INTRODUCCIÓN A LAS RELACIONES

Hasta este momento **hemos estudiado una de las dos partes de lo que constituye la abstracción en el mundo del paradigma orientado a objetos: La definición y modelado de clases** con sus respectivos atributos y operaciones.

Las aplicaciones orientadas objetos raramente definen clases cuyos objetos trabajen de manera aislada. Un buen diseño orientado a objetos utilizará objetos de diferentes clases que cooperan para alcanzar el objetivo central del dominio del negocio de la aplicación.

Por este motivo, en esta sección abordaremos **el otro elemento central de la abstracción en el paradigma orientado a objetos: la forma en que los objetos colaboran** entre ellos para realizar alguna tarea específica y como representarla en el diagrama de clases; conocida también con el nombre de **relaciones entre clases**. Se definirán los diferentes tipos de relaciones y se brindará ejemplos clarificadores.

RELACIONES ENTRE CLASES

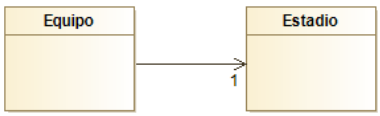
Las relaciones permiten establecer cómo interactúan las clases. En UML se representan mediante una línea. Existen diferentes tipos de relaciones:

- ✓ Asociación
- ✓ Agregación
- ✓ Dependencia
- ✓ Implementación
- ✓ Herencia

La asociación

Permite establecer una relación conceptual entre dos clases. Especifica una conexión entre los objetos de una clase y los objetos de otra clase. Describiremos las características de las asociaciones mediante ejemplos.

Ejemplo 1:



La relación de asociación se establece mediante una línea. La idea detrás de esta línea es indicar que se **establece una relación no jerárquica entre dos clases**. Es decir, ninguna de las dos tiene un estatus superior respecto de la otra (como sucede y se verá en la herencia, las clases abstractas y las interfaces). Por este motivo **también se la conoce como una relación de igual a igual, o de mano a mano**. Es la relación que más se utiliza en los modelos de clases.

La asociación debe expresar en alguno (o en ambos) de los extremos una flecha o un triángulo que hace de flecha, denominado **navegador**, y se dice que la asociación es dirigida o direccional.

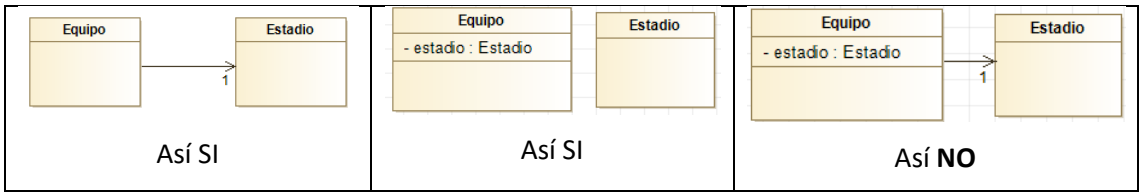
El extremo con la flecha es el **objetivo u objeto** hacia el que se puede navegar. El extremo sin la flecha se denomina **fuente**.

Al menos que se exprese una personalización del significado conceptual de la asociación, se asume que la fuente" **tiene o conoce"** un atributo cuyo tipo es de la otra clase; esto aplicado a nuestro ejemplo sería: **un equipo tiene un solo estadio**. El significado conceptual también recibe el nombre de **frase semántica**.

¿Por qué estoy tan seguro de que un equipo tiene un estadio, y no dos o ninguno? Porque es posible indicar en los extremos de una asociación la multiplicidad; es decir ese número 1 que está del lado del objetivo me indica que la fuente apunta únicamente a un objeto objetivo.

¿Esto no les recuerda a que ya fue visto cuando definíamos atributos? Absolutamente, si. La multiplicidad de un atributo se puede expresar en su definición o como se indica en este caso, mediante una relación.

¿Cuándo usar una y cuando usar otra? **En general se prefiere usar las relaciones, para que quede explícitamente expresada la relación entre clases**, sin embargo, si posee un diagrama de clases extenso, una gran cantidad de relaciones puede dificultar su lectura. En estos casos, se preferirá expresar la multiplicidad en el atributo. Lo que debe quedar claro es que son autoexcluyentes. Es decir, **la multiplicidad de un atributo o es expresada en el atributo o es expresada en la relación**. Gráficamente, esto es



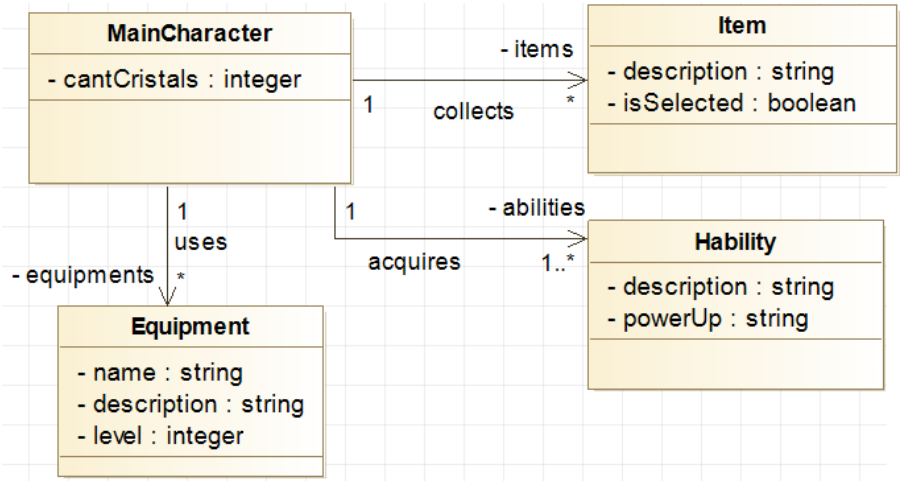
Puede observar que la relación de asociación brinda mucha información. Por ejemplo, **Equipo** tiene un atributo (aunque no se lo vea definido explícitamente) y se denomina estadio.

Esto se debe a que la relación de asociación es una implementación de punteros: un objeto de una clase apunta a uno o varios objetos de otra clase. Como se mencionó anteriormente, la cantidad de objetos a los que apunta la fuente se define mediante la **multiplicidad**. Hay varios indicadores de multiplicidad. A continuación, se los indican en la siguiente tabla:

1	Solo 1
*	Muchos
0..1	Cero o uno
0 ..*	Cota inferior a cero y cota superior a infinito. Es equivalente a *
1..1	Uno y solo uno. Equivalente a 1
1..*	Cota inferior de al menos 1 y cota superior a infinito.
m, n	Indicación de una multiplicidad no contigua, por ejemplo 3, 5 indica 3 o 5. Esta relación ya no es válida en UML

¿Es posible modificar la frase semántica de la asociación? Si, veámoslo con un ejemplo que encara esta situación y a la vez refuerza el modo de usar la multiplicidad

Ejemplo 2: la famosa lista de power ups que puede tener un personaje. En este caso nos remitiremos a Legend of Zelda: A Link to the Past. Observe la siguiente imagen e intente modelar las relaciones entre Link y el conjunto de power ups que puede usar en el juego:



En este modelo podemos observar muchos aspectos muy importantes a considerar para quienes empiezan en el modelado de clases. En primer lugar, observe que la clase de la cual se obtiene un objeto que representa a Link se denominada MainCharacter. **Es un error muy común denominar a la clase con el nombre del personaje**, es decir definir una clase que se llamaría en este caso Link; pero **debe recordar que una clase es un molde del cual se generan objetos**. El **proceso por el cual a partir de una clase se genera un objeto se denomina instanciación**; así un objeto es una instancia de una clase. Por tanto, no sería correcto que la clase se llame como el objeto.

En segundo lugar, observe **que es posible cambiar la frase semántica por defecto de una asociación para expresar con mayor detalle la relación conceptual entre dos clases**. Por ejemplo, entre MainCharacter e Item se expresa la relación con la palabra "collects"; es decir "*objetos de la clase MainCharacter coleccionan objetos de la clase Item*".

Esta relación es refinada mediante la multiplicidad para indicar que "*Un objeto de la clase MainCharacter colecciona muchos objetos de la clase Item, pudiendo al inicio no poseer ningún item*", esto significa que **el * del lado de Item en realidad es equivalente a 0..*.**

Además, la relación se puede refinar aún más mediante la especificación del rol de los objetos en la relación. En este ejemplo, observe que el **rol especifica el nombre del atributo que posee o tiene la clase fuente**, que en este caso es ítems. Entonces la relación queda formalmente "*Un objeto de la clase MainCharacter colecciona 0 o muchos ítems*".

Así las otras relaciones de asociación presentes en este ejemplo son:

- Un objeto de la clase MainCharacter usa 0 o muchos equipamientos
- Un objeto de la clase MainCharacter adquiere 1 o muchas habilidades

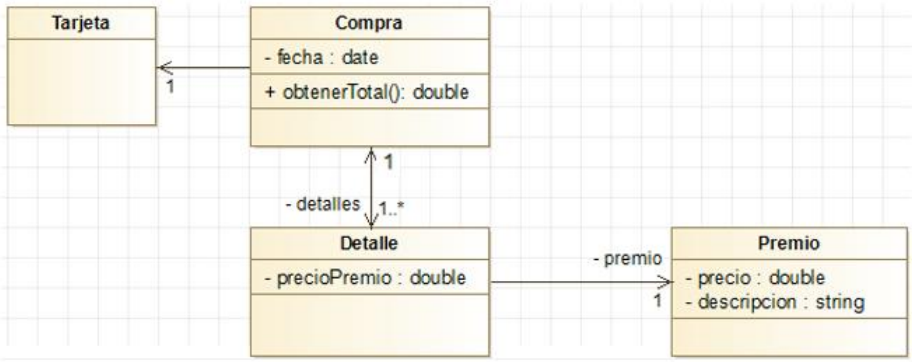
Observe que esta última relación es diferente a las anteriores con respecto a la multiplicidad, ya que indica que Link tendrá por defecto al menos una habilidad (que en el juego es su capacidad de hablar con los demás personajes del juego).

Observe que se debe armar clases y relaciones entre clases cuando sean absolutamente necesarias. Por ejemplo, en este ejemplo MainCharacter no se relaciona con una clase Cristal, sino que simplemente se indica que Link posee una cierta cantidad de cristales mediante su atributo cantCristalls. Dado que los cristales no otorgan habilidades ni modifican el desempeño de Link, y además no poseen características u operaciones específicas; no requieren ser agrupados en una entidad denominada Cristal.

¿Pueden las asociaciones ser bidireccionales?

Si, esto **se aplica esencialmente cuando se desea mantener un registro de la asociación**. En el caso de los juegos se puede referir a la compra de premios mediante el uso de tarjeta. En estos casos resulta interesante que tanto el comprador como el vendedor del premio tengan un registro de la transacción (operación de compra).

Ejemplo 3: Intente interpretar este diagrama de clases

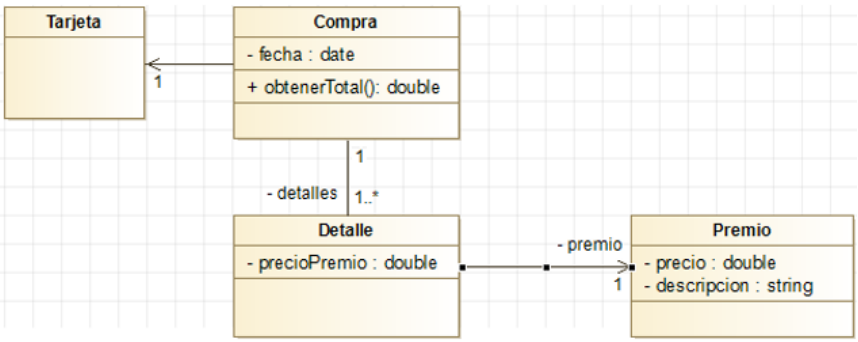


Puede observar que la clase principal es **Compra**, la cual mantiene un atributo de un objeto de tipo **Tarjeta**. De la clase **Premio** se pueden instanciar objetos, tales como aquellos que día a día sugiere el juego que se adquieran (o aún incluso cuando no lo jugamos, especialmente en juegos de celulares). Al autorizar una compra se guarda la fecha y… aquí viene lo interesante, **Compra mantiene un conjunto de detalles** (que sería el registro de cada una de las compras realizadas), pero a la vez cada uno de esos detalles mantiene un registro de la compra a la que pertenece; por tanto, es una relación bidireccional. ¿Por qué no se utiliza una relación de asociación desde Compra a Premio directamente? Porque cada objeto de tipo Premio solo puede ser comprado por un cliente únicamente (algo similar a cuando compra un producto en un negocio: si bien hay varios del mismo tipo en la consola o mostrador, ud selecciona alguno de ellos y es ese el producto que adquiere). Además, el precio del premio puede cambiar en cualquier momento, sin embargo, cuando se realiza la compra, ese precio se registra y no se cambia. Entonces, **Detalle** se vuelve muy importante, porque permite llevar el registro a cuál compra pertenece, cual es el premio comprado y con que precio adquirió el premio en el momento de la compra.

Por otra parte, un jugador podría comprar un premio y no usarlo hasta el momento que lo requiera. Entonces ¿de dónde va a extraer el premio comprado si no es del detalle?

Entonces la bidirección se logra con los navegadores a ambos extremos ¿Hay otra forma de representarla?

Si, curiosamente si no coloca la navegación en ninguno de los extremos será equivalente a una bidirección, es decir para el ejemplo anterior este sería su diagrama equivalente:



Para practicar: Observe las siguientes imágenes y/o narrativas y modele el diagrama de clases correspondiente.

Ejemplo 4: Juego Vendetta

En este juego pueden interactuar hasta cuatro jugadores. Cada peleador puede golpear (con las manos), patear y saltar. Una característica común en este tipo de juegos es la posibilidad de usar armas que se encuentran tiradas en el lugar o que dejan los enemigos al ser golpeados. Para recogerlos simplemente el peleador debe pasar por encima del arma. Al recoger un arma, el peleador golpeará con ella en lugar de usar sus manos. Una característica de este juego es que esas armas tienen una cantidad de veces a ser usadas. Entonces si un garrote es utilizado por un personaje y este lo pierde (ya sea porque lo tira o porque lo golpean), puede ser recogido por un enemigo o por un compañero; como consecuencia quien lo use, solo podrá usarlo la cantidad de veces que le quede disponible.

Ejemplo 5: Observe las siguientes imágenes. Es del juego arcade Carrier Air Wing (aunque su versión japonesa se denominaba US Navy, y a mi entender mucho mejor que la de EEUU). Modele las clases presentes y establezca las relaciones de asociación que aparecen en la misma.

La agregación

La agregación es una relación de asociación, pero jerárquica, en el sentido de que se establece que **una de las clases representa un "Todo" mientras que la otra clase representa a las "Partes"** del todo. La agregación no es diferente de una asociación y se suele preferir referenciarla como asociación.

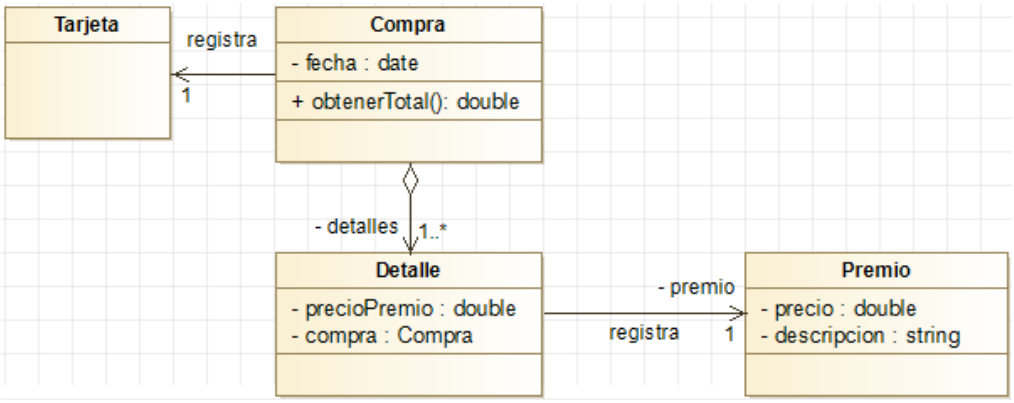
Sin embargo, dado que el propósito de los modelos es reflejar una realidad de la manera más fiel posible dentro de los parámetros de abstracción que se adopten, UML ofrece esta relación para indicar explícitamente la relación conceptual entre un "todo" y sus "partes", aunque luego en la implementación se interpreten de la misma manera.

En definitiva, esta relación representa el hecho de que una clase puede constar de otras clases. **La jerarquía se expresa indicando la clase "todo" en la parte superior, y los componentes "partes" por debajo de ella**. Algunos autores no consideran lo anterior, debido a que la relación indica la clase "todo" mediante un rombo o diamante denominado **conector**. Nuevamente se vuelve a destacar que las clases agregadas son "**atributos**" de la clase completa y, por lo tanto, no se las expresa en manera explícita porque la relación gráfica se encarga de ello.

Del lado de las "partes" la relación puede o no finalizar en una flecha. Además, **como la relación "todo" siempre se refiere a un elemento, no se coloca multiplicidad de este lado**, mientras que del lado "partes" se debe indicar explícitamente la multiplicidad.

La frase semántica que referencia una relación de agregación es "**está formado/a por**". Esta frase debería ayudar a confirmar o desestimar si una relación de asociación representa una agregación.

Ejemplo 6: Analice el diagrama de clases del ejemplo 3 y discierna si las relaciones de asociación indicadas confieren una agregación, en cuyo caso actualice el diagrama de clases.

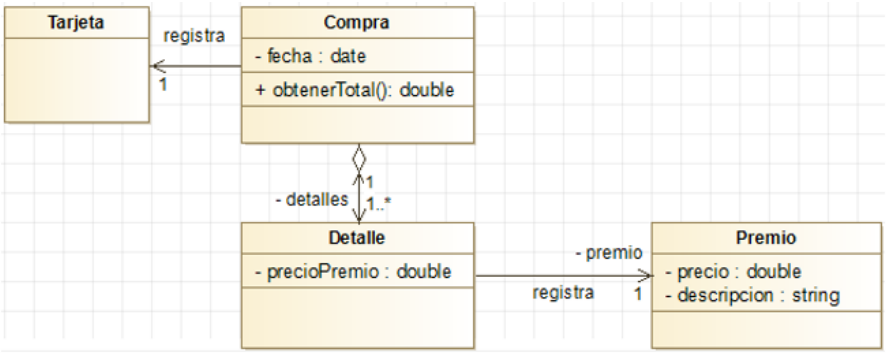


Bien, el diagrama de clases anterior se ha actualizado de la siguiente manera:

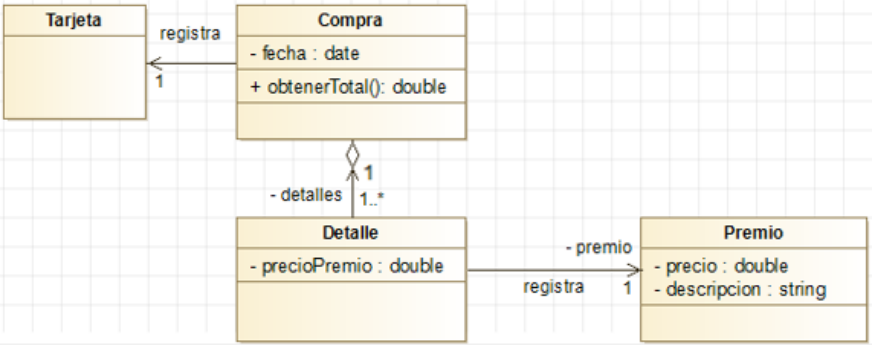
- Entre Compra y Tarjeta hay una asociación. La compra ¿está formada por la tarjeta? No. Una compra no está física o conceptualmente formada por una tarjeta, sino que registra los datos de la tarjeta.
- Entre Compra y Detalle hay una asociación. La compra ¿está formada por detalles? Si, una transacción (operación de compra) necesariamente "está formada" por detalles, que son los que registran los premios comprados. Por tanto, aquí, se coloca del lado "todo" que es Compra el conector. Así, se debe leer "Un objeto compra está formado por 1 o muchos detalles". Observe además que en el conector **no** se coloca multiplicidad.

- Finalmente, entre Detalle y Premio también hay una asociación. Como se indicó previamente, en un detalle se registran los premios adquiridos, pero esto no significa que física o conceptualmente un detalle "esté formado" por un premio.

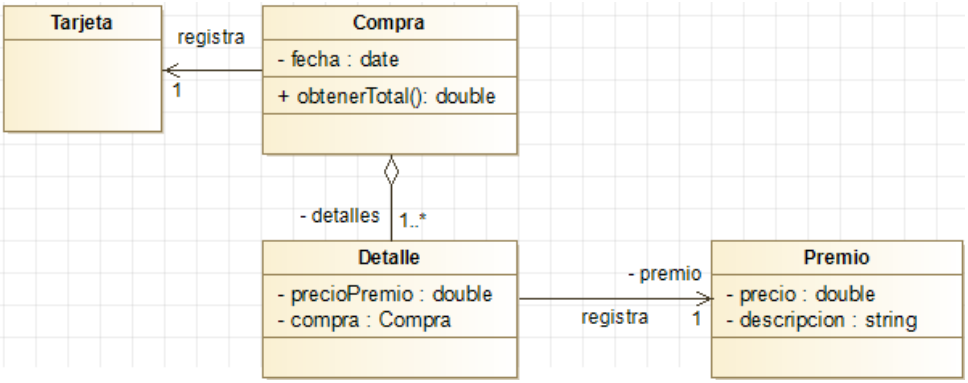
También observe que, con el objetivo de mantener la bidirección de la navegación, se ha agregado un atributo denominado *compra* en Detalle, para explicitar la relación de bidirección. Esto se ha realizado con el objetivo de mantener la claridad de lectura, aunque también se podría haber realizado de la siguiente manera



Así, no hace falta colocar el atributo de tipo Compra en la clase Detalle, pero tenga cuidado de malinterpretar la multiplicidad 1 que está debajo del conector: no pertenece a este; sino que indica que un detalle usa o conoce su compra. También, **debido a que la navegabilidad del lado de las "partes" es opcional**, el siguiente diagrama de clases también sería válido

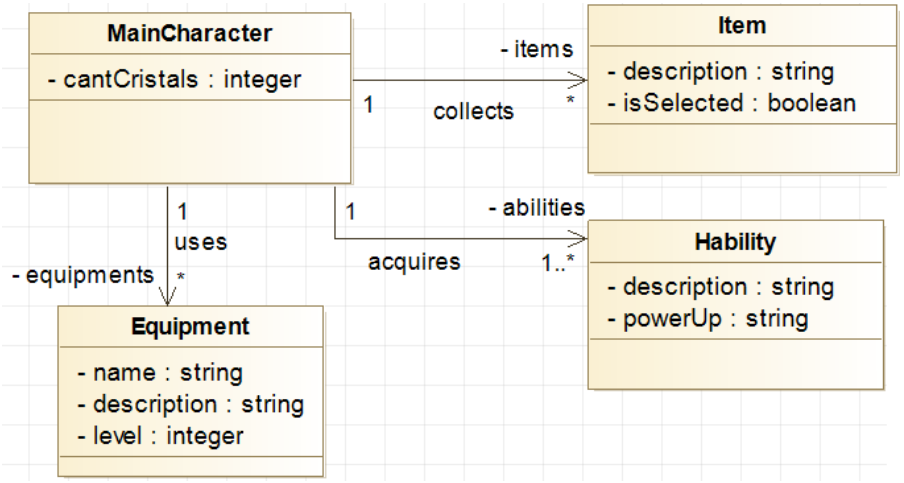


Donde se mantiene la lectura de que "un objeto de tipo Compra está formado por uno o más detalles" y que "un detalle forma parte de una compra". Además, un detalle usa o conoce su compra. Esta forma, aunque parece clara podría causar una mala interpretación, tal como sucedía con el caso anterior, por lo que se suele en realidad usar esta otra forma de expresión



Mientras se cumpla la nomenclatura particular, puede utilizar cualquiera de estas formas.

Ejemplo 7: Analice ejemplo 2 e indique si se debiese aplicar agregación en algunas de las relaciones creadas.



Como puede observar no se ha introducido ningún cambio al diagrama de clases generado anteriormente. Esto se debe a que la frase semántica de la agregación "**forma parte de** " o "**está formado por**" expresa claramente una jerarquía entre clases. Si las analizamos en detalle el diagrama evaluado observaremos lo siguiente:

- ¿Un personaje principal (mainCharanter) está formado por 0 o muchos ítems? Absolutamente no; de hecho, por algo pudimos explicitar la relación conceptual entre MainCharacter e Item mediante "colecciona". Suena mucho más coherente que el personaje principal coleccione 0 o muchos ítems durante el desarrollo del juego.
- ¿Un personaje principal (mainCharanter) está formado por 0 o muchos equipos? Sucede exactamente lo mismo que con la relación anterior: un personaje principal utiliza 0 o muchos equipamientos.
- Finalmente, ¿Un personaje principal (mainCharanter) está formado por 1 o muchas habilidades? Ciertamente que el personaje principal adquiere las habilidades que usará durante el juego.

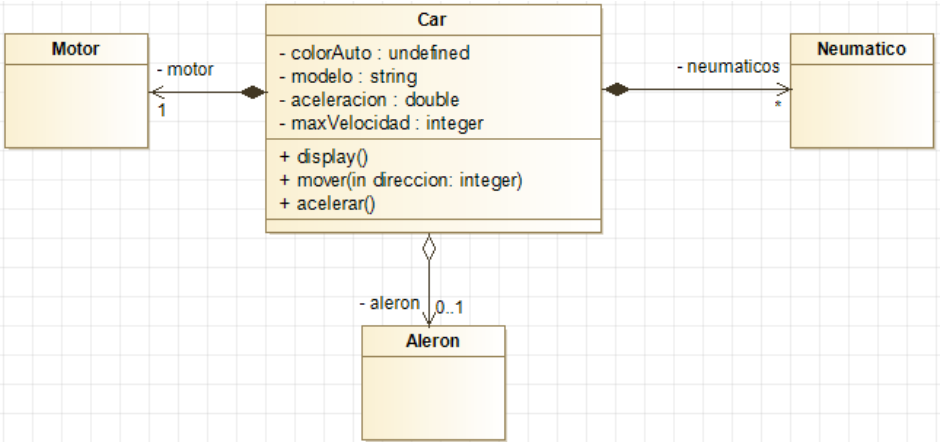
La composición

Es un tipo muy representativo de una agregación, que expresa una relación más fuerte entre el todo y sus partes. **Cada componente puede pertenecer tan solo a un todo**. En la composición la clase "todo" es responsable de la creación y de la destrucción de la/s clase/s parte/s, y esta última no puede existir en alguna otra relación al mismo tiempo. **Por este motivo la composición se denomina agregación fuerte y la agregación propiamente dicha agregación débil**.

La clase compuesta debe garantizar que se crean todas sus partes y se fijan a la compuesta, antes de que ésta esté por completo construida. **En tanto exista la clase compuesta, se puede confiar que ninguna de sus partes será destruida por cualquier otra entidad**, pero cuando se destruya la compuesta, se deben destruir las partes, o puede eliminar en forma explícita las partes y llevarlas hacia algún otro objeto. La relación en UML se expresa mediante un símbolo similar al de la agregación, pero el conector en este caso es un diamante con color de relleno.

Ejemplo 8: Observe la siguiente imagen perteneciente al juego arcade The Fast & Furious

Al inicio del juego y a medida que se van ganando carreras, es posible adquirir diversas partes para mejorar el auto, entre los que se encuentra motores, nitro, neumáticos, alerones, etc. Modele un diagrama de clases que establezca la relación entre un auto y estas partes.



En este caso se ha modelado “parcialmente” lo solicitado con el objetivo de que ud lo pueda completar. Observe que “un auto está formado por un motor”. Hasta allí tenemos una agregación. Para que el auto pueda participar de la carrera, ese motor debe estar instalado previamente en el auto. Además, un motor solo puede estar en un auto a la vez y en caso de que el auto se destruyera por completo el motor probablemente también desaparecería. En este caso vemos que es una agregación fuerte, por lo cual es una composición.

Ahora observe la relación entre **Auto** y **Neumatico**. En este caso podemos aplicar el mismo criterio que en el caso anterior. Un auto no podría correr una carrera si no tiene instalado los 4 neumáticos. Podemos decir que un auto está formado por 4 neumáticos y que estos no pueden formar parte de dos autos diferentes. Observe un dato curioso: la multiplicidad del lado de las partes es *. En lo típico sería razonable pensar que se podría leer “un auto está formado por 0 o muchos neumáticos”, pero como es una composición, sería una interpretación conceptual

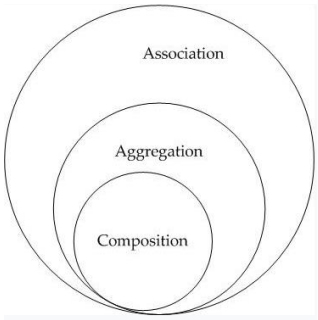
errónea, debido a que las partes se deben crear al momento de la instanciación del objeto de tipo Auto. Por lo tanto, lo correcto es leer literalmente "un auto está formado por muchos neumáticos", aunque sabemos que en un análisis necesitamos exactamente 4 neumáticos para empezar, pero lamentablemente en UML ese tipo de multiplicidad ya no es válido.

Finalmente podemos observar el caso del alerón. En este caso podemos decir que "un auto está formado por un alerón" o que un "alerón forma parte de un auto". Sin embargo, podríamos correr una carrera sin el alerón. Por ese motivo resulta claro que es una agregación débil, o simplemente una agregación. Y la multiplicidad expresa el hecho de que podría utilizar o no un alerón.

Ejemplo 9: retome el ejemplo 6 y determine si la agregación entre Compra y Detalle debería considerarse una composición, en cuyo caso actualice el diagrama de clases considerando los siguientes casos particulares

a) El detalle de una compra puede ser eliminado de una compra debido a que el comprador tiene la posibilidad de anularla por arrepentimiento.
b) No se puede anular ningún detalle de la compra.
c) La empresa que desarrolló el juego utiliza los detalles en su propio sistema de ventas para la generación de las facturas.

Nota: Finalmente para terminar con los conceptos vertidos hasta ahora, se puede expresar gráficamente la relación que existe entre estas tres asociaciones con la siguiente gráfica, donde la agregación y la composición son subconjuntos de asociación, lo que significa que son casos específicos de asociación:

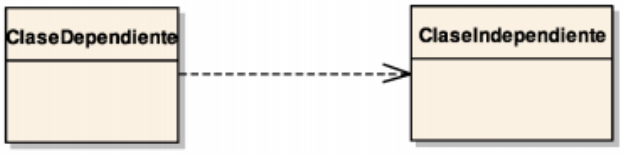


La dependencia

Se lo suele definir como una **"asociación de uso"**. Esto significa que expresa una situación donde **una clase A utiliza otra clase B, y donde un cambio en la especificación de B puede afectar a A, pero no necesariamente a la inversa.**

Por ese motivo la relación se denomina de dependencia**: la clase A se denomina cliente, mientras que la clase B se conoce como proveedor**. **Un objeto de la clase cliente depende de un objeto de la clase proveedor para poder proporcionar o completar un servicio**. El símbolo para una relación de dependencia luce como una asociación unidireccional, excepto que la línea es punteada en lugar de continua.

Esta situación expresa que, a diferencia de las anteriores relaciones tratadas, la dependencia no opera con



punteros. Es un recurso que sirve para mantener la estructura de clases con el menor nivel de relacionamiento posible, es decir su objetivo consiste en contribuir en el diseño de un modelo con bajo **acoplamiento**. El acoplamiento es una medida que indica cuan relacionadas están las clases. Si las clases están muy relacionadas, se dice que tienen un alto acoplamiento, como consecuencia de esto, cualquier cambio en ambas clases puede afectar seriamente a la otra clase dificultando la posibilidad de modificar, actualizar o mejorar el funcionamiento del sistema, lo cual no es deseado. El acoplamiento no se puede evitar, porque para que el sistema realice su objetivo requiere que los objetos interactúen entre ellos, pero si se debería minimizar.

Cada vez que se establece una relación de asociación, se establece un puntero entre dos clases. Esto genera acoplamiento, y una relación bidireccional genera aún mayor acoplamiento.

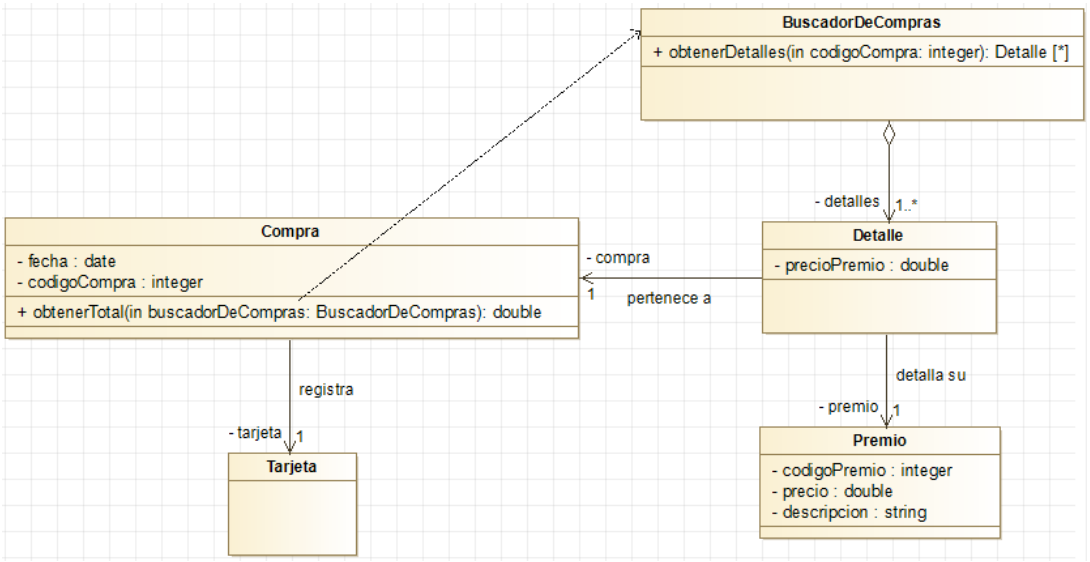
Por este motivo, **para evitar la bidirección cuando se necesita una visibilidad invertida, se aplican dependencias**. La dependencia **se implementa mediante un concepto denominado visibilidad por parámetros**. Esto significa que la operación del cliente debe especificar que uno de los parámetros sea un objeto del servidor.

De esta manera el cliente "visualizará" el servidor únicamente mientras dure el tiempo de vida de la operación, momento en el cual la visibilidad desaparece.

Entonces si bien **existe el acoplamiento, es mucho más débil que aquel que se generaría si se estableciera una bidirección como consecuencia la creación de un puntero**.

Las dependencias suelen aparecer cuando ud ya ha modelado su diagrama de clases y está a punto de programarlo, o cuando quiere visualizar sus clases programadas en la forma de un diagrama de clases. Esto significa que, si su diagrama de clases presenta dependencias, su modelo está en etapas avanzadas y generalmente representan las clases programadas.

Ejemplo 12: Considerando lo que indica el apartado anterior, se debe procurar un modelado intentando evitar usar relaciones bidireccionales. Entonces usando la dependencia, retome el ejemplo 6 y elimine el acoplamiento generado por la bidirección provocada por la agregación entre Compra y Detalle.



Observe que se ha reemplazado la agregación bidireccional entre Compra y Detalle (que produce un alto acoplamiento) por una asociación unidireccional y una dependencia. Desde Detalle hacia Compra se mantiene la asociación, ya que es necesario que cada Detalle sepa a que Compra

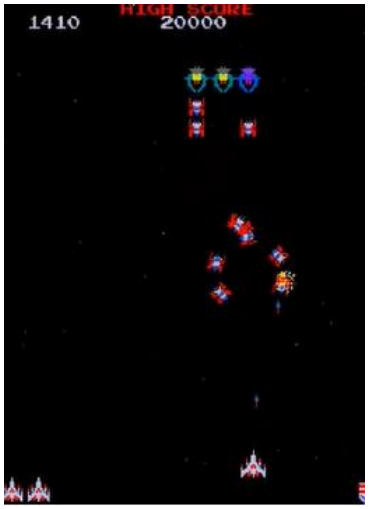
pertenece. Mientras que del otro lado, Compra ahora no mantiene una relación con Detalle, por lo cual ¿cómo obtiene el total de la compra? Ahora posee una operación denominada obtenerTotal() que recibe por parámetro un objeto de tipo BuscadorDeCompras. Esta clase posee una colección de objetos con todos los detalles existentes y además ofrece una operación por medio de la cual, si se le pasa el código de la compra, podrá devolver los detalles de esta. Al recibir por parámetro el buscador de compras, una compra podrá consultar cuales son sus detalles y de esa manera obtener el total de la compra. La relación de dependencia sale de una operación de la clase "Cliente" (la clase Compra) y para que se puede ejecutar requiere (o es dependiente) de una clase "Proveedora" (en este caso BuscadorDeCompras).

Un uso importante de la dependencia: la gestión de colisiones

En el mundo de los videojuegos, la interacción entre los objetos se conoce en primera instancia como colisión. Esto es, dos o más objetos se encuentran en un mismo espacio de la pantalla. La interacción de cada uno de ellos puede ser diferente: hablar, golpear, defenderse, explotar, etc.

Si ud programa juegos usando un lenguaje de programación puro y el paradigma orientado a objetos, una manera de gestionar estas interacciones es crear clases que mantengan una colección de objetos del mismo tipo y usar la dependencia para determinar alguno de los objetos de esta colección colisiona con otros objetos.

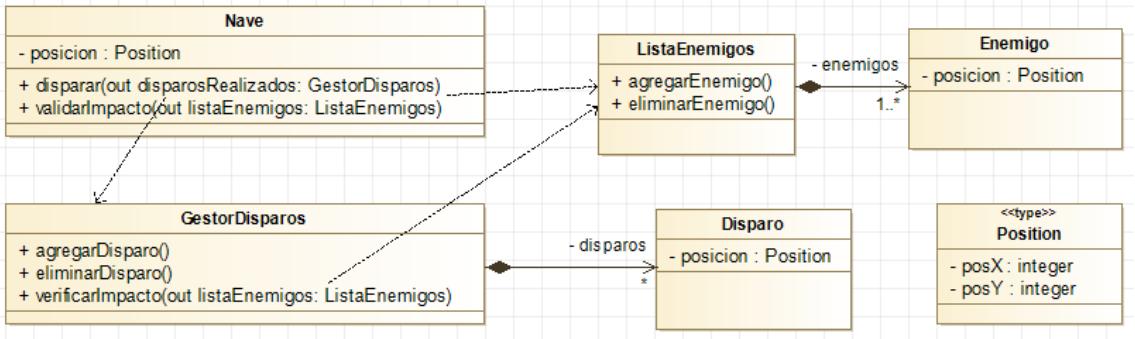
Ejemplo13: Observe la siguiente imagen. Pertenece al juego Galaga. En este juego, al inicio de cada nivel los enemigos se van ubicando en su posición original en el espacio. Luego de que se forman empiezan a atacar ya sea impactando sus disparos contra nuestra nave o chocándola, lo cual genera la pérdida de una vida.



Por otro lado, mientras se forman nosotros podemos atacarlos disparándoles (tal como lo muestra la imagen). Cada disparo que realizamos viaja por el espacio. Este disparo puede impactar en un enemigo o desaparecer de la pantalla.

Realice un diagrama de clases que permita gestionar la ubicación de cada enemigo generado y evaluar si nuestros disparos impactan contra alguno de ellos. De la misma manera valide si nuestra nave choca con un enemigo o si el disparo de uno de ellos impacta contra ella.

El siguiente diagrama de clases muestra parcialmente el modelado solicitado.



Observe que hemos creado dos clases que serán los contenedores de una colección de objetos. En primer lugar, se genera una colección de enemigos, representada por la clase ListaEnemigos. Esta clase posee un atributo denominado enemigos que es de tipo Enemigo.

En segundo lugar, se genera una colección de los disparos que nuestra nave realiza, denominada GestorDisparos, que posee un atributo denominado disparos que es de tipo Disparo.

Tanto Nave, como Enemigo y Disparo poseen un atributo denominado posición que es del Tipo Position, el cual posee las coordenadas en el plano. Esta clase está etiquetada con el estereotipo <<type>> por lo cual sabemos que es usada para generar un tipo de datos no existente en UML.

Ahora observemos el funcionamiento de este diagrama de clases:

- Un objeto de tipo **Nave** puede realizar una operación denominada *disparar()*. Cada vez que se dispara se agrega un nuevo objeto de tipo **Disparo** al gestor de disparos (seguramente se invoca la operación *agregarDisparo()*). **Para poder realizar esto, la única manera de hacerlo es que la operación *disparar()* reciba como parámetro el gestor de disparos.**
- Los disparos almacenados en el gestor de disparos van actualizando su posición constantemente. Por ese motivo se debe invocar en todo momento la operación *verificarImpacto()* para determinar si el disparo desaparece de la pantalla o impacta con un enemigo. Como se podrá dar cuenta, para verificar si impacta con un enemigo, esta operación requiere poder disponer de la lista de enemigos. Es por esta razón que la operación mantiene una dependencia con **ListaEnemigos**. Entonces, verificarImpacto() seguramente tomará cada disparo del Gestor de disparos y comparará su posición con respecto a la posición de cada enemigo de la lista de enemigos, en cuyo caso en principio eliminará al enemigo de su lista y el disparo también desaparecerá del gestor.
- Observe que Nave posee una operación validarImpacto(). Esta operación se encarga de determinar si la nave choca con un enemigo. Por este motivo, se requiere de la lista de enemigos y de esa manera se establece una dependencia. Nuevamente, este impacto será confirmado por el hecho de que la posición de la nave coincide con la posición de un enemigo. Si esto sucede, en principio se disminuirá la cantidad de vidas de la nave y se eliminará el enemigo de su lista.

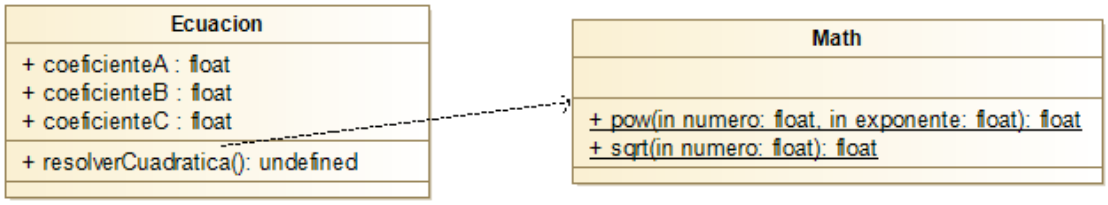
Con la información brindada por este ejemplo intente completar el diagrama de clases para que posea los atributos y operaciones faltantes para validar que sucede si un disparo de los enemigos impacta con nuestra nave.

Como afecta una clase degenerada la relación de dependencia

Si la clase proveedor es una clase “degenerada” no se requerirá definirlo como parámetro en la operación del cliente. Una clase degenerada es aquella que no posee estado, su función es brindar un conjunto de operaciones que deberían estar disponibles para todos los objetos. Muchos lenguajes de programación implementan las clases degeneradas como clases estáticas.

Si bien, las clases estáticas se estudiarán al momento de realizar la programación de clases, se puede adelantar que su objetivo fundamental es permitir que se puedan utilizar las operaciones de una clase sin necesidad de crear previamente un objeto de esa clase.

Ejemplo14: Con motivo de expresar en un diagrama de clases como afecta la dependencia el uso de una clase degenerada se propone el siguiente ejemplo: En Java **Math** es un ejemplo de clase degenerada, por lo cual hay una relación entre la definición de clases degeneradas y las clases que no poseen atributos (a menos que sean constantes) y sus operaciones son estáticos (operaciones de clase, que estudiarán posteriormente). Suponga que desea modelar la dependencia de una clase **Ecuacion** respecto del cálculo de una ecuación cuadrática al aplicar la clase **Math**



Observe que en Math se han definido operaciones de clase (están subrayadas, tal como sucede con los atributos de clase).

La operación *resolverCuadratica()* presenta un tipo de datos de retorno undefined, ya que ninguno de los tipos de datos UML por defecto se ajusta al hecho de que debe devolver el valor de x_1 y x_2.

¿Pero que es lo que le llama la atención de esta dependencia?

Como habrá notado, la operación no recibe como parámetro un objeto de la clase Math. Esto se debe al hecho que Math es degenerada, por lo cual no se crean objetos a partir de ella, por lo cual no hay forma de pasar un objeto como parámetro. Pero al ser degenerada, es posible utilizar sus operaciones sin ningún problema: pow() permite obtener la potencia de un número, para lo cual recibe como parámetro el número y el exponente, devolviendo la potencia generada; mientras que sqrt() devuelve la raíz cuadrada de un número. Si recuerda la ecuación que permite obtener las raíces de una ecuación cuadrática utilizan ambas operaciones.