# Desenvolupament de programari basat en reutilització

PID_00184443

Macario Polo Usaola

Temps mínim de dedicació recomanat: 10 hores

Universitat Oberta de Catalunya

**Macario Polo Usaola**

Llicenciat en Informàtica per la Universitat de Sevilla i doctor per la Universitat de Castella - la Manxa. Professor titular de Llenguatges i sistemes informàtics a la UCLM i tutor del postgrau d'Enginyeria del programari a la UOC. Ha estat investigador principal de diversos projectes de recerca relacionats amb l'automatització de processos de programari, especialment en l'àrea del *testing*, que és actualment la seva línia de recerca principal. Ha publicat diferents articles sobre proves de programari i també diferents obres de narrativa: la seva última novel·la es titula *Si yo soy yo*.

# Índex

# Introducció

Tothom que hagi programat alguna vegada haurà sentit, en algun moment, la necessitat de recuperar alguna funció o algun bloc de codi que tenia escrit d'algun projecte previ per a incorporar-lo en el que l'estigui ocupant en aquest moment. Les possibilitats que ens donen poder copiar i enganxar ens permeten aprofitar d'aquesta manera l'esforç que hi vam dedicar en moments anteriors.

No obstant això, aquesta utilització d'elements predesenvolupats no casa completament amb el concepte *reutilització* en el sentit de l'enginyeria del programari, en què *reutilitzar* té un significat més relacionat amb el següent:

**1)** L'encapsulació de funcionalitats (prèviament desenvolupades i provades) en elements que, posteriorment, podran ser directament integrats en altres sistemes.

**2)** El desenvolupament d'aquestes funcionalitats mitjançant mètodes d'enginyeria de programari.

**3)** L'aprofitament del coneixement i l'experiència produïts durant dècades de pràctica en la construcció de programari.

Aquest mòdul s'estructura en quatre parts: en la primera es presenta una visió general de la reutilització, se'n posen alguns exemples i es fa un breu repàs per la seva història. Les aportacions a la reutilització de l'orientació a objectes, que ha exercit un paper importantíssim en aquest camp, es descriuen i il·lustren en la segona part. Algunes solucions tècniques, moltes d'elles evolucions gairebé naturals de l'orientació a objectes, es presenten en la tercera part. La quarta i última part s'enfoca més a solucions metodològiques per a desenvolupament de programari, que prenen la reutilització com un valor essencial.

# Objectius

En finalitzar aquest curs, l'alumne ha de ser capaç del següent:

**1.** Ser conscient de la conveniència de reutilitzar programari i de desenvolupar programari reutilitzable.

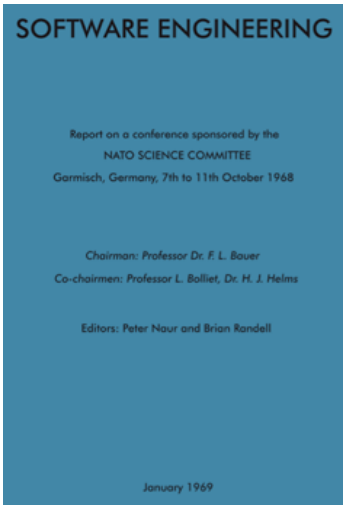
**2.** Conèixer les principals tècniques per al desenvolupament de programari reutilitzable.

**3.** Conèixer tècniques per a integrar programari reutilitzable en els projectes nous.

**4.** Conèixer metodologies que permetin el desenvolupament de projectes que facin un ús intensiu de la reutilització.

# 1. Introducció a la reutilització

La primera vegada que es va introduir de manera pública i rellevant la idea de la reutilització del programari va ser el 1968, en un cèlebre congrés sobre enginyeria del programari que va organitzar el comitè de ciència de l'OTAN. En aquesta trobada, a més d'encunyar-se també el terme *enginyeria del programari*, l'enginyer de Bell Laboratories M. D. McIlroy va afirmar que "La indústria del programari s'assenta sobre una base feble, i un aspecte important d'aquesta debilitat és l'absència d'una subindústria de components". En un context tecnològic molt més primitiu que l'actual, McIlroy assimilava un component a una rutina o conjunt de rutines reutilitzable: explicava que en la seva empresa utilitzaven més d'un centenar de màquines diferents de processament d'informació, però que gairebé totes necessitaven una sèrie de funcionalitats en comú (rutines per a convertir cadenes a nombres, per exemple) que havien d'implementar una vegada i una altra.

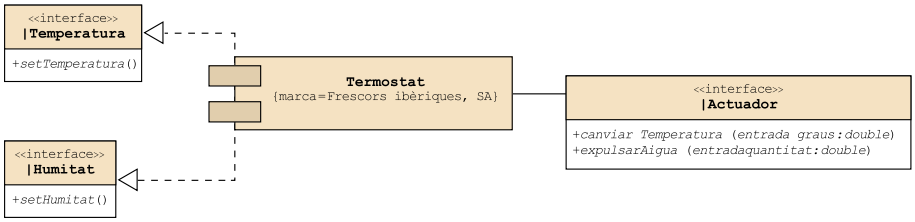



La conferència es va organitzar a la ciutat de Garmish (Alemanya), del 7 a l'11 d'octubre de 1968. Les actes s'han escanejat i processat mitjançant OCR i es troben publicades a Internet: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

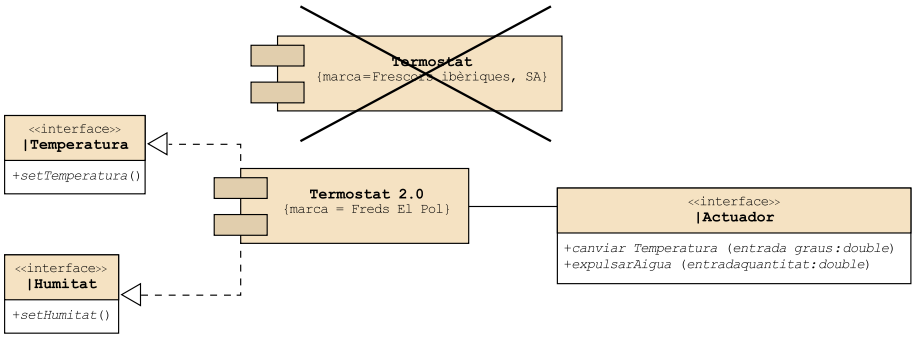
En el context actual de l'enginyeria del programari, tothom entén que un component es correspon amb un fragment reemplaçable d'un sistema, dins del qual es troben implementades un conjunt de funcionalitats. Un component es pot desenvolupat aïllat de la resta del món.

> Un botó d'un editor visual d'aplicacions és un petit component que permet, per exemple, que l'usuari el premi, i que ofereix al desenvolupador, d'una banda, una interfície pública perquè aquest li adapti la mida a la finestra, li canviï el color, el text que mostra, etc.; i, d'una altra, la possibilitat de col·locar diferents instàncies d'aquest mateix botó a la interfície o en altres aplicacions, i evitar la tasca tediosa de programar-ne el codi cada vegada que se'n vulgui fer ús.

Perquè un component sigui, en efecte, reutilitzable, ha d'encapsular la seva funcionalitat, i l'ha d'oferir a l'exterior per mitjà d'una o més interfícies que el component mateix ha d'implementar. Així, quan el sistema en el qual incloem el component vol utilitzar una de les funcionalitats que ofereix aquest, la hi demana per mitjà de la interfície. Però, de la mateixa manera, el component pot també requerir altres interfícies, per exemple, per a comunicar els resultats del seu còmput. En l'exemple de la figura següent es mostra un component *Termòstat* que rep dades de dos sensors (un de temperatura i un altre d'humitat): aquests li comuniquen periòdicament la informació per mitjà de les dues interfícies que els ofereix el component (*ITemperatura* i *IHumitat*). Aquest, en funció dels paràmetres rebuts, ordena per mitjà d'algun element que implementa la interfície *IActuador* que s'apugi o s'abaixi la temperatura (operació *canviarTemperatura*) o que s'expulsi aigua a l'estada per a incrementar la humitat.




Suposem que el component de la figura de dalt no respon amb la precisió requerida davant certs canvis atmosfèrics: per exemple, incrementa o disminueix la temperatura solament quan la diferència amb la mesura anterior és superior a 2 graus. En aquest cas, és possible que trobem un altre component del mateix o d'un altre fabricant perquè puguem construir una versió nova del sistema sense massa esforç: en la il·lustració següent, el vell *Termòstat* és substituït per un *Termòstat* 2.0 adquirit a un proveïdor diferent. Perquè aquesta nova versió del component es pugui integrar perfectament en el nostre sistema, ha d'oferir i requerir exactament les mateixes interfícies.




L'enginyeria del programari basada en components (CBSE, *component-based software engineering*) s'ocupa del desenvolupament de sistemes programari a partir de components reutilitzables, com en l'exemple senzill que acabem de presentar. Però la CBSE, no obstant això, és només una de les línies de treball de la comunitat científica i de la indústria del programari per a afavorir la reutilització i, d'aquesta manera, evitar la reinvenció contínua de la roda o, en el nostre entorn, impedir l'anàlisi, el disseny, la implementació i la prova de la mateixa solució, desenvolupant-la una vegada i una altra en mil i un projectes diferents.

**Vegeu també**

Vegeu l'assignatura *Enginyeria del programaridecomponents i sistemes distribuïts* del grau d'Enginyeria Informàtica.

D'aquesta manera, la reutilització[1] abarateix els costos del desenvolupament: en primer lloc, perquè no es necessita implementar una solució de la qual ja es disposa; en segon lloc, perquè augmenta la productivitat, en poder dedicar els recursos a altres activitats més en línia amb el negoci[2]; en tercer lloc, perquè probablement l'element que reutilitzem ha estat suficientment provat pel seu desenvolupador, i llavors els provadors es podran dedicar a provar altres parts més crítiques del sistema[3], i obtindran llavors uns nivells de qualitat molt més alts que si el sistema es desenvolupa des de zero.

(1)Krueger la defineix com "el procés de crear sistemes programari a partir de programari preexistent, en lloc de crear-los començant de zero".

(2)Imaginem que, per cada aplicació visual que fos necessari construir, s'hagués d'escriure el codi necessari per a implementar un botó corrent.

## 1.1. Una mica d'història

(3)Encara que també hauran de provar la integració de l'element reutilitzat amb la resta del sistema.

La introducció, en els anys vuitanta, del paradigma orientat a objectes, va representar un avenç importantíssim quant a reutilització de programari: en efecte, una classe ben construïda encapsula en si mateixa una estructura i un comportament que es poden aprofitar en altres projectes; a més, un conjunt de classes que col·laboren entre si per a servir algun propòsit es poden agrupar en biblioteques, que després, si volem, s'importen per a utilitzar-les més d'una vegada.

**Consulta recomanada**

**C. W. Krueger** (1992). "Software Reuse". *ACM Computing Surveys* (vol. 24, pàg. 131-183).

En els anys noranta es van desenvolupar biblioteques que permetien la programació dirigida per esdeveniments, biblioteques d'ús compartit i biblioteques d'enllaç dinàmic. Les *Microsoftfoundationclasses* (MFC, classes fonamentals de Microsoft), dels anys noranta, agrupaven gran quantitat de classes d'ús comú en una sèrie de biblioteques d'enllaç dinàmic, que el programador podia utilitzar per a construir tipus molt diversos d'aplicacions.

També en els anys noranta es comencen a desenvolupar els primers components, aptes per a ser integrats en aplicacions sense massa esforç. Algunes companyies construeixen i venen el que es van anomenar *components COTS* (*commercial off-the self*), que no eren sinó components tancats i posats a la venda al costat de l'especificació de les seves interfícies i els seus manuals d'integració i utilització.

**Vegeu també**

Vegeu els components COTS en l'assignatura *Enginyeria de requisits* del grau d'Enginyeria Informàtica.

Es construeixen també biblioteques que encapsulen funcionalitats de comunicacions, que permeten, per exemple, la compartició d'una mateixa instància per part d'altres objectes situats en màquines remotes. En aquest sentit, cada gran companyia de programari va desenvolupar la seva tecnologia de comunicacions pròpia.

Així, per exemple, Sun desenvolupa RMI (*remote method invocation*) i Microsoft fa el mateix amb DCOM (*distributed component object model*), dos sistemes incompatibles però que persegueixen el mateix propòsit. Tots dos models d'objectes distribuïts comparteixen una idea senzilla: l'elaboració d'un protocol per a compartir objectes i transmetre missatges entre objectes remots.

Aquests models no són, al cap i a la fi, res gaire diferent d'un sòcol que envia pel seu canal cadenes de bytes amb la informació codificada d'alguna manera. Lamentablement, els formats d'aquestes cadenes de bytes eren diferents en RMI i en DCOM, per la qual cosa un objecte Java no es podia comunicar (almenys directament) amb un objecte Microsoft.

**Vegeu també**

Vegeu RMI i DCOM en l'assignatura *Enginyeria del programari de components i sistemes distribuïts* del grau d'Enginyeria Informàtica.

Suposem que dues companyies diferents desenvolupen una calculadora remota senzilla començant des de zero. Totes dues calculadores implementen i ofereixen, via accés remot, les operacions aritmètiques bàsiques, com *suma (x : int, i : int) : int*. Cada companyia oferirà accés a la seva calculadora imposant un format diferent: la primera pot dir al client que, quan vulgui invocar una operació, enviï per un sòcol una cadena de bytes amb el format *operacio#parametre1#parametre2*, mentre que la segona potser utilitzarà el símbol @ com a caràcter de separació i, a més, un coixinet al final. Òbviament, una cadena de bytes per a invocar el primer servei no serveix per a cridar el segon, que retornarà un error. Els desenvolupadors d'RMI i DCOM no imposaven, evidentment, un format tan simple, sinó altres de més complexos que, no obstant això, romanen transparents per al

desenvolupador, que els utilitza per mitjà de les biblioteques corresponents, que són les que codifiquen els missatges.

Aquest problema d'incompatibilitats se soluciona parcialment en els primers anys d'aquest segle amb **la introducció dels serveis web**.

Un servei web no és més que una funcionalitat que resideix en un sistema remot i que es fa accessible a altres sistemes per mitjà d'una interfície que, normalment, s'ofereix per mitjà de protocol HTTP.

Les invocacions als serveis oferts per la màquina remota (que actua de servidor) viatgen des del client en un format estandarditzat de pas de missatges anomenat *SOAP* (*simple object access protocol*: protocol simple d'accés a objectes) per a la descripció dels quals, per fortuna, es van posar d'acord Microsoft, IBM i altres grans empreses. SOAP utilitza notació XML i, encara que una invocació a l'operació remota `suma(x : int, i: int) : int` no sigui exactament com es mostra en la figura següent (en la qual se sol·licita la suma dels nombres 5 i 8), la idea no és gaire diferent.

```
<SOAP_Invocation>
   <operation name='suma'>
      <parameter type='int' value='5'/>
      <parameter type='int' value='8'/>
   </operation>
</SOAP_Invocation>
```

Gràcies a aquesta estandardització, podem utilitzar una funcionalitat oferta en una màquina remota sense importar-nos en quin llenguatge ni en quin sistema operatiu s'estigui executant. Des de cert punt de vista, un servei web és un component remot que ens ofereix una interfície d'accés en un format estandarditzat.

**Vegeu també**

Vegeu els serveis web en l'assignatura *Enginyeria del programari de components i sistemes distribuïts* del grau d'Enginyeria Informàtica.

La reutilització que permeten els serveis web és molt evident, ja que s'integren en l'aplicació, sense necessitat de fer cap instal·lació, les funcionalitats que un tercer ha desenvolupat. L'únic que el desenvolupador necessita serà una classe que constituirà el punt d'accés al servidor (un servidor intermediari) que:

**1)** codifiqui adequadament les crides al servei remot[4]

**2)** les hi enviï;

**3)** rebi els resultats (també en format SOAP); i

**4)** els descodifiqui a un format que entengui l'aplicació.

(4)És a dir, que les tradueixi a cadenes de bytes en format SOAP.

(5)És a dir, el desenvolupador que integra un servei web en la seva aplicació.

A més, l'usuari del servei web[5]tampoc no s'ha de preocupar d'implementar el servidor intermediari esmentat, ja que qualsevol entorn de desenvolupament modern el construeix a partir de l'especificació de la interfície del servei, que també s'ofereix en un format estandarditzat anomenat *WSDL* (*web service description language*: llenguatge de descripció de serveis web).

A partir de la utilització més o menys massiva de serveis web es comença a parlar d'arquitectura orientada a serveis (SOA: *service oriented architecture*) que, bàsicament, consisteix en el desenvolupament d'aplicacions utilitzant un nombre important de funcionalitats exposades com a serveis que, no obstant això, no han de ser obligatòriament serveis web.

**Vegeu també**

Vegeu SOA en l'assignatura *Enginyeria* del *programari de components i sistemes distribuïts* del grau d'Enginyeria Informàtica.

D'això s'ha avançat ràpidament a la computació en el núvol (*cloud computing*), mitjançant la qual s'ofereixen multitud de serveis (execució d'aplicacions, serveis d'emmagatzematge, eines col·laboratives, etc.) als usuaris, que els utilitzen de manera totalment independent de la seva ubicació i, en general, sense gaires requisits de maquinari.

**Web recomanat**

A la Wikipedia hi ha una dissertació molt completa sobre la informàtica en el núvol: http://en.wikipedia.org/wiki/Cloud_computing

Una altra línia interessant respecte de la reutilització es troba en **la fabricació industrial del programari**, tant en els anomenats *mercats de masses*[6] com en els mercats de client[7]. En alguns entorns s'ha fet un pas més i s'apliquen tècniques de línies de producte a la fabricació de programari: el desenvolupament de programari mitjançant línies de producte de programari (en endavant, LPP) sorgeix fa una dècada apriximadament (encara que és ara quan està prenent més interès) amb l'objectiu de flexibilitzar i abaratir el desenvolupament de productes de programari que comparteixen un conjunt ampli de característiques (*common features* o característiques comunes).

(6)El mateix producte es pot vendre diverses vegades –el que es diuen economies d'escala–, copiant els prototips mecànicament.

(7)Cada producte és únic i el benefici s'aconsegueix per mitjà de la reutilització sistemàtica.

**Nota**

En anglès, les LPP es coneixen com a *software product lines*, *software product architectures* i *software product families* (SPL, SPA i SPF, respectivament).

Les línies de producte fa anys que s'apliquen en altres entorns industrials, com la fabricació de vehicles en cadenes de muntatge. Les diferents versions d'un mateix model de cotxe difereixen en certes característiques del seu equipament (presència o absència d'aire condicionat, alçavidres elèctrics, etc.), la qual cosa no impedeix que tots comparteixin les mateixes cadenes de muntatge i producció, de manera que imiten així les indústries de productes manufacturats que es van engegar a partir de la Revolució Industrial, al segle XIX, i que Charles Chaplin va parodiar en el film *Temps moderns*.

**Vegeu també**

Veurem les LPP detalladament en l'apartat "Enginyeria del programari per a línies de producte basada en UML 87" d'aquest mòdul.

Més recentment, les línies de producte s'utilitzen també per al desenvolupament i construcció de telèfons mòbils que, a més del conjunt bàsic de funcionalitats, ofereixen diferents mides de pantalla, presència o absència de Bluetooth, diverses resolucions de la càmera, etc. En general, el desenvolupament de productes de programari més o menys similars mitjançant LPP permet aprofitar gairebé al màxim els esforços dedicats a la construcció de les *common*

*features* (característiques comunes) reutilitzant-les i dedicant els recursos a la implementació de les *variable features* (característiques pròpies o variables) de cada producte i al muntatge.

### 1.2. Reutilització de coneixement: patrons de disseny

Quan en enginyeria de programari es parla de *reutilització*, no hem de creure que aquesta consisteix solament a tornar a utilitzar funcionalitats predesenvolupades i empaquetades en classes, components o serveis. També es pot reutilitzar el coneixement i l'experiència d'altres desenvolupadors. Efectivament, hi ha multitud de problemes que es presenten una vegada i una altra quan es desenvolupa un producte programari.

La gestió de la persistència d'objectes, per exemple, es presenta cada vegada que es construeix una aplicació que hagi de manipular informació connectant-se a una base de dades.

Com a enginyers de programari, hem de ser capaços d'identificar aquestes situacions que es presenten freqüentment i que han estat ja, amb tota probabilitat, resoltes amb antelació per altres persones de manera eficient. Aquest tipus de coneixement, en el qual es descriuen problemes que apareixen de manera més o menys freqüent, i que inclouen en la seva descripció una o més maneres adequades de resoldre'l, conforma el que es diu un patró. Un **patró**, llavors, descriu una solució bona a un problema freqüent, com aquest que hem esmentat de la gestió de la persistència.

**Vegeu també**

Vegeu l'assignatura *Anàlisi idisseny amb patrons* del grau d'Enginyeria Informàtica.

Revisant l'exemple del component *Termòstat* que s'ha reemplaçat per un altre, de vegades ocorre que el component nou no es pot integrar directament en el sistema: en la vida real, de vegades cal col·locar una mica d'estopa o algun adhesiu entre dues canonades que s'han de connectar i que no arriben a encaixar perfectament. En el cas de la substitució de components o serveis el problema és molt semblant: disposem d'un component o servei que ens ofereix les funcionalitats que requerim, però les interfícies ofertes i esperades no coincideixen amb el context en el qual el volem integrar. Aquest és un problema recurrent que es pot solucionar aplicant *glue code*[8] potser mitjançant un adaptador, que és la solució que proposa el patró *Wrapper*. El servidor intermediari que s'esmentava en l'epígraf anterior, i que dèiem que s'utilitza per a connectar el sistema a un servei web extern, és també la solució que es proposa en el patró de disseny *Proxy* per a connectar un sistema a un altre sistema remot.

(8)Literalment "codi cola", que correspondria a aquesta estopa o a aquest adhesiu.

Tot aquest coneixement produït durant anys de recerca i desenvolupament en enginyeria del programari es pot reutilitzar i, de fet, es recopila i es fa públic en catàlegs de patrons. El més famós és el de Gamma, Helm, Johnson i Vlissides, però tots els anys tenen lloc, en diversos llocs del món, conferències sobre patrons: la PLoP (Conference on Pattern Languages of Programs) a escala internacional, l'EuroPLoP a Europa o la KoalaPLoP a Oceania. Per publicar un patró en aquestes conferències, l'autor ha de demostrar que el problema és important i que es presenta amb freqüència, i ha de presentar almenys tres casos diferents en els quals s'hagi aplicat la mateixa solució: d'aquesta manera, es demostra que la solució ha estat utilitzada satisfactòriament més d'una vegada, i ofereix a la comunitat la possibilitat de reutilitzar aquest coneixement.

**Consulta recomanada**

La primera edició del llibre *Design patterns* es va publicar el 1994. Ha estat traduït a multitud d'idiomes, entre els quals el castellà: **E.Gamma; R. Helm; R. Johnson; J. Vlissides** (2002). *Patrones de diseño: elementos de software orientado a objetos reutilizables*. Addison Wesley.


### 1.3. Beneficis i costos de la reutilització

Hi ha dos enfocaments ben diferents quant a reutilització: l'enfocament **oportunista** aprofita elements programari construïts en projectes anteriors, però que no van ser desenvolupats especialment per a ser reutilitzats; en **un enfocament** més **planificat** o **proactiu**, els elements es construeixen pensant que seran reutilitzats, la qual cosa pot significar, en el moment del desenvolupament, més inversió de recursos, ja que s'ha de dotar l'element de suficient generalitat.

El nostre enfocament en aquest material se centra en **la reutilització proactiva**.

De manera general, la reutilització permet:

- Disminuir els terminis d'execució de projectes, perquè s'evita construir novament funcions, funcionalitats o components.

- Disminuir els costos de manteniment, cosa que es discuteix més extensament a continuació.

- Augmentar la fiabilitat, sempre que s'utilitzin elements reutilitzables procedents de subministradors segurs.

- Augmentar l'eficiència, sobretot si es reutilitza programari desenvolupat sota l'enfocament planificat o proactiu, perquè els desenvolupadors implementaran versions optimitzades d'algorismes i d'estructures de dades. En un desenvolupament nou, en el qual la reutilització sorgirà més endavant potser per un enfocament oportunista, la pressió dels terminis de lliurament impedeix, en molts casos, millorar o optimitzar els algorismes.

- Augmentar la consistència dels dissenys i millorar l'arquitectura del sistema, especialment quan s'utilitzen *frameworks* o biblioteques ben estructu-

rades que, en certa manera, imposen a l'enginyer de programari un bon estil de construcció del sistema.

- Invertir, perquè la despesa ocasionada a dotar un element de possibilitats de reutilització es veurà recompensada amb estalvis importants en el futur.

### 1.3.1. L'impacte positiu de la reutilització en el procés de manteniment

Un treball de Kung i Hsu de 1998 assimila la taxa d'arribades de peticions de manteniment al model d'ecologia de poblacions de May. En un ecosistema tancat en el qual hi ha només predadors i preses, l'augment de predadors implica una disminució del nombre de preses, la qual cosa implica que, en haver-hi menys aliment, disminueixi el nombre de predadors i augmenti novament el de preses. Amb el pas del temps, aquests alts i baixos van tendint envers un punt d'equilibri.

**Lectura addicional**

**H.-J. Kung; C. Hsu** (1998). "Software Maintenance Life Cycle Model". *International Conference on Software Maintenance* (pàg. 113-121). IEEE Computer Society.

**Model de Kung i Hsu**

En el model de Kung i Hsu, els errors al programari són preses, que són "devorades" per les accions de manteniment correctiu, que són els predadors. La introducció d'un sistema implica la introducció massiva d'errors, que es consumeixen en l'etapa de creixement, en ser corregits pels informes dels usuaris. En la maduresa amb prou feines queden preses de la primera generació; no obstant això, les peticions de perfectiu (addició de noves funcionalitats) introdueixen nous errors (noves preses, per tant), que són novament consumides per les correccions que fan els programadors.

**Lectura addicional**

**R. M. May** (1974). "Biological Populations with Nonoverlapping Generations: Stable Points, Stable Cycles, and Chaos". *Science* (vol. 186, núm. 4164, pàg. 645-647).

El procés de manteniment és un dels processos principals del cicle de vida del programari. Comença una vegada que el sistema ha estat instal·lat i posat en explotació. En la seva caracterització, Kung i Hsu identifiquen quatre etapes en la vida d'un sistema:

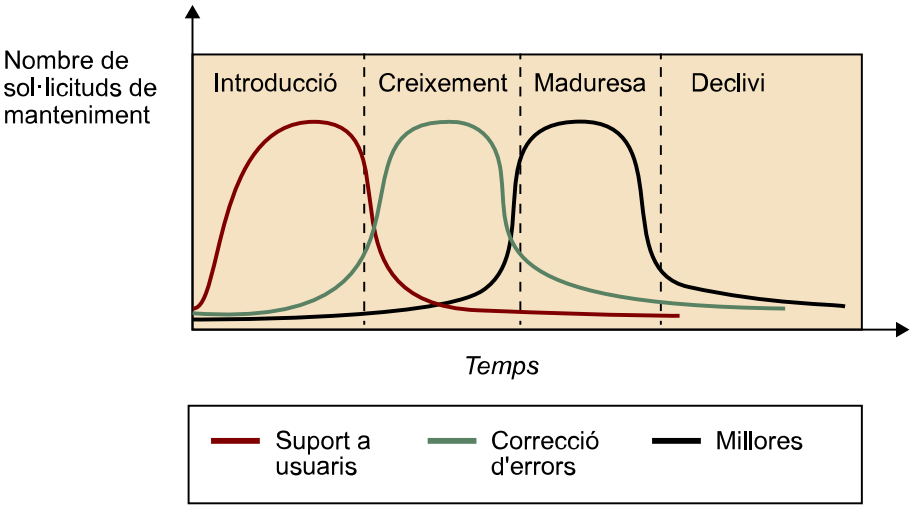
- **Introducció**, que té una durada relativament breu. Durant aquest període, els usuaris aprenen a manejar el sistema i envien a l'equip de manteniment peticions de suport tècnic, a fi d'aprendre a utilitzar-lo per complet.

- **Creixement**. En aquesta etapa disminueix el nombre de peticions de suport perquè els usuaris ja coneixen bé el sistema i no necessiten ajuda tècnica. No obstant això, els usuaris comencen a trobar errors en el sistema i envien peticions de manteniment correctiu.

- **Maduresa**. Les correccions que s'han anat fent durant l'etapa de creixement han disminuït el nombre de peticions de manteniment correctiu i el sistema, ara, té molt pocs errors. Els usuaris, no obstant això, que ja coneixen el sistema, demanen noves funcionalitats que es tradueixen a peticions de manteniment perfectiu.

- **Declivi**. Amb el pas del temps, nous sistemes operatius i nous entorns afavoreixen la substitució del sistema per un de nou, o la migració envers

plataformes més modernes, la qual cosa pot donar lloc o bé a la retirada del sistema o bé a la migració, que implicarà la presentació de peticions de manteniment adaptatiu.




En servir una petició de manteniment, l'enginyer de programari, en primer lloc, ha de situar l'àrea del codi que ha de ser modificada, entendre la lògica de l'aplicació i les seves connexions amb sistemes externs (tasques que són, en general, les més costoses), implementar el canvi i fer les proves que siguin necessàries, la qual cosa inclourà proves de regressió per a comprovar que la modificació no ha introduït errors que abans no existien.

La reutilització de programari implica una disminució del manteniment correctiu, ja que la probabilitat de trobar errors en una funció utilitzada en diversos projectes anteriors és molt petita. També disminueix els costos de manteniment perfectiu (addició de funcionalitats), perquè resulta més senzill entendre un sistema en el qual molts dels seus elements són "caixes negres" que ofereixen correctament una funcionalitat determinada i que, per tant, no és necessari examinar.

### 1.3.2. Els costos de reutilitzar

La reutilització de programari té, sempre, uns costos associats. Meyer indica que, almenys, aquests costos són els següents:

**Consulta recomanada**

**B. Meyer** (1997). *Object-oriented software construction* (2a. ed.). Prentice Hall Professional Technical Reference.

- **Cost d'aprenentatge** (almenys la primera vegada).

- **Cost d'integració en el sistema**. Per exemple, en el cas de la reutilització de components, moltes vegades cal escriure adaptadors per a connectar adequadament el component en el sistema que s'està desenvolupant.

**Vegeu també**

Vegeu la reutilització de components en l'apartat "Components".

Òbviament, l'enfocament proactiu en el desenvolupament de programari perquè aquest sigui reutilitzable té també uns costos associats. Una visió a curt termini desaconsellaria dedicar esforços especials a la construcció de compo-

nents acoblats o d'elements de programari genèrics (parlarem de tot això més endavant). Griss (1993) aconsella el desenvolupament de models de càlcul del ROI (*return of investment*, tornada de la inversió) per a convèncer la direcció de la conveniència de tractar la reutilització com un actiu important.

**Lectura addicional**

**M. L. Griss** (1993). "Software reuse: from library to factory". *IBM Software Journal* (vol. 32, núm. 4, pàg. 548-566).

# 2. Reutilització en disseny de programari orientat a objectes

Bertrand Meyer defineix la construcció de programari orientat a objectes com la

> "Construcció de sistemes programari en forma de col·leccions estructurades, possiblement, d'implementacions de tipus abstractes de dades."

Els tipus abstractes de dades (TAD) descriuen matemàticament el conjunt d'operacions que actuen sobre un tipus d'element determinat. Cada operació es descriu algebraicament en termes del següent:

**1)** les transformacions que fa sobre les ocurrències del tipus de dada;
**2)** els axiomes de l'operació; i
**3)** les precondicions necessàries perquè l'operació es pugui executar.

Com que són abstractes i segueixen el principi d'abstracció, els TAD exhibeixen una vista d'alt nivell de les possibilitats de manipulació d'informació que permet el tipus de dada, i mostren a l'exterior la part que interessa i amaguen els detalls que no interessa mostrar. Així, es fa també gala de l'encapsulació i de l'ocultament de la informació.

**Vegeu també**

Repasseu els TAD i l'orientació a objectes en les assignatures *Disseny d'estructures de dades* i *Disseny i programació orientada a objectes* del grau d'Enginyeria Informàtica.

Com en altres àrees del desenvolupament de programari, els TAD descriuen el que succeeix o ha de succeir, però no com succeeix.

Ocorre el mateix, per exemple, en l'anàlisi funcional: l'enginyer de programari identifica els requisits del sistema i els pot representar en un diagrama de casos d'ús. En aquest cas, s'indiquen les funcionalitats que ha de servir el sistema, però sense entrar en els detalls de com les ha de servir.

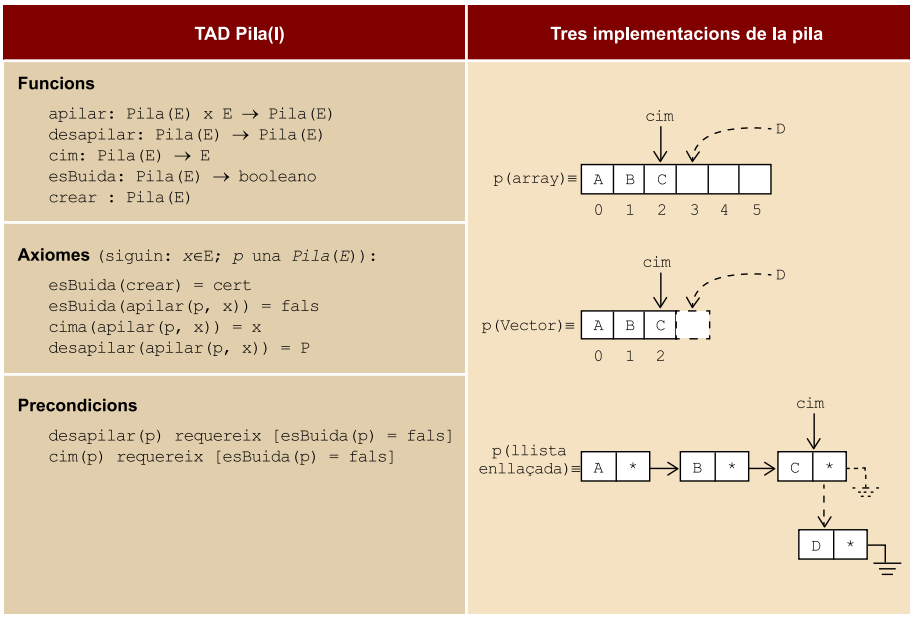
Al costat d'altres principis fonamentals del desenvolupament de programari (com el manteniment de l'alta cohesió i del baix acoblament), l'abstracció, l'encapsulació i l'ocultament de la informació són tres eines essencials del disseny de programari en general, no solament de l'orientat a objectes. Però l'orientació a objectes aporta també les possibilitats que ofereixen l'herència i el polimorfisme. En aquest apartat es presenten les principals característiques que ofereix l'orientació a objectes per permetre i fomentar la reutilització del programari.

## 2.1. Introducció

En la majoria dels llibres d'introducció a les estructures de dades, s'utilitzen estructures d'emmagatzematge d'informació (llistes, piles, cues, arbres, grafs) com a exemples habituals.

> Una pila, per exemple, es correspon amb una estructura de dades en la qual els elements es van col·locant l'un damunt d'un altre, i de la qual podem treure elements en ordre invers al qual han estat col·locats: d'aquesta manera, l'estructura de dades *pila* abstreu el comportament d'una pila real, com la pila que es crea quan es renten els plats, i després quan es van prenent en ordre invers per a parar la taula: se segueix una política LIFO (*last-in, first-out*: últim a entrar, primer a sortir) per a gestionar els elements.

Quan es programa el codi de la pila en l'ordinador, es pot representar l'estructura de dades mitjançant una matriu estàtica, un vector dinàmic, una llista enllaçada, etc.; no obstant això, als dissenyadors i als usuaris del TAD *Pila* els interessa tan sols la descripció del comportament, sense que interessi oferir els detalls de la implementació. D'aquesta manera, el TAD *Pila* és una abstracció de l'estructura real *Pila* que, a més, encapsula els seus detalls d'estructura i funcionament.




En l'especificació del TAD *Pila* que es dóna en la figura anterior, la pila treballa amb elements genèrics de tipus `E` que, en la pràctica, poden ser nombres enters o reals, lletres, cadenes de caràcters, o estructures més complexes, com persones, comptes corrents, abstraccions de l'objecte real plat o, fins i tot, altres piles. Totes les piles (emmagatzemin el tipus d'element que emmagatzemin) es comporten exactament igual: si es col·loca un element al cim, aquest mateix element serà el primer a sortir, independentment que apilem nombres o lletres.

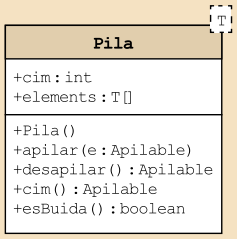
El comportament que descrivim en el TAD *Pila* és llavors un comportament genèric: el llenguatge `C++` és un dels pocs que permeten la implementació de TAD genèrics, com la *Pila* que es mostra en la figura següent, en la qual es

defineix, mitjançant una plantilla (*template*) de C++, una pila que actua sobre un tipus de dada genèrica `T`. Si, en el codi, instanciem una pila `int` i una altra del tipus *Persona*, el compilador genera una versió en codi objecte de la pila d'enters i una altra de la pila de persones.

**Pila genèrica en C++**

```
template <class T>
class Pila {
public:
  Pila() { cima = -1; }
  void apilar(T e)
    { elements [++cim] = e; }
  T desapilar()
    { return elements [cim--]; }
      T cim()
            { return elements [cim];}
     int esBuida() { return cim==-1; }
private:
  int cim;
  T elements[100];
};
```

**Pila genèrica en UML**




Altres llenguatges de programació orientats a objectes no permeten la definició de plantilles, encara que sí deixen la descripció de piles de qualsevol tipus d'element mitjançant l'herència. Aquest és el cas de C# i de les primeres versions de Java, amb el tipus `Object` que representa qualsevol tipus d'objecte. En aquests dos llenguatges, tots els objectes són també instàncies del tipus genèric `Object`, per la qual cosa es pot implementar una pila d'objectes en la qual serà possible emmagatzemar qualsevol cosa, com cadenes, enters o reals, però també *Termòstats* o *IActuadors* com els que mostràvem en l'apartat anterior.

**Nota**

A partir de la versió 5 de Java es poden utilitzar els genèrics, que permeten crear plantilles similars a C++:

```
public class Pila <T>
```

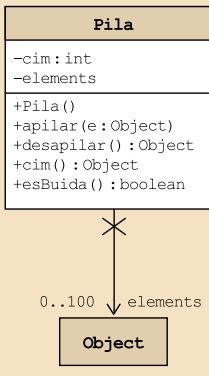
**Vegeu també**

Veurem la programació genèrica en l'apartat "Fàbriques de programari".

**Pila d'objectes en Java**

```
public class Pila {
     private int cim;
     private Object[] elements;

     public Pila() {
          this.cim=-1;
          this.elements=new Object[100];
     }

     public void apilar(Object e) {
          elements[++cim] = e;
     }

     public Object desapilar() {
          return elements[cim--];
     }

     public Object cim() {
          return elements [cim];
     }

     public boolean esBuida() {
          return this.cim==-1;
     }
}
```

**Pila d'objectes en UML**

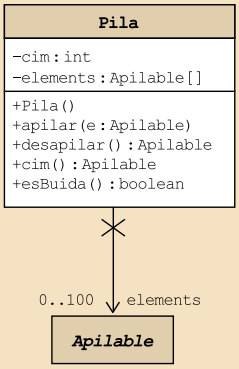



De la mateixa manera, fent ús de l'herència es pot definir un tipus de dada *Apilable* que representi els objectes que, en el domini del problema que s'estigui resolent, es puguin col·locar en la *Pila*:

**Pila d'objectes apilables implementada en Java**

```
public class Pila {
     private int cim;
     private Apilable[] elements;

     public Pila() {
          this.cim=-1;
          this.elements=new Apilable[100];
     }

     public void apilar(Apilable e) {
          elements[++cim] = e;
     }

     public Apilable desapilar() {
          return elements[cim--];
     }

     public Apilable cim() {
          return elements [cim];
     }

     public boolean esBuida() {
          return this.cim==-1;
     }
}
```

**Pila d'objectes apilables en UML**



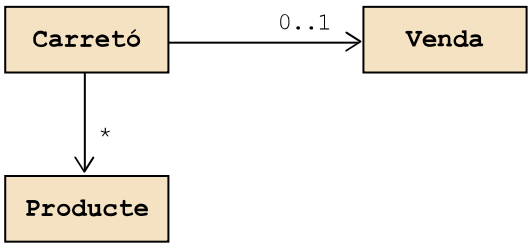

## 2.2. Abstracció, encapsulació i ocultament

En el nostre context, l'abstracció és el mecanisme que utilitza l'enginyer de programari per a representar conceptes del món real utilitzant alguna notació que, en algun moment i d'alguna manera (tal vegada mitjançant diversos passos intermedis), pugui ser processada i traduïda a algun format que entengui un ordinador.

Com s'ha suggerit en paràgrafs anteriors, l'orientació a objectes sorgeix com una evolució dels tipus abstractes de dades: d'acord amb Jacobson (un dels pares de l'UML) i col·laboradors, tant un TAD com una classe són abstraccions definides en termes del que són capaces de fer, no de com ho fan: són, llavors, generalitzacions d'una cosa específica, en les quals l'encapsulació té un paper fonamental. En el disseny d'una tenda virtual, per exemple, s'utilitzen conceptes (abstraccions) com *Carretó*, *Producte* o *Venda*, que es poden representar utilitzant una notació abstracta (com UML), incloent-hi les relacions entre aquestes abstraccions.

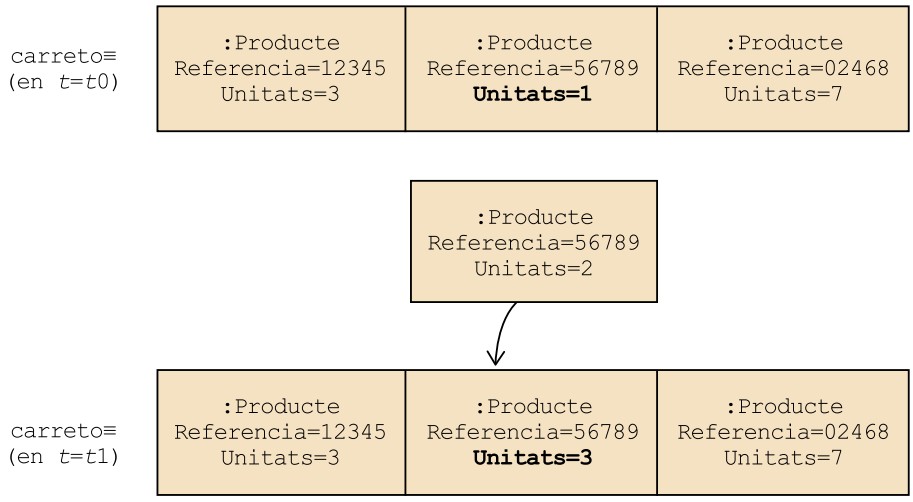
**Consulta recomanada**

**I. Jacobson; M. Christerson; P. Jonsson; G. Övergaard** (1992). *Object-Oriented Software Engineering. A use case approach*. Addison Wesley.



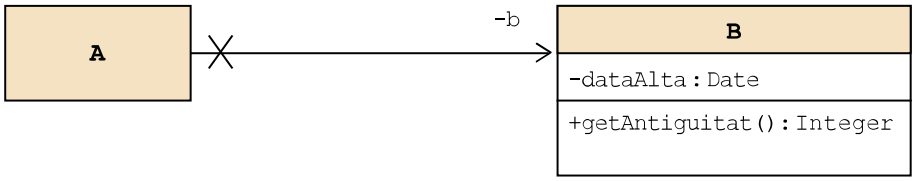

A l'hora d'implementar el *Carretó*, el *Producte* i la *Venda*, el programador haurà de considerar, per exemple, si la col·lecció de productes continguts en el carretó la representarà amb un vector, amb una taula de dispersió (*hash*) o amb una estructura de dades d'un altre tipus. Independentment de l'estructura de dades seleccionada, el programador ha d'aconseguir, per exemple, que si

s'afegeixen dues unitats d'un producte que ja estava contingut en el carretó (el producte amb referència 56789, per exemple), tan sols s'incrementi el nombre d'unitats. La manera amb la que el programador aconsegueixi aquest comportament no forma part del principi d'abstracció (perquè és un detall massa concret sobre el funcionament dels carretons en la botiga virtual). Per contra, aquest comportament que el programador ha implementat (i que, des del punt de vista de l'abstracció, no ens interessa) sí que es correspon amb el principi d'encapsulació: diem que no sabem com funciona l'operació, però sí que sabem que funciona.




En orientació a objectes, una classe és una plantilla d'estructura i comportament que s'utilitza per a crear objectes. La part d'estructura s'utilitza per a representar l'estat dels objectes d'aquesta classe (és a dir, de les seves instàncies), mentre que la de comportament descriu la manera amb la que es pot alterar o consultar l'estat d'aquests objectes, o sol·licitar-los que executin algun servei sobre altres. Des del punt de vista pràctic, l'estructura es compon d'un conjunt de camps o atributs, mentre que el comportament es descriu mitjançant una sèrie d'operacions o mètodes. En general, l'estructura d'una classe (és a dir, el seu conjunt de camps) roman oculta a la resta d'objectes, de manera que només l'objecte mateix és capaç de conèixer directament el seu estat: cada objecte mostra a l'exterior només allò que l'objecte mateix està interessat a mostrar.

En la figura següent, cada objecte de classe A coneix una instància B; les instàncies de B posseeixen un estat que està determinat per un camp privat (és a dir, ocult a l'exterior) anomenat *dataAlta*; no obstant això, B exhibeix a A una porció del seu estat (l'antiguitat en anys), que fa accessible mitjançant l'operació pública *getAntiguitat()*, i que probablement es calcularà en funció de la data del sistema i de la data d'alta.




De la mateixa manera, quan un objecte A vol fer una cosa sobre un altre objecte B, A només podrà actuar utilitzant els elements de B que B mateix li permeti: és a dir, B ofereix a A un conjunt de serveis (operacions o mètodes públics) que són un subconjunt de les operacions incloses i implementades en B. B, per tant, és l'objecte que té realment la responsabilitat de controlar els seus canvis d'estat: A pot intentar desapilar un element d'una pila buida, però la instància mateixa de *Pila* ha de ser responsable de dir, d'alguna manera (potser mitjançant el llançament d'alguna excepció), que no s'hi pot desapilar perquè no té elements, de manera que el seu estat roman inalterable.

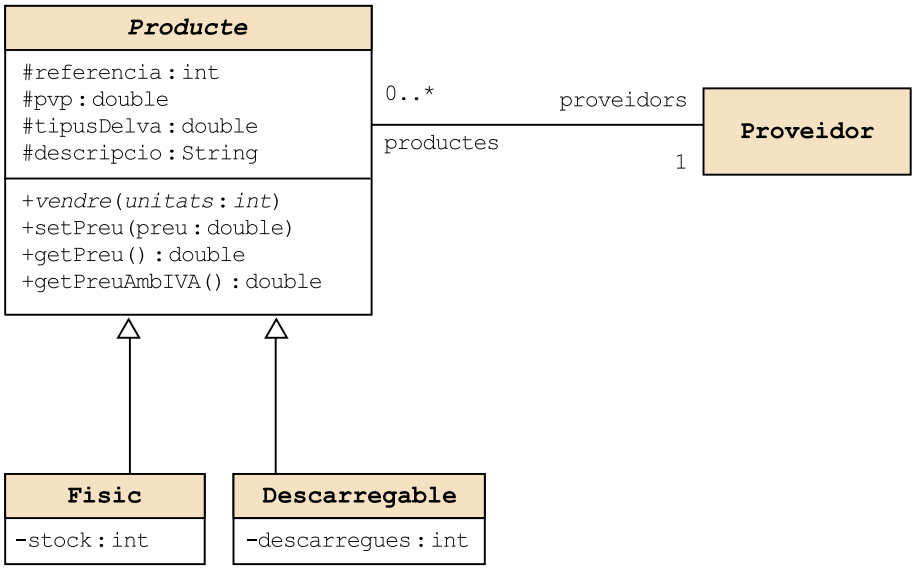
## 2.3. Herència i polimorfisme

Suposem que la botiga virtual ofereix dos tipus de productes: **productes convencionals**, que requereixen un magatzem i dels quals hi ha un nombre determinat d'unitats en estoc, i **productes electrònics**, que es venen per descàrregues i que no tenen necessitat d'emmagatzematge físic. És molt probable que tots dos tipus de productes comparteixin un conjunt ampli de característiques[9] i part del comportament[10]. El producte físic, no obstant això, aporta a la seva estructura (és a dir, al seu conjunt de camps) el nombre d'unitats que es tenen en el magatzem, mentre que el producte descarregable aporta, per exemple, el nombre de descàrregues que ha tingut.

[9] Per exemple: referència, descripció, proveïdor, preu i tipus d'IVA.

[10] Tots dos es donen d'alta i de baixa, tots dos es venen, se'n pot modificar el preu, etc.

Així doncs, la botiga manipularà *Productes*, que podran ser *Fisics* o *Descarregables*, ja que tots dos tenen en comú part de l'estructura i del comportament, és possible definir, en un disseny orientat a objectes, una superclasse *Producte* en la qual situarem tots aquells elements comuns a les dues subclasses que, en aquest cas, seran *Fisic* i *Descarregable*.




En la figura anterior es representa l'estructura de la classe *Producte*, en la qual s'han situat tots els elements comuns als dos tipus de productes que ven la botiga. El fet que aparegui en cursiva el nom de la superclasse i en rodona els de les subclasses denota que *Producte* és una classe abstracta, mentre que *Fisic* i *Descarregable* són classes concretes: és a dir, a la botiga hi ha *Productes*, però han de ser o bé *Físics*, o bé *Descarregables*, i no hi pot haver *Productes* com a tals.

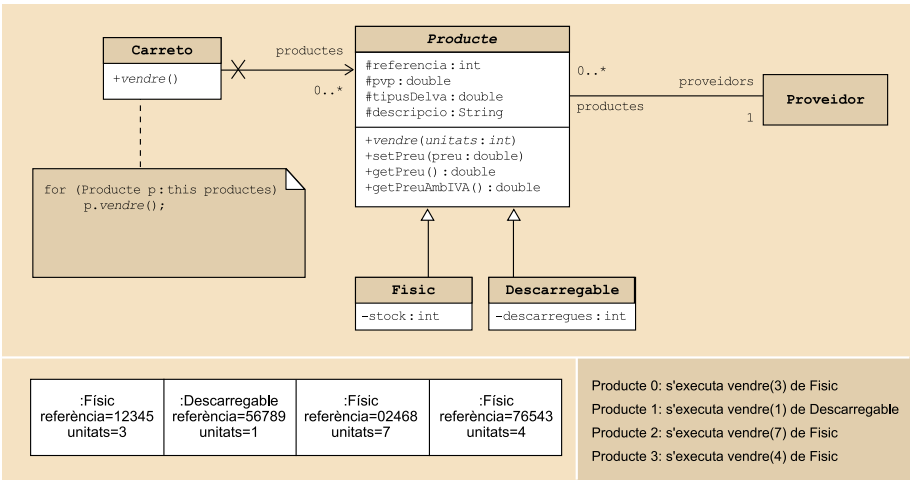
La figura il·lustra la capacitat de reutilització que ofereix l'herència: la definició genèrica (d'estructura i de comportament) que es dóna en la superclasse es reutilitza completament en les subclasses que, en aquest exemple, aporten una mica d'estructura a l'heretat.

Noteu, d'altra banda, que l'operació *Vendre()* de *Producte* apareix (igual que el nom de la classe contenidora) també en cursiva, la qual cosa denota que es tracta d'una operació abstracta: com que és diferent vendre un producte físic d'un producte descarregable (en el primer es disminueix l'estoc; en el segon s'incrementen les descàrregues), cal donar una implementació diferent en les subclasses a l'operació abstracta que s'està heretant.

*Producte*, a més, té una sèrie d'operacions concretes (*setPreu()*,*getPreu()* i *getPreuAmbIVA()*), que es comporten exactament igual amb tots els productes, siguin físics o descarregables: en aquest cas, l'herència ens permet reutilitzar ja no solament part de l'estructura, sinó també part del comportament, per mitjà de les operacions heretades.

A més, la inclusió de l'operació abstracta *vendre()* en *Producte* ens permet manejar, per exemple, col·leccions de productes genèrics[11] i executar-hi l'operació *Vendre()*. En temps d'execució, i en funció del tipus concret de cada *Producte* contingut en la col·lecció, es decideix quina de les dues versions de l'operació *Vendre()* s'ha d'executar: si la instància és del subtipus *Fisic*, l'operació *Vendre()* serà de *Fisic*; si és del subtipus *Descarregable*, *Vendre()* serà de *Descarregable*.

(11)En temps d'execució, estaran òbviament instanciats a productes concrets dels subtipus *Fisic* i *Descarregable*, ja que no hi pot haver instàncies reals de la classe abstracta *Producte*.




El carretó conté una llista de productes "genèrics". En temps d'execució, cada producte està instanciat a un dels dos subtipus concrets. En executar *vendre()* en el carretó, es crida l'operació abstracta *vendre(unitats:int)* de cada producte. S'executa una versió de *vendre()* o l'altra en funció del tipus concret de cada producte.

L'operació *vendre(unitats:int)* és una operació polimòrfica, ja que "té moltes formes"[12]. Com s'il·lustra en el cas de la figura anterior, el polimorfisme permet fer referència a diferents comportaments de diferents objectes, però que es troben designats pel mateix missatge (*vendre*, en aquest cas).

[12]En aquest exemple realment només dues, però hi podria haver més versions diferents de l'operació si hi hagués més subtipus de *Producte*.

**Polimorfisme**

Quan una instància envia un estímul a una altra instància, però sense que aquella estigui segura de saber a quina classe pertany la instància receptora, es diu que tenim polimorfisme.

**Consulta recomanada**

**I. Jacobson; M. Christerson; P. Jonsson; G. Övergaard** (1992). *Object-Oriented Software Engineering. A use case approach*. Addison Wesley.

# 3. Reutilització a petita escala: solucions tècniques de reutilització

## 3.1. Introducció

Com s'ha comentat en les primeres línies del primer apartat, la necessitat de produir programari reutilitzable es va posar de manifest en el mateix congrés en el qual es va emprar per primera vegada, fa més de quaranta anys, el terme *enginyeria del programari*. Aquesta coincidència permet prendre consciència de la importància que, des del naixement mateix d'aquesta disciplina, té la reutilització de programari i els esforços duts a terme per a poder establir mètodes que en permetin la producció industrial.

Escriure una vegada i una altra les mateixes rutines o funcions és una pràctica que s'hauria d'evitar costi el que costi: la recerca i el desenvolupament en enginyeria del programari ha portat a la implementació de diferents tipus de solucions per a afavorir la reutilització. Després de presentar un fragment molt il·lustratiu de la reutilització de programari de fa només 20 anys, aquest capítol revisa les solucions de reutilització més properes a la implementació de codi: biblioteques, classes, patrons arquitectònics, patrons de disseny, components, *frameworks* i serveis.

## 3.2. Una història il·lustrativa

dBase va ser un sistema gestor de bases de dades, molt popular a la fi dels anys vuitanta i començaments dels noranta, i que va ser desenvolupat i comercialitzat per l'empresa Ashton-Tate. A més de permetre la creació de taules i índexs, dBase incorporava un llenguatge de programació amb el qual es podien implementar de manera relativament senzilla aplicacions per a gestionar les nostres pròpies bases de dades, la qual cosa incloïa la possibilitat de crear pantalles (en aquell temps, basades en text i no en gràfics) amb menús, formularis de manteniment de registres, obtenció de llistes, impressió, etc.


dBase III Plus era un dels llibres sobre biblioteques i rutines per a la reutilització de codi més venut en aquells anys. Se'n pot trobar més informació a la Wikipedia: http://es.wikipedia.org/wiki/DBase.

Un dels problemes del dBase és que els seus programes eren interpretats i no compilats, per la qual cosa els desenvolupadors havien de lliurar als seus clients el codi font dels programes que escrivien, amb el perjudici consegüent quant a la conservació dels seus drets d'autor i propietat intel·lectual. Coneixedors d'aquest desavantatge, l'empresa Nantucket Corporation (posteriorment adquirida per Computer Associates) va crear el 1985 el sistema Clipper que era, al començament, un compilador del llenguatge de dBase que generava executables en format binari, la qual cosa impedia l'accés al codi font dels programes.

Gràcies als seus fabricants, el Clipper va evolucionar de manera paral·lela al dBase i va incorporar, amb el temps, noves funcions i millores respecte de les ofertes pel sistema interpretat d'Ashton-Tate. D'aquesta manera, es va arribar al fet que qualsevol programa escrit en dBase podia ser compilat pel Clipper; a més, els programes que estaven escrits directament per a Clipper permetien la inclusió en el seu codi de crides a funcions addicionals que, gràcies al fet que el Clipper estava implementat en assemblador i en C, podien residir en fitxers separats i que no eren, llavors, executables des de l'entorn de dBase. El Clipper traduïa els programes dBase a fitxers objecte amb extensió *.obj*.


EL LIBRO DEL
Clipper SUMMER'87
Esta guía trata de ser una útil herramienta de trabajo sobre la mesa del programador de CLIPPER. Se basa en la versión Summer '87, aunque recoje también datos de versiones anteriores. Comprende lo siguiente:
• Datos técnicos.
• Proceso de instalación.
• Uso del compilador CLIPPER y del PLINK 86.
• Códigos devueltos por INKEY.
• Guía de mandatos y funciones con múltiples ejemplos.
• Palabras reservadas.
• Errores más frecuentes.
EIDOS es un grupo formado por profesionales de distintas áreas de la disciplina informática que confluyen en un trabajo común: la investigación y la docencia en el mundo de las bases de datos relacionales. Expertos en el entorno dBASE y sus diferentes dialectos y herramientas complementarias, los miembros de EIDOS han publicado libros sobre dBASE, CLIPPER, SQL, además de artículos sobre entornos verticales de aplicación de los SGBDR. Asimismo, han diseñado planes de formación e impartido cursos en centros públicos y privados acerca de esta materia.
ISBN 84-86381-84-3
ra-ma

Contraportada d'*Ellibro del Clipper Summer'87*, publicat el 1989 per l'editorial Ra-Ma. Explicava, per exemple, els mètodes de compilació i enllaçament

Per a construir el fitxer que, finalment, pogués ser executat pel sistema operatiu de què es tractés, calia enllaçar el programa objecte amb els fitxers externs en els quals hi havia les funcions auxiliars utilitzades pel programador. Aquests fitxers externs (que s'emmagatzemaven en disc amb extensió *.lib*) s'anomenaven *biblioteques* (*libraries*) i oferien al programador funcions d'ús comú. Les biblioteques estàndard distribuïdes amb el Clipper eren *CLIPPER.LIB*, *EXTEND.LIB* i *OVERLAY.LIB*, i oferien funcions com *abs*[13], *aCopy*[14] o *Overlay*[15].

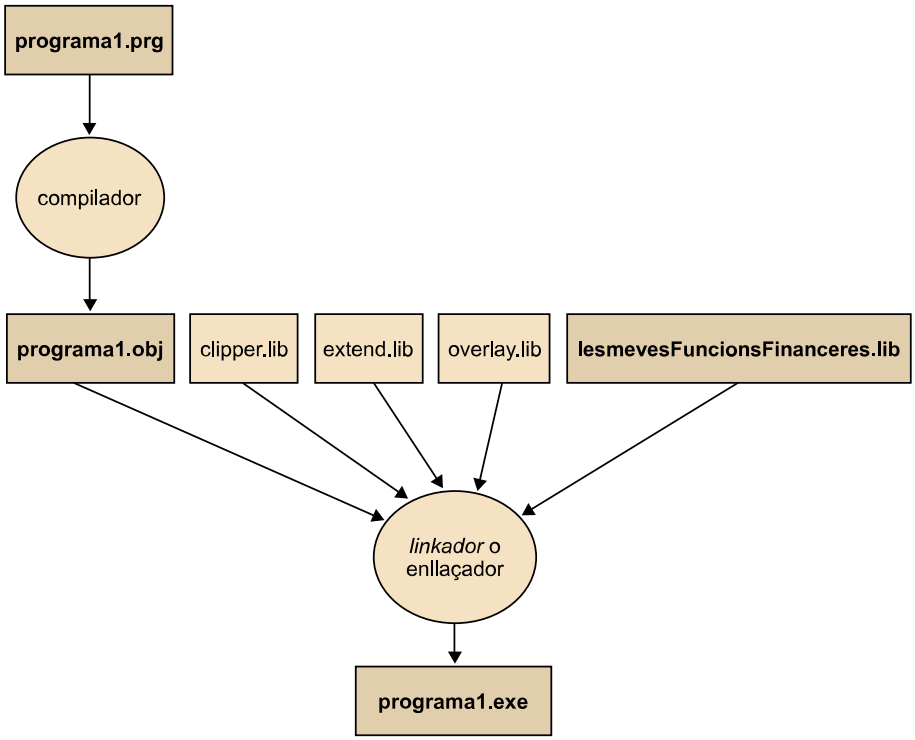
No obstant això, qualsevol programador de C podia implementar funcions addicionals en altres biblioteques que, amb un enllaçament previ, podien ser utilitzades en qualsevol programa[16]. Una d'aquestes biblioteques no estàndard que va gaudir de cert èxit era *FiveWin*, del fabricant espanyol FiveSoftware, i que es va utilitzar per a migrar les primitives aplicacions Clipper (desenvolupades per a terminals basats en text) envers el llavors emergent sistema de Microsoft Windows, que ja oferia als usuaris un entorn gràfic de finestres.

(13)Per a calcular el valor absolut, que era a *CLIPPER.LIB*.

(14)Utilitzada per a copiar matrius, i que es trobava a *EXTEND.LIB*.

(15)Permetia "pontejar" el sistema operatiu per gestionar més eficaçment la memòria de l'ordinador, llavors molt limitada, i que es trobava a *OVERLAY.LIB*.

(16)En la figura següent es representa, amb traç més gruixut, una biblioteca no estàndard que inclou funcions financeres que s'utilitzen en el *programa1.prg*.




L'enllaçament estàtic produïa programes executables de gran mida, la qual cosa podia implicar problemes de gestió de memòria amb els sistemes operatius més comuns en aquells anys, com MS-DOS.

### 3.3. Biblioteques

El concepte de *biblioteques* s'ajusta bastant bé a la idea que, respecte d'aquestes, s'ha presentat en l'epígraf anterior dedicat al dBase i el Clipper.

En efecte, una biblioteca és un conjunt reutilitzable de funcions que s'agrupen normalment en un sol fitxer.

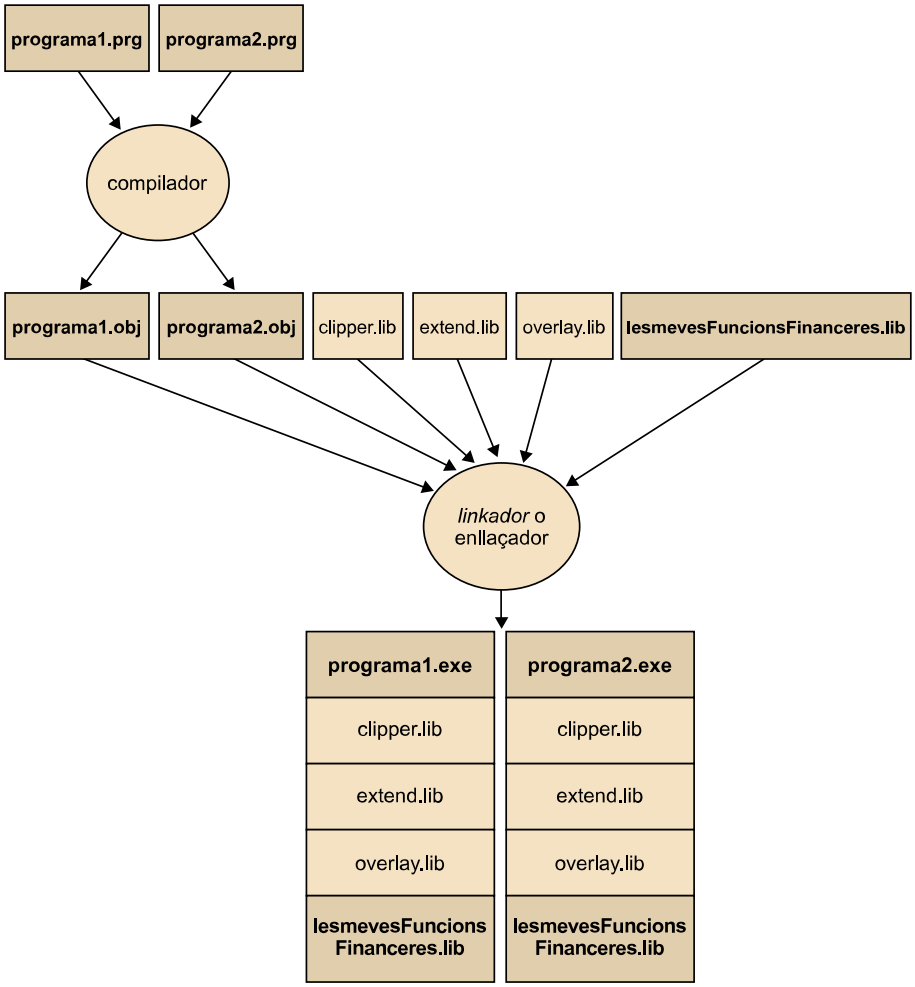
Si el format de la biblioteca i l'entorn operatiu ho permeten, els desenvolupadors poden incloure en els seus programes crides a les funcions ofertes en la biblioteca.

En general, cada biblioteca agrupa un conjunt cohesionat de funcions, com per exemple: biblioteques de funcions matemàtiques i trigonomètriques, biblioteques per a la creació de gràfics, biblioteques per a funcions de cerca en textos mitjançant expressions regulars, etc.

Hi ha dos enfocaments importants a l'hora de fer ús de les funcions contingudes en una biblioteca, i que estan motivats pel fet que les biblioteques es troben en fitxers externs al sistema que s'està desenvolupant: l'enllaç **estàtic** i l'enllaç **dinàmic**.

### 3.3.1. Biblioteques d'enllaç estàtic

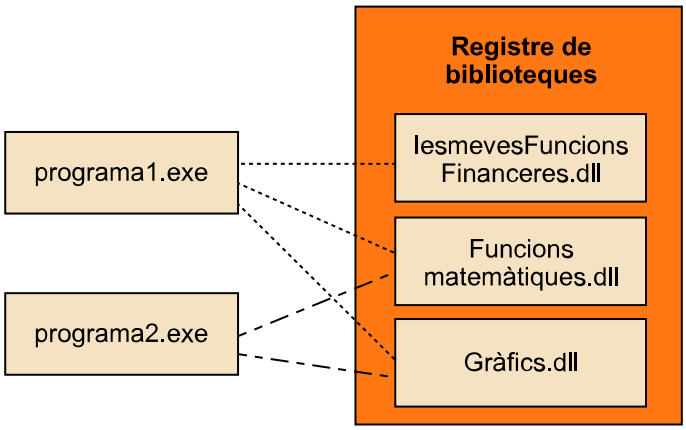
En els primitius sistemes de biblioteques, com els que s'han il·lustrat amb l'exemple del Clipper, tots els programes que es desenvolupaven i que fessin ús, per exemple, de la funció *abs*, havien de ser enllaçats (per a generar l'executable) al costat de la biblioteca *clipper.lib*. Aquest fet causava desavantatges importants quant a l'ús del disc i de la memòria, ja que tot el codi objecte de la biblioteca era combinat, durant el procés d'enllaçat, amb el codi objecte del programa, la qual cosa produïa programes de mida molt gran: si els programes *programa1* i *programa2* usaven la funció *abs*, els dos executables corresponents incloïen la biblioteca completa.



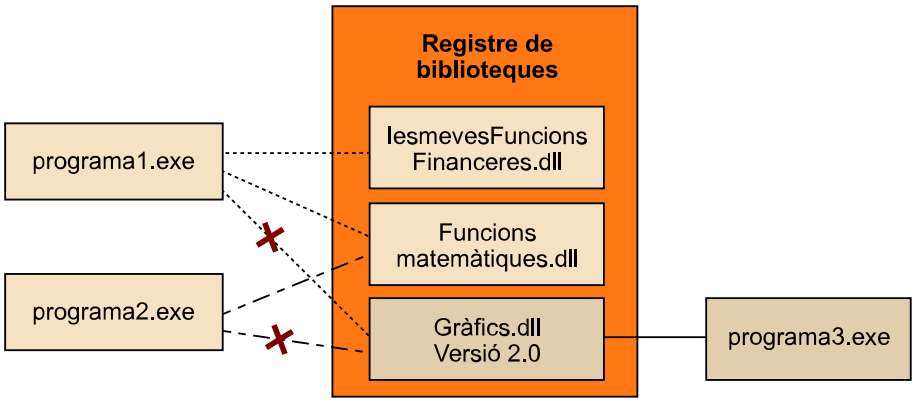

### 3.3.2. Biblioteques d'enllaç dinàmic

La solució al problema anterior consisteix en la implementació de biblioteques d'enllaç dinàmic (*dynamic-link libraries* o DLL), que no requereixen enllaçar el codi escrit pel programador amb el codi inclòs en la biblioteca. Ara, quan un executable necessita cridar una funció de biblioteca, la demana al sistema operatiu, que desa un registre de totes les DLL amb les seves funcions. D'aquesta manera, dos programes que requereixin usar la mateixa funció no tenen codi duplicat, sinó que comparteixen un únic exemplar de la biblioteca.

En la figura següent s'il·lustra aquest concepte: tots dos programes utilitzen (comparteixen) les biblioteques de funcions matemàtiques i de maneig de gràfics, i el programa1 és l'únic que utilitza les funcions financeres. Com s'observa, les biblioteques es troben en fitxers separats que no s'enllacen per a generar els executables.

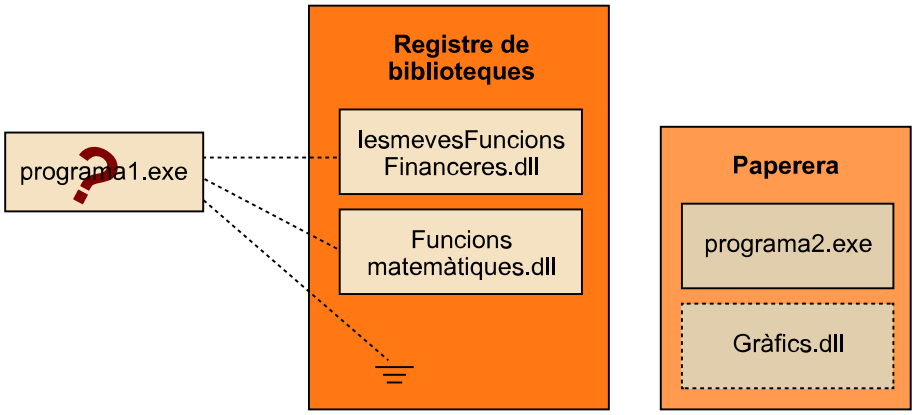



No obstant això, la introducció de biblioteques d'enllaç dinàmic va representar també l'aparició de problemes nous que, amb l'enllaç estàtic, no es presentaven: així, la instal·lació d'una nova aplicació en el sistema podia reemplaçar una versió antiga d'una DLL per una de més moderna, i de vegades passava que aquesta nova versió era incompatible amb altres programes instal·lats prèviament. Si, en la figura anterior, s'instal·la un nou *programa3* en el sistema que instal·la també una versió nova de la biblioteca *Gràfics.dll* i aquesta és incompatible amb els dos programes preexistents, aquests podrien deixar de funcionar.




Anàlogament, la desinstal·lació d'una aplicació pot representar l'eliminació de les seves biblioteques d'enllaç dinàmic associades. Si el sistema operatiu no porta un control adequat de les aplicacions que fan ús de les biblioteques registrades, o si hi ha hagut instal·lacions o desinstal·lacions manuals, és possible que s'eliminin biblioteques requerides per altres aplicacions. La figura següent il·lustra aquesta situació: per un error en el registre de biblioteques,

l'eliminació del *programa2* comporta també l'eliminació de *Gràfics.dll*, per la qual cosa és previsible que el *programa1* deixi de funcionar o tingui comportaments inesperats.




### 3.3.3. Altres biblioteques

El terme *biblioteca* s'utilitza normalment per a fer referència a biblioteques de funcions, però es poden referir a qualsevol tipus de magatzem estàtic de recursos reutilitzables, com arxius d'imatges o de sons.

## 3.4. Classes

La creixent implantació de l'orientació a objectes, amb les capacitats per a l'abstracció que permeten l'herència, el polimorfisme i l'encapsulació de comportament i dades en una mateixa unitat, va afavorir l'aparició de diversos tipus d'elements reutilitzables: per al desenvolupament d'un nou projecte, una companyia pot reutilitzar els dissenys fets per a un projecte desenvolupat amb anterioritat; pot també reutilitzar classes que ofereixin un conjunt de serveis que, novament, puguin ser útils; i pot, en tercer lloc, reutilitzar el codi font, mitjançant algunes modificacions, per a adaptar-lo al nou desenvolupament.

Com que la documentació tècnica del projecte, en molts casos, no evoluciona d'acord amb la implementació a la qual va donar lloc, la reutilització pura i simple del disseny d'un sistema porta el risc greu de reutilitzar un disseny que contingui defectes que es van detectar amb el producte ja implementat, i que van ser corregits només sobre el codi font: com indica Meyer, la sola reutilització del disseny pot portar a reutilitzar elements incorrectes o obsolets. Per a aquest autor, la reutilització de dissenys s'ha de dur a terme amb un enfocament que elimini la distància entre disseny i implementació, de tal manera que el disseny, per exemple, d'una classe, pugui ser considerat com una classe a la qual falta una mica d'implementació; i al revés: gràcies als mecanismes d'abstracció, una classe implementada es pot considerar un element de disseny més.

La reutilització del codi font, d'altra banda, té l'avantatge que permet al reutilitzador modificar-lo per a adaptar-lo a les seves necessitats reals; no obstant això, en lliurar tot el detall de la lògica del sistema s'està violant el principi d'ocultament d'informació, i el desenvolupador pot estar temptat, en no arribar a entendre exactament la política de privadesa o publicitat de les operacions, d'alterar el codi, i introduir efectes col·laterals no desitjats.

**Nota**

No sempre es vol posar a la disposició de tercers el codi de les aplicacions: recordeu que una de les raons que van portar Nantucket Corporation a crear el Clipper va ser que els programadors del dBase es veien obligats a lliurar el seu codi.

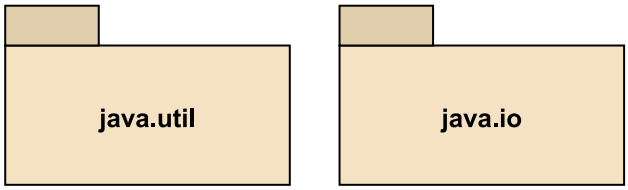
L'element bàsic de reutilització en orientació a objectes és la classe. Normalment, s'agrupa en un paquet (*package*, en UML) un conjunt cohesionat de classes reutilitzables.

Un paquet d'UML és un mecanisme utilitzat per a agrupar elements de qualsevol tipus (casos d'ús, classes, diagrames de seqüència d'un cert cas d'ús, una barreja de casos d'ús i classes, altres paquets, etc.).

Si bé els paquets de classes són els més interessants des del punt de vista de la reutilització: el paquet *java.util* ofereix un conjunt molt ampli de classes i interfícies per a manejar col·leccions, esdeveniments, dates i hores, internacionalització i unes quantes més; *java.io* conté classes per a manejar l'entrada i sortida.

**Nota**

Dins del JDK, Java ofereix una API àmplia amb paquets formats per classes i interfícies relacionades per a una varietat enorme d'usos utilitaris que permet als desenvolupadors fer-ne ús per construir aplicacions Java. L'API per a la versió 7 de Java SE es troba a docs.oracle.com/javase/7/docs/api/.



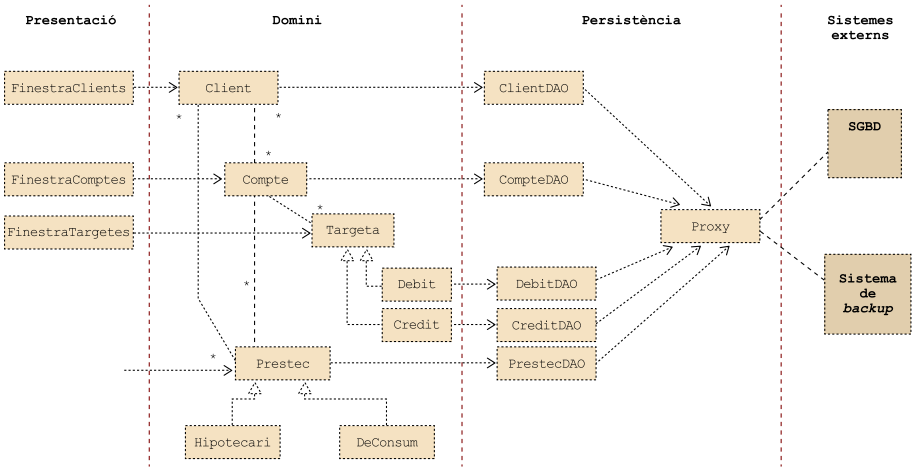

### 3.5. Patrons arquitectònics

L'arquitectura (o disseny arquitectònic) d'un sistema es correspon amb el seu disseny al més alt nivell. En plantejar l'arquitectura d'un sistema es prescindeix dels detalls, enfocant l'atenció en aquells elements generals que, gràcies al principi d'abstracció, duen a terme una o més tasques de computació. Tal com un disseny arquitectònic equivocat portarà probablement al fracàs del projecte, un de bo permet:

- Reconèixer relacions d'alt nivell entre sistemes o subsistemes.
- Afavorir la reutilització i l'ampliació de sistemes o subsistemes ja existents.
- Fer una aportació fonamental per a entendre i descriure, a alt nivell, un sistema complex.

### 3.5.1. Arquitectura multicapa

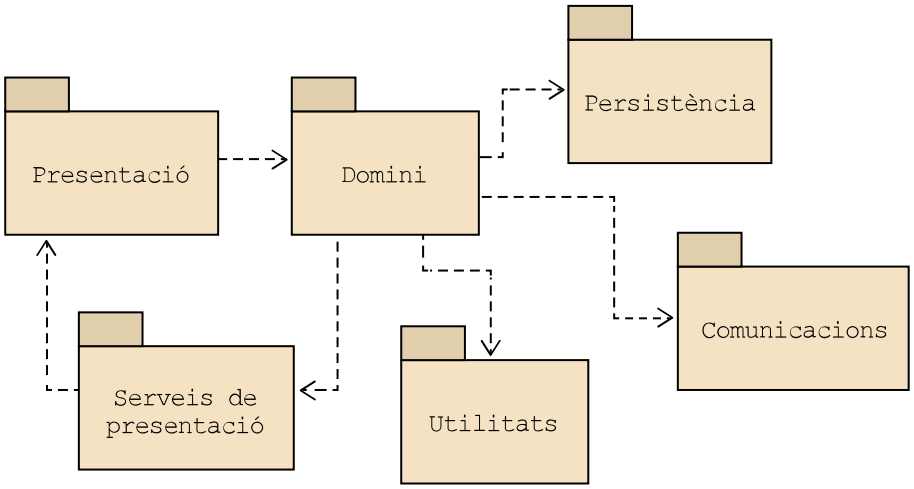
En un disseny arquitectònic multicapa, el sistema es descompon en capes (*tiers* o *layers*), en cadascuna de les quals se situa un conjunt més o menys relacionat de responsabilitats. En rigor, una capa ha de conèixer només els seus adjacents. La figura següent mostra el disseny d'un sistema amb tres capes: una primera capa de presentació, en la qual se situen les classes que utilitza l'usuari per a interactuar amb el sistema, i que estaran constituïdes per finestres; una segona capa de domini, en la qual es troben les classes de l'enunciat del problema, i que són les que tenen la responsabilitat real de resoldre'l; una tercera capa de persistència, en la qual se situen les classes que s'encarreguen de gestionar la persistència de les instàncies de classes de domini. Hi ha, a més, una quarta capa, en la qual es troben altres sistemes externs amb els quals es comunica aquest però que, en rigor, està fora de l'abast del sistema objecte d'estudi.




Arquitectònicament, les capes es poden representar com a paquets d'UML. Les relacions de dependència representen habitualment relacions d'ús: en la figura següent, la capa de presentació coneix *domini*; com que *domini* és la capa que conté les classes que veritablement resolen el problema (i que tenen la major part de la complexitat i, probablement, el principal interès a ser reutilitzades) interessa que el domini estigui poc acoblat a la resta de capes: per això, es desacobla el domini de la presentació mitjançant un subsistema de serveis (que inclourà observadors, patró que es veurà més endavant), i es desacobla (mitjançant indireccions, façanes, fabricacions pures, servidors intermedis, etc.), de les capes de persistència i comunicacions (patrons que també s'expliquen més endavant).

**Vegeu també**

Vegeu els patrons a l'apartat 3.6. En particular, a l'apartat 3.6.2. trobareu el patró *Fabricació pura* que explica les classes DAO de la figura.




### 3.5.2. Arquitectura de *pipes and filters* (canonades i filtres)

Els dissenys arquitectònics de *pipes and filters* es poden representar com a grafs en els quals els nodes es corresponen amb filtres[17] i els arcs es corresponen amb canonades (*pipes*)[18].

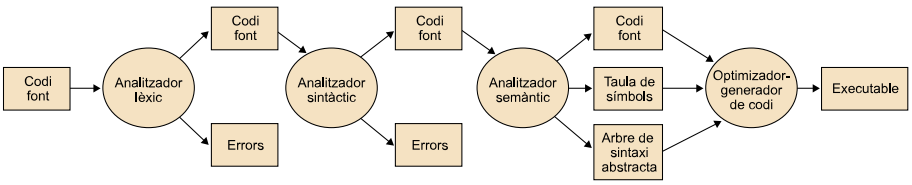
[17] Elements o components que agafen entrades i produeixen sortides.

[18] Mecanismes de comunicació entre els nodes.

Cada filtre llegeix cadenes de dades de les seves entrades, les transforma (les "filtra") i produeix sortides, que deixa en les canonades perquè siguin llegides per altres filtres. Els filtres són independents (no comparteixen el seu estat amb altres filtres) i no coneixen ni els seus filtres predecessors ni successors.

> Un filtre és un element que simplement rep informació, la transforma i la deixa en algun lloc, i se'n desentén en aquest moment.

L'arquitectura de *pipes and filters* s'utilitza, per exemple, en el desenvolupament de compiladors:




Algunes característiques d'aquestes arquitectures són:

- És fàcil comprendre el funcionament general del sistema.
- És fàcil reutilitzar filtres.
- El manteniment és relativament senzill.

- També és relativament senzill fer alguns tipus d'anàlisis especialitzades, com la identificació d'interbloquejos i colls d'ampolla.

- És fàcil fer execucions concurrents, posant cada filtre a executar-se en un procés separat.

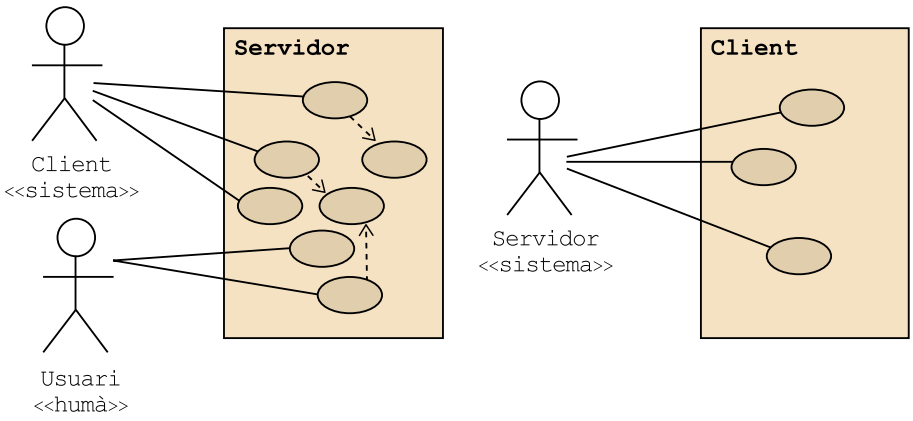
- No obstant això, són difícilment aplicables a sistemes interactius, servint gairebé exclusivament per a sistemes en lot.

**Nota**

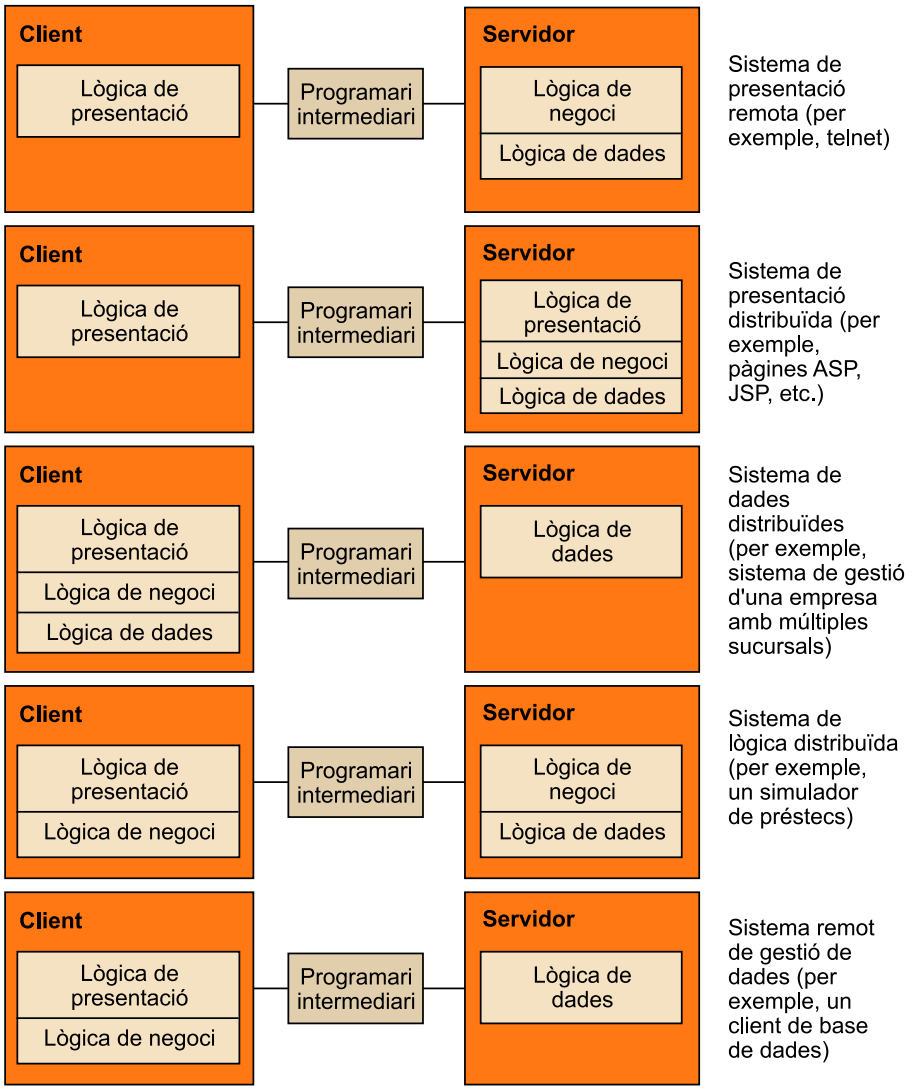
Els sistemes en lot també es coneixem com *batch*.

### 3.5.3. Arquitectures client-servidor

Els **sistemes client-servidor** consisteixen en aplicacions amb lògica distribuïda entre dues màquines remotes: el client i el servidor, que es comuniquen per mitjà d'algun tipus de programari intermediari. En construir el sistema, és convenient entendre'l com dos sistemes: el client s'interpreta com un actor que actua sobre el servidor, i el servidor com un actor que actua sobre el client.

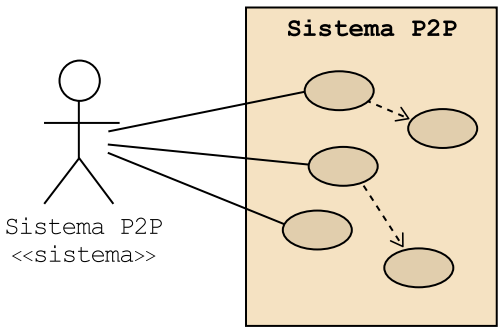



Amb aquesta consideració, cadascun dels dos subsistemes pot tenir un disseny arquitectònic diferent, si bé serà necessari que els desenvolupadors de tots dos determinin les responsabilitats que es trobaran en el client i quines es trobaran en el servidor. En funció d'aquesta distribució, es poden donar diferents situacions, com per exemple les cinc que es mostren a continuació:

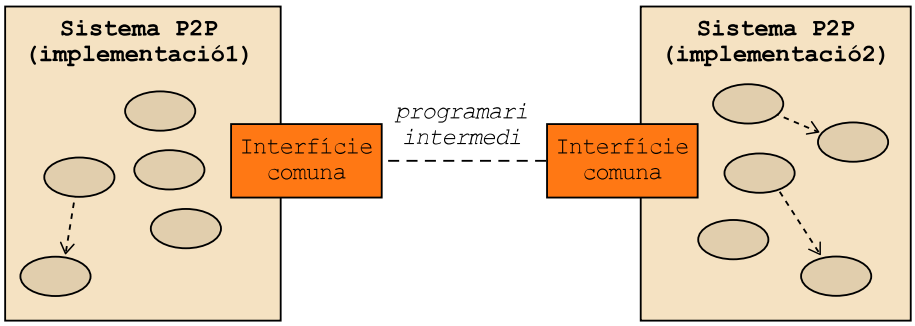



### 3.5.4. Arquitectures P2P (*peer-to-peer*, d'igual a igual)

En aquest cas, dos sistemes iguals es relacionen entre si. A molt alt nivell, un sistema P2P es pot entendre com un sistema que es relaciona amb altres sistemes que són com ell: vegeu, en la figura, que el nom del sistema (*sistema P2P*) coincideix exactament amb el nom de l'actor amb el qual es comunica:



En realitat, els dos sistemes poden ser diferents, però s'han de comunicar entre si oferint i implementant les mateixes interfícies. Aquestes, al costat de la seva implementació, es poden incloure en un component reutilitzable que s'integri en diferents tipus de clients P2P. Amb una notació una mica lliure, encara que de tipus UML, podem representar això de la manera següent:




Per a implementar les interfícies i permetre la comunicació entre els parells es pot reutilitzar Gnutella, un protocol per a compartir arxius entre parells que és P2P "pur": no hi ha servidor central i tots els nodes tenen exactament les mateixes funcions dins de la xarxa. Gnutella defineix la sintaxi dels missatges perquè un node s'enllaci a una xarxa ja existent, busqui arxius dins de la xarxa i descarregui arxius.

### 3.6. Patrons de disseny

Els patrons de disseny són solucions bones a problemes de disseny de programari que es presenten freqüentment. De fet, perquè una solució es pugui considerar un patró, ha d'haver estat provada amb èxit en més d'una situació. La solució, a més, s'ha de descriure de manera suficientment general perquè sigui reutilitzable en contextos diferents.

**Nota**

En l'assignatura *Anàlisi i disseny amb patrons* del grau d'Enginyeria Informàtica ja es descriuen molts d'aquests patrons. En aquest text els intentem presentar amb un punt de vista més orientat a la reutilització.

Com es veurà, molts patrons requereixen l'aplicació a nivells successius d'indirecció (és a dir, en lloc d'enviar el missatge directament a l'objecte interessat, s'envia a un intermediari) que, en molts casos, poden aparentar ser massa artificials. En general, tots els patrons persegueixen el disseny de sistemes amb alta cohesió i baix acoblament, la qual cosa produeix elements programari més reutilitzables. De vegades, no obstant això, la utilització exagerada de patrons pot donar lloc a dissenys molt complexos, difícils de seguir i entendre, la qual cosa en pot dificultar el manteniment.

El conjunt de patrons més conegut és el de Gamma i col·laboradors que, el 1995, van publicar un cèlebre llibre titulat *Design Patterns: Elements of Reusable Object-Oriented Software*, del qual s'han imprès multitud d'edicions, i ha estat traduït a nombrosos idiomes. Altres autors, com Craig Larman, han publicat també llibres molt reeixits d'enginyeria del programari en els quals inclouen

els seus catàlegs de patrons propis: en el cas de Larman, aquests patrons es coneixen com a *patrons GRASP* (*general responsibility assignment software patterns*: patrons generals d'assignació de responsabilitats).

### 3.6.1. Patrons de Gamma

En el seu llibre, Gamma i altres presenten 23 patrons agrupats en tres categories:

- **Patrons de creació**, que pretenen resoldre el problema de com crear objectes complexos.

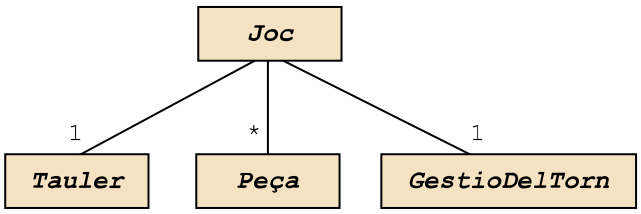
- **Patrons estructurals**, que descriuen solucions sobre com unir objectes perquè formin estructures complexes.

- **Patrons de comportament**, que presenten algorismes i mecanismes per a comunicar objectes.

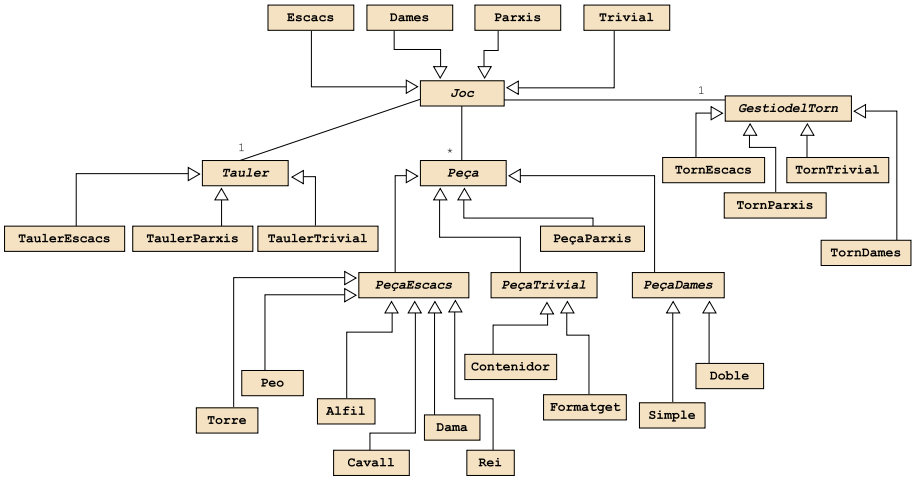
A continuació se'n descriuen alguns.

#### Patró de creació *Abstract Factory* (fàbrica abstracta)

Suposem que hem de crear un sistema de programari per a jugar a diferents tipus de jocs de taula: dames, escacs, parxís i Trivial. A més de les regles i polítiques de gestió del torn pròpies, cada joc té un tipus diferent de tauler i unes peces diferents. Aprofitant aquesta estructura comuna dels jocs, podem crear una estructura de classes abstractes per a representar els tres jocs en general:



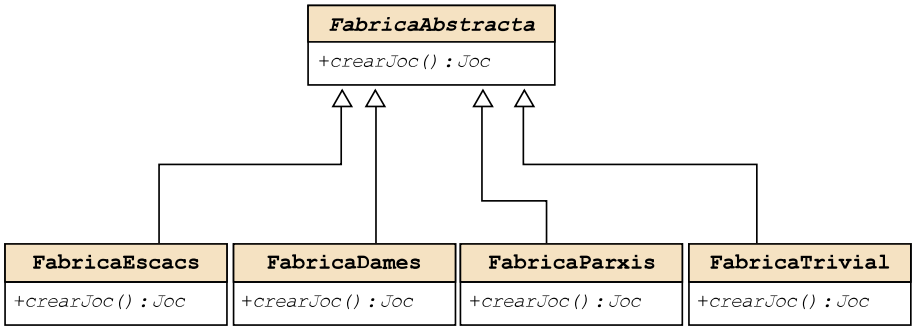

El nombre d'especialitzacions d'aquestes classes abstractes dependrà del nombre de jocs i de les característiques específiques de cada joc:



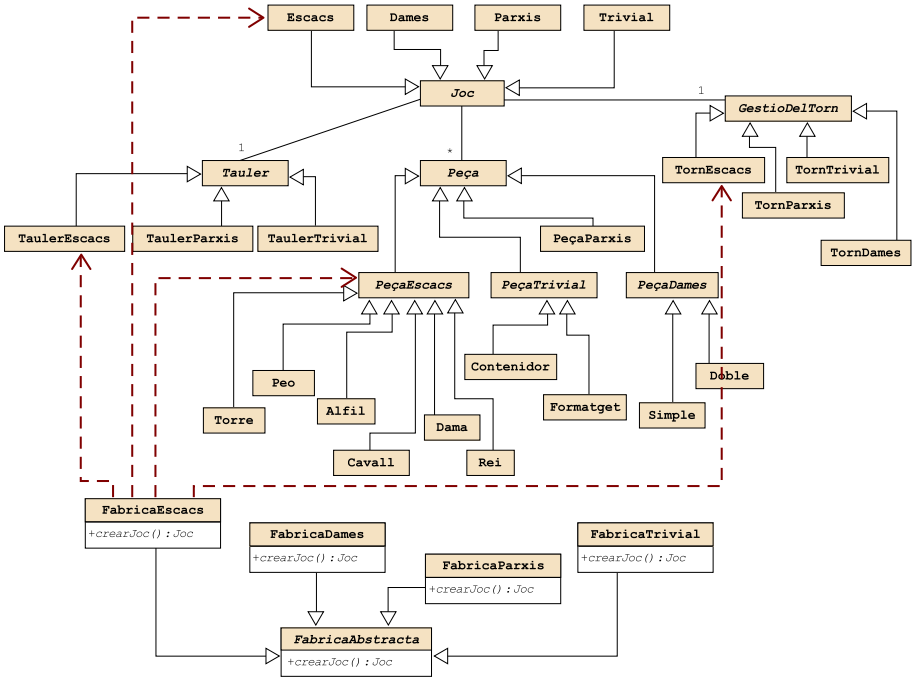

A l'hora de crear un objecte, per exemple, de la classe *Escacs*, caldrà instanciar un *Tauler* d'escacs, diverses peces (peons, alfils, etc.) de colors diversos i la seva política concreta de gestió del pas del torn entre els jugadors. El patró *Abstract Factory* ens ajuda a resoldre la pregunta següent:

A qui hem d'assignar la responsabilitat de crear una família d'objectes complexos?

La resposta és assignar-la a una classe externa, que fabriqui els objectes. Hi haurà una fàbrica abstracta i tantes especialitzacions concretes com a famílies d'objectes hi hagi:




Cada fàbrica concreta és responsable de crear tots els objectes corresponents a la família d'objectes a la qual està associada. La *FabricaEscacs*, per exemple, s'ocupa solament amb els subtipus de *Tauler*, *Peça* i *GestioDelTorn* que li corresponen:




La implementació del mètode *crearJoc()*, per exemple, a *FabricaEscacs*, serà d'aquest estil:

```
public Joc crearJoc() {
   Joc resultat=new Escacs();
   resultat.setTauler(new TaulerEscacs());
   resultat.setPeces(buildPeces());
   resultat.setTorn(new TornEscacs());
   return resultat;
}
```

### Patró de creació *Builder* (constructor)

El *Builder* proporciona una solució diferent al mateix problema plantejat en *Abstract Factory*. En essència, la diferència és que, ara, un dels objectes de la família és el responsable de crear la resta d'objectes. Si en l'exemple assignem a la classe *Joc* les responsabilitats de construir els objectes, el codi de la classe pot prendre aquesta forma:

```
public abstract class Joc {
   private Tauler tauler;
   private Vector<Peces> peces;
   private GestioDelTorn torn;

   public Joc() {
      this.tauler=crearTauler();
      this.peces=crearPeces();
      this.torn=crearTorn();
   }

   protected abstract Tauler crearTauler();
   protected abstract Vector<Peces> crearPeces();
   protected abstract GestioDelTorn  crearTorn();
}
```

El codi anterior mostra també un exemple del patró de comportament *template method* (mètode plantilla): noteu que, des d'una operació concreta (el constructor *Joc()*) d'una classe abstracta (*Joc*), s'està cridant tres operacions declarades com a abstractes en la classe mateixa (*crearTauler(),crearPeces()* i *crearTorn()*). El patró mètode plantilla descriu el comportament genèric d'una jerarquia d'objectes, però deixant a les subclasses la responsabilitat concreta d'executar les operacions: en el constructor, es diu que el primer que cal fer és crear el tauler, després les peces i després el sistema de gestió del torn; no obstant això, deixa a les classes concretes (*Escacs*, *Dames*, *Parxis* i *Trivial*) la responsabilitat concreta de com fer-ho.

### Patró de creació *Singleton* (creació d'una única instància)

De vegades és necessari garantir que, per a alguna classe, es pot crear un màxim d'una instància: pot interessar, per exemple, dotar el sistema d'un únic punt d'accés a una base de dades externa, de manera que aquest punt d'accés gestioni les múltiples connexions que es puguin establir.

Per a garantir que es crea una sola instància d'aquest punt d'accés (que, d'altra banda, es pot entendre com un patró *Proxy* o un patró *Broker* (agent) de base de dades, de Larman), la classe que l'implementa pot ser un *Singleton*. En general, un *Singleton* té:

- Un camp estàtic del mateix tipus que la classe *Singleton* que, per exemple, es diu *jo*.

- Un constructor de visibilitat reduïda (privat o protegit).

- Un mètode públic estàtic per a recuperar l'única instància existent de la classe que, per exemple, es pot anomenar *getInstancia*. En cridar aquest mètode, es pregunta si existeix *jo* (és a dir, si és diferent de *null*): si *jo* sí que és diferent de *null*, el mètode retorna la instància; si no, es diu al constructor (que sí que pot veure *getInstancia*), amb la qual cosa es construeix *jo* i, després, es retorna la instància creada recentment.

- El conjunt de mètodes de negoci que correspongui, que no han de ser necessàriament estàtics.

El codi següent mostra un exemple d'un agent de base de dades *Singleton* que, malgrat ser únic, és capaç de gestionar fins a mil connexions simultànies a la base de dades: quan un client necessita executar una operació sobre la base de dades, en lloc de crear ell mateix la seva pròpia connexió, recupera l'agent (cridant *getInstancia*); una vegada que té la referència a l'agent, el client li pot demanar que li retorni la connexió.

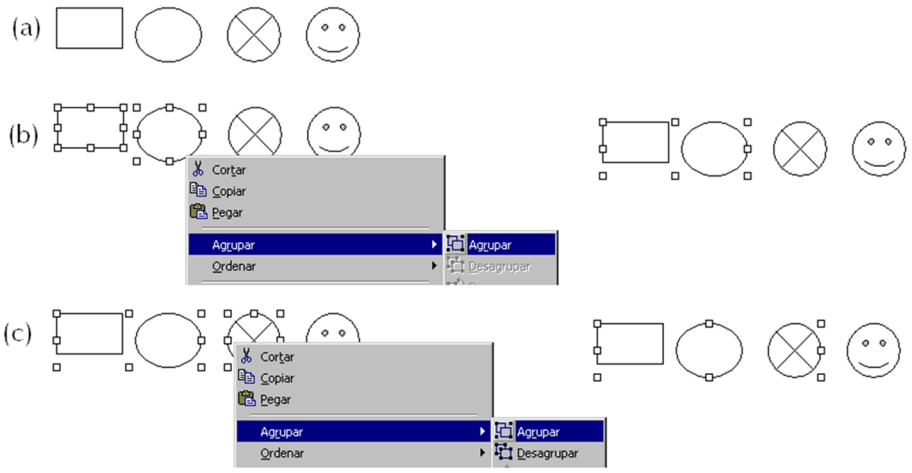
```
package persistencia;

import …;

public class Broker {
   protected static Broker jo;
   private Vector<Connexio> lliures, ocupades;
   private static int CONEXIONES = 1000;

   protected Broker () throws ClassNotFoundException, SQLException {
      this.lliures=new Vector<Connexio>();
      this.ocupades=new Vector<Connexio>();
      for (int i=0; i<CONNEXIONS; i++) {
         this.lliures.add(new Connexio(i+1));
      }
   }


   public static Broker getInstancia() throws
            NoHihaConnexionsLliuresException, SQLException {
      if (jo==null) {
         try {
            synchronized(Broker.class) {
               jo=new Broker ();
            }
         } catch (ClassNotFoundException e) {
            throw new NoHihaConnexionsLliuresException();
         }
      }
      return jo;
   }

   public Connexio donamConnexio() throws
         NoHihaConnexionsLliuresException {
      …
   }
   …
```

### Patró estructural *Adapter* (adaptador)

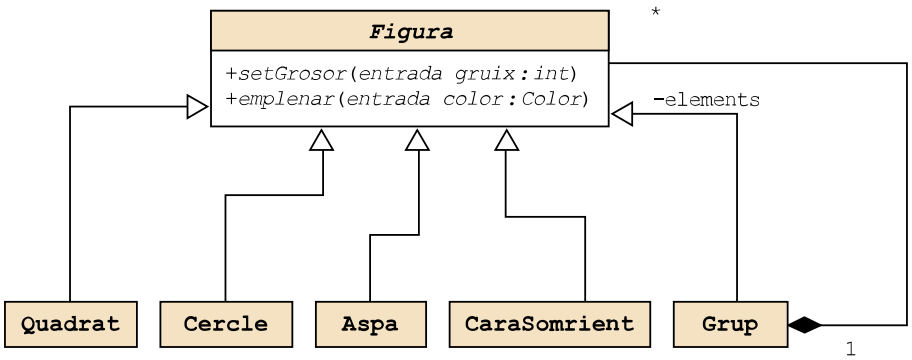
Aquest patró s'aplica quan una classe client vol utilitzar una altra classe servidora que, no obstant això, li ofereix una interfície diferent de la que espera el client. Ho veurem amb cert detall en la secció de components.

### Patró estructural *Composite* (compost)

De vegades, un objecte està format per objectes que, al seu torn, posseeixen en el seu interior objectes que són de la classe de l'objecte original. En el processador de textos Microsoft Word, per exemple, és possible agrupar diversos objectes de dibuix en un "grup"; aquest grup es maneja com un únic objecte, i pot ser també agrupat al costat d'altres objectes, de manera que un grup pot contenir, barrejats, objectes simples i grups. La figura següent mostra quatre objectes senzills dibuixats amb el Word (a); en el costat esquerre de (b) creem un grup a partir de dos objectes, que ja es mostren agrupats en el costat dret; en el costat esquerre de (c), creem un nou grup format pel grup creat anteriorment i un nou objecte simple, el resultat del qual es mostra a la dreta.




Tots aquests objectes de dibuix poden respondre, per exemple, a una operació *setGruix(pixels:int)*, que assigna el *gruix* que es passa com a paràmetre a les línies que conformen el dibuix, o a *emplenar* (*color:Color*) per a emplenar la figura amb el *color* que s'estableixi. Per a manejar de manera uniforme tot el grup de figures es pot utilitzar el patró *Composite*, en el qual les figures simples i compostes es representen d'aquesta manera:




S'està dient que les figures poden ser figures simples (quadrats, cercles, aspes o cares somrients) o figures compostes (grups); aquestes, al seu torn, estan formats per figures que també poden ser simples o compostes. Com s'observa, totes les subclasses són concretes, la qual cosa significa que implementen les dues operacions abstractes que es mostren en la superclasse. Les figures "simples" donaran a les operacions la implementació que correspongui; la implementació d'aquestes operacions en *Grup*, tanmateix, consistirà a cridar la mateixa operació en totes les figures que coneix, que són accessibles per mitjà de la relació d'agregació:

```
public void setGruix(int gruix) {
   for (Figura f : elements)
      f.setGruix(gruix);
}
```

En temps d'execució es decidirà, en funció del tipus dels objectes continguts en el *Grup*, quina versió de l'operació s'ha d'executar.

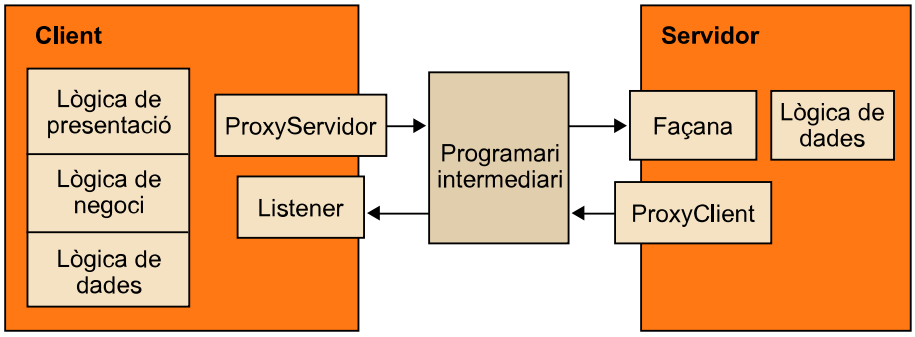
### Patró estructural *Façade* (façana)

Una façana representa un punt d'accés únic a un subsistema. Si, en una aplicació multicapa, disposem de diverses finestres en la capa de presentació i de diverses classes en la capa de negoci, pot ser convenient crear una façana que representi el punt d'accés únic des de la capa de presentació a la de domini. Quan un objecte de presentació li vol dir alguna cosa a un de domini no l'hi dirà directament, sinó que farà la petició a la façana, que l'encaminarà a l'objecte de domini que correspongui.

### Patró estructural *Proxy*

Un *Proxy* representa un substitut d'"alguna cosa" que, en general, serà un sistema o dispositiu extern. En lloc de permetre que qualsevol part del sistema accedeixi directament al sistema extern, col·loquem un *Proxy* intermedi que serà el seu únic punt d'accés. Podem considerar que una *Façana* és un *Proxy* d'accés a un subsistema, i és cert; quan utilitzem una classe intermèdia per a accedir a un sistema o dispositiu extern (un servidor o un escàner, per exemple), ho anomenarem *Proxy*. Els *proxies*, sovint, són a més classes *Singleton*.

La figura següent completa un dels dissenys arquitectònics que mostràvem en parlar dels estils arquitectònics client-servidor amb dos *proxies*: un, en el sistema client, per a accedir al servidor; un altre, en el servidor, per a comunicar amb els clients. El servidor, a més, ofereix els seus serveis per mitjà d'una façana (que, d'altra banda, ha d'implementar algun tipus que permeti la comunicació remota); el client escolta el servidor per mitjà d'un *listener*, que es pot entendre com una façana més simplificada.



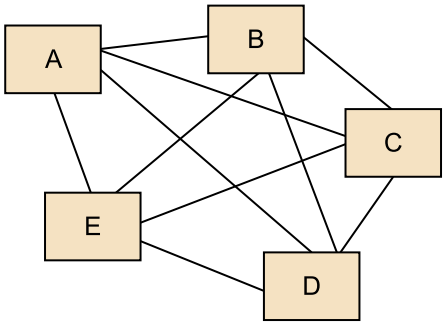

### Patró de comportament *Chain of responsibility* (cadena de responsabilitat)

Aquest patró s'utilitza per a desacoblar l'objecte emissor d'una petició de l'objecte que l'ha de rebre. La idea implica tenir una cadena d'objectes que passin la petició fins que aparegui un objecte que sigui capaç d'atendre-la.

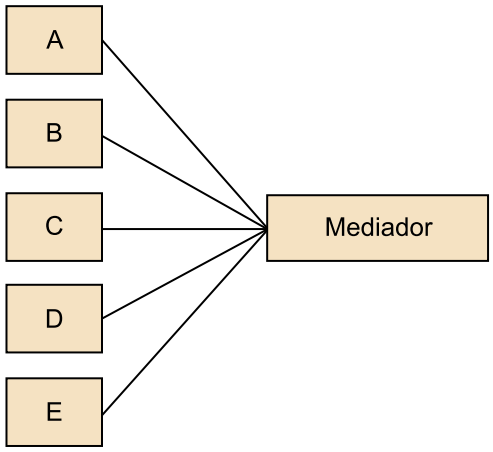
La solució es modelitza com una superclasse abstracta amb una operació abstracta que es correspon amb la petició; aquesta classe, a més, coneix un objecte del seu mateix tipus, que s'anomena *successor*. Totes les subclasses concretes implementen l'operació: les que poden atendre la petició, l'atenen; les que no, la deriven al successor, que estan heretant de la superclasse.

### Patró de comportament *Mediator* (mediador)

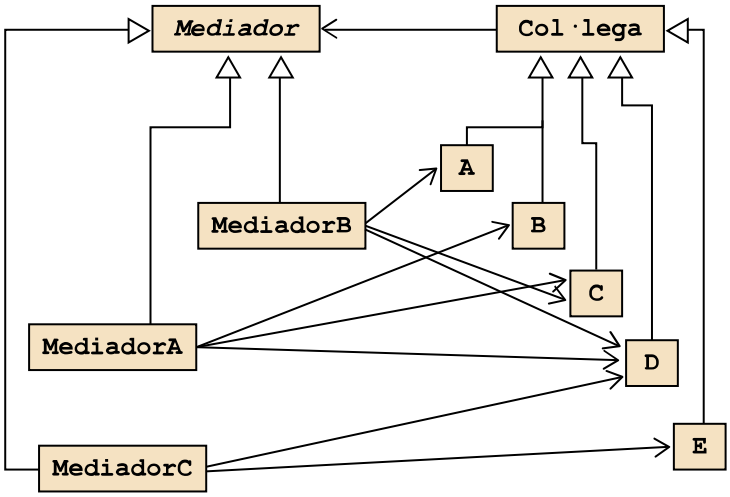
De vegades, un canvi en un objecte qualsevol d'una família d'objectes ha de ser notificat a tota la resta d'objectes. Suposem que tenim cinc classes les instàncies de les quals han d'estar pendents dels canvis d'estat que es produeixen en els objectes de les altres quatre. Una solució possible és estructurar el sistema de tal manera que totes les classes es coneguin entre elles, com es mostra en aquesta figura:




Aquest disseny, no obstant això, és difícil de comprendre, de mantenir i de provar, i té un altíssim acoblament que, probablement, el faci molt propens a errors. El patró *Mediador* aconsella la creació d'una classe que rebi les notificacions dels canvis d'estat i que les comuniqui a les interessades:




El patró admet variants que aprofiten l'herència: si, per exemple, els canvis d'estat de A només interessen a B, C i D, els de B només a A, C i D, els de C només a D i E, i els altres no interessen, es pot replantejar el disseny d'aquesta manera:




**Patró de comportament *Observer* (observador)**

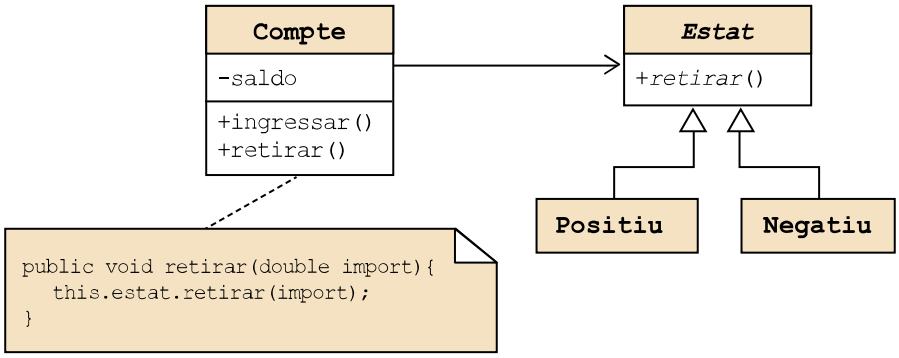
Si en el patró anterior es resolia el problema de com notificar canvis en múltiples objectes a altres múltiples objectes, el patró *Observador* dóna una solució a com notificar els canvis d'estat d'un únic objecte a un conjunt d'objectes que l'estan observant.

La solució consisteix en la creació d'una classe intermèdia (l'observador), a la qual subscriuen els objectes interessats a observar a l'objecte observable. Quan aquest experimenta un canvi d'estat, ho comunica a l'observador, que ho notifica als objectes que manté subscrits.

El patró *Alta Cohesió* (un dels patrons de Larman), que es presenta més endavant, inclou un exemple del patró *Observador*.

**Patró de comportament *State* (estat)**

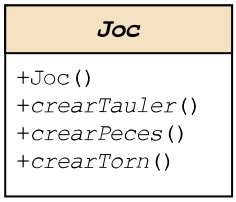
Aquest patró delega en una classe associada aquelles operacions el comportament de les quals depèn del seu estat. Suposem que l'operació *retirar()* d'un compte corrent depèn de si el seu estat és positiu o negatiu: l'operació es pot implementar en la classe mateixa *Compte* i, abans d'efectuar la retirada, preguntar si el saldo és suficient per a afrontar l'import que es vol retirar. Mitjançant el patró *Estat*, es crea una classe abstracta *Estat* amb dues especialitzacions: *Positiu* i *Negatiu*. Quan a la classe Compte li arriba una crida a *retirar()*, aquesta invoca l'operació *retirar()* del seu estat associat, que estarà instanciat a una de les dues subclasses.




**Patró de comportament *Template-Method* (mètode plantilla)**

Aquest patró s'utilitza per a representar de manera general el comportament d'una operació determinada: en una classe abstracta es crea una operació concreta que crida operacions abstractes i concretes.

Hem mostrat un exemple de codi en presentar el patró *Builder*, que correspon a la classe següent en UML:




### 3.6.2. Patrons de Larman

Larman presenta nou patrons que ajuden l'enginyer de programari a determinar a quin objecte s'ha d'assignar cada responsabilitat. L'autor explica que les responsabilitats dels objectes són de dos tipus: de conèixer[19] i de fer[20].

[19] Les seves dades pròpies, els d'objectes relacionats, i altres coses que pot derivar o calcular.

[20] Pot fer alguna cosa l'objecte mateix, pot iniciar una acció en un altre objecte o pot coordinar i controlar activitats entre objectes.

**Patró *Expert***

Aquest patró ens diu que la responsabilitat s'ha d'assignar a la classe "experta en la informació": és a dir, a aquella classe que té la informació necessària per a fer la responsabilitat.

**Patró *Creador***

Aquest patró diu que s'ha d'assignar a la classe A la responsabilitat de crear instàncies de B quan:

- A agrega o conté objectes de B.
- A registra objectes de B.
- A utilitza "més estretament" objectes de B.
- A té la informació d'inicialització necessària per a crear instàncies de B.

### Patró *Baix acoblament*

L'acoblament indica el grau en què una classe està relacionada amb d'altres: com més gran sigui l'acoblament, més dependència de la classe respecte de canvis en les altres. A més, en diversos estudis experimentals, s'ha comprovat que l'alt acoblament és el millor indicador de la propensió a fallades d'una classe.

Més que un patró, el manteniment del baix acoblament és un principi general de disseny de programari, incloent-hi el disseny de programari orientat a objectes.

La classe A està acoblada a B quan:

- A té un atribut de tipus B.
- A invoca algun mètode de B.
- En algun mètode de A es declara una variable de tipus B.
- En algun mètode de A s'utilitza algun paràmetre de tipus B.
- A és una subclasse, directa o indirecta, de B.
- B és una interfície, i A una implementació de B.

Per a mantenir el baix acoblament, s'han d'assignar les responsabilitats de manera que s'evitin, en la mesura del possible, relacions com les enumerades a dalt.

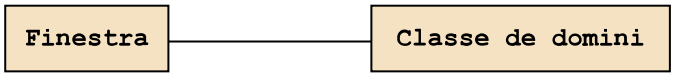
### Patró *Alta cohesió*

La cohesió dóna una mesura del grau en què estan relacionats els elements d'una classe. Una classe amb baixa cohesió fa diverses coses diferents que no tenen relació entre si, la qual cosa porta a classes més difícils d'entendre, de reutilitzar i de mantenir.

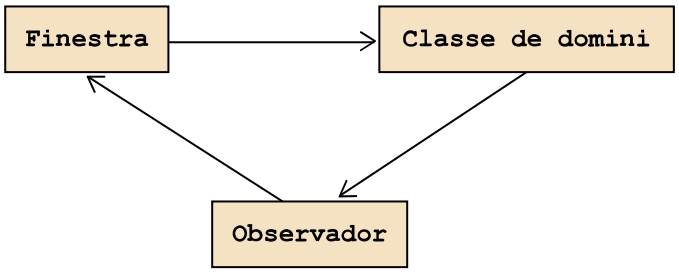
Igual que el baix acoblament, el manteniment de l'alta cohesió és també un principi general en el disseny de programari.

Moltes vegades, aconseguir una cohesió alta implica apujar també l'acoblament, i disminuir-lo sol implicar una baixada d'aquella. En la figura següent es mostra un petit diagrama amb dues classes: una **finestra**, situada en la capa de presentació, i una **classe de domini**, que tanca part de la lògica realment complexa del sistema. Els canvis d'estat en l'objecte de domini s'han de notificar a la finestra, a fi que l'usuari estigui al tant de l'evolució del sistema. En la solució que s'està representant, la finestra coneix el domini, i

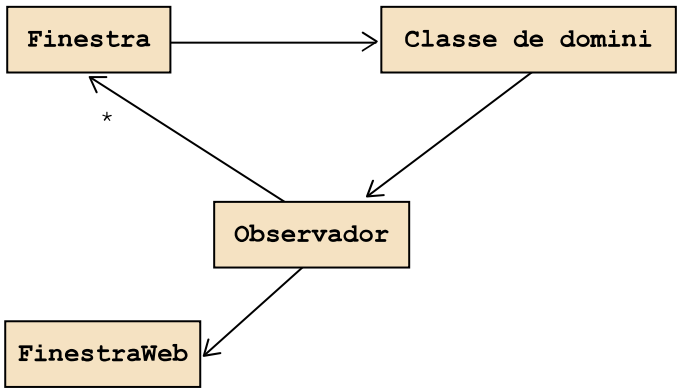
el domini coneix l'objecte. Des d'un punt de vista de manteniment de l'alta cohesió, l'objecte de domini no hauria de ser responsable de notificar a la finestra els seus canvis d'estat.




Podem introduir un observador intermedi que es responsabilitzi únicament de notificar a la finestra els canvis d'estat en l'objecte de domini:




D'aquesta manera, l'objecte de domini queda més cohesionat, ja que delega a una altra classe (l'observador) la responsabilitat de notificar els seus canvis a qui estigui interessat. L'acoblament general del sistema, Tanmateix, augmenta, ja que passem a tenir tres relacions en lloc de dues (de finestra a domini i de domini a finestra). Tanmateix, aquest acoblament nou és "menys dolent" que l'anterior, ja que la classe de domini deixa d'estar acoblada amb una finestra: si, en un futur, hi hagués més tipus de finestres interessades en l'objecte de domini, o si fos necessari notificar a més instàncies de la finestra actual, no caldrà reconstruir l'objecte de domini, ja que aquesta responsabilitat s'ha assignat a l'observador:




**Patró *Controlador***

Aquest patró recomana assignar els esdeveniments del sistema (els provocats per actors: usuaris o altres sistemes externs) a una classe controlador que:

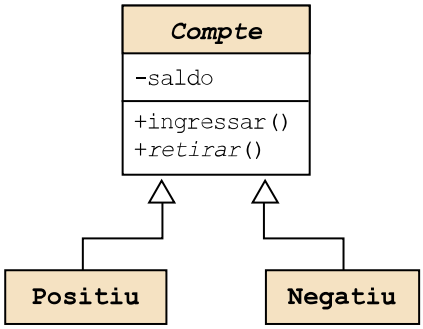
- Representi el sistema global (el que es correspon amb un "controlador de façana").

- Representi un escenari del cas d'ús en el qual es rep l'esdeveniment ("controlador de cas d'ús").

### Patró *Polimorfisme*

L'ús de múltiples instruccions condicionals en un mètode dificulta la reutilització de la classe en la qual està contingut. Per això, quan el comportament d'una operació variï segons el tipus de la classe, s'ha d'assignar la responsabilitat en funció dels subtipus per als quals variï el comportament.

Quan hem presentat el patró *Estat*, hem delegat a una classe associada l'execució de l'operació *retirar*; amb el patró *Polimorfisme*, deixem l'operació *ingressar* (el comportament de la qual sempre és igual) com a concreta en la superclasse *Compte*, que és abstracta perquè *retirar* sí que es comporta de manera diferent en funció del subtipus:




Per descomptat, aquest patró ens prohibeix la utilització d'operacions de comprovació del tipus, com la *instanceof* que permet Java.

### Patró *Fabricació pura*

Suposem que el *Compte* que hem utilitzat en l'exemple anterior és una classe persistent (és a dir, que les seves instàncies s'han d'emmagatzemar com a registres en una base de dades). Les operacions de gestió de la seva persistència[21] es poden situar en la classe mateixa i implementar-les aquí, incloent-hi la construcció d'instruccions SQL per a accedir a la base de dades.

[21] Anomenades habitualment *CRUD*, sigla de *create*, *read*, *update* i *delete*: respectivament, *Crear*, que correspondria a una instrucció SQL de tipus *Insert into*; *Llegir*, que correspon a un *Select from*; *Actualitzar*, que és un *Update*; i *Esborrar*, corresponent a *Delete from*.

Des del punt de vista del patró *Expert*, aquesta és la solució que s'hauria de seguir, ja que quina classe millor que el *Compte* coneix les seves dades pròpies i té més capacitat per a construir les instruccions SQL que corresponguin. No obstant això, incloure SQL en una classe de domini no és gaire convenient, ja que l'acobla directament al gestor de base de dades que s'estigui utilitzant. En situacions com aquesta, és més convenient crear una classe artificial, que no existeix en l'enunciat del problema que es pretén resoldre, i delegar a aquesta les operacions de persistència. Aquest tipus de classes artificials són les fabricacions pures: classes en les quals es deleguen una sèrie d'operacions que no corresponen realment a una altra. La cohesió augmenta, perquè la classe de

domini no fa coses que no li corresponen, i també la fabricació pura és una classe altament cohesiva. D'aquesta manera, quan una instància de la classe *Compte* es vol inserir, li ho demana a la seva fabricació pura associada.

Les classes *ClientDAO*, *CompteDAO*, etc., que es mostraven en el diagrama amb el qual hem il·lustrat l'arquitectura multicapa, són fabricacions pures que tenen delegades les responsabilitats de persistència de les classes de domini.

**Patró *Indirecció***

Mitjançant aquest patró, s'assigna la responsabilitat a un objecte intermedi. S'utilitza per a minimitzar l'acoblament directe entre dos objectes importants i per a augmentar el potencial de reutilització.

Noteu que una fabricació pura com la que hem comentat a dalt, per la qual s'assignen a una classe artificial les responsabilitats de persistència d'una altra classe de domini és, essencialment, una indirecció que desacobla el domini de la base de dades, i que permet que la classe *Compte* de l'exemple sigui més reutilitzable. Els patrons *Observador* i *Mediador* són també formes específiques del patró *Indirecció*.

**Patró *Variacions protegides***

Per a evitar que les classes o subsistemes inestables tinguin un impacte negatiu en la resta del sistema, aquest patró recomana crear una interfície estable entorn seu.

### 3.6.3. El patró *Model-vista-controlador*

El patró *Model-vista-controlador* (MVC, *Model-view-controller*) és un patró de disseny molt utilitzat per a mantenir la separació entre capes (en particular, per a separar les capes de presentació i domini).

El model es correspon amb el conjunt d'objectes que representen l'enunciat del problema, i que estan normalment situats en la capa de domini; la vista es correspon amb la interfície d'usuari (la GUI: *graphical user interface*), que aquest utilitza per a manipular i conèixer l'estat dels objectes de domini. Al mig se situa un controlador. Si bé hi ha implementacions diverses del patró MVC, en la més habitual el controlador rep esdeveniments des de la GUI i els passa al model; els canvis d'estat en els objectes del model es passen a la GUI també per mitjà del controlador, que actua llavors com una espècie de façana bidireccional.

## 3.7. Components

A diferència d'una classe, un component és, segons Sommerville (2007), una unitat d'implementació executable, construïda mitjançant un conjunt de classes, que ofereix un conjunt de serveis per mitjà d'una interfície. En general, l'usuari entén el component com una caixa negra (de fet, l'habitual és no disposar del seu codi font), de la qual coneix tan sols els serveis oferts per mitjà de la seva interfície, i també els resultats que l'execució d'aquests serveis li permetrà obtenir.

**Consulta recomanada**

**I. Sommerville** (1992). *Software Engineering* (8a. ed.). Addison Wesley.

A fi de que l'usuari pugui utilitzar amb garanties la funcionalitat oferta, Szyperski indica que la interfície del component ha d'estar especificada contractualment. Per a una operació determinada, el contracte ha d'especificar quins resultats es podran obtenir a partir de les entrades subministrades a l'operació, en termes de precondicions i postcondicions.

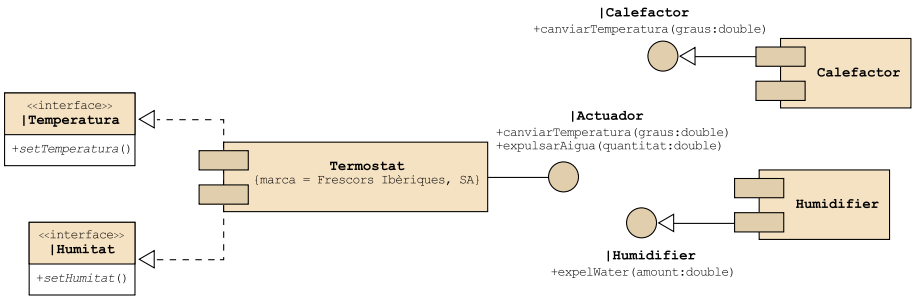
**Consulta recomanada**

**C. Szyperski** (2002). *Component Software: Beyond Object-Oriented Programming* (2a. ed.). Boston: Addison Wesley.

Szyperski també apunta que el component ha de ser "componible" (o, en altres paraules, susceptible de ser compost amb altres components), la qual cosa indica que ha de ser possible integrar un component amb un altre per a construir una aplicació basada en components. Per a això, pot ser necessari que el component disposi també d'una interfície requerida: és a dir, d'un mitjà de comunicació per a sol·licitar l'execució de serveis a altres dispositius, com en l'exemple del *Termòstat* que hem utilitzat en l'apartat 1 i que, com a bons reutilizadors, veiem novament en la figura següent amb la idea de poder connectar-lo a dos components més:

**1)** El component *Termòstat* ofereix dues interfícies (*ITemperatura* i *IHumitat*), que utilitza per a recollir dades de dos sensors externs. Aquest component, a més, requereix un dispositiu que implementi la interfície *IActuador*.

**2)** *Calefactor* i *Humidifier* són dos components que ofereixen dues interfícies diferents. L'operació *canviarTemperatura* oferta pel *Calefactor* és un subconjunt de la interfície requerida per *IActuador*. *Humidifier*, no obstant això, que podria ser un component importat del Regne Unit, ofereix una operació (*expelWater*) semblant a l'operació *expulsarAigua* requerida pel *Termòstat*.



Com s'observa, no podem connectar directament el *Termòstat* als altres dos components: en primer lloc, perquè les interfícies ofertes *ICalefactor* i *IHumidifier* són semblants, però no iguals, a la interfície *IActuador*, que és la que requereix el *Termòstat*; en segon lloc perquè, encara que totes dues interfícies eren absolutament compatibles, el *Termòstat* coneix un únic *IActuador*, però hi ha dos components als quals es poden connectar, ja que un actua sobre la temperatura i un altre sobre la humitat. Com es pot solucionar? A continuació explorarem algunes maneres de solucionar aquest problema.

### 3.7.1. Enginyeria del programari basada en components

L'enginyeria del programari basada en components (CBSE, per les sigles en anglès: *component-based software engineering*) es basa en el fet que els sistemes programari tenen una base comuna suficient que justifica, en molts casos, la utilització de components reutilitzables. Enfront d'això, l'enfocament tradicional de desenvolupament de sistemes involucra generalment la construcció de cadascuna de les peces des de zero, cadascuna implementada a propòsit per al projecte en curs. Algunes d'aquestes peces o elements provenen de projectes anteriors i són reutilitzats amb alguns ajustos addicionals.

**Vegeu també**

Vegeu l'assignatura *Enginyeria de programari de components i sistemes distribuïts*.

La tendència envers la reutilització i la disminució dels esforços de desenvolupament ha conduït a la CBSE, que tendeix a concentrar l'esforç en la composició de peces ja desenvolupades (els components) que, en general, posseeixen una funcionalitat autocontinguda. Alguns components, no obstant això, poden requerir una vinculació a altres components per a proveir la funcionalitat completa. Tant la interconnexió de components com la integració de components amb els elements del sistema que es van desenvolupant es fa entenent els components com a caixes negres, ja que s'utilitzen tal com estan disponibles i sense considerar cap modificació interna. De fet, la CBSE tracta del desenvolupament de sistemes a partir de la mera composició i integració de components reutilitzables.

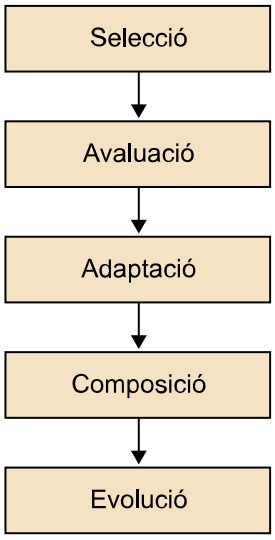
Així, el procés de desenvolupament tradicional es modifica en CBSE per a incorporar tasques de:

- **Selecció i avaluació de components**: és a dir, identificació de components candidats a servir una funcionalitat o part d'aquesta que ha de ser proveïda pel sistema.

- **Adaptació**: és possible que el component no pugui ser directament utilitzable i calgui fer algun tipus d'adaptació en el sistema per a poder usar-lo.

- **Composició**: per a enllaçar diversos components per mitjà de les seves interfícies.

- **Evolució**: es du a terme durant el manteniment del sistema. De vegades requereix la substitució d'algun component per una versió més moderna, o per un component diferent que ofereixi la mateixa interfície.




### Selecció

Una vegada que s'han determinat els requisits del sistema i se n'ha establert el disseny arquitectònic, l'equip de desenvolupament identifica aquelles funcionalitats que, en lloc de ser implementades de nou, puguin ser servides per un component preexistent. En particular, es preguntarà:

**1)** Disposem en l'empresa de components reutilitzables que hàgim desenvolupat nosaltres mateixos (*in-house components)*, i que serveixin la funcionalitat requerida?

**2)** Hi ha en el mercat components (*comercial off-the-self components, COTS*) que serveixin la funcionalitat requerida i que puguem utilitzar?

**3)** En cas que puguem utilitzar un component ja existent, són les seves interfícies compatibles amb l'arquitectura del nostre sistema?

En funció de les respostes a les preguntes anteriors, és possible que es redefineixin alguns requisits (funcionals o no funcionals) per a facilitar la integració dels possibles components reutilitzables identificats. Per a aquells requisits que, aparentment, sí que es puguin servir amb components, se segueixen quatre activitats importants: avaluació de l'adequació del component per al sistema; si és necessari, adaptació per a integrar-lo correctament en l'entorn operatiu; integració del component, i actualització del component.

### Avaluació

Un component se sol descriure en termes d'un conjunt d'interfícies que proveeixen l'accés a la funcionalitat del component. Sovint, aquestes interfícies són la principal font d'informació sobre el component, atès que la documentació adjunta ofereix, en general, tan sols una guia de les operacions disponibles en la seva interfície.

La integració de components en un sistema fa que l'equip de desenvolupament tingui el doble paper de desenvolupadors (perquè integra i utilitza el component dins de l'aplicació que està construint) i d'usuaris (utilitza la funcionalitat subministrada pel component), per la qual cosa és convenient analitzar doblement cada component candidat:

- Com a usuaris, s'avalua la qualificació del component per a complir la seva comesa: és a dir, s'avalua la seva capacitat per a servir els requisits.

- Com a desenvolupadors, s'avaluen característiques com la interfície, les eines de desenvolupament i integració necessàries, requisits maquinari (memòria, etc.), requisits programari (necessitat d'altres components, sistema operatiu, etc.), seguretat i manera com maneja les excepcions.

### Adaptació

Una vegada que s'han identificat un o més components per a ser integrats en el sistema, el pas següent és reparar qualsevol incompatibilitat que s'hagi detectat. En l'exemple del *Termòstat* observàvem que aquest component era incompatible amb els dos components amb els quals es volia integrar. Per a dur a terme les adaptacions se solen utilitzar adaptadors (*wrappers*, en anglès) que, per exemple, alteren la manera com es criden els serveis oferts. Els tres tipus d'incompatibilitats més comuns són:

**Nota**

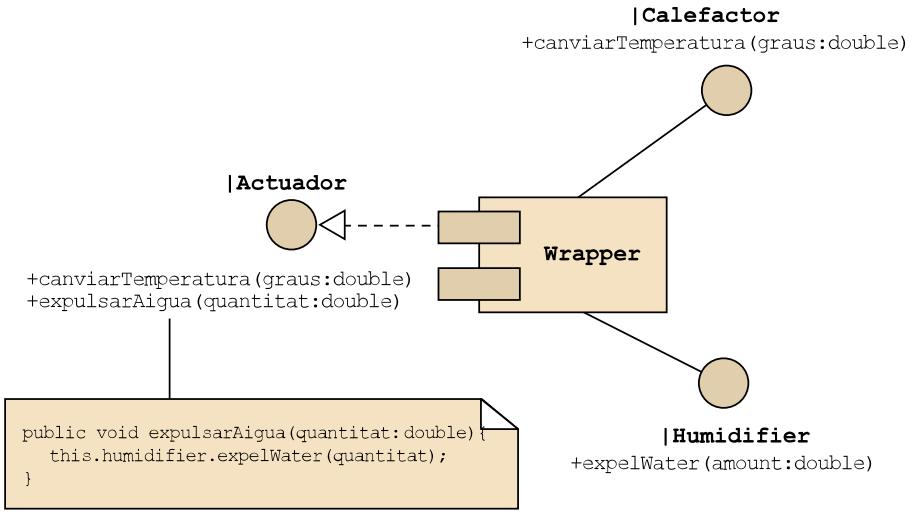
Alguns textos anomenen *embolcalls* els *wrappers*.

- **Incompatibilitat de paràmetres**: les operacions tenen el mateix nombre de paràmetres però estan en diferent ordre o són de tipus diferents. En aquest cas, el *wrapper* haurà de canviar l'ordre dels paràmetres o fer les conversions de tipus que corresponguin.

- **Incompatibilitat d'operacions**: els noms de les operacions en les interfícies esperada i oferta són diferents. En aquest cas, el *wrapper* rebrà una crida a l'operació esperada, i la reconduirà a la interfície oferta.

- **Incompletesa d'operacions**: les operacions ofertes en la interfície són només un subconjunt de les esperades. Aquest cas no és sempre solucionable, ja que el component client espera més del component servidor.

Per a l'exemple que estem utilitzant, un possible *wrapper* hauria d'oferir la interfície esperada pel *Termòstat*, i requerir les dues ofertes pel *Calefactor* i l'*Humidifier*. En aquest exemple, les dues operacions de *IActuador* estarien implementades en el *wrapper*, entre les quals *expulsarAigua(quantitat:double)*, que hauria d'adaptar les seves crides perquè fossin compatibles amb l'operació *expelWater(amount:double)*.




El disseny de *wrappers* comporta l'aplicació d'algun patró de disseny. Els tipus de *wrappers* més comuns i la seva relació pel que fa a patrons són:

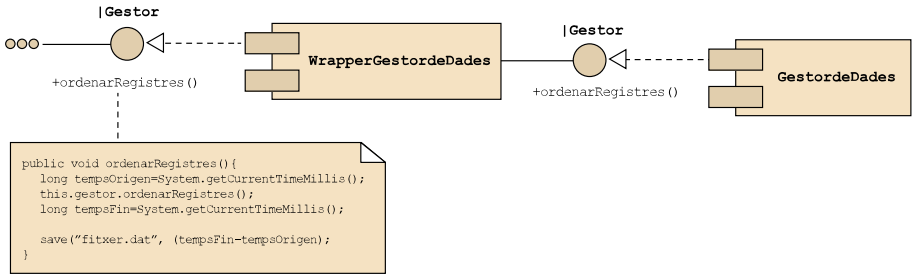
**1) *Wrappers* d'aïllament**, que proveeixen un únic punt d'accés mitjançant una sola API a múltiples interfícies, i es relaciona amb el patró *Façana*.

Un exemple d'això és l'API ODBC, que s'utilitza en els sistemes Windows per a proveir un punt únic d'accés a diferents tipus de bases de dades.

**2) Embolcalls**, que oculten o adapten algun aspecte del component, com en l'exemple de la figura anterior, i que es relaciona amb el patró *Adaptador* (*Adapter*).

**3) *Wrappers* d'instrumentació**, que s'utilitzen per a instrumentar, depurar, agregar assercions, etc.

Pot interessar, per exemple, conèixer quant temps triga un component a fer una operació determinada de les que ofereix per mitjà de la seva interfície: amb un *wrapper* d'instrumentació es captura la crida a l'operació i, abans d'executar-la, inicia un cronòmetre; quan el resultat es retorna, el *wrapper* el captura i deté el temps. Aquest tipus de *wrappers* es relacionen, entre altres, amb els patrons *Decorador* (*Decorator*) i *Proxy*.

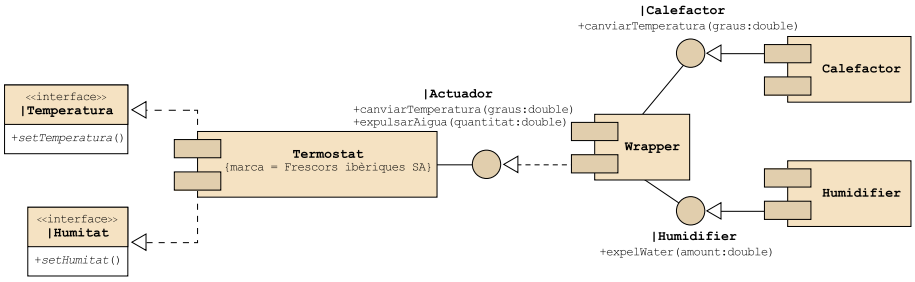



**4) *Wrappers* de mediació**, que coordinen diferents interpretacions d'una mateixa propietat del sistema.

Per exemple, es pot usar un *wrapper* d'aquest tipus per a gestionar la seguretat de l'accés a diversos components. Es relacionen amb els patrons *Mediador* (*Mediator*) i *Proxy*, entre altres.

### Composició

Una vegada que s'han construït els *wrappers* necessaris per a endollar uns components amb altres, els components es componen i integren per a servir la funcionalitat. La figura següent il·lustra com, finalment, el *Termòstat* es pot connectar amb els dos actuadors per mitjà del *wrapper* que es va dissenyar i implementar en el punt anterior.




Hi ha diversos enfocaments per a dur a terme la composició de components:

**1) Composició seqüencial**: els components es van connectant un a l'altre seqüencialment, i es creen per a cada connexió els *wrappers* necessaris.

**2) Composició jeràrquica**: els components es connecten un amb l'altre, de manera jeràrquicament ascendent o descendent.

**3) Composició additiva**: les interfícies de dos components s'uneixen per a crear un "supercomponent" que ofereixi les funcionalitats de tots dos, i que requereixi també les funcionalitats de tots dos.

### Actualització

Com tot sistema de programari, els sistemes basats en component també arriben, en algun moment del seu cicle de vida, a la fase de manteniment, que pot ser correctiu (correcció d'errors), perfectiu (addició de noves funcionalitats), preventiu (millora de propietats sense alterar-ne la funcionalitat) o adaptatiu (adaptació del sistema a canvis d'entorn). La modificació del sistema pot implicar la necessitat de substituir un component per un altre; però, a més, els fabricants produeixen també versions noves dels seus components en funció de les seves correccions pròpies, millores o necessitats competitives del mercat. Davant nous lliuraments (*releases*, en anglès), el desenvolupador ha de decidir si se substitueix o no el nou component per la nova versió.

**Vegeu també**

Vegeu l'assignatura *Projecte de desenvolupament de programari* per a més informació sobre manteniment de programari.

La substitució d'un component hauria de ser tan senzilla com el reemplaçament d'una peça en un puzle per una altra de manera exactament igual. De vegades, no obstant això, cal analitzar la possible necessitat de reescriure *wrappers*, i també fer proves unitàries del component candidat. El fet que el compo-

nent sigui accessible únicament per mitjà de les seves interfícies permet abaratir els costos de les proves: si se superen les proves d'unitat, probablement no és necessari fer proves d'integració ni proves de sistema.

### 3.7.2. Components i classes

És possible que una classe ofereixi, per mitjà d'una interfície, una funcionalitat ben determinada i que pugui necessitar una classe tercera per a executar algun servei. Vist així, es podria considerar que aquesta classe és un component. No obstant això, no totes les classes són components:

- Un component és una entitat directament instal·lable (*deployable*, en anglès).
- Un component no defineix un tipus.
- La implementació d'un component és una caixa negra.
- Un component és una entitat estandarditzada.

Si se segueixen els principis d'alta cohesió i baix acoblament, les classes són mòduls de mida més o menys petita i tindran unes poques dependències amb altres classes. Per contra, els components són més grans, perquè inclouen en l'interior la implementació de diverses classes; són autocontinguts i independents; en general, utilitzen com a paràmetres dels seus serveis solament tipus bàsics (a fi que siguin, precisament, el més independents possible del context i de l'entorn); es poden connectar ("endollar") a altres components per a construir una aplicació; implementen interfícies ben definides; són reemplaçables.

### 3.7.3. Desenvolupament de components

La probabilitat de reutilitzar un component és més gran com més relacionat estigui amb un domini d'abstracció estable: això és, com més representi un objecte de negoci (*business object*). En un enfocament "oportunista", es detecta la possibilitat de reutilitzar un component després d'haver-lo implementat per a un propòsit específic. Per a fer-lo efectivament reutilitzable en altres contextos, el component ha de ser modificat i generalitzat:

**Vegeu també**

Parlarem amb més profunditat el tema dels dominis en tractar les línies de producte programari, els DSL i la programació generativa a l'apartat 4 d'aquest material.

- Eliminant les crides a mètodes específics de l'aplicació per a la qual va ser construït.
- Canviant els noms dels serveis oferts perquè siguin més generals.
- Afegint nous serveis perquè cobreixi un espectre tant més general com més particular.
- Fent que cada servei llanci un conjunt ben clar d'excepcions (és a dir, evitant el llançament d'una *Exception* genèrica, que impedeix a l'usuari del component conèixer amb precisió la situació anòmala que s'ha produït).

- Afegint una interfície de configuració que permeti adaptar el component al seu context d'execució.

- Integrant en el component altres components, de manera que se'n redueixi la dependència i pugui ser integrat de manera més fàcil en entorns, contextos i aplicacions diferents.

### 3.8. *Frameworks*

Tal com un component ens brinda, per la seva construcció mateixa, una o més funcionalitats empaquetades a punt per a ser utilitzades, un *framework* ofereix un conjunt de classes (algunes d'abstractes i altres de concretes) i interfícies que ajuda a la resolució de problemes freqüents i que, a més, imposarà (o, en tot cas, guiarà) al desenvolupador un cert plantejament del disseny arquitectònic del sistema. Larman ho explica dient que:

> "Fins i tot a risc de simplificar en excés, un *framework* és un conjunt extensible d'objectes per a funcions relacionades [...]. El senyal de qualitat d'un *framework* és que proporciona una implementació per a les funcions bàsiques i invariables, i inclou un mecanisme que permet al desenvolupador connectar les funcions que varien, o estendre les funcions."

**Consulta recomanada**

**C. Larman** (2002). *UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Prentice Hall.

Els *frameworks*, en efecte, estan formats normalment per un conjunt interrelacionat de classes fàcilment extensibles (de fet, les seves classes estan pensades per a ser esteses: això és, especialitzades). Algunes de les característiques dels *frameworks* que esmenta Larman són:

- Disposen d'un conjunt cohesiu d'interfícies i classes que col·laboren per a proporcionar els serveis de la part central i invariable d'un subsistema lògic.

- Contenen classes concretes i abstractes que defineixen les interfícies a les quals s'ajusten, interaccions d'objectes en les quals participen, i altres d'invariants.

- Normalment requereixen que l'usuari del *framework* defineixi subclasses de les classes que s'hi inclouen per a utilitzar, adaptar i estendre els serveis del *framework*. Aquestes subclasses rebran missatges des de les classes predefinides del *framework* que, normalment, es manegen implementant mètodes abstractes heretats de les classes abstractes del *framework*, i que l'usuari haurà hagut de redefinir.

### 3.8.1. *Framework* per a la implementació d'interfícies d'usuari

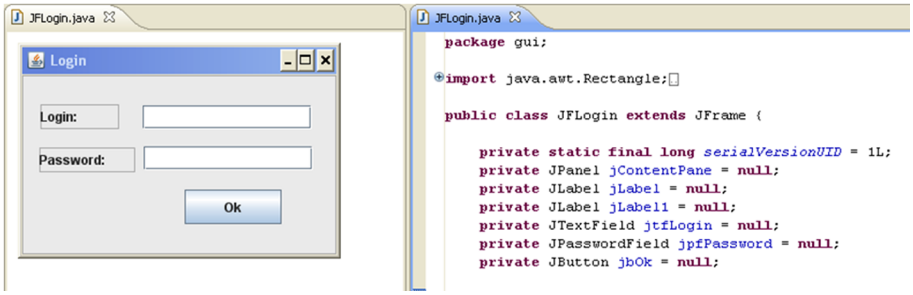
**Web recomanat**

A http://www.javaswing.org/ hi ha una llista completa dels components visuals d'aquest *framework*.

Swing és un *framework* de Java que inclou un conjunt molt ampli de components visuals (botons, caixes de text, etc.) per a crear interfícies gràfiques d'usuari per a aplicacions Java, com ara *JMenu*, *JButton*, *JTextarea*, etc.

La figura següent mostra un moment del disseny d'una finestra per a identificar-se amb un nom d'usuari i una contrasenya: mentre l'usuari va dissenyant-la amb l'assistent visual del seu entorn de desenvolupament, aquest va afegint el codi corresponent a l'aspecte de la finestra, que després el programador haurà de completar. Noteu com, en el codi font, la classe que s'ha creat (*JFLogin*) és una especialització de la classe *JFrame*, que és una de les proporcionades per Swing. Aquest és un exemple clar de la tercera característica dels *frameworks* que indicava Larman i que hem esmentat més amunt:

> "Normalment requereixen que l'usuari del *framework* defineixi subclasses de les classes que s'hi inclouen per a utilitzar, adaptar i estendre els serveis del *framework*."




**Vegeu també**

Hem vist el patró MVC en l'apartat "Patrons de disseny".

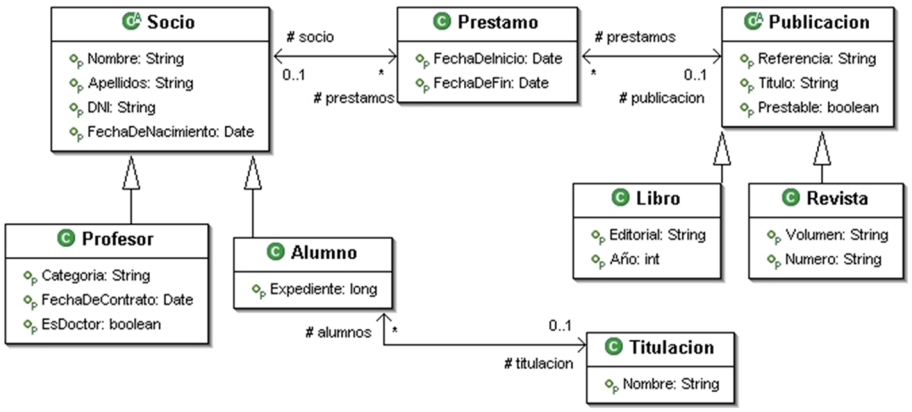
Swing permet, a més, adequar-se al patró *model-vista-controlador* (MVC), ja que cada component visual està empaquetat en una classe i pot, a més, associar-se a un tipus de model específic. En Swing s'inclouen també models adequats per a cada tipus de component. A les taules (*JTable*), per exemple, es pot associar un objecte de tipus *TableModel* que contingui les dades mostrades en la taula. Si un altre objecte actualitza el *TableModel* associat, l'objecte corresponent visual (la *JTable*) canvia automàticament.

### 3.8.2. *Frameworks* de persistència per a "mapatge" objecte-relacional

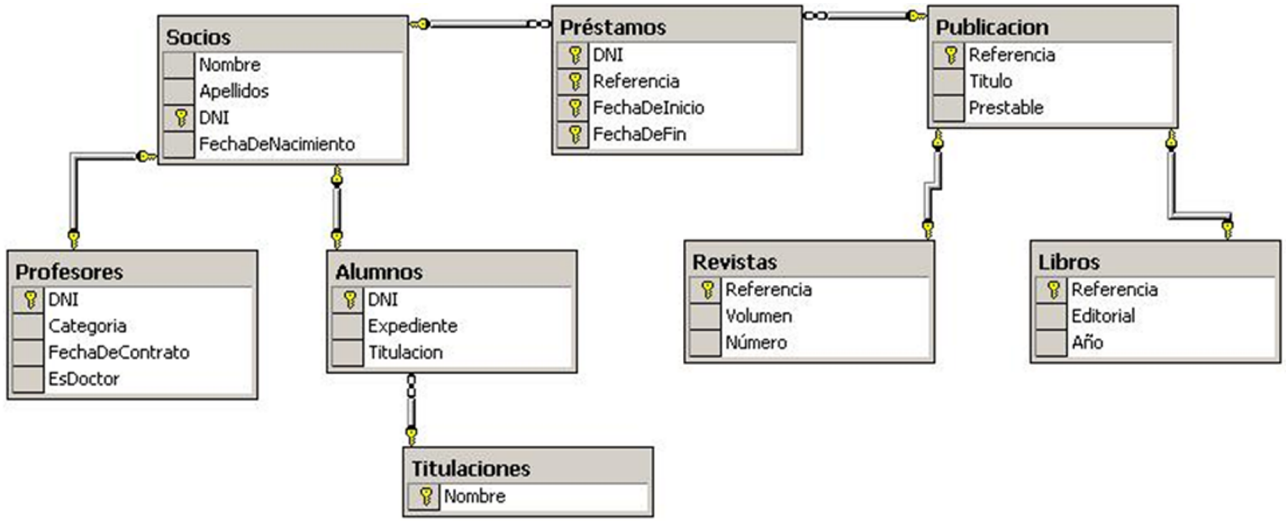
La gestió de la persistència d'objectes és un problema molt recurrent en el desenvolupament d'aplicacions. Els "món d'objectes" i el "món de taules" són molt diferents, per la qual cosa es necessiten bones polítiques de mapatge entre un i l'altre.

En primer lloc, és convenient mantenir dissenys equivalents entre l'estructura de classes i l'estructura de taules, que es pot aconseguir mitjançant l'aplicació de patrons de transformació, com el patró "Una classe, una taula": de cada

classe persistent es construeix una taula; associacions i agregacions es transformen en relacions de clau externa; l'herència es transforma en una relació entre dues taules relacionades per les seves claus primàries.



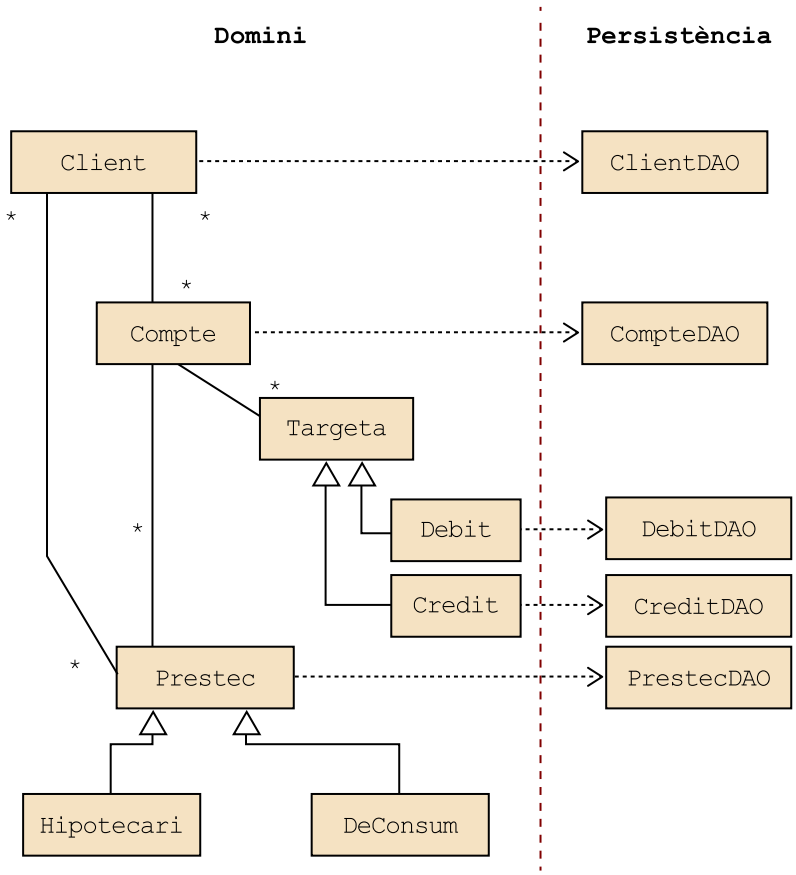

L'aplicació del patró "Una classe una taula" al diagrama de classes anterior, que representa una biblioteca, pot donar lloc al diagrama relacional següent:




En segon lloc, s'han d'assignar les responsabilitats de persistència de manera que es compleixin els principis bàsics de disseny de programari (especialment el manteniment de l'alta cohesió i el baix acoblament): quan s'ha presentat el patró "fabricació Pura", s'ha explicat que s'utilitza amb freqüència per a delegar a altres classes la persistència de classes de domini; com es mostra en la figura següent, determinades classes de domini tenen associada una fabricació pura que s'encarrega de gestionar-ne la persistència. La implementació de les operacions de persistència en les classes DAO (*data access objects*: objectes d'accés a dades) ha de ser coherent amb el disseny de la base de dades.

**Vegeu també**

Vegeu les classes DAO en l'assignatura *Enginyeria de programari de components i sistemes distribuïts* del grau d'Enginyeria Informàtica.




Hi ha diversos *frameworks* que ajuden a resoldre el problema de la gestió de la persistència (es coneixen com a *frameworks ORM*: *object-relational mappling*) que proposen solucions semblants a la mostrada en la figura anterior. En general, aquests *frameworks* proporcionen:

- Una API per a gestionar les operacions CRUD.
- Un llenguatge per a especificar les consultes que es refereixen a les classes i a les seves propietats.
- Una manera de definir i especificar les metadades del mapatge.
- Una tècnica per a la implementació del mapatge que permeti la interacció amb objectes transaccionals amb suport per a diverses funcions.

Alguns d'aquests *frameworks* són: Hibernate i NHibernate (per a Java i .NET, respectivament), Torque i Object-Relational Bridge.

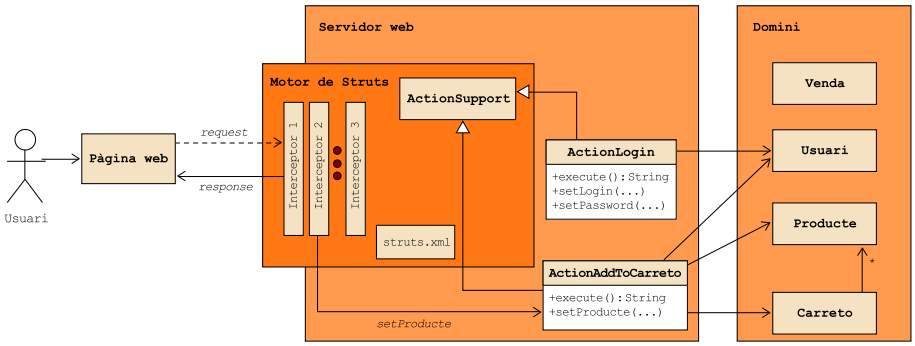
### 3.8.3. *Framework* per al desenvolupament d'aplicacions web

En desenvolupament web tradicional, tota la informació que envien els usuaris des dels seus navegadors es recull en el servidor en forma de cadenes de caràcters (*strings*). Els programadors web recullen aquests paràmetres, els converteixen al tipus de dada que correspongui[22] en comproven la validesa segons les característiques del sistema i, després, criden la lògica de negoci.

[22] Es pot rebre en el servidor una cadena com "03-12-2011", que ha de ser traduïda a un objecte de tipus *Date*.

Struts2 és un *framework* de Java que inclou un conjunt molt ampli de classes per a desenvolupar aplicacions web, que evita als desenvolupadors l'escriptura i reescriptura de codi molt semblant i que, a més, imposa una arquitectura molt sòlida per al disseny de les aplicacions.

Aquest *framework* requereix l'extensió del servidor web amb un motor auxiliar d'execució que recull les peticions que arriben des dels clients. Els paràmetres es reben, com sempre, en forma de cadenes de caràcters, però els capturen una sèrie d'interceptors que els poden processar de maneres diferents. La figura següent il·lustra un fragment del disseny d'una tenda virtual: l'usuari interactua amb una sèrie de pàgines web que envien paràmetres al servidor; aquests són capturats per una pila d'interceptors que poden, per exemple, validar-ne el format.




Si els paràmetres enviats no són correctes, s'interromp l'execució de la pila d'interceptors i es retorna l'error (o la resposta que correspongui) a l'usuari. Alguns dels interceptors poden cridar accions concretes (especialitzacions de la classe *ActionSupport* del *framework* Struts2, que és la classe fonamental): l'*Interceptor* 2 de la figura, per exemple, executa el mètode *setProducte* sobre l'acció *ActionAddToCarreto* que, potser, afegeix el producte passat com a paràmetre al carretó de la compra.

**Web recomanat**

El *framework* i la documentació de Struts2 està disponible a http://struts.apache.org/

Cada acció es correspon, molt aproximadament, amb un requisit funcional o cas d'ús del sistema, per la qual cosa aquest *framework* resulta molt convenient per a migrar aplicacions d'escriptori envers entorns web, ja que se separa completament la lògica de negoci de la interfície d'usuari, i se n'afavoreix la reutilització. Quan s'ha executat l'últim interceptor, el motor d'execució crida el mètode *execute():String* de l'acció, que, normalment, consta d'un bloc *try* se-

guit d'un o més *catch*: el bloc *try* es correspon amb l'escenari normal del cas d'ús representat per l'acció, mentre que cada *catch* es correspon amb un escenari alternatiu.

La cadena retornada per *execute* la interpreta el motor d'execució a partir dels valors declarats en el fitxer *struts.xml*. En aquest es troba una llista de totes les accions al costat dels possibles valors de tipus `string` que els seus mètodes *execute* poden retornar: cada valor possible[23] té associada una pàgina web diferent, que es correspon amb la resposta que es retorna a l'usuari.

[23] Alguns són predefinits, com *success*, *error* o *input*, mentre que altres poden estar definits en l'aplicació mateixa, com NoExisteixElProducte.

A més de reutilitzar la classe *ActionSupport*, els desenvolupadors poden crear nous interceptors, alterar l'ordre de les piles d'interceptors per a modificar el tractament per defecte o crear piles noves. A més, cada acció disposa d'un context d'execució (*ActionContext*) a partir del qual es poden recuperar els objectes habituals (la *session*, el *request*, el *response*, etc.) que, en un altre cas, es manegen de manera transparent. Finalment, cada acció té, en la seva *ActionContext*, una *ValueStack* (pila de valors) en la qual es col·loquen els objectes manipulats per l'acció (en la figura, en la *ValueStack* de l'*ActionLogin* hi haurà un objecte de tipus *Usuari*; en la d'*ActionAddToCarreto* hi ha un *Usuari*, un *Producte* i un *Carreto*). Les accions poden compartir els objectes que emmagatzemen en les *value stacks* de les seves *action contexts* per mitjà d'un mapa d'objectes.

El fet d'utilitzar el *framework* Struts2 ja implica la reutilització de les classes que ell mateix inclou. A més, com hem dit, aporta una sèrie d'avantatges molt importants respecte de la reutilització de la lògica de domini d'aplicacions desenvolupades inicialment per a escriptori: si una aplicació d'escriptori està ben dissenyada (quant a bona separació de la lògica de domini respecte de la de presentació), es pot crear una acció per cada cas d'ús. L'acció, en aquest cas, es correspondria aproximadament amb un patró *Controlador* (de cas d'ús) dels proposats per Larman.

**Vegeu també**

Vegeu el patró *Controlador* en l'apartat "Patrons de disseny".

## 3.9. Serveis

Un servei és una funcionalitat que un proveïdor de serveis ofereix als seus clients de manera remota. Essencialment, el servei pot ser vist com un component, ja que el client o usuari del servei (en el nostre cas, un enginyer de programari que en farà ús en el seu sistema) només coneix les operacions que ofereix per mitjà de la seva interfície.

Els serveis, en enginyeria del programari, es poden entendre des de dos punts de vista: com a desenvolupadors del servei i com a usuaris. En aquesta secció presentem la tecnologia de serveis mostrant el cas particular dels serveis web, i presentant-la des d'aquests dos punts de vista.

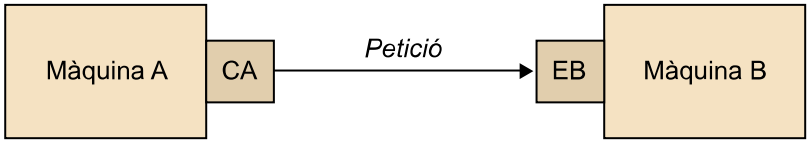
### 3.9.1. Introducció als serveis web

**Vegeu també**

Vegeu els serveis web en l'assignatura *Enginyeria de programari de components i sistemes distribuïts* del grau d'Enginyeria Informàtica.

Els serveis web s'executen sobre un tipus de programari intermediari mitjançant el qual es poden comunicar aplicacions remotes. En essència, funciona com qualsevol altre tipus de programari intermediari (RMI, CORBA...), però amb la diferència important que els missatges que s'envien i es reben s'adhereixen a un protocol estandarditzat anomenat *SOAP* (*simple object access protocol*). Tant la crida al servei remot com la resposta es codifiquen en SOAP i es transporten, normalment, mitjançant HTTP.

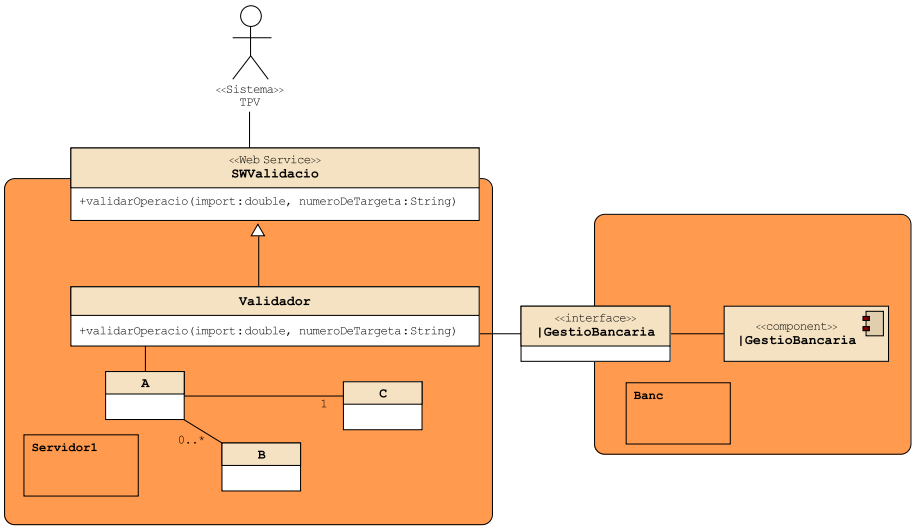
Tots els protocols de missatgeria es poden veure com en la figura següent, que mostra dues màquines qualssevol que es comuniquen: quan A vol enviar un missatge a B, prepara el missatge en un format equivalent al que espera B, i l'envia. En la figura, CA representa l'element de A que codifica el missatge, mentre que EB representa l'element de B per al qual aquest escolta. Aquest model és vàlid per a RMI, CORBA, serveis web o fins i tot per a un protocol de missatgeria totalment de propietat i que es podria implementar mitjançant sòcols: un requisit necessari perquè les dues màquines s'entenguin és la definició del format dels missatges que es volen enviar des de A fins a B i des de B fins a A, i després implementar el mecanisme de codificació i descodificació.




A fi d'estandarditzar la comunicació entre màquines remotes i, alhora, fer un pas important quant a l'oferta de serveis remots i les arquitectures orientades a serveis, s'ha proposat SOAP, un protocol de missatgeria basat en XML: així, la crida a una operació oferta per un servidor consisteix realment en la transmissió d'un missatge SOAP, el resultat retornat és un altre missatge SOAP, etc. D'aquesta manera, el client pot estar construït en Java i el servidor en .NET però tots dos aconseguiran comunicar-se gràcies a l'estructura dels missatges que intercanvien.

### 3.9.2. Desenvolupament de serveis web

Els serveis web s'ofereixen mitjançant interfícies remotes que es publiquen en algun servidor, i que s'hi troben implementades. En la figura següent s'il·lustra, a molt alt nivell, un *Servidor* 1 que ofereix un servei de validació d'operacions amb targetes de crèdit a terminals de punt de venda (TPV). Com es veu, la lògica del *Validador* es troba oculta a l'usuari del servei, que només coneix l'operació oferta per mitjà del *Web Service*. La implementació de l'operació, en el *Servidor* 1, utilitza diverses classes i, en l'exemple, un component extern (tal vegada un EJB o un DCOM) que es troba en un sistema remot.




En general, el desenvolupament d'un servei web segueix aquests passos:

**1)** Escriptura d'una classe que implementi les operacions que s'oferiran. En la figura anterior, seria el cas de *Validador*, que actua de façana.

**2)** Generació del servei web. Tots els entorns de desenvolupament actuals inclouen la possibilitat d'oferir les operacions contingudes en una classe mitjançant un servei web.

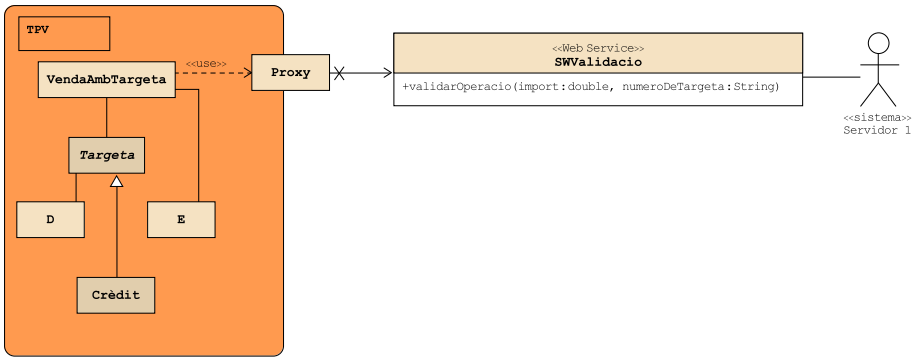
**3)** Desplegament i publicació del servei web en un servidor.

**4)** Alhora que es desplega el servei, es genera automàticament un document WSDL (*web services description language*) que els usuaris del servei (és a dir, els desenvolupadors que implementen els sistemes clients) necessiten per a conèixer les operacions ofertes. La figura següent mostra un fragment del document WSDL generat per al servei *WSValidador*: noteu que s'inclou una etiqueta (*tag*) de tipus *element* per a *validarOperacio* que pren dos paràmetres (*import* i *numeroDeTargeta*).

```
▼<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:ns1="http://org.apache.
 xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl" xmlns:http="http://schemas.xmlsoap.org
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" xmlns:soap="http://schemas.xmlsoap.org/wsd
 xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" targetNamespace="http://dominio">
   <wsdl:documentation>WSValidador</wsdl:documentation>
 ▼<wsdl:types>
   ▼<xs:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace
     ▼<xs:element name="validarOperacio">
       ▼<xs:complexType>
         ▼<xs:sequence>
            <xs:element minOccurs="0" name="import" type="xs:double"/>
            <xs:element minOccurs="0" name="numeroDeTargeta" nillable="true" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
     ▼<xs:element name="validarOperacioResponse">
       ▼<xs:complexType>
         ▼<xs:sequence>
            <xs:element minOccurs="0" name="return" type="xs:boolean"/>
          </xs:sequence>
```

### 3.9.3. Desenvolupament de clients

Una vegada que el document WSDL està publicat, l'usuari del servei el necessita recuperar per a crear, d'alguna manera, un mecanisme d'accés a aquest que el permeti utilitzar-lo. La figura següent mostra el mateix sistema, però ara des del punt de vista del desenvolupador del sistema TPV: a més de les seves classes de domini, finestres, etc., el desenvolupador afegirà al seu sistema un servidor intermediari que representi, per al sistema client, el punt d'accés al servei remot.



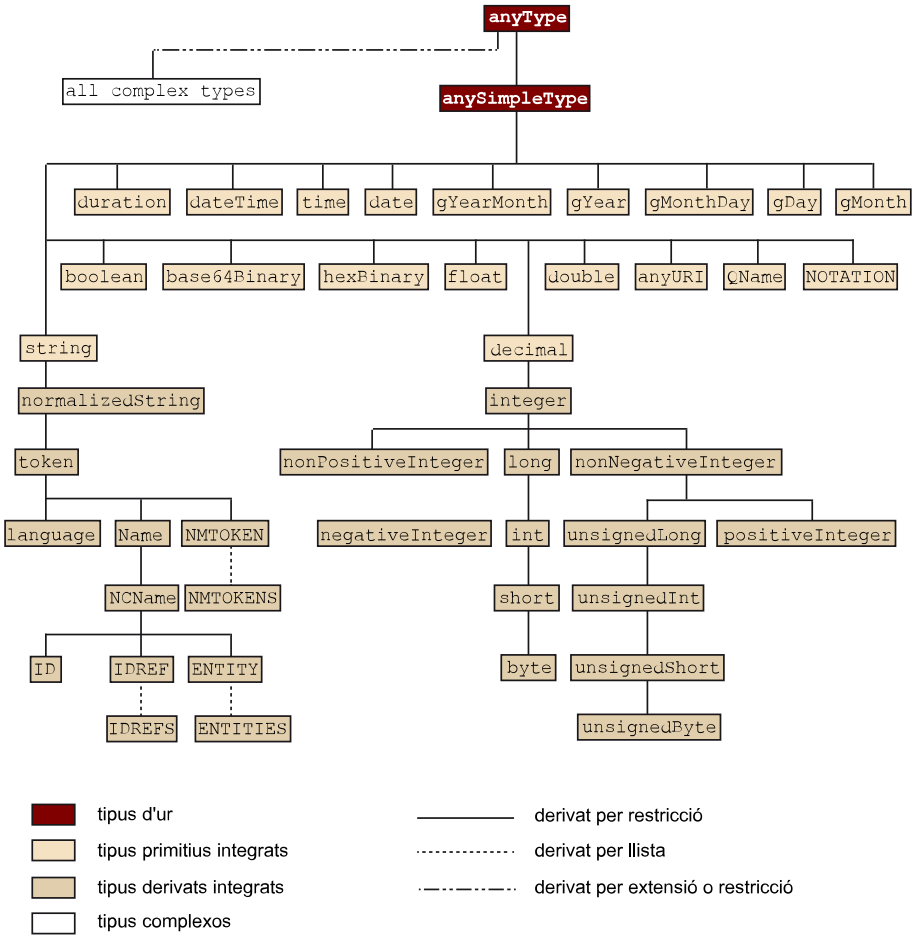

Igual que els entorns de desenvolupament actuals, com s'ha dit, incorporen assistents per publicar les operacions contingudes en classes en forma de serveis web, també inclouen assistents que analitzen el codi dels documents WSDL i generen servidors intermediaris de connexió de manera automàtica.

Una dels avantatges que s'ha comentat dels serveis web és la interoperabilitat entre plataformes: podem accedir al servei web *WSValidador*, que està implementat en Java, mitjançant un client escrit en el C# de .NET.

### 3.9.4. Tipus de dades

Els intercanvis d'informació entre el client i el servidor es duen a terme mitjançant missatges en format SOAP. Cada paràmetre enviat o cada resultat retornat estan definits segons un tipus de dada. Els tipus de dades que SOAP suporta són els recollits en el document *XML Schema Part 2: Datatypes Second Edition* que es mostren en la figura següent, agafada d'aquest document: com es veu, es preveuen els tipus primitius existents en la majoria dels llenguatges de programació (enters, reals, cadenes de caràcters, booleans, etc.).

Els tipus complexos (com *Targeta*, *ClientBancari*, etc.) es poden serialitzar per a ser codificats i transmesos, i apareixerien recollits, en la figura, en la categoria *all complex types*. Els assistents de generació i inclusió de serveis web breguen amb els tipus de dades no estàndard, amb la qual cosa es poden utilitzar sense problemes en la gran majoria de vegades.


anyType
all complex types
anySimpleType
duration
dateTime
time
date
gYearMonth
gYear
gMonthDay
gDay
gMonth
boolean
base64Binary
hexBinary
float
double
anyURI
QName
NOTATION
string
decimal
normalizedString
integer
token
nonPositiveInteger
long
nonNegativeInteger
language
Name
NMTOKEN
negativeInteger
int
unsignedLong
positiveInteger
NCName
NMTOKENS
short
unsignedInt
ID
IDREF
ENTITY
byte
unsignedShort
IDREFS
ENTITIES
unsignedByte
tipus d'ur
tipus primitius integrats
tipus derivats integrats
tipus complexos
derivat per restricció
derivat per llista
derivat per extensió o restricció

# 4. Reutilització a gran escala: solucions metodològiques de reutilització

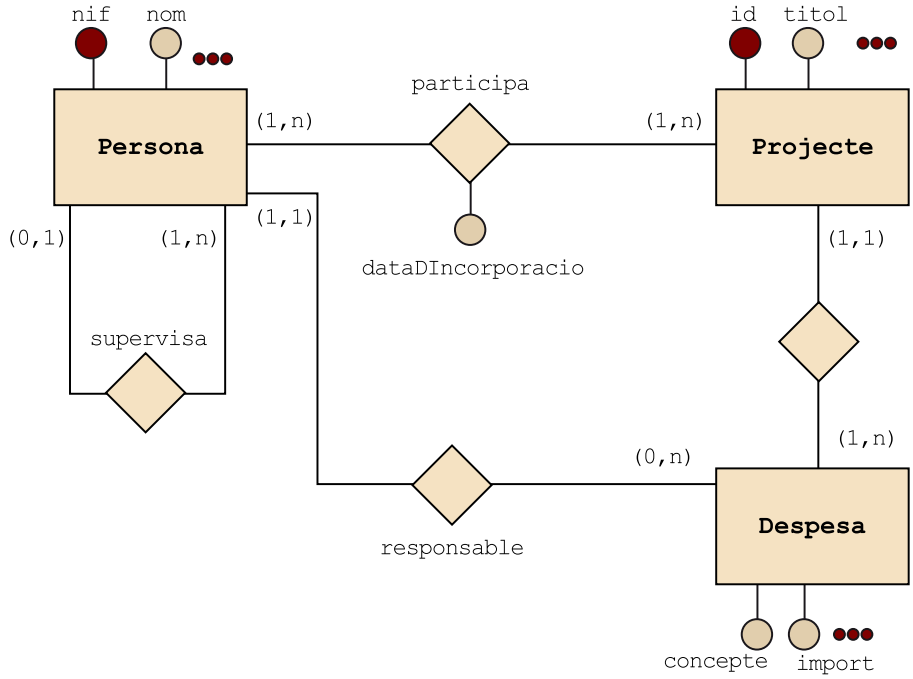
## 4.1. Introducció

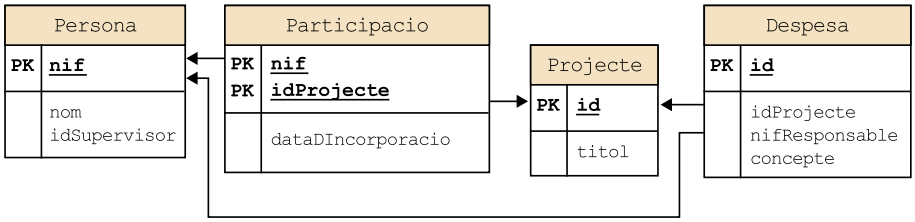
En l'apartat anterior s'han presentat algunes solucions tècniques per a afavorir la reutilització de programari, i hem posat l'atenció en alguns artefactes programari concrets, com components, biblioteques o serveis web. Però la fabricació i la reutilització de programari van més enllà del simple emmagatzematge d'elements reutilitzables en repositoris als quals un pugui accedir per agafar el que necessiti: per a millorar-ne i fomentar-ne la reutilització (per a no fer tan sols una reutilització oportunista de components predesenvolupats), és necessari desenvolupar el programari utilitzant metodologies específiques que la considerin des del principi.

En aquesta secció es presenten algunes metodologies de desenvolupament en les quals la reutilització té un paper essencial. Parlarem d'enginyeria del programari dirigida per models, línies de producte de programari, llenguatges específics de domini, programació generativa i, més breument, de fàbriques de programari.

## 4.2. Enginyeria del programari dirigida per models

En desenvolupament tradicional de bases de dades, els requisits d'emmagatzematge que es van capturant es representen utilitzant un model conceptual que, en molts casos, té la forma d'un diagrama entitat-interrelació. En una etapa posterior, el diagrama conceptual obtingut es transforma d'acord amb el model lògic que correspongui i, en general, s'obté un diagrama relacional. Aquest, finalment, s'enriqueix amb les anotacions que corresponguin per al sistema gestor de base de dades en el qual s'hagin de gestionar les dades, i se n'obté un model físic:




Hi ha moltes eines que fan aquest tipus de traduccions successives d'un tipus de diagrama a un altre i que, al final, implementen la base de dades sobre el gestor que s'hagi seleccionat. Aquest últim implica, en primer lloc, la generació del codi SQL que correspongui a la definició de les dades; i, en segon lloc, l'execució d'aquest sobre el gestor.

L'esquema conceptual és un artefacte de programari tan genèric que, en la pràctica, és útil per a construir un esquema vàlid per a qualsevol model lògic de base de dades: relacional, orientat a objectes, objecte-relacional o, fins i tot, per a models lògics més antics, com el jeràrquic. Per a la modelització conceptual, per tant, no es pren en consideració la plataforma final d'execució de la base de dades; igualment, el model lògic sí que depèn del suport tecnològic final, però no dels detalls concrets del gestor de base de dades.

**Exemple**

En l'SQL Server, per exemple, els valors lògics es poden representar amb columnes de tipus bit, mentre que en l'Oracle han de ser de tipus numèric.

La idea de l'enginyeria del programari dirigida per models (MDE: *model-driven engineering*) és molt semblant: l'ideal d'aquest paradigma es podria resumir a "Dibuixar un sistema i passar directament a executar-lo". Per a això s'han de representar tant l'estructura del sistema com el seu comportament. A més, i a fi de facilitar la portabilitat dels sistemes desenvolupats, es comença dissenyant models independents de la plataforma final d'execució que, en un pas posterior, es transformen a models dependents:

**Vegeu també**

Vegeu el mòdul sobre "Desenvolupament de programari dirigit per models" d'aquesta mateixa assignatura.

- L'enginyer de programari construeix un PIM (*platform-independent model*) que descriu el sistema sense considerar els detalls finals d'implementació.

- El model independent es tradueix, mitjançant una transformació, en un PSM (*platform-specific model*), en el qual sí que es tenen en compte les característiques concretes de la plataforma final d'execució.

- Finalment, una última etapa de generació de codi transforma el PSM a codi compilable i executable.

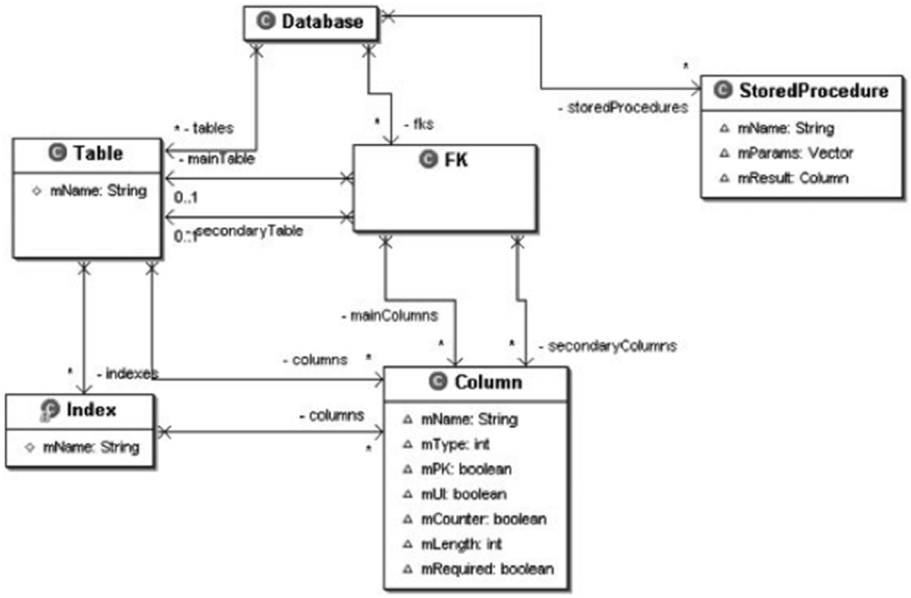
### 4.2.1. Diferents aproximacions: metamodels propis i metamodels estàndard

Igual que el llenguatge traduït per un compilador requereix adherir-se a una sintaxi específica per a poder ser entès, l'automatització de la transformació d'un model des d'un tipus a un altre necessita també que el model estigui representat amb alguna sintaxi específica. Aquesta és la funció principal dels metamodels, que no són sinó models que s'utilitzen per a representar altres models.

**Consulta recomanada**

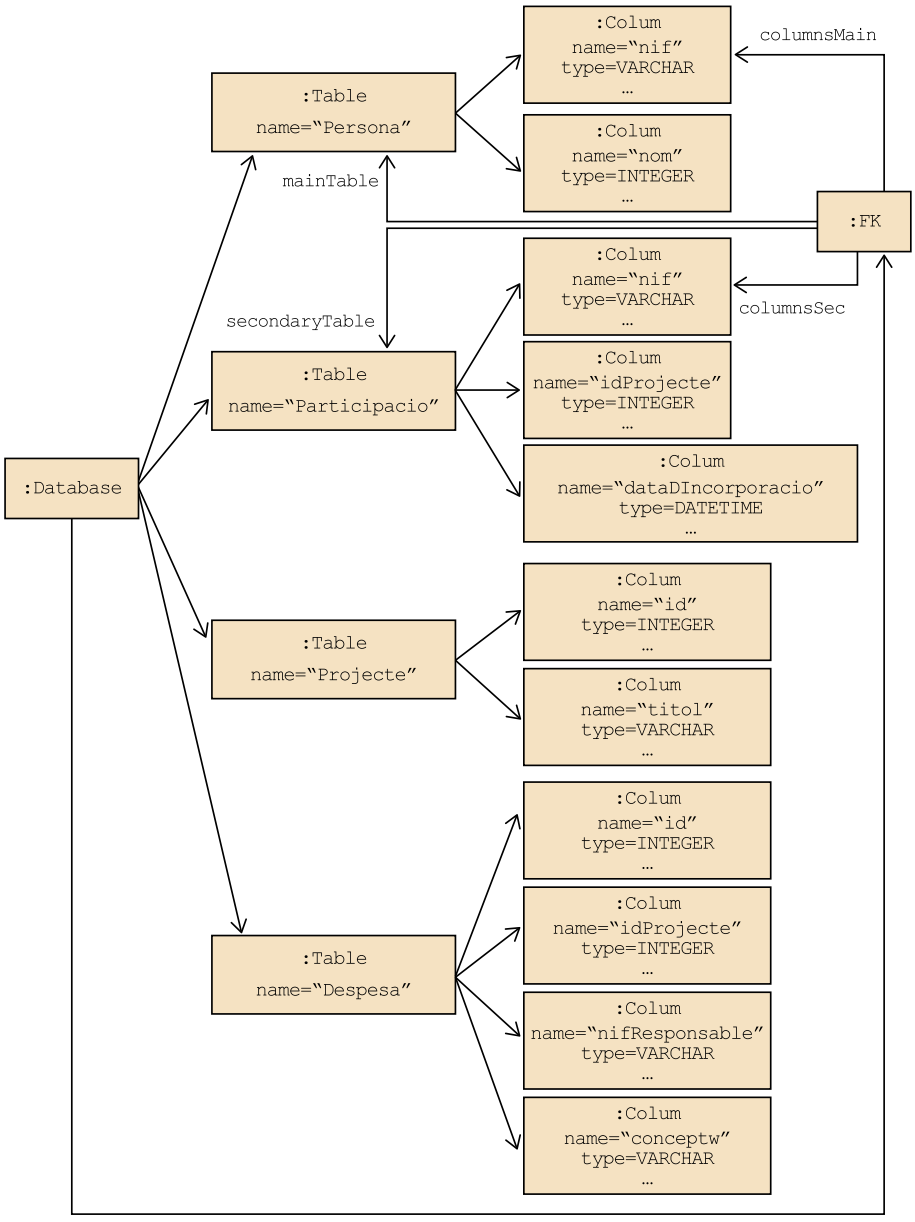
**M. Polo; I. García-Rodríguez; M. Piattini** (2007). "An MDA-based approach for database reengineering". *Journal of Software Maintenance & Evolution: Research and Practice* (vol. 19, núm. 6, pàg. 383-417).

La figura següent (presa de Polo i altres autors, 2007) mostra un dels metamodels utilitzats per una eina que, a partir d'una base de dades relacional implementada en Access, Oracle o SQL Server, és capaç de generar diferents tipus d'aplicacions per a gestionar la informació emmagatzemada en aquesta base de dades. L'eina llegeix la informació de taules, columnes i procediments emmagatzemats i construeix una instància del metamodel mostrat en la figura:

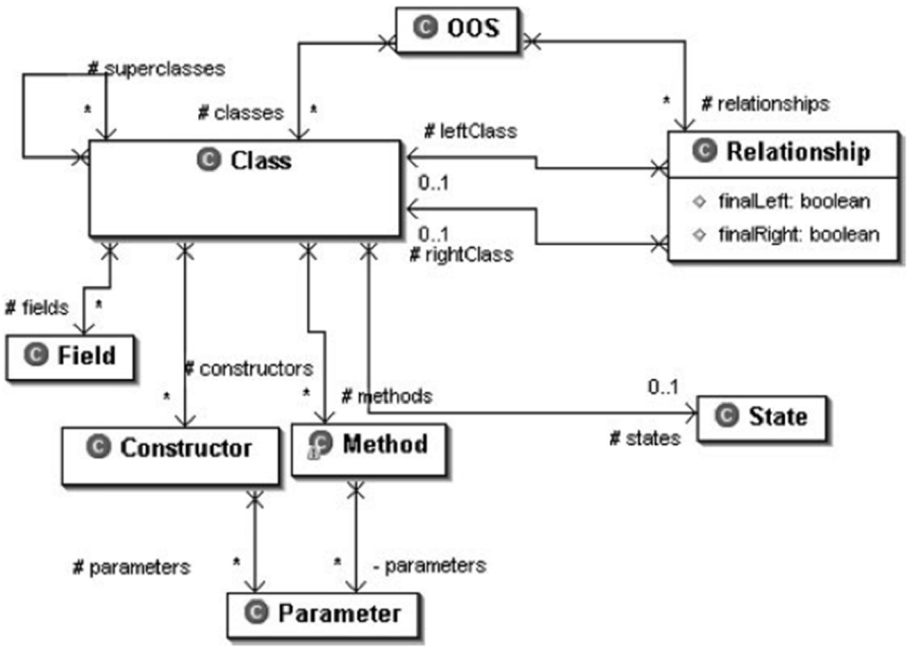



Així, a partir de la base de dades de persones, projectes i pressupostos que es mostrava a dalt, s'obtindria un conjunt d'instàncies de *Database*, *Table*, *Index*, *Column*, *FK* i *StoredProcedure*. En la figura següent es mostren part d'aquests ob-

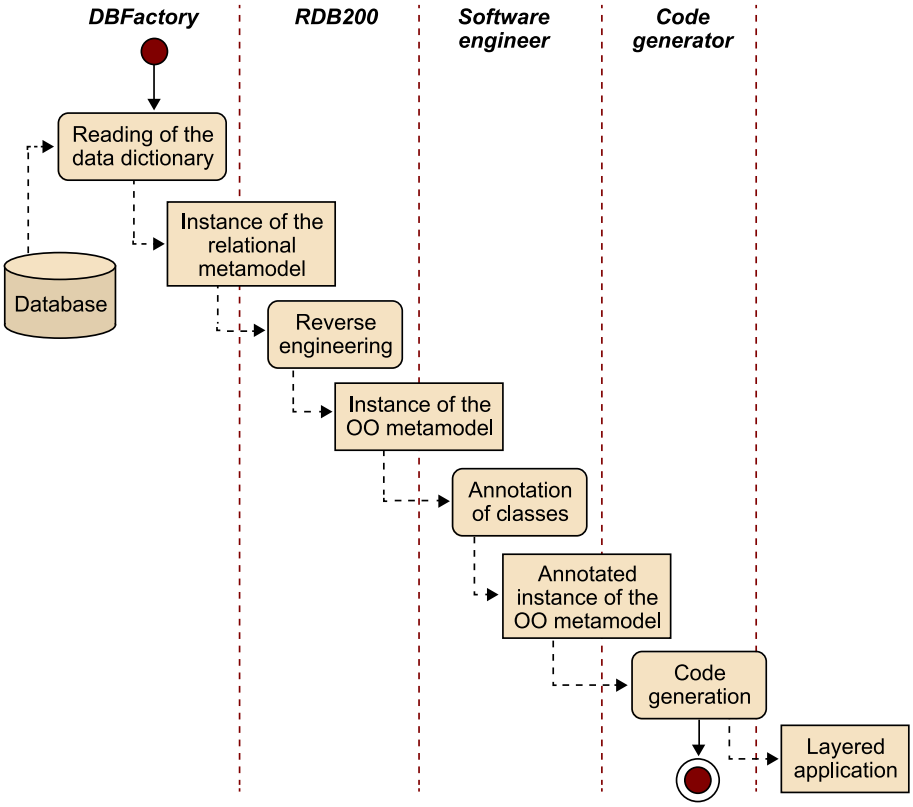
jectes: la instància de *Database* coneix quatre instàncies de *Table* i un objecte de tipus *FK* (en coneix realment més, però no s'han representat per no sobrecarregar el diagrama); cada objecte de tipus *Table* coneix diverses instàncies de tipus *Column*; la *foreign key* (objecte *FK*) relaciona les instàncies de tipus *Table*, els camps *name* dels quals són *Persona* i *Participació*, mitjançant les columnes apuntades per *columnsMain* i *columnsSec*.




Als models que es corresponen amb instàncies de metamodels ben definits els podem aplicar algorismes de transformació per fer "transformacions entre models". L'eina esmentada transforma, mitjançant un procés d'enginyeria inversa, models conformes al metamodel de base de dades relacional en models conformes a un metamodel que representa l'estructura de classes d'un sistema orientat a objectes:




Als models orientats a objectes els podem aplicar diferents algorismes de generació de codi. El procés complet s'il·lustra en la figura següent presa també de Polo i altres autors (2007): un procés llegeix la informació de disseny del gestor de base de dades a partir del diccionari de dades, i s'obté una instància del metamodel relacional; a aquest se li aplica un procés d'enginyeria inversa per a transformar el model en una instància del metamodel d'objectes; l'usuari pot, llavors, anotar les classes amb màquines d'estat per a descriure'n el comportament de les instàncies; finalment, un o més algorismes de generació de codi produeixen una aplicació multicapa capaç de gestionar la base de dades amb un conjunt bàsic d'operacions (les operacions CRUD) més aquelles que l'enginyer de programari hagi anotat en les màquines d'estat.




Un desavantatge important de l'ús de metamodels propis (com els dos mostrats anteriorment) és la dificultat per a portar els models entre eines. En els últims anys, l'OMG (Object Management Group) ha publicat i millorat diversos documents que s'han convertit en estàndards *de facto* per a afavorir el desenvolupament de programari seguint un enfocament MDE:

- UML, per a representar models de sistemes. Els seus mecanismes d'extensió basats en perfils (mitjançant estereotips, restriccions i valors etiquetats) permeten, per exemple, aplicar certs mecanismes de transformació només als objectes que tenen un estereotip específic: a una classe estereotipada com *ORM Persistable* se li pot aplicar una transformació, d'una banda, per a generar una taula i, de l'altra, per a generar una classe de domini i una classe *fabricaciópura* auxiliar que li gestioni la persistència.

- MOF (*meta object facility*), que és un llenguatge de metamodel estructurat en quatre nivells: en el nivell més baix (M0) es troben les dades reals (les persones emmagatzemades en la base de dades amb el seu nom, cognoms, etc.); en el següent (M1), l'estructura de la base de dades que, per exemple, pot ser relacional; en M2 el metamodel per a representar bases de dades relacionals; en M3, finalment, es troba MOF, que representa l'estructura genèrica que ha de tenir qualsevol metamodel.
  La figura següent il·lustra aquesta relació entre nivells: en M2 es troben els metamodels per a representar processos de negoci (SPEM), sistemes programari orientats a objecte (UML) i magatzems de dades (CWM); tots aquests metamodels són conformes a MOF (nivell M3), que és a més conforme

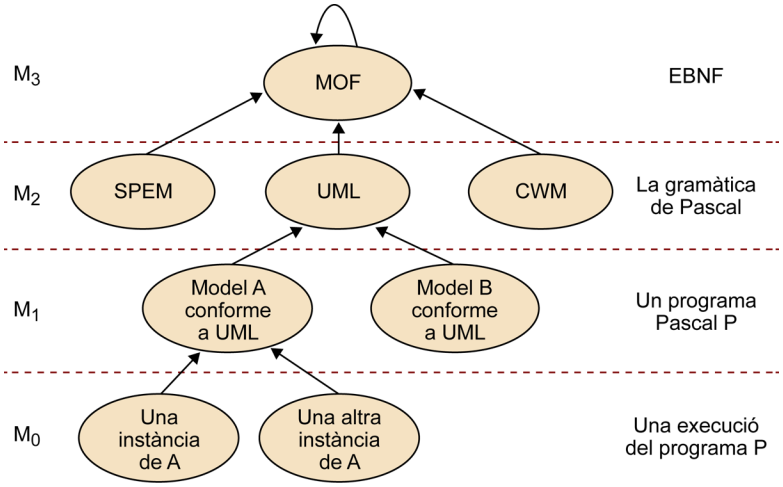
**Nota**

Les idees de la metamodelització i de la transformació de models tenen realment molts anys. Ja les eines primitives CASE disposaven dels seus metamodels i eines d'importació i exportació propis. Vegeu, per exemple, el llibre de **Piattini** et al. (1996). *Análisis y diseño detallado de Aplicaciones informáticas de gestión.* Madrid: Editorial Ra-Ma.

amb ell mateix; els models de sistemes programari orientats a objectes específics o el model de negoci d'una certa empresa es troben en M1 i són conformes al seu metamodel d'M2 corresponent; finalment, les dades, els objectes i els processos reals de cada execució del sistema o del procés de negoci es troben en M0.

**Nota**

EBNF: Extended Backus-Naur Form, defineix una manera formal de descriure la gramàtica d'un llenguatge de programació.




**Nota**

Han estat el desenvolupament d'UML, l'impuls de la investigació en universitats i instituts i les activitats d'OMG els que han portat als avenços actuals en MDE.

- OCL (*object constraint language*) és un llenguatge formal que, inicialment, va ser concebut per a anotar amb restriccions els models dissenyats amb UML (cosa que permet eliminar l'ambigüitat inherent als diagrames). En MDE, OCL permet llançar consultes sobre models i escriure precondicions i postcondicions per a les transformacions.

- QVT (*query-view-transformation*) és un llenguatge que permet programar transformacions entre models. Essencialment, una transformació QVT es pot veure com una funció el domini de la qual és el metamodel origen (per exemple, UML) i el recorregut de la qual és el metamodel destinació (per exemple, el metamodel per a representar bases de dades relacionals): així, a partir d'un model concret d'un sistema (és a dir, una instància del metamodel origen) s'obté un model concret (és a dir, una instància del metamodel destinació).

**Vegeu també**

En el mòdul sobre "Desenvolupament de programari dirigit per models" es tracten tots aquests conceptes amb més profunditat.

- XMI (*XMLmetadata interchange*) és l'estàndard d'OMG per a permetre l'intercanvi de models. Un model s'emmagatzema com un document XMI i pot ser processat per qualsevol altra eina.

- MOF2Text és l'estàndard per a generar text (codi font) a partir de models. El llenguatge QVT permet la transformació entre models, però fa difícil l'obtenció de codi a partir de models.

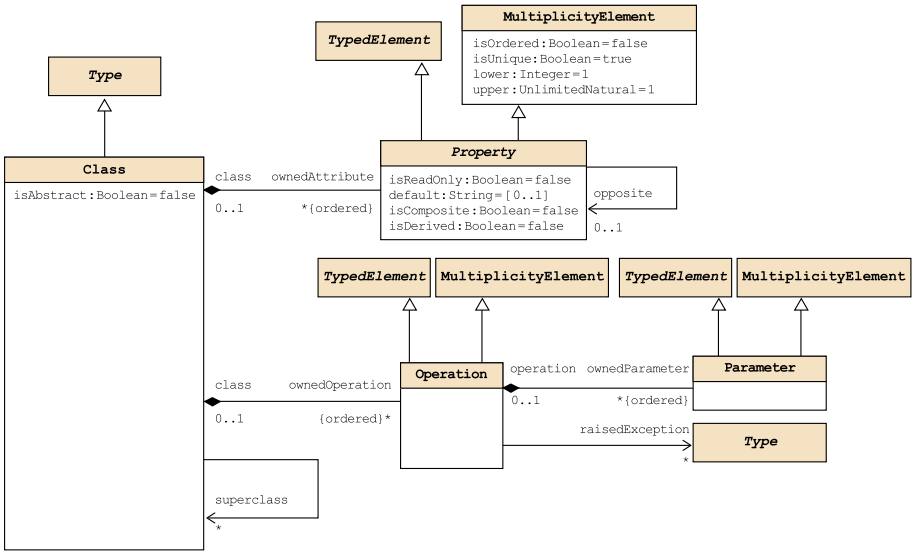
**Nota**

En l'edició de 2009 del conegut llibre de Pressman apareixen més de 200 sigles utilitzades en enginyeria de programari. No obstant això, no s'hi inclouen algunes de les sigles que hem vist aquí.

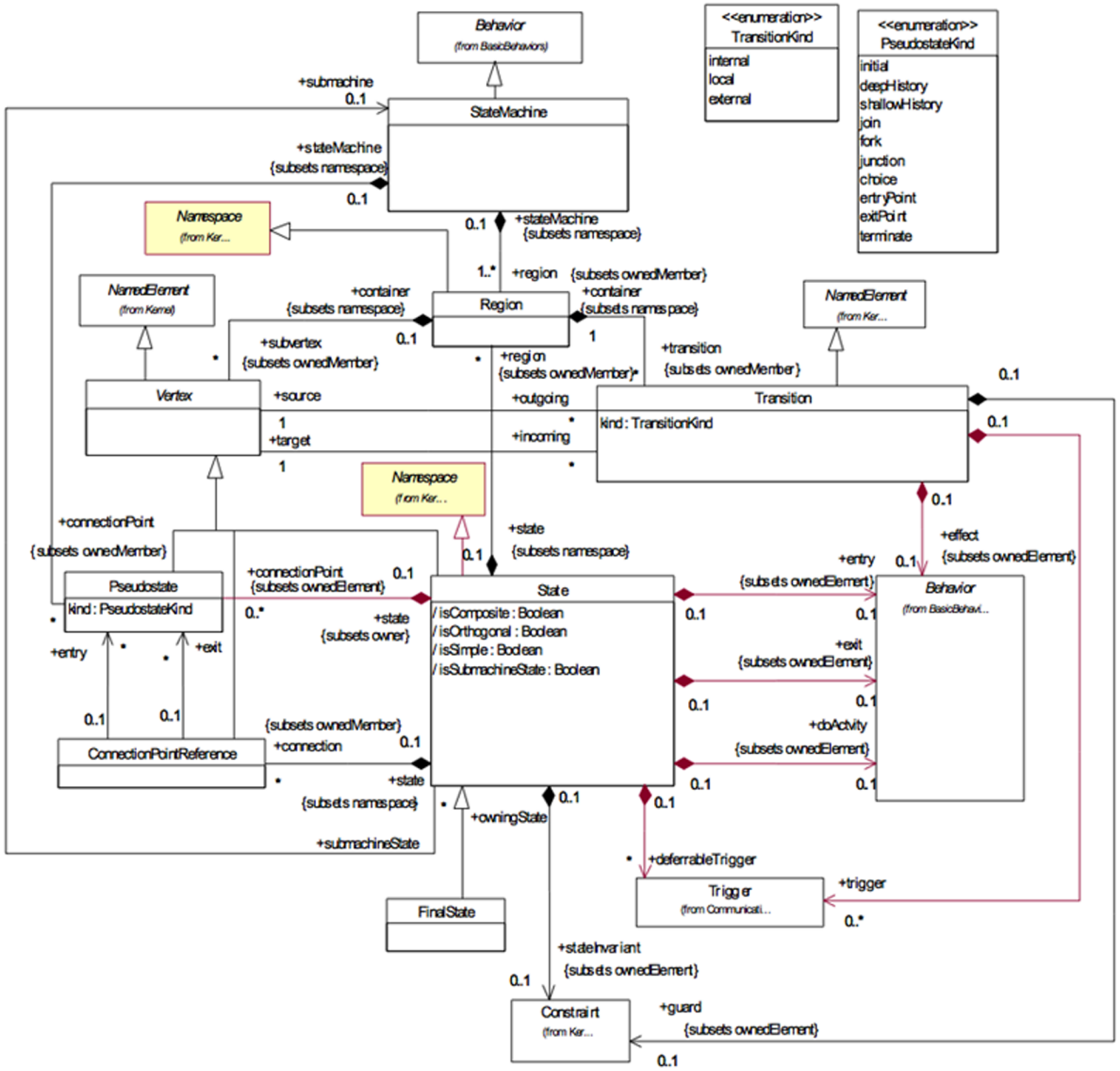
**R. S. Pressman**. *Ingeniería del software: un enfoque práctico*. Editorial McGraw-Hill.

### 4.2.2. UML com a llenguatge de modelització en MDE

L'especificació d'UML donada per l'OMG segueix un enfocament formal de metamodelització: tot model UML és una instància del seu metamodel. Aquestes relacions d'instanciació permeten aplicar mecanismes estàndard de transformació de models. Així, per exemple, tot diagrama de classes que sigui conforme ha de ser una instància del metamodel següent, ja que s'indica així en l'especificació oficial d'UML (document *Unified Modeling Language: Infrastructure*, versió 2.0, de març de 2006, pàg. 105).




D'altra banda, també tots els aspectes de comportament estan formalment definits en UML. Les màquines d'estat, per exemple, han de ser conformes al metamodel següent:

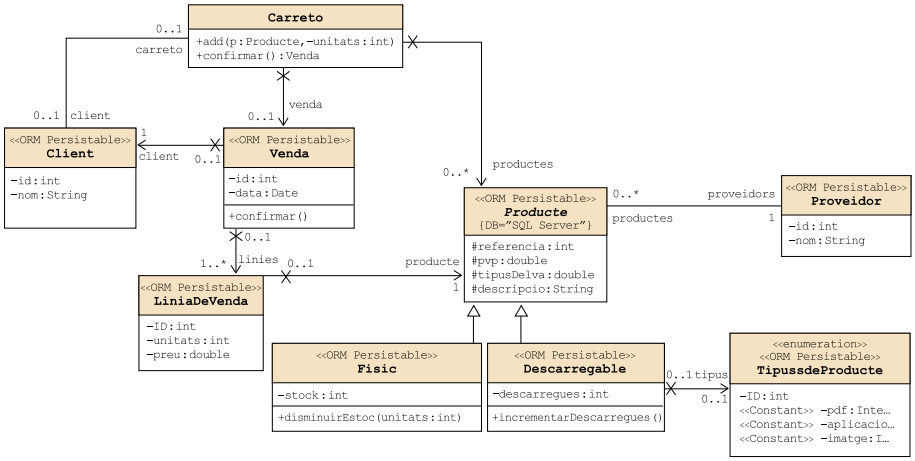



Un estat concret d'una màquina d'estats representada segons el model anterior és, realment, una instància de la classe *State* que apareix en la figura. Per això, a les classes incloses en els metamodels d'UML s'anomenen *metaclasses*. Per acabar la il·lustració, una classe d'un diagrama de classes és una instància de la metaclasse *Class*.

### Perfils d'UML

Assumint ja que coneixem què és una metaclasse, concepte que s'acaba d'introduir, els perfils permetran estendre metaclasses dels metamodels perquè siguin útils per a propòsits específics. Un perfil es compon d'estereotips i de valors etiquetats (*tagged values*). El diagrama de classes de la figura següent representa part de la capa de domini d'un sistema de gestió d'una tenda virtual: totes les classes, excepte *Carreto* (que és una classe volàtil), tenen l'estereotip *ORM Persistable*, la qual cosa denota que són classes persistents que hauran

de ser transformades a taules mitjançant algun mecanisme de mapatge objecte-relacional. La classe *Producte*, a més, té el *tagged value {DB="SQL Server"}*, que indica que ha de ser transformada a una taula per al gestor de bases de dades relacionals SQL Server.




Llavors, els perfils permeten dotar als elements dels models UML de noves característiques, com per exemple:

- Utilitzar una terminologia adaptada a una plataforma o domini particular. Dos possibles valors d'un *tagged value* d'una classe "*ORM Persistable*" podrien ser *{Database=SQL Server}* i *{Database=Oracle}*, la qual cosa pot representar una variació en el tractament que cal donar a transformació d'aquestes classes.

- Utilitzar una notació per als elements que no tenen notació (com les accions, per exemple).

- Utilitzar una notació diferent per a símbols ja existents (utilitzar una icona que mostri un formulari per a representar una classe de la capa de presentació).

- Dotar de més semàntica a algun element del metamodel.

- Crear semàntica per a elements que no existeixen en el metamodel (estereotipant, per exemple, un actor d'UML amb l'estereotip "*Timer*" per a denotar que és un rellotge que, periòdicament, envia algun tipus d'estímul al sistema).

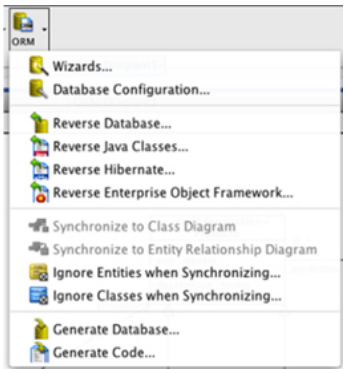
- Evitar construccions que sí que estan permeses en el metamodel.

- Afegir informació necessària per a fer transformacions entre models, com el cas de l'estereotip *ORMPersistable* que s'ha comentat.

L'especificació d'UML inclou diversos exemples de conjunts d'estereotips per a diverses plataformes de desenvolupament d'aplicacions basades en components, com JEE/EJB, COM, .NET o CCM.

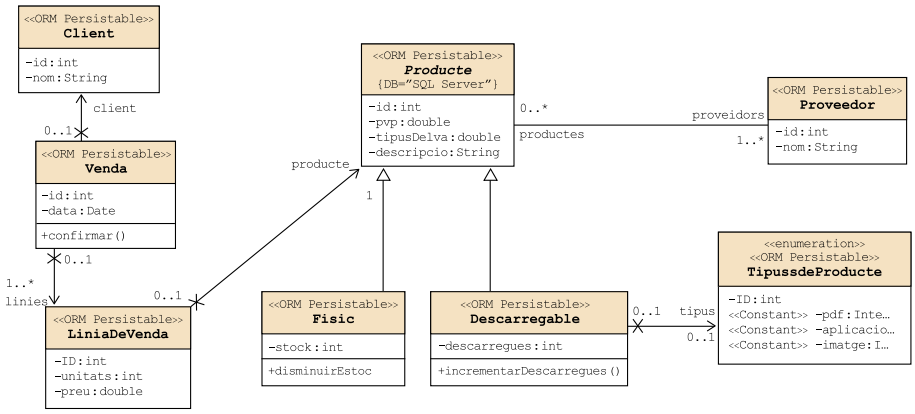
### 4.2.3. Suport automàtic

Cada vegada apareixen més eines (i connectors per a eines existents) capaces de bregar amb models i de fer transformacions entre models. Una d'aquestes eines, compatible amb UML 2, és el Visual Paradigm, un paquet d'eines comercials de modelització amb diferents edicions (estàndard, professional, empresarial), que dóna, per exemple, la possibilitat de generar codi en diferents llenguatges i d'obtenir diagrames a partir de codi.

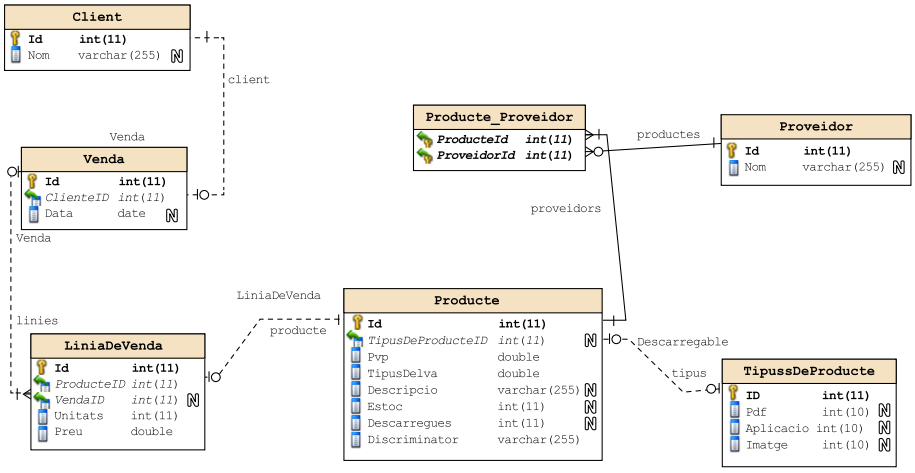



El Visual Paradigm té assistents per a generació de codi i enginyeria inversa.

Com s'ha comentat anteriorment, una de les transformacions de models més habituals és la transformació de diagrames de classes en bases de dades: en efecte, el diagrama de classes de la capa de domini es pot utilitzar en molts casos com a model de dades per a construir la base de dades del sistema. El Visual Paradigm disposa de l'estereotip "*ORM Persistable*", amb el qual s'anoten les classes que s'han de transformar en taules:

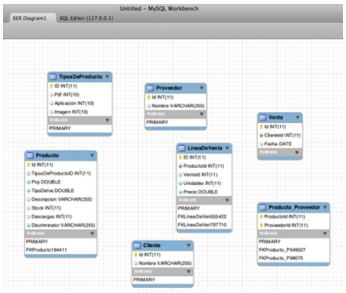



Mitjançant un assistent, en el qual es poden configurar molts aspectes de la transformació, l'eina genera un diagrama entitat-interrelació, que es mostra en la figura següent. Noteu, per exemple, la creació de l'entitat intermèdia *Producte-Proveïdor*, que procedeix de l'associació de molts a molts entre les classes *Producte* i *Proveïdor*. Igualment, el petit arbre d'herència del *Producte* i les seves dues especialitzacions s'han traduït a una sola entitat *Producte*, en la qual un atribut *Discriminator* s'utilitza per a diferències entre els dos subtipus (*Fisic* i *Descarregable*).




En un últim pas, l'eina connecta via JDBC amb el gestor de base de dades que vulguem, es generen les instruccions SQL de creació de taules que corresponguin i es crea el model físic de la base de dades. En aquest exemple, generem una base de dades per a MySQL:

```
CREATE DATABASE  IF NOT EXISTS `tienda`
USE `tienda`;
-- MySQL dump 10.13  Distrib 5.5.16, for osx10.5 (i386)
--
-- Host: localhost    Database: tienda
-- ------------------------------------------------------
-- Server version     5.1.50

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `Proveedor`
--

DROP TABLE IF EXISTS `Proveedor`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `Proveedor` (
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `Nombre` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`Id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;


    ...
```



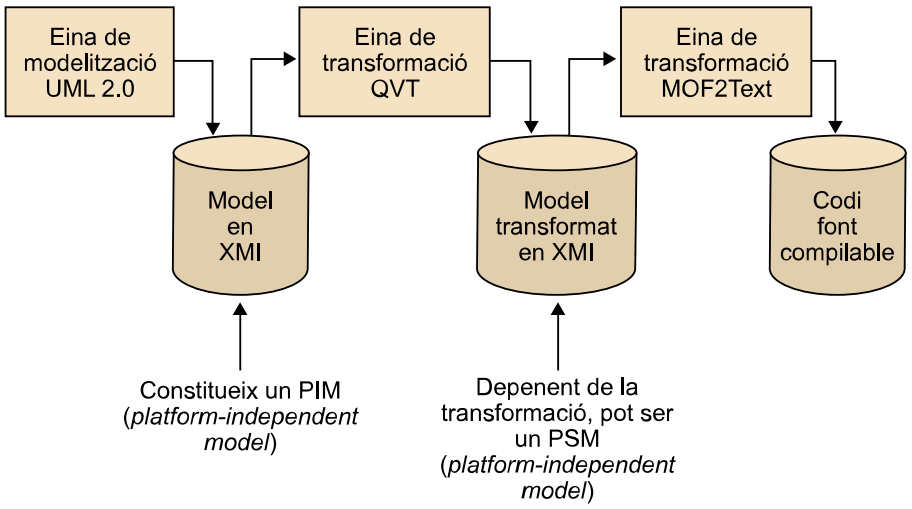
Algunes de les taules que es creen en la base de dades

A més, els models construïts amb aquesta eina es poden exportar en format XMI i importar, per exemple, amb Medini QVT, un conjunt d'eines (distribuïdes sota llicència EPL) que implementa l'estàndard QVT de l'OMG. D'aquesta manera, podem dissenyar un sistema orientat a objectes amb Visual-Paradigm (o amb una altra eina de modelització que exporti en XMI), crear un perfil

amb estereotips i valors etiquetats, anotar amb aquests els elements del sistema orientat a objectes, i finalment implementar les transformacions que vulguem en el Medini.

Als models obtinguts amb el Medini QVT els podrem aplicar una transformació amb el MOFScript (una implementació de MOF2Text, l'estàndard d'OMG per a transformació de models a text) i generar el codi que vulguem.

Amb tot això, podem donar suport a un procés complet d'MDE:




Una altra eina molt coneguda per a la transformació de models és Atlas, desenvolupada per AtlanMod, un grup de recerca de l'Institut Nacional de Recerca en Informàtica i Automàtica de l'Escola de Mines de Nantes, a França. Atlas inclou el llenguatge de transformació ATL i és conforme a MOF (és a dir, defineix el seu metamodel mateix), encara que no és extensió d'UML.

### 4.3. Llenguatges de domini específic (DSL: *domain-specific languages*)

Els llenguatges de domini específic[24] (DSL) permeten descriure sistemes des d'un nivell molt alt d'abstracció, utilitzant una terminologia i una notació molt propera a la dels experts del domini. De fet, un mateix sistema pot ser descrit amb diversos DSL adequats als diferents tipus d'experts. Aquesta característica permet que sigui l'expert mateix el que modelitzi el seu sistema; més endavant, aquesta descripció haurà de ser processable automàticament per un computador. Per a aconseguir aquest processament automàtic, el model construït per l'expert es transforma a un altre model que, ara sí, podrà ser aplicat en un entorn MDE per a generar el codi d'una aplicació.

Un DSL es compon del següent:

[24] Sovint es tradueix *domain-specific language* per *llenguatge específic de domini*. Considerem més encertat utilitzar *llenguatge de domini específic*, ja que un DSL és un llenguatge especialment adequat per a representar un domini d'aplicació ben concret.

**Vegeu també**

En el mòdul sobre "Desenvolupament de programari dirigit per models" es tracten els DSL amb més profunditat.

- Una **sintaxi abstracta**, que conté els conceptes del llenguatge, el vocabulari, les relacions entre aquests i les regles que permeten construir sentències vàlides.

- Una **sintaxi concreta**, que descriu la notació permesa per a construir models, i que pot ser visual o textual.

- Una **semàntica**, que associa significat al model i als seus elements, i que pot ser denotacional[25], operacional[26] o axiomàtica[27].

[25] Tradueix cada sentència textual o cada model visual a sentències o models en altres llenguatges.

[26] Associa comportament als models.

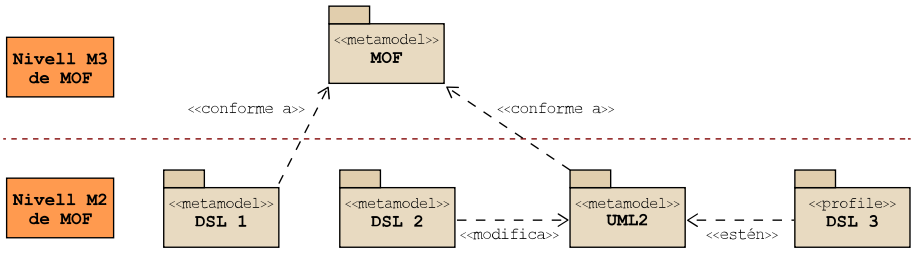
[27] Descriu axiomes, regles o predicats que han de complir les sentències del llenguatge perquè se'n pugui raonar.

### 4.3.1. Definició de DSL

Tenint en compte el requisit que els dominis modelitzats amb un DSL han de poder ser transformats a models processables en entorns MDE, la construcció de llenguatges de domini específic es pot fer de dues maneres:

- Definint un metamodel nou que sigui conforme a MOF. En la figura següent, seria el cas del DSL 1.

- Estenent el metamodel d'UML mitjançant l'addició, modificació o eliminació d'elements (el que es coneix com una "extensió pesant", *heavy extension*, que en la figura correspon al DSL 2) o mitjançant l'especialització d'alguns dels seus conceptes creant un perfil, com el DSL 3.




La definició de dominis és un concepte essencial tant en línies de producte de programari com en programació generativa.

**Nota**

Tots dos paradigmes es tracten a continuació. En les seccions que dediquem a aquests dos temes es podrà entendre la relació dels DSL amb la reutilització.

## 4.4. Línia de producte de programari

Clements i Nortrop (2001) defineixen una línia de producte programari (LPP) com

> "Un conjunt de sistemes programari que comparteixen un conjunt comú i gestionat de característiques (*features*) que satisfan les necessitats específiques d'un domini o segment particular de mercat, i que es desenvolupen a partir d'un sistema comú d'actius base (*core assets*) d'una manera preestablerta."

**Consulta recomanada**

**P. Clements; L. Northrop** (2001). *Software Product Lines: Practices and Patterns*. Addison Wesley.

Gomaa (2005) apunta que l'interès per les LPP sorgeix en el camp de la reutilització del programari, quan els desenvolupadors observen que resulta molt més profitós reutilitzar dissenys arquitectònics complets en lloc de components programari individuals. De fet, la idea de les LPP consisteix en la construcció de productes programari a partir de la reutilització dels *core assets* esmentats.

**Consulta recomanada**

**H. Gomaa** (2005). *Designing Software Product Lines with UML. From use cases to pattern-based software architectures*. Addison Wesley.

L'enfocament de reutilització en aquest context és més planificat que oportunista (Díaz i Trujillo, 2007): en desenvolupament de programari tradicional, s'aprecia la possibilitat de reutilitzar un component després d'haver-lo desenvolupat; en LPP, la reutilització és planificada, de manera que es reutilitzen la major part dels actius base en tots els productes de la línia.

**Consulta recomanada**

**Ó. Díaz; S. Trujillo** (2010). "Líneas de Producto Software". A: Piattini; Garzás (eds.). *Fábricas de Software*. Madrid: Editorial Ra-Ma.

Si volem, per exemple, desenvolupar un sistema client-servidor que permeti a diverses persones jugar a jocs de taula de manera remota (escacs, dames, parxís, Trivial, etc.), l'empresa de desenvolupament pot aplicar un enfocament d'LPP i identificar els *core assets* següents, que estaran presents en tots o gairebé tots els productes que s'implementin:

**1) Moure fitxa**. En tots aquests jocs es mou una fitxa en el torn del jugador.

**2) Gestió del torn**. En tots els jocs esmentats se segueix una certa política d'assignació del torn al jugador següent, que varia en funció del joc[28].

[28]Torns alternatius en escacs i dames; dos moviments seguits si es treuen dos sisos en el parxís o si s'encerta una pregunta en el Trivial, per exemple.

**3) Tirar els daus**. Alguns dels jocs (parxís, Trivial i altres) necessiten que el jugador tiri els daus per a moure.

**4) Romandre en la sala d'espera**. Per a poder començar una partida es necessiten exactament dos jugadors en escacs i dames, i dos o més de dos en els restants.

**5) Incorporació a una partida**. Perquè una partida pugui començar, cal que els jugadors s'hi incorporin.

**6) Sistema de registre**. Atès que es tracta d'un sistema client-servidor, és probable que s'exigeixi a tots els jugadors estar registrats prèviament en el sistema.

**7) Sistema d'identificació**. Per a incorporar-se a una partida és necessari estar registrat i haver-se identificat mitjançant, per exemple, un sistema basat en un nom d'usuari i una contrasenya.

**8) Menjar**. En alguns dels jocs es poden menjar fitxes del contrincant: en escacs i dames, la fitxa menjada desapareix; en el parxís, la fitxa menjada torna a la casella d'origen; en el Trivial no es mengen fitxes.

No tots els jocs de taula tenen totes aquestes característiques, i no tots les implementen de la mateixa manera; a més, cadascun té una altra sèrie de característiques pròpies.

> Per exemple, els escacs i les dames comparteixen el tauler, però no el parxís ni el Trivial; els moviments permesos són diferents en tots els jocs; en els escacs es guanya quan es fa escac i mat, en les dames venç el jugador que deixa al contrari sense fitxes, en el Trivial, aquell que reuneix sis "formatgets", i el que porta totes les seves fitxes a la casella final en el parxís.

En un desenvolupament de programari tradicional l'empresa començaria, probablement, per la implementació completa d'un joc concret: per exemple, el parxís. A partir d'aquest, és possible que abordés la construcció del Trivial, per a la qual cosa podria reutilitzar la lògica responsable del registre i identificació d'usuaris, i també l'encarregada tirar els daus. Potser aquestes funcionalitats estan empaquetades en components. Malgrat això, la reutilització, en aquest cas, es produeix *a posteriori*.

En LPP es treballa a dos nivells:

- En el nivell d'**enginyeria de domini** s'analitzen, dissenyen, implementen i proven les característiques comunes, que s'inclouran en tots o gairebé tots els productes de la línia, i que es du a terme mitjançant la construcció de components.

- En el nivell d'**enginyeria de producte** s'analitzen, dissenyen, implementen i proven els productes concrets, i s'incorporen a cadascun les característiques que els corresponguin i que procedeixin del nivell d'enginyeria de domini.

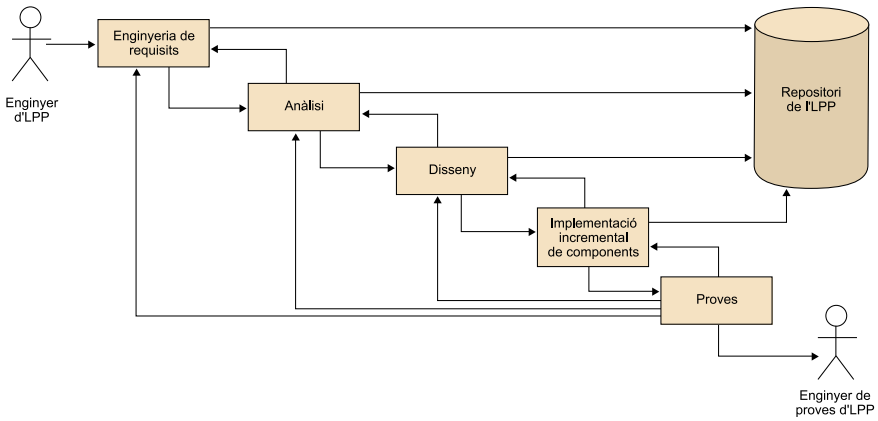
### 4.4.1. Enginyeria de domini

La figura següent mostra les etapes del procés de desenvolupament al nivell d'enginyeria de domini segons la proposta de Gomaa (2005).

**Consulta recomanada**

**H. Gomaa** (2005). *Designing Software Product Lines with UML. From use cases to pattern-based software architectures*. Addison Wesley.

Cada etapa alimenta el repositori de l'LPP. Com il·lustren les fletxes, l'execució de les tasques d'una etapa pot representar la revisió de l'etapa anterior. L'etapa final, de proves, pot implicar la introducció de canvis en els productes de qualsevol de les etapes anteriors.




## Enginyeria de requisits

Es desenvolupa un model de casos d'ús i un *feature model* (model de característiques). En LPP, no tots els actors ni tots els casos d'ús són necessaris per a tots els productes. Per això, els casos d'ús s'identifiquen i estereotipen d'acord si són:

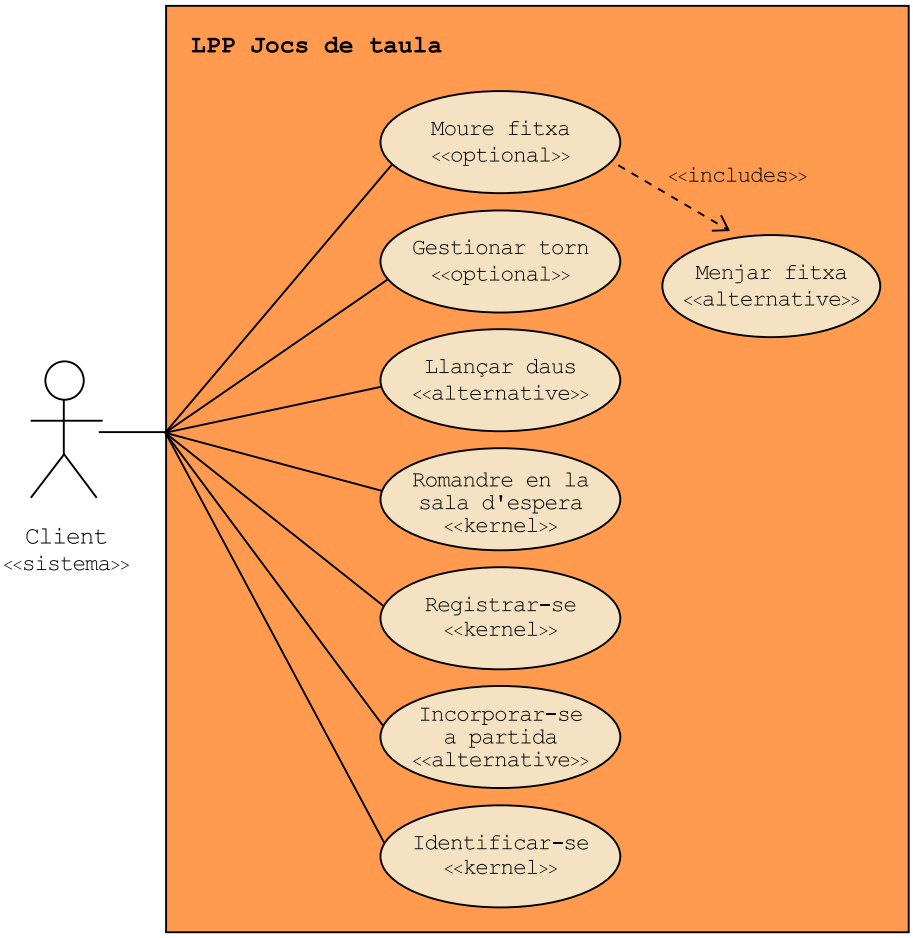
**Vegeu també**

Vegeu l'assignatura *Enginyeria de requisits* del grau d'Enginyeria Informàtica.

- *Kernel*, que són aquells casos d'ús que hauran d'estar presents en tots els productes de la línia i la implementació dels quals és fixa.

- *Optional*, que estaran presents en alguns productes, però no en tots. La implementació és també fixa: és a dir, els productes que incorporen un cas d'ús *optional* posseeixen la mateixa implementació del cas d'ús.

- *Alternative*, que són aquells casos d'ús que estan presents només en alguns productes i que, a més, poden tenir implementacions diferents en funció del producte. En aquests casos d'ús es poden identificar punts de variació (*variation points*), que són aquelles ubicacions del cas d'ús en les quals una situació es gestiona de manera diferent segons el producte.




En el *feature model*, d'altra banda, es recullen les característiques (*features*) de l'LPP i es representen les dependències entre aquestes.

Una *feature*, per tant, és un requisit o característica que ofereix almenys un dels productes de la línia.

S'utilitzen sis estereotips per a les *features*:

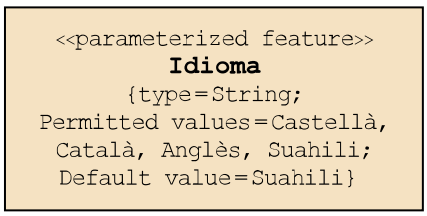
- *Common feature*, que són aquelles *features* presents en tots els productes de la línia.

- *Optional feature*, que són característiques opcionals, que no estaran presents en tots els productes.

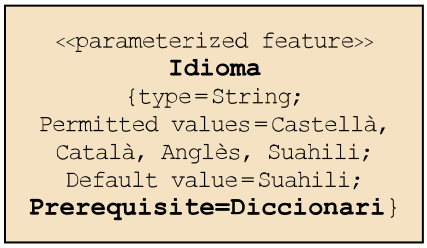
- *Alternative feature*, que són *features* que es presenten en grups i que són mútuament excloents (és a dir, un producte pot oferir només una de les *features* incloses en el grup).

- *Parameterized feature*, que és una *feature* que defineix un paràmetre el valor del qual s'ha de definir en la configuració del sistema. Una característica d'aquest tipus requereix un tipus del paràmetre, un rang de valors i, opci-

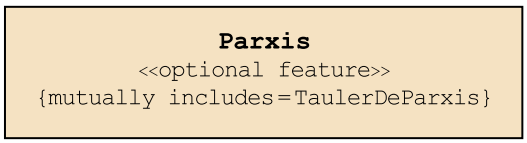
onalment, un valor per defecte. En el cas dels jocs de taula, i suposant que la interfície d'usuari és multiidioma, es podria descriure aquesta característica com en la figura següent:



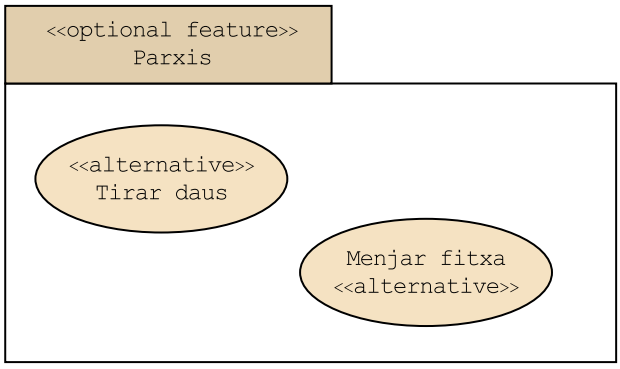

- *Prerequisite features*, que són aquelles *features* necessàries per a altres *features*. No cal indicar que les *common features* són prerequisits perquè ja ho són per defecte, però sí s'han d'anotar les opcionals o alternatives que siguin prerequisit d'altres. A continuació mostrem la figura anterior, però estesa amb la necessitat de disposar d'un diccionari:




- *Mutually inclusivament features*, que s'utilitzen per a representar que dues *features* es necessiten mútuament. En la figura següent es representa que el joc del parxís necessita, en efecte, un tauler de parxís.



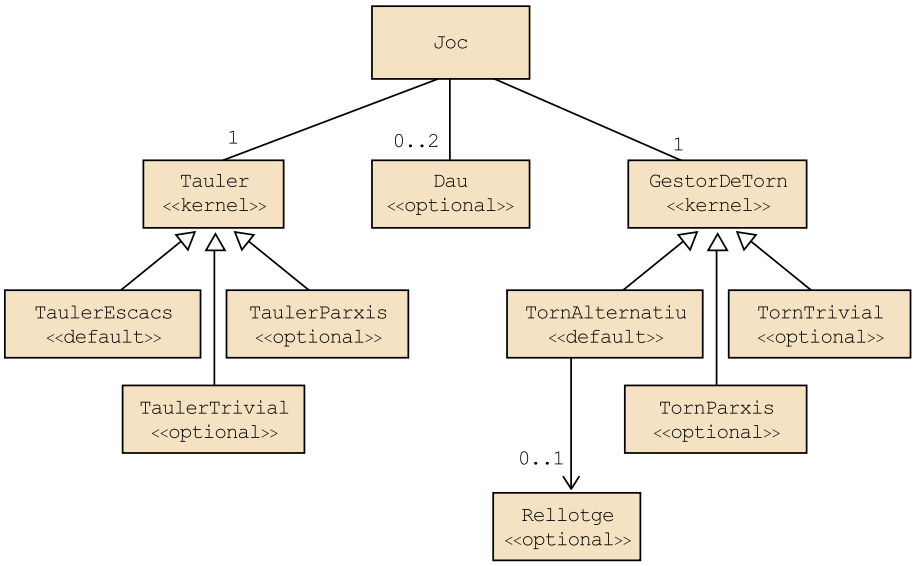

Òbviament, una *feature* pot agrupar diversos requisits funcionals. Per a representar els casos d'ús inclosos en una *feature* es poden utilitzar paquets d'UML:



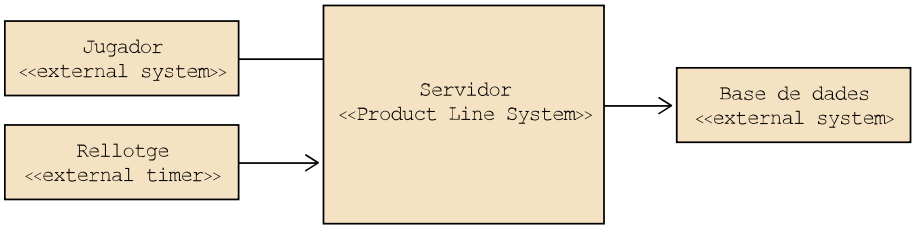
### Anàlisi

D'aquesta etapa s'obtenen almenys tres productes:

- Un **model de classes d'anàlisis**, en el qual cada classe s'estereotipa segons sigui *kernel, optional* o *variant*. Com que hi pot haver herència de classes (fins i tot en classes *kernel*), alguna de les subclasses es pot estereotipar com a *default* si serà la versió per defecte que s'utilitzi en els productes. En la figura següent es representa part de l'estructura d'un joc de taula: un joc té un tauler, potser un o dos daus i un gestor de torns. El tipus de tauler per defecte és el format per escacs, que s'utilitza per a jugar als escacs; el gestor de torn per defecte és l'adequat per a aquests dos jocs, que passa el torn alternativament d'un jugador a un altre i que pot, com s'observa, disposar d'un rellotge.




Es pot crear, a més, un diagrama de classes de context, en el qual es captura la variabilitat en els límits del sistema, tenint en compte els actors que el poden afectar, i que seran dispositius externs d'entrada, dispositius externs de sortida o dispositius externs d'entrada/sortida. En la pràctica, aquests sistemes externs (típicament actors) poden ser persones, dispositius, altres sistemes o rellotges (*timers*) que, periòdicament, envien un senyal al sistema (per exemple, fer una còpia de seguretat). Per a representar les relacions d'aquests actors amb el sistema, aquest es pot representar amb un símbol de classe (un requadre) estereotipat com a «*product line System*».



El model de classes de context proporciona una vista del sistema de molt alt nivell. El desenvolupament pot continuar identificant, per al *product line system*, les classes *boundary, entity, controllers*, etc.

- Un **model dinàmic del sistema**, que estarà format per diagrames de col·laboració entre objectes i, per a aquells objectes amb un comportament dinàmic significatiu, també per màquines d'estat. En general se seguirà el principi de *kernel first*: és a dir, modelitzar primer les interaccions i el comportament de les classes *kernel*, i seguir després amb les classes *optional* i *variant*.

- Una **especificació de les classes** que intervenen en *cada feature*.

**Disseny**

En aquesta fase es dissenya l'arquitectura de la línia, tenint en compte la pròxima implementació dels components. El model d'anàlisi, que representa el domini del problema, es tradueix a un model de disseny, que es correspon amb el domini de la solució. Les classes de disseny s'associen a components, que s'implementaran en la fase següent.

Els punts de variació, que s'utilitzen en la descripció dels casos d'ús per a representar en quins llocs pot variar el processament, es traslladen a les classes en aquesta fase de disseny. Per a això, s'utilitzen els estereotips que es mostren en l'arbre d'herència que apareix més avall i que s'expliquen a continuació. En tots els casos, el sufix *vp* denota que es tracta d'un punt de variació (*variation point*):

**1) *Kernel***: classe que s'inclou sense canvis en tots els productes.

- *Kernel-param-vp* (classe *kernel* parametritzada): classe *kernel* amb un paràmetre el valor del qual s'estableix en temps de configuració.

- *Kernel-abstract-vp* (classe *kernel* abstracta): classe abstracta inclosa en cada producte. Essencialment, representa una interfície de comunicació per a les seves subclasses.

- *Kernel-vp* (classe *kernel* concreta): representa una classe concreta que es redefineix mitjançant especialitzacions i que, a diferència de l'anterior, pot ser instanciada.

**2) *Optional***: classe que s'inclou en alguns productes de l'LPP. Si s'usa, s'utilitza sense canvis.

- *Optional-param-vp* (classe opcional parametritzada): representa una classe parametritzada que estarà present en alguns productes de l'LPP.

- *Optional-abstract-vp* (classe opcional abstracta): és una classe opcional no instanciable, i que proporciona un mitjà de comunicació amb les seves subclasses.

- *Optional-vp* (classe opcional concreta): classe opcional instanciable i que, igual que succeeix amb les estereotipades amb *kernel-vp*, es redefineix mitjançant especialitzacions.

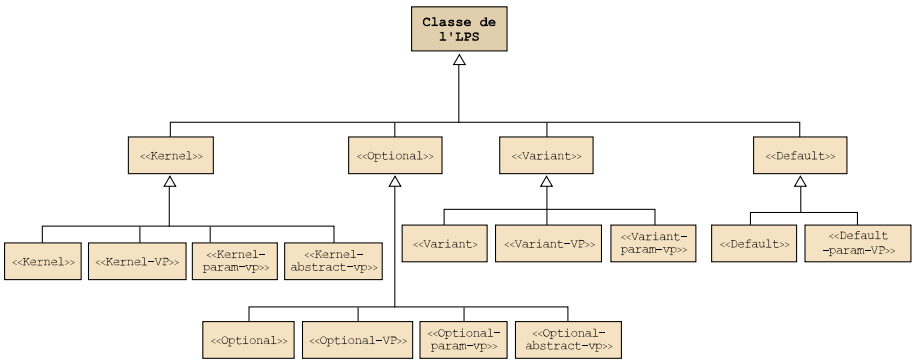
**3) Variant**: classe continguda en un conjunt de classes similars, amb algunes característiques idèntiques.

- *Variant-param-vp* (classe variant parametritzada): representa una classe variant, el comportament de la qual s'estableix en funció d'un paràmetre de configuració.

- *Variant-vp* (classe variant concreta): es tracta d'una classe variant instanciable que ofereix una interfície de comunicació per a les seves especialitzacions.

**4) Default**: classe variant per defecte, inclosa en alguns dels productes de la línia.

- *Default-param-vp* (classe per defecte parametritzable): es tracta de la classe per defecte entre les classes parametritzables per defecte.




### Implementació incremental de components

En aquesta fase es tria un conjunt de casos d'ús i s'implementen els components que hi participen. Es comença per la implementació dels components inclosos en casos d'ús *kernel*, seguits pels *optional* i els *variant*, segons la seqüència que s'hagi determinat durant la modelització dinàmica. A més, es fan proves unitàries de cada component.

**Proves**

L'enginyer de proves fa tant proves d'integració com funcionals, posant principalment la seva atenció en els casos d'ús *kernel*. Si és possible, es proven també els casos d'ús *optional* i *variant*, encara que aquestes proves potser s'han de posposar fins que es disposi de productes implementats.

### 4.4.2. Enginyeria del programari per a línies de producte basada en UML

PLUS és una metodologia per al desenvolupament d'LPP proposada per Gomaa que es recolza en UML i que es basa en el procés unificat de desenvolupament de programari. De la mateixa manera que aquest, consta de quatre fases.

**Inici**

De la fase d'inici s'obtenen els artefactes següents:

- Estudi de viabilitat, abast, mida, funcionalitats, identificació de *commonalities* i *variabilities*, i nombre estimat de productes.
- Identificació de casos d'ús per a cada producte potencial.
- Identificació dels casos d'ús *kernel*.
- Diagrama de classes de context.
- *Feature model* inicial, indicant quins són *kernel*, quins *optional* i quins *variant*.

**Elaboració**

Aquesta fase es du a terme en dues iteracions:

- En la primera (que Gomaa anomena *kernel first*: 'el *kernel*, primer') es revisa i desenvolupa el model de casos d'ús amb més detall. Es determinen els casos d'ús *kernel*, *optional* i *variant* i es fa una anàlisi més en profunditat de les *features*. El *feature model* obtingut ara serà el principal element que guiï en la identificació dels requisits comuns i variables. Els elements del *kernel* es modelitzen mitjançant diagrames d'anàlisi i disseny, i es construeix un esbós inicial de l'arquitectura del sistema.
- En la segona (denominada *product line evolution*: evolució de la línia de productes) es planifica l'evolució de la línia considerant els casos d'ús *optional* i *variant*, les *features* i els punts de variació. Es fan diagrames d'interacció per als casos d'ús *optional* i *variant*, i màquines d'estat per a les classes *optional* i *variant* que tenen una dependència forta de l'estat. El disseny ar-

quitectònic del sistema, esbossat en la iteració anterior, s'estén ara per a incloure els components *optional* i *variant*.

**Construcció**

En les diferents iteracions de què pot constar aquesta fase es construeixen els components que conformen el *kernel*. Per a això, es procedeix fent el disseny detallat, la codificació i la prova unitària de cada component; també es fan proves d'integració dels components del *kernel*.

**Transició**

En arribar a aquesta fase es disposa d'una versió mínima executable de l'LPP. Es fan proves funcionals i d'integració dels components del *kernel* de què es disposa. El sistema, en el seu estat actual, es pot lliurar als enginyers de producte.

**Iteracions addicionals**

Hi pot haver iteracions addicionals en les quals es desenvolupin components *optional* i *variant*. El cap de projecte ha de decidir si aquests components s'implementen durant l'enginyeria de domini o, per contra, es deixen per a més endavant, per al nivell d'enginyeria de producte.

### 4.5. Programació generativa

La programació generativa apareix el 1998 com una proposta de Krzysztof Czarnecki per a resoldre alguns dels problemes que presenta el desenvolupament tradicional de programari quant a reutilització, adaptabilitat, gestió de la complexitat creixent i rendiment.

De manera semblant al desenvolupament de programari en línies de producte, la programació generativa es dedica a la construcció, amb un enfocament de reutilització proactiu, de famílies de productes relacionats amb algun domini particular, en lloc de productes independents.

> La programació generativa tracta del disseny i implementació de programari reutilitzable per a generar famílies de sistemes d'un domini, i no sistemes independents.

Un dels objectius de la programació generativa és la disminució del salt conceptual entre els conceptes del domini i el codi que els implementa. Idealment, i tal com es mostra en el nivell 5 de la figura següent (adaptada de la tesi de màster de César Cuevas), es pretén que "el model sigui el codi".

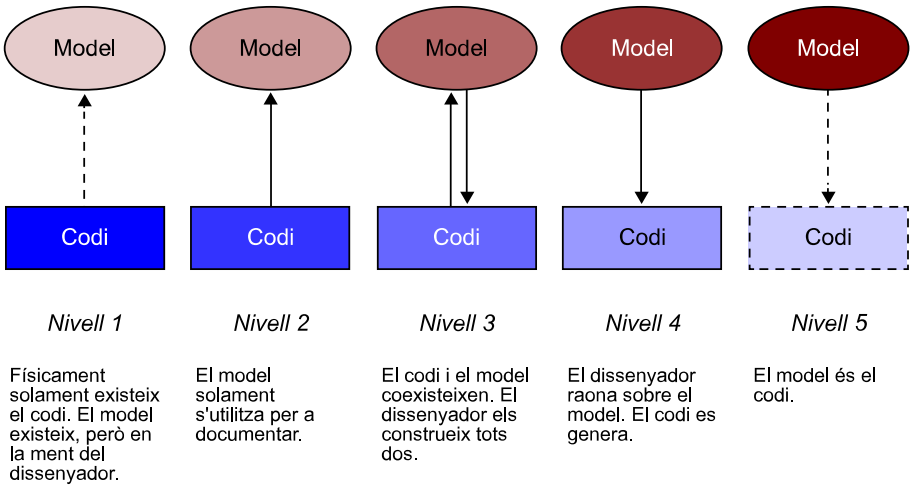
**Consultes recomanades**

A Internet es localitza fàcilment el document PDF amb la tesi doctoral de Czarnecki (*Generative Programming. Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*).

Poc després (el 2000), Czarnecki va publicar amb Eisenecker el llibre *Generative Programming: Methods, Tools and Applications*, amb l'editorial Addison Wesley.

**Consulta recomanada**

La tesi de màster de César Cuevas Cuesta (*Herramientas para el despliegue de aplicaciones basadas en componentes*), de 2009, dedica un capítol a la programació generativa. Està disponible (29 de febrer de 2012) al web de la Universitat de Cantàbria.




Per a això, aquest paradigma pretén la generació automàtica de programes a partir d'especificacions donades a molt alt nivell. Així, s'incrementa l'eficiència en la producció de programari mitjançant l'escriptura i construcció d'elements abstractes que, parametritzats i processats adequadament, proveeixin una implementació real de la solució del problema. Per a aconseguir que tals especificacions "a tan alt nivell" siguin directament processables es treballa a escala de dominis d'aplicació i, per això, la programació generativa es recolza fortament en l'ús de llenguatges de domini específic (DSL).

**Vegeu també**

Repasseu, si cal, l'apartat "Llenguatges de domini específic".

Amb la finalitat de facilitar la reutilització, una altra de les característiques de la programació generativa és la separació de funcionalitats[29] en mòduls o elements independents, de manera que hi hagi tan poca superposició com sigui possible. Per a això s'utilitza intensivament la modelització de característiques (*feature modeling*) i tècniques de programació orientada a aspectes (AOP: *aspect-oriented programming*). Amb aquestes tècniques es pretén obtenir un conjunt d'elements programari de funcionalitat autocontinguda, acoblables i reutilitzables.

[29]La "separació de funcionalitats" que esmentem és el que Dijkstra anomenava *separation of concerns* (E. W. Dijkstra (1982). "On the role of scientific thought". A: *Selected writings on Computing: A Personal Perspective* (pàg. 60-66). Springer-Verlag), terme que no resulta gens fàcil de traduir sense perdre semàntica o sense utilitzar una expressió massa llarga. De vegades es tradueix com a *separació d'incumbències*.

No obstant això, fins i tot dins d'un mateix domini, hi pot haver diferents particularitats que dificultin la generació del producte final. Per això, la programació generativa utilitza també tècniques de programació genèrica, la qual es basa en l'ús intensiu de mecanismes genèrics de programació, com les plantilles de C++ (*templates*) o Java (*generics*).

**Nota**

DSL: *domain specific language*
FM: *feature modeling*
AOP: *aspect-oriented programming*

No cal oblidar que, idealment, el model ha de ser transformable directament al codi executable del producte final. la programació generativa aprofita l'estat actual de la tecnologia combinant les tècniques que s'acaben d'indicar (DSL, FM, AOP, programació genèrica).

**Vegeu també**

Trobareu més informació sobre la programació genèrica més avall a l'apartat "Tecnologies".

### 4.5.1. Enginyeria de domini

Des del punt de vista de la programació generativa, un domini és un conjunt de conceptes i termes relacionats amb una àrea de coneixement determinada (per exemple: hospitals, gasolineres, universitats) i que entenen les persones involucrades en aquesta àrea (metges, gasoliners, professors i alumnes). A més, la identificació i estructuració dels conceptes que pertanyen al domini s'ha d'orientar envers la construcció de sistemes de programari per a aquesta àrea i han de maximitzar la satisfacció dels requisits de les persones involucrades.

Amb la finalitat de modelitzar adequadament el domini d'una família de sistema es necessiten aplicar de manera rigorosa tècniques d'enginyeria de domini, que és l'activitat en la qual, de manera similar al que es fa a l'enginyeria de domini a LPP, es recull i organitza i l'experiència obtinguda en la construcció de sistemes o parts de sistemes d'un domini particular. Aquesta experiència es conserva en forma d'actius reutilitzables (*reusable assets*). Òbviament, aquests actius s'han d'emmagatzemar de manera que es puguin reutilitzar quan es construeixin nous sistemes: tal infraestructura d'emmagatzematge ha d'oferir els mitjans per a recuperar, qualificar, distribuir, adaptar i acoblar els actius reutilitzables desats.

**Nota**

En l'apartat "Components" ja en parlàvem de la qualificació, adaptació i acoblament. La pràctica, en aquest context, és diferent, però la idea és exactament la mateixa.

L'enginyeria de domini té tres fases principals dedicades a l'anàlisi del domini (en la qual es defineix un conjunt de requisits reutilitzables per als sistemes d'aquest domini), al disseny del domini (en què es determina l'arquitectura comuna dels sistemes d'aquest domini) i a la implementació del domini (en la qual s'implementen tant els actius reutilitzables en forma de components, DSL, generadors, etc., com la infraestructura d'emmagatzematge).

**Nota**

Czarnecki explica que afegeix explícitament el postfix *de domini* a les fases d'anàlisi, disseny i implementació per a emfatitzar-ne precisament l'orientació a famílies de productes i no a productes independents o aïllats.

#### Anàlisi del domini

El "conjunt de requisits reutilitzables" que s'ha esmentat en el paràgraf anterior s'obté a partir de l'estudi del domini de què es tracti. Així, les fonts d'informació poden ser molt variades: sistemes ja existents, entrevistes amb experts, llibres, prototips, experimentació o requisits ja coneguts de sistemes futurs.

D'aquesta etapa s'obté un model de domini, en el qual s'expliciten les propietats comunes i variables dels sistemes del domini, i també les dependències entre les propietats variables.

**Vegeu també**

En la secció dedicada a les línies de producte programari ja vam fer una discussió extensa sobre les *commonalities*, les *variabilities* i els *feature models*.

Alguns continguts habituals del model de domini són:

- **Definició**: es descriu l'abast del domini, es donen exemples i contraexemples de sistemes que són a dins i fora, i s'indiquen regles genèriques d'inclusió i exclusió.

- **Vocabulari**: defineix els termes més importants del domini que es tracti.

- **Model de conceptes**: es descriuen els conceptes del domini utilitzant algun formalisme apropiat (diagrames d'objectes, d'interacció, entitat-interrelació...).

- **Model de característiques** (*feature model*): descriu els requisits reutilitzables i configurables amb un nivell de granularitat adequat, en forma de característiques (*features*).

#### Disseny del domini

Com que es desenvolupen famílies de sistemes i no sistemes independents, l'arquitectura que s'obtingui d'aquesta etapa de disseny ha de ser suficientment flexible per a suportar diversos sistemes i l'evolució de la família de sistemes.

#### Implementació del domini

En aquesta fase s'implementen els components, generadors per a composició automàtica de components i la infraestructura de reutilització que permeti la recuperació, qualificació, etc., de components.

### 4.5.2. Tecnologies

A més dels DSL i els *feature models*, que ja s'han tractat anteriorment, en programació generativa s'utilitzen conceptes de programació orientada a aspectes i de programació genèrica.

#### Programació orientada a aspectes

La programació orientada a aspectes sorgeix com una evolució de l'orientació a objectes que pretén incrementar la modularitat i la reutilització: en els sistemes programari hi ha moltes funcionalitats "transversals", en el sentit que s'utilitzen en altres funcionalitats múltiples. Abans de fer una operació sobre una base de dades, per exemple, potser cal fer una comprovació sobre el nivell de permisos de l'usuari que llança l'operació. En un desenvolupament orientat a objectes tradicional, el codi corresponent a aquesta comprovació s'inseriria abans de cada crida a una operació de persistència. La modificació de la política de seguretat o un altre tipus de canvi (potser fins i tot petit) pot representar la modificació d'aquest codi en tots els llocs en què apareix.

Aquest tipus de funcionalitats transversals és el que es coneix com a "aspecte". Cada aspecte s'implementa de manera modular i separada de la resta en el que es diu un *advice* (literalment, 'avís'). El lloc del programa en el qual es crida la funcionalitat donada per un *advice* (és a dir, la implementació de l'aspecte)

és el que es diu un *punt d'enllaç* (*join point*). Els punts d'enllaç s'anoten en *punts de tall* (*pointcuts*), que descriuen mitjançant patrons de noms, expressions regulars i una sèrie de paraules reservades quan cal cridar l'*advice*.

L'entreteixit (*weaving*) és el procés de combinar els aspectes amb la funcionalitat a la qual estan associats, de la qual cosa s'encarrega el *weaver*. En **l'entreteixit estàtic**, la combinació del codi de l'aplicació i dels aspectes es du a terme en temps de compilació. En **el dinàmic**, la combinació es fa en temps d'execució.

La figura següent resumeix un exemple donat en el llibre *Aspect-Oriented Programming with AspectJ*: la funció *main*, inclosa en la classe *A*, construeix una instància de *A* i crida *a(5)*. *a(5)* retorna el resultat de *b(5)* que, al seu torn, retorna el valor que rep com a paràmetre (5, en aquest cas). Mentrestant, la crida a *b* executa també una crida al mètode *c* de *B*.

**Consulta recomanada**

**I. Kiselev** (2003). *Aspect-Oriented Programming with AspectJ*. Editorial SAMS.

**Nota**

La majoria dels llenguatges orientats a aspectes són extensions d'altres "llenguatges base":

- AspectJ (de Java)
- AspectC (de C)
- AspectC++ (de C++)

En la part inferior apareix un aspecte amb tres punts de tall i els seus tres *advices* corresponents:

- L'expressió que hi ha a la dreta del nom del primer (anomenat `int_A_a_int())` enllaça aquest punt de tall amb la crida al mètode amb signatura `int A.a(int)` (és a dir, el mètode `a()` de la classe `A)`. La paraula reservada `call` és un "descriptor de punt de tall". Més avall es troba l'*advice* que correspon a aquest punt de tall: amb la paraula reservada `before()` es denota que l'*advice* s'ha d'executar abans de cridar aquest mètode concret.

- El segon *advice* (anomenat `int_A_all_int())` s'executa després (*after*) de les crides (també s'utilitza del descriptor de punt de tall `call`) a tots els mètodes de `A` que prenguin un paràmetre de tipus `int` i que retornin un `int`, la qual cosa es representa amb l'expressió `int A.*(int)`, que conté el caràcter comodí `(*)` de les expressions regulars.

- El tercer punt de tall `(all_all_c_all())` afecta tots els mètodes que s'anomenin `c`, independentment del tipus que retornin (primer comodí) i de la classe en la qual estan continguts (segon), però que prenguin un sol paràmetre (tercer caràcter comodí, situat entre els parèntesis que corresponen als paràmetres). La paraula reservada (*around*, en aquest cas) serveix per a substituir per complet les crides al punt d'enllaç capturat. No obstant això, amb la crida a *proceed* s'aconsegueix cridar el punt d'enllaç original, si volem.

```
public class A {
      int a(int x) {
            return b(x);
      }
      int b(int x) {
            c("x");
            return x;
      }

      String c(String x) {
            (new B()).c(3.14);
            return x;
      }

      public static void main(String
args[]) {
            A t = new A();
            t.a(5);
      }
}
```

```
public class B {
      void c(double x) {
            …
      }
}
```

```
public aspect Showcase {
      pointcut int_A_a_int():    call(int A.a(int));
      pointcut int_A_all_int(): call(int A.*(int));
      pointcut all_all_c_all(): call(* *.c(*));

      before(): int_A_a_int() {
            System.out.println("Before: " + thisJoinPoint);
      }

      after(): int_A_all_int() || all_all_c_all() {
            System.out.println("After: " + thisJoinPoint);
      }

      Object around(): all_all_c_all() {
            System.out.println("Start around: " + thisJoinPoint);
            Object o = proceed();
            System.out.println("End around: " + thisJoinPoint);
            return o;
      }
}
```

**Aspectes, *features* i programació generativa**

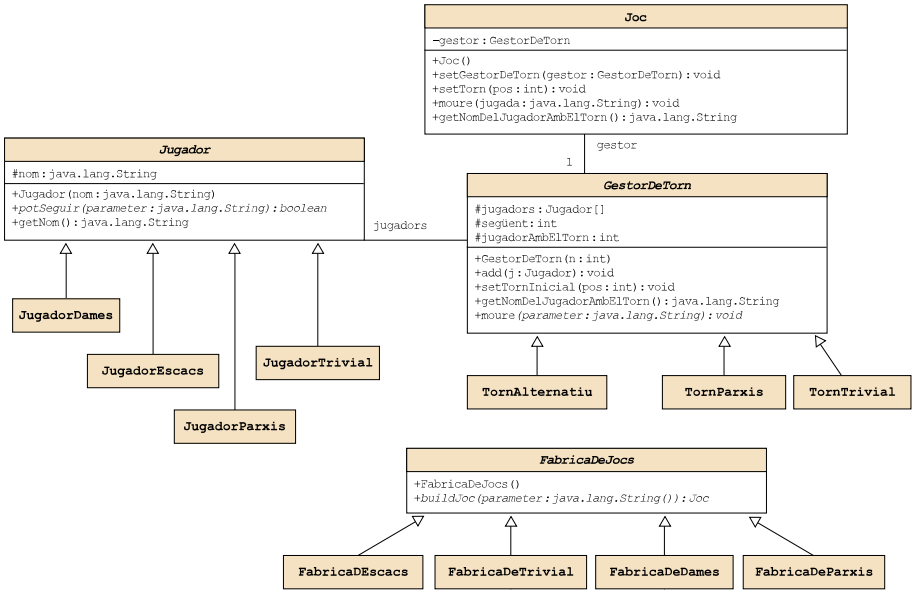
Amb no gaire marge d'error, podem assimilar una *feature* a un aspecte. Per això, el *feature model* que construïm en l'enginyeria de domini ens serveix també com a model per a la implementació dels aspectes; les relacions entre les *features*, igualment, es poden assimilar als punts d'enllaç entre els aspectes i les funcionalitats del sistema.

En compondre els aspectes, hem d'intentar que l'aspecte sigui tan independent de la funcionalitat com sigui possible. Així aconseguirem aspectes reutilitzables. Per a l'efecte recíproc (que les funcionalitats també ho siguin), és convenient utilitzar estàndards adequats de codificació quant a anomenar classes, operacions, camps, variables, etc.

## Programació genèrica

La programació genèrica es dedica més al desenvolupament d'algorismes que al disseny de les estructures de dades sobre les quals operen aquells. Ja en l'apartat dedicat a la reutilització en disseny de programari orientat a objectes es va recordar la noció de les *templates* de C++ i els *generics* de Java: mitjançant aquestes construccions, podem crear classes que actuïn sobre tipus de dades arbitraris, que es declaren en temps d'escriptura de codi, però que s'instancien en temps de compilació.

La figura següent mostra part d'un disseny del possible sistema per a jugar a jocs de taula que vèiem en la secció dedicada a línies de producte programari: es disposa d'una classe abstracta *Jugador* i d'una altra *GestorDeTorn*, que tenen diverses especialitzacions en funció del tipus de joc que es vulgui construir, la qual cosa s'aconsegueix a partir de la *FabricaDeJocs* (estructura que es correspon al patró *FàbricaAbstracta*). Hi ha, a més, una *classe Joc* que actua de *wrapper*.




El disseny mostrat a dalt aprofita l'herència, però no utilitza en absolut la programació genèrica. De fet, el codi de les classes *GestorDeTorn* i *TornAlternatiu* (la implementació utilitzada per a jugar als escacs i a les dames) ens ho confirma:

```
package domini.sinGenerics.torns;

import domini.sinGenerics.jugadors.Jugador;

public abstract class GestorDeTorn {
   protected Jugador[] jugadors;
   protected int següent, jugadorAmbElTorn;

   public GestorDeTorn(int n) {
      this.jugadors=new Jugador[n];
      this.següent=0;
      this.jugadorAmbElTorn=-1;
   }

   public void add(Jugador j) {
      if (this.següent<this.jugadors.length)
         this.jugadors[this.següent++]=j;
   }

   public void setTornInicial(int pos) {
      this.jugadorAmbElTorn=pos;
   }

   public String getNomDelJugadorAmbElTorn() {
      return this.jugadors[this.jugadorAmbElTorn].getNom();
   }

   public abstract void moure(String jugada);
}
```

```
package domini.sinGenerics.torns;

public class TornAlternatiu extends GestorDeTorn {

   public TornAlternatiu(int n) {
      super(n);
   }

   @Override
   public void moure(String jugada) {
      this.jugadorAmbElTorn=
         (this.jugadorAmbElTorn+1) % 2;
   }

}
```

El fragment de codi següent, d'altra banda, crea una partida d'escacs mitjançant la crida al mètode *buildJoc* de la fàbrica concreta *FabricaDEscacs*:

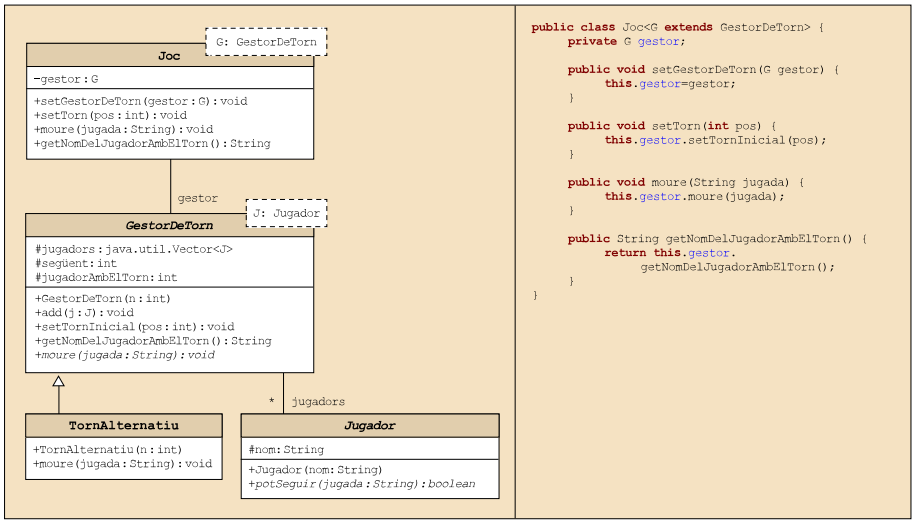
```
FabricaDeJocs f=new FabricaDEscacs();
String[] noms={"Capablanca", "Bobby Fischer"};
Joc j=f.buildJoc(noms);
```

Per donar un enfocament genèric al disseny del mateix sistema, podem pensar en les diferents configuracions de jocs que podem crear. D'aquesta manera, en lloc de tenir una classe *Joc* que coneix un *GestorDeTorn* abstracte que, al seu torn, coneix un *Jugador* també abstracte (aquests dos últims objectes s'instanciaran als subtipus adequats, de manera que la configuració estigui formada per objectes compatibles), i en lloc de delegar la responsabilitat de crear la configuració a cada fàbrica concreta, deixarem el codi preparat amb tipus genèrics perquè la configuració es creï en temps de compilació.

Amb un enfocament genèric, direm que el *Joc* conté un *GestorDeTorn*, i que aquest té, al seu torn, un *Jugador*. El *GestorDeTorn* conegut pel *Joc* es passa com a paràmetre i s'instancia en temps de compilació. Igualment, el *Jugador* que coneix el *GestorDeTorn* es passa a aquest com a paràmetre i també s'instancia en temps de compilació. Així, el resultat de la compilació és una configuració concreta. El tros de codi següent, per exemple, munta una configuració formada per un *TornAlternatiu* i dos jugadors d'escacs: noteu la instanciació que es fa del *Joc* en la primera línia, indicant que volem que es munti amb un *TornAlternatiu*:

```
public static void main(String[] args) {
     Joc<TornAlternatiu> joc=new Joc<TornAlternatiu>();
     String[] nomsDeJugadors={"Capablanca", "Bobby Fischer"};
     for (String s : nomsDeJugadors) {
          JugadorEscacs j=new JugadorEscacs(s);
          joc.gestor.add(j);
     }
          …
}
```

Visualment, el disseny del sistema és el que es mostra al costat esquerre de la figura següent; al costat dret apareix el codi corresponent a la classe *Joc*: noteu que conté un objecte gestor del tipus genèric G, que es declara com a paràmetre en la capçalera de la classe.




Com s'ha dit, la configuració concreta es munta en temps de compilació. El diagrama de classes de la figura anterior procedeix d'haver aplicat enginyeria inversa al codi que utilitzem com a exemple: els tipus parametritzats apareixen en caixes petites amb la vora puntejada.

## 4.6. Fàbriques de programari

D'acord amb Piattini i Garzás (2010), una fàbrica de programari requereix una sèrie de bones pràctiques que organitzin el treball "d'una manera determinada, amb una especialització considerable, i també una formalització i estandardització dels processos". Entre aquestes bones pràctiques, els autors esmenten:

- La definició de l'estratègia de fabricació, tant en desenvolupament tradicional, com en generació automàtica de codi, com en reutilització.

- La implantació d'un model de processos i d'un model de qualitat de programari. La certificació de les fàbriques de programari en algun nivell de maduresa és, moltes vegades, un requisit imprescindible per al desenvolupament de projectes amb grans empreses o amb l'administració pública.

- La implantació d'una metodologia de fabricació.

- Integració contínua i gestió de configuració.

- Control de qualitat exhaustiu, periòdic i automàtic.

- Disseny basat en el coneixement, que eviti la dependència excessiva de determinades persones.

- Construir el programari prenent els casos d'ús com a peça essencial del desenvolupament. El desenvolupament basat en casos d'ús facilita la traçabilitat des dels requisits al codi i permet estimar costos i esforços, moni-

torar l'avenç del projecte, planificar i executar les proves funcionals, guiar en el desenvolupament de la interfície d'usuari i, també, organitzar i redactar el manual de l'usuari d'acord amb tot això.

- Utilitzar cicles de desenvolupament evolutius i iteratius, i no en cascada.

- Estimar els costos utilitzant punts funció adaptats, per a la qual cosa hi ha diverses propostes en la bibliografia que donen bons resultats.

- Guiar les accions pel seu valor, avaluant el retorn de la inversió de cadascuna.

Com s'observa, hi ha una diferència d'actitud pel que fa al que Griss apuntava el 1993, i que ja hem citat en parlar dels costos de desenvolupar programari reutilitzable. Les fàbriques de programari han deixat enrere aquesta consideració de la reutilització com una despesa, i han passat a considerar-ho una inversió.

**La proactivitat en la reutilització**

Griss esmenta altres problemes de la proactivitat en la reutilització, molts dels quals es van eradicant progressivament. Alguns són els següents:

- La síndrome NIH (*not-invented here*, 'no inventat aquí'), que pot semblar un atemptat a la creativitat dels desenvolupadors.

- La falta de tècniques de gestió adequades per a projectes de desenvolupament de programari reutilitzable.

- L'avaluació individual del treball dels desenvolupadors es veu dificultada en projectes d'aquesta naturalesa, on poden no obtenir-se beneficis a curt termini.

Evidentment, el desenvolupament orientat a objectes, el desenvolupament de components, l'enfocament de línies de producte programari, la programació generativa i la resta de temes que hem tractat en aquestes pàgines entren en aquestes estratègies de fabricació, metodologies, etc., que s'esmenten en els punts anteriors.

# Resum

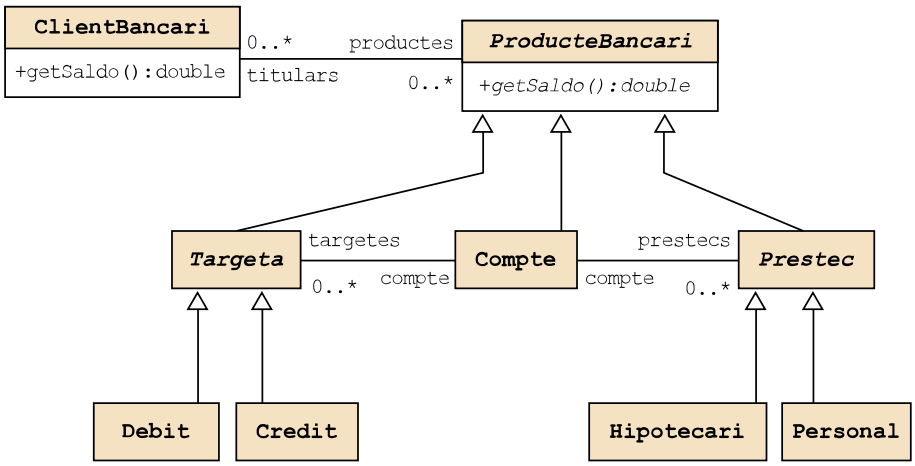
Al llarg d'aquest text, s'han donat unes pinzellades sobre l'evolució històrica quant a reutilització, i s'ha emfatitzat la importància que va representar la introducció del paradigma orientat a objectes. Aquest va ser el pilar fonamental sobre el qual es van desenvolupar altres tecnologies, com les biblioteques de classes, els *frameworks* i els components. Al seu torn, els components han estat la base per a l'aparició d'altres paradigmes com les línies de producte programari (en les quals la modelització exerceix un paper fonamental) o la programació generativa (amb l'ús intensiu de noves evolucions de l'orientació a objectes, com l'AOP o la genericitat). La reutilització proactiva implica la gestió adequada de tots els artefactes de programari que es van construint al llarg del desenvolupament, la qual cosa implica la monitorització no solament del codi font, sinó també dels models dels quals procedeix. En aquest sentit, també el desenvolupament dirigit per models ha representat (i continua representant, ja que és una matèria d'un enorme interès en l'R+D+I en informàtica) un avenç importantíssim.

Esperem que aquest text hagi servit al lector per a assolir els objectius que s'han plantejat a l'inici del mòdul: pensem que la varietat de tecnologies i metodologies presentades hauran ajudat a la consecució dels tres últims. Respecte del primer, potser la lectura d'aquest llibre i de les referències complementàries que s'han anat citant, la resolució dels exercicis plantejats i el pòsit que l'estudi haurà anat deixant hagin aconseguit finalment convèncer al lector que, en efecte, hem de tenir una mentalitat suficientment oberta i considerar la reutilització com un actiu fonamental dels nostres sistemes programari.

# Activitats

**1.** Un tipus de compte corrent determinat ofereix operacions per a ingressar diners (incrementant el saldo), retirar diners (disminuint-lo, sempre que hi hagi saldo suficient per a l'import que es vol retirar) i per a transferir un import a un altre compte (disminuint el saldo del compte origen en l'import transferit més un 1% de comissió, sempre que en el compte origen hi hagi saldo suficient). Quan el compte es crea, el saldo és zero. Proporcioneu una descripció del *Compte* com un tipus abstracte de dada.

**2.** El diagrama següent mostra l'estructura de classes d'un possible subsistema bancari. El saldo d'un client es defineix com la suma dels saldos de tots els seus comptes, menys la suma dels saldos de les seves targetes de crèdit (se n'exclouen les de dèbit) i menys la suma dels saldos dels seus préstecs. Escriviu el codi o el pseudocodi de l'operació *getSaldo()* de *ClientBancari* perquè retorni el resultat segons l'especificació que s'ha donat. Observeu que potser és convenient efectuar alguna modificació en el diagrama de classes.




**3.** El sistema informàtic d'una empresa de vendes fa totes les nits les tasques següents:

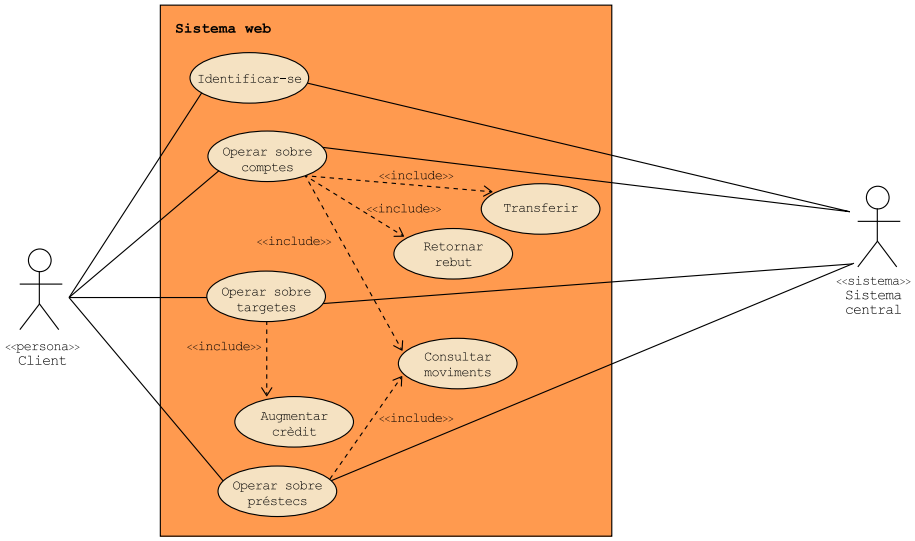
**a)** exporta totes les dades de totes les vendes i els clients nous, que es troben en una base de dades, a tres fitxers de text pla (vendes, línies de venda i clients);

**b)** un altre procés importa aquestes dades i les desa en una altra base de dades diferent;

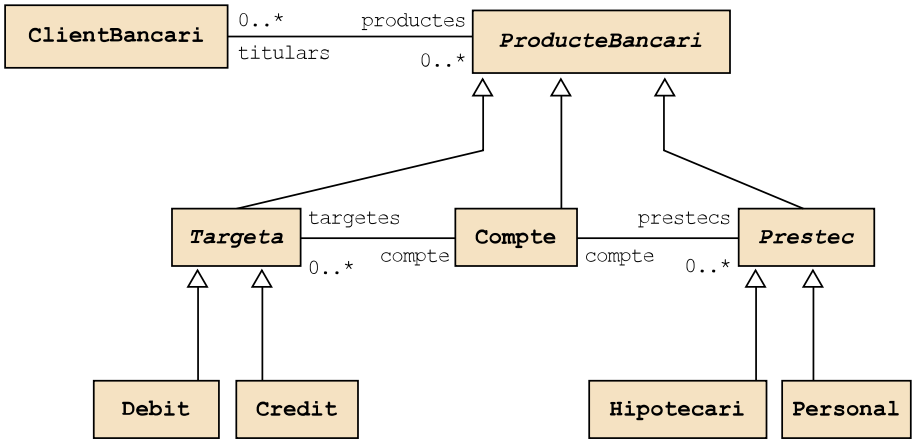
**c)** un altre procés agafa els tres fitxers de la tasca *a*, els totalitza i els desa en un fitxer de totals.

Es demana que representeu aquestes tasques amb una arquitectura de *pipes and filters*.

**4.** Un banc vol oferir als seus clients la possibilitat de consultar i manejar la informació dels seus productes bancaris mitjançant una aplicació web. Es disposa, d'una banda, del diagrama de casos d'ús del sistema que es vol desenvolupar. Com s'observa, tots els casos d'ús requereixen la interacció amb el sistema central del banc, que és el que disposa de tota la informació relativa a clients, comptes, targetes i préstecs:




D'altra banda, s'ha construït un diagrama amb les classes que faran aquests casos d'ús. Aquestes classes s'executaran en el servidor web i, per a executar les seves operacions, s'hauran de connectar a l'actor *Sistema central*, sistema que també en gestiona la persistència i que, per tant, no forma part d'aquest problema.

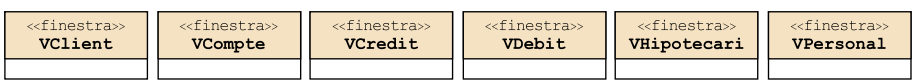



Us demanen que completeu el disseny de classes anterior perquè el sistema tingui una arquitectura multicapa i es pugui implementar utilitzant Struts2.

**5.** A partir del diagrama de casos d'ús i del diagrama de classes de l'exercici anterior, us demanen que dissenyeu un component que ofereixi totes les funcionalitats descrites i que sigui capaç de connectar-se amb el *Sistema central*.

**6.** Modifiqueu els dissenys obtinguts en l'exercici 4 suposant que podeu utilitzar el component dissenyat en l'exercici 5.

**7.** Suposeu que el diagrama de classes mostrat en l'exercici 4 forma part del sistema de gestió que utilitzen els empleats del sistema del banc, de manera que diversos empleats poden estar veient simultàniament l'estat d'una mateixa instància. Quan hi ha un canvi en algun dels objectes, volem que s'actualitzin les vistes corresponents a totes les finestres que estan "mirant" aquest objecte. Suposeu que es disposa de les sis finestres següents per a manipular clients, comptes, targetes de crèdit i dèbit, i préstecs hipotecaris i personals, i que són aquestes les finestres que s'han d'actualitzar quan hi ha algun canvi en l'objecte de domini que estan observant.




Proposeu una solució per a aquest problema basada en patrons.

**8.** La National Oceanic and Atmospheric Administration's dels Estats Units ofereix un servei web amb diverses funcions per a conèixer la previsió del temps als Estats Units i Puerto Rico. El document WSDL amb l'especificació de la interfície d'accés es troba a: http://graphical.weather.gov/xml/SOAP.

Una de les operacions ofertes és *NDFDgenByDay(latitud, longitud, diaInicial, numeroDeDias, unidad, formato)*, que retorna un document XML amb diverses dades de la previsió del temps en la latitud i longitud passades com a paràmetres, a partir del *diaInicial* i durant *numeroDeDiasdies*, en unitats britàniques o del Sistema Internacional, i amb les hores en format de 12 o de 24 hores.

La crida següent, per exemple, recupera la informació meteorològica d'Asheville, a Carolina del Nord (a 35,35 N, 82,33 W), des de l'11 de desembre de 2011 i durant els 7 dies següents, en unitats internacionals ("m") i en format de 24 hores.

```
result = proxy.NDFDgenByDay(35.35M, -82.33M,
      new DateTime(2011, 12, 11), "7", "m", "24 hourly");
```

Us demanen que escrigueu un client que es connecti al servei web i recuperi, per exemple, la previsió del temps a Asheville per als tres propers dies.

**9.** Es vol desenvolupar una eina CASE per a dibuixar i emmagatzemar diagrames de classes. Es vol dotar a aquesta eina d'un mecanisme de generació de codi per a diversos llenguatges de programació (Java, C++ i Visual Basic .NET). Es demana que, utilitzant el patró *Abstract Factory*, proposeu una solució per a generar codi a partir dels diagrames de classes.

**10.** Com s'explica en la secció dedicada a MDE, "una altra eina molt coneguda per a transformació de models és Atlas, desenvolupada per AtlanMod, un grup de recerca de l'Institut Nacional de Recerca en Informàtica i Automàtica de l'Escola de Mines de Nantes, a França. Atlas inclou el llenguatge de transformació ATL i és conforme a MOF (és a dir, defineix el seu mateix metamodel), encara que no és extensió d'UML". Podran els models ATL transformar-se a models UML 2.0 amb algun enfocament estàndard, com QVT?

**11.** Es vol desenvolupar una eina CASE que tindrà, segons el preu de venda, diferents funcionalitats, que s'oferiran en tres edicions: estàndard, executiva i empresarial.

L'edició estàndard permetrà la modelització de diagrames de classe, la generació de l'esquelet del codi (nom de la classe o interfície, camps i capçalera de les operacions) en llenguatge Java i la modelització de màquines d'estat per a les classes definides en el diagrama, però no permetrà modelitzar estats compostos.

L'edició executiva també permetrà la modelització de màquines d'estat amb estats compostos, la generació del codi que correspongui a les màquines d'estat i generar codi SQL a partir de diagrames de classes. A més de Java, aquesta eina permet generar codi per a altres llenguatges.

L'empresarial, a més de tot l'anterior, inclourà una funcionalitat per a generar bases de dades en diversos gestors a partir de diagrames de classes i una altra per a obtenir diagrames de classes a partir de codi escrit en els mateixos llenguatges que és capaç de generar. Us demanen que modelitzeu alguns aspectes d'aquesta eina seguint un enfocament de línies de producte programari.

## Exercicis d'autoavaluació

**1.** Expliqueu què significa donar a la reutilització un enfocament "proactiu".

**2.** Expliqueu com contribueixen les classes DAO a la reutilització.

**3.** Indiqueu en què es diferencia un cas d'ús *optional* d'un *alternative*.

**4.** Indiqueu les tecnologies principals utilitzades en la programació generativa.

**5.** Expliqueu com contribueix el manteniment d'un disseny arquitectònic multicapa d'un sistema si volem dotar a aquest sistema d'una interfície web.

# Solucionari

**Activitats**

**1.** L'enunciat evidencia l'existència explícita de quatre operacions sobre aquest TAD. A més, una operació implícita és la que permet conèixer el saldo i que necessitem per a descriure els axiomes:

- *Ingressar(import : real)*
- *Retirar(import : real)*
- *Transferir(import : real, destinació : Compte)*
- *Crear un compte*
- *Saldo() : real*

Una descripció possible d'aquest TAD és:

**a) Funcions**

- *ingressar: Compte x real → Compte*
- *retirar: Compte x real → Compte*
- *transferir: Compte x real x Compte Compte*
- *crear : Compte*
- *saldo : Compte → real*

**b) Axiomes** (siguin: x ∈ real; c, d ∈ Compte)

- *saldo(crear) = 0*
- *saldo(crear, ingressar(c, x)) = x*
- *saldo(retirar(c, x)) = saldo(c) - x*
- *saldo(transferir(c, x, d)) = saldo(c) – x - 0.01*x*

**c) Precondicions**

- *ingressar(c, x) requereix [x>0]*
- *retirar(c, x) requereix [x>0 ^ saldo(c)>= x]*
- *transferir(c, x, d) requereix [x>0 ^ saldo(c)>= 1.01*x*]

**2.** Si el saldo del client és la suma dels saldos de tots els productes associats, sense considerar les restriccions indicades en l'enunciat, un pseudocodi possible de l'operació *ClientBancari::getSaldo():double* podria ser:

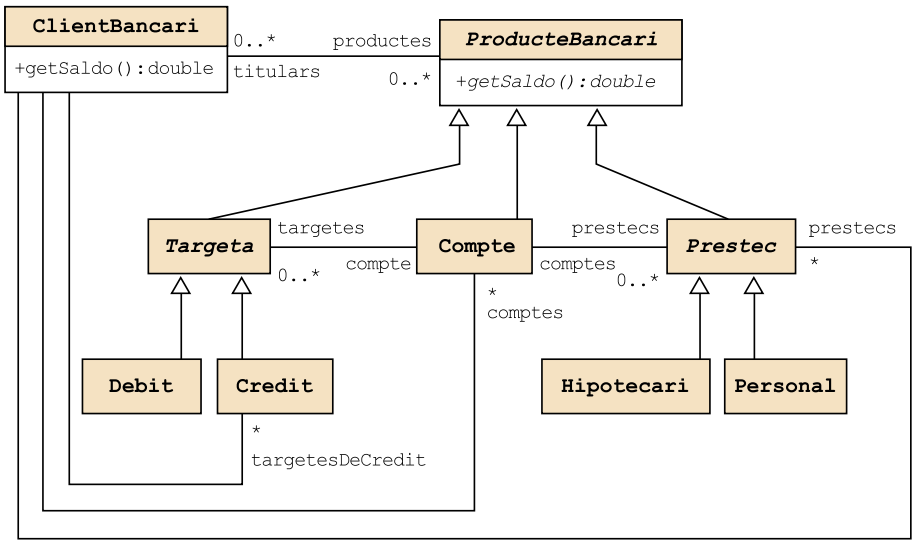
```
public double getSaldo() {
   double result=0.0;
   for (ProducteBancari p : this.productes) {
      result=result+p.getSaldo();
   }
   return result;
}
```

No obstant això, les restriccions imposades impedeixen donar aquesta implementació. Una solució possible recorreria la col·lecció de productes i interrogaria pel tipus de cadascun dels objectes continguts en la col·lecció (mitjançant una instrucció del tipus *instanceof* de Java), sumant, restant o no fent res, segons el cas:

```
public double getSaldo() {
   double result=0.0;
   for (ProducteBancari p : this.productes) {
      if (p instanceof Compte)
         result=result+p.getSaldo();
      else if (p instanceof Credit)
         result=result-p.getSaldo();
      else if (p instanceof Prestec)
         result=result-p.getSaldo();
   }
   return result;
}
```

Òbviament, el codi anterior n'impedeix completament la reutilització, ja que fa la seva implementació totalment dependent de la col·lecció de tipus existents en el sistema. Una solució possible implica la modificació del diagrama de classes per a fer que el client conegui di-

rectament tots els seus productes associats dels tipus *Credit*, *Compte* i *Prestec*, a més de conèixer-los per mitjà de l'associació amb *ProducteBancari*. És a dir:

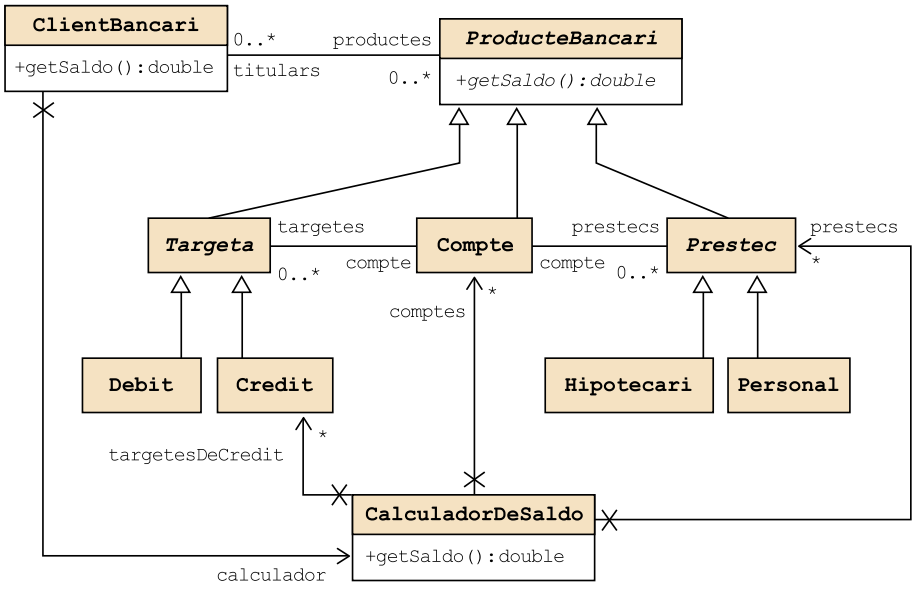



El codi de l'operació, en aquest cas, podria ser el següent:

```
public double getSaldo() {
   double result=0.0;
   for (Compte c : this.comptes)
         result=result+c.getSaldo();
   for (Credit t : this.targetesDeCredit)
         result=result-t.getSaldo();
   for (Prestec p : this.prestecs)
         result=result-p.getSaldo();
   return result;
}
```

La solució donada, no obstant això, augmenta la complexitat del sistema i en dificulta també la reutilització.

Una altra solució implica delegar la responsabilitat de calcular el saldo a una classe especialitzada que es dediqui només a això, la qual cosa es pot entendre com una aplicació del patró *Expert*: associem, a la classe *ClientBancari*, un *CalculadorDelSaldo* que recorre de la manera adequada els objectes corresponents:



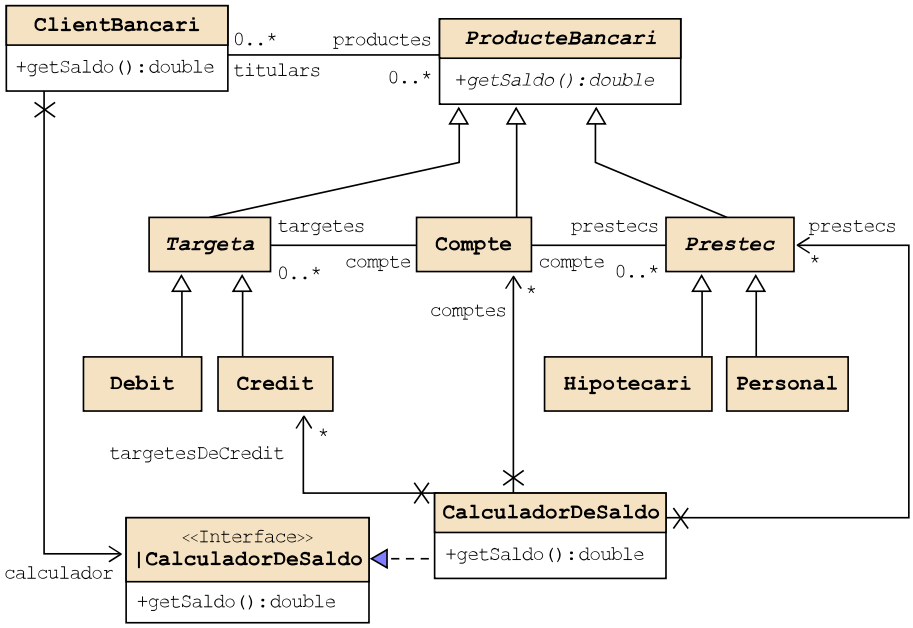

Ara, el codi de *getSaldo()* en *ClientBancari* consistiria, simplement, en una crida a l'operació *getSaldo()* de *CalculadorDelSaldo*:

```
public double getSaldo() {
   return this.calculador.getSaldo();
}
```

I, d'altra banda, el codi de *getSaldo()* en *CalculadorDelSaldo* seria:

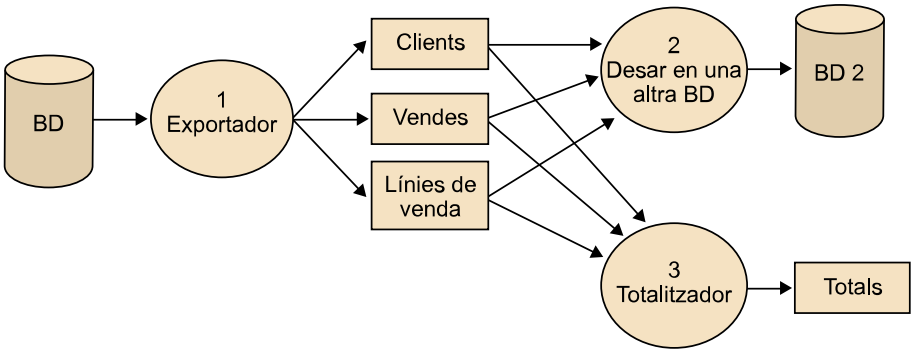
```
public double getSaldo() {
   double result=0.0;
   for (Compte c : this.comptes)
         result=result+c.getSaldo();
   for (Credit t : this.targetesDeCredit)
         result=result-t.getSaldo();
   for (Prestec p : this. prestecs)
         result=result-p.getSaldo();
   return result;
}
```

Aquesta solució, sense ser perfecta, deixa pràcticament sense tocar la classe *ClientBancari*, que és probablement important per al sistema i susceptible de ser reutilitzada, encara que obliga a l'addició del camp *calculador* a la classe *ClientBancari*, la qual cosa implica que aquesta classe queda acoblada a *CalculadorDelSaldo*. Per a disminuir l'acoblament es pot crear una interfície intermèdia per a la qual, no obstant això, l'últim codi continua essent vàlid: en la figura següent, el *ClientBancari* coneix un calculador genèric (la interfície *ICalculadorDeSaldo*), que està implementat pel *CalculadorDelSaldo* concret:

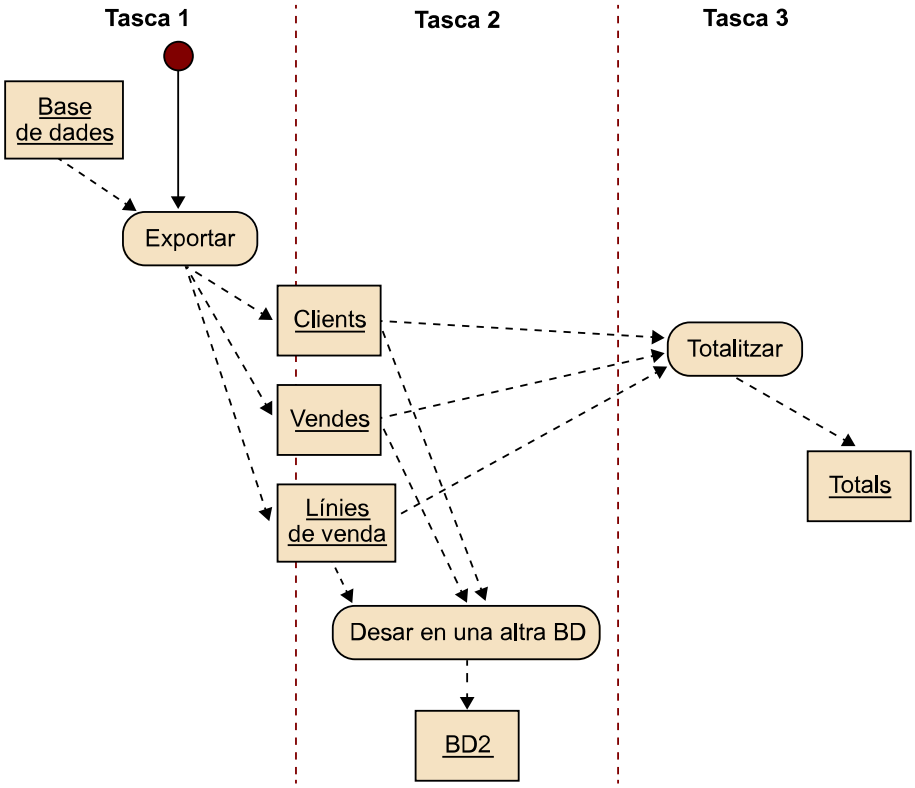



Aquesta última solució té l'avantatge que el *ClientBancari* no està directament acoblat al *CalculadorDelSaldo*: si, en algun moment, el saldo del client es vol calcular d'una altra manera, n'hi haurà prou de substituir la classe *CalculadorDelSaldo* per una altra versió que implementi la interfície *ICalculadorDeSaldo*.

**3.** Per a representar l'arquitectura de *pipes and filters* del sistema sol·licitat utilitzarem la notació dels diagrames de fluxos de dades. Hi ha tres processos, diversos magatzems de dades i dues entitats externes (la base de dades d'origen i la base de dades de destinació):

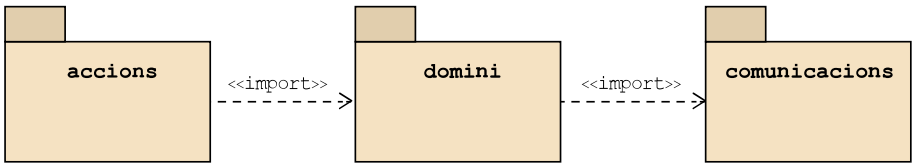



En la figura següent es mostra la mateixa solució utilitzant ara un diagrama d'activitat UML, en què cada *swimlane* representa una de les tres tasques:




**4.** Com s'ha vist en el desenvolupament del tema corresponent, es recomana crear una acció de Struts per a cada cas d'ús. A més, i a fi de comunicar amb el *Sistema central*, crearem també un subsistema de comunicacions.

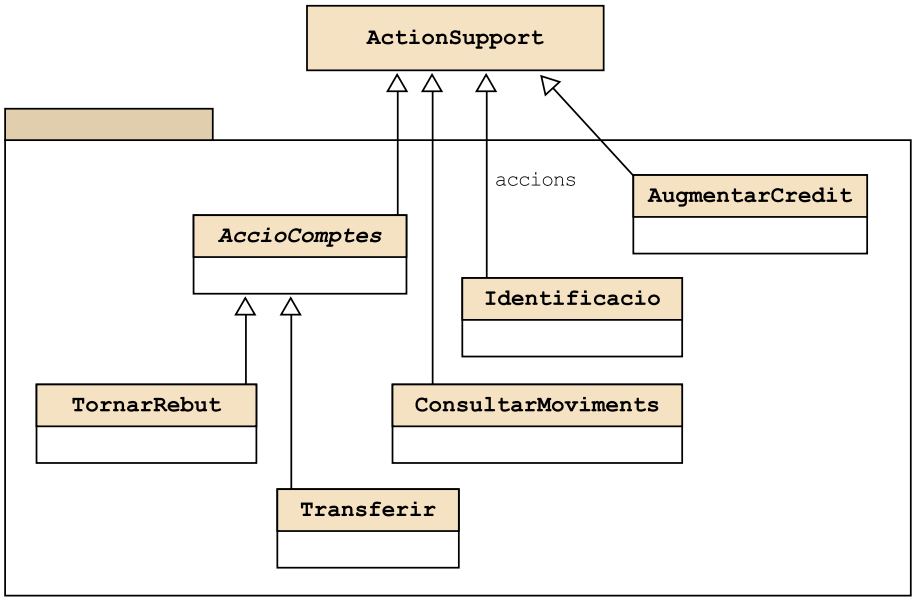
A molt alt nivell, el sistema web constarà, almenys, dels tres subsistemes següents:




En el paquet d'accions col·locarem les classes que implementen accions de Struts:

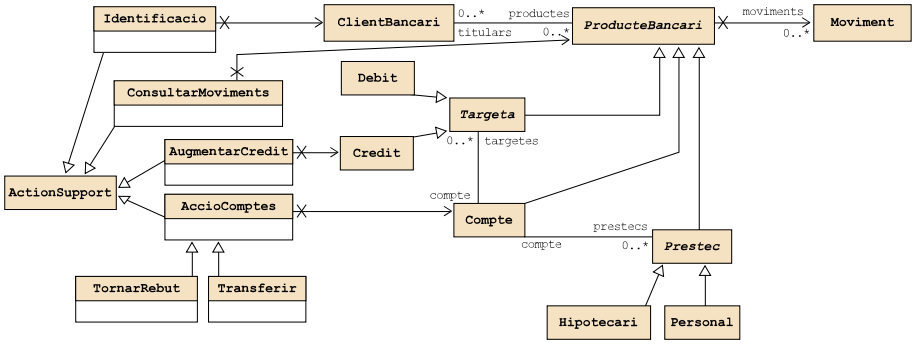
- Es requereix una acció per al cas d'ús *Identificar- se*.
- Es requereix una acció per al cas d'ús *Consultarmoviments*, que està inclòs en *Operar sobre comptes*, *Operar sobre targetes* i *Operar sobre préstecs*.
- Es requereix una acció per a *Transferir* i una altra per *Retornar rebut*. Aquests dos casos d'ús estan inclosos en *Operar sobre comptes*, per la qual cosa crearem una acció abstracta *AcciócComptes* que tingui la lògica concreta que pugui ser heretada en les accions *Transferir* i *TornarRebut*.

El disseny que podem donar al paquet d'accions és el següent:




Òbviament, és necessari connectar cada acció als objectes de domini adequats. En la figura següent:

- L'acció *Identificació* es connecta al *ClientBancari*, ja que s'assumeix que disposarà dels mecanismes necessaris per a identificar-se amb el *Sistema central*.
- *ConsultarMoviments* la connectem a la classe abstracta *ProducteBancari*, ja que del diagrama de casos d'ús sembla deduir-se que els moviments de tots els productes es consulten igual.
- *AugmentarCredit* s'associa solament al subtipus *CreditdeTargeta*.
- Connectem *AccióComptes* a *Compte*. Les dues especialitzacions d'aquesta acció reutilitzen l'estructura i la lògica definides en la superclasse.

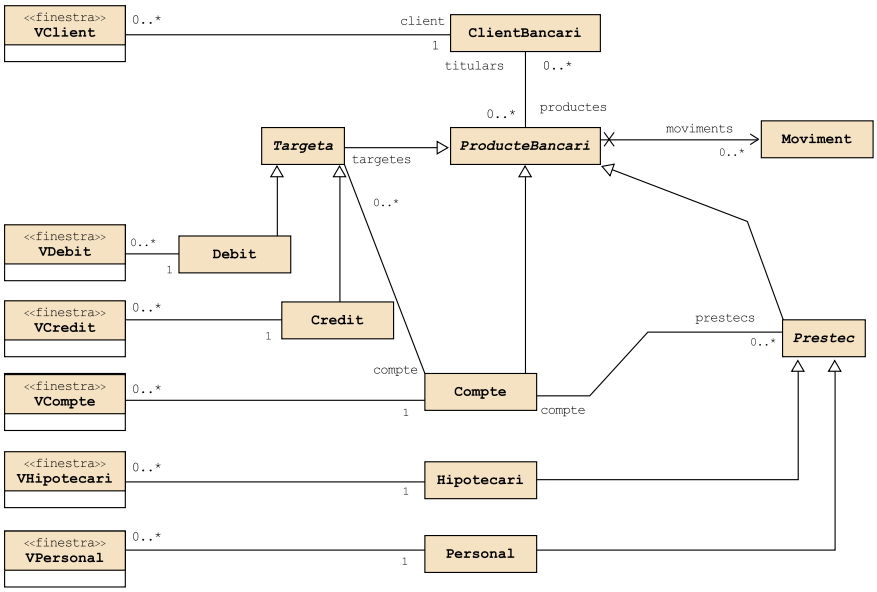



Finalment, en el paquet de comunicacions col·locarem un *Proxy*, que serà utilitzat pel sistema web per a connectar-se al *Sistema central*.

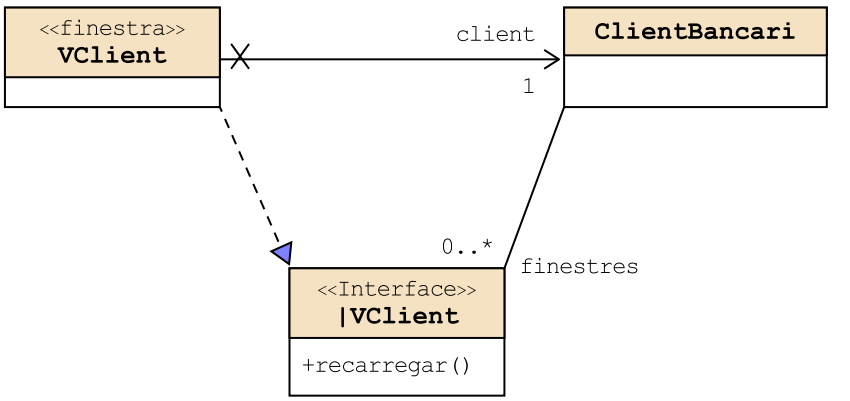
**5.** Per a solucionar aquest exercici, encapsularem totes les classes incloses en el diagrama de classes mostrat en l'enunciat de l'exercici anterior en un component. A més, inclourem en el component un patró façana que implementarà totes les operacions ofertes per la interfície oferta (*IGestioBancaria*). La façana encamina totes les crides a les operacions per mitjà del *ClientBancari*, únic objecte que coneix directament.

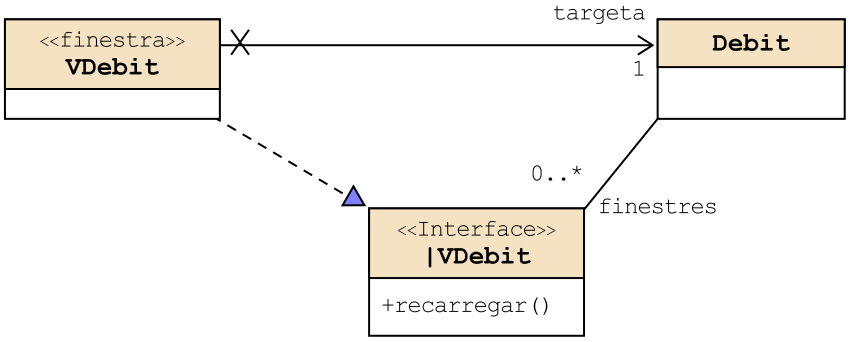
**6.** En aquest cas, ens podem limitar a tenir una acció genèrica que conegui una única instància del component i que pot ser abstracta. D'aquesta hereten tantes accions com funcionalitats és necessari servir. Cada acció dóna una implementació diferent al mètode *execute()* en funció de l'objectiu de la funcionalitat.

**7.** Una solució possible però, per descomptat, molt lluny de la ideal i que a més no utilitza patrons, implica acoblar les classes de domini a les de presentació: cada finestra observa un objecte de domini, i cada objecte de domini coneix totes les finestres que "l'estan mirant". Quan canvia l'estat de l'objecte de domini, ell mateix (la qual cosa seria conforme amb el patró *Expert*) actualitza totes les seves vistes. Aquesta solució, no obstant això, acobla prohibitivament el domini a la presentació i dóna a les classes de domini la responsabilitat d'actualitzar les seves vistes, la qual cosa en disminueix la cohesió.

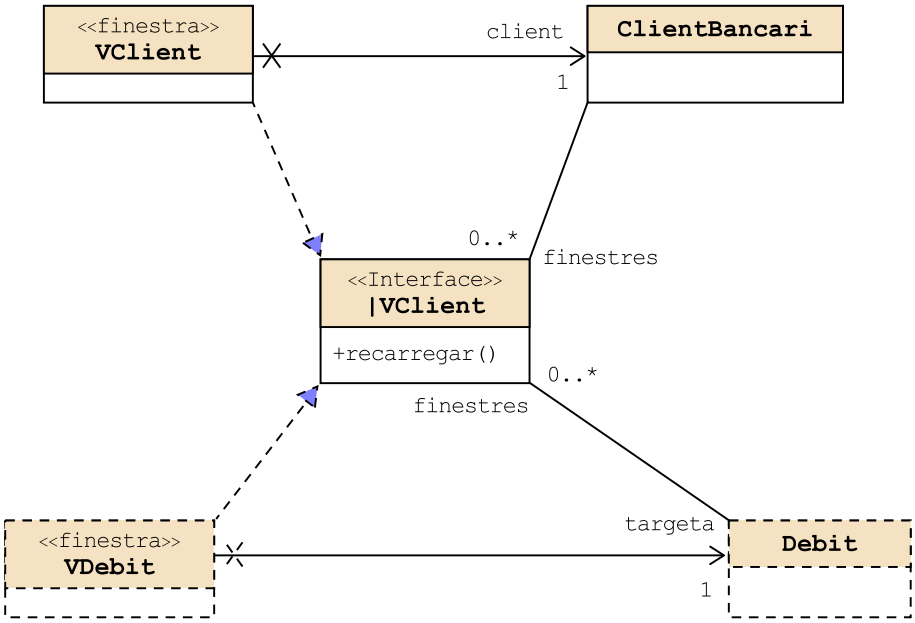



Per millorar el disseny, es poden crear interfícies que es poden situar en una capa intermèdia de serveis que desacoblin el domini de la presentació. Per no sobrecarregar massa el diagrama, la figura següent mostra aquesta solució per al *ClientBancari*, la targeta de *Debit* i les seves finestres corresponents: quan l'objecte de domini canvia, executa l'operació *recarregar()* sobre tots els objectes de tipus *IFClient* o *IFDebit* que coneix. Assumim que l'operació *recarregar()* en les finestres recarrega i mostra l'estat que interessi, en aquest exemple, del client o la targeta al qual la finestra està observant.

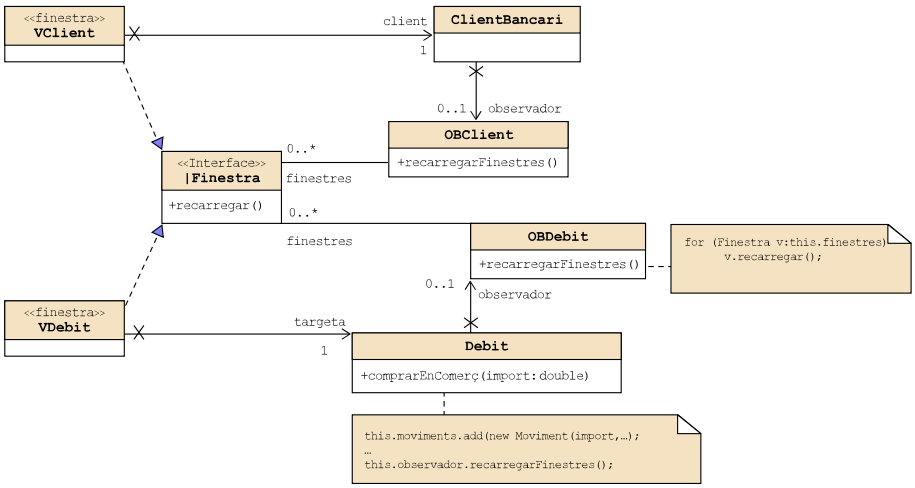



Amb aquesta solució es disminueix l'acoblamiento del domini respecte de la presentació: es té ara un acoblament "menys dolent", perquè el domini coneix només objectes genèrics (interfícies) en lloc d'objectes concrets (classes de tipus finestra). Si ens anem a l'adopció d'aquesta solució per a tot el sistema, podem crear una única interfície *IFinestra* que declari l'operació *recarregar()* i que sigui implementada per totes les finestres de la capa de presentació. Continuant amb la particularització per a *ClientBancari* i *Debit*:

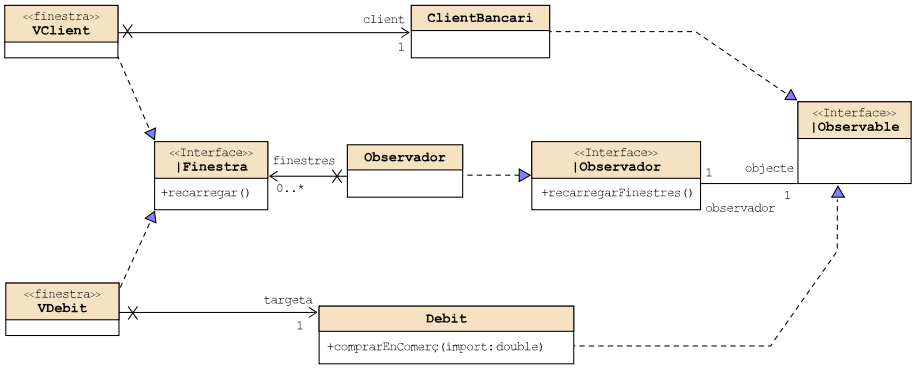



De totes maneres, la cohesió del domini continua essent baixa amb la solució anterior, perquè els objectes de domini continuen tenint la responsabilitat d'indicar a les finestres que es recarreguin. En el diagrama següent es delega la responsabilitat de recarregar a un observador que associem a cada objecte de domini: quan canvia l'estat d'un d'aquests (per exemple, perquè es compra en un comerç amb una targeta de dèbit), l'objecte executa les operacions de domini que corresponguin i, després, diu a l'observador que recarregui les finestres que l'estan mirant; l'observador, que les coneix totes, executa l'operació *recarregar* sobre cadascuna.

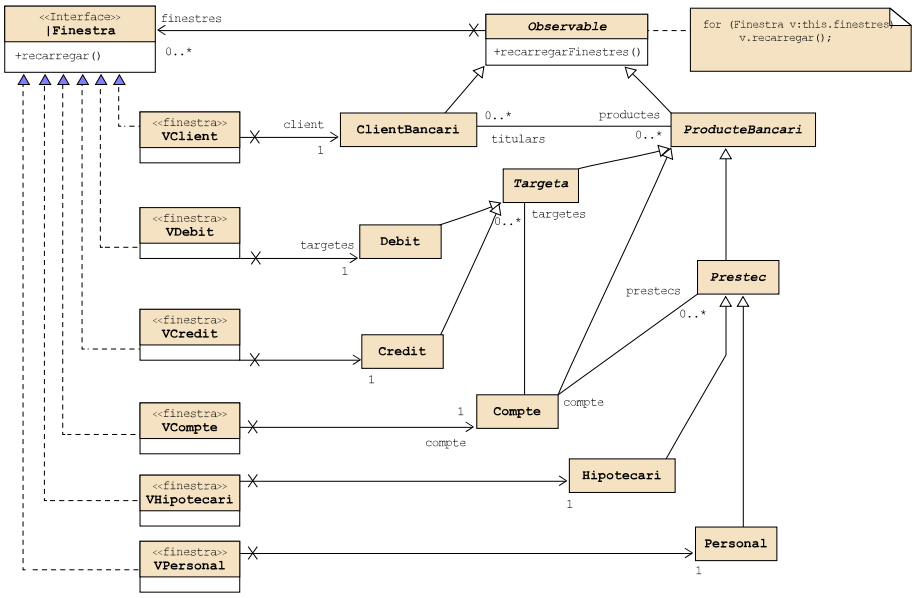



Ja que cada classe de domini tindrà un observador, podem generalitzar tots els objectes de domini a un supertipus *IObservable*, i crear també un observador genèric que conegui objectes *IObservables*.

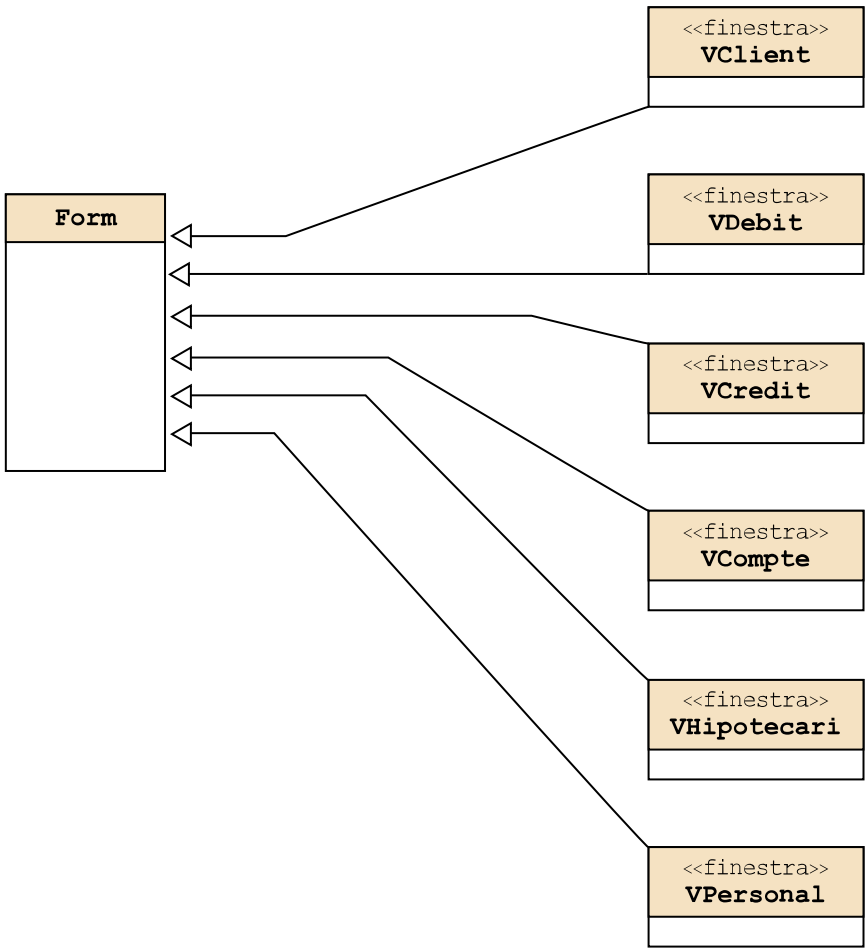
En el disseny següent, cada objecte de domini és un objecte *IObservable*; cada objecte observable coneix un *IObservador*, i cada *IObservador* observa un objecte observable:



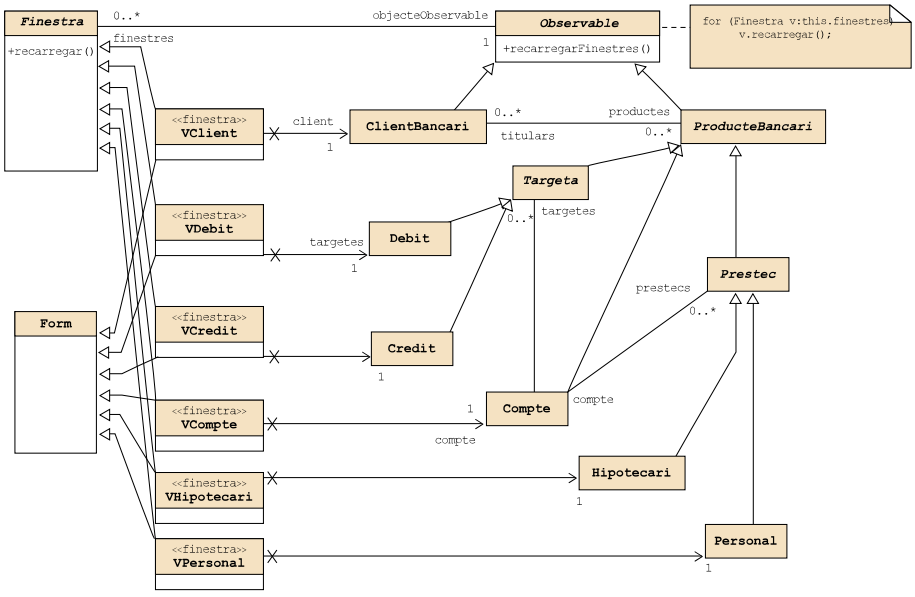

Una solució addicional està determinada per l'estructura d'herència que hi ha en el domini: podem fer que tant el *ClientBancari* com el *ProducteBancari* siguin especialitzacions d'una classe abstracta *Observable* que, no obstant això, té una operació concreta *recarregarFinestres()*. Aquesta classe abstracta coneix una col·lecció d'objectes de tipus *IFinestra* (una interfície), que és implementada per totes les finestres: quan canvia l'estat d'un objecte (es compra amb una targeta de *Dèbit*, per exemple), s'executen les operacions de domini que corresponguin i, després, es crida l'operació heretada *recarregarFinestres()*, la implementació de la qual s'hereta íntegrament de la superclasse.




Suposem, d'altra banda, que les finestres són especialitzacions d'un supertipus *Form* inclòs en un *framework* de creació d'interfícies d'usuari:



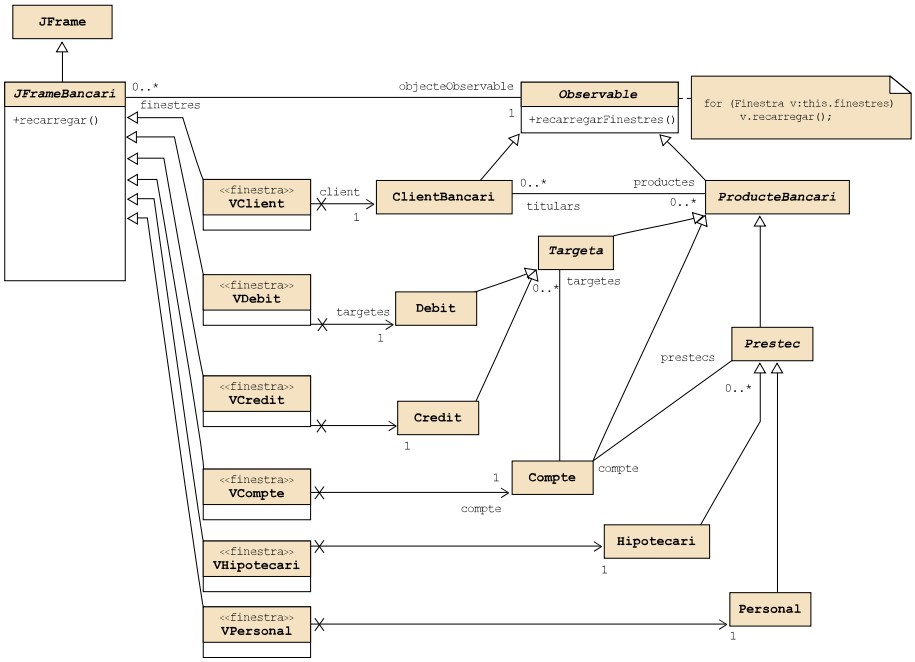

Suposem, a més, que el llenguatge de programació en el qual implementarem el sistema suporta herència múltiple. En aquest cas, la interfície *IFinestra* que hem mostrat en l'última solució es podria substituir per una superclasse abstracta *Finestra* que inclogui l'operació concreta *recarregar()*. Cada finestra concreta (*FCompte*, *FDebit*, etc.) hereta tant de *Form* com de *Finestra*:




En aquest últim disseny, cada instància de *Finestra* coneix dues vegades la mateixa *Instància* de l'objecte de domini: una pel camp *ObjecteObservable* que està heretant de *Finestra*, i una altra per l'associació que relaciona a la *Finestra* mateixa amb l'objecte de domini que correspongui (per exemple: *FCompte* coneix una instància de tipus *Compte* mitjançant el seu camp *Compte*, però també coneix la mateixa instància mitjançant el Camp *objecteObservable* que hereta de

*Finestra*). La referència heretada s'utilitza únicament per a recarregar la finestra; la referència pròpia, per a cridar els mètodes de negoci corresponents a l'objecte. El programador haurà de garantir que totes dues referències apunten al mateix objecte.

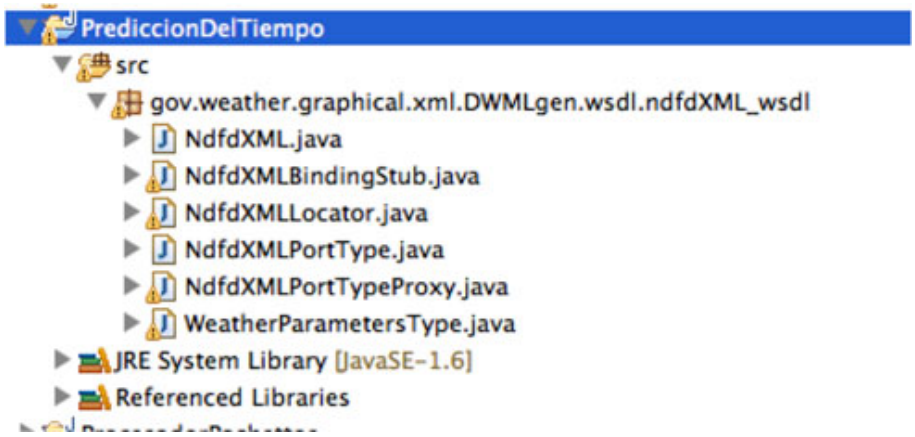
Suposem ara que el llenguatge de programació no admet herència múltiple (com podria ser el cas de Java). En aquest cas, totes les finestres seran, probablement, especialitzacions d'alguna classe inclosa en algun *framework* de disseny d'interfícies d'usuari (*FCompte*, *FDebit*, etc., poden ser especialitzacions de la classe *JFrame* inclosa en Swing). Si no volem utilitzar interfícies (com en una solució anterior), podem crear una classe abstracta *JFrameBancari* que hereti de *JFrame*, que connecti l'objecte *Observable*, de la qual heretin les finestres específiques de gestió i que doni implementació a l'operació *recarregar()*. Com en l'última solució, en la qual hem utilitzat herència múltiple, cada finestra concreta posseirà dues referències a cada instància de domini: una heretada del *JFrameBancari* i una altra a la qual apunta cada finestra concreta. Igual que abans, el programador ha de garantir que tots dos camps apuntin a la mateixa referència.




Ara que s'han presentat múltiples solucions per al mateix problema, es demana al lector que pensi i raoni sobre els diferents valors de cohesió, acoblament o reutilització que aporta cadascuna.

**8.** Per a resoldre aquesta activitat, hem d'utilitzar l'assistent del nostre entorn de desenvolupament per a connectar al document WSDL publicat en l'adreça especificada en l'enunciat del problema.

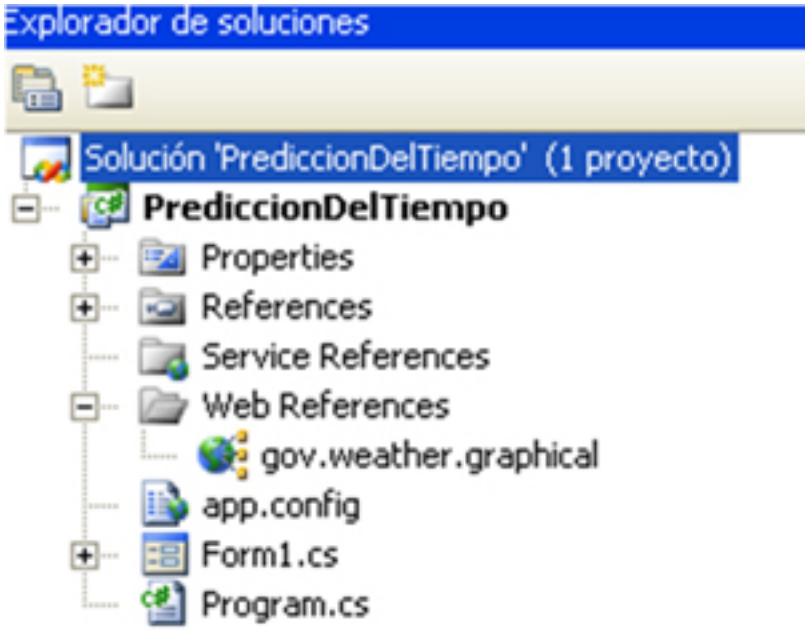
En l'entorn Eclipse, per exemple, podem crear un projecte anomenat PrediccioDelTemps i crear, mitjançant l'assistent, les classes necessàries per a accedir al servei:



A continuació, escrivim el codi per a fer la crida sol·licitada:

```
public static void main(String[] args) {
   BigDecimal latitud=new BigDecimal("+35.35");
   BigDecimal longitud=new BigDecimal("-82.33");
   GregorianCalendar gc=new GregorianCalendar(2011, 11, 11);
   Date dataDInici=new Date(gc.getTimeInMillis());
   BigInteger nombreDeDies=new BigInteger("7");
   String unitat="m";
   String format="24 hourly";
   try {
      NdfdXMLPortTypeProxy proxy = new NdfdXMLPortTypeProxy();
      String result = proxy.NDFDgenByDay(latitud, longitud, dataDInici,
                     nombreDeDies, unitat, format);
      System.out.println(result);
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

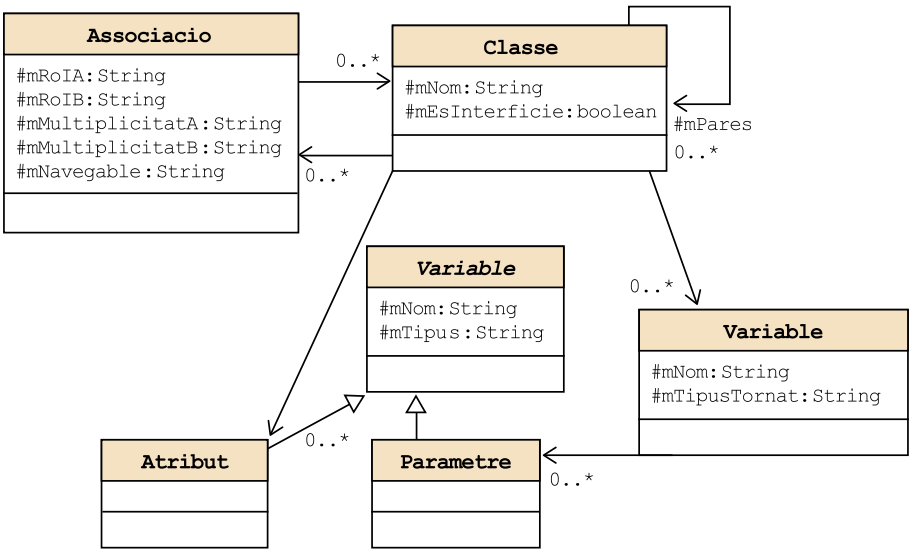
En el Microsoft Visual Studio, la manera de procedir és molt semblant: s'agrega la referencia web...



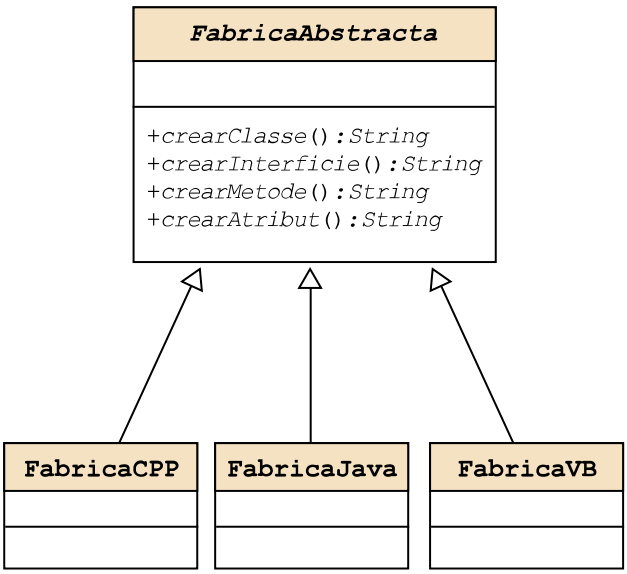

...i s'escriu el codi:

```
private void button1_Click(object sender, EventArgs e)
{
   gov.weather.graphical.ndfdXML proxy = new
   WeatherForecast.gov.weather.graphical.ndfdXML();
   string result = proxy.NDFDgenByDay(35.35M, -82.33M,
      new DateTime(2011, 12, 11), "7", "m", "24 hourly");
   Console.WriteLine(result);
}
```

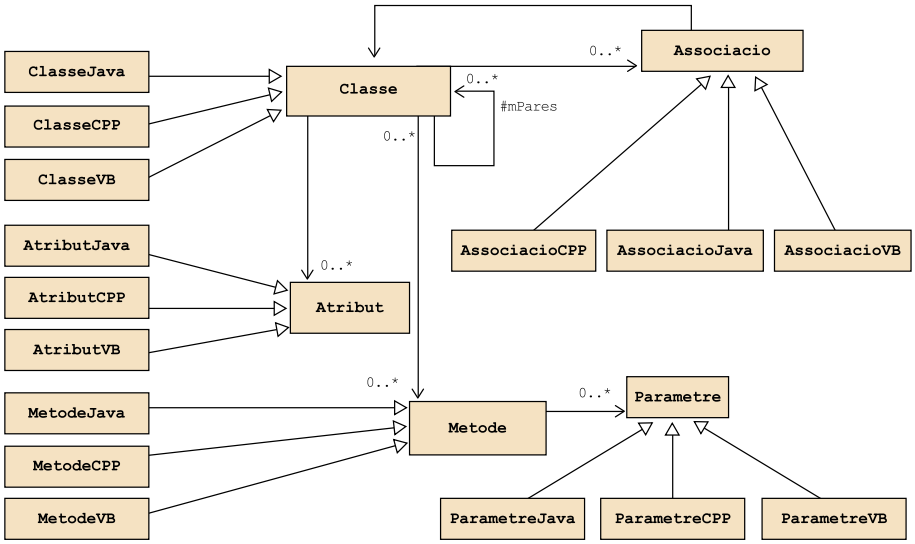
**9.** Aquest exercici combina conceptes del tema anterior (el patró *Abstract Factory*) amb la idea dels metamodels descrits en aquest. Idealment, el metamodel que hauria de suportar aquesta eina per a representar diagrames de classe hauria de ser del d'UML 2.0. No obstant això, i per simplificar, suposarem que utilitzem el metamodel reduït següent: una classe hereta, es relaciona amb altres classes mitjançant associacions, i té atributs i mètodes, que poden tenir paràmetres:



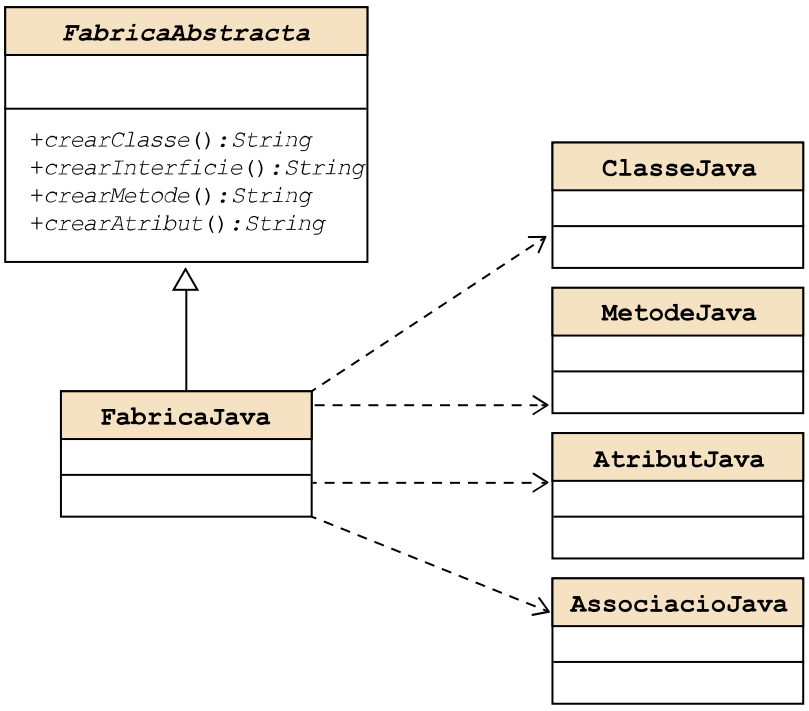

ja que es vol generar codi per a tres llenguatges diferents, crearem tres fàbriques concretes, que seran especialitzacions d'una abstracta. En la fàbrica abstracta declararem les operacions necessàries per a crear els tipus d'elements el codi dels quals es vol generar:



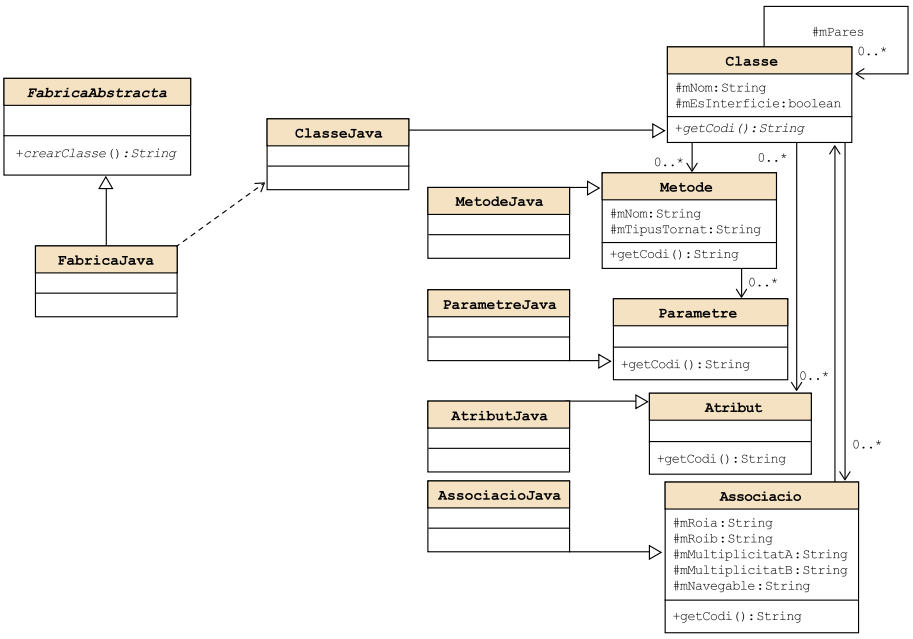

D'altra banda, especialitzem les classes definides en el metamodel creant, per exemple, les classes *ClasseJava*, *ClasseCPP* i *ClasseVB* que, respectivament, representen classes Java, C++ i de Visual Basic:




Finalment, enllacem cada fàbrica concreta amb els tipus d'elements concrets. Per exemple, fent que la fàbrica de Java creï classes, mètodes, atributs i associacions en Java:




En aquesta altra solució es combina el patró *Fàbrica Abstracta* amb el *Builder*: com que una classe és un agregat d'atributs i mètodes, es pot fer que cada fàbrica concreta creï solament una instància concreta del seu subtipus corresponent: per exemple, que *FabricaJava* creï objectes de tipus *ClasseJava*. Llavors, considerem que *ClasseJava* és el *Builder* i la dotem de la capacitat de generar el codi complet de la classe mitjançant un mètode *getCodi*, que és abstracte en *Classe* però concret en *ClasseJava*, *ClasseCPP* i *ClasseVB*:




**10.** La resposta a aquesta pregunta és un clar i contundent sí: QVT és un llenguatge de transformació definit per a models MOF. Com que tant el metamodel d'ATL com el metamodel d'UML 2.0 són conformes a MOF, qualsevol model ATL (que no és sinó una instància del seu metamodel) podrà ser transformat a una instància del metamodel d'UML 2.0 (és a dir, a un model). Recordeu que un model (nivell 2 de MOF) és una instància del seu metamodel corresponent (nivell 1 de MOF).

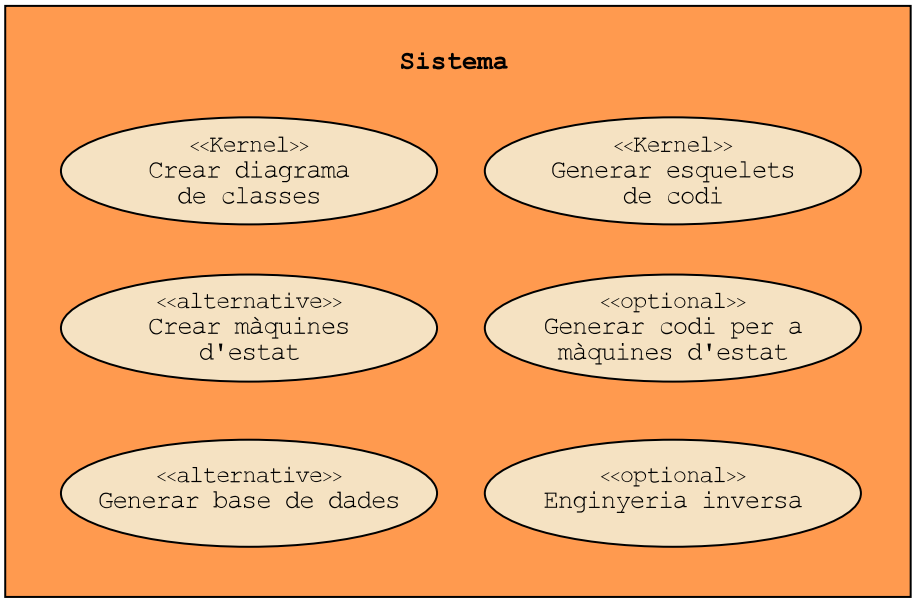
Una altra cosa és la possible dificultat inherent a la determinació de les equivalències entre els elements de cada metamodel, que pot fer molt complexes les transformacions.

**11.** L'enunciat queda suficientment obert (ens diu que modelitzem "alguns aspectes d'aquesta eina") perquè practiquem diferents aspectes quant a línies de producte.

La taula següent resumeix les característiques de cada edició:

| Edició | Diagrames de classe | Generació esquelets | Màquines d'estat | Generació codi m. estat | Generació de BBDD | Enginyeria inversa |
|---|---|---|---|---|---|---|
| Estàndard | Sí | Sí | Parcialment | | | |
| Executiva | Sí | Sí | Sí | Sí | Parcialment (només SQL) | |
| Empresarial | Sí | Sí | Sí | Sí | Sí | Sí |

Un conjunt possible de casos d'ús per a aquest sistema podria ser el següent:




- *Crear diagrames de classes* i *Generar esquelets de codi* són casos d'ús *kernel* perquè són funcionalitats que s'inclouran en totes les edicions i, a més, funcionaran en totes exactament de la mateixa manera.
- La *Creació de màquines d'estat* està inclosa en totes les edicions, però la implementació serà diferent en l'edició estàndard. Per això, s'ha decidit estereotipar aquest cas d'ús com a *alternative*. La *Generació de base de dades* només pertany a les edicions executiva i empresarial i, a més, és una mica diferent en una i en l'altra, per la qual cosa també s'ha usat el mateix estereotip.
- Finalment, *Enginyeria inversa* i *Generar codi per a màquines d'estat* són *optional* perquè s'inclouen, sempre de la mateixa manera, només en dos dels tres productes.

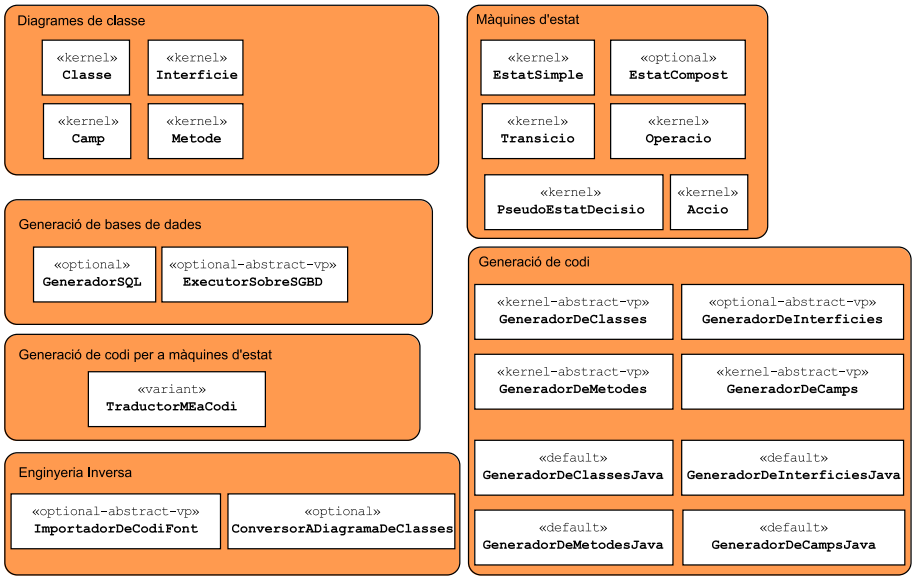
Si saltem a l'etapa de disseny, i tenint en compte els coneixements que tenim sobre metamodelització (exemples i activitats), podem utilitzar, a manera d'exemple i simplificant molt, les classes següents:

- Per a la creació de diagrames de classes: *Classe*, *Camp*, *Mètode*, *Interfície*.
- Per a la creació de màquines d'estat: *Estat simple*, *Estat compost*, *Pseudoestat de decisió*, *Transició*, *Operació*, *Acció*.
- Per a l'enginyeria inversa tindrem un *Importador de codi* i un *Convertidor* (que el transformarà en el diagrama de classes).
- Per a la generació d'esquelets de codi, un generador per a cada tipus d'element dels diagrames de classe.
- Per a la generació de codi per a màquines d'estat, un *Traductor*.
- Per a la generació de bases de dades, utilitzarem un *Generador d'SQL* i una altra classe *Executor*, que executa les sentències SQL sobre el gestor.

En aquesta figura es mostren les classes, agrupades segons la funcionalitat per a la qual estan pensades.

- Totes les classes del grup *Diagrames de classe* són *kernel* perquè estan incloses en les tres edicions de l'eina. A més, s'hi inclouran exactament de la mateixa manera.
- En *Generació de bases de dades*, *GeneradorSQL* és *optional* perquè està inclosa només en dues de les tres edicions i, a més, la implementació serà la mateixa en totes dues. L'*ExecutorSobreSGB* és *optional-abstract-vp* perquè és opcional (no està inclosa en tots els productes) però, a més, la implementació dependrà del gestor concret sobre el qual s'executin les sentències de creació de taula.
- És discutible (com, realment, qualsevol solució que es doni a un problema en enginyeria del programari) l'adequació de l'estereotip variant amb què s'ha anotat *TraductorMEaCodi* del grup *Generació de codi* per a màquines d'estat. Se li ha aplicat perquè és una classe que no estarà en tots els productes però, a més, té dues implementacions diferents segons es generi codi per a màquines d'estats simples o compostos. No obstant això, l'enunciat és suficientment obert per a utilitzar altres estereotips.
- En enginyeria inversa, l'*ImportadorDeCodiFont* és *optional-abstract-vp* perquè estarà inclòs solament en alguns productes i, a més, amb diferents implementacions segons el llenguatge de programació que s'hagi de llegir. Aquesta classe obtindrà un diagrama de classes (del grup *Diagrames de classe*, amb les quatre classes *kernel*) que serà processat per un *ConversorADiagramesDeClasses*, que és *optional*.

- En *Màquines d'estat* totes les classes són *kernel* excepte *EstatCompost*, que no està inclosa en l'edició estàndard.
- En *Generació de codi* hi ha diverses classes *kernel-abstract-vp* perquè totes les edicions generen l'esquelet de codi, però hi ha diversos llenguatges de programació. Il·lustrem el fet que l'edició estàndard genera en llenguatge Java mitjançant les classes estereotipades amb *default*.




**Exercicis d'autoavaluació**

**1.** L'enfocament "proactiu" representa el fet que la reutilització de programari es concep des del principi del desenvolupament, i no després. Així, el grau de reutilització dels artefactes programari és, des del començament del projecte, un requisit no funcional més.

**2.** Les classes de domini persistents deleguen a les classes DAO la responsabilitat de mantenir-les actualitzades en el sistema d'emmagatzematge secundari (fitxers o bases de dades). Així, les classes de domini, que són les que tenen la major part de la complexitat del sistema, queden desacoblades del sistema d'emmagatzematge, i tenen més potencial de reutilització.

**3.** Un cas d'ús *optional* està present en diversos productes d'una línia de producte programari, i en tots té la mateixa implementació. Un cas *alternative* també està present només en alguns productes però, no obstant això, en cadascun pot tenir una implementació diferent.

**4.** S'utilitzen fonamentalment: llenguatges de domini específic, modelització de característiques, programació orientada a aspectes i programació genèrica.

**5.** Un bon disseny arquitectònic multicapa manté la lògica de negoci ben separada de la capa de presentació, amb la qual cosa la lògica que s'hi trobi es limitarà, pràcticament, a la gestió dels esdeveniments de l'usuari, validació de dades, etc. Mantenir aquest desacoblament facilita la reutilització de la capa de negoci o domini, que es podrà connectar sense dificultats especials (excepte les purament tecnològiques) a la nova capa de presentació.

# Glossari

**acoblament** *m* Grau en el qual estan relacionats un element d'un conjunt amb els altres elements. En el cas d'un sistema programari, l'acoblament és una mesura del nombre i qualitat de les relacions existents entre classes, subsistemes, etc. Un element molt acoblat a d'altres es veurà molt afectat pels canvis en aquests elements. L'acoblament es pot mesurar quantitativament amb mètriques com *Coupling Between Objects, Depth of Inheritance of Tree* o *Number of Childs*.

**cohesió** *f* En enginyeria del programari, la cohesió dóna una mesura del grau en què estan relacionats els membres d'una classe o les classes pertanyents a un subsistema. Una classe amb baixa cohesió fa diverses coses diferents que no tenen relació entre si, la qual cosa porta a classes més difícils d'entendre, de reutilitzar i de mantenir. Es pot mesurar quantitativament amb la mètrica *Lack of Cohesion in Methods*.

**CBSE (*component-based software engineering*)** *f* Subdisciplina de l'enginyeria del programari que s'ocupa de la construcció de sistemes a partir de components reutilitzables.

**DAO (*data access object*)** *m* Objecte que s'encarrega de gestionar la persistència d'altres objectes de domini.

**DLL (*dynamic-link library*)** *f* Biblioteca de classes o funcions que es carrega en temps d'execució: això és, en el moment en què el programa en execució la necessita.

**DSL (*domain-specific language*)** *m* Llenguatge de modelització que proporciona els conceptes, les notacions i els mecanismes propis d'un domini determinat, semblants als que manegen els experts d'aquest domini, i que permet expressar els models del sistema a un nivell d'abstracció adequat.

**enginyeria de domini** *f* Un dels dos nivells de desenvolupament de programari en línies de producte i en programació generativa. En l'enginyeria de domini, es construeixen els elements comuns a tots els productes.

**enginyeria de producte** *f* Un dels dos nivells de desenvolupament de programari usats en línies de producte i en programació generativa. En l'enginyeria de producte, es construeix cadascun dels productes programari a partir de les pròpies característiques del producte i del que s'ha construït en el nivell d'enginyeria de domini.

**enllaç dinàmic** *m* Vegeu **DLL**.

**enllaç estàtic** *m* Biblioteca de classes o funcions que es carrega en temps de compilació per a produir el programa executable. Vegeu també, per comparar, *DLL*.

**fabricació industrial del programari** *f* Model de construcció de programari que s'empra a les anomenades *fàbriques de programari*, en les quals un conjunt nombrós d'enginyers desenvolupa aplicacions grans i complexes, utilitzant de manera més o menys rigorosa metodologies de desenvolupament, tècniques de reutilització, etc.

***feature*** *f* Característica important (des del punt de vista funcional) d'un producte de programari. L'anàlisi de *features* s'utilitza en paradigmes en què la reutilització és una activitat important.

***feature model*** *m* Model en què es representen les *features* o característiques d'un sistema de programari.

**genericitat** *f* Capacitat que tenen alguns llenguatges de programació per a definir i combinar classes parametritzant tipus de dades, que es declaren en temps d'escriptura del codi, però que es resolen i instancien en temps de compilació.

**informàtica en núvol (*cloud computing*)** *f* Paradigma de distribució de programari i compartició de recursos pel qual s'ofereixen multitud de serveis (execució d'aplicacions, serveis d'emmagatzematge, eines col·laboratives, etc.) als usuaris, que els utilitzen de manera totalment independent de la seva ubicació i, en general, sense gaire requisits de maquinari.

**LPP (línia de producte de programari)** *f* Mètode de producció de programari en el qual es fa un ús molt intensiu de la reutilització i es treballa a dos nivells de desenvolupament: enginyeria de domini i enginyeria de producte.

**MDE (*model-driven engineering*)** *m* Paradigma dins de l'enginyeria del programari que advoca per l'ús dels models i les transformacions entre ells com a peces clau per a dirigir totes les activitats relacionades amb l'enginyeria del programari.

**metamodel** *m* Model que especifica els conceptes d'un llenguatge, les relacions entre ells i les regles estructurals que restringeixen els possibles elements dels models vàlids, i també aquelles combinacions entre elements que respecten les regles semàntiques del domini.

**MOF2Text** *m* Especificació de l'OMG per a la transformació de models a text.

**MOFScript** *m* Implementació de MOF2Text distribuïda amb llicència GNU, integrable en l'entorn Eclipse i desenvolupada per la comunitat de desenvolupadors SINTEF amb el patrocini de la Unió Europea.

**OCL (*object constraint language*)** *m* Llenguatge que complementa UML i que s'utilitza principalment per a escriure restriccions sobre sistemes orientats a objecte, de manera que s'elimini l'ambigüitat inherent al llenguatge natural.

**OMG (*object management group*)** *m* Consorci per a l'estandardització de llenguatges, models i sistemes per al desenvolupament d'aplicacions distribuïdes i la seva interoperabilitat.

**perfil** *m* Extensió d'un subconjunt d'UML orientat a un domini, utilitzant estereotips, valors etiquetats i restriccions.

**programació generativa** *f* Mètode de producció de programari que tracta del disseny i la implementació de programari reutilitzable per a generar famílies de sistemes d'un domini, i no sistemes independents. Es basa de manera important en llenguatges específics de domini, aspectes i programació genèrica.

# Bibliografia


### Bibliografia bàsica

**Czarnecki, K.; Eisenecker, U.** (2000). *Generative Programming: Methods, Tools and Applications*. Addison Wesley.

**Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.** (2002). *Patrones de diseño: elementos de software orientado a objetos reutilizables*. Addison Wesley.

**Gomaa, H.** (2005). *Designing Software Product Lines with UML. From use cases to pattern-based software architectures*. Addison Wesley.

**Kiselev, I.** (2003). *Aspect-Oriented Programming with AspectJ*. Editorial SAMS.

**Larman, C.** (2002). *UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Prentice Hall.

**Piattini M.; Garzás, J.** (2010). *Fábricas de software: experiencias, tecnologías y organización*. Ra-Ma: Madrid.

**Szyperski, C.** (2002). *Component Software: Beyond Object-Oriented Programming* (2a. ed.). Boston: Addison Wesley.

### Bibliografia complementària

**Clements, P.; Northrop, L.** (2001). *Software Product Lines: Practices and Patterns*. Addison Wesley.

**Díaz, Ó.; Trujillo, S.** (2010). "Líneas de Producto Software". A: Piattini; Garzás (eds.). *Fábricas de Software*. Madrid: Editorial Ra-Ma.

**Griss, M. L.** (1993). "Software reuse: from library to factory". *IBM Software Journal* (vol. 32, núm. 4, pàg. 548-566).

**Jacobson, I.; Christeson, M.; Jonsson, P.; Övergaard, G.** (1992). *Object-oriented software engineering. A use case driven approach*. ACM Press.

**Krueger, C. W.** (1992). "Software Reuse". *ACM Computing Surveys* (vol. 24, pàg. 131-183).

**Kung, H.-J.; Hsu, C.** (1998). "Software Maintenance Life Cycle Model". *International Conference on Software Maintenance* (pàg. 113-121). IEEE Computer Society.

**May, R. M.** (1974). "Biological Populations with Nonoverlapping Generations: Stable Points, Stable Cycles, and Chaos". *Science* (vol. 186, núm. 4164, pàg. 645-647).

**Meyer, B.** (1997). *Construcción de software orientado a objetos* (2a. ed.). Prentice Hall: Santa Bárbara.

**Piattini M.; Calvo-Manzano, J. A.; Cervera, J.; Fernández, L.** (1996). *Análisis y diseño detallado de Aplicaciones informáticas de gestión*. Madrid: Editorial Ra-Ma.

**Polo, M.; García-Rodríguez, I.; Piattini, M.** (2007). "An MDA-based approach for database reengineering". *Journal of Software Maintenance & Evolution: Research and Practice* (vol. 19, núm. 6, pàg. 383-417).

**Pressman, R. S.** (2009). *Ingeniería del software: un enfoque práctico*. Editorial McGraw-Hill.

**Sommerville, I.** (1992). *Software engineering* (8a. ed.). Addison Wesley.