Herència (relacions entre classes)

PID_00272785

David García Solórzano

Temps mínim de dedicació recomanat: 4 hores

Universitat Oberta de Catalunya

David García Solórzano

Graduat Superior en Enginyeria en Multimèdia i Enginyer en Informàtica per la Universitat Ramon Llull des de 2007 i 2008, respectivament. És també doctor per la Universitat Oberta de Catalunya des de 2013, en la qual va presentar una tesi doctoral relacionada amb l'àmbit de l'*e-learning*. Des de 2008 és professor de la Universitat Oberta de Catalunya en els Estudis d'Informàtica, Multimèdia i Telecomunicació.

L'encàrrec i la creació d'aquest recurs d'aprenentatge UOC han estat coordinats pel professor: David García Solórzano (2020)

Primera edició: febrer de 2020

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

Índex

Introducció

Les classes, a més de relacionar-se mitjançant la interacció entre els seus objectes, també poden relacionar-se entre elles a nivell de la pròpia classe. La relació entre classes per excel·lència en el paradigma de la programació orientada a objectes (POO) és la generalització / especialització, comunament coneguda com a *herència*. Gràcies a l'herència, podrem utilitzar una classe (anomenada *superclasse*) per a crear noves classes (anomenades *subclasses*) que modifiquin alguns aspectes de la classe original. Així doncs, aprendrem a crear subclasses que heretaran els atributs i mètodes d'una superclasse. A més, veurem com es sobreescriuen mètodes per a fer que les subclasses es comportin amb les particularitats que volem que tinguin. També parlarem dels conceptes d'herència simple i múltiple, així com de transitivitat.

En relació amb l'herència, aprendrem què són una classe abstracta i una classe final. A continuació, presentarem la interfície, un element similar a una classe abstracta, però que té diferències importants que hem de conèixer. Finalment, explicarem un dels pilars de la POO que està íntimament relacionat amb l'herència, el polimorfisme.

A més, veurem com es representen aquestes relacions, tipus de classes i la interfície usant el llenguatge de modelatge de sistemes de *programari* UML (*Unified Modeling Language*).

Tret que s'indiqui un llenguatge de programació concret, els exemples de codificació estan escrits amb un llenguatge de programació inventat, és a dir, un pseudocodi. Si haguéssim de dir a quin llenguatge de programació real s'assembla el pseudocodi que emprem, diríem que és semblant al Java, però sense elements que en dificultin la comprensió dels exemples.

Així doncs, haurem de consultar la documentació del llenguatge de programació que volem utilitzar per a veure com es codifiquen els conceptes explicats i els exemples proporcionats.

Objectius

L'objectiu principal d'aquest mòdul és explicar com les classes es relacionen entre elles dins d'un programa basat en el paradigma de la programació orientada a objectes (POO). Com a conseqüència d'aquest objectiu, n'apareixen uns altres:

1. Conèixer la relació de generalització o especialització (més coneguda com a *herència*) que es pot donar entre classes, com també els termes *superclasse* i *subclasse*.

2. Entendre els tipus d'herència que hi ha i per què l'herència simple és més comuna en els llenguatges de programació que l'herència múltiple.

3. Comprendre la particularitat que té el constructor d'una subclasse.

4. Saber què és una classe abstracta i la seva utilitat.

5. Saber què és una classe final i la seva utilitat.

6. Saber què és una interfície des del punt de vista del paradigma de la POO i la seva utilitat.

7. Veure i entendre el potencial que l'herència, mitjançant diferents mecanismes (per exemple, el polimorfisme), pot oferir-nos a l'hora de dissenyar i implementar els nostres programes.

1. Herència o relació de generalització/especialització

1.1. Concepte

La relació de generalització/especialització també és coneguda com a *herència*, encara que és més freqüent l'ús d'aquest últim terme. Així doncs, a partir d'ara parlarem d'herència. L'herència, juntament amb l'abstracció, l'encapsulació i el polimorfisme (que veurem més endavant), és una de les característiques més importants de la programació orientada a objectes. Quan desenvolupis programes de mitjana o gran envergadura, necessitaràs usar el mecanisme d'herència.

Definició d'herència

La relació entre dues classes en la qual una classe (anomenada *classe pare*, *base* o *superclasse*) generalitza el comportament de l'altra classe (anomenada *classe filla*, *classe derivada* o *subclasse*). O, vist des de l'altre punt de vista, la subclasse especialitza el comportament de la superclasse.

Des d'un punt de vista més pràctic, podem definir l'herència de la manera següent:

Mecanisme de la programació orientada a objectes que permet definir una classe nova (la subclasse) a partir d'una classe que ja existeix (la superclasse) descrivint-ne solament les diferències.

Resumint, en termes generals:

- Una superclasse conté els atributs i mètodes comuns que són heretats per una o més subclasses.

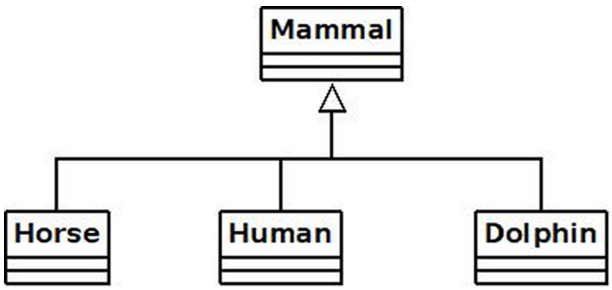
- Una subclasse reutilitza (hereta) els atributs i mètodes de la superclasse sense haver de recodificar-los. Així mateix, la subclasse pot modificar els mètodes heretats mitjançant el mecanisme de sobreescriptura –ho veurem més endavant. A més, pot definir atributs i mètodes nous que són propis de la subclasse.

Així doncs, gràcies a l'herència es reutilitza una gran quantitat de codi, per la qual cosa es redueix o s'elimina la redundància. En aquest punt, cal destacar que, a diferència de les associacions –que modelen la relació entre objectes–, l'herència és una relació exclusivament entre classes. Vegem-ne un exemple:

Exemple 1 (herència) – Mamífers

A la Marina li parlen dels mamífers a la llar d'infants i li expliquen que els mamífers respiren, mengen, etc. Així mateix, li expliquen que un cavall és un mamífer, però que els humans també ho som. A més, encara que no ho semblin, els dofins també ho són. Per tant, a la Marina, el que li estan dient en termes del paradigma de programació orientada a objectes és que les classes `Horse` ('cavall'), `Human` ('humà') i `Dolphin` ('dofí') especialitzen (o hereten) el comportament de la classe `Mammal` ('mamífer'). En aquest cas, la classe `Mammal` és la classe pare (superclasse) de la relació de la qual hereten les classes filles (subclasses): `Horse`, `Human` i `Dolphin`.

En UML, l'herència es representa de la manera següent:



Com podem veure, es dibuixa un triangle blanc en la superclasse (en l'exemple anterior, `Mammal`) i en surten línies cap a les subclasses (`Horse`, `Human` i `Dolphin`). És important adonar-se que la relació d'herència no té multiplicitat, ja que no hi intervenen instàncies o objectes, sinó que simplement diem que una subclasse especialitza les característiques d'una altra classe (i.e. superclasse). Tampoc no existeixen ni la navegabilitat ni el rol.

Es pot considerar l'herència com una relació del tipus `<<és-un>>`; per exemple, un cavall (subclasse) és un tipus de mamífer (superclasse), un humà (subclasse) és un tipus de mamífer (superclasse), un dofí (subclasse) és un tipus de mamífer (superclasse), un cotxe (subclasse) és un tipus de vehicle (superclasse), etc.

1.2. Traduint a codi

La sintaxi que permet codificar l'herència entre classes depèn del llenguatge de programació. No obstant això, molts llenguatges comparteixen sintaxi. Per exemple, Java, Scala, TypeScript i PHP utilitzen la paraula reservada `extends` per a indicar que una classe estén o hereta d'una altra.

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

Per la seva banda, C#, C++ i Kotlin utilitzen els dos punts (`:`).

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

En canvi, en Python, s'opta per la sintaxi següent:

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

1.3. Què hereta la subclasse de la superclasse?

Quan utilitzem el mecanisme d'herència és important saber quins membres de la superclasse hereten les subclasses i quins no. Això depèn, en part, del llenguatge de programació que s'utilitza. No obstant això, podem dir que de manera general les subclasses sí hereten:

- **Tots els atributs i mètodes.** No obstant això, la seva visibilitat dins de la subclasse depèn del modificador d'accés definit per a cadascun en la superclasse. Amb això volem dir que, depenent del modificador d'accés indicat en la superclasse, la subclasse pot usar un atribut o mètode com si s'hagués codificat en la pròpia subclasse o no. És a dir, segons el modificador d'accés declarat en la superclasse, la subclasse té accés directe, o no, a l'atribut o mètode heretat.

Per contra, les subclasses no hereten:

- **Constructors de la superclasse.** En el mòdul «Abstracció i encapsulació» vam dir que el constructor no és considerat membre d'una classe perquè precisament no s'hereta. Cada classe ha de definir els seus propis constructors, encara que molts llenguatges criden (o permeten cridar) un constructor de la superclasse des del constructor de la subclasse.

- **Destructor.** Els llenguatges que tenen un destructor *de facto* (no un mètode que actua com si fos un destructor, per exemple, `finalize` en Java) no permeten heretar el destructor en les subclasses pel mateix motiu explicat per al constructor.

- **Constructor estàtic.** Els llenguatges que permeten definir un constructor estàtic, com C#, no permeten heretar-lo en les subclasses.

Com hem dit, el modificador d'accés assignat a cada membre de la superclasse restringeix si la subclasse pot accedir a aquest de manera directa, és a dir, com si hagués estat definit en la pròpia subclasse. En general, els llenguatges de programació orientats a objectes compleixen les regles següents:

a) En els llenguatges en què no es pot indicar com es du a terme l'herència (per exemple, Java, C#, PHP, Scala, etc.), els atributs i els mètodes heretats en les subclasses són heretats amb el mateix modificador d'accés que l'especificat en la superclasse (tret que es digui el contrari mitjançant sobreescriptura). En altres llenguatges, com C++, es pot indicar si l'herència és `public`, `protected` o `private`. Per exemple, si l'herència és `protected`, llavors els atributs i mètodes de la superclasse que siguin `public` o `protected` seran `protected` en la subclasse.

Vegeu també

El concepte de *sobreescriptura* el veurem a l'apartat 1.7 d'aquest mòdul.

b) `Public`: els membres definits com a `public` en la superclasse, en ser heretats per les subclasses, són visibles tant dins de les subclasses com fora d'elles. És a dir, poden ser cridats com si aquests haguessin estat codificats en les subclasses i, al seu torn, definits com a `public`.

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

class B extends A{ //"extends" serveix per dir "hereta"
     public B(){
        super();
        method(); //el podem cridar perquè forma part de la
//classe B (ve heretat de A). Imprimirà "Hola".
     }
}

class C{
     public C(){ //Imprimeix 2 vegades "Hola"
        B objectB = new B();
        objectB.method(); //el podem cridar perquè és un mètode
//públic de la classe B que ve heretat de la classe A.
     }
```

Vegeu també

En l'apartat 1.6 d'aquest mòdul veurem com fer una crida explícita a un constructor de la superclasse des del constructor de la subclasse. En Java, s'usa la paraula reservada `super`.

c) `Protected`: els membres definits com a `protected` en la superclasse, en ser heretats per les subclasses, són solament visibles dins de les subclasses. És a dir, no són accessibles des de fora de les subclasses. No obstant això, dins de les subclasses es poden utilitzar com si aquests s'haguessin codificat en les subclasses.

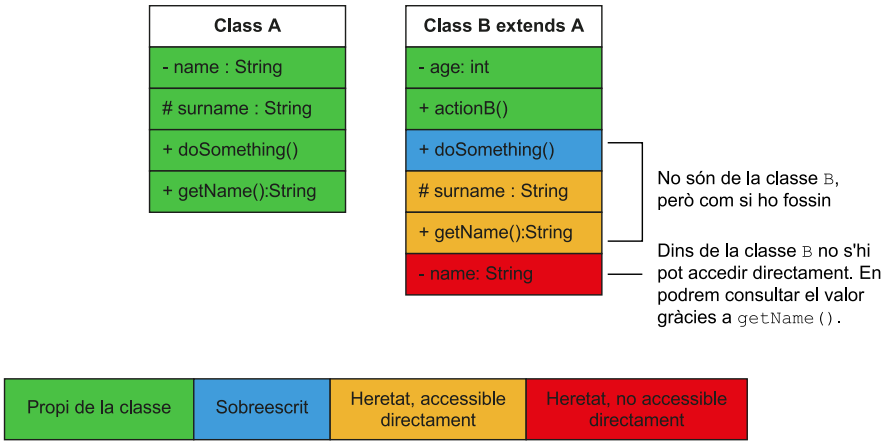
```
//Codi Java
class A{
     protected void method(){
          print "Hola";
     }
}

class B extends A{ //"extends" serveix per a dir "hereta"
     public B(){
        method(); //el podem cridar perquè forma part de la
//classe B (ve heretat de A). Imprimirà "Hola".
     }
}

class C{
     public C(){ //Imprimeix 1 vegada "Hola" amb la crida
        B objectB = new B();
        objectB.method(); //error, és protected en B, no public
     }
}
```

d) `Private`: els membres declarats com a `private` en la superclasse són heretats per les subclasses, però no són accessibles directament des de dins de les subclasses. En el cas dels atributs, si se'ls vol assignar un valor, o consultar el que tenen, l'operació s'ha de fer utilitzant un *getter* o *setter* `public` o `protected` heretat de la superclasse.

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

     public A(){
          value = 50; //només és accessible dins de la classe A
     }

     public int getValue(){
          return value;
     }
}

class B extends A{ //"extends" serveix per dir "hereta"
     public B(){
          print getValue(); //getValue és accessible i retorna 50
          print value; //error, value és private en A i, malgrat que és
//heretat per la subclasse B, dins d'aquesta no s'hi pot accedir
//directament.
     }
}

class C{
     public C(){
          B objectB = new B(); //Assumim que no dona error...
          objectB.getValue(); //getValue és accessible, retorna 50
          print objectB.value; //error, aquest atribut no és accessible
     }
}
```

Com ja vam comentar en el mòdul «Abstracció i encapsulació», alguns llenguatges de programació orientats a objectes defineixen altres modificadors d'accés; per exemple, `internal` (C#) i `package-private` (Java). Per a saber com aquests modificadors afecten durant l'herència, hem de consultar la documentació del llenguatge en qüestió. Així mateix, alguns llenguatges de programació defineixen comportaments especials per a les classes imbricades que hereten de la classe contenidora.

Finalment, vegem amb un exemple com cal imaginar-se el concepte d'herència.



Tot el que la classe `B` hereta de la classe `A` és com si estigués recollit en un objecte d'`A` que està dins de `B`. No obstant això, cal tenir present que quan s'instancia un objecte de la classe `B`, no es crea un objecte `B` i un objecte `A`, sinó que solament es crea un objecte `B` que conté el de `B` i el de `A`. Així doncs, cal anar amb compte en el disseny de les classes, ja que, com que s'hereta tot de la superclasse, si hi ha alguna cosa de la superclasse que no volem en la subclasse, ho tindrem igualment i, encara pitjor, ocupant memòria per a cada objecte instanciat de la subclasse.

1.4. Tipus d'herència

Hi ha dos tipus d'herència: simple i múltiple. Vegem cadascuna.

1.4.1. Herència simple

L'herència simple és el cas més comú d'herència. És el més utilitzat perquè és el tipus d'herència permès per tots els llenguatges de programació orientats a objectes. En què consisteix?

Definició d'herència simple

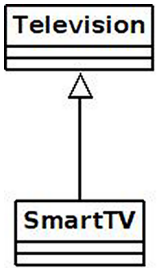
En l'herència simple, una subclasse solament pot heretar directament d'una superclasse.

L'exemple anterior dels mamífers és un cas clar d'herència simple. Vegem-ne algun més per acabar d'entendre el concepte.

Exemple 2 (herència simple) – Smart TV

L'Elena i el David han comprat un nou televisor, per ser més exactes, han adquirit una smart TV. Quina diferència amb el que tenien fins ara! Cert, però la funcionalitat principal, que consisteix a mostrar els canals de televisió, continua essent la mateixa. Així mateix, tant el nou televisor com el vell permeten pujar i baixar el volum, canviar de canal i fins i tot canviar el contrast. Llavors, què ha canviat? Doncs que el nou televisor (smart TV) ha especialitzat el concepte original de televisor. De fet, ha agafat com a punt de partida les funcionalitats d'un televisor i les ha millorat. La smart TV no deixa de ser

un televisor, l'única cosa és que, a més de fer tot el que fa el televisor antic, afegeix un comportament nou, que és accedir a Internet.



Per tant, la classe `SmartTV` hereta de la classe `Television`. D'aquesta manera, `SmartTV` és una subclasse que especialitza la superclasse `Television`.

1.4.2. Herència múltiple

L'herència múltiple està present en el nostre entorn, per tant, conceptualment té sentit i, per això, és possible representar-la en un diagrama de classes UML. No obstant això, aquesta és menys comuna en els programes que l'herència simple perquè molts llenguatges de programació orientats a objectes no la suporten, com és el cas de Java, C#, TypeScript i PHP. No obstant això, llenguatges com C++ i Python sí la suporten. Però, en què consisteix l'herència múltiple?

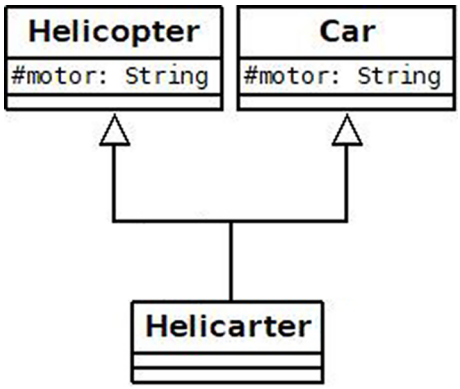
Definició d'herència múltiple

En l'herència múltiple, una classe filla pot heretar directament de més d'una superclasse.

Vegem-ne un parell d'exemples:

Exemple 3 (herència múltiple) – L'helicarter

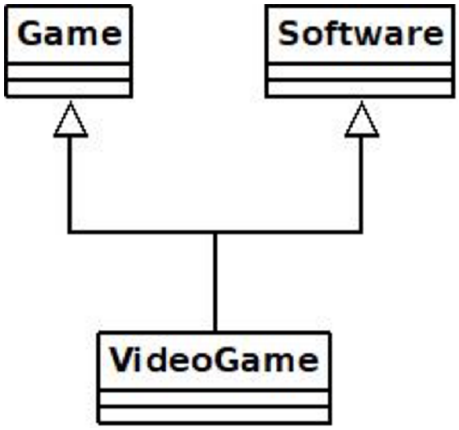
Imaginem que volem inventar un nou vehicle anomenat *helicarter* que té les virtuts de l'helicòpter i del cotxe. Així doncs, és fàcil pensar que l'`Helicarter` heretarà característiques de les classes `Helicopter` i `Car`, per la qual cosa el diagrama de classes UML quedaria de la manera següent:



Exemple 4 (herència múltiple) – El videojoc

Un exemple més real és el videojoc. Aquest element d'entreteniment és alhora un joc (classe `Game`) amb les seves regles, nombre de jugadors, etc., i un programa d'ordinador (classe `Software`) amb les seves línies de codi, amb la informació del llenguatge de pro-

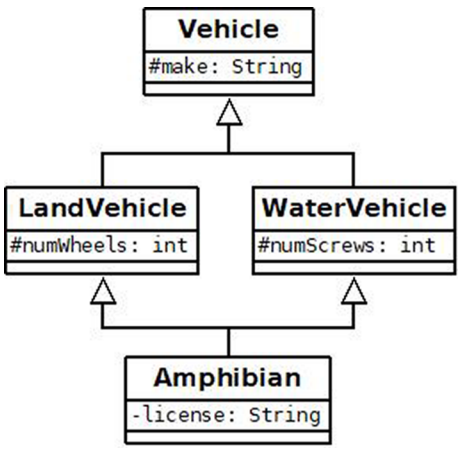
gramació utilitzat, etc. Així doncs, el diagrama de classes que relaciona aquestes tres classes seria:



Un cop vistos els exemples anteriors, ara intentarem explicar per què molts llenguatges –com Java o C#– van decidir no donar suport a l'herència múltiple. Si tornem a l'exemple de l'`Helicarter`, veiem que les dues superclasses tenen un atribut anomenat `motor` que és un `String`. Doncs bé, la subclasse `Helicarter` heretarà tots dos atributs. A causa d'això, el llenguatge de programació haurà de dotar el programador d'algun mecanisme que li permeti indicar a quin dels dos atributs repetits fa referència en cada instrucció del codi. Això fa confusa la lectura del codi i, al seu torn, pot ocasionar errors mentre es codifica.

Un altre dels problemes que pot provocar l'herència múltiple és una ambigüitat coneguda com a *problema del diamant*. Aquest problema consisteix, bàsicament, en una ambigüitat que sorgeix quan les classes `B` i `C` hereten d'una superclasse `A`, i, posteriorment, una quarta classe `D` hereta de `B` i `C`. Llavors, si des de la classe `D` es crida un mètode definit en la classe `A`, per on hereta `D` aquest mètode?, per `B` o per `C`? Cada llenguatge de programació que permet herència múltiple resol aquest problema de manera diferent. És per això que molts llenguatges, amb tal de simplificar els problemes potencials que cal resoldre, no permeten l'herència múltiple.

Exemple 5 (el problema del diamant) – El vehicle amfibi, un problema



El diagrama de classes anterior mostra clarament el problema del diamant. La classe `Amphibian` tindrà els atributs `license` ('matrícula'), `numWheels` ('nombre de rodes') i `numScrews` ('nombre d'hèlixs'). No obstant això, també tindrà l'atribut `make`, però per on el rep?, per `LandVehicle` o per `WaterVehicle`?

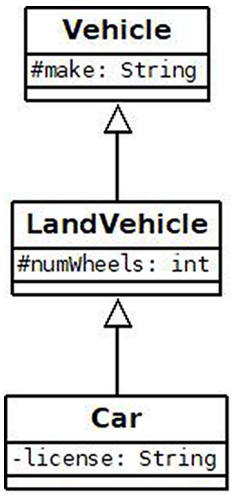
Finalment, cal dir que si tenim un diagrama de classes en el qual apareix herència múltiple i el llenguatge de programació que usarem no ho suporta, llavors haurem de modificar, durant la codificació, la relació entre classes basada en herència múltiple perquè s'adapti de la millor manera possible al llenguatge de programació escollit. Cal recordar que, en primera instància, un diagrama de classes UML ha de ser independent del llenguatge de programació utilitzat per a codificar-lo. Així doncs, si en modelar un problema ens apareix una herència múltiple, l'hem de dibuixar en el diagrama de classes, ja que conceptualment té sentit. Qualsevol canvi que sigui necessari a causa del llenguatge de programació que s'utilitzarà, el farem en el moment de la codificació. També podem fer dues versions del diagrama de classes, una de genèrica (i.e. independent del llenguatge) i una altra d'adaptada al llenguatge escollit. D'aquesta manera, delimitem perfectament què concerneix a la fase de disseny i què a la fase d'implementació del programa o projecte.

1.5. Transitivitat

Transitivitat

El problema del diamant de l'Exemple 5 també és un cas de transitivitat.

Molts llenguatges de programació permeten la transitivitat entre classes, és a dir, una classe `B` hereta d'una superclasse `A` i, al seu torn, una classe `C` hereta de la classe `B`, essent `A i B` superclasses per a la classe `C`. En aquest cas, la classe `B` serà la superclasse directa de la classe `C`.



En aquest exemple, la classe `LandVehicle` té l'atribut `numWheels` definit en la mateixa classe, i l'atribut `make` heretat de `Vehicle`, al qual podrà accedir directament tal com accedeix a `numWheels`. És a dir, `make` en la classe `LandVehicle` es comporta com si s'hagués definit en la mateixa classe `LandVehicle`. En la classe `LandVehicle`, l'atribut `make` continua essent `protected`. Per la seva banda, la classe `Car` tindrà l'atribut privat `license` declarat en la mateixa classe, com també els atributs protegits `numWheels` i `make` heretats de

les superclasses `LandVehicle` i `Vehicle` respectivament. La classe `Car` hereta l'atribut `make` perquè `LandVehicle` (la seva superclasse directa) l'ha heretat prèviament.

1.6. Constructor de la subclasse

Com ja hem comentat, una subclasse hereta automàticament, a més dels mètodes, tots els atributs de la superclasse. Aquests atributs normalment han de ser inicialitzats quan l'objecte és instanciat. Habitualment, fem aquesta inicialització en el constructor. En molts llenguatges, si no creem un constructor, el compilador en crearà un per defecte. Per tot això, és una bona pràctica que el constructor de la subclasse cridi el constructor de la seva superclasse com a part de la inicialització. De fet, els atributs privats heretats no són visibles en la subclasse i, per inicialitzar-los, o ho fem per mitjà de mètodes heretats que siguin visibles en la subclasse o mitjançant el constructor de la superclasse.

Així doncs, la majoria de llenguatges forcen al fet que si es vol cridar explícitament el constructor de la superclasse, aquesta crida hagi de ser la primera instrucció del constructor de la subclasse. Depenent del llenguatge, aquesta crida al constructor de la superclasse s'ha de fer d'una manera o una altra. Per exemple, en Java, dins del cos del constructor de la subclasse, s'usa el mètode especial `super`, la signatura del qual coincideix amb la signatura de qualsevol dels constructors de la superclasse.

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

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

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

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

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

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

Per la seva banda, en PHP, en comptes de `super`, s'usa `parent::__construct()`, en què `__construct()` funciona com `super`, és a dir, la seva signatura ha de coincidir amb la del constructor de la superclasse.

Finalment, llenguatges com C#, C++, Kotlin o Scala forcen al fet que s'invoqui el constructor de la superclasse en la pròpia signatura del constructor de la subclasse. En C#, per exemple, aquesta crida es fa mitjançant la paraula reservada `base`.

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

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

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

En C++, en permetre herència múltiple, no usa una paraula reservada, sinó que directament utilitza el nom de la superclasse. Scala, per la seva banda, és bastant similar.

```
//Codi C++ de la subclasse SubC
class SubC : SuperC{
     protected:
          string surname;

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

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

Kotlin permet usar els dos tipus de crida anteriors: en línia (en la definició de la classe) i amb l'ús de la paraula `super` (com en Java), però en comptes d'invocar en el cos del constructor de la subclasse, s'invoca en la signatura.

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

}
```

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

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

Arribats a aquest punt, cal saber que en molts llenguatges, com Java o C#, si no cridem explícitament en el constructor de la subclasse al constructor de la superclasse –mitjançant `super` en Java i `base` en C#–, llavors el compilador cridarà automàticament el constructor per defecte (és a dir, sense arguments) de la superclasse. En aquest cas, si la superclasse no té un constructor per defecte definit pel programador, llavors obtindrem un error en temps de compilació.

Mètode `this`

Molts llenguatges de programació com Java, Kotlin, C#, etc. permeten usar un mètode especial anomenat `this` per a cridar des d'un constructor a un altre constructor de la pròpia classe i així eliminar redundància de codi.

1.7. Sobreescriptura

Molts llenguatges de programació permeten que una subclasse modifiqui (redefineixi) els mètodes heretats de la superclasse als quals té accés directe. És a dir, dels tres modificadors d'accés que hem comentat en l'apartat 1.3, una subclasse pot redefinir o sobreescriure els mètodes declarats com a `public` o

`protected` en la superclasse, però no `private`. Aquesta modificació o redefinició d'un mètode de la superclasse en la subclasse s'anomena **sobreescriptura** (*overriding*).

En general, en molts llenguatges, la sobreescriptura es regeix per les regles següents:

a) A més de tenir accés directe al mètode des de dins de la subclasse, el mètode que es vol sobreescriure ha de ser declarat o marcat com a `virtual` en la superclasse. Depenent del llenguatge, els mètodes són marcats `virtual` per defecte o no. Per exemple, en Java o Scala, tots els mètodes són `virtual` per defecte, mentre que en C++ o C#, per defecte, no ho són. Altres llenguatges com Kotlin usen un concepte similar amb el *keyword* `open` i es comporten de manera semblant a C#.

b) Excepte en llenguatges més antics com el C++, el modificador d'accés (nivell de visibilitat) que hem d'assignar al mètode sobreescrit ha de ser igual o menys restrictiu que el definit per la superclasse. És a dir, si un mètode en la superclasse és `protected`, aquest no pot ser sobreescrit en la subclasse amb un nivell d'accés `private`. En aquest cas, el mètode sobreescrit solament pot ser definit en la subclasse com a `protected` (idèntic nivell o modificador d'accés que en la superclasse) o `public`.

c) Les modificacions realitzades en el mètode sobreescrit dins de la subclasse solament afecten la subclasse i les classes que heretin d'aquesta, però en cap cas afecten la superclasse.

d) Els mètodes de la superclasse marcats com a `final` (Java, Scala, Kotlin, PHP o C++11) o `sealed` (en C#) no poden ser sobreescrits per les seves subclasses (ni subclasses de les subclasses, etc.).

e) Els mètodes de la superclasse marcats com a `static` no poden ser sobreescrits per les seves subclasses (i subclasses de les subclasses, etc.). El motiu és que els mètodes estàtics són vinculats en temps de compilació usant el tipus de classe, no d'execució usant objectes.

f) Els atributs no poden ser sobreescrits en les subclasses, però sí amagats (*shadowed*). Com? Declarant un atribut en la subclasse amb el mateix nom que el que té un atribut de la superclasse, llavors l'atribut heretat de la superclasse queda ocult, no eliminat, la qual cosa hi fa més difícil l'accés.

```
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:

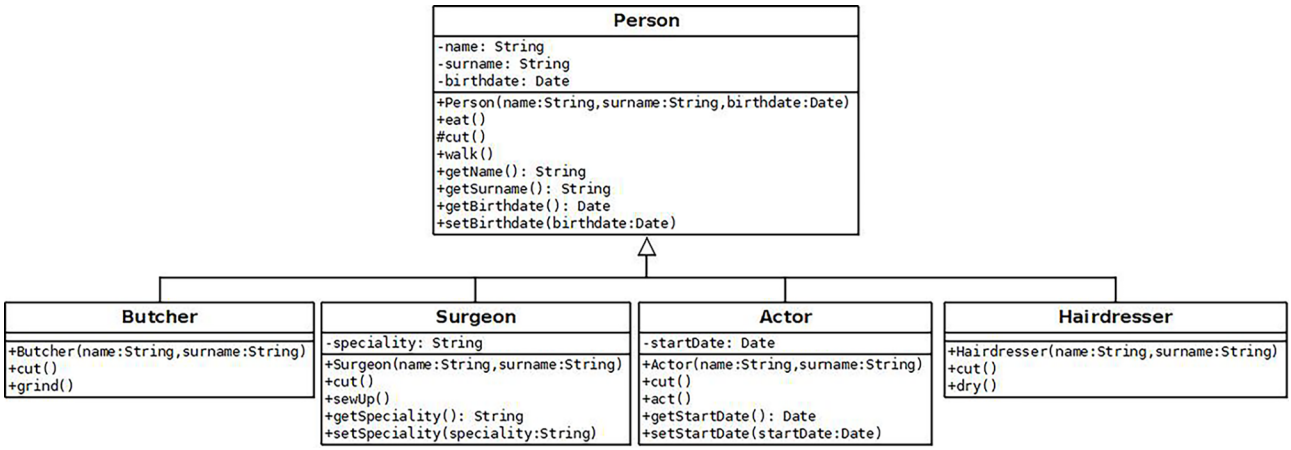
```
B objectB = new B();
objectB.hello(); //Imprimeix primer "David" i després "Elena".
```

No obstant això, alguns llenguatges com Scala o Kotlin permeten la sobreescriptura d'atributs (en Kotlin en veritat són *properties*, que és una extensió del concepte d'atribut).

Quan és útil sobreescriure un mètode en una subclasse? Vegem un exemple:

Exemple 6 (sobreescriptura) – No totes les persones tallem igual

El David, que és perruquer, ha quedat amb els seus amics: el Josep (que és carnisser), el Xavi (que és cirurgià) i el Carles (que és actor). Els quatre tenen una professió que qualsevol persona pot exercir. De fet, un carnisser, un cirurgià, un actor i un perruquer no deixen de ser persones. Així doncs, podem modelar aquest context de la manera següent:



En aquest exemple, les subclasses `Butcher` ('carnisser'), `Surgeon` ('cirurgià'), `Actor` i `Hairdresser` ('perruquer') hereten els atributs i mètodes de la superclasse `Person`. En el cas dels membres privats (`name`, `surname` i `birthdate`), aquests són heretats, però les quatre subclasses no tindran accés directe dins d'elles; però per a la resta de membres, sí que tindran accés directe. És a dir, és com si a `Butcher`, per exemple, haguéssim escrit o codificat `eat`, `cut`, `walk`, `getName`, `getSurname`, `getBirthdate` i `setBirthdate`. Recordem que el constructor de la superclasse no s'hereta.

Per la seva banda, les classes `Surgeon` i `Actor` afegeixen un atribut privat: `speciality` i `startDate`, respectivament. Així mateix, les quatre subclasses afegeixen un mètode públic: `grind` ('picar'), `sewUp` ('cosir'), `act` ('actuar') i `dry` ('eixugar'), respectivament.

Finalment, en les quatre subclasses apareix el mètode `cut`, que ja està definit en la superclasse `Person`. Això vol dir que les quatre classes sobreescriuen (és a dir, modifiquen el comportament) el mètode `cut` que hereten de `Person`. Òbviament, un carnisser fa l'acció de tallar (`cut`) de manera molt diferent de com la fa un cirurgià i un perruquer, i ja no diguem un actor, el «tallar» del qual (el famós «talleu» que criden els directors d'un film als actors) no té res a veure amb dividir coses en trossos.

D'altra banda, cal destacar que gràcies a l'herència reduïm la redundància de codi. Imagineu, si no, haver d'escriure exactament el mateix codi per al mètode `eat` tant en la

classe `Butcher` com en `Surgeon`, `Actor` i `Hairdresser`. Al mateix temps, especialitzem o concretem el comportament dels mètodes heretats que tenen particularitats segons la subclasse i, a més, afegim nous membres quan és necessari.

En l'exemple anterior, hem indicat la sobreescriptura en el diagrama de classes UML repetint el mètode sobreescrit en la subclasse que el sobreescriu. Així doncs, si no apareix el mètode heretat en la subclasse, s'entén que es comporta igual que en la superclasse. No obstant això, l'especificació 2.5.1 d'UML indica que els membres heretats (entesos com aquells als quals es pot accedir directament en la subclasse) per una subclasse (en el cas dels mètodes és independent si se sobreescriuen o no) es poden indicar anteposant el símbol «^» (*caret*). Així doncs, si volem fer èmfasi en els mètodes sobreescrits, podem usar el símbol «^», encara que el seu ús és molt poc freqüent.

Finalment, cal dir que molts llenguatges de programació permeten cridar des dels mètodes de les subclasses qualsevol mètode de la superclasse que sigui visible des de la subclasse amb la finalitat de reaprofitar al màxim el codi ja escrit. Així, si un mètode que sobreescrivim en la subclasse ha de fer el mateix que el de la superclasse i «una mica més», llavors solament hem de codificar aquest «una mica més». En Java, TypeScript, Kotlin i Scala s'usa la *keyword* `super`, en C# s'usa `base` i en C++ s'utilitza `::` entre el nom de la superclasse i el mètode d'aquesta que cal cridar; per exemple, `SuperClass::doSomething()`. A més, molts llenguatges (com Java o TypeScript) solament permeten accedir a la superclasse més immediata (recordem que hi pot haver transitivitat), altres en canvi sí que permeten escalar més en la jerarquia, per exemple, C++. Vegem un exemple d'ús de `super` en Java:

```
//Codi Java
public void doSomething(){ //Mètode sobreescrit
     super.doSomething(); //Crida al mateix mètode, però de la superclasse
     age = 36;
}
```

2. Classes i mètodes abstractes

El concepte *abstracte* està íntimament relacionat amb l'herència. En el paradigma de la programació orientada a objectes, tant les classes com els mètodes poden ser abstractes.

2.1. Classe abstracta

Comencem definint què és una classe abstracta.

Definició de classe abstracta

Una classe abstracta es defineix com aquella que no pot ser instanciada. És a dir, no podem crear objectes a partir d'aquesta classe.

Una de les utilitats de definir una classe abstracta és la d'actuar com a superclasse per unificar atributs i mètodes comuns de les seves subclasses. Així doncs, l'objectiu principal d'una classe abstracta és ser heretada i donar una codificació comuna per defecte a totes les subclasses que hereten d'ella. Per aquest motiu, una classe abstracta s'ha d'entendre com una abstracció d'una funcionalitat, comportament o lògica comuna, més que l'abstracció d'un conjunt d'entitats (objectes) concretes.

En general, una classe abstracta declara l'existència de tots els seus mètodes, però no els codifica tots. Els mètodes que no codifica, són declarats com a mètodes abstractes (ho veurem a continuació).

Una classe abstracta pot ser heretada de la mateixa manera que una classe «normal».

Per definir una classe abstracta, la majoria de llenguatges proveeixen el programador amb la paraula reservada `abstract`, la qual ha de ser escrita en la definició de la classe.

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

Classes i mètodes abstractes en Python

Aquest llenguatge no proporciona per si mateix la possibilitat de crear elements abstractes. No obstant això, existeix un mòdul anomenat `abc` que ofereix aquesta característica. Gràcies al mòdul `abc`, una classe és abstracta heretant d'`ABC` i un mètode és abstracte usant el decorador `@abstractmethod`.

Sabent que una classe abstracta no es pot instanciar, té sentit que tingui definit un o diversos constructors? Doncs sí, ja que si una classe hereta d'una classe abstracta, el primer que ha de fer el seu constructor és cridar el constructor de la superclasse, que és abstracta.

Finalment, cal destacar que, quan dues classes repeteixen el mateix codi (per exemple, un mètode anomenat igual i que fa el mateix), és senyal que hem de fer alguna cosa per a evitar aquesta duplicitat i facilitar el futur manteniment del programa. Una solució és posar tot el codi compartit en una superclasse que unifiqui la lògica de les dues classes, les quals passaran a ser subclasses de la superclasse unificadora. Si aquesta superclasse solament té sentit perquè fa d'element comú de les diferents subclasses, llavors aquesta nova superclasse ha de ser abstracta per a evitar que es creïn objectes d'aquest tipus.

2.2. Mètode abstracte

Una classe abstracta pot contenir, o no, mètodes abstractes.

Definició de mètode abstracte

Un mètode abstracte és aquell que solament fa constar la seva firma o signatura, res més. És a dir, no té claus i, per tant, no conté codi en el seu cos. Serveix per a avisar que aquest mètode s'ha de codificar en alguna de les subclasses (o subclasses de les subclasses, etc.) que heretin de la superclasse en la qual es troba aquest mètode abstracte.

Un mètode és abstracte si compleix les quatre condicions següents:

- La seva definició conté la paraula reservada `abstract`.
- No pot ser `private`, ja que no serà visible en les subclasses i, per tant, no pot ser sobreescrit.
- No pot ser `static`.
- No pot ser `final` (veurem què significa més endavant).

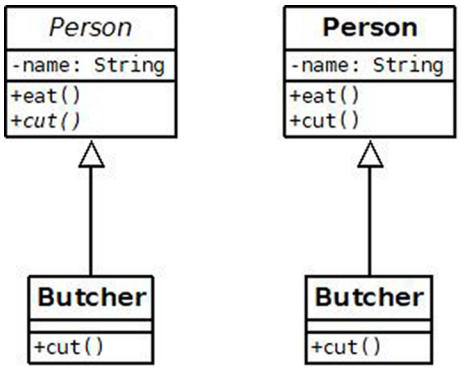
És important saber que **si una classe conté almenys un mètode abstracte**, llavors **la classe ha de ser declarada abstracta**. Això té molt sentit, perquè si no fos així, podríem instanciar un objecte de la classe abstracta, i si cridéssim el mètode abstracte, el programa no sabria com comportar-se, ja que no hem escrit codi per a aquest mètode. És a dir, podem veure un mètode abstracte com un mètode «inacabat» i, per tant, la classe que el conté també està inacabada o incompleta i, per tant, ha de 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.";
     }
}
```

Com podem veure, un mètode abstracte és molt útil si no té sentit donar una codificació per defecte o comuna a la classe abstracta i, al mateix temps, volem assegurar-nos que cada subclasse proporciona la seva pròpia codificació d'aquest mètode. En l'exemple anterior, els objectes de les quatre classes (recordem: `Butcher`, `Surgeon`, `Actor` i `Hairdresser`) mengen de la mateixa manera, però no tallen igual. D'aquesta manera, gràcies a l'herència i a l'ús d'un mètode abstracte, reaprofitem en les quatre classes el comportament comú definit en el mètode `eat` de la superclasse `Person`; i, al seu torn, permetem que cada subclasse especialitzi o concreti el comportament o codi del mètode `cut`. Òbviament, l'ús de mètodes abstractes es pot combinar amb la sobreescriptura de mètodes per a dissenyar el nostre programa segons ens interessi.

Exemple 7 (mètode abstracte versus mètode sobreescrit) - Trobar la diferència

Donats els dos diagrames de classes següents:



Pots veure la diferència entre ells? En el diagrama de l'esquerra, el mètode `cut` (escrit en cursiva) és abstracte en la superclasse (el nom del qual també està escrit en cursiva), és a dir, no té codi en la classe `Person` i és en la subclasse `Butcher` on es codifica. No obstant això, en el diagrama de la dreta, la classe `Person` proporciona un codi per al mètode `cut` (el qual no és abstracte perquè no està en cursiva), i aquest és sobreescrit en la subclasse `Butcher`. Així mateix, en el diagrama de l'esquerra no es poden crear objectes de la classe `Person` perquè és una classe abstracta (està escrita en cursiva), mentre que en el de la dreta sí.

2.3. Representació en un diagrama de classes UML

Com has pogut veure en l'apartat anterior, en un diagrama de classes UML representem una classe o un mètode abstracte escrivint el nom de la classe o el nom del mètode en cursiva. Una vegada que el mètode abstracte deixa de ser-ho perquè se li ha proporcionat codi, llavors ja no s'escriu en cursiva. Si, en l'exemple anterior, la classe `Butcher` del diagrama de l'esquerra no hagués codificat o sobreescrit el mètode `cut`, llavors tant `Butcher` com `cut` haurien d'estar en cursiva.

3. Classes, mètodes i atributs finals o segellats

El concepte *final* o *segellat* per a les classes i mètodes està íntimament relacionat amb l'herència.

3.1. Classe final

En primer lloc direm què és una classe final o segellada.

Definició de classe `final`

Una classe `final` és aquella que pot ser instanciada, però no heretada.

Així doncs, no podem crear subclasses a partir d'una classe `final`.

En Java, Scala, PHP i C++11 es denomina classe `final`, mentre que en C#, per exemple, s'anomena *classe segellada*. De fet, hem d'usar la *keyword* `final` en Java, PHP i C++11, i `sealed` en C#, davant de la classe per a indicar que volem que es comporti com a tal.

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

3.2. Mètode final

Una classe pot contenir o no, mètodes `final`.

Definició de mètode `final`

Un mètode `final` és aquell que no pot ser sobreescrit per una subclasse.

Un mètode és `final` si compleix les dues condicions següents:

- La seva definició conté la paraula reservada `final` (o l'equivalent del llenguatge). En C#, a causa que els mètodes per defecte no són `virtual`, solament podem indicar que un mètode és segellat a partir de la primera subclasse, per la qual cosa hem d'escriure `sealed override`.

- No pot ser abstracte.

Un mètode `final` permet establir un punt a partir del qual garantim que no es poden produir més canvis en el comportament d'aquest mètode per part de cap subclasse.

`final` en Kotlin

En Kotlin, totes les classes i mètodes són `final` per defecte. Així mateix, hi ha les *sealed classes*, que no tenen res a veure amb el `sealed` de C#. En Kotlin està relacionat amb els `enumeration`.

3.3. Atribut final

Aprofitem que estem parlant de `final` per explicar el concepte d'atribut `final`:

Definició d'atribut `final`

És l'atribut el valor del qual solament pot ser assignat una vegada. Seria com una constant.

`final` en Python

En Python, el concepte de `final` no existeix pròpiament dit.

Els llenguatges de programació defineixen aquest concepte de maneres molt diverses. Per exemple, Java, Scala i PHP utilitzen la *keyword* `final` en la declaració de l'atribut. En C#, el més semblant seria usar el modificador `readonly` en la declaració de l'atribut. No obstant això, en Python no hi ha una *keyword* equivalent a aquest concepte.

3.4. Representació en un diagrama de classes UML

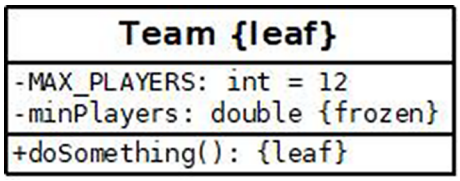
La manera de representar en un diagrama de classes un element `final` depèn de l'element en si mateix:

- **Una classe `final`:** es representa escrivint la propietat *{leaf}* a sota o a prop del nom de la classe. D'aquesta manera s'està dient que la classe és la fulla (*leaf*, en anglès) d'un arbre o, dit d'una altra manera, l'última classe d'una jerarquia. Per tant, no es pot heretar d'ella. Que una classe no pugui ser heretada és el mateix que dir que aquesta classe és `final`.

- **Mètode `final`:** no hi ha un estàndard, ja que UML modela un programa a més alt nivell i no recull totes les característiques que proporcionen els llenguatges de programació. Així doncs, tenim certa flexibilitat per a indicar les semàntiques que no estan recollides en l'estàndard. Per aquest motiu, una manera d'indicar que un mètode és final seria posant la propietat *{leaf}* al costat dret del mètode.

- **Atribut `final`:** sol ser una pràctica molt estesa distingir un atribut `final` de la resta d'atributs escrivint-lo seguint la convenció de noms de les constants usada en la majoria dels llenguatges de programació, és a dir, tot escrit en majúscules i els espais representats per guions baixos (*underscore*). També podem trobar l'ús de les propietats *{readOnly}*, *{frozen}* o *{const}* al costat dret de l'atribut. No obstant això, recomanem usar sempre la con-

venció de noms de les constants. Això sí, l'ús de les propietats es pot combinar amb la convenció de noms de les constants.



4. Interfície

Una interfície és un element molt utilitzat, però que no tots els llenguatges de programació orientats a objectes suporten; per exemple, C++ i Python. També hi ha llenguatges que, a causa de la seva pròpia evolució, han anat modificant el concepte original d'interfície per dotar-lo de més potencial. És el cas, per exemple, de Java.

4.1. Concepte original

La definició més essencial d'interfície seria:

Definició d'interfície

És una plantilla o esquelet que declara, però no codifica, un conjunt de mètodes que les classes que utilitzen la interfície han de codificar. És a dir, la interfície declara els mètodes (concretament les firmes o signatures), però en delega la codificació a les classes que la usen.

Les interfícies funcionen com a contractes, és a dir, quan una classe utilitza una interfície s'està comprometent a codificar les funcionalitats indicades en la interfície. Així doncs, la interfície és un contracte que diu què fan les classes, però no especifica com ho fan.

En alguns llibres, webs, anotacions, etc., potser llegeixes que una interfície és com una classe 100% abstracta, ja que no subministra el codi de cap dels mètodes declarats. Evidentment, una classe abstracta i una interfície, encara que s'assemblen, no són el mateix. Algunes de les diferències més significatives són les següents:

- Una classe no hereta una interfície, sinó que la implementa. Així doncs, direm que la classe `A` implementa la interfície `B`.

- No obstant això, una interfície sí que pot heretar una altra interfície, però mai implementar-la. Això és senzill d'entendre, ja que si una interfície `A` implementés una altra interfície `B`, la interfície `A`, pel mer fet de ser interfície, no podria codificar els mètodes de la interfície `B`.

- Una interfície no pot heretar d'una classe, ja que estaria heretant codi de la superclasse, i ja hem dit que una interfície no proporciona el codi dels seus mètodes.

- La majoria de llenguatges que suporten interfícies, però no suporten l'herència múltiple, sí que permeten que una classe implementi més d'una interfície. Aquest és el cas, per exemple, de Java o C#. D'aquesta manera, «simulen» l'herència múltiple gràcies a la implementació de múltiples interfícies.

- D'igual manera que les classes abstractes, les interfícies no poden ser instanciades.

- A diferència de les classes abstractes, les interfícies no tenen constructors ni destructors.

- Tots els membres d'una interfície han de ser `public`. De fet, en alguns llenguatges no és necessari (o fins i tot està prohibit) indicar el modificador d'accés `public` perquè és l'únic modificador d'accés vàlid.

- Conceptualment, una interfície no pot contenir atributs, ni tan sols estàtics. No obstant això, alguns llenguatges permeten saltar-se aquesta restricció, encara que amb limitacions.

El nom de les interfícies segueix la mateixa nomenclatura que la de les classes. No obstant això, és molt freqüent adjectivar el nom de la interfície amb el sufix anglès «`-able`», per exemple, `Runnable`, `Comparable`, etc.

4.2. Traduint a codi

En general, una interfície es declara amb la *keyword* `interface` (per exemple, Java, C#, Kotlin o TypeScript). Depenent del llenguatge, la classe que la implementa usa una sintaxi o una altra. Per exemple, en Java s'usa la *keyword* `implements`, i en C# s'usen els dos punts (`:`).

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

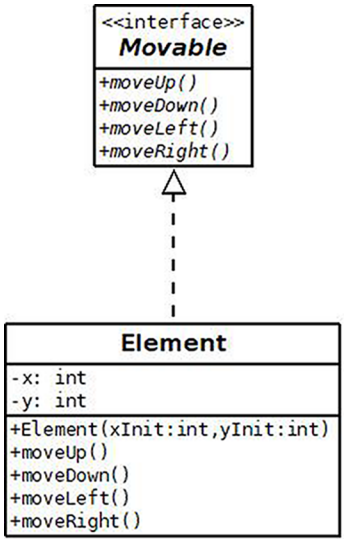
```java
//Codi Java
public class Element implements Movable{
     private int x, y;

     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. Representació en un diagrama de classes UML

En un diagrama de classes UML també podem definir interfícies. Aquestes es representen amb una caixa amb dos compartiments:

a) Nom de la interfície: el compartiment superior conté el nom de la interfície. Aquest s'escriu en negreta i en cursiva (com si fos una classe abstracta). A més, el nom va acompanyat de l'estereotip `<<interface>>`.

b) Mètodes: l'últim compartiment conté els mètodes de la interfície escrits en cursiva (són, per definició d'interfície, abstractes).



Com podem veure, es dibuixa un triangle blanc en la interfície i d'aquest surten línies discontínues cap a les classes que la implementen.

4.4. Quan s'ha d'usar una interfície?

És possible que arribats a aquest punt algú es pregunti quan s'ha d'usar una interfície i quan una classe abstracta, dos elements que són molt similars, però no iguals. Aquesta pregunta se la fa molta gent, per la qual cosa es converteix en el centre de molts debats. A continuació intentarem donar algunes directrius, encara que l'experiència i el problema o context en el qual ens trobem seran determinants per a la decisió que prenguem.

S'escull una classe abstracta (i herència) si es pot dir que «`B` és un tipus de la classe `A`» (`A` serà una classe abstracta i `B` una classe que heretarà de `A`). En canvi, s'usa una interfície si la relació encaixa millor amb la frase «`B` és capaç de fer `A`» (`A` serà un interfície i `B` una classe que la implementarà).

Així doncs, la classe abstracta s'usa per a representar el que els objectes d'aquesta classe són (la seva identitat), mentre que la interfície s'utilitza per a representar el que els objectes de les classes que la implementin són capaços de fer (habilitats). Per exemple, podem dir que «un chihuahua és un tipus de gos», però no té sentit dir que «un chihuahua és capaç de ser un gos», per tant, definiríem `Dog` ('gos') com una classe abstracta. En canvi, podem dir que les persones, els animals i els vehicles són capaços de moure's o de comparar-se, però no són ni un tipus de moviment ni de comparació. En aquest cas, les tres

classes esmentades –`Person`, `Animal` i `Vehicle`– implementaran les interfícies `Movable` i `Comparable`. D'aquesta manera, els objectes de tres classes *a priori* inconnexes com són `Person`, `Animal` i `Vehicle`, estarien relacionats entre ells gràcies al fet que les tres classes a les quals pertanyen els objectes es relacionen entre elles mitjançant la interfície que implementen, no d'unes superclasses que hereten. Per tant, **amb la interfície hem forçat una relació entre classes que no estan relacionades mitjançant una jerarquia d'herència**.

De vegades, la restricció de només poder heretar de manera simple que imposen molts llenguatges de programació pesarà en la decisió, ja que en tots els llenguatges la implementació múltiple d'interfícies està permesa.

Finalment, mostrem una taula comparativa que pot ajudar a veure les diferències (i similituds) entre interfície i classe abstracta (alguns llenguatges poden tenir matisos en algun aspecte indicat en la taula).

Característica	Interfície	Classe abstracta
Mecanisme d'ús per part de les classes	Implementació (pot ser múltiple).	Herència (múltiple, segons llenguatge).
Modificador d'accés dels membres	Els membres d'una interfície no tenen modificador d'accés, s'assumeix que són public.	Les classes abstractes poden definir diferents modificadors d'accés per als seus membres.
Atributs	No en pot tenir (o sí, però amb restriccions).	Sí en pot tenir.
Mètodes	Tots abstractes.	Concrets (amb codi) i abstractes (sense codi).
Funcionalitat	És 100% abstracta, per la qual cosa no proporciona codi.	Pot tenir codi que defineix un comportament per defecte per als mètodes no abstractes, els quals poden ser sobreescrits en les subclasses.
Keyword abstract	L'ús d'aquest *keyword* és opcional per a declarar els mètodes (s'assumeix que ho són).	És obligatori usar aquesta *keyword* per a declarar un mètode com a abstracte.
Modificació d'un mètode (p. ex. canvi de nom, en els paràmetres, tipus de retorn, s'afegeix un mètode, etc.)	S'han de revisar totes les classes que la implementen per a codificar les novetats del mètode, si no, tindrem errors de compilació.	Una modificació en la classe abstracta pot no afectar les subclasses si la classe abstracta proporciona el codi del mètode no abstracte i aquest no es crida a les subclasses.
Herència	No pot heretar de cap classe, però sí d'interfícies (herència simple o múltiple, segons el llenguatge).	Pot heretar de superclasses (herència simple o múltiple, segons el llenguatge).
Implementació	No pot implementar cap interfície.	Pot implementar múltiples interfícies.
Constructor o destructor	No en pot tenir.	En pot tenir.
Instanciació	No es pot instanciar un objecte a partir d'una interfície.	No es pot instanciar un objecte a partir d'una classe abstracta.

Característica	Interfície	Classe abstracta
Reutilització de codi	No facilita la reutilització.	Permet definir codi per als mètodes no abstractes, la qual cosa n'afavoreix la reutilització per part de les subclasses.
Ús	Defineix capacitats perifèriques d'una classe.	Defineix identitat d'una classe.

5. Polimorfisme

El mecanisme de polimorfisme és un dels pilars de la programació orientada a objectes (POO) i està íntimament relacionat amb l'herència.

5.1. Concepte

Podem definir el mecanisme de polimorfisme de la manera següent:

Definició de polimorfisme

Característica de la POO que permet modificar el tipus d'un objecte en temps d'execució basant-se en una jerarquia d'herència.

D'una manera més simple i resumida, podem dir:

El polimorfisme és un mecanisme que permet que una variable o referència d'una classe es comporti com un objecte de qualsevol de les seves subclasses (o subclasses de les subclasses, etc.).

Recordem que una subclasse té tots els atributs i mètodes de la seva superclasse (perquè els hereta). Així doncs, això significa que una subclasse pot fer qualsevol cosa que la seva superclasse sigui capaç de fer. Podríem dir, si no hi ha sobreescriptura, que en qualsevol part del nostre programa en què necessitem un objecte de la superclasse, aquest podria ser substituït per un objecte d'una de les seves subclasses i el codi funcionaria igual. Encara més, a una referència (variable) del tipus de la superclasse s'hi podria assignar un objecte de qualsevol de les seves subclasses i tot funcionaria correctament perquè la subclasse no deixa de ser la superclasse estesa (amb més atributs o mètodes o modificacions en el comportament dels mètodes). Per tant, qualsevol objecte d'una subclasse pot ser assignat a qualsevol referència (variable) de les seves superclasses (qualsevol classe que estigui per damunt en la jerarquia d'herència).

Ara pensa que una subclasse (`Actor`) sobreescriu un mètode anomenat `cut()` de la seva superclasse (`Person`). Si tenim el codi següent:

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

Quin mètode es crida?, el de la superclasse o el de la subclasse? En aquest cas es crida el mètode `cut` de la subclasse, ja que, a la variable `carlos`, li hem assignat un objecte de tipus `Actor` (subclasse). Aquest és el cas més comú de polimorfisme, és a dir, quan s'usa una referència (variable) d'una superclasse per a referenciar un objecte d'una subclasse. Aquest cas és conegut com a **polimorfisme dinàmic (o simplement *polimorfisme*)**, encara que també rep els noms de *run-time polymorphism*, *dynamic binding* o *late binding*. Com pots veure, els termes inclouen els conceptes de «temps d'execució» o «enllaçat tardà». Per què? Perquè com que la creació de l'objecte que s'assigna a la referència o variable es fa en temps d'execució, llavors la decisió de quin mètode es crida solament es pot prendre quan s'executa el programa, no quan aquest es compila.

En aquest punt, és important entendre dos conceptes: tipus estàtic i tipus dinàmic.

- **Tipus estàtic (*static type*):** és el tipus definit en la declaració d'una variable. Aquest tipus pot ser una classe concreta (i.e. normal), una classe abstracta o una interfície.

- **Tipus dinàmic (*dynamic type*):** és el tipus de l'objecte al qual es refereix o apunta una variable o, dit d'una altra manera, és el tipus de l'objecte que s'assigna a una variable.

Així doncs, el polimorfisme dinàmic es dona quan el tipus dinàmic d'una variable o referència determina el mètode que es crida. És a dir:

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

Tipus estàtic — Tipus dinàmic

En aquest exemple, `carlos` té com a tipus estàtic `Person` i com a tipus dinàmic `Actor`. El tipus estàtic d'una variable o referència defineix què es pot cridar amb aquesta variable o referència i què no.

- **Mètodes de la superclasse (tipus estàtic):** amb l'objecte `carlos` podrem cridar a tots els mètodes definits en la classe `Person`, perquè la subclasse `Actor` té tots els mètodes de la superclasse `Person`. Si el mètode està sobreescrit en la subclasse, llavors es crida el mètode *sobreescrit*.

- **Mètodes exclusius de la subclasse (tipus dinàmic):** amb l'objecte `carlos` no podrem cridar mètodes que siguin exclusius de la subclasse `Actor`, ja que `carlos` és variable de tipus `Person` i no sap res dels mètodes d'`Actor`.

En resum:

- Un objecte de la subclasse pot ser assignat a una variable o referència declarada amb el tipus d'una superclasse.

- Una variable o referència declarada com a superclasse (tipus estàtic) i assignada a un objecte subclasse (tipus dinàmic) pot cridar els mètodes definits en la superclasse, però si estan sobreescrits en la subclasse es cridaran les versions sobreescrites d'aquests mètodes.

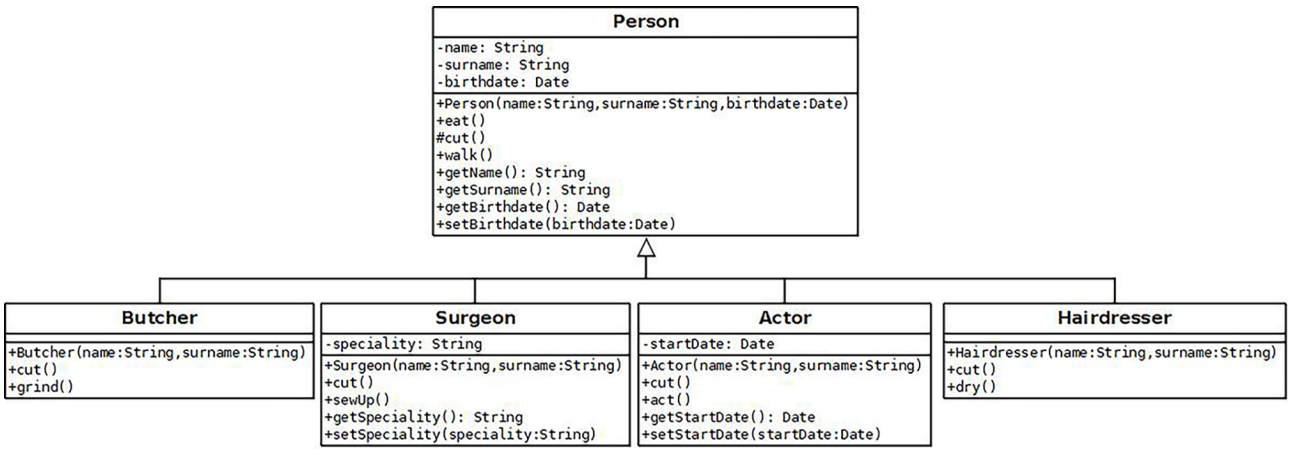
- Una variable o referència declarada com a superclasse (tipus estàtic) i assignada a un objecte subclasse (tipus dinàmic) no pot cridar els mètodes definits exclusivament en la subclasse.

Com a anècdota, podem dir que hi ha el terme, menys utilitzat, de **polimorfisme estàtic** (*static binding*, en anglès), que no és altre que el que es decideix en temps de compilació. I quan ocorre? Doncs succeeix quan se sobrecarrega un mètode. Com que en la sobrecàrrega el que canvia és el nombre o tipus d'arguments (el tipus de retorn també pot variar, però no és suficient per a fer sobrecàrrega), llavors el compilador pot saber quina versió del mètode ha de cridar gràcies al nombre i tipus de paràmetres passats en la crida. Així doncs, podem dir que polimorfisme estàtic i sobrecàrrega d'un mètode són sinònims.

5.2. Exemple d'ús

Un dels aspectes clau del polimorfisme és que permet separar les funcionalitats o habilitats dels objectes de la seva codificació, és a dir, separar la interfície (signatures dels mètodes) de la implementació per a poder programar basant-nos en la interfície, no en la codificació. Això és molt útil en programes complexos. Vegem-ne un exemple a partir d'un cas ja esmentat en aquestes anotacions: la manera de tallar de les persones.

Exemple 8 (polimorfisme) – Una llista de persones cadascuna amb les seves particularitats



Com hem vist més amunt, un carnisser no talla igual que un cirurgià, que un actor o que un perruquer. Així doncs, la superclasse `Person` defineix una manera de tallar genèrica per a qualsevol persona, i les diferents subclasses sobreescriuen el mètode `cut` per proporcionar-ne la codificació o comportament específic.

```
class Person{ //Segons el context, Person podria haver estat 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";
     }
}
```

Ara mira el codi següent:

```
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(); //crida cada codificació de cut en el
//següent ordre: Butcher, Surgeon, Actor, Hairdresser i Person
}

people[4].walk(); //crida walk de Person
people[0].walk(); //crida walk de Butcher que és igual al de Person
//perquè no l'ha sobreescrit

people[0].grind(); //error: mètode exclusiu de Butcher que Person
//desconeix... el tipus estàtic de people[0] és Person i el tipus dinàmic
//és Butcher
```

Si et fixes en l'exemple anterior, gràcies a l'herència i al polimorfisme, hem pogut crear un *array* de `Person` i emmagatzemar objectes de cinc classes diferents (perquè tots són implícitament la classe `Person`): `Butcher`, `Surgeon`, `Actor`, `Hairdresser` i `Person`. Serà el tipus dinàmic de cada variable o referència (en aquest cas, casella de l'*array*) qui determini quina versió del mètode `cut` s'ha de cridar en cada cas.

Si `Person` hagués estat una classe abstracta, o fins i tot una interfície, tot funcionaria igual, excepte que no podríem instanciar un objecte de tipus `Person` tal com fem en la casella 4 de l'*array* `people`.

5.3. *Càsting*: *upcasting* i *downcasting*

El concepte de *càsting* –o simplement *cast*– consisteix a fer una conversió de tipus, per exemple, de `int` a `float`, de nombre a text, etc. No obstant això, aquest procés, quan es fa amb objectes, llavors està íntimament relacionat amb el polimorfisme i, per tant, també amb l'herència. Hi ha dos tipus de *càsting*: *upcasting* i *downcasting*.

> **Definició d'*upcasting* (o *generalization* o *winderning*)**
>
> Consisteix a assignar un objecte d'una subclasse a una referència/variable el tipus estàtic de la qual és igual al d'alguna de les seves superclasses.

Així doncs, podem fer:

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

Aquest procés sempre és segur, ja que un objecte de tipus subclasse té els membres de la superclasse i pot fer tot el que fa la seva superclasse, és a dir, un actor és un tipus de persona (`Actor` hereta de `Person`). Com és obvi, no és necessari indicar l'*upcasting* de manera explícita:

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

El tipus de l'objecte (en aquest cas, `Actor`) no canvia a causa de l'*upcasting*, l'objecte `Actor` continua essent un `Actor`, no un `Person`. L'única cosa que canvia és la referència o variable (`carlos`) el tipus estàtic de la qual (superclasse `Person`) és d'un nivell superior en la jerarquia d'herència al de l'objecte assignat (subclasse `Actor`).

Com ja hem comentat, ara amb la referència `carlos` no podrem cridar els mètodes exclusius d'`Actor`, perquè no estan en la interfície (i.e. la llista de mètodes) de `Person`. Així doncs, fer *upcasting* (moure's cap amunt en la jerarquia d'herència) comporta la pèrdua d'accés a les característiques pròpies de la subclasse. Per què? Perquè `carlos` es tracta com `Person` encara que l'objecte sigui (i no deixa de ser-ho) un `Actor`. Així doncs, com que `carlos` es tracta com un `Person`, els membres d'`Actor` queden amagats (però no eliminats, podrem recuperar-los fent *downcasting*).

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

Consisteix a convertir un objecte amb tipus estàtic igual a una superclasse i tipus dinàmic igual a una subclasse en un objecte el tipus estàtic del qual sigui igual al tipus dinàmic.

Així doncs, podem fer:

```
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 procés de *downcasting* necessita un *cast* explícit, és a dir, indicar entre parèntesis la subclasse a la qual s'ha de convertir l'objecte. A diferència de l'*upcasting*, el *downcasting* no és un procés segur, ja que l'objecte al qual s'aplica el *downcasting* pot no pertànyer a la subclasse indicada. Si es produeix un error en fer el *downcasting*, aquest es produirà en temps d'execució. El *downcasting* ens porta a descendir per la jerarquia d'herència.

Gràcies al procés de *downcasting*, la referència pot usar els mètodes exclusius de la subclasse. Així doncs, en l'exemple anterior seria correcte fer:

```
    Actor actor = (Actor) carlos; //downcast tracta "carlos" com a Actor
    actor.act(); //OK:
    carlos.act();//KO, perquè el tipus estàtic de carlos és Person
    Butcher butcher = (Actor) carlos; //KO, butcher no és pare d'Actor,
//sinó germà
    Butcher butcher = (Butcher) carlos; //KO, el tipus dinàmic de carlos és
//Actor, no Butcher
```

El procés de *downcasting* és molt més freqüent que el d'*upcasting*. Usem *downcasting* quan volem accedir a membres específics o exclusius de la subclasse.

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

    if(p instanceof Actor) {
        //Només s'executa quan el tipus dinàmic de p és Actor
        Actor actor = (Actor) p; //downcast
        actor.act();
    }
}
```

Saber la classe d'un objecte

Molts llenguatges tenen un comparador que permet saber si un objecte és d'un tipus de classe concreta. Per exemple, Java i PHP tenen `instanceof`, C# i Kotlin tenen `is`, Python té el mètode `isinstance()`, Scala el mètode paramètric `isInstanceOf[Type]`, etc.

5.4. Problemes de rendiment

Cal tenir en compte que el fet de fer *càsting* (*upcasting* i sobretot *downcasting*) suposa una sèrie de verificacions en temps d'execució que penalitza el rendiment del programa. Òbviament, els dissenyadors dels llenguatges de programació ho saben i intenten minimitzar aquesta pèrdua de rendiment. El temps

extra que necessita el programa durant l'execució per a fer *upcasting* i *downcasting* apareix quan el programa ha de decidir quina versió d'un mètode s'ha de cridar. Aquesta verificació ocorre amb els mètodes declarats com a `virtual` (i que posteriorment han estat sobreescrits), però no amb els no virtuals. Amb aquests últims, el compilador ja sap quin mètode ha de cridar a partir del tipus amb el qual s'ha declarat la variable o referència.

En la majoria dels nostres programes no tindrem més problemes de rendiment a causa de l'ús del polimorfisme (i els avantatges en són molts), però és important saber que hi ha una possible pèrdua de rendiment amb el seu ús. Així doncs, en contextos en els quals el rendiment de l'aplicació és essencial, hem de saber que evitant el polimorfisme podem millorar el rendiment general del programa. En relació amb això, és important que en aquests contextos tinguem la màxima cura a l'hora de dissenyar les classes i solament definim, quan el llenguatge ens ho permeti (per exemple, C++ o C#), com a `virtual` els mètodes que realment es sobreescriuran. En Java, per exemple, tots els mètodes són `virtual` per defecte, així que en aquest cas cal fer-los `final`.

6. Consideracions

Una vegada vist què és l'herència i alguns aspectes relacionats amb ella (p.ex. polimorfisme, classe abstracta, interfície, etc.), val la pena reflexionar en termes de disseny dels programes. Si bé és cert que el fet de centralitzar en una classe els atributs i mètodes comuns a diferents objectes de diferents classes és, com hem vist, positiu, també hem de tenir en compte que pot augmentar la complexitat del nostre programa.

Quan dissenyem un programa, hem de buscar un equilibri entre ser el màxim de precisos a l'hora de modelar el problema o context que estem tractant i reduir la complexitat de l'ús i manteniment del nostre programari. Desgraciadament, no hi ha unes pautes i la decisió dependrà de diversos factors: la nostra experiència, el problema que cal resoldre, etc.

Sabies que hi ha gossos que no borden? Els gossos de la raça basenji no borden, en el seu lloc, emeten un so o udol similar al cant tirolès. Imaginem que volem modelar aquesta realitat fidelment. En aquest cas, separaríem la classe `Dog` ('gos') en dues subclasses: els gossos que borden (`BarkingDog`) i els que udolen a l'estil tirolès (`YodelingDog`). Té sentit conceptualment? Sí. Però per una raça de gos que no borda, val la pena crear dues classes noves i una relació d'herència? Segurament no. Aquest exemple és petit i es podria fer el disseny diferenciador esmentat, però imagina un programa molt més gran i complex: si vas prenent decisions d'aquest tipus sovint, al final el programa serà immanejable i difícil de mantenir. Així doncs, **en un programa gran, mantenir les coses el més simples possible és normalment la millor pràctica**.

> L'objectiu principal d'un disseny és aconseguir construir un programari que sigui prou flexible, però sense afegir massa complexitat (o no tanta que faci que el *software* sigui immanejable).

Finalment, és vital que quan dissenyem un programa en tinguem en ment possibles usos futurs i ampliacions. La idea és deixar la porta oberta, de manera que afegir nous elements –tant atributs com mètodes o fins i tot classes– sigui relativament senzill (i poc costós).

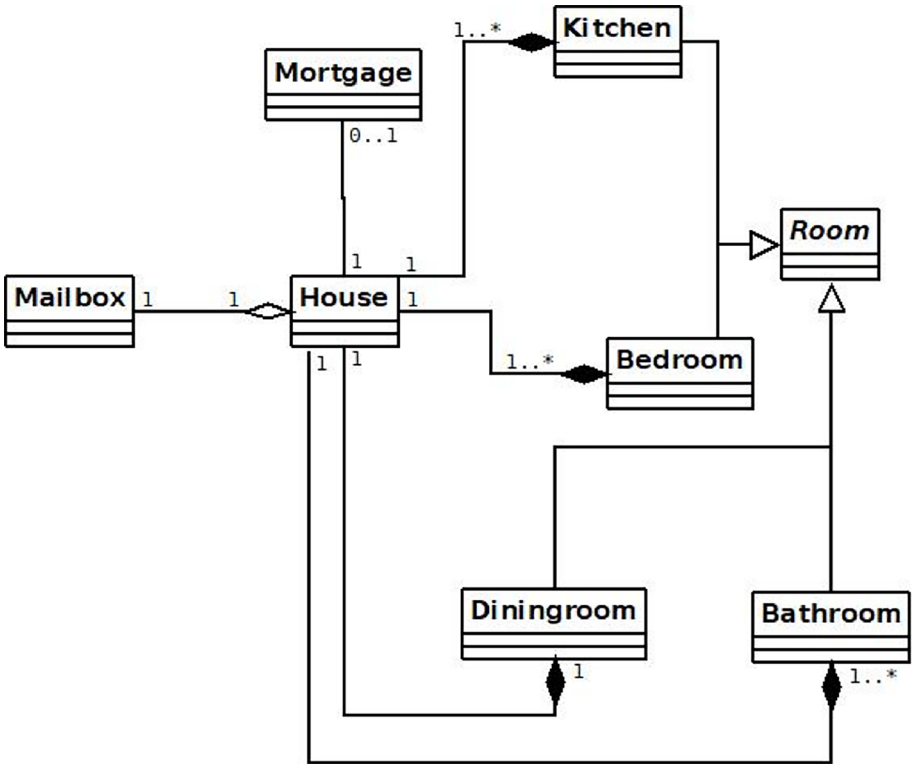
Com podem veure, la tasca d'analitzar i dissenyar un programari és més complexa del que sembla, ja que entren en joc moltes variables, entre aquestes, la capacitat d'abstracció del dissenyador.

7. Exemple resum

En aquest apartat presentem un exemple que inclou gran part dels conceptes que hem vist en aquest mòdul i en l'anterior. Primer, vegem el diagrama de classes UML i després analitzem-lo per acabar d'entendre'l.

Vídeo d'interès

Trobaràs aquest exemple en format audiovisual en el vídeo «Exemple de diagrama de classes UML» que tens disponible a l'aula de l'assignatura.



En general, una casa (`House`) està formada per una cuina (`Kitchen`), un o més dormitoris (`Bedroom`), un menjador (`Diningroom`), un o més banys (`Bathroom`), etc. Tots aquests elements són habitacions (`Room`); per aquest motiu hereten atributs i mètodes de la classe pare `Room` (la qual és abstracta).

Així doncs, podem dir que una casa està formada per habitacions o, dit d'una altra manera, les habitacions formen o són part d'una casa. Encara més, si la casa desapareix (és destruïda), llavors les habitacions desapareixen amb ella. No ocorre el mateix si desapareix un habitació (per exemple, perquè hem fet obres i canviem la distribució de la casa). Per aquest motiu, la relació de la classe `House` amb els quatre tipus d'habitació representades (`Kitchen`, `Bedroom`, `Diningroom` i `Bathroom`) és de tipus associació de composició.

D'altra banda, tota casa té una bústia. Estem pensant en una casa adossada amb jardí típica de les pel·lícules nord-americanes. La bústia, encara que forma part de la casa, pot existir sense que hi hagi casa (per exemple, fins i tot no estant

construïda la casa, la bústia ja pot estar instal·lada); així mateix, si traiem la bústia no es destrueix la casa. Per aquesta justificació, la relació entre les classes `House` i `Mailbox` és de tipus associació d'agregació.

Finalment, tenim la hipoteca (`Mortgage`). Si bé és cert que una casa pot tenir associada una hipoteca, aquesta no forma part de la casa. Per això, la relació és una associació binària. En aquest cas, com que no tenim informació addicional, hem decidit que la navegabilitat entre totes dues classes sigui bidireccional, és a dir, la casa sap de l'existència de la hipoteca i viceversa. Això podria haver estat diferent si ens haguessin donat més informació de context. Per exemple, es podria pensar que solament la hipoteca hauria de saber de l'existència de la casa (la punta de la fletxa acabaria en la classe `House`) i que la casa, en si mateixa, no hauria de saber que està hipotecada.

Resum

Herència

- Mecanisme que permet definir una classe nova (anomenada *subclasse*) a partir d'una altra ja existent (anomenada *superclasse*) descrivint-ne solament les diferències.
- La subclasse hereta tots els atributs i mètodes de la superclasse.
- En general, en la majoria dels llenguatges no s'hereten els constructors, destructors ni constructors estàtics.
- Hi ha llenguatges, com el C++, que permeten definir com es fa l'herència: `public`, `protected` o `private`. No obstant això, la majoria no permeten indicar la manera d'heretar.
- Els membres `public` o `protected` són heretats per les subclasses amb el mateix modificador d'accés (excepte en llenguatges en què es pot indicar la manera d'heretar), i s'hi pot accedir i es poden utilitzar en la subclasse com si s'haguessin definit en la pròpia subclasse.
- Els membres `private` en la superclasse són heretats per les subclasses, però aquestes no tenen accés directe a aquests membres. Per a accedir-hi han d'usar algun mètode `public` o `protected` heretat de la superclasse.
- Hi ha dos tipus d'herència: la simple, en la qual una subclasse solament pot heretar directament d'una superclasse; i la múltiple, en la qual una subclasse pot heretar directament de més d'una superclasse.
- La majoria de llenguatges permeten la transitivitat en la jerarquia d'herència.
- La sobreescriptura permet que una subclasse redefineixi el comportament (i.e. el codi) d'un mètode heretat i accessible directament (i.e. `public` o `protected` en la superclasse). Per a aconseguir-ho, el mètode en la superclasse ha de ser `virtual`.

Vídeo d'interès

Pots veure un resum de què és l'herència i els tipus que hi ha en el vídeo «Relació entre classes: Herència» que trobaràs a l'aula de l'assignatura.

Classes i mètodes abstractes

- Una classe abstracta és aquella que no pot ser instanciada. En general, declara l'existència de tots els seus mètodes, però no els codifica tots.

- Si una classe conté un mètode abstracte, llavors la classe també ha de ser abstracta. El codi del mètode abstracte ha de ser proporcionat per les subclasses (o subclasses de la subclasse, etc.) que hereten la superclasse abstracta en la qual està declarat el mètode abstracte.

- Una classe abstracta pot ser heretada, però no instanciada. No obstant això, pot tenir constructor.

- Les classes i mètodes abstractes s'escriuen en cursiva en els diagrames de classe UML.

Classes, mètodes i atributs finals

- Una classe `final` (o segellada) és la que pot ser instanciada però no heretada. En un diagrama de classes UML, normalment s'escriu la propietat *{leaf}* a sota o a prop del nom de la classe.

- Un mètode `final` és el que no pot ser sobreescrit per una subclasse. En un diagrama de classes UML normalment s'escriu la propietat *{leaf}* en el costat dret del mètode.

- Un atribut `final` és aquell el valor del qual solament pot ser assignat una vegada, és com una constant. En un diagrama de classes UML, normalment s'escriu el nom de l'atribut final tot en majúscules i amb els espais representats per guions baixos (*underscore*). També es poden usar les propietats *{readonly}*, *{frozen}* o *{const}* en el costat dret de l'atribut.

Interfície

- És una plantilla o esquelet que declara, però no codifica, un conjunt de mètodes que les classes que utilitzen la interfície han de codificar. És a dir, la interfície declara els mètodes (concretament les firmes o signatures), però en delega la codificació a les classes que la usen.

- Les classes no hereten una interfície, sinó que la implementen. En general, una classe pot implementar múltiples interfícies.

- Una interfície no pot implementar cap interfície, però sí la pot heretar. Tampoc no pot heretar una classe.

- Una interfície no pot ser instanciada ni tenir constructor ni destructor.

- Tots els membres d'una interfície han de ser `public`. Alguns llenguatges no permeten ni tan sols escriure el modificador d'accés `public` perquè és l'únic acceptat.

- Una interfície no pot contenir atributs, ni tan sols estàtics. No obstant això, alguns llenguatges se salten aquesta restricció sempre que es compleixin unes condicions.

- El nom d'una interfície segueix el mateix conveni que el de les classes, encara que és freqüent que estigui adjectivat amb el sufix anglès «`-able`», per exemple, `Comparable`.

- Les interfícies es representen en un diagrama de classes amb el nom de la interfície i dels mètodes en cursiva. A més a més, s'anteposa l'estereotip `«interface»` al nom de la interfície. En un diagrama de classes UML es dibuixa un triangle blanc en la interfície i d'ell surten línies discontínues cap a les classes que la implementen.

- S'escull una classe abstracta (i herència) si es pot dir que «`B` és un tipus de la classe `A`» (`A` serà una classe abstracta i `B` una classe que heretarà de `A`). En canvi, s'usa una interfície si la relació encaixa millor amb la frase «`B` és capaç de fer `A`» (`A` serà una interfície i `B` una classe que la implementarà).

Polimorfisme

- El polimorfisme és un mecanisme que permet que una variable o referència d'una classe es comporti com un objecte de qualsevol de les seves subclasses (o subclasses de les subclasses, etc.).

- Tipus estàtic: és el tipus definit en la declaració d'una variable. Aquest tipus pot ser una classe concreta (i.e. normal), una classe abstracta o una interfície.

- Tipus dinàmic: és el tipus de l'objecte al qual es refereix o apunta una variable. Dit d'una altra manera, és el tipus de l'objecte que s'assigna a una variable.

- Un objecte de la subclasse pot ser assignat a una variable o referència declarada amb el tipus d'una superclasse.

- Una variable o referència declarada com a superclasse (tipus estàtic) i assignada a un objecte subclasse (tipus dinàmic) pot cridar els mètodes definits en la superclasse, però si estan sobreescrits en la subclasse, es cridaran les versions sobreescrites d'aquests mètodes.

- Una variable o referència declarada com a superclasse (tipus estàtic) i assignada a un objecte subclasse (tipus dinàmic) no pot cridar els mètodes definits exclusivament en la subclasse.

- L'*upcasting* consisteix a assignar un objecte d'una subclasse a una referència el tipus estàtic de la qual és igual al d'alguna de les seves superclasses. L'*upcasting* sempre és segur.

- El *downcasting* consisteix a convertir un objecte amb tipus estàtic igual a una superclasse i tipus dinàmic igual a una subclasse en un objecte el tipus estàtic del qual sigui igual al tipus dinàmic. El *downcasting* no sempre és un procés segur.

Activitats

Exercici 1
Dibuixar i explicar el diagrama de classes d'una aplicació que permeti el control dels seients d'un teatre per a les diferents obres que s'hi representen. Per començar, recordem que cada obra té un títol, una sinopsi, el nom del director, la durada en minuts i l'edat a partir de la qual està recomanada. Les edats recomanades poden ser `TP` (i.e. tots els públics), `M7` (i.e. més grans de set anys), `M12` (i.e. més grans de dotze anys) i `M18` (i.e. més grans de divuit anys).

Cada obra té unes emissions en les quals es representa. Cada emissió està definida per una data (dia, mes i any; tipus `Date`) i una hora (tipus `Time`). A més, una emissió posa a la venda un conjunt de seients. Per a cada emissió, s'ha de guardar si el seient ja ha estat venut o no i el preu de venda d'aquest seient.

Cada seient té un número de fila i un número de butaca, així doncs, un seient seria «fila 5, butaca 2», per exemple. A més, cada seient té un atribut que indica el sector al qual pertany. Els sectors del teatre són: `PLATEA`, `BALCO`, `AMFITEATRE1` i `AMFITEATRE2`. Així doncs, hi ha més d'un seient «fila 5, butaca 2», però pertanyen a sectors diferents. A més dels seients normals, hi ha una varietat de seients, que es diuen VIP, que són iguals que els seients normals, però tenen uns braços més grans i inclouen un sistema de massatge (mitjançant un mètode anomenat `donarMassatge`). Aquests seients estan repartits per tot el teatre.

Nota: En el diagrama de classes s'han de recollir tant les classes com els seus atributs i mètodes rellevants (amb el seu nivell d'accés i tipus), com també les relacions i multiplicitats que hi ha entre les classes. No s'hi han d'incloure mètodes *getter* ni *setter*, com tampoc constructors de classe. Per a la resta de mètodes es pot considerar que no reben paràmetres ni tampoc retornen res. Els elements abstractes indiqueu-los anteposant `<<abstract>>`.

Exercici 2
Dibuixar i explicar el diagrama de classes d'una aplicació que permeti gestionar una lliga de futbol. En primer lloc, recordem que tots els equips tenen un nom, un any de fundació i un nom de president. Com és obvi, tots els equips estan compostos per un mínim d'onze jugadors fins a un màxim de vint-i-dos futbolistes. A més, tenen un entrenador. De tots ells volem saber-ne l'identificador (és un número proporcionat per la federació que pot contenir zeros per l'esquerra, per exemple, `001`), el nom, els cognoms i la data de naixement. Dels futbolistes, a més, volem saber-ne el dorsal i la demarcació en la qual juguen. Les demarcacions que hi ha són `PORTER`, `DEFENSA`, `MIGCAMPISTA` i `DAVANTER`. Quant als entrenadors, volem guardar l'any en què es van llicenciar.

Tant els futbolistes com l'entrenador viatgen i es concentren abans de cada partit. A més, els futbolistes entrenen i juguen, mentre que l'entrenador dirigeix.

Cada equip juga amb un altre en un partit, i cada partit té un equip local i un equip visitant. Volem saber quins partits disputa un equip com a local i quins com a visitant. A més, cada partit es disputa en una data concreta i, al final, té un resultat.

La lliga té àrbitres que s'encarreguen d'arbitrar els partits que se'ls assignen. Dels àrbitres, en guardem l'identificador (és un número proporcionat per la federació que pot contenir zeros per l'esquerra, per exemple, `001`), el nom, els cognoms i la data de naixement. Per qüestions d'estrès, un àrbitre no pot arbitrar més de quinze partits. Evidentment, un partit solament té quatre àrbitres. El diagrama, i per tant el programa, no ha de diferenciar entre àrbitre principal, linier i quart àrbitre.

Nota: En el diagrama de classes s'han de recollir tant les classes com els seus atributs i mètodes rellevants (amb el seu nivell d'accés i tipus), com també les relacions i multiplicitats que hi ha entre les classes. No s'hi han d'incloure mètodes *getter* ni *setter*, com tampoc constructors de classe. Per a la resta de mètodes es pot considerar que no reben paràmetres ni tampoc retornen res. Els elements abstractes indiqueu-los anteposant `<<abstract>>`.

Exercici 3
Dibuixar i explicar el diagrama de classes d'una aplicació que permeti el control de vols d'un aeroport. En aquesta aplicació guardarem informació sobre les companyies aèries que operen a l'aeroport que gestionem. Cada companyia aèria té un nom (per exemple, Iberia) i un identificador (per exemple, IB per Iberia), com també un conjunt de vols que operen. Si esborrem una companyia del sistema, n'esborrarem tots els vols.

Un vol es troba identificat per un codi i una companyia, i es du a terme des d'un aeroport d'origen a un aeroport de destinació en una data determinada. De cada aeroport volem guardar-ne el nom (per exemple, Madrid-Barajas) i l'identificador (per exemple, MAD). Així mateix, cada vol té assignat un sol avió, que està identificat per un codi alfanumèric i té unes

places determinades. Cada avió pot tenir assignats diversos vols. Des de l'aplicació volem saber tant l'avió assignat a un vol concret com els vols assignats a un avió determinat.

Per a operar un vol és necessari el treball de diferents persones. Aquestes persones poden ser, o bé personal de l'aeroport, o bé personal d'una companyia aèria. Tant el personal de l'aeroport com el de les companyies estan identificats per un número personal que coincideix amb el seu DNI (amb lletra) i un sou base.

Entre el personal necessari per a operar un vol, hi ha els controladors aeris. Aquests empleats, que són personal de l'aeroport, són els encarregats de controlar els vols de l'aeroport. Dins del control dels vols hi ha l'autorització de l'enlairament o de l'aterratge d'un vol (depèn de si el vol surt d'aquest aeroport o hi arriba). Per raons d'estrès, un controlador solament pot encarregar-se, com a màxim, de deu vols i, òbviament, un controlador solament gestiona un vol. A més, com que la seva feina és complexa, tenen un plus de responsabilitat que serà un percentatge del seu sou.

Finalment, hi ha el personal d'embarcament. Aquestes persones pertanyen a la companyia que opera el vol i per a aquest personal s'ha de guardar a quina companyia pertany. Aquestes persones facturen i realitzen l'embarcament dels vols de la companyia. Cadascuna d'aquestes persones pot ser assignada a un o més vols, i la facturació i embarcament d'un vol pot ser gestionat per una o més persones de la companyia. Si s'elimina una companyia del sistema perquè ja no opera més a l'aeroport, tot el seu personal també ha de ser eliminat de l'aplicació.

Nota: En el diagrama de classes s'han de recollir tant les classes com els seus atributs i mètodes rellevants (amb el seu nivell d'accés i tipus), com també les relacions i multiplicitats que hi ha entre les classes. No s'hi han d'incloure mètodes *getter* ni *setter*, com tampoc constructors de classe. Per a la resta de mètodes es pot considerar que no reben paràmetres ni tampoc retornen res. Els elements abstractes indiqueu-los anteposant `<<abstract>>`.

Exercici 4
Dibuixar i explicar el diagrama de classes que representa una aplicació que permeti a la Conselleria d'Educació d'una comunitat autònoma gestionar les escoles, els professors i els alumnes d'educació infantil i primària. Per començar, recordem que les escoles es defineixen amb un identificador alfanumèric, un nom, una adreça i un director que les dirigeix. A més, cadascuna d'elles pot ser pública, privada o concertada. Les escoles assignen els estudiants en les aules i decideixen qui són els tutors de cada aula.

Cada escola té un claustre de professors que volem conèixer en tot moment. Cada professor es defineix pel seu NIF, nom, cognoms i any en què es va graduar en magisteri. Cadascun també té una especialitat docent: matemàtiques, llengua, música o educació física. Un professor sempre pertany a una escola i, en un instant determinat, solament pot estar assignat a una. No interessa saber les escoles per les quals ha passat cada professor al llarg de la seva carrera docent, en canvi, donat un professor, sí que volem saber a quina escola pertany en el moment de la consulta. La tasca principal d'un professor és avaluar. Finalment, cal dir que un dels professors del claustre és el director de l'escola.

Així mateix, les escoles tenen alumnes. Cada alumne està definit amb un identificador format per nou dígits anomenat IDALU (identificador de l'alumne), el seu NIF (si en tenen, per defecte s'entendrà que no en tenen), el seu nom i els seus cognoms. A més, per a cada estudiant, l'escola ha de saber si té necessitats educatives especials (NEE). Un estudiant pot haver pertangut a diverses escoles al llarg de la seva vida acadèmica i aquesta és una informació que hem de guardar. És a dir, per a cada estudiant hem de tenir l'historial (o expedient) de cursos acadèmics que indiqui el següent: data del curs acadèmic (format per un any inicial i un de final, per exemple, 2018-2019), en quina escola va estudiar durant aquest curs acadèmic, en quin nivell acadèmic (es refereix a: P3 –tres anys–, P4, P5, primer –sis anys–, segon, tercer, quart, cinquè i sisè) i l'informe final obtingut –serà un text amb l'avaluació de les diferents assignatures. Si el curs acadèmic està en curs (i.e., és l'actual), aquest text estarà buit. Si un estudiant desapareix de l'aplicació (per exemple, mor, passa a l'ESO, etc.), el seu historial o expedient també.

Una escola està formada per aules. Aquestes tenen un nom (per exemple, 1-A), un àlies (per exemple, Tortugues), una ubicació definida per tres dígits (per exemple, 203, porta 03 de la planta 2 o 001, porta 01 de la planta 0), un nivell acadèmic, un tutor (és un professor) i un conjunt d'alumnes. El mínim d'alumnes per aula és de deu, amb un màxim de vint-i-sis. Un alumne solament pot pertànyer a una aula en un moment determinat. Donada una aula, volem saber quins alumnes i quin tutor té. Al seu torn, donat un alumne, volem saber a quina aula va. Donat un professor, no volem saber de quina aula és tutor (hi ha professors que no són tutors).

Nota: En el diagrama de classes s'han de recollir tant les classes com els seus atributs i mètodes rellevants (amb el seu nivell d'accés i tipus), com també les relacions i multiplicitats que hi ha entre les classes. No s'hi han d'incloure mètodes *getter* ni *setter*, com tampoc constructors

de classe. Per a la resta de mètodes es pot considerar que no reben paràmetres ni tampoc retornen res. Els elements abstractes indiqueu-los anteposant `<<abstract>>`.

Exercici 5
El servei de rodalies està format per línies, per exemple, R1 (Molins de Rei-Maçanet), C3 (Aranjuez-Chamartín), etc. Cada línia té un codi alfanumèric, un nom, un recorregut (és un text), una hora d'inici i una hora de finalització del servei. A més, en cada línia circula com a mínim un tren. El nombre màxim de trens per línia dependrà de la demanda, no hi ha límits. Volem saber, donada una línia, quins trens l'operen en aquest moment i si una línia desapareix, els trens que l'operen són ressituats en altres línies o desmuntats. Per defecte, un tren no té línia assignada.

Per a cada tren en guardarem l'identificador alfanumèric (per exemple: R1-0456, C3-0678, etc.). Un tren està format per una sola locomotora i un conjunt de vagons. A més, cada tren té definit l'espai en centímetres entre vagons. Així mateix, hem de ser capaços de saber la longitud total del tren a partir de la longitud de la seva locomotora, dels vagons i de l'espai entre aquests. Donat un tren, volem conèixer la línia que opera (evidentment, solament pot operar una línia en un moment determinat).

Tant la locomotora com els vagons poden considerar-se elements d'un tren amb les característiques comunes d'identificador alfanumèric i longitud. A més, la locomotora es defineix mitjançant la tipologia de propulsió, que pot ser elèctrica, dièsel o magnètica. Donada una locomotora, volem saber a quin tren pertany (solament pot estar vinculada a un tren) i viceversa.

Per la seva banda, els vagons es defineixen mitjançant el volum (en metres cúbics) i el tipus de contingut que poden transportar, és a dir, si és un vagó de passatgers, refrigerat o de mercaderies, per exemple, transporta paquets, cartes, etc. Cada tren té entre cinc i quinze vagons.

Cada dia, si és necessari, es configuren els trens, per tant, un mateix tren pot un dia tenir sis vagons i un altre, vuit. Tant els vagons com la locomotora que han estat trets del tren són utilitzats per a crear un altre tren o guardats a la cotxera. Evidentment, un vagó solament pot pertànyer a un tren en un moment determinat, i això és informació que ens interessa conèixer. Per contra, no ens interessa saber a quins trens ha pertangut cada vagó ni cada locomotora al llarg de la seva història. Per defecte, un vagó no pertanyerà a cap tren i el mateix ocorrerà amb les locomotores.

El personal que permet el bon funcionament del servei està format pels revisors i pels maquinistes. Aquests es defineixen amb el seu nom, cognoms, NIF i un horari (serà un text). Cada línia pot tenir assignats un màxim de tres revisors i volem saber qui són. Aquests s'encarreguen d'inspeccionar que els passatgers tinguin un bitllet vàlid. Un revisor pot tenir assignades diverses línies i volem saber quines.

Cada maquinista té associat un sol tren i s'encarrega de conduir-lo. Si no s'assigna una línia al tren, el maquinista és assignat a un tren amb línia. Donat un tren, volem saber qui n'és el maquinista i viceversa.

Nota: En el diagrama de classes s'han de recollir tant les classes com els seus atributs i mètodes rellevants (amb el seu nivell d'accés i tipus), com també les relacions i multiplicitats que hi ha entre les classes. No s'hi han d'incloure mètodes *getter* ni *setter*, com tampoc constructors de classe. Per a la resta de mètodes es pot considerar que no reben paràmetres ni tampoc retornen res. Els elements abstractes indiqueu-los anteposant `<<abstract>>`.

Exercici 6
Dibuixar i explicar el diagrama de classes que representa una aplicació d'escriptori que permet la gestió de la guia televisiva d'una plataforma de pagament. En primer lloc, sabem que el servidor multimèdia envia a l'aplicació informació sobre els programes de televisió. Cada programa està definit per un títol, una descripció, una durada en minuts, una data d'inici d'emissió, una data fins a la qual estarà disponible en la plataforma i un codi d'autoregulació que pot ser TP (i.e. tots els públics), +7, +12 o +18. Per a cada programa, hem de proporcionar un mecanisme que informi sobre la data de finalització del programa a partir de la data d'inici d'emissió i la durada del programa.

La interfície del programari ha de diferenciar els programes segons siguin sèries, pel·lícules o altres. Per a les pel·lícules, a més de la informació esmentada per als programes, aquestes tenen informació sobre el nom del director, el repartiment (és un text amb els noms dels actors separats per comes), l'any d'estrena i el gènere al qual pertanyen: acció, comèdia, romàntica, terror, suspens, oest o un altre.

Les sèries estan formades com a mínim per una temporada i proveeixen d'un mecanisme per a saber quantes temporades tenen. Cada temporada té un identificador numèric (per

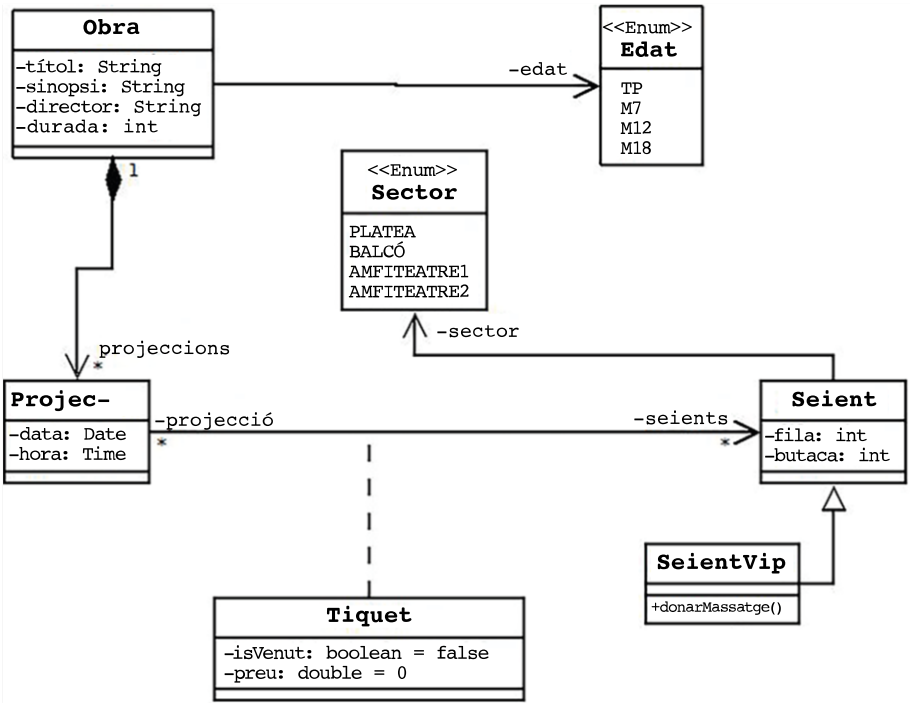
exemple, 1 per a la temporada 1) i un any. Cada temporada està formada per episodis (com a mínim un) i també permet saber quants episodis té. Per a cada episodi guardarem el títol, el director, la seva sinopsi i la durada en minuts. A més, per a cada episodi hem de saber quin és l'episodi previ i el posterior. Si una temporada desapareix, tots els seus episodis també. Si una sèrie desapareix del catàleg, totes les seves temporades també.

La informació que arriba del servidor al qual es connecta el programari també organitza els programes per cadenes; és a dir, l'usuari, a més de poder accedir per programa, també pot veure quins programes s'estan emetent en les diferents cadenes en aquest moment i a curt termini. Per a això, el servidor envia informació de les cadenes de televisió presents en la plataforma. En aquest sentit, cada cadena està definida per un nom i un número de canal. A més, juntament amb la informació anterior, també s'envia la graella televisiva de la cadena des del moment actual fins a un temps definit per la mateixa cadena. La graella televisiva és una llista ordenada de programes de televisió. Com a mínim, aquesta graella està formada pel programa que s'està emetent en el moment actual. Si una cadena de televisió desapareix, els seus programes no ho fan, ja que es deixen un temps en catàleg, concretament fins a la data de finalització de disponibilitat que hem esmentat més amunt. Així mateix, no tots els programes de la plataforma estan associats a un canal. Per exemple, la plataforma permet als seus clients veure pel·lícules de pagament que s'estan projectant als cinemes. Per a aquestes pel·lícules, a més de la informació esmentada per a les pel·lícules, també hem de guardar-ne el preu. Una vegada pagada, el client la pot veure a continuació, no més tard. En aquest sentit, no ens interessa guardar les compres fetes pels clients.

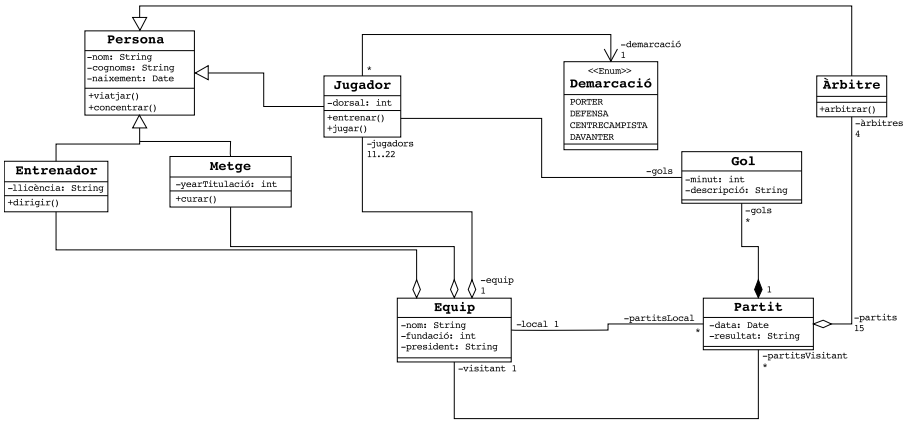
Nota: En el diagrama de classes s'han de recollir tant les classes com els seus atributs i mètodes rellevants (amb el seu nivell d'accés i tipus), com també les relacions i multiplicitats que hi ha entre les classes. No s'hi han d'incloure mètodes *getter* ni *setter*, com tampoc constructors de classe. Per a la resta de mètodes, es pot considerar que no reben paràmetres ni tampoc retornen res. Els elements abstractes indiqueu-los anteposant `<<abstract>>`.

Solucionari

Solució exercici 1

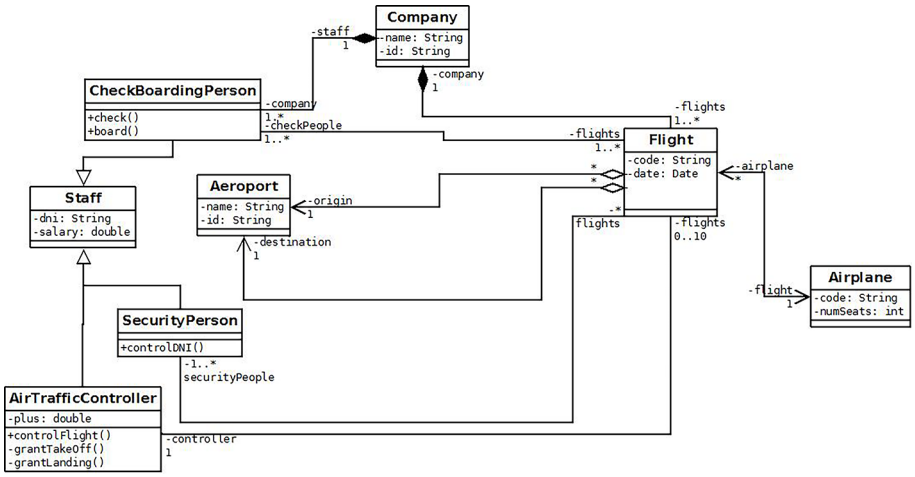


Solució exercici 2



La relació entre `Jugador` i `Partit` no pot tenir una classe associativa `Gol`, ja que les classes associatives tenen la restricció implícita que solament hi pot haver una instància de la classe associativa per cada parell d'objectes de la relació. Així doncs, si un jugador marca més d'un gol, llavors no es podria representar; però tal com està en el diagrama, sí.

Solució exercici 3



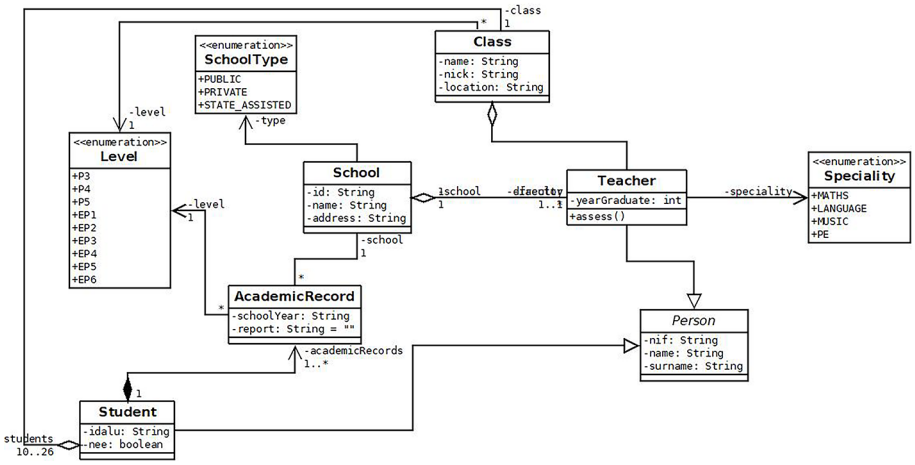
La relació de `Company` amb `CheckBoardingPerson` i `Flight` ha de ser de composició, perquè l'enunciat diu que si s'elimina la companyia s'esborren el seu personal i els seus vols.

La relació de `Flight` amb la resta d'elements pot ser binària o d'agregació, però no de composició.

És possible crear dues classes extra, `AirportStaff` i `CompanyStaff`, però no és necessari. En qualsevol cas, ha d'haver-hi relació d'herència entre les classes *staff* i els casos concrets.

En la classe del controlador aeri (`AirTrafficController`) hem considerat privats els mètodes de concedir enlairament (`grantTakeOff`) i aterratge (`grantLanding`).

Solució exercici 4



La classe `Persona` és abstracta i pare d'`Estudiant` i `Professor`. El professor avalua (és un mètode) i l'atribut `nee` d'`Estudiant` és un `boolean`.

L'expedient és una classe intermèdia entre `Estudiant` i `Escola`. La relació `Estudiant-Expedient` és de composició, perquè l'enunciat diu que si desapareix l'`Estudiant`, el seu expedient també. El valor de l'informe s'ha d'inicialitzar a «» (buit) o a `null`.

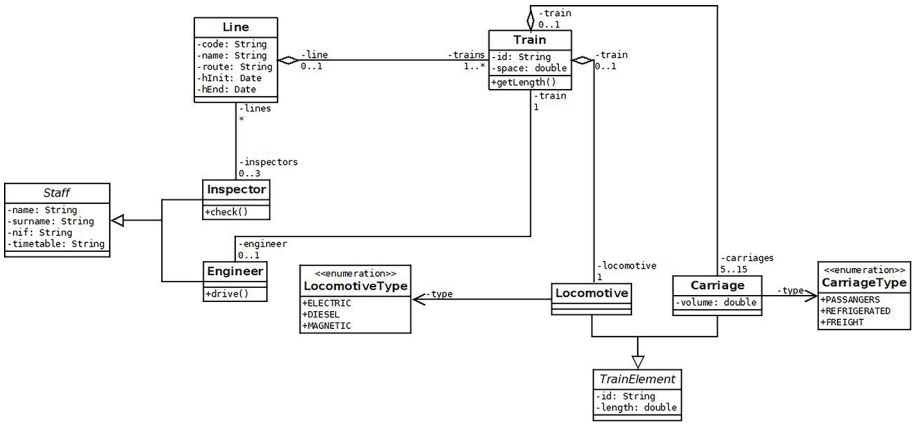
Les aules estan formades per estudiants. Aquesta relació és una agregació, perquè si s'elimina una aula, els estudiants no són eliminats, sinó ressituats. La navegabilitat és bidireccional, segons indica l'enunciat.

Les aules tenen un tutor que és un professor. La relació `Aula-Professor` és unidireccional, perquè l'enunciat deixa clar que es vol saber el tutor de l'aula, però, donat un professor, no es vol saber si és tutor d'un aula o no.

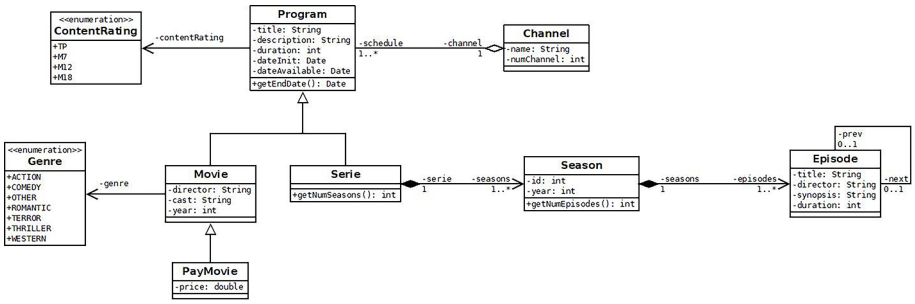
La relació `Escola-Professor` per a modelar el `Director` és agregació, encara que podria ser binària. L'enunciat no especifica la navegabilitat d'aquesta relació.

La relació `Escola-Professor` per a modelar el claustre és bidireccional. No es vol saber les escoles per les quals ha passat un professor, per tant, no ha d'haver-hi cap classe associativa ni i que modeli això.

Solució exercici 5



Solució exercici 6



Bibliografia

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* (2a. ed.). Addison-Wesley Professional. ISBN: 978-0135956977.

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

Phillips, D. (2018). *Python 3 Object-Oriented Programming* (3a. 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#* (3a. 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* (5a. ed.). Addison-Wesley Professional. ISBN: 978-0135182130.