Introducció al paradigma de la programació orientada a objectes

David García Solórzano

PID_00235245

UOC
Universitat Oberta de Catalunya

Índex

Introducció

Aquest mòdul situa l'assignatura dins del grau. Per a això, explicarem el punt de partida a partir del qual vam començar aquesta assignatura, que no és altre que el paradigma de la programació estructurada que vam aprendre en l'assignatura anterior de programació, Fonaments de programació en el cas de Telecomunicacions i Programació en el de Multimèdia. Un cop ens haguem situat, veurem com s'arriba al paradigma de la programació orientada a objectes (POO), element principal d'estudi d'aquesta assignatura.

Aquest mòdul és introductori i, per tant, molts dels conceptes que es comenten seran vistos amb més detall en futurs mòduls.

Objectius

L'objectiu principal d'aquest mòdul és contextualitzar l'assignatura. Concretament:

1. Ser conscient del punt de partida després d'haver cursat l'assignatura de programació anterior.

2. Conèixer el concepte de *paradigma de programació*, fent èmfasi en la programació estructurada –ja coneguda per l'estudiant– i la seva evolució cap a la programació orientada a objectes (PPO) –paradigma que s'estudiarà en aquesta assignatura.

3. Entendre el raonament que se segueix a l'hora de programar mitjançant el paradigma POO.

4. Comprendre, en general, els beneficis que ens aporta el paradigma de la programació orientada a objectes.

5. Ser conscients de les dificultats a les quals un programador novell, com nosaltres, s'enfronta quan intenta aprendre el paradigma de la programació orientada a objectes.

1. Context: programació estructurada

Després de cursar la primera de les assignatures de programació –Fonaments de programació o Programació–, segur que sou conscients que un programa ha de ser dissenyat (és a dir, pensat amb deteniment) abans de codificar-lo (és a dir, d'escriure'n el codi). Això és encara més cert com més complexa és la tasca que ha de fer el nostre programa. És a dir, no podem posar-nos a codificar directament un programa de la complexitat que sigui.

Per a garantir que els programes complexos funcionen, és a dir, fan el que s'espera d'ells, són fiables i estan ben escrits, va aparèixer a finals dels anys cinquanta una nova disciplina anomenada **enginyeria del programari**. Aquesta disciplina proposa mètodes i teories tant per a analitzar problemes complexos com per a dissenyar programes que resolguin aquests problemes. Cada conjunt de mètodes i teories que determinen una manera específica d'analitzar i resoldre un problema s'anomena enfocament de programació o, més formalment, **paradigma de programació**.

> Una definició més formal de paradigma de programació podria ser la de col·lecció de conceptes, mètodes i teories que guien el procés de construcció d'un programa i en determinen l'estructura.

Així doncs, un paradigma de programació determina la manera en què pensem i formulem els programes.

Durant els anys setanta, es va començar amb un paradigma anomenat **programació estructurada**. Aquest paradigma es basa a veure un programa –que és el problema complex que cal resoldre– com un conjunt de diferents subprogrames que fan tasques concretes. Aquests subprogrames poden estar dividits, al seu torn, en més subprogrames, i així successivament. Segons com es fa la creació dels subprogrames, es parla d'un enfocament *top-down* (o disseny descendent) o *bottom-up* (o disseny ascendent). Bàsicament:

1) ***Top-down***: per a resoldre un problema complex, es descompon el problema en diversos problemes més petits i senzills anomenats subproblemes. Per a resoldre cada subproblema, es treballa cadascun d'ells per separat considerant-los un nou problema que pot ser al seu torn descompost en problemes més petits, i així successivament. Finalment s'arriba a problemes que poden ser resolts directament sense necessitat de descompondre més.

Això implica tenir diferents nivells de descomposició, que van d'allò general –el problema complex inicial– a allò particular o a un nivell de detall més gran –el subproblema de nivell més baix de descomposició.

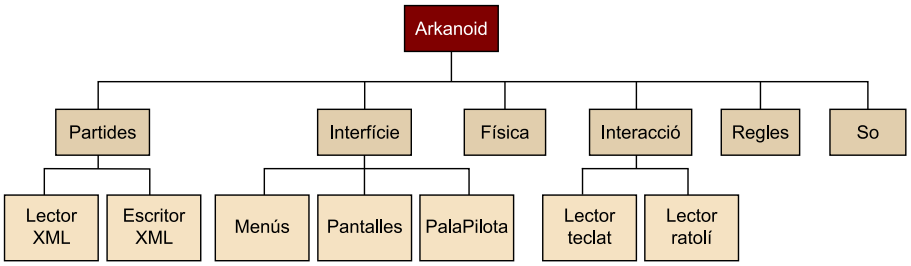
2) ***Bottom-up***: aquest enfocament comença per baix, és a dir, amb problemes que ja se sap com resoldre (i per als quals es pot reutilitzar codi ja fet). A partir d'aquí, es treballa cap amunt (unint trossos/blocs) a la recerca d'una solució per al problema complex inicial. Informalment es podria dir que és un enfocament d'estil LEGO®: tinc petites peces –per exemple, maons– que, unides, creen un objecte més gran, p. ex. una casa. Dit d'una altra manera, aquest enfocament consisteix a començar pels subproblemes de nivell més baix i anar pujant fins a arribar al problema inicial.

En aquest punt, si diem que el problema complex inicial és el programa, també podem dir que un subproblema és un subprograma. Aquests subprogrames se solen anomenar **mòduls**, i és per això que de vegades també es fa servir el nom de **programació modular** per a referir-se a la programació estructurada.

> Un mòdul es pot definir com una col·lecció de constants, variables, funcions i accions (és a dir, funcions que no retornen res) relacionades entre si que estan agrupades en un mateix fitxer.

Com es pot veure a la figura següent, després de descompondre el problema en subproblemes (mòduls), s'obté una estructura piramidal o jeràrquica en què els mòduls dels nivells superiors s'encarreguen de les tasques de coordinació, lògica de l'aplicació i manipulació dels mòduls inferiors. Per la seva banda, els mòduls inferiors fan tasques de càlcul, tractament i entrada/sortida d'informació. És a dir, els mòduls superiors fan tasques més generals/globals, mentre que els mòduls inferiors s'encarreguen de tasques més concretes.

Diagrama jeràrquic incomplet que mostra possibles mòduls (o subprogrames) del videojoc (o programa) Arkanoid



El nombre de mòduls que té un programa i com s'estructura cada un d'ells depèn del programador. L'única cosa que s'ha de tenir en compte és que cada mòdul ha de complir una funcionalitat del programa. Així doncs, si el nostre programa és un videojoc com el que es mostra a la figura anterior, un mòdul es podria encarregar de carregar i desar partides; un altre de totes les gestions relacionades amb el fet de pintar la interfície gràfica; un altre de controlar la

física dels objectes del joc, etc. Els diferents mòduls poden interactuar entre si. Per exemple, el mòdul de física dirà al mòdul de pintar on dibuixar una pilota que segueix un moviment parabòlic.

El paradigma de programació estructurada, amb independència de l'enfocament (és a dir, *top-down* o *bottom-up*), té els avantatges següents respecte al fet de no seguir cap paradigma concret:

- **La programació és més senzilla**, ja que es pot ajornar el desenvolupament d'algunes tasques en benefici del problema global.

- **Facilita el treball en grup**. Diversos programadors poden tractar el problema complex inicial encarregant-se cadascun d'ells d'un mòdul concret.

- **El manteniment és més senzill** de fer, ja que es poden modificar segments de codi (és a dir, subprogrames/mòduls) de manera independent, sense que calgui tocar la resta del programa.

- **La reutilització de codi ja escrit és més fàcil** tant en el mateix programa com en altres gràcies a l'organització basada en mòduls. Idealment, cada mòdul s'hauria de poder fer servir en qualsevol programa. Parlant de manera informal, podem dir que un mòdul hauria de ser un element *plug and play*.

- **Augmenta la llegibilitat del codi**. És millor tenir el codi organitzat/repartit en diferents mòduls que segueixen una certa lògica que tenir un únic mòdul amb milers de línies de codi.

Normalment, quan s'aborda un problema seguint el paradigma de programació estructurada, se sol combinar l'enfocament *top-down* amb l'enfocament *bottom-up*, encara que l'enfocament *top-down* té predominança. Alguns motius pels quals es combinen els dos enfocaments són:

1) Si el problema, a més de complex, és nou per al programador, sol ser més senzill pensar de manera descendent (*top-down*).

2) Si durant el descens es detecta algun mòdul que ja es té fet d'un altre programa anterior (p. ex. desar i carregar partides), es pot mirar d'aprofitar (és a dir, reutilitzar) i, en conseqüència, adaptar els mòduls superiors perquè en facin ús.

3) Si se segueix exclusivament un disseny descendent (*top-down*), les funcions desenvolupades de nivell més baix depenen molt del problema que volen resoldre i difícilment poden ser aprofitades per a altres programes.

4) Si s'ha seguit exclusivament un disseny descendent (*top-down*) i en el futur hi ha canvis funcionals en el programa, aquest queda molt afectat per aquests canvis i pot obligar a reescriure bona part del programa.

Independentment de l'enfocament predominant –*top-down* o *bottom-up*–, el grau d'èxit que podem tenir quan seguim el paradigma de programació estructurada és directament proporcional a la capacitat d'**abstracció** que siguem capaços d'aplicar al problema complex al qual ens enfrontem. La nostra capacitat d'abstracció ens permet reconèixer els diferents subproblemes del problema complex inicial i dividir-los (és a dir, enfocament *top-down*) o ajuntar-los (és a dir, enfocament *bottom-up*) segons sigui necessari.

Enfocament *bottom-up*

Com veiem, l'enfocament *top-down* té més inconvenients que avantatges. Per això, la programació orientada a objectes segueix un enfocament *bottom-up*, com ja veurem.

Tot el que hem comentat fins ara us hauria de ser familiar, ja que a l'assignatura de programació anterior vau implementar programes seguint el paradigma de programació estructurada i, més concretament, seguint un enfocament descendent (*top-down*). Si no ho recordeu, contesteu aquestes preguntes:

- **He fet servir mòduls?** Sí. Quan escrivies un programa en C o PHP organitzaves el codi en fitxers amb l'extensió .c o .php que contenien variables i funcions.

- **He seguit un enfocament descendent (*top-down*)?** Sí. A més de mòduls, quan escrivies una funció *A* i aquesta feia moltes tasques, la subdividies en funcions més senzilles que cridaves des de la funció *A*.

- **He aplicat abstracció?** Sí. A cada nivell de descomposició, has estat capaç de considerar només aquells aspectes necessaris per a analitzar el problema i prendre decisions. Per exemple, decidies sobre qüestions com: en quants mòduls dividies el programa, què feia cada mòdul, quines funcions implementaves en cada mòdul, quin encapçalament tenia cada funció, quines variables necessitaves, etc.

Enllaç d'interès

Per a entendre millor el concepte d'abstracció, llegiu l'apartat *Create by abstracting* de l'article web següent: http://worrydream.com/LearnableProgramming/

Cal tenir ben assimilats tots aquests conceptes –abstracció, modulació, reutilització, etc.– abans de començar aquesta assignatura, ja que la programació orientada a objectes no només els utilitza, sinó que els potencia i, a més, afegeix nous elements.

2. I arriba la programació orientada a objectes

Els llenguatges de programació, generalment, estan lligats a un paradigma de programació (o l'afavoreixen). D'aquesta manera, el llenguatge C està clarament alineat amb la programació estructurada.

Si bé la programació estructurada és útil, la complexitat que han anat adquirint els programes al llarg dels anys ha obligat a pensar nous paradigmes. Així és com a la dècada dels vuitanta va néixer un nou paradigma anomenat **programació orientada a objectes (o POO)**.

Si reprenem el paradigma que ja coneixem –la programació estructurada–, ens adonarem que per a un programador que vulgui fer servir un mòdul que no hagi programat ell mateix, no és tan important saber com està implementat aquest mòdul sinó saber què és el que pot fer amb ell i com ha d'utilitzar-lo. Així doncs, al programador li interessa saber quines funcions pot usar, què fan, com es fan servir i què ofereixen. Per contra, no li interessa saber si el programador que va fer el mòdul, quan va escriure una funció, va utilitzar una sèrie de sentències *if* o es va estimar més un *switch*, o si va fer servir un *while* en comptes d'un *for*.

Seguint amb el mateix raonament, molts mòduls inclouen variables globals que poden ser usades per altres mòduls, accedint-hi directament, o bé a través de funcions proporcionades pel mòdul que les declara. Al començament dels vuitanta es va popularitzar la segona manera de treballar, és a dir, els mòduls d'un programa no accedien directament a les variables d'un altre mòdul, sinó que ho feien per mitjà de funcions proporcionades pel mòdul que contenia les variables que calia utilitzar. Això té una sèrie d'avantatges respecte a accedir-hi directament, com per exemple:

- S'assegura la **consistència** de la variable. Per exemple, si una variable és una matriu, en fer servir una funció per a accedir-hi, aquesta funció comprovarà si l'índex al qual es vol accedir (passat per paràmetres) és correcte o no. A més es garanteix que tots els accessos a l'*array* fets per mitjà de la funció es facin de la mateixa manera i se'n facin les mateixes comprovacions.

- **S'eximeix de responsabilitats** al programador que fa servir el mòdul, ja que quan utilitza les funcions proporcionades, la responsabilitat recau en qui les ha programat.

- **Facilitat de manteniment**. Si es canvia el tipus de variable o hi ha cap error en una de les funcions proporcionades pel mòdul, només cal canvi-

ar el codi del mòdul que la conté. D'aquesta manera, els canvis queden reflectits automàticament en altres mòduls que l'utilitzin.

Com es pot veure, a poc a poc va sorgir la tendència d'anar ocultant informació sobre la implementació dels mòduls. O dit d'una altra manera, de fer d'un mòdul una caixa negra de la qual se sabés el mínim. Aquest costum d'ocultar informació i de controlar-ne l'accés –especialment dades, és a dir, variables– està íntimament relacionat amb el concepte d'**encapsulació**.

Vegeu també

El concepte d'encapsulació es tracta en el mòdul «Encapsulació i extensió» d'aquesta assignatura.

Atesos els avantatges, els programadors tenien cada vegada més interès a poder escriure mòduls que facilitessin l'encapsulació, i així és com van sorgir nous llenguatges (p. ex. C ++) que donaven suport a aquesta tècnica i a nous paradigmes de programació més adequats. D'aquesta manera va sorgir el paradigma de la programació orientada a objectes (POO).

3. De la programació estructurada a la programació orientada a objectes

3.1. Introducció als conceptes bàsics de la programació orientada a objectes

De manera molt introductòria –ja ho veurem tot amb més detall al llarg de l'assignatura–, cal destacar que l'element principal de la programació orientada a objectes (POO) és l'**objecte**. La POO modela la funcionalitat d'un sistema (és a dir, el problema que s'ha de resoldre) intentant assemblar-se el màxim possible a la realitat. Mira d'imitar la realitat definint un conjunt d'objectes que interactuen entre si enviant i responent **missatges**. Gràcies a la col·laboració entre objectes, el programa és capaç de fer la funcionalitat esperada (o de resoldre el problema plantejat).

De la mateixa manera que en el llenguatge C hi ha els tipus *integer* (*int*, sencer), *character* (*char*, caràcter), etc., per a definir una variable, en els llenguatges que admeten la POO, el tipus dels objectes és una **classe**. Una classe s'assembla a allò que en programació estructurada anomenem mòdul, ja que una classe conté dades (en forma de variables) i funcions. Les dades (variables) d'una classe se solen denominar **atributs**, mentre que les seves funcions reben el nom de **mètodes**. De la mateixa manera, tant els atributs com els mètodes s'acostumen a anomenar **membres d'una classe**.

Així doncs, quan definim un objecte, n'hem d'indicar el tipus. Aquest no serà ni un *int*, ni un *char*, sinó una classe. De fet, moltes vegades, els objectes s'anomenen ***instàncies*** d'una classe o, simplement, instàncies (de l'anglès, *instance*), encara que la traducció correcta al català hauria de ser «exemple/cas». Per tant, un objecte és un exemple/cas particular d'una classe. És important entendre que podem tenir tants objectes/instàncies com vulguem d'una mateixa classe.

Com hem dit, les classes –que defineixen el tipus dels objectes– hauran de modelar la realitat del problema que tractem, la qual cosa ens obliga a definir les classes mitjançant una estratègia totalment diferent de la que seguim quan definim mòduls en la programació estructurada. Si en la programació estructurada definíem els mòduls seguint un criteri, més o menys lògic, com agrupar funcions que tractin una mateixa tasca (p. ex. pintar, control de regles, control de físiques, etc.), en la POO, aquesta estratègia no ens servirà (del tot).

A continuació, vegem un exemple senzill que ens permetrà entendre els conceptes que han anat apareixent.

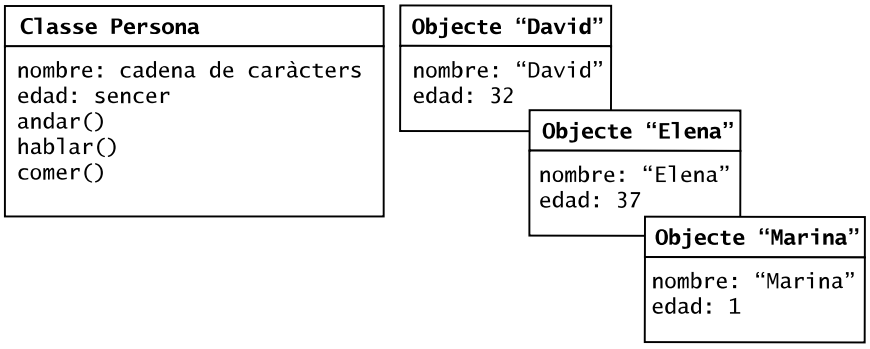
Elena, *Marina* i *David* són **objectes/instàncies** (exemples/casos) de la **classe** *Persona*. Una *Persona* té unes dades i unes funcions/comportaments, és a dir, uns **atributs** i uns **mètodes**. Exemples d'atributs que totes les persones compartim són: nom, cognom, data de naixement, lloc de naixement, gènere, raça, pes i alçada. Mentre que exemples de mètodes que compartim totes les persones poden ser: parlar, escoltar, caminar, menjar, augmentar l'estatura, augmentar el pes i reduir el pes, entre d'altres.

Un cop sabem que *Elena*, *Marina* i *David* són objectes de *Persona*, és fàcil intuir que cadascun d'ells tindrà uns valors diferents per als atributs de la classe *Persona* i faran certes accions (mètodes) d'una manera diferent, encara que es diguin igual. Per exemple, el valor de l'atribut *nom* de la classe *Persona* ja és diferent per a cadascun dels tres objectes, donat que un objecte té el valor *Elena*, un altre *Marina* i un altre *David*. Així mateix, mentre que el valor de l'atribut gènere per a *David* és *home*, per a *Elena* i *Marina* el valor és *dona*.

Si *Marina* és un nadó de tretze mesos, és fàcil imaginar que els valors per als atributs *estatura* i *pes* no poden ser iguals que els dels adults *Elena* i *David*, i, que al seu torn, els valors d'aquests atributs seran diferents entre *Elena* i *David*. De la mateixa manera, la forma de parlar de *Marina* serà molt diferent de la d'*Elena* i *David*, ja que encara no té el vocabulari suficient per a expressar-se.

D'altra banda, si imaginem que *David* és el pare de *Marina*, podria demanar-li a *Marina* que mengés enviant-li el missatge «*Marina, menja*» (és a dir, mitjançant el mètode *menjar* de *Marina*). Al seu torn, *Marina* podria respondre menjant (p. ex. retornant el seu mètode *menjar* un *true* o un *1*) o bé negant-s'hi (p. ex. retornant un *false* o *0*). D'aquesta manera, veiem com interactuen (es comuniquen o col·laboren) els diferents objectes.

De manera simplificada i visual, podem veure aquest exemple en la figura següent.



En la classe hem indicat el tipus simple/bàsic (és a dir, sencer, cadena de caràcters, booleà, etc.) dels atributs, mentre que els mètodes els hem distingit obrint i tancant parèntesis al costat del seu nom. En el cas dels objectes, posem el valor que té cada atribut.

3.2. Donat un problema real, quin és el procés que cal seguir?

La programació orientada a objectes (POO) intenta modelar el món real (o el problema real que volem resoldre) a partir de dos elements: els objectes i les classes.

El dissenyador del programa segueix un procés d'abstracció en què, a partir d'elements concrets (és a dir, els objectes), intenta generalitzar fins a arribar a diferents plantilles (és a dir, les classes). Concretament, el procés que sol seguir és:

1) Identificar els objectes que interactuen en el problema. Davant un text/enunciat del problema, els objectes solen ser substantius o sintagmes nominals.

En l'exemple del subapartat 3.1, hem identificat *Elena*, *Marina* i *David* com a objectes.

2) Analitzar els objectes identificats **en la recerca de característiques i accions/funcionalitats comunes**. Davant un text/enunciat, les característiques es corresponen amb substantius o sintagmes nominals, mentre que les accions se solen correspondre amb verbs o sintagmes verbals. Com veurem més endavant, les característiques passaran a anomenar-se atributs i les accions es diran mètodes.

En l'exemple del subapartat 3.1, hem identificat *nom* i *edat* com a atributs, mentre que *caminar*, *parlar* i *menjar* eren els mètodes.

3) Identificar les classes que agrupen els diferents objectes. Cada classe tindrà les característiques i accions/funcionalitats comunes dels objectes que representa. Davant un text/enunciat, les classes, si apareixen, solen ser substantius o sintagmes nominals. Si no apareixen, hem d'assignar-li un nom inventat per nosaltres que tingui una certa lògica.

En l'exemple del subapartat 3.1, hem identificat una única classe anomenada *Persona*.

4) Assignar cada objecte a una classe.

En l'exemple del subapartat 3.1, hem identificat que els objectes *Elena*, *Marina* i *David* compartien atributs i mètodes i que, per tant, pertanyien a la mateixa classe, en aquest cas, la classe *Persona*.

Com podem veure, el procés seguit es basa en un enfocament *bottom-up*, ja que anem des del més concret, o nivell baix (és a dir, els objectes), fins al més general, o nivell alt (és a dir, les classes).

Finalment, cal tenir en compte que no tots els substantius i verbs del text/enunciat són objectes, atributs, mètodes o classes. Es poden donar els casos següents:

- **Redundància (o sinònims)**: un objecte/classe/atribut/mètode és anomenat d'una manera diferent dins el text. És a dir, apareixen dos o més substantius/verbs per a indicar el mateix element. En aquests casos hem de triar una de les maneres de referir-nos a l'element.

- **Irrellevància**: pot donar-se que algun element que hem identificat, pel que sigui, tingui poc o res a veure amb el problema que volem solucionar. És a dir, no ens és útil per al problema/context en qüestió. Dit d'una altra manera, ens és irrellevant.

- **Rols**: hi ha substantius i verbs que indiquen quin és el paper d'una classe o objecte dins el problema. Unes vegades, aquesta informació és supèrflua i, per tant, la podem descartar. D'altres, els rols es corresponen amb atributs, operacions o relacions que són importants per a nosaltres.

Si ens fixem en l'exemple del subapartat 3.1, podem veure que hem seguit el procés *bottom-up* que acabem d'explicar. Fixeu-vos que hem anat de baix (objectes: *Elena*, *Marina* i *David*) cap amunt (classe: *Persona*). Concretament, primer hem identificat tres objectes –*Elena*, *Marina* i *David*– i a partir d'aquí hem detectat que aquests tres objectes compartien una sèrie d'atributs i mètodes, la qual cosa ens indicava que *Elena*, *Marina* i *David* podrien tractar-se del mateix tipus d'objecte, és a dir, que podrien pertànyer a la mateixa classe. Aquesta classe, l'hem anomenat *Persona*.

Aquesta identificació dels «elements compartits» ens requereix exercir un nivell d'abstracció més gran del que solem fer en el paradigma de programació estructurada.

Podríem haver fet una classe per a *Elena*, una altra per a *Marina* i una tercera per a *David*, però en comptes d'això, hem fet un exercici d'abstracció i hem buscat elements comuns el resultat dels quals ha estat una classe comuna anomenada *Persona*.

Aquesta manera de pensar un programa ens ajuda a fer un disseny millor organitzat, més reduït pel que fa a línies de codi i, per tant, més fàcil de mantenir i escalar.

A diferència de la programació estructurada que se centra en les funcions, la POO se centra en l'estructura de les dades (és a dir, com s'organitzen). És imprescindible comprendre i assimilar aquest canvi, encara que pugui semblar subtil, per a tenir èxit amb la POO.

4. Beneficis de la programació orientada a objectes

La POO ha aportat nous avantatges o millores pel que fa a la programació estructurada. En destaquem alguns:

- **Més facilitat** per a abordar problemes complexos:
  - **Naturalitat**: l'anàlisi i el disseny que divideixen un domini en classes/objectes concorden més amb la realitat que una descomposició funcional (com fa la programació estructurada).

  - **Modularitat**: gràcies a la definició de classes, els objectes són autocontinguts i tenen definida clarament la manera de comunicar-se amb altres objectes.

- **Seguretat**: la POO mitjançant l'encapsulació permet controlar l'accés als atributs i mètodes. D'aquesta manera, la consistència de les dades (és a dir, atributs) és més gran, ja que es pot evitar que s'hi accedeixin directament. Així mateix, en una classe, un mètode que fa una tasca complexa pot estar dividit en mètodes auxiliars més senzills que no han de ser ni accedits per altres objectes ni coneguts per algú que no sigui el programador de la classe.

- **Reutilització**: les classes ben dissenyades poden ser utilitzades com a base d'altres sistemes. Per a facilitar la reutilització, la POO ha definit principalment un mecanisme anomenat *herència*. L'herència permet sobretot dues coses:
  - Crear un mòdul (més ben dit, una *classe*) a partir d'un de ja existent, de manera que estén les funcionalitats del mòdul reutilitzat sense afectar-lo.

  - Definir i utilitzar de manera clara classes funcionalment incompletes, però que ajuden a definir el problema.

- **Escalabilitat**: gràcies a l'herència i la naturalesa del paradigma en si, permet que l'esforç no augmenti exponencialment amb la mida i la complexitat del projecte ni amb un canvi d'especificacions.

- **Comprensió**: millora la llegibilitat del codi en relació amb la programació estructurada. Com que les dades i els procediments que conformen els objectes estan encapsulats en un mateix compartiment (és a dir, la classe), els objectes poden ser desenvolupats i provats de manera independent. Així mateix, en provar un objecte, se'n prova de passada la classe.

5. Enfrontar-se a la programació orientada a objectes

En aquest apartat veurem els inconvenients que la POO té per a un programador novell:

- **Corba d'aprenentatge costosa**: aprendre el paradigma de POO i un llenguatge orientat a objectes concret és més complex que aprendre un llenguatge imperatiu tradicional basat en un paradigma de programació estructurada.

- **Canvi d'enfocament**: generalment s'aprèn a programar seguint un enfocament *top-down* –com ha estat el vostre cas–, però la POO es basa en un enfocament *bottom-up*. La POO se centra en els conceptes (abstraccions) del domini de l'aplicació. La capacitat d'abstracció d'un programador novell sol ser una de les seves debilitats.

- **Reutilitzacions ineficients**: les classes en si mateixes no són reutilitzables; cal programar pensant en la reutilització. Això requereix fer un pas més en la vostra capacitat d'abstracció.

Tot això es resumeix en el fet que des d'aquest moment heu de ser conscients que aquesta assignatura us exigirà moltes hores de dedicació davant de l'ordinador, perquè com se sol dir: «s'aprèn a programar programant».

Resum

En aquest mòdul hem vist en l'apartat 1 el punt de partida un cop superada l'assignatura Fonaments de programació o Programació, és a dir, el paradigma de programació estructurada. Hem recordat en què consisteix l'enfocament descendent (*top-down*) utilitzat en l'assignatura anterior i hem comentat breument l'enfocament ascendent (*bottom-up*) que serà la base de la programació orientada a objectes (POO).

Per fer que el pas del paradigma de programació estructurada *top-down* al paradigma de la programació orientada a objectes (afí a un enfocament *bottom-up*) sigui com més natural i simple possible, hem vist en l'apartat 2 les necessitats que van originar l'aparició d'aquest segon paradigma i, en l'apartat 3, les similituds que tenen o poden establir-se de manera informal entre els dos. Això ens ha permès, d'una banda, introduir conceptes i termes utilitzats en la POO –per exemple: objecte, classe, etc.–, i, de l'altra, conèixer el procés d'abstracció que hem de seguir a l'hora de dissenyar programes basats en POO. A partir d'aquí, en l'apartat 4 hem vist, a més, sense entrar en gaires detalls, els beneficis que presenta el paradigma de la programació orientada a objectes.

Per acabar, en l'apartat 5, com que sou programadors novells i en aquesta assignatura es planteja un enfocament (és a dir, un paradigma de programació) totalment diferent d'aquell al qual esteu acostumats, hem plantejat una sèrie de qüestions que heu de tenir en compte a l'hora de cursar aquesta assignatura.

Bibliografia

Booch, G.; Maksimchuk, R.; Engle, M.; Conallen, J.; Houston, K. (2007). *Object-Oriented Analysis and Design with Applications*. Pearson Education.

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

Meyer, B. (1999). *Construcción de software orientado a objetos*. Madrid: Prentice Hall.