

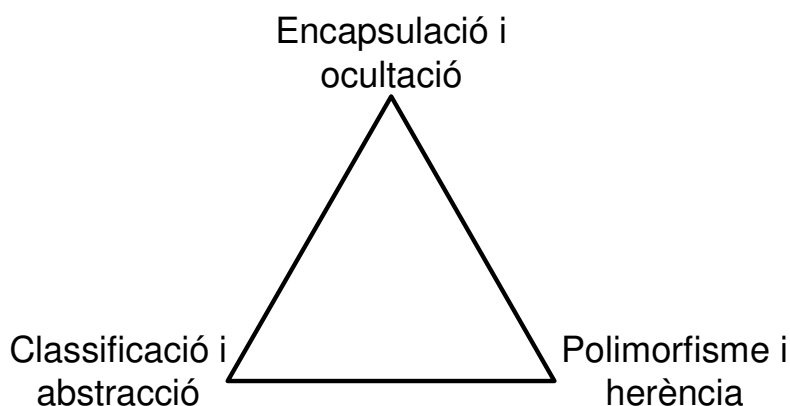
## Orientació a l'objecte

L'**Origen** se situa a principis dels 70 en un projecte d'un entorn amigable que l'empresa Xerox va desenvolupar a Palo Alto amb un equip format per Alan Kay, Adele Goldberg i Dan Ingalls. Aquest projecte ha aportat al món informàtic elements tan interessants com el ratolí, el primer entorn gràfic amb icones i el llenguatge Smalltalk que és el primer llenguatge OO (orientat a l'objecte). Per desenvolupar Smalltalk aprofitaren els conceptes aportats pel llenguatge Simula67 (Noruega, 1967).

Els conceptes d'OO van arribar al mercat a través del C++ i dels sistemes operatius. Avui tot llenguatge de programació, sistema operatiu, SGBD, etc. és OO. De fet, no és estrictament cert ja que el que s'ha fet en molts casos és incorporar els objectes a més a més, com una opció. Per exemple, amb Turbo Pascal que dóna suport a objectes puc fer programes OO, però també en puc fer sense cap objecte.

**Introducció a l'orientació a l'objecte:** la idea base us ha de resultar familiar, partim del concepte de tipus com una encapsulació de valors i de les operacions que es poden realitzar amb ells (dades i tractaments), aquest concepte el coneixeu com a tipus abstractes de dades (TAD). En l'OO només afegim al TAD un mecanisme d'herència.

Es considera que l'OO se sustenta en tres punts que se solen representar com als vèrtexs d'un triangle:



Tant els aspectes d'encapsulació i ocultació (vol dir que un TAD és un mòdul o càpsula de la qual no cal conèixer la implementació) com els de classificació i abstracció (un TAD és una abstracció que representa un element d'una classe o tipus determinat) us han de ser prou familiars ja que s'usen per als TAD. Per això només comentarem el tercer.

La idea de polimorfisme tot i que us pugui semblar nova, també la coneixeu, de fet, l'esteu usant, per exemple, amb la suma i resta numèrica. Si escric  $a := b + c$ , algú pot dir-me de quin tipus són  $a$ ,  $b$  i  $c$ ? Lògicament poden ser de qualsevol tipus enter o real, i en general numèric.

Es diu que el signe "+" ha estat sobrecarregat (*overloaded*), també podríem dir que la operació suma té diverses formes, que és polimorfa. Aquest polimorfisme de les operacions és imprescindible per al mecanisme d'herència.

(En la nomenclatura OO s'anomena *atributs* als camps d'un TAD, *mètodes* a les operacions, i *classe* al tipus.)

Diem que una classe n'hereta una altra quan conté tots els seus atributs i tots els seus mètodes. Per tant, no cal definir-los de nou ni fer-ne una còpia simplement se n'encarrega el compilador. Si al definir la classe B es fa que hereti la classe A, B tindrà tots els atributs i operacions d'A amb el mateix nom, per això ens cal el polimorfisme. Normalment s'afegirà a B algun o alguns atributs o mètodes nous que complementin aquells que ha heretat d'A.

Aquest mecanisme permet la definició de classes (TAD) no instanciables, que vol dir que han estat dissenyades per ser heretades. D'aquestes classes en direm classes *abstractes*. En alguns casos aquesta classe abstracta ho és simplement perquè les seves instàncies directes no tenen sentit, però en altres casos ho són perquè a més resulta impossible d'instanciar-les ja que no se n'ha fet una definició completa. Una classe abstracta pot *diferir* a qui l'hereti la definició d'una part de les seves operacions, d'aquesta classe abstracta se'n diu classe *diferida* (*deferred*). Per exemple, podríem definir una classe NUMERIC, que garanteixi, a través de l'herència, que tot tipus numèric té suma, resta, multiplicació i divisió, però com que pot convenir en alguns casos definir l'operació d'una manera diferent ens convindrà enunciar o poder obligar que qui hereti de NUMERIC tingui aquestes operacions, però no haver de definir com són. Definida així, NUMERIC serà una classe diferida.

Algun llenguatge OO, com Eiffel, permet definir classes genèriques, que per poder-les instanciar requeriran d'un o més paràmetres. Aquest mecanisme permetrà definir estructures genèriques com la taula (*array*) la pila (*stack*) o la cua (*queue*), on el tipus de l'element base serà un paràmetre. Això ens permet definir una única vegada la classe cua i instanciar, per exemple, cues de diferents tipus, com d'enter, text o de persona, pensant en una classe persona que pugui ser una estructura més o menys complexa.

**Nomenclatura OO:** cal tenir present que en la bibliografia OO podeu trobar diverses nomenclatures, depenent de l'autor o del traductor. La més habitual sol ser:

- Classe: el TAD o encapsulació de dades i operacions.
- Objecte: element concret d'una classe (variable, constant, etc.), que solem anomenar *instància*. Els programes OO estan formats per objectes.
- Atribut o camp: dada interna d'un objecte (una classe té estructura de *tupla*).
- Mètode o servei: operació (funció, procediment o rutina) accessible d'un objecte.

**Filosofia OO:** Es tracta d'una nova forma de fer *software*. Es vol construir un model informàtic del sistema real, on cada classe representi cada un dels elements del sistema, modelitzant la realitat. Per exemple, el subíndex d'una taula, tot i que pugui ser implementat amb un enter, haurà de ser de la classe que representi els subíndexs numèrics.

Amb el sistema informàtic es crea un model que simula la realitat, d'on n'extreu la funcionalitat que li interessa. És a dir, no es parteix de la funcionalitat que interessa

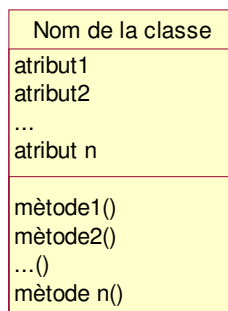
per definir el sistema, sinó del model de la realitat d'on s'extreu només la part que interessa.

Un altre dels objectius és la reusabilitat: l'aprofitament de la feina feta per a posteriors desenvolupaments. Aquest aprofitament pot arribar al 80% (vol dir el 80% del nou programa ha estat tret de les biblioteques) quan els elements han estat creats amb aquest concepte de reflectir la realitat convenientment acoblats.

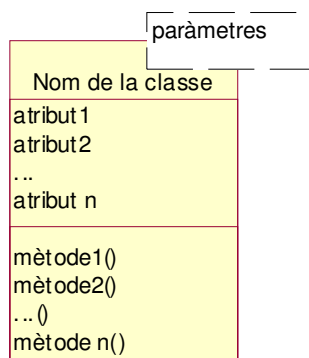
Un programa OO es compon per acoblament de components predefinitos, com es fa quan es munta un PC a base d'encaixar plaques. En alguna bibliografia podeu trobar el concepte del circuit integrat de *software* (*software-IC*).

**Notació gràfica i relacions:** per aquest projecte s'usa UML (llenguatge unificat de modelatge) i l'eina és el Rational Rose (opcionalment ArgoUML). De fet, UML és un estàndard per a OO força recent. Rational Rose és una eina CASE (enginyeria del software assistida per ordinador) que implementa UML, i és capaç de generar codi en diferents llenguatges, entre els quals hi ha Java. Tant UML com Rational Rose s'usen a les assignatures d'enginyeria del *software*.

Una classe és representada en UML:

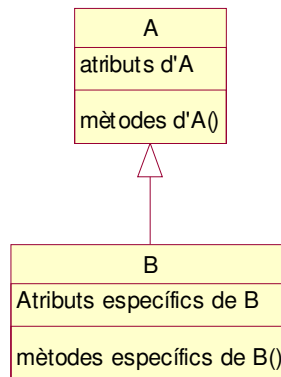


Si és abstracta el nom de la classe apareix en cursiva. Quan la classe és genèrica es representa així:



Les classes del diagrama es relacionen entre si de diverses maneres. La primera relació que veurem és la d'herència: es diu que la classe B hereta la classe A (o també que B és de la classe A amb característiques específiques o que A és una generalització de B). Per exemple, un alumne és una persona que estudia (o que hauria de fer-ho): alumne hereta persona.

A la classe B només hi veiem aquells atributs i mètodes que se li han afegit, però, a més d'aquest, té tots els de la classe A, encara que no es veuen explícitament. En UML es representa:



Una altra relació estàtica entre classes és la d'agregació. Quan una classe C té un o més atributs d'una altra classe D direm que D està agregada a C. S'entén que l'objecte de la classe D és una peça (*part*) del de la classe C que és un tot (*whole*). Aquesta relació es representa en UML —que n'anomena *composició*— d'una d'aquestes dues formes:

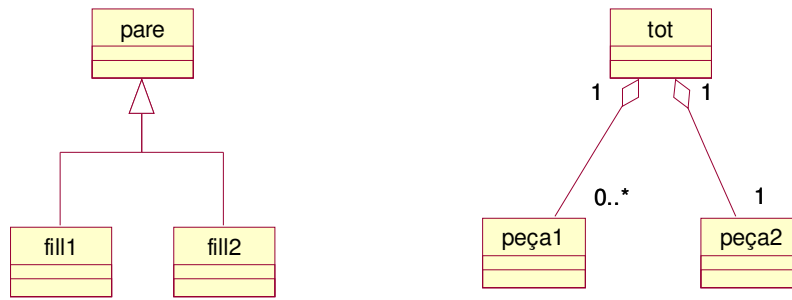


En una relació d'aquest tipus es pot explicitar un nom (el de l'atribut) o no. Una agregació sempre s'ha de quantificar, aquí s'indica que per cada D només hi ha una C i que a C hi ha un nombre no determinat (n) d'atributs que seran objectes de classe D. Com que el nombre d'objectes D que contindrà un objecte de la classe C en aquest cas és variable, l'atribut de C haurà de ser un contenidor (una taula, una llista, etc.) d'elements del tipus D.

Aquests detalls d'implementació, com el del contenidor o el del nom de l'atribut, no cal que s'explicitin fins que no sigui necessari. En canvi, convé fixar-se que quan hi ha una relació d'agregació l'atribut no apareix en la casella corresponent de la classe C, fóra redundant perquè ja s'ha fet explícit amb la relació.

Fixeu-vos també que aquestes dues relacions, agregació i herència, es representen en vertical, la classe heretada sobre de la qui l'hereta i la classe agregada sota de la classe on s'agrega.

La representació gràfica de la relació d'herència se sol fer de la classe pare a totes les filles, en forma de pinta, mentre que amb l'agregació no té sentit fer-ho així, sempre serà classe a classe, de forma que es pugui quantificar clarament cada una d'elles:



En les relacions d'herència és força normal de referir-se a les classes relacionades com si es tractes de relacions de emparentament familiar (pares, fills, avis, néts...)

Una altra possible relació entre dues classes és la de dependència (o missatge). És quan una classe *F* usa una altre classe *G* ja sigui com paràmetre d'una operació o sigui perquè *F* usa una operació de *G*. El primer tipus de dependència no cal explicitar-la si s'explicita com a paràmetre a la capçalera de l'operació. En cap cas *G* sap res de la classe *F*. La representació en UML és aquesta:



Finalment, parlarem de les associacions (o relacions d'instància). Les relacions definides fins aquí han de servir per modelitzar classes que representen diferents nivells d'importància o d'abstracció. Amb l'associació s'està modelitzant relacions entre classes que són iguals l'una que l'altre (pel seu paper en el sistema i pel nivell d'abstracció). Serveix per modelitzar les relacions estructurals. En les assignatures d'ES veureu que aquestes relacions són les que conformen el model entitat-relació o de Chen.

De fet, el que modelitzen és el comportament del sistema, les relacions poden variar segons l'estat del sistema. Són relacions existents entre determinades instàncies dels elements, per exemple, en un estat concret el professor Pau de l'assignatura ProPro té d'alumne a en Pere, el curs següent ja no l'hi té. Les estructures professor, assignatura i alumne són del mateix rang, només es relacionen valors concrets, instàncies concretes d'aquestes classes. Al professor Pau només l'interessa en Pere com alumne de ProPro i el que hi tingui a veure, però cap altre de les seves dades, la relació només té sentit per això.

Direm que una relació d'associació modelitza el fet que un objecte de classe *A* necessiti tenir coneixement d'un altre de classe *B* i viceversa. Aquesta relació també es quantifica sempre. En UML es representa:

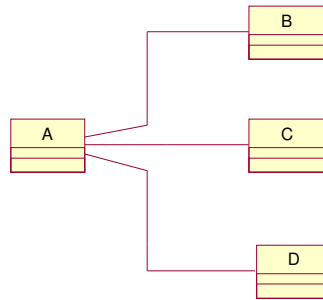


En aquest cas cada instància d'*A* es relaciona amb una sola instància de *B*, en canvi cada instància de *B* pot tenir coneixement d'una o varies instàncies d'*A* simultàniament. En alguns casos aquesta relació no és simètrica, es diu que només és navegable en un dels sentits. Quan la navegabilitat és limitada en UML es representa així, amb un cap de fletxa:



En aquest cas vol dir que des les instàncies d' $A$  es poden trobar les instàncies de  $B$  relacionades amb ell, però des de les de  $B$  no es pot accedir a les instàncies d' $A$  relacionades. És a dir, que només és possible navegar en un sentit de la relació. Això, la navegabilitat, també és un detall d'implementació a definir en el moment adequat, durant el disseny.

Per acabar, convé fixar-se que tant la dependència com l'associació, i especialment l'associació, se solen representar en horitzontal, amb les classes relacionades l'una al costat de l'altre, per dir-ho d'alguna manera.



Existeix un petit manual de Rational Rose a la pàgina de l'assignatura ES:E que usaré per explicar el que us convé saber sobre el funcionament d'aquest programa. (La pàgina de ES:E és <http://www.lsi.upc.edu/~es-e/>, llavors cal *clicar* sobre "Laboratori" i triar "Documents d'interès", el manual és la primera descàrrega que ofereix. N'hi ha una còpia a la meua plana <http://www.lsi.upc.edu/~ibarz/index.html>)

Per omplir les fitxes associades o especificacions d'una classe heu de tenir present que els elements de documentació més importants seran:

A la **documentació d'una classe** (a més dels seus atributs i operacions):

- Nom
- Descripció
- Tipus: abstracta, genèrica o instanciable. Per defecte és instanciable, que vol dir que hi haurà objectes d'aquesta classe.
- Visibilitat (quan se sàpiga)
- Persistència

A la **documentació d'un atribut**:

- Nom
- Tipus: classe
- Descripció
- Visibilitat (quan se sàpiga)
- Valor per defecte o inicial (si n'hi ha)
- Validació: valors admesos i altres condicionants del seu valor (relació amb altres atributs de la classe, amb atributs d'altres objectes, si ha de ser l'identificador de la classe, etc.)

Al web de l'assignatura hi ha un document enllaçat a "[Informació sobre els diagrames del primer lliurament i Rational Rose](#) (si es fa servir una altra eina, s'ha d'incloure la mateixa informació)" on es descriu el que és **normatiu adjuntar als diagrames**

**del primer lliurament.** Se us demana un llistat fet amb una eina de tractament de textos que ha de contenir:

Per a cada classe:

- Nom de la classe
- Breu explicació de la classe.
- Cardinalitat (*el nombre d'objectes de la classe que es preveu que poden existir*)\*.
- Per a cada atribut:
  - Nom de l'atribut
  - Valor inicial (*o per defecte*)\*.
  - Breu descripció de l'atribut (*quan el nom no és prou indicatiu*)
  - Estàtic (*quan és atribut de la classe, o sigui que és compartit per tots els objectes de la classe*).
- I per a cada associació o agregació\*:
  - Nom de les 2 classes implicades.
  - Breu descripció del significat de l'associació.

---

\* Opcional