Objecte i classe

David García Solórzano

PID_00235246

Universitat Oberta de Catalunya

Índex

Introducció

Aquest mòdul té com a objectiu principal explicar els elements bàsics de la programació orientada a objectes (POO). Concretament veurem què és una classe i quins són els membres d'una classe –els atributs i mètodes–, què és un objecte, què significa instanciar, què són l'estat i el comportament d'un objecte, i què és un missatge.

Dins d'aquest mòdul explicarem, mitjançant diferents exemples, com aquests elements es relacionen amb el món real, i fan més senzill i natural el disseny d'aplicacions. Simultàniament s'introduirà el llenguatge de modelització de sistemes de programari, UML (*Unified Modeling Language*). En concret, utilitzarem UML per a representar gràficament una classe de manera formal.

Objectius

L'objectiu principal d'aquest mòdul és introduir els conceptes bàsics de la programació orientada a objectes (POO). Concretament:

1. Saber diferenciar entre un objecte i una classe, i comprendre la relació que hi ha entre ells.

2. Conèixer i entendre les parts que componen una classe, és a dir, els atributs i els mètodes.

3. Tenir un primer contacte amb el llenguatge de modelització UML.

4. Saber els conceptes relacionats amb un objecte, com són: instància, estat, comportament i missatge.

5. Reflexionar sobre com el paradigma POO us anima i, fins i tot, us obliga, com a programadors, a veure el món que us envolta d'una manera diferent de com ho fèieu fins ara.

1. Definir els conceptes *objecte* i *classe*

1.1. Què és un objecte?

Com ja heu llegit i us podíeu haver imaginat, l'**objecte** és l'element principal de la programació orientada a objectes i al voltant del qual gira aquest paradigma. D'aquí ve el nom.

En aquest punt, el més lògic seria preguntar-se: *què és un «objecte»?* i *què s'entén per «objecte» en la POO?* Per a respondre aquestes preguntes, ho farem de manera formal i informal. La formal ens porta a la definició següent.

Un objecte en POO representa alguna entitat de la vida real, és a dir, algun dels objectes únics que pertanyen al problema al qual ens enfrontem i amb el qual podem interactuar.

Cada objecte, de la mateixa manera que l'entitat de la vida real que representa, té un estat (és a dir, uns atributs amb uns valors concrets) i un comportament (és a dir, té funcionalitats o sap fer unes accions concretes).

Entitat

Els objectes del món real solen rebre el nom d'*entitats* per diferenciar-los del seu homòleg en el món de la programació orientada a objectes, anomenat *objecte*.

És possible que alguns ja hagueu entès què és un objecte, però segurament que molts encara no. Per això, anem amb l'explicació informal que hauria d'aconseguir que tots entenguéssiu què és un objecte.

Informalment podem dir que un objecte és qualsevol element únic del món real amb el qual es pot interactuar.

1.2. Què és una classe?

El concepte *classe* està íntimament relacionat amb el concepte *objecte*.

Podem definir informalment una classe com una plantilla (o esquelet o pla) a partir de la qual es creen els objectes.

Classe

També podeu entendre una classe com l'abstracció d'un objecte o com la definició d'un objecte.

Per exemple, al món hi ha milions de televisors de la mateixa marca i model. Cadascun d'aquests televisors ha estat muntat a partir d'un mateix pla/esquelet/plantilla i, conseqüentment, tots tenen els mateixos components, connexions i funcionalitats. Aquest pla/esquelet/plantilla és, en termes de programació orientada a objectes, una classe.

Per tant, una classe descriu les característiques i el comportament d'un conjunt d'objectes similars en un context determinat.

1.3. Relació entre la classe i els objectes

Relació classe-objecte

Molts llenguatges de programació permeten conèixer la classe a la qual pertany un objecte en temps d'execució del programa.

En aquest punt ha de quedar clar que quan parlem d'objecte, fem referència a una estructura que hi ha en un moment determinat de l'execució del programa, mentre que quan parlem de classe ens referim a una estructura que representa un conjunt d'objectes.

Això implica que un objecte, en comptes de ser d'un tipus bàsic (p. ex, *int*, *char*, *float*, etc.), és del tipus d'una classe concreta definida pel programador.

Així doncs, en el cas de la classe *Persona*, en aquesta són representades les propietats que caracteritzen una persona (entitat del món real), mentre que els diferents objectes (p. ex. *Elena*, *Marina* i *David*) representen persones concretes.

El mateix passa amb la classe *Televisor* i amb els objectes *miTelevisor*, *tuTelevisor*, *elTelevisorDelVecino*, etc. D'una banda, la classe *Televisor* representa el concepte abstracte de televisor (és a dir, les característiques i accions comunes dels televisors), mentre que els tres objectes declarats –*miTelevisor*, *tuTelevisor*, *elTelevisorDelVecino*– són televisors concrets.

Per acabar, vegem dos exemples que haurien de servir per a acabar d'entendre què és un objecte i una classe.

Exemple 1

Si som en una reunió familiar, cada persona de la família és única i es pot interactuar amb ella. És a dir, cada persona d'aquesta reunió és un objecte. Així doncs, *Marina*, la seva mare *Elena* i el seu pare *David* són, cadascun d'ells, objectes. Encara més, si el besavi i l'avi paterns de *Marina* es diuen tots dos *Manuel*, cadascun d'ells és un objecte diferent, ja que si bé tenen el mateix nom, no són la mateixa persona (és a dir, el mateix objecte). El besavi i l'avi, com també la resta d'integrants de la família, són elements amb els quals es pot interactuar de manera individual. Això sí, tots els membres de la família són del mateix tipus *Persona* (que és la classe a la qual pertanyen).

Exemple 2

Quan anem a una botiga d'electrodomèstics a comprar un televisor, el més probable és que ens trobem no amb una tele, sinó amb més d'una. Cadascun d'aquests televisors és un objecte, encara que pertanyin a la mateixa marca i al mateix model. Pensem que dos televisors del mateix model, encara que aparentment semblin idèntics, no per força han de ser iguals. Només cal pensar que un pot estar encès –perquè és a l'aparador– i l'altre apagat perquè és dins de l'embalatge. És a dir, tenen estats diferents. Encara més, tot i que tots dos estiguin apagats i, per tant, tinguin el mateix estat, pots tocar-los i veure que són dos objectes diferents! Això sí, tots dos televisors pertanyen a la classe *Televisor*.

Així doncs, *el teu* gos, *el teu* televisor, *el teu* bolígraf són objectes diferents del *meu* gos, *el meu* televisor i *el meu* bolígraf, encara que siguin de la mateixa raça, model i color, respectivament. Encara més, si tens dos gossos de la mateixa raça, per molt iguals que siguin, cadascun d'ells és únic i, per tant, són dos objectes diferents.

2. Classe

En aquest apartat entrarem en detall en el concepte de *classe* i veurem qüestions relacionades amb el paradigma de la programació orientada a objectes.

2.1. Analogia amb la programació estructurada

Per a entendre millor què és una classe des d'un punt de vista més proper a la implementació, farem dues analogies amb dos conceptes que ja coneixeu de l'assignatura Fonaments de programació: el mòdul i la tupla.

2.1.1. Mòdul

Mòdul

Recordeu que un mòdul és cada fitxer .c que escriviu en llenguatge C i que conté tant variables com funcions.

La primera de les analogies ja l'havíem comentat en el mòdul «Introducció al paradigma de la programació orientada a objectes» quan explicàvem com es va evolucionar de la programació estructurada a l'orientada a objectes. La idea és veure una classe com un *mòdul* de la programació estructurada. D'aquesta manera, un objecte és del tipus d'un *mòdul*, el qual té variables (que en POO denominem *atributs*) i funcions (anomenats *mètodes* en POO).

D'aquesta manera, podem imaginar que tenim un mòdul *Moto* que conté la variable *velocidad* i dues funcions que permeten modificar i consultar el valor de la variable *velocidad*, respectivament. Així, si tinc l'objecte *m1* del tipus *Moto*, puc accedir a la variable *velocidad* i als seus mètodes de modificació i consulta.

2.1.2. Tupla

Si no veieu clara l'explicació anterior, us en proposem una altra. Recordeu les tuples? Una tupla és un tipus de dada estructurada, heterogènia, d'accés directe i de mesura fixa. Si ho teniu present, el més habitual és crear un tipus propi (és a dir, *ad hoc* per al problema que tractem) del tipus tupla. En pseudocodi és:

```
tipus
   coordenades = tupla
      x, y: real;
   ftupla
ftipus
var
   punt1, punt2: coordenades;
fvar
```

En llenguatge C, un tipus tupla es defineix com:

```
typedef struct coordenada{
x, y: float;
}coordenada_t;
coordenada_t punt1, punt2;
```

Gràcies a les tuples, podem tenir tipus personalitzats més complexos i abstractes que n'inclouen altres, ja siguin bàsics, de tipus taula o fins i tot de tipus tupla. Les tuples ens permeten fer operacions com:

```
punt1.x := punt1.x +1;
punt1.y := punt2.y;
...
```

Doncs bé, si les tuples només poden tenir variables en la seva definició, una classe pot tenir funcions a més de variables. Així doncs, si ens inventem un pseudocodi per a definir les classes tindrem:

```
tipus
   rectangle = classe
      x, y, alçada, amplada: real;
      area: funcio():real
             retorna alçada*amplada;
         ffuncio
   fclasse
ftipus
var
   rect1, rect2: rectangle;
fvar
```

D'aquesta manera tenim una classe anomenada *rectangulo* que ens permet crear objectes de tipus *rectangulo*. De fet, en l'exemple anterior hem creat dos objectes, *rect1* i *rect2*. Ara podríem fer operacions com les següents:

```
rect1.x = 5;
rect1.y = 10;
escriuReal(rect1.area());//retornarà el valor 50, resultat de 5*10
```

2.2. Membres d'una classe

Com hem vist, una classe està formada per unes variables i unes funcions. Parlant estrictament en termes de POO, les variables s'anomenen **atributs** i les funcions, **mètodes**. Al seu torn, el binomi format pels atributs i els mètodes rep el nom de **membres d'una classe**. Així doncs, tant un atribut com un mètode són membres de la classe.

2.2.1. Atributs

Els atributs, també anomenats camps, són variables que codifiquen l'estat d'un objecte.

Classe sense atributs

Una classe pot no tenir atributs.

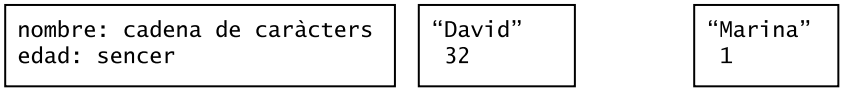
Si tenim la classe *Persona* amb els atributs *nombre* i *edad* –de tipus *cadena de caràcters* i *sencer*, respectivament–, cada objecte que es defineixi del tipus *Persona* tindrà aquests dos atributs.

Tipus dels atributs

Un atribut pot ser de tipus bàsic (és a dir, sencer, caràcter, etc.) o d'un tipus de classe concreta. Per exemple, de tipus *Persona*.

El tipus dels atributs es defineix com qualsevol altra variable.

L'estat de cada objecte *Persona* dependrà dels valors que s'assignin a aquests dos atributs, tal com es veu en la figura següent.



Classe *Persona* **Objectes de la classe *Persona***

Estat d'un objecte

L'estat d'un objecte ve definit pels valors que prenen en un instant determinat els atributs que defineixen a l'objecte.

Com podem veure en la figura, cada objecte *Persona* té estats diferents –és a dir, valors diferents per a cada atribut.

Encara que dos objectes comparteixin el mateix estat –és a dir, els mateixos valors per a tots els seus atributs–, aquests dos objectes són diferents. Només cal pensar que hi pot haver dues persones anomenades *David* de *32* anys d'edat al món i, òbviament, són persones (o objectes) diferents.

2.2.2. Mètodes

Els mètodes implementen el comportament d'un objecte o, dit d'una altra manera, les funcionalitats que un objecte és capaç de realitzar.

Fent una analogia amb la programació estructurada, els mètodes serien com les funcions (tant si retornen alguna cosa com si no). Per això, un mètode, a més de pel nom, es caracteritza pels arguments (també anomenats paràmetres) d'entrada que rep i pel valor de retorn que resulta d'executar el comportament que implementa. La descripció d'aquests elements es coneix com la **signatura del mètode**. En pseudocodi és:

```
nomMetode(param1:tipus,...,paramN:tipus):tipusRetorn
```

En el cas de la classe *Persona*, alguns mètodes poden ser:

```
hablar(texto:taula[30] de caràcters):void
andar(velocidad:sencer):void
```

El patró que segueix la signatura dels mètodes depèn de cada llenguatge de programació.

En aquest punt cal esmentar el concepte de **sobrecàrrega**.

La sobrecàrrega es produeix quan dos o més mètodes tenen el mateix nom, però diferent nombre i/o tipus d'arguments.

Dit d'una altra manera, la signatura del mètode es diferencia en la part dels arguments. Un exemple de sobrecàrrega del mètode *hablar* pot ser:

```
hablar(texto:taula[30] de caràcters):void
hablar(texto:taula[30] de caràcters, velocidad:sencer):void
```

El compilador decideix quin mètode (és a dir, quin dels dos *hablar*) ha d'invocar comparant els arguments de la crida amb els de la signatura.

2.3. Constructor i destructor

Les classes tenen dos tipus de mètodes especials anomenats *constructor* i *destructor* que no es consideren membres d'una classe, com a tals. No són membres de la classe perquè ni el constructor ni el destructor s'hereten.

Vegeu també

El concepte d'*herència* s'estudia en el mòdul «Associació i herència».

La majoria dels llenguatges de programació orientats a objectes implementen el mètode constructor, i fins i tot alguns obliguen a codificar-ne explícitament un. No passa el mateix amb el destructor, la codificació del qual es pot obviar en molts llenguatges –per exemple, en Java.

2.3.1. Constructor

El constructor es crida de manera automàtica quan es crea un objecte per a situar-lo en memòria i inicialitzar els atributs declarats en la classe. En la majoria de llenguatges, el constructor té les característiques següents:

1) Normalment el nom del constructor és el mateix que el de la classe.

Nom del constructor diferent del de la classe

En PHP5, el constructor es defineix mitjançant un mètode anomenat `__construct()`, mentre que en Python és `__init__()`.

2) El constructor no té tipus de retorn, ni tan sols *void*.

3) Pot rebre paràmetres (o també anomenats arguments) amb la finalitat d'inicialitzar els atributs de la classe per a l'objecte que es crea en aquell moment.

4) En general sol ser públic, però alguns llenguatges permeten que sigui privat.

Vegeu també

El concepte de *públic* s'estudia en el mòdul «Encapsulació i extensió» d'aquesta assignatura.

Hi ha llenguatges que permeten crear més d'un constructor –per exemple Java, C# i C++, entre d'altres. En aquests casos, el constructor sense paràmetres/arguments s'acostuma a anomenar **constructor per defecte o predeterminat**, mentre que aquells que tenen paràmetres, reben el nom de **constructors amb arguments**. Com es pot veure, dir «constructor per defecte» i «amb arguments» és el mateix que dir que es fa una sobrecàrrega del constructor. Degut a la sobrecàrrega, l'única limitació quan es vol (i es pot) crear més d'un constructor és que no es poden declarar diversos constructors amb el mateix nombre i el mateix tipus d'arguments.

En els llenguatges en els quals només es pot codificar un constructor –per exemple, PHP5 i Python–, aquest s'anomena simplement constructor.

En molts llenguatges, com ara Java o C#, si no es defineix cap constructor per a la classe, el compilador mateix crearà un constructor per defecte –és a dir, sense arguments– que no farà res especial més enllà de situar l'objecte en memòria. En el moment en què es defineix un constructor (encara que sigui amb arguments), el compilador no afegirà automàticament el constructor per defecte.

Per acabar, destaquem que, en molts llenguatges, no és obligatori que una classe tingui un constructor per defecte. Pot interessar-nos que tots els seus constructors siguin amb arguments.

2.3.2. Destructor

El destructor és el mètode que serveix per a eliminar un objecte concret definitivament de la memòria. Cal tenir en compte que:

1) No tots els llenguatges necessiten implementar un mètode destructor, com és el cas de Java i C#.

2) Per norma general, una classe té un sol destructor.

3) En alguns llenguatges no té tipus de retorn, ni tan sols *void*. En altres, generalment, té *void* com a tipus de retorn.

4) No rep paràmetres.

5) En general és públic.

La manera de declarar el mètode destructor varia entre llenguatges. Per exemple, en C ++ i C#, el nom del destructor és el mateix que el de la classe precedit pel símbol ~, p. ex. *~Persona()*.

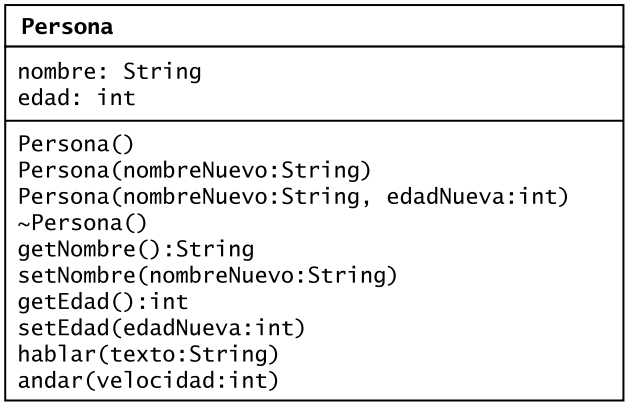
En Java, en canvi, es fa servir el mètode especial *finalize()*, que no retorna res (en aquest cas, sí que té tipus, concretament *void*). El compilador de Java no obliga a implementar el mètode *finalize()*. Així doncs, només s'ha de codificar si realment és necessari.

2.4. Representació d'una classe en UML

Per a definir una classe d'una manera formal i gràfica, se sol utilitzar el ***diagrama de classes*** del llenguatge UML (*Unified Modeling Language*).

> Utilitzar un llenguatge de modelització com l'UML permet llegir i entendre què representa una classe sense que sigui necessari veure el codi.

Aquest llenguatge gràfic defineix una classe com una caixa composta de tres parts. Vegem-ne un exemple en la figura següent per a la classe *Persona*.



Veiem que hi ha tres parts ben diferenciades mitjançant una ratlla:

1) La primera, a la part superior, indica el nom de la classe, en aquest cas, *Persona*.

2) La part central defineix els atributs de la classe –*nombre* i *edad*– i el tipus al qual pertanyen. Per a cada atribut cal indicar el tipus que hauria de ser amb independència de si el llenguatge de programació el suporta o no. Així doncs, podríem dir que un atribut hauria de ser del tipus *Date* per a indicar que ha de ser una data. En Java hi ha el tipus *Date* (és una classe), però en PHP no. Així doncs, en PHP hauríem de fer que aquest atribut fos del tipus permès per PHP que millor modeli les característiques d'una data (per exemple, un *String*) o bé fer diversos atributs de tipus *int*, és a dir, un per al dia, un altre per al mes i un altre per a l'any.

Independència del llenguatge

UML és un llenguatge conceptual, de manera que el diagrama de classes hauria de ser el més independent possible del llenguatge de programació que caldrà fer servir en el futur. D'aquesta manera, un mateix diagrama de classes serviria per a qualsevol llenguatge de programació.

No obstant això, de vegades pot succeir que un diagrama no pugui representar-se exactament en un llenguatge de programació. En aquests casos, haurem de ser creatius i intentar traduir el diagrama a codi tan bé com sapiguem.

Diagrama de classes adaptat al llenguatge de programació

Sovint, si es té clar el llenguatge de programació que es farà servir, el dissenyador adapta el diagrama de classes UML a les característiques d'aquell llenguatge. Així és com en el diagrama de classes d'exemple, hem indicat el destructor, suposant que el llenguatge de programació que fem servir permeti definir-lo. Si el llenguatge fos Java, no s'hi posaria, ja que no existeix. En Java, com a molt, posaríem el mètode *finalize()* quan l'implementéssim.

3) La part de sota defineix els mètodes de la classe: n'indica la signatura, és a dir, els paràmetres que rep i el tipus, com també el tipus de valor que retorna. En cas de no retornar res, no indica cap tipus de devolució.

Void en els diagrames de classes

En els diagrames de classes, algunes persones sí que indiquen explícitament amb la paraula *void* com a valor de retorn que un mètode no retorna res.

També podem observar que es defineixen els constructors (en aquest cas el *per defecte* i dos *amb arguments*) i també el destructor –en els llenguatges en què és possible implementar un destructor.

En aquest punt cal destacar que l'UML que hem representat per a la classe *Persona* no és complet. Ho acabarem de completar en el mòdul «Encapsulació i extensió». Aquest diagrama només defineix el que hem vist fins ara.

String

El tipus *cadena de caràcters* es defineix en molts llenguatges, com en Java, mitjançant una classe de tipus *String*.

Setters* i *getters

En les classes se solen crear mètodes per a assignar i consultar els valors dels seus atributs. El mètode que assigna el valor s'anomena *setter* (per això, el seu nom és *set* + el nom de l'atribut) i el que consulta es diu *getter* (per això, el seu nom és *get* + el nom de l'atribut). En l'exemple anterior, el mètode *setter* de l'atribut *nombre* és *setNombre* i el mètode *getter* del mateix atribut és *getNombre*.

2.5. Implementació d'una classe

A falta d'un petit detall, hem vist com representar una classe mitjançant el llenguatge gràfic UML. Fins i tot hem vist un pseudocodi d'una classe basant-nos en el concepte tupla.

Ara veurem com s'implementa una classe en un llenguatge de programació real. En aquest cas serà Java, però en altres llenguatges la sintaxi és molt similar. La classe que codificarem serà la classe *Persona* representada al diagrama de classes anterior.

```
class Persona{
   String nombre;
   int edad;

   //Constructor por defecte
   Persona(){
      nombre = "Fulanito";
      edad = 0;
   }
   //Constructor amb 1 argument
   Persona(String nombreNuevo){
      nombre = nombreNuevo;
   }
   //Constructor amb 2 arguments
   Persona(String nombreNuevo, int edadNueva){
      nombre = nombreNuevo;
      edad = edadNueva;
   }
```

```
    //Mètode getter de l'atribut "nombre"
    String getNombre(){
       return nombre;
    }
    // Mètode setter de l'atribut "nombre"
    void setNombre(String nombreNuevo){
       nombre = nombreNuevo;
    }
    // Mètode getter de l'atribut "edad"
    int getEdad(){
       return edad;
    }
    // Mètode setter de l'atribut "edad"
    void setEdad(int edadNueva){
       edad = edadNueva;
    }
    void hablar(String texto){
       // TODO: aquí va el codi. L'etiqueta "TODO" (de l'anglès "to do") se sol usar en comentaris
       //per dir que alguna cosa està pendent de fer
    }
    void andar(int velocidad){
       //TODO: aquí va el codi
    }
}
```

Vegeu també

El codi de la classe *Persona* anterior el teniu codificat en l'exemple 201 de la col·lecció d'exemples de l'assignatura.

Comentari sobre el codi font

No apareix el mètode destructor perquè en Java no existeix com a tal i, en aquest cas, no cal crear el mètode *finalize*.

En Java (i en molts altres llenguatges), el codi d'una classe va en un únic fitxer el nom del qual és el mateix que el de la classe.

Per tant, el codi de la classe *Persona* va en el fitxer *Persona.java*.

3. Objecte

En aquest apartat detallarem el concepte d'*objecte* i en veurem altres de nous, com **instància** i **missatge**.

3.1. Instància

Com ja hem comentat, els objectes són exemplars d'una classe. Així doncs, a l'hora de crear un objecte hem de seguir els passos següents:

1) Declarar l'objecte.

```
Persona persona1;
```

2) Instanciar l'objecte (és a dir, crear un objecte a partir d'una classe).

```
persona1 = new Persona("David");
```

En la majoria de llenguatges, per a instanciar/crear un objecte nou hem d'escriure la paraula *new* seguida d'un dels constructors de la classe. En aquest cas hem fet servir un dels dos constructors amb arguments, però també podríem haver utilitzat el constructor per defecte.

3) En aquest moment ja tenim l'objecte *persona1* del tipus *Persona* creat en la memòria. Així doncs, ja podem accedir als seus atributs i mètodes.

Els passos 1 i 2 es poden agrupar en un únic pas:

```
Persona persona1 = new Persona("David");
```

Com que l'acció de crear un objecte a partir d'una classe s'anomena *instanciar*, sovint els objectes reben el nom d'*instància*. Així doncs, *persona1* és una instància o un objecte de *Persona*.

3.2. Estat i comportament

Com hem llegit en la definició formal d'**objecte**, tot objecte en la POO té un estat i un comportament. Això és així perquè els objectes (o entitats) de la vida real comparteixen aquestes dues característiques.

Vegeu també

Podeu trobar la definició formal d'***objecte*** en el subapartat 1.1 d'aquest mòdul.

Hem d'entendre que l'**estat** d'un objecte és definit pels valors que prenen en un moment determinat els atributs que defineixen aquest objecte.

Per la seva banda, el **comportament** de l'objecte es pot entendre com les funcionalitats que aquest objecte és capaç de realitzar. Aquestes funcionalitats que defineixen el comportament d'un objecte les defineix la classe a la qual pertany aquest objecte mitjançant els mètodes de la classe.

Vegem diversos exemples per entendre millor aquests dos conceptes i veure com qualsevol entitat (o objecte) de la vida real té un estat i un comportament:

- Un *gos* té un estat (nom, raça, color, etc.) i un comportament (bordar, enterrar ossos, moure la cua, etc.).

- Un *cotxe* també té un estat (velocitat actual, marxa actual, color, longitud, amplada, etc.) i un comportament (pujar marxa, baixar marxa, posar intermitent, etc.).

- Un *rectangle* té un estat (coordenades d'origen x i y, alçada i amplada) i també un comportament (àrea, perímetre, modificar el valor d'x, modificar el valor d'y, modificar el valor de l'alçada, modificar el valor de l'amplada, etc.).

- De la mateixa manera, un *televisor* té un estat (encès o apagat, canal actual, volum actual, etc.) i un comportament (encendre, apagar, canviar a un canal concret, incrementar el número de canal, decrementar el número de canal, augmentar el volum, disminuir el volum, sintonitzar, etc.).

- Fins i tot una *factura* té un estat (cobrada o no, import total, paga i senyal abonada, etc.) i un comportament (canviar de no cobrada a cobrada i viceversa, modificar el valor de l'import total, etc.).

- Coses més abstractes/intangibles com un *contacte* del telèfon tenen també un estat (nom, cognom, telèfon, correu electrònic, etc.) i un comportament (introduir un atribut –és a dir, nom, cognom, telèfon, correu electrònic, etc.–, modificar el valor d'un atribut i consultar el valor d'un atribut).

Així doncs, si un televisor concret (és a dir, l'objecte) té l'atribut *canal actual* igual a 5, aquest televisor està en un estat diferent de si tingués el *canal actual* igual al número 6.

Si ens fixem en els exemples anteriors, ens adonarem que hi ha dos tipus de mètodes:

1) Aquells que fan accions que realitza l'entitat real (p. ex. bordar en el cas del gos, calcular l'àrea d'un rectangle, l'acció d'encendre d'un televisor, etc.).

2) Aquells que consisteixen a modificar o consultar el valor dels atributs de l'objecte (p. ex. modificar el número de telèfon d'un contacte o el canal actual d'un televisor) i, per tant, canvien l'estat de l'objecte. Aquests mètodes de modificació i consulta són els anomenats ***setter*** i ***getter***, respectivament.

3.3. Missatge

Quan els objectes volen interactuar entre ells, utilitzen **missatges**. Un missatge és la manera que hi ha d'accedir als atributs i mètodes d'un objecte. La forma d'un missatge, en la majoria de llenguatges, segueix la sintaxi següent:

```
variable_del_objecte.membre
```

on *membre* és un atribut o un mètode de la classe. Així doncs, per a l'exemple de la classe *Persona* tenim que:

```
Persona persona1 = new Persona("David",32);
Persona persona2 = new Persona("Marina");

//Accedim a l'atribut "nombre" per assignar un nou valor, canviem "David" per "Elena".
persona1.nombre = "Elena";

//Accedim a l'atribut "nombre" de l'objecte "persona1" per consultar el seu valor. El valor
//"Elena" es guardarà també en la variable "nombreAux".
String nombreAux = persona1.nombre;

//Accedim a un mètode de lectura/consultiu que ens retornarà "Elena", en ser aquest el valor de
//l'atribut "nombre" de l'objecte "persona1";
nombreAux = persona1.getNombre();

//Accedim al mètode d'escriptura que assigna el valor 1 a l'atribut "edad" de l'objecte "persona2".
persona2.setEdad(1);

//Podem saber l'edat de l'objecte "persona2" de dues formes, accedint a l'atribut "edad"
//directament o mitjançant la funció getter corresponent.
int edadAux = persona2.edad;
edadAux = persona2.getEdad();
```

Vegeu també

El codi anterior el teniu disponible en l'exemple 201 de la col·lecció d'exemples de l'assignatura.

L'exemple anterior és correcte, però no és la pràctica més habitual. Per què? Com ja hem comentat en el mòdul «Introducció al paradigma de la programació orientada a objectes», una de les raons que van motivar un canvi respecte al paradigma de programació estructurada va ser amagar les dades (és a dir, els atributs) i obligar els programadors a fer servir funcions per a consultar-ne i modificar-ne els valors, és a dir utilitzar mètodes *getter* i *setter*. En altres paraules, es considerava necessari crear algun mecanisme per a controlar l'accés a les dades. És a dir, no hauríem accedir –per a consultar o modificar– als atributs de la manera següent:

```
persona1.nombre = "Elena";
int edadAux = persona2.edad;
```

Sinó que hi hauríem d'accedir així:

```
persona1.setNombre("Elena");
int edadAux = persona2.getEdad();
```

Així doncs, la programació orientada a objectes va donar resposta a aquesta necessitat d'ocultar la informació mitjançant el que es coneix com a **encapsulació**.

Vegeu també

El concepte d'*encapsulació* el veurem en el mòdul «Encapsulació i extensió».

3.4. Veient el món d'una altra manera

És hora de fer un descans i aturar-nos un moment per reflexionar.

És igual on siguis, aparta la mirada d'aquesta pàgina i mira al teu voltant –però torna, que has de seguir llegint!–. Què veus? Potser respondràs «coses». Bé, no està malament, però intentem fer servir una paraula més precisa. Per exemple, *entitats*. No està malament, però, i si busquem una paraula que no sigui ni tan vulgar ni tan culta, i que a més hagi aparegut en aquests apunts? I si diem que el que veus al teu voltant són *objectes*?

A partir d'aquest moment, ja no miraràs el món que t'envolta de la mateixa manera. Ara hauries de veure *objectes* –inclosos els éssers vius– que interactuen entre ells.

Posem que ets al despatx on estudies o, millor, al sofà del menjador –l'estudi no és incompatible amb la comoditat. Segur que veus objectes molt diferents, uns de més simples i altres de més complexos. Per exemple, la làmpada que tens al costat només deu tenir dos estats (encès i apagat) i dos comportaments (encendre i apagar). D'altres, en canvi, com el televisor, tenen més estats i comportaments.

Encara més, un objecte pot ser tan complex que fins i tot pot estar format/compost per altres objectes. Sense anar més lluny, un televisor té un comandament a distància, que és un altre objecte. O, si no ho veus clar, un cotxe té un volant, quatre rodes, etc. i cadascuna d'aquestes coses és un objecte (és a dir, cada roda és un objecte independent). O, per veure-ho encara més clar, un telèfon intel·ligent és un objecte que dins té una llista d'*apps*, on cada *app* és un objecte.

Potser en aquest moment has aixecat els ulls i t'has adonat que, al teu despatx, hi ha una làmpada al sostre a més del fluorescent que hi ha damunt de la taula. Ostres, dos objectes «Làmpada»! Mmm, tenen característiques i comportaments similars... Tots dos objectes són del tipus Làmpada! O millor encara, tots dos objectes són de la classe Làmpada!

Després de llegir sobre el paradigma de la programació orientada a objectes (POO), la teva manera de mirar el món hauria d'haver canviat. Si encara no ho has fet, tranquil, que quan comencis a programar seguint la POO, aquest clic mental, el faràs irremeiablement.

Resum

En aquest mòdul hem parat especial atenció, com no podia ser d'una altra manera, als conceptes d'*objecte* i *classe*, i també a la relació que hi ha entre ells.

A més hem tingut un primer contacte amb el llenguatge de modelització de sistemes de programari, UML (*Unified Modeling Language*). En concret, hem vist com es representa una classe en un diagrama de classes fet amb aquest llenguatge.

Inevitablement, hem vist codi propi dels llenguatges orientats a objectes, sent Java el llenguatge utilitzat als exemples.

Bibliografia

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

Joyanes, L. (1998). *Programación orientada a objetos*. Madrid: McGraw-Hill.

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

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

Sommerville, I. (2005). *Ingeniería del software*. Madrid: Pearson-Addison Wesley.