Asociaciones (relaciones entre objetos)

PID_00269658

David García Solórzano

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

David García Solórzano

Graduado Superior en Ingeniería en Multimedia e Ingeniero en Informática por la Universitat Ramon Llull desde 2007 y 2008, respectivamente. Es también Doctor por la Universitat Oberta de Catalunya desde 2013, donde realizó una tesis doctoral relacionada con el ámbito del e-learning. Desde 2008 es profesor de la Universitat Oberta de Catalunya en los Estudios de Informática, Multimedia y Telecomunicación.

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: David García Solórzano

Primera edición: febrero 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

Introducción

Hasta ahora hemos visto una clase –formada por sus atributos y métodos– trabajar de manera aislada. Es obvio que una única clase no puede solucionar por sí sola un problema y que, por consiguiente, tendrá que colaborar con otras clases para resolverlo. En este módulo veremos cómo los objetos de diferentes clases pueden colaborar o relacionarse entre ellos. Principalmente veremos:

- asociaciones
- clase asociativa
- clase parametrizada

Además, veremos cómo se codifican y se representan estas relaciones usando el lenguaje de modelado de sistemas de *software* UML (*Unified Modeling Language*).

Salvo que se indique un lenguaje de programación concreto, los ejemplos de codificación están escritos con un lenguaje de programación inventado, es decir, un pseudocódigo. Si tuviéramos que decir a qué lenguaje de programación real se parece el pseudocódigo empleado, diríamos que es parecido a Java (pero sin elementos que dificultan el entendimiento de los ejemplos).

Así pues, deberás consultar la documentación del lenguaje de programación que desees utilizar para ver cómo se codifican los ejemplos proporcionados.

Objetivos

El objetivo principal de este módulo es explicar cómo los objetos se relacionan entre ellos dentro de un programa basado en el paradigma de la programación orientada a objetos. Como consecuencia de este objetivo, aparecen otros:

1. Entender qué es una asociación y distinguir entre los cuatro tipos más importantes: binaria, agregación, composición y reflexiva.

2. Conocer las tres propiedades básicas que tienen las asociaciones: multiplicidad, navegabilidad y rol.

3. Entender qué es una clase asociativa y saber cuándo su uso es necesario.

4. Comprender qué es una clase parametrizada y en qué contextos es comúnmente utilizada.

5. Saber representar en un diagrama de clases UML los cuatro tipos de asociación vistos en este módulo, la clase asociativa y la clase parametrizada.

6. Saber y entender cómo codificar los cuatro tipos de asociación explicados en este módulo, así como la clase asociativa y la clase parametrizada.

1. Asociaciones

La relación entre objetos o instancias es conocida con el nombre de *asociación*. Podemos distinguir principalmente cuatro tipos de asociaciones:

- asociación binaria
- asociación de agregación
- asociación de composición
- asociación reflexiva

Veremos estos cuatro tipos a continuación.

Nota

Es importante resaltar que los objetos de dos clases pueden relacionarse de manera diferente según sea el problema o contexto en el que están y el criterio del diseñador del programa.

1.1. Asociación binaria

Podemos definir la **asociación binaria** (*binary association*) como la relación que existe entre instancias/objetos de dos clases, donde los objetos de una clase existen de forma independiente a la existencia de los objetos de la otra clase.

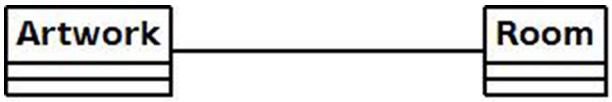
Así pues, la creación o destrucción de una instancia de la clase `A` implica únicamente la creación o destrucción de la relación que existe entre esa instancia y otra instancia de la clase `B`, pero nunca significa la creación o destrucción de la instancia de la clase `B`.

Hablando con propiedad se dice que no hay una relación fuerte entre ambas instancias. La asociación binaria responde a una relación que indica que el objeto de la clase `A` *usa* un objeto de la clase `B` y puede que viceversa también, como ya veremos.

Ejemplo 1 (asociación binaria) – Las obras de arte y las salas de un museo

Imagina un museo que alberga obras de arte. Cada obra de arte (instancia u objeto de la clase `Artwork`) está expuesta en una sala del museo (instancia u objeto de la clase `Room`). Ahora pensemos en la relación que existe entre los objetos de ambas clases. Si se destruye una obra de arte (por ejemplo, se quema o se deteriora), esto no significa que destruyamos la sala del museo (¡menuda gracia!), ya que puede haber más obras de arte expuestas en ella. Lo mismo pasa al revés, si destruimos una sala porque la vamos a fusionar con otra o vamos a usar su espacio para otra cosa –por ejemplo, ya no será una sala (`Room`), sino un despacho (`Office`)–, las obras de arte que alberga no las tenemos que destruir (¡solo faltaría!), en todo caso, las tendremos que reubicar (reasignar) a otras salas del museo.

En UML esta asociación se representa con una línea que une ambas clases:



Nota

Para facilitar la comprensión y reducir la extensión de estos materiales, se han eliminado los atributos y métodos de las clases de los diagramas de clases.

1.2. Asociación de agregación

Podemos definir la **asociación de agregación** (*aggregation* o *shared aggregation*) como la relación que existe entre instancias de dos clases cuando una de ellas (llamada *componente*) es parte de la otra (llamada *compuesto*).

Este tipo de asociación es también conocido como **composición débil**. Se asume una subordinación conceptual del tipo «todo/parte», «está formado por», o bien «tiene un». Así pues, se trata de un caso particular de asociación binaria en la que hay una cierta relación de ensamblaje, ya sea física o lógica, entre las dos instancias.

Una de las características de la agregación es que la **destrucción del compuesto no significa la destrucción de los componentes, ni viceversa**. Además, el **objeto componente puede estar presente en más de una asociación de agregación**.

Ejemplo 2 (asociación de agregación) – Las obras de arte y las colecciones

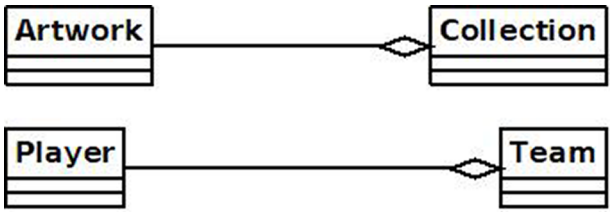
En muchos museos, las obras de arte (instancia u objeto de la clase `Artwork`) pertenecen a una colección (instancia u objeto de la clase `Collection`). Ahora pensemos en la relación que existe entre los objetos de ambas clases. Podemos ver que una colección está formada por obras de arte. Así pues, la colección es el todo (clase compuesta) y las obras de arte son la parte (clase componente). Además, si se destruye una obra de arte (por ejemplo, se quema o se deteriora) o simplemente decidimos que ya no pertenece a la colección que tiene asignada, esto no significa que destruyamos (que ya no exista) la colección a la que pertenecía hasta ahora, ya que puede haber más obras de arte que sí pertenezcan. Lo mismo pasa al revés, si decidimos que ya no queremos englobar las obras de arte bajo el paraguas de una colección, eliminamos la colección, pero no las obras de arte que formaban parte de ella. Por último, hay que decir que una obra de arte puede pertenecer a varias colecciones a la vez.

Veamos otro ejemplo:

Ejemplo 3 (asociación de agregación) – El equipo de baloncesto

Un equipo de baloncesto (instancia u objeto de la clase `Team`) está formado por o tiene jugadores (instancia u objeto de la clase `Player`). El equipo es el compuesto y cada jugador un componente. Si eliminamos un jugador del equipo (por ejemplo, lo hemos transferido o se ha retirado), el equipo no desaparece, obviamente. Asimismo, si el equipo desaparece (por ejemplo, entra en números rojos), los jugadores no desaparecen, se van a (formarán parte de) otros equipos.

La manera de representar en UML una asociación de agregación es mediante un rombo blanco en el extremo de la clase de los objetos que hacen de compuesto.



1.3. Asociación de composición

Podemos definir la **asociación de composición** (*composition* o *composite aggregation*) como un caso particular de la asociación de agregación en la que la vida o existencia de la instancia de la clase componente depende de la existencia de la instancia de la clase compuesta. Así pues, el objeto componente solo puede existir si existe el objeto compuesto. Esto conlleva que la destrucción del objeto compuesto supone la destrucción de todos sus componentes, pero no al revés.

Otra de de las características que diferencian la composición de la agregación es que **en la composición cada componente solo puede relacionarse mediante composición con un único compuesto**. Por todo ello, este tipo de asociación también se conoce como **composición fuerte**. No obstante, un objeto componente puede relacionarse con objetos de otras clases mediante otros tipos de asociación.

Ejemplo 4 (asociación de composición) – La mano

Mira una de tus manos. ¿Está formada por dedos, verdad? En este punto podemos decir que la relación entre mano (objeto de la clase `Hand`) y los dedos (objetos de la clase `Finger`) es de compuesto y componentes. ¿Es una asociación de agregación o de composición? Si amputamos la mano, implícitamente eliminamos los dedos que la componen, pero si amputamos un dedo de la mano, ¡no amputamos toda la mano! Además, los dedos solo existen para esa mano. Esto nos hace ver que para que existan los dedos, debe existir la mano que los contiene, por lo tanto, estamos ante un caso de composición.

Lo mismo ocurre con el siguiente ejemplo:

Ejemplo 5 (asociación de composición) – La silla

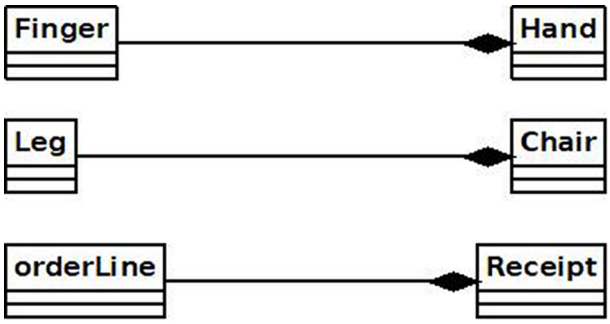
Una silla (objeto de la clase `Chair`) está compuesta o formada por patas (objetos de la clase `Leg`). Si destruimos la silla, estamos destruyendo sus patas implícitamente. Si le quitamos una pata a la silla, la silla continúa existiendo. Estamos claramente ante un caso de composición.

Finalmente:

Ejemplo 6 (asociación de composición) – El tique de compra

El tique de compra que nos dan en un supermercado (objeto de la clase `Receipt`) está formado por líneas de compra (objetos de la clase `orderLine`) que nos indican el producto adquirido, la cantidad y el precio. Cada línea del tique solo existe en ese tique y porque el tique existe. Si perdemos el tique, perdemos las líneas de compra. Si quitamos una línea de compra (por ejemplo, la tachamos), el tique no desaparece. Así pues, se trata de una asociación de composición.

La composición se representa igual que la agregación, pero el rombo ahora es de color negro.



1.4. Asociación reflexiva

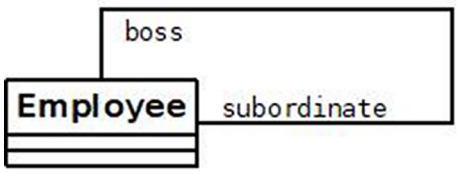
Podemos definir la **asociación reflexiva** (*reflexive association*) como una asociación binaria en la que los objetos que participan en la relación pertenecen a la misma clase, es decir, los objetos origen y destino son de la misma clase.

Veamos un ejemplo:

Ejemplo 7 (asociación reflexiva) – El jefe y sus subordinados

Piensa en cualquier trabajo. Siempre hay un empleado que es jefe de otros. Así pues, tanto el jefe como los trabajadores a su cargo son empleados. Eso sí, cada empleado existe independientemente del resto y un empleado no está formado o compuesto por otros empleados.

La manera de representar una asociación reflexiva es igual que la asociación binaria (no deja de ser un caso particular de esta), salvo que la línea sale y entra en la misma clase.



Como puedes ver en el diagrama de clases anterior, hay dos etiquetas o textos: `<<boss>>` y `<<subordinate>>`. Estas etiquetas se llaman **roles** (lo veremos en este módulo) y se escriben en los extremos de la asociación para identificar correcta y semánticamente qué papel juega cada objeto dentro de la relación.

1.5. Propiedades de las asociaciones

Las asociaciones anteriores tienen básicamente tres propiedades:

- multiplicidad
- navegabilidad
- rol

A continuación veremos cada una de ellas.

1.5.1. Multiplicidad

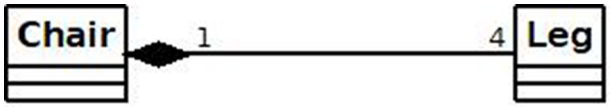
Debido a que una asociación entre dos clases representa en realidad una relación entre instancias u objetos de esas dos clases, debemos indicar cuántas instancias de cada clase están involucradas en la relación. Este número de instancias que participan en la relación es, precisamente, la multiplicidad de la asociación.

Definición de multiplicidad

Es el número mínimo y máximo de instancias de una clase con el que se puede relacionar una instancia de la otra clase de la relación.

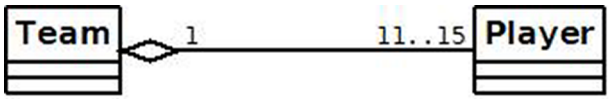
La manera de indicar la multiplicidad es mediante modificadores que se añaden encima o debajo de la línea que representa la relación y cerca de la clase. Cada asociación tiene dos multiplicidades, una por extremo, es decir, una por cada clase que participa en la asociación. Como hemos dicho, cada multiplicidad indica un número mínimo (llamado *opcionalidad* o *participación*) y máximo (llamado *cardinalidad*) de objetos de esa clase que se relacionan con un objeto de la otra clase. Así pues, la multiplicidad de una clase está formada por la participación (valor mínimo) y cardinalidad (valor máximo). Hay diferentes maneras de indicar la multiplicidad.

1) Número exacto: indica que una instancia de la clase `A` tiene que estar relacionada exactamente con el número indicado de instancias de la clase `B`. Así pues, la participación y la cardinalidad que conforman la multiplicidad de la clase coinciden.

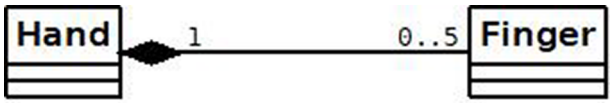


En el ejemplo anterior, un objeto de tipo `Chair` (‘silla’) se relaciona con cuatro objetos de la clase `Leg` (‘pata’). En este caso, la cardinalidad y la participación coinciden siendo `4`. Asimismo, un objeto de tipo `Leg` se relaciona con un único objeto de tipo `Chair` (lógico al tratarse de una composición). En este caso, también la cardinalidad y la participación coinciden, pero en el valor `1`.

2) Rango de valores: permite indicar el número mínimo y máximo de instancias de la clase `B` con las que un objeto de la clase `A` debe relacionarse.

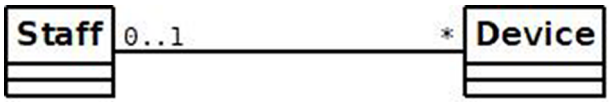


Según la normativa de la NBA, un equipo que juega en dicha liga de baloncesto debe tener entre `11` y `15` jugadores. Así pues, no puede tener ni más ni menos. En este caso, la multiplicidad de la clase `Player` es `11..15`, siendo la participación `11` y la cardinalidad `15`. Por su parte, un jugador solo puede pertenecer a un equipo. Como la participación es `1`, no existe la opción de que un jugador no tenga equipo (cuando se usa el rango `0..1`, se está diciendo que el objeto puede relacionarse o no, de ahí que la participación –el valor mínimo de la multiplicidad– también se conozca como *opcionalidad*).



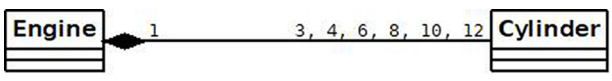
Una mano humana puede tener entre `0` y `5` dedos. Un dedo pertenece a una única mano. En este caso es opcional que una mano tenga dedos, puede que no tenga ninguno, de ahí que indiquemos que la participación sea `0` dentro de la multiplicidad de la clase `Finger`.

3) Símbolo *: existe la posibilidad de utilizar el símbolo `*` (asterisco) para indicar que el número de instancias es indefinido, es decir, con `*` estamos diciendo que un objeto de la clase `A` puede estar relacionado con cualquier número de instancias de la clase `B`, incluyendo ninguna (cero). Escribir `*` es equivalente a indicar la multiplicidad `0..*`.



El ejemplo anterior indica que un miembro del equipo (`Staff`) puede tener o usar entre `0` (ninguno) y un número indeterminado de dispositivos (`Device`), por ejemplo, un ordenador de sobremesa, un portátil, un móvil, etc. Además, cada dispositivo es tenido o usado, como máximo, por una persona a la vez (también puede no ser usado por nadie).

4) Rangos no consecutivos: separando números exactos o rangos por comas, se pueden hacer rangos no consecutivos.



Un motor (`Engine`) puede tener 3, 4, 6, 8, 10 ó 12 cilindros (pero no 9, por ejemplo), y un cilindro (`Cylinder`) solo puede pertenecer a un motor. Esta relación se trata de una composición, puesto que la destrucción del objeto `Engine` debe suponer la destrucción de los objetos `Cynlinder`, ya que la existencia de los objetos `Cylinder` depende de la existencia de un objeto `Engine` que los contenga.

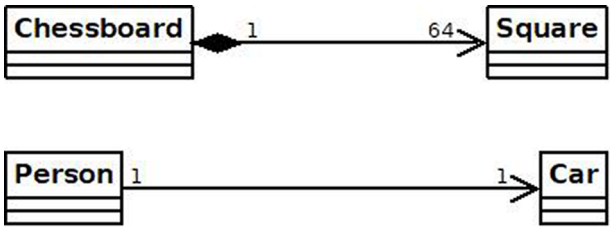
Multiplicidad en la composición

Dado que el objeto componente en una asociación de composición solo puede existir si existe el objeto compuesto al que pertenece, y no puede ser parte de otra composición, la multiplicidad en la clase que hace de compuesto (donde está el rombo negro) es `0..1` o `1`. El caso `0` es un poco especial, y quiere decir que si el objeto componente es extraído de la composición antes de eliminar el objeto compuesto, entonces el objeto componente que ha sido extraído puede seguir existiendo. Sin embargo, si no se ha extraído el objeto componente del objeto compuesto y se elimina este último, entonces el objeto componente también se elimina. Por lo tanto, lo más frecuente es una multiplicidad igual a `1` en el lado del compuesto.

1.5.2. Navegabilidad

Otra propiedad que podemos expresar en una asociación dentro de un diagrama de clases es su navegabilidad o direccionalidad, es decir, indicar si una instancia de una clase conoce la existencia de una instancia de la otra clase. Básicamente hay dos tipos de navegabilidad:

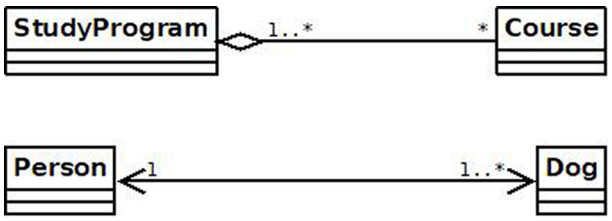
1) Unidireccional: solo una de las dos instancias tiene constancia de la existencia de la otra. La navegabilidad unidireccional se representa con una punta de flecha que apunta a la clase que no tiene constancia.



En el primero de los dos ejemplos podemos ver cómo un tablero de ajedrez (`Chessboard`) está compuesto por sesenta y cuatro cuadrados (`Square`), y un cuadrado solo pertenece a un tablero. Es el tablero (`Chessboard`) el que sabe de la existencia de los sesenta y cuatro cuadrados (`Square`), es decir, dado un tablero, podemos conocer sus sesenta y cuatro cuadrados, pero no al revés.

En el segundo ejemplo, una persona (`Person`) conduce un coche (`Car`), y un coche solo puede ser conducido por una persona en un momento determinado. En este caso, la navegabilidad nos está diciendo que, dado un objeto de tipo `Person`, podemos saber qué coche está conduciendo, pero, sin embargo, dado un objeto `Car`, no podemos saber qué persona lo conduce.

2) Bidireccional: las dos instancias tienen constancia de la existencia de la otra. La navegabilidad bidireccional se representa sin puntas de flecha. También puedes ver en algunos diagramas de clases que la bidireccionalidad está representada dibujando puntas de flecha en ambos lados de la línea (mira el ejemplo `Person-Dog`). Esto es correcto, aunque es menos habitual.



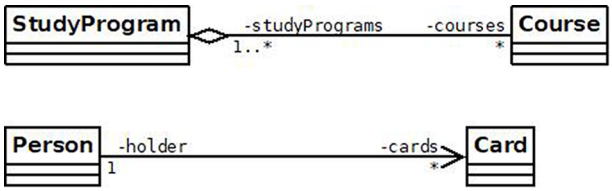
En el primer ejemplo, `StudyProgram-Course`, un plan de estudios (`StudyProgram`) está formado por diferentes asignaturas (`Course`). En este caso, se contempla el caso de que un plan de estudios no esté formado por ninguna asignatura. Una asignatura, a su vez, puede formar parte de uno o diversos planes de estudios. Por ejemplo, la asignatura de *Diseño y programación orientada al objeto* forma parte de los planes de estudio de los grados de Ingeniería Informática, Tecnologías de Telecomunicación, Multimedia, Ciencia de Datos Aplicados, entre otros. Gracias a la navegación bidireccional, dado un plan de estudios, podemos saber sus asignaturas y, dada una asignatura, podemos saber a qué planes de estudio pertenece.

En el segundo ejemplo estamos diciendo que una persona (`Person`) tiene uno o más perros (`Dog`), y que un perro tiene solo una persona como dueña. Dada una persona, sabemos qué perros tiene y dado un perro, podemos saber quién es su dueño.

1.5.3. Rol

Esta propiedad es un poco diferente a las dos anteriores. A diferencia de la multiplicidad y la navegabilidad, el rol no modifica el comportamiento de las asociaciones, sino que añade información contextual que debe ayudar al lector a comprender la relación.

El rol no es más que una etiqueta que se pone a cada lado de la asociación para llamar a las clases participantes de una manera diferente o, dicho de otro modo, para indicar el papel que juega cada clase. El rol tiene relevancia cuando traducimos una asociación del diagrama de clases a código. Concretamente cada rol se convertirá en un atributo en la clase conocedora y será del tipo de la otra clase. Es por eso muy habitual anteponer al nombre del rol el símbolo correspondiente al modificador de acceso que deseemos, es decir: `+`, `-`, `#` o `~`. Si no se pone, se entiende que es privado.



En el primer ejemplo, la clase `StudyProgram` tendrá un atributo llamado `courses` que será privado (fijémonos en el símbolo – precediendo el nombre del rol). Este atributo será un *array* (o una colección) que almacenará objetos de la clase `Course` (puede estar vacío). Asimismo, la clase `Course` tendrá un atributo privado llamado `studyPrograms` que será un *array* (o colección). Este *array* o colección guardará, como mínimo, un objeto de la clase `StudyProgram`.

En el segundo ejemplo, una persona (`Person`) puede tener varias tarjetas de fidelización (`Card`) o ninguna, mientras que una tarjeta solo será de una persona. En este caso, la clase `Person` tendrá un atributo privado llamado `cards` que almacenará objetos de tipo `Card` (puede ser un *array* vacío). Sin embargo, como la navegabilidad es unidireccional, la clase `Card` no tendrá un atributo llamado `holder`. De hecho, se podría haber omitido la aparición de dicho rol en la asociación. Entendemos que en este contexto las tarjetas no tienen nombre de titular.

1.6. Traduciendo a código

Hemos visto cuatro tipos de asociaciones desde un punto de vista teórico, pero ¿cómo se codifican en la práctica? Antes de explicar cómo se codifican los diferentes tipos de asociación que hemos visto, cabe tener muy presente que:

> La codificación de cualquier asociación debe tener en cuenta sus propiedades, es decir, la multiplicidad, navegabilidad y roles.

Veamos cómo afecta cada propiedad a la codificación:

a) Navegabilidad: como ya sabemos, denota la conciencia, por parte de una clase, de la propia relación. En términos de implementación, la navegabilidad nos indica en cuál de las clases debemos definir un atributo que sea del tipo de la otra clase. Dicho de otro modo, la navegabilidad nos dice en qué clase debe haber una referencia a objetos de la otra clase que participa en la relación. Así pues, tenemos dos casos:

- **Unidireccional**: la clase conocedora (i.e. aquella de la que sale la flecha) será la única que tenga una referencia a la otra clase (i.e. aquella a la que llega la punta de la flecha).

- **Bidireccional**: las dos clases de la asociación tendrán una referencia a la otra clase.

b) Multiplicidad: debido a que la multiplicidad puede tener diferentes formatos (por ejemplo, número exacto, rango de valores, etc.), la referencia a la otra clase dentro de la clase conocedora puede implementarse de muchas maneras. Sin embargo, podemos diferenciar dos grandes tipos de multiplicidad según la manera en la que se codifican:

- **Un objeto**: cuando la multiplicidad es `1` o `0..1`, entonces nos está diciendo que tenemos un atributo en la clase conocedora que es del tipo de la otra clase.

- **Más de uno**: cuando la multiplicidad es diferente a `1` o `0..1` (por ejemplo, `*`, `1..5`, `8..12`, etc.), entonces el atributo en la clase conocedora tiene que ser una colección de objetos de la otra clase. Por *colección* entendemos cualquier estructura que permita almacenar más de un objeto, por ejemplo: un *array*, una pila, una cola, una lista, etc. ¿Cuándo usar una u otra? Dependerá del problema que haya que resolver.

c) Rol: esta propiedad nos ayuda a determinar dos aspectos de la implementación. Por un lado, nos da a conocer cómo ha llamado el programador (o sugiere el diseñador llamar) al atributo que hace referencia a la otra clase. Por otro lado, también nos indica el nivel de acceso de dicho atributo. Como ya sabemos, el nombre del rol suele estar formado por el modificador de acceso y el nombre en sí, por ejemplo, `-person`, `-dogs`, etc.

Ahora que sabemos cómo afectan las propiedades de la asociación a la codificación de la misma, veamos cómo afecta el tipo de asociación.

1.6.1. Asociaciones binarias, reflexivas y de agregación

Las asociaciones binarias, reflexivas y de agregación son idénticas a nivel de codificación. En el caso de las binarias y reflexivas es evidente, ya que, como hemos visto, estas últimas son un caso particular de las binarias en las que las clases origen y destino son la misma. Por su parte, las asociaciones binarias y de agregación solo se diferencian a nivel conceptual, puesto que la asociación de agregación, a diferencia de la binaria, añade la semántica de que en la relación hay una clase componente que incluye a la otra, pero nada más.

Como en los tres tipos de asociación, la existencia de un objeto de una clase es independiente de la existencia de un objeto de la otra clase, el modo de codificar estas tres asociaciones es asignando un objeto ya existente al atributo que hace de rol en la clase conocedora. Es decir, la clase conocedora no instancia un objeto de la otra clase, sino que lo recibe y lo asigna al atributo que hace de rol.

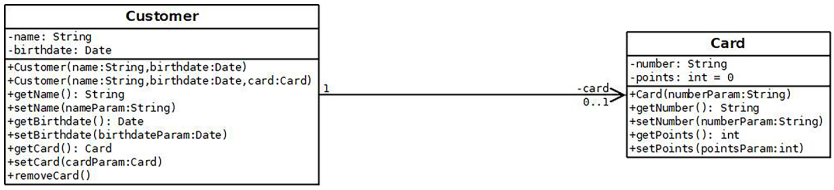
Veamos varios ejemplos para entenderlo mejor:

Aclaración

Los siguientes ejemplos sirven tanto si la asociación es binaria, reflexiva o de agregación.

Ejemplo 8 (codificación) – Asociación binaria o agregación unidireccional 1-1

Dada la siguiente relación entre cliente (`Customer`) y tarjeta de fidelización (`Card`).



Customer

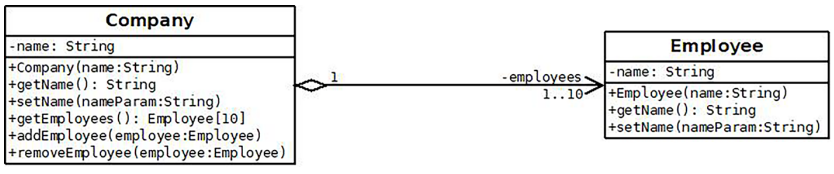
```
class Customer{
     private String title;
     private Date birthdate;
     private Card card;

     public Customer(String name, Date birthdate){
          setName(name);
          setYearBirth(yearBirth);
          card = null;
     }

     public Customer(String name, Date birthdate, Card card){
          setName(name);
          setBirthdate(birthdate);
          setCard(card);
     }

     public setCard(Card cardParam){
          if(cardParam != null){
               card = cardParam;
          }
     }
     public removeCard(){
          card = null;
     }
     …
}
```

Card

```
class Card{
     private String number;
     private int points; //No incluye un objeto de tipo Customer

     public Card(String numberParam){
          number = numberParam;
          points = 0;
     }
     …
}
```

Ejemplo 9 (codificación) – Asociación binaria o agregación unidireccional 1-N

Dada la siguiente relación entre compañía (`Company`) y empleado (`Employee`).



Company

```
class Company{
     private String title;
     private Employee[10] employees;


     public Company(String name){
          setName(name);
          employees = new Employee[10];
     }

     public addEmployee(Employee employee){
          //Para simplificar, suponemos que los arrays tienen un método llamado
          //contains que dice si está el objeto
          if(employee!= null && !employees.contains(employee)){
               employees.add(employee); //simplificación: suponemos que los
     //arrays tienen el método "add" que añade el objeto en la primera posición libre
          }
     }

     …
}
```

Employee

```
class Employee{
     private String name; //No incluye un objeto de tipo Company

     public Employee(String name){
          setName(name);
     }
     …
}
```

Ejemplo 10 (codificación) – Asociación binaria o agregación bidireccional 1-1

Dada la siguiente relación en la que a cada estudiante (`Student`) se le da un ordenador (`Computer`) a principio de curso.

Computer

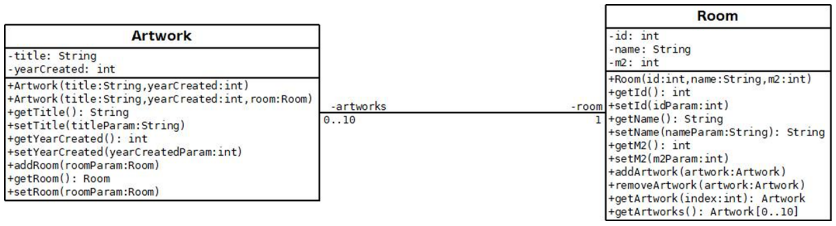
```
class Computer{
     private String name;
     private Student student;

     public Computer(String name, Student student){
          setName(name);
          addStudent(student);
     }

     public addStudent(Student studentParam){
          if(studentParam != null){
               //si student ya tiene ordenador, a ese ordenador le debemos
               //quitar la referencia al student
               if(studentParam.getComputer() != null){
                    studentParam.getComputer().setStudent(null);
               }
               setStudent(studentParam); //asignamos student a este objeto Computer
               studentParam.setComputer(this);//asignamos a student este objeto
                                              //Computer  (this)
          }
     }
     public setStudent(Student studentParam){
          student = studentParam;
     }

     public Student getStudent(){
          return student;
     }
     …
}
```

Student

```
class Student{
     private String title;
     private Computer computer;

     public Student(String name, Computer computer){
          setName(name);
          addComputer(computer);
     }

     public addComputer(Computer computerParam){
          //Same reasoning as the one in addStudent of the class Computer
     }
     public setComputer(Computer computerParam) {
          computer = computerParam;
     }

     public Computer getComputer(){
          return computer;
     }
     …
}
```

Ejemplo 11 (codificación) – Asociación binaria o agregación bidireccional 1-N

Dado la siguiente relación entre obras de arte (`Artwork`) y salas (`Room`) donde se exponen.



Artwork

```
class Artwork{
     private String title;
     private int yearCreated;
     private Room room;

     public Artwork(String title, int yearCreated){
          setTitle(title);
          setYearCreated(yearCreated);
     }

     public addRoom(Room roomParam){
          if(roomParam != null){
               if(!roomParam.getArtworks().contains(this)){
                    if(room != null){
                         room.removeArtwork(this);
                    }
                    setRoom(roomParam);
                    room.getArtworks().add(this);//simplificación:
                    //suponemos que los arrays tienen el método "add" que
                    //añade el objeto en la primera posición libre

               }
          }
     }
     public Room getRoom(){
          return room;
     }

     public setRoom(Room roomParam){
          room = roomParam;
     }

     …
}
```

Room

```
class Room{
     private int id, m2;
     private String name;
     private Artwork[] artworks;

     public Room(int id, String name, int m2){
          setId(id);
          setName(name);
          setM2(m2);
          artworks = new Artwork[10];
     }

     public addArtwork(Artwork artwork){
          if(artwork != null && !artworks.contains(artwork)){
               if(artwork.getRoom()!=null){
                    artwork.getRoom().removeArtwork(artwork);
               }
               artwork.setRoom(this);
               artworks.add(artwork); //simplificación: suponemos que los arrays
               // tienen el método "add" que añade el objeto en la primera
               //posición libre
          }
     }
     public removeArtwork(Artwork artwork){
          artworks.remove(artwork);
          //simplicación: suponemos que los arrays tienen un
          //método "remove" que busca el objeto en el array y lo borra.
     }

     public Artworks[] getArtworks(){
          return artworks;
     }
     …
}
```

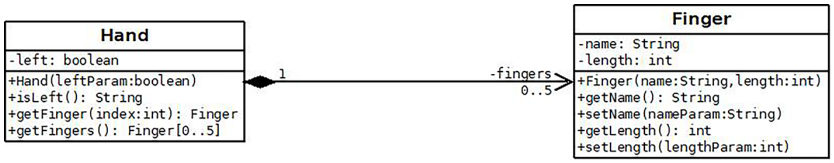
1.6.2. Asociación de composición

Como ya sabemos, a diferencia de los tres tipos de asociación anteriores, la composición añade una restricción fuerte: la existencia de un objeto de la clase componente depende de la existencia de la clase compuesta (o contenedora). Además, el objeto de la clase componente solo puede relacionarse con una clase compuesta mediante una asociación de composición.

Por todo lo anterior, la asociación de composición se codifica instanciando el objeto (u objetos) de la clase componente dentro del objeto de la clase compuesta. Esta instanciación normalmente se hace dentro del constructor de la clase compuesta. Esta manera de codificar la asociación de composición también implica que el objeto de la clase compuesta siempre conoce el objeto de la clase componente.

Ejemplo 12 (codificación) – Asociación de composición

Dada la siguiente relación entre una mano (`Hand`) y sus dedos (`Finger`).



Hand

```
class Hand{
     private boolean left;
     private Finger[] fingers;

     public Hand(boolean leftParam){
          left = leftParam;
          fingers = new Finger[5];
          fingers[0] = new Finger("Thumb",2); //creamos el objeto del dedo pulgar
          fingers[1] = new Finger("Index",5); //creamos el objeto del dedo índice
          fingers[2] = new Finger("Middle",8); //creamos el objeto del dedo corazón
          fingers[3] = new Finger("Ring",6); //creamos el objeto del dedo anular
          fingers[4] = new Finger("Little",4); //creamos el objeto del dedo meñique
     }
     …
}
```

Finger

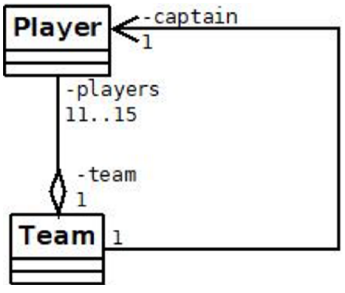
```
class Finger{
        private String name; //En este caso, no incluye un objeto de tipo Hand
        private int length;

        public Finger(String name, length){
                setName(name);
                setLength(length);
        }
        …
}
```

1.7. ¿Dos objetos pueden estar relacionados entre sí más de una vez?

La respuesta es «sí». Dos objetos pueden relacionarse entre sí por medio de diferentes asociaciones y, además, estas asociaciones no tienen por qué ser del mismo tipo. Veamos un ejemplo:

Ejemplo 13 (dos objetos relacionados entre sí más de una vez) – Capitán y jugador

Un equipo, por ejemplo de baloncesto, tiene jugadores. Normalmente uno de los jugadores de la plantilla es el capitán del equipo. Así pues, el objeto de ese jugador, además de relacionarse con el equipo como jugador, también se relaciona con el equipo como capitán.



2. Clase asociativa

2.1. Explicación

Este tipo de clase tiene sentido en el diseño de nuestro *software* y, por lo tanto, puede aparecer en un diagrama de clases UML. Sin embargo, en la mayoría de lenguajes de programación no existe una sintaxis especial que nos permita definir una clase asociativa.

Las clases asociativas aparecen en aquellas asociaciones (binarias, reflexivas o de agregación) en las que necesitamos guardar información específica de la relación o asociación (de ahí su nombre). Además añade semántica a la relación en forma de dos restricciones:

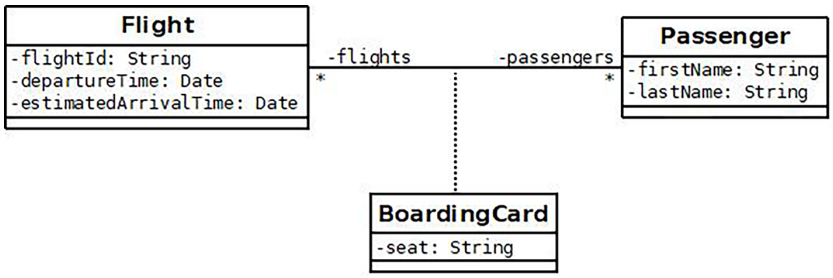
1) Dada una asociación entre una clase `A` y una clase `B` en la que hay una clase asociativa `C`, **solo puede haber un objeto de la clase `C` (clase asociativa) por cada combinación de objeto `A` y objeto `B` que estén relacionados**.

2) El objeto de la clase asociativa debe destruirse/eliminar cuando uno de los dos objetos que están relacionados por la asociación desaparece (i.e. se destruye) o cuando se elimina la relación entre ambos objetos. Esto es así porque la clase asociativa solo tiene sentido cuando los dos objetos de la asociación existen y están relacionados.

Las clases asociativas se representan con una línea discontinua que va de la clase asociativa a la línea continua que relaciona la asociación.

Ejemplo 14 (clase asociativa) – La tarjeta de embarque

Imaginemos la relación en la que un pasajero (`Passenger`) tiene un asiento asignado en un vuelo (`Flight`). En este caso, el asiento no es ni del pasajero ni del vuelo en sí, sino de la relación entre el pasajero y el vuelo.

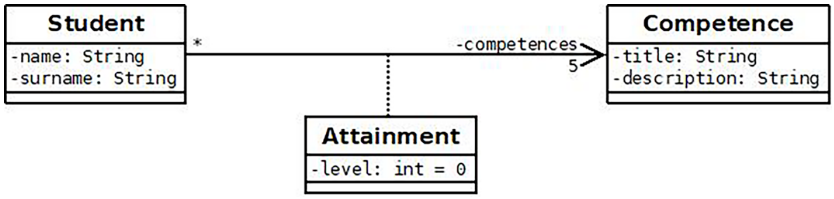


Como vemos, hemos tenido que crear una clase llamada `BoardingCard` que se relaciona con la asociación. Esta nueva clase incorpora información, en este caso el asiento (`seat`), que no es ni del vuelo ni del pasajero propiamente dicho, sino de los dos. Además se cumple la restricción que impone la clase asociativa de que solo puede haber un objeto `BoardingCard` para un pasajero concreto en un vuelo concreto. ¿Un mismo pasajero tiene dos asientos en un vuelo? No.

Veamos otro ejemplo:

Ejemplo 15 (clase asociativa) – Nivel competencial de los estudiantes

Imaginemos que queremos guardar el nivel de logro que tiene cada estudiante (`Student`) en una competencia transversal (`Competence`), como puede ser, comunicación oral, escrita, trabajo en grupo, etc. Un estudiante será evaluado en cinco competencias.



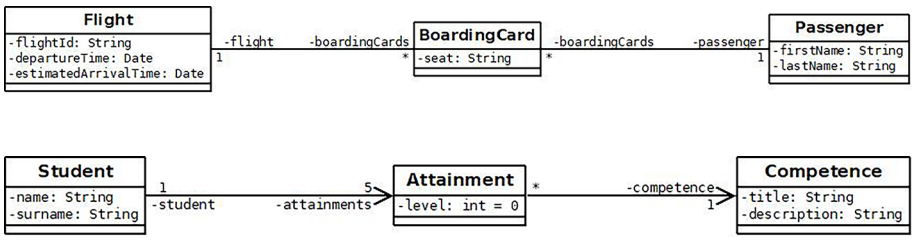
Como vemos, hemos tenido que crear una clase llamada `Attainment` que se relaciona con la asociación. Esta nueva clase incorpora información, en este caso el nivel de logro (`level`), que solo tiene sentido para el binomio estudiante-competencia. Igual que en el ejemplo anterior, un estudiante concreto para una competencia concreta solo puede tener un nivel de logro, no más.

2.2. Traduciendo a código

Como ya hemos comentado, los lenguajes de programación, en general, no proporcionan un mecanismo directo para implementar una clase asociativa. Por este motivo, lo más habitual es traducir o modificar la asociación en la que participa la clase asociativa por dos asociaciones binarias que van de las clases de la asociación binaria original a la clase asociativa.

Asimismo, para mantener la restricción de que solo puede haber un objeto de la clase asociativa por cada combinación de objetos de la asociación, es necesaria una multiplicidad 1 en el lado de las clases no asociativas. La multiplicidad en la clase asociativa es la misma que había en la asociación original. De igual manera, la navegabilidad original también se debe respetar.

Veamos cómo se modifican los ejemplos anteriores para poderlos codificar con un lenguaje de programación:



Con estas modificaciones, ya podemos codificar las clases como hemos comentado para las asociaciones binarias.

3. Clase parametrizada

3.1. Explicación

Una clase parametrizada, también conocida como *clase genérica* (en Java y C#) o *template* (en C++), es aquella en la que definimos atributos de tipo genérico. Gracias a esto, la clase parametrizada encapsula operaciones cuya funcionalidad no depende de un tipo de objeto determinado. El uso más común de este tipo de clase es el de crear clases que modelan el comportamiento de una colección, como por ejemplo: una pila, una cola, una lista, un árbol, una tabla de *hash*, etc.

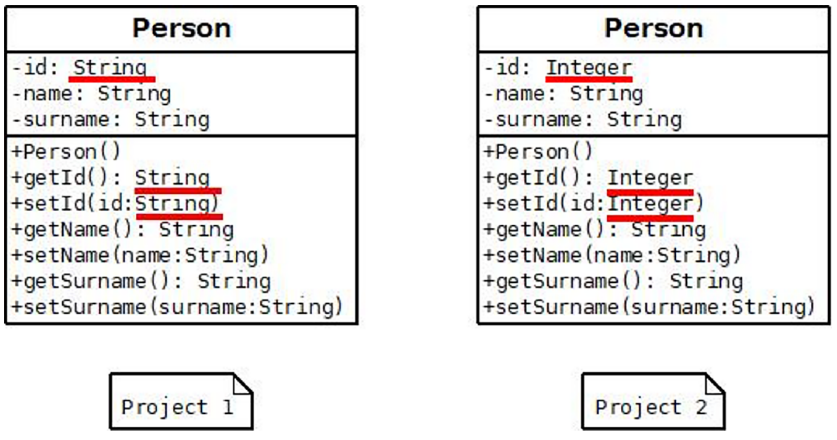
Las colecciones tienen operaciones o métodos que se realizan de la misma manera, independientemente del tipo de objeto que almacenan. Por ejemplo, si queremos añadir un elemento a una pila, da igual si guardamos un objeto de tipo `Person` o tipo `Dog`, la manera de guardar el objeto en la pila es la misma: poniéndolo encima de la colección (i.e. en primer lugar; *last-in*, *first-out* - LIFO).

Lenguajes como Java o C# tienen en su API (es decir, en el propio lenguaje) una gran cantidad de clases parametrizadas que modelan colecciones. Por ejemplo, Java proporciona las clases `ArrayList`, `LinkedList`, `HashMap`, `Stack`, `PriorityQueue`, entre otras. Por su parte, C# tiene `Dictionary`, `List`, `Queue`, `Stack`, etc.

Veamos un ejemplo donde nos puede ser útil hacer una clase parametrizada:

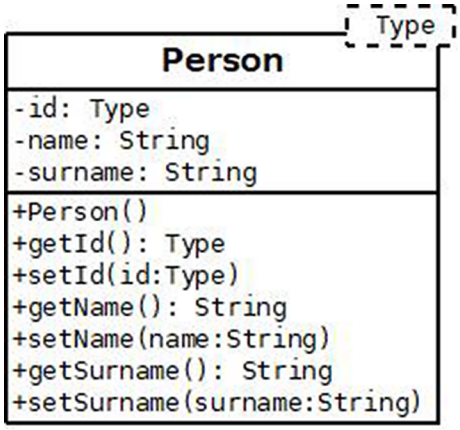
Ejemplo 16 (clase parametrizada) – Una clase `Person` para varios programas

Imagina que estamos programando en varios proyectos a la vez y en ambos nos piden que hagamos una clase `Person` a partir de los siguientes diagramas de clases.



Si nos fijamos, vemos que las clases son prácticamente idénticas, lo único que cambia es el tipo del atributo `id`, el cual es un `String` en el primer proyecto y un entero (`Integer`) en el segundo proyecto.

Así pues, en vez de codificar dos veces la clase `Person` con el pequeño cambio del tipo del atributo `id`, decidimos hacer una clase parametrizada.



Como podemos ver, ahora el tipo del atributo `id` es `Type`, que puede ser cualquier clase. Así pues, podemos instanciar un objeto de la clase `Person` de las siguientes formas (entre otras):

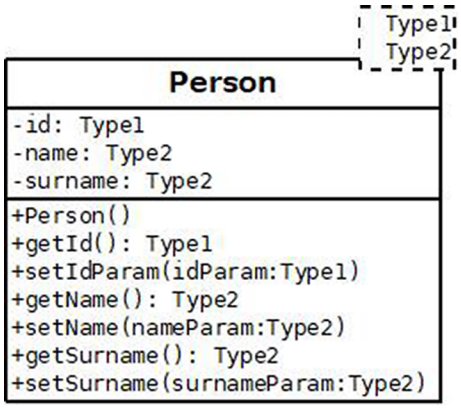
```
Person<String> personProject1 = new Person<String>();
Person<Integer> personProject2 = new Person<Integer>();
```

El tipo del atributo `id` será un parámetro de la propia clase. En nuestro ejemplo de código, hemos indicado el tipo mediante <> (piensa que el compilador sustituye `Type` por `String` o `Integer` o cualquier clase que indiquemos, según sea el caso).

En el UML se indica el nombre que se le da al tipo genérico (lo hemos llamado `Type`, pero podría haber sido otro nombre) mediante una pequeña caja situada en la parte superior derecha.

En el ejemplo anterior, gracias al uso de una clase parametrizada, hemos logrado codificar una sola vez la clase `Person` abstrayéndonos del tipo del atributo `id`. Ahora la clase `Person` es genérica, es decir, sirve para cualquier contexto donde solo el atributo `id` cambia de tipo.

Una clase parametrizada puede recibir tantos tipos como se necesiten. Veamos un ejemplo en el que hay dos tipos:



3.2. Traduciendo a código

Cada lenguaje de programación tiene sus particularidades y reglas a la hora de definir las clases parametrizadas. Para que puedas entender las clases parametrizadas desde un punto de vista de la codificación, en términos generales podríamos decir que su codificación sería como sigue:

```
class Person<Type>{
     private Type id;
     private String name, surname;

     public Person>(){}

     public Type getId(){ return id;}

     public setId(Type idParam){ id = idParam;}
     …
}
```

Asimismo, no todos los lenguajes de programación soportan la definición y uso de clases parametrizadas.

Resumen

Asociaciones

- Una asociación es la relación entre objetos o instancias. Dos objetos pueden estar relacionados entre sí por más de una asociación.

- Los principales tipos de asociación son: binaria, de agregación, de composición y reflexiva.

- La asociación binaria se da entre dos instancias donde cada una de ellas existe de forma independiente a la existencia de la otra.

- La asociación de agregación es similar a la binaria, pero incorpora la semántica de que uno de los dos objetos es componente de la otra (llamada *compuesta*).

- La asociación reflexiva es un caso particular de la asociación binaria en el que un objeto de una clase se relaciona con otro objeto de la misma clase.

- Tanto la asociación binaria como la de agregación y la reflexiva se codifican de igual manera.

- La asociación de composición es como la de agregación, pero la existencia del objeto componente depende de la existencia del objeto compuesto. Además, el objeto componente solo puede relacionarse mediante composición con un único objeto compuesto. No obstante, puede relacionarse con otros objetos mediante otros tipos de asociación.

- Toda asociación tiene tres propiedades: multiplicidad, navegabilidad y rol.

- La multiplicidad está formada por el número mínimo y máximo de objetos que participan de una clase en la relación. Al número mínimo se le llama *opcionalidad* o *participación*, mientras que al número máximo se le denomina *cardinalidad*. Cada asociación tiene dos multiplicidades, una por cada clase que participa en la relación.

- La navegabilidad indica si un objeto de una clase conoce la existencia de un objeto de la otra clase. Si solo un objeto de la asociación tiene conocimiento de la relación, entonces la navegabilidad es unidireccional. En caso contrario, es bidireccional.

- El rol es una etiqueta que sirve para aportar información contextual a la asociación. Además, ayuda a la hora de codificar porque sugiere un nom-

bre para el atributo de una clase que hace de referencia a los objetos de la otra clase de la relación.

Clase asociativa

- Las clases asociativas aparecen en aquellas asociaciones (binarias, reflexivas o de agregación) en las que necesitamos guardar información específica de la relación o asociación.

- Solo puede haber un objeto de la clase asociativa por cada combinación de objeto `A` y objeto `B` que estén relacionados mediante la asociación.

- Cuando uno de los dos objetos que están relacionados por la asociación desaparece (i.e. se destruye) o cuando se elimina la asociación entre ambos objetos, entonces el objeto de la clase asociativa también debe destruirse.

- La mayoría de los lenguajes de programación no permiten crear clases asociativas. Por este motivo, debemos modificar o transformar la clase asociativa por una clase normal y convertir la asociación en la que participa la clase asociativa en dos asociaciones binarias que van de las clases de la asociación binaria original a la clase asociativa.

Vídeo de interés

Puedes ver un resumen de los tipos de asociación y de la clase asociativa en el vídeo «Relación entre objetos» que encontrarás en el aula de la asignatura.

Clase parametrizada

- Una clase parametrizada nos permite abstraernos del tipo de todos o algunos de los atributos de la clase encapsulando operaciones cuya funcionalidad no depende del tipo de objeto que sea el atributo de esa clase.

- Gracias a una clase parametrizada, generalizamos el uso de la clase a diferentes contextos y, al mismo tiempo, ahorramos en horas de codificación.

- El uso más frecuente de una clase parametrizada es en la implementación de colecciones: pila, cola, lista, etc.

- La clase parametrizada también es conocida como *clase genérica* o *template*.

Bibliografía

Booch, G.; Maksimchuk, R.; Engle, M.; Young, B. J.; Conallen, J.; Houston, K. y otros (2007). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional. ISBN: 978-0201895513.

Griffiths, D.; Griffiths, D. (2019). *Head First Kotlin*. O'Reilly Media, Inc. ISBN: 978-1491996690.

Hunt, A.; Thomas, D. (2019). *The Pragmatic Programmer: your journey to mastery, 20th Anniversary Edition* (2.ª ed.). Addison-Wesley Professional. ISBN: 978-0135956977.

Microsoft (s. f.). «C# Guide» [en línea]. Microsoft. <https://docs.microsoft.com/en-us/dotnet/csharp/>

Phillips, D. (2018). *Python 3 Object-Oriented Programming* (3.ª ed.). Packt Publishing. ISBN: 978-1789615852.

Pollice, G.; West, D.; McLaughlin, B. (2006). *Head First Object-Oriented Analysis and Design*. O'Reilly Media, Inc. ISBN: 978-0596008673.

Robinson, S.; Nagel, C.; Watson, K.; Glynn, J.; Skinner, M.; Evjen, B. (2004). *Professional C#* (3.ª ed.). Wrox. ISBN: 978-0764557590.

Seidl, M.; Scholz, M.; Huemer, C.; Kappel, G. (2015). *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Heidelberg: Springer.

Sharp, J. (2007). *Microsoft Visual C# 2008 Step by Step*. Microsoft Press. ISBN: 978-0735624306.

Weisfeld, M. (2019). *The Object-Oriented Thought Process* (5.ª ed.). Addison-Wesley Professional. ISBN: 978-0135182130.