



Tema 3: Disseny

Anna Puig

Enginyeria Informàtica

Facultat de Matemàtiques i Informàtica,

Universitat de Barcelona

Curs 2019/2020



UNIVERSITAT DE
BARCELONA

Temari

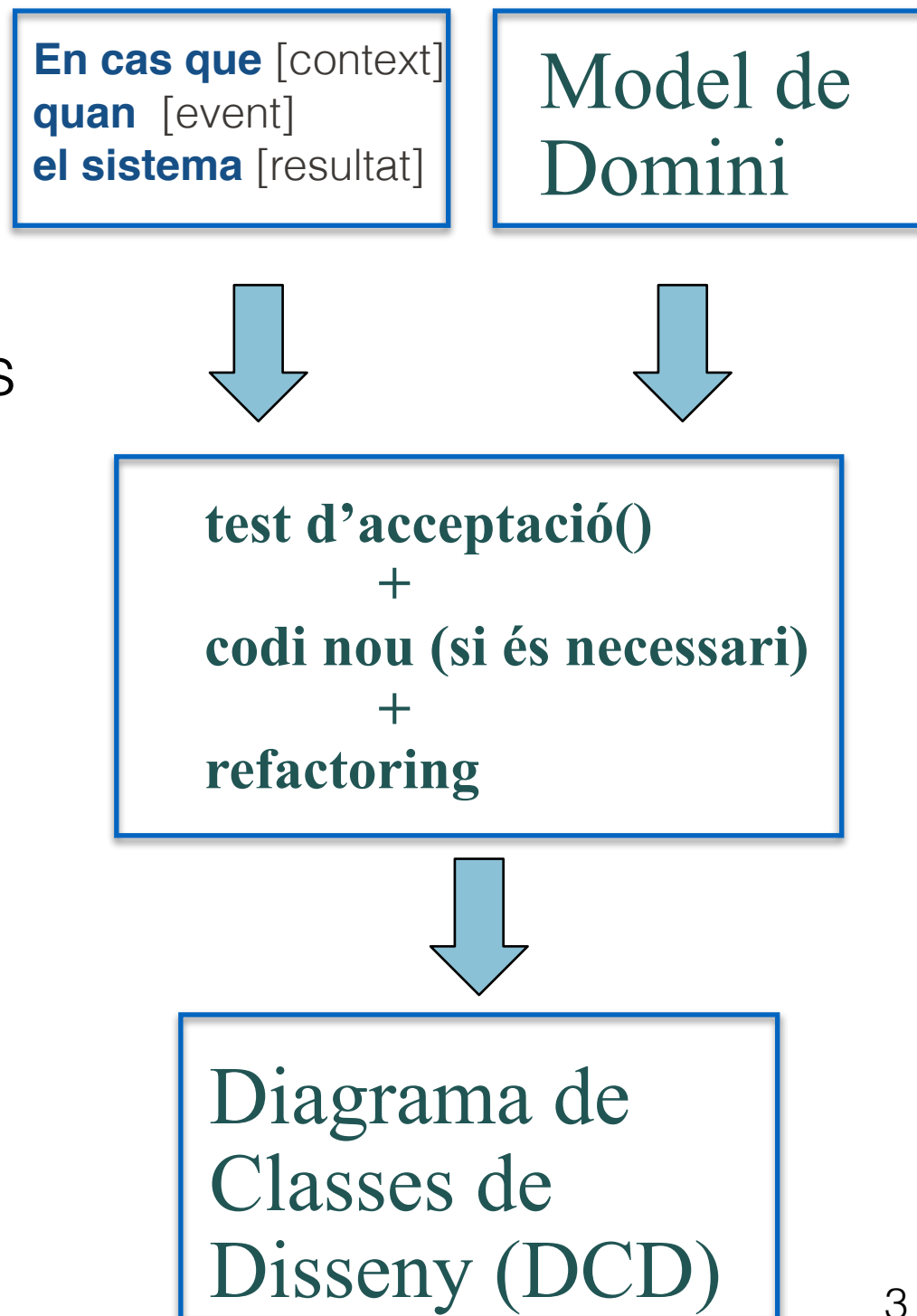
1	Introducció al procés de desenvolupament del software	
2	Anàlisi de requisits i especificació	
3	Disseny	
4	Del disseny a la implementació	3.1 Introducció
5	Ús de frameworks de testing	3.2 Principis de Disseny: S.O.L.I.D.
		3.3 Patrons arquitectònics
		3.4 Patrons de disseny

3.4. Patrons de disseny

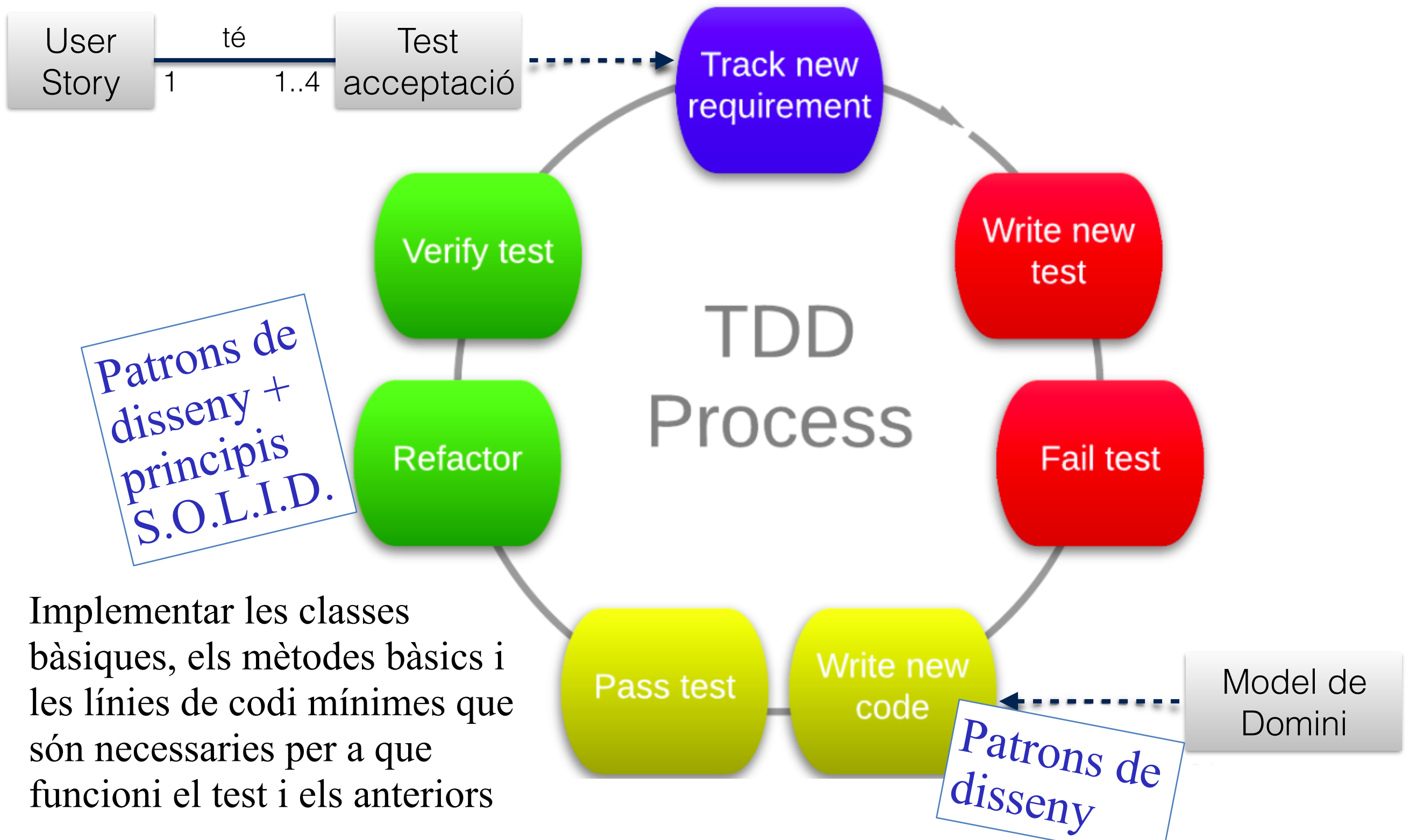
Per a cada **criteri d'acceptació** identificat en l'especificiació es defineix un **test d'acceptació**

1. Es dissenya la seqüència de crides que produirà **l'event** navegant per les classes necessàries
2. A mesura que es necessiten classes s'afegeixen en el **Diagrama de Classes** definint la **navegabilitat** entre elles a partir, si és possible de les classes conceptuais del **Model de Domini**

Com es distribueix el codi per les classes? Usant els patrons GRASP



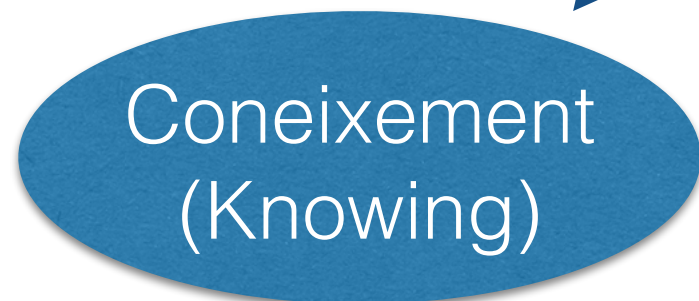
TDD en el Model o Controlador



Implementar les classes bàsiques, els mètodes bàsics i les línies de codi mínimes que són necessaries per a que funcioni el test i els anteriors

3.4.1. Patrons de disseny GRASP

- **Principis generals (GRASP):** descripció dels principis bàsics d'assignació de **responsabilitats** a les classes expressades com a patrons
- L'assignació de **responsabilitats** a objectes és determinar quines són les obligacions concretes dels objectes del DCD per donar resposta als events externs.



- Saber sobre els **atributs** privats de l'objecte
- Saber sobre els **objectes** associats
- Saber dades que se'n poden **derivar o calcular**



- Fer alguna cosa en el propi objecte (com crear o fer càlculs).
- Iniciar una acció en altres objectes
- Controlar i coordinar les activitats d'altres objectes

3.4. Patrons de disseny

Hi han diferents tipus de patrons (o regles generals per a solucionar un problema):

- **Principis generals (GRASP):** descripció dels principis bàsics d'assignació de responsabilitats a les classes expressades com a patrons (*són molt generals*)

General Responsibility Assignment Software Patterns (GRASP)

- Distribuir responsabilitats és la part més difícil del disseny OO. Consumeix una bona part del temps

- Patrons GRASP:

Expert

Baix Acoblament ✓

Controlador

Indirecció

Creador

Alta Cohesió ✓

Polimorfisme ✓

Variacions Protegides

3.4. Patrons de disseny

Per ajudar a la part de **refactoring** hi han diferents tipus de patrons (o regles generals per a solucionar un problema):

- **Patrons de disseny més específics:** es defineixen solucions concretes a problemes més concrets (de creació, d'estructura i de comportament) a nivell de *classe* i d'objecte.

Gang of Four (GoF)

Creació:

- **Factory method**
- **Abstract Factory**
- Builder
- Prototype
- **Singleton**
- Object pool

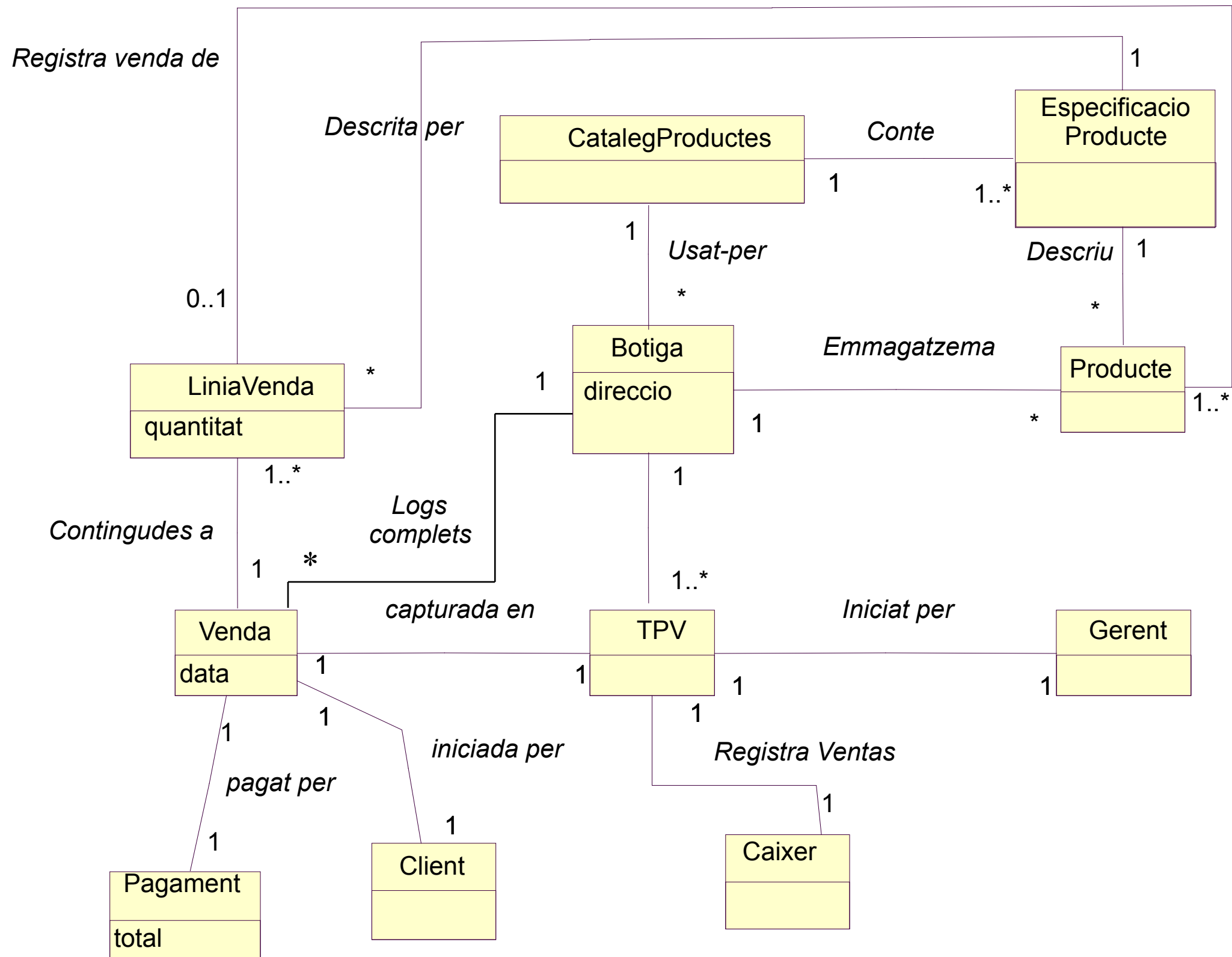
Estructura:

- *Class Adapter*
- Object Adapter
- **Facade**
- **Composite**
- Decorator
- Flyweight
- Proxy

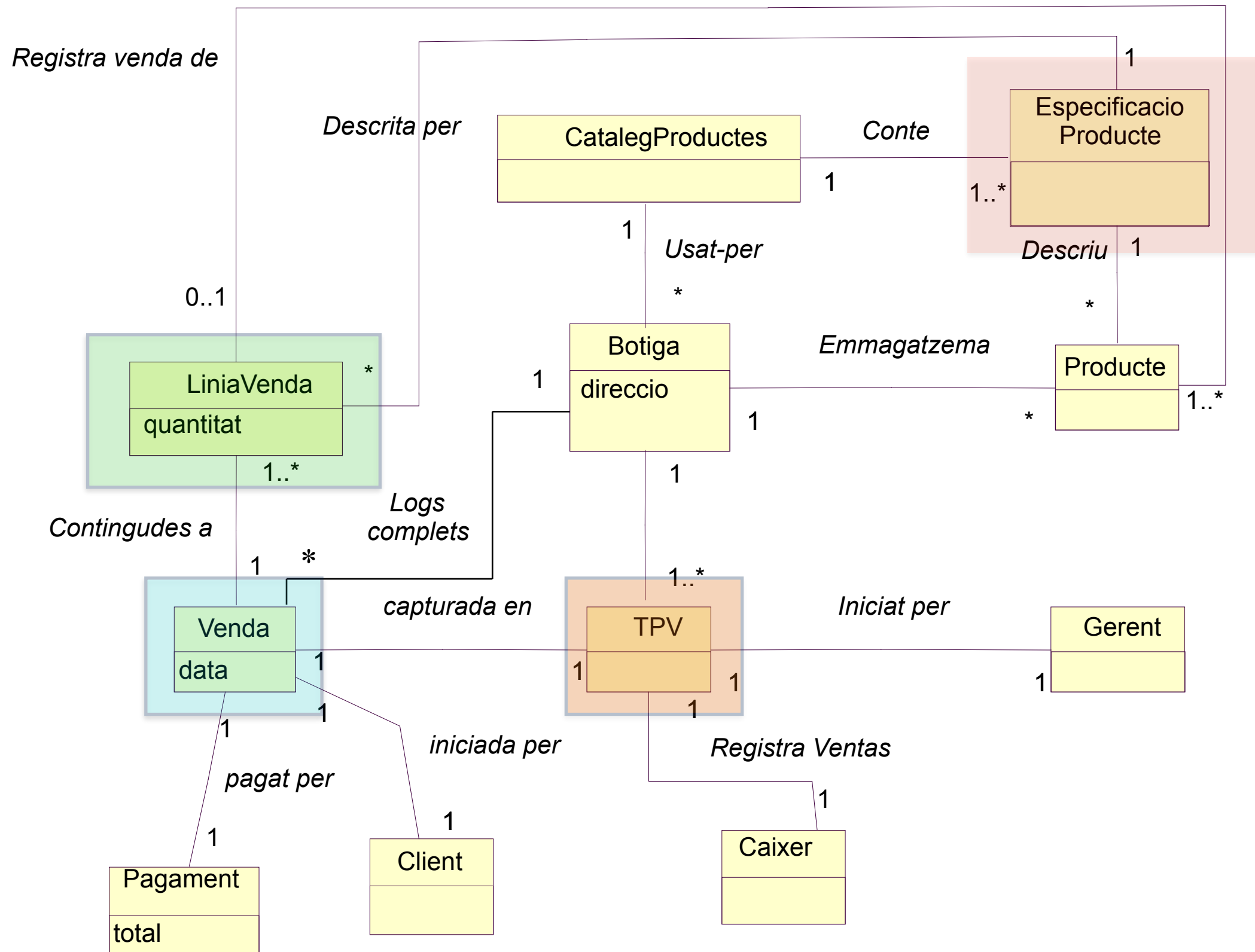
Comportament:

- *Interpreter*
- *Template method*
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- **Observer**
- **State**
- **Strategy**
- Visitor

Exemple model domini TPV



Càlcul del total d'una venda



3.4.1. Patrons de disseny GRASP

Nom del patró: Patró Expert

Context: Assignació de responsabilitats a objectes

Problema

- Baixa cohesió en algunes classes
- Alt acoblament entre diferents classes

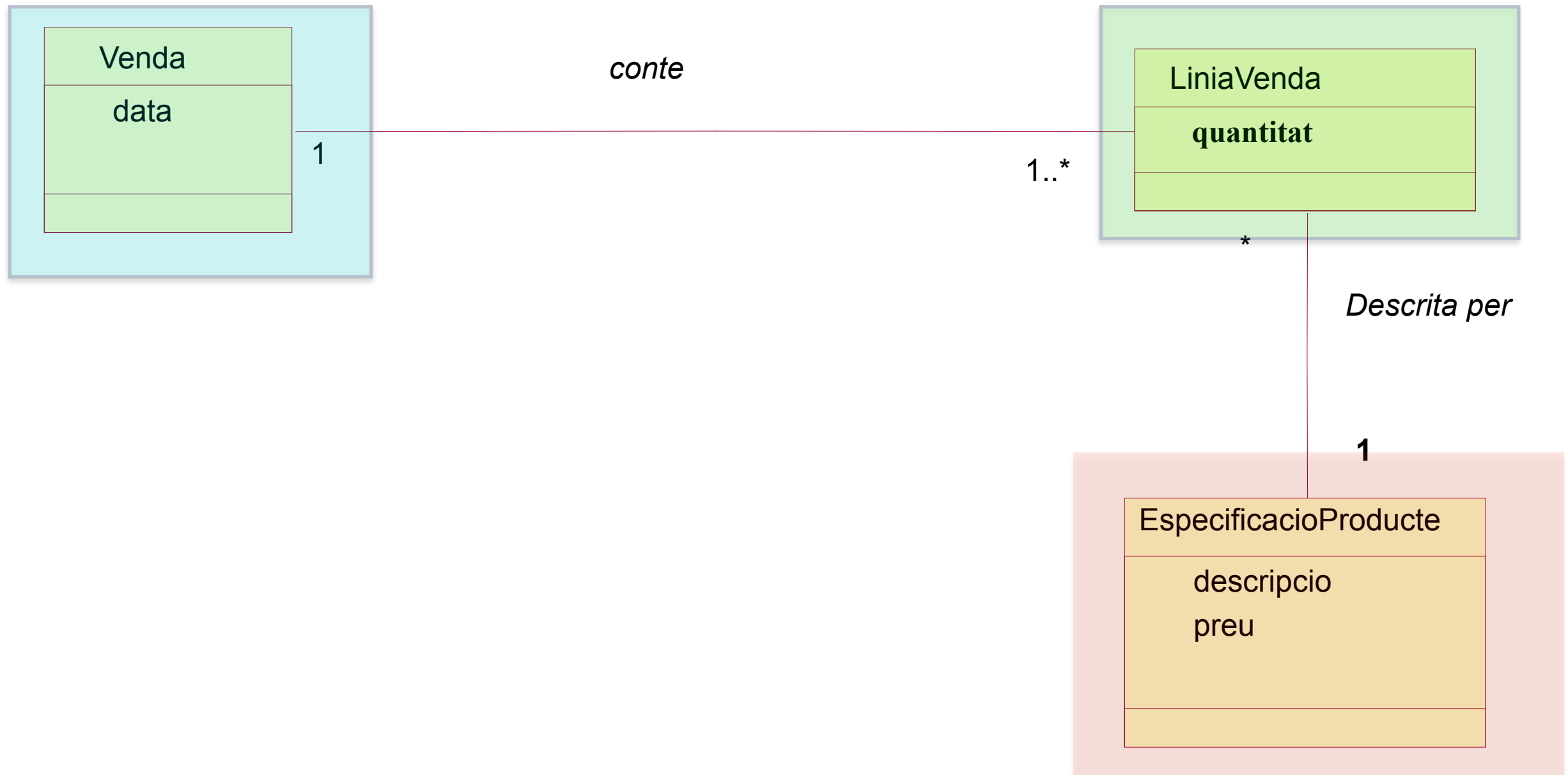
Solució

- Assignar la responsabilitat a la classe que té la informació necessària per fer aquella responsabilitat
- Cada objecte és responsable de mantenir la seva pròpia informació (principi d'encapsulament).
- Si té relacions de composició amb altres objectes (les seves parts) també serà el responsable de:
 - conèixer la informació d'ells,
 - crear-los (patró creador)
 - delegar-li les seves operacions.

NOTA: No sempre existeix un únic expert, sinó que poden existir experts parcials que hauran de col·laborar.

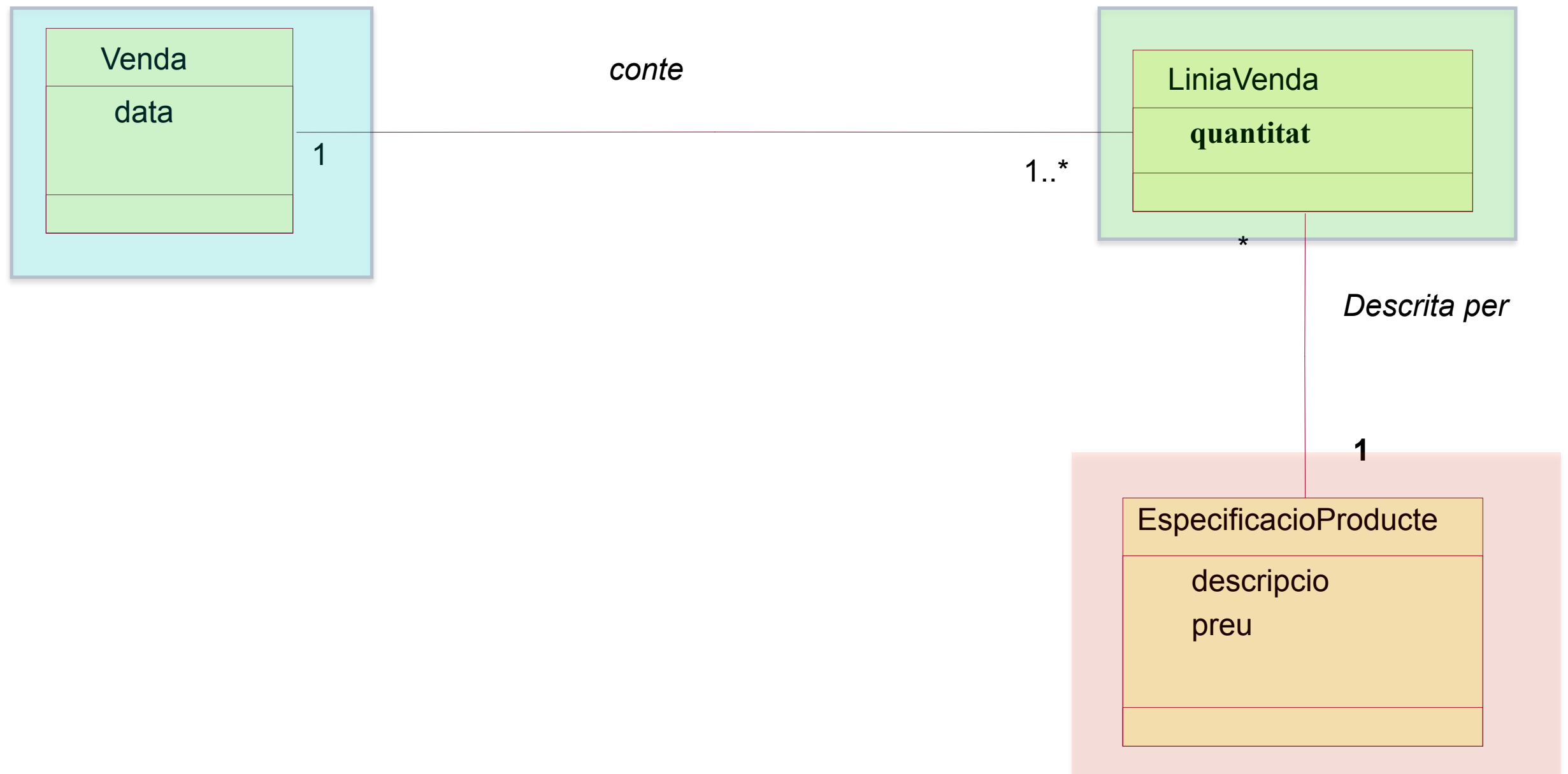
Exemple d'aplicació

- Qui és el responsable de calcular el total d'una venda?
Com seria el seu DS?



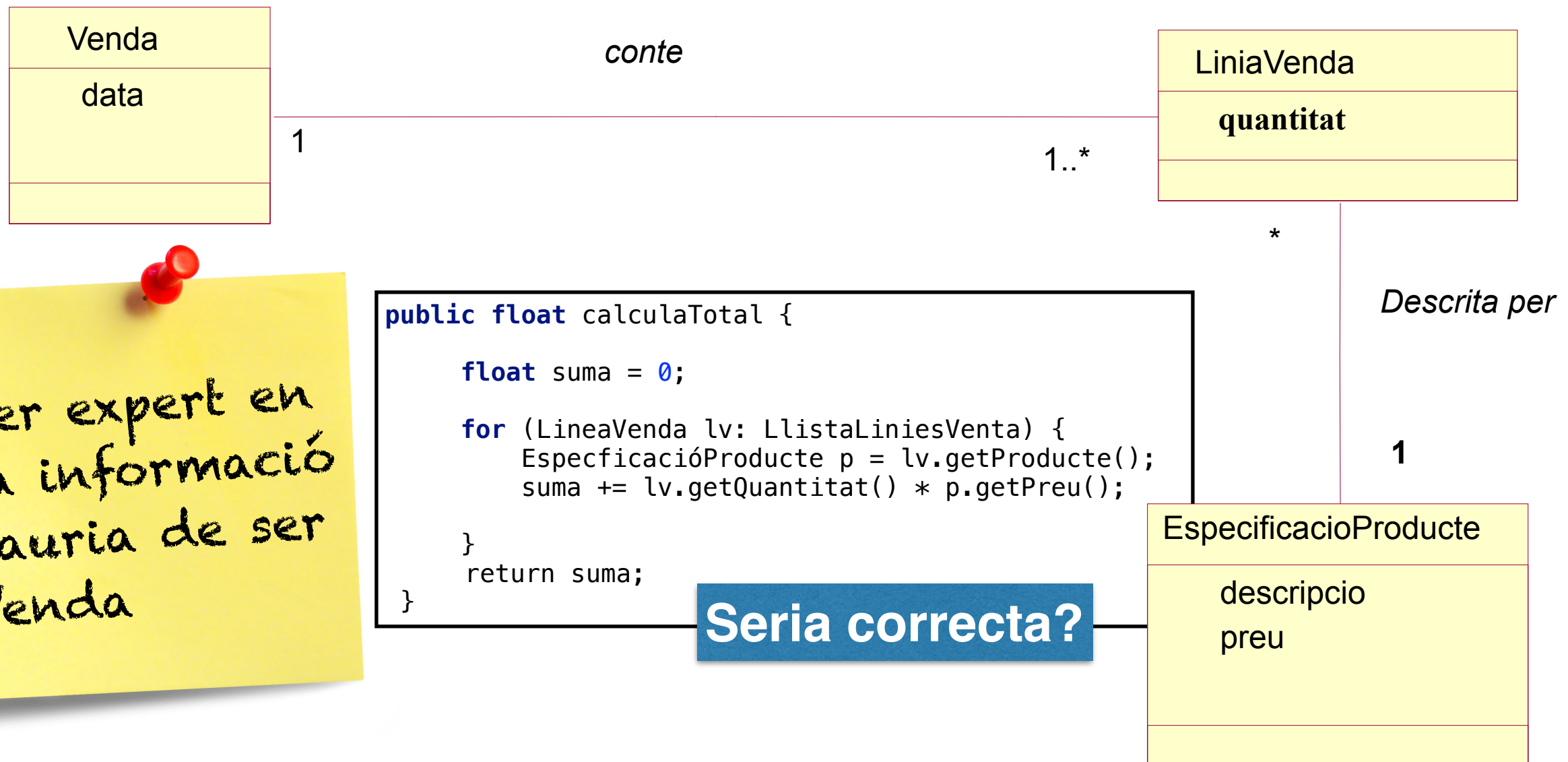
Exemple d'aplicació

- Qui és el responsable de calcular el total d'una venda?
Com seria el seu DS?



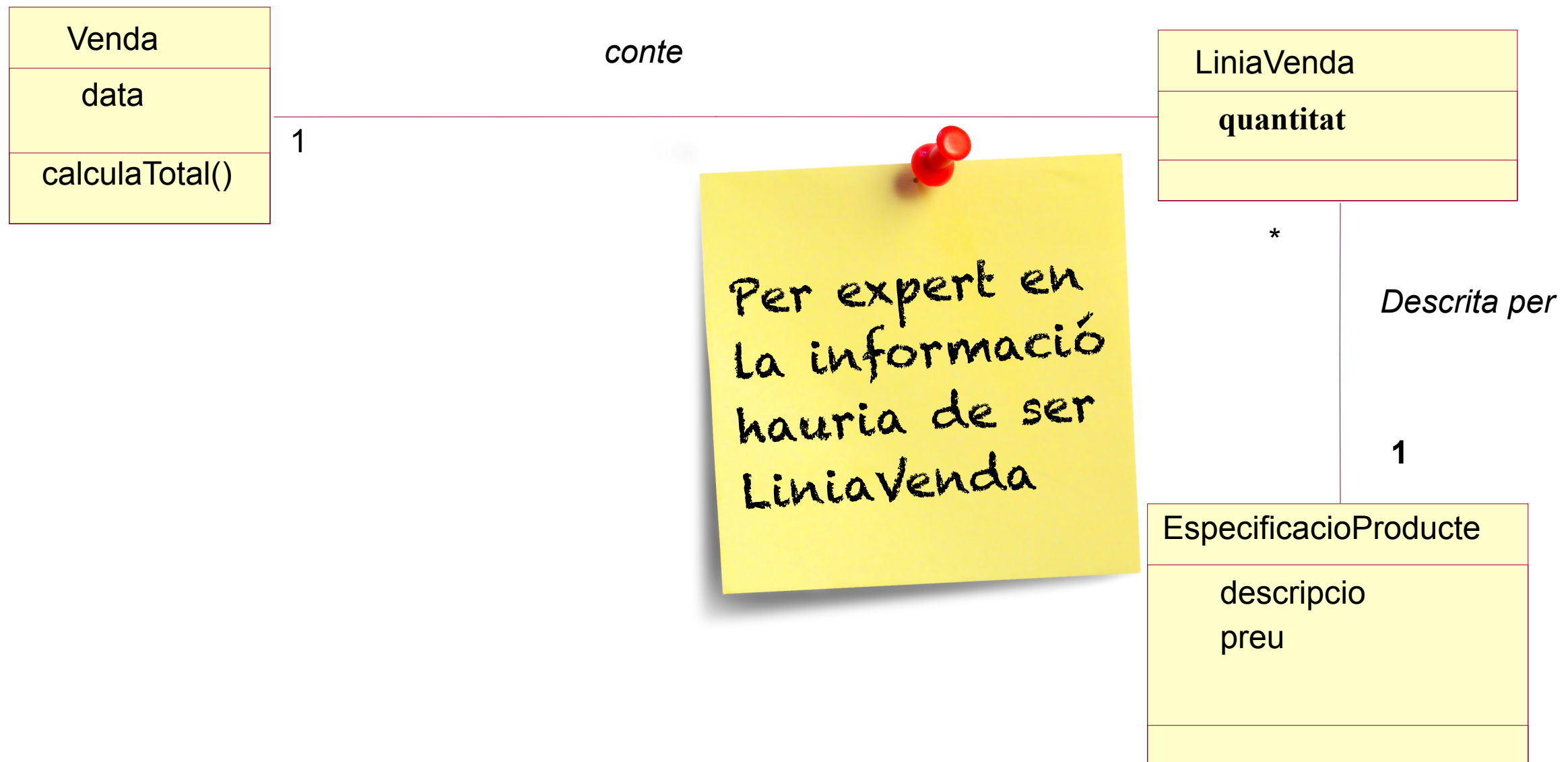
Exemple d'aplicació

- Qui és el responsable de conèixer el **total d'una venda** ?



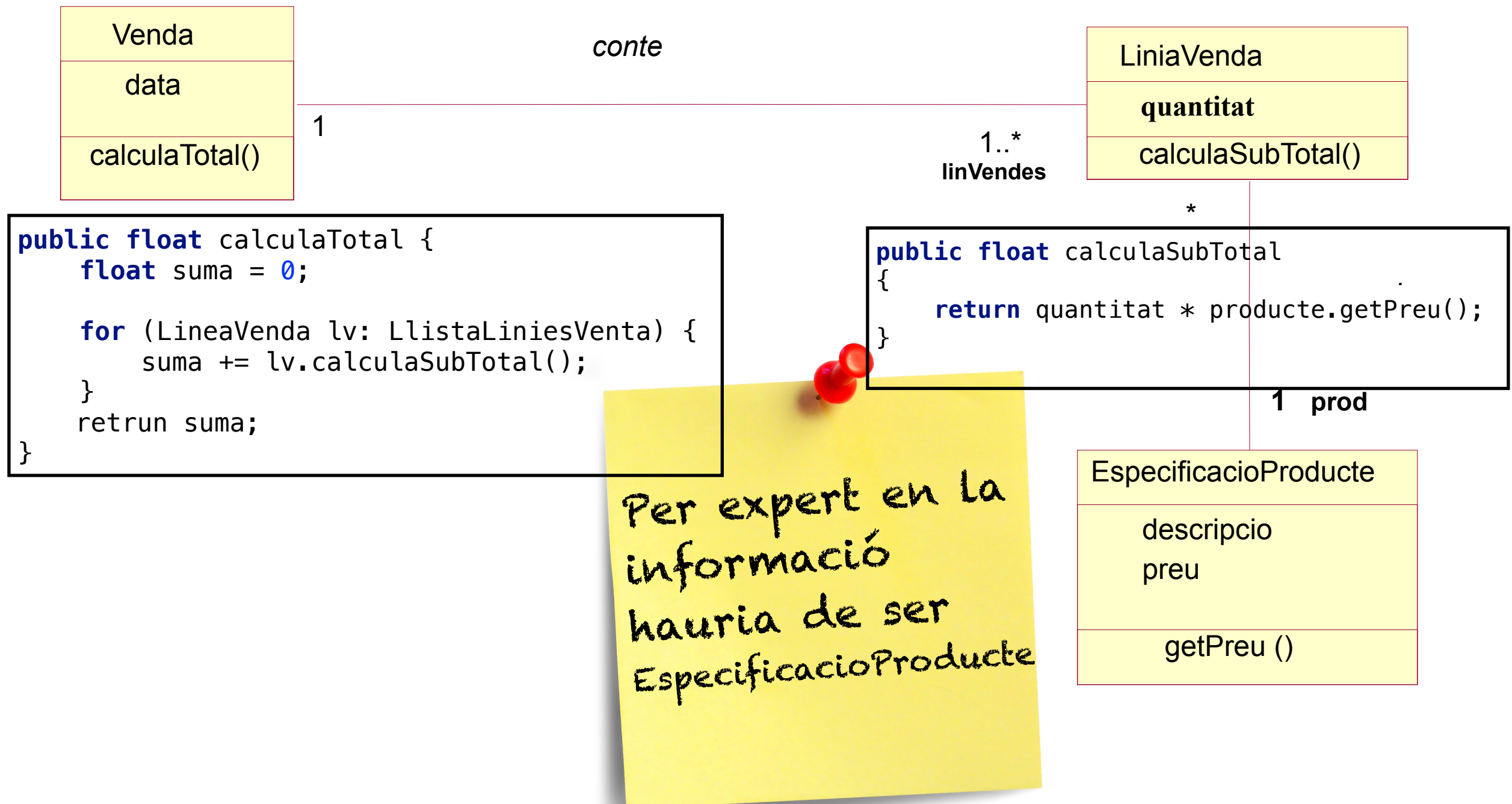
Exemple d'aplicació

- Qui és el responsable de conèixer el **subtotal** de cada línia de Venda?



Exemple d'aplicació

- Qui és el responsable de conèixer el **preu d'un producte**?





3.4.1. Patrons de disseny GRASP

Nom del patró: **Patró Expert**

Context: Assignació de responsabilitats a objectes

Pros:

- Es manté encapsulament  Baix acoblament
- Conducta distribuïda entre classes  Alta cohesió.
- No es creen classes “Deu”.

Cons:

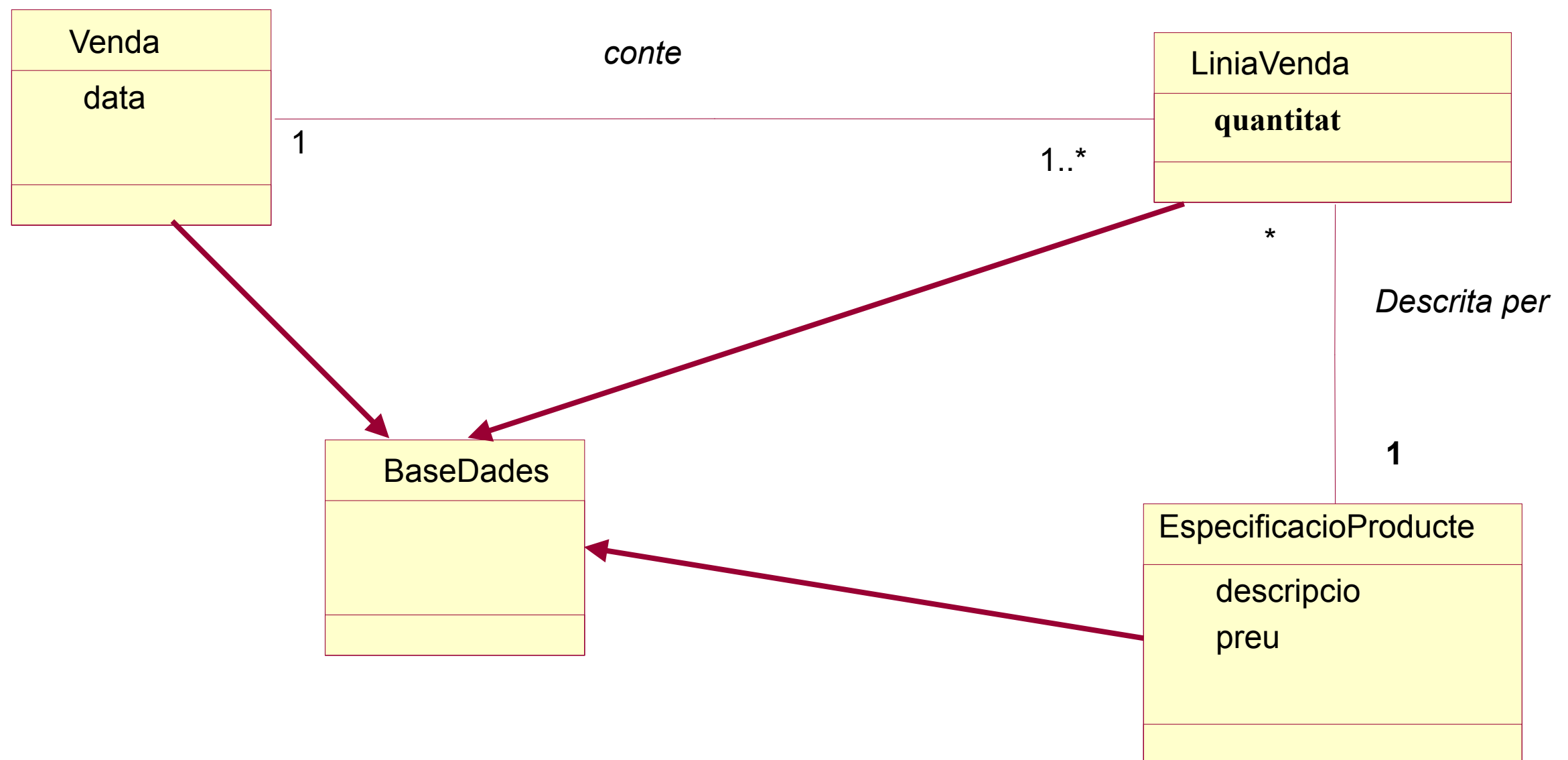
- Hi ha situacions en què no porta a bons dissenys, usualment per problemes d'acoblament i cohesió. Es trenquen principis arquitectònics (p.e., arquitectura a 3 capes)

Ex: **Interfície d'usuari**: Visualitzar les línies de venda en una finestra

Ex: **Persistència**: Emmagatzemar una venda a la base de dades

Inconvenients

- En alguns casos, aplicat al límit, pot portar a mals dissenys i acoblaments. Cal combinar-lo amb altres patrons



3.4.1. Patrons de disseny GRASP

Nom del patró: Patró Creador

Context: Assignació de la creació d' objectes

Problema:

- Baixa cohesió en algunes classes
- Alt acoblament entre diferents classes

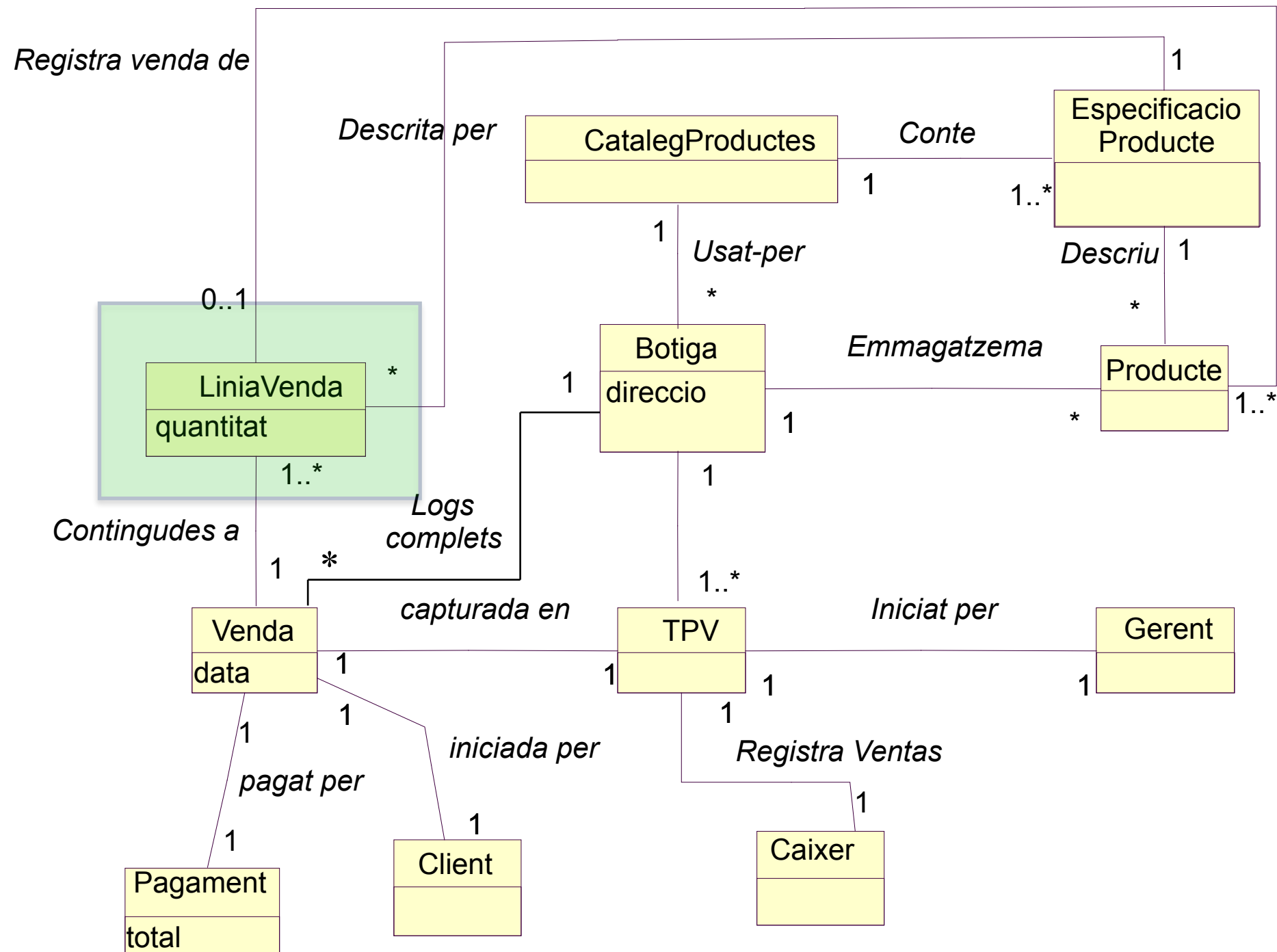
Solució:

Assignar a una classe **B** la responsabilitat de **crear** una instància d'A si es compleixen una o més de les següents condicions:

- B agrega objectes de classe A
- B conté objectes de classe A
- B manté un registre de les instàncies de la classe A
- B usa molts objectes de classe A
- B conté les dades per inicialitzar A en el moment de la seva creació
- Si més d'una classe compleix les condicions, seleccionar aquella que agrega o conté objectes de classe A

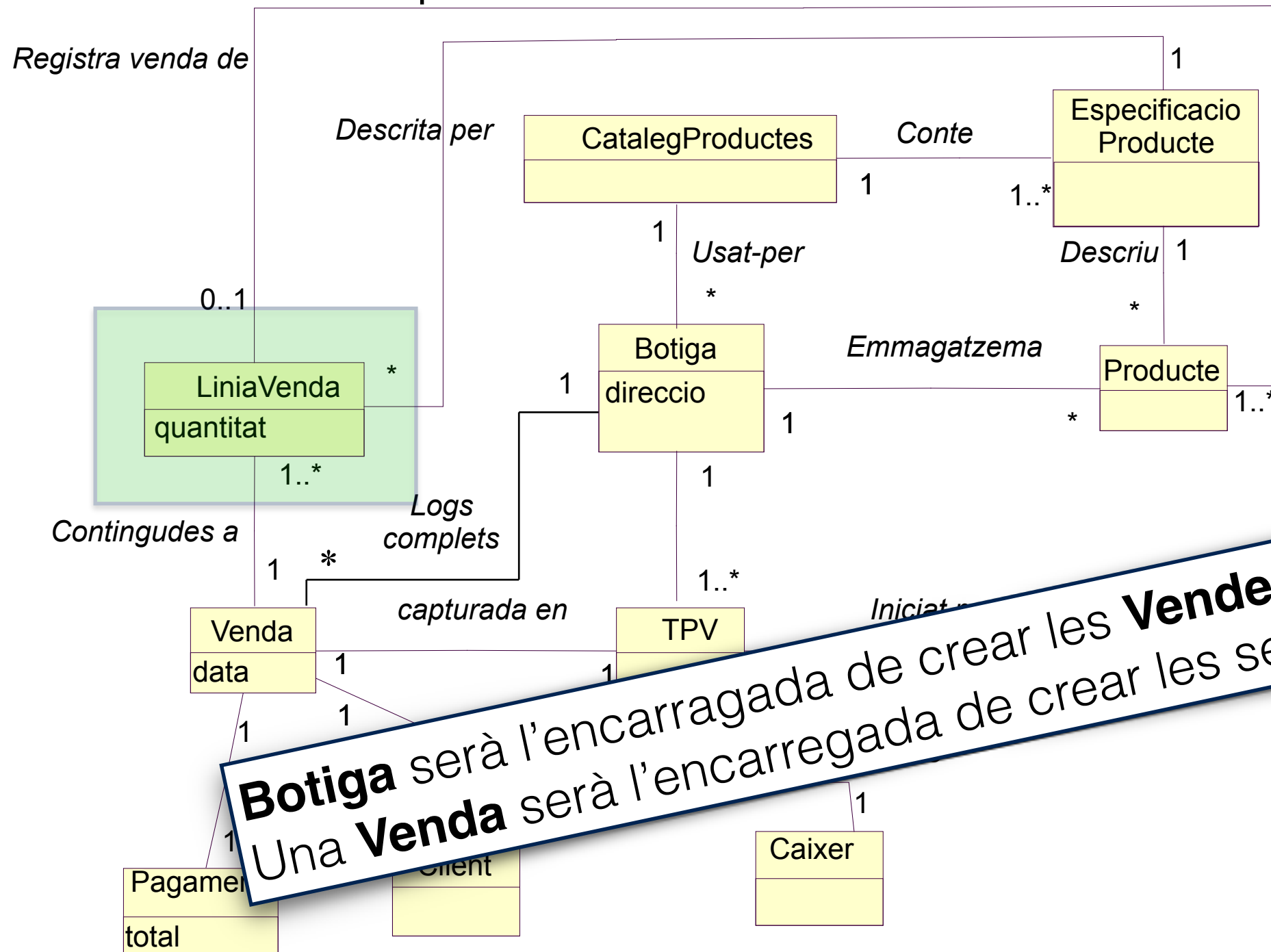
Exemple d'aplicació

- Qui és el responsable de crear les instàncies de *LiniaVenda*?



Exemple d'aplicació

- Qui és el responsable de crear les instàncies de *LiniaVenda*?



Botiga serà l'encarragada de crear les **Vendes**
Una **Venda** serà l'encarregada de crear les seves **Linia de Venda**

3.4.1. Patrons de disseny GRASP

Nom del patró: Creador

Context: Assignació de responsabilitats de creació d' objectes

Pros:

- Es manté encapsulament  Baix acoblament

Cons:

- Sovint la creació d'objectes requereix una complexitat significativa, per motius d'eficiència o d'altres. En aquests casos és raonable delegar la creació a una classe (per això s'utilitza el patró **Factory Template** o **Abstract Factory** (es veurà en la secció 3.4.2))
- Poden portar problemes d'acoblements alts trencant principis arquitectònics.

Patró Controller

Nom del patró: Controller

Context:

Els (sub)sistemes software reben esdeveniments que un cop interceptats algun objecte del sistema els ha de rebre i executar accions corresponents (*dispatcher*)

Problema:

Quina classe és el responsable de rebre un esdeveniment (**event**) i servir-lo?

Solució:

Assignar aquesta responsabilitat a un controlador.

Un **controlador** és un objecte d'una certa classe que:

- rep l'event i
- delega a un o més objectes del sistema el tractament de l'esdeveniment.

Patró Controller

Observacions:

- Un controlador **NO** és un objecte de la interfície
- El Controlador estableix el diàleg i interpretació dels errors del Model per a traduir-los a la Vista.

Pros:

- El Controlador és un Mediator entre la Vista i el Model. Desacobla funcionalitats

Cons:

- Single Responsibility Principle ✗
- Esdevenir una classe Deu (poca cohesió)
 - El Controller pot utilitzar diferents API's (o **façanes**) per a demanar serveis complexes del Model

3.4.1. Patrons de disseny GRASP

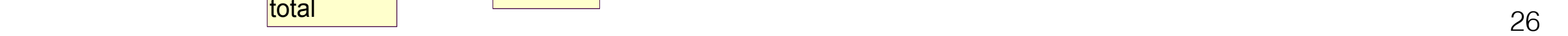
Inicialització de l'aplicació: creació de les instàncies principals de les classes de domini

- S'ha d'escollir un **objecte inicial de domini** (una classe prop de l'arrel de la jerarquia d'agregació) i invocar un mètode *create* d'aquell objecte que serà qui delegarà la creació de les instàncies de la resta d'objectes.
- Qui invoca la creació de l'objecte inicial de domini? Es crida des del programa principal o des del controlador?
- NOTA:
 - Si s'usen dades persistents (Bases de Dades, fitxers,...) potser el controlador serà l'encarregat de delegar a algun servei la càrrega de la Base de Dades o d'un fitxer (usant el patró **DAO**)

3.4.1. Patrons de disseny GRASP

Inicialització de l'aplicació: Si es segueix un Model-Vista-Controlador:

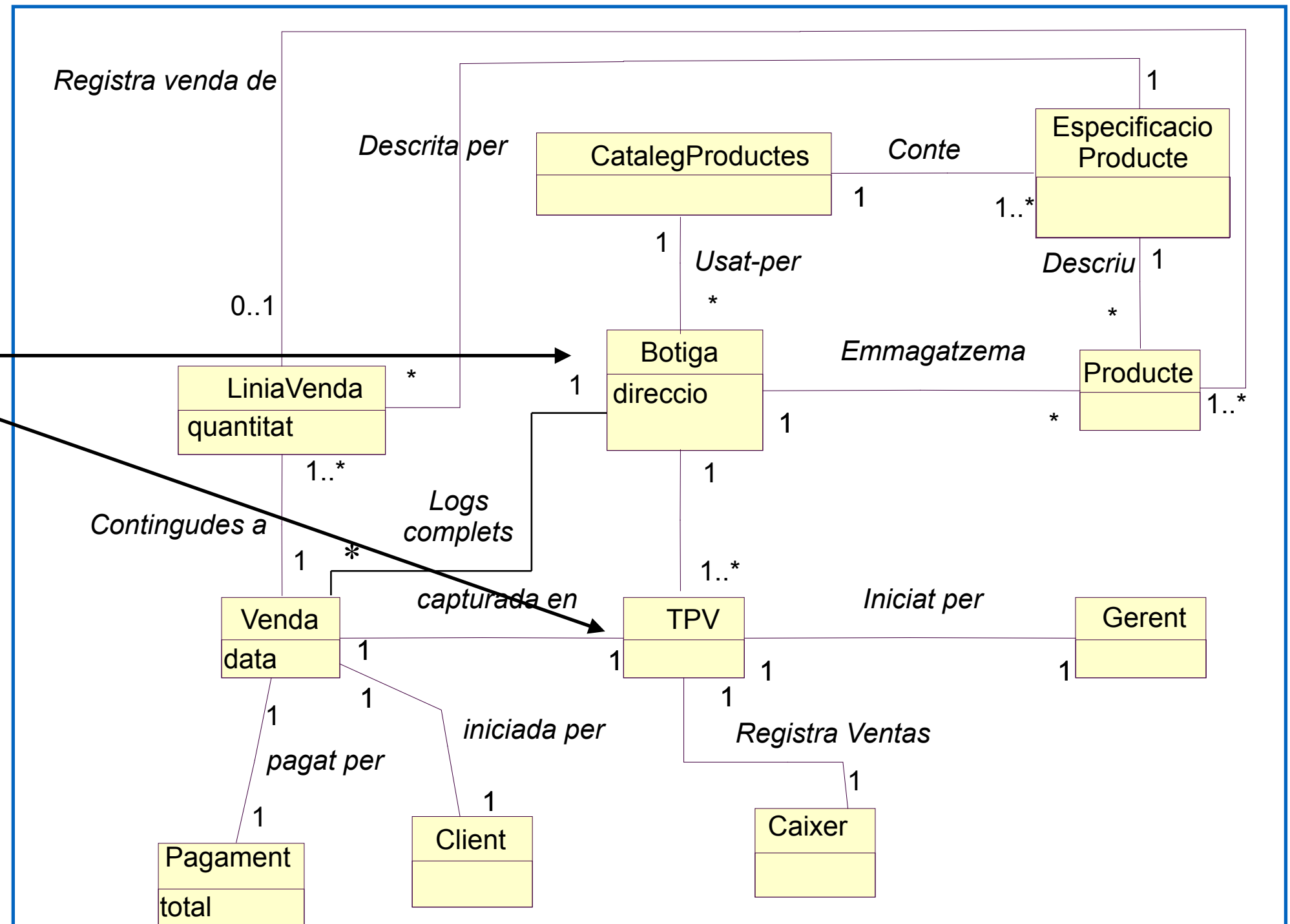
- Qui té la responsabilitat d'iniciar la creació les dades?
 - **Possibilitat 1:** L'**objecte del domini** que contingui més agregacions o que sigui l'arrel de l'aplicació
 - **Possibilitat 2:** L'objecte **controlador**
 - **Possibilitat 3:** L'objecte **façana** corresponent.
 - **Possibilitat 4:** La capa de **Persistència**



Exemple d'aplicació

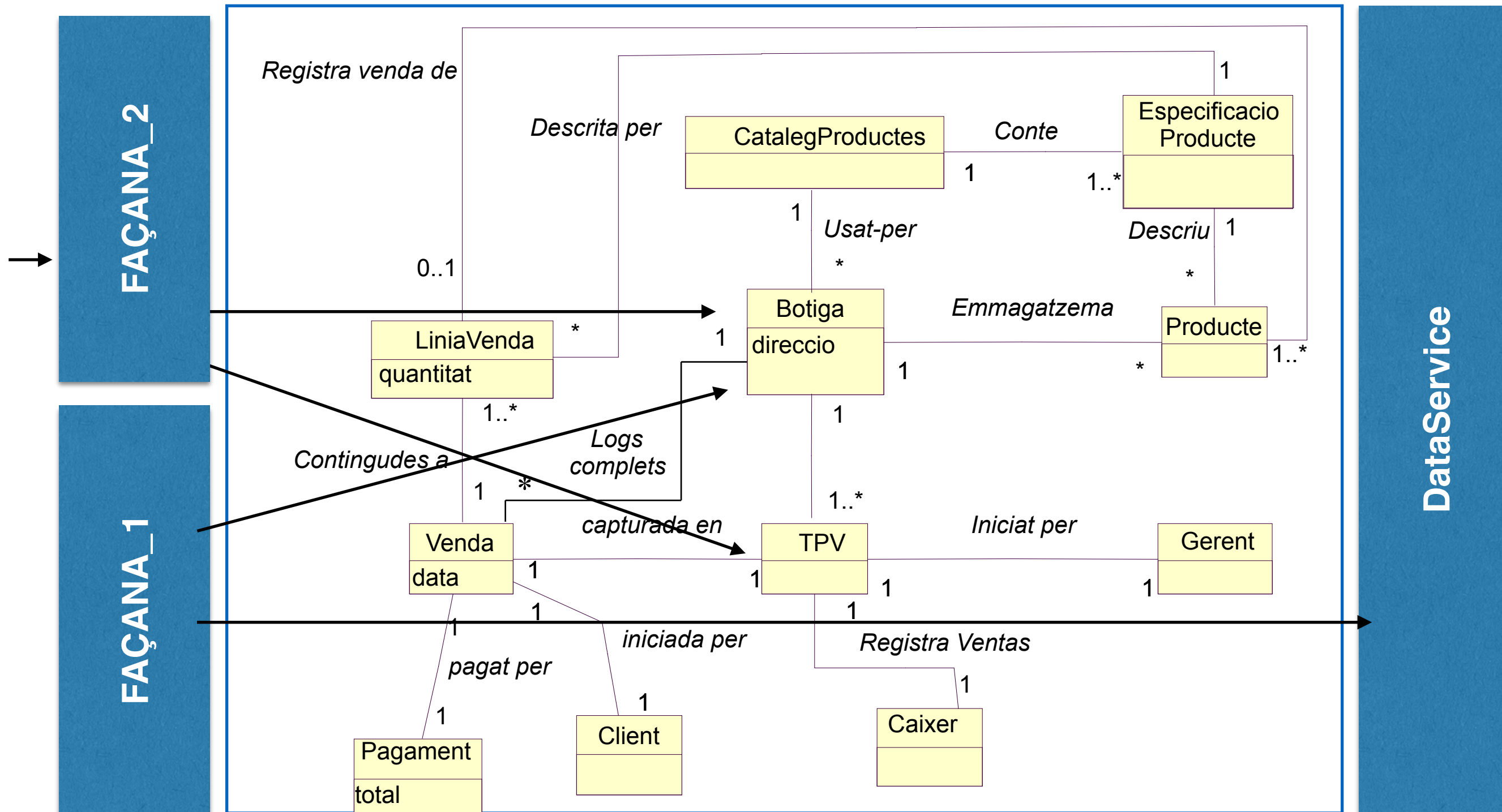
- Qui és l'objecte inicial de domini?

FAÇANA



Exemple d'aplicació

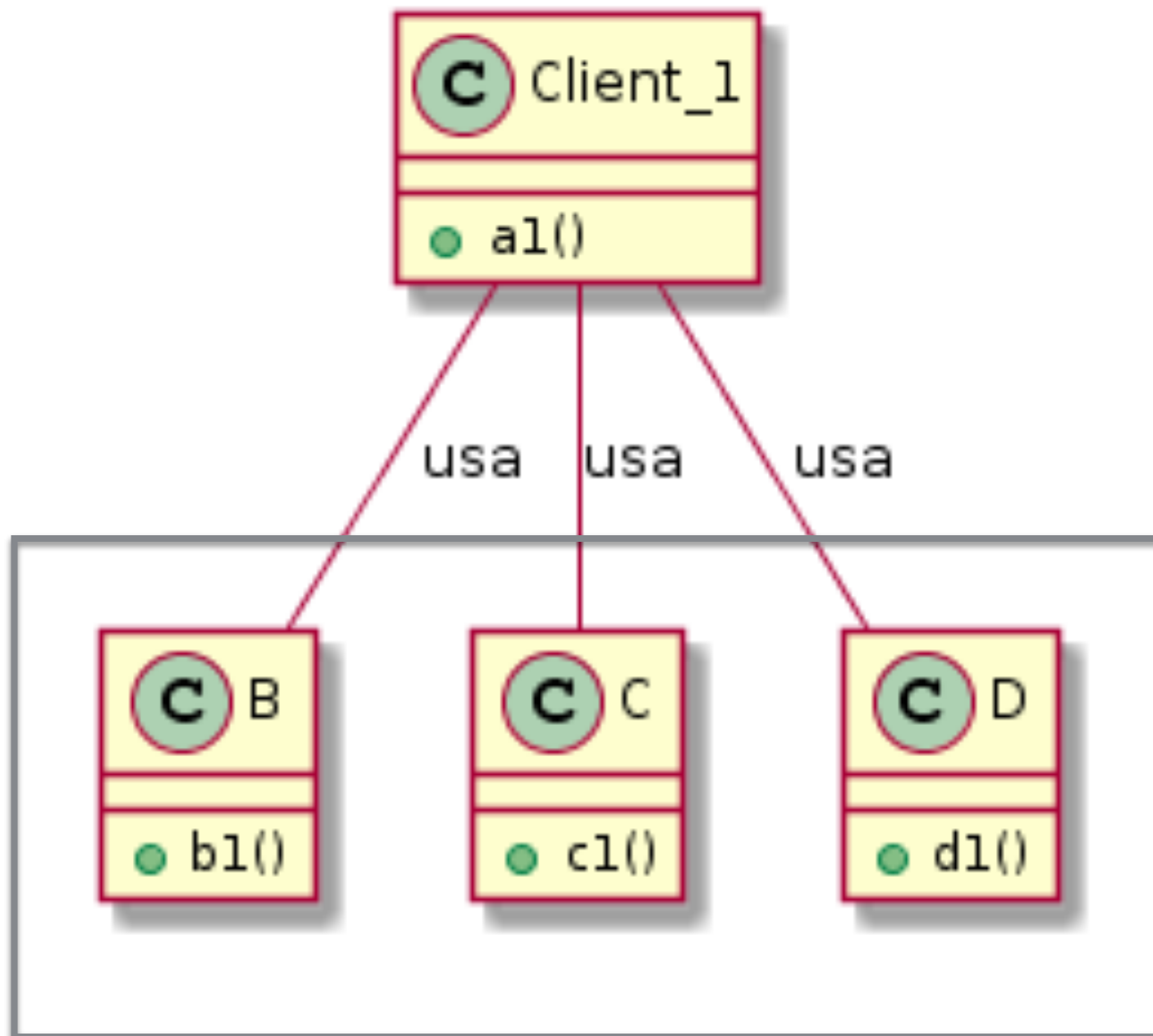
- Poden haver més d'una façana?



3.4. Patrons de disseny

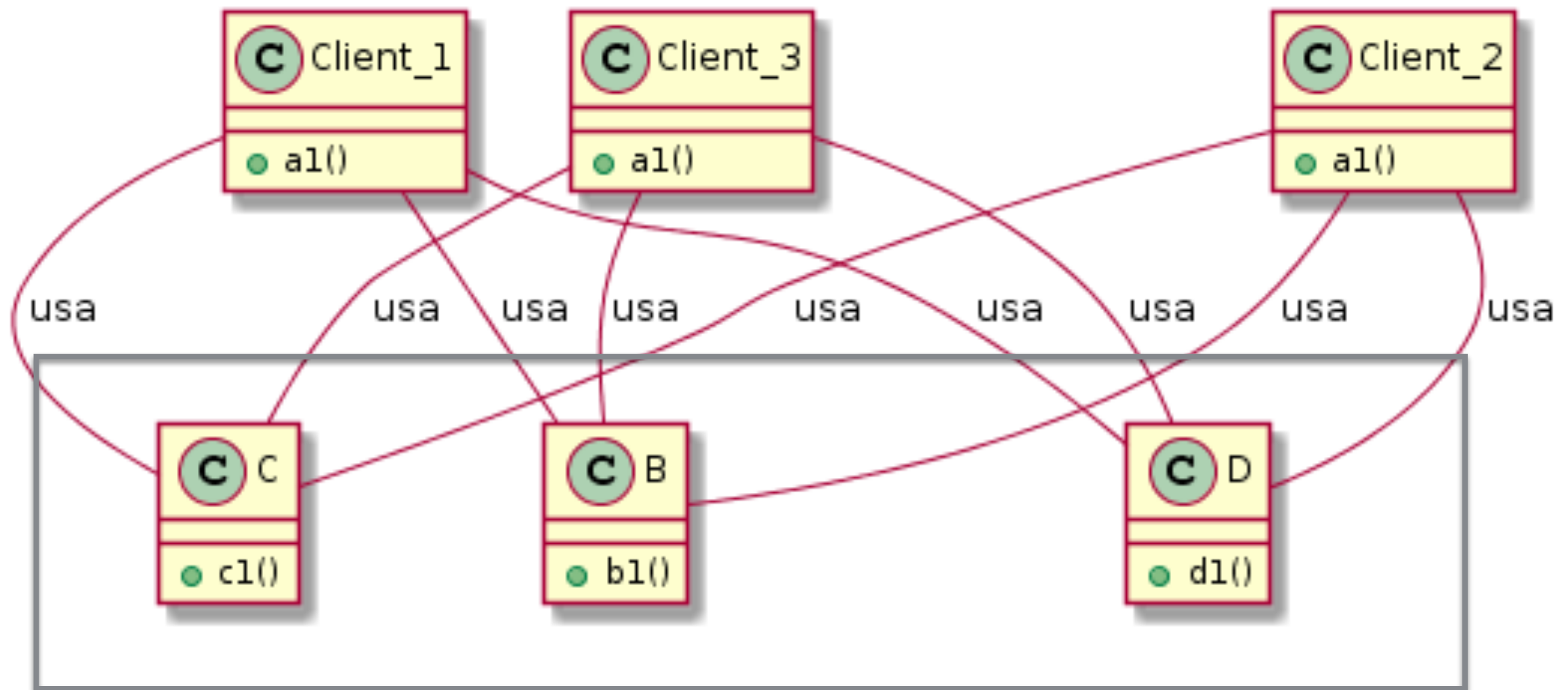
Propòsit → Àmbit ↓	CREACIÓ	ESTRUCTURA	COMPORTAMENT
CLASSE	<ul style="list-style-type: none"> • Factory method 	<ul style="list-style-type: none"> • class Adapter 	<ul style="list-style-type: none"> • Interpreter • Template method
OBJECTE	<ul style="list-style-type: none"> • Abstract Factory • Builder • Prototype • Singleton • Object pool 	<ul style="list-style-type: none"> • Object Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy 	<ul style="list-style-type: none"> • Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

3.4.2. Patrons de disseny: Patró Façana (Facade)



- La classe Client_1 ha de conèixer quina és la classe que exactament li proporciona el servei:
 - b1() es de B, c1() es de C, d1() es de D, etc.
- **Alt acoblament !!**

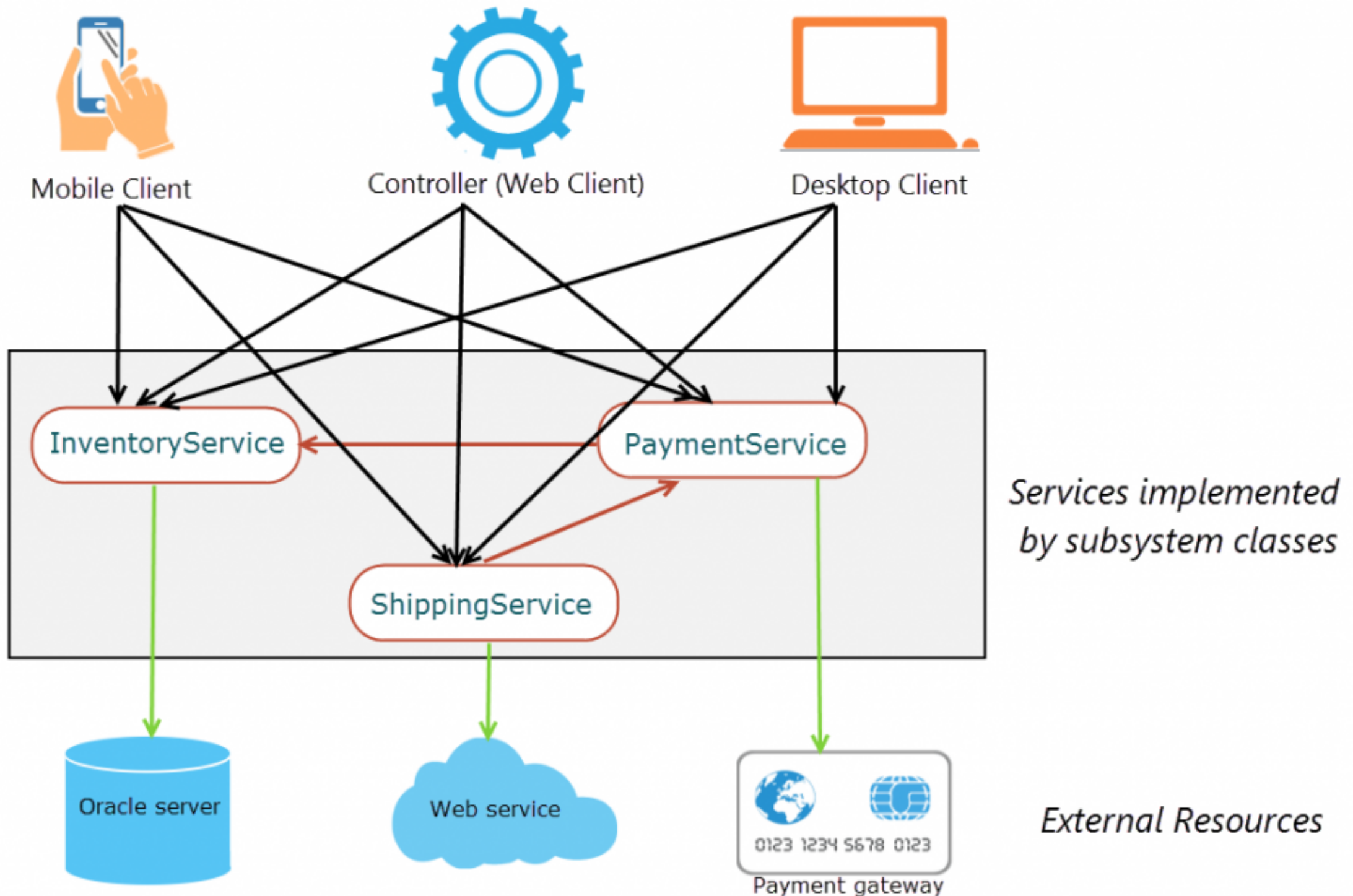
3.4.2. Patrons de disseny: Patró Façana (Facade)



PROBLEMA

A més a més, poden haver-hi moltes classes client

3.4.2. Patrons de disseny: Patró Façana (Facade)



Quin principi SOLID vulnera?

3.4.2. Patrons de disseny: Patró Façana (Facade)

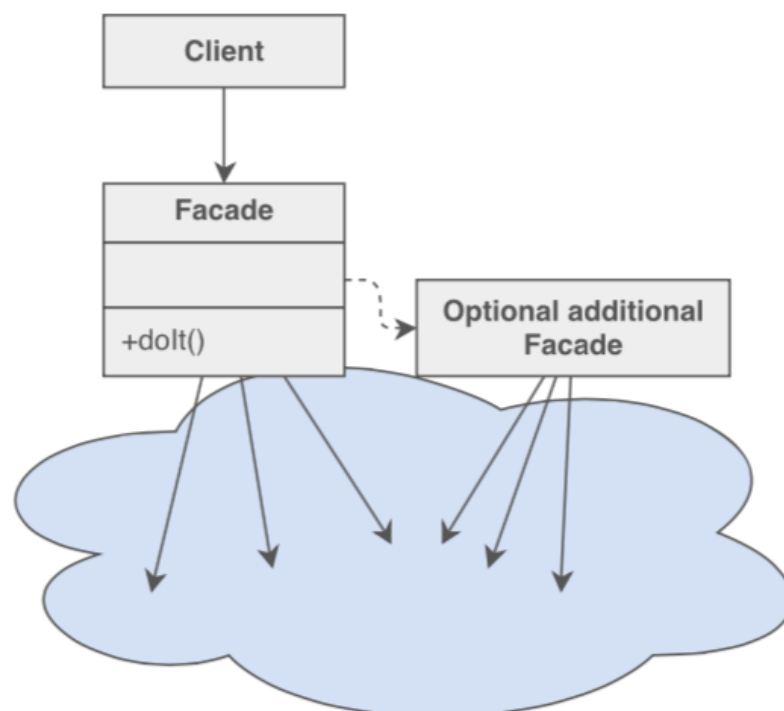
Nom del patró: **Façana** (*Facade*)

Context:

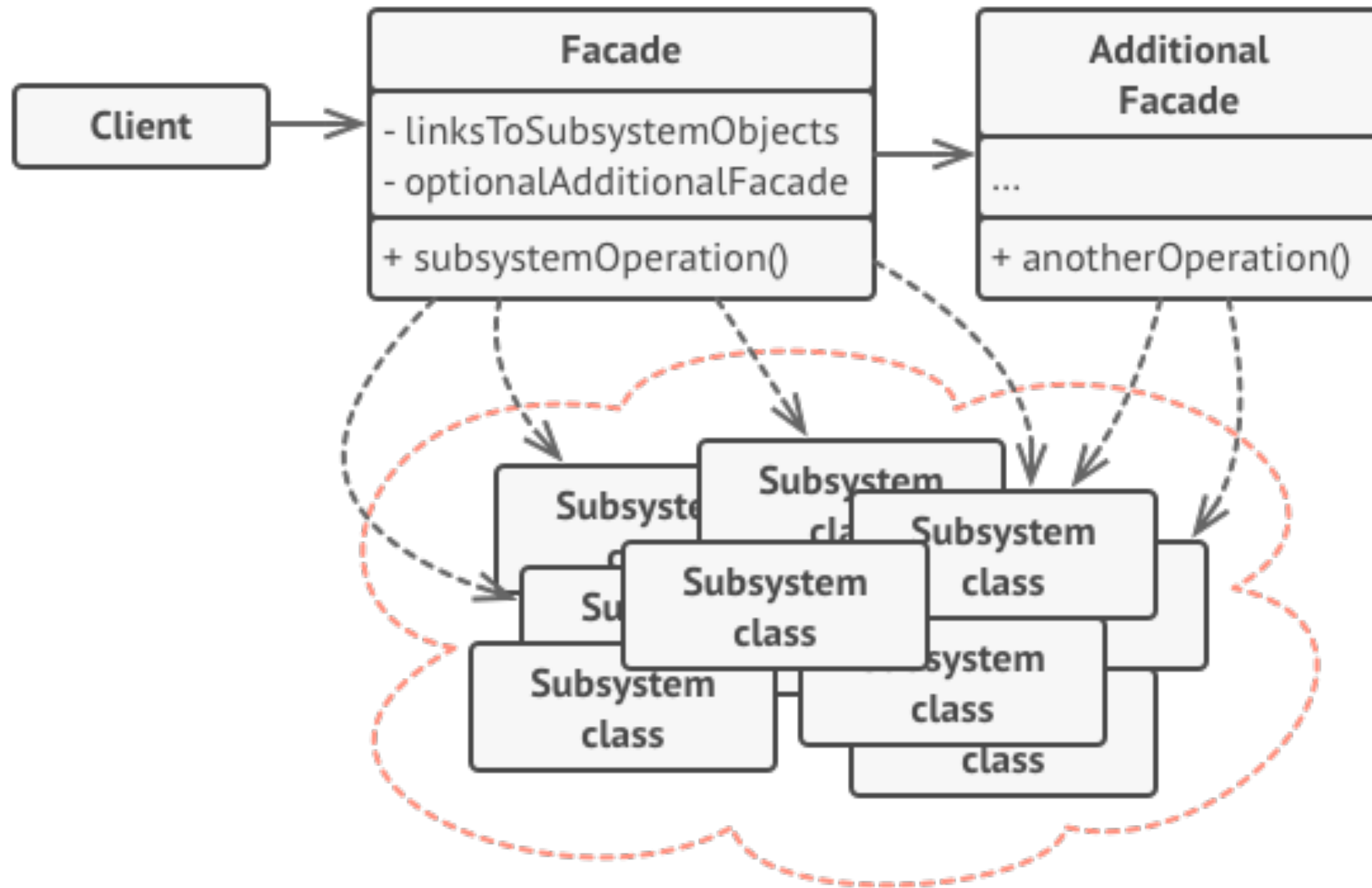
Subsistema amb moltes utilitats diferents

Solució:

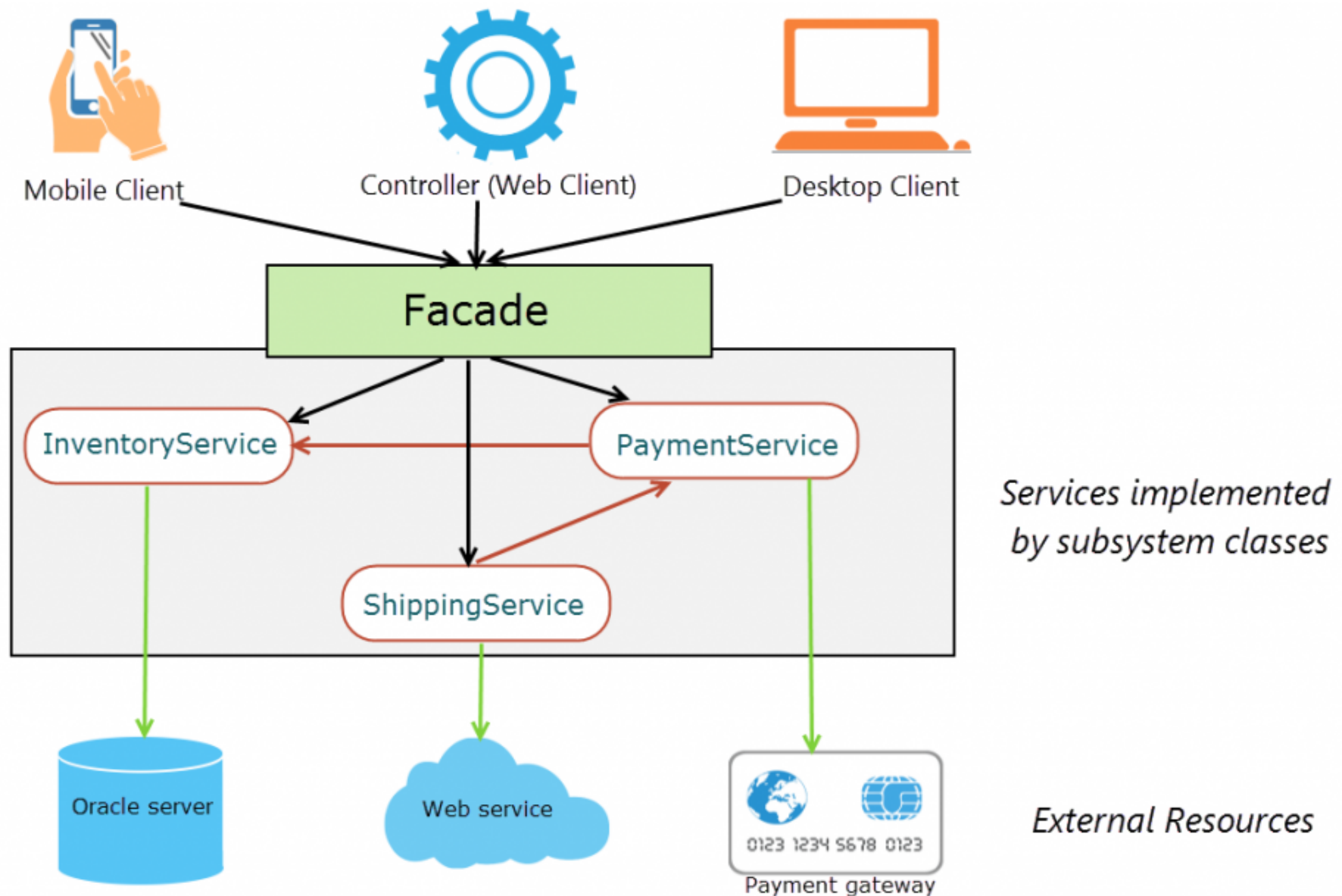
Proporciona una interfície unificada a un conjunt d'utilitats d'un subsistema complex. Redueix la corba d'aprenentatge necessària per aprofitar amb èxit el subsistema.



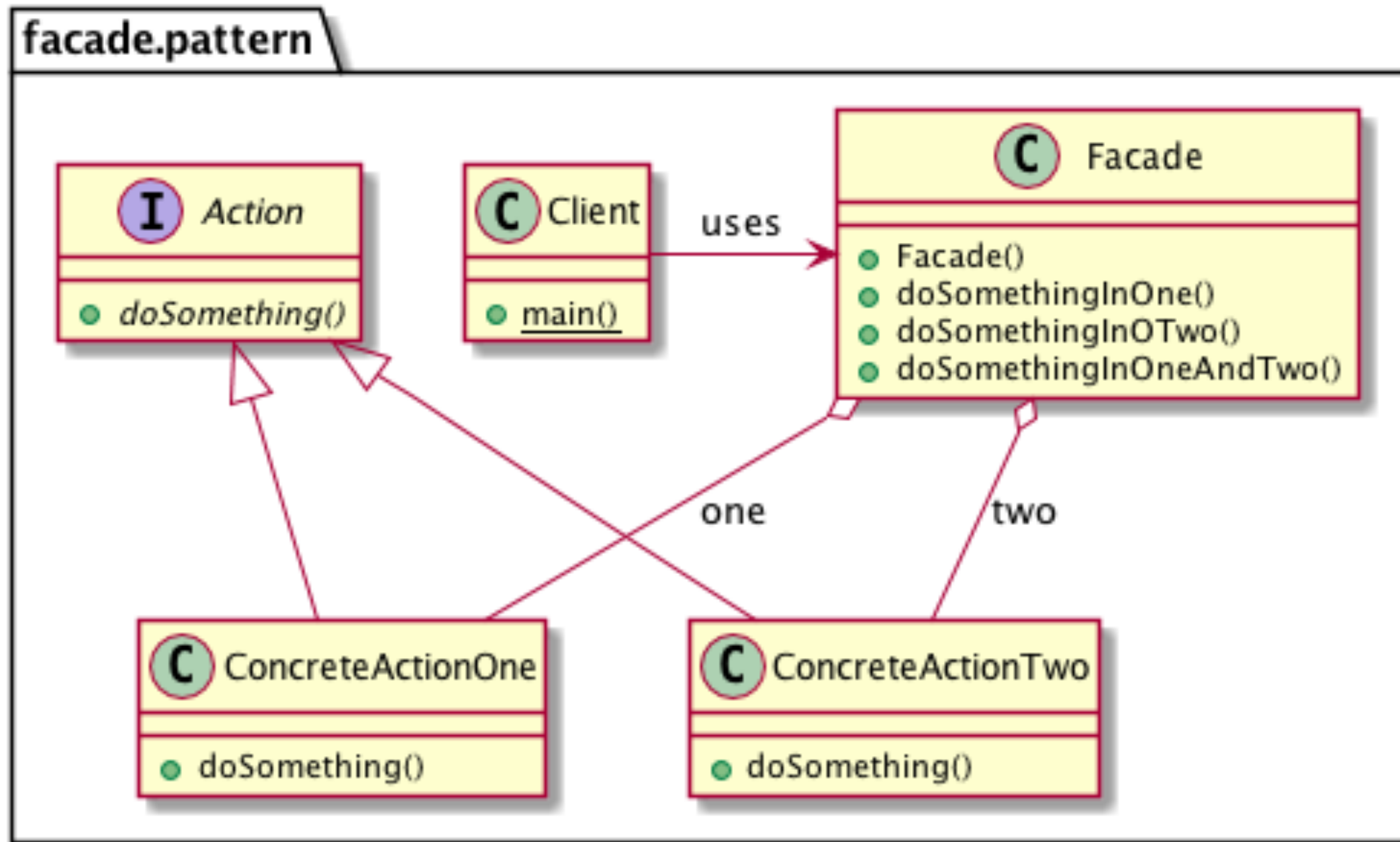
3.4.2. Patrons de disseny: Patró Façana (Facade)



3.4.2. Patrons de disseny: Patró Façana (Facade)



3.4.2. Patrons de disseny: Patró Façana (Facade)



3.4.2. Patrons de disseny: Patró Façana (Facade)

```
public class Facade {  
  
    private ConcreteActionOne one;  
    private ConcreteActionTwo two;  
  
    public Facade() {  
        System.out.println("This is the FACADE pattern...");  
        this.one = new ConcreteActionOne();  
        this.two = new ConcreteActionTwo();  
    }  
  
    public void doSomethingInOne() {  
        System.out.println("Calling doSomething in action ONE:");  
        one.doSomething();  
    }  
  
    public void doSomethingInTwo() {  
        System.out.println("Calling doSomething in action TWO:");  
        two.doSomething();  
    }  
  
    public void doSomethingInOneAndTwo() {  
        System.out.println("Calling doSomething in action ONE and TWO:");  
        one.doSomething();  
        two.doSomething();  
    }  
}
```

3.4.2. Patrons de disseny: Patró Façana (Facade)

Conseqüència

- Simplifica l'accés a un conjunt de classes proporcionant una única classe que tots utilitzen per comunicar-se amb el conjunt de classes

Pros:

- Les aplicacions client no necessiten conèixer les classes que hi ha darrera la classe FACADE
- Es poden canviar les classes “ocultes” sense necessitat de canviar els clients. Només s'ha de fer els canvis necessaris a FACADE
- Es minimitzen les comunicacions i dependències entre subsistemes

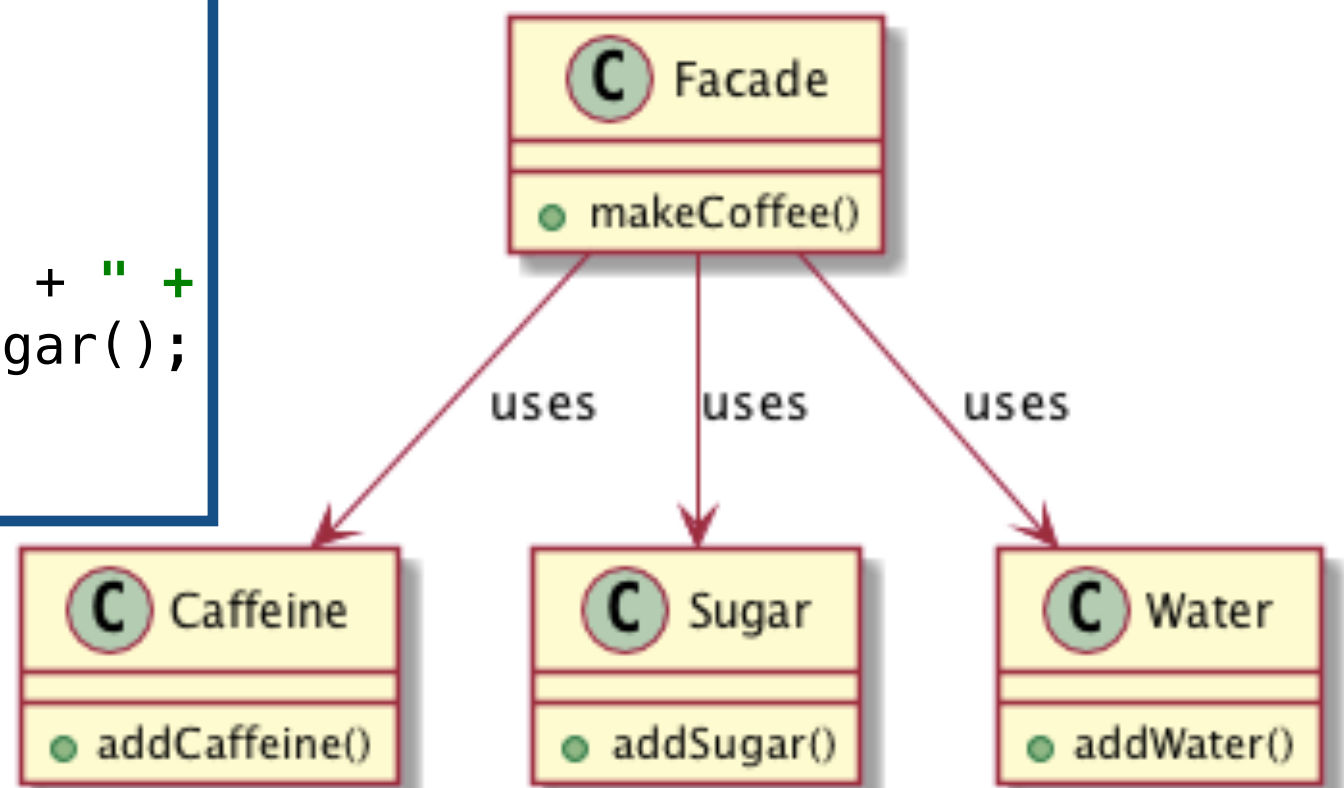
Cons:

- Dóna funcionalitats més limitades que les que realment té el subsistema
- Pot esdevenir un objecte Deu

3.4.2. Patrons de disseny: Patró Façana (Facade)

```
class Facade {  
  
    public String makeCoffee() {  
        Sugar s = new Sugar();  
        Caffeine c = new Caffeine();  
        Water w = new Water();  
  
        return "Coffee = " + w.addWater() + " +  
" + c.addCaffeine() + " + " + s.addSugar();  
    }  
}
```

FACADE's Class Diagram



3.4.2. Patrons de disseny: Patró Façana (Facade)

- Reducció de l'acoblament client-subsistema
 - Es pot reduir l'acoblament fent que la **façana** sigui una **classe abstracta** amb subclasses concretes per les diferents implementacions del subsistema. Els clients es comuniquen amb el subsistema utilitzant la interfície de la classe façana abstracta (patró **Adapter**)
 - Una altra possibilitat és configurar l'objecte façana amb **diferents objectes abstractes del subsistema**. Per personalitzar la façana només cal canviar un o varis objectes del subsistema
 - El patró **Abstract Factory** es pot utilitzar junt amb la **Facade** per crear objectes del subsistema de manera independent
- Classes del subsistema privades i públiques
 - En Java es pot usar els paquets per determinar les classes que són visibles fora o que no seran visibles.

Exemple: <https://springframework.guru/gang-of-four-design-patterns/facade-pattern/>