

A.- El llenguatge de modelat UML	4
A.1- Orígens UML	4
A.2- UML. Què és i que no és	5
A.3- OCL	5
A.4- Models i Diagrames	5
A.5- Diagrames coberts pel curs	7
A.6- Seqüència d'utilització habitual dels diagrames UML	9
B.- Modelat Estructural	10
B.1- Diagrama de classes	10
B.1.1.- Aspecte general	10
B.1.2.- Les fases del diagrama de classes	10
B.1.3.- Classes	11
a) Objecte o instància	11
B.1.4.- Control d'accés	12
B.1.5.- Paquetització o espais de noms	12
B.1.6.- Atributs	13
a) Valor per defecte:	13
b) Valor de només lectura	13
c) Multiplicitat	13
d) Concepte d'Atribut derivat	13
e) Atributs estàtics	14
B.1.7.- Mètodes	14
a) Mètodes estàtics	14
B.1.8.- Propietats	14
B.1.9.- Tipus de relacions entre classes	16
a) Dependència	16
b) Associació	16
c) Associacions i navegabilitat	17
d) Consells d'ús d'Atributs i Associacions !	19
e) Agregació i composició	20
f) Associació amb classe associativa (≡ Chen relació amb atributs...)	21
g) Associació múltiple (≡ Chen relacions n-àries)	22
h) Associacions Reflexives	22
i) Associació Qualificada	22
B.1.10.- Generalització / especialització (Herència)	23
a) Herència simple	23
b) Jerarquies, classes abstractes i finals	23
B.1.11.- Interfícies	24
a) Herència múltiple	25
b) Polimorfisme	26
B.1.12.- Plantilles	26
C.- Modelat de comportament	28
C.1- Casos d'ús:	28
C.2- Diagrama de Casos d'Ús	29
C.2.1.- Ítems bàsics del diagrama de cas d'ús	29
C.2.2.- Exemple complet	29
C.2.3.- Eines d'unió dins del diagrama de casos d'ús	29
C.2.4.- Relació d'inclusió entre Casos d'Ús	30
C.2.5.- Relació d'extensió entre Casos d'Ús	30
C.3- Diagrama d'activitats	31
C.3.1.- Exemple	31
a) Nivell de detall 1	32
b) Nivell de detall 2	33

c) Exemple de gestió de concurrència	34
C.4- Diagrama de seqüència	35
C.4.1 .- Visió general	35
C.4.2 .- Fases del diagrama de seqüència	35
a) Diagrama de Seqüència del Sistema (DSS)	35
b) Nivell de detall: Diagrama de seqüència	36
C.4.3 .- Notació bàsica	37
a) Línies de vida i missatges	37
b) Exemple marc	38
c) Auto-Missatges o missatges a this	39
d) Retorn	39
e) creació i destrucció d'instàncies	39
f) frames o blocs	39
g) Crides síncrones i asíncrones	40
C.5- Diagrama de col·laboració	41
C.6- Diagrama d'estats	41
C.6.1 .- Elements del diagrama	42
C.6.2 .- Guardes i accions	42
C.6.3 .- Tipus d'events	42
C.6.4 .- Autotransicions	43
C.6.5 .- Accions d'entrada i sortida	43
C.6.6 .- Exemple complet	43
C.6.7 .- Submàquines d'estats	45
a) Disjoint Substates	45
b) Paral·lel substates	45
c) Indicador d'història d'estats	46
D.- Patrons de disseny	47
D.1- Disseny per capes	47
D.2- Capa de persistència	48
a) Implementació manual	49
D.3- Patrons GRASP	50
D.3.1 .- Creador	50
D.3.2 .- Expert	50
D.3.3 .- Controlador	51
D.3.4 .- Alta cohesió	52
D.3.5 .- Polimorfisme	53
D.3.6 .- Classes fabricades	54
D.4- Patrons de disseny GoF	55
D.4.1 .- Adaptador o driver	55
D.4.2 .- Factory	55
D.4.3 .- Singleton	60

A.- El llenguatge de modelat UML

UML són les sigles de *Unified Modeling Language* o Llenguatge Unificat de Modelat.

Un model és una simplificació de la realitat, que ens permet descriure'l des de diferents punts de vista. Els models ens ajuden a entendre millor els sistemes que estem desenvolupant i a estudiar-los.

UML és un **llenguatge gràfic** de modelat que ens permet **visualitzar**, **especificar** i **documentar** cadascuna de les parts de la que es compon el procés de desenvolupament de programari.

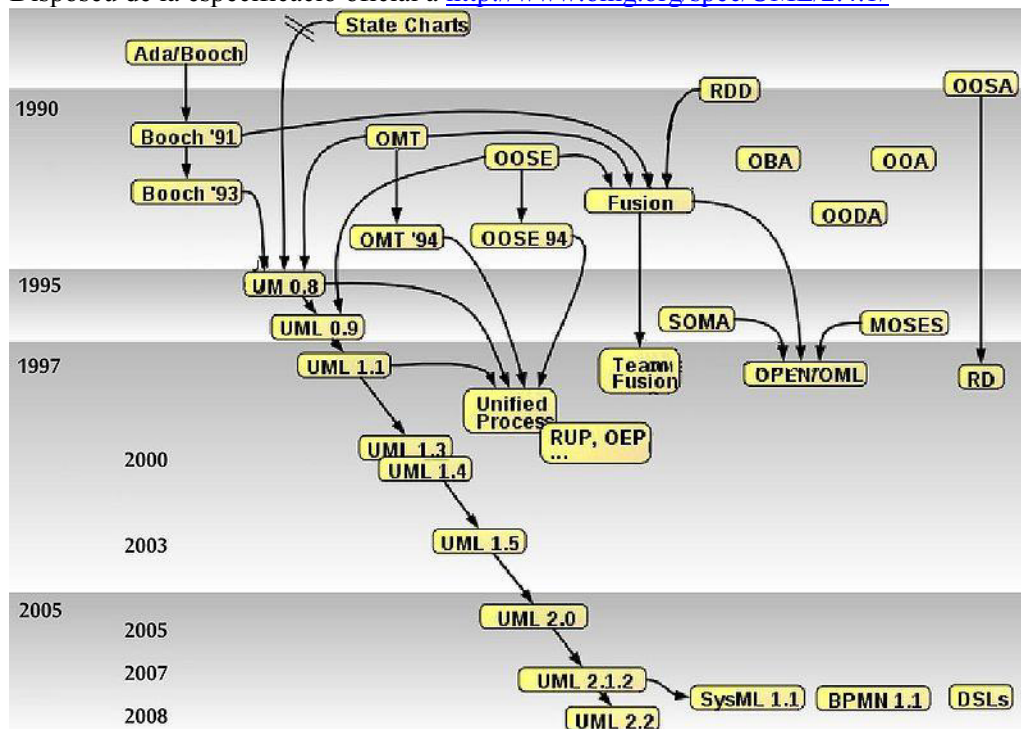
- **Llenguatge:** UML té un vocabulari (elements) i una gramàtica (regles de composició) que permet construir models.
- **Visualitzar:** Mitjançant un llenguatge gràfic. Tot i que l'expressió escrita pot ser molt concisa, el format gràfic permet amb unes poques regles comunicar de forma molt directa i concreta, sense ambigüitats, les funcionalitats del sistema.
- **Especificar:** Especificar significa construir models precisos, no ambigus i concrets.
- **Documentar:** UML dona suport a la documentació del qualsevol projecte, inclusive la fase de requisits i proves.
- **Construir:** Encara que no és un llenguatge de programació, els seus models poden connectar fàcilment amb ells, generant codi i viceversa (enginyeria inversa)
UML és suficientment expressiu i no ambigu com per a permetre l'execució directa de models, simulació de sistemes i instrumentació de sistemes en execució.

A.1- Orígens UML

Actualment estem per la versió 2.4.1 de l'estàndard (<http://www.uml.org/>).

UML fou adoptat com a standard per l'*Object Management Group* (OMG) el novembre de 1997.

Disposau de la especificació oficial a <http://www.omg.org/spec/UML/2.4.1/>



A.2- UML. Què és i que no és

- UML defineix un llenguatge gràfic compost de símbols i diagrames amb unes regles específiques d'interpretació.
 - **No és una metodologia.** UML no diu ni quan ni com usar els diagrames per a fer l'anàlisi i el disseny.
 - **És independent del llenguatge de programació.**
- UML permet modelar tot el cicle de vida d'un projecte.

A.3- OCL

Els diagrames UML es poden enriquir mitjançant l'ús del *Object Constraint Language* o OCL. És un llenguatge formal per a escriure expressions que permet contrarestar la interpretació i posterior implementació d'un diagrama, expressant restriccions que no poden ser representades amb la sintaxi gràfica d'UML. Aquestes expressions poden ser típicament dels següents tipus:

- Condicions invariants per tipus i classes
- Post-condicions per mètodes
- Pre-condicions per mètodes
- Restriccions sobre operacions
- Requeriments de test i especificacions

OCL també defineix com construir expressions per tal de navegar pel model UML d'una forma independent del llenguatge de programació.

Existeixen d'altres llenguatges de restricció disponibles, però OCL és l'únic oficial i estandaritzat per la OMG.

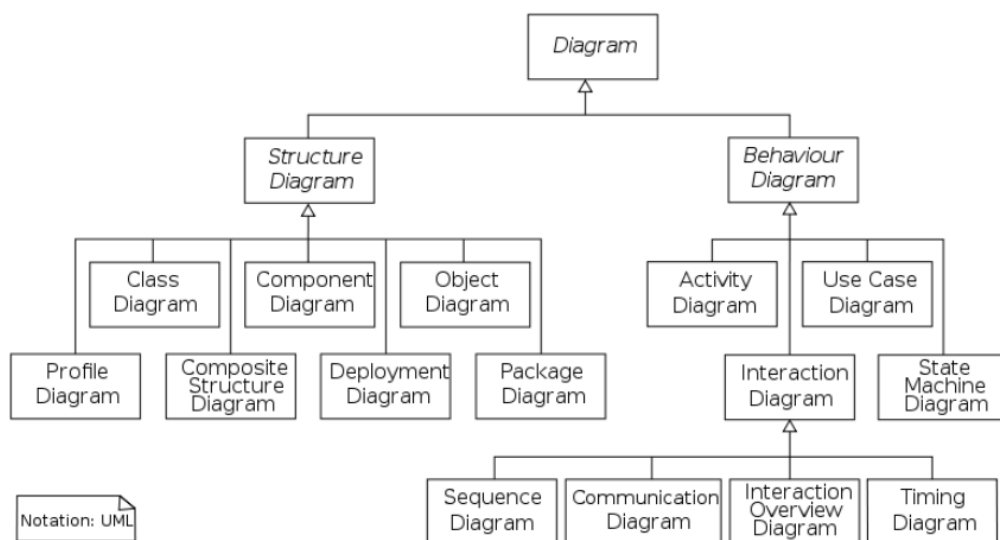
Podeu trobar l'especificació completa a:

<http://www.omg.org/spec/OCL/2.3.1>

A.4- Models i Diagrames

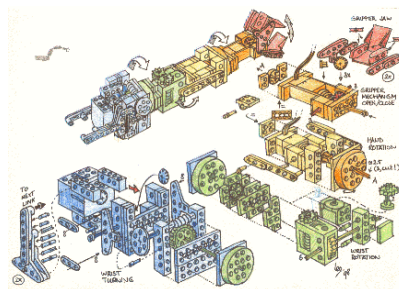
Un model captura certs aspectes d'un sistema per a un propòsit concret. UML defineix un conjunt de diagrames que intenten modelar diferents parts del sistema. Segons el punt de vista de l'anàlisi que fan, s'estableixen dues grans categories de diagrames:

- **STRUCTURE DIAGRAMS:** El **modelat estructural o estàtic**, que descriu l'estructura organitzativa dels mòduls i components que formen el sistema des d'un punt de vista intemporal.
- **BEHAVIOUR DIAGRAMS:** El **modelat de comportament o dinàmic**, que s'encarrega de mostrar la comunicació i interacció entre les parts del sistema tenint en compte la dimensió temps.
 - Com a subconjunt dels diagrames de comportament, apareixen els diagrames d'interacció, que s'especialitzen en detallar la interacció entre objectes.

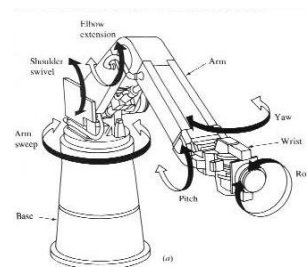
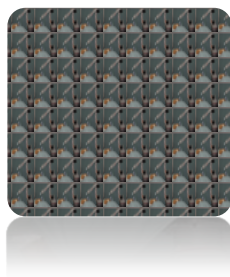


Modelat estructural	Diagrama de classes Diagrama d'objectes Diagrama de components Diagrama de desplegament
Modelat de comportament	Diagrama de seqüència. Diagrama d'estats. Diagrama de col·laboració. Diagrama de casos d'ús. Diagrama d'activitats.

Estructura

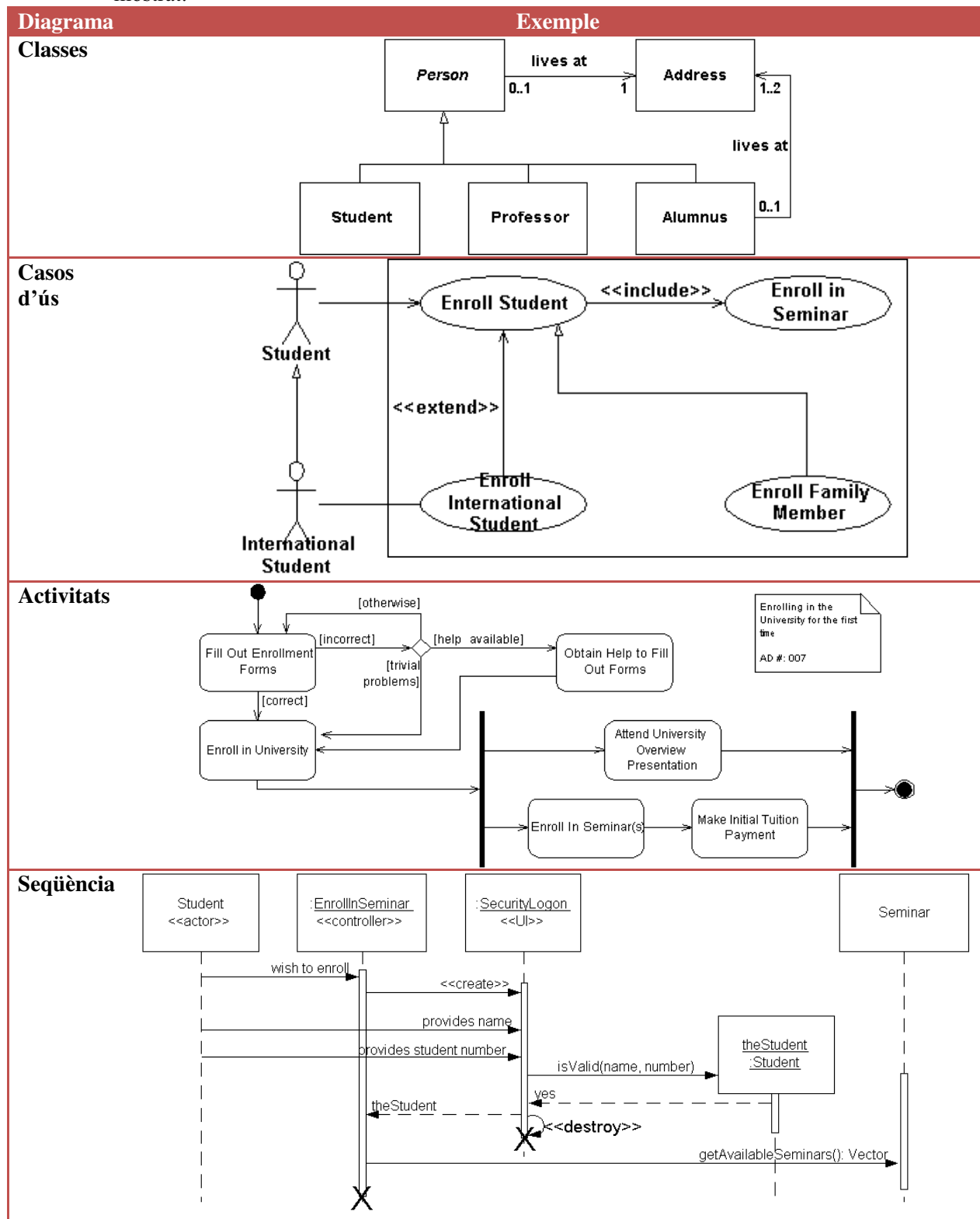


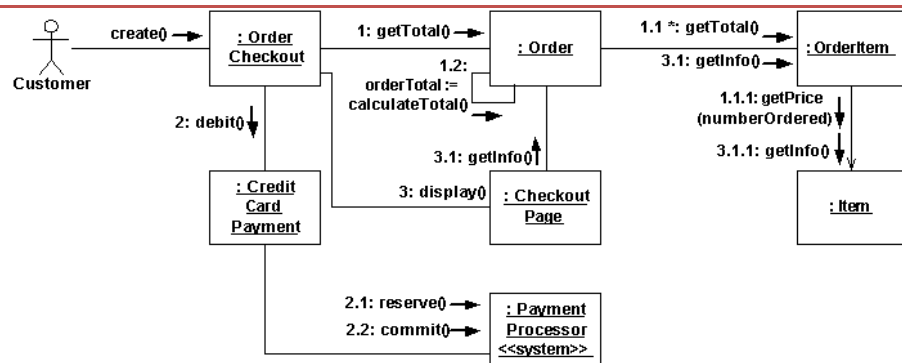
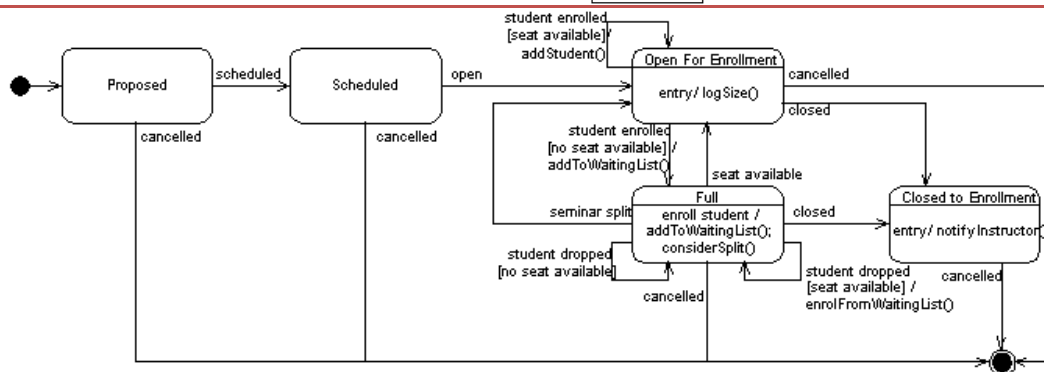
Comportament
(Funcionalitats, Interacció) dels components del sistema.



A.5- Diagrames coberts pel curs

Durant aquesta unitat formativa estudiarem els següents diagrames segons l'ordre mostrat:

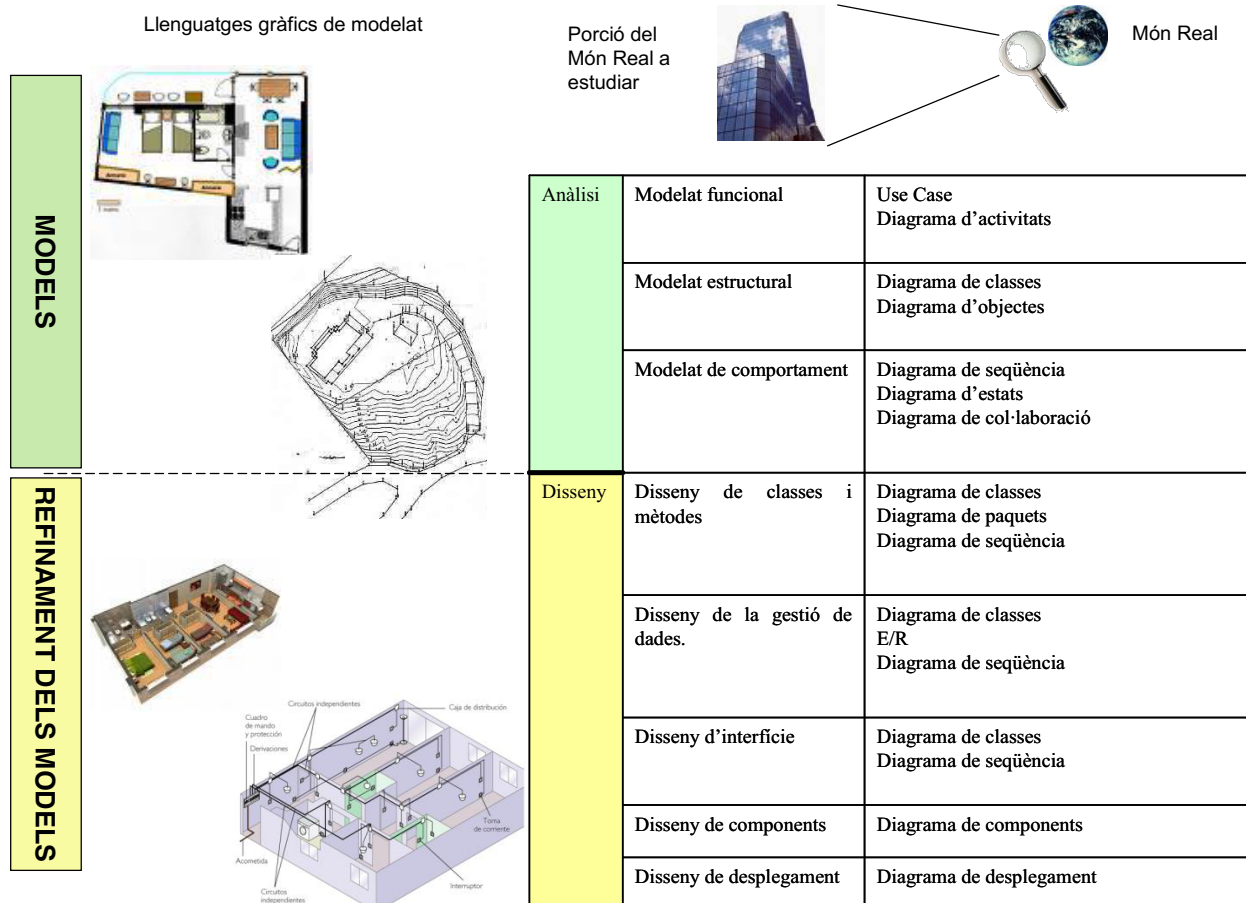


Col·laboració**Estats**

A.6- Seqüència d'utilització habitual dels diagrames UML

UML és una eina de modelat que no defineix cap metodologia, el seu ús és lliure i es pot adaptar a les necessitats i sistemes de treball de cada organització. No és obligatori fer tots els diagrames del sistema, sinó que ens podem concentrar en aquells que siguin més significatius pel nostre problema.

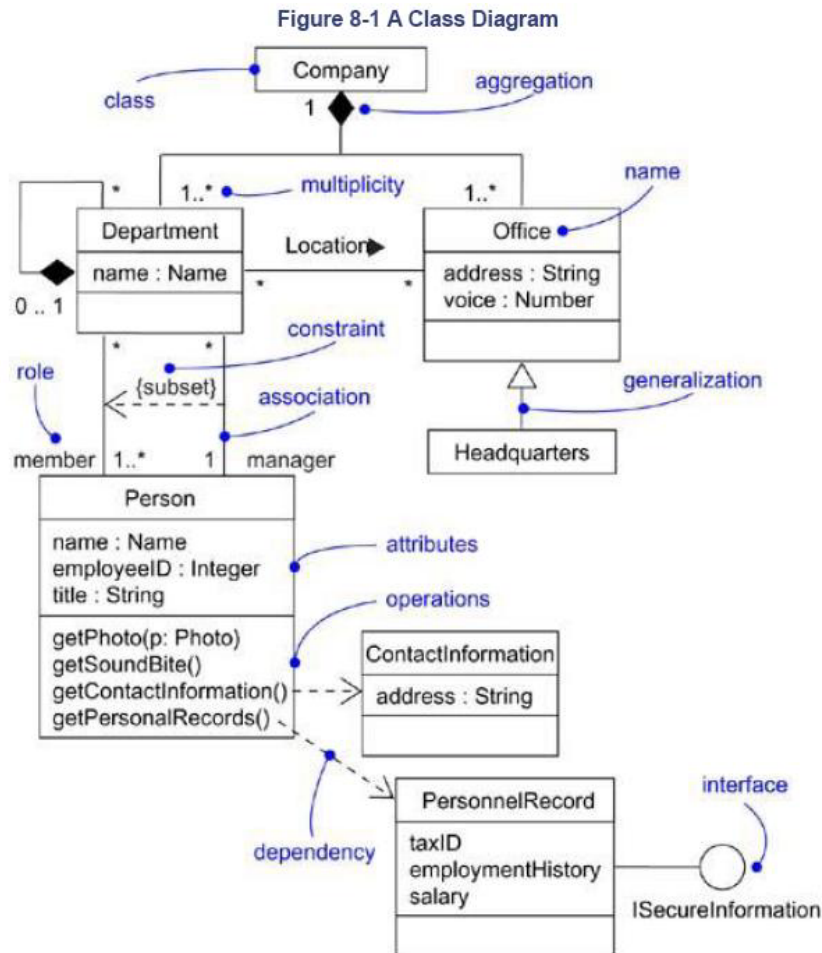
Amb l'objectiu de poder triar en cada fase del nostre projecte quins són els diagrames més escaients en aquell moment, a continuació es fa una classificació dels diagrames a llarg de les fases de anàlisi i disseny.



B.- Modelat Estructural

B.1- Diagrama de classes

B.1.1.- Aspecte general



B.1.2.- Les fases del diagrama de classes

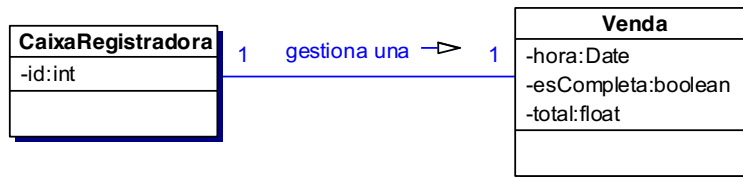
El procés de generació d'un diagrama de classes és fruit de múltiples iteracions del procés de creació del programari.

Podem distingir, però, tres estadis dins de la confecció d'aquest diagrama:

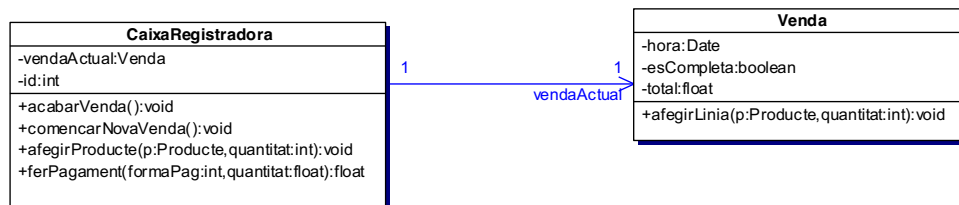
- **Model de domini:** ens centrem només a modelar les classes que detectem en el domini del problema. També indiquem quines relacions hi ha entre elles i les seves cardinalitats.



- **Model de domini amb atributs:** Afegim a les classes els atributs necessaris, indicant-ne el tipus.



- **Model de disseny:** Enfoquem el mateix model des del punta de vista del programari, refinant la definició de classes amb conceptes com:
 - Per a les classes:
 - Mètodes: signatures, tipus d'accés...
 - herències, interfícies, classes abstractes
 - Per a les relacions: la navegabilitat, noms de rols, tipologies d'associació (composició/associació), etc.

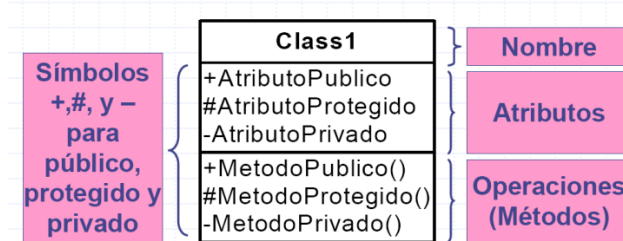


- **IMPORTANT:** En fases posteriors de refinament és habitual afegir noves classes per a aspectes més tècnics (gestió de la interfície d'usuari, gestió de la capa de base de dades, etc.)

B.1.3 .- Classes

En el món real hi ha molts d'objectes del mateix tipus. Hi ha milers de cotxes del mateix model i marca i tots ells s'han construït a partir de la mateixa plantilla i per tant tenen les mateixes components.

En termes de programació orientada a objectes, el nostre cotxe és una ocurrència (instància) d'una classe d'objectes coneguda com a *Cotxe*. Una classe és, doncs, un patró per a construir objectes.

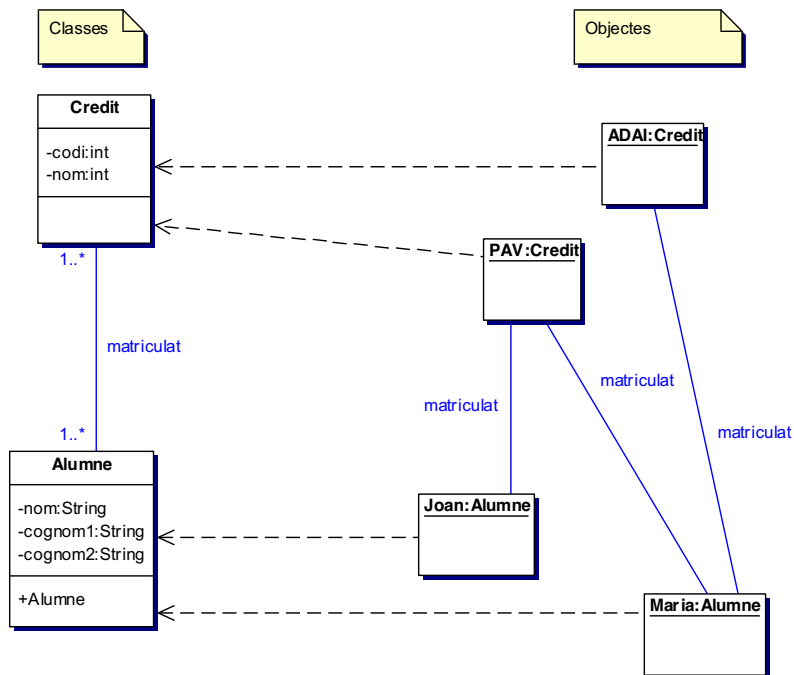


a) Objecte o instància

Un objecte és una component software formada per un estat i un comportament . S'utilitza per modelar objectes del món real.

El vostre cotxe, per exemple, té un estat (una matrícula concreta, una velocitat actual, ocupants, carburant restant, marxa actual,...) i un comportament (partir, frenar, aturar, canviar de marxa, obrir porta, ...)

En UML els objectes es mostren en capses, amb el nom de l'objecte subratllat, i indicant el nom de la classe a la que pertany en format [nomObjecte]:[nomClasse]



B.1.4 .- Control d'accés

És el que es coneix com ocultament de la informació: l'usuari no pot accedir directament a les parts de l'objecte o a la seva implementació

L'accés es limita a les operacions definides per l'objecte. S'estableixen uns nivells d'accés generals:

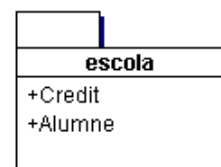
- **Pública (+)** : Un mètode o atribut públic és accessible des de qualsevol classe.
- **Protegida (#)** : Un mètode o atribut protegit és només accessible des de la pròpia classe o subclasses.
- **Privada (-)** : Un mètode o atribut privat és només accessible des de la pròpia classe.

B.1.5 .- Paquetització o espais de noms

Les classes es poden agrupar de forma lògica en paquets, de forma que podem trobar les classes convivint amb d'altres classes relacionades dins del mateix paquet o en paquets fills. És el que es coneix com a jerarquia de paquets.

Aquesta jerarquització ajuda a afinar el control d'accés, apareixent el concepte d'accés de paquet (*package*)

Els espais de noms són una altra forma d'agrupació lògica. La classe "compte" pot tenir varis significats dins d'una organització, a tal efecte tenim espais de noms. Per exemple a l'espai de noms "bancs" la classe "compte" és un compte bancari, a l'espai de noms vendes el "compte" és la factura al client.



B.1.6 .- Atributs

Són un conjunt de variables que tenen un nom propi i un tipus de dades específic dins de la classe on habiten. Un cop instanciada la classe, a l'objecte, aquestes variables poden prendre valors i ser modificades de forma dinàmica. El valor de l'atribut és propi de l'objecte, no de la classe. Per exemple, per la classe *Alumne*, que té un atribut *nom*, el valor del *nom* per cadascun dels objectes d'aquesta classe (en Pep, en Pau, la Maria...) és totalment independent l'un de l'altre.

Els atributs s'especifiquen indicant el seu nom com a mínim:

```
nom
```

Si no s'indica res més es dona un tipus d'accés privat per defecte, però si es vol canviar podem usar els modificadors +, -, # (*public*, *private* i *protected* respectivament) davant del nom de la variable per a indicar-ne l'accessibilitat:

```
+nom
```

És força aconsellable que s'indiqui també el tipus de dades al que pertany. El tipus pot ser una classe o bé un tipus bàsic (int, char, float, double i arrays dels mateixos)

```
+nom:String  
+nom:char[]
```

a) Valor per defecte:

```
+pi:double ="3.14159"
```

b) Valor de només lectura

```
+pi:double ="3.14159" {read-only}
```

c) Multiplicitat

- Per indicar un valor opcional:

```
+nom:String [0..1]
```

- Per indicar un rang d'ocurrències

```
+tutorsLegals:Persona [1..2]
```

- Per indicar un valor repetit sense restriccions

```
+clients:Client *
```

d) Concepte d'Atribut derivat

Un atribut derivat és aquell que pot ser calculat a partir dels valors d'altres atributs (d'aquesta o d'altres classes). S'indica posant al davant del nom una barra de divisió "/" Per exemple, si definim una classe "LiniaFactura", tindriem els següents atributs:

```
- preuUnitat:double  
- unitats:double  
- /total:double
```

Observem que l'atribut total és derivat, doncs es pot calcular a partir del preu i del nombre d'unitats.

NOTA: Aquest modificador no està suportat per Borland Together.

e) Atributs estàtics

Els atributs estàtics poden ser usats directament des de la classe, no ens cal tenir una instància. Per indicar-ho usem un subratllat simple:

```
+impresoraCompartida:Impressora
```

B.1.7 .- Mètodes

Els mètodes al igual que els atributs tenen el seu nom i modificador d'accés. Per defecte si no s'indica el contrari els mètodes són públics.

També és molt aconsellable indicar els paràmetres del mètode i el tipus de retorn. Ho farem tal i com es mostra a la figura:

```
+nomMètode( [nomParam1:Tipus1], [nomParam2:Tipus2]...) : tipusRetorn
+ ferComanda( prod:Producte, clt:Client, quantitat:int ) : Comanda
```

Si el mètode no retorna res, no cal indicar tipus de retorn:

```
#enviarEmail( to:String, subject:String, body:String)
```

a) Mètodes estàtics

Els mètodes estàtics poden ser cridats directament sobre la classe, no ens cal tenir una instància. Per indicar-ho usem un subratllat simple:

```
+agafarImpressoraCompartida():Impressora
```

B.1.8 .- Propietats

Com ja s'ha comentat, és habitual que els atributs es mantinguin privats i que només en donem accés des de l'exterior a través de mètodes. Quan volem **encapsular** un atribut primer el crearem amb visibilitat privada, i posteriorment escriurem els dos mètodes *get* i *set* que permetin llegir i modificar el seu contingut.

Els atributs fets visibles a l'exterior usant aquest mecanisme es coneixen com a **propietats**. La gran avantatge de les propietats sobre la manipulació directa dels atributs públics, és que les propietats **asseguraran el compliment de les regles** de validació i ús de les dades de la classe.

Les propietats poden ser de lectura, d'escriptura o de lectura i escriptura segons si els mètodes *get* o/i *set* estan disponibles.

```
- ingressos: BigDecimal // només de lectura.
- dataNaixement:Date    // lectura i escriptura
-----
+ getIngressos ():BigDecimal
+ getDataNaixement (): Date
+ setDataNaixement ( nouValor: Date )
```

Moltes eines CASE ja han previst les necessitats d'encapsulació, i ens faciliten la feina en el cas de voler crear propietats:

- creen automàticament per nosaltres els *setters* i *getters*

- per augmentar la claredat els mètodes *get* i *set* no es mostren, només el nom de la propietat

Les propietats estan íntimament relacionades amb el concepte de “**Bean**” (mongeta), que és aquella classe que està pensada per a ser reutilitzable, i compleix una sèrie de condicions:

- conté un nombre variat de propietats (mètode set/get) , que són la interfície útil de la classe
- tots els atributs són privats
- té un constructor públic sense paràmetres
- [al llenguatge Java] ha de ser *serialitzable* (s’ha de poder convertir en una tira de bytes per poder ser desada en un dispositiu d’emmagatzemament)

B.1.9 .- Tipus de relacions entre classes

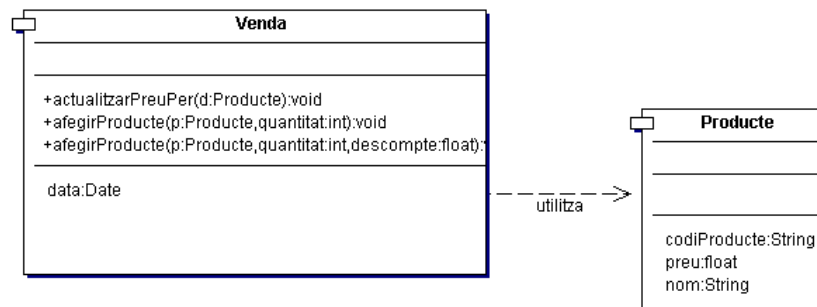
a) Dependència

- Indica una dependència funcional d'una classe respecte d'una altra.
- Una classe depèn d'una altra quan la fa servir, ja sigui instanciant-la i/o invocant-ne mètodes. Si la classe de la que depenem es veu modificada, això pot afectar a la classe dependent
- També pot haver-hi dependència indirecta.

Per exemple: Classe *Venda* i *Producte*. La classe *Venda* utilitza la descripció del producte per a obtenir el preu de venda. Així doncs, un canvi a la interfície de *Producte* **podria** afectar al codi de *Venda*:

```
public class Venda
{
    public void actualitzarPreuPer(Producte p )
    {
        float preuBase = p.getPreu();
        //...
    }
    public void afegirProducte(Producte p, int quantitat)
    {
        //...
    }
    // ...
}
```

Representació:



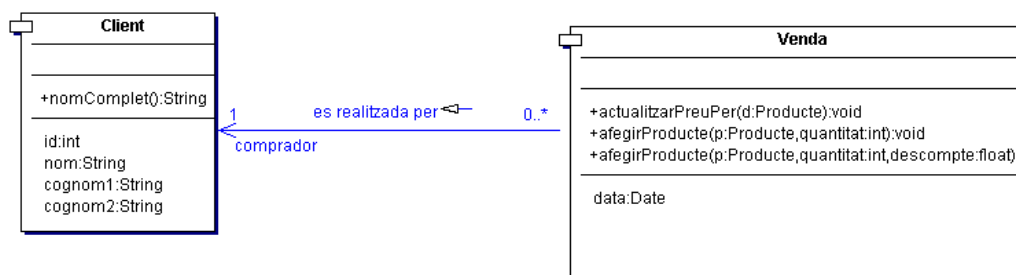
b) Associació

Indica una relació estructural entre instàncies (objectes), que especifica que els objectes d'una classe estan connectats amb els objectes de l'altre.

Aquesta associació pot tenir un caràcter bidireccional o unidireccional

Poden existir relacions d'autoassociació (reflexives) que impliquen que un objecte d'una classe està lligat amb un altra objecte de la mateixa classe.

Donades les classes *Empleat* i *Departament*, es pot establir una relació d'associació bidireccional doncs un empleat treballa a un departament, i a un departament hi treballen empleats.

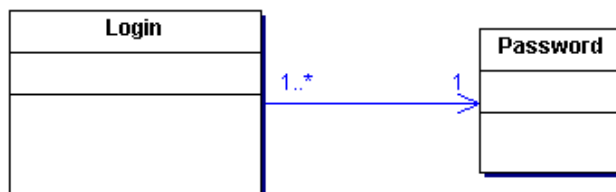


L'associació és una línia que pot tenir opcionalment les següents “decoracions”:

Punta de fletxa o navegabilitat:	Indica quina classe fa referència a l'altre (conté un atribut del tipus referenciat)
Cardinalitats als dos extrems.	La cardinalitat s'expressa usant la mínima i la màxima (en d'altres notacions correspon a l'obligatorietat de participació i la cardinalitat respectivament) La cardinalitat mínima va primera, separada per punts (indicant un rang de valors) de la màxima. Si la cardinalitat és exacta (p.ex. 5) podem posar un valor numèric únic. El valor N (cardinalitat múltiple indefinida) es representa amb un asterisc. Veiem-ne alguns exemples: <div style="border: 1px solid black; padding: 5px; margin: 10px auto; width: fit-content;"> 0..1 1 0..* 1..* </div>
Etiqueta o “verb” de l'associació	es realitzada per
Fletxa de lectura de l'etiqueta.	es realitzada per ←
Rols d'origen i destí	<div style="display: flex; align-items: center;"> <div style="border: 1px solid black; padding: 5px; margin-right: 10px;">1</div> <div style="border: 1px solid black; padding: 5px; margin-right: 10px;">comprador</div> <div style="border: 1px solid black; padding: 5px; margin-right: 10px;">es realitzada per ←</div> <div style="border: 1px solid black; padding: 5px;">0..*</div> </div> <p>Podem donar un nom a la participació de la classe en la relació, tant a l'origen com al destí.</p>

c) Associacions i navegabilitat

La navegabilitat consisteix en el fet de poder arribar fàcilment d'una classe a una altra. Hi ha casos on ens interessa que entre dues classes relacionades només sigui possible arribar d'una a l'altra, però no al revés. Seria el cas de les classes següents:

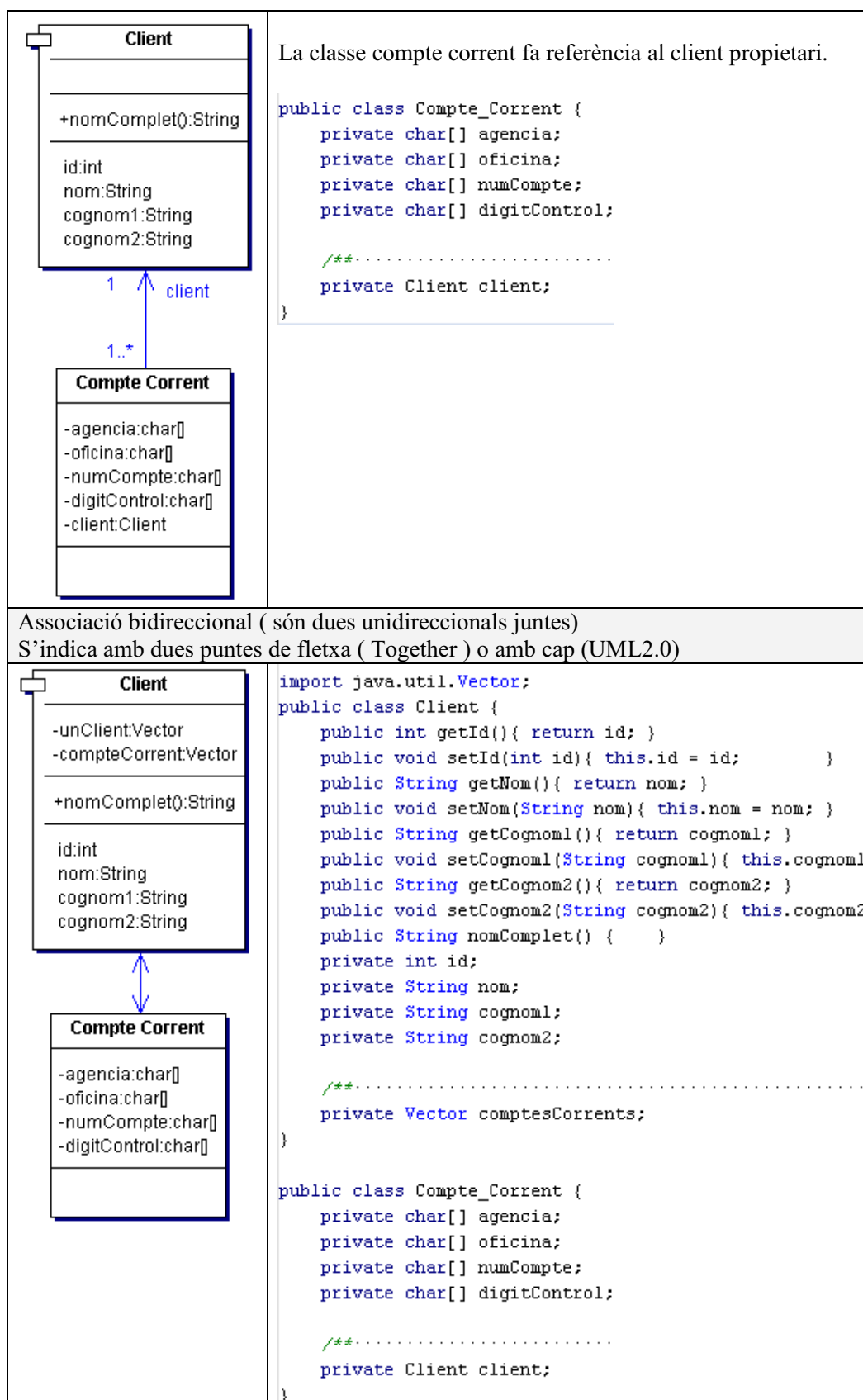


A partir de login volem conèixer fàcilment la contrassenya, de forma que internament puguem autenticar ràpidament els intents d'accés. No ens fa falta però obtenir l'usuari a partir del password (i a més en aquest cas no és possible).

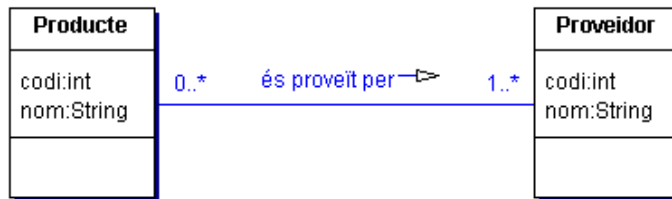
Sovint la navegabilitat és total o bidireccional, cosa que indica que es vol navegar en les dues direccions. És el cas per defecte, i s'indica posant la línia sense fletxes.

Des del punt de vista intern, la navegabilitat indica quina des les dues (o ambdues) classes fa referència **explícita** a l'altre. La referència explícita és el fet de que una classe tingui com a membre una instància de l'altre classe. En el cas de que la cardinalitat sigui major a 1, la classe que fa referència pot contenir un *array*, un *vector* o qualsevol altre contenidor que permeti tenir diverses instàncies de la classe referenciada.

Associació navegable en un sentit (unidireccional)



Implementació d'una relació N:M amb navegabilitat bidireccional



```
import java.util.Vector;
```

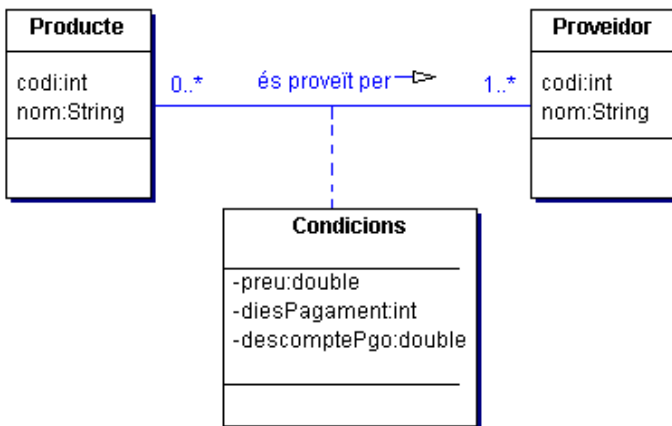
```
public class Producte {
    int codi;
    String nom;

    Vector<Proveidor> proveidorsDisponibles;
}
```

```
import java.util.Vector;
```

```
public class Proveidor {
    int codi;
    String nom;
    Vector<Producte> productesSuministrats;
}
```

Implementació d'una relació N:M amb navegabilitat bidireccional usant una classe associativa (ho veurem més endavant...)



```
import java.util.Vector;
```

```
public class Producte {
    int codi;
    String nom;

    Vector<Condicions> condicionsProvs;
}
```

```
public class
Condicions {
    Proveidor prov;
    Producte prod;

    double preu;
    int diesPagament;
    double deptePPGO;
}
```

```
import java.util.Vector;
```

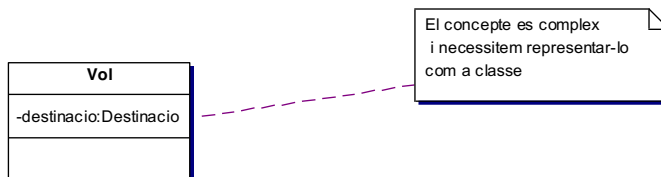
```
public class Proveidor {
    int codi;
    String nom;
    Vector<Condicions> condicionsProd;
}
```

d) Consells d'ús d'Atributs i Associacions !



No mostreu conceptes complexos com a atributs, feu servir associacions:

Pitjor:

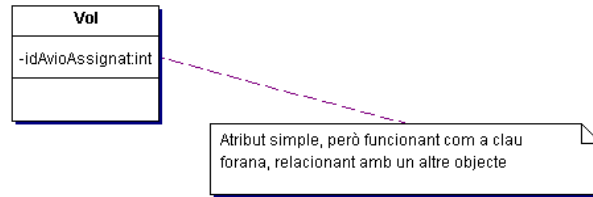


Millor:

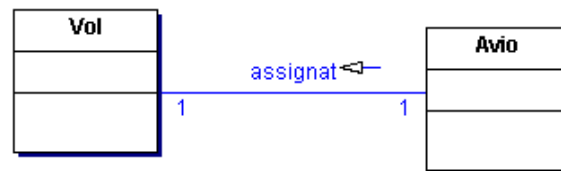


No useu claus foranes com a atributs, mostreu associacions:

Pitjor:



Millor:

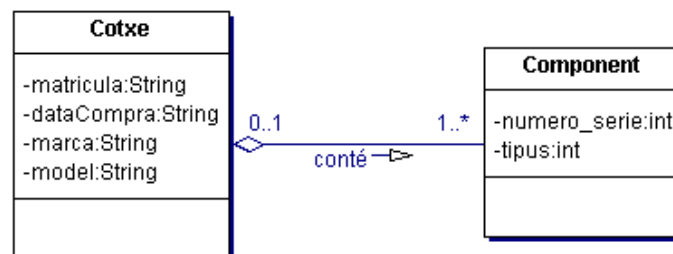


e) Agregació i composició

Es defineixen dos tipus especials d'associació: les agregacions i les composicions. **Ambdues són un cas especial d'associació, on la relació es del tipus conjunt/element.** Una classe (el *conjunt*) consta de diversos components més petits (els *elements*). L'objecte de la classe *tot* conté o apunta a objectes de la classe *part*. Dins d'aquesta casuística, diferenciem entre agregació i composició:

- **Agregació**

Si la classe "conjunt" pot existir sense necessitat de la classe "element", i fins i tot els elements són intercanviables entre diferents conjunts, podem considerar la relació com agregació. Entre la classe *Cotxe* i *Component* podríem dir que hi ha una relació d'agregació:



Nota: és molt difícil distingir entre agregació i una associació normal. En cas de dubte useu associacions.

- **Composició**

Si la classe “conjunt” no pot existir sense les parts (i viceversa), la relació es de composició. És el cas de la classe *Tauler* i la classe *Casella*, o de la classe *Bosc* i *Arbre*. No poden existir una sense l'altra.



Des del punt de vista de la programació, una relació de composició implica que el cicle de vida de la classe *element* és totalment controlat per la classe *conjunt*. Això voldrà dir que la creació i la destrucció de la classe *part*, són la responsabilitat de la classe *conjunt*. En C++ la diferència entre Agregació i Composició és que en les primeres el *tot* conté una referència (punters) a les parts, mentre que en la composició és habitual que el *tot* conté directament la *part*.

Agregació

```

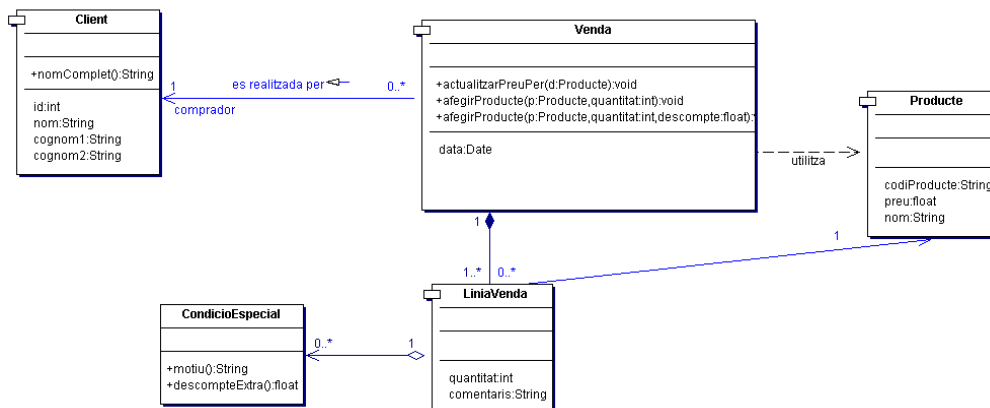
class Departament
{
private:
    Vector<Professor*> profes;
}
  
```

Composició

```

class Cercle
{
private:
    Punt centre;
}
  
```

A continuació s'inclou un exemple combinat:



f) Associació amb classe associativa (≡ Chen relació amb atributs...)

En alguns casos la relació entre dues classes pot implicar l'existència d'una nova classe, anomenada classe associativa o d'associació. La classe que neix de l'associació conté atributs que només tenen sentit quan posem en relació les dues classes associades.

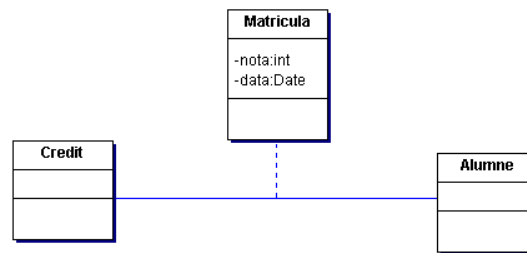
Aquest concepte equival a les situacions següents dels diagrames Chen:

- relació amb atributs. Els atributs de la relació es poden “empaquetar” en una classe associativa.
- relació ternària amb una de les entitats participant amb cardinalitat 1. Aquesta última és la classe associativa.

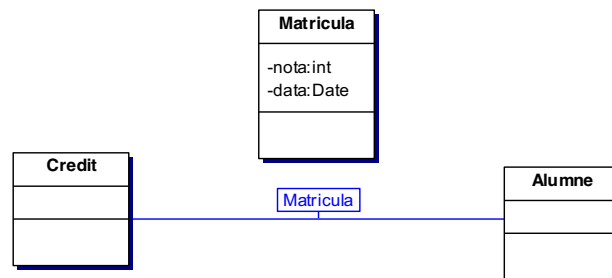
Per exemple, les classes *Alumne* i *Crèdit* estan associades mitjançant una classe *Matricula*, que té com atributs el curs acadèmic i la nota obtinguda.

La classe associativa en aquest cas és *Matricula*, i es representa unint-la al centre de l'associació usant una línia discontinua.

Notació UML



Notació Together



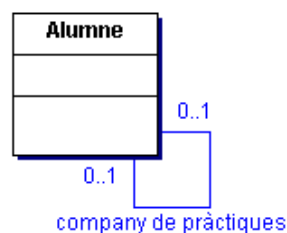
g) Associació múltiple (\equiv Chen relacions n-àries)

Quan més de dues classes intervenen en una associació parlem d'associacions múltiples. Tot i que en el modelat poden succeir situacions en les que necessitem fer servir aquest tipus d'associació, la majoria d'eines CASE no admeten relacions amb grau major a 2, doncs no hi ha una implementació clara i directe d'aquestes en codi font.

Així doncs, el mecanisme recomanat en aquestes situacions serà crear una classe que representi la l'associació, i que estigui associada en grau 2 amb totes i cadascuna de les classes que participaven a l'associació n-ària.

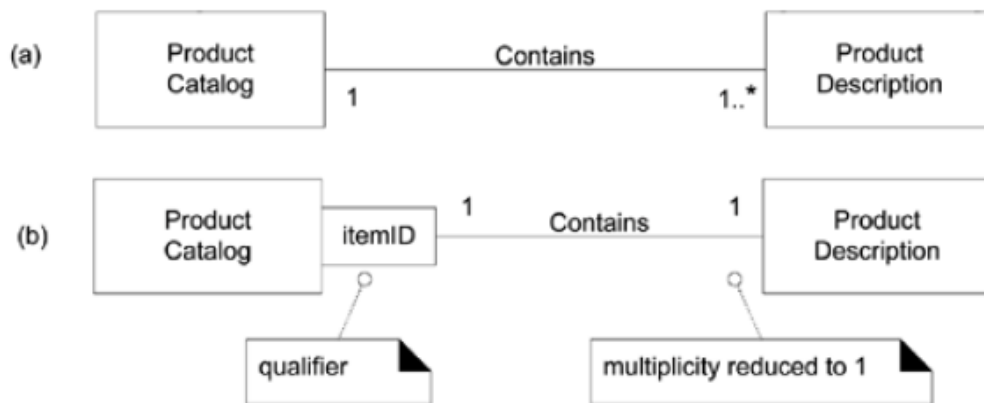
h) Associacions Reflexives

Una classe pot tenir una o varies associacions amb ella mateixa. Es mostra usant un llac tancat:



i) Associació Qualificada

L'associació qualificada és un concepte complex de definir. Es tracta d'associacions en les que indiquem un qualificador, que no és més que un atribut especial lligat a l'associació que permet identificar un objecte dins del conjunt d'objectes implicats a l'associació. Des d'un punt de vista informal, representa la possibilitat de fer cerca del destinatari de l'associació per una clau o índex (força típic si usem per a la implementació una taula *Hash* o un *array*/llista respectivament)



El cas (b) en Java es podria codificar com una Hashtable:

```

1  package hospital;
2  import java.util.Hashtable;
3  public class ProductCatalog{
4      Hashtable productDescriptions;
5      void addProductDescription( String itemID, prod ProductDescription)
6      {
7          productDescriptions.put( itemID, prod);
8      }
9      ProductDescription getProductDescription( String itemID)
10     {
11         return productDescriptions.get( itemID );
12     }
13 }
  
```

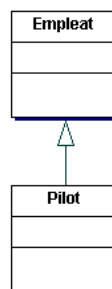
Per la gent que se sent còmoda amb la notació Chen, podem dir que les entitats Dèbils poden ser representades per associacions qualificades, tot i que el potencial representatiu de l'associació qualificada és més ampli.

NOTA: Borland Together **no dona** suport directe a aquest tipus d'associacions.

B.1.10 .- Generalització / especialització (Herència)

a) Herència simple

UML permet representar herències usant una fletxa amb punta triangular de color blanc. La classe apuntada és la superclasse, l'altre és a subclasse.



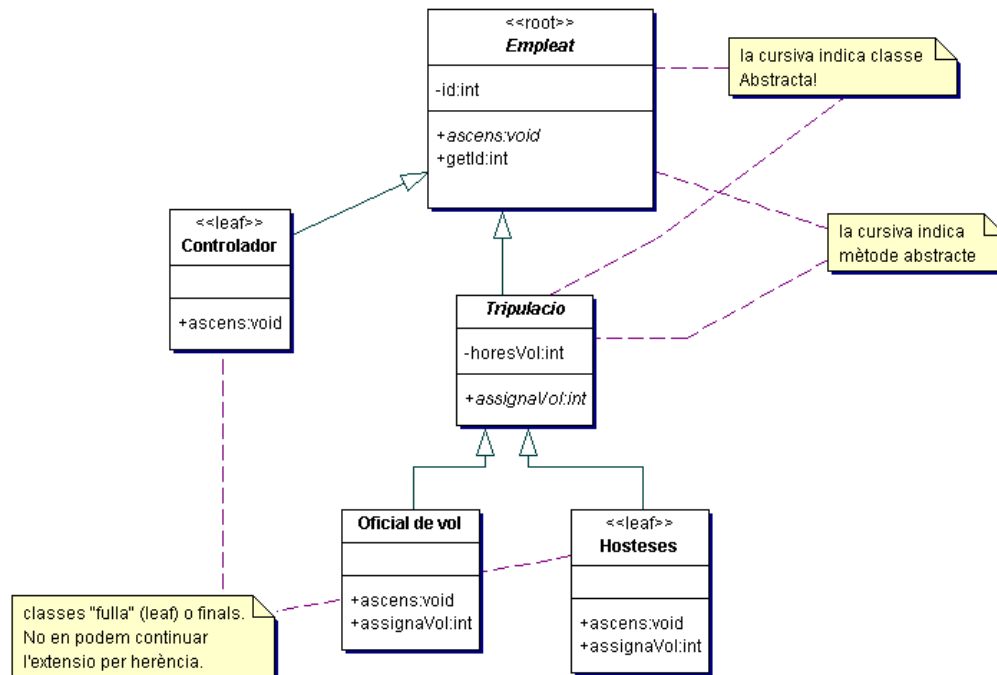
b) Jerarquies, classes abstractes i finals

L'ús apilat de l'herència ens porta a tenir jerarquies de classes, cosa molt freqüent en el món del la orientació a objectes.

Adicionalment, dins de les jerarquies podem marcar certes classes com a abstractes o finals. Recordem el significat de cada concepte:

- classe **abstracta**: és aquella que no podem instanciar, i per tant caldrà especialitzar-la (fer-ne subclasses) per a utilitzar-la. S'indiquen posant el nom de la classe en cursiva.
- classe **final**: és aquella classe que no admet més especialització, cap classe en pot heretar. Les indiquem amb l'estereotip <<leaf>> (fulla d'arbre en anglès)

Cal recordar que les classes abstractes poden tenir alhora mètodes abstractes, que són mètodes dels quals només se'n defineix la signatura (nom, paràmetres i tipus de retorn). Els mètodes abstractes no tenen codi i han de ser definits obligatòriament a les classes filles per tal de que puguin ser instanciables.



B.1.11 .- Interfícies

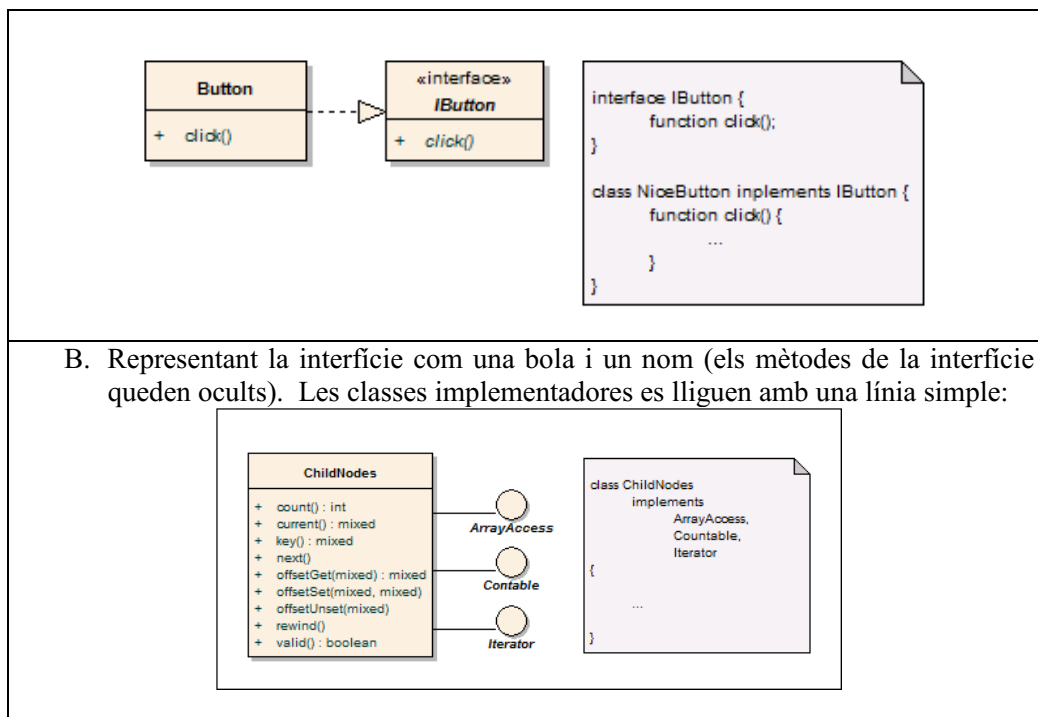
Una interfície és un conjunt de capçaleres de mètodes agrupats sota un nom comú. Podríem entendre una interfície com la “closca” d’una classe. És a dir, és una classe que no té atributs, tots els seus mètodes són públics i no tenen cap implementació (codi font que els descriu).

D’aquesta forma podem crear classes que implementin (defineixin i tinguin codi) a tots i cadascun els mètodes d’una interfície. Si això succeeix, podem dir que la classe implementa la interfície.

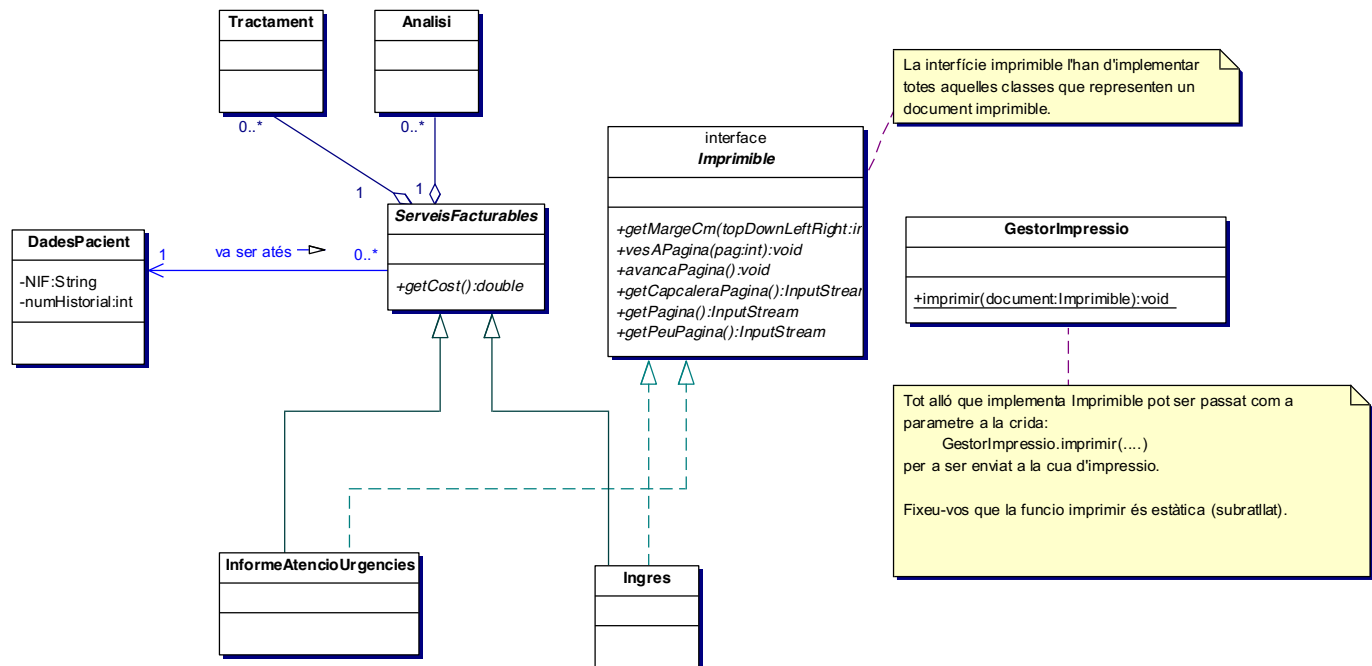
Les interfícies són molt útils, especialment en aquells llenguatges que no donen suport a l’herència múltiple, com ara Java.

En UML hi ha dos notacions possibles per a representar-les:

- una molt similar a la classe, però indicant el nom en cursiva i l’estereotip <<Interface>>. En aquest cas les classes que implementen la interfície es lliguen amb una fletxa molt similar a la de l’herència, però discontinua:



Veiem-ne un exemple d'aplicació:



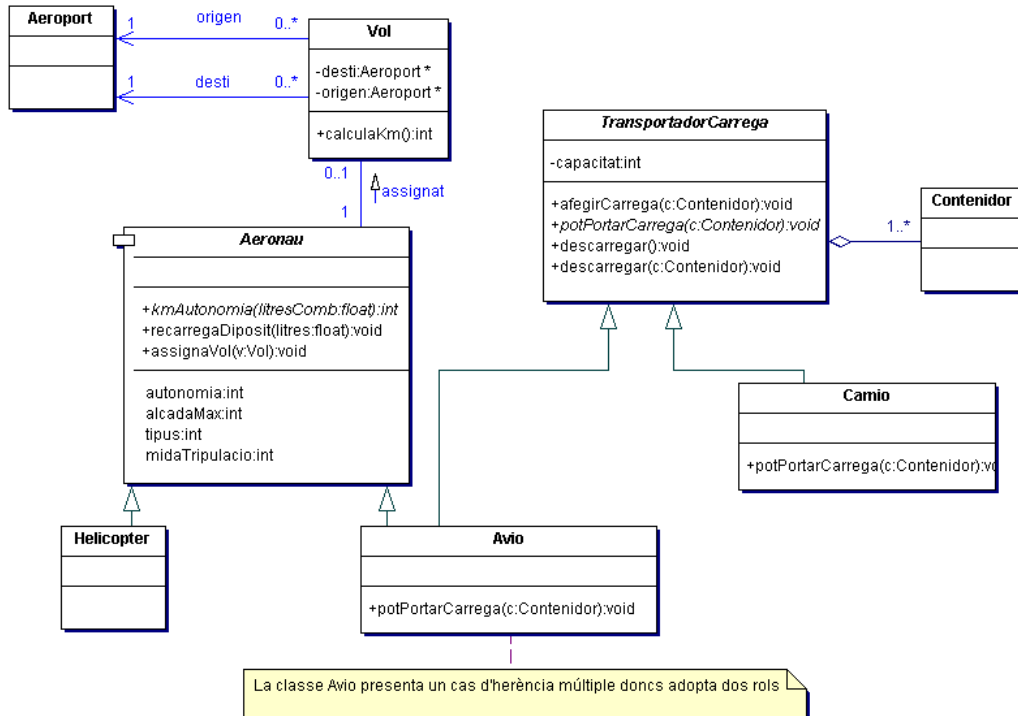
a) Herència múltiple

És possible que una mateixa classe sigui subclasse de dues superclasses diferents (serà l'origen de dues fletxes d'herència per tant).

L'herència múltiple ens duu al problema de la implementació, doncs no tots els llenguatges de programació admeten el seu ús. En el cas de que modelem una herència

múltiple quan el llenguatge de suport no l'admeti, haurem d'aplicar tècniques de “transformació” a herència simple.

Veiem un exemple d'herència múltiple en la classe *Avio* de l'esquema següent:



b) Polimorfisme

Podem tractar com a objecte de la superclasse qualsevol element de la subclasse.

Una extensió d'aquesta propietat són les interfícies. Quan una o més classes implementen una interfície, totes elles poden ser tractades com la interfície.

Cal destacar que quan a la classe filla sobreescrigui un mètode de la pare, quan invoquem aquest mètode sobre una instància sempre cridem a la implementació de la subclasse, encara que la tractem com una instància de la superclasse.

```

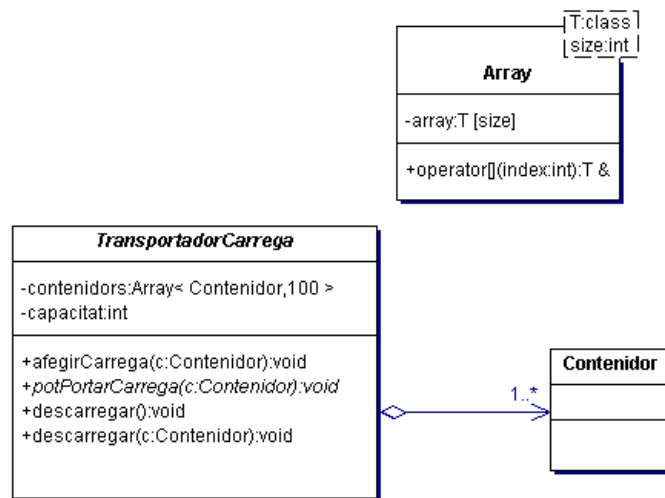
ObjecteUI oui= new FinestraUI(); // ús del polimorfisme
oui.mostrar(); // estem cridant a FinestraUI.mostrar() i no
               // a ObjecteUI.mostrar()
    
```

B.1.12 .- Plantilles

Són classes que defineixen una funcionalitat específica, però que pot ser utilitzada amb conjunt d'atributs o classes associades genèriques.

És el cas de les plantilles en llenguatge C++, on podem fer un *template* Cua que sigui genèric, podent usar la cua per a contenir enters, *floats*, o d'altres classes definides per l'usuari.

Veiem-ne la notació UML de la plantilla en un exemple. En aquest cas la classe *Array* és una plantilla, i com s'utilitza a nivell de codi per a implementar un contenidor:



A continuació veiem el codi font del *TransportadorCarrega*, on tenim ressaltada la línia que representa l'agregació de contenidor. Hem triat implementar l'agregació usant un Array de Contenedors de mida 100. Fixeu-vos que això no queda reflexat a l'UML, és una decisió d'implementació:

```

1  /* Generated by Together */
2  #ifndef TRANSPORTADORCARREGA_H
3  #define TRANSPORTADORCARREGA_H
4  #include "Contenedor.h"
5  #include "Array.h"
6  class TransportadorCarrega {
7  public:
8      virtual void afegirCarrega(Contenedor c);
9      virtual void potPortarCarrega(Contenedor c) =0;
10     void descarregar();
11     void descarregar(Contenedor c);
12 private:
13
14     /** @link aggregation
15      * @supplierCardinality 1..*
16      * @associates <{Contenedor}>*/
17     Array< Contenedor, 100 > contenedors;
18     int capacitat;
19 };
20 #endif //TRANSPORTADORCARREGA_H
  
```

C.- Modelat de comportament

C.1- Casos d'ús:

Informalment, els Casos d'Us (*use cases* en anglès) són històries textuais d'un actor usant un sistema per aconseguir els seus objectius. A continuació tenim un exemple breu d'un cas d'ús:

Cas d'ús "Realitzar venda":
<ol style="list-style-type: none"> 1. Un client arriba a la botiga i tria els productes que vol comprar. 2. El caixer utilitza la màquina registradora per a enregistrar cada ítem comprat. 3. El sistema imprimeix automàticament una factura amb el total i les línies de detall. 4. El client paga, amb tarja o en metàl·lic. <ol style="list-style-type: none"> 4.1. Si paga en metàl·lic i el caixer introdueix el import abonat al sistema. El sistema calcula el canvi, que és tornat pel caixer al client. El sistema emmagatzema tota la transacció monetària. 4.2. Si paga amb tarja, es validen les dades d'identitat i es passa la tarja. 4.3. Si el client no pot pagar, retorna els productes i deixa la botiga. Cal cancel·lar la transacció. 5. Si s'ha pagat, el sistema valida la transacció monetària i actualitza l'inventari de productes. 6. Es dona la factura al client, i aquest marxa de la botiga amb els productes.

Es poden indicar també els diferents **escenaris** que es poden donar en un cas d'ús. Els escenaris són ocurrències o instàncies possibles d'un mateix Cas d'Ús. Continuant amb l'exemple anterior tenim:

- **escenari A:** compra realitzada sense problemes
- **escenari B:** algun codi de producte no s'ha pogut llegir correctament amb el lector. Cal introduir el codi manualment
- **escenari C:** El client paga amb tarja de crèdit, però hi ha problemes amb la tarja o amb el datàfon. En aquest cas es pagarà en efectiu.
- **escenari D:** Si el sistema té un error de comunicacions amb el sistema de magatzem....


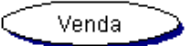

IMPORTANT:

- Els casos d'ús són purament textuais ! ! !
- El diagrama de casos d'ús només els enumera i en mostra les seves relacions.
- Per a definir millor un cas d'us ens podem ajudar d'un diagrama d'interacció.

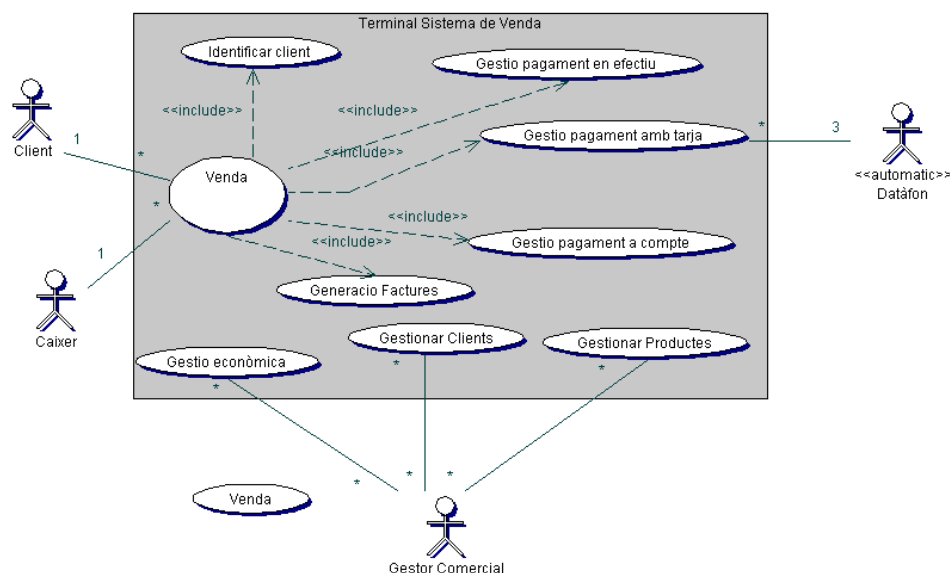
Els casos d'ús poden escriure's usant diferents nivells de detall i formalitat. Es subministra adjunt el document "plantilla use case.doc" que conté un model de documentació formal d'un cas d'ús. La plantilla està completada descrivint el cas d'ús de l'exemple anterior.

C.2- Diagrama de Casos d'Ús

C.2.1 .- Ítems bàsics del diagrama de cas d'ús

Actor	Cas d'ús	Límits del sistema (system boundary)
 Client	 Venda	
Representa un actor (persona/sistema/ens extern) que interacciona amb el sistema.	El cas d'ús que prèviament hem redactat.	Els límits del sistema que estem estudiant. Generalment marca el límit del que s'ha d'automatitzar i el que no, i ens indica la interfície del sistema !

C.2.2 .- Exemple complet



C.2.3 .- Eines d'unió dins del diagrama de casos d'ús

El DCU pot mostrar també relacions entre els diferents elements que el conformen. Bàsicament s'utilitzen:

- **Les associacions:** (relació de participació d'un actor en un cas d'ús) . Opcionalment es pot indicar sobre l'associació unes cardinalitats.
- **Les inclusions:** que permeten fragmentar un gran *Use Case* en *Use Cases* més petits. A més, aquests casos d'ús "menors" poden ser reutilitzats des d'altres casos d'ús.
- **Les extensions:** Ampliacions funcionals d'un cas d'ús per reflexar certes situacions.
- **Les herències:** << no les farem servir mai>>

Relationship	Function	Notation
association	The communication path between an actor and a use case that it participates in	—
extend	The insertion of additional behavior into a base use case that does not know about it	«extend» - - - - ->
use case generalization	A relationship between a general use case and a more specific use case that inherits and adds features to it	—>
include	The insertion of additional behavior into a base use case that explicitly describes the insertion	«include» - - - - ->

C.2.4 .- Relació d'inclusió entre Casos d'Ús

Durant algun moment de l'execució d'un cas d'ús s'executa tot el comportament associat a l'altre cas d'ús. Simplificant, podem dir que un cas d'ús es pot descomposar en varis casos d'ús. El total “inclou” les parts:

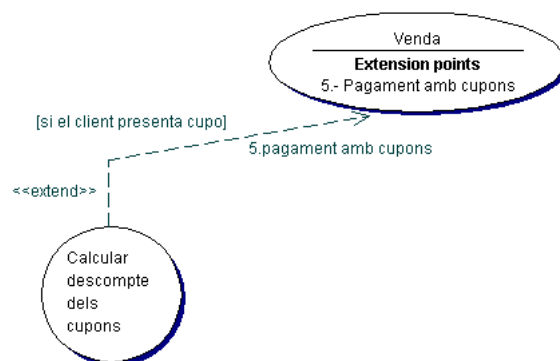


C.2.5 .- Relació d'extensió entre Casos d'Ús

Si en un moment determinat ens trobem que tenim un cas d'ús base pel que, en determinades circumstàncies, es dona un comportament que cal estendre amb un cas d'ús addicional, s'utilitza la relació d'extensió.

Tot i que es força similar al cas d'inclusió cal remarcar certs trets diferencials:

- El cas d'ús d'extensió es dona condicionalment, només en certes situacions, i caldrà indicar en el model les condicions d'aplicació.
- Considerem un cas d'ús d'extensió quan no és essencial per a resoldre el cas d'ús principal amb èxit. Són “Extres”...





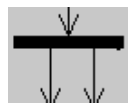

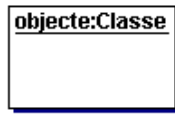
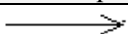
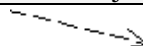
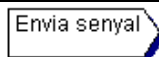
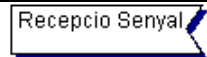


Sovint, i per simplificar, els analistes prefereixen treballar exclusivament amb **inclusions**

C.3- Diagrama d'activitats

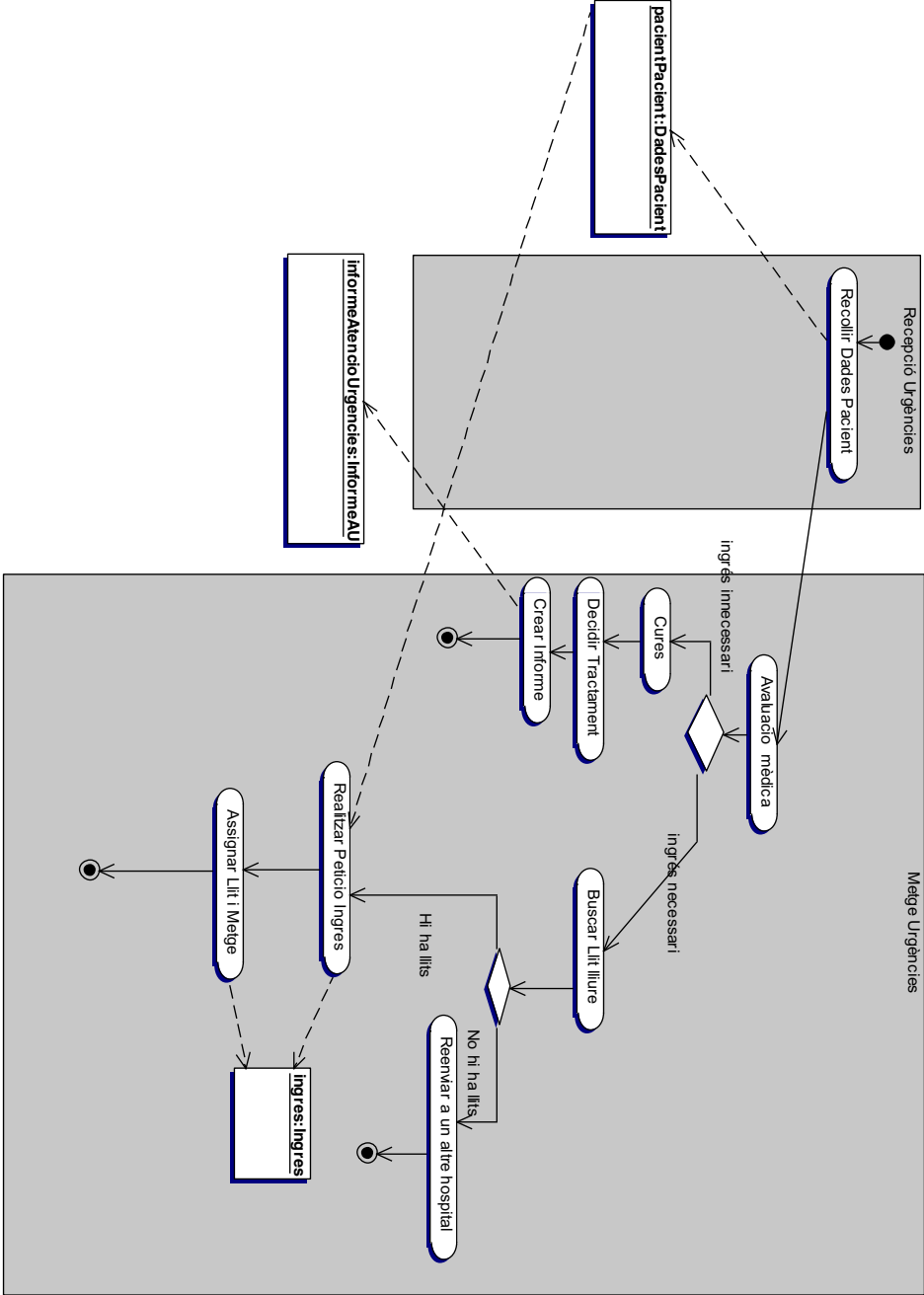
Per aquells analistes provinents de l'anàlisi estructurat, el diagrama d'activitats pot recordar en certa forma als diagrames DFD (Diagrames de Flux de Dades). En un diagrama d'activitats efectivament podem representar la mateixa informació que a un DFD, tot i que ordenada i disposada d'una forma diferent.

Els components del diagrama són:

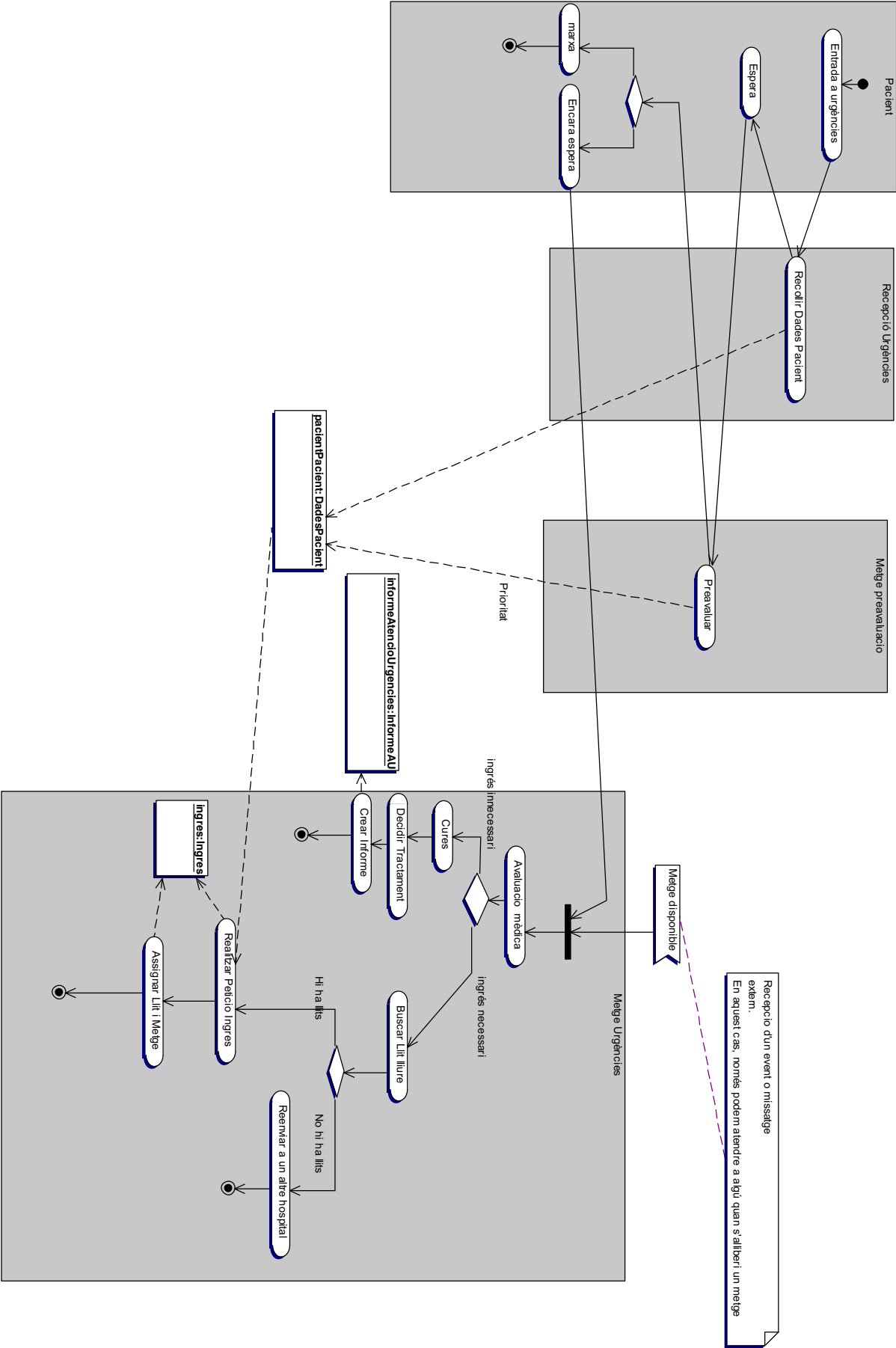
Inici	Fi	Activitat
		
Punt d'inici del diagrama	Punt de fi del diagrama	Representa un procés desenvolupat pel sistema.
Bifurcació/trobada	Fork (concurrència)	Join (concurrència)
		
Bifurcació del flux. Representa una presa de decisió. Permet també unir els fluxos divergents posteriorment.	El fork permet llançar diverses línies d'execució simultànies .	Uneix fluxos concurrents. Fins que tots els fluxos no han arribat a aquest punt, l'execució no continua.
Objecte	Línia de seqüència	Línia d'Objecte
		
És un objecte (informació) que es genera o consumeix en una activitat.	Indica quina és la següent activitat/bifurcació/fork...	Uneix una activitat a un objecte
Swimlane o Carril	Enviar senyal	Rebre senyal
		
Permet agrupar visualment totes les activitats realitzades per un mateix actor.	Envia una senyal degut a l'ocurrència d'un event.	Activa una línia d'execució al rebre un senyal o estímul extern. Quan el fluxe arriba a un receptor de senyal, ens quedem en espera fins a rebre la senyal esperada.

C.3.1 .- Exemple

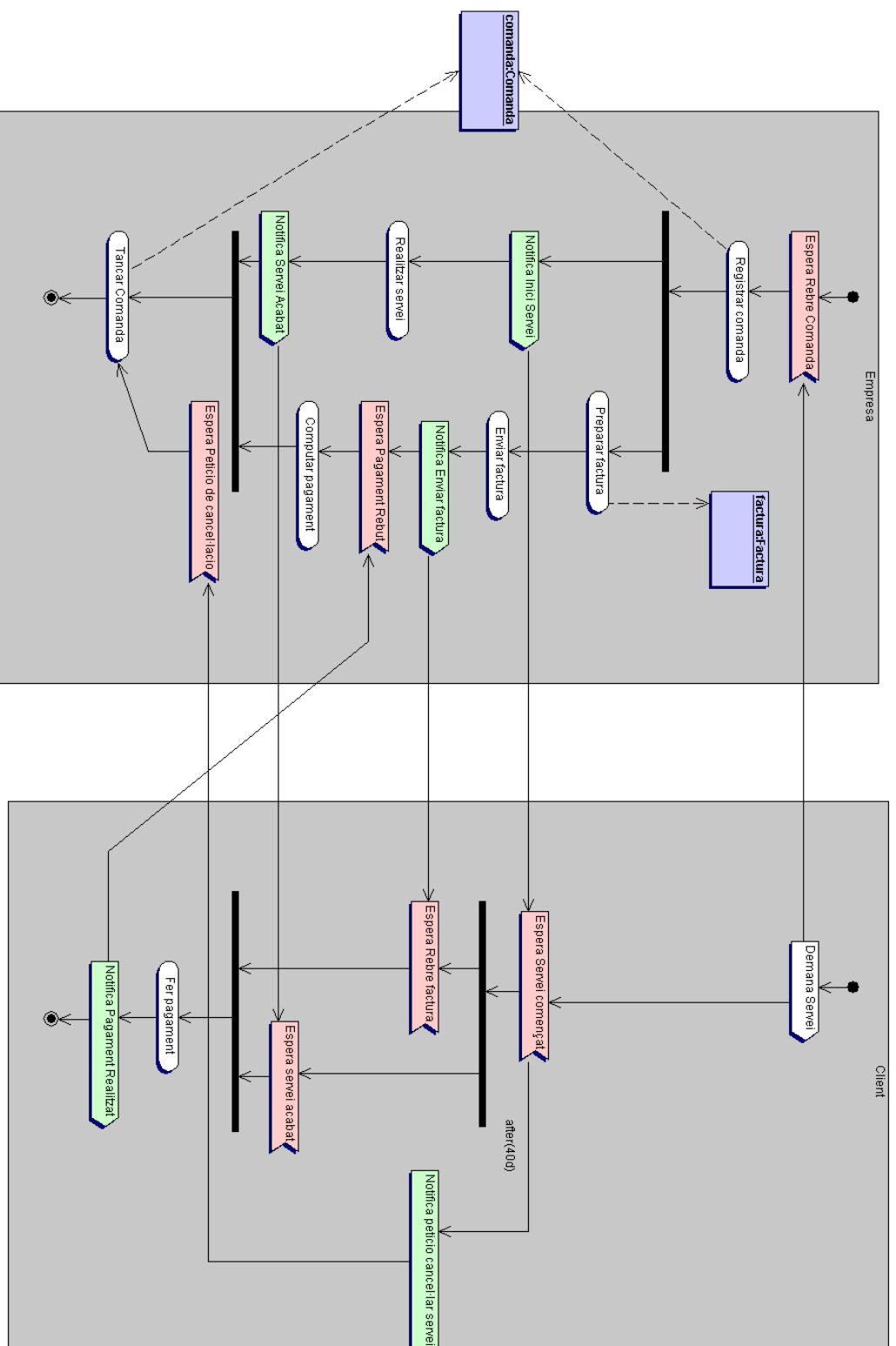
a) **Nivell de detall 1**



b) *Nivell de detall 2*

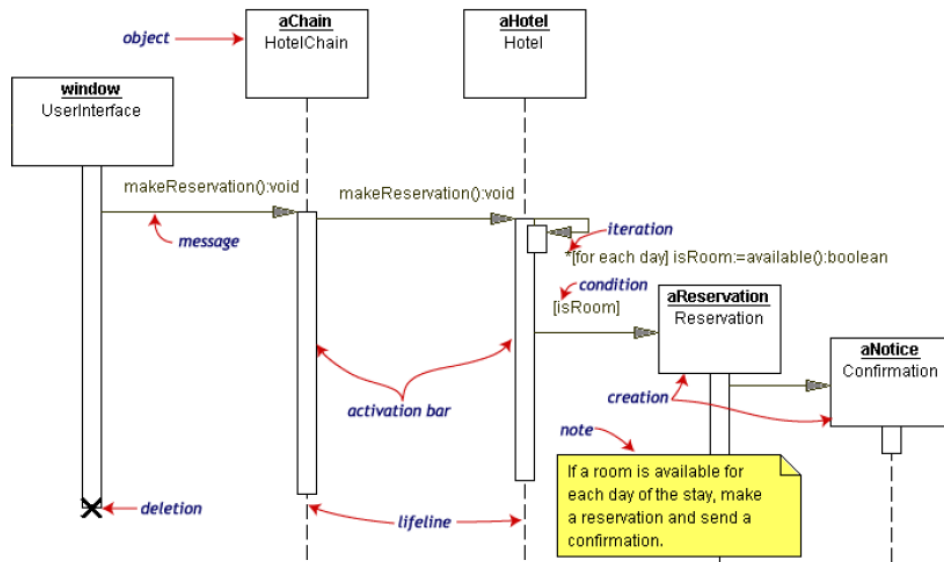


c) Exemple de gestió de concurrència



C.4- Diagrama de seqüència

C.4.1 .- Visió general



C.4.2 .- Fases del diagrama de seqüència

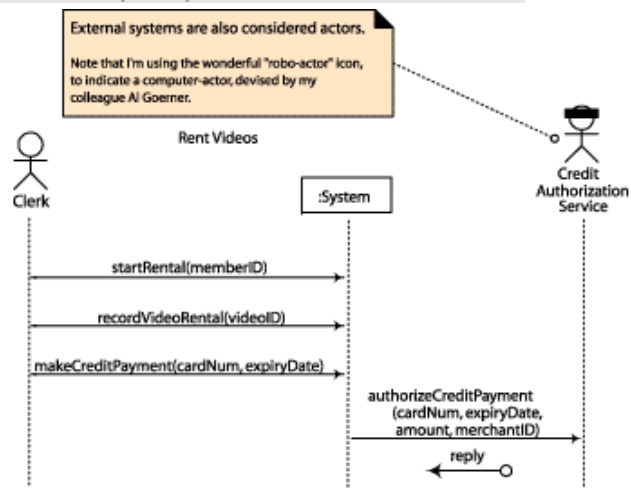
a) Diagrama de Seqüència del Sistema (DSS)

Els primers diagrames d'activitat que cal fer són del tipus *Diagrama de Seqüència del Sistema* (DSS). En ell representem la interacció del/s usuari amb el sistema format com a únic bloc (el sistema és una “capsa negra”). En ell es veuen:

- Els events que genera l'usuari
- Les respostes del sistema

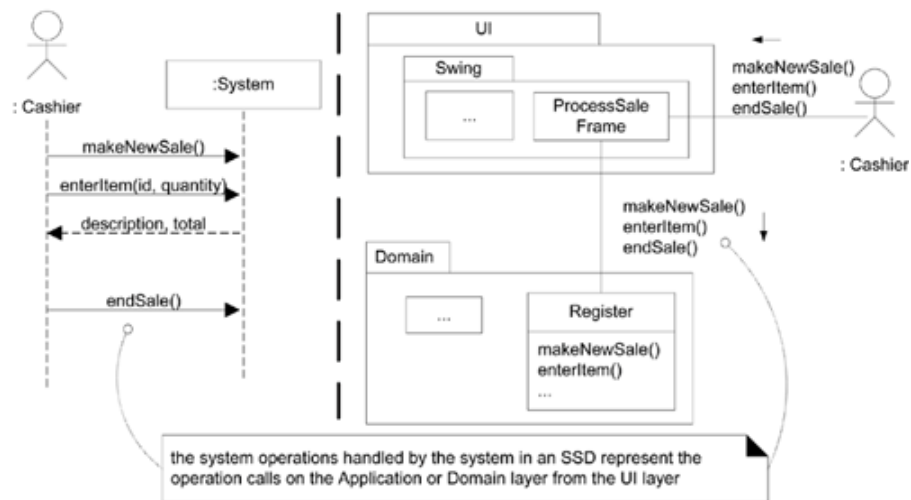
Generalment elaborem **un o més** DSS per cadascun dels casos d'ús:

- (obligatori) per a l'escenari d'èxit més comú de cada cas d'ús,
- (opcional) algun pels escenaris alternatius més freqüents de cada cas d'ús.



A la figura superior veiem el DSS d'un cas d'ús senzill : “lloguer d'una pel·lícula”. Veiem tota la interacció dels actors implicats amb el sistema com una caixa negra. En aquest cas els sistemes externs (autorització de crèdit) es consideren actors també.

Molt Important: les operacions que apareixen en el DSS entre l'actor principal i el sistema representen les crides de la capa d'Interfície d'Usuari a la capa de model de Domini:



b) Nivell de detall: Diagrama de seqüència

Un cop els DSS han estat elaborats, el pas següent es descriure en detall cadascuna de les crides del sistema. Idealment, caldria fer un diagrama de seqüència per a cadascuna de les crides de l'usuari al sistema, on es mostressin les classes del sistema que participen en l'acció i la seva interacció.

A efectes pràctics, només es faran el diagrama de seqüència de les accions més importants de cadascun dels casos d'ús.

C.4.3 .- Notació bàsica

a) Línies de vida i missatges

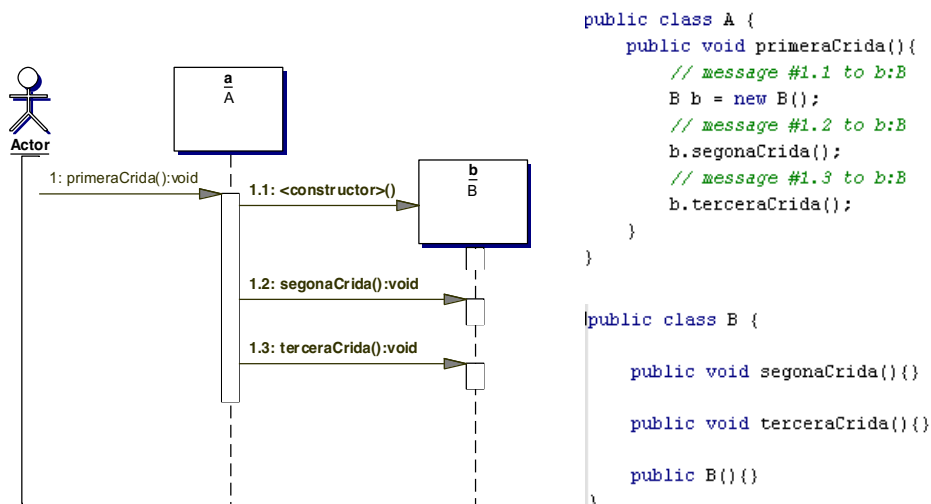
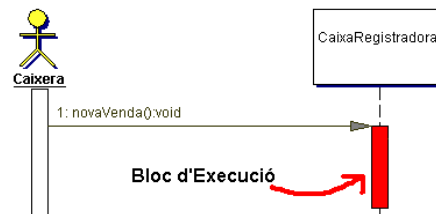
Un diagrama de seqüència es basa en la representació d'una sèrie d'objectes i actors que interaccionen mitjançant el pas de missatges. Realment aquests missatges no són més que crides a mètodes.

Podem associar una línia de vida a cada actor i objecte que intervé a la conversa. Els objectes generalment duren associada la classe a la que pertanyen. Quan donem nom a una línia de vida de tipus objecte usarem la notació *nomObjecte:nomClasse*.

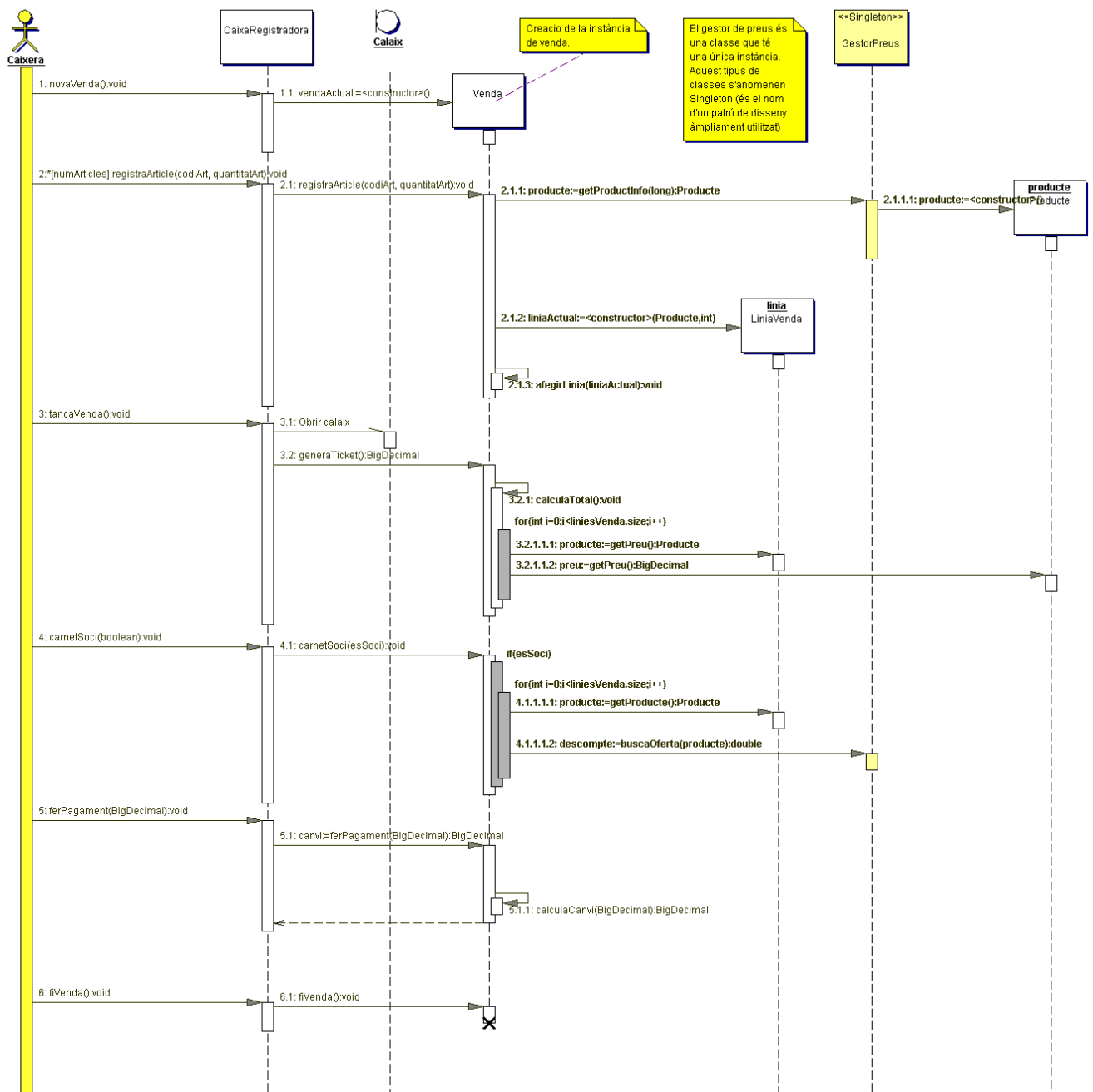
El pas de missatges Basic es representa amb una fletxa, on es mostra el nom del mètode, la signatura i el tipus de retorn. La sintaxis a utilitzar per especificar mètodes és la següent:

nomMetode(param1:tipusParam1, param2:tipusParam2.....):tipusRetorn

La fletxa que indica el pas de missatge inicia sobre la línia de vida de l'altre objecte un bloc d'execució, que representa l'**execució del codi del mètode invocat**.



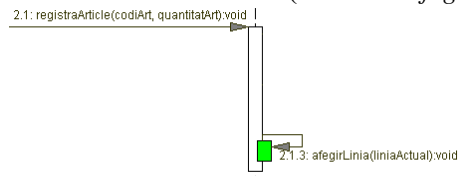
b) Exemple marc



c) Auto-Missatges o missatges a *this*

Els missatges es duen a terme entre objectes de diferents classes (i per tant entre línies de vida diferents) o bé dins del mateix objecte, també anomenat **this**. Això suposa en la representació gràfica un missatge dins de la mateixa línia de vida.

Pels automissatges, veiem que s'obre un nou bloc d'execució aniuat dins del anterior. En la figura següent es mostra un diagrama de seqüència del procés de gestió d'una compra d'un caixer de supermercat. En la línia de vida de l'objecte Venda veiem dos casos de crida a un mètode de la mateixa classe. (mètodes *afegirLinia* i *calculaTotal*)

**d) Retorn**

Cal remarcar que podem indicar també a quina variable s'emmagatzema el valor retornat per una funció, usant l'operador ":=":

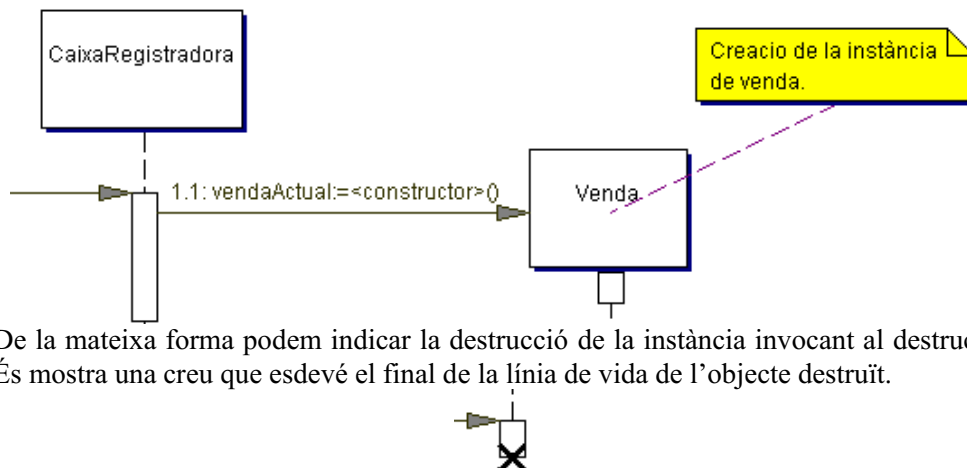
```
canvi := ferPagament(BigDecimal):BigDecimal
```

e) creació i destrucció d'instàncies

Podem mostrar la creació d'una instància usant la invocació al constructor:

```
vendaActual:={<constructor>}() // NOTACIÓ TOGETHER
```

Això es traduirà en la invocació de l'operador *new* (Java/C++) sobre la classe.



De la mateixa forma podem indicar la destrucció de la instància invocant al destructor. És mostra una creu que esdevé el final de la línia de vida de l'objecte destruït.

f) frames o blocs

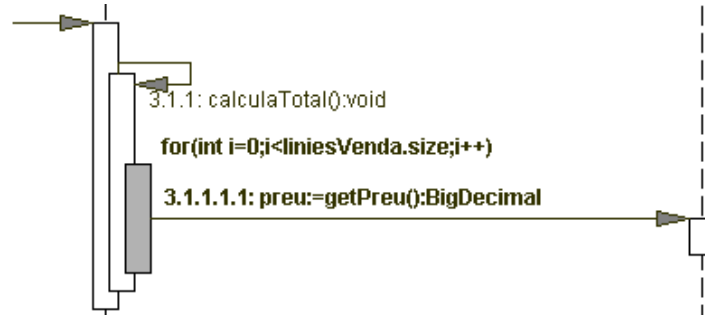
Els *frames* (marcs) o blocs són elements agrupadors que permeten representar estructures de tipus iteratiu, i condicional. Si bé la notació UML2.0 és molt concisa, cada eina CASE sol implementar-ne variants. En el cas de Borland Together les possibilitats s'agrupen en les que es mostren a continuació:

- loops: representen iteracions de qualsevol tipus (for, while, do-while)
- condicional: representen operacions if, else-if, else
- gestió d'errors: try, catch i finally

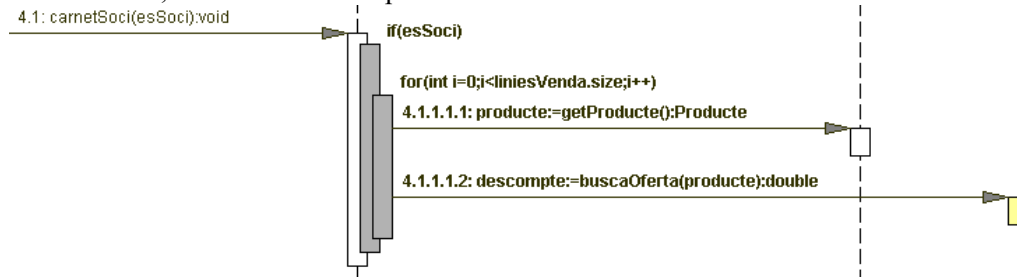
Els blocs queden aniuats dins del bloc d'execució en el que es troben, doncs formen part de l'execució del mètode.

Podem també aniuar blocs dins d'altres blocs (p.ex. fer iteracions aniuades per recórrer una matriu bidimensional)

Exemple de frame iterador dins del mètode *calculaTotal*, que recorre totes les línies de la venda:



Exemple de frame condicional amb iterador a dins. Si la persona és soci de l'establiment, es fa la cerca dels productes en oferta:



g) Crides síncrones i asíncrones

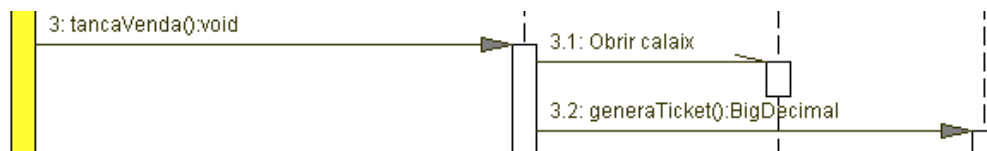
En general totes les crides representades en els diagrames de seqüència són síncrones. Això vol dir que el procés que invoca el mètode espera fins que el mètode retorna per a continuar treballant.

Hi ha però un altre tipus de missatges, els asíncrons que permeten cridar a un mètode i continuar executant el codi sense esperar-ne la finalització. Això implica l'ús de múltiples fils d'execució (*Threads*), el principal que s'encarrega d'executar el codi director, i un de secundari que es crea quan s'invoca asíncronament el mètode.

Recordeu, una crida asíncrona:

1. No bloqueja
2. Es du a terme en un fil d'execució diferent

La notació UML per a distingir entre crides síncrones i asíncrones es basa en la punta de la fletxa d'invocació del mètode. Si és completa es tracta d'una crida síncrona. Si és mitja fletxa, és una crida asíncrona:



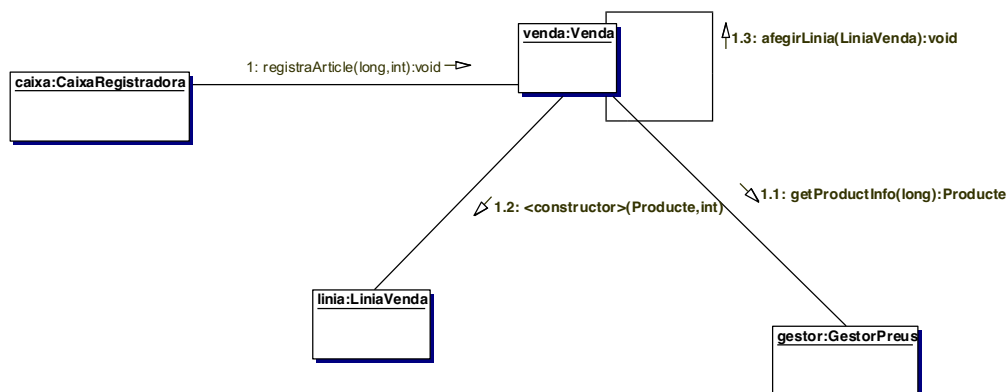
En aquest cas el mètode *tancaVenda()* (síncron) fa en primer lloc l'apertura del calaix i després genera el Ticket. Si l'apertura de calaix durés molt de temps, això no afectaria a la generació del ticket doncs *tancarVenda()* no espera a que *ObrirCalaix* hagi acabat per continuar l'execució.

C.5- Diagrama de col·laboració

El diagrama de col·laboració es pot entendre com una variant del diagrama de seqüència, doncs mostrem la mateixa informació: crides entre objectes. El canvi és fonamentalment en l'aspecte gràfic, té una organització diferent.

Si en el diagrama de seqüència teníem estrictament ordenats els objectes sobre les línies de vida i es defineix el sentit de lectura militarment d'esquerra a dreta i d'adalt a baix, en el diagrama de col·laboració tenim plena llibertat per disposar on vulguem els objectes, que típicament es disposen seguint la seqüència de la crida.

És imprescindible usar una numeració per indicar l'ordre de les crides, doncs en aquest tipus de diagrama esdevé molt difícil de llegir la seqüència. Per contra, l'avantatge que té és que veiem de forma fàcil quines classes treballen juntes i el tipus de col·laboració que s'hi estableix.



Val a dir que el diagrama que en vistes de l'equivalència en contingut dels dos diagrames, generalment els dissenyadors s'inclinen a utilitzar el diagrama de seqüència, deixant el diagrama de col·laboració per a mostrar casos senzills d'una forma més gràfica.

C.6- Diagrama d'estats

Un diagrama d'estat és un diagrama que mostra precisament els diferents estats pels que passa un sistema/objecte/component, i com es fan les transicions entre aquests estats. Els seus components bàsics són:

- **Estat:** descriu un període de temps de la vida d'un objecte de duració indeterminada, que està caracteritzat per uns certs valors o accions que s'estan realitzant en aquell període.
- **Event:** Succés real que implica un canvi d'estat. Per exemple: despenjar un telèfon.
- **Transició:** Una transició és una relació entre dos estats que indica el que passarà quan un event succeeixi, indicant el camí des de l'estat inicial a l'estat de destí. Seguint amb l'exemple, quan l'event "*despenjar telèfon*" succeeix, la transició del telèfon de l'estat "*espera*" a l'estat "*actiu*" s'activa.

Els diagrames d'estats permeten modelar màquines d'estats, que són sistemes basats en estats, events i transicions.

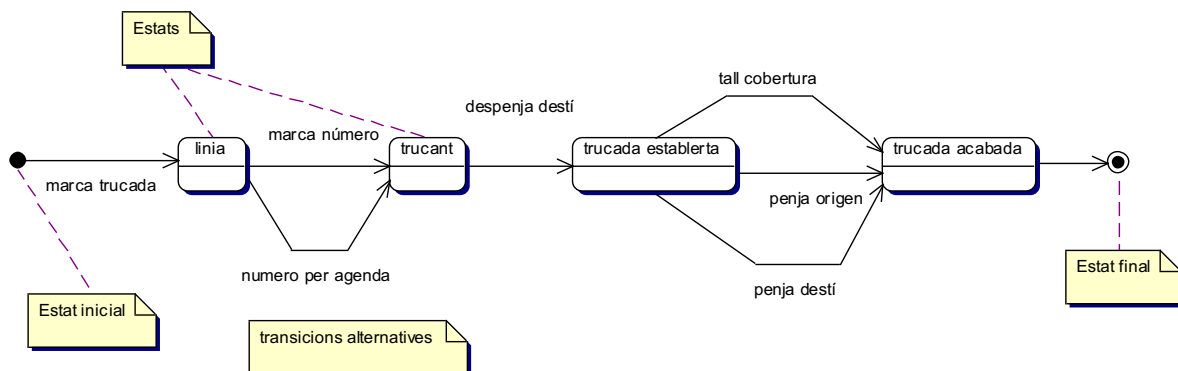
Exemples on ens pot ser útil un diagrama d'estats per a modelar el comportament:

- Protocols de comunicacions: TCP
- Flux de finestres/navegació a la interfície gràfica d'usuari.

- Objectes complexes : Comandes (hi ha molts possibles events i estats : cancel·lació, enviament, confirmació d'arribada, pagament...)

C.6.1 .- Elements del diagrama

Tota màquina d'estats té un únic punt d'inici o estat inicial, i un estat final . Ambdós són estats “ficticis” que indiquen per on cal començar i acabar de llegir el diagrama. Cada estat s'indica amb una capsula etiquetada amb el seu nom. Les transicions són les fletxes que apareixen entre els estats, i sobre elles indiquen el nom de l'event que activa la transició.



C.6.2 .- Guardes i accions

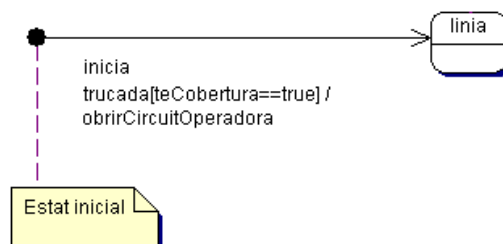
Podem detallar més la transició afegint guardes i accions:

- **Guarda:** Condició que s'ha de complir a banda de l'ocurrència de l'event per a que es pugui donar la transició
- **Acció:** Operació que desencadena la transició.

S'indica en el format següent:

```
nom_event (paràmetres_event) [condicio_guarda] / acció
```

Per exemple, aplicat a l'exemple anterior:



C.6.3 .- Tipus d'events

Hi ha diversos tipus d'events:

- Per recepció de **missatge**: l'event es dispara per la recepció d'un missatge (una crida d'un mètode)
- Per recepció de **senyal**: l'event es dispara quan succeeix un senyal com ara podria ser apretar certa tecla, un click del mouse, l'activació d'un relé, etc.
- Per **condició**: l'event es dispara quan certa condició s'avalua a *true* (p.ex. preu >100). Sintaxi:

```
when (condicio)
```

```
when (preu>100)
```

- Per **temps**: l'event es dispara en un temps concret, ja sigui absolut (demà a les 10) o relatiu (d'aquí a 30 segons)

```
after (temps)
```

```
after (5 sec)
```

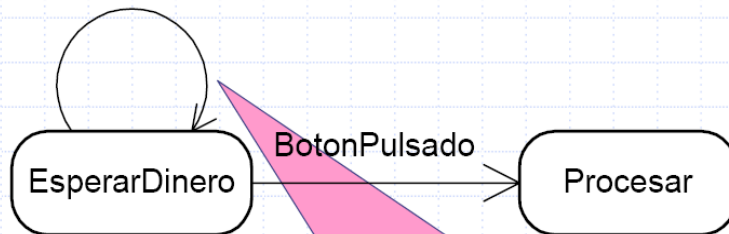
```
when ( expressió temps )
```

```
when (now >= 10/10/2008)
```

C.6.4 .- Autotransicions

Una autotransició es dona quan l'estat origen i l'estat destí d'una transició són el mateix.

MonedaIntroducida/
GuardarMoneda

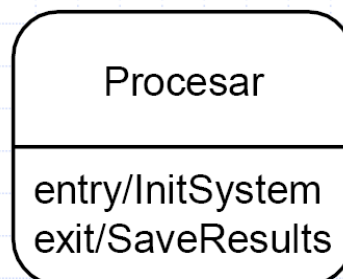


AutoTransición: Si se está en **EsperarDinero** y llega el evento **MonedaIntroducida**, ejecutamos la acción **GuardarMoneda** y volvemos al musmo estado **EsperarDinero**

C.6.5 .- Accions d'entrada i sortida

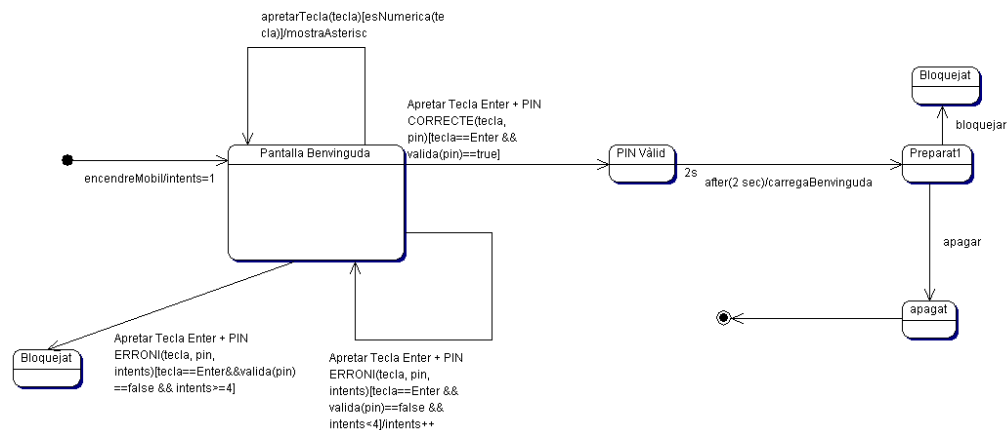
Són aquelles accions que es disparen en arribar a l'estat i al sortir-ne. Són pròpies i específiques de cada estat.

S'indiquen dins de la capsa d'estat, usant les paraules clau *entry* i *exit*, separades per una barra i el nom de l'acció que s'invoca:



C.6.6 .- Exemple complet

A continuació s'indiquen els estats propis de l'engegada d'un telèfon mòbil:



Fixem-nos que quan estem a l'estat “Pantalla de Benvinguda”, el sistema només respon als events “*Apretar Tecla*”. Quan es dispara l’event, hi ha tres transicions d’estat possibles:

1. Que sigui una tecla numèrica, i per tant es mostra un asterisc per pantalla
2. Que la tecla sigui enter, cas en el que es presenten tres variants:
 - 2.1.1. Que el número sigui el correcte, amb el que passarem a l’estat PIN vàlid
 - 2.1.2. Que el número sigui incorrecte, cas en el que:
 - 2.1.2.1. permetrem tornar a escriure el PIN (estat Pantalla benvinguda) sempre i quan no haguem superat el límit d’intents.
 - 2.1.2.2. Bloquejarem el mòbil si ja hem superat el límit.

Cal recordar que la sintaxis usada és:

Nom event (paràmetres event) [condicions a complir per a fer la transició] / accions a realitzar en la transició

Per exemple:

Apretar tecla (tecla) [esNumerica(tecla)] / mostraAsterisc(); desaTecla(tecla);

C.6.7 .- Submàquines d'estats

UML ens permet definir estats dins d'un estat, és el que es coneix com a submàquina d'estats. Hi ha dos tipus principals de submàquines:

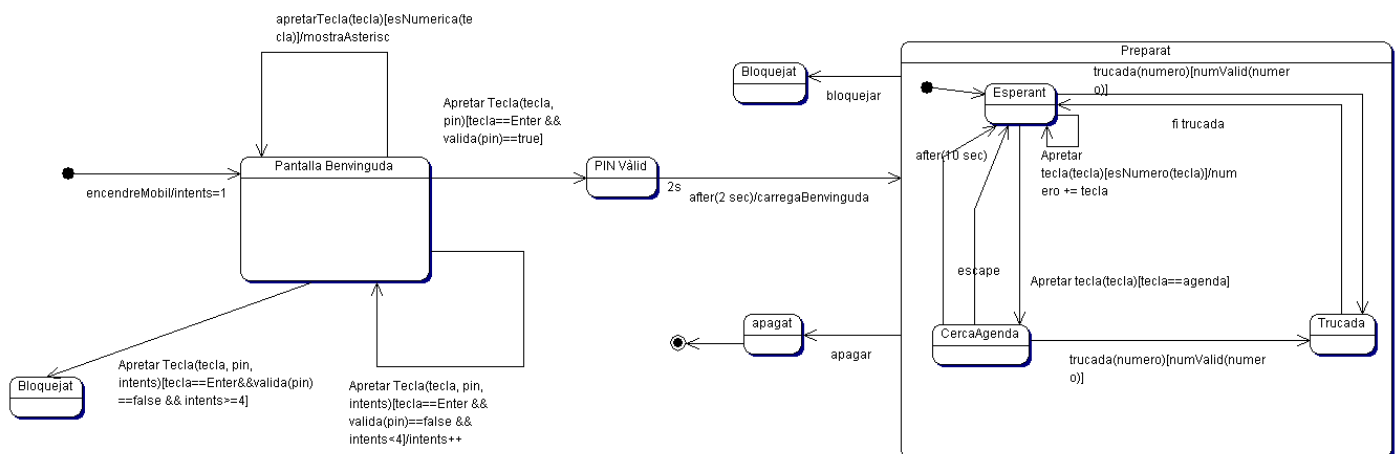
- *Disjoint substates*: Estats disjunts
- *Paral·lel substates*: Estats en paral·lel. L'estat té múltiples subestats simultànies.

a) Disjoint Substates

De vegades ens interessa començar amb estats més genèrics per, posteriorment, refinar el comportament del sistema dins d'aquests estats. Aquesta és la idea usada en les submàquines de d'estats: dins d'un estat hi ha una altra màquina d'estats !

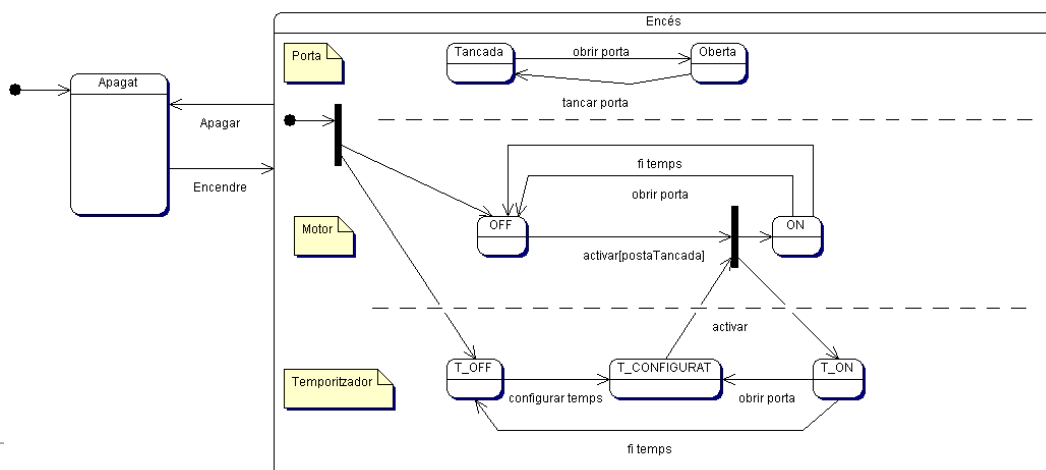
S'ha ampliat l'exemple anterior, baixant un nivell de detall extra per a l'estat "Preparat" del nostre telèfon mòbil. Quan passem de l'estat "PIN Vàlid" a "Preparat", s'inicia un estat intern indicat pel punt. De forma immediata es passa al primer estat intern "Esperant". A partir d'aquest punt ja es respon als events segons la màquina d'estats interna.

Si **en qualsevol moment** (des de qualsevol estat intern de "preparat") arriba l'event "apagar", això trenca abruptament el flux intern de l'estat, i saltarem directament a l'estat "apagat". El mateix passa per a l'estat "Bloquejat".



b) Paral·lel substates

Generalment aquestes submàquines d'estat representen varies cicles de vida que succeeixen en paral·lel dins de l'àmbit d'un estat general. Ens permeten representar diferents fil d'execució en un estat, i per tant ens faran falta eines de sincronisme: les barres fork/join que feiem servir al diagrama d'activitats.

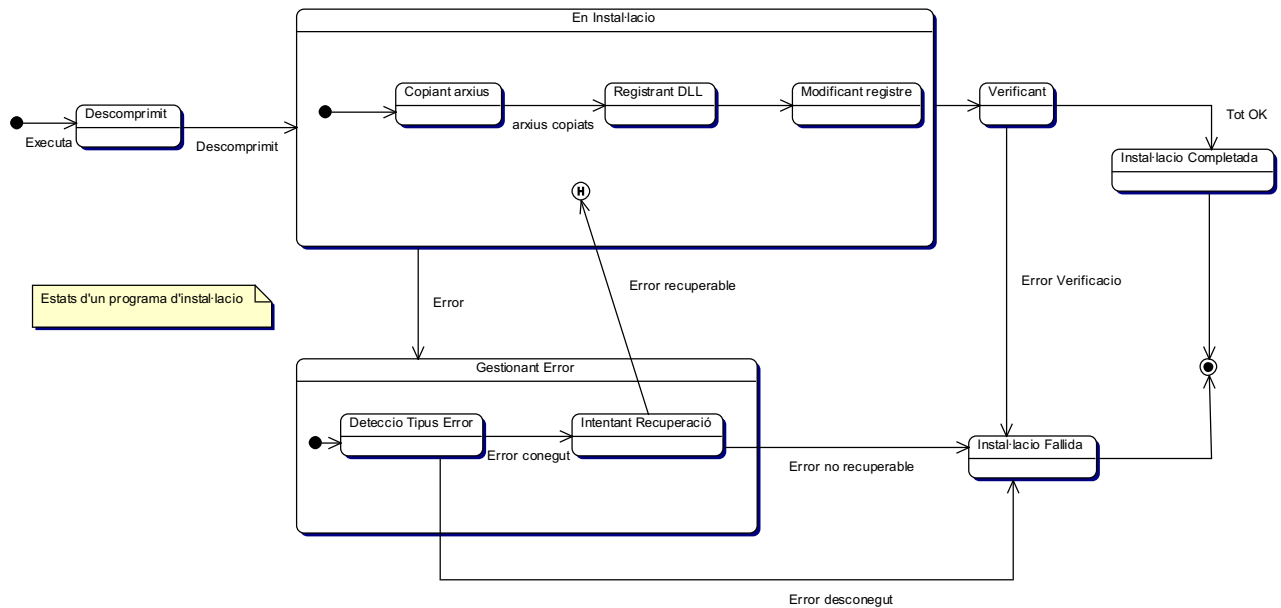


c) Indicador d'història d'estats

De vegades, per un event concret el sistema surt d'un subestat de forma abrupta. En aquestes situacions, sovint ens interessa poder tornar al punt del subestat on estàvem estat anteriorment. Per a representar això utilitzem un indicador d'història d'estats, identificat amb una icona **H**. Aquest indicador d'història és un pseudoestat que s'activa quan una transició ens fa tornar a un subestat de forma que ens redirigeix automàticament al subestat on es va aturar l'operació.

L'indicador d'història d'estats pot apuntar cap a un estat per defecte, que serà al que ens dirigirem si anteriorment no havíem passat activat mai aquella submàquina d'estats.

Veiem-ne un exemple en el diagrama d'estats d'una eina d'instal·lació de programari:



D.- Patrons de disseny

D.1- Disseny per capes

A banda de l'ús de la notació pròpia i de l'ús recomanat per a cadascun dels diagrames UML, ens trobem sovint amb la necessitat d'organitzar els elements que componen la nostra aplicació en agrupacions per tal de facilitar-ne la gestió i fragmentar la complexitat.

Un dels patrons més utilitzats avui dia és el del disseny per capes. Aquesta metodologia ens permet organitzar tots els components de la nostra aplicació, de forma que cada grup de components o capa només es comunica amb les dues capes adjacents. Això facilita el disseny del programa i ens permet tenir ben localitzats els components.

A més, si necessitem fer canvis dins d'una capa, mentre no se'n modifiqui la interfície amb les capes adjacents, el programa continuarà funcionant sense cap problema.

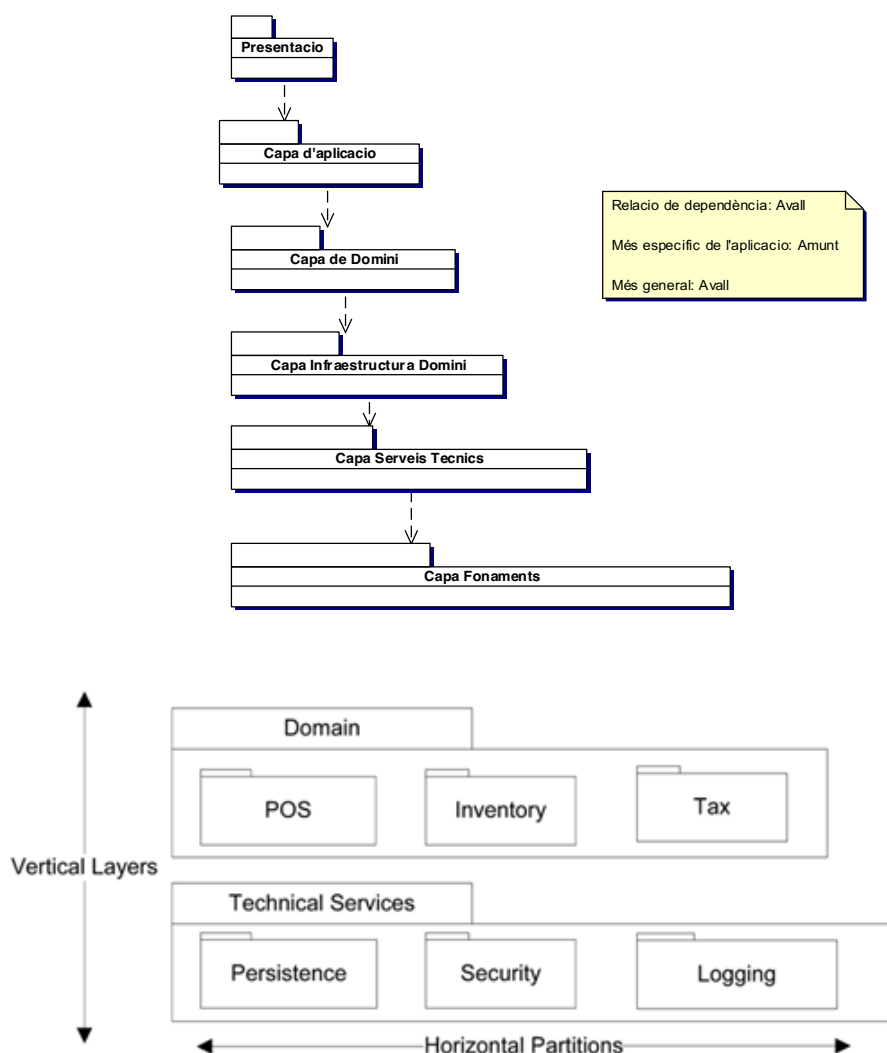
Veiem quines són les capes típiques que podem trobar a una aplicació estàndard:

Capa de Presentació o Vista (UI, GUI, User Interface, Presentation, View....)	És la capa que gestiona la generació de la interfície d'usuari, amb les complexitats específiques de cada tecnologia (Windows, Web...)
Capa d'aplicació (fluxe, mediació de procés, controlador d'aplicació)	És la capa que gestiona les peticions de la capa de presentació i controla el fluxe de pantalles. Gestiona també l'estat de la sessió d'usuari. Preprocessa les dades per a ser mostrades en la UI. Preprocessa les dades provinents de la UI per ser passades al proper nivell...
Capa de Domini (Model, lògica de negoci, lògica d'aplicació)	Gestiona les peticions de la capa d'aplicació. Controla la lògica del negoci i les regles específiques d'implementació. Aquí és on es codifiquen els comportaments específics de cada cas.
Capa d'infraestructura de Domini	Serveis de domini a baix nivell, ofereixen utilitats a tota la capa de domini. Per exemple un convertidor de divises aniria dins d'aquesta capa.
Capa de serveis tècnics	Els serveis tècnics d'alt nivell anirien aquí: <ul style="list-style-type: none"> • Persistència: Gestió de l'emmagatzemament de les classes. • Seguretat: Gestió d'accés a les classes.
Capa de Fonaments	Serveis tècnics de baix nivell, utilitats i frameworks. Estructures de dades, fils, math. Gestió I/O d'arxius, bases de dades i xarxa.

Tot i que el nombre adequat de capes dependrà de la complexitat de l'aplicació, és una bona mesura aplicar com a mínim les tres primeres i la última.

A nivell de disseny UML l'element organitzador bàsic és la carpeta, on podem col·locar els diagrames i classes relacionades amb cada capa.

A més, podem crear més nivells de subcarpetes dins de cadascuna de les carpetes de cada nivell, per tal de compartimentar millor l'anàlisi i el disseny



D.2- Capa de persistència

Separar la capa de domini de la capa de gestió de base de dades o capa de persistència és un dels patrons arquitectònics fonamentals.

Es molt poc aconsellable barrejar les consultes i el tractament de dades relacionat amb un SGBD o sistema de gestió d'arxius específic amb les regles que gestionen el nostre model de negoci. El plantejament "ràpid" de barreja-ho tot té nombrosos inconvenients:

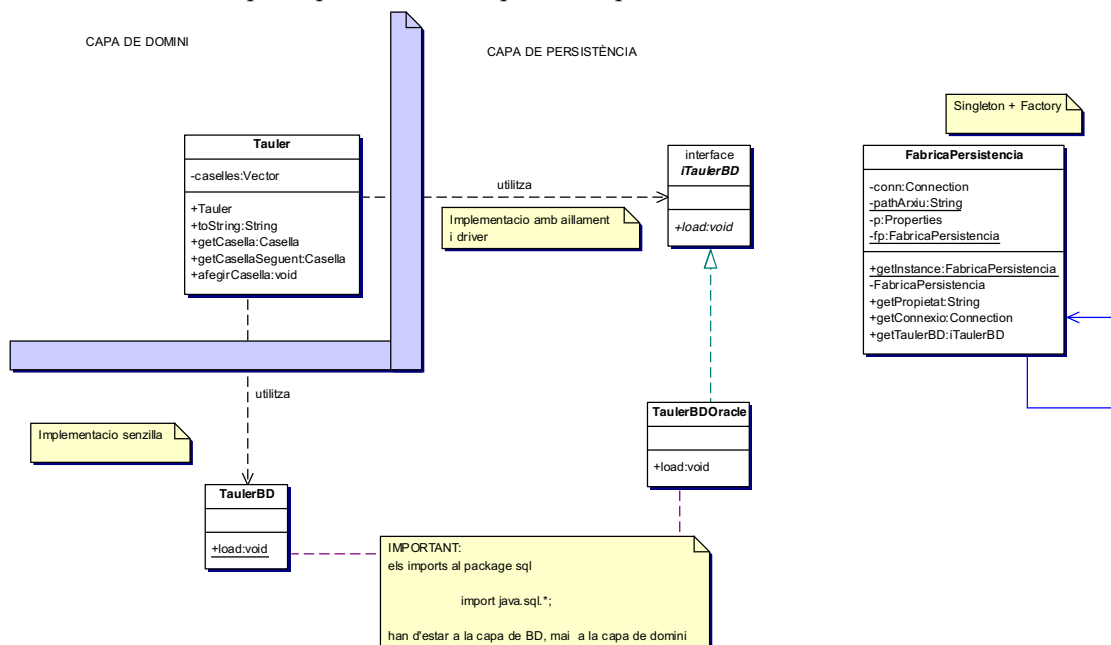
1. **DEPENDÈNCIA DEL SGBD:**
 - a. **Falta d'aïllament del fabricant d'SGBD:** Es fa molt difícil plantejar un canvi de SGBD, doncs hauríem d'apedaçar tot el codi de dalt a baix.
 - b. **Manca suport multiplataforma SGBD:** No ens podem plantejar donar suport a múltiples plataformes de SGBD simultàniament
 - c. **Falta Aïllament de l'estructura de dades:** Qualsevol canvi a l'estructura del SGBD (per exemple un canvi de nom d'una columna) ens obligarà a buscar i modificar codi que ja funcionava, corrent el perill d'espallar el que teníem.
2. **MANTENIMENT:** El codi es fa difícil de mantenir, doncs se'ns barregen conceptes propis del SGBD i de la programació del model. Tot sovint es mesclen els dos i no sabem on comença i on acaba la funcionalitat del SGBD i la del codi.
3. **REPARTIMENT DE TASQUES:** Si es fa una bona separació per capes, podem deixar la feina d'interacció amb el SGBD als especialistes del tema.

Existeixen dos enfocaments a l'hora de dissenyar una capa de persistència:

- **Implementació manual**
- **Implementació automatitzada:** Suposa l'ús d'una capa o *framework* de persistència comercial /integrat/opensource que ja ens dona un entorn de treball per la gestió de persistència. L'únic que ha de fer l'usuari és realitzar el *mapping* (mapejat) entre les seves classes i les columnes de les taules del SGBD. Aquests sistemes admeten treballar amb múltiples plataformes SGBD. Per exemple, per a Java disposem de:
 - **Enterprise Java Beans:** sistema de persistència integrat al J2EE. Utilitza la estratègia d'heredar d'una superclasse que implementa les funcions de persistència.
 - **Hibernate: framework** de persistència opensource àmpliament utilitzat. Utilitza una estratègia de introspecció o reflexió per a guardar les dades, no cal heretar de cap classe concreta.

a) Implementació manual

Hi ha multitud de possibles implementacions manuals de la persistència. Aquí es presenten dues possibilitats, una de senzilla i una de més elaborada que permet donar aïllament de la capa de persistència i suport multiplataforma.



D.3- Patrons GRASP

GRASP és l'acrònim de “*General Responsibility Assignment Software Patterns*”. GRASP són uns patrons generals de programari (“receptes”) per l'assignació de determinades responsabilitats a les classes. Encara que més que patrons, és un conjunt de recomanacions o llistat de bones pràctiques en la programació OO.

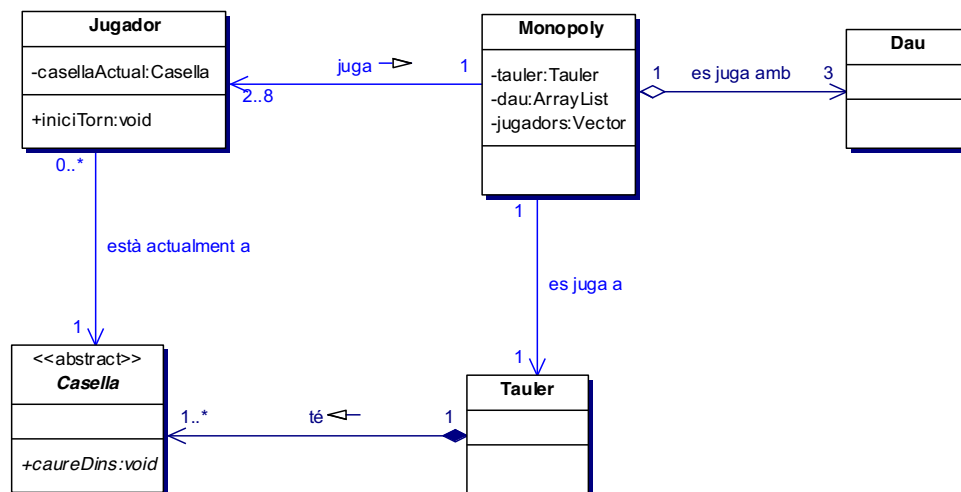
Cadascun dels patrons o recomanacions GRASP s'aplicaran sobre el cas de l'implementació del joc “Monopoly”.

D.3.1 .- Creador

El patró “creador” ajuda a identificar qui ha de ser el responsable de la creació o instanciació dels nous objectes o classes. La nova instància haurà de ser creada per la classe que:

- Té tota la informació necessària per a realitzar la creació de l'objecte, o
- Utilitza directament les instàncies creades per l'objecte, o
- Emmagatzema o manipula varies instàncies de la classe.

En el cas del joc de *Monopoly*, tenim el següent diagrama de classes:

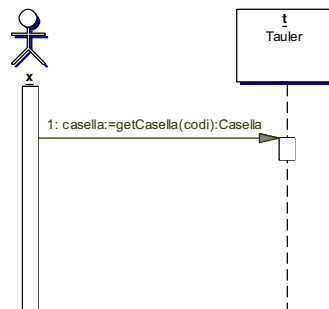


Per saber qui serà el responsable de crear les caselles (*Casella*), apliquem les recomanacions “Creador”. En aquest cas, el tauler és precisament qui conté les caselles, i per tant és un bon candidat per a ser el creador. En canvi el jugador, tot i ser usuari de les caselles, té una relació temporal.

D.3.2 .- Expert

Qui ens donarà la casella, donat un identificador? En aquestes situacions, hem de definir l'objecte que tindrà aquesta determinada responsabilitat. La classe idònia serà aquella que tingui totes les dades necessàries per a realitzar les cerques, o que tingui el camí més fàcil a aquesta informació.

Donat que en el nostre exemple, és el tauler el que té coneixement de totes les caselles, serà lògic que sigui ell mateix el responsable d'aquesta tasca.



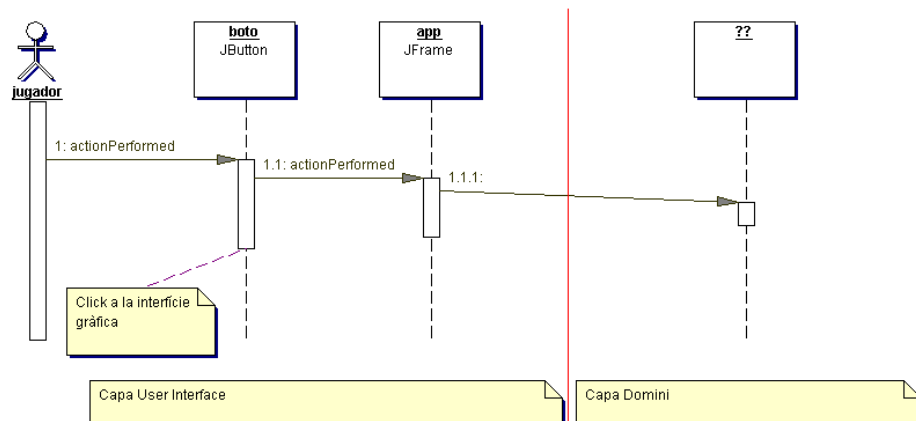
D.3.3 .- Controlador

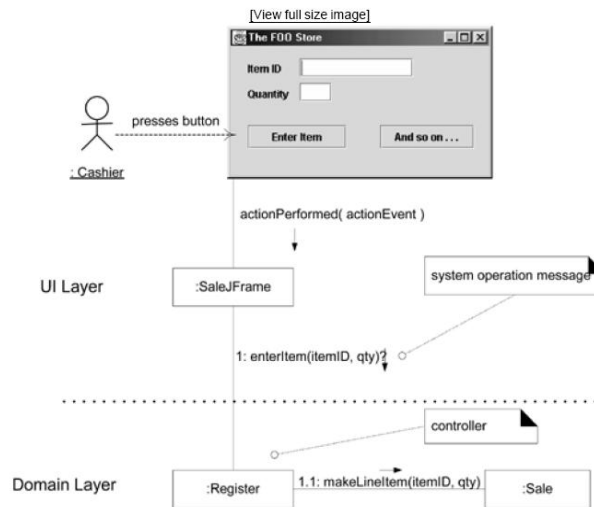
El controlador ens ajuda a prendre decisions quan arribem a un dels punts crucials en qualsevol aplicació: la interfície gràfica i la seva interacció amb el programari.

Típicament el programador dissenya de forma intuïtiva dos “capes”:

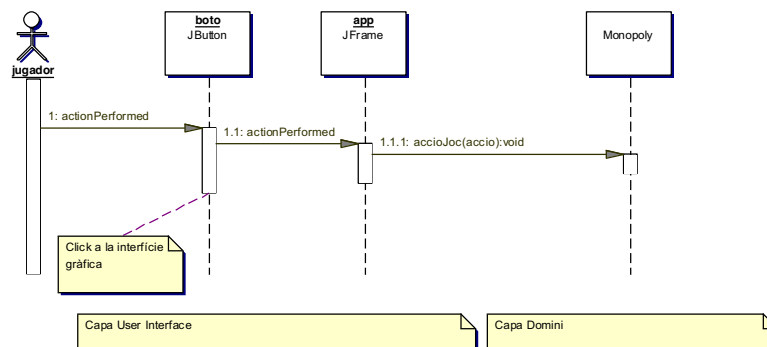
- Les classes pròpies per la generació de pantalles (JFrames, JButtons, i les necessàries per a embastar les pantalles)
- Les classes pròpies del domini

Però arribarà un punt que l’usuari premi un botó i les dues capes s’hagin de posar en contacte. Aquest patró de disseny vol respondre a la pregunta “quina serà la primera classe que es cridarà des de la capa UI per a gestionar un event ?”





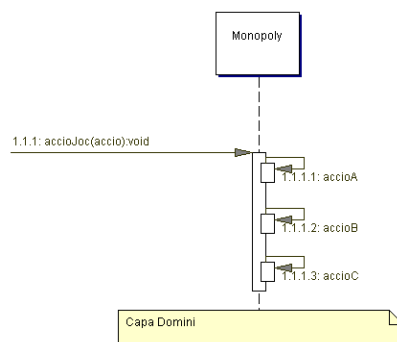
La resposta passa per localitzar/crear una classe en la capa de domini que anomenarem “Controlador”. Poden existir varies classes controlador, i fins i tot podem crear una capa de classes controlador que faci d’intermediari entre la interfície i el domini. Sovint les classes amb rol Controlador són classes artificials que tenen com a únic propòsit gestionar aquest tràfic d’events. En el nostre exemple la classe ideal és “Monopoly”, que serà el controlador principal de joc, el distribuïdor.



D.3.4 .- Alta cohesió

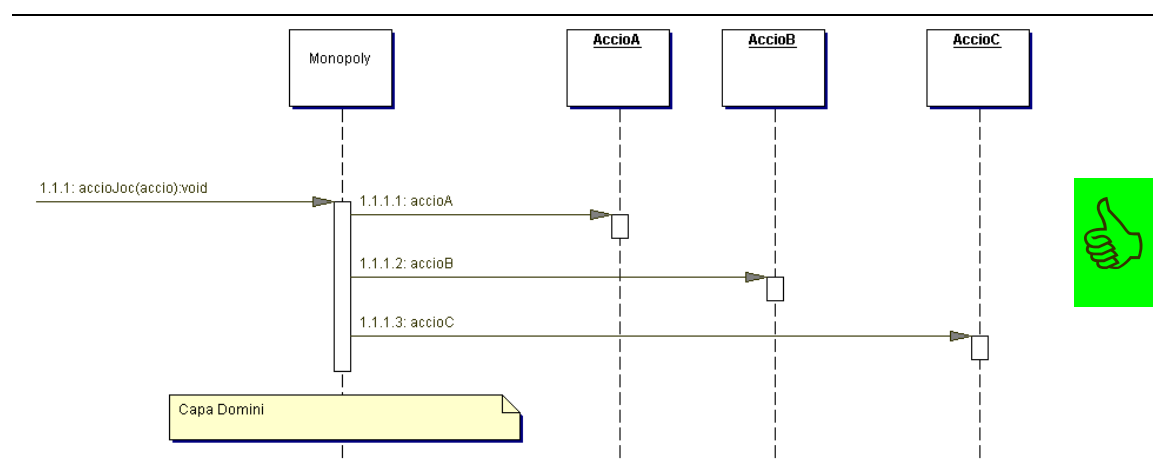
La pregunta següent és, com gestiona les accions de joc la classe Monopoly? Tenim dues alternatives:

Monopoly gestiona tots els casos: La classe creixerà i pot tornar-se inestable.



Monopoly delega la realització de les accions a d’altres classes





La versió de delegació és més flexible, i aïlla millor l'aplicació als possibles canvis i ampliacions

D.3.5 .- Polimorfisme

Una de les coses que resulta molt interessant pel dissenyador és poder crear components “plug-and-play” que ens permetin afegir noves accions al joc sense haver de modificar el codi en absolut.

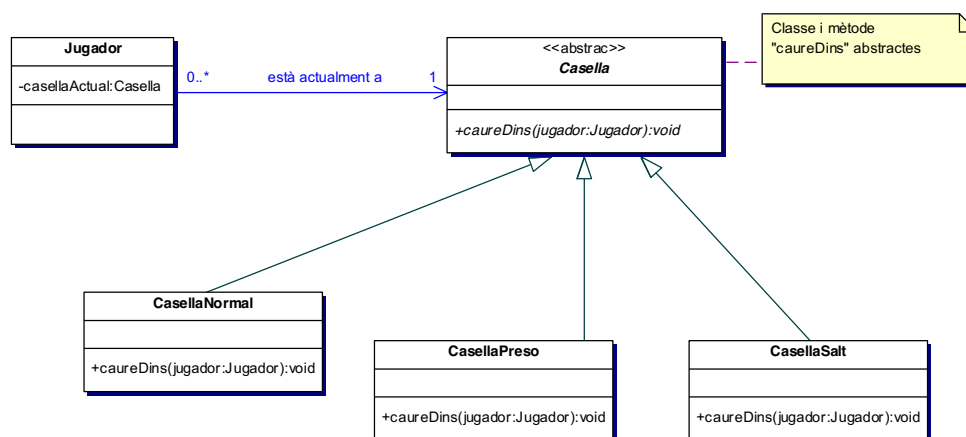
Aplicat al “Monopoly”, podríem usar aquest sistema sobre les accions provocades per les diferents caselles existents al joc. Ens agradaria poder afegir nous tipus de casella sense haver de modificar codi existent, simplement voldríem afegir noves classes al projecte que ja tenim.

Anteriorment hem fet una aproximació força millorable:

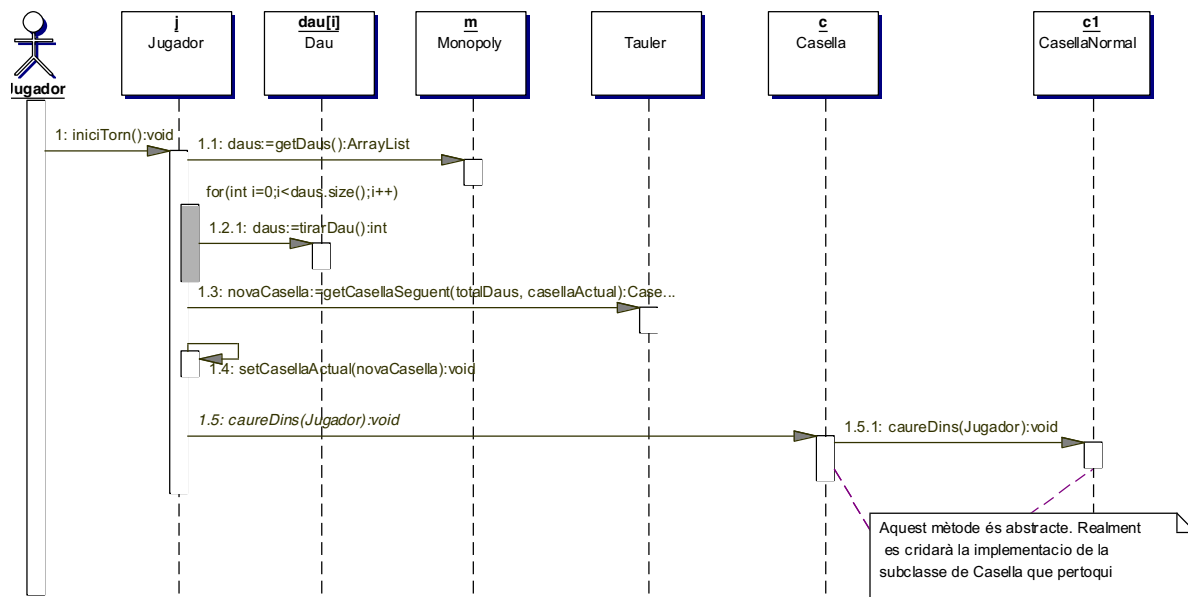
```

switch(tipusCasella)
{
    case 1: casellaNormal(); break;
    case 2: casellaPreso(); break;
    case 3: casellaPremi(); break;
    // ....
}
  
```

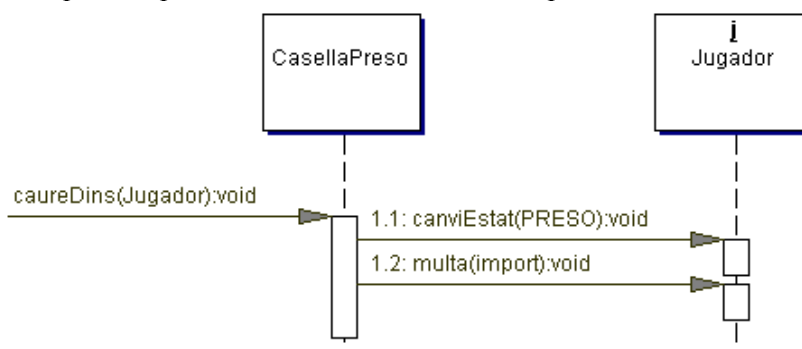
El polimorfisme ens dona una alternativa d'implementació que ens permetrà afegir tants tipus de casella com vulguem, però sense haver de tocar el codi existent. La proposta de solució és la següent:



Veiem com evoluciona temporalment aquesta estructura:



Exemple d'implementació d'una subclasse de tipus casella:



D.3.6 .- Classes fabricades

Arriba un punt en el que necessitem afegir funcionalitats complexes, sovint de caràcter tècnic a alguna classe.

En aquest cas segons el patró “Expert” hauríem d'assignar-les a aquell objecte que tingues la informació que cal manipular. Això de vegades esdevé poc pràctic o recomanable, doncs aquestes funcionalitats podrien ser reutilitzades per d'altres classes i si les assignem i tanquem dins d'una classe específica tanquem aquesta via.

A més podem tenir el problema de barrejar dins d'una classe les regles del negoci i accions de caràcter tècnic que poden enfosquir molt el codi.

Quina és la solució? Doncs “fabricar” una classe.

El cas paradigmàtic de classe fabricada són aquelles per a gestionar la persistència dels objectes.

Imaginem que volem salvar l'estat de la partida de Monopoly, i continuar jugant demà.

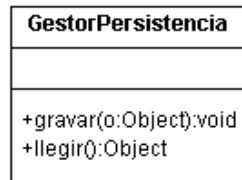
Això passa per guardar les dades de tots els jugadors a disc.

Si escrivim el codi directament dins de la classe jugador, estem :

- Augmentant la complexitat de la classe

- Afegint codi que no es propi del domini del problema
- Afegir acoblament, doncs treballem amb una nova interfície: APIs de bases de dades (com per exemple JDBC a tecnologies Java)

Solució: creem una nova classe *GestorPersistencia* que s'encarregui de desar els objectes a disc:



D.4- Patrons de disseny GoF

D.4.1 .- Adaptador o driver

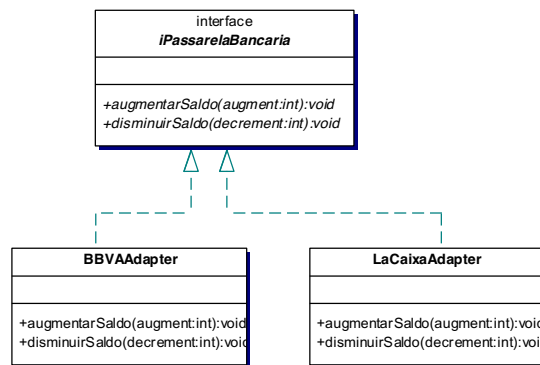
Imaginem que volem fer una “perillosa” versió de Monopoly que treballi amb diners de veritat. Inicialment els jugadors aportarien les seves dades bancàries i el sistema faria transaccions bancàries reals sobre els comptes.

En aquest cas ens trobarem que en funció de l'entitat bancària, la forma d'accedir als sistemes de pagament remots és diferent. Però, i si volem donar suport a varies dins del joc?

El patró ideal en aquest cas és l'adaptador o *driver*. Consisteix en els següents passos:

- Crear una interfície senzilla, que serà la que la nostra aplicació utilitzarà per cridar als serveis externs. La interfície és la mateixa siguin quines siguin les entitats bancàries que hi hagi al darrera. En aquest cas només necessitem dues operacions : *augmentarSaldo*(*augment:int*):void i *disminuirSaldo*(*decrement:int*):void)
- Crear classes adaptador que implementin la interfície anterior. Hi hauran tantes classes adaptador (*drivers*) com entitats bancàries haguéssim de connectar.

La cosa quedaria així:

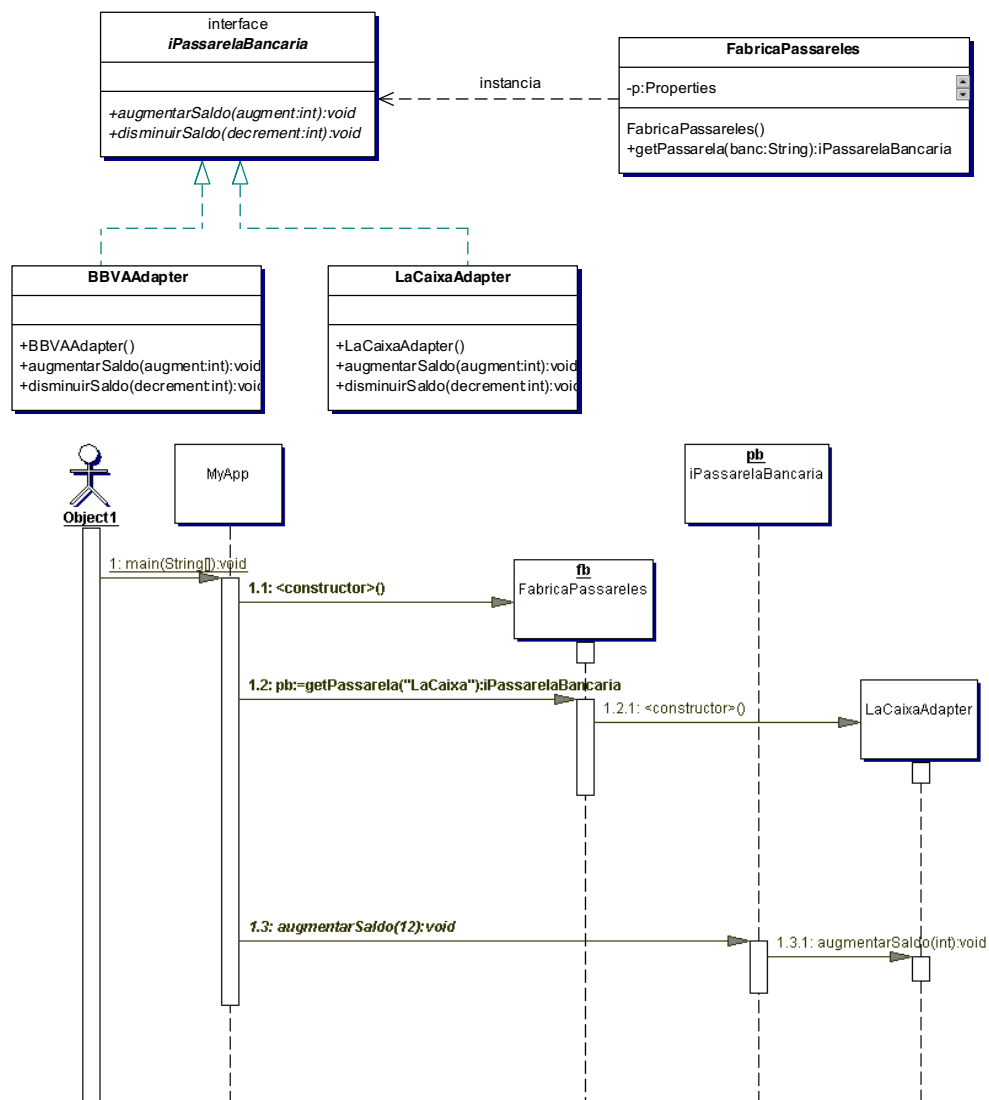


D.4.2 .- Factory

Seguint amb la mateixa problemàtica del patró anterior, encara no s'ha definit qui seria el responsable de crear/instanciar les classes adaptador. Sembla lògic pensar que hi haurà una classe que s'encarregarà de crear una nova instància de la classe adaptador que pertorqui en cada cas. Per exemple, si el jugador té compte al BBVA, aquesta classe hauria de crear una instància del BBVA adapter.

El nom per aquesta nova classe és *factory* o fabrica de classes. En la implementació d'aquesta classe solem usar la càrrega dinàmica de classes. Obtenim el nom de la classe

a carregar des d'una propietat del sistema i després carreguem la classe de forma dinàmica:



🔗 Implementació de la classe *factory*:

```

import java.io.*;
import java.util.*;

public class FabricaPassareles {

    private Properties p;
    private static String pathArxiu = "propietats.txt";

    FabricaPassareles() throws Exception
    {
        try {
            ClassLoader loader = ClassLoader.getSystemClassLoader();
            //aquest mètode carrega la classe des d'un path relatiu al CLASSPATH...ens evita posar paths absoluts.
            InputStream arxiuConfiguracioIS= loader.getResourceAsStream (pathArxiu);
            if(arxiuConfiguracioIS==null) throw new Exception ("Arxiu de propietats no trobat:"+pathArxiu);
            p = new Properties(); // Crea un nou contenidor de propietats
            p.load(arxiuConfiguracioIS); // Carrega les propietats de l'arxiu
        } catch (java.io.FileNotFoundException e) {
            throw new Exception("No he pogut trobar l'arxiu "+pathArxiu);
        } catch (java.io.IOException e) {
            throw new Exception("Error I/O");
        }
    }

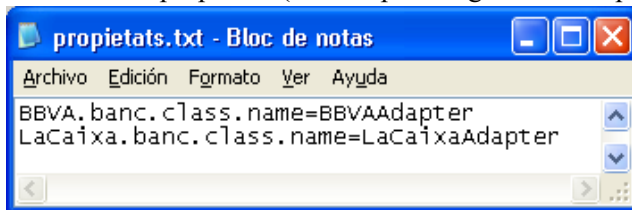
    public iPassarelaBancaria getPassarela(String banc) throws Exception
    {
        // Recuperem el nom de la classe de les propietats carregades
        String nomClasse = p.getProperty(banc + ".banc.class.name");
        // verifiquem que el nom de la classe existeixi
        if(nomClasse == null ) throw new Exception("L'adaptador pel banc "+ banc +" no està disponible");
        iPassarelaBancaria iPb;
        try
        {
            iPb = (iPassarelaBancaria) Class.forName( nomClasse ).newInstance();
        }
        catch(Exception e)
        {
            throw new Exception("Error al carregar l'adaptador del banc "+ banc );
        }
        return iPb;
    }
}

```


Exemple d'ús de la *factory*:

```
public static void main(String args[]) throws Exception
{
    FabricaPassareles fb= new FabricaPassareles();
    iPassarelaBancaria pb = fb.getPassarela("LaCaixa");
    pb.aumentarSaldo(12);
}
```

Arxiu de propietats (a la carpeta origen del classpath):



D.4.3 .- Singleton

Si ens fixem en la implementació que hem fet, cada cop que necessitem construir un adaptador haurem de crear una instància de la fabrica, i llavors invocar el mètode *getPassarela(String)*.

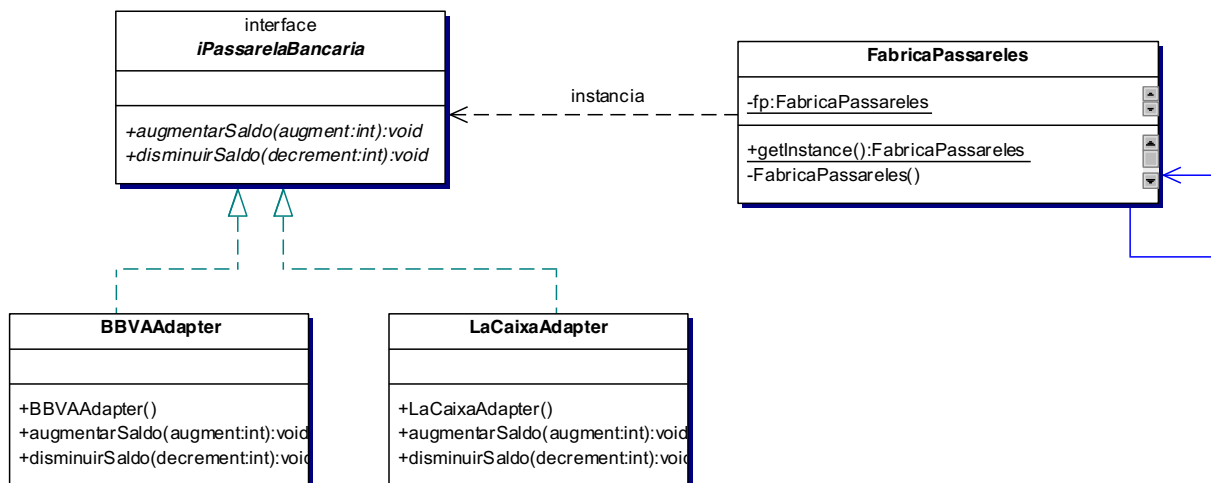
Això comporta un baix rendiment, doncs cada vegada tornem a carregar l'arxiu de propietats.

Una possible solució és forçar a que sempre hi hagi una única instància de la classe *Factory*, de forma que no sigui necessari tornar a inicialitzar cada cop la instància. Això ens obligarà a donar un únic punt d'accés a la classe.

Un *singleton* precisament és una classe que aconsegueix aquestes dues propietats:

- Només té una única instància, i el constructor de la classe és privat.
- Per accedir a ella passarem per un mètode estàtic o de classe.

Aplicat al cas anterior la millora seria la següent:



La classe *FabricaPassareles* contindrà ara una instància privada i estàtica de *FabricaPassareles*, que serà el SINGLETON (la instància única). Podrem accedir a aquesta instància cridant el mètode estàtic *getInstance ()*, que ens tornarà la instància única compartida per tothom.

Exemple d'ús de la *factory* convertida en *singleton*:

```
public static void main(String args[]) throws Exception
{
    FabricaPassareles fb= FabricaPassareles.getInstance();
    iPassarelaBancaria pb = fb.getPassarela("LaCaixa");
    pb.augmentarSaldo(12);
}
```

Marcats en vermell tenim els passatges modificats/afegits a la classe *FabricaPassareles* per a que implementi el patró *singleton*:

```
public class FabricaPassareles {

    private static FabricaPassareles fp; // SINGLETON o instància única de la classe

    private Properties p;
    private static String pathArxiu = "propietats.txt";

    // únic punt d'accés a la instància de la classe
    public static FabricaPassareles getInstance() throws Exception
    {
        if(fp==null) {fp = new FabricaPassareles();}
        return fp;
    }

    /** @link dependency .....
    /** iPassarelaBancaria lnkiPassarelaBancaria; */

    // Atencio, en un SINGLETON, el constructor és privat, per forçar a
    // passar pel getInstance()
    private FabricaPassareles() throws Exception
    {
        try {
            ClassLoader loader = ClassLoader.getSystemClassLoader();
            //aquest mètode carrega la classe des d'un path relatiu al CLASSPATH...ens evita posar paths absoluts.
            InputStream arxiuConfiguracioIS= loader.getResourceAsStream (pathArxiu);
```

Altres patron GoF....

- Strategy
- Composite
- Facade
- Observer