

Relaciones entre clases: Delegaciones, asociaciones, agregaciones, herencia

17.1. Relaciones entre clases

17.2. Dependencia

17.3. Asociación

17.4. Agregación

17.5. Jerarquía de clases: generalización y especialización

17.6. Herencia: clases derivadas

17.7. Accesibilidad y visibilidad en herencia

17.8. Un caso de estudio especial: herencia múltiple

17.9. Clases abstractas

CONCEPTOS CLAVE

RESUMEN

EJERCICIOS

INTRODUCCIÓN

En este capítulo se introducen los conceptos fundamentales de relaciones entre clases. Las relaciones más importantes soportadas por la mayoría de las metodologías de orientación a objetos y en particular por UML son: asociación, agregación y generalización/especialización. En el capítulo se describen estas relaciones así como las notaciones gráficas correspondientes en UML.

De modo especial se introduce el concepto de *herencia* como exponente directo de la relación de generalización/especialización y se muestra cómo crear *clases derivadas*. La herencia hace posible crear je-

rarquías de clases relacionadas y reduce la cantidad de código redundante en componentes de clases. El soporte de la herencia es una de las propiedades que diferencia los lenguajes *orientados a objetos* de los lenguajes *basados en objetos* y *lenguajes estructurados*.

La *herencia* es la propiedad que permite definir nuevas clases usando como base a clases ya existentes. La nueva clase (*clase derivada*) hereda los atributos y comportamiento que son específicos de ella. La herencia es una herramienta poderosa que proporciona un marco adecuado para producir software fiable, comprensible, bajo coste, adaptable y reutilizable.

17.1. RELACIONES ENTRE CLASES

Una relación es una conexión semántica entre clases. Permite que una clase conozca sobre los atributos, operaciones y relaciones de otras clases. Las clases no actúan aisladas entre sí, al contrario las clases están relacionadas unas con otras. Una clase puede ser un tipo de otra clase —generalización— o bien puede contener objetos de otra clase de varias formas posibles, dependiendo de la fortaleza de la relación entre las dos clases.

La fortaleza de una relación de clases [Miles, Hamilton 2006] se basa en el modo de dependencia de las clases implicadas en las relaciones entre ellas. Dos clases que son fuertemente dependientes una de otra se dice que están *acopladas fuertemente* y en caso contrario están *acopladas débilmente*.



Figura 17.1. Relaciones entre clases.

Las relaciones entre clases se corresponden con las relaciones entre objetos físicos del mundo real, o bien objetos imaginarios en un mundo virtual. En UML las formas en las que se conectan entre sí las clases, lógica o físicamente, se modelan como relaciones. En el modelado orientado a objetos existen tres clases de relaciones muy importantes: *dependencias*, *generalizaciones-especializaciones* y *asociaciones* [Booch 2006]:

- Las *dependencias* son relaciones de uso.
- Las *asociaciones* son relaciones estructurales entre objetos. Una relación de asociación “todo/parte”, en la cual una clase representa una cosa grande (“el todo”) que consta de elementos más pequeños (“las partes”) se denomina *agregación*.
- Las *generalizaciones* conectan clases generales con otras más especializadas en lo que se conoce como relaciones subclase/superclase o hijo/padre.

Una *relación* es una conexión entre elementos. En el modelado orientado a objetos una relación se representa gráficamente con una línea (continua, punteada) que une las clases.

17.2. DEPENDENCIA

La relación más débil que puede existir entre dos clases es una relación de *dependencia*. Una dependencia entre clases significa que una clase utiliza, o tiene conocimiento de otra clase, o dicho de otro modo “lo que una clase necesita conocer de otra clase para utilizar objetos de esa clase” (Russ Miles & Kim Hamilton, *Learning UML 2.0*, O’Reilly, páginas 81-82). Normalmente es una relación transitoria y significa que una clase dependiente interactúa brevemente con la clase destino, pero normalmente no tiene con ella una relación de un tiempo definido. Una *dependencia* es una relación de uso que declara que un elemento utiliza la información y los servicios de otro elemento pero no necesariamente a la inversa.

La dependencia se lee normalmente como una relación “...usa un...”. Por ejemplo, si se tiene una clase *Ventana* que envía un aviso a una clase llamada *EventoCerrarVentana* cuando está próxima a abrirse. Entonces se dice que *Ventana* utiliza *EventoCerrarVentana*.

Otro ejemplo puede ser una clase *InterfazUsuario* que depende de otra clase *EntradaBlog* ya que necesita leer el contenido de la entrada de un *blog* (página web) para visualizar al usuario.

En un diagrama de clases, la dependencia se representa utilizando una línea discontinua dirigida hacia el elemento del cual depende. La flecha punteada de dependencia (Figura 17.2) de la página siguiente que significa “se utiliza simplemente cuando se necesita y se olvida luego de ella”.

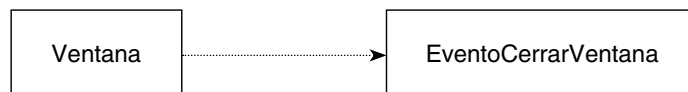


Figura 17.2. Relación de dependencia. *Ventana* depende de la clase *EventoCerrarVentana* porque necesitará leer el contenido de esta clase para poder cerrar la ventana

Otro ejemplo de dependencia se muestra entre la clase `Interfaz` y la clase `EntradaBlog` ya que ambas clases trabajan juntas. `Interfaz` necesitará leer el contenido de las entradas del *blog* para visualizar estas entradas al usuario.

Las dependencias se usarán cuando se quiera indicar que un elemento utiliza a otro. Una dependencia implica que los objetos de una clase pueden trabajar juntos; por consiguiente, se considera que es la relación directa más débil que puede existir entre dos clases

17.3. ASOCIACIÓN

Una **asociación** es más fuerte que la dependencia y normalmente indica que una clase recuerda o retiene una relación con otra clase durante un período determinado de tiempo. Es decir, las clases se conectan juntas conceptualmente en una asociación. La asociación realmente significa que una clase contiene una referencia a un objeto u objetos, de la otra clase en la forma de un atributo. La asociación se representa utilizando una simple línea que conecta las dos clases, como se muestra en la Figura 17.3.

Una *asociación* es una relación estructural que especifica que los objetos de una clase están conectados con los objetos de otra clase. En general, si se encuentra que una clase *trabaja* con un objeto de otra clase, entonces la relación entre esas clases es una buena candidata para una asociación en lugar de una dependencia.

Gráficamente, una asociación se representa como una línea continua que conecta la misma o diferentes clases. Las asociaciones se deben utilizar cuando se desee representar relaciones estructurales. Los adornos de una asociación son: línea continua, nombre de la asociación, dirección del nombre mediante una flecha que apunta en la dirección de una clase a la otra. La *navegabilidad* se aplica a una relación de asociación que describe qué clase contiene el atributo que soporta la relación.

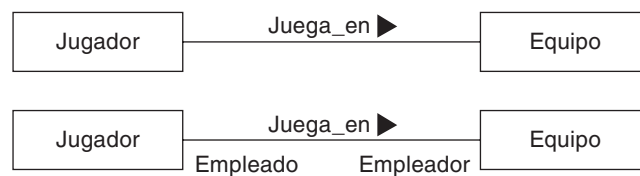


Figura 17.3. Relación de asociación (Jugador-Equipo).

Si se encuentra que una clase *trabaja con* un objeto de otra clase, entonces la relación entre clases es candidata a una asociación en lugar de a una dependencia. Cuando una clase se asocia con otra clase, cada una juega un rol dentro de la asociación. El rol se representa cerca de la línea próxima a la clase. En la asociación entre un `Jugador` y un `Equipo`, si esta es profesional, el equipo es el `Empleador` y el jugador es el `Empleado`.

Una asociación puede ser bidireccional. Un `Equipo` *emplea* a jugadores.

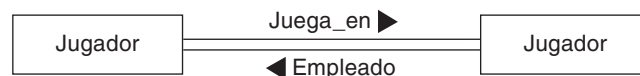


Figura 17.4. Relación de asociación bidireccional.

También pueden existir asociaciones entre varias clases, de modo que varias clases se pueden conectar a una clase.

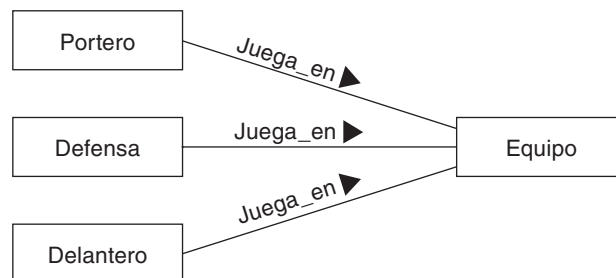


Figura 17.5. Asociación entre varias clases.

Una **asociación** es una conexión conceptual o semántica entre clases. Cuando una asociación conecta dos clases, cada clase envía mensajes a la otra en un diagrama de colaboración. *Una asociación es una abstracción de los enlaces que existen entre instancias de objetos*. Los siguientes diagramas muestran objetos enlazados a otros objetos y sus clases correspondientes asociadas. Las asociaciones se representan de igual modo que los enlaces. La diferencia entre un enlace y una asociación se determina de acuerdo al contexto del diagrama.

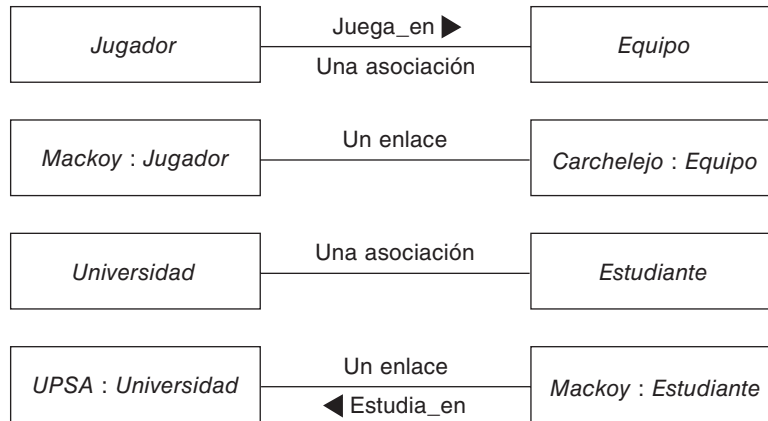


Figura 17.6. Asociación entre clases.

Regla

El significado más típico es una conexión entre clases, es una relación semántica entre clases. Se dibuja con una línea continua entre las dos clases. La asociación tiene un nombre (cerca de la línea que representa la asociación), normalmente un verbo, aunque está permitido los nombres o frases nominales. Cuando se modela un diagrama de clases, se debe reflejar el sistema que se está construyendo y por ello los nombres de la asociación deben deducirse del dominio del problema, al igual que sucede con los nombres de las clases.



Figura 17.7. Un programador utiliza un computador.
La clase *Programador* tiene una asociación con la clase *Computador*.

Es posible utilizar asociaciones navegables añadiendo una flecha al final de la asociación. La flecha indica que la asociación sólo se puede utilizar en la dirección de la flecha.



Figura 17.8. Una asociación navegable representa a una persona que posee (es propietaria) de varios carros, pero no implica que un auto pueda ser propiedad de varias personas.

Las asociaciones pueden tener dos nombres, uno en cada dirección.

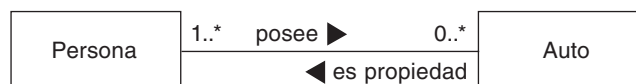


Figura 17.9. Una asociación navegable en ambos sentidos, con un nombre en cada dirección.

Las asociaciones pueden ser bidireccionales o unidireccionales. En UML las asociaciones bidireccionales se dibujan con flechas en ambos sentidos. Las asociaciones unidireccionales contienen una flecha que muestra la dirección de navegación.

En las asociaciones se pueden representar los roles o papeles que juegan cada clase dentro de las mismas. La Figura 17.10 muestra como se representan los roles de las clases. Un nombre de rol puede ser especificado en cualquier lado de la asociación. El siguiente ejemplo ilustra la asociación entre la clase Universidad y la clase Persona. El diagrama especifica que algunas personas actúan como estudiantes y algunas otras personas actúan como profesores. La segunda asociación también lleva un nombre de rol en la clase Universidad para indicar que la universidad actúa como un empresario (empleador) para sus profesores. Los nombres de los roles son especialmente interesantes cuando varias asociaciones conectan dos clases idénticas.



Figura 17.10. Roles en las asociaciones.

17.3.1. Multiplicidad

Entre asociaciones existe la propiedad de la *multiplicidad*: número de objetos de una clase que se relaciona con un único objeto de una clase asociada (un equipo de futbol tiene once jugadores).

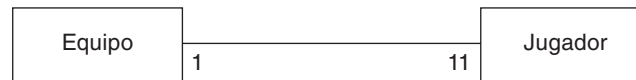


Figura 17.11. Multiplicidad en una asociación.

En notación moderna, a esta relación se le suele llamar también “*tiene un*”, pero hay que tener cuidado porque este concepto es sutil y de hecho siempre ha representado a la agregación, pero como se verá después UML contempla que la agregación *posee* (... *owns a...*). Por esta razón nos inclinaremos en considerar la relación “*tiene-un*” (*has-a*) como la relación de agregación.

La multiplicidad representa la cantidad de objetos de una clase que se relacionan con un objeto de la clase asociada. La información de multiplicidad aparece en el diagrama de clases a continuación del rol correspondiente. La multiplicidad se escribe como una expresión con un valor mínimo y un valor máximo, que pueden ser iguales; se utilizan dos puntos consecutivos para separar ambos valores. Cuando se indica una multiplicidad en un extremo de una asociación se está especificando cuántos objetos de la clase de ese extremo pueden existir por cada objeto de la clase en el otro extremo. UML utiliza un asterisco (*) para representar *más* y representa *muchos*. La Tabla 17.1 resume los valores más típicos de multiplicidad.

Tabla 17.1. Multiplicidad en asociaciones

Símbolo	Significado
1	Uno y sólo uno
0 .. 1	Cero o uno
m .. n	De m a n (enteros naturales)
*	De cero a muchos (cualquier entero positivo)
0 .. *	De cero a muchos (cualquier entero positivo)
1 .. *	De uno a muchos (cualquier entero positivo)
2	Dos
5 .. 11	Cinco a once
5, 10	Cinco o diez

Si no se especifica multiplicidad, es uno (1) por omisión. La multiplicidad se muestra cerca de los extremos de la asociación, en la clase donde es aplicable.

EJEMPLO 17.1

Relación de asociación entre las clases Empresa y Persona.



Cada objeto Empresa tiene como empleados, 1 o más objetos Persona (Multiplicidad 1.. *); pero cada objeto Persona tiene como patrón a cero o más objetos Empresa.

17.3.2. Restricciones en asociaciones

En algunas ocasiones, una asociación entre dos clases ha de seguir una regla. En este caso, la regla se indica poniendo una restricción cerca de la línea de la asociación que se representa por el nombre encerrado entre llaves.

EJEMPLO 17.2

Un cajero de un banco (humano o electrónico) atiende a clientes. La atención a los clientes se realiza en el orden en que se colocan ante la ventanilla o mostrador, o bien en función del momento de la petición electrónica de acceso al cajero.



Figura 17.12. Restricción en una asociación.

En algunas ocasiones las asociaciones pueden establecer una restricción entre las clases. Las restricciones típicas pueden ser {ordenado} {or}.

17.3.3. Asociación cualificada

Cuando la multiplicidad de una asociación es de uno a muchos, se puede reducir esta multiplicidad de uno a uno con una cualificación. El símbolo que representa la cualificación es un pequeño rectángulo adjunto a la clase correspondiente.

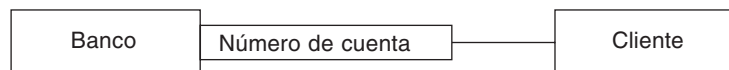


Figura 17.13. Asociación cualificada.

17.3.4. Asociaciones reflexivas

A veces, una clase es una asociación consigo misma. Esta situación se puede presentar cuando una clase tiene objetos que pueden jugar diferentes roles.

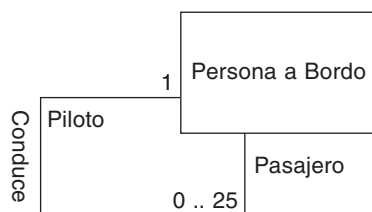


Figura 17.14. Asociación reflexiva.

17.3.5. Diagrama de objetos

Los objetos se pueden representar en diagramas de objetos. Un diagrama de objetos en UML tiene la misma notación y relaciones que un diagrama de clases, dado que los objetos son instancias de las clases. Así, un diagrama de clases muestra los tipos de clases y sus relaciones, mientras que el diagrama de objetos muestra instancias específicas de esas clases y enlaces específicos entre esas instancias en un momento dado. El diagrama de objetos muestra también cómo los objetos de un diagrama de clases se pueden combinar con cada uno de los restantes en un cierto instante de tiempo.

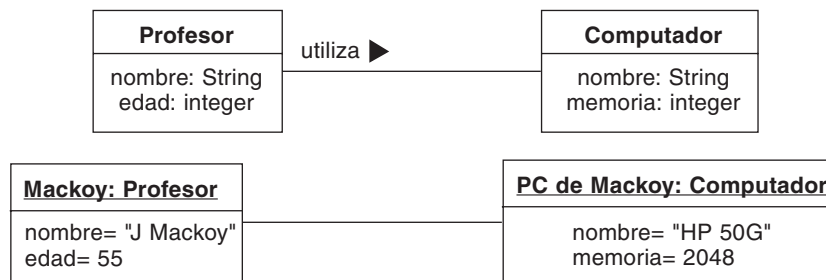


Figura 17.15. Diagrama de clases y diagrama de objetos.

Enlaces

Al igual que un objeto es una instancia de una clase, una asociación tiene también instancias. Por ejemplo, la asociación de la Figura 17.16.

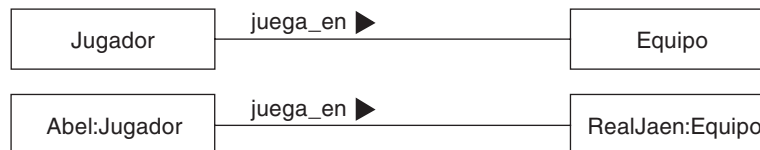


Figura 17.16. Instancia de una asociación.

17.3.6. Clases de asociación

Es frecuente encontrarse con una asociación que introduce nuevas clases. Una clase se puede conectar a una asociación, en cuyo caso se denomina *clase asociación*. De hecho una asociación puede tener atributos y operaciones tal como una clase, este es el caso de la clase asociación.

La clase asociación no se conecta a ninguno de los extremos de la asociación, sino que se conecta a la asociación real, a través de una línea punteada. La clase asociación se utiliza para añadir información extra en un enlace, por ejemplo, el momento en que fue creado. Cada enlace de la asociación se relaciona a un objeto de la clase asociación. La clase asociación se utiliza para añadir información extra en un enlace, por ejemplo, el momento en que se crea el enlace. Cada enlace de la asociación se relaciona a un objeto de la clase asociación.

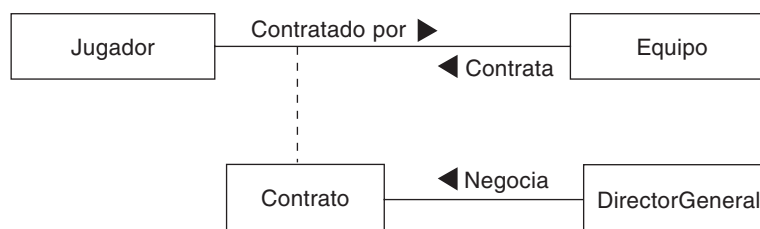


Figura 17.17. La clase asociación Contrato está asociada con la clase DirectorGeneral.

Una clase asociación es una asociación —con métodos y atributos— que es también una clase normal. La clase asociación se representa con una línea punteada que la conecta a la asociación que representa.

Las clases de asociación se pueden aplicar en asociaciones binarias y n-arias. De modo similar a como una clase define las características de sus objetos, incluyendo sus características estructurales y sus características de comportamiento, una clase asociación se puede utilizar para definir las características de sus enlaces, incluyendo sus características estructurales y características de comportamiento. Estos tipos de clases se utilizan cuando se necesita mantener información sobre la propia relación.

EJEMPLO 17.3

Clase asociación `EquipoFutbol`.

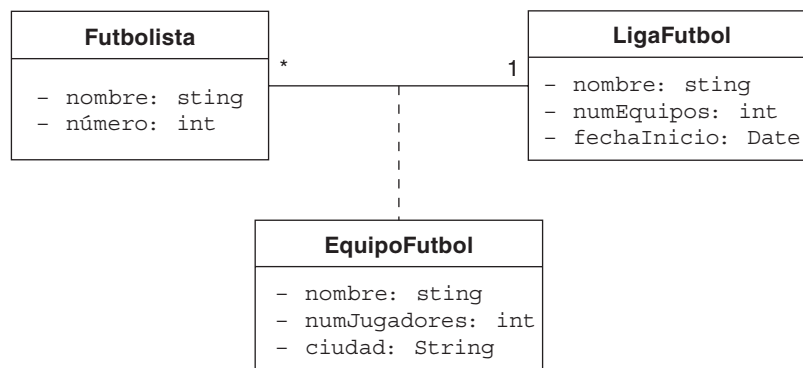


Figura 17.18. Clase asociación `EquipoFutbol`.

Criterios de diseño

Cuando se traducen a código, las relaciones con clases asociación se obtienen, normalmente, tres clases: una por cada extremo de la asociación y una por la propia clase asociación.

EJERCICIO 17.1

Diseñar el control de 5 ascensores (elevadores) de un edificio comercial que tenga presente las peticiones de viaje de los diferentes clientes, en función del orden y momento de llamada.

Análisis

El control de ascensores tiene un enlace con cada uno de los 5 ascensores y otro enlace con el botón (pulsador) de llamada de “subida/bajada”. Para gestionar el control de llamadas de modo que responda el ascensor, que cumpla con los requisitos estipulados (situado en piso más cercano, parado, en movimiento, etc.) se requiere una clase `Cola` que almacene las peticiones tanto del `ControlAscensor` como del propio ascensor (los motores interiores del ascensor). Cuando el control del ascensor elige un ascensor para realizar la petición de un pasajero externo al ascensor, un pasajero situado en un determinado piso o nivel, el control del ascensor lee la cola y elige el ascensor que está situado, disponible y más próximo en la `Cola`. Esta elección normalmente se realizará por algún algoritmo inteligente.

En consecuencia, se requieren cuatro clases: `ControlAscensor`, `Ascensor` (elevador), `Botón` (pulsador) y `Cola`. La clase `Cola` será una clase asociación ya que puede ser requerida tanto por el control de ascensores como por cualquier ascensor.

Recuerde el lector que una estructura de datos cola es una estructura en la que cada elemento que se introduce en la cola es el primer elemento que sale de la cola (al igual que sucede con la cola para sacar una entrada de cine,

comprar el pan o una cola de impresoras conectadas a una computadora central). En cada enlace entre los ascensores y el control de ascensores hay una cola. Cada cola almacena las peticiones del control del ascensor y el propio ascensor (los botones internos del ascensor).

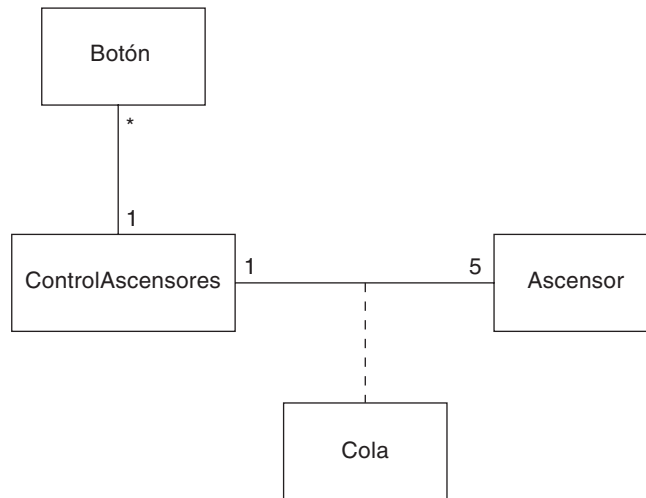


Figura 17.19. Diagrama de clases para control de ascensores.

Asociaciones ternarias

Las clases se pueden asociar una a una o bien se pueden asociar unas con otras. La asociación ternaria es una asociación especial que asocia a tres clases. La asociación ternaria se representa con la figura geométrica “rombo” y con los roles y multiplicidad necesarios, pero no están permitidos los cualificadores ni la agregación. Se puede conectar una clase asociación a la asociación ternaria, dibujando una línea punteada a uno de los cuatro vértices del rombo.

EJERCICIO 17.2

Dibujar un modelo de seguros de automóviles que represente: compañía de seguros, asegurados, póliza de seguro y el contrato de seguro.

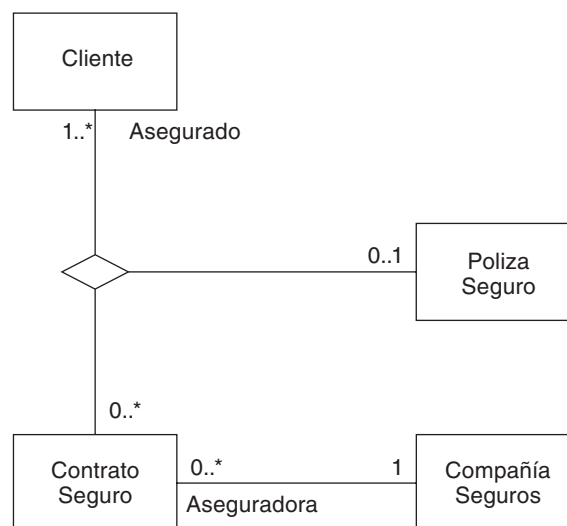


Figura 17.20. Diagrama de clases con una asociación ternaria.

El núcleo muestra un cliente —juega el rol o papel de asegurado— que puede tener 0 o muchos contratos de seguros y cada contrato de seguro está asociado con una única compañía de seguros que juega el rol de aseguradora. En la asociación entre cliente y contrato de seguro, hay una o ninguna pólizas de seguros.

Asociaciones cualificadas

Una asociación cualificada se utiliza con asociaciones una-a-muchas o muchas-a-muchas para reducir su multiplicidad a una, con objeto de especificar un objeto único (o grupos de objetos) desde el destino establecido. La asociación cualificada es muy útil para modelar cuando se busca o navega para encontrar objetos específicos en una colección determinada.

Ejemplos típicos son los sistemas de reservas de pasaje de avión, de entradas de cine, de reservas de habitaciones en hoteles. Cuando se solicita un pasaje, una entrada o una habitación, es frecuente que nos den un localizador de la reserva (H234JK, o similar) que se ha de proporcionar físicamente a la hora de sacar el pasaje en el aeropuerto, la entrada en el cine o ir a alojarse en el hotel.

El atributo o calificador se conoce normalmente como *identificador* (número de ID). Existen numerosos calificadores tales como: ID de reserva, nombre, número de la tarjeta de crédito, número de pasaporte, etc. En terminología de proceso de datos, también se conoce como clave de búsqueda. Este identificador o calificador, al especificar una clave única resuelve la relación uno a muchos y lo convierte en uno a uno.

El calificador se representa como una caja o rectángulo pequeño que se dibuja en el extremo correspondiente de la asociación, al lado de la clase a la cual está asociada. El calificador representa un añadido a la línea de la asociación y fuera de la clase. Las asociaciones cualificadas reducen la multiplicidad real en el modelo, de uno-a-muchos a uno-a-uno indicando con el calificador una identidad para cada asociación.

EJEMPLO 17.4

1. Lista de reservas. Calificador, ID de la reserva.

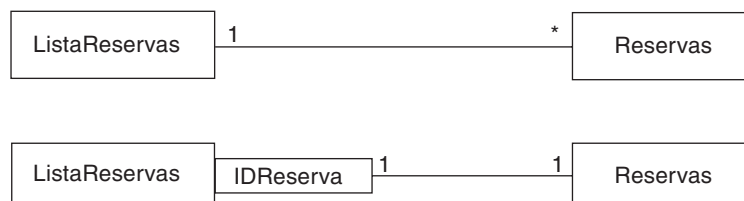


Figura 17.21. Asociación cualificada.

2. Lista de pasajes: vuelo Madrid-Cartagena de Indias con la compañía aérea Iberia.

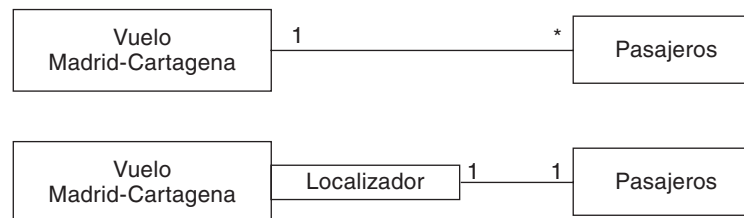


Figura 17.22. Asociación cualificada.

Asociaciones reflexivas

En ocasiones una clase es una asociación consigo misma. En este caso la asociación se denomina asociación reflexiva. Esta situación se produce cuando una clase tiene objetos que pueden jugar diferentes roles. Por ejemplo, un ocupante de un avión de pasajeros puede ser: un pasajero, un miembro de tripulación o un piloto. Este tipo de asociación

se representa gráficamente dibujando la línea de asociación con origen y final en la propia clase y con indicación de los roles y multiplicidades correspondientes.

EJEMPLO 17.5

1. Asociación reflexiva `OcupanteAvión`.

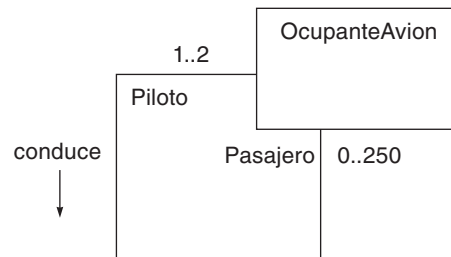


Figura 17.23. Asociación reflexiva.

2. Asociación reflexiva `OcupanteAuto`.

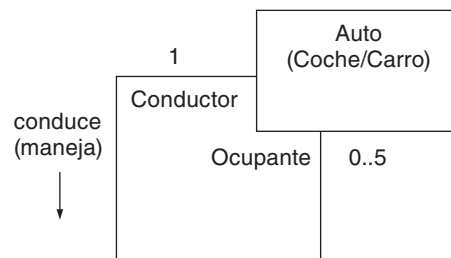


Figura 17.24. Asociación reflexiva.

17.3.7. Restricciones en asociaciones

Una asociación entre clases, a veces, tiene que seguir una regla determinada. Esta regla se indica poniendo una restricción cerca de la línea de la asociación. Una restricción típica se produce cuando una clase (un objeto) atiende a otra clase (un objeto) en función de un determinado orden o secuencia. Por ejemplo, un vendedor de entradas de cine (taquillero) atiende a los espectadores a medida que se sitúan delante de la ventanilla de entradas. En este caso, esta restricción se representa en el modelo con la palabra *ordered* encerrada entre llaves.



Figura 17.25. Restricciones entre asociaciones.



Figura 17.26. Asociación con una restricción. (El cajero atiende al cliente por orden de llegada a caja.)

Otro tipo de restricciones se pueden presentar y se representan con relaciones *or* o bien *xor*, y se representan gráficamente con una línea de asociación y las palabras *or*, *xor* entre llaves.

EJEMPLO 17.6

Un estudiante se matricula en una universidad en estudios de ingeniería o de ciencias.

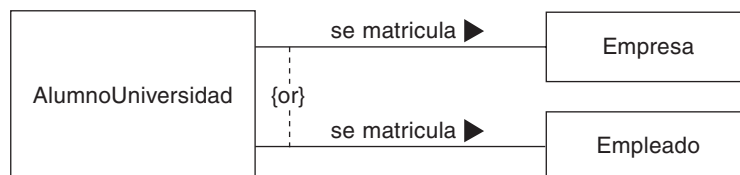


Figura 17.27. Restricción en una asociación.

EJEMPLO 17.7

La relación xor implica una u otra asociación y no pueden ser nunca las dos. El caso de una póliza de seguro de una empresa que puede ser o corporativa o de empleado, pero son entre sí, excluyentes.

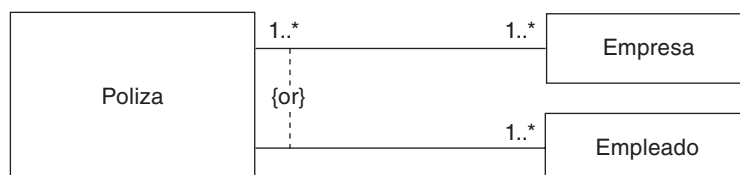


Figura 17.28. Relación xor en una asociación.

EJEMPLO 17.8

Un cajero de un banco (humano o electrónico) atiende a clientes. La atención a los clientes se realiza en el orden en que se colocan ante la ventanilla o mostrador, o bien en función del momento de la petición electrónica de acceso al cajero.

Enlaces

Al igual que un objeto es una instancia de una clase, una asociación también tiene instancia. Por ejemplo la asociación *Juega_en* y su instancia se muestran en la Figura 17.29.

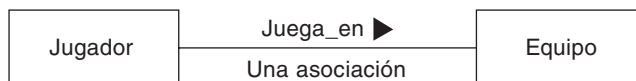


Figura 17.29. Instancia de una asociación.

17.4. AGREGACIÓN

Una **agregación** es un tipo especial de asociación que expresa un acoplamiento más fuerte entre clases. Una de las clases juega un papel importante dentro de la relación con las otras clases. La agregación permite la representación

de relaciones tales como “maestro y esclavo”, “todo y parte de” o “compuesto y componentes”. Los componentes y la clase que constituyen son una asociación que conforma un todo.

Las agregaciones representan conexiones bidireccionales y asimétricas. El concepto de agregación desde un punto de vista matemático es una relación que es transitiva, asimétrica y puede ser reflexiva.

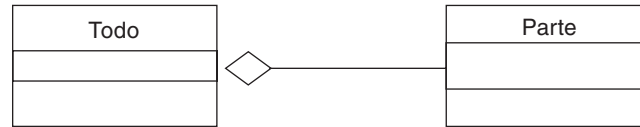


Figura 17.30. Relación de agregación.

La *agregación* es una versión más fuerte que la asociación. Al contrario que la asociación, la agregación implica normalmente propiedad o pertenencia. La agregación se lee normalmente como relación “... *posee un...*” o relación “*todo-parte*”, en la cual una clase (“el todo”) representa un gran elemento que consta de elementos más pequeños (“las partes”). La agregación se representa con un rombo a continuación de la clase “propietaria” y una línea recta que apunta a la clase “poseída”. Esta relación se conoce como “*tiene-un*” ya que el todo tiene sus partes; un objeto es parte de otro objeto.

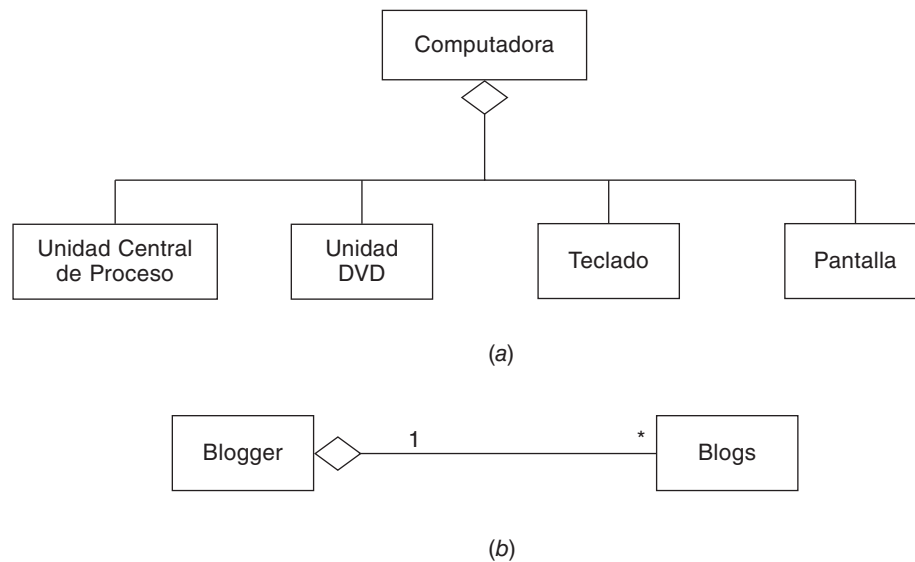


Figura 17.31. Relaciones de agregación: (a) *Computadora* con sus componentes; (b) Un *Blogger* propietario de muchos (*) *Blogs*.

Desde el punto de vista conceptual una clase realmente *posee*, pero puede *compartir* objetos de otra clase. Una agregación es un caso especial de asociación. Un ejemplo de una agregación es un automóvil que consta de cuatro ruedas, un motor, un chasis, una caja de cambios, etc. Otro ejemplo es un árbol binario que consta de cero, uno o dos nuevos árboles. Una agregación se representa como una jerarquía con la clase “todo” (por ejemplo, un sistema de computadora) en la parte superior y sus componentes en las partes inferiores (por ejemplo CPU, discos, web-cam,...). La representación de la agregación se realiza insertando un rombo vacío en la parte *todo*.

EJEMPLO 17.9

Una computadora es un conjunto de elementos que consta de una unidad central, teclado, ratón, monitor, unidad de CD-ROM, módem, altavoces, escáner, etc.

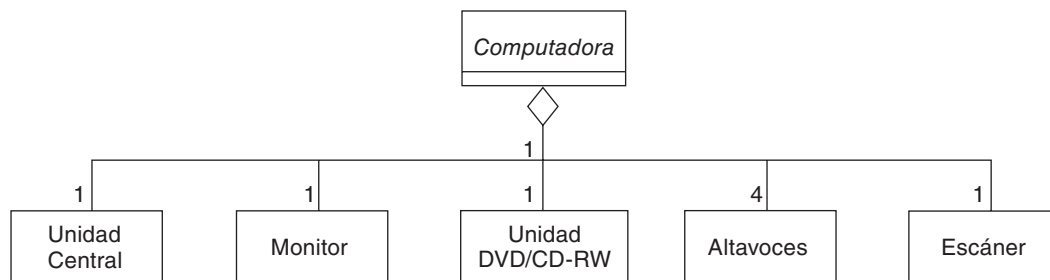


Figura 17.32. Una agregación computadora.

Restricciones en las agregaciones

En ocasiones el conjunto de componentes posibles en una agregación se establece dentro de una relación *O*. Así, por ejemplo, el menú del día en un restaurante puede constar de: un primer plato (a elegir entre dos-tres platos), el segundo plato (a elegir entre dos-tres platos) y un postre (a elegir entre cuatro postres). El modelado de este tipo se realiza con la palabra reservada *O* dentro de llaves con una línea discontinua que conecte las dos líneas que conforman el todo.

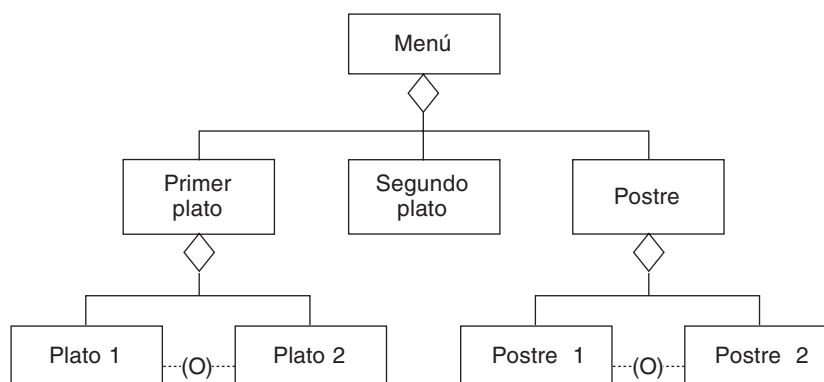


Figura 17.33. Restricción en agregaciones.

17.4.1 Composición

Una **composición** es un tipo especial de agregación que impone algunas restricciones: si el objeto completo se copia o se borra (elimina), sus partes se copian o se suprimen con él. La composición representa una relación fuerte entre clases y se utiliza para representar una relación *todo-parte* (*whole-part*). Cada componente dentro de una composición puede pertenecer tan sólo a un todo. El símbolo de una composición es el mismo que el de una agregación, excepto que el rombo está relleno (Figura 17.34). Es como una agregación pero con el rombo pintado y no vacío.

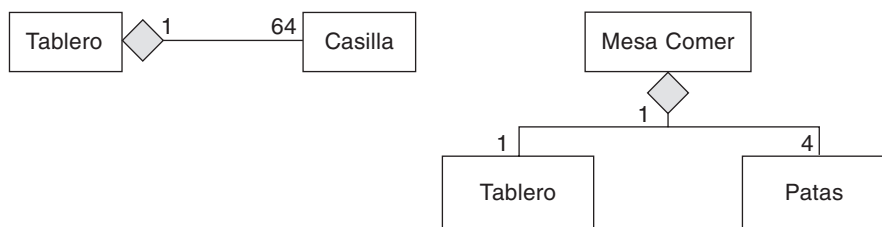


Figura 17.34. Relaciones de composición.

Una relación de composición se lee normalmente como “... es *parte de*...”, que significa se necesita leer la composición de la parte al todo. Por ejemplo, si una ventana de una página web tiene una barra de títulos, se puede representar que la clase `BarraTitulo` es *parte de* una clase denominada `Ventana`.

EJEMPLO 17.10

Una mesa para jugar al póker es una composición que consta de una superficie de la mesa y cuatro patas.

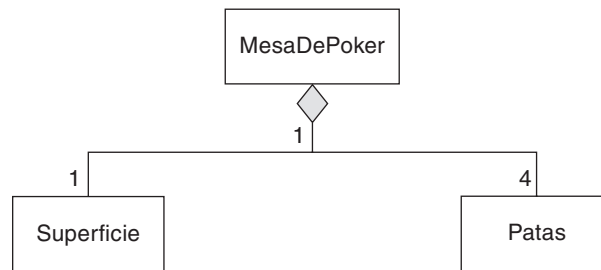


Figura 17.35. Composición

EJEMPLO 17.11

Un auto tiene un motor que no puede ser parte de otro auto. La eliminación completa del auto supone la eliminación de su motor.

17.5. JERARQUÍA DE CLASES: GENERALIZACIÓN Y ESPECIALIZACIÓN

La jerarquía de clases (o clasificaciones) hace lo posible para gestionar la complejidad ordenando objetos dentro de árboles de clases con niveles crecientes de abstracción. Las jerarquías de clases más conocidas son: **generalización** y **especialización**.

La relación de generalización es un concepto fundamental de la programación orientada a objetos. Una *generalización* es una relación entre un elemento general (llamado *superclase* o “*padre*”) y un caso más concreto de ese elemento (denominado *subclase* o “*hijo*”). Se conoce como relación *es-un* y tiene varios nombres, *extensión*, *herencia*... Las clases modelan el hecho de que el mundo real contiene objetos con propiedades y comportamientos. La herencia modela el hecho de que estos objetos tienden a ser organizados en jerarquías. Estas jerarquías representan la relación *es-un*.

La generalización normalmente se lee como “...*es un*...” comenzando en la clase específica, derivada o subclase y derivándose a la superclase o clase base. Por ejemplo, un `Gato` *es-un tipo de* `Animal`. La relación de generalización se representa con una línea continua que comienza en la subclase y termina en una flecha cerrada en la superclase.

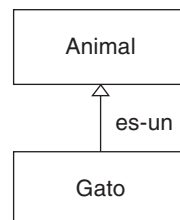


Figura 17.36. Relaciones de generalización.

En UML la relación se conoce como generalización y en programación orientada a objetos como herencia. Al contrario que las relaciones de asociación, las relaciones de generalización no tienen nombre ni ningún tipo de multiplicidad.

Booch, para mostrar las semejanzas y diferencias entre clases, utiliza las siguientes clases de objetos: flores, margaritas, rosas rojas, rosas amarillas y pétalos. Se puede constatar que: Una margarita *es un* tipo (una clase) de flor.

- Una rosa *es un* tipo (diferente) de flor.
- Las rosas rojas y amarillas son *tipos de* rosas.
- Un pétalo *es una parte* de ambos tipos de flores.

Como Booch afirma, las clases y objetos no pueden existir aislados y, en consecuencia, existirán entre ellos relaciones. Las relaciones entre clases pueden indicar alguna forma de compartición, así como algún tipo de conexión semántica. Por ejemplo, las margaritas y las rosas son ambas tipos de flores, significando que ambas tienen pétalos coloreados brillantemente, ambas emiten fragancia, etc. La conexión semántica se materializa en el hecho de que las rosas rojas y las margaritas y las rosas están más estrechamente relacionadas entre sí que lo están los pétalos y las flores.

Las clases se pueden organizar en estructuras jerárquicas. La *herencia* es una relación entre clases donde una clase comparte la estructura o comportamiento, definida en una (*herencia simple*) o más clases (*herencia múltiple*). Se denomina *superclase* a la clase de la cual heredan otras clases. De modo similar, una clase que hereda de una o más clases se denomina *subclase*. Una subclase heredará atributos de una superclase más elevada en el árbol jerárquico. La herencia, por consiguiente, define un “tipo” de jerarquía entre clases, en las que una subclase hereda de una o más superclases. La Figura 17.37 ilustra una jerarquía de clases *Animal* con dos subclases que heredan de *Animal*, *Mamífero* y *Reptil*.

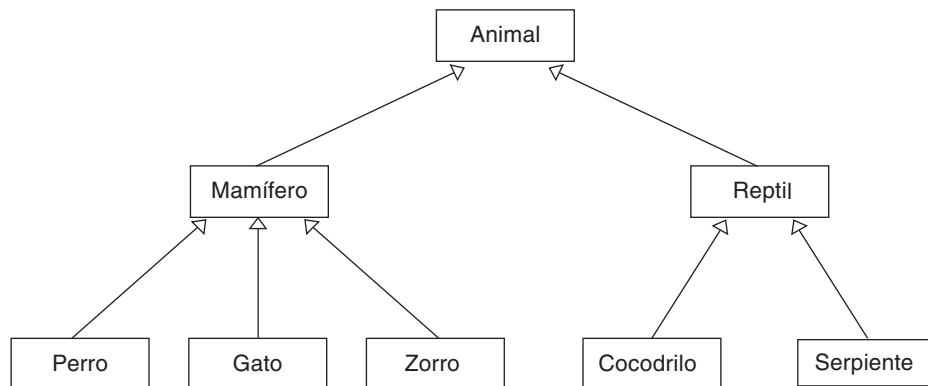


Figura 17.37. Jerarquía de clases.

Herencia es la propiedad por la cual instancias de una clase hija (o subclase) puede acceder tanto a datos como a comportamientos (métodos) asociados con una clase padre (o superclase). La herencia siempre es transitiva, de modo que una clase puede heredar características de superclases de nivel superior. Esto es, la clase *Perro* es una subclase de la clase *Mamífero* y de *Animal*.

Una vez que una jerarquía se ha establecido es fácil extenderla. Para describir un nuevo concepto no es necesario describir todos sus atributos. Basta describir sus diferencias a partir de un concepto de una jerarquía existente. La herencia significa que el comportamiento y los datos asociados con las clases hija son siempre una extensión (esto es, conjunto estrictamente más grande) de las propiedades asociadas con las clases padres. Una subclase debe tener todas las propiedades de la clase padre y otras. El proceso de definir nuevos tipos y reutilizar código anteriormente desarrollado en las definiciones de la clase base se denomina *programación por herencia*. Las clases que heredan propiedades de una clase base pueden, a su vez, servir como clases base de otras clases. Esta jerarquía de tipos normalmente toma la estructura de árbol, conocido como *jerarquía de clases* o *jerarquía de tipos*.

La jerarquía de clases es un mecanismo muy eficiente, ya que se pueden utilizar definiciones de variables y métodos en más de una subclase sin duplicar sus definiciones. Por ejemplo, consideremos un sistema que representa varias clases de vehículos manejados por humanos. Este sistema contendrá una clase genérica de vehículos, con subclases para todos los tipos especializados. La clase *Vehículo* contendrá los métodos y variables que fueran propios de todos los vehículos, es decir, número de matrícula, número de pasajeros, capacidad del depósito de combustible. La subclase, a su vez, contendrá métodos y variables adicionales que serán específicos a casos individuales.

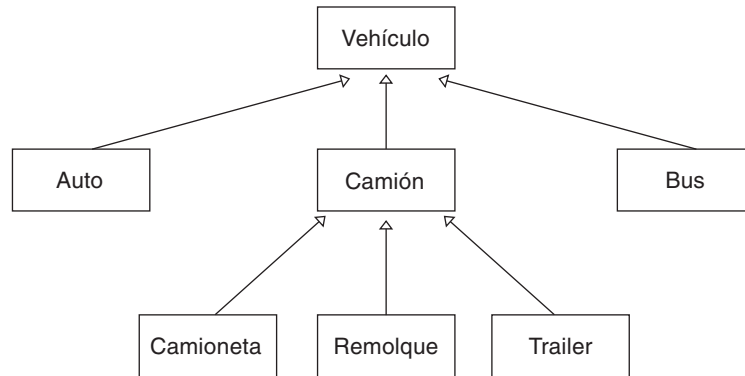


Figura 17.38. Subclases de la clase Vehículo.

La flexibilidad y eficiencia de la herencia no es gratuita; se emplea tiempo en buscar una jerarquía de clases para encontrar un método o variable, de modo que un programa orientado a objetos puede correr más lentamente que su correspondiente convencional. Sin embargo, los diseñadores de lenguajes han desarrollado técnicas para eliminar esta penalización en velocidad en la mayoría de los casos, permitiendo a las clases enlazar directamente con sus métodos y variables heredados, de modo que no se requiera realmente ninguna búsqueda.

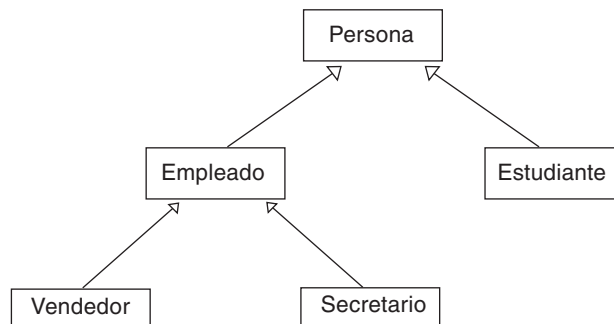


Figura 17.39. Una jerarquía Persona.

Regla

- Cada objeto es una instancia de una clase.
- Algunas clases —abstractas— no pueden instanciar directamente.
- Cada enlace es una instancia de una asociación.

17.5.1. Jerarquías de generalización/especialización

Las clases con propiedades comunes se organizan en superclases. Una **superclase** representa una *generalización* de las subclases. De igual modo, una subclase de una clase dada representa una *especialización* de la clase superior (Figura 17.40). La clase derivada *es-un* tipo de clase de la clase base o superclase.

Una superclase representa una *generalización* de las subclases. Una subclase de la clase dada representa una *especialización* de la clase ascendente (Figura 17.41).

En la *modelización* o *modelado* orientado a objetos es útil introducir clases en un cierto nivel que puede no existir en la realidad, pero que son construcciones conceptuales útiles. Estas clases se conocen como **clases abstractas** y su propiedad fundamental es que no se pueden crear instancias de ellas. Ejemplos de clases abstractas son *vehículo de pasajeros* y *vehículo de mercancías*. Por otra parte, de las subclases de estas clases abstractas, que corresponden a los objetos del mundo real, se pueden crear instancias directamente por sí mismas. Por ejemplo, de BMW se pueden obtener, dos instancias, *Coche1* y *Coche2*.

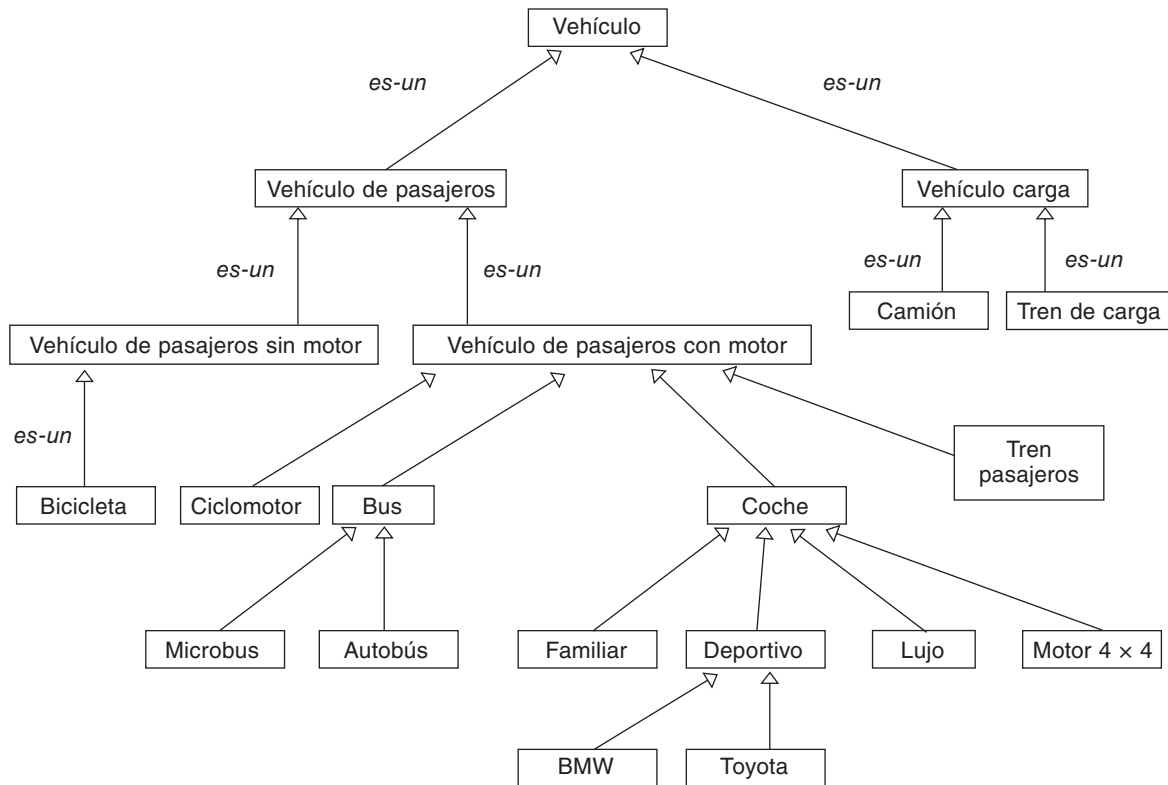


Figura 17.40. Relaciones de generalización.

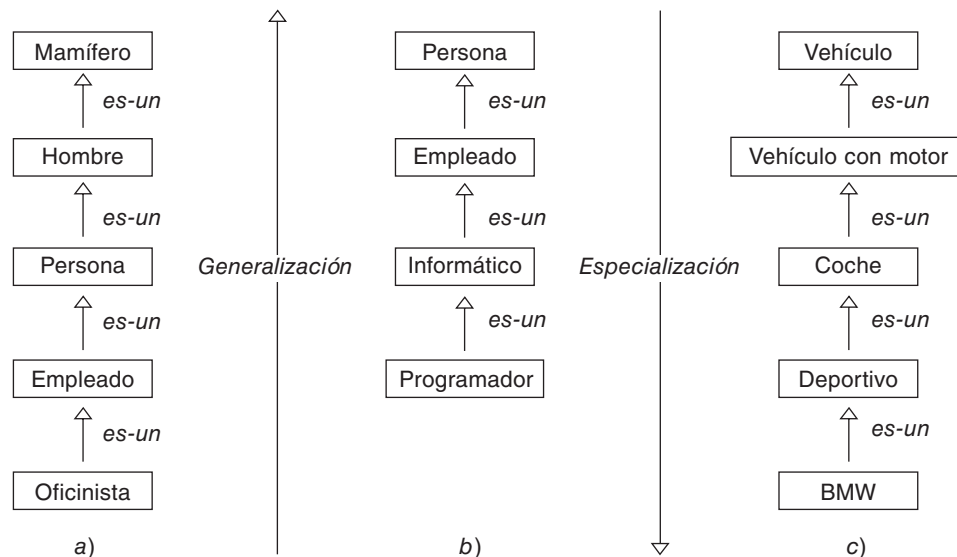


Figura 17.41. Relaciones de jerarquía es-un (is-a).

La generalización, en esencia, es una abstracción en que un conjunto de objetos de propiedades similares se representa mediante un objeto genérico. El método usual para construir relaciones entre clases es definir generalizaciones buscando propiedades y funciones de un grupo de tipos de objetos similares, que se agrupan juntos para formar un nuevo tipo genérico. Consideremos el caso de empleados de una compañía que pueden tener propiedades comunes (nombre, número de empleado, dirección, etc.) y funciones comunes (calcular_nómina), aunque dichos empleados pueden ser muy diferentes en atención a su trabajo: oficinistas, gerentes, programadores, ingenieros, etc. En este caso, lo normal será crear un objeto genérico o superclase **Empleado**, que definirá una clase de empleados individuales.

Por ejemplo, Analistas, Programadores y Operadores se pueden generalizar en la clase informático. Un programador determinado (Mortimer) será miembro de las clases Programador, Informático y Empleado; sin embargo, los atributos significativos de este programador variarán de una clase a otra.

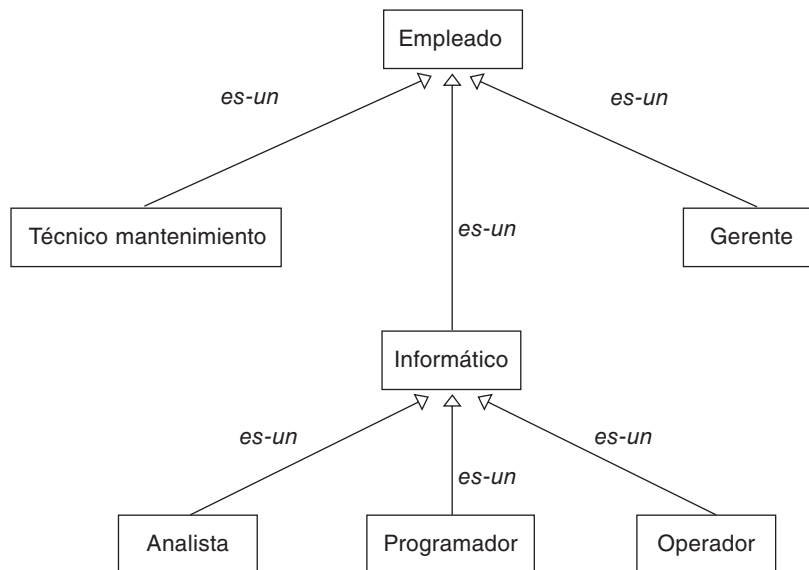


Figura 17.42. Una jerarquía de generalización de empleados.

La jerarquía de generalización/especialización tiene dos características fundamentales y notables. Primero, un tipo objeto no desciende más que de un tipo objeto genérico; segundo, los descendientes inmediatos de cualquier nodo no necesitan ser objetos de clases exclusivas mutuamente. Por ejemplo, los Gerentes y los Informáticos no tienen por qué ser exclusivos mutuamente, pero pueden ser tratados como dos objetos distintos; es el tipo de relación que se denomina *generalización múltiple*.

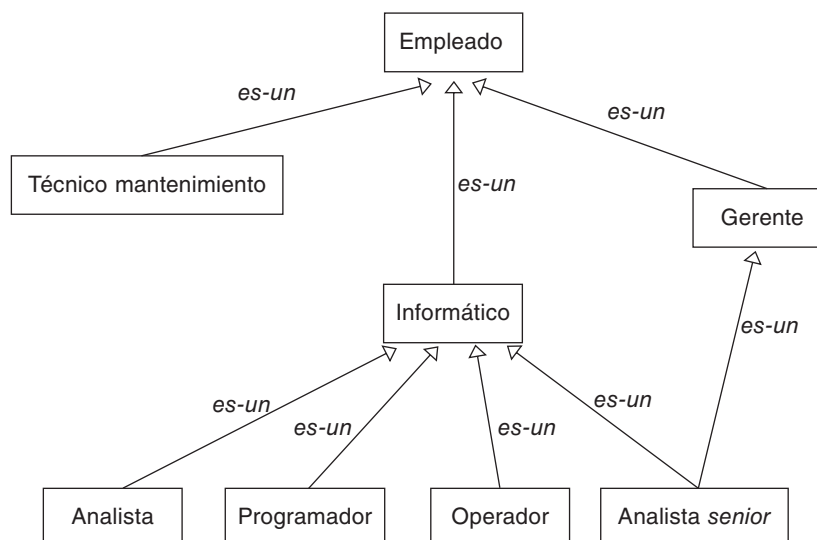


Figura 17.43. Una jerarquía de generalización múltiple.

UML define la generalización como herencia. De hecho, generalización es el concepto y herencia se considera la implementación del concepto en un lenguaje de programación.

Síntesis de generalización/Especialización [Muller 97]

1. La generalización es una relación de herencia entre dos elementos de un modelo tal como clase. Permite a una clase heredar atributos y operaciones de otra clase. En realidad es la factorización de elementos comunes (atributos operaciones y restricciones) dentro de un conjunto de clases en una clase más general denominada **superclase**. Las clases están ordenadas dentro de una jerarquía; una superclase es una abstracción de sus subclases.
2. La flecha que representa la generalización entre dos clases apunta hacia la clase más general.
3. La especialización permite la captura de las características específicas de un conjunto de objetos que no han sido distinguidos por las clases ya identificadas. Las nuevas características se representan por una nueva clase, que es una subclase de una de las clases existentes. La especialización es una técnica muy eficiente para extender un conjunto de clases de un modo coherente.
4. La generalización y la especialización son dos puntos de vista opuestos del concepto de jerarquía de clasificación; expresan la dirección en que se extiende la jerarquía de clases.
5. Una generalización no lleva ningún nombre específico; siempre significa “es un tipo de”, “es un”, “es uno de”, etc. La generalización sólo pertenece a clases, no se puede instanciar vía enlaces y por consiguiente no soporta el concepto de multiplicidad.
6. La generalización es una relación no reflexiva: una clase no se puede derivar de sí misma.
7. La generalización es una relación asimétrica: si la clase B se deriva de la clase A, entonces la clase A no se puede derivar de la clase B.
8. La generalización es una relación transitiva: si la clase C se deriva de la clase B que a su vez se deriva de la clase A, entonces la clase C se deriva de la clase A.

17.6. HERENCIA: CLASES DERIVADAS

Como ya se ha comentado, la herencia es la manifestación más clara de la relación de generalización/especialización y a la vez una de las propiedades más importantes de la orientación a objetos y posiblemente su característica más conocida y sobresaliente. Todos los lenguajes de programación orientados a objetos soportan directamente en su propio lenguaje construcciones que implementan de modo directo la relación entre clases derivadas.

La *herencia* o relación **es-un** es la relación que existe entre dos clases, en la que una clase denominada *derivada* se crea a partir de otra ya existente, denominada *clase base*. Este concepto nace de la necesidad de construir una nueva clase y existe una clase que representa un concepto más general; en este caso la nueva clase puede *heredar* de la clase ya existente. Así, por ejemplo, si existe una clase *Figura* y se desea crear una clase *Triángulo*, esta clase *Triángulo* puede derivarse de *Figura* ya que tendrá en común con ella un estado y un comportamiento, aunque luego tendrá sus características propias. *Triángulo es-un* tipo de *Figura*. Otro ejemplo, puede ser *Programador* que *es-un* tipo de *Empleado*.

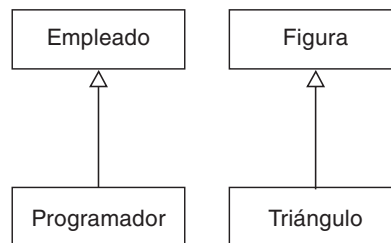


Figura 17.44. Clases derivadas.

17.6.1. Herencia simple

La implementación de la generalización es la herencia. Una clase hija o subclase puede heredar atributos y operaciones de otra clase padre o superclase. La clase padre es más general que la clase hija. Una clase hija puede ser, a su vez, una clase padre de otra clase hija. Mamífero es una clase derivada de Animal y Caballo es una clase hija o derivada de Mamífero.

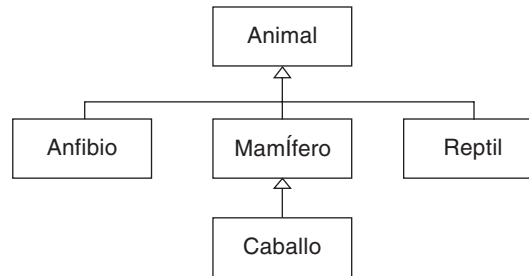


Figura 17.45. Herencia simple con dos niveles.

En UML, la herencia se representa con una línea que conecta la clase padre con la clase hija. En la parte de la línea que conecta a la clase padre se pone un triángulo abierto (punta de flecha) que apunta a dicha clase padre. Este tipo de conexión se representa como “*es un tipo de*”. Caballo *es-un-tipo de* Mamífero que a su vez es un tipo de Animal.

Una clase puede no tener padre, en cuyo caso se denomina *clase base* o *clase raíz*, y también puede no tener ninguna clase hija, en cuyo caso se denomina *clase terminal* o *clase hija*.

Si una clase tiene exactamente un padre, tiene **herencia simple**. Si una clase tiene más de un padre, tiene **herencia múltiple**.

17.6.2. Herencia múltiple

Herencia múltiple o **generalización múltiple** —en terminología oficial de UML— se produce cuando una clase hereda de dos o más clases padres (Figura 17.46).

Aunque la herencia múltiple está soportada en UML y en C++ (no en Java), en general, su uso no se considera una buena práctica en la mayoría de los casos. Esta característica se debe al hecho de que la herencia múltiple presenta un problema complicado cuando las dos clases padre tienen solapamiento de atributos y comportamientos. ¿A qué se debe la complicación? Normalmente a conflictos de atributos o propiedades derivadas. Por ejemplo, si las clases A1 y A2 tienen el mismo atributo nombre, la clase hija de ambas A1-2, ¿de cuál de las dos clases hereda el atributo?

C++, que soporta herencia múltiple, debe utilizar un conjunto propio de reglas del lenguaje C++ para resolver estos conflictos. Estos problemas conducen a malas prácticas de diseño y ha hecho que lenguajes de programación como **Java** y **C#** no soportan herencia múltiple. Sin embargo como C++ soporta esta característica, UML incluye en sus representaciones este tipo de herencia.

Regla

La generalización es una relación “**es-un**” (un Carro es-un Vehículo; un Gerente **es-un** Empleado, etc.). También se utiliza “**es-un-tipo-de**” (*is a kind of*).

En orientación a objetos la relación se conoce como *herencia* y en UML como *generalización*.

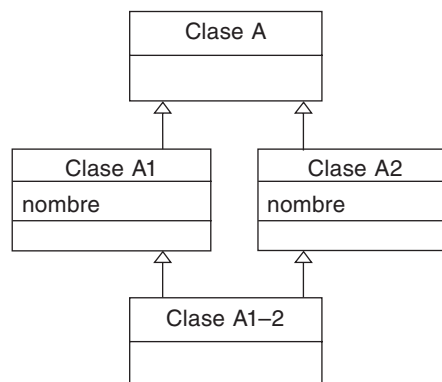


Figura 17.46. Herencia múltiple en la clase A1-2.

17.6.3. Niveles de herencia

La jerarquía de herencia puede tener más de dos niveles. Una clase hija puede ser una clase padre, a su vez, de otra clase hija. Así, una clase Mamífero es una clase hija de Animal y una clase padre de Caballo.

Las clases hija o subclases añaden sus propios atributos y operaciones a los de sus clases base. Una clase puede no tener clase hija, en cuyo caso es una *clase hija*. Si una clase tiene sólo un padre, se tiene *herencia simple* y si tiene más de un padre, entonces, se tiene *herencia múltiple*.

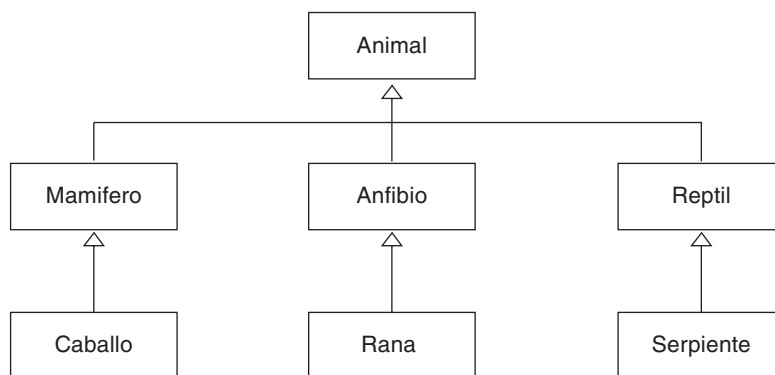


Figura 17.47. Dos niveles en una jerarquía de herencia simple.

EJEMPLO 17.12

Representaciones gráficas de la herencia.

1. Vehículo es una *superclase* (clase base) y tiene como clase derivadas (subclase) Coche (Carro), Barco, Avión y Camión. Se establece la jerarquía Vehículo, es una jerarquía *generalización-especialización*.

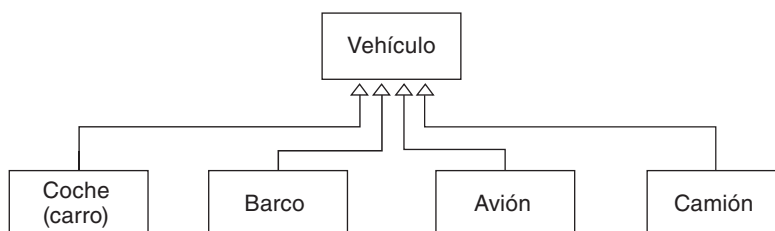


Figura 17.48. Diagrama de clases de la jerarquía Vehículo.

2. Jerarquía Vehículo (segunda representación gráfica, tipo árbol).

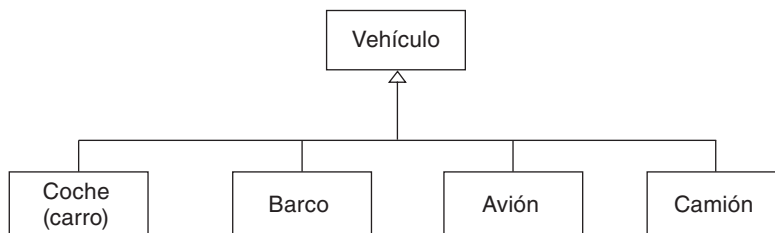


Figura 17.49. Jerarquía Vehículo en forma de árbol.

Evidentemente, la clase base y la clase derivada tienen código y datos comunes, de modo que si se crea la clase derivada de modo independiente, se duplicaría mucho de lo que ya se ha escrito para la clase base. C++ soporta el

mecanismo de *derivación* que permite crear clases derivadas, de modo que la nueva clase *hereda* todos los miembros datos y las funciones miembro que pertenecen a la clase ya existente.

La declaración de derivación de clases debe incluir el nombre de la clase base de la que se deriva y el especificador de acceso que indica el tipo de herencia (*pública, privada y protegida*). La primera línea de cada declaración debe incluir el formato siguiente:

```
class nombre_clase hereda_de tipo_herencia nombre_clase_base
```

Regla

En general, se debe incluir la palabra reservada **publica** en la primera línea de la declaración de la clase derivada, y representa herencia pública. Esta palabra reservada produce que todos los miembros que son públicos en la clase base permanecen públicos en la clase derivada.

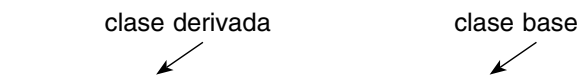
EJEMPLO 17.13

Declaración de las clases Programador y Triangulo.

```
1. class Programador hereda_de Empleado
    publica:
    // miembros públicos
    privada:
    // miembros privados
    fin_clase

2. class Triangulo hereda_de Figura
    publica:
    // sección pública
    ...
    privado:
    // sección privada
    ...
```

Una vez que se ha creado una clase derivada, el siguiente paso es añadir los nuevos miembros que se requieren para cumplir las necesidades específicas de la nueva clase.



```
class derivada      clase base
    ↓               ↓
class Director hereda_de Empleado

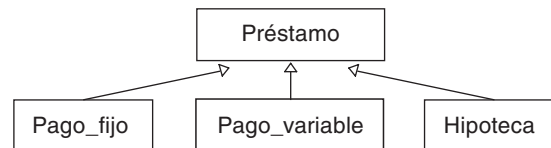
publica:
    nuevas funciones miembro
privada:
    nuevos miembros dato
fin_clase
```

En la definición de la clase *Director* sólo se especifican los miembros nuevos (funciones y datos). Todas las funciones miembro y los miembros dato de la clase *Empleado* son heredados automáticamente por la clase *Director*. Por ejemplo, la función *calcular_salario* de *Empleado* se aplica automáticamente a los directores:

```
Director d;
d.calcular_salario(325000);
```

EJEMPLO 17.14

Considérese una clase *Prestamo* y tres clases derivadas de ella: *Pago_fijo*, *Pago_variable* e *Hipoteca*.



```

class Prestamo
protegida:
    real capital;
    real tasa_interes;
publica:
    Prestamo(float, float);
    virtual int crearTablaPagos(float [MAX_TERM] [NUM_COLUMNAS] = 0;
fin_clase
  
```

Las variables `capital` `tasa_interes` no se repiten en la clase derivada

```

class Pago_fijo hereda_de Prestamo
privada:
    real pago;          // cantidad mensual a pagar por cliente
publica:
    Pago_Fijo(float, float, float);
    ent CrearTablaPagos(float [MAX_TERM] [NUM_COLUMNAS]);
};

class Hipoteca hereda_de Prestamo
privada:
    entero num_recibos;
    entero recibos_por_año;
    real pago;
publica:
    Hipoteca(int, int, float, float, float);
    entero CrearTablaPagos(float [MAX_TERM] [NUM_COLUMNAS]);
fin_clase
  
```

17.6.4. Declaración de una clase derivada

La sintaxis para la declaración de una clase derivada es:

```

class ClaseDerivada hereda_de ClaseBase
publica:
    // sección privada
    ...
privada:
    // sección privada
    ...
fin_clase
  
```

Diagrama de anotaciones:

- Nombre de la clase derivada* → `ClaseDerivada`
- Especificador de acceso (normalmente público)* → `hereda_de`
- Tipo de herencia* → `hereda_de`
- Nombre de la clase base* → `ClaseBase`
- Símbolo de derivación o herencia* → `hereda_de`

Especificador de acceso publica, significa que los miembros públicos de la clase base son miembros públicos de la clase derivada.

Herencia pública, es aquella en que el especificador de acceso es publica (*público*).

Herencia privada, es aquella en que el especificador de acceso es privado (*privado*).

Herencia protegida, es aquella en que el especificador de acceso es protegida (*protegido*).

El especificador de acceso que declara el tipo de herencia es opcional (publica, privada o protegida); si se omite el especificador de acceso, se considera por defecto privada. La *clase base* (*ClaseBase*) es el nombre de la clase de la que se deriva la nueva clase. La *lista de miembros* consta de datos y funciones miembro:

```

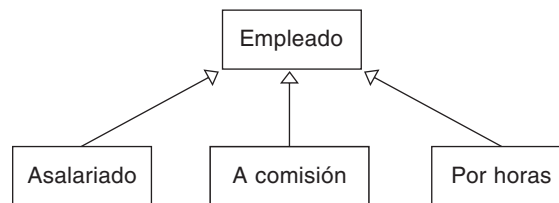
clase nombre_clase hereda_de [especificador_acceso] ClaseBase
    lista_de_miembros;
fin_clase

```

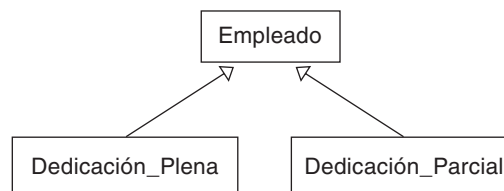
17.6.5. Consideraciones de diseño

A veces es difícil decidir cuál es la relación de herencia más óptima entre clases en el diseño de un programa. Consideremos, por ejemplo, el caso de los empleados o trabajadores de una empresa. Existen diferentes tipos de clasificaciones según el criterio de selección (se suele llamar *discriminador*) y pueden ser: modo de pago (sueldo fijo, por horas, a comisión); dedicación a la empresa (plena o parcial) o estado de su relación laboral con la empresa (fijo o temporal).

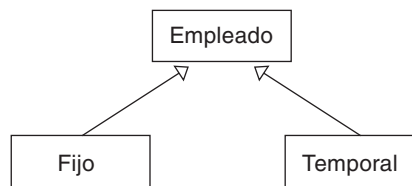
Una vista de los empleados basada en el modo de pago puede dividir a los empleados con salario mensual fijo; empleados con pago por horas de trabajo y empleados a comisión por las ventas realizadas



Una vista de los empleados basada en el estado de dedicación a la empresa: dedicación plena o dedicación parcial.



Una vista de empleados basada en el estado laboral del empleado con la empresa: fijo o temporal.



Una dificultad a la que suele enfrentarse el diseñador es que en los casos anteriores un mismo empleado puede pertenecer a diferentes grupos de trabajadores. Un empleado con dedicación plena puede ser remunerado con un salario mensual. Un empleado con dedicación parcial puede ser remunerado mediante comisiones y un empleado fijo puede

ser remunerado por horas. Una pregunta usual es ¿cuál es la relación de herencia que describe la mayor cantidad de variación en los atributos de las clases y operaciones?, ¿esta relación ha de ser el fundamento del diseño de clases? Evidentemente la respuesta adecuada sólo se podrá dar cuando se tenga presente la aplicación real a desarrollar.

17.7. ACCESIBILIDAD Y VISIBILIDAD EN HERENCIA

En una clase existen secciones públicas, privadas y protegidas. Los elementos públicos son accesibles a todas las funciones; los elementos privados son accesibles sólo a los miembros de la clase en que están definidos y los elementos protegidos pueden ser accedidos por clases derivadas debido a la propiedad de la herencia. En correspondencia con lo anterior existen tres tipos de herencia: *pública*, *privada* y *protegida*. Normalmente el tipo de herencia más utilizada es la herencia pública.

Con independencia del tipo de herencia, una clase derivada no puede acceder a variables y funciones privadas de su clase base. Para ocultar los detalles de la clase base y de clases y funciones externas a la jerarquía de clases, una clase base utiliza normalmente elementos protegidos en lugar de elementos privados. Suponiendo herencia pública, los elementos protegidos son accesibles a las funciones miembro de todas las clases derivadas.

Tabla 17.2. Acceso a variables y funciones según tipo de herencia

Tipo de herencia	Tipo de elemento	¿Accesible a clase derivada?
Pública	pública	sí
	protegida	sí
	privada	no
Privada	pública	no
	protegida	no
	privada	no

Norma

Por defecto, la herencia es privada. Si accidentalmente se olvida la palabra reservada `pública`, los elementos de la clase base serán inaccesibles. El tipo de herencia es, por consiguiente, una de las primeras cosas que se debe verificar si un compilador devuelve un mensaje de error que indique que las variables o funciones son inaccesibles.

17.7.1. Herencia pública

En general, *herencia pública* significa que una clase derivada tiene acceso a los elementos públicos y privados de su clase base. Los elementos públicos se heredan como elementos públicos; los elementos protegidos permanecen protegidos. La herencia pública se representa con el especificador `pública` en la derivación de clases.

Formato

```
class ClaseDerivada hereda_de pública Clase Base
    pública:
        // sección pública
    privada:
        // sección privada
fin_clase
```

17.7.2. Herencia privada

La herencia privada significa que una clase derivada no tiene acceso a ninguno de sus elementos de la clase base. El formato es:

```

class ClaseDerivada hereda_de privada ClaseBase
publica:
    // sección pública
protegida:
    // sección protegida
privada:
    // sección privada
fin_clase

```

Con herencia privada, los miembros públicos y protegidos de la clase base se vuelven miembros privados de la clase derivada. En efecto, los usuarios de la clase derivada no tienen acceso a las facilidades proporcionadas por la clase base. Los miembros privados de la clase base son inaccesibles a las funciones miembro de la clase derivada.

La herencia privada se utiliza con menos frecuencia que la herencia pública. Este tipo de herencia oculta la clase base del usuario y así es posible cambiar la implementación de la clase base o eliminarla toda junta sin requerir ningún cambio al usuario de la interfaz. Cuando un especificador de acceso no está presente en la declaración de una clase derivada, se utiliza herencia privada.

17.7.3. Herencia protegida

Con herencia protegida, los miembros públicos y protegidos de la clase base se convierten en miembros protegidos de la clase derivada y los miembros privados de la clase base se vuelven inaccesibles. La herencia protegida es apropiada cuando las facilidades o aptitudes de la clase base son útiles en la implementación de la clase derivada, pero no son parte de la interfaz que el usuario de la clase ve. La herencia protegida es todavía menos frecuente que la herencia privada.

Tabla 17.3. Tipos de herencia y accesos que permiten

Tipo de herencia	Acceso a miembro clase base	Acceso a miembro clase derivada
publica	publica protegida privada	publica protegida <i>inaccesible</i>
protegida	publica protegida privada	protegida protegida <i>inaccesible</i>
privada	publica protegida privada	privada privada <i>inaccesible</i>

La Tabla 17.3 resume los efectos de los tres tipos de herencia en la accesibilidad de los miembros de la clase derivada. La entrada *inaccesible* indica que la clase derivada no tiene acceso al miembro de la clase base.

EJERCICIO 17.3

Declarar una clase base (*Base*) y tres clases derivadas de ella, *D1*, *D2* y *D3*

```

class Base {
    publica:
        entero i1;
    protegida:
        entero i2;
    privada:
        entero i3;
};

```

```

clase D1: privada Base {
    nada f();
};
clase D2: protegida Base {
    nada g();
};
clase D3: publica Base {
    nada h();
};

```

Ninguna de las subclases tiene acceso al miembro `i3` de la clase `Base`. Las tres clases pueden acceder a los miembros `i1` e `i2`. En la definición de la función miembro `f()` se tiene:

```

void D1::f() {
    i1 = 0;      // Correcto
    i2 = 0;      // Correcto
    i3 = 0;      // Error
};

```

17.8. UN CASO DE ESTUDIO ESPECIAL: HERENCIA MÚLTIPLE

Herencia múltiple es un tipo de herencia en la que una clase hereda el estado (estructura) y el comportamiento de más de una clase base. En otras palabras hay herencia múltiple cuando una clase hereda de más de una clase; es decir, existen múltiples clases base (*ascendientes* o *padres*) para la clase derivada (*descendiente* o *hija*).

La herencia múltiple entraña un concepto más complicado que la herencia simple, no sólo con respecto a la sintaxis sino también al diseño e implementación del compilador. La herencia múltiple también aumenta las operaciones auxiliares y complementarias y produce ambigüedades potenciales. Además, el diseño con clases derivadas por derivación múltiple tiende a producir más clases que el diseño con herencia simple. Sin embargo, y pese a los inconvenientes y ser un tema controvertido, la herencia múltiple puede simplificar los programas y proporcionar soluciones para resolver problemas difíciles. En la Figura 17.50 se muestran diferentes ejemplos de herencia múltiple.

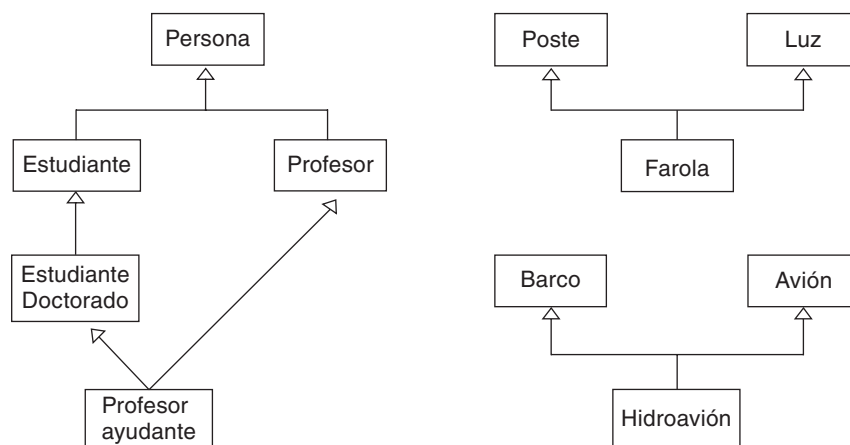


Figura 17.50. Ejemplos de herencia múltiple.

Regla

En herencia simple, una clase derivada hereda exactamente de una clase base (tiene sólo un padre). Herencia múltiple implica múltiples clases base (tiene varios padres una clase derivada).

En herencia simple, el escenario es bastante sencillo, en términos de concepto y de implementación. En herencia múltiple los escenarios varían ya que las clases base pueden proceder de diferentes sistemas y se requiere a la hora de la implementación un compilador de un lenguaje que soporte dicho tipo de herencia (C++ o Eiffel). ¿Por qué utilizar herencia múltiple? Pensamos que la herencia múltiple añade fortaleza a los programas y si se tiene precaución en la base del análisis y posterior diseño, ayuda bastante a la resolución de muchos problemas que tomen naturaleza de herencia múltiple.

Por otra parte, la herencia múltiple siempre se puede eliminar y convertirla en herencia simple si el lenguaje de implementación no la soporta o considera que tendrá dificultades en etapas posteriores a la implementación real. La sintaxis de la herencia múltiple es:

```
class CDerivada hereda_de Base1, Base2,...

publica:
    // sección pública
privada:
    // sección privada
...
fin_clase

CDerivada                Nombre de la clase derivada
Base1, Base2,...          Clases base con nombres diferentes
```

Funciones o datos miembro que tengan el mismo nombre en Base1, Base2, Basen,... serán motivo de ambigüedad.

```
class A hereda_de publica B, C {...}
class D hereda_de publica E, publica F, publica G {...}
```

La palabra reservada *publica* ya se ha comentado anteriormente, define la relación “*es-un*” y crea un subtipo para herencia simple. Así en los ejemplos anteriores, la clase A “*es-un*” tipo de B y “*es-un*” tipo de C. La clase D se deriva públicamente de E y G y privadamente de F. Esta derivación hace a D un subtipo de E y G pero no un subtipo de F. *El tipo de acceso sólo se aplica a una clase base.*

```
class Derivada hereda_de publica Base1, Base2 {...};
```

Derivada específica derivación pública de Base1 y derivación privada (por defecto u omisión) de Base2,

Regla

Asegúrese de especificar un tipo de acceso en todas las clases base para evitar el acceso privado por omisión. Utilice explícitamente *privada* cuando lo necesite para manejar la legibilidad.

```
Class Derivada: publica Base1, privada Base2 {...}
```

EJEMPLO 17.15

```
class Estudiante {
...
};
class Trabajador {
...
};
class Estudiante_Trabajador: publica Estudiante, publica Trabajador {
...
};
```

17.8.1. Características de la herencia múltiple

La herencia múltiple plantea diferentes problemas tales como la *ambigüedad* por el uso de nombres idénticos en diferentes clases base, y la *dominación* o *preponderancia* de funciones o datos.

Ambigüedades

Al contrario que la herencia simple, la herencia múltiple tiene el problema potencial de las ambigüedades.

EJEMPLO 17.16

```

clase Ventana {
privada:
...
publica:
    nada dimensionar();    // dimensiona una ventana
...
fin_clase

clase Fuente {
privada:
...
publica:
    nada dimensionar();    // dimensiona un tipo fuente
...
fin_clase

```

Una clase Ventana tiene una función `dimensionar()` que cambia el tamaño de la ventana; de modo similar, una clase Fuente modifica los objetos Fuente con `dimensionar()`. Si se crea una clase `Ventana_Fuente` (VFuente) con herencia múltiple, se puede producir ambigüedad en el uso de `dimensionar()`

```

clase VFuente: publica Ventana, publica Fuente {...};
VFuente v;
v.dimensionar();    // se produce un error ¿cuál?

```

La llamada a `dimensionar` es ambigua, ya que el compilador no sabrá a qué función `dimensionar` ha de llamar. Esta ambigüedad se resuelve fácilmente con el operador de resolución de ámbito (`::`)

```

v.Fuente::dimensionar();    // llamada a dimensionar() de Fuente
v.Ventana::dimensionar();    // llamada a dimensionar de Ventana

```

Precaución

No es un error definir un objeto derivado con multiplicidad con ambigüedades. Estas se consideran ambigüedades potenciales y sólo produce errores en tiempo de compilación cuando se llaman de modo ambiguo.

Regla

Incluso es mejor solución que la citada anteriormente resolver la ambigüedad en las propias definiciones de la función `dimensionar()`

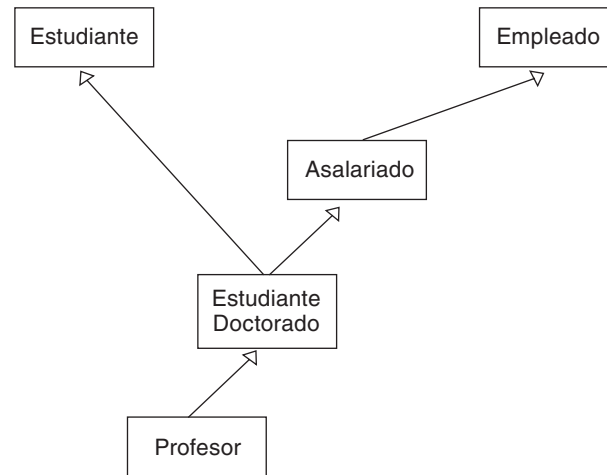
```

clase VFuente: publica Ventana, publica Fuente
...
void v_dimensionar() { Ventana::dimensionar(); }
void f_dimensionar() { Fuente::dimensionar(); }
fin_clase

```

EJEMPLO 17.17

Diseñar e implementar una jerarquía de clases que represente las relaciones entre las clases siguientes: estudiante, empleado, empleado asalariado y un estudiante de doctorado que es a su vez profesor de prácticas de laboratorio.

**Nota**

Se deja la resolución como ejercicio al lector.

17.9. CLASES ABSTRACTAS

Una clase abstracta es una clase que no tiene ningún objeto; o con mayor precisión, es una clase que no puede tener objetos instancias de la clase base. Una clase abstracta describe atributos y comportamientos comunes a otras clases, y deja algunos aspectos del funcionamiento de la clase a las subclases concretas. Una clase abstracta se representa con su nombre en cursiva.

EJERCICIO 17.4

Clase abstracta *Vehículo* con clases derivadas *Coche (Carro)* y *Barco*.

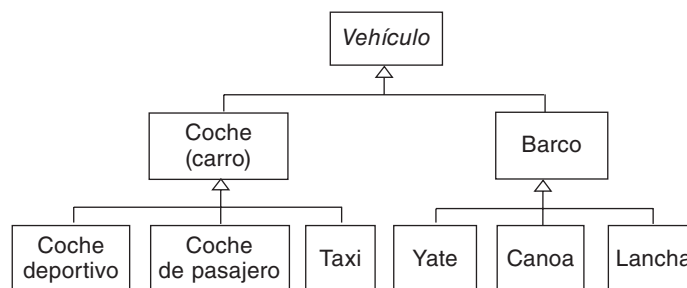


Figura 17.51. Jerarquía con clase base abstracta *Vehículo*.

Una clase abstracta se representa poniendo su nombre en cursiva o añadiendo la palabra {abstract} dentro del compartimento de la clase y debajo del nombre de la clase.

EJERCICIO 17.5

Clase abstracta *Futbolista* de la cual derivan las clases concretas *Portero*, *Defensa* y *Delantero*.

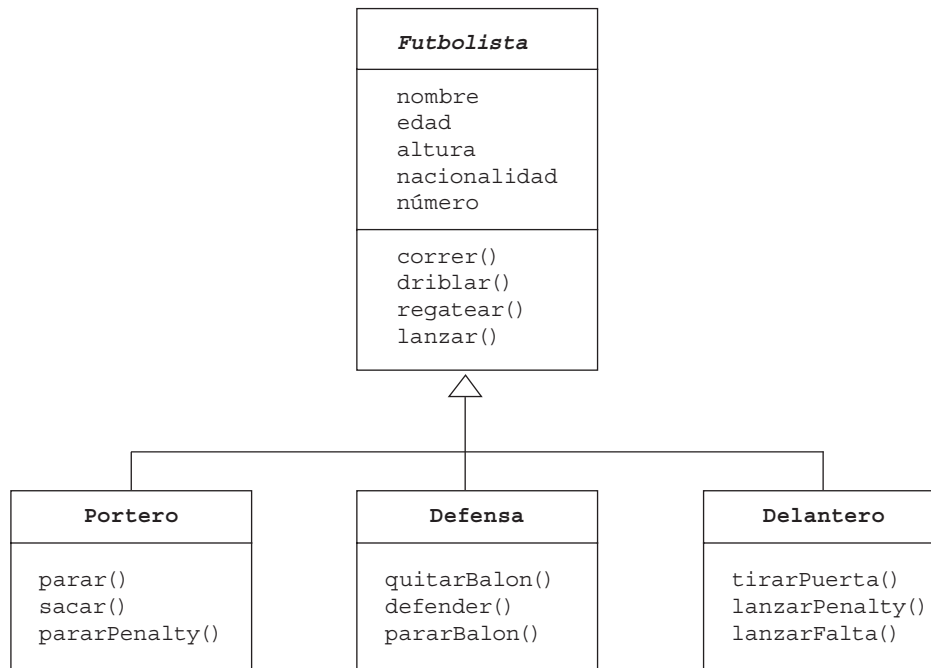


Figura 17.52. Jerarquía con clase base abstracta *Futbolista*.

17.9.1. Operaciones abstractas

Una clase abstracta tiene operaciones abstractas. Una *operación abstracta* no tiene implementación de métodos, sólo la signatura o prototipo. Una clase que tiene al menos una operación abstracta es, por definición, abstracta.

Una clase que hereda de una clase que tiene una o más operaciones abstractas debe implementar esas operaciones (proporcionar métodos para esas operaciones). Las operaciones abstractas se muestran con la cadena **{abstract}** a continuación del prototipo o signatura de la clase. Las operaciones abstractas se definen en las clases abstractas para especificar el comportamiento que deben tener todas las subclases. Una clase *Vehículo* debe tener operaciones abstractas que especifiquen comportamientos comunes de todos los vehículos (conducir, frenar, arrancar...).

Los modeladores suelen proporcionar siempre una capa de clases abstractas como superclases, buscando elementos comunes a cualquier relación de herencia que se puede extender a las clases hijas. En el ejemplo de las clases abstractas, *Coche* y *Barco* representan a clases que requieren implementar las operaciones abstractas “conducir” y “frenar”.

Una *clase concreta* es una clase opuesta a la clase abstracta. En una clase concreta, es posible crear objetos de la clase que tienen implementaciones de todas las operaciones. Si la clase *Vehículo* tiene especificada una operación abstracta conducir, tanto las clases *Coche* como *Barco* deben implementar ese método (o las propias operaciones deben ser especificadas como abstractas). Sin embargo, las implementaciones son diferentes. En un coche, la operación conducir hace que las ruedas se muevan; mientras que conducir un barco hace que el barco navegue (se mueva). Las subclases heredan operaciones de una superclase común, pero dichas operaciones se implementan de modo diferente.

Una subclase puede redefinir (modificar el comportamiento de la superclase) las operaciones de la superclase, o bien implementan la superclase tal y como está definida. Una operación redefinida debe tener la misma signatura o prototipo (tipo de retorno, nombre y parámetros) que la superclase. La operación que se está redefiniendo puede ser o bien *abstracta* (no tiene implementación en la superclase) o *concreta* (tiene una implementación en la superclase). En cualquier caso, la redefinición en las subclases se utiliza para todas las instancias de esa clase.

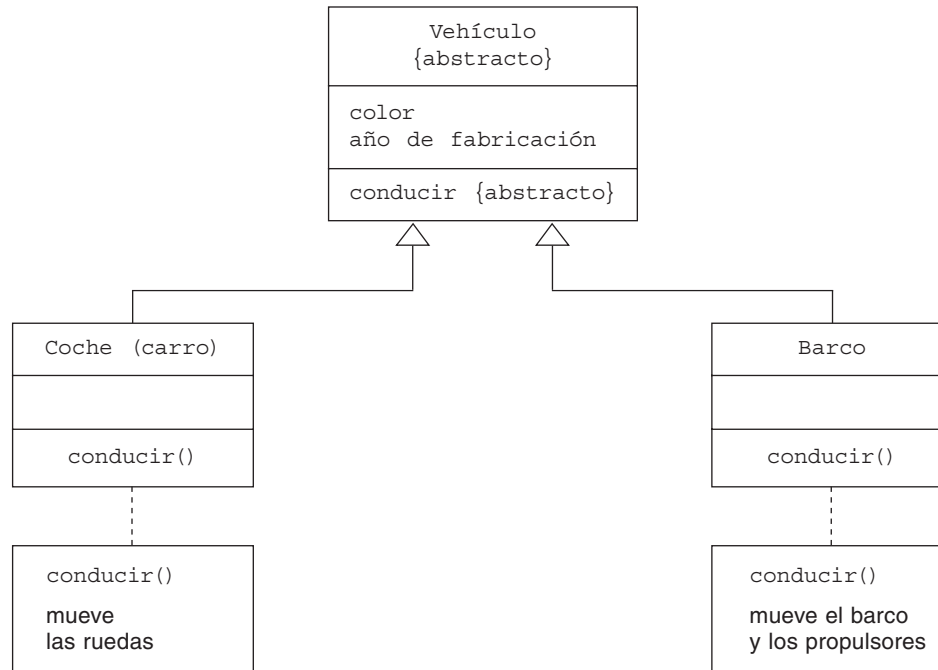


Figura 17.53. La clase `Vehículo` (abstracta) hereda los atributos `color` y `añoDeFabricación`, y la operación `conducir`.

Se pueden añadir a las subclases nuevas operaciones, atributos y asociaciones. Un objeto de una subclase se puede utilizar en cualquier situación donde sea posible utilizar objetos de la superclase. En ese caso, la subclase tendrá una implementación diferente dependiendo del objeto implicado.

CONCEPTOS CLAVE

- Agregación.
- Asociación.
- Clase abstracta.
- Clase base.
- Clase derivada.
- Composición.
- Constructor.
- Declaración de acceso.
- Destructor.
- Especificadores de acceso.
- Función virtual.
- Generalización.
- Herencia.
- Herencia múltiple.
- Herencia protegida.
- Herencia pública y privada.
- Herencia simple.
- Ligadura dinámica.
- Ligadura estática.
- Multiplicidad.
- Polimorfismo.
- Relación *es-un*.
- Relación *todo-parte*.

RESUMEN

Una asociación es una conexión semántica entre clases. Una asociación permite que una clase conozca de los atributos y operaciones públicas de otra clase.

Una agregación es una relación más fuerte que una asociación y representa una clase que se compone de otras clases. Una agregación representa la relación *todo-parte*; es decir una clase es el todo y contiene a todas las partes.

Una generalización es una relación de herencia entre dos elementos de un modelo tal como clases. Permite a una

clase heredar atributos y operaciones de otra clase. Su implementación en un lenguaje orientado a objetos es la herencia. La especialización es la relación opuesta a la generalización.

La relación **es-un** representa la herencia. Por ejemplo, una rosa es un tipo de flor; un pastor alemán es un tipo de perro, etc. La relación *es-un* es transitiva. Un pastor alemán es un tipo de perro y un perro es un tipo de mamífero; por consiguiente, un pastor alemán es un mamífero. Una clase

nueva que se crea a partir de una clase ya existente, utilizando herencia, se denomina clase derivada o subclase. La clase padre se denomina clase base o superclase.

1. *Herencia* es la capacidad de derivar una clase de otra clase. La clase inicial utilizada por la clase derivada se conoce como *clase base*, *padre* o *superclase*. La clase derivada se conoce como *derivada*, *hija* o *subclase*.
2. *Herencia simple* es la relación entre clases que se produce cuando una nueva clase se crea utilizando

las propiedades de una clase ya existente. Las relaciones de herencia reducen código redundante en programas. Uno de los requisitos para que un lenguaje sea considerado orientado a objetos es que soporte herencia.

3. La *herencia múltiple* se produce cuando una clase se deriva de dos o más clases base. Aunque es una herramienta potente, puede crear problemas, especialmente de colisión o conflicto de nombres, cosa que se produce cuando nombres idénticos aparecen en más de una clase base.

EJERCICIOS

- 17.1.** Definir una clase base *Persona* que contenga información de propósito general común a todas las personas (nombre, dirección, fecha de nacimiento, sexo, etc.). Diseñar una jerarquía de clases que contemple las clases siguientes: *Estudiante*, *Empleado*, *Estudiante_empleado*. Escribir un programa que lea un archivo de información y cree una lista de personas: a) general; b) estudiantes; c) empleados; d) estudiantes empleados. El programa debe permitir ordenar alfabéticamente por el primer apellido.

- 17.2.** Implementar una jerarquía *Librería* que tenga al menos una docena de clases. Considérese una *librería* que tenga colecciones de libros de literatura, humanidades, tecnología, etc.

- 17.3.** Diseñar una jerarquía de clases que utilice como clase base o raíz una clase *LAN* (red de área local). Las subclases derivadas deben representar diferentes topologías, como *estrella*, *anillo*, *bus* y *hub*. Los miembros datos deben representar propiedades tales como *soporte de transmisión*, *control de acceso*, *formato del marco de datos*, *estándares*, *velocidad de transmisión*, etc. **Se desea simular la actividad de los nodos de tal LAN.**

La red consta de **nodos**, que pueden ser dispositivos tales como computadoras personales, estaciones de trabajo, máquinas FAX, etc. Una tarea principal de LAN es soportar comunicaciones de datos entre sus nodos. El usuario del proceso de simulación debe, como mínimo, poder:

- Enumerar los nodos actuales de la red LAN.
- Añadir un nuevo nodo a la red LAN.
- Quitar un nodo de la red LAN.
- Configurar la red, proporcionándole una topología de *estrella* o en *bus*.
- Especificar el tamaño del paquete, que es el tamaño en bytes del mensaje que va de un nodo a otro.
- Enviar un paquete de un nodo especificado a otro.

- Difundir un paquete desde un nodo a todos los demás de la red.
- Realizar estadísticas de la LAN, tales como tiempo medio que emplea un paquete.

- 17.4.** Implementar una jerarquía *Empleado* de cualquier tipo de empresa que le sea familiar. La jerarquía debe tener al menos cuatro niveles, con herencia de miembros datos, y métodos. Los métodos deben poder calcular salarios, despidos, promoción, dar de alta, jubilación, etc. Los métodos deben permitir también calcular aumentos salariales y primas para Empleados de acuerdo con su categoría y productividad. La jerarquía de herencia debe poder ser utilizada para proporcionar diferentes tipos de acceso a Empleados. Por ejemplo, el tipo de acceso garantizado al público diferirá del tipo de acceso proporcionado a un supervisor de empleado, al departamento de nóminas, o al Ministerio de Hacienda. Utilice la herencia para distinguir entre al menos cuatro tipos diferentes de acceso a la información de Empleado.

- 17.5.** Implementar una clase *Automovil* (*Carro*) dentro de una jerarquía de herencia múltiple. Considere que, además de ser un *Vehículo*, un automóvil es también una *comodidad*, un *símbolo de estado social*, un *modo de transporte*, etc. *Automovil* debe tener al menos tres clases base y al menos tres clases derivadas.

- 17.6.** Escribir una clase *FigGeometrica* que represente figuras geométricas tales como *punto*, *línea*, *rectángulo*, *triángulo* y similares. Debe proporcionar métodos que permitan dibujar, ampliar, mover y destruir tales objetos. La jerarquía debe constar al menos de una docena de clases.

- 17.7.** Implementar una jerarquía de tipos de datos numéricos que extienda los tipos de datos fundamentales

tales como `int` y `float`, disponibles en C++. Las clases a diseñar pueden ser `Complejo`, `Fracción`, `Vector`, `Matriz`, etc.

- 17.8.** Implementar una jerarquía de herencia de animales tal que contenga al menos seis niveles de derivación y doce clases.

- 17.9.** Diseñar la siguiente jerarquía de clases:

<i>Persona</i>			
Nombre			
edad			
visualizar()			
<i>Estudiante</i>		<i>Profesor</i>	
nombre	heredado	nombre	heredado
edad	heredado	edad	heredado
id	definido	salario	definido
visualizar()	<i>redefinido</i>	visualizar()	<i>heredada</i>

Escribir un programa que manipule la jerarquía de clases, lea un objeto de cada clase y lo visualice.

- 17.10.** Crear una clase base denominada `Punto` que conste de las coordenadas `x` e `y`. A partir de esta clase, definir una clase denominada `Circulo` que tenga

las coordenada del centro y un atributo denominado `radio`. Entre las funciones miembro de la primera clase, deberá existir una función `distancia()` que devuelva la distancia entre dos puntos, donde:

$$\text{Distancia} = ((x_2 - x_1)^2 + (y_2 - y_1)^2)^{1/2}$$

- 17.11.** Utilizando la clase construida en el Ejercicio 17.10 obtener una clase derivada `Cilindro` derivada de `Circulo`. La clase `Cilindro` deberá tener una función miembro que calcule la superficie de dicho cilindro. La fórmula que calcula la superficie del cilindro es $S = 2r(l + r)$ donde r es el radio del cilindro y l es la longitud.

- 17.12.** Crear una clase base denominada `Rectangulo` que contenga como miembros `datos`, `longitud` y `anchura`. De esta clase, derivar una clase denominada `Caja` que tenga un miembro adicional denominado `profundidad` y otra función miembro que permita calcular su volumen.

- 17.13.** Dibujar un diagrama de objetos que represente la estructura de un coche (carro). Indicar las posibles relaciones de asociación, generalización y agregación.