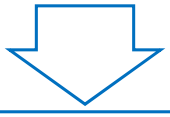
Disseny de Software en UML

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona
FIB
Departament d'Enginyeria de Serveis i Sistemes d'Informació
UNIVERSITAT POLITÈCNICA DE CATALUNYA

De l'Especificació al Disseny: Contractes de les Operacions

De l'Especificació al Disseny

- Al disseny hi tenim **components de software** i no conceptes de domini.
- Limitació tecnològica
  - No podem implementar directament tots els conceptes que hem usat a l'especificació



- Cal una **transformació** prèvia dels diagrames d'especificació
  - Obtenció del diagrama de classes de disseny:
    - Eliminar elements no compatibles amb la tecnologia
  - Obtenció dels contractes de disseny de les operacions
    - Controlar les restriccions d'integritat
    - Tractar la informació derivada.

Disseny sense precondicions

- A diferència d'especificació, a disseny no hi ha cap sistema que ens protegeixi de violar restriccions d'integritat, multiplicitats, precondicions o qualsevol protecció que ens doni el diagrama de classes de forma natural.

- Necessitem un mètode de poder controlar i informar de les restriccions que es violin al cridar una funció.
- Les Precondicions ja no tenen sentit!

Excepcions

- Els llenguatges de programació ens proporcionen diverses maneres de poder controlar els casos en els que la operació no s'ha pogut realitzar per causa d'algun problema.
- La més utilitzada és mitjançant l'ús **d'Excepcions**.

Gestió d'Excepcions

- Segons Wikipedia:
  - La gestió d'excepcions és una tècnica de programació que permet al programador controlar els errors ocasionats durant l'execució d'un programa informàtic. Quan es produeix algun tipus d'error, el sistema reacciona executant un fragment de codi que resolgui la situació, per exemple, retornant un missatge d'error o retornant un valor per defecte.
- Informal
  - És la manera de controlar i gestionar problemes. Quan es produeixi un problema, llencem una excepció per a notificar-ho, això atura la execució.

Com programaríem això?

```
Context: Sistema::FesLaResta(
             num1: Enter Positiu,
             num2: Enter Positiu
             ): Enter Positiu
Pre: num1 >= num2
Body: result = num1 – num2;
```


Així?

```
unsigned int FesLaResta(
             unsigned int num1,
             unsigned int num2)
{
       return num1 – num2;
}
```

- Si num1 és més petit que num2, això provocarà un error al sistema que no volem que passi. La precondició ens protegia d'això, i ara ja no hi és.

Així!

```
Class Num1Menor: públic std::exception
{}

unsigned int FesLaResta(
             unsigned int num1,
             unsigned int num2)
{
       if (num1 < num2)
       {
              throw Num1Menor();
       }
       return num1 – num2;
}
```


Contractes de les Operacions a Disseny

- S'elimina la secció de precondicions
- S'afegeix una secció d'excepcions
  - Per cada possible problema que la operació pugui provocar crearem una nova excepció. La definiríem amb un **nom explicatiu i una descripció**.

Contractes de les Operacions a Disseny

```
Context: Sistema::FesLaResta(
             num1: Enter Positiu,
             num2: Enter Positiu
             ): Enter Positiu
Excepcions:
    - Num1Menor: num1 és menor que num2
Body: result = num1 – num2;
```


Excepcions possibles

- Les excepcions que es poden produir en una operació poden ser dels següents tipus:
  - Violació d'una precondició
  - Violació d'una restricció textual
  - Violació d'una restricció gràfica
  - Violació d'una restricció del model (implícita)
- Per cada operació haurem de fer una anàlisi i veure quines es poden produir.

Violació d'una precondició

Especificació

Context: AltaPersona(
nom: String,
edat: Enter,
nomCiutat: String)

Pre: nomCiutat existeix

Post: Es dóna d'alta una nova persona amb el nom, i l'edat introduïts, associada a la Ciutat nomCiutat

Disseny

Context: AltaPersona(
nom: String,
edat: Enter,
nomCiutat: String)

Excepcions:

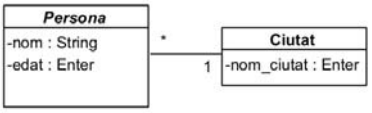
- [CiutatNoExisteix] la Ciutat nomCiutat no existeix

Post: Es dóna d'alta una nova persona amb el nom, i l'edat introduïts, associada a la Ciutat nomCiutat

- Per cada Precondició que hi hagi a la nostra operació, haurem d'afegir una excepció nova

Violació d'una restricció textual

Disseny



RT1 (Persona, nom), (Ciutat, nom)
RT2: edat de Persona >= 18

Context: AltaPersona(
nom: String,
edat: Enter,
ciutat: String)

Pre: nomCiutat existeix

Post: Es dóna d'alta una persona nova amb el dni, ciutat i edats introduïts.

Disseny

Context: AltaPersona(
nom: String,
edat: Enter,
ciutat: String)

Excepcions:

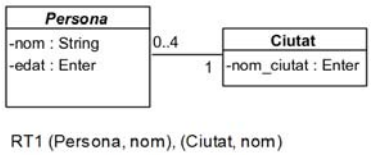
- [CiutatNoExisteix] la Ciutat nomCiutat no existeix
- **[PersonaExisteix] Persona amb nom = nom existeix**
- **[MenorD'Edat] edat < 18**

Post: Es dóna d'alta una persona nova amb el nom, ciutat i edats introduïts.

- Per cada restricció d'integritat que es pugui violar, haurem d'afegir una nova excepció.

Violació d'una restricció gràfica

Disseny



RT1 (Persona, nom), (Ciutat, nom)

Context: AltaPersona(
nom: String,
edat: Enter,
nomCiutat: String)

Pre: nomCiutat existeix

Post: Es dóna d'alta una persona nova amb el dni, ciutat i edats introduïts.

Disseny

Context: AltaPersona(
nom: String,
edat: Enter,
ciutat: String)

Excepcions:

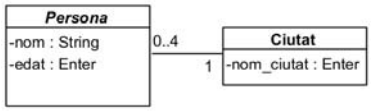
- [CiutatNoExisteix] la Ciutat nomCiutat no existeix
- [PersonaExisteix] Persona amb nom = nom existeix
- **[MassaGent] ciutat ja te 4 persones**

Post: Es dóna d'alta una persona nova amb el nom, ciutat i edats introduïts.

- Per cada multiplicitat que es pugui violar, cal una nova excepció

Violació d'una restricció del model (implícita)

Disseny



RT1 (Persona, nom), (Ciutat, nom)

Context: AfegirACiutat(
nomPersona: String,
nomCiutat: String)

Pre: nomCiutat existeix
nomPersona existeix

Post: Es dóna d'alta una persona nova amb el dni, ciutat i edats introduïts.

Disseny

Context: AfegirACiutat(
nomPersona: String,
ciutat: String)

Excepcions:

- [CiutatNoExisteix] Ciutat amb nom_ciutat = nomCiutat no existeix
- [PersonaNoExisteix] Persona amb nom = nomPersona no existeix
- **[PersonaJaViu] La Persona amb nom = nomPersona ja viu a la ciutat amb nom_ciutat = nomCiutat**

Post: Es dóna d'alta una persona nova amb el nom, ciutat i edats introduïts.

- Per cada multiplicitat que es pugui violar, cal una nova excepció

Elements de Disseny en UML

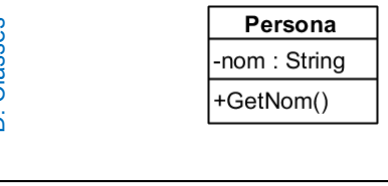
- Crides i resultats
- Crides entre instàncies
- Navegabilitat
- Agregats
- Polimorfisme
- Elements Abstractes
- Creadores

Obtenir el diagrama de seqüència de disseny

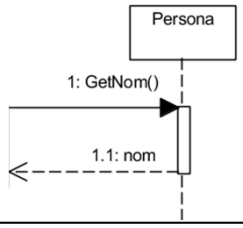
- A disseny, el diagrama de seqüència ens ha de mostrar les crides entre les funcions de les diverses classes, per tant, ha de ser un diagrama que entri a fons en les interaccions entre classes.
- **Avís**: Alguns elements del nostre diagrama de seqüència, repercutiran en canvis al diagrama de classes

Crides i Resultats

D. Classes



D. Seqüència



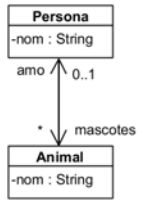
Codi

```
class Persona
{
public:
    string GetNom()
    {
        return nom;
    }

private:
    string nom;
};
```

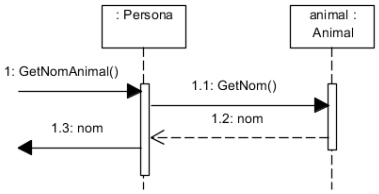
- La forma de cridar una funció del diagrama de Seqüència és la mateixa en que es cridaven els esdeveniments a Especificació

Crides entre Instàncies

D. Classes



D. Seqüència



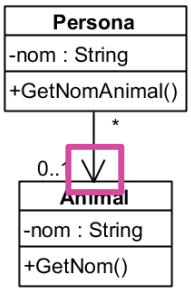
Codi

```
class Persona
{
public:
    string GetNomAnimal() {
        return animal->GetNom();
    }

private:
    Animal* animal;
    string nom;
};

class Animal
{
public:
    string GetNom() {
        return nom;
    }
private:
    string nom;
};
```

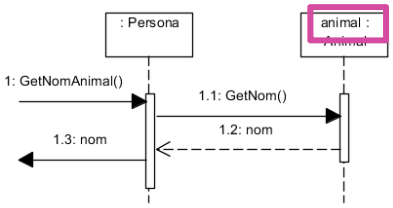
- Una instància d'una classe pot cridar funcions públiques de les instàncies amb que està relacionada, privades d'ella mateixa i protegides dels seus ancestres a la jerarquia.

Crides entre Instàncies (II)

D. Classes (Disseny)



D. Seqüència (Disseny)



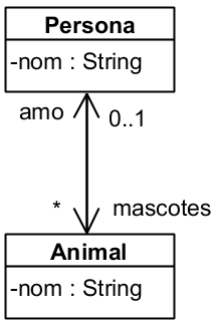
- Si una classe es comunica amb una altra, ho hem de marcar al diagrama de classes. Això indica la **navegabilitat**.
- Si una classe crida funcions d'una altra amb la que està relacionada, es pot posar el nom de rol/membre per a identificar l'objecte cridat.

Navegabilitat

- Indica si és possible o no travessar una associació binària d'una classe a una altra
  - Si A és navegable cap a B, des d'un objecte d'A es poden obtenir els objectes de B amb els que està relacionat.
- La navegabilitat d'una associació resultant del procés de disseny pot ser *bidireccional*, *unidireccional* o *no navegable.*

Navegabilitat bidireccional

D. Classes (Disseny)



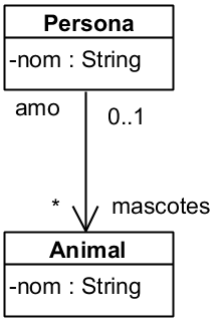
Codi

```
class Animal
{
public:
    string nom;
    Persona* amo;
};

class Persona
{
public:
    string nom;
    set<Animal*> mascotes;
};

void func(Animal* animal)
{
    string nomAmo = animal->amo->nom;
}

void func2(Persona* persona)
{
    for (Animal* mascota : persona->mascotes)
    {
        string nomAnimal = mascota->nom;
    }
}
```

- Les relacions navegables de forma bidireccional necessiten mantenir punters/referències a ambdues classes

Navegabilitat unidireccional

D. Classes (Disseny)



Codi

```
class Animal
{
public:
    string nom;
};

class Persona
{
public:
    string nom;
    set<Animal*> mascotes;
};

void func(Animal* animal)
{
    //string nomAmo = animal->amo->nom; //No compila!
}

void func2(Persona* persona)
{
    for (Animal* mascota : persona->mascotes)
    {
        string nomAnimal = mascota->nom;
    }
}
```

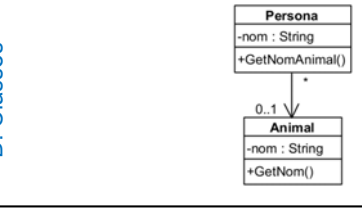
- Les relacions navegables de forma unidireccional necessiten mantenir punters/referències a una sola de les classes

Navegabilitat nul·la

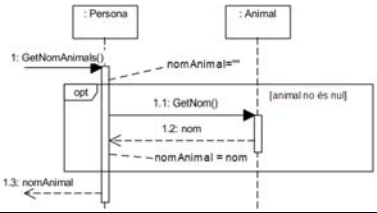
- Si una relació no és navegable en cap sentit, aleshores vol dir que no hi ha cap referència a ella dins del codi.
- Per tant, la pròpia relació pot desaparèixer del diagrama de classes de Disseny

Opcions / Condicionals

D. Classes



D. Seqüència



Codi

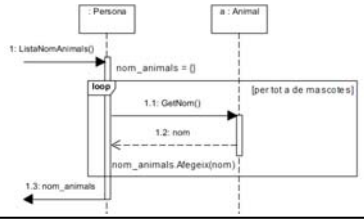
```
class Persona
{
public:
    string GetNomAnimal() {
        string nomAnimal = "";
        if (animal != nullptr) {
            nomAnimal = animal->GetNom();
        }
        return nomAnimal;
    }
private:
    Animal* animal;
    string nom;
};
```

- La forma de representar *Opcions* o *Condicions* al diagrama de seqüència serà mitjançant l'ús dels frames de **opt** i **alt**, de la mateixa manera que es feia als diagrames de seqüència d'especificació.

Agregats

D. Classes

D. Seqüència



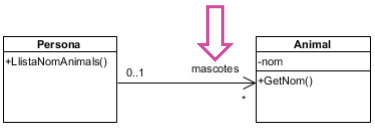
Codi

```
class Persona
{
private:
    set<Animal*> mascotes;
public:
    set<string> LlistaNomAnimals(){
        set<string> nomAnimals;
        for (Animal* a : mascotes){
            nomAnimals.insert(a->GetNom());
        }
        return nomAnimals;
    }
};

class Animal
{
private:
    string nom;
public:
    string GetNom() {
        return nom;
    }
};
```

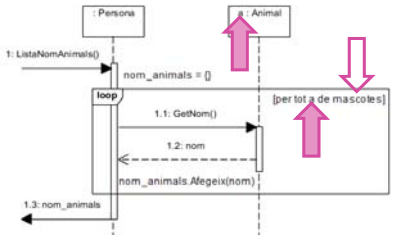
- Un agregat és una col·lecció d'objectes
- La forma de representar *iteracions sobre agregats* es fa mitjançant al frame de **loop** de la mateixa manera que es feia als diagrames de seqüència d'especificació

Agregats (II)

D. Classes



D. Seqüència



- Els noms de l'element sobre el que iterem ha de coincidir amb el digrama de classes.
- El nom que donem a cada element ha de coincidir amb el nom que donem a la instància

Agregats (recopilació)

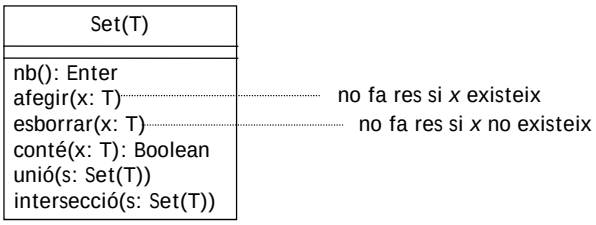
Un agregat és una col·lecció d'objectes

Sorgeixen en diversos contexts:

- Rols navegables amb multiplicitat més gran que 1
- Operacions que reben o retornen una col·lecció de valors
- Atributs multivaluats

En tots aquests casos, considerem l'agregat com un conjunt (Set)

- S'hi poden aplicar operacions següents (i només aquestes):

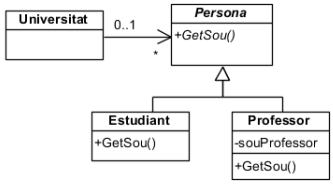


Polimorfisme

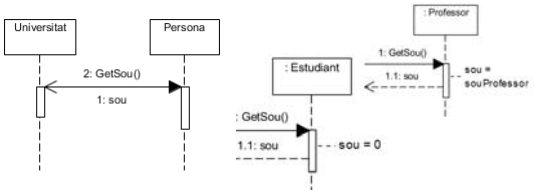
- És un mecanisme dels llenguatges de programació que permet que, en el cas de que dues funcions es diguin igual en una jerarquia de classes, aleshores el sistema cridarà en temps d'execució a la funció que es trobi implementada al nivell més inferior a la jerarquia.

Polimorfisme

D. Classes



D. Seqüència



Codi

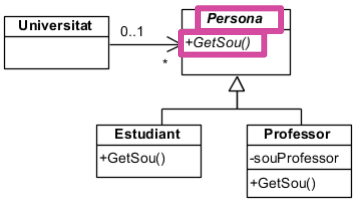
```
class Persona
{
public:
    virtual int GetSou() = 0;
private:
    string nom;
};

class Estudiant: public Persona
{
public:
    int GetSou() override {
        return 0;
    }
};

class Professor : public Persona
{
public:
    int GetSou() override {
        return souProfessor;
    }
private:
    int souProfessor;
};
```

- La forma de marcar el polimorfisme en el Diagrama de Seqüència és creant funcions amb el mateix nom dins de la mateixa estructura de Generalització / Especialització

Polimorfisme: Elements Abstractes

- Quan una operació dins d'una classe no te cap mena de codi, aquesta s'ha de definir com a **abstracta**.
- Això provoca que:
  - Tots els seus descendents dins de la jerarquia estan obligats a tenir aquesta funció.
  - La pròpia classe esdevé **abstracta** i no es poden declarar instàncies que siguin exclusivament del seu tipus.
  - Una superclasse només pot ser abstracta si la seva especialització és *complete*.

Polimorfisme: Elements Abstractes

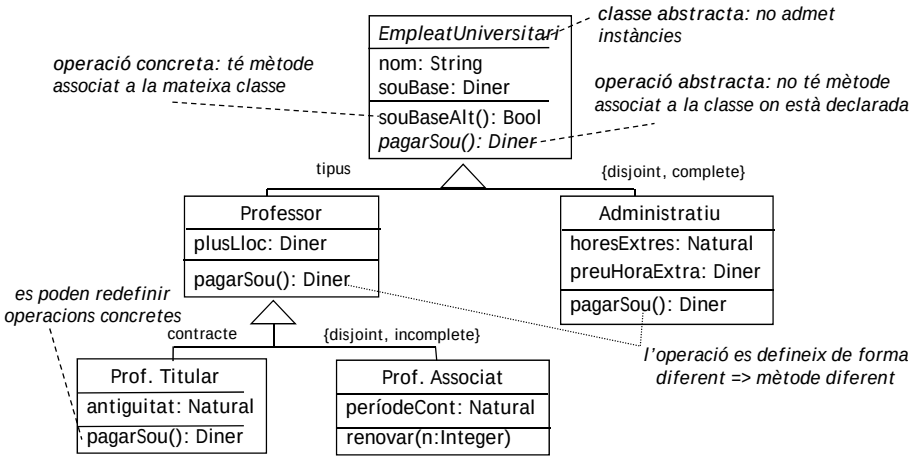


- La forma de marcar elements abstractes és posant el nom en cursiva
- A IES podeu posar una nota al costat de la classe/funció per a indicar-ho.
- **Extra**: Aquesta és una manera de tenir especialitzacions / generalitzacions de tipus **Complete**.

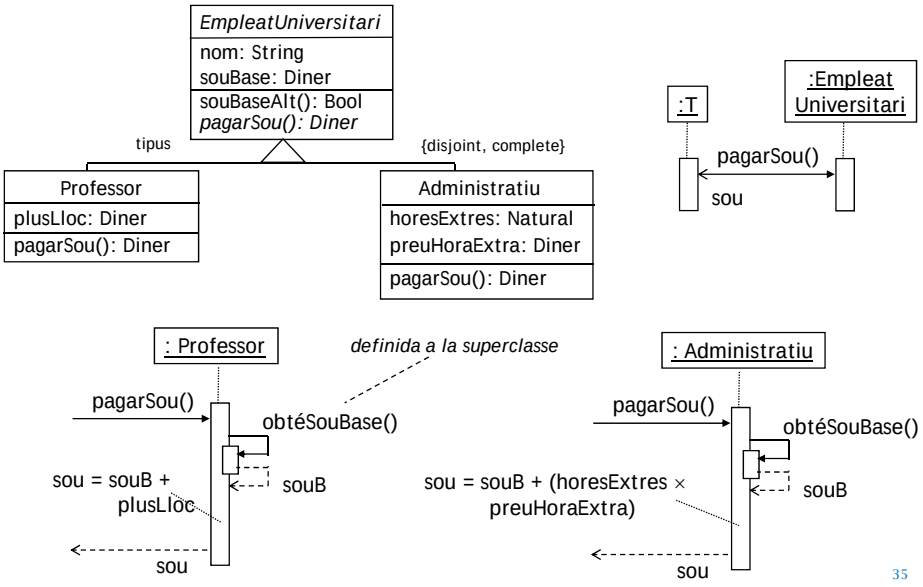
Polimorfisme

Operació polimòrfica:

– operació que s'aplica a diverses classes d'una jerarquia tal que la seva semàntica depèn de la subclasse concreta on s'aplica



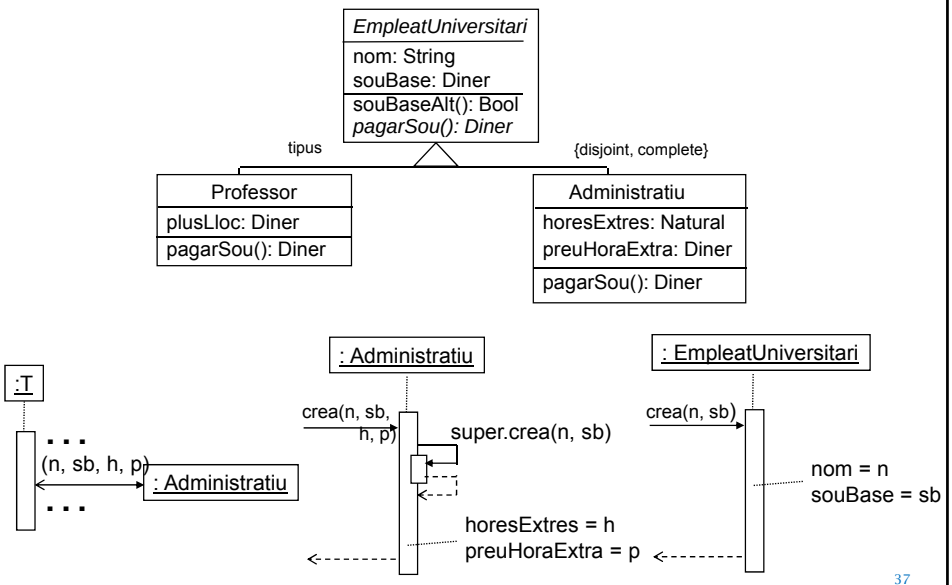
Polimorfisme: vinculació dinàmica



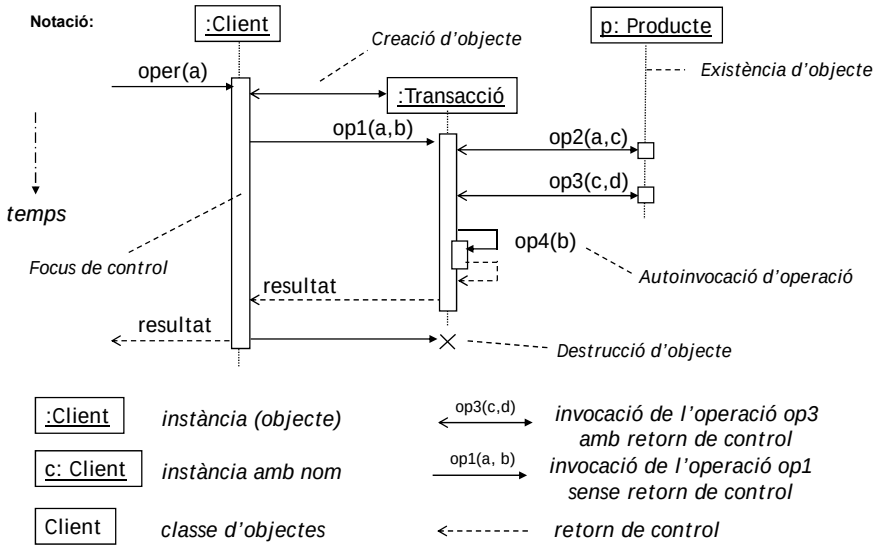
Creadores

- La forma de crear instàncies als llenguatge de programació és, normalment, mitjançant l'ús d'unes funcions **estàtiques** especials anomenades **Creadores**
- Una creadora rep com a paràmetres alguns (o tots, o cap) dels atributs de la classe que ha de crear i en retorna una instància.
- En el cas de les creadores de subclasses d'una jerarquia, estan obligades a cridar a la creadora de la seva superclasse per a completar la seva construcció.
- A IES assumirem la existència d'una creadora amb tots els atributs de la classe com a paràmetres.

Creació d'objectes en una jerarquia d'herència



Diagrames de seqüència: sintaxi completa



Diagrames de seqüència: convencions (1)

Noms dels objectes: batejar-los quan calgui per identificar el seu origen
- si són resultat d'una operació, usar el mateix nom en el resultat i en l'objecte
- si s'obtenen recorrent una associació amb multiplicitat 1, usar el nom del rol
- si han arribat com a paràmetres, usar el nom dels paràmetres

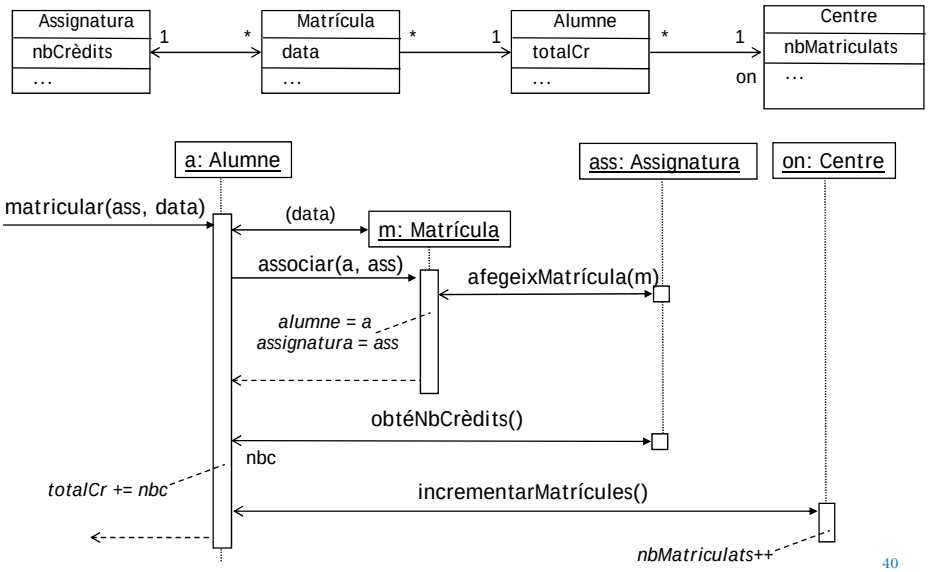
Paràmetres de les operacions:
- cal indicar explícitament amb quins paràmetres s'invoquen les operacions

Resultats de les operacions:
- donar nom al resultat si surt en algun altre lloc del diagrama
- també als valors retornats en els paràmetres

Comentaris:
- no deixar cap aspecte rellevant sense comentar
- en particular, ha de quedar clar:
  - ✓ com es calculen els resultats de les operacions
  - ✓ Quins valors es passen com a paràmetres a les operacions creadores
  - ✓ com es modifiquen els valors dels atributs i pseudo-atributs de les classes
- usar noms d'atributs, associacions, etc.
  - ✓ usar sintaxi Java per a operacions aritmètiques

No cal especificar el comportament de les operacions *getter* i *setter*

Diagrames de seqüència: convencions (2)



Bibliografia

- Larman, C. "*Applying UML and Patterns. An Introduction to Object-oriented Analysis and Design*", Prentice Hall, 2005, (3ª edició).
- http://www.uml.org/#UML2.3
- Meyer, B. "*Object-Oriented Software Construction*", Prentice Hall, 1997, cap. 3