Herencia (relaciones entre clases)

PID_00272786

David García Solórzano

Tiempo mínimo de dedicación recomendado: 4 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 (2020)

Primera edición: febrero 2020

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

Índice

Introducción

Las clases, además de relacionarse mediante la interacción entre sus objetos, también pueden relacionarse entre ellas a nivel de la propia clase. La relación entre clases por excelencia en el paradigma de la programación orientada a objetos (POO) es la generalización / especialización, comúnmente conocida como *herencia*. Gracias a la herencia podremos utilizar una clase (llamada *superclase*) para crear nuevas clases (llamadas *subclases*) que modifiquen algunos aspectos de la clase original. Así pues, aprenderemos a crear subclases que heredarán los atributos y métodos de una superclase. Además, veremos cómo sobrescribir métodos para hacer que las subclases se comporten con las particularidades que queremos que tengan. También hablaremos de los conceptos de herencia simple y múltiple, así como de transitividad.

En relación con la herencia, conoceremos qué son una clase abstracta y una clase final. A continuación, presentaremos la interfaz, un elemento similar a una clase abstracta, pero que tiene diferencias importantes que debemos conocer. Finalmente, explicaremos uno de los pilares de la POO que está íntimamente relacionado con la herencia, el polimorfismo.

Además, veremos cómo se representan estas relaciones, tipos de clases y la interfaz 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 las clases se relacionan entre sí dentro de un programa basado en el paradigma de la programación orientada a objetos (POO). Como consecuencia de este objetivo aparecen otros:

1. Conocer la relación de generalización o especialización (más conocida como *herencia*) que se puede dar entre clases, así como los términos *superclase* y *subclase*.

2. Entender los tipos de herencia que existen y por qué la herencia simple es más común en los lenguajes de programación que la herencia múltiple.

3. Comprender la particularidad que tiene el constructor de una subclase.

4. Saber qué es una clase abstracta y su utilidad.

5. Saber qué es una clase final y su utilidad.

6. Saber qué es una interfaz desde el punto de vista del paradigma de la POO y su utilidad.

7. Ver y entender el potencial que la herencia, mediante diferentes mecanismos (por ejemplo, el polimorfismo), puede ofrecernos a la hora de diseñar e implementar nuestros programas.

1. Herencia o relación de generalización / especialización

1.1. Concepto

La relación de generalización / especialización es también conocida como *herencia*, aunque es más frecuente el uso de este último término. Así pues, a partir de ahora hablaremos de herencia. La herencia, junto con la abstracción, la encapsulación y el polimorfismo (que veremos más adelante), es una de las características más importantes de la programación orientada a objetos. Cuando desarrolles programas de media o gran envergadura, necesitarás usar el mecanismo de herencia.

Definición de herencia

La relación que existe entre dos clases en la que una clase (llamada *clase padre*, *base* o *superclase*) generaliza el comportamiento de la otra clase (llamada *clase hija*, *clase derivada* o *subclase*). O, visto desde el otro punto de vista, la subclase especializa el comportamiento de la superclase.

A nivel más práctico podemos definir la herencia como:

Un mecanismo de la programación orientada a objetos que permite definir una clase nueva (la subclase) a partir de una clase ya existente (la superclase) describiendo solamente las diferencias entre las mismas.

Resumiendo, en términos generales:

- Una superclase contiene los atributos y métodos comunes que son heredados por una o más subclases.

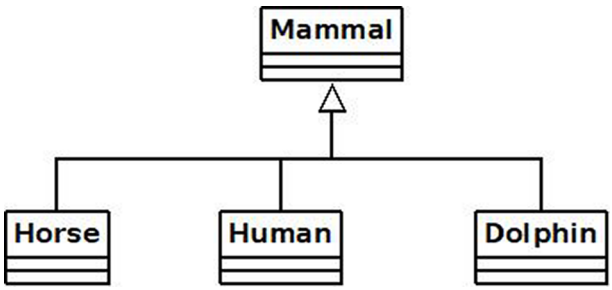
- Una subclase reutiliza (hereda) los atributos y métodos de la superclase sin tener que recodificarlos. Asimismo, la subclase puede modificar los métodos heredados mediante el mecanismo de sobrescritura –lo veremos más adelante. Además, puede definir atributos y métodos nuevos que son propios de la subclase.

Así pues, gracias a la herencia se reutiliza una gran cantidad de código, por lo que se reduce o elimina la redundancia. En este punto, cabe resaltar que, a diferencia de las asociaciones –que modelan la relación entre objetos–, la herencia es una relación exclusivamente entre clases. Veamos un ejemplo:

Ejemplo 1 (herencia) – Mamíferos

A Marina le hablan de los mamíferos en la guardería y le explican que los mamíferos respiran, comen, etc. Asimismo, le cuentan que un caballo es un mamífero, pero que los humanos también lo somos. Además, aunque no lo parezcan, los delfines también lo son. Por lo tanto, a Marina lo que le están diciendo, en términos del paradigma de programación orientada a objetos, es que las clases `Horse` ('caballo'), `Human` ('humano') y `Dolphin` ('delfín') especializan (o heredan) el comportamiento de la clase `Mammal` ('mamífero'). En este caso, la clase `Mammal` es la clase padre (superclase) de la relación de la cual heredan las clases hijas (subclases): `Horse`, `Human` y `Dolphin`.

En UML la herencia se representa de la siguiente manera:



Como podemos ver, se dibuja un triángulo blanco en la superclase (en el ejemplo anterior, `Mammal`) y de él salen líneas hacia las subclases (`Horse`, `Human` y `Dolphin`). Es importante darse cuenta de que la relación de herencia no tiene multiplicidad, puesto que no intervienen instancias u objetos, sino que simplemente estamos diciendo que una subclase especializa las características de otra clase (i.e. superclase). Tampoco existen la navegabilidad ni el rol.

Se puede considerar la herencia como una relación del tipo `<<es-un>>`, por ejemplo, un caballo (subclase) es un tipo de mamífero (superclase), un humano (subclase) es un tipo de mamífero (superclase), un delfín (subclase) es un tipo de mamífero (superclase), un coche (subclase) es un tipo de vehículo (superclase), etc.

1.2. Traduciendo a código

La sintaxis que permite codificar la herencia entre clases depende del lenguaje de programación. Sin embargo, muchos lenguajes comparten sintaxis. Por ejemplo, Java, Scala, TypeScript y PHP utilizan la palabra reservada `extends` para indicar que una clase extiende o hereda de otra.

```
class Subclass extends Superclass{
    //TODO
}
```

Por su parte, C#, C++ y Kotlin utilizan los dos puntos (`:`).

```
class Subclass : Superclass{
    //TODO
}
```

En Python, sin embargo, se opta por la siguiente sintaxis:

```
class Subclass(Superclass):
    //TODO
```

1.3. ¿Qué hereda la subclase de la superclase?

Cuando utilizamos el mecanismo de herencia es importante saber qué miembros de la superclase heredan las subclases y cuáles no. Esto depende, en parte, del lenguaje de programación que se utiliza. No obstante, podemos decir que de manera general las subclases sí heredan:

- **Todos los atributos y métodos.** Sin embargo, su visibilidad dentro de la subclase dependerá del modificador de acceso definido para cada uno de ellos en la superclase. Con esto queremos decir que, dependiendo del modificador de acceso indicado en la superclase, la subclase podrá usar un atributo o método como si se hubiera codificado en la propia subclase o no. Es decir, según el modificador de acceso declarado en la superclase, la subclase tendrá acceso directo o no al atributo o método heredado.

Por el contrario, las subclases no heredan:

- **Constructores de la superclase:** en el módulo «Abstracción y encapsulación» dijimos que el constructor no es considerado miembro de una clase porque precisamente no se hereda. Cada clase debe definir sus propios constructores, aunque muchos lenguajes llaman (o permiten llamar) a un constructor de la superclase desde el constructor de la subclase.

- **Destructor:** aquellos lenguajes que tienen un destructor *de facto* (no un método que actúa como si fuera un destructor, por ejemplo, `finalize` en Java) no permiten heredar el destructor en las subclases por el mismo motivo explicado para el constructor.

- **Constructor estático:** aquellos lenguajes que permiten definir un constructor estático, como C#, no permiten heredarlo en las subclases.

Como hemos dicho, el modificador de acceso asignado a cada miembro de la superclase restringe si la subclase puede acceder a él de manera directa, es decir, como si hubiera sido definido en la propia subclase. En general, los lenguajes de programación orientados a objetos cumplen las siguientes reglas:

a) En los lenguajes en los que no se puede indicar cómo se realiza la herencia (por ejemplo, Java, C#, PHP, Scala, etc.), los atributos y métodos heredados en las subclases son heredados con el mismo modificador de acceso que el especificado en la superclase (a no ser que se diga lo contrario mediante sobrescritura). En otros lenguajes, como C++, se puede indicar si la herencia es `public`, `protected` o `private`. Por ejemplo, si la herencia es `protected`, entonces los atributos y métodos de la superclase que sean `public` o `protected` serán `protected` en la subclase.

Ved también

El concepto de *sobrescritura* lo veremos en el apartado 1.7 de este módulo.

b) `Public`: los miembros definidos como `public` en la superclase, al ser heredados por las subclases, son visibles tanto dentro de las subclases como fuera de ellas. Es decir, pueden ser llamados como si estos hubieran sido codificados en las subclases y, a su vez, definidos como `public`.

```
//Código Java
class A{
     public void method(){
          print "Hola";
     }
}

class B extends A{ //"extends" sirve para decir "hereda".
     public B(){
        super();
        method(); //podemos llamarlo porque forma parte de la
//clase B (viene heredado de A). Imprimirá "Hola".
     }
}

class C{
     public C(){ //Imprime 2 veces "Hola"
        B objectB = new B();
        objectB.method(); //podemos llamarlo porque es un método
//público de la clase B que viene heredado de la clase A.
     }
```

Ved también

En el apartado 1.6 de este módulo veremos cómo llamar explícitamente a un constructor de la superclase desde el constructor de la subclase. En Java se usa la palabra reservada `super`.

c) `Protected`: los miembros definidos como `protected` en la superclase, al ser heredados por las subclases, son solo visibles dentro de las subclases. Es decir, no son accesibles desde fuera de las subclases. Sin embargo, dentro de las subclases se pueden utilizar como si estos se hubieran codificado en las subclases.

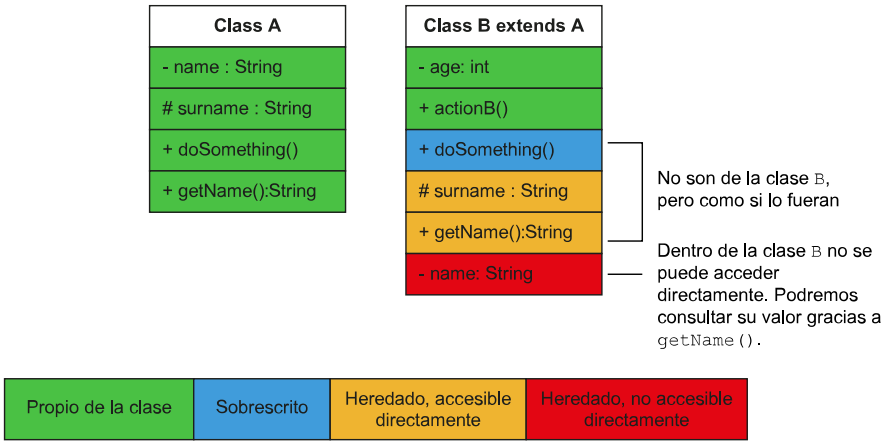
```
//Código Java
class A{
     protected void method(){
          print "Hola";
     }
}

class B extends A{ //"extends" sirve para decir "hereda".
     public B(){
        method(); //podemos llamarlo porque forma parte de la
//clase B (viene heredado de A). Imprimirá "Hola".
     }
}

class C{
     public C(){ //Imprime 1 vez "Hola" con la llamada "new B()"
        B objectB = new B();
        objectB.method(); //error, es protected en B, no public
     }
}
```

d) `Private`: los miembros declarados como `private` en la superclase son heredados por las subclases, pero no son accesibles directamente desde dentro de las subclases. En el caso de los atributos, si se les quiere asignar un valor, o consultar el que tienen, la operación se debe hacer utilizando un *getter* o *setter* `public` o `protected` heredado de la superclase.

```
//Código Java
class A{
     private int value;

     public A(){
          value = 50; //sólo es accesible dentro de la clase A
     }

     public int getValue(){
          return value;
     }
}

class B extends A{ //"extends" sirve para decir "hereda".
     public B(){
          print getValue(); //getValue es accesible y devuelve 50
          print value; //error, value es private en A y, aunque es
//heredado por la subclase B, dentro de ella no puede ser accedido
//directamente.
     }
}

class C{
     public C(){
          B objectB = new B(); //Asumimos que no da error…
          objectB.getValue(); //getValue es accesible, devuelve 50
          print objectB.value; //error, este atributo no es accesible
     }
}
```

Como ya comentamos en el módulo «Abstracción y encapsulación», algunos lenguajes de programación orientados a objetos definen otros modificadores de acceso, por ejemplo, `internal` (C#) y `package-private` (Java). Para saber cómo estos modificadores afectan durante la herencia, debemos consultar la documentación del lenguaje en cuestión. Asimismo, algunos lenguajes de programación definen comportamientos especiales para las clases anidadas que heredan de la clase contenedora.

Finalmente, vamos a ver con un ejemplo cómo hay que imaginarse el concepto de herencia.



Todo lo que la clase `B` hereda de la clase `A` es como si estuviera recogido en un objeto de `A` que está dentro de `B`. Sin embargo, hay que tener presente que cuando se instancia un objeto de la clase `B`, no se crea un objeto `B` y un objeto `A`, sino que solo se crea un objeto `B` que contiene lo de `B` y lo de `A`. Así pues, hay que tener cuidado en el diseño de las clases, ya que, como se hereda todo de la superclase, si hay algo de la superclase que no queremos en la subclase, lo tendremos igualmente y, lo que es peor, ocupando memoria para cada objeto instanciado de la subclase.

1.4. Tipos de herencia

Existen dos tipos de herencia: simple y múltiple. Vamos a ver cada una de ellas.

1.4.1. Herencia simple

La herencia simple es el caso más común de herencia. Es el más utilizado porque es el tipo de herencia permitido por todos los lenguajes de programación orientados a objetos. ¿En qué consiste?

Definición de herencia simple

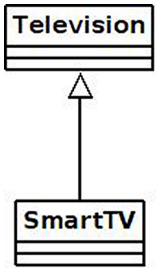
En la herencia simple, una subclase solo puede heredar directamente de una superclase.

El ejemplo anterior de los mamíferos es un claro caso de herencia simple. Veamos alguno más para acabar de entender el concepto.

Ejemplo 2 (herencia simple) – Smart TV

Elena y David han comprado un nuevo televisor, para ser más exactos, han adquirido una smart TV. ¡Qué diferencia con el que tenían hasta ahora! Cierto, pero la funcionalidad principal, que consiste en mostrar los canales de televisión, sigue siendo la misma. Asimismo, tanto el nuevo como el viejo televisor permiten subir y bajar el volumen, cambiar de canal e incluso cambiar el contraste. Entonces, ¿qué ha cambiado? Pues que el nuevo televisor (smart TV) ha especializado el concepto original de televisor. De hecho, ha cogido como punto de partida las funcionalidades de un televisor y las ha mejorado.

La smart TV no deja de ser un televisor, lo único es que además de hacer todo lo que hace el televisor antiguo, le añade un comportamiento nuevo, que es acceder a internet.



Por lo tanto, la clase `SmartTV` hereda de la clase `Television`. De este modo, `SmartTV` es una subclase que especializa la superclase `Television`.

1.4.2. Herencia múltiple

La herencia múltiple está presente en nuestro entorno, por lo tanto, conceptualmente tiene sentido y, por ello, es posible representarla en un diagrama de clases UML. Sin embargo, esta es menos común en los programas que la herencia simple debido a que muchos lenguajes de programación orientados a objetos no la soportan, como es el caso de Java, C#, TypeScript y PHP. Sin embargo, lenguajes como C++ y Python, sí la soportan. Pero ¿en qué consiste la herencia múltiple?

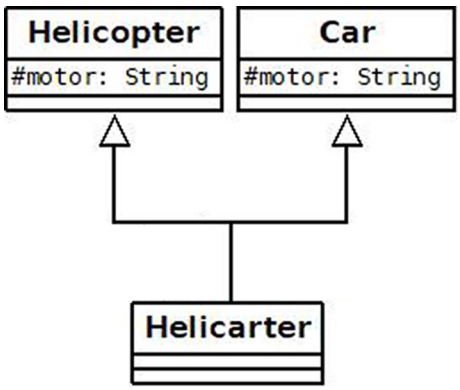
Definición de herencia múltiple

En la herencia múltiple una clase hija puede heredar directamente de más de una superclase.

Veamos un par de ejemplos:

Ejemplo 3 (herencia múltiple) – El helicarter

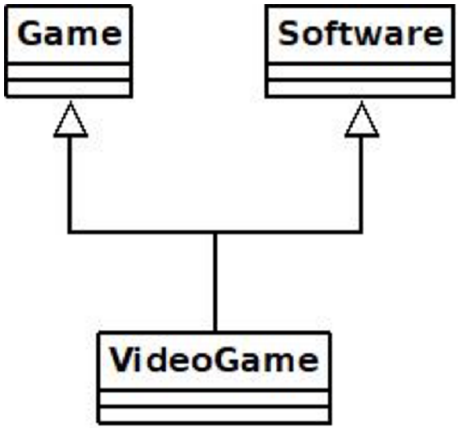
Imaginemos que queremos inventar un nuevo vehículo llamado «helicarter» que tiene las virtudes del helicóptero y del coche. Así pues, es fácil pensar que el `Helicarter` heredará características de las clases `Helicopter` y `Car`, por lo que el diagrama de clases UML quedaría de la siguiente manera:



Ejemplo 4 (herencia múltiple) – El videojuego

Un ejemplo más real es el videojuego. Este elemento de entretenimiento es a la vez un juego (clase `Game`) con sus reglas, número de jugadores, etc., y un programa de ordenador

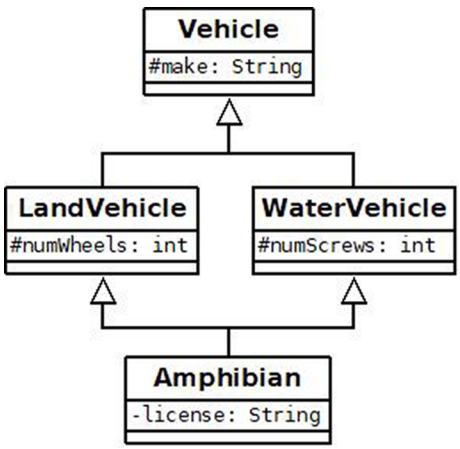
(clase `Software`) con sus líneas de código, con la información del lenguaje de programación utilizado, etc. Así pues, el diagrama de clases que relaciona estas tres clases sería:



Vistos los ejemplos anteriores, vamos a intentar explicar por qué muchos lenguajes –como Java o C#– decidieron no dar soporte a la herencia múltiple. Si volvemos al ejemplo del `Helicarter`, vemos que las dos superclases tienen un atributo llamado `motor` que es un `String`. Pues bien, la subclase `Helicarter` heredará ambos atributos. Debido a esto, el lenguaje de programación deberá dotar al programador de algún mecanismo que le permita indicar a cuál de los dos atributos repetidos está haciendo referencia en cada instrucción del código. Esto hace confusa la lectura del código y, a su vez, puede ocasionar errores mientras se codifica.

Otro de los problemas que puede provocar la herencia múltiple es una ambigüedad conocida como *problema del diamante*. Este problema básicamente consiste en una ambigüedad que surge cuando las clases `B` y `C` heredan de una superclase `A` y, posteriormente, una cuarta clase `D` hereda de `B` y `C`. Entonces, si desde la clase `D` se llama a un método definido en la clase `A`, ¿por dónde hereda `D` dicho método?, ¿por `B` o por `C`? Cada lenguaje de programación que permite herencia múltiple resuelve este problema de distinta manera. Es por eso que muchos lenguajes, con tal de simplificar los potenciales problemas que hay que resolver, no permiten la herencia múltiple.

Ejemplo 5 (el problema del diamante) – El vehículo anfibio, un problema



El diagrama de clases anterior muestra claramente el problema del diamante. La clase `Amphibian` tendrá los atributos `license` ('matrícula'), `numWheels` ('número de ruedas') y `numScrews` ('número de hélices'). Sin embargo, también tendrá el atributo `make`, pero ¿por dónde lo recibe?, ¿por `LandVehicle` o por `WaterVehicle`?

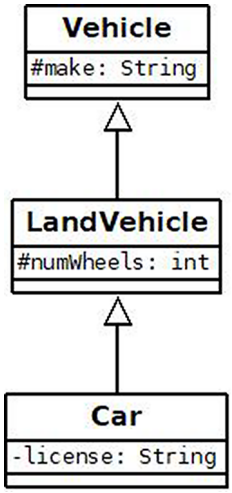
Finalmente, decir que si tenemos un diagrama de clases en el que aparece herencia múltiple y el lenguaje de programación que usaremos no lo soporta, entonces deberemos modificar, durante la codificación, la relación entre clases basada en herencia múltiple para que se adapte de la mejor manera posible al lenguaje de programación escogido. Hay que recordar que, en primera instancia, un diagrama de clases UML debe ser independiente al lenguaje de programación utilizado para codificarlo. Así pues, si al modelar un problema nos aparece una herencia múltiple, la debemos dibujar en el diagrama de clases, puesto que conceptualmente tiene sentido. Cualquier cambio que sea necesario hacer debido al lenguaje de programación que se va a utilizar, lo realizaremos en el momento de la codificación. También podemos hacer dos versiones del diagrama de clases, una genérica (i.e. independiente del lenguaje) y otra adaptada al lenguaje escogido. De esta manera, delimitamos perfectamente qué concierne a la fase de diseño y qué a la fase de implementación del programa o proyecto.

1.5. Transitividad

Muchos lenguajes de programación permiten la transitividad entre clases, es decir, una clase `B` hereda de una superclase `A` y, a su vez, una clase `C` hereda de la clase `B`, siendo `A` y `B` superclases para la clase `C`. En este caso, la clase `B` será la superclase directa de la clase `C`.

Transitividad

El problema del diamante del Ejemplo 5 es también un caso de transitividad.



En este ejemplo, la clase `LandVehicle` tiene el atributo `numWheels` definido en la propia clase y el atributo `make` heredado de `Vehicle`, al que podrá acceder directamente tal como accede a `numWheels`. Es decir, `make` en la clase `LandVehicle` se comporta como si se hubiera definido en la propia clase `LandVehicle`. En la clase `LandVehicle`, el atributo `make` sigue siendo `protected`. Por su parte, la clase `Car` tendrá el atributo privado `license` declarado en la propia clase, así como los atributos protegidos `numWheels` y `ma-`

ke heredados de las superclases LandVehicle y Vehicle, respectivamente. La clase Car hereda el atributo make porque LandVehicle (su superclase directa) lo ha heredado previamente.

1.6. Constructor de la subclase

Como ya hemos comentado, una subclase automáticamente hereda, además de los métodos, todos los atributos de la superclase. Estos atributos normalmente deben ser inicializados cuando el objeto es instanciado. Habitualmente hacemos esta inicialización en el constructor. En muchos lenguajes, si no creamos un constructor, el compilador creará uno por defecto. Por todo ello, es una buena práctica que el constructor de la subclase llame al constructor de su superclase como parte de la inicialización. De hecho, los atributos privados heredados no son visibles en la subclase, y para inicializarlos, o lo hacemos por medio de métodos heredados que sean visibles en la subclase o mediante el constructor de la superclase.

Así pues, la mayoría de lenguajes fuerzan a que si se quiere llamar explícitamente al constructor de la superclase, dicha llamada tenga que ser la primera instrucción del constructor de la subclase. Dependiendo del lenguaje, esta llamada al constructor de la superclase se hará de una manera u otra. Por ejemplo, en Java se usa, dentro del cuerpo del constructor de la subclase, el método especial super, cuya firma coincide con la firma de cualquiera de los constructores de la superclase.

```java
//Código Java
public class SuperC{
     private String name;

     public SuperC(){
          name = "David";
     }

     public SuperC(String nameParam){
          name = nameParam;
     }
}
```

```java
//Código Java
public class SubC extends SuperC{
     protected String surname;

     public SubC(){
          super("Elena");
          surname = "Lazaro";
     }

     public SubC(String surnameP){
          super();
          surname = surnameP;
     }
}
```

Por su parte, en PHP, en vez de super, se usa parent::__construct(), donde __construct() funciona como super, es decir, su firma debe coincidir con la del constructor de la superclase.

Finalmente, lenguajes como C#, C++, Kotlin o Scala fuerzan a que se invoque al constructor de la superclase en la propia firma del constructor de la subclase. En C#, por ejemplo, esta llamada se hace mediante la palabra reservada base.

```
//Código C# de la subclase SubC
public class SubC : SuperC{
     protected String surname;

     public SubC():base("Elena"){
          surname = "Lazaro";
     }

     public SubC(String surnameP):base(){
          surname = surnameP;
    }
}
```

En C++, al permitir herencia múltiple, no usa una palabra reservada, sino que directamente utiliza el nombre de la superclase. Scala, por su parte, es bastante similar.

```
//Código C++ de la subclase SubC
class SubC : SuperC{
     protected:
          string surname;

     public:
          SubC():SuperC("Elena"){
               surname = "Lazaro";
          }

          SubC(String surnameP):SuperC(){
               surname = surnameP;
          }
}
```

Kotlin permite usar los dos tipos de llamada anteriores: en línea (en la definición de la clase) y con el uso de la palabra `super` (como en Java), pero en vez de en el cuerpo del constructor de la subclase, se invoca en la firma.

```
//Código Kotlin de la superclase SuperC
open class SuperC(var name:String = "David"){

}
```

```
//Código Kotlin de la subclase SubC
class SubC(var surname:String ="Lazaro"):SuperC(){

     constructor():this("Lazaro"){
          name = "Elena"
     }
}
```

Llegados a este punto, cabe saber que en muchos lenguajes, como Java o C#, si no llamamos explícitamente en el constructor de la subclase al constructor de la superclase –mediante `super` en Java y `base` en C#–, entonces el compilador llamará automáticamente al constructor por defecto (es decir, sin argumentos) de la superclase. En este caso, si la superclase no tiene un constructor por defecto definido por el programador, entonces obtendremos un error en tiempo de compilación.

Método `this`

Muchos lenguajes de programación como Java, Kotlin, C#, etc., permiten usar un método especial llamado `this` para llamar desde un constructor a otro constructor de la propia clase y así eliminar redundancia de código.

1.7. Sobrescritura

Muchos lenguajes de programación permiten que una subclase modifique (redefina) los métodos heredados de la superclase a los que tiene acceso directo. Es decir, de los tres modificadores de acceso que hemos comentado en el apartado 1.3, una subclase puede redefinir o sobrescribir los métodos declarados

como `public` o `protected` en la superclase, pero no `private`. A esta modificación o redefinición de un método de la superclase en la subclase se le llama **sobrescritura** (*overriding*).

Por lo general, la sobrescritura se rige, en muchos lenguajes, por las siguientes reglas:

a) Además de tener acceso directo al método desde dentro de la subclase, el método que se quiere sobrescribir debe ser declarado o marcado como `virtual` en la superclase. Dependiendo del lenguaje, los métodos son marcados `virtual` por defecto o no. Por ejemplo, en Java o Scala todos los métodos son `virtual` por defecto, mientras que en C++ o C#, por defecto, no lo son. Otros lenguajes como Kotlin usan un concepto similar con el *keyword* `open` y se comportan parecido a C#.

b) Excepto en lenguajes más antiguos como C++, el modificador de acceso (nivel de visibilidad) que le debemos asignar al método sobrescrito debe ser igual o menos restrictivo que el definido para la superclase. Es decir, si un método en la superclase es `protected`, este no puede ser sobrescrito en la subclase con un nivel de acceso `private`. En este caso, el método sobrescrito solo puede ser definido en la subclase como `protected` (idéntico nivel o modificador de acceso que en la superclase) o `public`.

c) Las modificaciones realizadas en el método sobrescrito dentro de la subclase solo afectan a la subclase y a aquellas clases que hereden de esta, pero en ningún caso afectan a la superclase.

d) Los métodos de la superclase marcados como `final` (Java, Scala, Kotlin, PHP o C++11) o `sealed` (en C#) no pueden ser sobrescritos por sus subclases (ni subclases de las subclases, etc.).

e) Los métodos de la superclase marcados como `static` no pueden ser sobrescritos por sus subclases (y subclases de las subclases, etc.). El motivo es que los métodos estáticos son vinculados en tiempo de compilación usando el tipo de clase, no de ejecución usando objetos.

f) Los atributos no pueden ser sobrescritos en las subclases, pero sí escondidos (*shadowed*). ¿Cómo? Declarando un atributo en la subclase con el mismo nombre que el que tiene un atributo de la superclase, entonces el atributo heredado de la superclase queda oculto, que no eliminado, lo que hace más difícil el acceso a él.

```
class A{
     protected String name;

     public A(){
          name = "David";
     }

     public String getName(){
          return name;
     }
}
```

```
class B extends A{
     protected String name;

     public B(){
          name = "Elena";
     }

     public hello(){
          print getName();
          print name;
     }
}
```

Si en otra clase hacemos:

```
B objectB = new B();
objectB.hello(); //Imprime primero "David" y luego "Elena".
```

Sin embargo, algunos lenguajes como Scala o Kotlin permiten la sobrescritura de atributos (en Kotlin en verdad son *properties*, que es una extensión del concepto de atributo).

¿Cuándo es útil sobrescribir un método en una subclase? Veamos un ejemplo:

Ejemplo 6 (sobrescritura) – No todas las personas cortamos igual

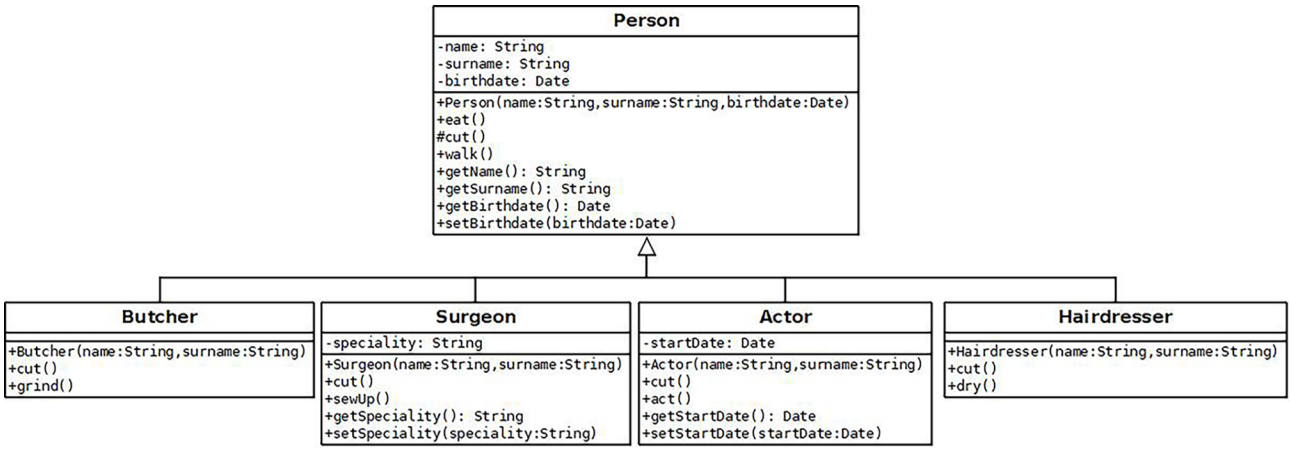
David, que es peluquero, ha quedado con sus amigos: José (que es carnicero), Javi (que es cirujano) y Carlos (que es actor). Los cuatro ejercen una profesión que toda persona puede realizar. De hecho, un carnicero, un cirujano, un actor y un peluquero no dejan de ser personas. Así pues, podemos modelar este contexto como se muestra en la imagen siguiente.

En este ejemplo, las subclases `Butcher` ('carnicero'), `Surgeon` ('cirujano'), `Actor` y `Hairdresser` ('peluquero') heredan los atributos y métodos de la superclase `Person`. En el caso de los miembros privados (`name`, `surname` y `birthdate`), estos son heredados, pero las cuatro subclases no tendrán acceso directo dentro de ellas; pero para el resto de miembros, sí que tendrán acceso directo. Es decir, es como si en `Butcher`, por ejemplo, hubiéramos escrito o codificado `eat`, `cut`, `walk`, `getName`, `getSurname`, `getBirthdate` y `setBirthdate`. Recordemos que el constructor de la superclase no se hereda.

Por su parte, las clases `Surgeon` y `Actor` añaden un atributo privado: `speciality` y `startDate`, respectivamente. Asimismo, las cuatro subclases añaden un método público: `grind` ('picar'), `sewUp` ('coser'), `act` ('actuar') y `dry` ('secar'), respectivamente.

Finalmente, en las cuatro subclases aparece el método `cut`, que ya está definido en la superclase `Person`. Esto quiere decir que las cuatro clases sobrescriben (es decir, modifican el comportamiento) el método `cut` que heredan de `Person`. Obviamente, un carnicero hace la acción de cortar (`cut`) de manera muy distinta a como la hace un cirujano y un peluquero, y ya no digamos un actor que nada tiene que ver su «cortar» (el famoso «corten» que gritan los directores de un film a los actores) con dividir cosas en trozos.

Por otro lado, cabe destacar que gracias a la herencia reducimos la redundancia de código. Imagina, si no, tener que escribir exactamente el mismo código para el método `eat` tanto en la clase `Butcher` como en `Surgeon`, `Actor` y `Hairdresser`. Al mismo tiempo, especializamos o concretamos el comportamiento de aquellos métodos heredados que tienen particularidades según la subclase y, además, añadimos nuevos miembros cuando es necesario.



En el ejemplo anterior hemos indicado la sobrescritura en el diagrama de clases UML repitiendo el método sobrescrito en la subclase que lo sobrescribe. Así pues, si no aparece el método heredado en la subclase, se entiende que se comporta de igual manera que en la superclase. No obstante, la especificación 2.5.1 de UML indica que los miembros heredados (entendidos como aquellos a los que se puede acceder directamente en la subclase) por una subclase (en el caso de los métodos es independiente si se sobrescriben o no) se pueden indicar anteponiendo el símbolo «^» (*caret*). Así pues, si queremos hacer énfasis en los métodos sobrescritos, podemos usar el símbolo «^», aunque su uso es muy poco frecuente.

Por último, cabe decir que muchos lenguajes de programación permiten llamar desde los métodos de las subclases a cualquier método de la superclase que sea visible desde la subclase con el fin de reaprovechar al máximo el código ya escrito. Así, si un método que sobrescribimos en la subclase debe hacer lo mismo que el de la superclase y «algo más», entonces solo tenemos que codificar ese «algo más». En Java, TypeScript, Kotlin y Scala se usa la *keyword* `super`, en C# se usa `base` y en C++ se utiliza `::` entre el nombre de la superclase y el método de esta que hay que llamar, por ejemplo, `SuperClass::doSomething()`. Además, muchos lenguajes (como, por ejemplo, Java o TypeScript) solo permiten acceder a la superclase más inmediata (recordemos que puede existir transitividad), otros en cambio sí que permiten escalar más en la jerarquía, por ejemplo, C++. Veamos un ejemplo de uso de `super` en Java:

```
//Código Java
public void doSomething(){ //Método sobrescrito
     super.doSomething(); //Llamada al mismo método, pero de la superclase
     age = 36;
}
```

2. Clases y métodos abstractos

El concepto *abstracto* está íntimamente relacionado con la herencia. En el paradigma de la programación orientada a objetos, tanto las clases como los métodos pueden ser abstractos.

2.1. Clase abstracta

Empecemos definiendo qué es una clase abstracta.

Definición de clase abstracta

Una clase abstracta se define como aquella que no puede ser instanciada. Es decir, no podemos crear objetos a partir de dicha clase.

Una de las utilidades de definir una clase abstracta es la de actuar como superclase para unificar atributos y métodos comunes de sus subclases. Así pues, el principal objetivo de una clase abstracta es ser heredada y dar una codificación común por defecto a todas las subclases que heredan de ella. Por este motivo, una clase abstracta se debe entender como una abstracción de una funcionalidad, comportamiento o lógica común, más que la abstracción de un conjunto de entidades (objetos) concretos.

En general, una clase abstracta declara la existencia de todos sus métodos, pero no los codifica todos. Los métodos que no codifica son declarados como métodos abstractos (lo veremos a continuación).

Una clase abstracta puede ser heredada de igual manera que una clase «normal».

Para definir una clase abstracta la mayoría de lenguajes proveen al programador con la palabra reservada `abstract`, la cual debe ser escrita en la definición de la clase.

```
abstract class A{
    //TODO
}
```

Clases y métodos abstractos en Python

Este lenguaje no proporciona de por sí la posibilidad de crear elementos abstractos. Sin embargo, existe un módulo llamado `abc` que ofrece esta característica. Gracias al módulo `abc`, una clase es abstracta heredando de `ABC` y un método es abstracto usando el decorador `@abstractmethod`.

Sabiendo que una clase abstracta no se puede instanciar, ¿tiene sentido que tenga definido uno o varios constructores? Pues sí, puesto que si una clase hereda de la clase abstracta, su constructor lo primero que tiene que hacer es llamar al constructor de la superclase, que es abstracta.

Finalmente, destacar que cuando dos clases repiten el mismo código (por ejemplo, un método llamado igual y que hace lo mismo), es señal de que debemos hacer algo para evitar dicha duplicidad y facilitar el futuro mantenimiento del programa. Una solución es poner todo el código compartido en una superclase que unifique la lógica de las dos clases, las cuales pasarán a ser subclases de la superclase unificadora. Si esta superclase solo tiene sentido porque hace de elemento común de las diferentes subclases, entonces esta nueva superclase debe ser abstracta para evitar que se creen objetos de este tipo.

2.2. Método abstracto

Una clase abstracta puede o no contener métodos abstractos.

Definición de método abstracto

Un método abstracto es aquel que solo hace constar su firma o signatura, nada más. Es decir, no tiene llaves y, por consiguiente, no contiene código en su cuerpo. Sirve para avisar de que ese método se debe codificar en alguna de las subclases (o subclases de las subclases, etc.) que hereden de la superclase en la que se encuentra dicho método abstracto.

Un método es abstracto si cumple las siguientes cuatro condiciones:

- Su definición contiene la palabra reservada `abstract`.
- No puede ser `private`, puesto que no será visible en las subclases y, por lo tanto, no puede ser sobrescrito.
- No puede ser `static`.
- No puede ser `final` (veremos qué significa más adelante).

Es importante saber que **si una clase contiene al menos un método abstracto**, entonces **la clase debe ser declarada como abstracta**. Esto tiene mucho sentido, porque si no fuera así, podríamos instanciar un objeto de la clase abstracta y si llamamos al método abstracto, el programa no sabría cómo comportarse, puesto que no hemos escrito código para dicho método. Es decir, po-

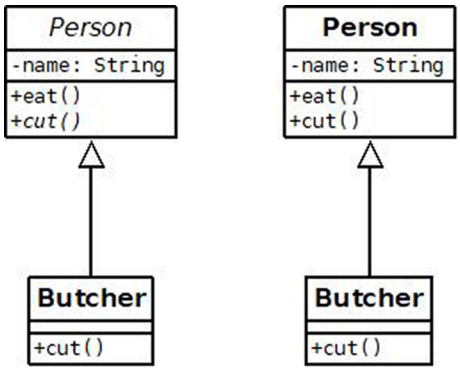
demos ver un método abstracto como un método «inacabado» y, por lo tanto, la clase que la contiene también está inacabada o incompleta y, por ende, debe ser abstracta.

```
abstract class Person{

     protected String name;

     public void eat(){
          print "I am eating…";
     }

     protected abstract void cut();
}

public class Butcher extends Person{

     protected void cut(){
          print "I am a butcher and I prefer chopping.";
     }
}
```

Como podemos ver, un método abstracto es muy útil si no tiene sentido dar una codificación por defecto o común en la clase abstracta y, al mismo tiempo, queremos asegurarnos de que cada subclase proporciona su propia codificación de ese método. En el ejemplo anterior, los objetos de las cuatro clases (recordemos: `Butcher`, `Surgeon`, `Actor` y `Hairdresser`) comen de la misma manera, pero no cortan igual. De esta forma, gracias a la herencia y al uso de un método abstracto, reaprovechamos en las cuatro clases el comportamiento común definido en el método `eat` de la superclase `Person;` y, a su vez, permitimos que cada subclase especialice o concrete el comportamiento o código del método `cut`. Obviamente, el uso de métodos abstractos se puede combinar con la sobrescritura de métodos para diseñar nuestro programa según nos interese.

Ejemplo 7 (método abstracto frente a método sobrescrito) - Encontrar la diferencia

Dados los siguientes dos diagramas de clases:



¿Puedes ver la diferencia entre ellos? En el diagrama de la izquierda el método `cut` (escrito en cursiva) es abstracto en la superclase (cuyo nombre también está escrito en cursiva), es decir, no tiene código en la clase `Person` y es en la subclase `Butcher` donde se codifica. Sin embargo, en el diagrama de la derecha, la clase `Person` proporciona un código para el método `cut` (el cual no es abstracto porque no está en cursiva), y este es sobrescrito en la subclase `Butcher`. Asimismo, en el diagrama de la izquierda no se pueden crear objetos

de la clase `Person` porque es una clase abstracta (está escrito en cursiva), mientras que en el de la derecha sí.

2.3. Representación en un diagrama de clases UML

Como has podido ver en el apartado anterior, en un diagrama de clases UML representamos una clase o un método abstracto escribiendo el nombre de la clase o el nombre del método en cursiva. Una vez que el método abstracto deja de serlo porque se le ha proporcionado código, entonces ya no se escribe en cursiva. Si en el ejemplo anterior la clase `Butcher` del diagrama de la izquierda no hubiera codificado o sobrescrito el método `cut`, entonces tanto `Butcher` como `cut` deberían estar en cursiva.

3. Clases, métodos y atributos finales o sellados

El concepto *final* o *sellado* para las clases y métodos está íntimamente relacionado con la herencia.

3.1. Clase final

En primer lugar diremos qué es una clase final o sellada.

Definición de clase `final`

Una clase `final` es aquella que puede ser instanciada, pero no heredada.

Así pues, no podemos crear subclases a partir de una clase `final`.

En Java, Scala, PHP y C++11 se denomina clase `final`, mientras que en C#, por ejemplo, se llama *clase sellada*. De hecho, debemos usar la *keyword* `final` en Java, PHP y C++11, y `sealed` en C#, delante de la clase para indicar que queremos que se comporte como tal.

```
final class A{
    //TODO
}
```

3.2. Método final

Una clase puede o no contener métodos `final`.

Definición de método `final`

Un método `final` es aquel que no puede ser sobrescrito por una subclase.

Un método es `final` si cumple las siguientes dos condiciones:

- Su definición contiene la palabra reservada `final` (o la equivalente del lenguaje). En C#, debido a que los métodos por defecto no son `virtual`, solo podemos indicar que un método es sellado a partir de la primera subclase, por lo que debemos escribir `sealed override`.

- No puede ser abstracto.

Un método `final` permite establecer un punto a partir del cual garantizamos que no se pueden producir más cambios en el comportamiento de dicho método por parte de ninguna subclase.

`final` en Kotlin

En Kotlin todas las clases y métodos son `final` por defecto. Asimismo, existen las *sealed classes*, que nada tienen que ver con el `sealed` de C#. En Kotlin está relacionado con los `enumeration`.

3.3. Atributo final

Aprovechamos que estamos hablando de `final` para explicar el concepto de atributo `final`:

Definición de atributo `final`

Es aquel atributo cuyo valor solo puede ser asignado una vez. Sería como una constante.

`final` en Python

En Python el concepto de `final` no existe propiamente dicho.

Los lenguajes de programación definen este concepto de maneras muy diversas. Por ejemplo, Java, Scala y PHP utilizan la keyword `final` en la declaración del atributo. En C# lo más parecido sería usar el modificador `readonly` en la declaración del atributo. Sin embargo, en Python no existe una *keyword* equivalente a este concepto.

3.4. Representación en un diagrama de clases UML

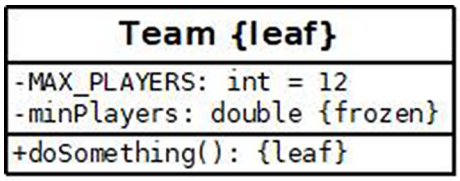
La manera de representar en un diagrama de clases un elemento `final` depende del elemento en sí:

- **Una clase `final`:** se representa escribiendo la propiedad *{leaf}* debajo o cerca del nombre de la clase. De este modo se está diciendo que la clase es la hoja (*leaf,* en inglés) de un árbol o, dicho de otro modo, la última clase de una jerarquía. Por lo tanto, no se puede heredar de ella. Que una clase no pueda ser heredada es lo mismo que decir que esa clase es `final`.

- **Método `final`:** no existe un estándar, ya que UML modela un programa a más alto nivel y no recoge todas las características que proporcionan los lenguajes de programación. Así pues, tenemos cierta flexibilidad para indicar las semánticas que no están recogidas en el estándar. Por este motivo, una manera de indicar que un método es final sería poniendo la propiedad *{leaf}* en el lado derecho del método.

- **Atributo `final`:** suele ser una práctica muy extendida distinguir un atributo `final` del resto de atributos escribiéndolo siguiendo la convención de nombres de las constantes usada en la mayoría de los lenguajes de programación: todo escrito en mayúsculas y los espacios representados por guiones bajos (*underscore*). También podemos encontrar el uso de las propiedades *{readOnly}*, *{frozen}* o *{const}* en el lado derecho del atributo. No obstante, recomendamos usar siempre la convención de nombres de las

constantes. Eso sí, el uso de las propiedades se puede combinar con la convención de nombres de las constantes.



4. Interfaz

Una interfaz es un elemento muy utilizado, pero que no todos los lenguajes de programación orientados a objetos lo soportan, por ejemplo, C++ y Python. También hay lenguajes que, con su propia evolución, han ido modificando el concepto original de interfaz para dotarla de más potencial. Este es el caso, por ejemplo, de Java.

4.1. Concepto original

La definición más esencial de *interfaz* sería:

Definición de interfaz

Es una plantilla o esqueleto que declara, pero no codifica, un conjunto de métodos que las clases que utilizan la interfaz deben codificar. Es decir, la interfaz declara los métodos (concretamente las firmas o signaturas), pero delega su codificación en las clases que la usan.

Las interfaces funcionan como contratos, es decir, cuando una clase utiliza una interfaz, se está comprometiendo a codificar las funcionalidades indicadas en la interfaz. Así pues, la interfaz es un contrato que dice qué hacen las clases, pero no especifica cómo lo hacen.

En algunos libros, webs, apuntes, etc., quizás leas que una interfaz es como una clase 100 % abstracta, dado que no suministra el código de ninguno de los métodos declarados. Evidentemente, una clase abstracta y una interfaz, aunque se parecen, no son lo mismo. Algunas de las diferencias más significativas son las siguientes:

- Una clase no hereda una interfaz, sino que la implementa. Así pues, diremos que la clase `A` implementa la interfaz `B`.

- Sin embargo, una interfaz sí que puede heredar otra interfaz, pero nunca implementarla. Esto es sencillo de entender, puesto que si una interfaz `A` implementara otra interfaz `B`, la interfaz `A`, por el mero hecho de ser interfaz, no podría codificar los métodos de la interfaz `B`.

- Una interfaz no puede heredar de una clase, ya que estaría heredando código de la superclase, y ya hemos dicho que una interfaz no proporciona el código de sus métodos.

- La mayoría de lenguajes que soportan interfaces, pero no soportan la herencia múltiple, sí que permiten que una clase implemente más de una interfaz. Este es el caso, por ejemplo, de Java o C#. De este modo, «simulan» la herencia múltiple gracias a la implementación de múltiples interfaces.

- De igual modo que las clases abstractas, las interfaces no pueden ser instanciadas.

- A diferencia de las clases abstractas, las interfaces no tienen constructores ni destructores.

- Todos los miembros de una interfaz deben ser `public`. De hecho, en algunos lenguajes no es necesario (o incluso está prohibido) indicar el modificador de acceso `public` porque es el único modificador de acceso válido.

- Conceptualmente, una interfaz no puede contener atributos, ni siquiera estáticos. No obstante, algunos lenguajes permiten saltarse esta restricción, aunque con limitaciones.

El nombre de las interfaces sigue la misma nomenclatura que la de las clases. No obstante, es muy frecuente adjetivar el nombre de la interfaz con el sufijo inglés «`-able`», por ejemplo, `Runnable`, `Comparable`, etc.

4.2. Traduciendo a código

Por lo general, una interfaz se declara con la *keyword* `interface` (por ejemplo, Java, C#, Kotlin o TypeScript). Dependiendo del lenguaje, la clase que la implementa usa una u otra sintaxis. Por ejemplo, en Java se usa la *keyword* `implements` y en C# se usan los dos puntos (`:`).

```java
//Código Java
public interface Movable{
     void moveUp();
     void moveDown();
     void moveLeft();
     void moveRight();
}
```

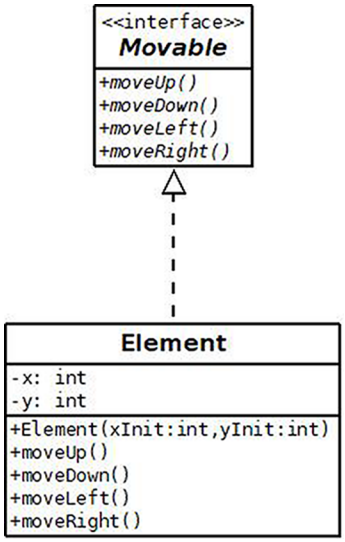
```java
//Código Java
public class Element implements Movable{
     private int x, y;

     public Element(int xInit, int yInit){
          x = xInit;
          y = yInit;
     }

     public void moveUp(){ y++;}
     public void moveDown(){ y--;}
     public void moveLeft(){ x--;}
     public void moveRight(){ x++;}

}
```

4.3. Representación en un diagrama de clases UML

En un diagrama de clases UML también podemos definir interfaces. Estas se representan con una caja con dos compartimentos:

a) Nombre de la interfaz: el compartimento superior contiene el nombre de la interfaz. Este se escribe en negrita y en cursiva (como si fuera una clase abstracta). Además, el nombre va acompañado del estereotipo `<<interface>>`.

b) Métodos: el último compartimento contiene los métodos de la interfaz escritos en cursiva (son, por definición de interfaz, abstractos).



Como podemos ver, se dibuja un triángulo blanco en la interfaz y de él salen líneas discontinuas hacia las clases que la implementan.

4.4. ¿Cuándo usar una interfaz?

Es posible que llegados a este punto uno se pregunte cuándo usar una interfaz y cuándo una clase abstracta, dos elementos que son muy similares, pero no iguales. Esta pregunta se la hace mucha gente, por lo que se convierte en el centro de muchos debates. Vamos a intentar dar algunas directrices, aunque la experiencia y el problema o contexto en el que nos encontremos serán determinantes para la decisión que tomemos.

Se escoge una clase abstracta (y herencia) si se puede decir que «`B` es un tipo de la clase `A`» (`A` será una clase abstracta y `B` una clase que heredará de `A`). En cambio, se usa una interfaz si la relación casa mejor con la frase «`B` es capaz de hacer `A`» (`A` será un interfaz y `B` una clase que la implementará).

Así pues, la clase abstracta se usa para representar lo que los objetos de esa clase son (su identidad), mientras que la interfaz se utiliza para representar lo que los objetos de las clases que la implementen son capaces de hacer (habilidades). Por ejemplo, podemos decir que «un chihuahua es un tipo de perro», pero no tiene sentido decir que «un chihuahua es capaz de ser un perro», por lo tanto, definiríamos `Dog` ('perro') como una clase abstracta. En cambio, podemos decir que las personas, los animales y los vehículos son capaces de moverse o de compararse, pero no son ni un tipo de movimiento ni de compara-

ción. En este caso, las tres clases mencionadas –`Person`, `Animal` y `Vehicle`– implementarán las interfaces `Movable` y `Comparable`. De este modo, los objetos de tres clases *a priori* inconexas como son `Person`, `Animal` y `Vehicle`, estarían relacionados entre sí gracias a que las tres clases a las que pertenecen los objetos se relacionan entre sí mediante la interfaz que implementan, no de unas superclases que heredan. Por consiguiente, **con la interfaz hemos forzado una relación entre clases que no están relacionadas mediante una jerarquía de herencia**.

A veces la restricción de solo poder heredar de manera simple que imponen muchos lenguajes de programación pesará en la decisión, ya que en todos los lenguajes la implementación múltiple de interfaces está permitida.

Finalmente, mostramos una tabla comparativa que puede ayudar a ver las diferencias (y similitudes) ente interfaz y clase abstracta (algunos lenguajes pueden tener matices en algún aspecto indicado en la tabla).

Característica	Interfaz	Clase abstracta
Mecanismo de uso por parte de las clases	Implementación (puede ser múltiple).	Herencia (múltiple, según lenguaje).
Modificador de acceso de los miembros	Los miembros de una interfaz no tienen modificador de acceso, se asume que son public.	Las clases abstractas pueden definir diferentes modificadores de acceso para sus miembros.
Atributos	No puede tener (o sí, pero con restricciones).	Sí puede tener.
Métodos	Todos abstractos	Concretos (con código) y abstractos (sin código)
Funcionalidad	Es 100 % abstracta, por lo que no proporciona código.	Puede tener código que define un comportamiento por defecto para los métodos no abstractos, los cuales pueden ser sobrescritos en las subclases.
Keyword abstract	El uso de este *keyword* es opcional para declarar los métodos (se asume que lo son).	Es obligatorio usar esta *keyword* para declarar un método como abstracto.
Modificación de un método (p.ej., cambio de nombre, en los parámetros, tipo de retorno, se añade un método, etc.)	Se deben revisar todas las clases que la implementan para codificar las novedades del método, si no, tendremos errores de compilación.	Una modificación en la clase abstracta puede no afectar a las subclases si la clase abstracta proporciona el código del método no abstracto y este no se llama en las subclases.
Herencia	No puede heredar de ninguna clase, pero sí de interfaces (herencia simple o múltiple, según el lenguaje).	Puede heredar de superclases (herencia simple o múltiple, según el lenguaje).
Implementación	No puede implementar ninguna interfaz.	Puede implementar múltiples interfaces.
Constructor o destructor	No puede tener.	Puede tener.

Característica	Interfaz	Clase abstracta
Instanciación	No se puede instanciar un objeto a partir de una interfaz.	No se puede instanciar un objeto a partir de una clase abstracta.
Reutilización de código	No facilita la reutilización.	Permite definir código para los métodos no abstractos, lo que favorece su reutilización por parte de las subclases.
Uso	Define capacidades periféricas de una clase.	Define identidad de una clase.

5. Polimorfismo

El mecanismo de polimorfismo es uno de los pilares de la programación orientada a objetos (POO) y está íntimamente relacionado con la herencia.

5.1. Concepto

Podemos definir el mecanismo de polimorfismo de la siguiente manera:

Definición de polimorfismo

Característica de la POO que permite modificar el tipo de un objeto en tiempo de ejecución basándose en una jerarquía de herencia.

De una manera más simple y resumida, podemos decir:

El polimorfismo es un mecanismo que permite que una variable o referencia de un clase se comporte como un objeto de cualquiera de sus subclases (o subclases de las subclase, etc.).

Recordemos que una subclase posee todos los atributos y métodos de su superclase (porque los hereda). Así pues, esto significa que una subclase puede hacer cualquier cosa que su superclase sea capaz de hacer. Podríamos decir, si no hay sobrescritura, que en cualquier parte de nuestro programa donde necesitemos un objeto de la superclase, este podría ser sustituido por un objeto de una de sus subclases y el código funcionaría igual. Es más, a una referencia (variable) del tipo de la superclase se le podría asignar un objeto de cualquiera de sus subclases y todo funcionaría correctamente porque la subclase no deja de ser la superclase extendida (con más atributos o métodos o modificaciones en el comportamiento de los métodos). Por lo tanto, cualquier objeto de una subclase puede ser asignada a cualquier referencia (variable) de sus superclases (cualquier clase que esté por encima en la jerarquía de herencia).

Ahora piensa que una subclase (`Actor`) sobrescribe un método llamado `cut()` de su superclase (`Person`). Si tenemos el siguiente código:

```
Person carlos = new Actor("Carlos","González");
carlos.cut();
```

¿Qué método se llama?, ¿el de la superclase o el de la subclase? En este caso se llama el método `cut` de la subclase, puesto que a la variable `carlos` le hemos asignado un objeto de tipo `Actor` (subclase). Este es el caso más común de polimorfismo, es decir, cuando se usa una referencia (variable) de una superclase para referenciar un objeto de una subclase. Este caso es conocido como **polimorfismo dinámico (o simplemente *polimorfismo*)**, aunque también recibe los nombres de *run-time polymorphism*, *dynamic binding* o *late binding*. Como puedes ver, los términos incluyen los conceptos de «tiempo de ejecución» o «enlazado tardío». ¿Por qué? Porque como la creación del objeto que se le asigna a la referencia o variable se realiza en tiempo de ejecución, entonces la decisión de qué método se llama solo se puede tomar cuando se ejecuta el programa, no cuando este se compila.

En este punto es importante entender dos conceptos: tipo estático y tipo dinámico.

- **Tipo estático (*static type*):** es el tipo definido en la declaración de una variable. Este tipo puede ser una clase concreta (i.e. normal), una clase abstracta o una interfaz.

- **Tipo dinámico (*dynamic type*):** es el tipo del objeto al que se refiere o apunta una variable o, dicho de otro modo, es el tipo del objeto que se asigna a una variable.

Así pues, el polimorfismo dinámico se da cuando el tipo dinámico de una variable o referencia determina el método que se llama. Es decir:

```
Person carlos= new Actor("Carlos","González");
```

Tipo estático — Tipo dinámico

En este ejemplo, `carlos` tiene como tipo estático `Person` y como tipo dinámico `Actor`. El tipo estático de una variable o referencia define qué se puede llamar con esa variable o referencia y qué no.

- **Métodos de la superclase (tipo estático):** con el objeto `carlos` podremos llamar a todos los métodos definidos en la clase `Person`, porque la subclase `Actor` posee todos los métodos de la superclase `Person`. Si el método está sobrescrito en la subclase, entonces se llama al método *sobrescrito*.

- **Métodos exclusivos de la subclase (tipo dinámico):** con el objeto `carlos` no podremos llamar a métodos que sean exclusivos de la subclase `Actor`, puesto que `carlos` es variable de tipo `Person` y no sabe nada de los métodos de `Actor`.

En resumen:

- Un objeto de la subclase puede ser asignado a una variable o referencia declarada con el tipo de una superclase.

- Una variable o referencia declarada como superclase (tipo estático) y asignada a un objeto subclase (tipo dinámico) puede llamar a los métodos definidos en la superclase, pero si están sobrescritos en la subclase se llamarán a las versiones sobrescritas de dichos métodos.

- Una variable o referencia declarada como superclase (tipo estático) y asignada a un objeto subclase (tipo dinámico) no puede llamar a los métodos definidos exclusivamente en la subclase.

Como anécdota, decir que existe el término, menos utilizado, de **polimorfismo estático** (*static binding,* en inglés), que no es otro que aquel que se decide en tiempo de compilación. ¿Y cuándo ocurre? Pues sucede cuando se sobrecarga un método. Como en la sobrecarga lo que cambia es el número o tipo de argumentos (el tipo de retorno también puede variar, pero no es suficiente para hacer sobrecarga), entonces el compilador puede saber a qué versión del método llamar gracias al número y tipo de parámetros pasados en la llamada. Así pues, podemos decir que polimorfismo estático y sobrecarga de un método son sinónimos.

5.2. Ejemplo de uso

Uno de los aspectos clave del polimorfismo es que permite separar las funcionalidades o habilidades de los objetos de su codificación, es decir, separar la interfaz (firmas de los métodos) de la implementación para poder programar basándonos en la interfaz, no en la codificación. Esto es muy útil en programas complejos. Veamos un ejemplo a partir de un caso ya mencionado en estos apuntes: la manera de cortar de las personas.

Ejemplo 8 (polimorfismo) – Una lista de personas cada una con sus particularidades

Como vimos, un carnicero no corta igual que un cirujano, que un actor o que un peluquero. Así pues, la superclase `Person` define una manera de cortar genérica para cualquier persona y las diferentes subclases sobrescriben el método `cut` para proporcionar su codificación o comportamiento específico.

```
class Person{ //Según el contexto, Person podría haber sido abstracta
     //TODO: fields and methods
     protected void cut(){
          print "I do not know how to cut!!";
     }
}

class Butcher{
     //TODO: fields and methods
     public void cut(){
          print "I am a butcher and I prefer chopping";
     }
}

class Surgeon{
     //TODO: fields and methods
     public void cut(){
          print "I am a surgeon and I cut carefully";
     }
}

class Actor{
     //TODO: fields and methods
     public void cut(){
          print "I am an actor and when cutting, I shut up";
     }
}
class Hairdresser{
     //TODO: fields and methods
     public void cut(){
          print "I am a hairdresser and I only cut hair and beard";
     }
}
```

Ahora mira el siguiente código:

```
Person[] people = new Person[5];
people[0] = new Butcher("Jose", "Rodríguez");
people[1] = new Surgeon("Javier" "Vázquez");
people[2] = new Actor("Carlos", "González");
people[3] = new Hairdresser("David", "García");
people[4] = new Person("Elena", "Lázaro", new Date(1978,11,8));

people[0].cut(); //"I am a butcher and I prefer chopping"
people[4].cut(); //"I do not know how to cut!!";

for(int i = 0; i<5; i++){
     people[i].cut(); //llama a cada codificación de cut en el
//siguiente orden: Butcher, Surgeon, Actor, Hairdresser y Person
}

people[4].walk(); //llama a walk de Person
people[0].walk(); //llama a walk de Butcher que es igual al de Person
//porque no lo ha sobrescrito

people[0].grind(); //error: método exclusivo de Butcher que Person
//desconoce… el tipo estático de people[0] es Person y el tipo dinámico
//es Butcher
```

Si te fijas en el ejemplo anterior, gracias a la herencia y al polimorfismo, hemos podido crear un *array* de `Person` y almacenar objetos de cinco clases distintas (porque todos son implícitamente de la clase `Person`): `Butcher`, `Surgeon`, `Actor`, `Hairdresser` y `Person`. Será el tipo dinámico de cada variable o referencia (en este caso, casilla del *array*) quien determine qué versión del método `cut` se debe llamar en cada caso.

Si `Person` hubiera sido una clase abstracta, o incluso una interfaz, todo funcionaría igual, excepto que no podríamos instanciar un objeto de tipo `Person` tal y como estamos haciendo en la casilla 4 del *array* `people`.

5.3. *Casting*: *upcasting* y *downcasting*

El concepto de *casting* –o simplemente *cast*– consiste en hacer una conversión de tipo, por ejemplo, de `int` a `float`, de número a texto, etc. No obstante, este proceso, cuando se hace con objetos, entonces está íntimamente relacionado con el polimorfismo y, por ende, también con la herencia. Hay dos tipos de *casting*: *upcasting* y *downcasting*.

Definición de *upcasting* (o *generalization* o *winderning*)

Consiste en asignar un objeto de una subclase a una referencia/variable cuyo tipo estático es igual al de alguna de sus superclases.

Así pues, podemos hacer:

```
Person carlos= new Actor("Carlos","González"); //upcast
```

Este proceso es siempre seguro, ya que un objeto de tipo subclase posee los miembros de la superclase y puede hacer todo lo que hace su superclase, es decir, un actor es un tipo de persona (`Actor` hereda de `Person`). Como es obvio, no es necesario indicar el *upcasting* de manera explícita:

```
Person carlos = (Person) new Actor("Carlos","González");//OK: explicit upcast
Person carlos = new Actor("Carlos","González"); //OK: implicit upcast
```

El tipo del objeto (en este caso, `Actor`) no cambia debido al *upcasting*, el objeto `Actor` continúa siendo un `Actor`, no un `Person`. Lo único que cambia es la referencia o variable (`carlos`) cuyo tipo estático (superclase `Person`) es de un nivel superior en la jerarquía de herencia al del objeto asignado (subclase `Actor`).

Como ya hemos comentado, ahora con la referencia `carlos` no podremos llamar a los métodos exclusivos de `Actor`, porque no están en la interfaz (i.e. el listado de métodos) de `Person`. Así pues, hacer *upcasting* (moverse hacia arriba en la jerarquía de herencia) conlleva la pérdida de acceso a las características propias de la subclase. ¿Por qué? Porque `carlos` se trata como `Person` aunque el objeto sea (y no deja de serlo) un `Actor`. Así pues, como `carlos` se trata como un `Person`, los miembros de `Actor` quedan escondidos (pero no eliminados, podremos recuperarlos haciendo *downcasting*).

Definición de *downcasting* (o *specialization* o *narrowing*)

Consiste en convertir un objeto con tipo estático igual a una superclase y tipo dinámico igual a una subclase en un objeto cuyo tipo estático sea igual al tipo dinámico.

Así pues, podemos hacer:

```
Person carlos= new Actor("Carlos","González"); //OK: upcast
Actor actor = (Actor) carlos; //OK: downcast
Person elena = new Person("Elena", "Lázaro", new Date(1978,11,8));
Butcher butcher = (Butcher) elena; //KO
Surgeon surgeon = (Surgeon) carlos; //KO
```

El proceso de *downcasting* necesita un *cast* explícito, es decir, indicar entre paréntesis la subclase a la que se debe convertir el objeto. A diferencia del *upcasting*, el *downcasting* no es un proceso seguro, ya que el objeto al que se le aplica el *downcasting* puede no pertenecer a la subclase indicada. Si se produce un error al hacer el *downcasting*, este se producirá en tiempo de ejecución. El *downcasting* nos lleva a descender por la jerarquía de herencia.

Gracias al proceso de *downcasting*, la referencia puede usar los métodos exclusivos de la subclase. Así pues, en el ejemplo anterior sería correcto hacer:

```
    Actor actor = (Actor) carlos; //downcast: trata "carlos" como Actor
    actor.act(); //OK:
    carlos.act();//KO, porque el tipo estático de carlos es Person
    Butcher butcher = (Actor) carlos; //KO, butcher no es padre de
//Actor, sino hermano
    Butcher butcher = (Butcher) carlos; //KO, el tipo dinámico de carlos es
//Actor, no Butcher
```

El proceso de *downcasting* es mucho más frecuente que el de *upcasting*. Usamos *downcasting* cuando queremos acceder a miembros específicos o exclusivos de la subclase.

```
//Código Java
public void doSomething(Person p){
    p.walk();
    p.eat();

    if(p instanceof Actor) {
        //Sólo se ejecuta cuando el tipo dinámico de p es Actor
        Actor actor = (Actor) p; //downcast
        actor.act();
    }
}
```

Saber la clase de un objeto

Muchos lenguajes tienen un comparador que permite saber si un objeto es de un tipo de clase concreta. Por ejemplo, tanto Java como PHP tienen `instanceof`, C# y Kotlin tienen `is`, Python tiene el método `isinstance()`, Scala el método paramétrico `isInstanceOf[Type]`, etc.

5.4. Problemas de rendimiento

Hay que tener en cuenta que el hecho de hacer *casting* (ya sea *upcasting* y sobre todo *downcasting*) supone una serie de verificaciones en tiempo de ejecución que penaliza al rendimiento del programa. Obviamente los diseñadores

de los lenguajes de programación lo saben e intentan minimizar esta pérdida de rendimiento. El tiempo extra que necesita el programa durante la ejecución para hacer *upcasting* y *downcasting* aparece cuando el programa debe decidir qué versión de un método debe ser llamado. Esta verificación ocurre con los métodos declarados como `virtual` (y que posteriormente han sido sobrescritos), pero no con los no virtuales. Con estos últimos el compilador ya sabe qué método debe llamar a partir del tipo con el que se ha declarado la variable o referencia.

En la mayoría de nuestros programas no tendremos mayores problemas de rendimiento debido al uso del polimorfismo (y las ventajas son muchas), pero es importante saber que existe una posible pérdida de rendimiento con su uso. Así pues, en contextos en los que el rendimiento de la aplicación es esencial, debemos saber que evitando el polimorfismo podemos mejorar el rendimiento general del programa. En relación con esto, es importante que en estos contextos tengamos el máximo cuidado a la hora de diseñar las clases y solo definamos, cuando el lenguaje nos los permita (por ejemplo, C++ o C#), como `virtual` aquellos métodos que realmente van a ser sobrescritos. En Java, por ejemplo, todos los métodos son `virtual` por defecto, así que en este caso hay que hacerlos `final`.

6. Consideraciones

Una vez visto qué es la herencia y algunos aspectos relacionados con ella (por ejemplo, polimorfismo, clase abstracta, interfaz, etc.), merece la pena reflexionar en términos de diseño de los programas. Si bien es cierto que el hecho de centralizar en una clase los atributos y métodos comunes a diferentes objetos de diferentes clases es, como hemos visto, algo positivo, también debemos tener en cuenta que puede aumentar la complejidad de nuestro programa.

Cuando diseñamos un programa, debemos buscar un equilibrio entre ser lo más precisos a la hora de modelar el problema o contexto que estamos tratando y reducir la complejidad del uso y mantenimiento de nuestro software. Desgraciadamente no existen unas pautas y la decisión dependerá de varios factores: nuestra experiencia, del problema que hay que resolver, etc.

¿Sabías que hay perros que no ladran? Los perros de la raza basenji no ladran, en su lugar, emiten un sonido o aullido similar al canto tirolés. Imaginemos que queremos modelar esta realidad fielmente. De ser así, separaríamos la clase `Dog` ('perro') en dos subclases: los perros que ladran (`BarkingDog`) y los que aúllan al estilo tirolés (`YodelingDog`). ¿Tiene sentido conceptualmente? Sí. Pero ¿por una raza de perro que no ladra, merece la pena crear dos clases nuevas y una relación de herencia? Seguramente no. Este ejemplo es pequeño y se podría hacer el diseño diferenciador mencionado, pero imagina un programa mucho más grande y complejo: si vas tomando decisiones de este tipo a menudo, al final el programa será inmanejable y difícil de mantener. Así pues, **en un programa grande, mantener las cosas lo más simples posibles es normalmente la mejor práctica**.

> El objetivo principal de un diseño es lograr construir un software que sea lo suficientemente flexible, pero sin añadir demasiada complejidad (o no tanta que haga que el software sea inmanejable).

Finalmente, es vital que cuando diseñemos un programa tengamos en mente posibles usos futuros y ampliaciones del mismo. La idea es dejarse la puerta abierta, de manera que añadir nuevos elementos –ya sean atributos, métodos o incluso clases– sea relativamente sencillo (y poco costoso).

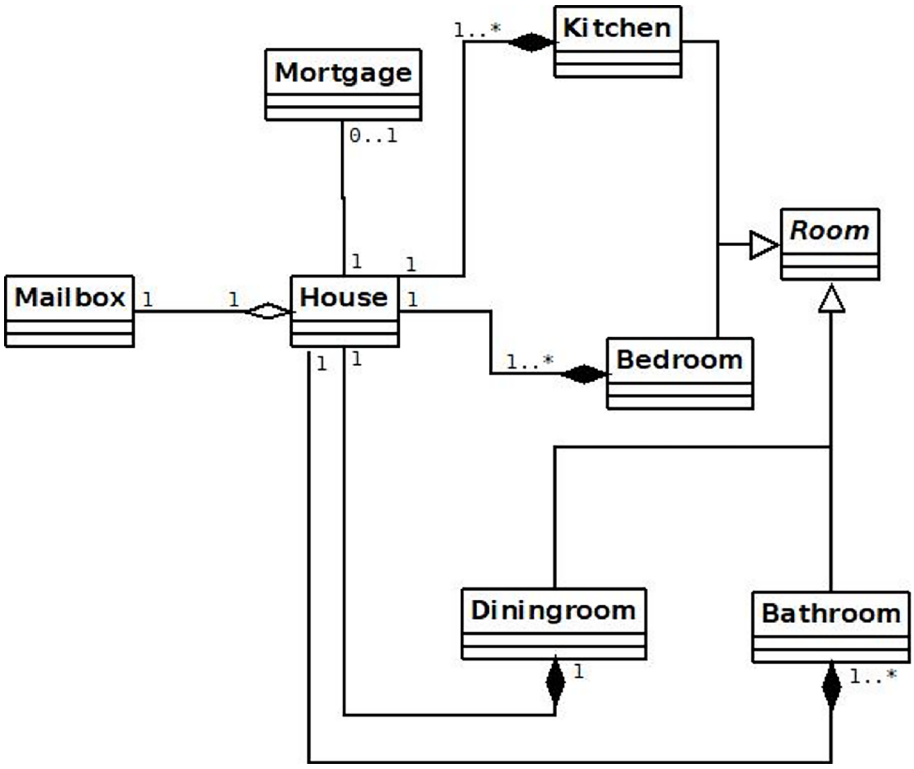
Como podemos ver, la tarea de analizar y diseñar un software es más compleja de lo que parece, puesto que entran en juego muchas variables, entre ellas, la capacidad de abstracción del diseñador.

7. Ejemplo resumen

En este apartado presentamos un ejemplo que incluye gran parte de los conceptos que hemos visto en este módulo y en el anterior. Primero veamos el diagrama de clases UML y después analicémoslo para acabar de entenderlo.

Vídeo de interés

Encontrarás este ejemplo en formato audiovisual en el vídeo «Ejemplo de diagrama de clases UML» que tienes disponible en el aula de la asignatura.



Una casa (`House`), por lo general, está formada por una cocina (`Kitchen`), uno o más dormitorios (`Bedroom`), un comedor (`Diningroom`), uno o más baños (`Bathroom`), etc. Todos ellos son habitaciones (`Room`), de ahí que hereden atributos y métodos de la clase padre `Room` (la cual es abstracta).

Así pues, podemos decir que una casa está formada por habitaciones o, dicho de otro modo, las habitaciones forman o son parte de una casa. Es más, si la casa desaparece (es destruida), entonces las habitaciones desaparecen con ella. No ocurre lo mismo si desaparece un habitación (por ejemplo, porque hemos hecho obras y cambiamos la distribución de la casa). Por este motivo, la relación de la clase `House` con los cuatro tipos de habitación representadas (`Kitchen`, `Bedroom`, `Diningroom` y `Bathroom`) es de tipo asociación de composición.

Por otro lado, toda casa tiene un buzón. Estamos pensando en una casa adosada con jardín típica de las películas norteamericanas. El buzón, aunque forma parte de la casa, puede existir sin que haya casa (por ejemplo, aun no estando

construida la casa, el buzón puede estar ya instalado); asimismo, si quitamos el buzón, no se destruye la casa. Por esta justificación, la relación entre las clases `House` y `Mailbox` es de tipo asociación de agregación.

Por último, tenemos la hipoteca (`Mortgage`). Si bien es cierto que una casa puede tener asociada una hipoteca, esta no forma parte de la casa. Por ello, la relación es una asociación binaria. En este caso, debido a que no tenemos información adicional, hemos decidido que la navegabilidad entre ambas clases sea bidireccional, es decir, la casa sabe de la existencia de la hipoteca y viceversa. Esto podría haber sido diferente si nos hubieran dado más información de contexto. Por ejemplo, uno podría pensar que solo la hipoteca debería saber de la existencia de la casa (la punta de la flecha acabaría en la clase `House`) y que la casa, en sí, no debería saber que está hipotecada.

Resumen

Herencia

- Mecanismo que permite definir una clase nueva (llamada *subclase*) a partir de otra ya existente (llamada *superclase*) describiendo solamente sus diferencias.
- La subclase hereda todos los atributos y métodos de la superclase.
- Por lo general, en la mayoría de los lenguajes no se heredan los constructores, destructores ni constructores estáticos.
- Hay lenguajes, como C++, que permiten definir cómo se hace la herencia: `public`, `protected` o `private`. La mayoría, sin embargo, no permiten indicar la manera de heredar.
- Los miembros `public` o `protected` son heredados por las subclases con el mismo modificador de acceso (excepto en lenguajes donde se puede indicar la manera de heredar) y pueden ser usados o accedidos en la subclase como si se hubieran definido en la propia subclase.
- Los miembros `private` en la superclase son heredados por las subclases, pero estas no tienen acceso directo a dichos miembros. Para acceder a ellos deben usar algún método `public` o `protected` heredado de la superclase.
- Existen dos tipos de herencia: la simple, en la que una subclase solo puede heredar directamente de una superclase; y la múltiple, en la que una subclase puede heredar directamente de más de una superclase.
- La mayoría de lenguajes permiten la transitividad en la jerarquía de herencia.
- La sobrescritura permite que una subclase redefina el comportamiento (i.e. el código) de un método heredado y accesible directamente (i.e. `public` o `protected` en la superclase). Para lograrlo, el método en la superclase debe ser `virtual`.

Vídeo de interés

Puedes ver un resumen de qué es la herencia y los tipos que hay en el vídeo «Relación entre clases: Herencia» que encontrarás en el aula de la asignatura.

Clases y métodos abstractos

- Una clase abstracta es aquella que no puede ser instanciada. Por lo general, declara la existencia de todos sus métodos, pero no los codifica todos.

Aquellos métodos para los que no se proporciona una codificación deben ser declarados como abstractos.

- Si una clase contiene un método abstracto, entonces la clase también debe ser abstracta. El código del método abstracto debe ser proporcionado por las subclases (o subclases de la subclase, etc.) que heredan la superclase abstracta en la que está declarado el método abstracto.

- Una clase abstracta puede ser heredada, pero no instanciada. Sin embargo, puede tener constructor.

- Las clases y métodos abstractos se escriben en cursiva en los diagramas de clase UML.

Clases, métodos y atributos finales

- Una clase `final` (o sellada) es aquella que puede ser instanciada pero no heredada. En un diagrama de clases UML normalmente se escribe la propiedad *{leaf}* debajo o cerca del nombre de la clase.

- Un método `final` es aquel que no puede ser sobrescrito por una subclase. En un diagrama de clases UML normalmente se escribe la propiedad *{leaf}* en el lado derecho del método.

- Un atributo `final` es aquel cuyo valor solo puede ser asignado una vez, es como una constante. En un diagrama de clases UML normalmente se escribe el nombre del atributo final todo en mayúsculas y con los espacios representados por guiones bajos (*underscore*). También se pueden usar las propiedades *{readonly}*, *{frozen}* o *{const}* en el lado derecho del atributo.

Interfaz

- Es una plantilla o esqueleto que declara, pero no codifica, un conjunto de métodos que las clases que utilizan la interfaz deben codificar. Es decir, la interfaz declara los métodos (concretamente las firmas o signaturas), pero delega su codificación en las clases que la usan.

- Las clases no heredan una interfaz, sino que la implementan. En general, una clase puede implementar múltiples interfaces.

- Una interfaz no puede implementar ninguna interfaz, pero sí la puede heredar. Tampoco puede heredar una clase.

- Una interfaz no puede ser instanciada ni tener constructor ni destructor.

- Todos los miembros de una interfaz deben ser `public`. Algunos lenguajes no permiten ni siquiera escribir el modificador de acceso `public` porque es el único aceptado.

- Una interfaz no puede contener atributos, ni siquiera estáticos. Sin embargo, algunos lenguajes se saltan esta restricción siempre y cuando se cumplan unas condiciones.

- El nombre de una interfaz sigue el mismo convenio que el de las clases, aunque es frecuente que estén adjetivados con el sufijo inglés «`-able`», por ejemplo, `Comparable`.

- Las interfaces se representan en un diagrama de clases con el nombre de la interfaz y de los métodos en cursiva. Además, se antepone el estereotipo `«interface»` al nombre de la interfaz. En un diagrama de clases UML se dibuja un triángulo blanco en la interfaz y de él salen líneas discontinuas hacia las clases que la implementan.

- Se escoge una clase abstracta (y herencia) si se puede decir que «`B` es un tipo de la clase `A`» (`A` será una clase abstracta y `B` una clase que heredará de `A`). En cambio, se usa una interfaz si la relación casa mejor con la frase «`B` es capaz de hacer `A`» (`A` será una interfaz y `B` una clase que la implementará).

Polimorfismo

- El polimorfismo es un mecanismo que permite que una variable o referencia de un clase se comporte como un objeto de cualquiera de sus subclases (o subclases de las subclase, etc.).

- Tipo estático: es el tipo definido en la declaración de una variable. Este tipo puede ser una clase concreta (i.e. normal), una clase abstracta o una interfaz.

- Tipo dinámico: es el tipo del objeto al que se refiere o apunta una variable. Dicho de otro modo, es el tipo del objeto que se asigna a una variable.

- Un objeto de la subclase puede ser asignado a una variable o referencia declarada con el tipo de una superclase.

- Una variable o referencia declarada como superclase (tipo estático) y asignada a un objeto subclase (tipo dinámico) puede llamar a los métodos definidos en la superclase, pero si están sobrescritos en la subclase, se llamarán a las versiones sobrescritas de dichos métodos.

- Una variable o referencia declarada como superclase (tipo estático) y asignada a un objeto subclase (tipo dinámico) no puede llamar a los métodos definidos exclusivamente en la subclase.

- El *upcasting* consiste en asignar un objeto de una subclase a una referencia cuyo tipo estático es igual al de alguna de sus superclases. El *upcasting* siempre es seguro.

- El *downcasting* consiste en convertir un objeto con tipo estático igual a una superclase y tipo dinámico igual a una subclase en un objeto cuyo tipo estático sea igual al tipo dinámico. El *downcasting* no siempre es un proceso seguro.

Actividades

Ejercicio 1
Dibujar y explicar el diagrama de clases de una aplicación que permita el control de los asientos de un teatro para las diferentes obras que se representan en él. Para empezar, recordemos que cada obra tiene un título, una sinopsis, el nombre del director, la duración en minutos y la edad a partir de la cual está recomendada. Las edades recomendadas pueden ser `TP` (i. e. todos los públicos), `M7` (i. e. mayores de siete años), `M12` (i. e. mayores de doce años) y `M18` (i. e. mayores de dieciocho años).

Cada obra tiene unos pases en los que dicha obra se representa. Cada pase viene definido por una fecha (día, mes y año; tipo `Date`) y una hora (tipo `Time`). Además, un pase pone a la venta un conjunto de asientos. Para cada pase, se debe guardar si el asiento ya ha sido vendido o no y el precio de venta de dicho asiento.

Cada asiento tiene un número de fila y un número de butaca, así pues, un asiento sería «fila 5, butaca 2», por ejemplo. Además, cada asiento tiene un atributo que indica el sector al que pertenece. Los sectores del teatro son: `PLATEA`, `BALCON`, `ANFITEATRO1` y `ANFITEATRO2`. Así pues, existe más de un asiento «fila 5, butaca 2», pero pertenecen a sectores diferentes. Además de los asientos normales, hay una variedad de asientos, que se llaman VIP, que son iguales que los asientos normales, pero tienen un mayor reposabrazos e incluyen sistema de masaje (mediante un método llamado `darMasaje`). Estos asientos están repartidos por todo el teatro.

Nota: En el diagrama de clases se deben recoger tanto las clases como sus atributos y métodos relevantes (con su nivel de acceso y tipos), así como las relaciones y multiplicidades que existen entre las clases. No se deben incluir métodos *getter* ni *setter*, así como tampoco constructores de clase. Para el resto de métodos se puede considerar que no reciben parámetros ni tampoco devuelven nada. Los elementos abstractos indícalos anteponiendo `<<abstract>>`.

Ejercicio 2
Dibujar y explicar el diagrama de clases de una aplicación que permita gestionar una liga de fútbol. En primer lugar, recordemos que todos los equipos tienen un nombre, un año de fundación y nombre de presidente. Como es obvio, todos los equipos están compuestos por un mínimo de once jugadores hasta un máximo de veintidós futbolistas. Además, tienen un entrenador. De todos ellos queremos saber su identificador (es un número proporcionado por la federación que puede contener ceros por la izquierda, por ejemplo, `001`), su nombre, sus apellidos y la fecha de nacimiento. De los futbolistas además queremos saber su dorsal y la demarcación en la que juegan. Las demarcaciones que existen son `PORTERO`, `DEFENSA`, `CENTROCAMPISTA` y `DELANTERO`. En cuanto a los entrenadores, queremos guardar el año en que se licenciaron.

Tanto los futbolistas como el entrenador viajan y se concentran antes de cada partido. Además, los futbolistas entrenan y juegan, mientras que el entrenador dirige.

Cada equipo juega con otro en un partido y cada partido tiene un equipo local y un equipo visitante. Queremos saber qué partidos disputa un equipo como local y cuáles como visitante. Además, cada partido se disputa en una fecha concreta y, al final del mismo, tiene un resultado.

La liga tiene árbitros que se encargan de arbitrar los partidos que se les asigna. De los árbitros, guardamos su identificador (es un número proporcionado por la federación que puede contener ceros por la izquierda, por ejemplo, `001`), su nombre, sus apellidos y la fecha de nacimiento. Por cuestiones de estrés, un árbitro no puede arbitrar más de quince partidos. Evidentemente, un partido solo tiene cuatro árbitros. El diagrama, y por consiguiente el programa, no debe diferenciar entre árbitro principal, linier y cuarto árbitro.

Nota: En el diagrama de clases se deben recoger tanto las clases como sus atributos y métodos relevantes (con su nivel de acceso y tipos), así como las relaciones y multiplicidades que existen entre las clases. No se deben incluir métodos *getter* ni *setter*, así como tampoco constructores de clase. Para el resto de métodos se puede considerar que no reciben parámetros ni tampoco devuelven nada. Los elementos abstractos indícalos anteponiendo `<<abstract>>`.

Ejercicio 3
Dibujar y explicar el diagrama de clases de una aplicación que permita el control de vuelos de un aeropuerto. En dicha aplicación guardaremos información sobre las compañías aéreas que operan en el aeropuerto que gestionamos. Cada compañía aérea tiene un nombre (por ejemplo, Iberia) y un identificador (por ejemplo, IB para Iberia), así como un conjunto de vuelos que operan. Si borramos una compañía del sistema, borraremos todos sus vuelos.

Un vuelo se encuentra identificado por un código, una compañía y se realiza desde un aeropuerto origen a un aeropuerto destino en una determinada fecha. De cada aeropuerto quere-

mos guardar su nombre (por ejemplo, Madrid-Barajas) y su identificador (por ejemplo, MAD). Asimismo, todo vuelo tiene asignado un solo avión, que está identificado por un código alfanumérico y tiene unas plazas determinadas. Cada avión puede tener asignados varios vuelos. Desde la aplicación queremos saber tanto el avión asignado a un vuelo concreto como los vuelos asignados a un avión determinado.

Para operar un vuelo es necesario el trabajo de diferentes personas. Estas personas pueden ser, o bien personal del aeropuerto, o bien personal de una compañía aérea. Tanto el personal del aeropuerto como el de las compañías están identificados por un número personal que coincide con su DNI (con letra) y un sueldo base.

Entre el personal necesario para operar un vuelo, están los controladores aéreos. Estos empleados, que son personal del aeropuerto, son los encargados de controlar los vuelos del aeropuerto. Dentro del control de los vuelos está la autorización del despegue o del aterrizaje de un vuelo (depende de si el vuelo parte de ese aeropuerto o llega). Por razones de estrés, un controlador solo puede encargarse de, como máximo, diez vuelos y, obviamente, un controlador solo gestiona un vuelo. Además, como su trabajo es complejo, tienen un plus de responsabilidad que será un porcentaje de su sueldo.

Por último está el personal de embarque. Estas personas pertenecen a la compañía que opera el vuelo y para este personal se debe guardar a qué compañía pertenece. Estas personas facturan y realizan el embarque de los vuelos de la compañía. Cada una de estas personas pueden ser asignadas a uno o más vuelos, y la facturación y embarque de un vuelo puede ser gestionado por una o más personas de la compañía. Si se elimina una compañía del sistema porque ya no opera más en el aeropuerto, todo su personal debe ser eliminado también de la aplicación.

Nota: En el diagrama de clases se deben recoger tanto las clases como sus atributos y métodos relevantes (con su nivel de acceso y tipos), así como las relaciones y multiplicidades que existen entre las clases. No se deben incluir métodos *getter* ni *setter*, así como tampoco constructores de clase. Para el resto de métodos se puede considerar que no reciben parámetros ni tampoco devuelven nada. Los elementos abstractos indícalos anteponiendo `<<abstract>>`.

Ejercicio 4
Dibujar y explicar el diagrama de clases que representa una aplicación que permita a la Consejería de Educación de una comunidad autónoma gestionar las escuelas, profesores y alumnos de educación infantil y primaria. Para empezar, recordemos que las escuelas se definen con un identificador alfanumérico, un nombre, una dirección y un director que las dirige. Además, cada una de ellas puede ser pública, privada o concertada. Las escuelas asignan a los estudiantes en las aulas y deciden quiénes son los tutores de cada aula.

Cada escuela tiene un claustro de profesores que queremos conocer en todo momento. Cada profesor se define por su NIF, nombre, apellidos y año en que se graduó de magisterio. Cada uno de ellos tiene también una especialidad docente: matemáticas, lengua, música o educación física. Un profesor siempre pertenece a una escuela y, en un instante determinado, solo puede estar asignado a una. No interesa saber las escuelas por las que ha pasado cada profesor a lo largo de su carrera docente, en cambio, dado un profesor, sí que queremos saber a qué escuela pertenece en el momento de la consulta. La tarea principal de un profesor es evaluar. Finalmente, hay que decir que uno de los profesores del claustro es el director de la escuela.

Asimismo, las escuelas tienen alumnos. Cada uno de ellos está definido con un identificador formado por nueve dígitos llamado IDALU (IDentificador del ALUmno), su NIF (si tienen, por defecto se entenderá que no tienen), su nombre y sus apellidos. Además, para cada estudiante, la escuela debe saber si tiene necesidades educativas especiales (NEE). Un estudiante puede haber pertenecido a varias escuelas a lo largo de su vida académica y esta es una información que debemos guardar. Es decir, para cada estudiante debemos tener el historial (o expediente) de cursos académicos que indique: fecha del curso académico (formado por un año inicial y uno final, por ejemplo, 2018-2019), en qué escuela estudió durante dicho curso académico, en qué nivel académico (se refiere a: P3 –tres años–, P4, P5, primero –seis años–, segundo, tercero, cuarto, quinto y sexto) y el informe final obtenido –será un texto con la evaluación de las diferentes asignaturas. Si el curso académico está en curso (i. e. es el actual), este texto estará vacío. Si un estudiante desaparece de la aplicación (por ejemplo, fallece, pasa a la ESO, etc.), su historial o expediente también.

Una escuela está formada por aulas. Estas tienen un nombre (por ejemplo, 1-A), un sobrenombre (por ejemplo, Tortugas), una ubicación definida por tres dígitos (por ejemplo, 203, puerta 03 de la planta 2 o 001, puerta 01 de la planta 0), un nivel académico, un tutor (es un profesor) y un conjunto de alumnos. El mínimo de alumnos por aula es diez, con un máximo de veintiséis. Un alumno solo puede pertenecer a un aula en un momento determinado. Dada un aula, queremos saber qué alumnos y tutor tiene. A su vez, dado un alumno,

queremos saber a qué aula va. Dado un profesor, no queremos saber de qué aula es tutor (hay profesores que no son tutores).

Nota: En el diagrama de clases se deben recoger tanto las clases como sus atributos y métodos relevantes (con su nivel de acceso y tipos), así como las relaciones y multiplicidades que existen entre las clases. No se deben incluir métodos *getter* ni *setter*, así como tampoco constructores de clase. Para el resto de métodos se puede considerar que no reciben parámetros ni tampoco devuelven nada. Los elementos abstractos indícalos anteponiendo `<<abstract>>`.

Ejercicio 5
El servicio de cercanías está formado por líneas, por ejemplo, R1 (Molins de Rei-Maçanet), C3 (Aranjuez-Chamartín), etc. Cada línea tiene un código alfanumérico, un nombre, un recorrido (es un texto), una hora de inicio y una hora de finalización del servicio. Además, cada línea es operada como mínimo por un tren. El número máximo de trenes por línea dependerá de la demanda, no hay límites. Queremos saber, dada una línea, qué trenes la operan en ese momento y si una línea desaparece, los trenes que la operan son reubicados en otras líneas o desmontados. Por defecto, un tren no tiene línea asignada.

Para cada tren guardaremos su identificador alfanumérico (por ejemplo: R1-0456, C3-0678, etc.). Un tren está formado por una sola locomotora y un conjunto de vagones. Además, cada tren tiene definido el espacio en centímetros entre vagones. Asimismo, debemos ser capaces de saber la longitud total del tren a partir de la longitud de su locomotora, los vagones y el espacio entre estos. Dado un tren, queremos conocer la línea que opera (evidentemente solo puede operar una línea en un momento determinado).

Tanto la locomotora como los vagones pueden considerarse elementos de un tren con las características comunes de identificador alfanumérico y longitud. Además, la locomotora se define mediante la tipología de propulsión, que puede ser: eléctrica, diésel o magnética. Dada una locomotora, queremos saber a qué tren pertenece (solo puede estar vinculada a un tren) y viceversa.

Por su parte, los vagones se definen mediante su volumen (en metros cúbicos) y el tipo de contenido que pueden transportar, es decir, si es un vagón de pasajeros, refrigerado o de mercancías, por ejemplo, transporta paquetes, cartas, etc. Cada tren tiene entre cinco y quince vagones.

Cada día, si es necesario, se configuran los trenes, por lo tanto, un mismo tren puede un día tener seis vagones y otro ocho. Tanto los vagones como la locomotora que han sido quitados del tren son utilizados para crear otro tren o guardados en la cochera. Evidentemente, un vagón solo puede pertenecer a un tren en un momento determinado, y eso es información que nos interesa conocer. Por el contrario, no nos interesa saber a qué trenes ha pertenecido cada vagón ni cada locomotora a lo largo de su historia. Por defecto, un vagón no pertenecerá a ningún tren y lo mismo ocurrirá con las locomotoras.

El personal que permite el buen funcionamiento del servicio son los revisores y maquinistas. Estos se definen con su nombre, apellidos, NIF y un horario (será un texto). Cada línea puede tener asignados un máximo de tres revisores y queremos saber quiénes son. Estos se encargan de inspeccionar que los pasajeros tengan un billete válido. Un revisor puede tener asignadas varias líneas y queremos saber cuáles.

Cada maquinista tiene asociado un solo tren y se encarga de conducirlo. Si al tren no se le asigna una línea, el maquinista es asignado a un tren con línea. Dado un tren, queremos saber su maquinista y viceversa.

Nota: En el diagrama de clases se deben recoger tanto las clases como sus atributos y métodos relevantes (con su nivel de acceso y tipos), así como las relaciones y multiplicidades que existen entre las clases. No se deben incluir métodos *getter* ni *setter*, así como tampoco constructores de clase. Para el resto de métodos se puede considerar que no reciben parámetros ni tampoco devuelven nada. Los elementos abstractos los indícalos anteponiendo `<<abstract>>`.

Ejercicio 6
Dibujar y explicar el diagrama de clases que representa una aplicación de escritorio que permite la gestión de la guía televisiva de una plataforma de pago. En primer lugar, sabemos que el servidor multimedia envía a la aplicación información sobre los programas de televisión. Cada programa está definido por un título, una descripción, una duración en minutos, una fecha de inicio de emisión, una fecha hasta la cual estará disponible en la plataforma y un código de autorregulación que puede ser TP (i. e. todos los públicos), +7, +12 o +18. Para cada programa debemos proporcionar un mecanismo que informe sobre la fecha de finalización del programa a partir de la fecha de inicio de emisión y la duración del programa.

La interfaz del software debe diferenciar los programas según sean series, películas u otros. Para las películas, además de la información mencionada para los programas, estas tienen información sobre el nombre del director, el reparto (es un texto con los nombres de los actores separados por comas), el año de estreno y el género al que pertenece: acción, comedia, romántica, terror, suspense, oeste u otro.

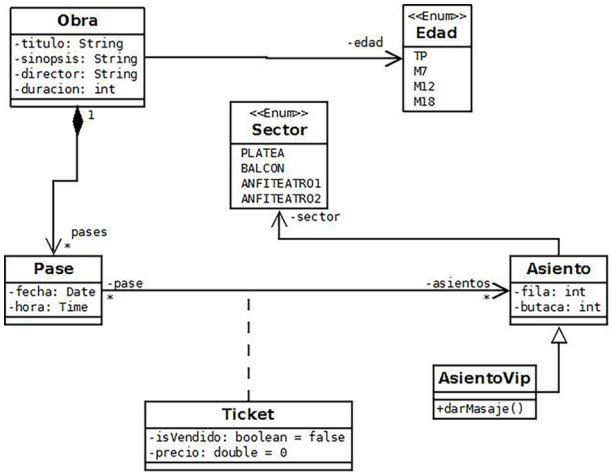
Las series están formadas como mínimo por una temporada y proveen un mecanismo para saber cuántas temporadas tienen. Cada temporada tiene un identificador numérico (por ejemplo, 1 para la temporada 1) y un año. Cada temporada está formada por episodios (como mínimo uno) y también permite saber cuántos episodios hay en ella. Para cada episodio guardaremos el título, el director, su sinopsis y duración en minutos. Además, para cada episodio debemos saber cuál es el episodio previo y el posterior. Si una temporada desaparece, todos sus episodios también. Si una serie desaparece del catálogo, todas sus temporadas también.

La información que llega del servidor al que se conecta el software también organiza los programas por cadenas; es decir, el usuario además de poder acceder por programa, también puede ver qué programas se están emitiendo en las diferentes cadenas en ese momento y a corto plazo. Para ello, el servidor envía información de las cadenas de televisión presentes en la plataforma. En este sentido, cada cadena viene definida por un nombre y un número de canal. Además, junto con la información anterior, se envía también la parrilla televisiva de la cadena desde el momento actual hasta un tiempo definido por la propia cadena. La parrilla televisiva es un listado ordenado de programas de televisión. Como mínimo, esta parrilla está formada por el programa que se está emitiendo en el momento actual. Si una cadena de televisión desaparece, sus programas no lo hacen, puesto que se dejan un tiempo en catálogo, concretamente hasta la fecha de fin de disponibilidad que mencionamos anteriormente. Asimismo, no todos los programas de la plataforma están asociados a un canal. Por ejemplo, la plataforma permite a sus clientes ver películas de pago que se están proyectando en los cines. Para estas películas, además de la información mencionada para las películas, debemos guardar también su precio. Una vez pagada, el cliente la puede ver a continuación, no más tarde. En este sentido no nos interesa guardar las compras realizadas por los clientes.

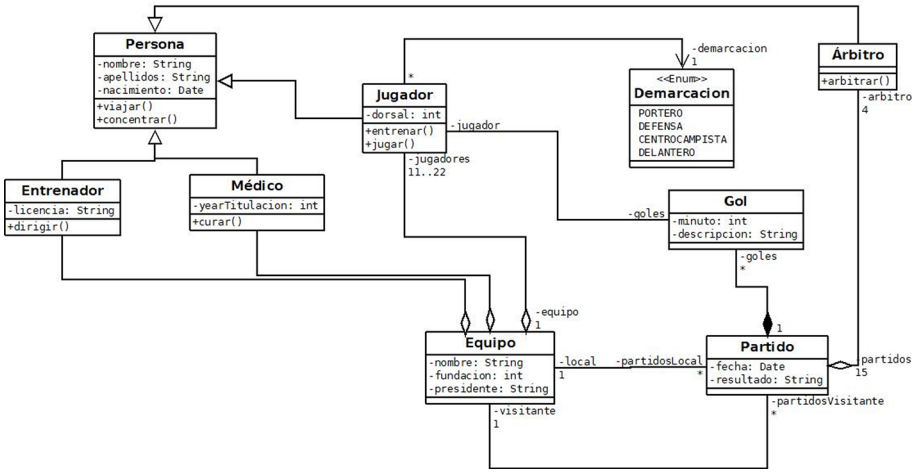
Nota: En el diagrama de clases se deben recoger tanto las clases como sus atributos y métodos relevantes (con su nivel de acceso y tipos), así como las relaciones y multiplicidades que existen entre las clases. No se deben incluir métodos *getter* ni *setter*, así como tampoco constructores de clase. Para el resto de métodos, se puede considerar que no reciben parámetros ni tampoco devuelven nada. Los elementos abstractos indícalos anteponiendo `<<abstract>>`.

Solucionario

Solución ejercicio 1

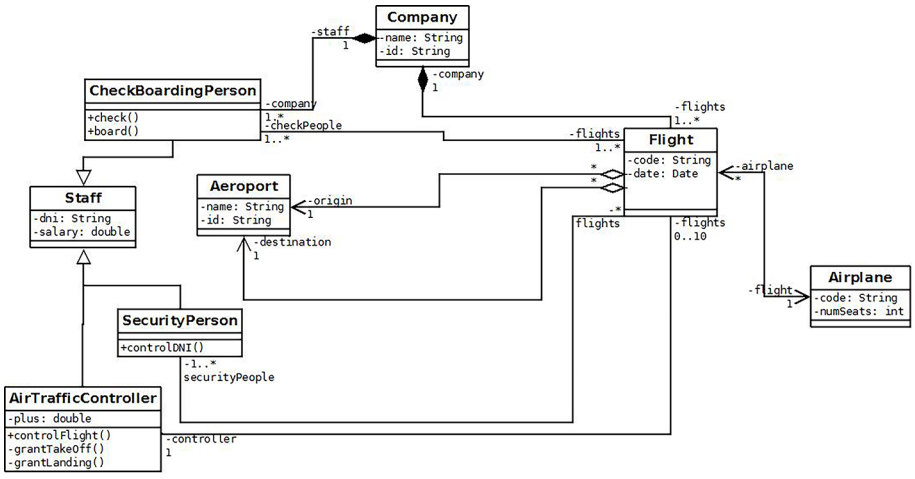


Solución ejercicio 2



La relación entre `Jugador` y `Partido` no puede tener una clase asociativa `Gol`, puesto que las clases asociativas tienen la restricción implícita de que solo puede existir una instancia de la clase asociativa por cada par de objetos de la relación. Así pues, si un jugador marca más de un gol, entonces no se podría representar; pero tal y como está en el diagrama, sí.

Solución ejercicio 3



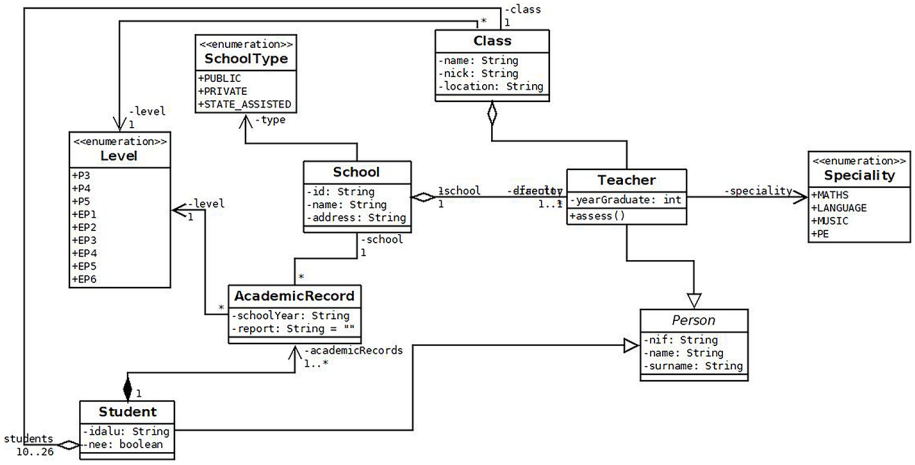
La relación de `Company` con `CheckBoardingPerson` y `Flight` debe ser de composición, porque el enunciado dice que si se elimina la compañía se borran su personal y sus vuelos.

La relación de `Flight` con el resto de elementos puede ser binaria o de agregación, pero no de composición.

Es posible crear dos clases extra, `AirportStaff` y `CompanyStaff`, pero no es necesario. En cualquier caso, debe haber relación de herencia entre las clases *staff* y los casos concretos.

En la clase del controlador aéreo (`AirTrafficController`) hemos considerado privados los métodos de conceder despegue (`grantTakeOff`) y aterrizaje (`grantLanding`).

Solución ejercicio 4



La clase `Persona` es abstracta y padre de `Estudiante` y `Profesor`. El profesor evalúa (es un método) y el atributo `nee` de `Estudiante` es un `boolean`.

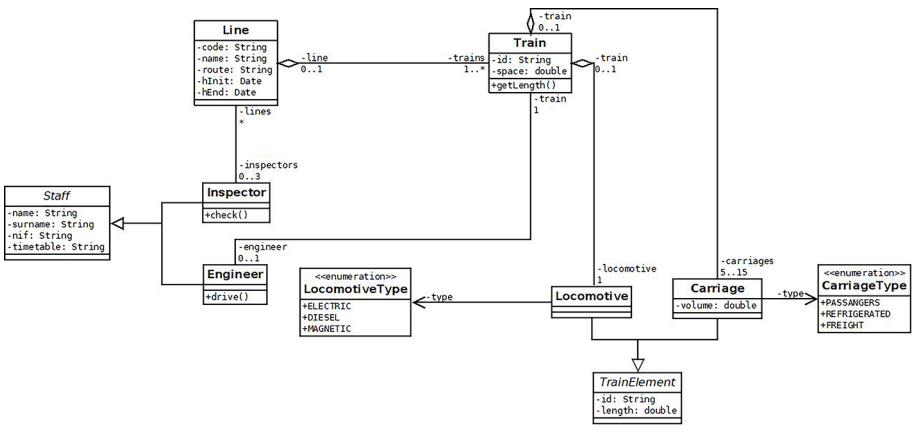
El expediente es una clase intermedia entre `Estudiante` y `Escuela`. La relación `Estudiante-Expediente` es de composición, porque el enunciado dice que si desaparece el `Estudiante`, su expediente también. El valor del informe se debe inicializar a «» (vacío) o a `null`.

Las aulas están formadas por estudiantes. Esta relación es una agregación, porque si se elimina un aula, los estudiantes no son eliminados, sino reubicados. La navegabilidad es bidireccional, según indica el enunciado. Asimismo, las aulas tienen un tutor que es un profesor. La relación `Aula-Profesor` es unidireccional, porque el enunciado deja claro que se quiere saber el tutor del aula, pero, dado un profesor, no se quiere saber si es tutor o no de un aula.

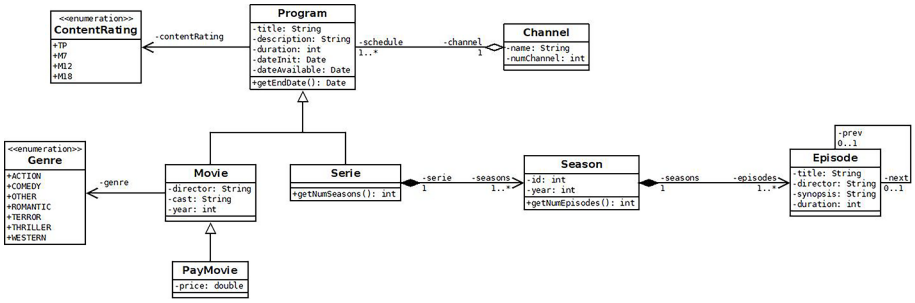
La relación `Escuela-Profesor` para modelar el `Director` es agregación, aunque podría ser binaria. En el enunciado no se especifica la navegabilidad de esta relación.

La relación `Escuela-Profesor` para modelar el claustro es bidireccional. No se quiere saber las escuelas por las que ha pasado un profesor, por lo tanto, no debe haber ninguna clase asociativa ni intermedia que modele esto.

Solución ejercicio 5



Solución ejercicio 6



Bibliografía

Booch, G.; Maksimchuk, R.; Engle, M.; Young, B. J.; Conallen, J.; Houston, K. (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.

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.