

Tema 3: Disseny

Anna Puig

Enginyeria Informàtica

Facultat de Matemàtiques i Informàtica,

Universitat de Barcelona

Curs 2019/2020



UNIVERSITAT DE
BARCELONA

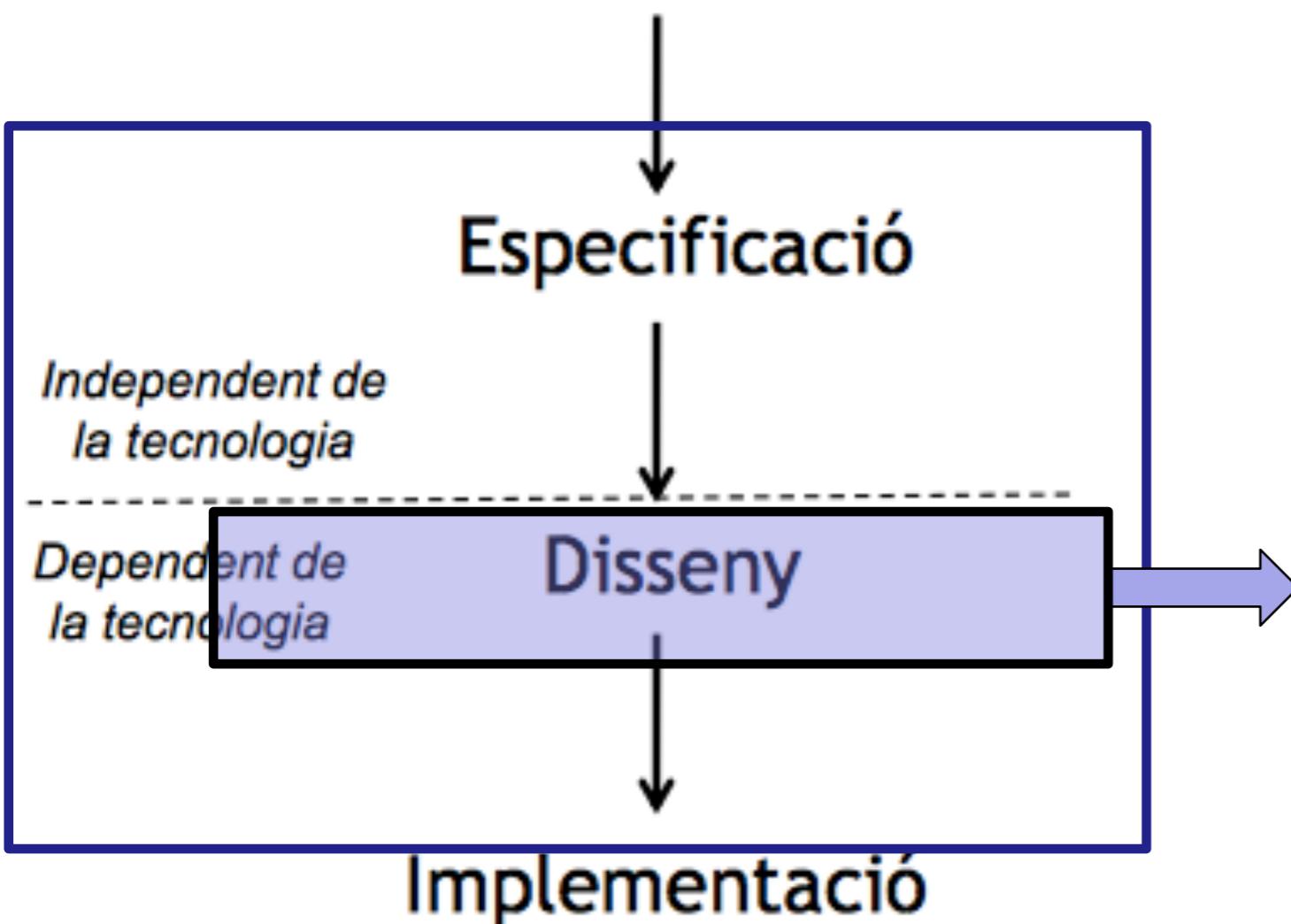
Temari

1	Introducció al procés de desenvolupament del software		
2	Anàlisi de requisits i especificació		
3	Disseny	3.1	Introducció
4	Del disseny a la implementació	3.2	Principis de Disseny: S.O.L.I.D.
5	Ús de frameworks de testing	3.3	Patrons arquitectònics
		3.4	Patrons de disseny

3.1. Introducció

Procés sistemàtic (Tema 3):

Anàlisi de requisits



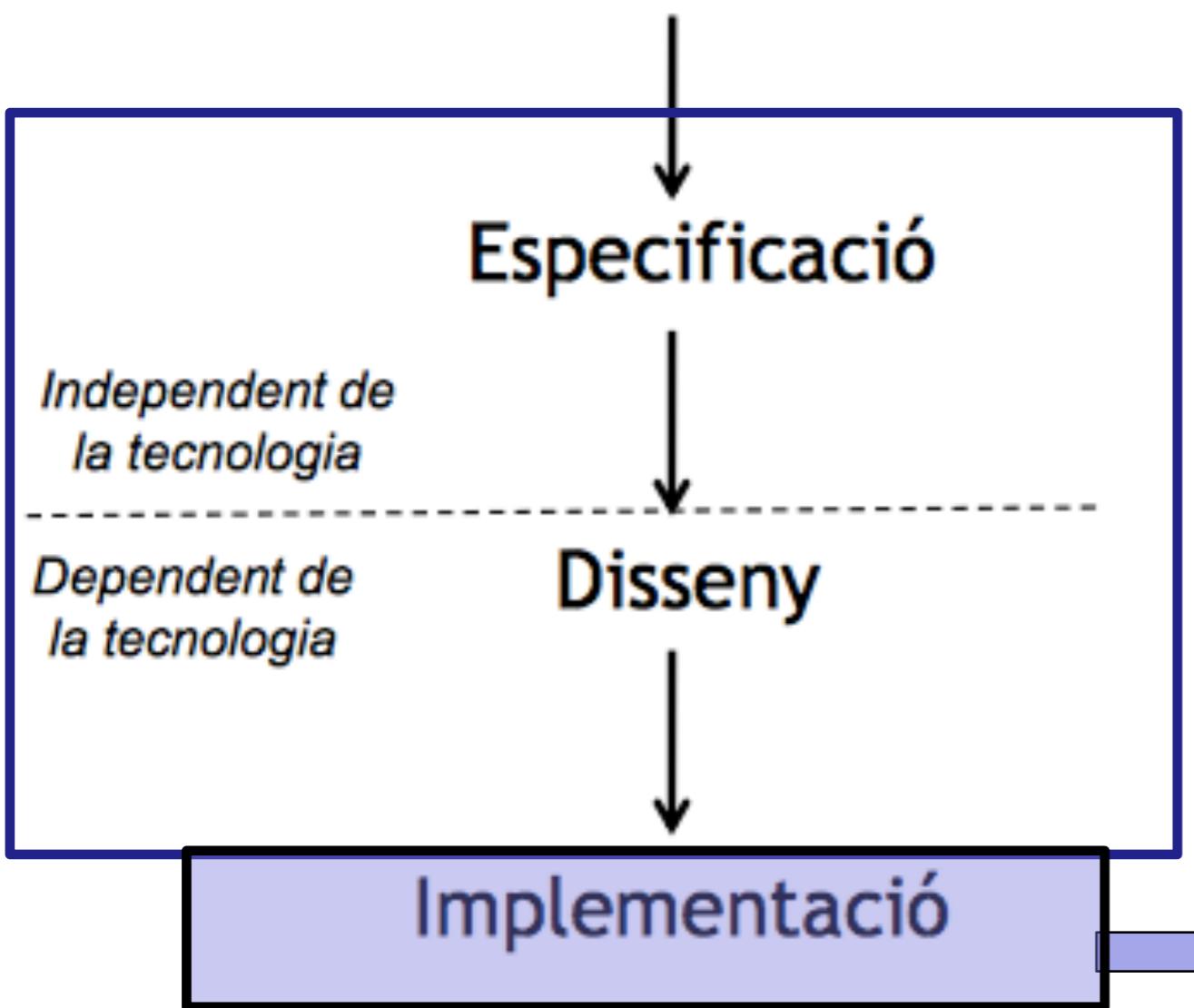
Com ho fa el sistema?

- Tests d'acceptació
- Descripció de l'arquitectura dels subsistemes i components del sistema software
- Diagrama de classes
- Diagrames d'interacció

3.1. Introducció

Procés sistemàtic (Laboratori):

Anàlisi de requisits



Generació de Codi:

- Tests unitaris
- Codi en llenguatge concret de les classes

3.1. Introducció

Per a què es vol un **bon** disseny de software?

- Per manegar de forma **fiable** la **complexitat** del problema
- Per a **desenvolupar ràpid** i lliurar-lo a temps
- Per poder incloure **canvis** fàcilment

- Preservar els principis de disseny
- Adaptació de solucions genèriques a problemes coneguts de disseny (**patrons**)

3.1. Introducció

“Each **pattern** (patró) describes a problem which occurs over and over again in our environment, and then describes the **core of the solution** to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

Christopher Alexander, arquitecte (1977)

Nom del patró

Problema

- Descripció del problema a resoldre
- Enumeració de les forces a equilibrar

Solució

- **Aspecte estàtic**: impacte en el diagrama de classes del disseny
- **Aspecte dinàmic**: establiment del comportament de les noves operacions

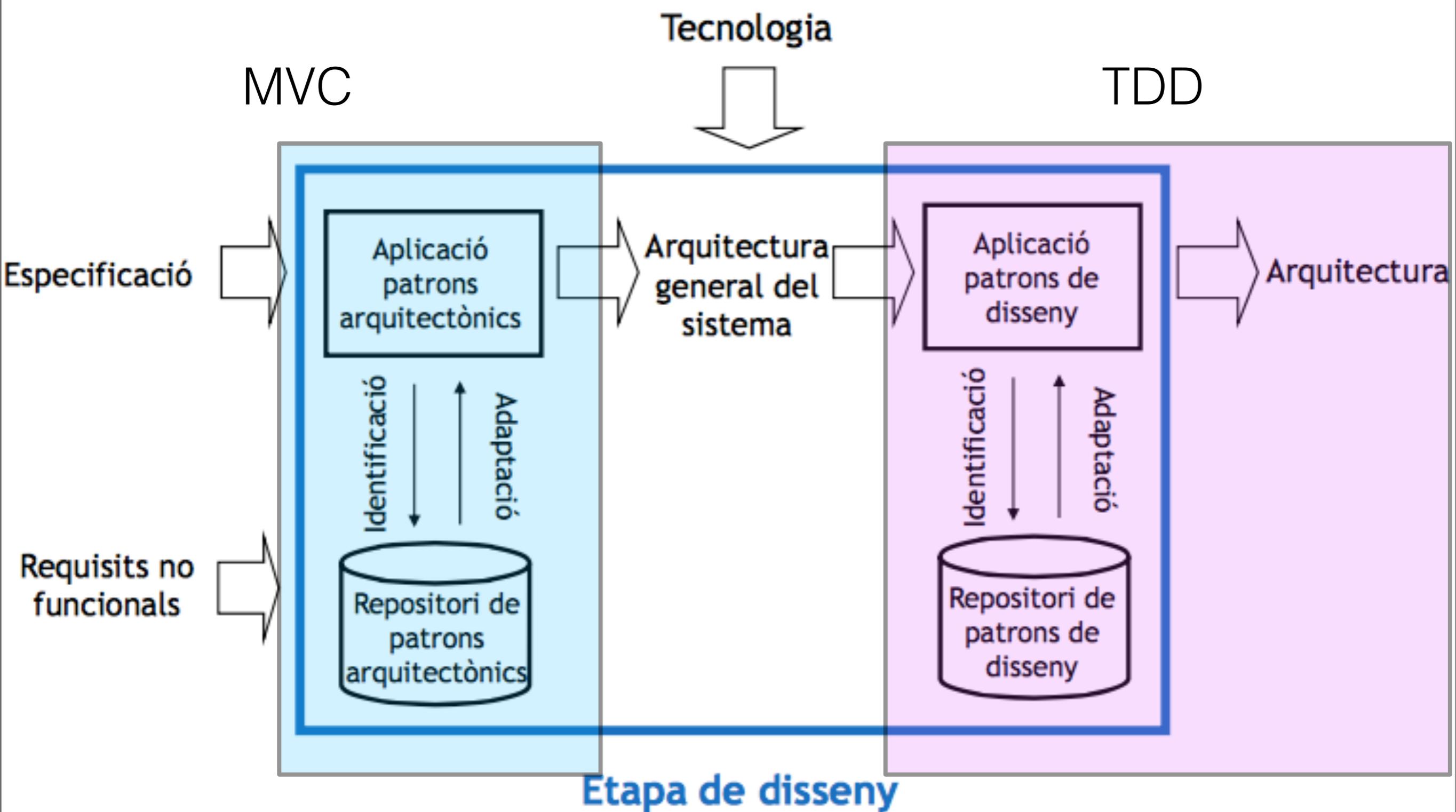
Conseqüències

Avantatges i desavantatges

3.1. Introducció

- **Patrons d'arquitectura:** usats en el disseny a gran escala i de gra guixut.
 - *Per exemple:* patró de **Capes**
- **Patrons de disseny:** utilitzats en el disseny d'objectes i frameworks de petita i mitjana escala. (micro-arquitectura)
 - *Per exemple,* patró **Façana** (*Facade*) per connectar les capes o el patró **Estratègia** per permetre algorismes connectables
- **Patrons d'estils:** solucions de baix nivell orientades a la implementació o al llenguatge.
 - *Per exemple,* patró **Singleton** per fer una única instància d'una classe.

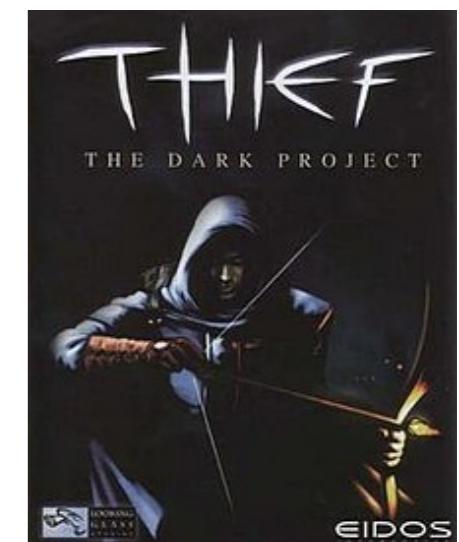
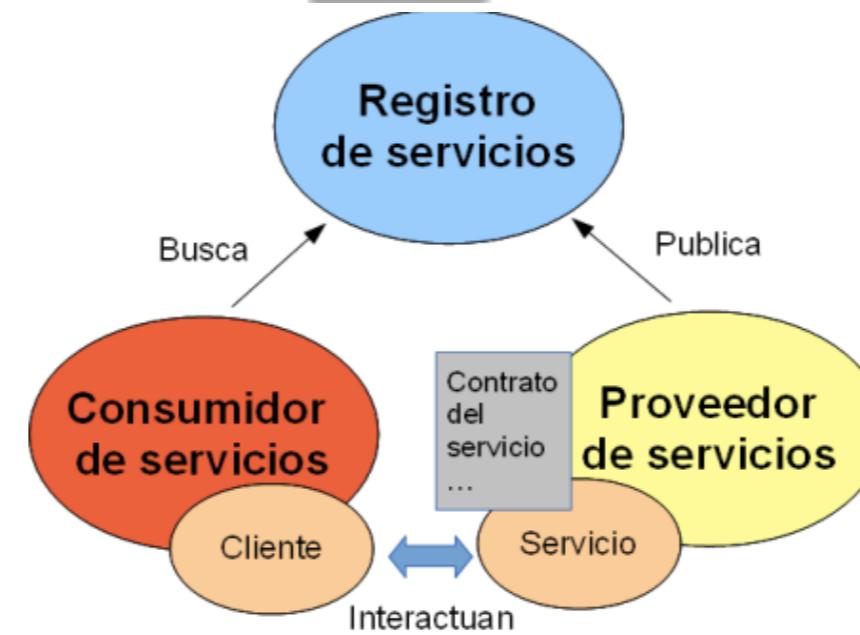
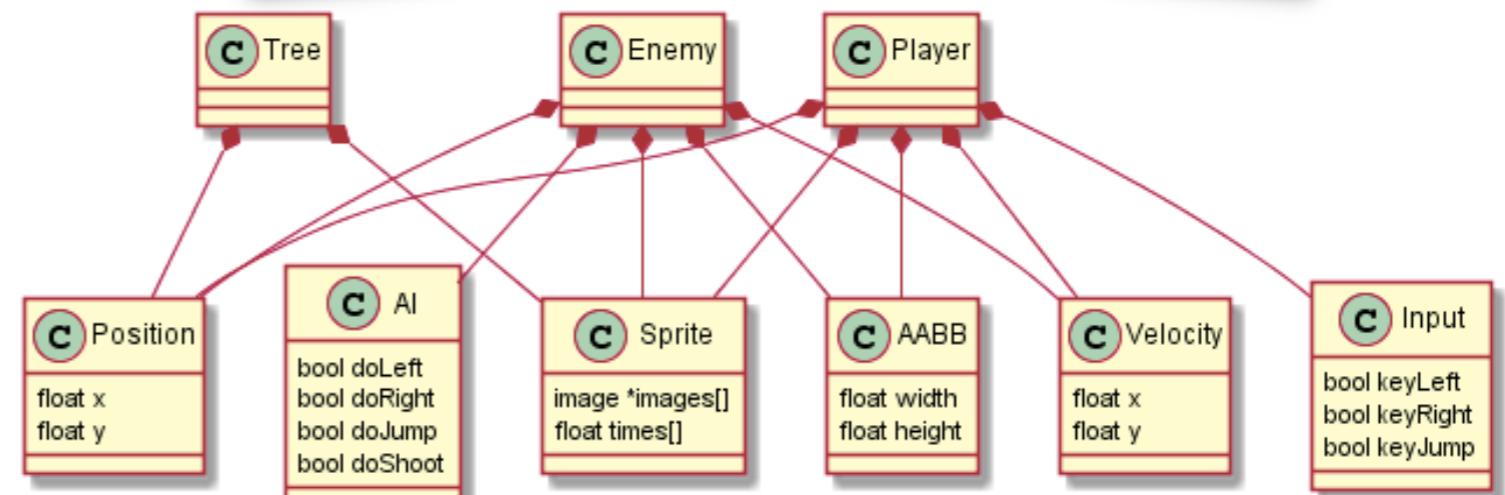
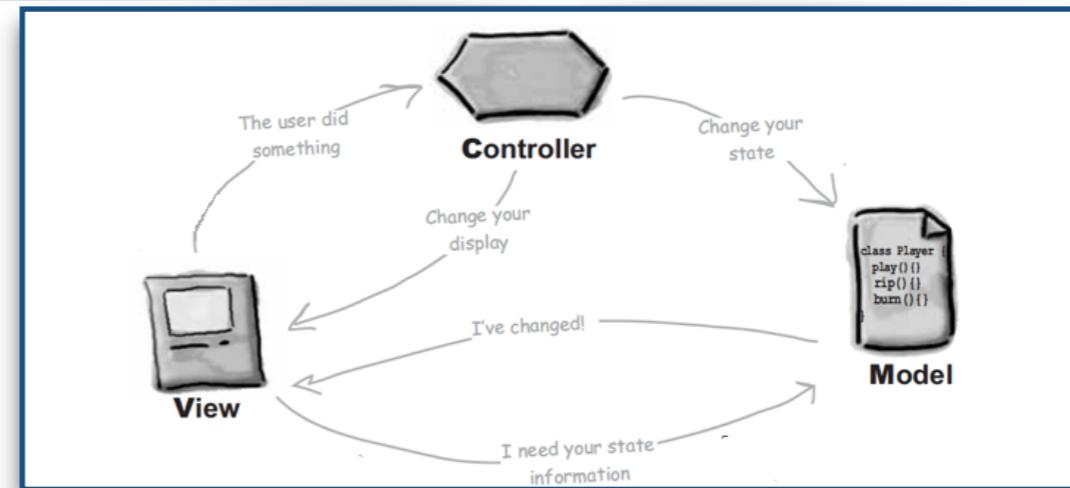
3.1. Introducció



3.1. Introducció

Patrons arquitectònics (entre d'altres):

- Model-Vista-Controller
- Entity-Component System
- Service Oriented Architecture (SOA)



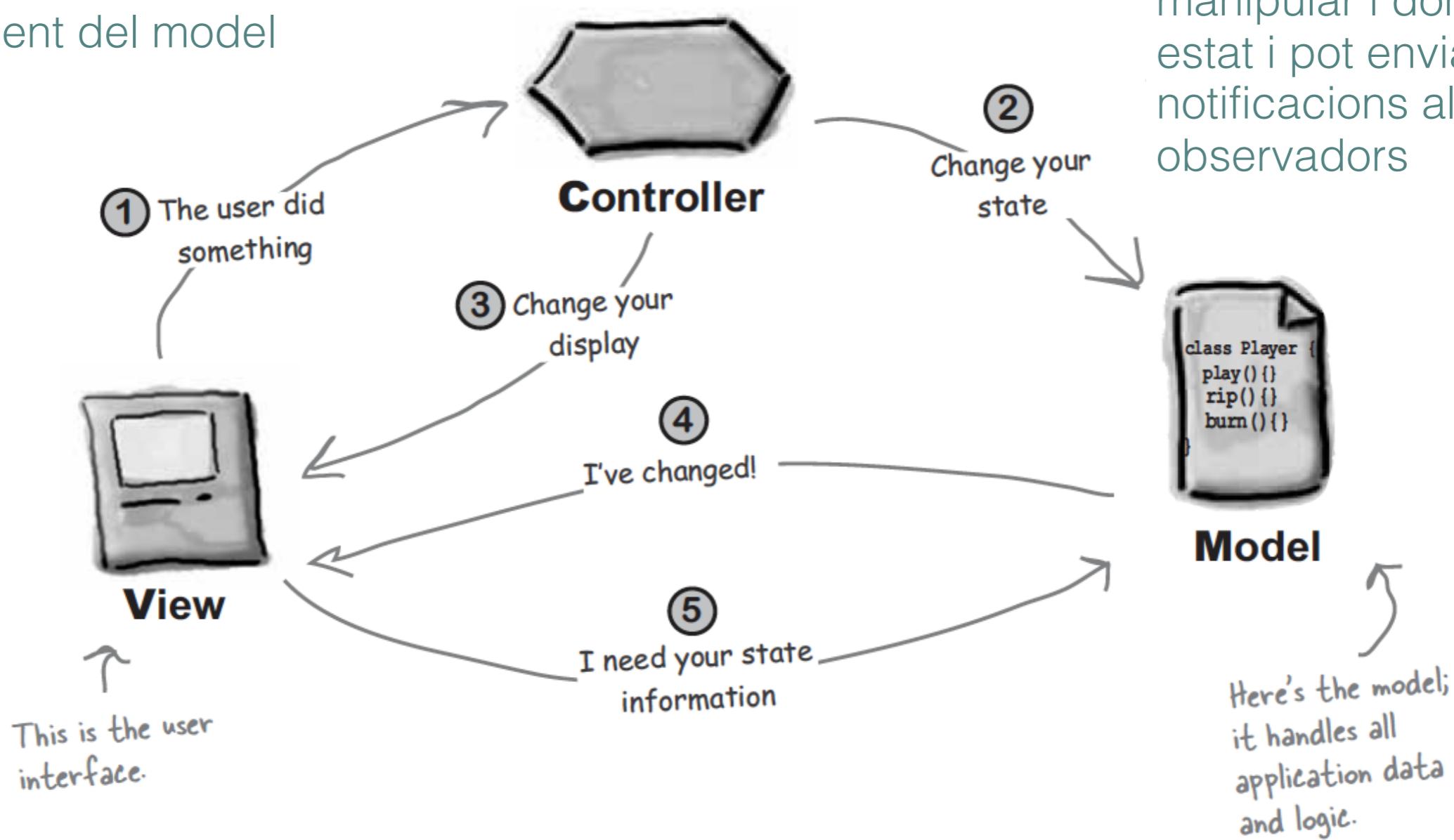
3.1. Introducció

VISTA:

Dóna la presentació del model. La vista normalment mostra l'estat de les dades i el seu valor directament del model

CONTROLADOR:

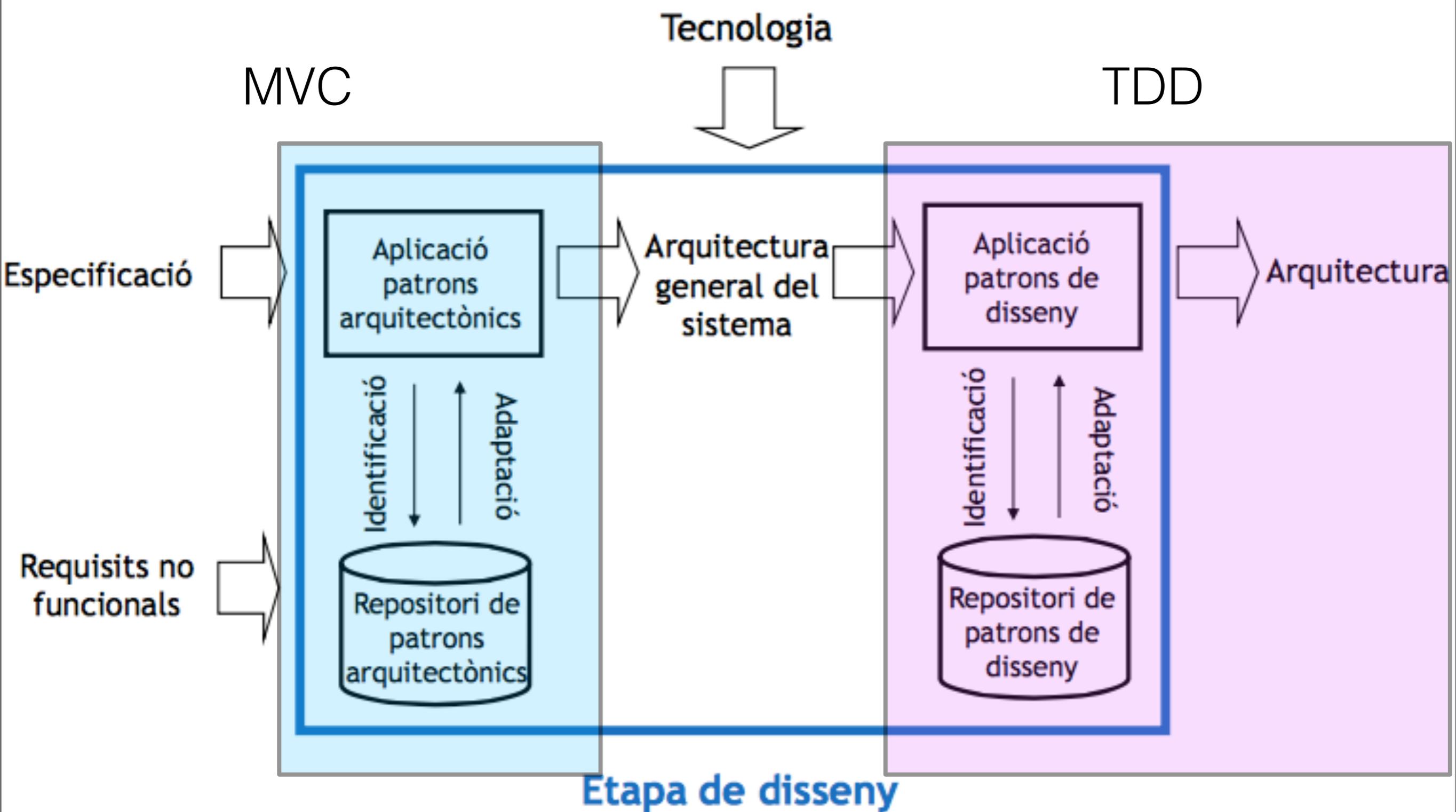
Agafa l'entrada de l'usuari i li dóna el què significa al model



MODEL:

El model guarda totes les dates, l'estat i la lògica de l'aplicació. Dóna una interfície per manipular i donar el seu estat i pot enviar notificacions als observadors

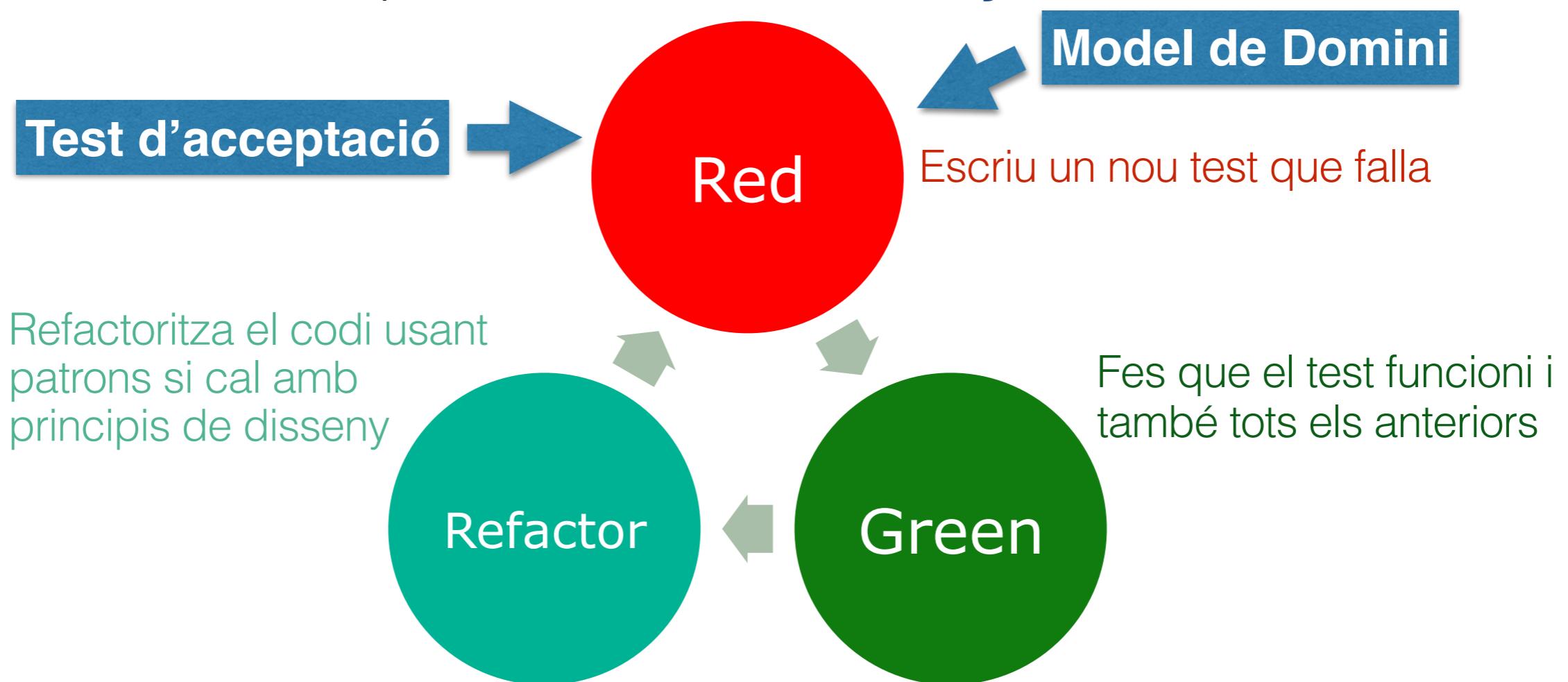
3.1. Introducció



3.1. Introducció

TDD (Test Driven Development): Basat en **dues** senzilles regles:

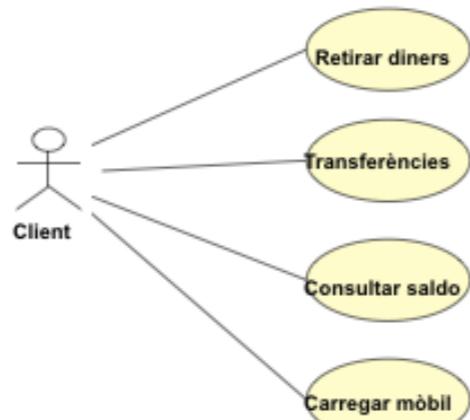
1. Escriu el nou codi només si el test automàtic ha fallat (**Disseny i Implementació**)
2. Elimina la duplicació de codi. (**Disseny**)



3.1. Introducció

Escenari simple de ProcesarVenta para el pago en efectivo

1. El Cliente llega al terminal PDV.
2. El Cajero inicia una nueva venta.
3. El Cajero inserta el identificador del artículo.
4. El Sistema registra la línea de venta y presenta la descripción del artículo, precio y suma parcial.
- El Cajero repite los pasos 3 y 4 hasta que se indique.
5. El Sistema muestra el total con los impuestos calculados.
6. El Cajero le dice al Cliente el total, y pide que le pague.
7. El Cliente paga y el Sistema gestiona el pago.
- ...



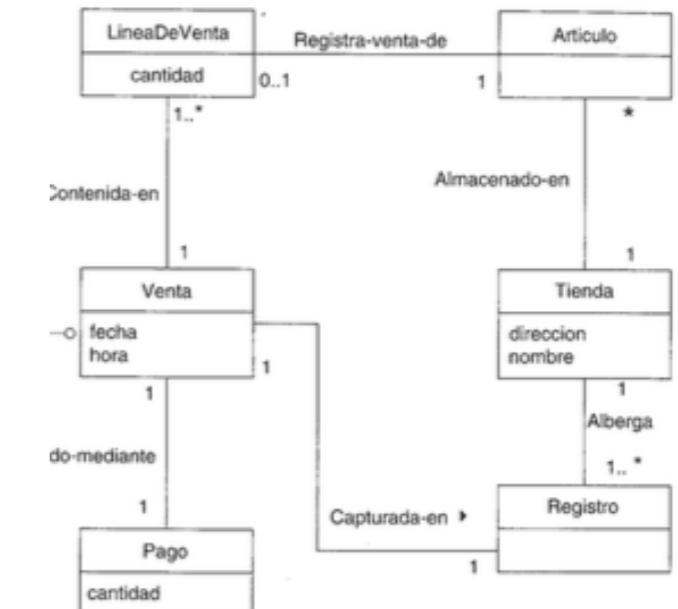
Com a [rol d'usuari]
vull [objectiu]
per què així [raó]

En cas que [context]
quan [event]
el sistema [resultat]

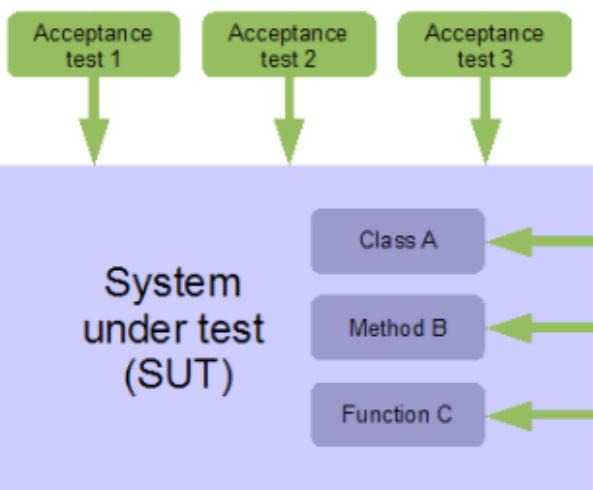
Casos d'ús

DCU

User Stories+Criteris
d'acceptació



Model de Domini



Tests d'acceptació
+ Tests unitaris

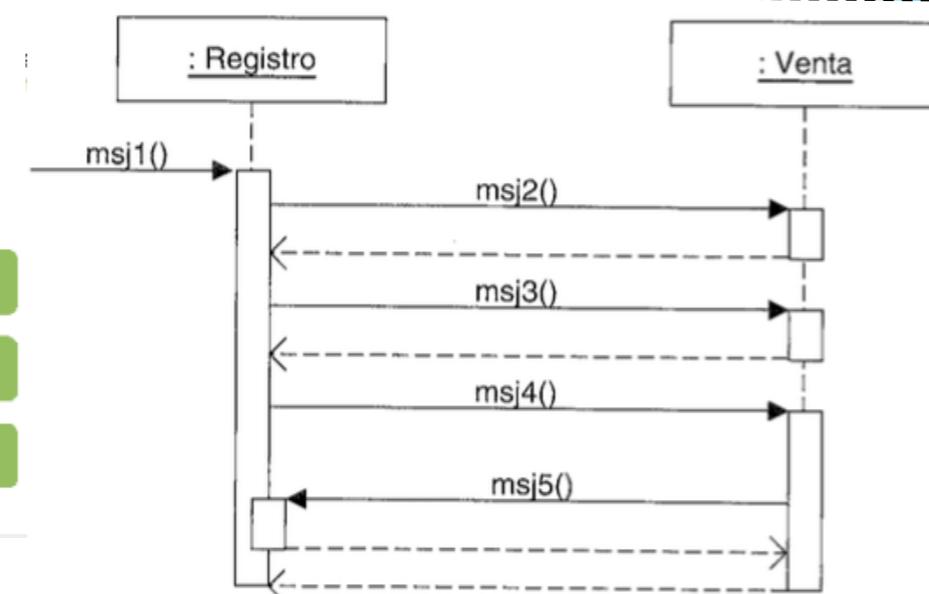


Diagrama de Seqüència (DS)

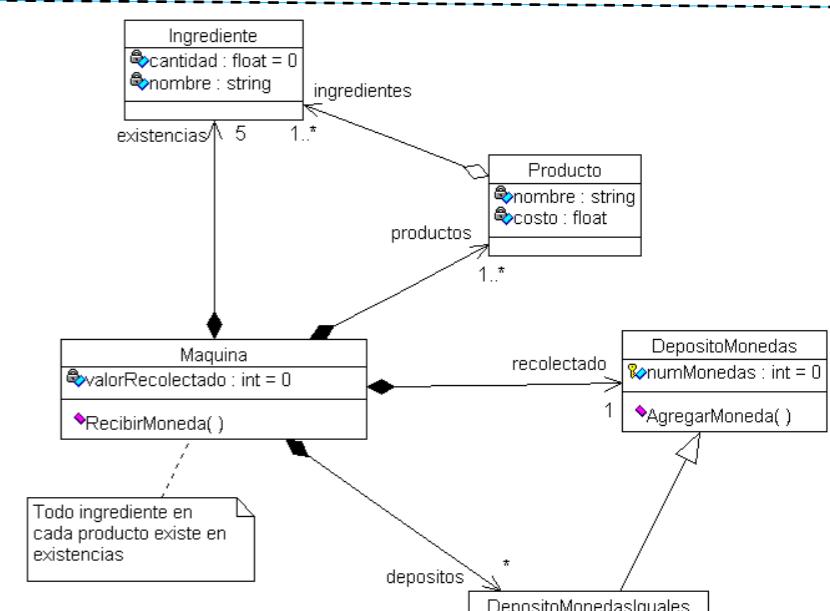


Diagrama de Classes de Disseny (DCD)

3.1. Introducció

- La creació del **model de disseny** és la construcció d'una solució basada en el paradigma orientat a objectes.
 - **Esquema de classes de les dades:** (**model estàtic**)
Construcció els **diagrames de classes** que resumeixen la definició de les classes de software que s'estan implementant.
 - **Esquema de comportament:** (**model dinàmic**)
Construcció de **diagrames d'interacció** que representen com els objectes col·laboren per satisfer els requisits. (**Diagrames de Seqüència**)
- Són eines de disseny, que ajudaran a avaluar el disseny i *aprendre a dissenyar amb patrons*

3.1. Introducció

Un **diagrama d'interacció** consisteix en un conjunt d'objectes i les seves relacions, incloent-hi els missatges que es poden enviar entre ells

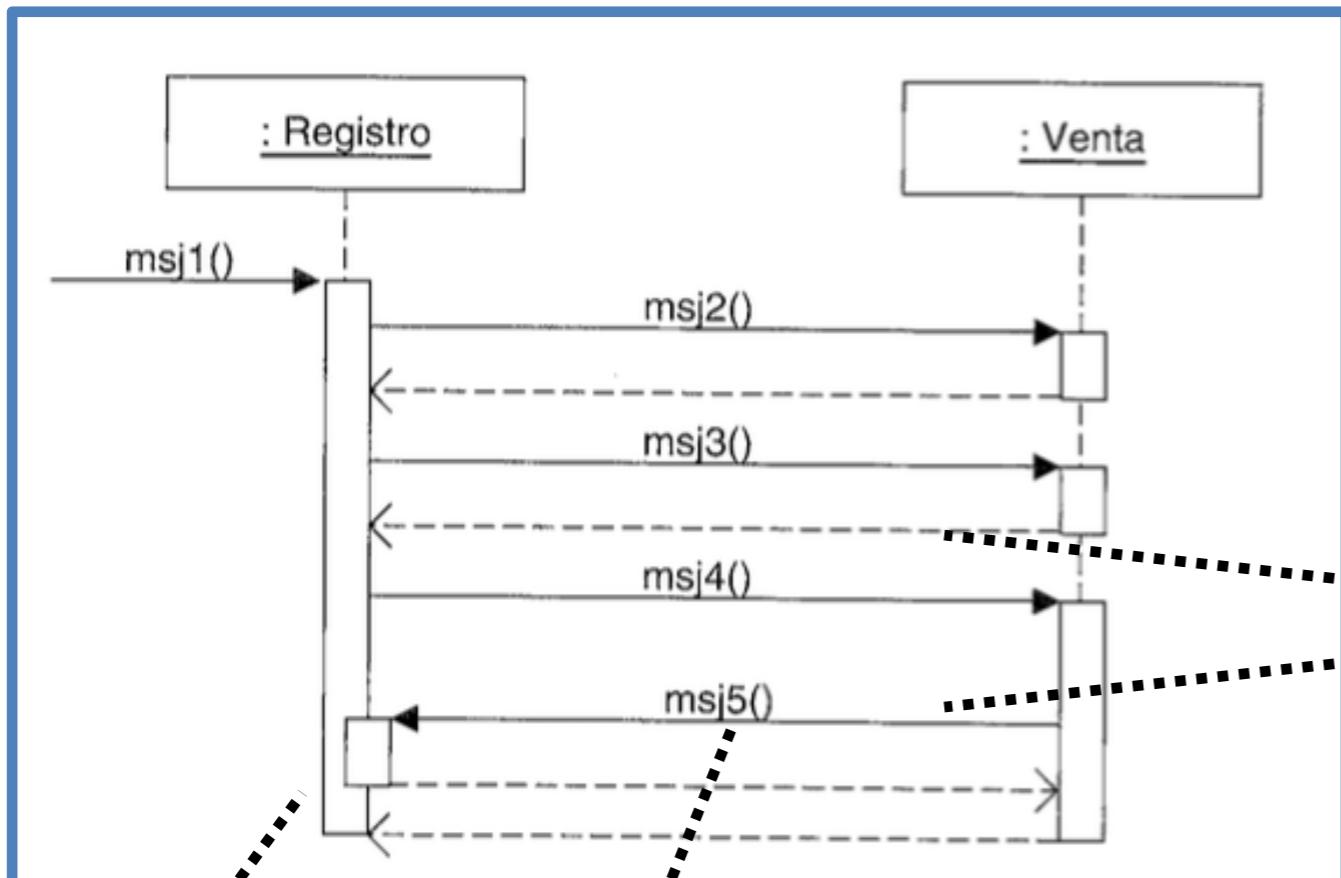
[modelar el **comportament** d'un sistema]

Aquests diagrames es desenvolupa en paral·lel amb diagrama de classes

1. **Diagrames de seqüència:** destaquen l'ordre temporal dels missatges
2. **Diagrames de col·laboració:** destaquen l'organització estructural dels objectes

Ambdós diagrames (seqüència i col·laboració) són semànticament equivalents. Es pot passar d'un a l'altre sense pèrdua d'informació

Tipus de diagrames d'interacció

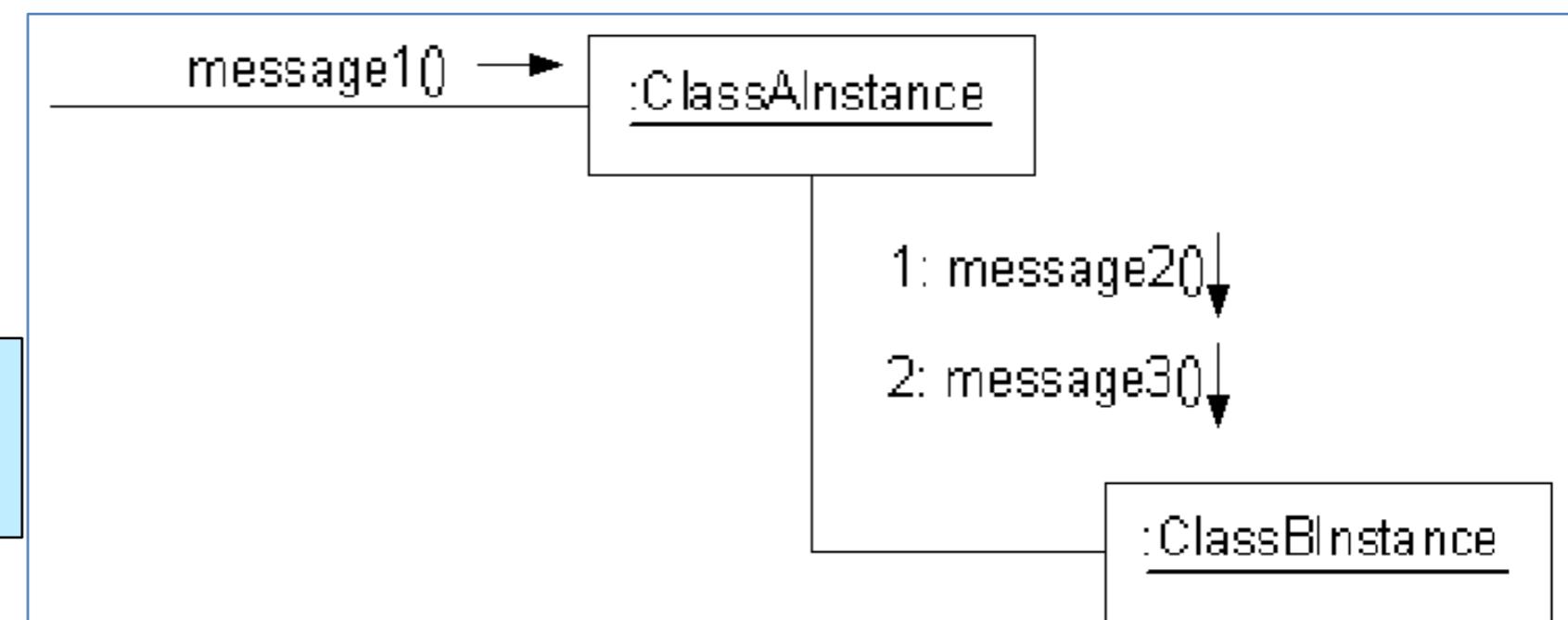


(a) Diagrama de seqüència

Bloc activació

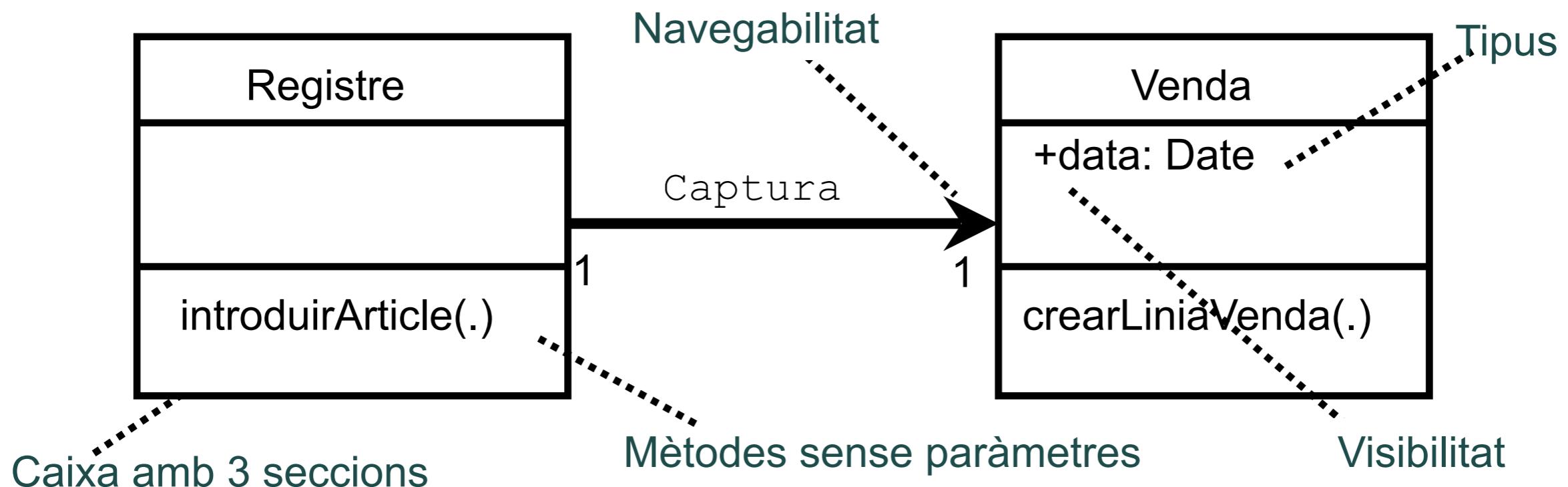
Missatge

(b) Diagrama de col·laboració

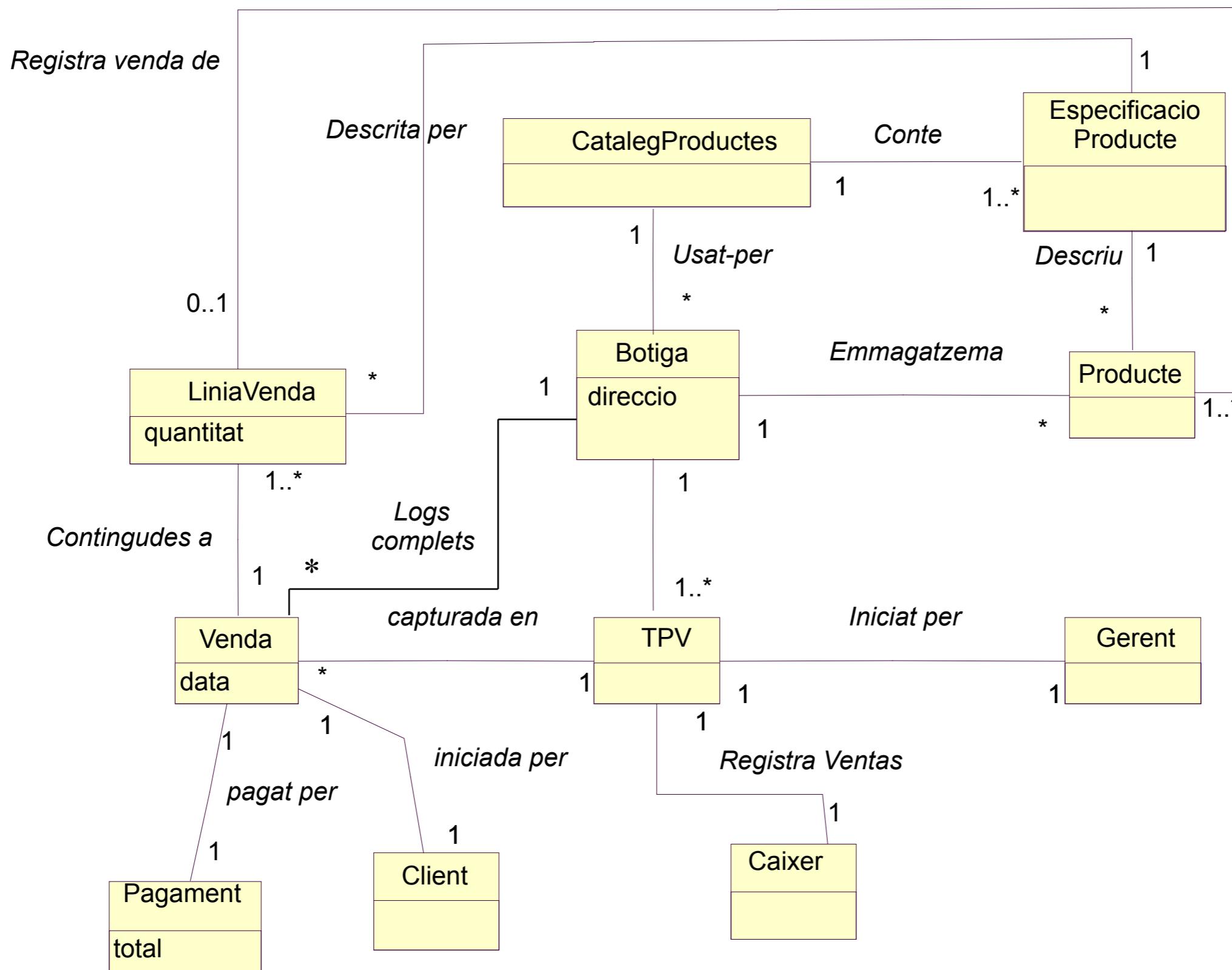


3.1. Introducció

Un **Diagrama de Classes de Disseny** (DCD) il·lustra les especificacions per classes software i interfícies en una aplicació



Exemple model domini TPV



Criteri d'acceptació: fesPagament

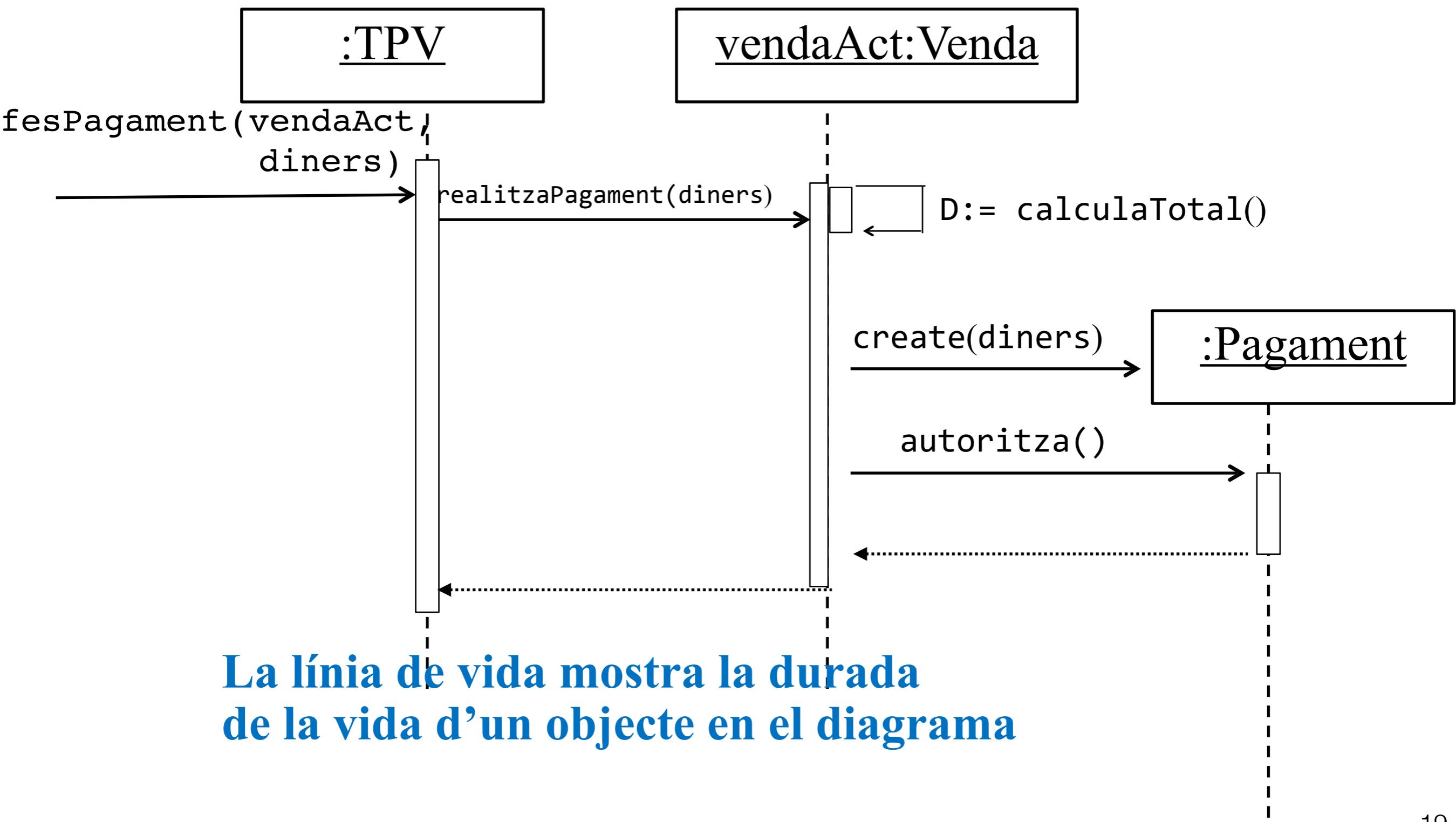
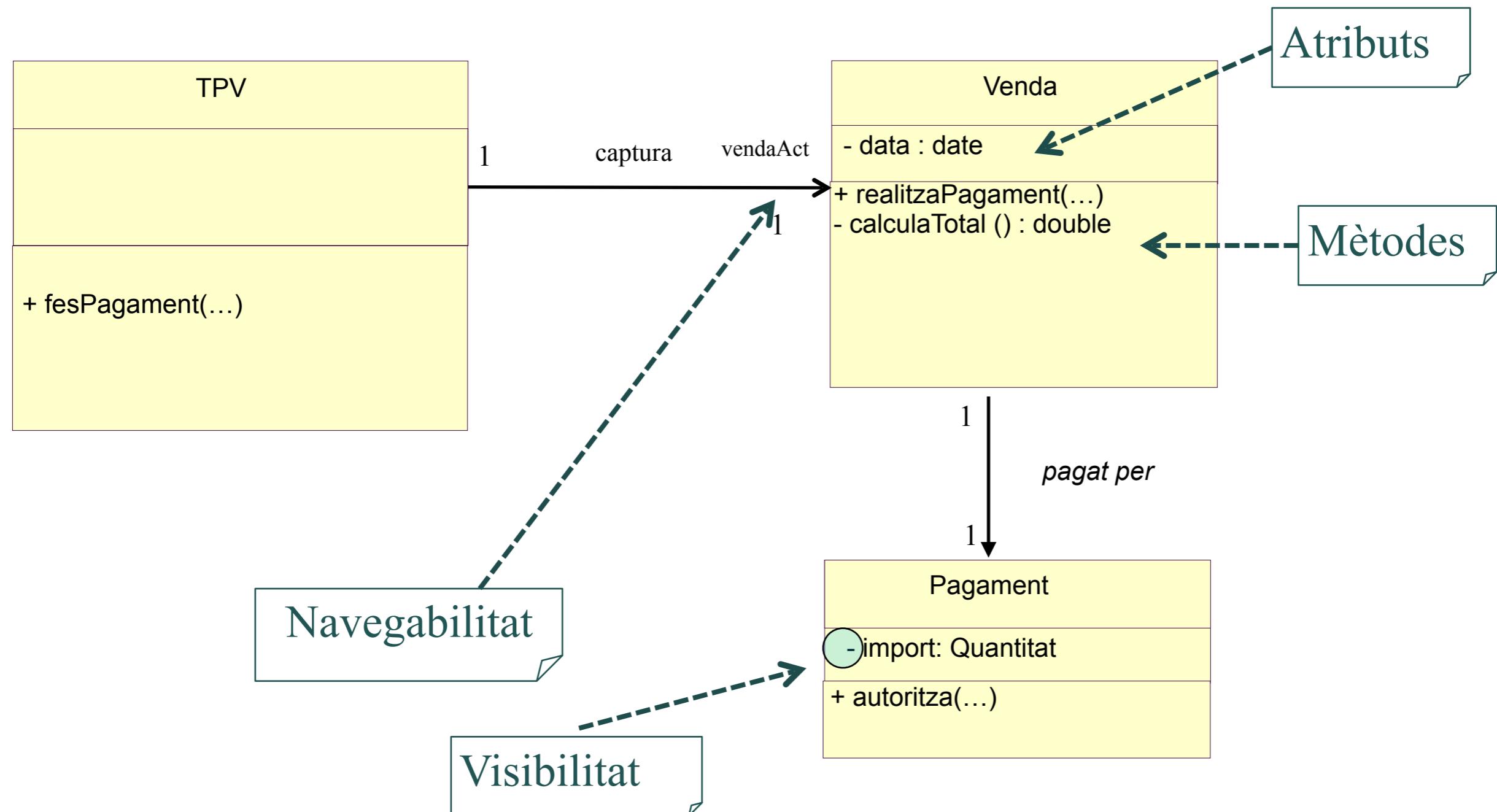


Diagrama de classes

Classes, Associacions, navegabilitat i visibilitat en una part del Terminal Punt de Venda



3.1. Introducció

Aspectes que denoten un disseny “dolent” (*code smell*)

- **Rigidesa**: l'impacte d'un canvi en el software és impredecible (cada canvi produeix una cascada de canvis en moltes altres classes o el codi és tan complicat que costa entendre'l)
- **Inmobilitat**: No es pot reutilitzar codi o parts de codi
- **Fragilitat**: A cada canvi, el software es trenca en llocs on no hi han relacions conceptuais
- **Viscositat**: Impossibilitat de canviar el codi sense canviar el disseny. Provoca que fer un pedaç addicional al codi és més fàcil que no pas canviar tot el disseny



Dependències entre classes

3.1. Introducció

Característiques que permeten avaluar si un disseny és “bo”:

- **Acoblament:**

mesura del grau de connexió, coneixement i dependència d'una classe respecte d'altres classes.

Acoblament BAIX

Quan més acoblament té una classe:

- més difícil resulta comprendre-la aïlladament.
- més difícil de reutilitzar-la, perquè requereix la presència de les altres classes.

- **Cohesió:**

mesura del grau de relació i de concentració de les diverses responsabilitats d'una classe (atributs, associacions, mètodes,...)

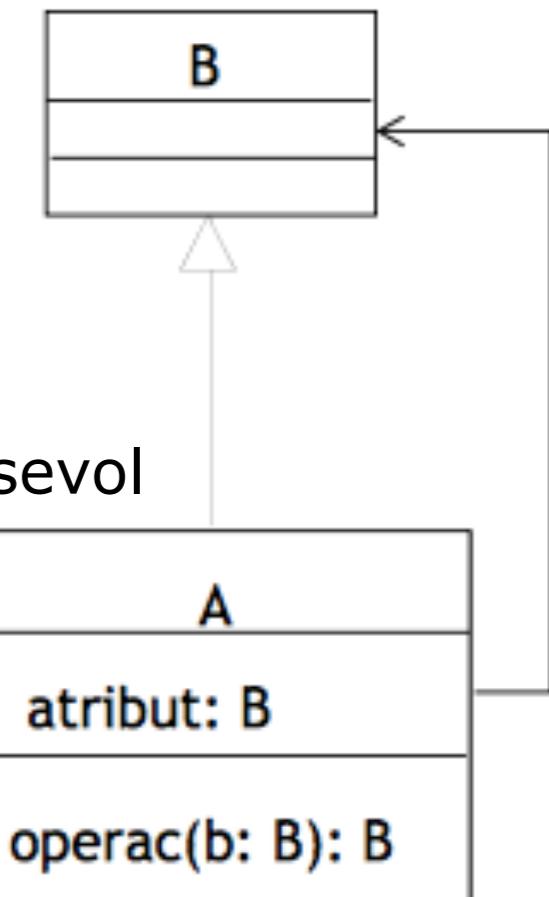
ALTA Cohesió

Una classe amb alta cohesió **no** té molts mètodes, que estan molt relacionats entre ells i **no** realitza molt de treball. Col·labora amb altres classes.

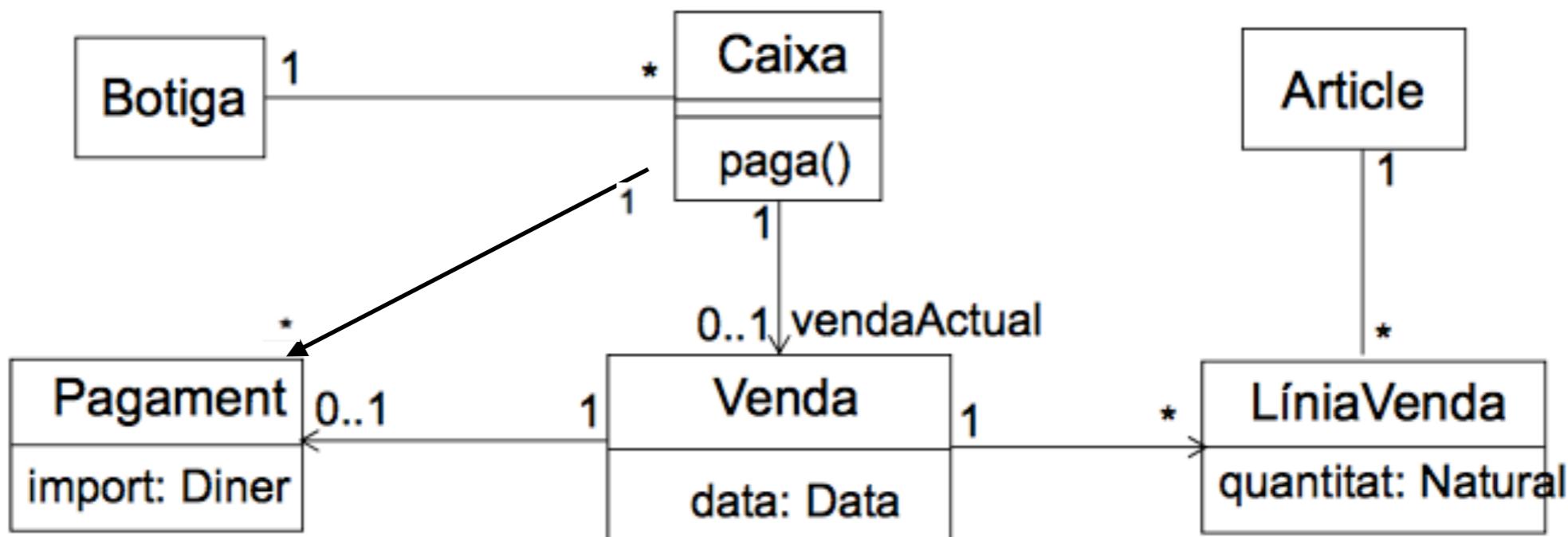


Quan hi ha acoblament?

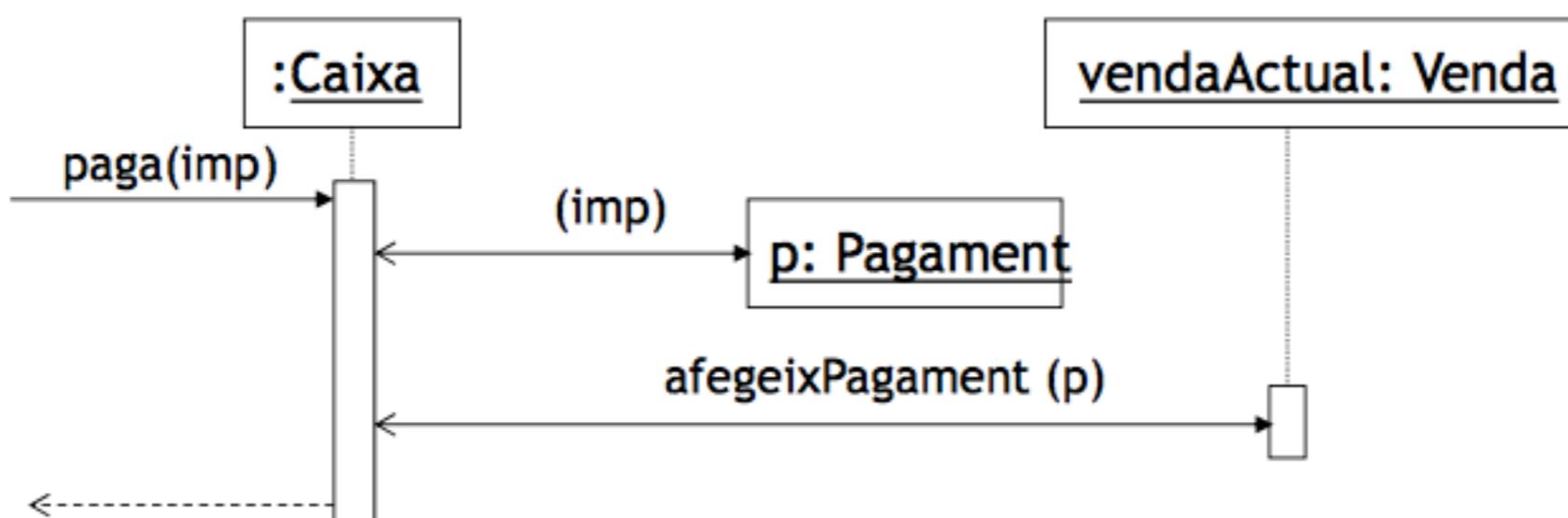
- Tipus d'acoblament entre les classes A i B:
 - A té un atribut que es refereix a una instància de B
 - Un objecte A invoca mètodes d'un objecte B
 - A té un mètode que referència a una instància de B de qualsevol forma (paràmetre o retorn)
 - A és una subclasse directa o indirecta de B
 - B és una interfície i A implementa B
- Convé que l'acoblament sigui **baix**:
 - Si hi ha un acoblament de A a B, un canvi en B pot implicar canviar A.
- Excepcions:
 - L'acoblament amb classes estables ben conegeudes o estables no acostuma a ser problema, sobretot si estan protegides de canvis



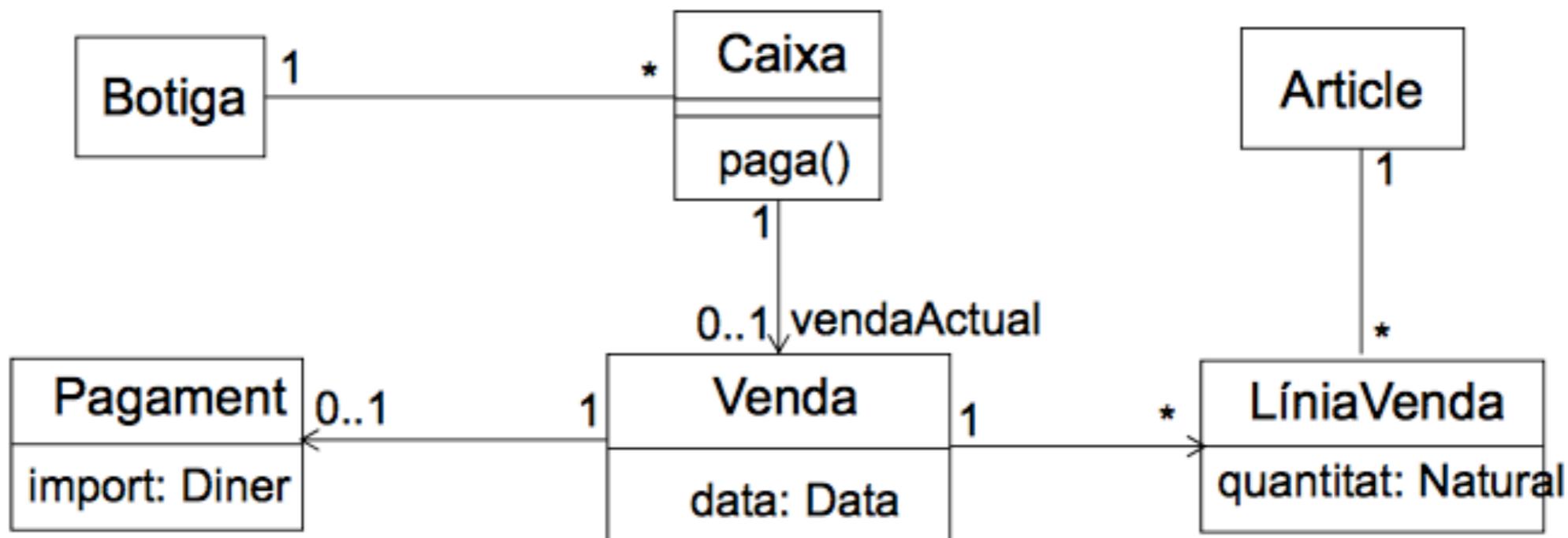
Acoblament alt



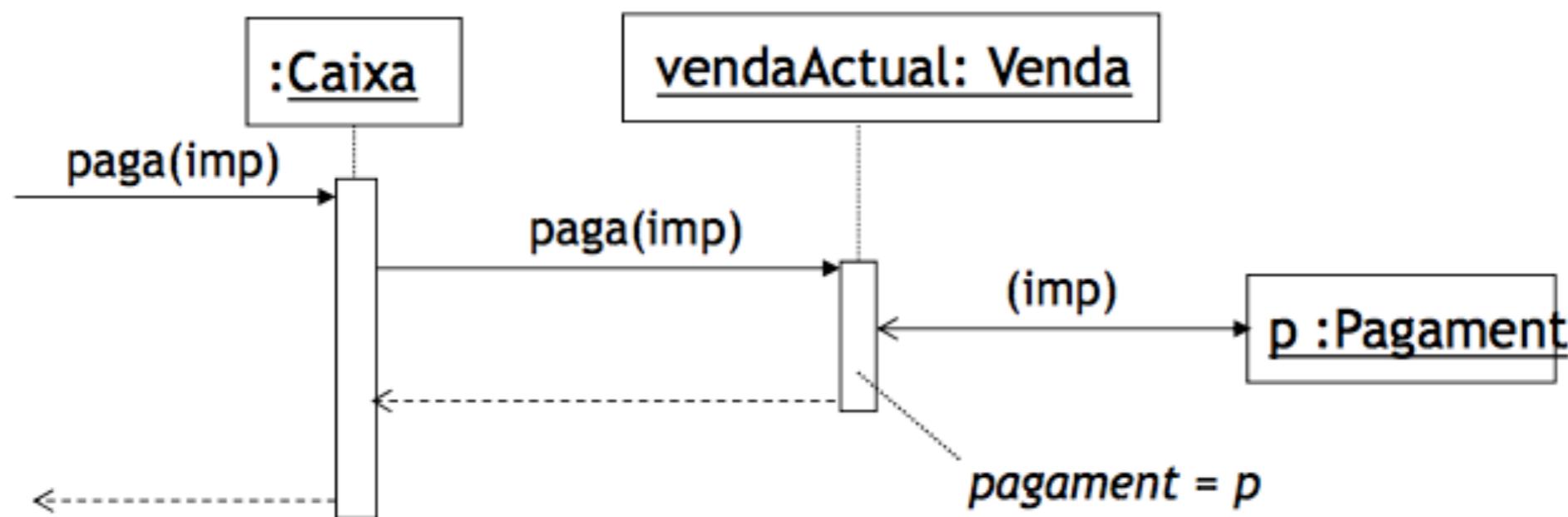
Alternativa 1:



Acoblament baix

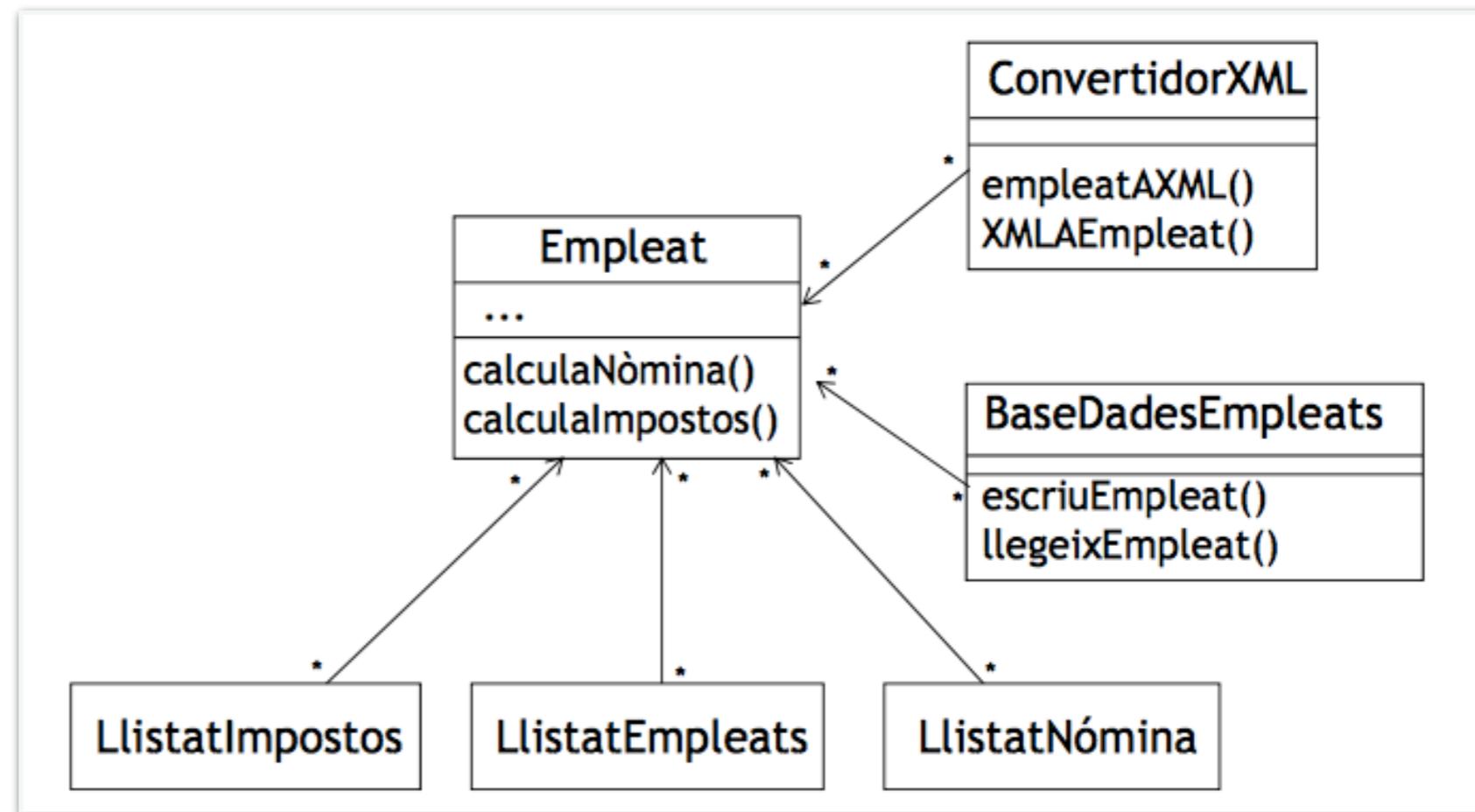


Alternativa 2:



Exemple cohesió

Empleat
...
calculaNòmina()
calculalImpostos()
escriuADisc()
llegeixDeDisc()
creaXML()
llegeixDeXML()
mostraEnLlistatNòmina()
mostraEnLlistatImpostos()
mostraEnLlistatEmpleats()



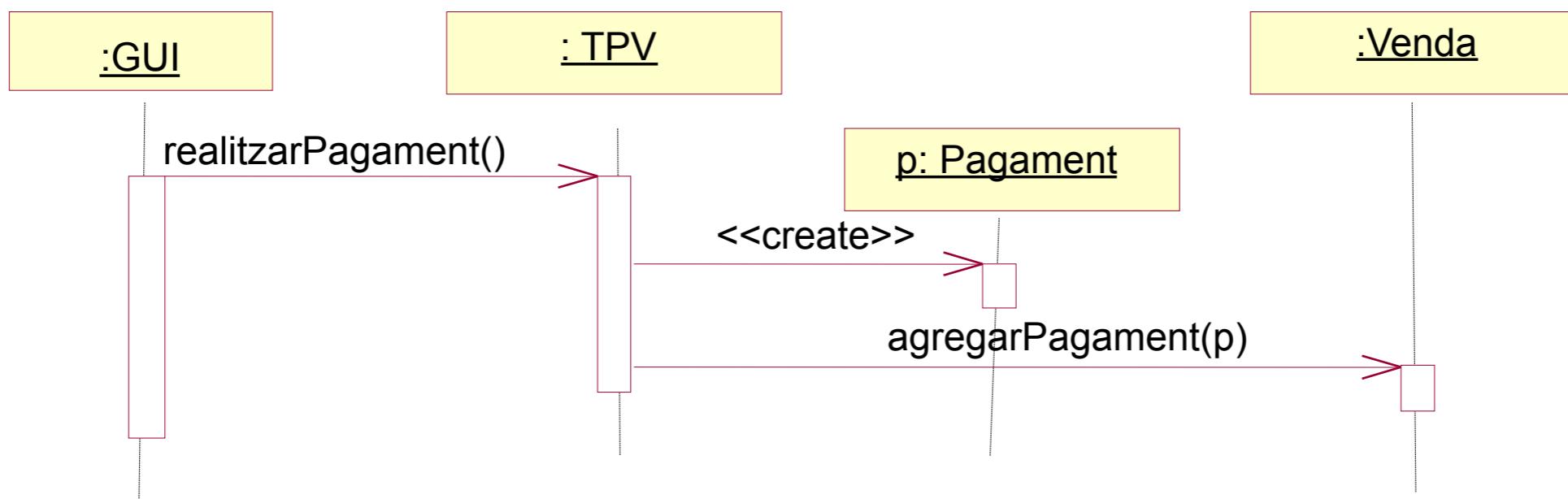
BAIXA Cohesió

ALTA Cohesió

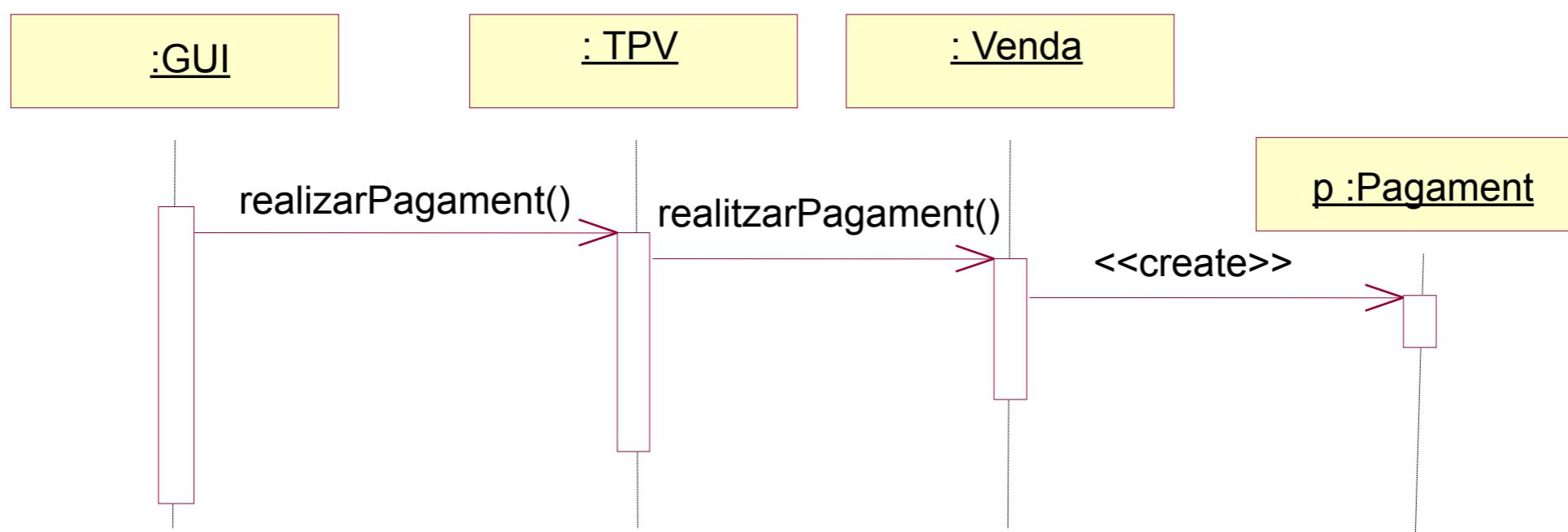
Exemple d'aplicació

- Quin d'aquests dissenys produeix una major cohesió en *TVP*?

(a)

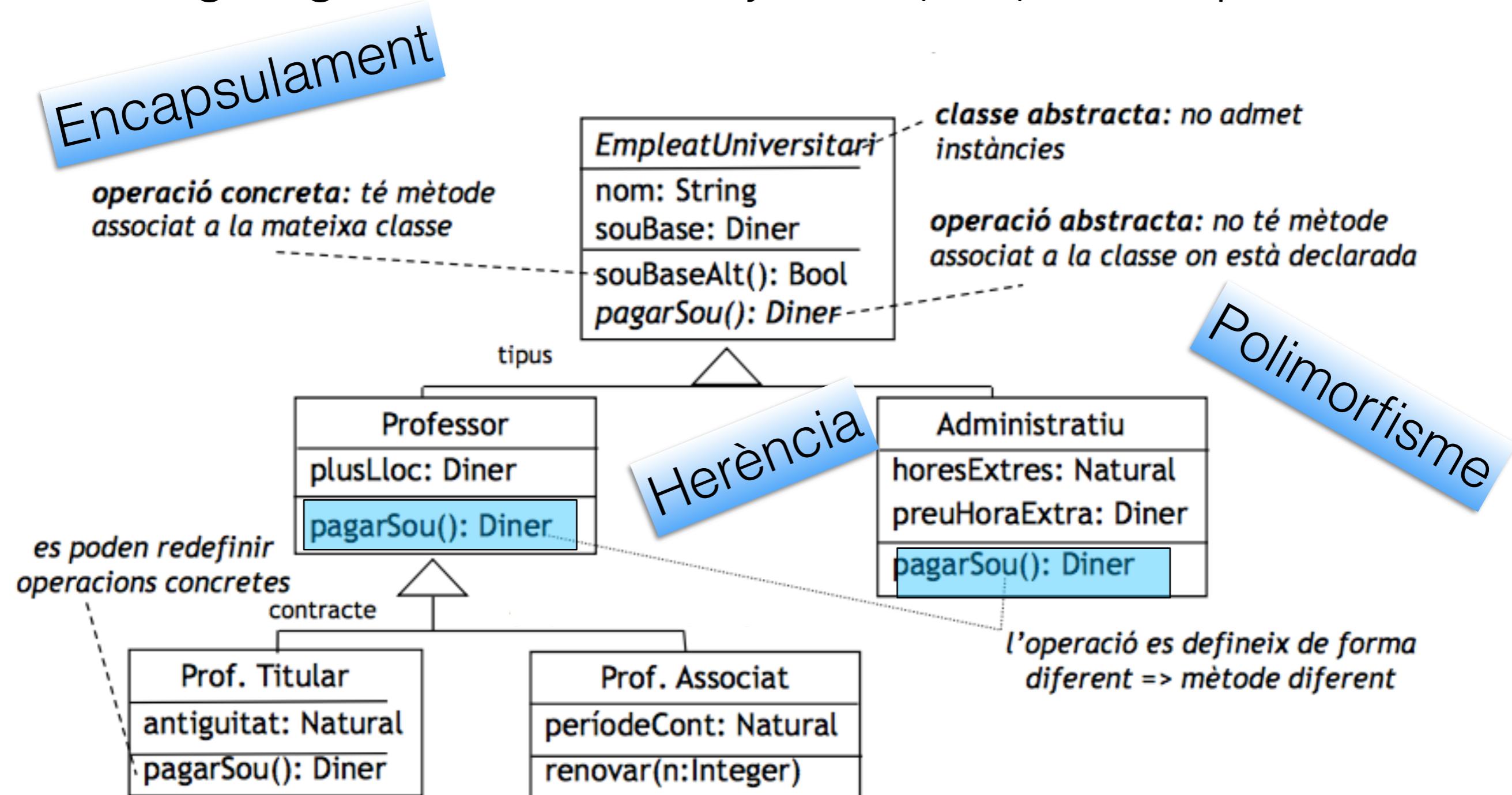


(b)



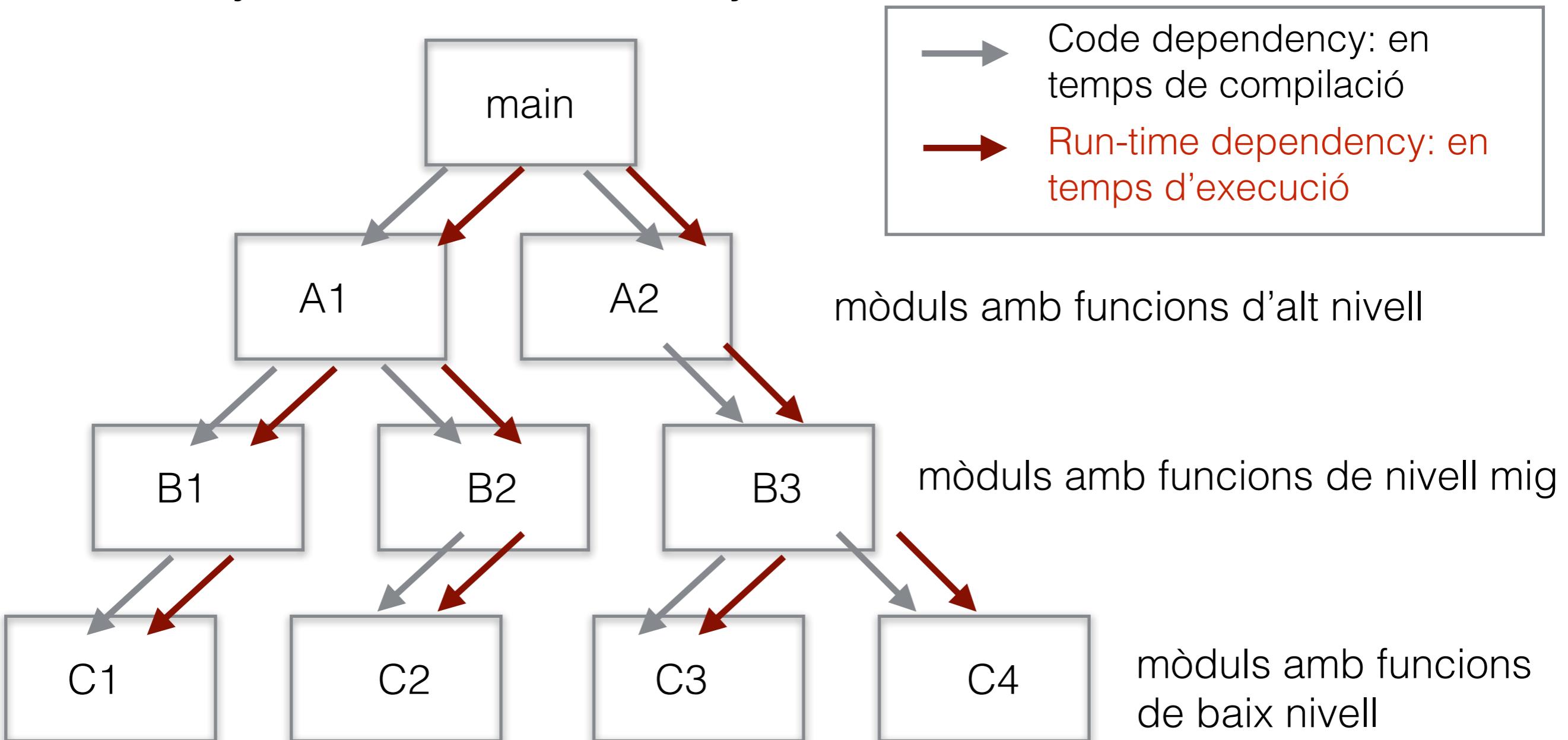
3.1. Introducció

Llenguatges Orientat a Objectes (OO): Què aporta?



3.1. Introducció

En dissenys no orientats a objectes:



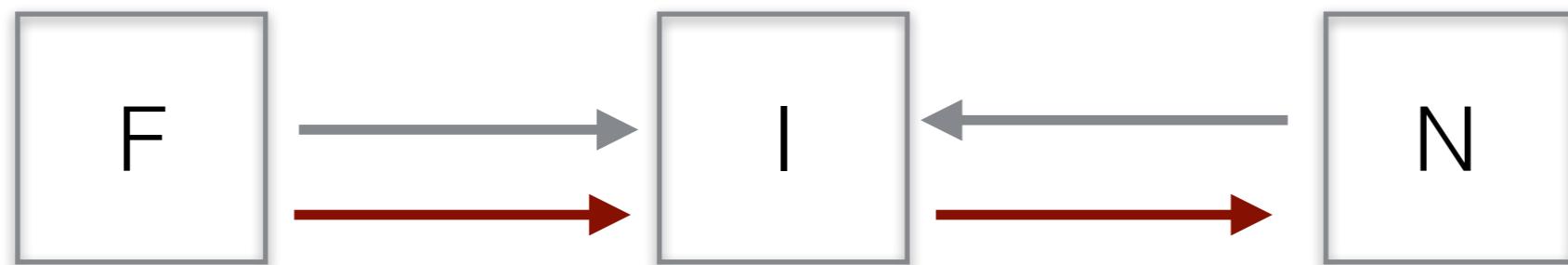
El sentit de les dependències de codi és el mateix que les dependències en execució

3.1. Introducció

Aportació del disseny Orientat a Objectes



→ Code dependency: en temps de compilació
→ Run-time dependency: en temps d'execució



→ L'arquitectura de'execució no restringeix l'arquitectura del codi

3.2. Principis de Disseny: S.O.L.I.D.

Principis de disseny [Robert C. Martin 98]:

- **S**: Single Responsibility Principle
- **O**: Open-Close Principle
- **L**: Liskov Substitution Principle
- **I**: Interface Segregation Principle
- **D**: Dependency Inversion Principle



[Article resum de R.C. Martin \(fins la pàgina 18\)](#)

Agile Software Development, Principles, Patterns, and Practices

3.2. Principis de Disseny: S.O.L.I.D.

Single Responsibility Principle:

"One class should have one and only one responsibility"

De Marco 79 and Page-Jones 88



- Una classe ha de tenir **una i només una responsabilitat**
- Responsabilitat s'entén com a raó que fa canviar la classe
- Una classe hauria de tenir només una raó per a ser canviada
- Això permet tenir **ALTA** cohesió i evitar classes **fràgils**

3.2. Principis de Disseny: S.O.L.I.D.

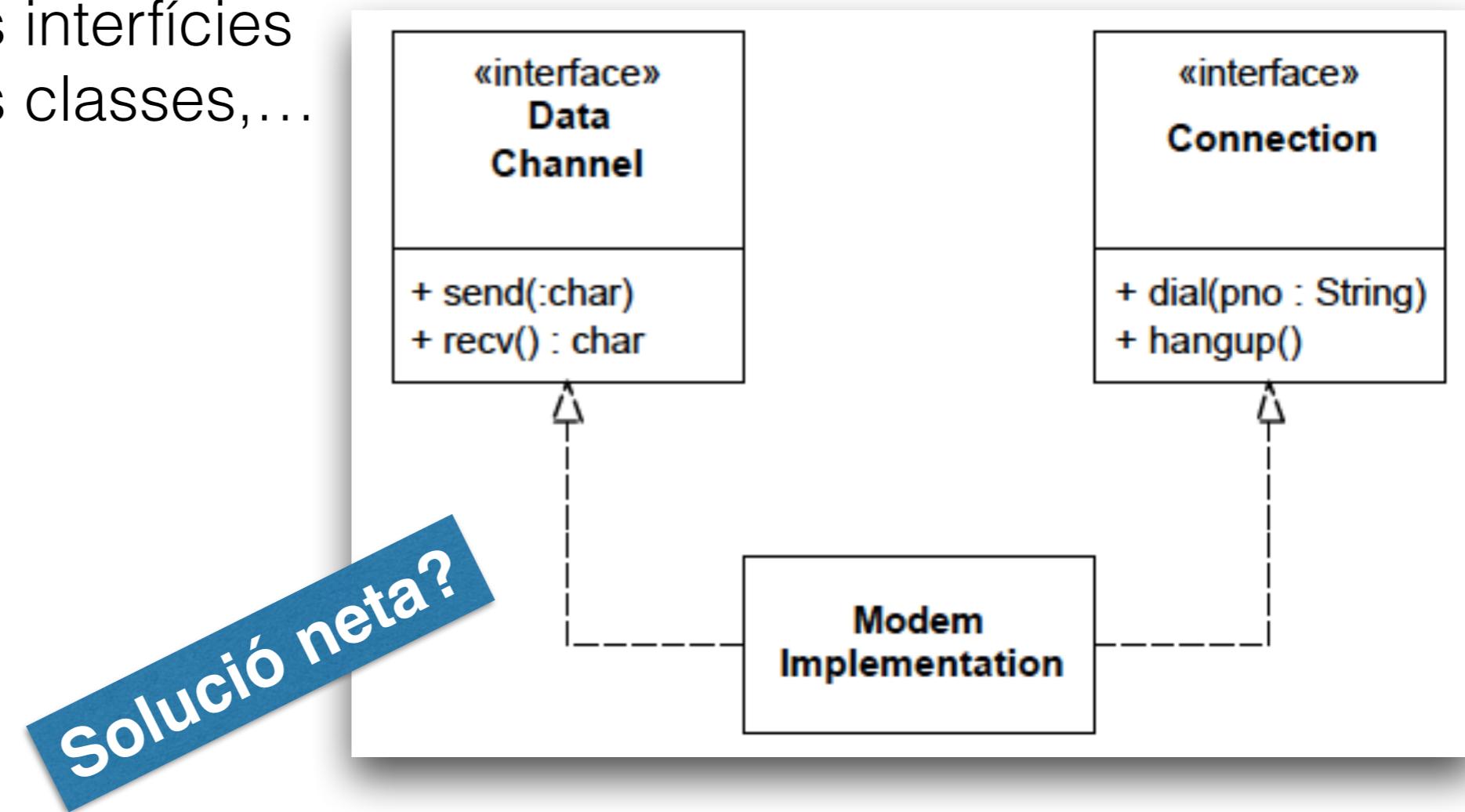
Exemple de SRP: Modem

- Quants actors actuen?
- Quins d'ells tenen comportaments diferents?

```
interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
}
```

3.2. Principis de Disseny: S.O.L.I.D.

- Cal identificar característiques que canvien per diferents raons i per a quines raons canvien
- Cal agrupar les característiques que canvien per les mateixes raons
- Per exemple,
 - usant diferents interfícies
 - usant diferents classes,...

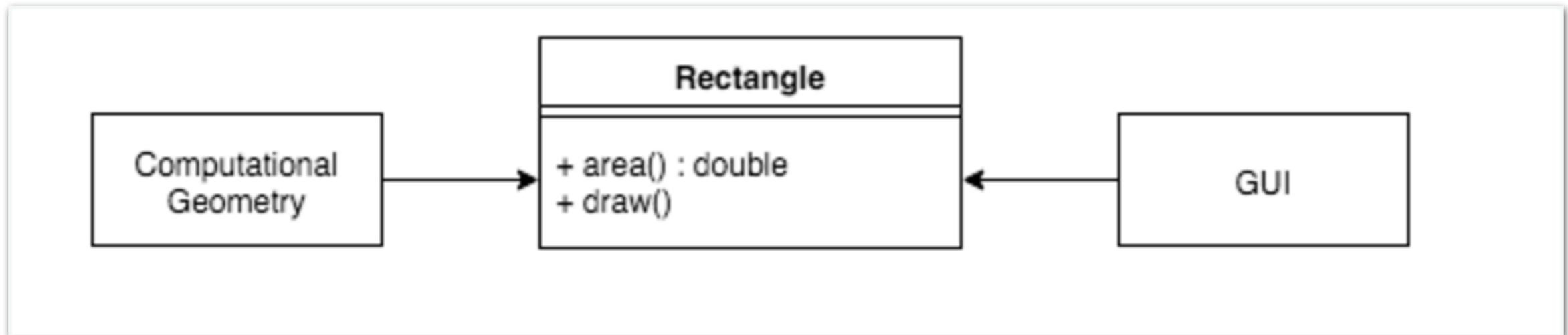


3.2. Principis de Disseny: S.O.L.I.D.

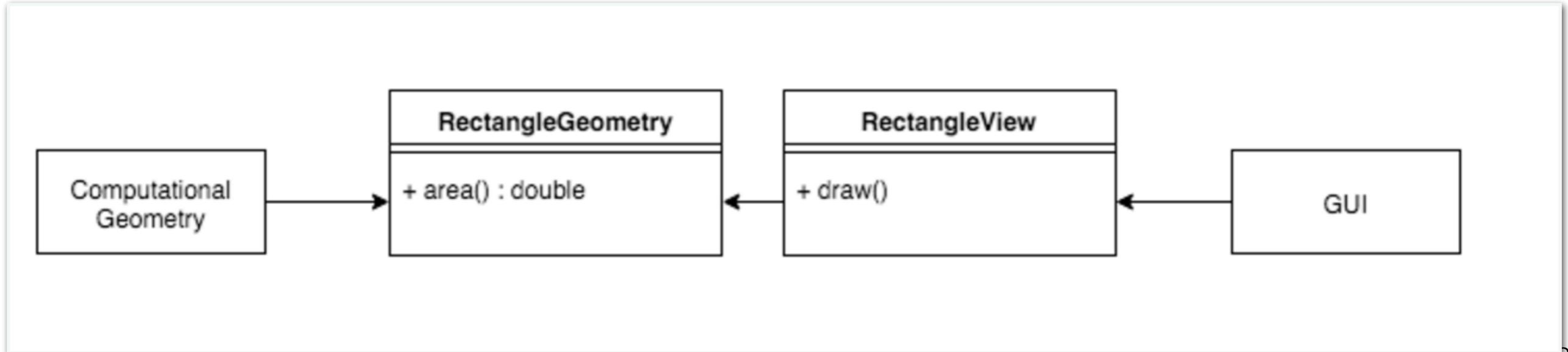
```
interface Employee
{
    public Pay calculate();
    public void report(Writer w);
    public void save();
    public void reload();
}
```

- Cal identificar característiques que canvien per diferents raons
- Cal agrupar les característiques que canvien per les mateixes raons

3.2. Principis de Disseny: S.O.L.I.D.



- Cal identificar característiques que canvien per diferents raons
- Cal agrupar les característiques que canvien per les mateixes raons



3.2. Principis de Disseny: S.O.L.I.D.

Open Closed Principle:

"Software components should be open for extension, but closed for modification"

Meyer, 88



- Els mòduls (classes, funcions, operacions, etc.) haurien de ser:
 - **Oberts** per extensió: per satisfer nous requisits
 - **Tancats** per 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

3.2. Principis de Disseny: S.O.L.I.D.

- Segueix el principi obert-tancat?

```
void DrawAllShapes(ShapePointer list[], int n)
{
    for (int i=0; i<n; i++)
    {
        struct Shape* s = list[i];
        switch (s->type)
        {
            case square: DrawSquare((struct Square*)s);
                           break;
            case circle: DrawCircle((struct Circle*)s);
                           break;
        }
    }
}
```

3.2. Principis de Disseny: S.O.L.I.D.

- Segueix el principi obert-tancat?

```
public void draw(Shape[] shapes) {  
    for( Shape shape : shapes ) {  
        switch (shape.getType()) {  
            case Shape.SQUARE:  
                draw( (Square)shape);  
                break;  
            case Shape.CIRCLE:  
                draw( (Circle)shape);  
                break;  
        }  
    }  
}
```

3.2. Principis de Disseny: S.O.L.I.D.

- Ara si segueix el principi obert-tancat:

```
public void draw(Shape[] shapes) {  
    for( Shape shape : shapes ) {  
        shape.draw();  
    }  
}
```

Consell (R. Martin):

- fés que les coses que canvien sovint estiguin lluny de les que no canvien (estàtiques)
- si unes depenen de les altres, **les coses que canvien sovint han de ser les que depenen de les que no canvien**

3.2. Principis de Disseny: S.O.L.I.D.

- Què passa si es vol dibuixar primer tots els Cercles i després tots el Quadrats?

```
public void draw(List<Shape> shapes) {  
    for (Shape shape : shapes) {  
        if (shape.getClass() == Shape.Circle) {  
            shape.draw();  
        }  
        for (Shape shape : shapes) {  
            if (shape.getClass() == Shape.Square) {  
                shape.draw();  
            }  
        }  
    }  
}
```

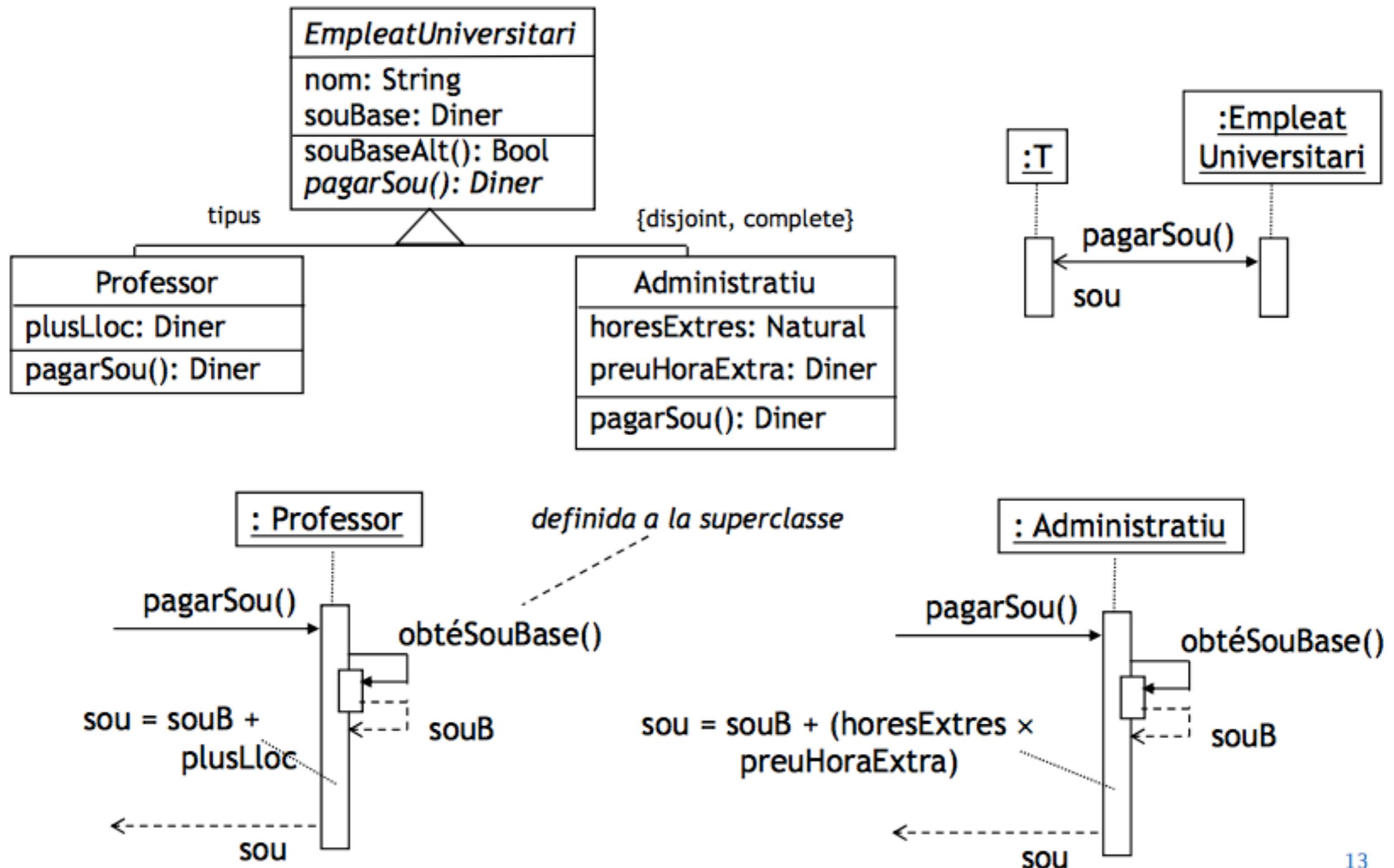
3.2. Principis de Disseny: S.O.L.I.D.

- Què passa si es vol dibuixar primer tots els Cercles i després tots el Quadrats?

```
public void draw(List<Shape> shapes) {  
    shapes.sort(new ShapeComparator());  
    shapes.forEach(draw());  
}
```

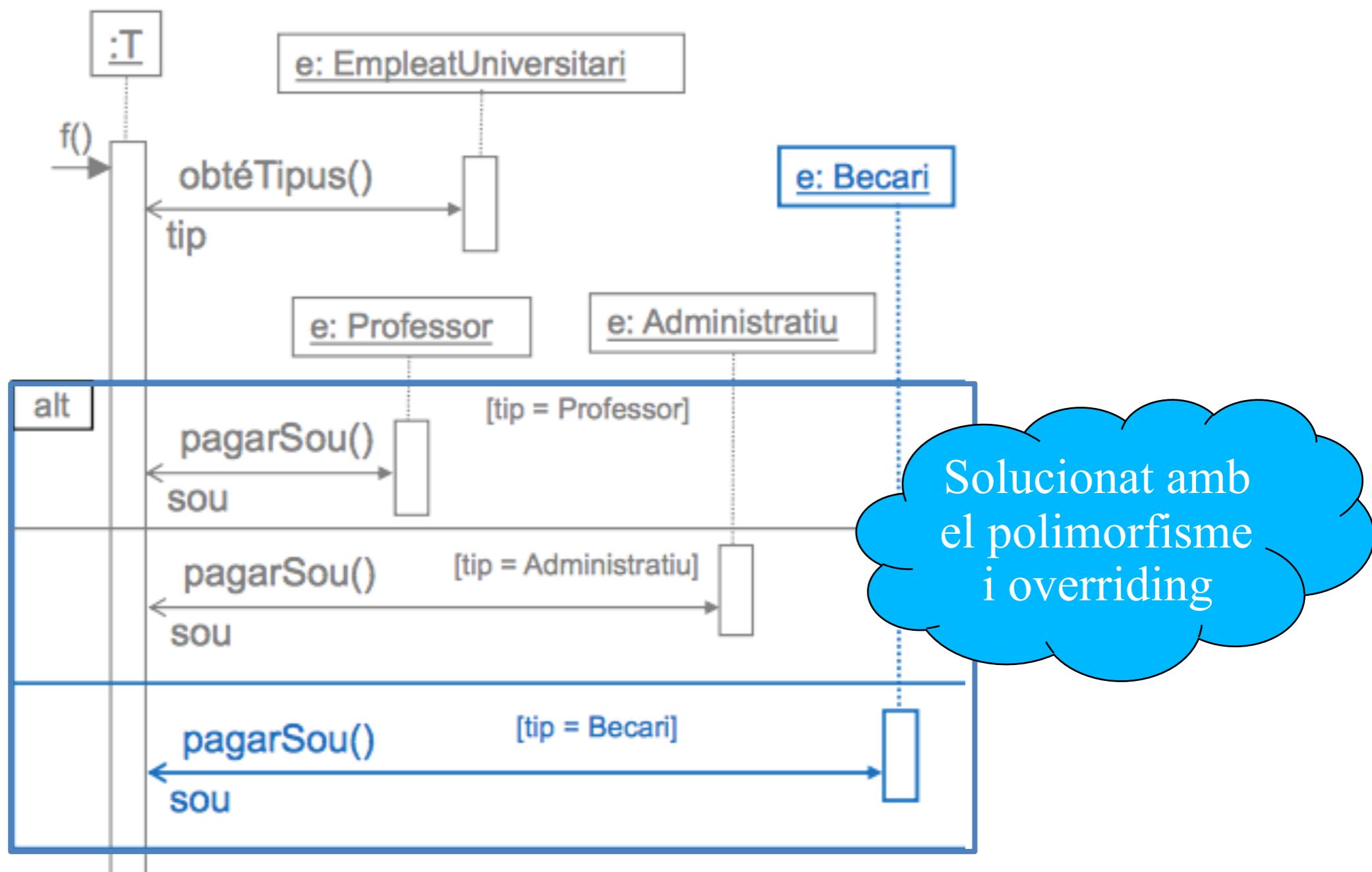
```
public class ShapeComparator implements Comparator<Shape> {  
    @Override  
    public int compare(Shape p1, Shape p2) {  
        return (p1 instanceof Shape.Square);  
    }  
}
```

3.2. Principis de Disseny: S.O.L.I.D.



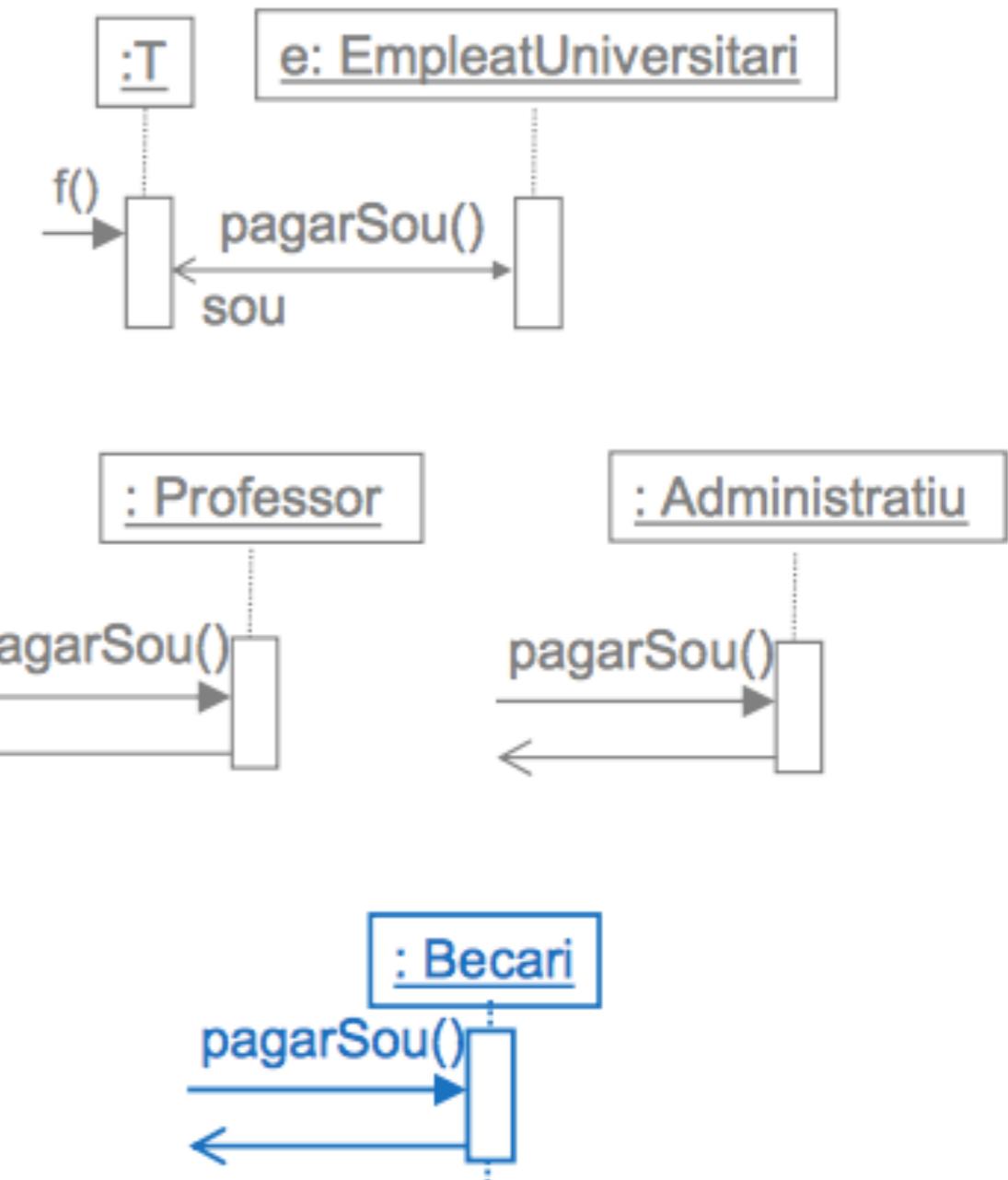
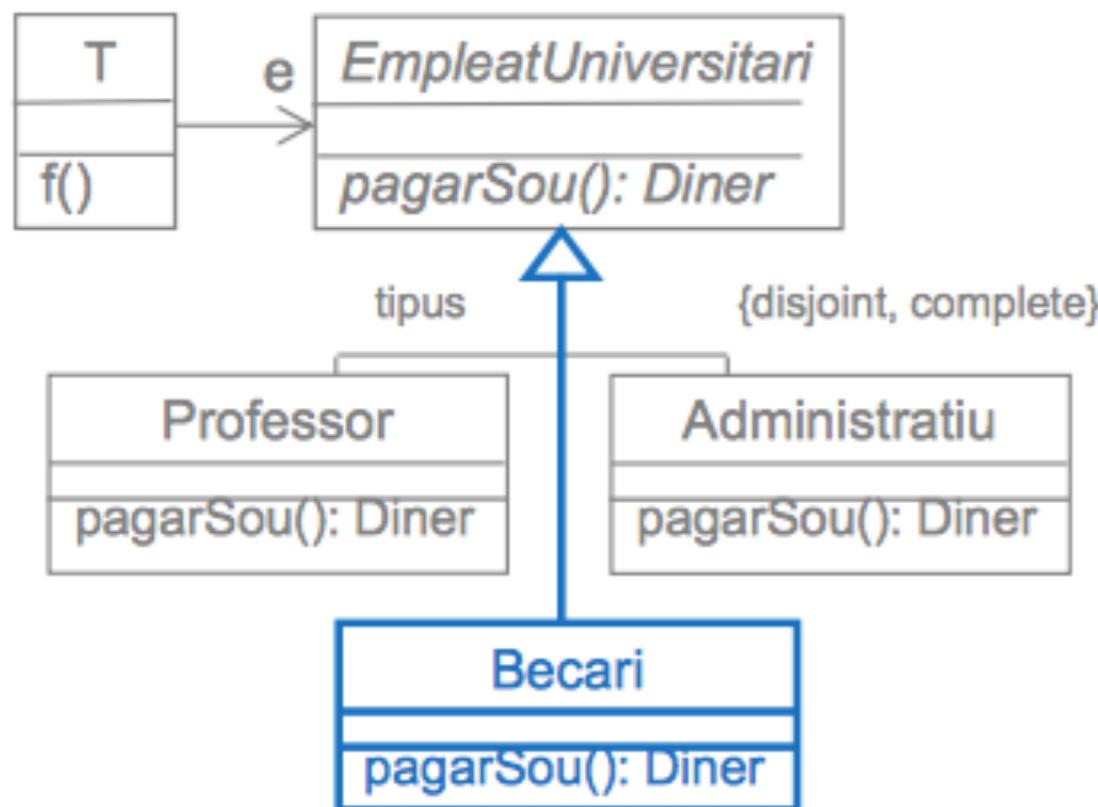
3.2. Principis de Disseny: S.O.L.I.D.

NouEmpleat Universitari: Becari



3.2. Principis de Disseny: S.O.L.I.D.

NouEmpleat Universitari: Becari



3.2. Principis de Disseny: S.O.L.I.D.

- Com comprometen el principi Obert-Tancat ...
 - Els atributs públics?

```
public class Device {  
    public boolean status;  
}
```

- Les variables globals?

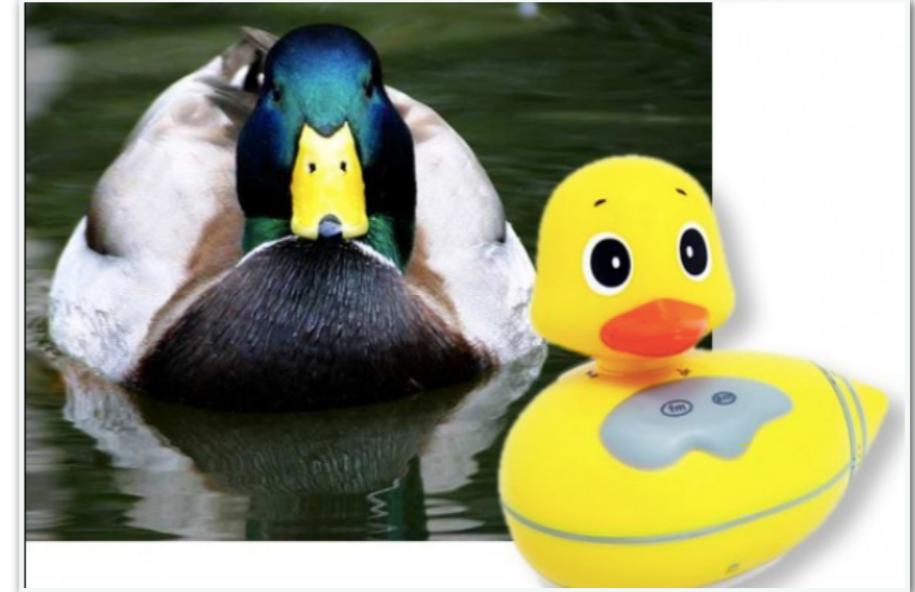
```
public class Time {  
    int hours; int minutes; int seconds;  
}
```

3.2. Principis de Disseny: S.O.L.I.D.

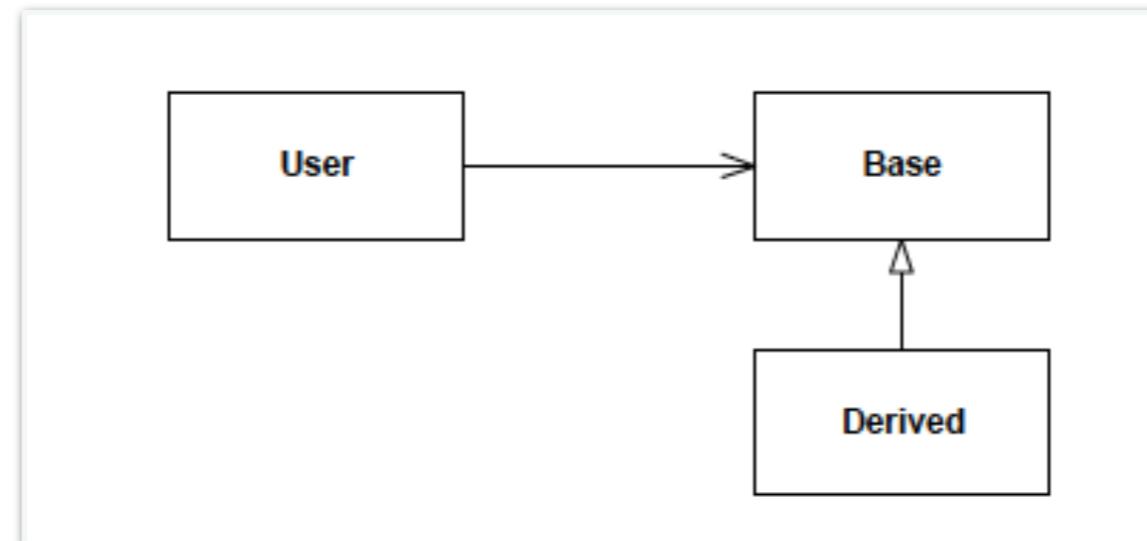
Liskov Substitution Principle:

"Derived types must be completely substitutable for their base types"

Barbra Liskov 1988

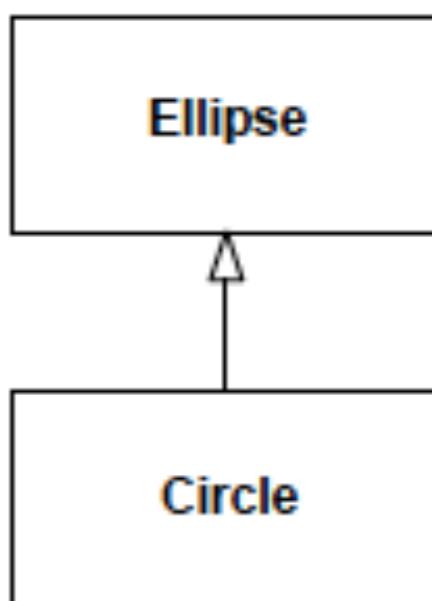


- Donada una entitat **Base** amb un cert mètode i altres subentitats **Derivades** que implementen el mètode original, el comportament d'un objecte **o1** que era de tipus Base no ha de variar si s'usa **o2** de tipus Derivat



3.2. Principis de Disseny: S.O.L.I.D.

Liskov Substitution Principle:



Ellipse	
-	itsFocusA : Point
-	itsFocusB : Point
-	itsMajorAxis : double
+	Circumference() : double
+	Area() : double
+	GetFocusA() : Point
+	GetFocusB() : Point
+	GetMajorAxis() : double
+	GetMinorAxis() : double
+	SetFoci(a:Point, b:Point)
+	SetMajorAxis(double)

Què passa amb f quan e és de tipus Circle?

```
void f(Ellipse& e)
{
    Point a(-1,0);
    Point b(1,0);
    e.SetFoci(a,b);
    e.SetMajorAxis(3);
    assert(e.GetFocusA() == a);
    assert(e.GetFocusB() == b);
    assert(e.GetMajorAxis() == 3);
}
```

```
void Circle::SetFoci(const Point& a, const Point& b)
{
    itsFocusA = a;
    itsFocusB = a;
}
```

3.2. Principis de Disseny: S.O.L.I.D.

Liskov Substitution Principle:

```
void f(Ellipse& e)
{
    if (typeid(e) == typeid(Ellipse))
    {
        Point a(-1, 0);
        Point b(1, 0);
        e.SetFoci(a, b);
        e.SetMajorAxis(3);
        assert(e.GetFocusA() == a);
        assert(e.GetFocusB() == b);
        assert(e.GetMajorAxis() == 3);
    }
    else
        throw NotAnEllipse(e);
}
```

- Violacions en el LSP son violacions latents del Open-Closed Principle

3.2. Principis de Disseny: S.O.L.I.D.

Interface Segregation Principle:

"Clients should not be forced to implement unnecessary methods which they will not use"

Gamma et al., 1995



- No s'ha d'obligar a les classes a dependre de classes o mètodes que no han d'usar (deriva del SRP)
- Tot i que hi hagin classes molt grans que inclouen molts mètodes (interfícies no cohesionades), els clients (altres classes) només haurien de conèixer classes abstractes que tinguin interfícies cohesionades.
- Això permet tenir **ALTA** cohesió i evitar classes fràgils

3.2. Principis de Disseny: S.O.L.I.D.

Interface Segregation Principle:

```
interface IWorker {  
    public void work();  
    public void eat();  
}  
  
class Worker implements IWorker{  
    public void work() {  
        // ....working  
    }  
    public void eat() {  
        // ..... eating in launch  
    }  
}  
  
class SuperWorker implements IWorker{  
    public void work() {  
        //.... working much more  
    }  
    public void eat() {  
        //.... eating in launch break  
    }  
}
```

```
class Manager {  
    IWorker worker;  
  
    public void setWorker(IWorker w) {  
        worker=w;  
    }  
  
    public void manage() {  
        worker.work();  
    }  
}
```

I si afegim un Robot que no menja?

3.2. Principis de Disseny: S.O.L.I.D.

Interface Segregation Principle:

```
interface IWorker extends Feedable, Workable {  
}  
  
interface IWorkable {  
    public void work();  
}  
  
interface IFeedable{  
    public void eat();  
}  
  
class Worker implements IWorkable, IFeedable{  
    public void work() {  
        // ....working  
    }  
  
    public void eat() {  
        //.... eating in launch break  
    }  
}  
  
class Robot implements IWorkable{  
    public void work() {  
        // ....working  
    }  
}
```

3.2. Principis de Disseny: S.O.L.I.D.

Dependency Inversion Principle:

"Depend on abstractions, not on concretions"

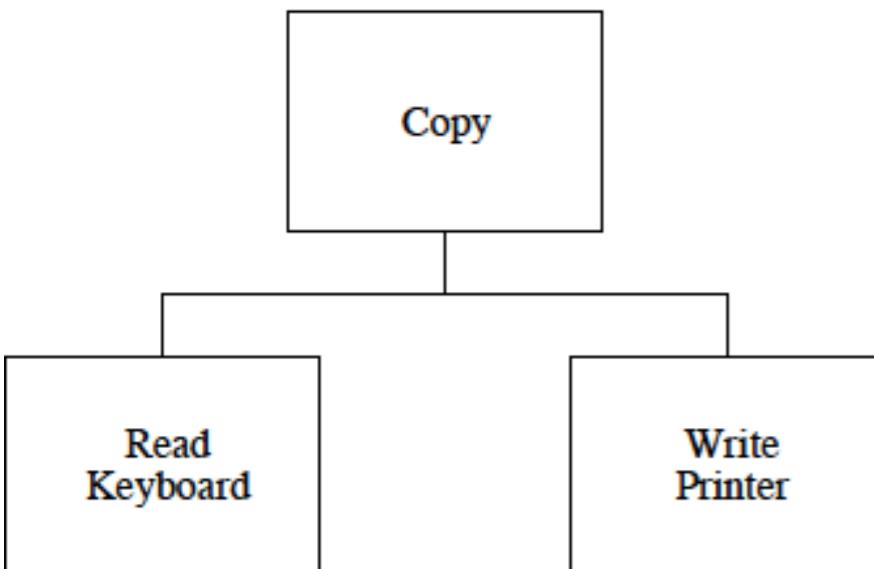
Martin, 1997



- Les classes d'alt nivell no han de canviar per que canvien les classes més senzilles o de baix nivell. Les dues haurien de dependre d'abstraccions.
- Les abstraccions no han de dependre de detalls tecnològics, són els detallls que haurien de dependre de les abstraccions.
- Aplicar aquest principi dona **BAIX** acoblament

3.2. Principis de Disseny: S.O.L.I.D.

Dependency Inversion Principle?



```
void Copy()
{
    int c;
    while ((c = ReadKeyboard()) != EOF)
        WritePrinter(c);
}
```

I si el volem extender per a guardar a disc?

3.2. Principis de Disseny: S.O.L.I.D.

Dependency Inversion Principle?

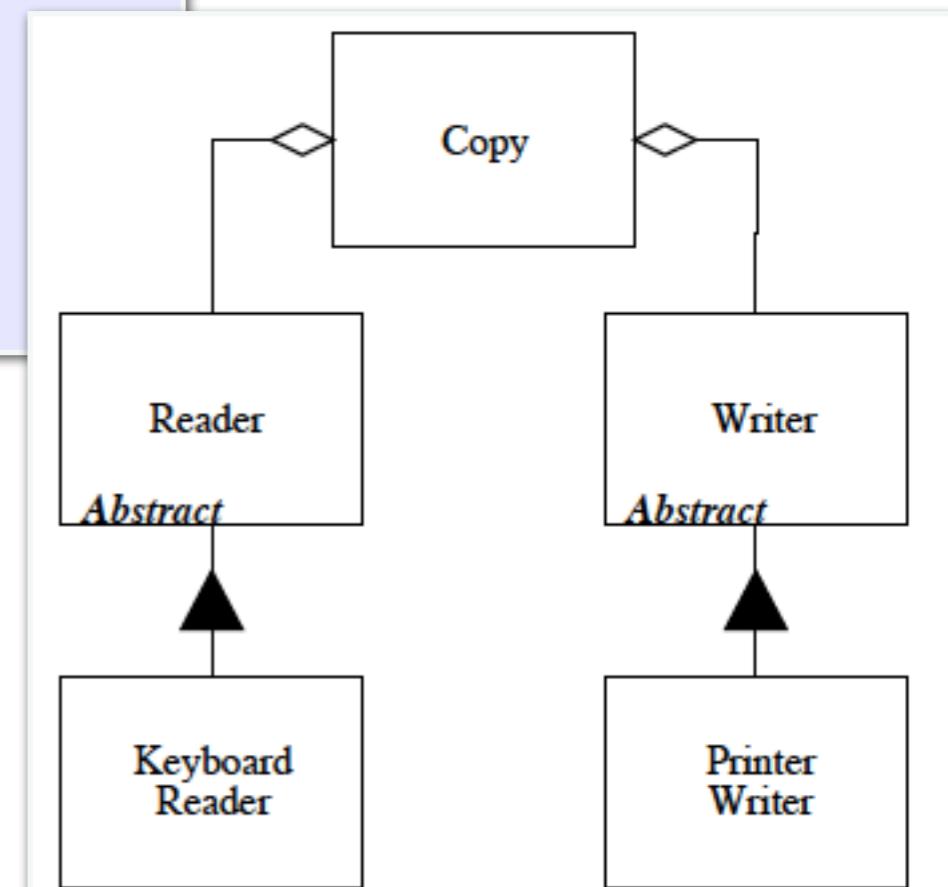
```
enum OutputDevice {printer, disk};

void copy(OutputDevice dev)
{
    int c;
    while( (c=readKeyboard()) != EOF)
    {
        if (dev == printer)
            writePrinter(c);
        else
            writeDisk(c);
    }
}
```

3.2. Principis de Disseny: S.O.L.I.D.

Dependency Inversion Principle:

```
void copy(Reader input, Writer output)
{
    int c;
    while((c=input.read()) != EOF) {
        output.write(c);
    }
}
```



3.2. Principis de Disseny: S.O.L.I.D.

Dependency Inversion Principle:

- Com es creen les classes dels diferents tipus concret?
- Mitjançant el patró d'**Injecció de Dependències**, es subministren objectes a la classe, en lloc de crear-los dins de la mateixa classe.

```
public class Vehiculo {  
  
    private Motor motor = new Motor();  
  
    /** @retorna la velocidad del vehículo*/  
    public Double enAceleracionDePedal(int presionDePedal) {  
        motor.setPresionDePedal(presionDePedal);  
        int torque = motor.getTorque();  
        Double velocidad = ... //realiza el cálculo  
        return velocidad;  
    }  
}
```

3.2. Principis de Disseny: S.O.L.I.D.

Dependency Inversion Principle:

```
public class Vehiculo {  
  
    private Motor motor = null;  
  
    public void setMotor(Motor motor){  
        this.motor = motor;  
    }  
  
    /** @retorna la velocidad del vehículo*/  
    public Double enAceleracionDePedal(int presionDePedal) {  
        Double velocidad = null;  
        if (null != motor){  
            motor.setPresionDePedal(presionDePedal);  
            int torque = motor.getTorque();  
            velocidad = ... //realiza el cálculo  
        }  
        return velocidad;  
    }  
}
```

```
public class VehiculoFactory {  
  
    public Vehiculo construyeVehiculo() {  
        Vehiculo vehiculo = new Vehiculo();  
        Motor motor = new Motor();  
        vehiculo.setMotor(motor);  
        return vehiculo;  
    }  
}
```