Associació i herència

David García Solórzano

PID_00235248

UOC
Universitat Oberta de Catalunya

Índex

Introducció

Fins ara hem vist una classe –formada pels seus atributs i mètodes– treballar de manera aïllada. No obstant, en un exemple del mòdul «Encapsulació i extensió» hem vist que una classe podia tenir com a atribut un objecte que pertanyia a una altra classe. Aquest era el cas de la classe *SortedArray*, la qual tenia un atribut que era un *array* d'objectes de la classe *Object*. Això és, clarament, una relació entre classes: la classe A (és a dir, *SortedArray*) conté/utilitza la classe B (és a dir, *Object*).

És obvi que una única classe no pot solucionar per si sola un problema i que, per tant, haurà de col·laborar amb altres classes per a resoldre-ho. En els mòduls que queden veurem les diferents relacions (és a dir, maneres de col·laborar) que es poden donar entre les classes d'un programa seguint el paradigma de la programació orientada a objectes.

A més, veurem com es representen aquestes relacions mitjançant el llenguatge de modelització de sistemes de programari, UML (*Unified Modeling Language*).

En aquest punt cal destacar que, en veritat, són els objectes els que col·laboren entre si per a resoldre un problema i que cada objecte pertany a una classe. Si som rigorosos, hem de distingir entre dos tipus de relacions:

1) les que són entre instàncies i

2) les que són entre classes.

Les relacions entre instàncies són, com veurem en aquest mòdul, anomenades associacions i n'hi pot haver, principalment, de quatre tipus:

1) associació binària,

2) associació d'agregació,

3) associació de composició i

4) associació reflexiva.

Per la seva banda, la relació entre classes per excel·lència en la POO és la generalització/especialització o, com es coneix normalment, l'herència. En parlarem en aquest mòdul i aprendrem, entre altres coses, el concepte d'herència simple i múltiple, com també la característica coneguda com a polimorfisme, que és una conseqüència de l'herència i a la vegada una peça clau de la POO.

Objectius

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

1. Entendre què és una associació (o relació entre instàncies) i distingir entre els quatre tipus més importants: binària, d'agregació, de composició i reflexiva.

2. Conèixer la relació de generalització/especialització (més coneguda com a herència) que es pot donar entre classes.

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

1. Associacions: relació entre instàncies

La relació entre instàncies és coneguda amb el nom d'**associació**. Podem distingir principalment quatre tipus d'associacions:

- Associació binària
- Associació d'agregació
- Associació de composició
- Associació reflexiva

Veurem aquests quatre tipus a continuació.

1.1. Associació binària

Una associació expressa la relació que hi ha entre dues instàncies.

En el cas de l'associació binària (o simplement associació), les dues instàncies relacionades entre si existeixen de forma independent.

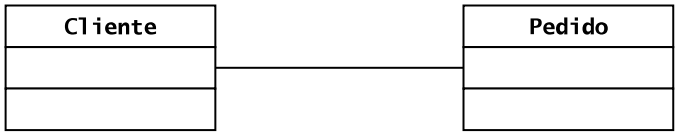
Per tant, la creació o destrucció d'una d'aquestes implica únicament la creació o destrucció de la relació que hi ha entre elles, però mai no significa la creació o destrucció de l'altra instància. Dit d'una altra manera, les instàncies d'una classe no depenen de l'existència de les de l'altra classe.

Parlant amb propietat, es diu que no hi ha una relació forta entre les dues instàncies. L'associació binària respon a una relació que indica que l'objecte de la classe A fa servir un objecte de la classe B i pot ser que viceversa també.

Exemple

Per exemple, un client (és a dir, una instància/objecte de la classe *Cliente*) pot tenir diverses comandes de compra o cap (és a dir, instàncies/objectes de la classe *Pedido*). Si es destrueix el client, no vol dir que en destruïm les comandes, ja que ens pot servir per a fer el càlcul del nombre de comandes sol·licitades anualment. Més clar és el cas contrari, si s'elimina una comanda d'un client no significa que aquest client deixi d'existir. És més, si no hi ha cap demanda d'un client, pot existir aquest client, donem-li temps perquè pugui fer una comanda!

En UML, l'associació binària es representa amb una línia que uneix les dues classes.



Nota

Per facilitar la comprensió i reduir l'extensió dels apunts, s'han eliminat dels diagrames de classes els atributs i mètodes de les classes.

1.2. Associació d'agregació

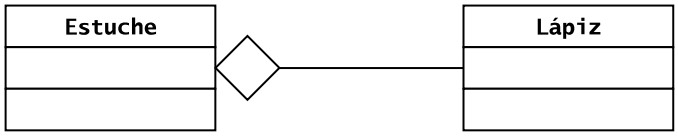
Podem definir l'associació d'agregació com aquella que es dóna quan una instància d'una classe (anomenada *component*) és part d'una instància d'una altra classe (anomenada *compost*).

Aquest tipus d'associació també es coneix com a **composició dèbil**. S'assumeix una subordinació conceptual del tipus «tot/part» o bé «té un». Així doncs, es tracta d'un cas particular de l'associació binària en que hi ha una certa relació d'acoblament, ja sigui física o lògica, entre les instàncies.

Una de les característiques d'aquesta associació és que la destrucció del compost no significa la destrucció dels components, ni viceversa.

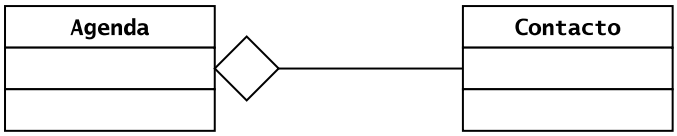
Exemple 1

Un estoig escolar està format per llapis. Si l'estoig es trenca/destrueix, no vol dir que els llapis es trenquin/destrueixin amb ell. El mateix passa a l'inrevés: si un llapis es perd o es trenca (es destrueix), no vol dir que l'estoig també es trenqui amb ell.



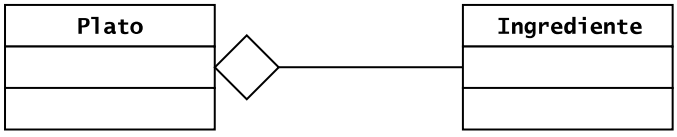
Exemple 2

Una agenda està formada per contactes. Si l'agenda es destrueix, els contactes existeixen com a tals, de fet podrien estar en una altra agenda. Si un contacte és eliminat de l'agenda, l'agenda continua existint.



Exemple 3

Les instàncies de la classe *Ingrediente* formen part d'una instància o objecte de la classe *Plato*. Si eliminem un plat, no eliminem els ingredients (ja que poden formar part d'altres plats). Si eliminem un ingredient, el plat es diu igual però queda modificat.



Com hem pogut veure, en UML, l'associació d'agregació es representa amb un rombe blanc a la part de l'objecte agregador (és a dir, el compost, el que representa el tot).

1.3. Associació de composició

Podem definir l'associació de composició com un cas particular de l'associació d'agregació en què la vida/existència de la instància de la classe continguda (anomenada *component*) ha de coincidir amb la de la instància de la classe contenidora (anomenada *compost*). O, dit d'una altra manera, la destrucció de l'objecte compost comporta la destrucció dels seus components.

A més a més, en el cas de la composició, cada component només pot estar present en un únic compost.

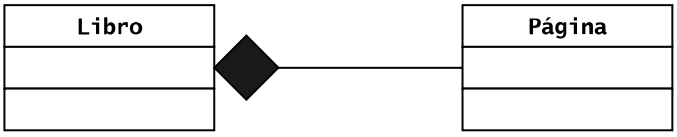
Per tot el que hem esmentat, aquest tipus d'associació és també coneguda com a **composició forta**.

Exemple 1

Un exemple molt visual i tangible d'associació de composició és el cas de la mà i la seva relació amb els dits. Si amputem la mà, implícitament eliminem els dits que la componen. Però si traiem un dit de la mà, no amputem tota la mà!

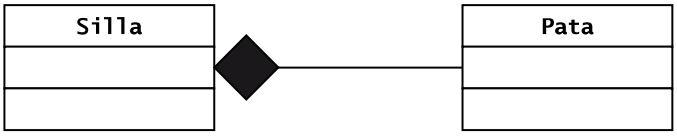
Exemple 2

Un llibre està compost per pàgines. La pàgina 1 d'un exemplar del Quixot només té sentit en aquest llibre i no en un altre. Per tant, la destrucció del llibre implicaria la destrucció de les seves pàgines. En canvi, si s'arrenca una pàgina i aquesta desapareix, el llibre seguiria existint, encara que minvat.



Exemple 3

Una cadira està composta/formada per potes. Si destruïm la cadira, en destruïm les potes implícitament. Si traiem una pota a la cadira, la cadira continua existint.



Com hem pogut veure, en UML, l'associació de composició es representa amb un rombe negre a la part de l'objecte que representa el tot (és a dir, el compost).

En aquest punt pot ser que us costi veure la diferència entre agregació i composició. El matís rau principalment en el fet que la composició afegeix una restricció de vida/existència en la relació entre instàncies. És a dir, si es destrueix la instància agregadora/composta, han de morir/destruir-se les instàncies agregades/components.

El tipus de relació entre dues instàncies de classes diferents pot canviar segons el context/problema en el qual es trobin.

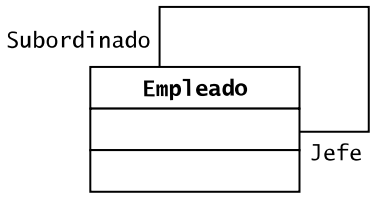
Per això és important entendre el problema/context que treballem en cada moment per a saber si estem davant d'un cas d'agregació o de composició o d'associació binària.

1.4. Associació reflexiva

Podem definir l'associació reflexiva com una associació binària en la qual les dues classes, origen i destinació de la relació, són la mateixa classe.

Exemple

L'exemple típic d'associació reflexiva és el d'un empleat que és cap d'un altre empleat. El cap no deixa de ser un empleat.



Com hem vist, en UML, l'associació reflexiva es representa amb una línia que surt i entra a la mateixa classe. Per poder identificar correctament els rols dels dos extrems de l'associació, cal definir-los amb un text aclaridor en cada un dels extrems.

1.5. Propietats de les associacions

Les associacions tenen bàsicament tres propietats:

- Cardinalitat
- Navegabilitat
- Rol

A continuació veurem cadascuna d'elles.

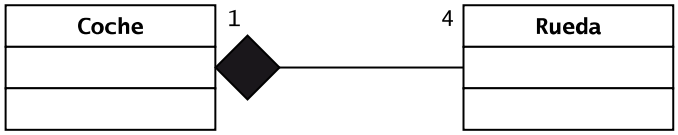
1.5.1. Cardinalitat

Atès que una associació entre dues classes representa en realitat una relació entre instàncies d'aquestes dues classes, hem d'indicar quantes instàncies/objectes de cada classe poden estar involucrats com a mínim o com a màxim en la relació. Aquest nombre d'instàncies/objectes és la cardinalitat de l'associació.

La cardinalitat (també coneguda com a multiplicitat) és el nombre d'instàncies d'una classe amb el qual es pot relacionar una instància de l'altra classe de la relació.

La manera d'indicar la cardinalitat és mitjançant modificadors que s'afegeixen sobre o sota la línia que representa la relació. Hi ha diferents notacions per als modificadors de cardinalitat:

1) Nombre exacte: indica que una instància de la classe A ha d'estar relacionada exactament amb el nombre indicat d'instàncies de la classe B.

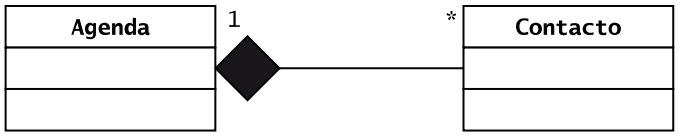


En aquest cas, un objecte de tipus *Coche* es relaciona amb quatre objectes de tipus *Rueda* i un objecte de tipus *Rueda* es relaciona amb un únic objecte de tipus *Coche*. En aquest cas a més, es tracta d'una composició, ja que la destrucció de l'objecte de tipus *Coche* ha de significar la destrucció dels objectes *Rueda*, perquè per ells sols no tenen sentit.

2) Rang de valors: permet indicar el nombre mínim i màxim d'instàncies de la classe A amb les quals ha d'estar relacionada una instància de la classe B.

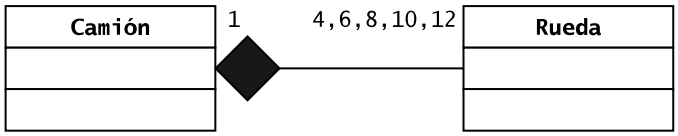
En aquest cas, una microempresa, segons la Unió Europea, és aquella que, entre altres criteris, té menys de deu treballadors. Posem com a mínim 1 perquè entenem que no hi ha cap empresa sense cap treballador (encara que sigui el propietari). En canvi, en aquest context, hem decidit que un treballador només pot ser en una microempresa. En un altre context, aquesta cardinalitat podria ser diferent.

3) Molts: hi ha la possibilitat d'utilitzar el símbol * (asterisc) per a indicar que el nombre d'instàncies és «molts» o «infinit». Dit d'una altra manera, amb l'asterisc diem que la instància de la classe A pot estar relacionada amb qualsevol nombre d'instàncies de la classe B (o fins i tot amb cap) i no en coneixem el nombre. Escriure * és equivalent a indicar el rang 0.. *.



En aquest cas, una agenda pot estar buida (és a dir, zero contactes) o tenir molts contactes (no en sabem el límit en aquest context), mentre que un contacte només pot estar en una agenda. A diferència d'un exemple anterior, ara hem decidit que entre les instàncies de les classes *Agenda* i el *Contacto* hi hagi una associació de composició.

4) **Rangs no consecutius**: si separem nombres exactes o rangs per comes, es poden fer rangs no consecutius.



Un camió pot tenir 4, 6, 8, 10 o 12 rodes (però no 9, per exemple) i una roda només pot pertànyer a un camió.

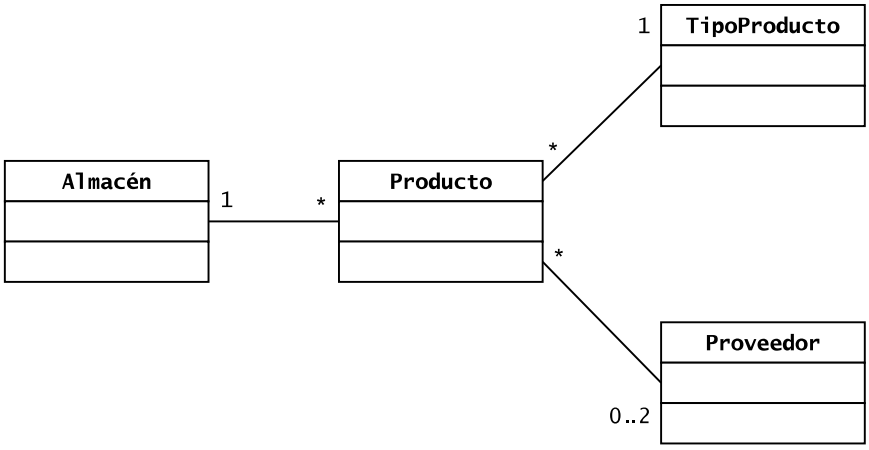
5) **Combinacions de les anteriors**: això ja s'ha vist en els exemples anteriors, hi ha extrems en què hi ha un nombre exacte i en l'altre un asterisc, altres en què hi ha un rang en un extrem i un nombre en l'altre, altres casos en què a tots dos extrems hi ha un asterisc, etc.

Reflexió

Si ens parem a pensar, la cardinalitat, en el cas de l'associació de composició, no té sentit indicar-la a l'extrem de la classe composta, ja que sempre serà 1.

Exemple

A partir del següent diagrama de classes UML, sabem com s'organitzen els magatzems de la nostra botiga.



Sabem que un producte (és a dir, l'objecte concret) només pot ser en un magatzem, com és lògic. Una ampolla concreta d'aigua no pot ser en dos magatzems a la vegada, no té el poder de l'omnipresència.

Així mateix, un producte només pot ser d'un tipus: o és un refresc o és un *snack*, però no tots dos alhora. Això sí, un tipus de producte (p. ex. un refresc) pot associar-se a diferents productes. Per exemple, el tipus de producte *refresc* és l'adequat tant per a l'*aigua*, com per a la *llimonada* i la *cervesa*.

Per la cardinalitat del diagrama veiem que un producte ens el poden facilitar entre 0 i 2 proveïdors com a màxim. El zero ens diu que potser per a algun producte no tenim proveïdor en un instant determinat. Per la seva banda, el número 2 ens diu que, com a màxim, la nostra empresa treballarà amb dos proveïdors per a un producte concret.

Si pensem en instàncies podríem inventar-nos les següents, ja que no sabem quins atributs i mètodes té cada classe:

- Objectes/instàncies de la classe *Almacen*: "Magatzem d'Ateca", "Magatzem de Barcelona", "Magatzem de Madrid", etc.
- Objectes/instàncies de la classe *Producto*: "Cervesa Marca X de 33 cl.", "Cervesa Marca X d'1 l.", "Cervesa Marca Y d'1 l.", "Aigua Marca Z de 33 cl.", "Patates fregides Marca M de 130 g. ", etc.
- Objectes/instàncies de la classe *TipoProducto*: "Refresc", "Snack", "Detergent", etc.
- Objectes/instàncies de la classe *Proveedor*: "Proveïdor 1", "Proveïdor 2", etc.

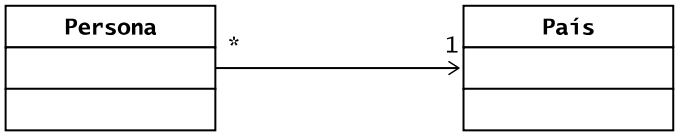
Per acabar, veiem que les instàncies de les classes es relacionen entre si mitjançant associacions binàries. El raonament és lògic: la destrucció d'un magatzem no comporta la destrucció d'un producte ni viceversa; si deixen de fabricar un producte –per exemple, "Cervesa Marca X de 33 cl."–, aquest fet no implica la desaparició d'un tipus de producte –en aquest cas, "Refresc"–; la desaparició d'un tipus de producte no significa la destrucció d'un producte: potser s'ha decidit canviar el tipus de producte per un altre; el tancament d'una empresa proveïdora no vol dir que el producte desaparegui, ni viceversa.

Així doncs, en aquest cas, no hi ha associacions d'agregació ni de composició. Algú de vosaltres podria pensar que un magatzem «està format» per productes i, per tant, entre les instàncies d'aquestes dues classes hi ha una associació d'agregació o composició. Doncs no, cal anar amb compte amb l'ús d'«està format» per a detectar tipus d'associacions. Un magatzem guarda productes, però el magatzem no està fet (és a dir, format) o compost per productes. En tot cas, el magatzem estarà fet/compost/format de maons, pilars, parets, sales, etc., és a dir, d'un altre tipus d'objectes diferents dels productes.

1.5.2. Navegabilitat

Una altra propietat que podem expressar en una relació dins d'un diagrama UML és la seva navegalitat o direccionalitat, és a dir, quina de les dues instàncies coneix l'altra. La navegabilitat només té sentit en associacions binàries i reflexives. Hi ha dues opcions de navegabilitat:

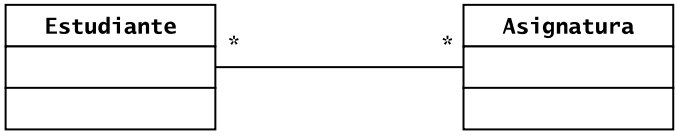
1) Unidireccional: només una de les dues instàncies té constància de l'existència de l'altra.



En aquest cas, una persona sap que viu en un país (necessita accedir als atributs i mètodes de *País*), però no ens cal tenir emmagatzemades quines persones hi viuen, és a dir, la instància del *País* no ha d'interactuar amb els objectes de tipus *Persona*.

Com es pot veure, la navegabilitat unidireccional es representa amb una fletxa que va de la classe que té constància a l'altra.

2) Bidireccional: les dues instàncies tenen constància de l'existència de l'altra.



En aquest cas, ens interessa saber, per a cada assignatura, la llista d'estudiants matriculats i, de cada estudiant, la llista d'assignatures matriculades.

La navegabilitat bidireccional es representa amb una simple línia, sense fletxes. Aquest és el tipus de navegabilitat més freqüent.

Representació de la bidireccionalitat

En alguns diagrames de classes veureu que la navegabilitat bidireccional es representa dibuixant fletxes a banda i banda de la línia. Això és correcte, encara que menys habitual.

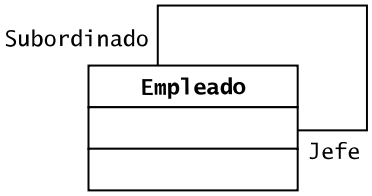
La navegabilitat, com la cardinalitat, depèn del context/problema que tractem en cada moment.

1.5.3. Rol

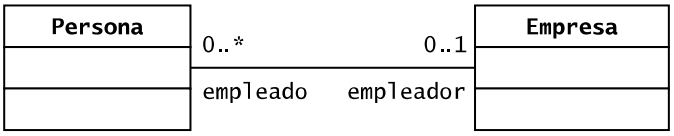
El modificador de tipus rol és una mica diferent dels dos anteriors. A diferència de la cardinalitat i la navegabilitat, no modifiquen el comportament de les associacions, sinó que afegeixen informació contextual que ha d'ajudar el lector a comprendre la relació.

El modificador de rol no és més que una etiqueta que es posa a cada costat de l'associació per a anomenar les classes participants d'una manera diferent o, dit en altres paraules, per a indicar el rol de cada classe. Si la relació s'entén sense posar els rols, no cal posar-los.

Havíem vist el modificador de rol en l'associació reflexiva:



En aquest exemple, els rols o etiquetes són «subordinado» i «jefe». Però, altres exemples podrien ser:



1.6. Traduint a codi...

Hem vist els quatre tipus d'associacions a nivell teòric, però, en la pràctica, com s'implementa cada tipus?

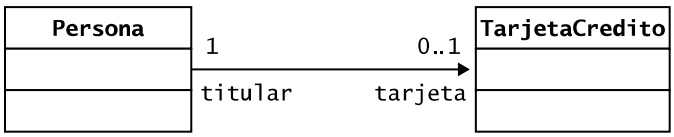
Les **associacions binàries, reflexives i d'agregació** es poden veure com a similars. En el cas de les binàries i reflexives és evident ja que aquestes últimes són un cas particular de les binàries en què les classes origen i destinació són la mateixa.

En el cas de les binàries i d'agregació només hi ha una diferenciació conceptual: l'agregació ens diu que hi ha una classe dominant respecte a una altra, mentre que en les associacions binàries no.

Per aquest motiu, els tres tipus d'associacions es codifiquen de la mateixa manera. Per codificar-les, es passa l'objecte de la classe que cal fer servir (en el cas de la binària) o component (en el cas de l'agregació) per paràmetre en un mètode de la classe que la utilitza (en el cas de la binària) o de la classe composta (en el cas de l'agregació). És a dir, la classe que cal fer servir/component no s'instancia en la classe que la utilitza o compon.

Exemple d'associació binària

Davant el diagrama de classes següent:



Veiem que, en aquest context determinat, una persona fa servir zero o una targeta i que una targeta només pot ser utilitzada per una persona que és titular. El codi Java d'aquest exemple és:

```
public class Persona{
   private String nombre;
   private TarjetaCredito tarjeta;

   public Persona(){

      nombre = "fulanito";
      tarjeta = null;
   }

   public void addTarjetaCredito(TarjetaCredito tarjetaNueva)
   {
      tarjeta = tarjetaNueva;
   }
}
```

D'aquesta manera, si eliminem l'objecte de tipus *Persona*, no eliminaríem la targeta associada, ja que la targeta és creada fora de la classe *Persona*.

En aquest exemple apliquem el concepte de navegabilitat, ja que és la persona qui coneix l'existència de la targeta i no a l'inrevés. Si es tractés d'una relació bidireccional, hauríem de tenir una manera semblant de passar l'objecte *Persona* a l'objecte de la classe *TarjetaCredito*. En tal cas, la classe *TarjetaCredito* tindria un atribut anomenat *titular* de tipus *Persona*.

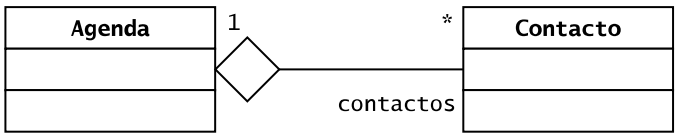
Fixeu-vos com en la classe *Persona* hem afegit un atribut anomenat *tarjeta* de tipus *TarjetaCredito* que implementa l'associació. Hem fet coincidir el nom de l'atribut amb el rol de la relació. Aquesta és una pràctica molt habitual si és que hi ha un rol. Fins i tot sovint, en el diagrama de classes, es posa al costat del rol el símbol de visibilitat de l'atribut (és a dir, +, -, #, ~).

Vegeu també

Aquest exemple el teniu codificat en l'exemple 401 de la col·lecció d'exemples de l'assignatura.

Exemple d'associació d'agregació

Davant l'exemple d'agregació de l'agenda i els contactes:



El codi en Java del diagrama de classes anterior és:

```
public class Agenda{
   private   Contacto [] contactos;
   private int posicionLibre;

   public Agenda(int numContactos){
      contactos = new Contacto[numContactos];
      posicionLibre = 0;
   }
   public boolean addContacto(Contacto c){
      boolean ok = false;
       int numContactos = contactos.length;

       if(posicionLibre<numContactos){
         contactos[posicionLibre] = c;
         posicionLibre++;
         ok = true;
         }
      return ok;
   }
}
```

D'aquesta manera, si eliminem l'objecte agenda, no eliminaríem els contactes, ja que els objectes de la classe *Contacto* són creats en una altra part del programa.

Vegeu també

Aquest exemple el teniu codificat en l'exemple 402 de la col·lecció d'exemples de l'assignatura.

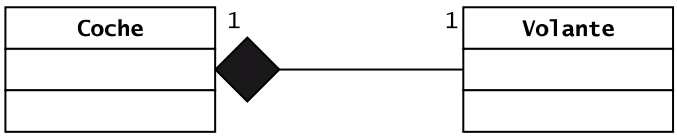
ArrayList

La manera ideal de codificar el llistat de contactes és mitjançant una col·lecció, com pot ser un *ArrayList*. Hem fet servir un *array* per a simplificar i no afegir nous conceptes propis del llenguatge Java.

Per la seva banda, l'**associació de composició** es codifica instanciant l'objecte component dins de l'objecte compost, normalment en el constructor. Això és així ja que l'associació de composició ens indica que la vida de l'objecte component depèn de la vida de l'objecte compost.

Primer exemple d'associació de composició

Davant l'exemple de composició del cotxe i el volant:



El codi en Java és:

Vegeu també

Aquest exemple el teniu codificat en l'exemple 403 de la col·lecció d'exemples de l'assignatura.

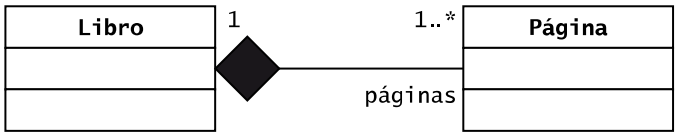
```
public class Coche{
   private Volante vol;

   public Coche(){
      vol = new Volante();
   }
}
```

Veiem que en el cas del *Coche*, aquest té un únic volant. Aquest volant es crea quan es crea el cotxe. Així doncs, quan l'objecte *Coche* és destruït, es destrueix amb ell l'objecte *Volante*.

Segon exemple d'associació de composició

Davant l'exemple de composició del llibre i les pàgines:



El codi en Java és:

```
public class Libro{
   private Pagina [] paginas;

   public Libro(int numPaginas){
      paginas = new Pagina[numPaginas];
   }
}
```

Veiem que en el cas del *Libro*, aquest té un *array* (una col·lecció) de pàgines. És a dir, no només té un objecte *Pagina* sinó tants com indiqui *numPaginas*. Gràcies a aquesta implementació basada en una associació de composició, quan l'objecte llibre és destruït, es destrueixen amb ell les pàgines que la inclouen.

Vegeu també

Aquest exemple el teniu codificat en l'exemple 404 de la col·lecció d'exemples de l'assignatura.

A partir dels exemples anteriors, ja us deveu haver adonat del següent:

La implementació ha de tenir en compte les propietats de les associacions, és a dir, la cardinalitat, la navegabilitat i els rols.

Com hem vist, en el cas de la cardinalitat, cal tenir en la classe coneixedora una referència a objectes de l'altra classe. Atès que hi ha diferents tipus de cardinalitat (és a dir, nombre exacte, rang de valors, etc.), són necessaris diferents tipus d'implementació per donar resposta a cada tipus de cardinalitat. Així doncs, podem tenir en la classe A un atribut que sigui del tipus de la classe B, o fins i tot un atribut que sigui una col·lecció d'objectes de la classe B. La implementació de la col·lecció pot ser diversa, ja que pot ser des d'un simple *array*, fins a una llista, una pila, o classes específiques del llenguatge de pro-

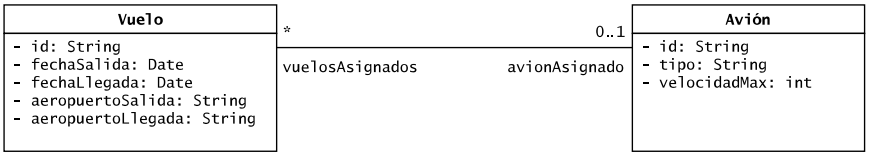
gramació que implementen diferents estructures de dades. En el cas de Java, podria ser un atribut de tipus *Set* (p. ex. *HashSet*), de tipus *List* (p. ex. *ArrayList*, *LinkedList*, etc.) o de tipus *Map* (p. ex. *HashMap*), etc.

Pel que fa a la navegabilitat, com ja sabem, denota la consciència, per part d'una classe, de la pròpia relació. En termes d'implementació, la navegabilitat ens indica en quina de les classes hem de definir l'atribut del tipus de l'altra classe. Serà en la classe coneixedora (és a dir, de la qual surt la fletxa), en el cas de les relacions unidireccionals, i en les dues classes, en el cas de les relacions bidireccionals.

Per acabar, el rol ens ajuda a saber com ha anomenat el programador l'atribut que farem servir en la classe coneixedora. Tot i que els atributs que utilitzem per a definir explícitament la navegabilitat poden tenir el nom que ens sembli més adequat, és recomanable fer servir el rol emprat en el diagrama de classes UML, si és que existeix.

Exemple resum de codificació d'associacions

Imaginem que modelem la relació entre els vols d'una companyia i els avions que té, és a dir, resolem quins avions operen quins vols. El diagrama de classes UML podria ser el següent (s'han omès els mètodes):



Aquest exemple inclou les tres propietats de les associacions: cardinalitat, navegabilitat i rols.

Gràcies a la cardinalitat, sabem que un vol pot no tenir assignat cap avió o, com a molt, un, mentre que un avió pot operar diferents vols o cap.

La navegabilitat ens diu que es tracta d'una associació binària bidireccional, és a dir, el vol i l'avió saben l'un de l'altre.

Per acabar, els rols ens indiquen (o ens recomanen) com podem anomenar l'objecte de l'altra classe en la classe coneixedora. Així doncs, la classe *Vuelo* tindrà un atribut de la classe *Avion* anomenat *avionAsignado* i la classe *Avion* tindrà un atribut –que per la cardinalitat serà una col·lecció, per exemple un *ArrayList* de Java– de la classe *Vuelo* anomenat *vuelosAsignados*. Fixeu-vos que en el diagrama de classes UML, la classe *Vuelo* no inclou un atribut que faci la mateixa funcionalitat que *avionAsignado* ni que es digui igual, ja que s'encavalcaria amb el rol. El mateix passa amb la classe *Avion*, que no té un atribut anomenat *vuelosAsignados*. En aquests casos, ja s'entén que el rol serà un atribut en la classe coneixedora i, per tant, no es posa en la classe perquè seria duplicar informació. Així mateix, hi ha persones que posen davant els rols el símbol corresponent amb el nivell d'accés (és a dir, +, -, #, ~). Normalment no s'hi posa, ja que es dedueix que en ser un atribut, ha de ser privat.

Així doncs, el codi en Java és:

Vuelo.java

```
import java.util.Date;
public class Vuelo{
   String id = null, aeropuertoSalida = null, aeropuertoLlegada = null;
    Date fechaSalida = null, fechaLlegada = null;
    Avion avionAsignado = null;

    public Vuelo(){}
```

Vegeu també

Aquest exemple el teniu codificat en l'exemple 405 de la col·lecció d'exemples de l'assignatura.

Import

La paraula clau *import* ens permet importar classes ja fetes d'altres paquets. En aquest cas farem servir la classe *Date* que hi ha al paquet *java.util*.

Treballarem el concepte *paquet* a l'assignatura quan fem exercicis de codificació.

```
   public   Vuelo(String idNuevo, String aeropuertoS, String
   aeropuertoL, Date fechaS, Date fechaLl, Avion avionA){
      id = idNuevo;
      aeropuertoSalida = aeropuertoS;
      aeropuertoLlegada = aeropuertoL;
      fechaSalida = fechaS;
      fechaLlegada = fechaL;
      avionAsignado = avionA;
   }
   //TODO: Més mètodes, entre ells, setter's i getter's
}
```

Avion.java

```
import java.util.ArrayList;
public class Avion{
   String id = null, tipo = null;
   int velocidadMax = 0;
   ArrayList<Vuelo> vuelosAsignados = null;

   public Avion(){}

   public Avion(String idNuevo, String tipoNuevo,
   int velocidadMaxNueva){
      id = idNuevo;
      tipo = tipoNuevo;
      velocidadMax = velocidadMaxNueva;
      vuelosAsignados = new ArrayList<Vuelo>();
   }

   public void addVuelo(Vuelo v){
      vuelosAsignados.add(v);
   }
   //TOT: Més métodes, entre ells, eliminar un vol i
   //els setter's i getter's dels atributs de la classe
}
```

2. Herència: relació de generalització/especialització o entre classes

2.1. Concepte

La relació de generalització/especialització equival en el paradigma de la programació orientada a objectes (POO) al mecanisme d'herència. Per això, en el dia a dia se sol fer servir indistintament herència com generalització/especialització, encara que és més freqüent el terme herència.

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

D'una manera més pràctica, podem definir *herència* com un mecanisme que permet definir una classe nova a partir d'una anterior descrivint-ne les diferències.

A diferència de l'associació, que era una relació entre instàncies, la generalització/especialització és una relació entre classes.

Exemple

La classe *Vehiculo* generalitza el comportament de les classes *Camion*, *Motocicleta*, *Autobus*, etc. Tot vehicle té una marxa, una velocitat, un nombre de rodes, un mètode per a accelerar, un altre mètode per a frenar, etc. Aquests són elements comuns que tenen tots els vehicles.

D'altra banda, una *Motocicleta* especialitza el comportament general d'un *Vehiculo*, és a dir, una *Motocicleta* és un *Vehiculo* que es comporta d'una manera concreta i té diferències amb la resta de *Vehiculos* –p. ex., no té finestres i requereix l'ús d'un casc.

Per tant, la classe *Vehiculo* és la classe pare (o base o superclasse) i la classe *Motocicleta* és la classe filla (o subclasse o classe derivada).

És important tenir clar que una classe filla (o subclasse) hereta tots els atributs i mètodes declarats com a públics o protegits en la classe pare.

Això vol dir que la classe filla no pot ni accedir directament als atributs privats declarats en la classe pare ni fer servir els mètodes privats declarats en la classe pare, ja que en tots dos casos no han estat heretats.

2.2. Herència simple

El cas més estès d'herència és el que s'anomena *herència simple*. És el més utilitzat perquè és el tipus d'herència permès per tots els llenguatges de programació orientats a objectes.

Exemple

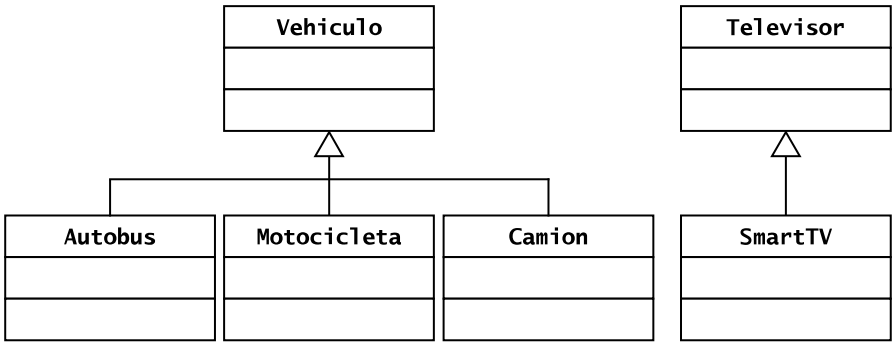
La classe *Televisor* generalitza el comportament de la classe *SmartTV*. Tot televisor té una cadena actual, un volum, un mètode per a pujar i baixar el volum, un altre per a canviar de canal, etc. Aquests són elements comuns que tenen tots els televisors.

D'altra banda, una *SmartTV* especialitza el comportament general d'un *Televisor*. A més de fer tot el que fa la classe *Televisor*, afegeix un comportament nou, com ara accedir a internet.

Quan parlem des del paradigma de POO, diem que **la classe filla hereta de la classe pare**. És a dir, *Motocicleta* hereta de *Vehiculo* i *SmartTV* ho fa de *Televisor*.

Cal adonar-se que aquesta relació no té cardinalitat, ja que no estem parlant d'una relació entre instàncies, sinó que simplement diem que una classe (*filla*) conté i incrementa/modifica les propietats d'una altra (és a dir, la classe *pare*).

En UML, la relació de generalització/especialització (o herència) es representa mitjançant un triangle blanc a la classe pare.



2.3. Herència múltiple

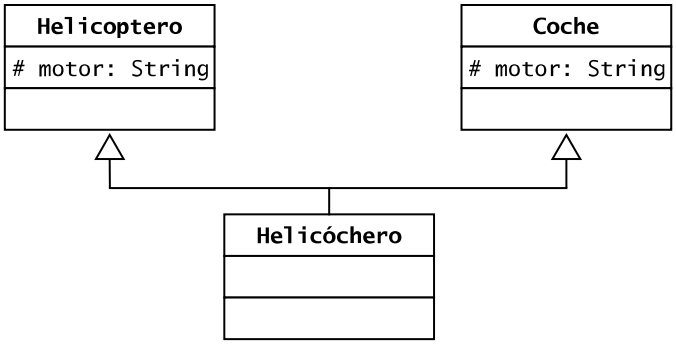
També hi ha la possibilitat que una classe hereti de més d'una classe pare.

Exemple

Imaginem que volem inventar un nou vehicle anomenat *Helicóchero* que té les virtuts de l'helicòpter i del cotxe. És fàcil pensar que l'*Helicóchero* heretarà atributs i mètodes de les classes *Helicoptero* i *Coche*, i el diagrama de classes UML quedarà de la següent manera:

Representació dels atributs heretats en el diagrama de classes

Fixeu-vos que els atributs heretats no es tornen a indicar a la classe filla. En l'exemple, no s'han posat dos atributs motors (un d'*Helicóptero* i un altre de *Coche*) dins de la classe *Helicóchero* perquè ja se sobreentén que els té.



Representació dels mètodes heretats en el diagrama de classes

A la classe filla només es posen aquells mètodes heretats que han estat sobreescrits en la classe filla. Així doncs, si no apareix un mètode heretat en la classe filla, s'entén que es comporta de la mateixa manera que a la classe pare. El concepte de *sobreescriptura* el veurem al subapartat 2.7 d'aquest mòdul.

Com es pot veure, l'exemple anterior té sentit conceptualment i és possible representar-lo mitjançant UML.

El problema rau en el fet que no tots els llenguatges de programació permeten l'herència múltiple –és a dir, heretar de més d'una classe pare.

Per exemple, ni Java ni C# ni PHP permeten l'herència múltiple, mentre que C++ i Python sí.

Així doncs, en el moment de passar del disseny a la implementació, haurem d'adaptar tan bé com puguem al llenguatge de programació escollit allò que haguem explicitat en el diagrama de classes. És a dir, si conceptualment té sentit que hi hagi una herència múltiple, el nostre diagrama de classes ho ha de contemplar així, ja que el diagrama de classes UML ha de ser independent del llenguatge de programació. Per tant, els canvis que siguin necessaris per a l'ús d'un llenguatge de programació concret, els farem en el moment de la implementació.

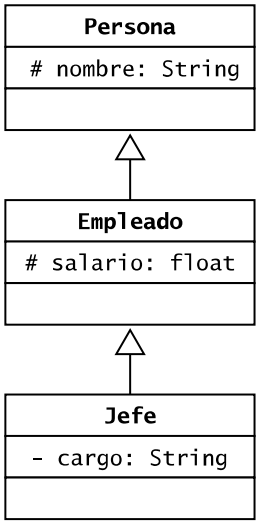
A més, l'herència múltiple implica un problema que han de resoldre els llenguatges que la permeten.

Exemple

Si tornem a l'exemple de l'*Helicóchero*, veiem que les dues classes pare tenen un atribut anomenat *motor* que és *String*. Doncs bé, la classe filla *Helicóchero* 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. Aquest és un problema típic de l'herència múltiple.

2.4. Transitivitat

Una classe B pot heretar d'una classe pare A i, al seu torn, una classe C pot heretar de la classe B. El diagrama de classes UML seria el següent:



En aquest cas *Empleado* tindrà l'atribut *salario* i l'atribut heretat *nombre*. Per la seva banda, *Jefe* tindrà tres atributs: el seu propi, *cargo*, més els heretats d'*Empleado* i *Persona*, que són *salario* i *nombre*, respectivament (s'hereten perquè la seva visibilitat és *protected*). És a dir, l'herència no es limita a un nivell de profunditat en la jerarquia (és a dir, primer grau de parentiu), sinó que l'herència és, per dir-ho d'alguna manera, una relació transitiva: sempre que un element es relaciona amb un altre mitjançant el mecanisme d'herència i aquest últim amb un tercer també per mitjà de l'herència, llavors el primer es relaciona amb el tercer.

2.5. Traduint a codi...

Hi ha diferents sintaxis per a definir l'herència entre classes. Alguns llenguatges de programació comparteixen sintaxis. Per exemple, en Java i en PHP es fa servir la paraula reservada ***extends***:

```
class Hija extends Padre{
}
```

En C# y C++ s'utilitzen els **dos punts** (:). Així doncs:

```
class Hija : Padre{
}
```

En C++, l'anterior declaració seria correcta si la classe filla només heretés d'una classe pare (és a dir, herència simple), però en permetre herència múltiple (és a dir, més d'una classe pare), les diferents classes se separen amb comes després dels dos punts.

```
class Hija : public Padre1, public Padre2{
}
```

Com es pot veure, en C++, a més, s'ha d'indicar si l'herència és pública, protegida o privada (per defecte, si no s'indica res). A més, C++ divideix una classe en dos fitxers: un amb l'extensió *.h* (anomenat *header file*) per a la declaració de la classe, és a dir, la definició dels atributs i encapçalaments/signatures dels mètodes; i un altre amb extensió *.cpp*, que conté la implementació de la classe. Queda a les vostres mans cercar informació sobre aquest tema si és que us interessa.

2.6. El constructor de la classe filla

El mínim que ha d'implementar la classe filla és el seu propi constructor. El codi del constructor d'una subclasse (és a dir, classe filla) ha de comprendre dos passos:

1) La invocació del constructor de la classe pare.

2) La resta d'instruccions pròpies del constructor de la classe filla.

El pas 1 és l'«especial» respecte a la implementació d'un constructor d'una classe que no hereta. La manera de cridar/invocar el constructor de la classe pare varia segons el llenguatge de programació utilitzat.

Vegem el cas de Java, en el qual es fa servir el mètode reservat ***super***:

`Persona.java`

```
public class Persona{
   private String nombre, apellidos;
   private int edad;

   //Constructor

   public Persona (String nombreNuevo, String apellidosNuevos, int edadNueva) {
      nombre = nombreNuevo;
       apellidos = apellidosNuevos;
       edad = edadNueva;
   }

   //Mètodes

   public String getNombre () { return nombre;  }
   public String getApellidos () { return apellidos;  }
   public int getEdad () { return edad;   }
}
```

`Profesor.java`

```
public class Profesor extends Persona {
    //Campos específicos de la clase hija.
    private String idProfesor;

    //Constructores de la subclase

   public Profesor(){
      super("David", "García Solórzano", 32);
      idProfesor = "Unknown";
   }

//Aquí incloem com a paràmetres els del constructor de la classe pare

   public Profesor (String nombre, String apellidos, int edad) {
      super(nombre, apellidos, edad);
       idProfesor = "Unknown";
   }

   public Profesor (String nombre, String apellidos, int edad, String idProfesorNuevo) {
       super(nombre, apellidos, edad);
       idProfesor = idProfesorNuevo;
   }

    //Mètodes específics de la subclasse

    public void setIdProfesor (String idProfesorNuevo) {idProfesor = idProfesorNuevo;   }

    public String getIdProfesor () { return idProfesor;   }

    public void mostrarNombreApellidosYCarnet() {
   //nombre = "David"; Si tractéssim d'accedir directament a un camp privat de la classe pare,
   //donaría un error
   //Si podem accedir a variables d'instància a través dels mètodes d'accés públic de la
   //classe pare
      System.out.println ("Profesor de nombre: " + getNombre() + " " +  getApellidos() + " con
      Id de profesor: " + getIdProfesor() );
   }
}
```

Vegeu també

Aquest exemple el teniu codificat en l'exemple 406 de la col·lecció d'exemples de l'assignatura.

Pel que fa al mètode especial ***super***, hem de tenir present que té com a signatura qualsevol de les signatures dels possibles constructors de la classe pare.

En PHP, en comptes de *super*, es fa servir la sintaxi ***parent::__ construct()***, on *__construct()* funciona com *super*, és a dir, la seva signatura ha de coincidir amb la del constructor de la classe pare (en PHP, una classe només pot tenir un constructor).

C++

C++ és semblant a C#, però com que permet herència múltiple, no s'hi fa servir la paraula reservada *base*, sinó directament el nom de la classe pare/base.

En C# se sol invocar el constructor de la classe pare en la signatura mateixa del constructor de la classe filla mitjançant la paraula reservada ***base***:

```
public class Profesor : Persona{
    //Camps específics de la classe filla
    private String idProfesor;

    public Profesor(String nombre, String apellidos, int edad,
     String idProfesorNuevo):base(nombre,apellidos,edad){
      idProfesor = idProfesorNuevo;
   }
}
```

És important saber que si en el constructor de la classe filla no sol·licitem explícitament el constructor de la classe pare –mitjançant *super* a Java o *base* en C#–, el compilador sol·licita automàticament el constructor per defecte (és a dir, sense arguments) de la classe pare. En aquest cas, si la classe pare no té constructor per defecte, obtindrem un error de compilació.

Per acabar, alguns llenguatges com Java i C#, a més del mètode especial *super* i *base*, respectivament, poden utilitzar la funció ***this*** per a cridar, des d'un constructor, un altre constructor de la mateixa classe.

```
public class Persona{
   private String nombre, apellidos;
   private int edad;

   //Constructor per defecte que utilitza el mètode especial this() per trucar
   // al constructor amb 3 arguments de la pròpia classe Persona
   public Persona () {
      this("David","García Solórzano",32);
   }

   //Constructor amb 3 arguments
   public Persona (String nombreNuevo, String apellidosNuevos, int edadNueva) {
      nombre = nombreNuevo;
      apellidos = apellidosNuevos;
      edad = edadNueva;
   }
}
```

Vegeu també

L'ús de la funció especial *this* també s'inclou en l'exemple 406 de la col·lecció d'exemples de l'assignatura.

Atesa l'anterior implementació de la classe *Persona*, les dues instanciacions següents de la classe *Persona* crearien objectes idèntics pel que fa als valors dels seus atributs i estats, però serien dos objectes independents:

```
Persona persona1 = new Persona();

Persona persona2 = new Persona("David","García Solórzano",32);
```

2.7. Sobreescriptura (*override*)

Com ja hem comentat en el subapartat 2.1, quan una classe filla hereta d'una classe pare, n'hereta els atributs i mètodes de la classe pare que siguin públics (és a dir, *public*) o protegits (és a dir, *protected*). Això vol dir que els podem fer servir en la classe filla com si els haguéssim declarat en la mateixa classe filla.

En aquest punt és important saber que en heretar no podem anul·lar/eliminar res de la classe pare. L'únic que podem fer és no fer servir l'atribut o el mètode heretat que no ens fa falta en la classe filla, però hi serà en la memòria.

Una classe filla pot definir nous atributs i mètodes totalment diferents dels de la classe pare i que, per tant, seran exclusius de la classe filla, és a dir, no existiran en la classe pare. Una particularitat en definir nous mètodes en la classe filla és el que es coneix com a **sobreescriptura de mètodes**.

La sobreescriptura de mètodes consisteix a modificar, en la classe filla, el comportament d'un mètode definit prèviament en la classe pare perquè faci altres tasques, tant si es tracta de tasques similars com si són completament diferents. Per a parlar de sobreescriptura pròpiament dita, la **signatura del mètode en la classe filla ha de ser idèntica de la de la classe pare**, i només pot canviar el nivell d'accés, el qual no pot ser més restrictiu que el del mètode de la classe pare (p. ex. *protected* en la classe pare i *private* en la classe filla). Si canviem la signatura del mètode (és a dir, el nombre o el tipus dels arguments), parlaríem de sobrecàrrega del mètode.

Vegeu també

Reviseu el concepte de *sobrecàrrega* en el mòdul «Objecte i classe».

Vegeu també

Veurem el concepte de *mètode final* en el mòdul «Tipus de classe i interfície».

També cal tenir en compte que les modificacions fetes en la classe filla afecten només la classe filla i les classes que heretin d'ella. En cap cas no afecten la classe pare.

Cal tenir present també que en la classe filla no es poden sobreescriure mètodes que siguin *final* en la classe pare.

En el cas dels mètodes *static*, només poden ser sobreescrits si en la classe filla continuen essent *static*.

Exemple

Imaginem que tenim una classe pare *Autor* i una altra que hereta d'aquesta que es diu *AutorVIP*. La classe filla –és a dir, *AutorVip*– fa referència a aquells autors que són *best-sellers* i que, per tant, tenen un tractament diferent de l'editorial.

Vegeu també

Aquest exemple el teniu codificat en l'exemple 407 de la col·lecció d'exemples de l'assignatura.

La classe pare *Autor* té un mètode públic anomenat *getGananciasPorVenta(): double*, el qual ens retorna la quantitat de diners en euros que ha guanyat un autor pels llibres que s'han venut d'ell. El codi en Java de la classe *Autor* podria ser tan simple com:

```
public class Autor{
   private int numLibros;

   //Aquí vindrien més atributs

   public Autor(){numLibros = 0;}

   public double getGananciasPorVenta(){
      return numLibros * 5.5;
   }

   public void setNumLibros(int numLibrosActual){
```

```
        numLibros = numLibrosActual;
    }

    public int getNumLibros(){
        return numLibros;
    }
    //TODO: Aquí vindrien més mètodes
}
```

És a dir, per cada llibre venut, l'autor guanya 5,5 €. Si la classe *AutorVip* hereta d'*Autor* i no sobreescriu el mètode *getGananciasPorVenta():float*, quan sol·licitem aquesta funció des d'una instància de tipus *AutorVip*, el resultat seria el mateix. Però, l'editorial no vol això, vol que els autors VIP tinguin una fórmula diferent. Per això hem de sobreescriure el mètode:

```
public class AutorVIP extends Autor{
    public AutorVIP(){super();}

    @Override
    public double getGananciasPorVenta(){
        return getNumLibros() * 10.5;
    }
    //TODO: Aquí vindrian més mètodes
}
```

Ara, la fórmula per als objectes de tipus *AutorVIP* serà de 10,5 € per exemplar. Fixeu-vos que la classe *AutorVip* no declara l'atribut *numLibros* ni tampoc els mètodes *getter* i *setter* d'aquest atribut, és a dir, *getNumLibros* i *setNumLibros*. Per què? Perquè els hereta de la classe pare *Autor*. En aquest punt cal adonar-se que com l'atribut *numLibros* està declarat com a privat en la classe pare, la classe filla *AutorVIP* no pot accedir-hi/consultar-la directament i ho ha de fer amb el mètode públic *getNumLibros()*. Si haguéssim declarat l'atribut *numLibros* com a *public* o *protected*, llavors podríem haver definit el mètode *getGananciasPorVenta()* en la classe *AutorVIP* de la manera següent:

```
@Override
public double getGananciasPorVenta(){
    return numLibros * 10.5;
}
```

Ja que podríem accedir directament a l'atribut sense necessitat de fer servir el mètode *getter*, l'ús del qual seguiria sent vàlid.

Amb el codi anterior:

```
Autor autor1 = new Autor();
AutorVIP autor2 = new AutorVIP();
autor1.setNumLibros(10);
autor2.setNumLibros(11);
autor1.getGananciasPorVenta(); //retornarà el resultat d'aplicar la
//fórmula numLibros * 5.5 = 10 * 5.5 = 55
autor2.getGananciasPorVenta(); //retornarà el resultat d'aplicar la
//fórmula numLibros * 10.5 = 11 * 10.5 = 115.5
```

Fixeu-vos també que en el constructor d'*AutorVIP* hem cridat el mètode especial *super* perquè cridi el constructor d'*Autor*. Si la fórmula que cal aplicar en el mètode *getGananciasPorVenta()* de la classe *AutorVIP* hagués estat (*numLibros * 5.5*) + *8*, hauríem pogut aprofitat la implementació del mètode homònim de la classe pare mitjançant la paraula especial ***super*** de la manera següent:

```
@Override
public double getGananciasPorVenta(){
   double valor = super.getGananciasPorVenta();
   return valor + 8;
}
```

Amb *super.getGananciasPorVenta()* cridem el mètode *getGananciasPorVenta* de la classe pare, és a dir, de la classe *Autor*.

En l'exemple anterior, us deveu haver adonat que apareix *@Override* abans de la signatura del mètode *getGananciasPorVenta* en la classe *AutorVip*. Això és el que es coneix com a **anotació**. No és obligatori posar les anotacions, però sí que són recomanables. Per què? Perquè d'aquesta manera facilitem la feina al compilador i també a nosaltres. En primer lloc, diem al compilador que aquest mètode està sobreescrit (en anglès, *override*) i, per tant, ens podrà avisar si ens equivoquem quan creem un mètode diferent. En segon lloc, en posar l'anotació, avisem un altre programador –i ens ho recordem quan revisem el codi al cap d'un cert temps– que aquest mètode és una sobreescriptura d'un mètode de la classe pare de la qual hereta. D'aquesta manera, mantenir el codi és una tasca més senzilla.

Enllaç d'interès

Hi ha més tipus d'anotacions. Per a saber-ne més coses en Java, consulteu l'enllaç web següent:

https://docs.oracle.com/javase/tutorial/java/annotations/.

Diferencia C++, C# i Java

Hi ha una diferència molt important entre C++ i C# i Java. En C++ i C#, només es pot sobreescriure un mètode de la classe base si ha estat declarat com a virtual i en la classe derivada el mètode és declarat com a *override*. En Java, per defecte, els mètodes són virtuals.

2.8. Polimorfisme

El concepte de polimorfisme és un dels pilars de la POO.

El polimorfisme es pot definir com la característica de la programació orientada a objectes que permet modificar el tipus d'un objecte en temps d'execució basat en una jerarquia d'herència.

D'una manera molt resumida i més senzilla, podem dir que és un mecanisme que permet que un objecte d'una classe es comporti com un objecte de qualsevol de les seves subclasses (o subclasses de les subclasses, i així successivament).

Vegem-ho més clar amb un exemple.

Exemple

Imaginem que tenim les classes *Rectangulo* i *Triangulo* que hereten de la classe *Figura*.

`Figura.java`

```
class Figura{
   protected float base, altura;
   public Figura(float baseNueva, float alturaNueva){
      base = baseNueva;
      altura = alturaNueva;
   }
   public float perimetro(){  return -1;}
   public float area(){  return -1;}
   public String getTexto(){ return "Mi base es "+base+" y mi
   altura es "+altura;}
```

Vegeu també

Aquest exemple el teniu codificat en l'exemple 408 de la col·lecció d'exemples de l'assignatura.

```
}
```

Rectangulo.java

```
class Rectangulo extends Figura{
   public Rectangulo(float base, float altura){super(base,altura);}
   public float perimetro(){  return 2*(base+altura);}
   public float area(){  return base*altura;}
   public boolean isCuadrado(){ return base==altura;}
}
```

Triangulo.java

```
class Triangulo extends Figura{
   public Triangulo (float base, float altura){super(base,altura);}
   public float perimetro(){  return 2*altura + base;}
   public float area(){  return (base*altura)/2;}
}
```

Veiem que tant *Rectangulo* com *Triangulo* sobreescriuen els mètodes *perimetro* i *area* per a ajustar-se a les característiques de totes dues figures. El polimorfisme ens permet fer el següent:

```
Figura fig = new Figura(2,3);
Rectangulo rect = new Rectangulo(2,3);
Triangulo triang = new Triangulo(2,3);
fig.perimetro(); //retorna -1
fig = rect; //assignació polimòrfica
fig.perimetro(); //retorna 10 perquè "fig" es comporta com un
//rectangle
fig = triang;
fig.perimetro(); //retorna 8 perquè "fig" es comporta com un triangle
triang = fig; // error, no es pot perquè la classe base (tipus
//estàtic) de "fig" és Figura, no Triangulo
```

Un dels usos del polimorfisme es mostra en el següent exemple.

Exemple

```
Figura [] figuras = new Figura[10];
for(int i=0;i<10;i++){
   if(i==0){ //si i és igual a zero
      figuras[i] = new Figura(10,10);

   }else if(i%2 == 0){ //si i és un nombre parell, creem un
   //rectangle amb alçada 3 vegades la seva base
      figuras [i] = new Rectangulo(i,i*3);

   }else{ //si i és un nombre imparell, creem un triangle amb
   //altura 4 vegades la seva base
      figuras [i] = new Triangulo(i,i*4);
   }
   figuras[i].getTexto();
}
```

És a dir, declarant un sol *array* podem gestionar, en aquest cas, fins a tres tipus (és a dir, classes) diferents d'objecte: *Figura*, *Rectangulo* i *Triangulo*. A més, com que *Rectangulo* i *Triangulo* són classes derivades de *Figura*, podem cridar el mètode *getTexto()* directament, ja que les tres classes tenen aquest mètode – *Rectangulo* i *Triangulo* perquè l'hereten.

Això ens redueix l'extensió del nostre codi i ajuda en la gestió de les diferents classes, ja que ens facilita la tasca d'escriure codi genèric.

Vegeu també

Aquest exemple el teniu codificat en l'exemple 408 de la col·lecció d'exemples de l'assignatura.

Un altre ús típic del polimorfisme és el següent:

Exemple

Partint de l'exemple anterior, imaginem que ara tenim una quarta classe en la qual tenim el mètode següent:

```
public void pinta(Figura fig){
   float perimetro = fig.perimetro();
   //TODO: Més línies de codi aquí
}
```

Aquest mètode *pinta(Figura fig)* podria rebre com a paràmetre un objecte de tipus *Figura* o *Rectangulo* o *Triangulo*, gràcies a l'herència i el polimorfisme. Així mateix, serà en temps d'execució, no de compilació, quan es decidirà a quina classe –és a dir, *Figura*, *Rectangulo* o *Triangulo*– pertany el mètode *perimetro():float* que s'executa.

Vegeu també

Aquest exemple el teniu codificat en l'exemple 408 de la col·lecció d'exemples de l'assignatura.

Un altre ús típic del polimorfisme en el qual es veurà clarament la seva utilitat és en la creació de codi genèric.

Exemple

Partint de l'exemple anterior, imaginem que en la classe *Figura* canviem el mètode *getTexto* de la manera següent:

Figura.java

```
public String getTexto(){
   return "Soy un objeto de tipo "+getTipo()+" y mi base es
   "+base+" y mi altura es "+altura;
}
```

En cada clase escriurem el mètode *getTipo():String* que retorna el nom de la classe:

Figura.java

```
class Figura{
   //Aquí aniria el codi escrit anteriorment per a la classe Figura
   public String getTipo(){ return "Figura";}
}
```

Rectangulo.java

```
class Rectangulo extends Figura{
//Aquí aniria el codi escrit anteriorment per a la classe Rectangulo
   public String getTipo(){ return "Rectangulo";}
}
```

Triangulo.java

```
class Triangulo extends Triangulo{
//Aquí aniria el codi escrit anteriorment per a la classe Triangulo
   public String getTipo(){ return "Triangulo";}
}
```

Cada vegada que s'executi el mètode *getTexto():String* de la classe *Figura*, s'executarà un mètode *getTipo():String* diferent en funció de si s'ha fet una assignació polimòrfica a l'objecte o no. D'aquesta manera, no cal escriure dos mètodes *getTexto():String* addicionals en les classes *Rectangulo* i *Triangulo*.

Vegeu també

Aquest exemple el teniu codificat en l'exemple 408 de la col·lecció d'exemples de l'assignatura.

En resum, el polimorfisme:

1) Es basa i es veu restringit per l'herència.

2) Implica que un objecte té un ***tipus estàtic*** (és a dir, l'indicat en la declaració) i un ***tipus dinàmic*** (és a dir, el que s'assigna en temps d'execució seguint les regles del polimorfisme).

3) El tipus dinàmic d'un objecte és igual al tipus (= classe) de la declaració o igual a una de les classes derivades de la seva.

4) Sobre un objecte declarat del tipus classe pare no es poden invocar mètodes propis/exclusius d'una classe filla.

Tenint present les idees clau anteriors, tornem a posar un exemple per a acabar d'entendre el polimorfisme.

Exemple

```
Figura fig = new Figura(2,3); //tipus estàtic = Figura i tipus dinàmic = Figura

Rectangulo rect = new Rectangulo(2,3); //tipus estàtic = Rectangulo i tipus dinàmic =
//Rectangulo.
fig = rect; // Assignació polimòrfica, tipus estàtic = Figura i tipus dinàmic = Rectangle,
//el tipus dinàmic ha canviat en temps d'execució; També podria haver-se fet així: fig = new
//Rectangulo(2,3)

fig.perimetro(); // S'està trucant al mètode perimetro de la classe Rectangulo, tipus estàtic
//= Figura i tipus dinàmic = Rectangulo

fig.area(); //OK mètode que està en Figura i Rectangulo i es decidirà en temps d'execució quin

//trucar. En aquest cas, executarà el mètode de la classe Rectangulo
fig.getTexto(); //OK mètode de Figura
fig.isCuadrado(); //error de compilació, aquest mètode no està en Figura
```

Vegeu també

Aquest exemple el teniu codificat en l'exemple 409 de la col·lecció d'exemples de l'assignatura.

2.9. Càsting

El concepte de *càsting* –o simplement *cast* o també conegut com a *narrowing*– està íntimament relacionat amb el polimorfisme i, per tant, també amb l'herència.

Formalment, el procés de *càsting* es pot definir com el procés en el qual el compilador permet fer una conversió d'un objecte polimòrfic a un dels seus possibles tipus dinàmics.

D'una manera més informal diríem que si es vol accedir als mètodes de la classe filla tenint una referència d'una classe pare, cal convertir explícitament la referència de tipus/classe pare a tipus/classe filla. Aquesta conversió és el que s'anomena *càsting*.

Exemple

Així doncs, partint de l'exemple de les figures anteriors:

Vegeu també

Aquest exemple el teniu codificat en l'exemple 410 de la col·lecció d'exemples de l'assignatura.

```
Figura [] figuras = new Figura[10];
for(int i=0;i<10;i++){
   if(i==0){ //si i és igual a zero
      figuras[i] = new Figura(10,10);

   }else if(i%2 == 0){ //si i és un nombre parell
       figuras [i] = new Rectangulo(i,i*3);
       Rectangulo rect = (Rectangulo) figuras[i];
       rect.isCuadrado(); //funciona perquè hem fet el càsting a la
       //línia anterior. Si féssim directament figures [i]
       //.isCuadrado() no funcionaria perquè el mètode isCuadrado no
      //pertany a la classe Figura. Per estalviar una línia es podria
       //haver fet: ((Rectangulo) figures [i]). IsCuadrado (); sense
       //necessitat de crear l'objecte temporal "rect" de tipus
       //Rectangulo.

   }else{ //si i és un nombre senar
       figuras [i] = new Triangulo(i,i*4);
   }

   figuras[i].getTexto();
}
```

En resum, si volem accedir a un mètode d'una classe derivada des d'un objecte declarat com a classe base/pare, hem d'aplicar un *cast* (o *càsting*) a aquest objecte.

El procés de *càsting* té un problema: si la conversió no és possible, falla el programa en temps d'execució llançant una excepció de tipus *ClassCastException* i l'execució del programa queda avortada.

Per a assegurar-nos que no hi haurà errors a l'hora de fer la conversió, haurem d'assegurar-nos de quin tipus dinàmic és l'objecte. Per a això hi ha dues estratègies en Java: mitjançant *instanceof* o el mètode *getClass()*:

Tipus dinàmic en altres llenguatges

Queda a les vostres mans, si us interessa, buscar com es pot saber el tipus dinàmic d'un objecte en altres llenguatges de programació.

1) *instanceof*: és una paraula reservada que ens permet comprovar si el tipus dinàmic de l'objecte és **compatible** amb un tipus/classe determinat.

La manera de fer servir *instanceof* és la següent:

```
B objeto = new B();
if(objeto instanceof A){...}
```

instanceOf

instanceOf també es pot utilitzar amb interfícies. Veurem aquest concepte més endavant en el mòdul «Tipus de classe i interfície».

Un tipus dinàmic B serà compatible amb un tipus A, si B i A són el mateix tipus (= classe) o B és una classe derivada d'A.

2) *getClass()*: mètode que retorna la classe amb la qual ha estat instanciat l'objecte. És a dir, aquest mètode ens retorna el tipus dinàmic de l'objecte.

La manera d'utilitzar *getClass()* és la següent:

```
B objeto = new B();
if(objeto.getClass() == A.class){...}
```

Aquesta comprovació mira si el tipus dinàmic de l'objecte és **exactament** la classe que s'indica a l'altra banda de la comparació. Dit d'una altra manera, l'expressió booleana de la instrucció *if* només serà certa si els tipus són idèntics.

En general es recomana fer servir *instanceof* en comptes del mètode *getClass()*.

Exemple

Seguint amb l'exemple de les figures, imaginem que tenim el tros de programa següent en Java:

```
Figura [] figuras = new Figura[3];
figuras[0] = new Figura(2,3);
figuras[1] = new Rectangulo(2,3);
figuras[2] = new Triangulo(2,3);

//Aquesta forma de fer un "for" en veritat es diu "for each" i
//permet recórrer un array d'objectes, en aquest cas "figuras",
//assignant a cada cicle la "casella actual" a l'objecte temporal
//"fig".
for(Figura fig : figuras){
   if(fig.getClass() == Figura.class){
       fig.perimetro();
   }
}
```

Amb el codi anterior només s'executarà un cop el mètode *perimetro* de *Figura*. Ara veurem «el mateix codi» amb *instanceof*:

```
Figura [] figuras = new Figura[3];
figuras[0] = new Figura(2,3);
figuras[1] = new Rectangulo(2,3);
figuras[2] = new Triangulo(2,3);
for(Figura fig : figuras){
   if(fig instanceof Figura){
       fig.perimetro();
   }
}
```

Aquest cop s'executa tres vegades el mètode *perimetro*: un cop la que pertany a la classe *Figura*, una altra en la que s'executa la de *Rectangulo* i una altra la de *Triangulo*. Per què? Perquè *Rectangulo* i *Triangulo* són classes derivades (és a dir, subclasses) de *Figura* i, per tant, són *instàncies* de (*instanceof*) *Figura*.

Vegeu també

Aquest exemple el teniu codificat en l'exemple 411 de la col·lecció d'exemples de l'assignatura.

Per acabar, cal tenir clar que el procés de *càsting*, atès que s'han de fer comprovacions en temps d'execució per a assegurar que el *càsting* és possible, consumeix recursos i, per tant, penalitza el rendiment del programa. Per tant, no s'ha d'abusar del mecanisme de *càsting*.

2.10. La classe Object

Totes les classes en Java i C# hereten d'una classe pare que no hereta de ningú anomenada *Object*. Dit d'una altra manera, és la classe arrel de la jerarquia de classes.

Com es pot deduir, la classe *Object* es presta a fer polimorfisme i *càsting*.

Totes les classes –creades per nosaltres o no–, tenen –perquè els hereten– els atributs i mètodes públics i protegits de la classe *Object*.

Enllaç d'interès

Podeu consultar la classe *Object* de Java en l'enllaç següent:

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

Això vol dir que un objecte de la classe *Object* pot apuntar qualsevol objecte de la classe que sigui.

Exemple

```
Object obj = new Figura(2,3);
```

Amb aquesta instanciació, l'objecte *obj* només pot accedir a membres (és a dir, atributs i mètodes) de la classe *Object*, ja que es tracta d'una assignació polimòrfica. Per a poder accedir als membres de la classe *Figura*, haurem de fer un *càsting*.

```
Figura fig = (Figura) obj;
fig.perimetro();
```

2.11. *Boxing*

En el cas de Java, els tipus primitius (*int*, *double*, etc.) no són objectes, però hi ha la seva classe equivalent *Integer*, *Double*, etc. Aquestes classes equivalents es coneixen com a **classes *wrapper* (o embolcall)**. Gràcies a això podrem fer el que es mostra en l'exemple següent:

Tipus primitius

En C#, a diferència de Java, cada tipus primitiu no és més que un àlies (o nom curt) d'una classe. Per tant, els tipus primitius són sempre classes.

Exemple

```
Integer integer1 = 15; //procés d'"autoboxing"
int entero1 = integer1; //procés d'"unboxing"
int entero2 = 10;
Object obj = entero2; //procés d'"autoboxing", també podria ser
//Integer obj = entero2;
int entero3 = (Integer) obj; //procés d'"unboxing", en aquest cas
//com a obj és de la classe Object hem d'explicitar la classe wrapper
//que volem fer servir per al procés d'"unboxing"
```

Vegeu també

Aquest exemple el teniu codificat en l'exemple 412 de la col·lecció d'exemples de l'assignatura.

El procés de convertir un tipus bàsic/primitiu en un objecte de la classe *wrapper* equivalent o en la classe *Object* s'anomena *boxing* o *autoboxing*. El procés contrari rep el nom d'*unboxing*.

Boxing i unboxing

Abans de la versió 5 de Java, els processos de *boxing*/*autoboxing* i *unboxing* requerien més línies de codi.

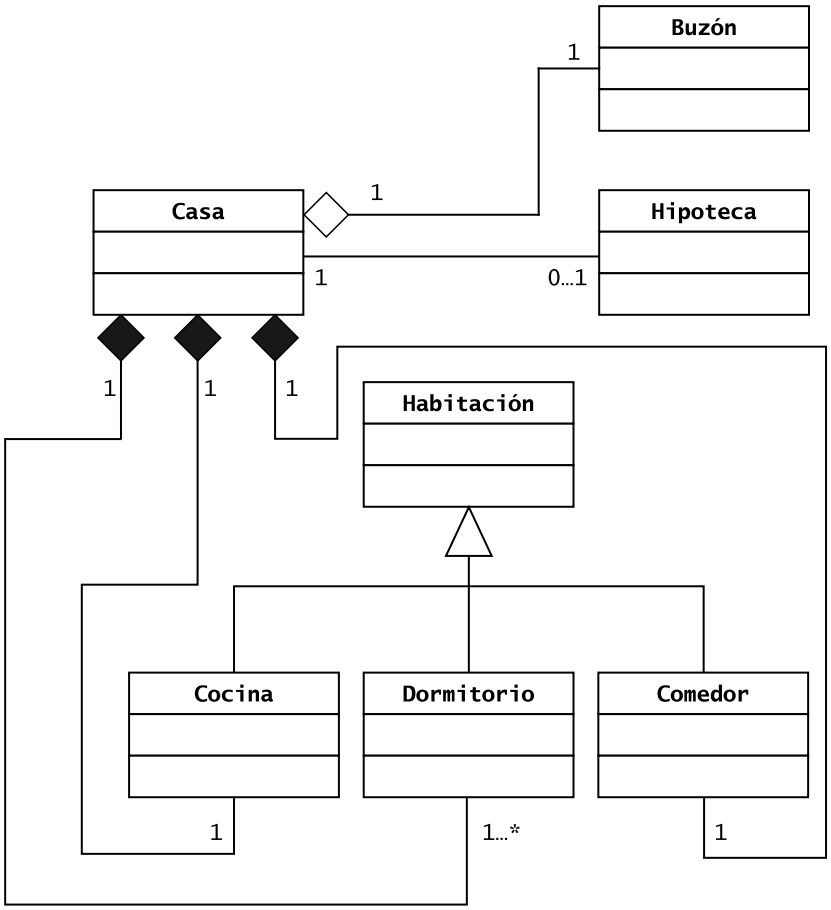
Cal tenir present que els processos de *boxing* i *unboxing* consumeixen molts recursos.

En Java, les classes embolcall s'anomenen igual que els tipus primitius, però amb la inicial en majúscula, tret d'*Integer* i *Character*. Aquests són:

- *Byte* per a *byte*.
- *Short* per a *short*.
- *Integer* per a *int*.
- *Long* per a *long*.
- *Boolean* per a *Boolean*.
- *Float* per a *float*.
- *Double* per a *double*.
- *Character* per a *char*.

3. Exemple resum

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



Una casa està formada per una cuina, un o més dormitoris, un menjador, un o més banys (no apareix per no fer més gran l'exemple), etc. Tots són habitacions, i és per això que hereten atributs i mètodes de la classe pare *Habitacion*.

Així doncs, podem dir que una casa està formada per habitacions o, dit d'una altra manera, les habitacions formen/són part d'una casa. Encara més, si la casa desapareix (és a dir, és destruïda), les habitacions desapareixen amb ella. No passa el mateix si desapareix un habitació (p. ex. perquè hem fet obres i canviem la distribució de la casa). Per aquest motiu, la relació de la classe *Casa* amb les tres habitacions representades (és a dir, *Cocina*, *Dormitorio* i *Comedor*) és de tipus associació de composició.

D'altra banda, tota casa té una bústia. Pensem en una casa adossada amb jardí, típica de les pel·lícules americanes. La bústia, tot i que forma part de la casa, pot existir sense que hi hagi casa –p. ex. encara no està construïda la casa, però la bústia ja està instal·lada. Així mateix, si la traiem, no es destrueix la casa. Per aquesta justificació, la relació entre les classes *Casa* i *Buzon* és de tipus agregació.

Per acabar, tenim la hipoteca. 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, pel fet que no tenim informació addicional, hem decidit que la navegabilitat entre les 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, podríem pensar que només la hipoteca hauria de saber de l'existència de la casa (és a dir, la punta de la fletxa acabaria en la classe *Casa*) i que la casa, en si, no hauria de saber que està hipotecada.

Resum

En aquest mòdul hem vist que quan analitzem un problema des del paradigma de la programació orientada a objectes (POO), hem de pensar necessàriament en dos nivells íntimament relacionats: les classes i els objectes.

La POO ens ensenya que les classes, mitjançant instanciacions/objectes seus, no treballen de manera aïllada dins dels programes, sinó que es relacionen entre si. En aquest sentit, aquest mòdul ens ha explicat dos tipus de relacions:

1) entre objectes/instàncies i

2) entre classes.

La primera d'elles, la relació entre objectes, és el que s'anomena formalment **associació** i s'ha tractat en l'apartat 1. En aquest mòdul hem vist les associacions més utilitzades en la POO: binàries, d'agregació, de composició i reflexiva.

Nota

Es recomana cercar informació sobre les relacions/associacions *n*-àries també existents en el llenguatge UML.

Si una idea ens ha quedat clara després de llegir aquest mòdul és que, si bé és cert que els diagrames realitzats amb UML es basen en les classes –per això s'anomenen *diagrames de classes*– hem de tenir presents els objectes (o instàncies) a l'hora de fer un disseny correcte, ja que afegeixen informació extra com la **cardinalitat**, la **navegabilitat** i el **rol**, que són tractats en el subapartat 1.5.

Pel que fa a les relacions entre classes, hem vist la relació de generalització/especialització o **herència**, com es coneix més normalment, que es tracta en l'apartat 2. Aquest mòdul ha mostrat el potencial que aquest tipus de relació té a l'hora de dissenyar els nostres programes amb mecanismes com la **sobreescriptura** (subapartat 2.7) i el **polimorfisme** (subapartat 2.8).

Bibliografia

Booch, G.; Jacobson, I.; Rumbaugh, J. (1999). *El lenguaje de modelado unificado UML*. Madrid: Addison-Wesley Iberoamericana.

Casas, J.; Conesa, J. (2013). *Diseño conceptual de bases de datos en UML*. Barcelona: Editorial UOC.

Meyer, B. (1997). *Object-oriented software construction*. Santa Bárbara: Prentice Hall.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995). *Design patterns. Elements of reusable object-oriented design*. Addison Wesley.

Rumbaugh, J.; Blaha, M.; Premernali, W.; Eddy, F.; Lorensen, W. (1996). *Modelado y diseño orientado a objetos*. Madrid: Prentice Hall.

Valeri, M.; Naccarato, G. (2006). *Java 5 – Novedades del lenguaje*. Lulu.com. ISBN: 9781430301288.