Encapsulació i extensió

David García Solórzano

PID_00235247

UOC
Universitat Oberta de Catalunya

Cap part d'aquesta publicació, incloent-hi el disseny general i la coberta, no pot ser copiada, reproduïda, emmagatzemada o transmesa de cap manera ni per cap mitjà, tant si és elèctric com químic, mecànic, òptic, de gravació, de fotocòpia o per altres mètodes, sense l'autorització prèvia per escrit dels titulars del copyright.

Índex

Introducció 5

Objectius 6

1. Què és l'encapsulació? 7

2. Reutilització de codi 9

3. Ocultació 11
- 3.1. Nivells d'ocultació 13
- 3.2. Exemple d'ocultació dels membres d'una classe 14
- 3.3. L'ocultació en UML 18

4. Extensió d'atributs i mètodes d'una classe 19
- 4.1. Explicació 19
- 4.2. Casos d'ús 21
- 4.3. Ús abusiu 21
- 4.4. L'extensió en UML 22

Resum 23

Bibliografia 25

Introducció

Aquest mòdul explica el concepte d'encapsulació, un dels més importants del paradigma de la programació orientada a objectes (POO). Veurem què vol dir encapsular i dues de les seves principals implicacions:

1) la reutilització de codi i

2) l'ocultació d'atributs i mètodes.

A més, veurem el concepte d'estendre atributs i mètodes d'una classe mitjançant el modificador *static*.

De la mateixa manera, veurem com l'ocultació i l'extensió d'atributs i mètodes afecten la representació de classes quan fem servir el llenguatge de modelització de sistemes de programari, UML (*Unified Modeling Language*).

Objectius

L'objectiu principal d'aquest mòdul és introduir el concepte d'encapsulació. Com a conseqüència d'aquest objectiu, n'apareixen d'altres, com ara:

1. Entendre què significa encapsular en el paradigma de la programació orientada a objectes.

2. Veure el procés d'abstracció que cal seguir a l'hora de dissenyar una classe amb el propòsit de reutilitzar codi.

3. Conèixer l'ocultació com a un mecanisme que defineix diferents nivells d'accés a atributs i mètodes.

4. Saber què és l'extensió d'atributs i mètodes, i conscienciar del seu ús correcte.

1. Què és l'encapsulació?

Per a entendre el concepte d'encapsulació, primer veurem una analogia amb el món real per mitjà dels dos exemples següents:

Exemple 1

Quan estem malalts i decidim prendre una pastilla, les dues úniques coses que ens interessen *a priori* com a usuaris són:

1) què cura la pastilla i

2) com s'administra (via oral/rectal, freqüència, quantitat, etc.).

És a dir, fem servir la pastilla i ja està, no ens preocupa amb quins ingredients està elaborada. Sabem que la pastilla blava és la de la tos i la vermella la de la tensió, en coneixem l'aspecte (és a dir, interfície/exterior), la utilitat i la manera d'administrar-les, però no sabem res de la composició (és a dir, el seu interior).

Exemple 2

El mateix passa amb les càpsules de cafè que s'han posat tan de moda. D'una banda, sabem que el color de la càpsula –que és la interfície– ens indica el gust i la intensitat del cafè i, de l'altra, que la manera d'utilitzar-la és posar-la d'una determinada manera dins de la cafetera especialment dissenyada. En aquest punt, sabem poca cosa de la composició exacta de cafè que hi ha a dins, però com a usuaris tampoc no ens interessa gaire, sabem que el cafè que surt de la màquina és bo i ens el prenem.

En el paradigma de la programació orientada a objectes (POO), l'encapsulació és precisament això: utilitzar una classe sabent el bàsic per a poder-la fer servir i prescindir dels detalls d'implementació.

Així doncs, podem veure una classe com una pastilla o una càpsula, i d'aquí ve el nom d'**encapsulació**.

Des d'un punt de vista més formal, podríem dir que l'encapsulació en el paradigma de la POO consisteix a agrupar, seguint un procés d'abstracció, totes les dades (és a dir, atributs) i operacions/accions (és a dir, mètodes) relacionats en una mateixa classe, de manera que sigui senzill reutilitzar la classe i ocultar els detalls (és a dir, atributs i mètodes).

De fet, ja havíem vist l'encapsulació en els mòduls anteriors quan definíem una classe després d'un procés d'abstracció, encara que no li havíem posat nom. Llavors, per què dediquem un mòdul a l'encapsulació? Doncs perquè, com s'ha comentat, l'encapsulació comporta altres característiques de la POO,

com ara la **reutilització de codi** i l'**ocultació d'atributs i mètodes**. En aquest mòdul veurem aquests dos conceptes, i posarem un èmfasi especial en el d'ocultació.

2. Reutilització de codi

L'encapsulació permet que, en tenir tots els atributs i mètodes que estan relacionats entre si centralitzats en una classe, sigui fàcil poder reaprofitar aquest codi en diferents programes simplement instanciant un objecte d'aquesta classe.

Ens és igual com és implementada la classe per dins (p. ex. quants bucles hi ha, si el codi està optimitzat o no, etc.), només ens interessa saber què ens permet fer i com s'utilitza.

Dit d'una altra manera, si tenim una classe anomenada *SortedArrayInteger*, ens interessa saber que té cinc mètodes:

1) un per a afegir un element (que serà un sencer) en una posició determinada,

2) un altre per a eliminar l'element d'una posició determinada,

3) un per a consultar l'element situat en una posició determinada,

4) un que ens diu el nombre d'elements que hi ha a l'*array*, i

5) un mètode que retorna l'*array* ordenat de manera ascendent o descendent segons el valor del paràmetre *boolean* que se li passa. Com és implementada l'ordenació (és a dir, bucles, condicionals, etc.), no ens interessa. Si la classe *SortedArrayInteger* funciona bé, la podrem utilitzar en qualsevol programa. És a dir, la podrem reutilitzar.

Això que pot semblar tan senzill, per al programador novell no ho és tant, ja que requereix un procés d'abstracció al qual no està acostumat durant el disseny de les classes. Cal pensar molt bé quins atributs i mètodes tindrà la classe per a cobrir necessitats actuals i deixar la porta oberta per a futurs usos/millores. Així doncs, si preveiem que la classe que dissenyarem la farem servir més enllà del problema que resolem en aquest moment –és a dir, en preveiem la reutilització–, hem de tenir una capacitat d'abstracció més gran.

Exemple

Disposar d'una classe que sigui un *array* i que ens permeti obtenir aquest *array* ordenat segurament serà una cosa que ens serà molt útil en molts programes. Amb el plantejament anterior –és a dir, *SortedArrayInteger*–, hauríem de fer una classe per a cada tipus d'*array* que tinguem, és a dir, *integer*, *double*, *float*, *String*, etc. Com es pot veure, això no és pràctic. Així doncs, si preveiem que farem ús d'aquesta classe en un futur i que podem tenir elements molt diversos, per què no fem una classe en què el tipus d'elements que accepta l'*array* és genèric? D'aquesta manera, ja no tindríem *SortedArrayInteger*, ni *SortedArrayDouble*, etc., sinó simplement *SortedArray*. Si implementem aquesta nova classe ge-

Vegeu també

Aquest exemple el teniu codificat en l'exemple 301 de la col·lecció d'exemples de l'assignatura.

nèrica en Java, el mètode *addElement* ja no tindria les signatures *addElement(int element, int position)* o *addElement(double element, int position)* segons com es digui la classe, sinó simplement *addElement(Object element, int position)*, on la classe *Object* és la classe pare de totes les classes. La classe *Object* permet assignar qualsevol tipus de classe (de Java: *Integer*, *Double*, *Float*, *String*, etc.; o nostra: *Persona*) a l'objecte declarat com a tipus/classe *Object*.

SortedArray
array: Object [ ]
SortedArray() SortedArray(numElements:int) ~SortedArray () addElement(element:Object, position:int):boolean removeElement(position:int):boolean getElement(position:int):Object getNumElements():int getSortedArray(ascOrDesc:boolean):Object [ ]

Així, doncs, fent un procés d'abstracció més gran i coneixent les característiques del llenguatge de programació que cal utilitzar, hem pogut codificar una única classe genèrica que ens dóna la funcionalitat d'un *array* per a qualsevol tipus d'element amb un mètode d'ordenació i que podrem fer servir en qualsevol programa futur.

Destructor UML

En el cas de l'exemple, hem posat el destructor en el diagrama UML per a evitar condicionar l'exemple a un llenguatge genèric. En el cas de Java, no hi hauria mètode destructor.

Solució alternativa

En el mòdul «Tipus de classe i interfície» d'aquesta assignatura veurem una altra manera d'abordar aquest problema mitjançant una classe genèrica o parametritzada. Aquesta segona opció és millor, pel que fa a rendiment del programa, que la solució proposada aquí que utilitza la classe *Object*.

3. Ocultació

Arribats a aquest punt, podem intuir que com menys informació es dóna d'una classe –és a dir, com més petita és la part visible d'una classe per a les altres classes que hi poden interactuar–, menys probable és que hi hagi problemes i més senzill és assegurar certs factors de qualitat com la reusabilitat, la portabilitat i la facilitat d'ús de les classes.

L'encapsulació garanteix la integritat de les dades que conté l'objecte mitjançant el mecanisme d'ocultació. L'**ocultació** d'informació és un mecanisme molt útil i una conseqüència directa de l'encapsulació.

Terminologia

En altres explicacions veureu el concepte d'*ocultació* referit com a «visibilitat dels membres d'una classe» (és a dir, atributs i mètodes) o «restricció d'accés a atributs i mètodes».

De manera informal, podem dir que el mecanisme d'ocultació permet amagar tot allò que no interessa que se sàpiga o se'n tingui accés.

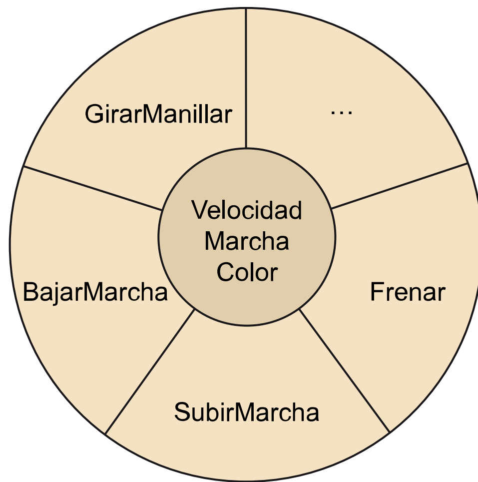
D'aquesta idea sorgeix la bona pràctica següent, a la qual tot programador orientat a objectes s'ha d'acostumar.

El programador que utilitzarà la classe ha de conèixer i poder accedir d'una manera directa als atributs i mètodes estrictament necessaris, res més. Així, s'evita l'accés als membres de la classe per qualsevol altre mitjà diferent dels especificats.

Vegem un exemple per a entendre el concepte d'ocultació.

Exemple 1

Si tenim la classe *Bicicleta*, podríem representar-la seguint la metàfora de la pastilla. La zona exterior serien els atributs i mètodes visibles/accessibles per altres classes, és a dir, la zona pública. En canvi, la zona central seria la privada, la que quedaria amagada i no seria accessible des de fora. Fent el símil amb la pastilla, la zona exterior seria la tapa/embolcall/càpsula de la pastilla, mentre que la zona interior seria el granulat que hi ha dins de la pastilla.



D'altra banda, com ja hem comentat abans, és una bona pràctica ocultar els atributs d'una classe sempre que es pugui i obligar a accedir-hi per a consultar-los o modificar-los mitjançant mètodes *getter* i *setter*. D'aquesta manera es garanteix la integritat de les dades i que compleixin així les expectatives que es requereixen dins de la classe. En l'exemple de la bicicleta, caldria crear els següents mètodes *getter* i *setter* públics per a poder consultar i modificar els atributs de la classe: *getVelocidad*, *getMarcha*, *getColor*, *setVelocidad*, *setMarcha* i *setColor*.

Vegem un altre exemple relacionat amb el concepte d'ocultació.

Exemple 2

Imaginem que volem implementar el mètode *hablar(texto: String)* de la classe *Persona*. Podem intuir que es tracta d'un mètode complex de codificar i que, segurament, la seva implementació ha de ser dividida en diferents funcions (és a dir, mètodes) més simples. Aquests nous mètodes en què es descompon el mètode *hablar(texto: String)* no han de ser coneguts per força pel programador que farà servir la classe *Persona*, és a dir, els hi podríem ocultar i res no canviaria per a ell. Si fem servir la terminologia de la programació orientada a objectes, direm que el mètode *hablar(texto: String)* serà *públic*, mentre que la resta de mètodes que el componen seran *privats*.

Per acabar, vegem un últim exemple.

Exemple 3

Si bé és cert que quan obtenim el permís de conduir cotxes se suposa que tenim unes nocions de mecànica, la realitat és que la majoria de conductors no tenen ni idea de com funciona el seu cotxe per dins. Això no els limita a l'hora de conduir-lo.

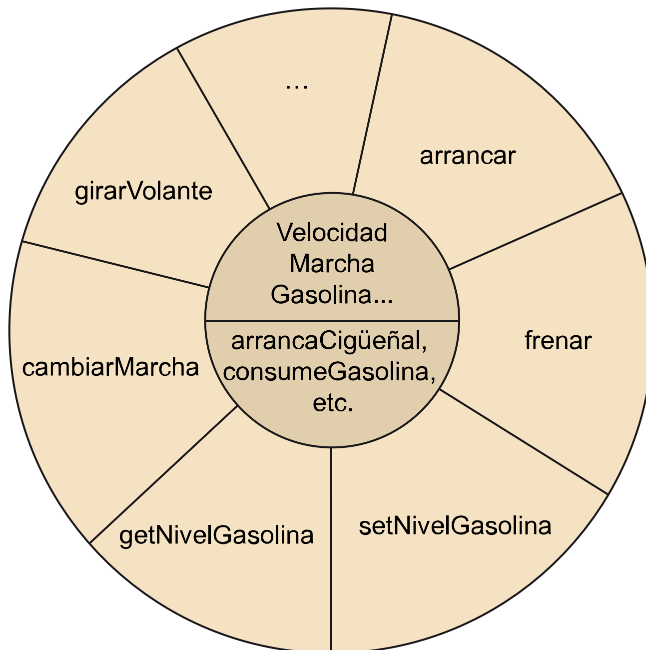
El conductor sap com es fa servir el volant, com es canvia de marxa, per a què serveix cada pedal i quan els ha d'utilitzar, sap on és la palanca que activa els intermitents, els llums, etc. En definitiva, el conductor coneix la interfície del vehicle, no la seva implementació. És a dir, no sap quins mecanismes s'activen perquè, quan pitja el botó del parabrisa, es posi en marxa. De la mateixa manera, sap que ha d'introduir la clau al contacte i girar-la perquè el cotxe engegui, però desconeix quines accions fa la mecànica del cotxe per posar-se en marxa.

Si veiéssim el cotxe com una classe de la POO, podríem dir que els mètodes *arrancar()*, *frenar()*, *girarVolante(derechaOIzquierda: boolean, grados: float)*, etc. són la part visible/pública i, per tant, coneguda pel conductor (és a dir, el programador que utilitza la classe *Coche*). No obstant això, quan el conductor sol·licita el mètode *arrancar()*, per dins, aquest mètode crida a molts altres mètodes ocults/privats que el conductor desconeix – per exemple, *arrancaCigüeñal*, *consumeGasolina*, etc.– i que només coneix el mecànic (és a dir, el programador que ha dissenyat i codificat la classe *Coche*).

Així mateix, la classe *Coche* té atributs que com a conductors podem consultar i modificar per mitjà de mètodes *getter* i *setter*. Per exemple, el nivell de benzina del dipòsit és un atribut privat que consultem amb el mètode *getNivelGasolina()* que ens retorna el valor al tauler de control del vehicle en forma de nombre o d'agulla. Per a posar benzina no

modifiquem directament aquest valor, sinó que el modifiquem mitjançant el mètode *setNivelGasolina(float litros)* que invoquem quan introduïm la mànega de l'assortidor de la benzinera a l'orifici específic del cotxe.

D'una manera gràfica i resumida, un *Coche*, vist com una classe POO i seguint la metàfora de la pastilla, seria:



A la part externa hi ha els mètodes i atributs públics, i a l'interior els privats.

3.1. Nivells d'ocultació

Abans de continuar cal emfatitzar les següents idees clau.

> Quan parlem de visibilitat/ocultació d'un atribut o mètode d'una classe, l'hem d'entendre en relació amb la resta de classes.
>
> Així doncs, l'altra idea clau que ha de quedar clara és que una classe sempre pot accedir a tots els atributs i mètodes que té definits.
>
> Com a última idea clau, convé dir que la interacció entre dues classes es fa mitjançant la interacció entre objectes d'aquestes dues classes. Aquesta interacció es fa per mitjà de missatges.

Un cop clares les tres idees claus anteriors, podem dir que l'encapsulació defineix els nivells d'accés per als membres d'una classe. Podem restringir l'accés als atributs o als mètodes en diferent mesura. En general, els llenguatges de POO defineixen tres nivells d'accés:

Altres nivells d'accés segons el llenguatge

Hi ha altres llenguatges que defineixen més nivells d'accés. Per exemple, C# defineix el nivell *internal* i Java, el nivell *package*.

1) Públic (*public*): quan un atribut o mètode és públic en una classe, qualsevol altra classe pot accedir-hi directament i utilitzar-lo.

2) Protegit (*protected*): quan un atribut o mètode és protegit en una classe, qualsevol altra classe filla (o subclasse) d'aquesta classe pot accedir-hi i utilitzar-lo. Dit d'una altra manera, una classe que hereta pot accedir i utilitzar atributs i mètodes que han estat declarats com a "protegits" en la classe pare.

Vegeu també

Els conceptes d'*herència*, *classe pare* i *subclasse* (o *classe filla*) són tractats en el mòdul «Associació i herència» d'aquesta assignatura.

3) Privat (*private*): quan un atribut o mètode és privat en una classe, cap altre classe no pot accedir-hi. Així doncs, només pot ser accedit des de dins de la classe mateixa que el defineix.

Si no indiquem la visibilitat d'un membre de la classe, el compilador dóna una visibilitat per defecte (*default*) als atributs i mètodes. En Java, aquesta visibilitat per defecte és *package*.

Paquet

El concepte de paquet (*package*) el veurem quan fem exercicis de codificació.

A continuació es mostra una taula resum amb els diferents nivells d'accés a Java.

Nivells d'accés definits a Java

Nivell d'accés \Accés des de	La mateixa classe	Subclasse	Package	Altres/Món
public	x	x	x	x
protected	x	x	x	
default (*package*)	x		x	
private	x			

3.2. Exemple d'ocultació dels membres d'una classe

En aquest subapartat exemplifiquem l'ocultació dels membres d'una classe, concretament els casos públic i privat. Per a això, reprendrem la implementació de la classe *Persona* del mòdul «Objecte i classe», i hi afegim la visibilitat als membres de la classe:

Vegeu també

Per a poder explicar amb detall el nivell d'accés *protected*, primer hem de conèixer el concepte d'*herència* entre classes. Aquest concepte es tracta en el mòdul «Associació i herència».

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

   public Persona(){
      nombre = "Fulanito";
      edad = 0;
   }
   public Persona(String nombreNuevo){
      nombre = nombreNuevo;
   }
   public Persona(String nombreNuevo, int edadNueva){
      nombre = nombreNuevo;
      edad = edadNueva;
   }
   public String getNombre(){
      return nombre;
   }
   public void setNombre(String nombreNuevo){
      nombre = nombreNuevo;
   }
   public int getEdad(){
      return edad;
   }
   public void setEdad(int edadNueva){
      edad = edadNueva;
   }
   public void hablar(String texto){
      //TODO: aquí va el codi
   }
   public void andar(int velocidad){
      //TODO: aquí va el codi
   }
}
```

Vegeu també

Aquest exemple el teniu codificat en l’exemple 302 de la col·lecció d’exemples de l’assignatura.

En la implementació anterior, podem veure com els atributs són privats i els mètodes són públics. Aquesta estratègia d’ocultació respecta la bona pràctica de restringir al màxim l’accés a les dades (és a dir, als atributs) per a garantir-ne la integritat.

Ara, després del mètode *setEdad*, afegirem el codi d’un nou mètode anomenat *checkEdadOK* que retornarà *true* quan la nova edat (és a dir, atribut *edadNueva*) sigui un nombre positiu o igual a zero, i *false* en cas contrari. Aquest mètode serà d’ús intern, és a dir, serà privat.

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

   public Persona(){
      nombre = "Fulanito";
      edad = 0;
   }
   public Persona(String nombreNuevo){
      nombre = nombreNuevo;
   }
   public Persona(String nombreNuevo, int edadNueva){
      nombre = nombreNuevo;
      if(checkEdadOK(edadNueva)){
         edad = edadNueva;
      }else{
         edad = 0;
      }
   }
   public String getNombre(){
      return nombre;
   }
   public void setNombre(String nombreNuevo){
      nombre = nombreNuevo;
   }
   public int getEdad(){
      return edad;
   }
   public void setEdad(int edadNueva){
      if(checkEdadOK(edadNueva)){
         edad = edadNueva;
      }else{
         edad = 0;
      }
   }
   private boolean checkEdadOK(int edad){
      boolean ok = false;
      if(edad>=0){
         ok = true;
      }
      return ok;
   }
   public void hablar(String texto){
      //TODO: aquí va el codi
     }
   public void andar(int velocidad){
      //TODO: aquí va el codi
   }
}
```

Vegeu també

Aquest exemple el teniu codificat en l'exemple 302 de la col·lecció d'exemples de l'assignatura.

Com podem veure, hem declarat el nou mètode *checkEdadOK* com a privat. Això vol dir que només es pot utilitzar dins de la mateixa classe. També implica que, si volem que un altre programador utilitzi la classe *Persona*, no li direm res sobre l'existència del mètode *checkEdadOK*, ja que no n'ha de fer res amb ell.

Imaginem que tenim una segona classe anomenada *Example302* amb un únic constructor i que hi fa servir la classe *Persona* mitjançant la instanciació de dos objectes.

```
class Example302{
   Persona persona1 = null, persona2 = null;

   public Example302(){
      persona1 = new Persona("David");
      persona2 = new Persona("Elena",37);
      persona1.checkEdadOK(32);
      persona1.edad = 32;
      persona1.setEdad(32);
      persona2.nombre = "Helena";
      persona2.setNombre("Helena");
   }
}
```

En el codi anterior, veiem en vermell les instruccions que donarien error a causa de les limitacions que imposa el nivell d'accés *private* en el mètode *checkEdadOK* i en els atributs *edad* i *nombre*.

Gràcies a aquest exemple, podem veure com n'és, de bona, la pràctica de declarar els atributs d'una classe com a privats i obligar a accedir-hi mitjançant mètodes públics de tipus *setter* i *getter*. Si ens fixem en l'atribut *edad*, gràcies a l'ús de *setEdad*, a més d'assignar un valor a l'atribut *edad*, comprovem que el nou valor que cal assignar sigui correcte (és a dir, superior o igual a zero), ja que a dins crida el mètode privat *checkEdadOK*. Per tant, garantim la integritat de les dades i ho estem fent de manera transparent a la classe *Example302*.

Per acabar, cal dir que és una bona pràctica indicar el nivell d'accés de les classes. En general, les classes són públiques, així doncs:

```
public class Persona{
   //TODO: Atributs i mètodes
}
public class Example302{
   //TODO: Atributs i mètodes
}
```

Classes privades

Es poden crear classes privades. Això se sol fer quan es tracta d'una classe auxiliar que es fa servir i es defineix dins d'una altra classe pública. Aquestes classes que són dins d'una altra es coneixen com a *nested classes*, és a dir, classes imbricades. En realitat, les *nested classes* no han de ser necessàriament privades.

3.3. L'ocultació en UML

Si recordem, en el mòdul «Objecte i classe» vam dir que la representació de la classe *Persona* mitjançant el llenguatge UML no era completa. El que ens faltava era indicar el nivell d'accés dels atributs i mètodes. Com es representen els nivells d'accés a UML? Amb un d'aquests símbols: +, -, # o ~.

- El símbol + al costat d'un atribut o mètode indica que és públic.
- El símbol - al costat d'un atribut o mètode indica que és privat.
- El símbol # al costat d'un atribut o mètode indica que és protegit.
- El símbol ~ al costat d'un atribut o mètode indica que el seu nivell d'accés és de tipus paquet.

Així doncs, la classe *Persona* representada en UML seria la que es mostra en la figura següent.

Persona
- nombre: String - edad: int
+ Persona() + Persona(nombreNuevo:String) + Persona(nombreNuevo:String, edadNueva:int) + ~Persona() + getNombre():String + setNombre(nombreNuevo:String) + getEdad():int + setEdad(edadNueva:int) - checkEdadOK(edad:int):boolean + hablar(texto:String) + andar(velocidad:int)

Gràcies a l'UML anterior, veiem de seguida que hi ha dos atributs privats –*nombre* i *edad*– i un mètode privat, *checkEdadOK*.

Per acabar, cal comentar que quan representen una classe en UML, algunes persones ordenen els atributs i els mètodes per ordre alfabètic, altres per rellevància de l'atribut o mètode (p. ex. primer o més amunt aquells que són més importants), altres per nivell d'accés (p. ex. primer els privats), altres posen primer els mètodes *getter* i *setter*, i altres, simplement, no segueixen una lògica concreta.

4. Extensió d'atributs i mètodes d'una classe

4.1. Explicació

Per a entendre el concepte d'extensió d'atributs i mètodes d'una classe, vegem un exemple.

Exemple

Imaginem que tenim dues instàncies de la classe *Persona* anomenades *persona1* i *persona2*, respectivament, i que aquesta vegada la classe *Persona* l'hem implementat només amb quatre mètodes (és a dir, *setNombre*, *getNombre*, *setEdad* i *getEdad*), a més del constructor i destructor. En ser *persona1* i *persona2* dues instàncies (o objectes), aquestes ocupen espai en la memòria i cadascuna té els atributs –amb els seus valors assignats– i mètodes de la classe.

nombre = "David" edad = 32 getNombre():String setNombre(nombreNuevo:String) getEdad():int setEdad(edadNueva:int)	nombre = "Marina" edad = 1 getNombre():String setNombre(nombreNuevo:String) getEdad():int setEdad(edadNueva:int)
persona1	persona2

En aquest cas, si fem:

```
persona1.getNombre();
```

obtindrem el valor "David", mentre que si fem:

```
persona2.getNombre();
```

obtindrem el valor "Marina".

Com veiem, cada instància pot fer ús dels mètodes de la classe, però el comportament de cada mètode dependrà de l'estat de la instància. Així mateix, els atributs de cada instància són els de la classe *Persona*, però els seus valors són els assignats a la instància. Per això, se sol dir que en instanciar una classe, es creen atributs i mètodes de la instància.

En aquest punt, potser algú de vosaltres us pregunteu: és possible tenir un atribut o mètode comú/compartit entre instàncies? Doncs sí, és possible. Molts llenguatges moderns orientats a objectes permeten utilitzar un modificador que podem aplicar en la definició de mètodes i atributs de les classes, **i fan que aquests elements constin com a pertanyents a la classe, enlloc de pertanyents a la instància**. Aquest modificador és la paraula clau ***static***. Així doncs, els atributs i mètodes que van precedits per la paraula *static*, a més de ser anomenats atributs i mètodes estàtics, reben també el nom d'elements de la classe, no de la instància, ja que no pertanyen a un objecte (instància de classe) en concret, sinó a la classe com a entitat.

Per a veure l'ús d'aquest modificador, continuarem amb l'exemple anterior, en el qual afegirem un atribut estàtic anomenat *numPersonas*.

```
class Persona{
   private String nombre;
   private int edad;
   public static int numPersonas = 0;

   public Persona(){
      nombre = "Fulanito";
      edad = 0;
   }

   …
}
```

Vegeu també

Aquest exemple el teniu codificat en l'exemple 303 de la col·lecció d'exemples de l'assignatura.

Hem declarat l'atribut *numPersonas* públic i estàtic. Això ens permetrà accedir-hi directament per mitjà d'un objecte de la classe *Persona* instanciat en qualsevol altra classe. Ara, si fem:

```
persona1.numPersonas=1;
persona2.numPersonas=2;
int suma = 3 + persona1.numPersonas;
```

El valor de la variable *suma* serà 5, ja que l'atribut *numPersonas* és comú per a totes les instàncies de la classe *Persona*. Això vol dir que tant és des de quina instància es modifiqui o es consulti, ja que només hi ha un atribut *numPersonas* per a totes les instàncies pel fet d'haver-se declarat com a *static*. És a dir, pel fet de ser *numPersonas* un element de la clase i no de la instància.

Així mateix, un membre de la classe declarat com a *static*, pot ser accedit sense fer servir cap instància, ja que és un element de la classe. Per a això, només cal crear el missatge anteposant el nom de la classe a l'atribut o mètode, en comptes del nom de l'objecte. És a dir:

```
Persona.numPersonas=1;
Persona.numPersonas=2;
int suma = 3 + Persona.numPersonas;
```

El resultat seria el mateix d'abans: 5.

Si el que volem és crear un mètode estàtic, llavors ho farem així:

```
public static int getNumPersonas(){
   return numPersonas;
}
```

La manera d'accedir als mètodes és la mateixa que amb els atributs, és a dir, podem accedir mitjançant un objecte de la classe *Persona* o directament amb el nom de la classe.

Tanmateix, en aquest cas és important tenir present que un mètode estàtic no pot accedir a atributs i mètodes no estàtics directament encara que siguin de la mateixa classe. Per a poder accedir-hi, s'ha de crear una instància dins el mètode estàtic i accedir als atributs i mètodes per mitjà de la instància.

En canvi, un atribut estàtic pot ser accedit tant des d'un mètode estàtic com no estàtic.

El motiu és simple. L'atribut o mètode estàtic és creat quan és creada la classe, no la instància. Recordem una vegada més que els atributs/mètodes estàtics són elements de la classe, no de la instància. Les classes es creen en iniciar el programa. És a dir, estan disponibles des del començament de l'execució del programa i accedir a atributs o mètodes no estàtics donaria lloc a situacions incontrolades/estranyes. Per exemple, el mètode estàtic no sabria el valor de l'atribut no estàtic que vol utilitzar perquè no té assignat cap valor.

4.2. Casos d'ús

Després de veure l'explicació de l'extensió, ens podem preguntar: per a què pot ser útil declarar un atribut o mètode com a estàtic? Hi ha diversos usos coneguts:

1) Tenir un comptador del nombre d'instàncies creades d'una classe.

2) Per a atributs que tenen el mateix valor per a totes les instàncies d'una classe i, per tant, si es modifica, s'ha de modificar el valor de l'atribut en totes les instàncies.

Per exemple, si el preu de viatjar en autobús està inclòs en la classe *Autobus* i en el nostre context tots els autobusos tenen el mateix preu, seria suficient tenir un atribut de classe (és a dir, estàtic). Si el declarem estàtic, estalviem espai en la memòria, ja que l'atribut només apareix una vegada en la memòria, per moltes instàncies de la classe que fem. En canvi, si no fos estàtic, apareixeria en la memòria tantes vegades com objectes de la classe instanciem.

El mateix passa amb les constants, com el valor de π, e, etc. En aquest cas són valors de consulta, ja que ningú no els en canviarà el valor, però no per això deixen d'ocupar memòria.

4.3. Ús abusiu

Els membres estàtics han d'estar subjectes a una atenció addicional per a no cometre errors de disseny.

De vegades es tendeix a fer servir atributs estàtics com una manera d'emular el comportament de les variables globals, que són presents a qualsevol lloc de l'aplicació. Això pot originar exactament els mateixos problemes que coneixem per l'ús de variables globals en el paradigma de la programació estructurada.

D'altra banda, sovint, les classes es fan servir com a contenidors de mètodes estàtics. Per exemple, la classe *Math* de Java no és més que un contenidor de dos atributs (*PI* i *E*) i diversos mètodes estàtics relacionats amb les matemàtiques, p. ex. *abs*, *cos*, *exp*, *max*, *min*, etc.

Enllaç d'interès

Sobre la classe Math de Java podeu consultar l'enllaç següent: https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html

Encara que això és correcte des d'un punt de vista de la implementació, no ho és tant des d'un punt de vista del disseny de la classe. Crear una classe per a ser un contenidor d'atributs i mètodes estàtics –sense afegir més valor a la classe, ja que no caldria instanciar-la–, és caure en una pràctica similar a la de la programació estructurada. En el cas de la classe *Math*, la seva implementació basada en atributs i mètodes estàtics pot estar justificada, i en altres casos no. Per tant, ha de quedar clar que cal fer servir els atributs i mètodes estàtics amb sentit comú.

4.4. L'extensió en UML

Per a indicar que un atribut o un mètode és *static* en UML, ha d'estar subratllat. Així doncs:

```
Persona
-------------------------------------------------
- nombre: String
+ numPersonas: int
- edad: int
-------------------------------------------------
+ Persona()
+ Persona(nombreNuevo:String)
+ Persona(nombreNuevo:String, edadNueva:int)
+ ~Persona()
+ getNombre():String
+ setNombre(nombreNuevo:String)
+ getEdad():int
+ setEdad(edadNueva:int)
- checkEdadOK(edad:int):boolean
+ hablar(texto:String)
+ andar(velocidad:int)
+ getNumPersonas():int
```

Resum

En aquest mòdul hem vist en l'apartat 1 què és l'encapsulació. Per a això hem fet el símil amb les pastilles farmacèutiques i amb les càpsules de cafè.

A més, hem vist com l'encapsulació facilita dues característiques del paradigma de la programació orientada a objectes (POO). La primera d'aquestes característiques és la reutilització de codi, tal com es veu en l'apartat 2. Dissenyar (és a dir, encapsular) correctament una classe significa no només recollir les necessitats del programa/problema que resolem en aquell moment, sinó ser capaços també de veure més enllà –mitjançant un procés d'abstracció– per tal de crear una classe que doni resposta a necessitats futures de manera que pugui ser reutilitzada en futurs programes.

La segona característica que fomenta l'encapsulació és l'ocultació d'atributs i mètodes, tal com s'ha vist en l'apartat 3. Hem vist que l'ocultació –definida per diferents nivells d'accés (principalment, privat, públic i protegit)– és un mecanisme que ajuda a garantir la integritat de les dades (és a dir, atributs) i a alliberar el programador usuari de la classe de la necessitat de conèixer els detalls de la implementació de la classe que es vol utilitzar.

Per acabar, en l'apartat 4 hem vist l'extensió d'atributs i mètodes, que és una arma de doble tall. D'una banda, és interessant a l'hora de minimitzar l'impacte en la memòria i, de l'altra, el seu ús abusiu i descontrolat pot portar a un disseny inadequat de les classes.

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.