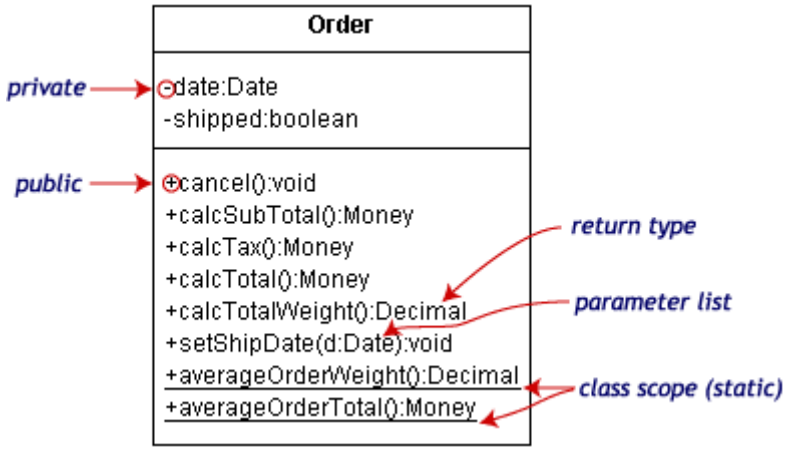
Classes

Amb UML una classe es representa per mitjà d'un rectangle dividit en tres parts que contenen:

- la part superior: el *nom* de la classe;
- la part central: els *atributs* de la classe;
- la part inferior: les *operacions* de la classe (*mètodes* a la fase de disseny).



Els atributs

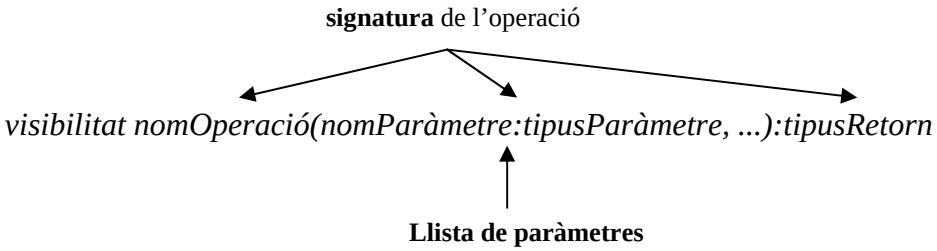
La sintaxi d'un atribut és:

visibilitat nomAtribut: tipus = valorInicial

on *nomAtribut* és la única part obligatòria.

Les operacions

La sintaxi d'una operació és:



En Java i C++ el *tipusRetorn* és ignorat.

Quan un objecte rep un missatge, en compara la signatura amb la signatura de les seves operacions, i si troba una coincidència llavors invoca la corresponent operació.

La representació d'una classe acostumen a ometre les operacions de creació/destrucció i set/get.

Visibilitat d'atribut i d'operació

La visibilitat d'un atribut o d'una operació indica com aquests poden ser accedits des d'objectes d'altres classes. La visibilitat s'indica situant davant de l'atribut o de l'operació un dels símbols de la taula següent :

Símbol	Accés	Accessibilitat per a
+	Públic	Tots els objectes de l'aplicació
-	Privat	Instàncies de la pròpia classe
#	Protegit	Instàncies de la pròpia classe i les seves subclasses
~	Paquet	Tots els objectes del mateix paquet

Amb UML, si la visibilitat no s'indica explícitament, se sobreenten que els atributs són privats i els mètodes públics.

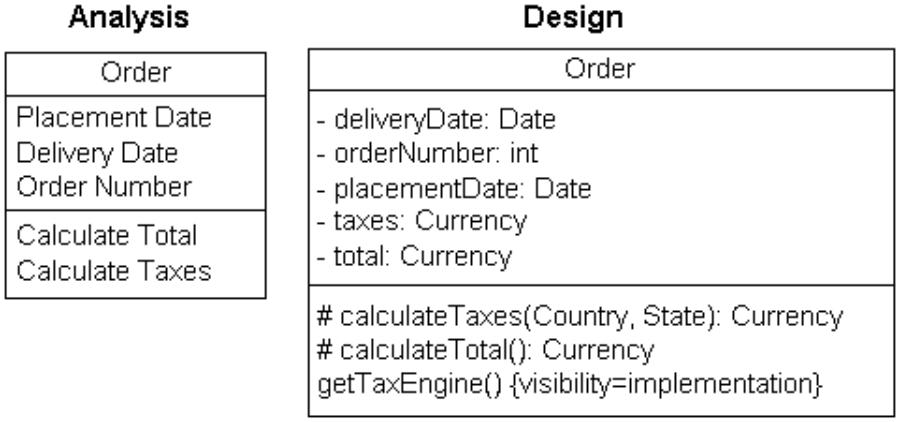
Àmbit

S'indica que una operació és operació de classe subratllant-la. Si una operació no està subratllada se sobreentén que és operació d'objecte.

Classes abstractes

Una classe abstracta és una classe de la qual no se'n poden instanciar directament objectes. Per tant els objectes lligats a una classe abstracta s'han de crear necessàriament en alguna de les seves subclasses (especialització). Una classes abstracta s'acostuma a posar en *itàliques*.

Convé observar que depenent de la fase de desenvolupament del programari en la qual es fan servir, la representació de les classes és més o menys completa.



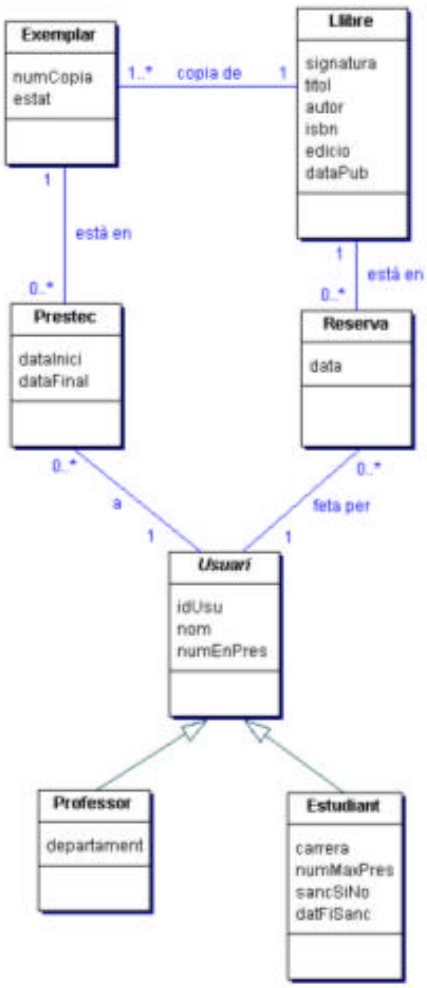
Diagrames de Classes

Un diag*rama de classes* conté:

- classes;
- relacions entre classes.

Una *relació* entre classes es representa en el diagrama per un segment que les connecta.

Exemple (inicial) per a la gestió d'una biblioteca universitària:



Els diagrames de classes es fan servir per a:

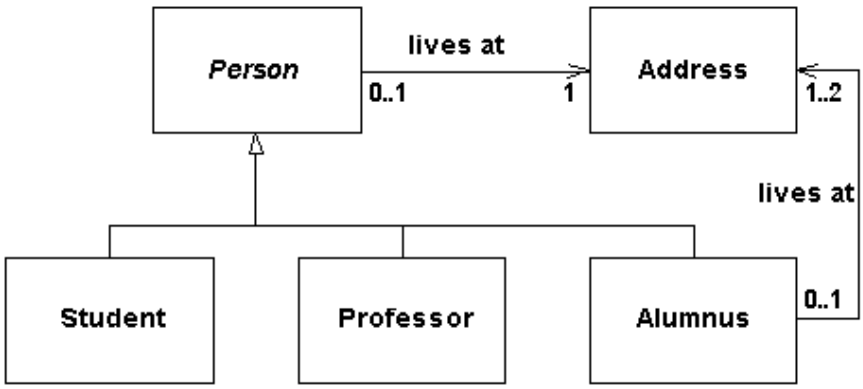
- explorar conceptes del domini del problema a la fase de definició de requeriments;
- analitzar/especificar requeriments a l'hora de construir el model d'anàlisi;
- descriure detalladament el programari a construir a la fase de disseny.

A l'igual que passa amb les classes, la representació dels diagrames de classe és més o menys completa depenent de la fase de desenvolupament del programari en la qual es fan servir.

Un diagrama de classes proporciona una visió estàtica del sistema a desenvolupar:

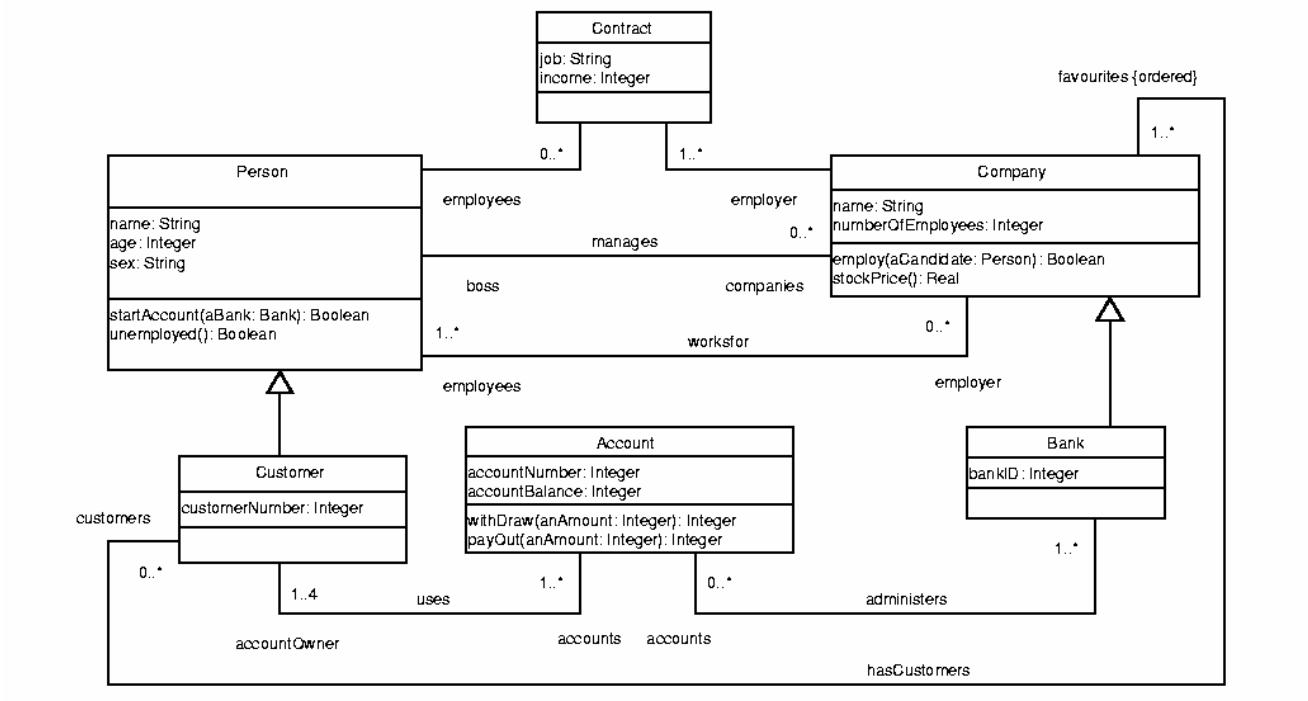
- mostra les classes que interactuen;
- no mostra què passa quan les classes interactuen

Generalització/Especialització: és una relació d'herència en la qual una classe és superclasse (subclasse) de l'altra. Una generalització/especialització es representa per un segment amb un extrem triangular apuntant cap a la superclasse.



En el diagrama anterior s'ha afegit una relació entre **Alumnus** i **Address** perquè té una multiplicitat (veure més endavant) diferent de la que tenen les relacions entre **Student** i **Professor** amb **Address**.

A continuació donem un altre exemple de diagrama de classes:



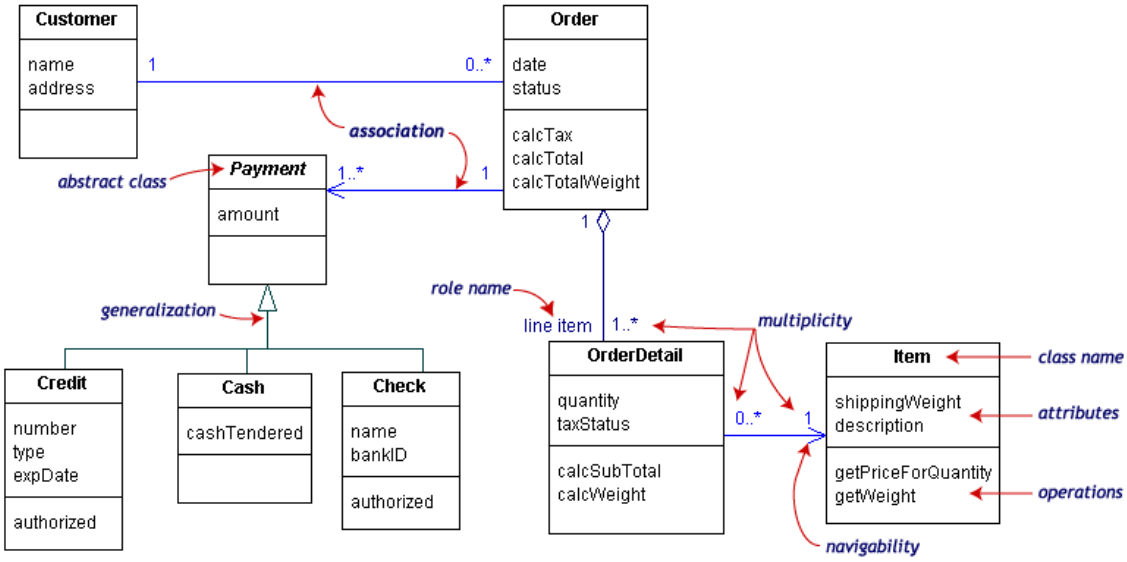
Existeixen tres tipus de relacions:

- *Associació*: interrelació entre instàncies de dues classes. Una instància d'una classe ha de disposar d'informació d'una instància de l'altra classe per a poder fer el seu treball. A un diagrama, una associació es representa per mitja d'un segment que connecta les dues classes.

- *Agregació*: una associació en la qual es distingueixen un tot i una part: una instància d'una classe (tot) es relaciona amb una col·lecció d'instàncies de l'altra classe (part). Una agregació es representa per un segment amb un extrem en forma de diamant apuntant cap a la classe singular.

- *Composició*: una associació en la qual una instància d'una classe (feble) no pot existir sense estar relacionat amb una instància d'una altra classe. Una composició es representa per un segment amb un extrem en forma de diamant ple apuntant cap a la classe no feble.

Exemple de gestió d'una comanda:



A continuació descriurem amb més de detall el tres tipus de relacions:

Associació

A un dels dos extrems d'una associació s'hi pot posar el nom d'un rol per a clarificar la natura de l'associació.

A una associació se li pot afegir una fletxa per a mostrar el sentit (de navegació) en el qual es poden fer consultes. Una fletxa serveix per a mostrar qui és el "propietari" de l'associació. Només es poden enviar missatges en la direcció de la fletxa. Les associacions sense sentit de navegació són bidireccionals.

La *multiplicitat* situada a un extrem d'una associació indica el nombre de possibles instàncies de la classe d'aquest extrem que es poden associar amb una instància de la classe de l'altre extrem.

A continuació donem una taula que mostra les multiplicitats més comunes:

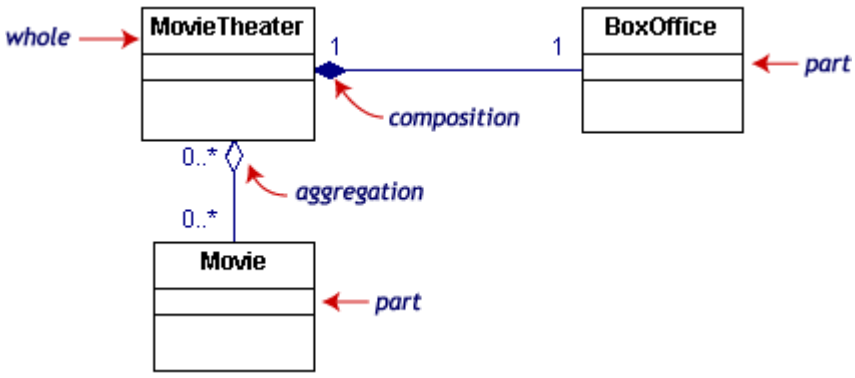
Multiplicitats	**Significat**
0..1	Zero o una instàncies
n ..m	De n a m instàncies
0..* or *	Nombre indeterminat d'instàncies (el zero inclòs)
1	Exactament una instància
1..*	Al menys una instància

A un diagrama de classes hi han de figurar obligatòriament: les classes, les associacions i les multiplicitats. El sentit de navegació i els rols es poden posar de forma opcional per a fer més entenedor el diagrama.

Agregació i composició

El diagrama de l'exemple següent mostra que:

- una guixeta (classe feble) va inseparablement lligada a l'existència d'una sala de cinema (composició);
- a una sala de cinema s'hi passen diverses pel·lícules (agregació).

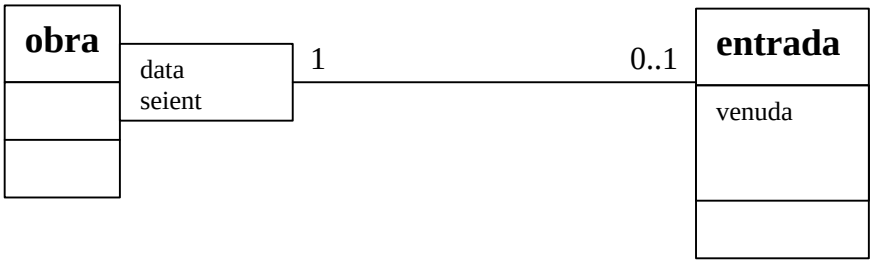


Convé observar que a l'hora de generar codi hi ha molt poca diferència entre una associació, una agregació i una composició. En cas de dubte, sobre tot a la fase d'anàlisi, no és cap problema deixar les relacions en forma d'associació.

Associacions qualificades

Un *qualificador* és un atribut d'una *associació binària* que serveix per a determinar unívovament un objecte o conjunt d'objectes d'una de les classes, la *classe objectiu*, que estan relacionats a través de l'associació.

Un qualificador es representa per un rectangle que s'adjunta a l'extrem de l'associació situat al costat de la *classe font*, que és la classe oposada a la classe objectiu.



Els qualificadors, bàsicament, es fan servir com a identificadors: un objecte de la classe font conjuntament amb un valor del qualificador selecciona unívocament un objecte o conjunt d'objectes de la classe objectiu.

La multiplicitat assignada a la classe objectiu acostuma a ser: 0..1, 1 i * .

Convé comparar aquesta situació amb la que tindriem si no utilitzesim una associació qualificada:



En aquest cas, per a saber si una entrada ha estat venuda, hauriem de buscar entre les entrades d'una obra fins a arribar a la que busquem, i llavors fer-ne la comprovació.

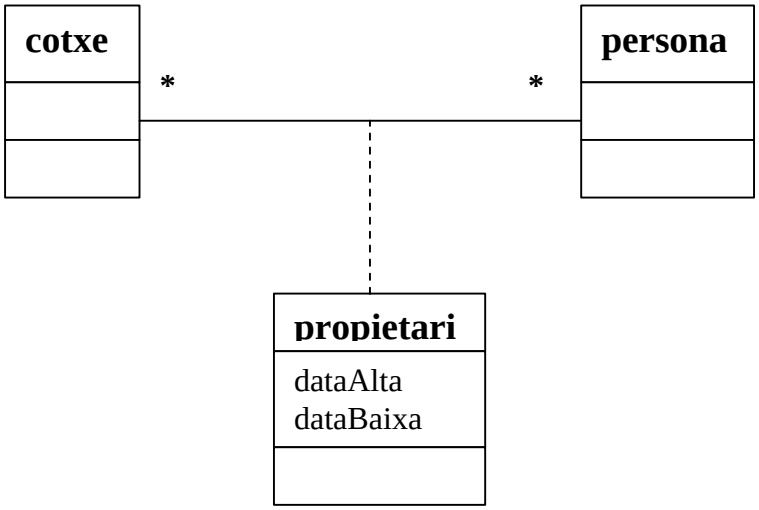
Classes associació

Una classe associació és una associació que té el mateix comportament que una classe.

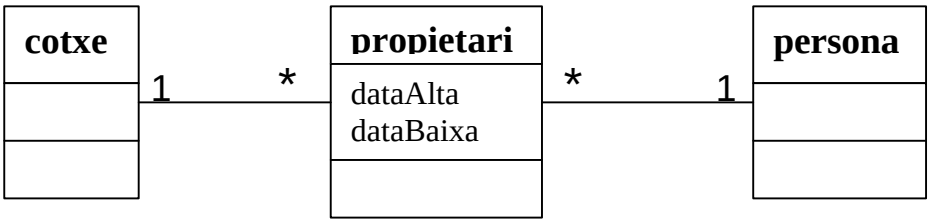
Una classe associació es representa amb un rectangle de classe que s'uneix per mitjà d'una línia puntejada a la línia que representa l'associació.

L'associació i la classe associació representen un sol element del model i, per tant, els noms de l'associació i de la classe associació han de coincidir.

Només es pot fer servir una classe associació quan hi ha una única relació entre dos objectes en un moment donat en el temps.



Algunes eines CASE (i tampoc Java) no permet treballar amb classes assocciació. En aquest cas cal transformar el model. A continuació donem un exemple de transformació per la classe associació **propietari** que acabem de veure.



Dependències

Una *dependència* entre dues classes A i B és una relació, que no sigui una associó (visibilitat d'atribut), en la qual un canvi a la classe B (proveidora) pot forçar canvis a la classe A (client). A un diagrama de classes, la dependència d'A respecte de B s'indica amb una fletxa puntejada que va des d'A fins a B.

Una classe depén d'una altra classe si:

- U objecte de la classe B és fa servir de paràmetre a una operació de la classe A (visibilitat de paràmetre).

- Una operació de la classe A retorna un objecte de la classe B (visibilitat de paràmetre).

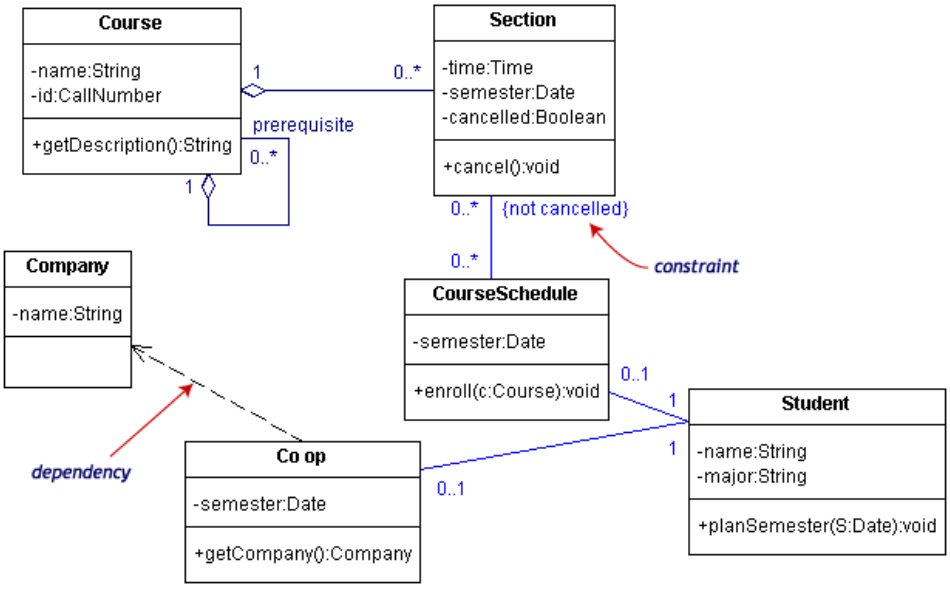


- Una operació de la classe A crea un objecte de la classe B i el fa servir dins d'un mètode, però sense mantenir-ne cap referència (visibilitat local).

- Una operació de la classe A fa servir un mètode estàtic de la classe B (visibilitat global).

Restriccions

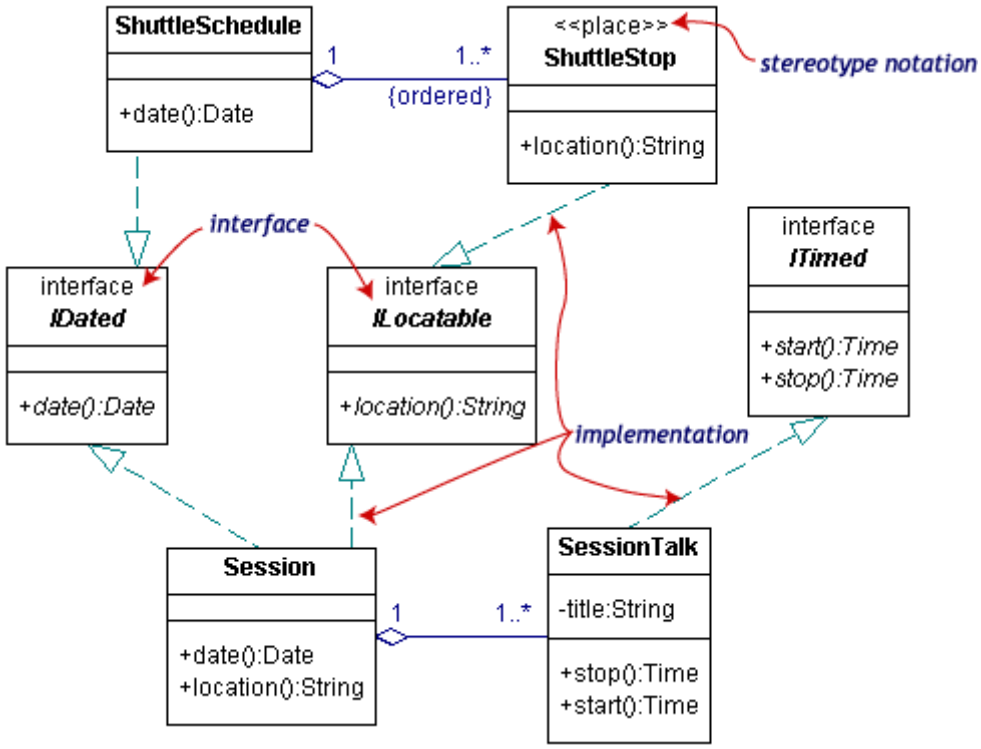
Una *restricció* és una condició que tota implementació del disseny ha de satisfer. En un diagrama de classes les restriccions s'escriuen entre { }. En el diagrama de l'exemple una **Section** pot ser part d'una **CourseSchedule** només si no ha estat cancel·lat.



Estereotips

Un *estereotip* permet afegir informacions extres als elements del model (classes, dependències, color/tipus de rol, …). El nom d'un estereotip s'escriu entre << >> sobre del nom de l'element.

El següent diagrama de classes és un model d'un congrés:



Les classes del model són **SessionTalk**, que és refereix a una presentació, i **Session**, que són el conjunt **SessionTalk**s d'un dia. **ShuttleSchedule,** la llista de **ShuttleStop**s, és important pels congressistes allotjats a hotels allunyats de la seu del congrés.

El diagrama té una restricció, les **ShuttleStop**s estan ordenades. La classe **ShuttleStop** té l'*estereotip* <<place>>.

Interfícies

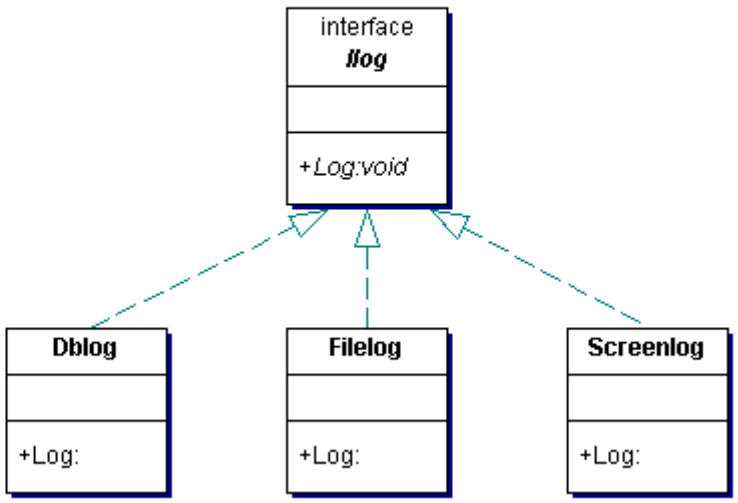
Una *interfície* és una classe que dóna nom a una llista d'operacions abstractes, sense indicar-ne la implementació. Una interfície només defineix la signatura completa de les seves operacions (nom, tipus de paràmetre i tipus de retorn), però no té atributs ni implementa les operacions. D'aquesta forma l'*especificació* d'una funcionalitat queda separada de la seva *implementació*.

Una interfície pot ser una especialització d'una interfície de més alt nivell. Una interfície no pot ser instanciada.

La implementació d'una interfície per una classe concreta s'anomena *realització*. La classe interfície porta l'estereotip «interface» sobre el nom de la interfície. Una realització es simbolitza per una linea a trossos amb una fletxa que apunta des de la classe implementadora a la classe interfície. No hi pot haver cap realització que tingui per origen la interfície. Una classe pot implementar diverses interfícies, i una interfície pot ser implementada per més d'una classe.

Exemples

1.

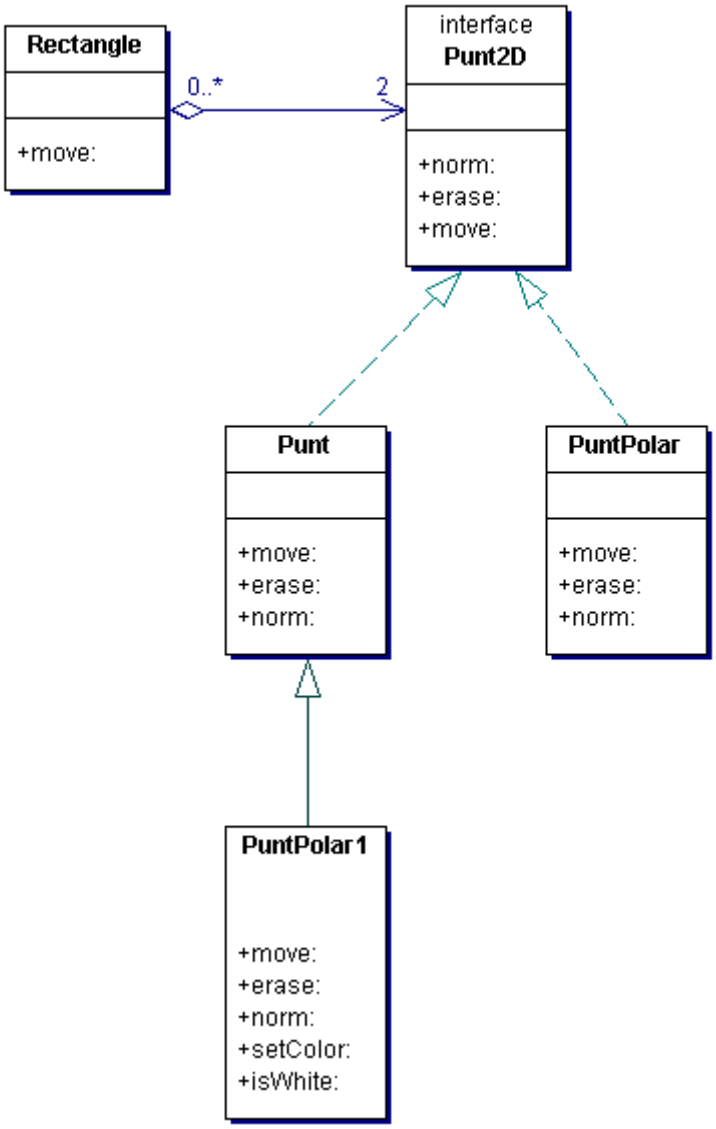


2. A l'anterior diagrama del model d'un congrés hi ha tres interfícies: *IDated*, *ILocatable*, and *ITimed*. El nom de les interfícies acostuma a començar amb la lletra I. Els noms de les interfícies i de les seves operacions s'escriuen en itàliques. Una classe com ShuttleStop, amb una operació que coincideix amb la de la interfície, com *ILocatable*, és una implementació de la interfície.

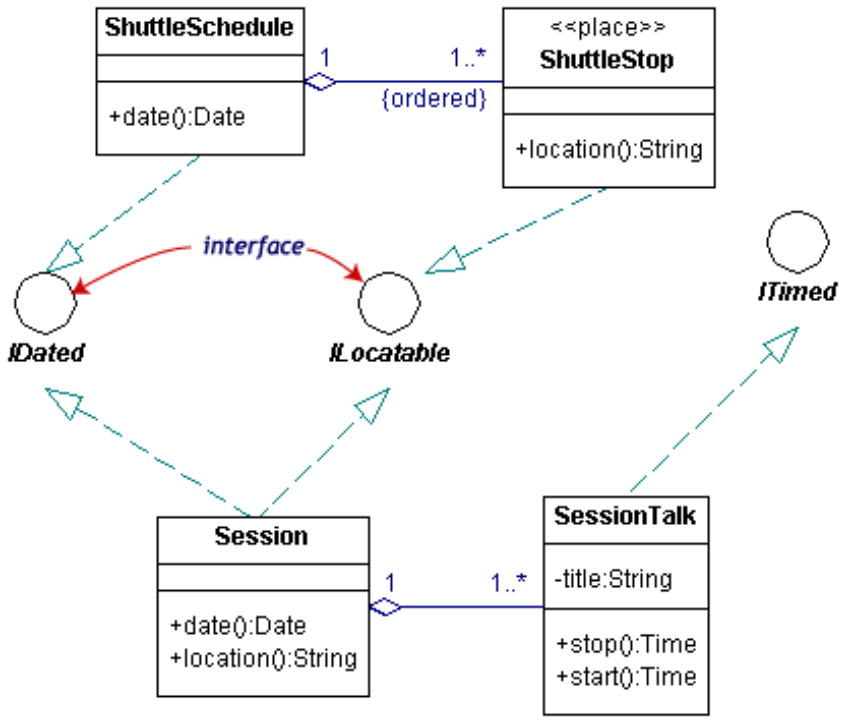
Interficies i associacions

Es pot establir una associació entre una classe C i una interfície I, sempre que la navegabilitat sigui des de C cap a I. Tot i que la interfície I no pot ser instanciada, cal interpretar aquesta associació en el sentit que un objecte de la classe C té referències a objectes de classes que implementen la interfície I, i que per tant poden rebre missatges de C.

Exemple



A més a més de la notació per a representar interfícies que acabem de descriure i utilitzar a l'exemple anterior, a UML hi ha una segona notació que fa servir cercles per a representar interfícies. Amb la notació cercles, l'exemple anterior es representaria com:



Amb la nova notació, les interfícies són cercles connectats per fletxes puntejades amb origen a les classes que les implementen. Al ser més compacta, però amb menys detall, la notació cercles simplifica els diagrames originals.

Classes d'anàlisi

El següent cas d'ús representa la funcionalitat que permet a un empleat de banca transferir diners des d'un compte a un altre:



El procés consta de dos passos:

1. Treure diners del **Compte1**;
2. Dipositar diners al **Compte2**.

Els objectes **Compte1** i **Compte2** de la classe **Compte** no saben res un de l'altre. Per tant el primer compte normalment no tindrà cap relació amb el segon. A més a més probablement voldrem enregistrar la transferència. Podríem fer que un dels comptes fos el responsable d'enregistrar la transferència, però això tampoc seria natural. Per a solventar-ho podem introduir un nou objecte **Transferir** que serà el responsable de la transferència. L'empleat demanarà a l'objecte **Transferir** que faci la transferència i aquest traurà diners d'un compte i els dipositarà a l'altre. L'objecte **Transferir**, afegit per l'analista, pren la responsabilitat de la transacció. A més a més pot ocupar-se també d'enregistrar-la. D'aquesta forma amb els comptes es fa allò que se suposa que s'ha de fer, treure-hi i dipositar-hi diners. Normalment, a la fase d'anàlisi, un cas d'ús tindrà un objecte de control d'aquest tipus. Amb la solució anterior encara hi ha un problema: a la realitat l'empleat no pot interactuar directament amb l'objecte **Transferir**. Haurem de dissenyar una interfície de pantalla, l'objecte **PantallaTransferir**, de manera que l'empleat hi pugui interactuar entrant-hi els detalls necessaris per a poder fer la transferència. Després **PantallaTransferir** demanarà a **Transferir** que faci la transferència.

Diferents tipus de classes d'anàlisi

Classes d'Entitats

Les classes d'entitats contenen els objectes bàsics que corresponen al mon real. Normalment s'emmagatzemen a una base de dades. Sovint són independents de l'aplicació. Generalment els objectes d'una d'aquestes classes no necessiten conèixer res uns del altres. Es comuniquen a través dels objectes de les classes de control. Els objectes de las classes d'entitats només tenen coneixement de les coses que poden fer per a ells mateixos.

Classes de Control

Cada cas d'ús ha de tenir almenys un objecte d'una classe de control que s'ocupi de la comunicació entre objectes. L'objecte de control serà el que s'ocuparà de dirigir els diferents camins del cas d'ús. L' especificació de l'objecte de control s'obtindrà dels diversos camins del cas d'ús, probablement expressats en forma de diagrames d'activitat.

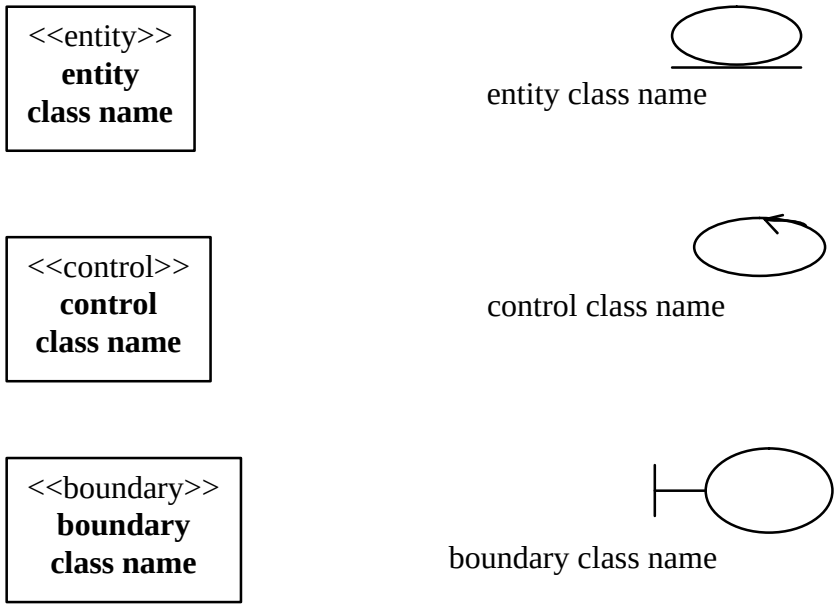
Les classes de control generalment no tenen atributs. Les operacions de les classes de control contenen els algorismes de l'aplicació. Entre les tasques típiques dels objectes de les classes de control hi trobem: enregistrar l'activitat del cas d'ús, cridar els controls de seguretat, etc.

Classes de Frontera

Els objectes de les classes frontera defineixen les interfícies amb els actors: finestres, diàlegs per pantalla i menús. Generalment seran pantalles de computador. S'utilitzen també per a cridar altres subsistemes. Un cas d'ús pot tenir diversos objectes frontera, i de vegades dos o més casos d'ús poden compartir un mateix objecte frontera. N'hi ha d'haver almenys un per a cada actor.

El modelat de les classes de control i de frontera s'acostuma a fer a la fase de disseny.

Notacions alternativa per a les classes d'anàlisi



Disseny estàndard de sistemes interactius

L'ampliació d'un cas d'ús fent servir classes d'anàlisi porta a una construcció estàndard dels sistemes interactius que pren una forma semblant a:

1) els actors interactuen amb els objectes frontera;
2) els objectes frontera fan peticions als objectes de control;
3) els objectes de control manipulen els objectes entitat.

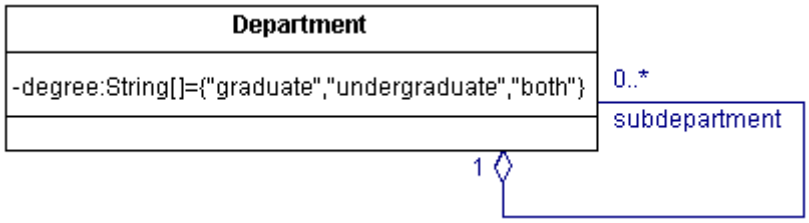
A la fase de disseny, generalment, s'incorpora en forma d'operacions:

- una classe de control a una classe de frontera;
- una classe de control a una classe d'entitat;
- una classe de control a una altra classe de control;

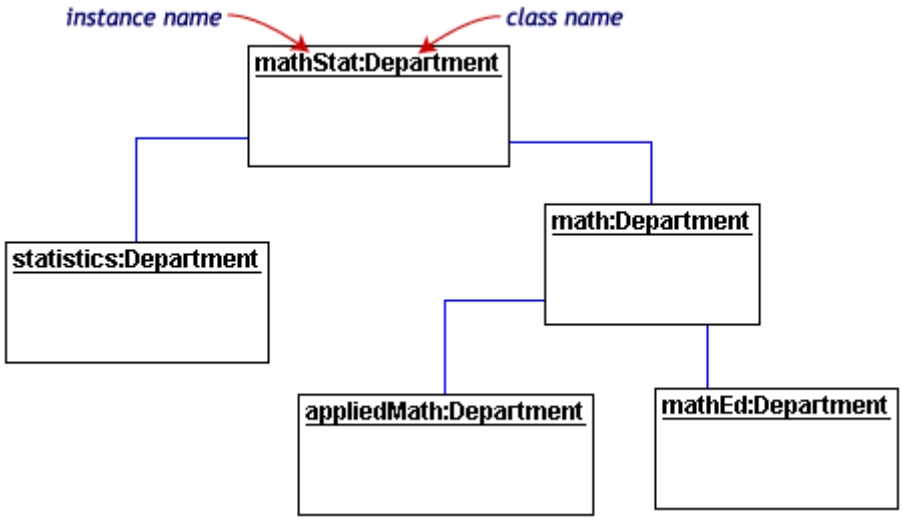
Diagrames d'objectes

Els diagrames d'objectes són tipus especials de diagrames de classes, que mostren instàncies en comptes de classes. Són útils per a explicar relacions que tenen una certa complexitat, com per exemple les relacions recursives.

El diagrama de classes de l'exemple següent mostra que un **Departament** universitari pot estar format per molts altres **Departament**s.



El diagrama d'objectes que segueix, instància el diagrama de classes tot substituint-lo per un exemple concret.



Cada rectangle del diagrama correspon a un únic objecte de la classe. El noms dels objectes (instàncies) es subratllen. El nom de la classe d'un objecte va precedit de (:).

Els diagrames de classes i d'objectes són models estàtics del sistema.

Paquets

Un *paquet* és una col·lecció d'elements del model (classes, altres paquets, casos d'ús, …) relacionats lògicament.

Els paquets permeten organitzar diagrames complexos, agrupant diversos elements en un paquet.

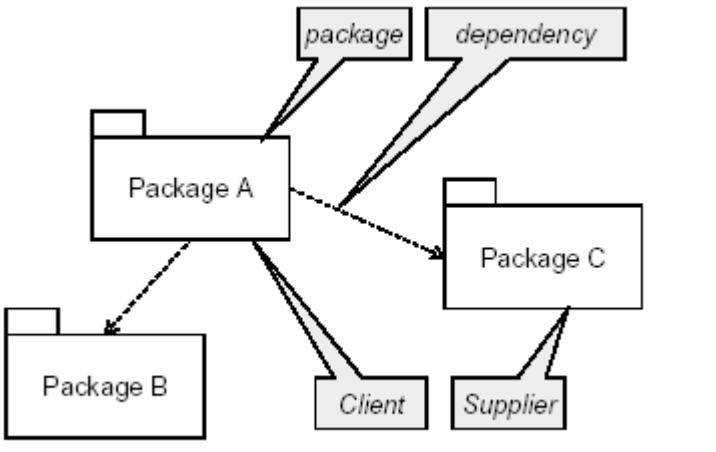
Els paquets es representen per rectangles amb petites pestanyes a la part superior. El nom del paquet es posa dins del rectangle o de la pestanya.

Un paquet depèn d'un segon paquet, si els canvis fets al segon paquet poden forçar canvis en el primer. Les dependències entre paquets s'indiquem amb fletxes puntejades.

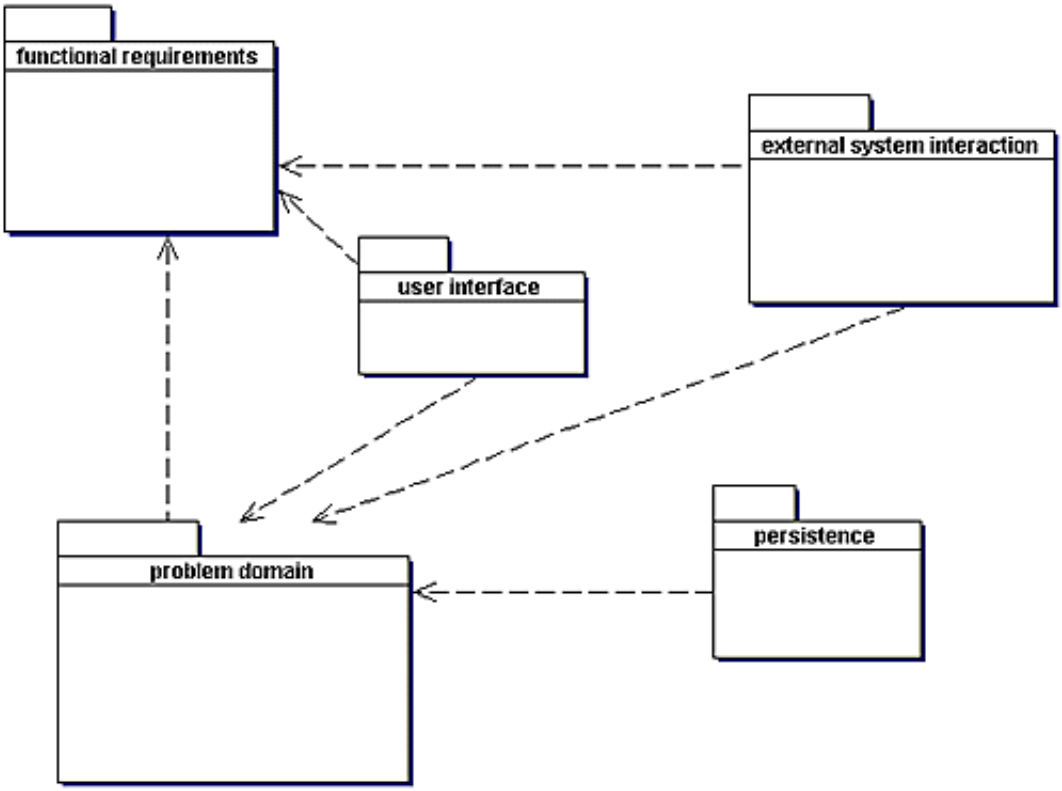
Naturalment, la dependència entre classes de diferents paquets implica la depència entre els corresponents paquets.

Els paquets continguts en altres paquets veuen tot allò que el paquet contenidor importa.

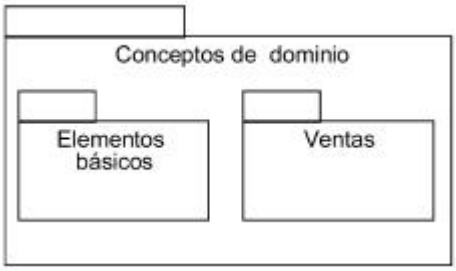
Exemple d'un diagrama de paquets:



Exemple de paquets:

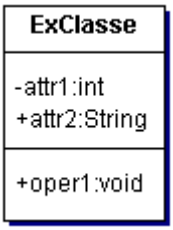


Exemple d'un paquet de paquets:



APÈNDIX. DELS DIAGARAMES DE CLASSES AL CODI JAVA

Classe



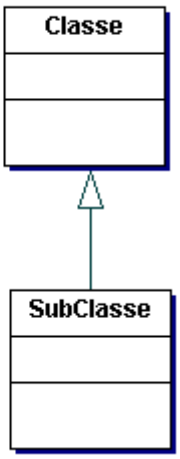

```
public class ExClasse {

    private int attr1;
    public String attr2;

    public void oper1() {
    }

}
```

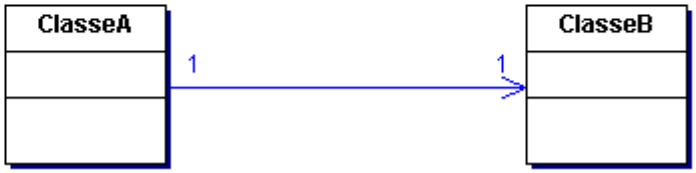
Generalització




```
public class Classe {
}

public class SubClasse extends Classe {
}
```

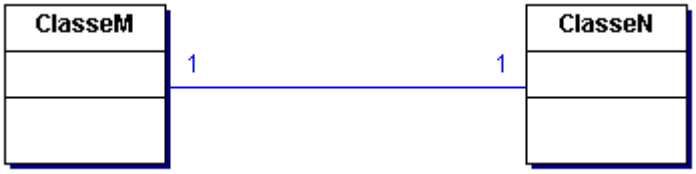
Associació amb multiplicitats 1 – 1 dirigida




```
public class ClasseB {
}

public class ClasseA {
   private ClasseB lnkClasseB;
}
```

Associació amb multiplicitats 1 – 1 no dirigida



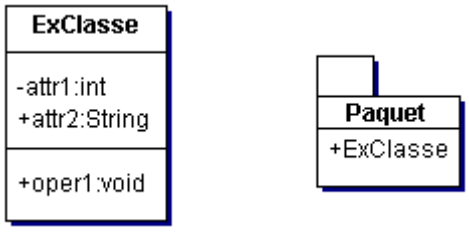

```
public class ClasseM {

    private ClasseN lnkClasseN;
}


public class ClasseN {

    private ClasseM lnkClasseM;
}
```

Classe d'un paquet




```
package Paquet;

public class ExClasse {

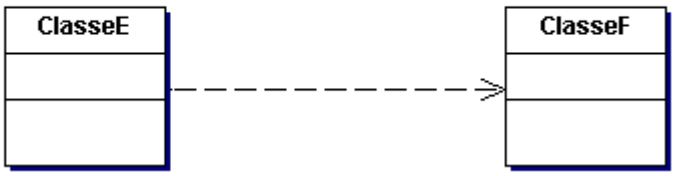
    private int attr1;
    public String attr2;

    public void oper1() {
    }

}
```

Dependències

Importar una classe d'un mateix paquet




```
public class ClasseF {
}
```

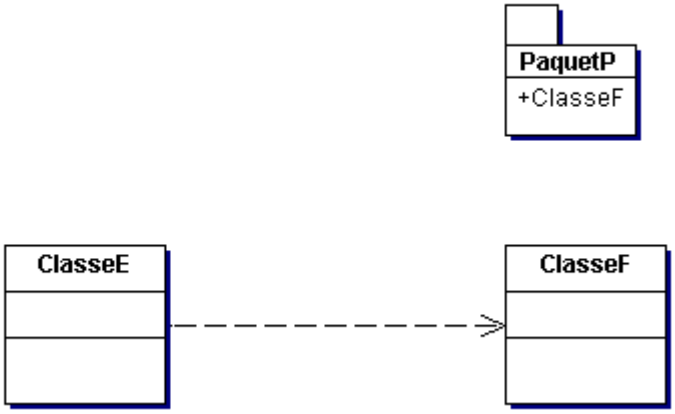
i la classe E podria ser, per exemple, de la forma:

```
import ClasseF;
public class ClasseE {
  Classe F f
}
```

o de la forma:

```
import ClasseF;
public class ClasseE {
a(ClasseF f){}
}
```

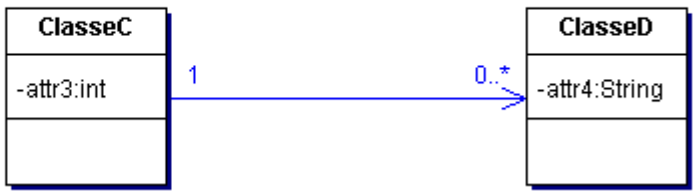
Importar una classe d'un paquet diferent




```
package PaquetP;
public class ClasseF {
}

import PaquetP.ClasseF;
public class ClasseE {
}
```

Associació amb multiplicitats 1 - 0..*



Es fa servir la classe List de java.util

```
public class ClasseD {

    private String attr4;
}

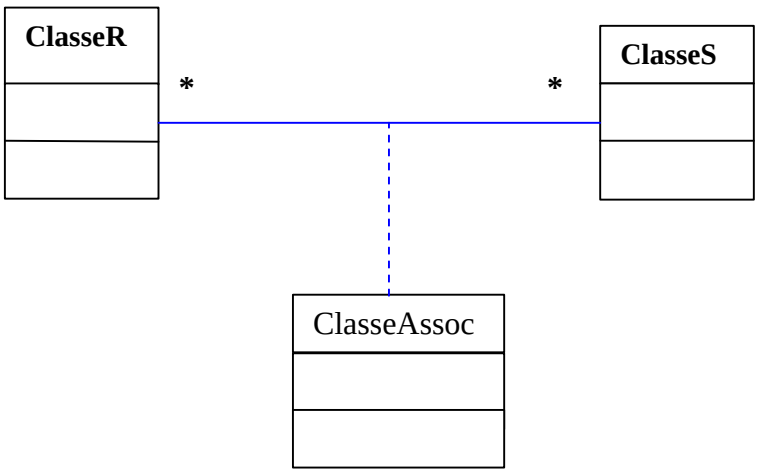

import java.util.List;
public class ClasseC {

    private int attr3;

    private List lnkClasseD;
}
```

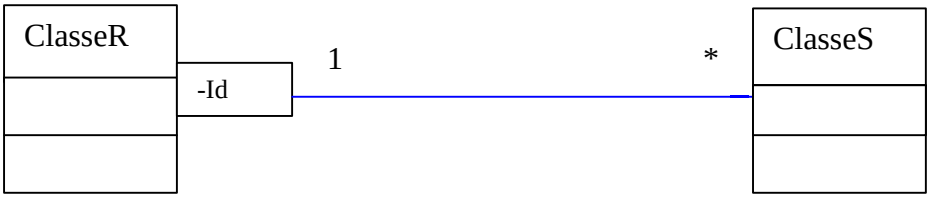
A nivell d'atributs, les agregacions i composicions es tracten com les asociacions.

Classe Associació



Es decomposa en dues associacions i després es transforma a codi Java d'acord amb les regles donades anteriorment.

Associació Qualificada



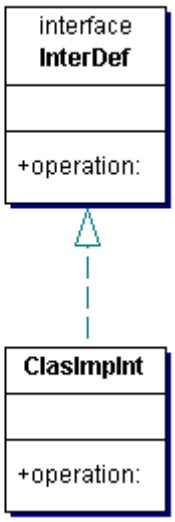
És semblant a la transformació amb multiplicitat 1 – 0..*, però en comptes de la classe List de java.util es fa servir la classe Map. El qualificador es fa servir com a clau de la Map.

```
public class ClasseS {
}

import java.util.Map;
public class ClasseR {

    private Map lnkClasseS;
}
```

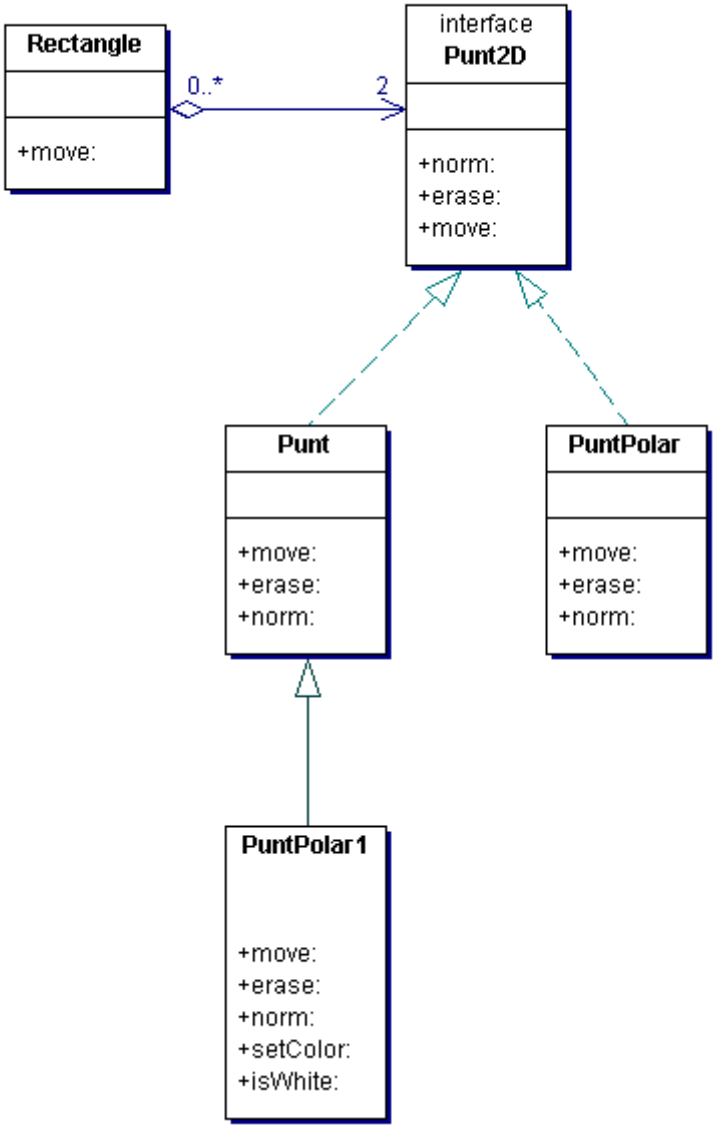
Interficie




```
public interface InterDef{
void operation()
{}
}
```

```
public class ClasImpInt implements InterDef{
void operation()
{…}
}
```

Exemple amb una interfície



```
interface Punt2D {
float norm() {}
Punt2D erase() {}
Punt2D move(int dx, int dy) {}
}

class Punt implements Punt2D {
int x,y;
float norm() {
return x*x+y*y;
}
Punt erase() {
x = y = 0;
return this;
}
Punt move(int dx, int dy) {
x += dx;
y += dy;
return this;
}

class PuntColor extends Punt {
String color;
PuntColor setColor(String color) {
this.color=color;
return this;
}
boolean isWhite() {
return color.equals(“white”);
}
Punt erase() {
x = y = 0;
color = “white”;
return this;
}
}

class PuntPolar implements Punt2D {
float rho, theta;
float norm() {…}
PuntPolar erase() {…}
PuntPolar move(int dx, int dy) {…}
}

class Rectangle {
Punt2D supEsq, infDre;
Rectangle move(int dx,int dy){
supEsq.move(dx,dy);
infDre.move(dx,dy);
return this;
}
}
```