

UML diagrama de classe

Introducció

Un diagrama de classes és un tipus de diagrama estàtic que descriu l'estructura d'un sistema mostrant les seues classes, atributs i les relacions entre ells.

Els diagrames de classes són utilitzats durant el procés d'anàlisi i disseny dels sistemes, on es crea el disseny conceptual de la informació que es manejarà en el sistema, i els components que s'encarregaran del funcionament i la relació entre l'un i l'altre.

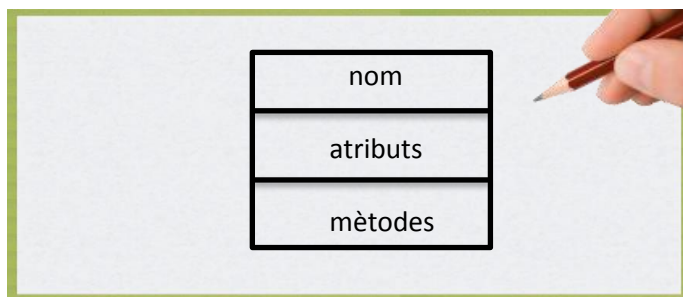
En un diagrama de classes es poden distingir principalment dos elements: les classes i les seues relacions.

Classe

Una classe és una descripció d'un conjunt d'objectes amb les mateixes propietats (atributs) i els mateixos comportaments (mètodes).

En programació orientada a objectes, **una classe és una plantilla** que representa un concepte del món real i s'utilitza per a crear objectes. Per exemple, si pensem en les llavadores, podem abstraure-les en una classe Llavadora que reuneix els seus atributs (marca, model, consum, velocitat) i comportaments (llavar, esbandir, centrifugar).

A partir d'una classe es poden crear objectes, també dits instàncies de la classe. Els objectes són concrecions d'una classe: tots els objectes d'una classe comparteixen els mateixos comportaments, però els seus atributs poden tindre valors diferents.



Una classe es representa mitjançant un rectangle dividit en 3 parts, la part superior està el nom de la classe, en la part central estan els atributs i en la part inferior estan els mètodes

Una classe pot no tindre atributs o no tindre mètodes, però es pinten les tres parts.

Per exemple, per a una llavadora podríem tindre la classe següent:



Llavadora
- marca : string
- model : string
- velocitatCentrifugat : integer
- consum : double
+ llavar()
+ esbandir()
+ centrifugar()

Atributs

Els atributs o característiques són les propietats de la classe, tenen el format següent:

visibilitat Nom : Tipus de dada

- La **visibilitat** defineix qui pot veure aquest atribut, també, es diu accessibilitat. Hi ha quatre visibilitats diferents:
 - + accés públic (public en Java). Qualsevol classe pot accedir a qualsevol atribut o mètode declarat com a públic.
 - # accés protegit (protected en Java). Només poden veure l'atribut o mètode protegit els elements de la pròpia classe o les classes filles.
 - ~ accés de paquet (no s'escriu res en Java). Qualsevol classe del paquet pot accedir a qualsevol atribut o mètode declarat com de paquet.
 - accés privat (private en Java). Només poden veure l'atribut o mètode privat els elements de la pròpia classe.

La taula següent mostra a quins tipus d'element d'una classe pot accedir una altra classe en funció de la seua visibilitat.

ACEDEIX	La pròpia classe	Una classe en el mateix paquet	Una subclasse en altre paquet	Una classe en altre paquet
Public	Sí	Sí	Sí	Sí
Protected	Sí	Sí	Sí	No
	Sí	Sí	No	No
Private	Sí	No	No	No

- El **nom** és un text relacionat amb la propietat que es vol descriure.
- El **tipus de dada** pot ser un tipus definit pel llenguatge (String, Integer, Double, etc) o una classe definida en el projecte (Punt, Euro, Nota, Alumne, etc)

Per exemple, per a definir la marca de la llavadora hem de tindre en compte: l'encapsulació ens diu que un atribut ha de ser privat, volem guardar un text i això o permet el tipus String de Java i com representa la marca aqueix serà el seu nom. La seua representació en el diagrama és

- marca: String

Es pot posar un valor inicial per a l'atribut

- velocitatCentrifugat : int = 0



Si la classe no té atributs, llavors es deixa el requadre del mig buit.

Comportaments

Els comportaments, mètodes o operacions són la forma que té la classe d'interactuar amb el seu entorn, tenen el format següent:

visibilitat Nom (Paràmetres) : Tipus de dada retornada

- La **visibilitat** és el mateix concepte que el dels atributs.
- El **nom** és un text relacionat amb el comportament que es vol definir.
- Els **paràmetres** són la informació que se proporciona al mètode quan es crida, pot tindre 0 o molts paràmetres

Un paràmetre té el format següent:

direcció Nom : Tipus de dada

- La **direcció** indica si el paràmetre és d'entrada in, eixida out o d'entrada/eixida in/out. Els paràmetres d'entrada, només es lliguen, els d'eixida només s'escriuen i els d'entrada/eixida es lliguen i escriuen. A Java tots els paràmetres són d'entrada.
- El **nom** és un text relacionat amb el paràmetre que es vol descriure.
- El **tipus de dada** pot ser algun definit per Java (String, Integer, etc) o un definit en el projecte (Cotxe, Bola, Aula, etc)

Si el mètode retorna un resultat, això s'indica escrivint el tipus de dada retornada, si el mètode no retorna res, llavors no s'escriu res.

Si la classe no té mètodes, llavors es deixa el requadre inferior buit.

EXEMPLE 1

```
public class Alumne {  
    private int edat;  
    private String nom;  
    private String cognoms;  
  
    public Alumne() {  
        nom = "John";  
        cognoms = "Doe";  
        edat = 18;  
    }  
  
    public void setNomCompleto(String nom, String cognoms) {  
        this.nom = nom;  
        this.cognoms = cognoms;  
    }  
  
    public String toString() {  
        return nom + " " + cognoms + " té " + edat + " anys";  
    }  
}
```



```

public int getEdat() {
    return this.edat;
}

public void setEdat(int edat) {
    this.edat = edat;
}

public String getNom() {
    return this.nom;
}


public void setNom(String nom) {
    this.nom = nom;
}

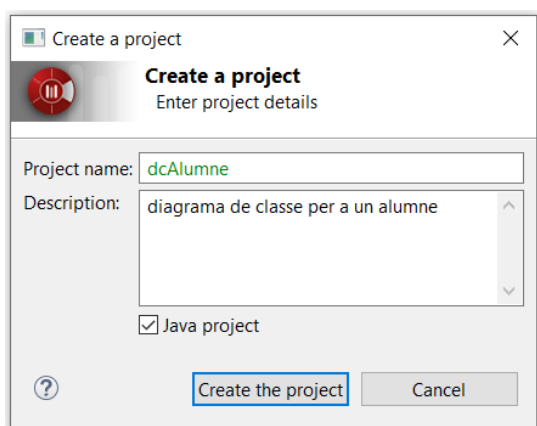
public String getCognoms() {
    return this.cognoms;
}

public void setCognoms(String cognoms) {
    this.cognoms = cognoms;
}
}

```

Anem a veure com crear el diagrama de classe de Alumne.

En Modelio, hem de crear un projecte nou, usant l'opció de menú **File > Create a Project...**, prement (Ctrl+N) o amb el botó .

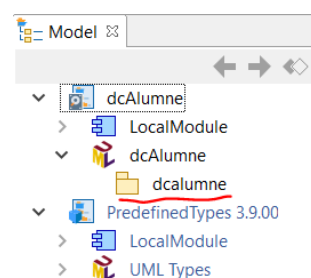


En la finestra de creació del projecte cal donar-li un nom (obligatori) i una descripció (optatiu). Per a tindre les opcions de Java, cal marcar la casella **Java project**, llavors, es carrega el mòdul del **Java Designer** en el projecte, aquest mòdul es pot carregar després de crear el projecte.

En la imatge tenim un projecte que s'anomena dcAlumne, amb la descripció "diagrama de classe per a un alumne" i s'ha marcat que és un projecte de Java

En prémer el botó **[Create the project]** es crea el projecte, cal desplegar l'arbre del projecte, allí està la carpeta de UML i a dins un paquet amb el nom del projecte, en eixe paquet és on anem a crear el diagrama de classe.

A nivell del sistema operatiu, s'ha creat una carpeta anomenada dcAlumne en l'espai de treball. Eixa carpeta és la has de copiar per a endur-te el projecte a altre Modelio.

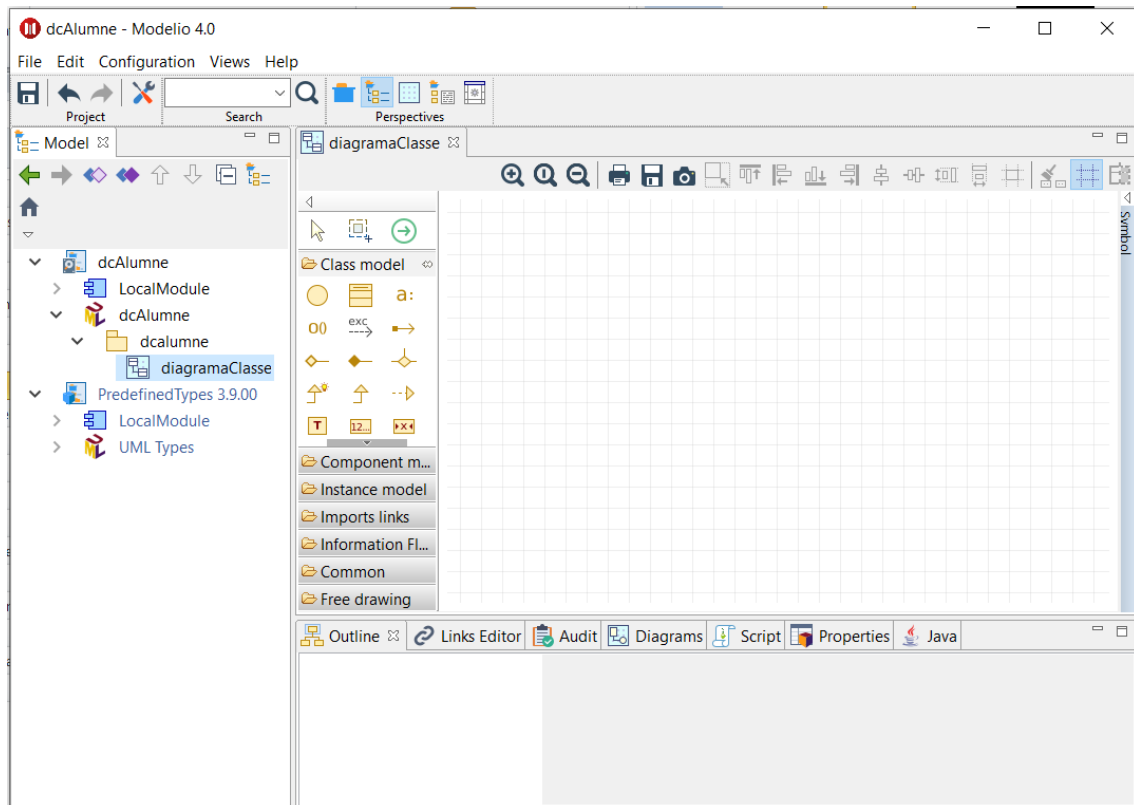


Sobre el paquet obrim el menú emergent i elegim [Create diagram...](#) que obri la finestra de creació de diagrames, en [Type selection](#) elegim [Class diagram](#), podem canviar el nom del diagrama en [Identification](#).

En la imatge següent s'ha canviat el nom per defecte "Class diagram" per "diagramaClasse".

En prémer el botó [\[OK\]](#) es crea una zona de dibuix amb les eines per a crear un diagrama de classe.

En un projecte es poden crear tots els diagrames que es vulga, del tipus que siga.



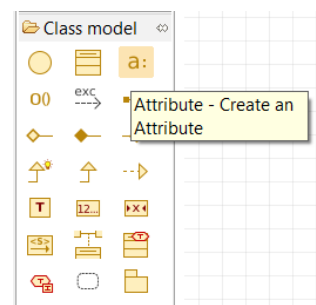
De totes les finestres amb eines, nosaltres anem a usar la finestra [Class model](#).

Per a triar un element de la finestra, fem clic sobre ell i després fem clic en la zona de dibuix per a col·locar-ho allí. Prémer la tecla (esc) anul·la la selecció.

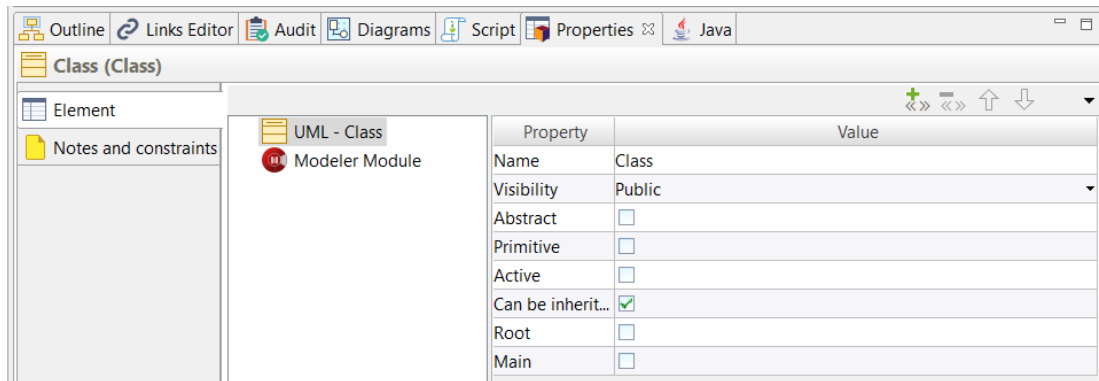
Si es posa el ratolí a sobre de l'element apareix un tooltip indicant quin tipus d'element és

Es poden utilitzar algun element de les finestres [Common](#) i [Free drawing](#).

Triem l'element  [Class – Create a class](#) i l'afegim a la zona de dibuix.

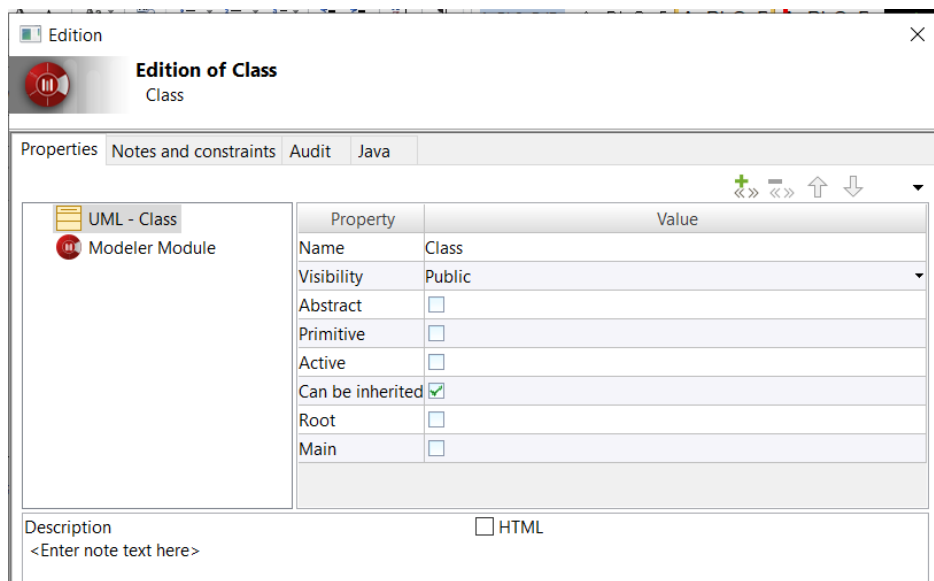


Quan es selecciona un element en la zona de dibuix, es mostren unes pestanyes amb informació de l'element seleccionat (la posició depèn de l'estructura del IDE)



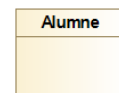
En la pestanya **Properties > Element** podem canviar el **Name** de la classe, també, es pot fer prement (F2)

Si fem doble clic sobre l'element o elegim **Edit element...** amb el botó dret, llavors s'obri la seua finestra de propietats.



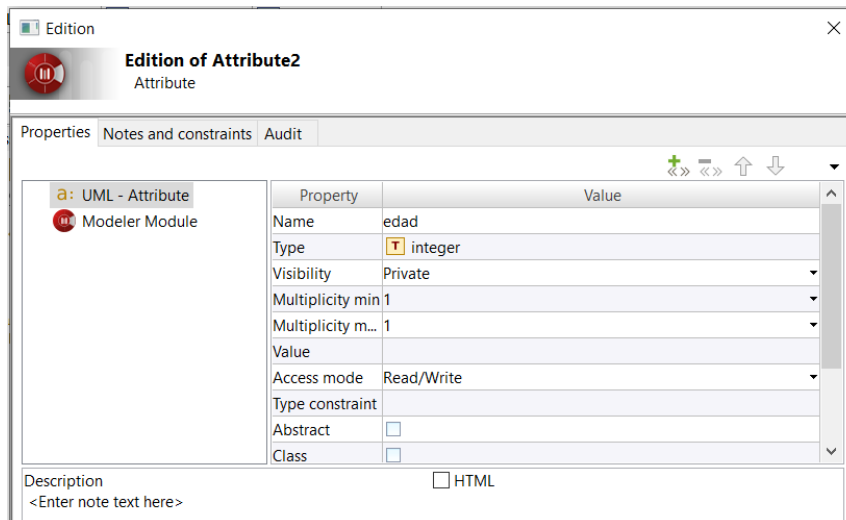
Com es pot veure tenim pràcticament el mateix.

Canviem el nom de la classe a Alumne



Triem l'element **a: Attribute – Create an attribute** i l'afegim (3 vegades). Obrim les seues propietats per a canviar el seu nom, la seua accessibilitat i el seu tipus.






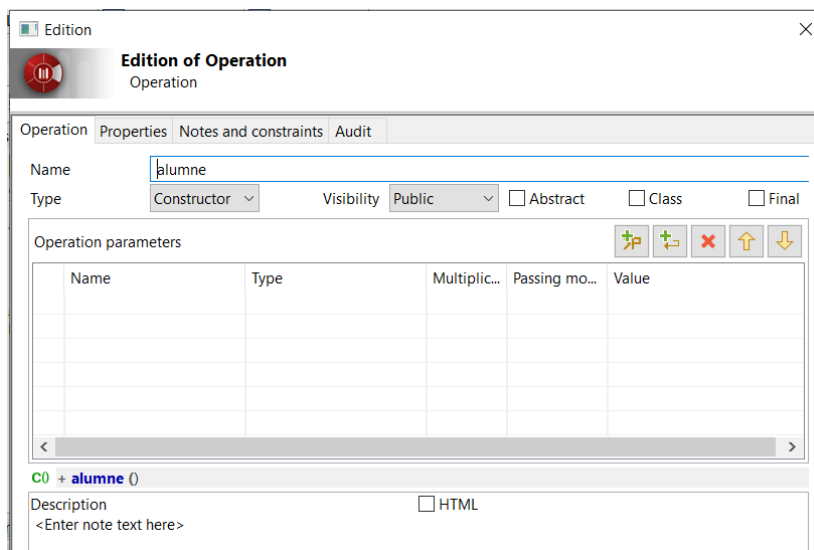
Cal canviar el nom. Per al tipus cal escriure l'inici del tipus i prémer (Control + barra espaiadora) per a triar-ne un dels tipus que ens ofereix. Per a l'accessibilitat cal triar-la de la llista desplegable, normalment seran privats.

Alumne
- nom : string
- cognoms : string
- edat : integer

Per defecte es creen atributs públics de tipus String, però es pot canviar en l'opció de menú [Configuration > Preferences... > Project default values](#).

El nom d'un atribut no es pot repetir, l'has de controlar tu, Modelio no ho fa.

Triem l'element  [Operation – Create an operation](#) i l'afegim tantes vegades com operacions volen definir. En les propietats de cada operació canviem els seus valors.



On podem canviar el tipus, la visibilitat, els modificadors, els paràmetres, el valor de retorn, etc.

En el tipus tenim [Constructor](#), [Operation](#) i [Destructor](#) (en Java no hi ha destructors)

La visibilitat dels mètodes, normalment, serà pública.

En la taula de paràmetres, tenim les accions següents:





afegeix un paràmetre al mètode



afegeix un valor de retorn, el nom **return** no es pot canviar, i sols hi ha un



esborra l'element seleccionat



mou l'element seleccionat a dalt o a baix en la taula

En la taula de paràmetres, podem posar el nom, el tipus, la multiplicitat, la direcció i el valor. Quan es crea un element es posen uns valors per defecte, que es poden canviar.

La multiplicitat indica quants elements del tipus indicat tenim, ens ofereix les possibilitats **0..1** (un o cap), **1** (un), **0..*** (zero o molts), però poden escriure qualsevol valor enter positiu, per exemple **2..4** indica que el nombre mínim és 2 elements i 4 el màxim.

La direcció indica el sentit de la informació dels paràmetres, ens ofereix les possibilitats **In**, **Out** i **In/Out**. **In**, la informació entra al mètode, **Out** la informació eix del mètode i **In/Out** la informació entra i ix del mètode. En Java tots els paràmetres són **In**.

A sota de la taula es mostra la capçalera del mètode que s'està definint.

La capçalera d'un mètode no es pot repetir, l'has de controlar tu, Modelio no ho fa.

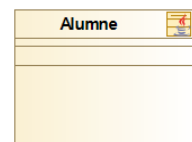
Alumne	
- nom : string	
- cognoms : string	
- edat : integer	
+ Alumne()	C0
+ toString(): string	
+ setNom(in nom: string)	
+ getNom(): string	

Si la classe està marcada com element de Java, podem crear la codificació de forma automàtica (la generació de codi es veu més avant).


Outline	Links Editor	Audit	Diagrams	Script	Properties	Java
Property	Value					
Java element	<input checked="" type="checkbox"/>					
No code	<input type="checkbox"/>					
Annotation	<input type="checkbox"/>					

El mòdul **Java Designer** ha d'estar activat en projecte per a tindre la pestanya Java. Si no està cal afegir-lo en **Configuration > Modules... > Add...**

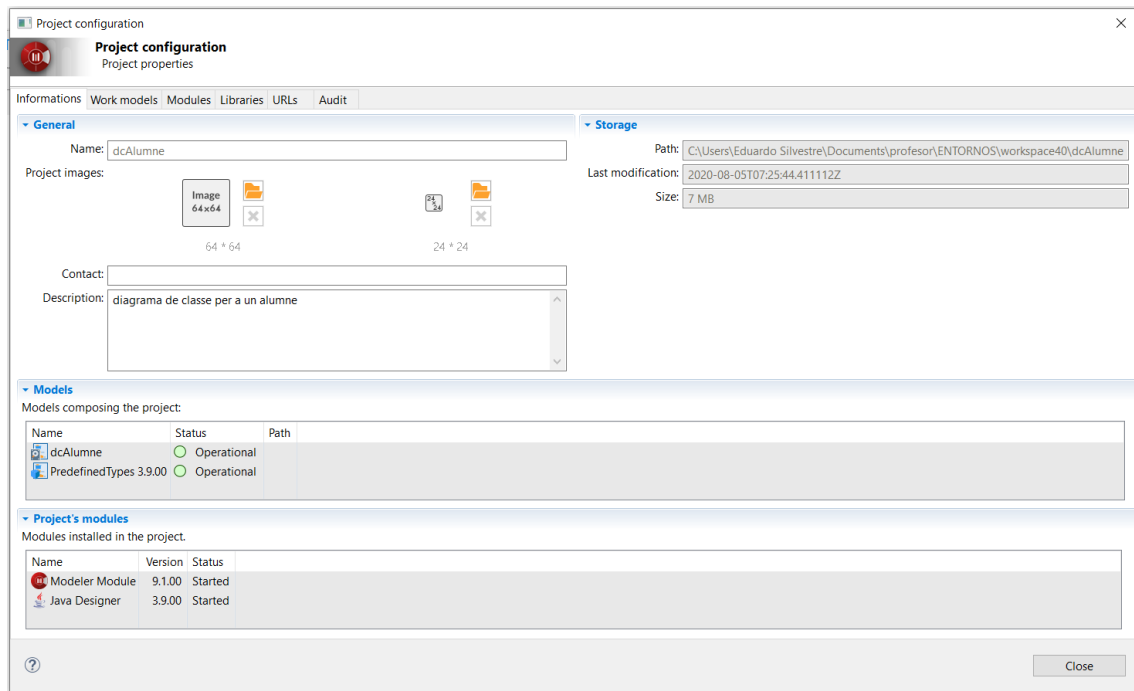
A la classe se li afeg una icona al costat del nom. Quan s'afegeix un atribut, llavors es creen el getter i el setter de forma automàtica.



Si el que s'ha creat no és el que vols, llavors has d'obrir la finestra de propietats i fer els canvis pertinents.

Si cliquem el botó  la configuració del projecte on podem vore informació i canviar algunes coses





Estàtic

El modificador estàtic es pot aplicar a atributs (variables o constants) i a mètodes, bàsicament indica que sols existeix una còpia de l'element, i es pot utilitzar sense necessitat d'instanciar la classe.

Als atributs (variables o constants) i mètodes que porten el modificador `static` se'ls anomenen estàtics o de classe. Als atributs i mètodes que no són estàtics (el comportament per defecte), se'ls anomenen dinàmics o d'instància.



Quan un mètode no necessita accedir als atributs de l'objecte i només depèn dels paràmetres que se li passen, es pot definir com a estàtic.

Quan un atribut (constant o variable) és estàtic, aquest es comparteix entre tots els objectes de la mateixa classe.

En l'exemple següent, tenim la classe `Matematic` amb diferents elements matemàtics (el valor de pi, el valor de e, el càlcul del sinus i del cosinus), és a dir, tenim dos atributs (`PI` i `E`) i dos mètodes (`si` i `cosinus`) estàtics.



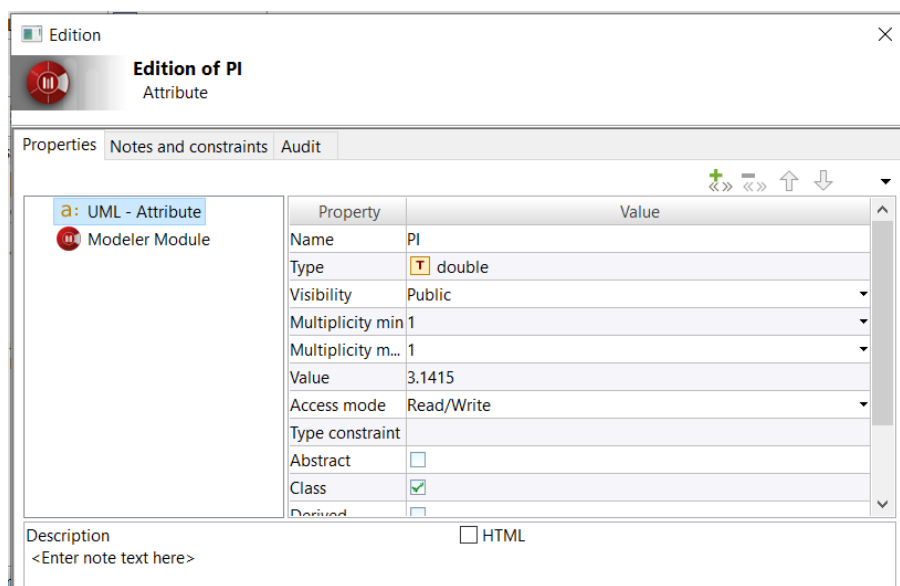
Matematic
+ PI : double
+ E : double
+ sinus(in x: double): double
+ cosinus(in x: double): double

La codificació de la classe Matematic en Java és

```
public class Mats {
    public static final double PI = 3.1415;
    public static final double E = 2.71828;
    public static double sinus(double x) {...}
    public static double cosinus(double x) {...}
}
```

Modelio

En Modelio per a definir un element com a estàtic cal marcar la casella de **Class** en la finestra d'edició de l'element.



Per a donar un valor a l'atribut PI, aquest s'ha d'escriure en la propietat **Value**, el valor no apareix en el diagrama.

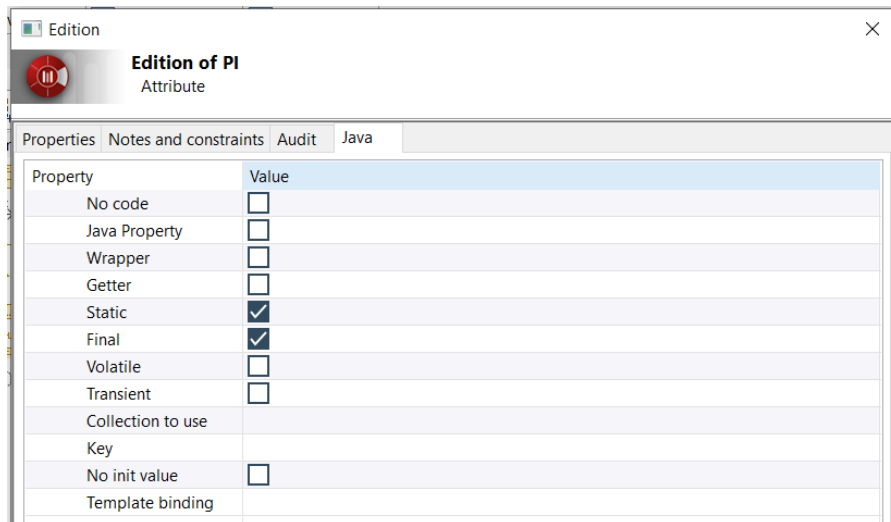
La propietat final no és una qualitat de UML, és de Java i indica que aqueix element no va a patir més canvis.

Per a un atribut, final el converteix en una constant, el noms d'una constant s'escriu en majúscules.

Per a un mètode, final indica que aqueix mètode no es pot sobreescrivre.

Si volem expressar aqueixa qualitat hem de marcar la casella **final** en la pestanya de Java, per a tindre eixa pestanya cal haver activat el mòdul **Java Designer**.





La icona que hi ha al costat del constructor Alumne marca l'estereotip de constructor.

Estereotip

Un estereotip és un mecanisme que permet estendre el llenguatge UML afegint informació extra als elements del diagrama. Per exemple, l'estereotip «Create» aplicat a una operació indicaria que aqueixa operació és el constructor de la classe.



Els estereotips es poden aplicar a qualsevol element del diagrama: classes, operacions, atributs, relacions, etc.

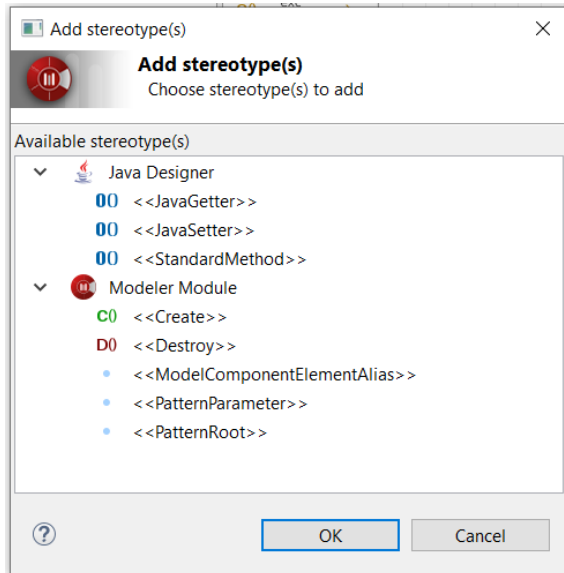
UML té diversos estereotips predefinits, però podem inventar-nos nous, per exemple en un diagrama de components de xarxa podríem distingir els switches dels hubs afegint els respectius estereotips: «switch» i «hub».

Alguns dels estereotips estàndard de UML aplicables als diagrames de classes són:

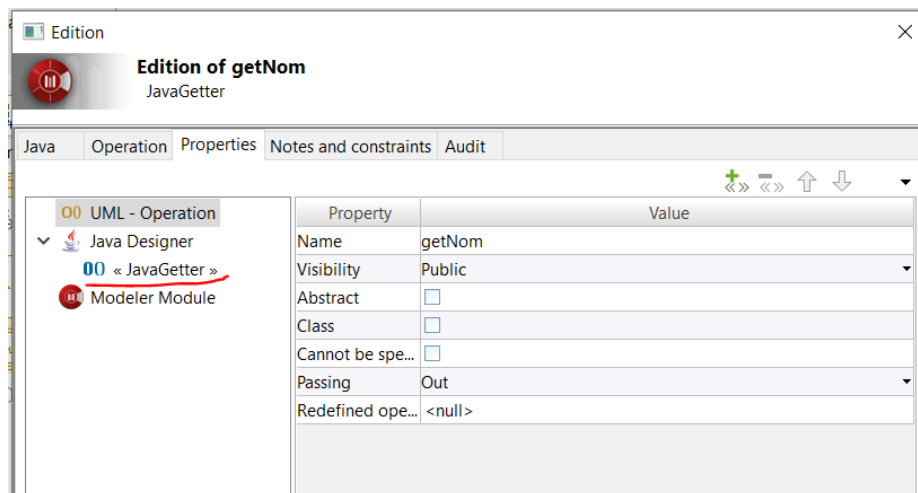
- «Metaclass» aplicat a una classe, indica que les instàncies d'aquesta classe són també classes.
- «Utility» aplicat a una classe, indica que la classe és una col·lecció de mètodes estàtics, no fa falta instanciar-la
- «Create» aplicat a una operació, indica que l'operació és un constructor
- «Destroy» aplicat a una operació, indica que l'operació és un destructor (Java no té destructors)

Modelio

En Modelio podem afegir estereotips a un element, triant l'opció [Add stereotype](#) del menú emergent.



En afegir algun estereotip es canvia o afeg una icona a l'element. En la finestra d'edició de l'element ha aparegut l'estereotip que hem afegit, si el volem eliminar cal obrir el menú emergent i triar [Remove](#).



Abstracte

Una classe o un mètode poden ser abstractes, això significa que tenim un comportament, però no se sap com implementar-lo.

Volem que la classe MitjaDeTransport es moga, llavors, es crea el mètode mouTe, però un Cotxe, un Avió o un Vaixell que són un tipus de MitjaDeTransport es mouen de forma molt



diferent, per tant, el mètode mouTe de MitjaDeTransport no sap com és el seu codi, però un Cotxe, un Avió o un Vaixell sí que saben codificar el mètode mouTe.

 <p>Classe «abstracte» mètode «abstracte»</p>	un element abstracte s'escriu seguit de l'estereotip «abstracte»
--	--

Una classe abstracta no s'instancia, ja que no té el codi complet.

Una classe abstracta s'hereta, i les classes que la hereten han de codificar tots els mètodes abstractes.

Un mètode abstracte només té capçalera, però no té codi.

Si una classe té un mètode abstracte, llavors passa a ser abstracta.

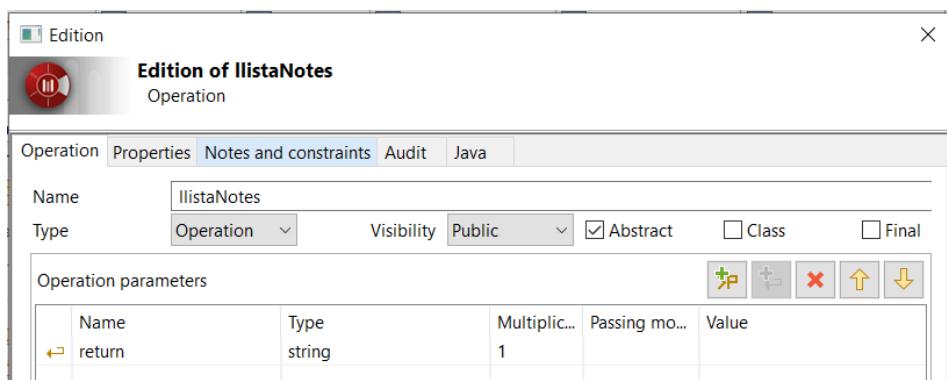
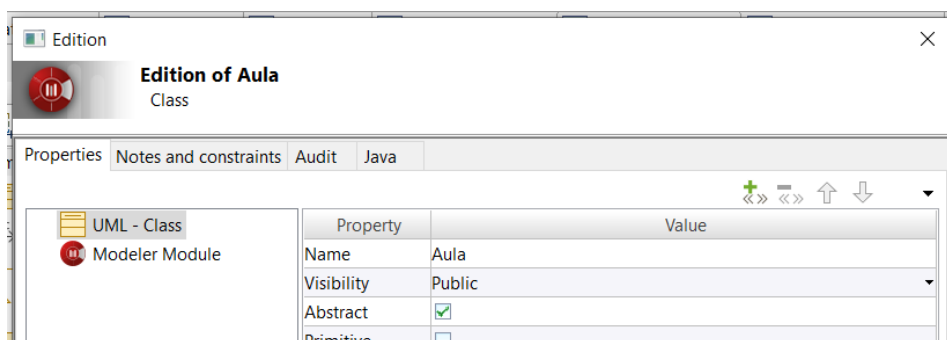
La classe saps utilitzar el mètode, coneix la capçalera, però no sap quin codi s'executarà.

Si la classe que hereta a una classe abstracta no implementa tots els mètodes abstractes, llavors serà abstracta.

En l'apartat **Generalització** hi ha un exemple.

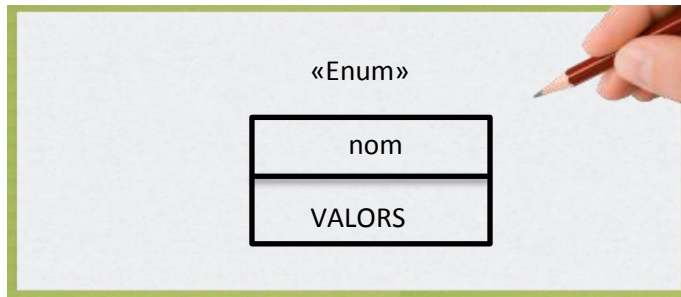
Modelio

En Modelio, en la finestra d'edició de l'element cal marcar la casella **Abstract**, per a transformar-lo en abstracte. Els elements abstractes es representen en itàlica.



Enumeració



Una enumeració és un conjunt discret de valors constants, per exemple, els dies de la setmana (DILLUNS, DIMARTS, DIMECRES, ...), els punts cardinals (NORD, SUD, EST, OEST)



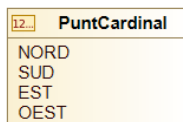
Una enumeració es representa amb un rectangle dividit en dos parts, en la superior està el nom i la inferior estan els valors. Damunt del rectangle s'escriu l'estereotip «Enum»

Els valors se solen escriure en majúscules (són constants)

Modelio

En Modelio el botó  [Enumeration – Create an Enumeration](#) crea una enumeració, i el botó  [Enumeration Literal – Create an Enumeration Literal](#) permet afegir un valor a una enumeració.

El valor d'una enumeració s'escriu en majúscula, és una convenció de Java, ja que és una constant.



Relacions entre classes

Una aplicació mínimament complexa tindrà diverses classes. Per a que la classe A pugui usar la classe B, la classe A ha de tenir una referència de la classe B. En lloc de definir referències es defineix relacions.

Quan es codifiquen les classes, les relacions es converteixen en referències.

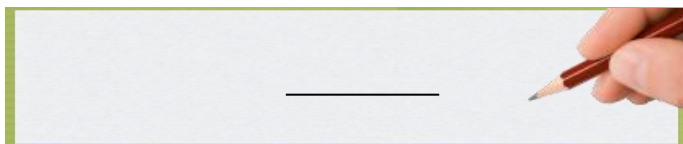
Una relació entre les classes, en el diagrama, es representa amb una línia que uneix les classes, les relacions poden ser de

- Associació
- Agregació
- Composició
- Dependència



Associació

Existeix una relació d'associació entre la classe A i la classe B quan en la classe A hi ha un atribut del tipus de la classe B i/o viceversa. És una forma gràfica de definir un atribut de tipus classe.

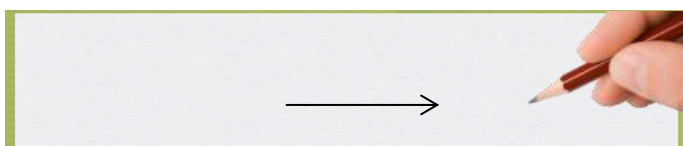


una associació es representa per una línia continua

És el tipus de relació més freqüent entre classes.

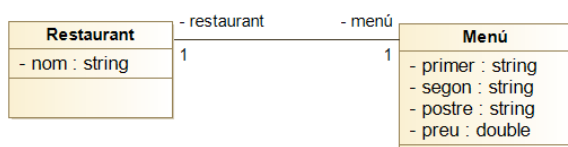
Si no es posa puntes de fletxa a la línia, l'associació és en els dos sentits, sols ens interessa indicar la relació entre les dos classes.

Si es posa una punta de fletxa oberta a la línia, el que volem indicar és quina classe té accés a l'altra classe, és a dir, quina classe té la referència de l'altra classe.



una associació es representa per una fletxa acabada en punta oberta

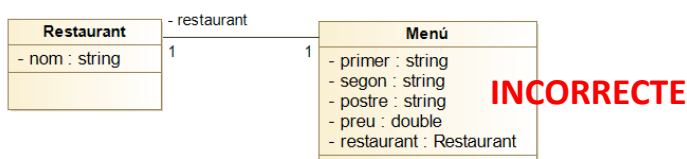
Tenim dues classes Restaurant i Menú que compleixen que un restaurant té un menú i un menú és d'un restaurant. L'associació seria



Si no es pinta l'associació, llavors cada classe té un atribut de l'altra classe



NO POSEU LA REFERÈNCIA I L'ASSOCIACIÓ, és redundant. En l'exemple següent la classe Menú tindria dos referències restaurant en la seua codificació.



Al costat de l'associació se sol indicar el seu rol i la seua multiplicitat.



rol

El rol és el significat de la relació, en la codificació es transforma en un atribut per a la classe.



L'associació es llig *“una comanda té un client”*. El rol és client, és a dir, mitjançant l'associació, la Comanda rebrà un atribut de tipus Client que s'anomena client (el rol).

El – indica que l'accés és privat, els símbols que es poden usar per al accés són els mateixos que els vists als atributs.

A Java la codificació seria

```
public class Comanda {
    private Client client; // client és el rol
}
```

En l'altre sentit l'associació es llig *“un client té 0 o moltes comandes”*, però com ni hi ha accés des de Client a Comanda, llavors no es posa rol.

Multiplicitat

La multiplicitat indica quants elements del tipus participen en la relació. Es pot usar un número o el * per a indicar molts (un número indeterminat superior a 1), té el format

multiplicitat mínima .. multiplicitat màxima

Es pot utilitzar el 0. Si la multiplicitat mínima i màxima és la mateixa s'escriu un sol valor. En l'exemple següent, s'expressa que un cotxe té de 3 a 5 rodes, i que una roda pertany a un únic cotxe.



Vegem diversos exemples de multiplicitat

- 1..1 significa té 1 i només 1
- 1 Significa té 1 i només 1
- 2 Significa té 2 i només 2
- 0..* Significa té 0 o moltes (sense límit)
- * Significa té 0 o moltes (sense límit)
- 0..3 Significa té entre 0 i 3.
- 2..* Significa té 2 o moltes



Navegabilitat

Una relació entre dues classes, representa el coneixement que té una classe respecte a una altra. Una relació pot anar en els dos sentits (bidireccional) o en un únic sentit (unidireccional).

Si ens interessa només la relació, llavors, es pinta una línia (relació bidireccional).

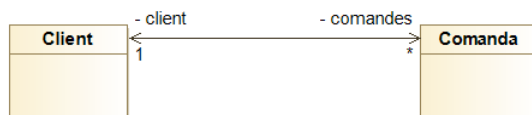
Si ens interessa l'accés d'una classe a una altra (un disseny més proper al codi), llavors haurem de posar fletxes en la relació, i la punta de la fletxa indica a qui té accés la classe, el sentit de les fletxes ens marquen la navegabilitat entre les classes.

Bidireccional

En una relació bidireccional, la classe A té accés a la classe B i la classe B té accés a la classe A, és a dir es pot navegar de la classe A a la classe B i viceversa.

La línia de l'associació es pot pintar amb o sense fletxes, amb les fletxes es vol representar la navegabilitat.

A Java, les dues classes rebran un atribut amb el nom del rol i del tipus de l'altra classe.



En l'exemple anterior, tenim una associació bidireccional entre Client i Comanda. Una comanda té un client (el rol és client i la multiplicitat és 1) i el client té moltes comandes (el rol és comandes i la multiplicitat és moltes). La comanda té accés al seu client i el client a les seues comandes.

La codificació a Java és la següent. Els atributs que apareixen en les classes es diuen client i comandes pel rol expressat en el diagrama, i la multiplicitat 1 dona el tipus Client, i el * dona lloc a una llista de tipus Comanda

```
public class Client {
    private List<Comanda> comandes; // el tipus és Comanda, el rol és comandes admet moltes (List)
}
public class Comanda {
    private Client client; // el tipus és Client, el rol és client, admet un únic client
}
```

Unidireccional

Si l'associació de les classes es representa amb una fletxa que va de la classe A a la classe B, significa que la classe A té accés a la classe B, però la classe B no té accés a la classe A, es diu que l'associació o navegabilitat és unidireccional.

A Java, això significa que apareixerà un atribut en la classe A que fa referència a la classe B.





El client té accés a les dades de les seues comandes, però la comanda no té accés al client que l'ha realitzat.

La codificació a Java és

```

public class Client {
    private List<Comanda> comandes;    // el Client té una referencia a la llista de Comanda
}
public class Comanda {
}
  
```

Per al diagrama, amb la navegabilitat en sentit contrari




La comanda té accés al client que l'ha fet i el client no té accés a les comandes que ha realitzat. La codificació a Java és

```

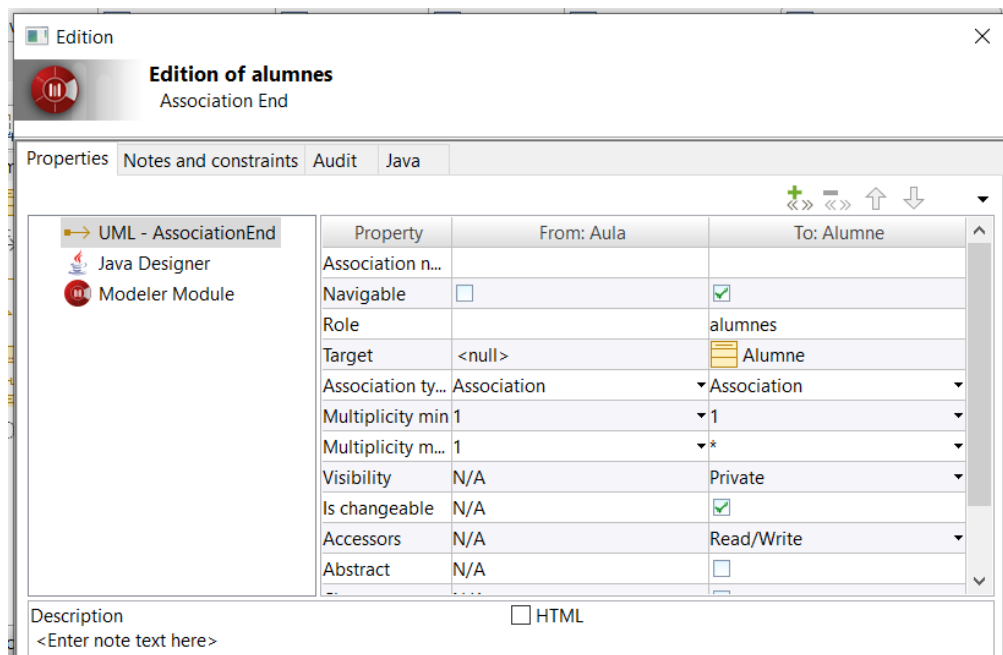
public class Client {
}
public class Comanda {
    private Client client;    // la Comanda té una referència al Client
}
  
```

Modelio

En Modelio en la finestra de **Class model** triem l'element  **Association – Create an association** i en la zona de dibuix fem clic en les classes que volem relacionar. Per defecte es crea una associació navegable des de la primera classe on hem fet clic a la segona.

En la vista d'element o en les propietats de l'associació tenim els dos costats de l'associació. Podem canviar la navegabilitat, els rols, les multiplicitats de cada costat. El **Target** és el tipus de dada corresponent al rol d'aqueix costat.





La visibilitat dels rols és **Private** per a complir amb l'encapsulació.

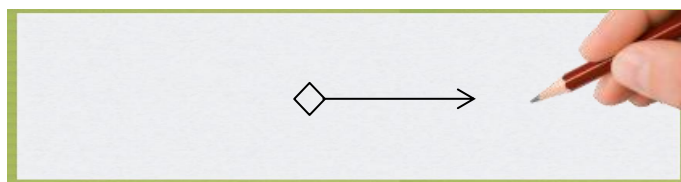
En la navegabilitat es pot escriure qualsevol valor enter positiu.

Agregació i composició

L'agregació i la composició són associacions que representen una relació entre el tot i les seues parts.

Si la classe A el tot i la classe B la part, es pot llegir com "la classe B és part de la classe A" o com "la classe A està feta de la classe B".

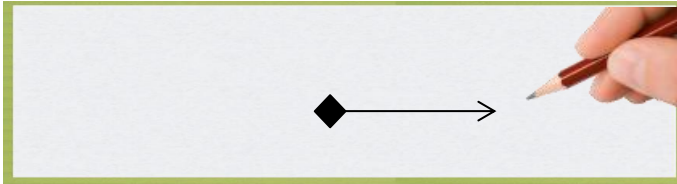
En l'agregació, les "part" són independents, i continuen existint encara que el "tot" es destrueixca.



Una agregació es representa amb una fletxa del "tot" a la "part" i posant un rombe buit en el "tot"

UML no precisa amb massa detall la semàntica de l'agregació així que a vegades és difícil distingir-la d'una associació. Davant el dubte, usarem l'associació.

En la composició, les "part" són dependents del "tot", de manera que si el "tot" es destrueix, les "parts" també es destrueixen.

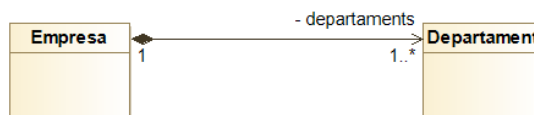


Una agregació es representa amb una fletxa del “tot” a la “part” i posant un rombe reomplit en el “tot”

La relació següent indica que el Cotxe té de 3 a 5 Roda. Els objectes Roda poden existir encara que no existisca l'objecte Cotxe, per tant la relació és d'agregació.



La relació següent indica que una Empresa està composta per molts Departament. Un departament no pot existir si l'empresa a la que pertany no existeix, per aquest motiu, la relació és de composició. Els objectes Departament es destruiran si es destrueix l'objecte Empresa.



Una conseqüència de la composició és que una “part” només pot relacionar-se amb 1 i només 1 “tot”, ja que si aqueix “tot” deixa d'existir, les seues parts també hauran de deixar d'existir. Per tant, la multiplicitat en la classe “tot” sempre serà 1 en una composició.

La implementació a Java de l'agregació és idèntica a l'associació

```

public class Cotxe {
    private Roda rodes[5]; // el tipus és Roda, el rol és rodes permet 5 (matriu)
}
  
```

```



public class Empresa {
    private List<Departament> departaments; // el tipus és Departament, el rol és departaments permet molts (List)
}
  
```

La composició s'implementaria cridant als destructors de les classes “part” des del destructor de la classe “tot”.

A Java, la gestió de memòria està basada en el “*garbage collector*” i no existeixen els destructors, així que si es perd la referència a Empresa (el “tot”), llavors es perden totes les referències que conté (les “parts”), és a dir, a Java la composició s'implementa exactament igual que l'agregació i l'associació



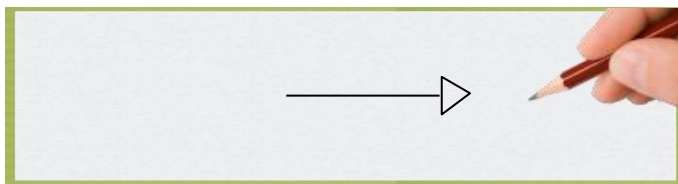
Modelio

En Modelio en la finestra de **Class model**, triem l'element  **Aggregation – Create a Shared Aggregation** o  **Composition – Create a Composite Aggregation**. Primer es fa clic sobre la classe “tot” i després sobre la classe “part”.

En la finestra d'edició de l'associació tenim les propietats de l'agregació o composició en la propietat **Association type**.

Generalització (herència)

La relació de generalització entre dues classes indica que la subclasse és una especialització de la superclasse, és a dir, la especialització fa tot el que fa la superclasse i més coses.

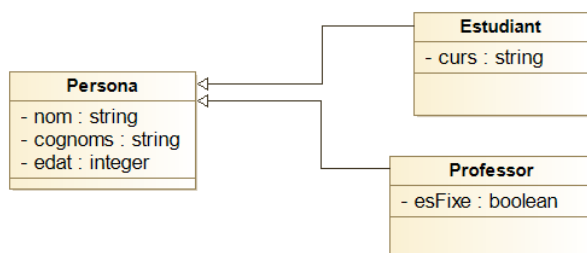


una generalització es representa amb una fletxa amb la punta tancada de la subclasse a la superclasse

A la subclasse també se li pot dir classe filla o classe derivada. A la superclasse també se li pot dir classe pare o classe base.

La generalització, també, s'anomena herència, ja que les subclasses hereten totes les propietats de la superclasse (atributs i comportaments). L'herència s'utilitza per a abstraure en una superclasse els mètodes i/o atributs comuns a diverses subclasses.

Si la classe A és la superclasse i la classe B la subclasse, la generalització obliga a que es compleixca “la classe B és una classe A” o “classe B és un tipus de classe A”. Per exemple tenim la classe Persona (superclasse) i les subclasses Estudiant i Professor, es compleix “un Estudiant és una Persona” i “un Professor és un tipus de Persona”.



La codificació de l'exemple en Java és

```
public class Persona {  
    private String nom;  
    private String cognoms;  
    private int edat;  
}  
public class Estudiant extends Persona {  
    private String curs;  
}
```



```
public class Professor extends Persona {
    private boolean esFixe;
}
```

Modelio

En Modelio en la finestra de **Class model**, triem l'element [Generalization – Create a Generalization](#). Primer es fa clic sobre la classe filla i després sobre la classe pare.

Es poden ajuntar dues línies de generalització, fent clic sobre una línia de generalització ja existent o modificant els punts de les línies perquè s'unisquen.

En l'exemple següent, la superclasse **Figura** és abstracta ja que el mètode `getArea` és abstracte (no sap calcular l'àrea d'una figura), la subclasse **Cercle** que hereta de **Figura** ja pot codificar el mètode `getArea`, ja que sap calcular l'àrea de un cercle.

```
public abstract class Figura {
    public abstract double getArea(); // només està la capçalera del mètode
}
public class Cercle extends Figura {
    private double radi; // radi del cercle
    public Cercle(double radi) { // constructor de Cercle
        this.radi = radi;
    }
    public double getArea() {
        return 2 * Math.PI * radi; //càlcul de l'àrea d'un cercle
    }
}
```

El diagrama de classes és el següent



Realització (interfícies)

En UML, una realització equival a la implementació d'una interfície.

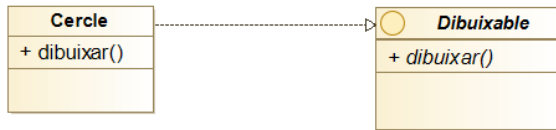
A Java, una interfície és a una classe completament abstracta, és a dir és una classe on tots els mètodes són abstractes (no s'implementa cap).

Una interfície permet al dissenyador de classes establir la forma d'una classe (noms dels mètodes, els paràmetres i els tipus de retorn), però no la seua codificació, per tant, la classe que implementa la interfície ha de codificar tots els mètodes presents en ella.

Per exemple, la interfície **Dibuixable** defineix que s'ha de fer per a que una classe es puga dibuixar a si mateixa, i defineix que s'ha de codificar el mètode `dibuixar`. La classe **Cercle** es



considera que es pot dibuixar, llavors implementa la interfície Dibuixable que obliga a codificar el mètode dibuixar.



La codificació a Java és

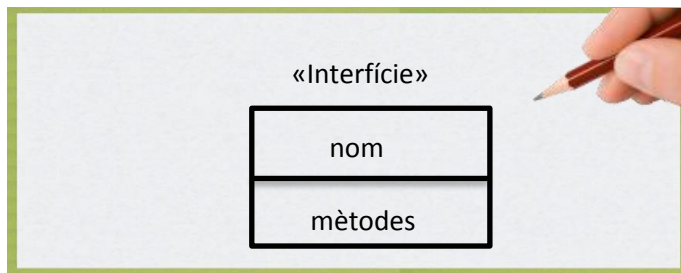
```
public interface Dibuixable {
    void dibuixar();        // és abstracte, només s'escriu la capçalera del mètode
}
public class Cercle implements Dibuixable {
    void dibuixar() {
        // implementació de com es dibuixa un cercle
    }
}
```

Una interfície es pot aplicar a classes molt diferents, si la classe **Persona** es considera que es pot dibuixar, llavors pot implementar la interfície **Dibuixable** que l'obligarà a implementar el mètode **dibuixar**.

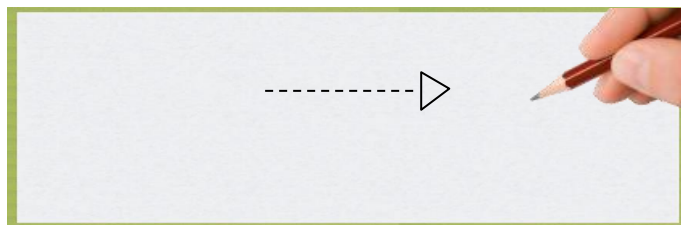
Una interfície permet relacionar totes les classes que la implementen, per exemple podríem crear un tractament per a totes les classes que es consideren dibuixable, és a dir que implementa la interfície **Dibuixable**.

Una classe pot implementar múltiples interfícies, però només pot heretar de (com a màxim) una classe pare. Una interfície pot heretar d'una o més interfícies.

Una interfície pot també contenir variables, però sempre **static** i **final**.




Una interfície es representa com una classe amb l'estereotip «Interfície» al damunt

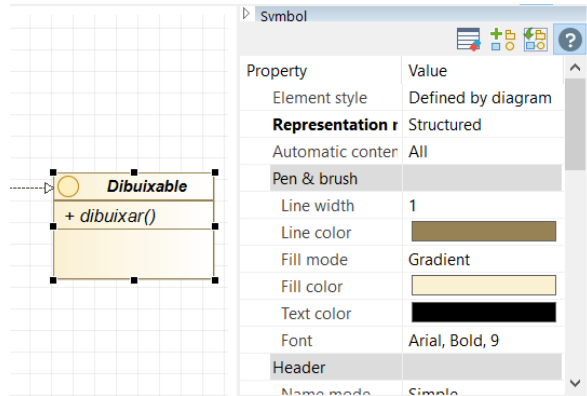



la implementació d'una interfície es representa amb una fletxa discontinua acabada en punta tancada de la classe a la interfície

Modelio

En Modelio en la finestra de **Class model**, triem l'element  **Interface – Create an Interface** i el col·loquem en la zona de dibuix.

Per defecte, una interfície es visualitza en la forma simple (un cercle) per a canviar la seua representació a la forma estructurada (el mateix dibuix que una classe, però amb un cercle com a icona) on podem veure els seus atributs i mètodes, cal obrir la vista de **Symbol** (està a la dreta de la zona de dibuix) i en l'atribut **Representation mode** donar-li el valor **Structured**.



En la finestra de **Class model**, triem l'element  **Interface Realization – Create an Interface Realization** i en la zona de dibuix es fa clic en la classe i després en la interfície.

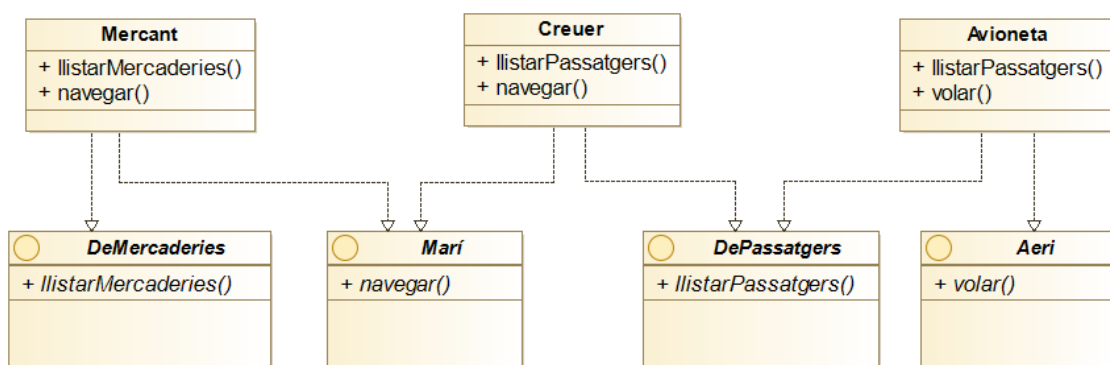
En l'exemple següent tenim les interfícies: **Marí**, **Aeri**, **DeMercaderies** i **DePassatgers**, i les classes: **Mercant**, **Creuer** i **Avioneta**.

Un **Mercant** és un vehicle marí i és un vehicle de transport de mercaderies

Un **Creuer** és un vehicle marí i és un vehicle de transport de persones

Una **Avioneta** és un vehicle aeri i és un vehicle de transport de persones

Aquestes relacions es poden expressar amb la realització de múltiples interfícies, cosa impossible d'expressar utilitzant la generalització.

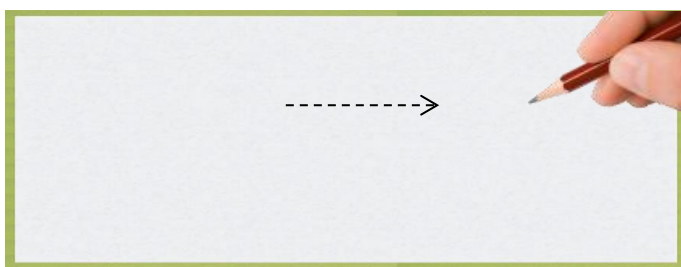


Relació de dependència

Segons el Manual de Referència de UML [RJB05], una dependència indica una relació semàntica entre dues o més classes del model. La relació pot ser entre les classes pròpiament

dites, és a dir, no té per què involucrar a instàncies de les classes (com en l'associació, composició i agregació). Si la classe A depèn de la classe B, això significa que un canvi en la classe B provoca un canvi en el significat de la classe A.

Segons aqueixa definició, totes les relacions que hem vist: associació, agregació, composició, generalització i realització són dependències, però com tenen un significat molt específic se'ls ha donat un nom concret. Per tant, es pot dir que les dependències són totes aquelles relacions que no encaixen en cap de les categories anteriors.



una dependència es representa per una fletxa discontinua acabada en punta oberta que va de la classe que usa a la classe usada

En la pràctica, i segons coincideixen molts autors, la dependència s'usa quan la classe A utilitza breument a la classe B, concretant, utilitzarem una relació de dependència quan una classe utilitza un objecte d'una altra classe però sense emmagatzemar-lo en cap atribut (ja que llavors seria una associació, agregació o composició), per exemple

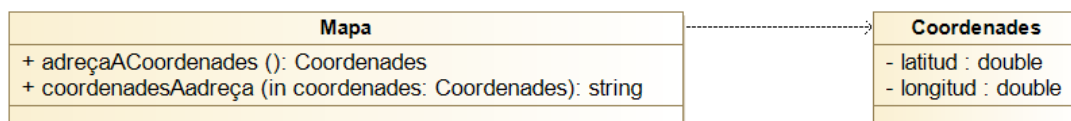
- Quan a un mètode de la classe A se li passa com a paràmetre un objecte de la classe B.
- Quan un mètode de la classe A retorna un objecte de la classe B
- Quan un mètode de la classe A té una variable local de tipus B

La classe **Coordenades** conté les dades (latitud i longitud) d'un punt geogràfic.

Tenim una classe **Mapa** capaç d'obtenir les coordenades (latitud i longitud) a partir d'una adreça postal i viceversa, té dos mètodes:

- El mètode `adreçaACoordenades` retorna les coordenades geogràfiques a partir d'una adreça donada.
- El mètode `coordenadesAadreça` retorna l'adreça més pròxima a les coordenades donades.

Els dos mètodes utilitzen breument un objecte de la classe **Coordenades**, entenent per breu el fet que no l'emmagatzemen en cap atribut de la classe. En conseqüència, **Mapa** depèn de **Coordenades**. El diagrama que representa aquesta relació és:



La codificació a Java és




```

public class Mapa {
    public Coordenades adreçaACoordenades (String adreça) { . . . }
    public String coordenadesAadreça (Coordenades coordenades) {    }
}

public class Coordenades {
    private double longitud;
    private double latitud;
}

```

Modelio

En Modelio la dependència està en la finestra de [Common](#), triem l'element  [Dependency – Create an Dependency Link](#), es fa clic en la classe que usa i després en la usada.

