Mòdul 5 Entorns de desenvolupament

UF1. Desenvolupament de programari
NF1. Desenvolupament de programari ('software')
A4. Característiques dels llenguatges més difosos

Característiques dels llenguatges més difosos

Existeixen molts llenguatges de programació diferents, fins al punt que moltes tecnologies tenen el seu llenguatge propi. Cada un d'aquests llenguatges té un seguit de particularitats que el fan diferent de la resta.

Els llenguatges de programació més difosos són aquells que més es fan servir en cadascun dels diferents àmbits de la informàtica. En l'àmbit educatiu, per exemple, es considera un llenguatge de programació molt difós aquell que es fa servir a moltes universitats o centres educatius per a la docència de la iniciació a la programació.

Els llenguatges de programació més difosos corresponents a diferents àmbits, a diferents tecnologies o a diferents tipus de programació tenen una sèrie de característiques en comú que són les que marquen les similituds entre tots ells.

Característiques de la programació estructurada

La programació estructurada va ser desenvolupada pel neerlandès Edsger W. Dijkstra i es basa en el denominat teorema de l'estructura. Per això utilitza únicament tres estructures: seqüència, selecció i iteració, essent innecessari l'ús de la instrucció o instruccions de transferència incondicional (GOTO, EXIT FUNCTION, EXIT SUB o múltiples RETURN).

D'aquesta forma les característiques de la programació estructurada són la claredat, el teorema de l'estructura i el disseny descendent.

Claredat

Hi haurà d'haver prou informació al codi per tal que el programa pugui ser entès i verificat: comentaris, noms de variables comprensibles i procediments entenedors… Tot programa estructurat pot ser llegit des del principi a la fi sense interrupcions en la seqüència normal de lectura.

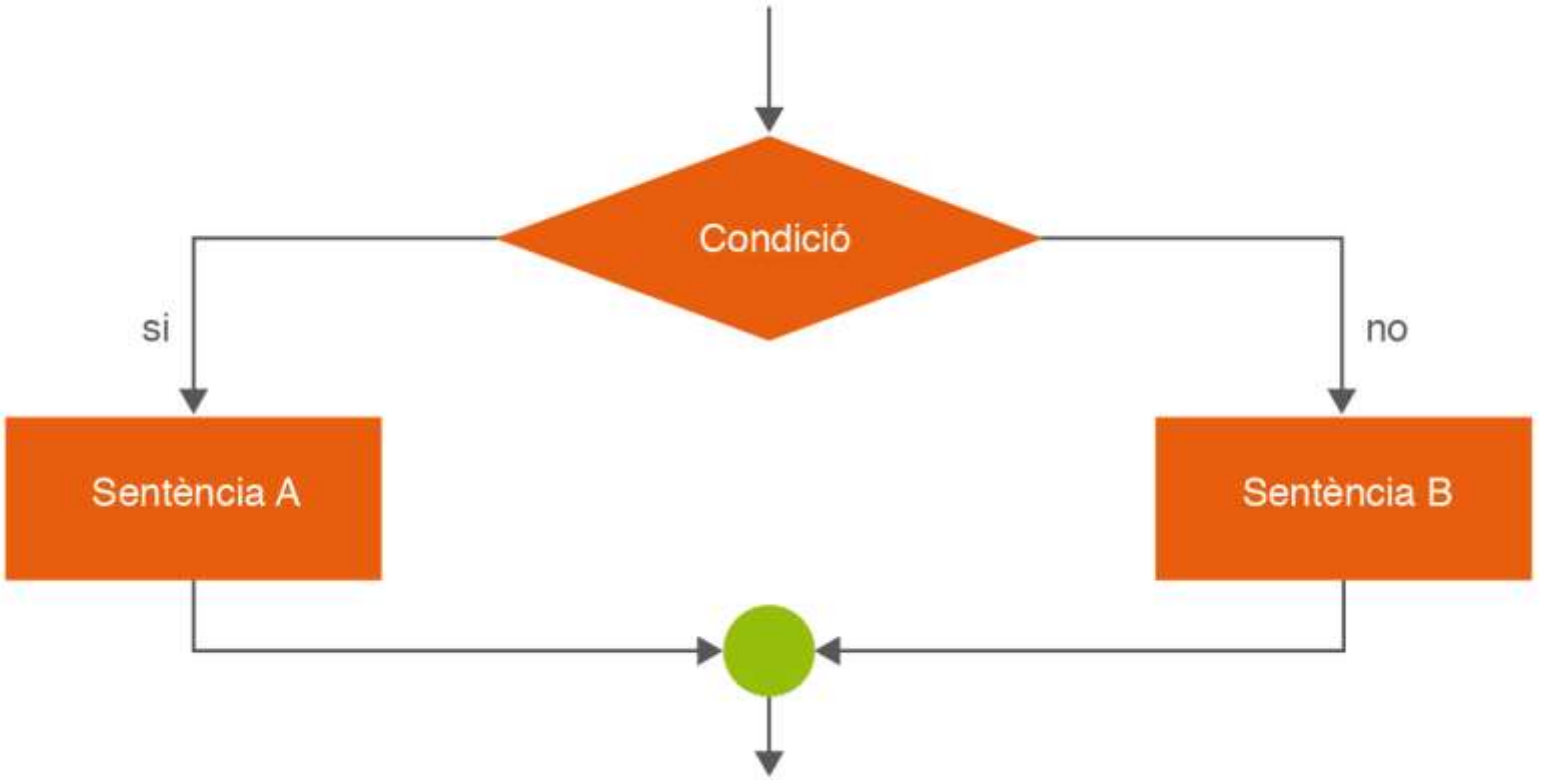
Teorema de l'estructura

Demostra que tot programa es pot escriure utilitzant únicament les tres estructures bàsiques de control:

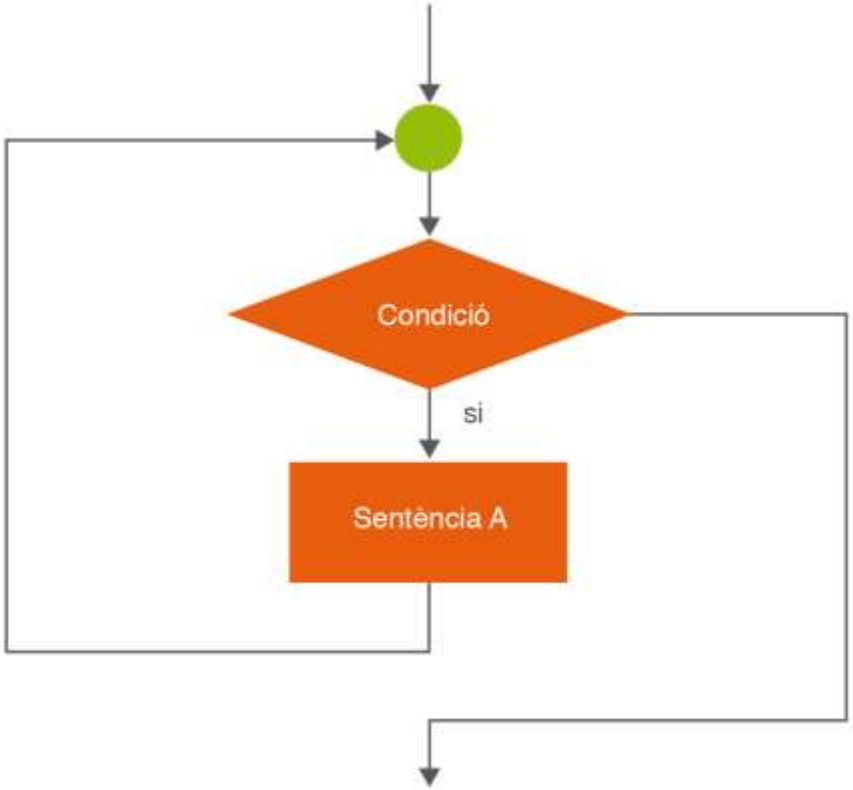
- Seqüència
- Selecció
- Iteració

Seqüència: instruccions executades successivament, una darrere l'altra. A la següent figura es pot observar un exemple de l'estructura bàsica de seqüència, on primer s'executarà la sentència A i, posteriorment, la B.

Selecció: la instrucció condicional amb doble alternativa, de la forma “si condició, llavors SentènciaA, sinó SentènciaB”. A la següent figura es pot observar un esquema que exemplifica l’estructura bàsica de selecció.



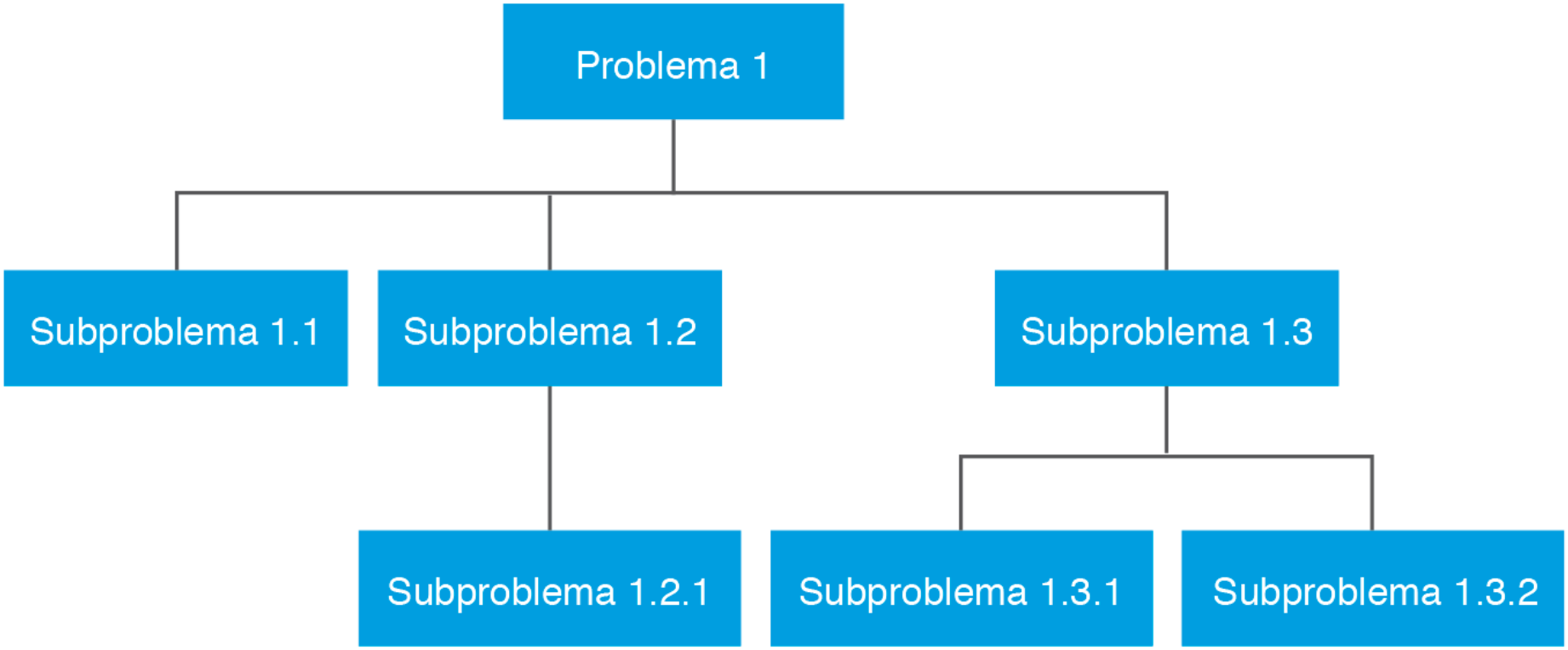
Iteració: el bucle condicional “mentre condició, fes SentènciaA”, que executa les instruccions repetidament mentre la condició es compleixi. A la següent figura es pot observar un esquema que exemplifica l’estructura bàsica d’iteració.



Disseny descendent

El disseny descendent és una tècnica que es basa en el concepte de “divideix i venceràs” per tal de resoldre un problema en l’àmbit de la programació. Es tracta de la resolució del problema al llarg de diferents nivells d’abstracció partint d’un nivell més abstracte i finalitzant en un nivell de detall.

A la següent figura es pot observar un exemple del disseny descendent. A partir del problema1 s’obtenen diversos subproblemes (subproblema 1.1, subproblema 1.2 i subproblema 1.3). La resolució d’aquests subproblemes serà molt més senzilla que la del problema original per tal com se n’ha reduït considerablement l’abast i la mida. De forma iterativa es pot observar com aquests subproblemes es tornen a dividir, a la vegada, en altres subproblemes.



La visió moderna de la programació estructurada introdueix les característiques de programació modular i tipus abstractes de dades (TAD).

Programació modular

La realització d'un programa sense seguir una tècnica de programació modular produeix sovint un conjunt enorme de sentències l'execució de les quals és complexa de seguir, i d'entendre, amb la qual cosa es fa gairebé impossible la depuració d'errors i la introducció de millores. Fins i tot, es pot donar el cas d'haver d'abandonar el codi preexistent perquè resulta més fàcil començar de nou.

Quan es parla de programació modular, ens referim a la divisió d'un programa en parts més manejables i independents. Una regla pràctica per aconseguir aquest propòsit és establir que cada segment del programa no excedeixi, en longitud, d'un pam de codificació.

En la majoria de llenguatges, els mòduls es tradueixen a:

- Procediments: són subprogrames que duen a terme una tasca determinada i retornen 0 o més d'un valor. S'utilitzen per estructurar un programa i millorar la seva claredat.
- Funcions: són subprogrames que duen a terme una determinada tasca i retornen un únic resultat o valor. S'utilitzen per crear operacions noves que no ofereix el llenguatge.

Tipus abstractes de dades (TAD)

En programació, el *tipus de dades* d'una variable és el conjunt de valors que la variable pot assumir. Per exemple, una variable de tipus booleà pot adoptar només dos valors possibles: vertader o fals. A més, hi ha un conjunt limitat però ben definit d'operacions que tenen sentit sobre els valors d'un tipus de dades; així, operacions típiques sobre el tipus booleà són AND o OR.

Els llenguatges de programació assumeixen un nombre determinat de tipus de dades, que pot variar d'un llenguatge a un altre; així, en Pascal tenim els *enters*, els *reals*, els *booleans*, els *caràcters*… Aquests tipus de dades són anomenats *tipus de dades bàsics* en el context dels llenguatges de programació.

Fins fa uns anys, tota la programació es basava en aquest concepte de tipus i no eren pocs els problemes que apareixien, lligats molt especialment a la complexitat de les dades que s'havien de definir. Va aparèixer la possibilitat de poder definir *tipus abstractes de dades*, on el programador pot definir un nou tipus de dades i les seves possibles operacions.

Exemple d'implementació d'un tipus abstracte de dades implementat en el llenguatge C

```
struct TADpila {
   int top;
   int elements[MAX_PILA];
}

void crear(struct TADpila *pila) {
   Pila.top = -1;
}

void apilar(struct TADpila *pila, int elem) {
   Pila.elements[pila.top++] = elem;
}

void desapilar(struct TADpila *pila) {
   Pila.top--;
}
```

Característiques de la programació orientada a objectes

Un dels conceptes importants introduïts per la programació estructurada és l'abstracció de funcionalitats a través de funcions i procediments. Aquesta abstracció permet a un programador utilitzar una funció o procediment coneixent només què fa, però desconeixent el detall de com ho fa.

Aquest fet, però, té diversos inconvenients:

- Les funcions i procediments comparteixen dades del programa, cosa que provoca que canvis en un d'ells afectin a la resta.
- Al moment de dissenyar una aplicació és molt difícil preveure detalladament quines funcions i procediments necessitarem.
- La reutilització del codi és difícil i acaba consistint a copiar i enganxar determinats trossos de codi, i retocar-los. Això és especialment habitual quan el codi no és modular.

L'orientació a objectes, concebut als anys setanta i vuitanta però estesa a partir dels noranta, va permetre superar aquestes limitacions.

L'**orientació a objectes** (en endavant, OO) és un paradigma de construcció de programes basat en una abstracció del món real.

En un programa orientat a objectes, l'abstracció no són els procediments ni les funcions, són els objectes. Aquests objectes són una representació directa d'alguna cosa del món real, com ara un llibre, una persona, una organització, una comanda, un empleat…

En un programa orientat a objectes, l'abstracció no són els procediments ni les funcions, són els objectes. Aquests objectes són una representació directa d'alguna cosa del món real, com ara un llibre, una persona, una organització, una comanda, un empleat…

Un **objecte** és una combinació de dades (anomenades atributs) i mètodes (funcions i procediments) que ens permeten interactuar amb ell. En OO, doncs, els programes són conjunts d'objectes que interactuen entre ells a través de missatges (crides a mètodes).

Els llenguatges de POO (programació orientada a objectes) són aquells que implementen més o menys fidelment el paradigma OO. La programació orientada a objectes es basa en la integració de 5 conceptes: abstracció, encapsulació, modularitat, jerarquia i polimorfisme, que és necessari comprendre i seguir de manera absolutament rigorosa. No seguir-los sistemàticament o ometre'ls puntualment, per pressa o altres raons, fa perdre tot el valor i els beneficis que aporta l'orientació a objectes.

Abstracció

És el procés en el qual se separen les propietats més importants d'un objecte de les que no ho són. És a dir, per mitjà de l'abstracció es defineixen les característiques essencials d'un objecte del món real, els atributs i comportaments que el defineixen com a tal, per després modelar-lo en un objecte de programari. En el procés d'abstracció no ha de ser preocupant la implementació de cada mètode o atribut, només cal definir-los.

En la tecnologia orientada a objectes l'eina principal per suportar l'abstracció és la **classe**. Es pot definir una classe com una descripció genèrica d'un grup d'objectes que comparteixen característiques comunes, les quals són especificades en els seus atributs i comportaments.

Encapsulació

Permet als objectes triar quina informació és publicada i quina informació és amagada a la resta dels objectes. Per això els objectes solen presentar els seus mètodes com a interfícies públiques i els seus atributs com a dades privades o protegides, essent inaccessibles des d'altres objectes. Les característiques que es poden atorgar són:

- Públic: qualsevol classe pot accedir a qualsevol atribut o mètode declarat com a públic i utilitzar-lo.
- Protegit: qualsevol classe heretada pot accedir a qualsevol atribut o mètode declarat com a protegit a la classe mare i utilitzar-lo.
- Privat: cap classe no pot accedir a un atribut o mètode declarat com a privat i utilitzar-lo.

Modularitat

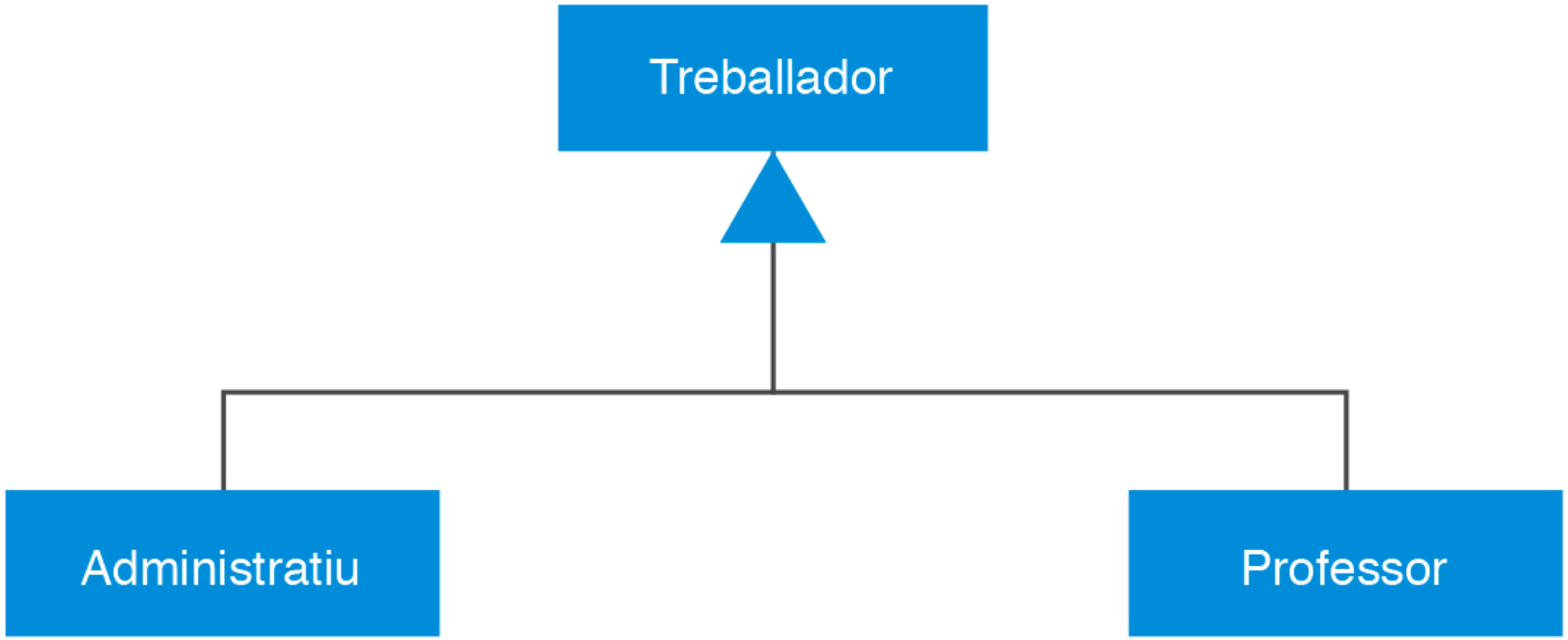
Permet poder modificar les característiques de cada una de les classes que defineixen un objecte, de forma independent de la resta de classes en l'aplicació. En altres paraules, si una aplicació es pot dividir en mòduls separats, normalment classes, i aquests mòduls es poden compilar i modificar sense afectar els altres, aleshores aquesta aplicació ha estat implementada en un llenguatge de programació que suporta la modularitat.

Jerarquia

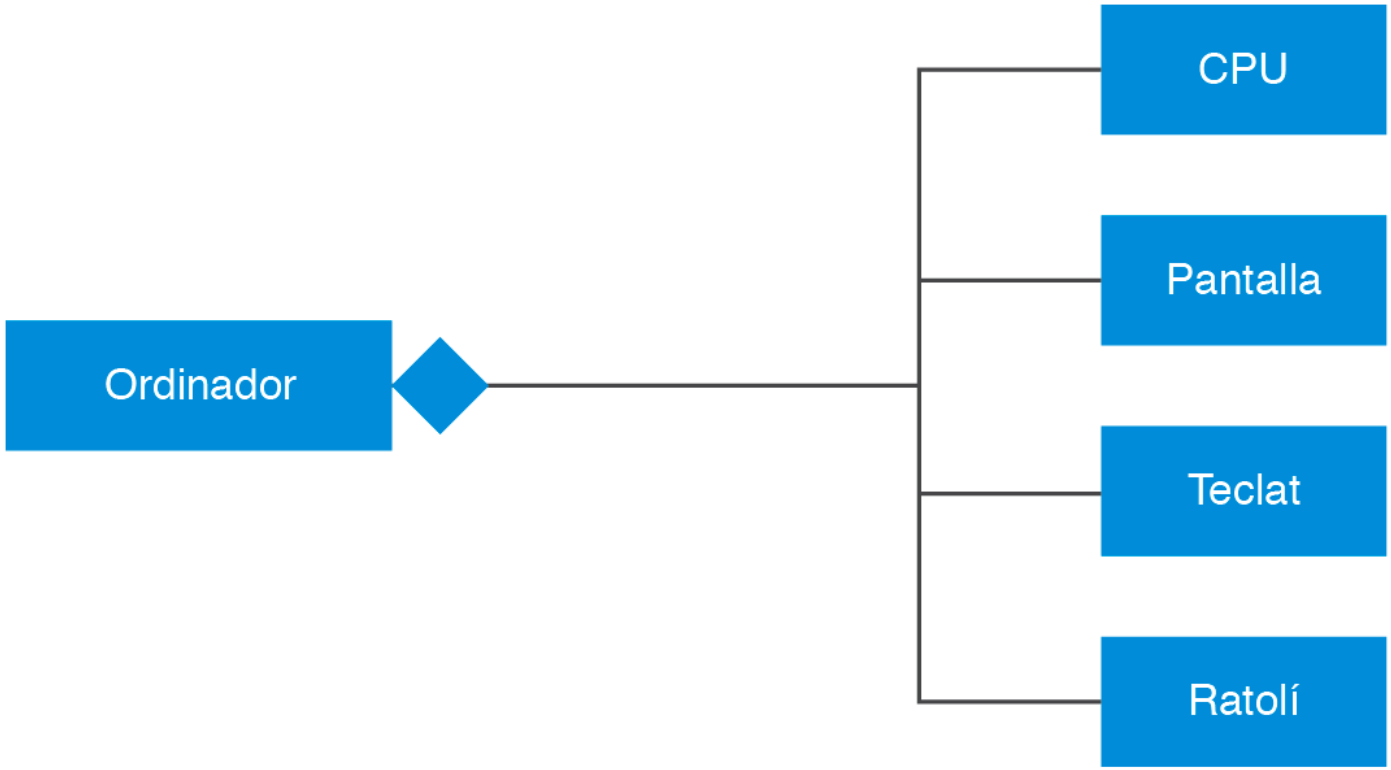
Permet l'ordenació de les abstraccions. Les dues jerarquies més importants d'un sistema complex són l'herència i l'agregació.

L'**herència** també es pot veure com una forma de compartir codi, de manera que quan s'utilitza l'herència per definir una nova classe només s'ha d'afegir allò que sigui diferent, és a dir, reaprofita els mètodes i variables, i especialitza el comportament.

Per exemple, es pot identificar una classe *pare* anomenada *treballador* i dues classes filles, és a dir dos subtipus de treballadors, *administratiu* i *professor*. Les classes *administratiu* i *professor* hereten de la classe *treballador*.



L'**agregació** és un objecte que està format de la combinació d'altres objectes o components. Així, un ordinador es compon d'una CPU, una pantalla, un teclat i un ratolí, i aquests components no tenen sentit sense l'ordinador. A la següent figura es pot observar un exemple d'agregació en què la classe ordinador està composta per les altres quatre classes.



El polimorfisme

És una característica que permet donar diferents formes a un mètode, ja sigui en la definició com en la implementació.

La sobrecàrrega (*overload*) de mètodes consisteix a implementar diverses vegades un mateix mètode però amb paràmetres diferents, de manera que, en invocar-lo, el compilador decideix quin dels mètodes s'ha d'executar, en funció dels paràmetres de la crida.

Un exemple de mètode sobrecarregat és aquell que calcula el salari d'un treballador en una empresa. En funció de la posició que ocupa el treballador tindrà més o menys conceptes a la seva nòmina (més o menys incentius, per exemple).

El mateix mètode, que podríem anomenar *CàlculSalari* quedarà implementat de forma diferent en funció de si es calcula el salari d'un operari (amb menys conceptes en la seva nòmina, la qual cosa provoca que el mètode rebi menys variables) o si es calcula el salari d'un directiu.

La sobreescriptura (*override*) de mètodes consisteix a reimplementar un mètode heretat d'una superclasse exactament amb la mateixa definició (incloent nom de mètode, paràmetres i valor de retorn).

Un exemple de sobrecàrrega de mètodes podria ser el del mètode Area(). A partir d'una classe Figura que conté el mètode Area(), existeix una classe derivada per a alguns tipus de figures (per exemple, Rectangle o Quadrat).

La implementació del mètode Area() serà diferent a cada una de les classes derivades; aquestes poden implementar-se de forma diferent (en funció de com es calculi en cada cas l'àrea de la figura) o definir-se de forma diferent.