Patró arquitectònic: Orientació a Objectes

- Context
- Problema
- Solució
- Elements d'UML propis de l'etapa de disseny
- Principis de disseny d'una arquitectura orientada a objectes
- Disseny per contracte
- Bibliografia

Context

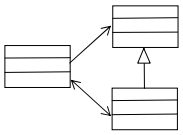
Un sistema que pot veure's com una col.lecció d'objectes que, en resposta a certs estímuls externs o esdeveniments interns, intercanvien informació, canvien el seu estat i eventualment produeixen resultats observables.

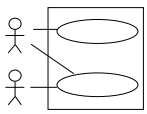
Problema

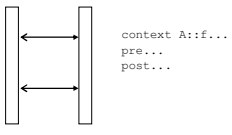
- Cal dissenyar un sistema com una col.lecció d'objectes tals que cadascun d'ells té unes responsabilitats assignades.
- El sistema ha de combinar dues visions: la visió estàtica, que defineix les propietats dels objectes que configuren els estats; i la visió dinàmica, que defineix les respostes dels objectes als esdeveniments que es produeixen (canvis d'estat, intercanvi d'informació, resultats observables, ...)
- Forces a equilibrar:
  - Canvis en el codi no haurien de propagar-se en tot el sistema (mantenibilitat)
  - Els components s'haurien de poder reutilitzar i reemplaçar per implementacions alternatives (reusabilitat, separació d'interfície i implementació)
  - Responsabilitats semblants s'haurien d'agrupar per afavorir la comprensibilitat i mantenibilitat (cohesió)
  - Es desitja portabilitat a d'altres plataformes

Solució

- Estructurar el sistema com una col.lecció d'objectes que tots junts configuren (part de) l'arquitectura.
- Assignar responsabilitats a aquests objectes de manera sistemàtica.
- Proporcionar una visió estàtica del sistema, declarant les classes a les que pertanyen els objectes, amb els seus atributs, operacions i interrelacions (associacions, herències, ...) juntament amb tota la informació que es considera rellevant a nivell de disseny (visibilitat, ...).
- Proporcionar una visió dinàmica del sistema, que identifica els esdeveniments que provoquen canvis en el sistema, i per a cadascun d'ells, la seqüència d'accions que en resulten.







Elements d'UML propis de l'etapa de disseny

- Conceptes generals
- Atributs
- Associacions
- Operacions
- Diagrames de seqüència
- Polimorfisme
- Interfícies
- Paquets

Conceptes generals: Visibilitat

- La visibilitat defineix quins objectes tenen dret a consultar i eventualment modificar informació declarada en un diagrama de classes
- La visibilitat dels elements d'un diagrama de classes pot influir en els factors de qualitat de l'arquitectura, per això cal establir-la en la seva vista lògica
- En UML, la visibilitat s'estableix sobre:
  - Atributs d'una classe
  - Operacions d'una classe
  - Rols d'associacions accessibles des d'una classe
- En UML, l'element *x* definit a la classe *C* pot ser:
  - *Public* (+): qualsevol classe que veu *C*, veu *x*
  - *Private* (-): *x* només és visible des de *C*
  - *Protected* (#): *x* és visible des de *C* i des de tots els descendents de *C* (si n'hi ha)

Conceptes generals: Àmbit

- L'àmbit defineix si determinats elements de disseny són aplicables a objectes individuals o a la classe que defineix els elements
- En UML, l'àmbit s'estableix sobre:
  - Atributs d'una classe
  - Operacions d'una classe
- En UML, l'element *x* definit a la classe *C* pot ser d'àmbit:
  - D'instància (no estàtic): *x* està associat als objectes de *C*
    referència: obj.x, essent *obj* una instància de la classe *C*
  - De classe (estàtic): *x* està associat a *C*
    referència: C.x

Atributs: Sintaxi completa

[visibilitat] nom [: tipus] [multiplicitat] [= valor-inicial] [{propietats}]

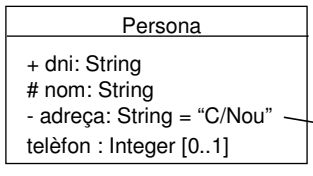
Univaluat (per defecte)
Admet valors nuls: [0..1]
Multivaluat: [1..*]
etc.

No les usem en aquesta assignatura

Public (+): qualsevol classe que veu la classe, veu l'atribut
Protected (#): només la pròpia classe i els seus descendents veuen l'atribut
Private (-): només la pròpia classe veu l'atribut

Suposarem que els atributs són privats, a no ser que especifiquem una altra visibilitat

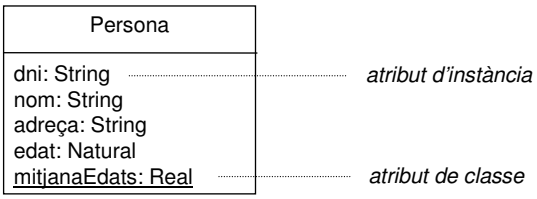
operacions *getter* i *setter*



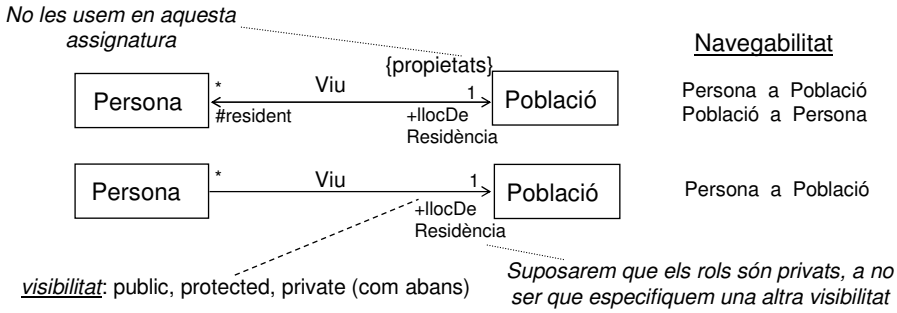
obtéAdreça(): String
posaAdreça(n: String)

Atributs: Àmbit

- Atribut d'instància:
  - representa una propietat aplicable a tots els objectes d'una classe.
  - cada objecte pot tenir un valor diferent d'aquest atribut.
- Atribut de classe:
  - propietat aplicable a la classe d'objectes com a tal.
  - no és aplicable a les instàncies de la classe.



Associacions



Navegabilitat: resultant del procés de disseny

- 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 què està relacionat
  - efecte: *A* té un pseudo-atribut amb la mateixa multiplicitat del rol
- Les associacions no navegables en cap sentit podrien desaparèixer de la vista lògica
  - però les podem deixar si el disseny és incomplet o per motius de canviabilitat (extensibilitat)
- La navegabilitat té un fort impacte en l'avaluació de l'arquitectura

Operacions: Sintaxi completa

No les usem en aquesta assignatura

Signatura d'una operació:

[visibilitat] nom [(llista-paràmetres)] [: tipus-retorn] [{propietats}]

Public (+): l'operació pot ser invocada des de qualsevol objecte
Protected (#): pot ser invocada pels objectes de la classe i els seus descendents
Private (-): només els objectes de la pròpia classe poden invocar l'operació

Suposarem que les operacions són públiques, a no ser que especifiquem una altra visibilitat

Paràmetres d'una operació:

[direcció] nom: tipus [multiplicitat] [= valor-per-defecte]

in, out, inout

Suposarem que els paràmetres són d'entrada (in), a no ser que especifiquem una altra direcció

Operacions: Àmbit

- Operació d'instància:
  - l'operació és invocada sobre objectes individuals
- Operació de classe:
  - l'operació s'aplica a la classe pròpiament dita
  - exemple: operacions constructores que serveixen per donar d'alta noves instàncies d'una classe

Alumne
nom: String edat: Natural
nom?(): String novaAssignatura (nom: String): Boolean alumne (nom: String, edat: Natural) mitjanaEdats(): Real

Operació constructora:

Suposarem que, si no s'indica el contrari, cada classe d'objectes té una operació constructora amb tants paràmetres com atributs té la classe.
Si es consideren altres constructores, caldrà definir-ne la seva signatura.

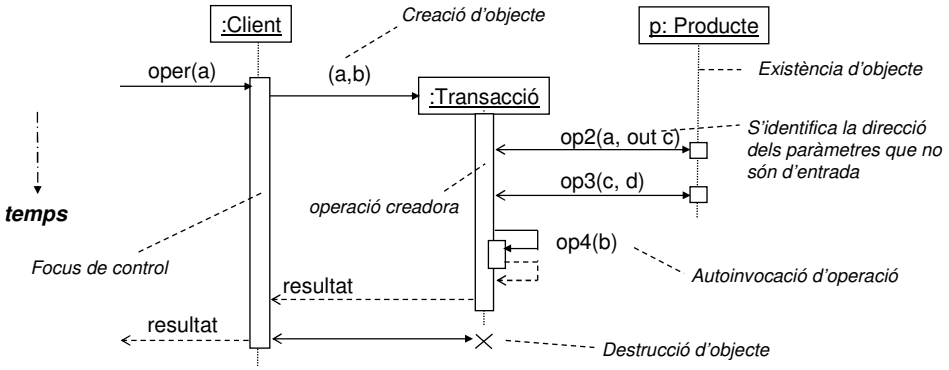
Agregats

- 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 multi-valuats
- En tots aquests casos, considerem l'agregat com un conjunt (Set)
  - S'hi poden aplicar operacions següents (i només aquestes):

Set(T)	
nb(): Enter	
afegir(x: T)	no fa res si *x* existeix
esborrar(x: T)	no fa res si *x* no existeix
conté(x: T): Bool	
unió(s: Set(T))	
intersecció(s: Set(T))	

  - La constructura, l'abreugem amb el símbol de conjunt buit Ø → creació de s: s = Ø

Diagrames de seqüència: Sintaxi completa

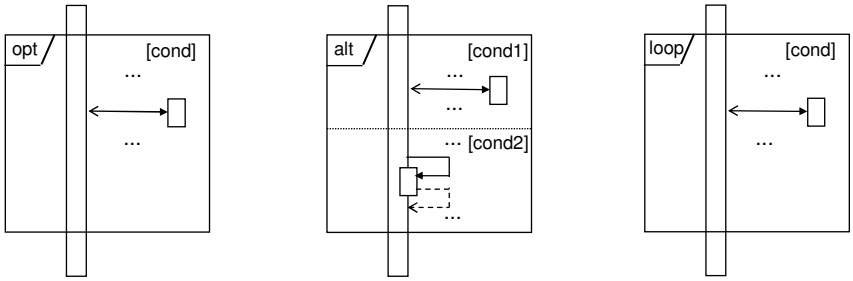


Notació:

:Client	*instància (objecte)*	op3(c, d)	*invocació de l'operació op3 amb retorn de control*
c: Client	*instància amb nom*	(a, b)	*invocació de l'operació creadora sense retorn de control*
Client	*classe d'objectes*		*retorn de control*

Diagrames de seqüència: Ús de *frames*

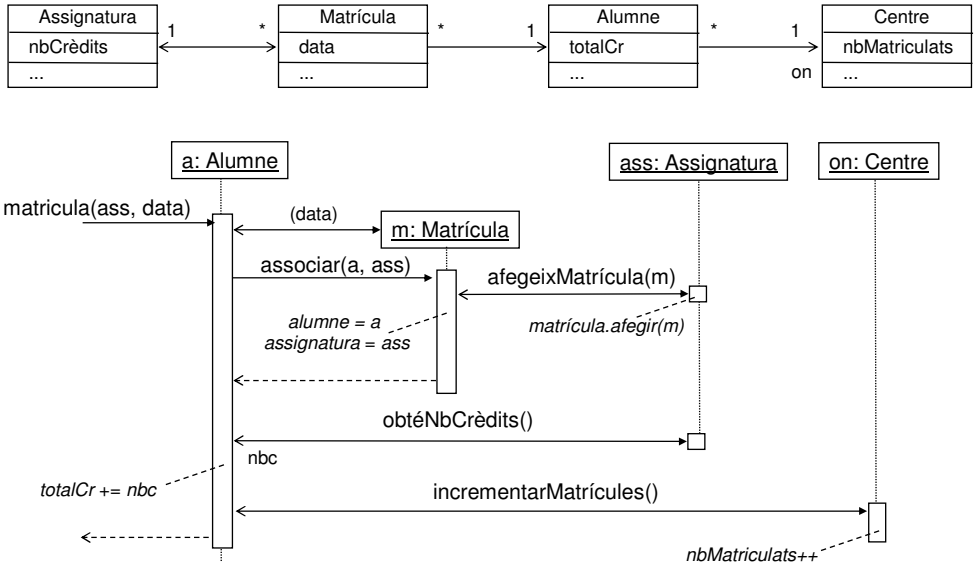
- Els *frames* permeten estructurar informació en els diagrames UML
  - aquí ens centrem en el seu ús en el context dels diagrames de seqüència
- Tres tipus principals de *frames*:
  - execució opcional (*opt*)
  - execució alternativa (*alt*)
  - execució repetida (*loop*)
- Els *frames* es poden aniuar lliurement



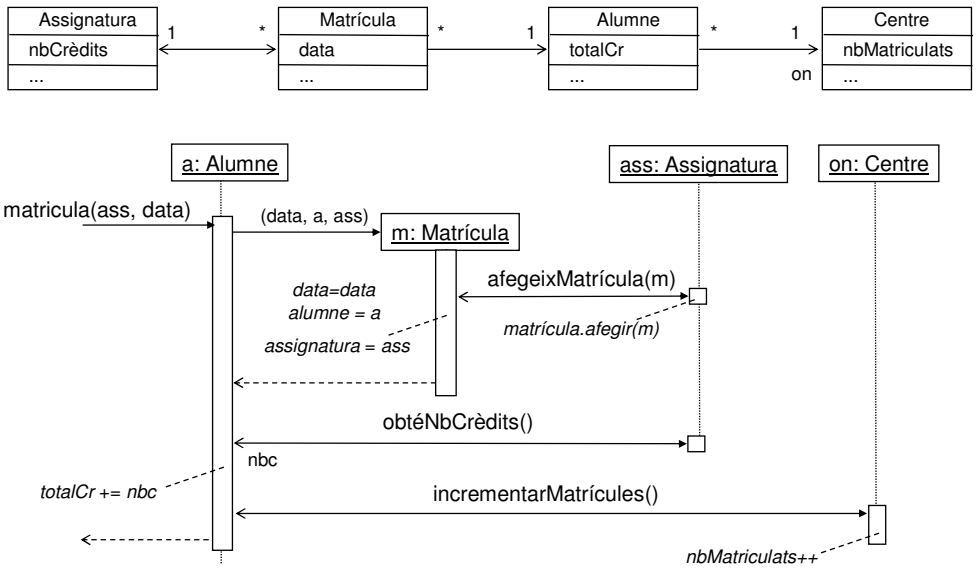
Diagrames de seqüència: Convencions (1)

- 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
  - si no són d'entrada, declarar la direcció
- Resultats de les operacions:
  - donar nom al resultat si surt en algun altre lloc del diagrama
- Comentaris:
  - no deixar cap aspecte rellevant sense comentar
  - en particular, ha de quedar clar:
    - ✓ com es calculen els resultats de les operacions
    - ✓ quins són els valors que 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*

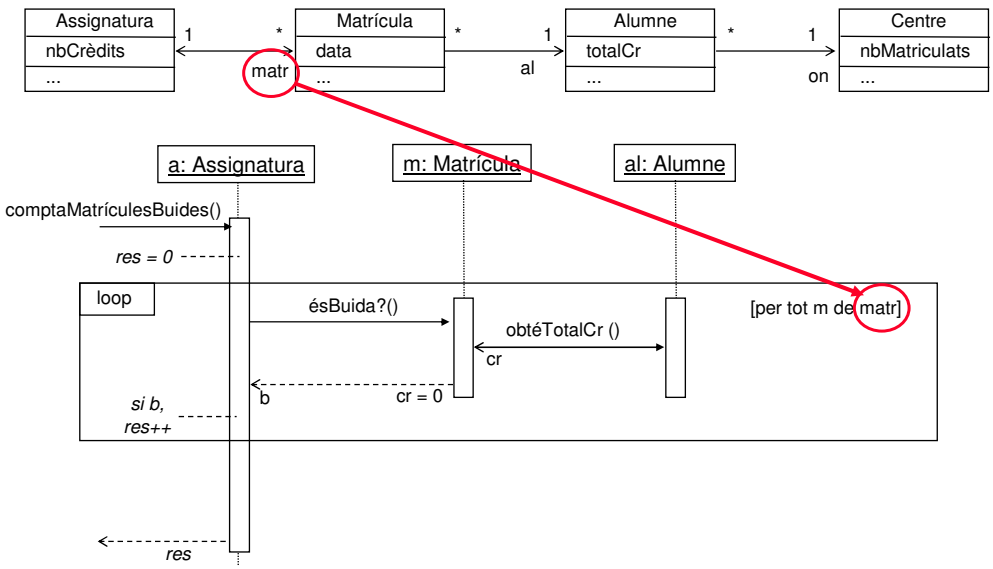
Diagrames de seqüència: Convencions (2)



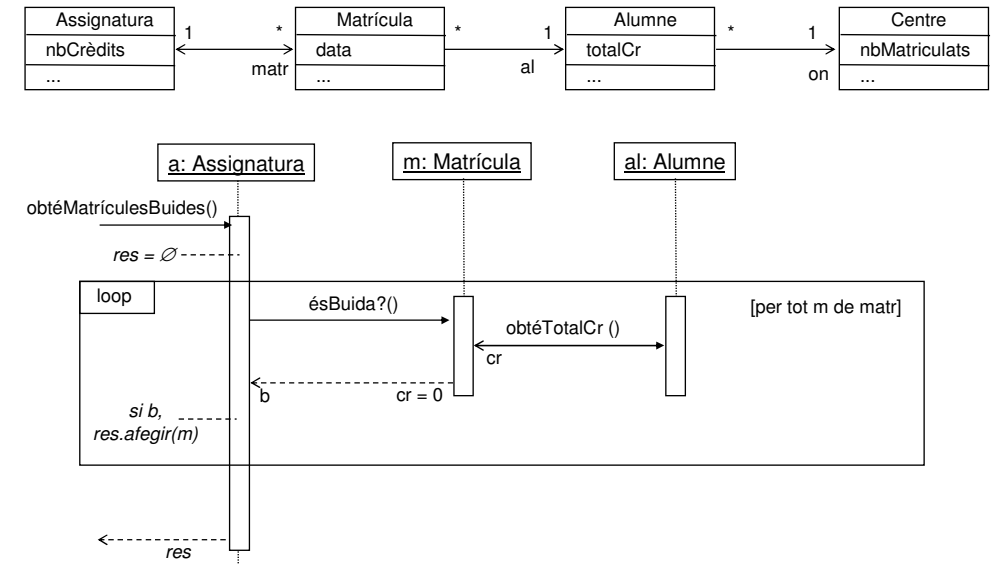
Diagrames de seqüència: Convencions (3)



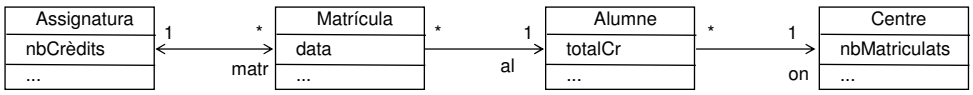
Diagrames de seqüència amb agregats (1)

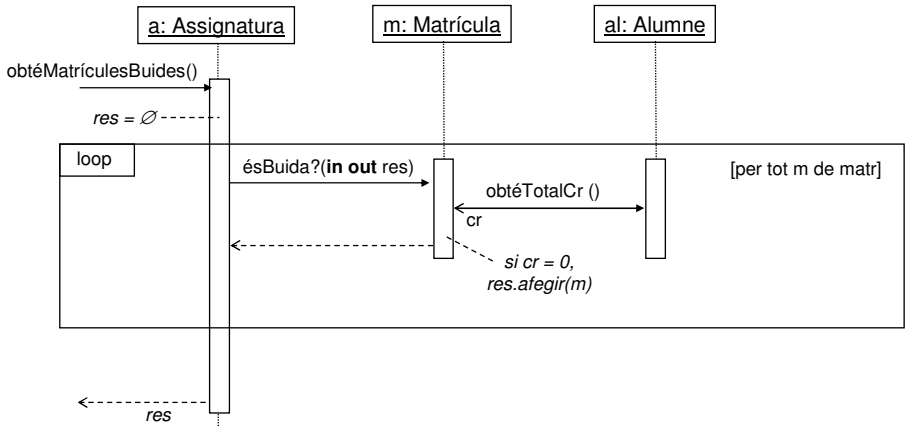


Diagrames de seqüència amb agregats (2)



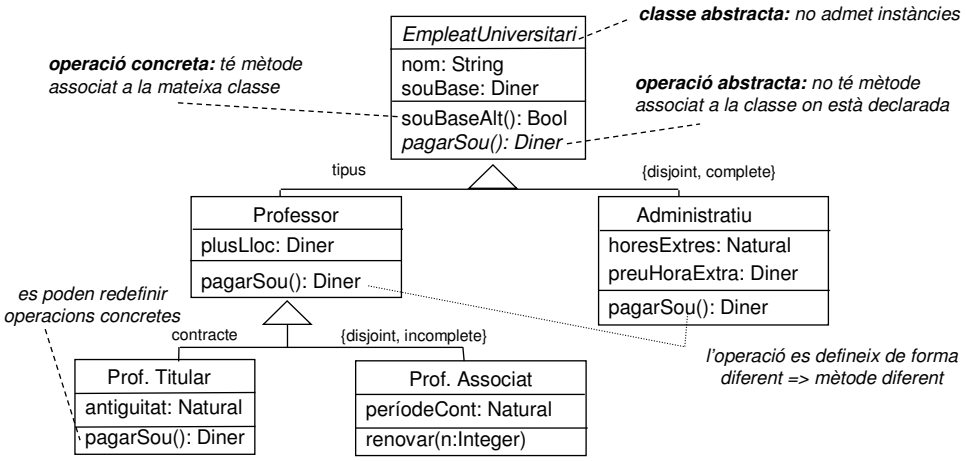
Diagrames de seqüència amb agregats (3)



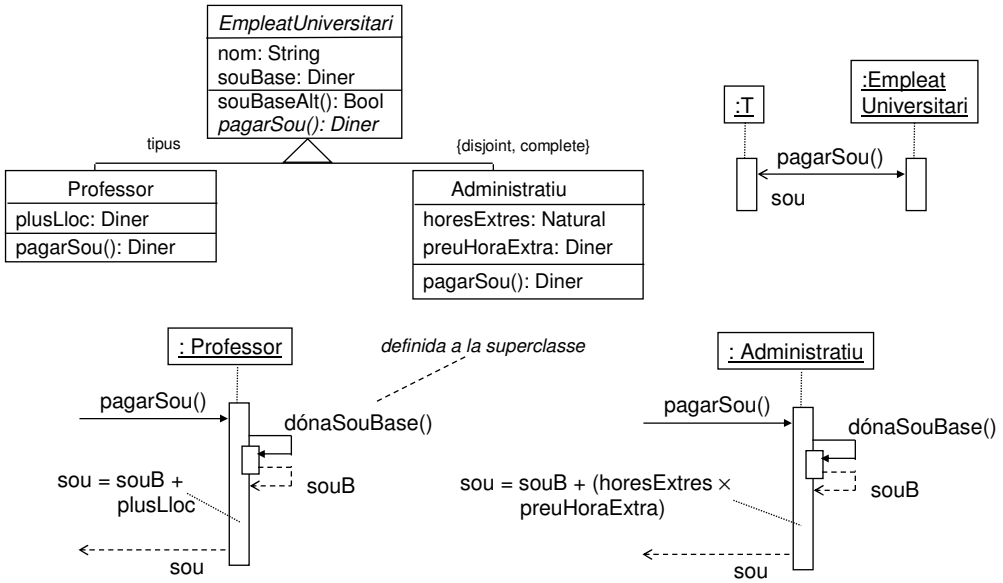


Polimorfisme

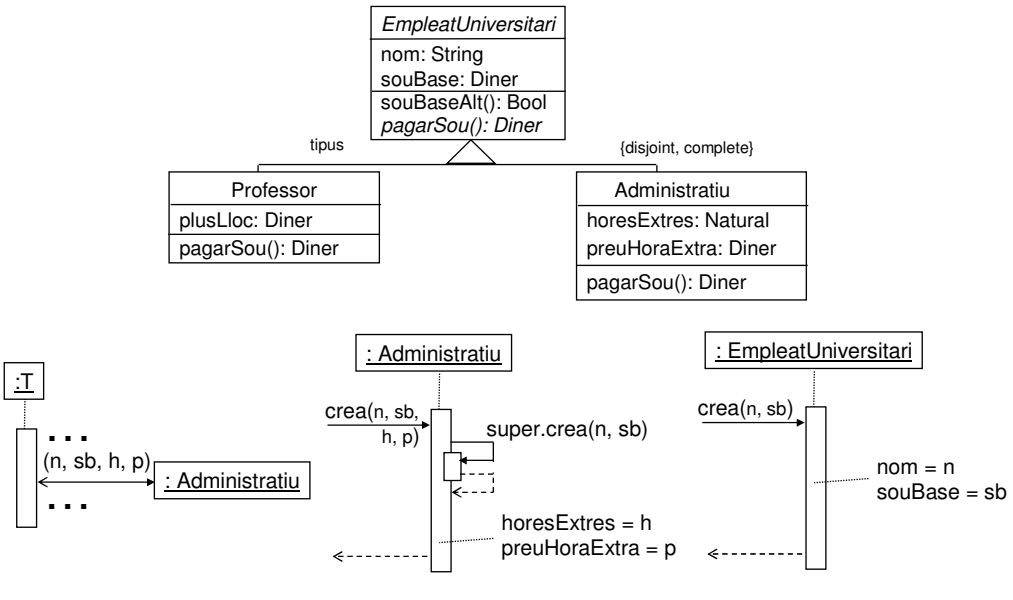
- Operació polimòrfica:
  - operació que s'aplica a diverses classes d'una jerarquia tal que la seva semàntica depèn de la subclasse concreta on s'aplica



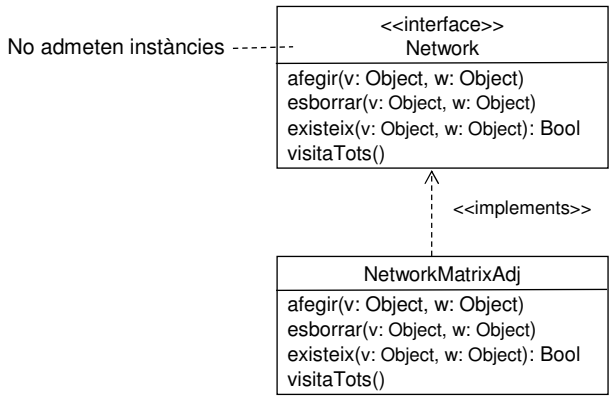
Polimorfisme: Vinculació dinàmica



Creació d'objectes en una jerarquia d'herència



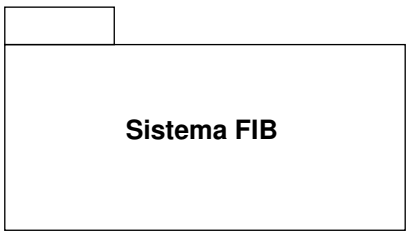
Interfícies



En els diagrames de seqüència, es comporten com classes abstractes

Paquets

Encapsulen classes interrelacionades



Es converteixen en l'unitat atòmica per al desplegament, la reusabilitat i el manteniment:

- totes les classes d'un paquet es despleguen juntes
- totes les classes d'un paquet es reusen juntes
- un canvi que afecta un paquet, afecta totes les classes d'aquest paquet i cap altre paquet

Principis de disseny d'una arquitectura orientada a objectes
Símptomes d'un disseny deficient

- *Rigidesa*. Difícil de canviar. Canvis simples impliquen canvis en molts mòduls.
- *Fragilitat*. Quan es fa un petit canvi, sorgeixen problemes en parts que no haurien d'estar afectades.
- *Immobilitat*. Conté parts útils en altres llocs, però difícils d'extreure.
- *Viscositat*. Difícil de fer canvis preservant el disseny original.
- *Complexitat innecessària*. Conté elements que de moment no s'usen, i que probablement no s'usaran en el futur.
- *Repetició innecessària*. Codi redundant que dificulta el canvi.
- *Opacitat*. Difícil d'entendre.

[cf. Martin (2003), pp. 88-89]

Principis de disseny d'una arquitectura orientada a objectes

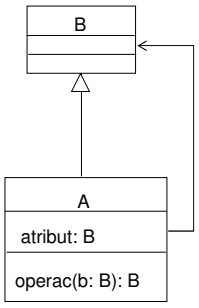
- Regeixen la construcció d'arquitectures de qualitat
- Tracten de satisfer els següents objectius dels mòduls software:
  - satisfer la seva funcionalitat prevista
  - estar preparat per als canvis
  - comunicar-se amb els seus lectors
- Aquests objectius donen lloc a diversos principis; ens centrem en l'estudi de:
  - l'acoblament → principi de l'acoblament baix
  - la cohesió → principi de la cohesió alta
  - l'extensió → principi obert-tancat
- L'aplicació dels patrons de disseny es farà tenint en compte aquests principis

Principi de l'acoblament baix

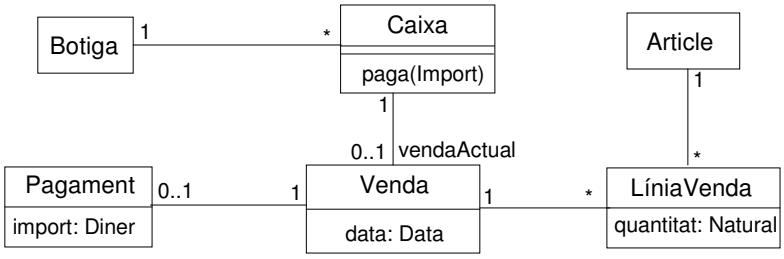
- **Acoblament.** Mesura fins a quin punt un mòdul és connectat a, té coneixement de, o es recolça en, altres mòduls. En el nostre context, els dos tipus de mòduls que ens interessen són paquets i classes
- Convé que l'acoblament sigui baix:
  - Si hi ha un acoblament de A a B, un canvi en B pot implicar canviar A.
  - Quan més acoblament té un mòdul, més difícil resulta comprendre'l aïlladament.
  - Quan més acoblament té un mòdul, és més difícil de reutilitzar-lo, perquè requereix la presència dels altres mòduls.
- Consideracions addicionals:
  - L'acoblament amb mòduls estables ben coneguts no acostuma a ser problema (tipus de dades, biblioteques ofertes pel llenguatge de programació, ...).
  - Cal evitar especialment els acoblaments cap a elements de més baix nivell d'abstracció
  - Cal evitar especialment els acoblaments entre paquets.

Acoblament entre classes

- **Acoblament** d'una classe és una mesura del grau de connexió, coneixement i dependència d'aquesta classe respecte d'altres classes.
- Hi ha un acoblament de la classe A a la classe B si:
  - A té un atribut de tipus B
  - A té una associació navegable amb B
  - B és un paràmetre o el retorn d'una operació de A
  - Una operació de A referencia a un objecte de B
  - A és una subclasse directa o indirecta de B



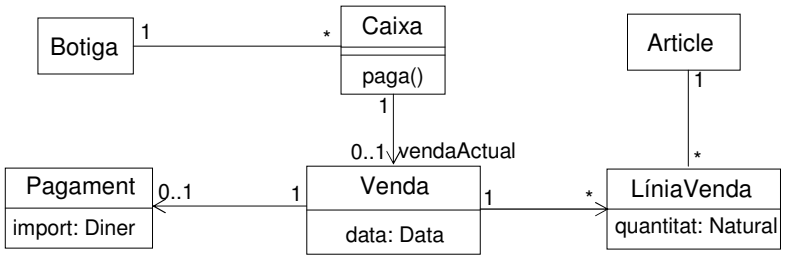
Acoblament: Exemple (1)



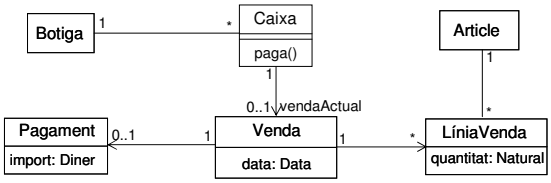
L'operació *paga* crea un nou pagament *p* amb l'import donat, i l'associa amb la venda actual de la caixa (se suposa que la caixa té venda actual)

Acoblament: Exemple (2)

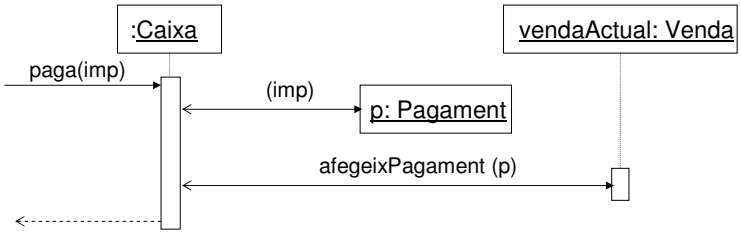
Suposem un disseny que en un moment determinat presenta la navegabilitat següent i no té més acoblaments dels que es dedueixen del diagrama:



Acoblament: Exemple (3)

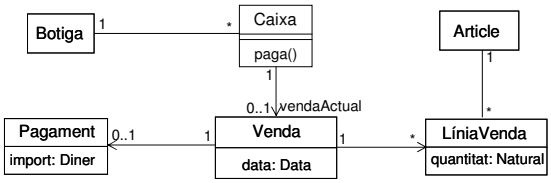


Alternativa 1: *Caixa* crea pagament i l'associa a *Venda*

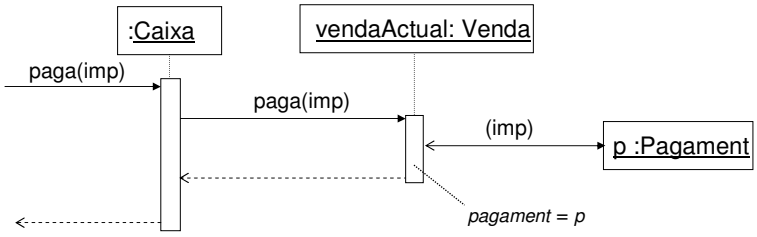


Introdueix un nou acoblament entre *Caixa* i *Pagament*

Acoblament: Exemple (4)



Alternativa 2: *Caixa* propaga l'operació a *Venda*



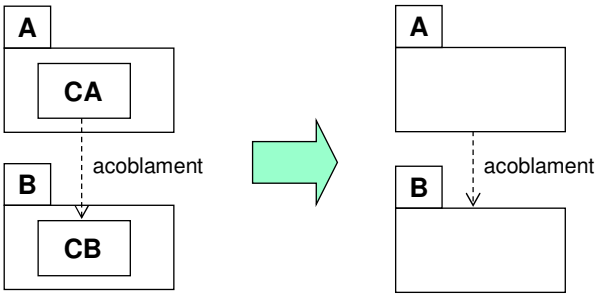
No introdueix cap nou acoblament

Llei de Demeter

- Una operació només hauria d'invocar operacions ("parlar") d'objectes accessibles des de *self* ("familiars"), que són:
  - L'objecte que està executant l'operació (*self*)
  - Un paràmetre rebut per l'operació
  - Els valors dels atributs de l'objecte *self*
  - Els objectes associats amb *self*
  - Els objectes creats per la pròpia operació
- Tots els altres objectes són "estranys". Per això, la llei també es coneix com a "No parleu amb estranys".
- La llei de Demeter ajuda a mantenir l'acoblament baix

Acoblament entre paquets

- Un paquet A està acoblat amb un paquet B si alguna classe d'A està acoblada amb alguna classe de B



Principi important de l'acoblament entre paquets:

- Aciclicitat en el graf de dependències

Cohesió

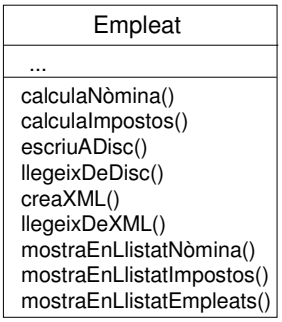
- **Cohesió** d'una classe és una mesura del grau de relació i de concentració de les diverses responsabilitats (atributs, associacions i operacions)

Principi de la cohesió alta

- Convé que la cohesió sigui alta
- Una classe amb cohesió alta:
  - Té poques responsabilitats en una àrea funcional
  - Col·labora (delega) amb d'altres classes per a fer les tasques
  - Acostuma a tenir poques operacions. Aquestes operacions estan molt relacionades funcionalment
- Avantatges:
  - Fàcil comprensió
  - Fàcil reutilització i manteniment
- No existeix una mètrica quantitativa simple de la cohesió
  - Avaluació qualitativa

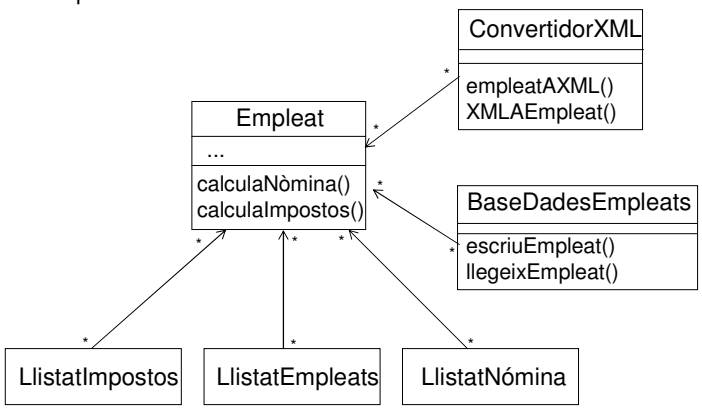
Cohesió: exemple (1)

Exemple de cohesió baixa:



Cohesió: exemple (2)

Exemple de cohesió alta:

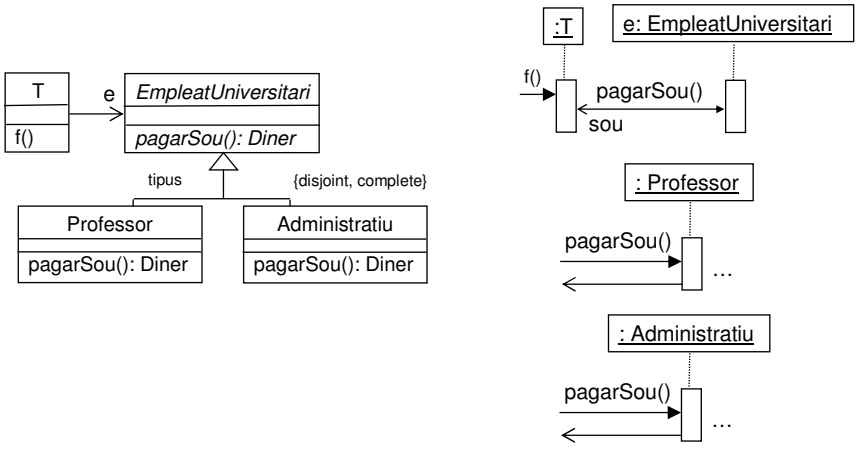


El principi Obert-Tancat (OCP)

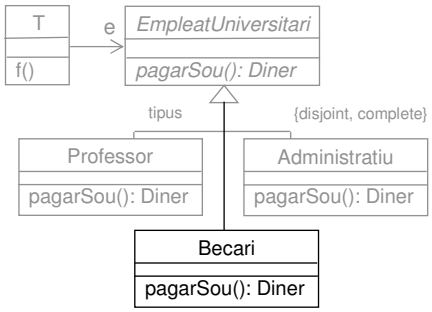
- Els mòduls (classes, funcions, etc.) haurien de ser:
  - *Oberts* per a l'extensió. El comportament del mòdul es pot estendre per tal de satisfer nous requisits.
  - *Tancats* per a la modificació. L'extensió no implica canvis en el codi del mòdul. No s'ha de tocar la versió executable del mòdul.
- El comportament dels mòduls que satisfan aquest principi es canvia afegint nou codi, i no pas canviant codi existent.
- L'ús correcte del polimorfisme afavoreix aquest principi

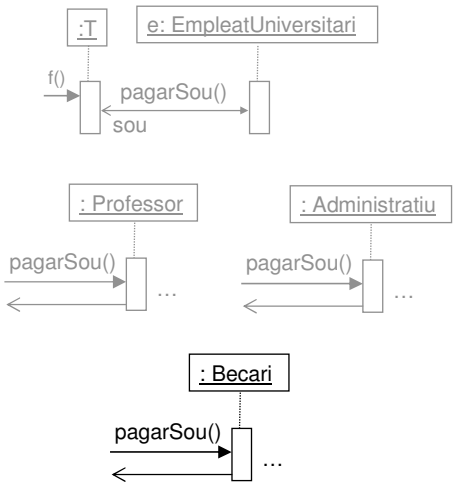
El principio Obert-Tancat (OCP)
Satisfacció

Totes les entitats software (classes, mòduls, funcions, etc.) haurien d'estar obertes per extensió, però tancades per modificació



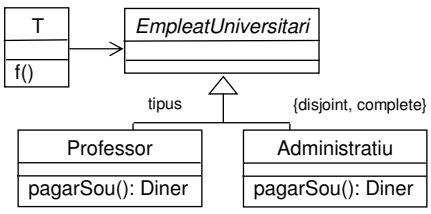
El principio Obert-Tancat (OCP)
Nou tipus d'empleat: becari

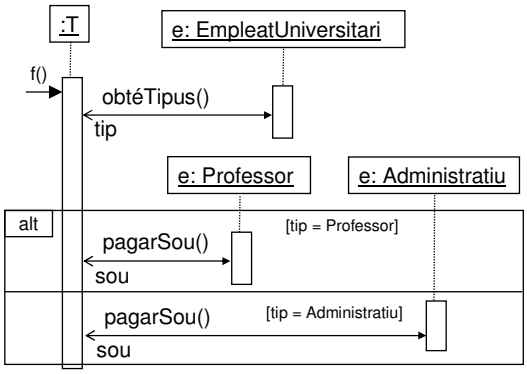




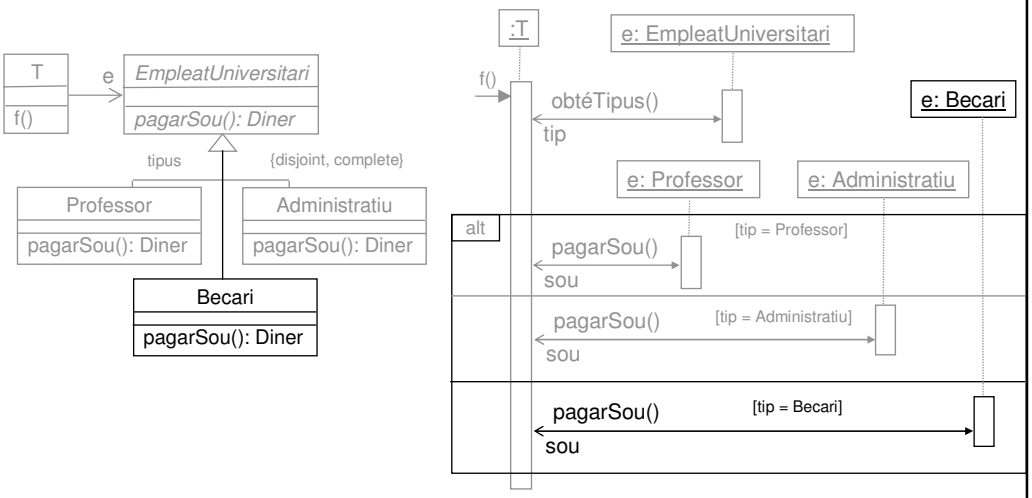
El principio Obert-Tancat (OCP)
Violació

Totes les entitats software (classes, mòduls, funcions, etc.) haurien d'estar obertes per extensió, però tancades per modificació





El principio Obert-Tancat (OCP)
Nou tipus d'empleat: becari



Disseny per Contracte
Pre/postcondicions i la fórmula de correctesa

- Tota operació *A* del sistema té:
  - una precondició *P*
  - una postcondició *Q*

  on *P* i *Q* són assercions tals que es satisfà l'anomenada *fórmula de correctesa*:

$$\{P\}\ A\ \{Q\}$$

⇨ "Qualsevol execució de *A* en un estat que satisfà *P* acabarà en un estat que satisfà *Q*."

- Sinecures:
  - { false } A { ... }
  - { ... } A { true }

Disseny per Contracte
Obligacions i beneficis

"Si em prometeu cridar l'operació *O* amb *P* (pre) satisfeta, llavors, a canvi, jo us prometo donar-vos un estat final en què *Q* (post) és satisfeta".

	Obligacions	Beneficis
Client	Invoca l'operació en un estat que satisfà P	Obté un estat que satisfà Q
Proveïdor	En acabar l'operació, se satisfà Q	Suposa que P ja se satisfà

Disseny per Contracte
Exemple de contracte

Article
estoc: Natural
potsServir(quantitat: Natural): Boolean serveix(quantitat: Natural)

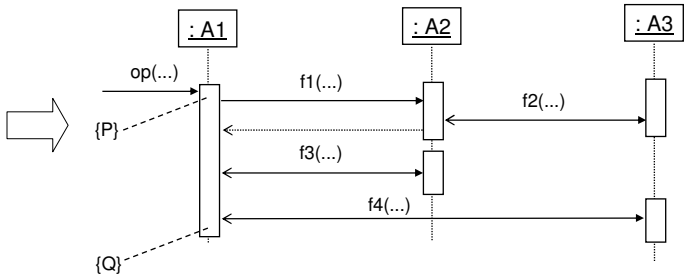
```
context Article::serveix(quantitat: Natural)
  pre hi-ha-estoc: la quantitat a servir és <= l'estoc
  post actualitza-estoc: el nou estoc és l'anterior menys la quantitat

context Article::potsServir(quantitat: Natural): Boolean
  post valida-estoc: cert si quantitat <= estoc; fals en cas contrari
```

Disseny per Contracte
Utilització a la fase de disseny

- Els serveis de cada capa del sistema seran especificats amb pre/post
- El disseny d'una operació ha de satisfer la fórmula de correctesa del contracte:
  - No cal controlar explícitament les precondicions
  - Cal assegurar que la solució donada satisfà la postcondició

```
context A1::op(...)
  pre nom-P: P
  post nom-Q: Q
```



Disseny per Contracte
Tractament dels errors

- No sempre les condicions anòmales s'han d'expressar com a precondicions:
  - El client no té tota la informació per assegurar que l'error no es produirà
  - Fins i tot si la té, es pot considerar que no és convenient
    - Operació vulnerable
    - Massa feina per al client

- Distingim doncs dues situacions:
  - L'operació dóna per suposat que en invocar-la, no es pot produir la condició anòmala → precondició
  - L'operació verifica si es compleix o no la situació d'error → excepció

```
context A1::op(...)
  pre nom-P: P
  exc nom-E: E
  post nom-Q: Q
```

Disseny per Contracte
Obligacions i beneficis en presència d'excepcions

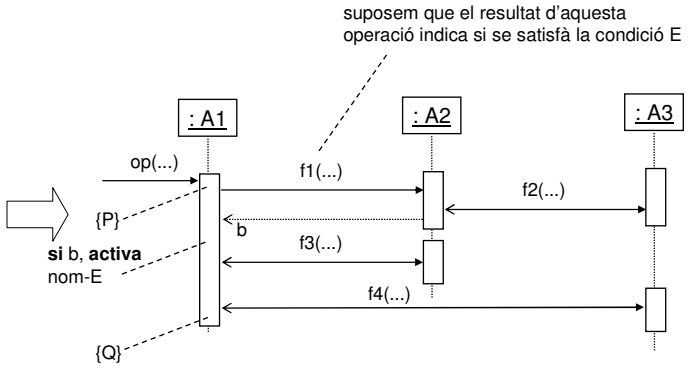
"Si em prometeu cridar l'operació *O* amb *P* (pre) satisfeta, llavors, a canvi, jo us prometo donar-vos un estat final en què *Q* (post) és satisfeta, a no ser que se satisfaci la condició *E* (exc), en el qual cas l'estat no canviarà".

	Obligacions	Beneficis
Client	Invoca l'operació O en un estat que satisfà P	• Si E se satisfà, reb notificació i sap que l'estat no canvia • Si no reb notificació que E se satisfà, obté un estat que satisfà Q
Proveïdor	• Detecta i notifica si se satisfà E, en el qual cas no canvia l'estat • Si E no se satisfà, en acabar l'operació, se satisfà Q	Suposa que P ja se satisfà

Disseny per Contracte
Detecció i notificació de les excepcions

- Quan una operació detecta una situació d'error declarada a l'apartat d'excepcions, diem que "activa" l'excepció

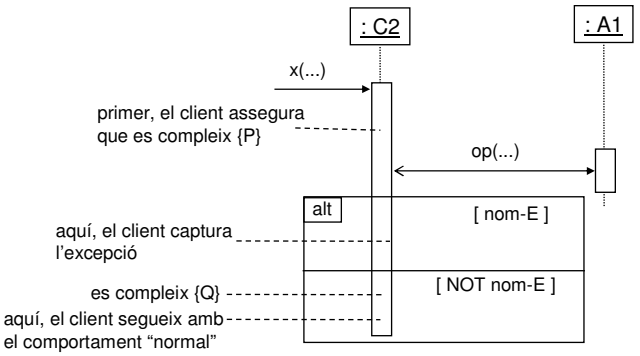
```
context A1::op(...)
  pre nom-P: P
  exc nom-E: E
  post nom-Q: Q
```



- Convenció important: suposem que "activa" provoca que es desfacin els possibles canvis de l'estat del sistema que l'operació hagués efectuat

Disseny per Contracte
Captura de les excepcions

- Quan un client rep notificació que s'ha produït una excepció, pot tractar-la si ho considera convenient

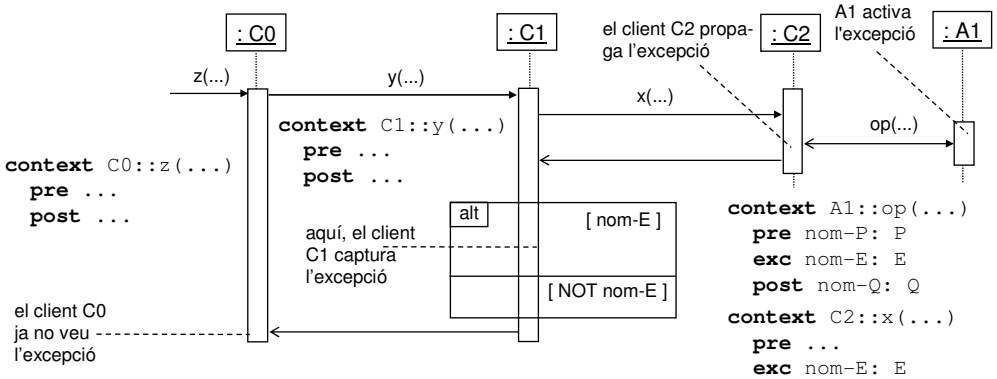



```
context C2::x(...)
  pre ...
  post ...
```

```
context A1::op(...)
  pre nom-P: P
  exc nom-E: E
  post nom-Q: Q
```


Disseny per Contracte
Propagació de les excepcions

- Quan un client no vol capturar una possible excepció, simplement la propaga al seu propi client
  - En algun punt de la cadena de crides, algun client ha de capturar-la
  - Suposem que en propagar, també es desfan els possibles canvis efectuats en l'estat del sistema durant l'execució de l'operació




```
context C0::z(...)
  pre ...
  post ...
```

```
context C1::y(...)
  pre ...
  post ...
```

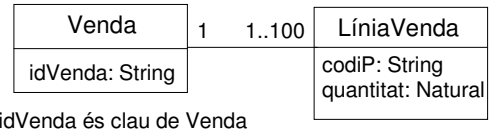
```
context A1::op(...)
  pre nom-P: P
  exc nom-E: E
  post nom-Q: Q
```

```
context C2::x(...)
  pre ...
  exc nom-E: E
  post ...
```

Disseny per Contracte
Invariants de classe

- L'invariant d'una classe és un predicat que tota instància de la classe ha de satisfer durant la seva existència sempre que el sistema estigui en una situació estable
  - considerem que el sistema no està estable mentre s'està executant un cas d'ús
- Donat un cas d'ús que executa, en aquest ordre, les operacions A1, A2, ..., A*n*, tals que $\{P_k\}\ A_k\ \{Q_k\}$, i essent *X* la conjunció dels invariants de les classes implicades en el cas d'ús, es compleix $\{P_1 \wedge X\}\ A_1\ \{Q_1\}$ i $\{P_n\}\ A_n\ \{Q_n \wedge X\}$
  - si el cas d'ús té una única operació {P} A {Q}, llavors es compleix $\{P \wedge X\}\ A\ \{Q \wedge X\}$
  - si només una de les P_k modifica l'estat del sistema, llavors sí que podem dir $\{P_k \wedge X\}\ A_k\ \{Q_k \wedge X\}$ per totes les P_k
- Típicament, l'invariant es dedueix de les restriccions d'integritat del model conceptual de dades:
  - gràfiques: multiplicitat, subset, etc.
  - textuals: de clau, etc.

Disseny per Contracte
Invariants de classe, exemple

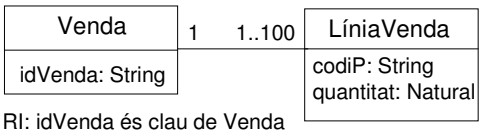


RI: idVenda és clau de Venda

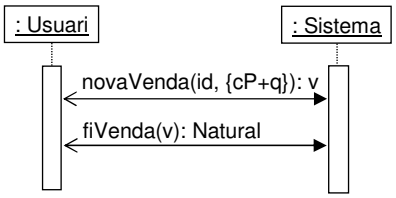
```
context Venda inv venda-té-clau: idVenda és clau de Venda
context Venda inv venda-té-línies: una Venda té entre 1 i 100 LíniaVenda
context LíniaVenda inv línia-és-de-venda: una LíniaVenda està associada amb una única Venda
```

- Convenció: per no repetir feina innecessàriament, i si no es diu el contrari, no escriurem explícitament els invariants, atès que són deduïbles de l'especificació del sistema
  - restriccions de multiplicitat de rol: s'associen a la classe de l'altre extrem
  - restriccions de qualificació d'herència (completitud, solapament): s'associen a la superclasse
  - restriccions textuals: s'associen a la classe més fortament implicada

Disseny per Contracte
Invariants de classe, impacte en els contractes



RI: idVenda és clau de Venda




```
context Venda::venda(id: String, s: Set(cP: String + q: Natural))
post nova-venda: crea una nova Venda v amb id
post noves-línies: per a cada element de s,
            crea una nova LíniaVenda associada amb v
```

post venda-té-clau, venda-té-línies

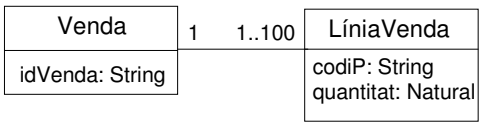
```
context Venda::nbLínies(): Natural
post dóna-resultat: resultat = nombre de línies de la venda
```

pre, post venda-té-clau, venda-té-línies

```
context LíniaVenda::líniaVenda(cp: String, q: Natural, v: Venda)
post nova-línia: crea una nova LíniaVenda lv amb c i q
post associa-venda: associa lv amb v
```

post línia-és-de-venda

Disseny per Contracte
Connexió entre invariants, precondicions i excepcions




```
context Venda inv venda-té-clau: ...
context Venda inv venda-té-línies: ...
```

```
context Venda::venda(id: String, s: Set(cP: String + q: Natural))
post nova-venda: crea una nova Venda v amb id
post noves-línies: per a cada element de s,
            crea una nova LíniaVenda associada amb v
```

post venda-té-clau, venda-té-línies

Per assegurar que es compleix l'invariant de Venda, l'operació:

- pot establir com a precondició que no hi ha cap Venda amb id i que 101 > s.nb() > 0
- pot declarar una excepció (o dues) per notificar la violació d'alguna d'aquestes condicions

En altres paraules, les operacions seran responsables de preservar l'invariant de les classes

Bibliografia

- *Applying UML and Patterns*
  C. Larman
  Prentice Hall, 2005 (Tercera edició), caps. 16, 17 i 32

- http://www.uml.org/#UML2.0

- *Object-Oriented Software Construction*
  B. Meyer
  Prentice Hall, 1997, cap. 3

- *Agile Software Development: Principles, Patterns and Practices*
  R.C. Martin
  Prentice Hall, 2003, caps. 7 i 9