Universitat de Barcelona

Tema 4 **Model de disseny**

Maria Salamó Llorente

Disseny de Software

Enginyeria Informàtica

Facultat de Matemàtiques, Universitat de Barcelona

Universitat de Barcelona

Contingut

PART 2. Dissenyar amb patrons

4.3.a Introducció als patrons

4.3.b Patrons generals d'assignació de responsabilitats en software

- Responsabilitats
- Patrons:
  1) Expert en informació
  2) Creador
  3) Alta cohesió
  4) Baix acoblament
  5) Controlador

Universitat de Barcelona

4.3.a Introducció als Patrons

Introducció als patrons

Dissenyar software orientat a objectes és difícil, i dissenyar software OO **reutilitzable** encara més difícil

- Capítol 1. Introduction. Design patterns, Gang of Four

... i un software que sigui capaç d'evolucionar ha de ser reutilitzable (almenys per les versions futures)

A més, no és bona idea resoldre tots els problemes des de zero, normalment és millor **tractar de reutilitzar solucions** (conceptes) que **ja han funcionat en el passat**

Introducció als patrons

- Els dissenyadors experts no resolen els problemes des de l'inici, reutilitzen solucions que han usat en el passat
- A més a més,
  - El software canvia
  - Per anticipar-se als canvis en els requisits s'ha de dissenyar pensant en quins aspectes poden canviar

- Els patrons de disseny estan orientats al canvi

Introducció als patrons

Com ser un mestre jugador d'escacs?

- Aprendre les regles
  - Nom de les peces, moviments legals, geometría i orientació al tauler,etc.
- A continuació aprendre els principis
  - Valor de les peces, valor estratègic de les caselles centrals, etc.
- Per arribar a ser un mestre, cal estudiar els dissenys d'altres mestres
  - Aquestes partides contenen patrons que han de ser entesos, memoritzats i aplicats repetidament
- Hi ha centenars de patrons

Introducció als patrons

Com ser un mestre del software?

- Aprendre les regles
  - Algorismes, estructures de dades, llenguatges de programació, etc.
- A continuació aprendre els principis
  - Programació estructurada, programació modular, programació OO, programació genèrica, etc.
- Per arribar a ser un mestre, cal estudiar els dissenys d'altres mestres
  - Aquests dissenys contenen patrons que han de ser entesos, memoritzats i aplicats repetidament
- Hi ha centenars de patrons

Definicions de patró

- (Gamma) Un patró de disseny és una descripció de classes i objectes comunicant-se entre sí adaptada per resoldre un problema de disseny general en un context particular
- **Un patró és**:
  - Una solució a un problema en un context particular
  - Recurrent (la solució és rellevant en altres situacions)
  - Ensenya (permet entendre com adaptar-ho a la variant particular del problema on es vol aplicar)
  - Té un nom per referirse al patró

Universitat de Barcelona

Motivació dels patrons

- Capturen l'experiència i la fan accessible als no experts
- El conjunt dels seus noms forma un vocabulari que ajuda als desenvolupadors a que es comuniquin millor
- Ajuden a la gent a comprendre un sistema més ràpidament quan està documentat quins patrons usa
- Faciliten la reestructuració d'un sistema tant si va ser o no va ser concebut amb patrons en ment

Com es descriu un patró

- **Nom del patró**: com es diu el patró? És una forma general de descriure un problema de disseny, les seves solucions i les conseqüències en una o dues paraules (Comunicació/Vocabulari tècnic)
- **Descripció del problema:** a resoldre (o contexte) Una descripció de quan (en quins casos) aplicar el patró i quan no s'ha d'aplicar. Ha d'explicar amb més detall el problema i el seu contexte
- **Solució:** Descriu els elements de disseny que contitueixen la solució (no és un disseny o implemenció concreta per un cas particular, és com una plantilla que es pot aplicar a molts casos particulars)
- **Conseqüències** Són els resultats d'aplicar el patró. Descriu avantatges i desavantatges

Tipus de patrons

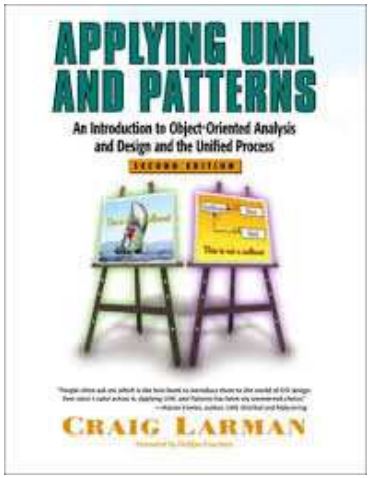
- **Generals** – Grand
  - Delegació, Interface, Inmutable, Marker Interface, Proxy
- **Estructurals** – Gamma i Grand
  - Adapter, Iterator, Bridge, Facade, Flyweight, Virtual Proxy, etc.
- **Creació** – Gamma i Grand
  - Factory method, Abstract factory, Builder, Prototype, Singleton, Object Pool, etc.
- **Particionament** – Gamma i Grand
  - Layered Initialization, Filter, Composite

Tipus de patrons

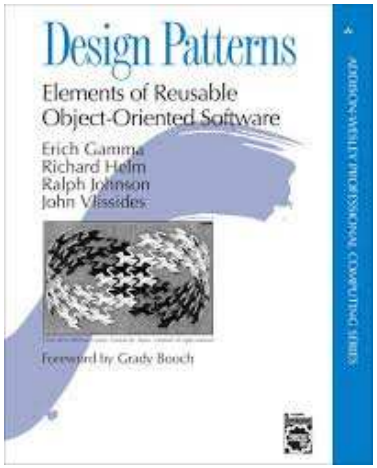
- **Comportament** – Gamma i Grand
  - Command, Mediator, Observer, State, Strategy, Visitor
- **Concurrència** – Grand
  - Single Thread Execution, Guarded Suspension, Balking, Read/Write Block, Produce-Consumer
- **Assignació de responsabilitats** – Larman
  - GRASP (són 10 patrons)

Buschmann	17 patrons
Gamma	23 patrons
Grand	41 patrons
Larman	10 patrons

Introducció als patrons



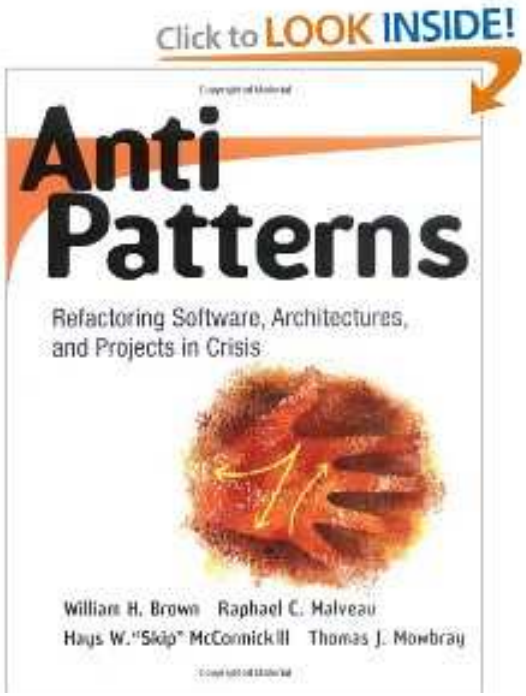
- Craig Larman. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.**



- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. **Design Patterns: Elements of Reusable Object-Oriented Software**. *Addison-Wesley*.

  S'anomenen Gang of Four (GoF), descrit a la dècada dels 90 i descriu en detall 23 patrons de disseny

Introducció als patrons



William J. Brown,
Raphael C. Malveau,
Hays W. McCormick,
Thomas J. Mowbray
Wiley
1ra. Edició

Si el concepte de patrons **(bones solucions a problemes coneguts)** esulta interessant, potser ıcara és més interessant el concepte d'**anti-patró (errors comuns solucionant problemes coneguts)**

Universitat de Barcelona

4.3.b Patrons generals d'assignació de responsabilitats en el software

Patrons GRASP

Introducció

- **GRASP**: descripció dels principis bàsics d'assignació de responsabilitats a les classes expressades com a patrons
- Distribuir responsabilitats és la part més difícil del disseny OO. Consumeix una bona part del temps
- Patrons GRASP:

Expert	***Creador***
Baix Acoblament	***Alta Cohesió***
Controlador	*Polimorfisme*
Indirecció	*Variacions Protegides*

En què consisteix la responsabilitat en el software?

- **Responsabilitat**: obligació d'un objecte en termes del seu comportament
- Poden ser de dos tipus:
  - De **realització**:
    - Fer alguna cosa ell mateix tal com crear un objecte o realitzar un càlcul.
    - Iniciar una acció en altres objectes
    - Controlar i coordinar les activitats d'altres objectes
  - De **coneixement**:
    - Conèixer les dades privades encapsulades
    - Conèixer els objectes relacionats
    - Conèixer les coses que pot derivar o calcular

Responsabilitats

- Exemples:
  - Una venda és responsable de crear LiniesDeVenda
  - Una venda és responsable de conèixer el seu total

- Les responsabilitats s'implementen usant mètodes que actuen en solitari o en col·laboració amb altres mètodes i objectes per acomplir-les.
  - Per exemple, una venda pot acomplir la responsabilitat de conèixer el seu total mitjançant un mètode *getTotal*, que a la seva vegada crida a les línies de venda per sol·licitar el subtotal de cadascuna

Patrons en GRASP

- Un **patró** és una descripció nominativa d'un problema i una solució que pot ser aplicada en un nou context, juntament amb consells sobre com aplicar-la en noves situacions i una discussió dels possibles compromisos

- Molts patrons ofereixen una guia sobre com **assignar les responsabilitats als objectes** en la solució d'un problema

- És important recordar els noms perquè ajuda a la comunicació

Exemple de patró GRASP

- **Nom**: Expert en la informació
- **Descripció del problema**: Quin pot ser el principi bàsic mitjançant el qual assignarem responsabilitats als objectes?
- **Solució**: Assigna una responsabilitat a la classe que té la informació necessària per satisfer-la

Del model de domini al model de disseny

- Les classes per aplicar un patró (com per exemple *Expert en la informació*) han de ser classes de disseny.
  - Si el **model de disseny** conté un *Expert en la informació* per la responsabilitat que es pretén assignar, l'usarem.
  - Sinó, buscarem en el **model de domini** per convertir una classe conceptual en una classe de disseny

Patrons GRASP

Els **patrons GRASP** (*General Responsibility Assignment Software Patterns*) descriuen principis fonamentals del disseny orientat a objecte i de l'assignació de responsabilitats.

En aquest curs veurem:

1. Expert en la informació
2. Creador
3. Baix acoblament
4. Alta cohesió
5. Controlador (Controller)

L'aplicació de qualsevol patró GRASP permet refinar el disseny

1. Expert en la informació

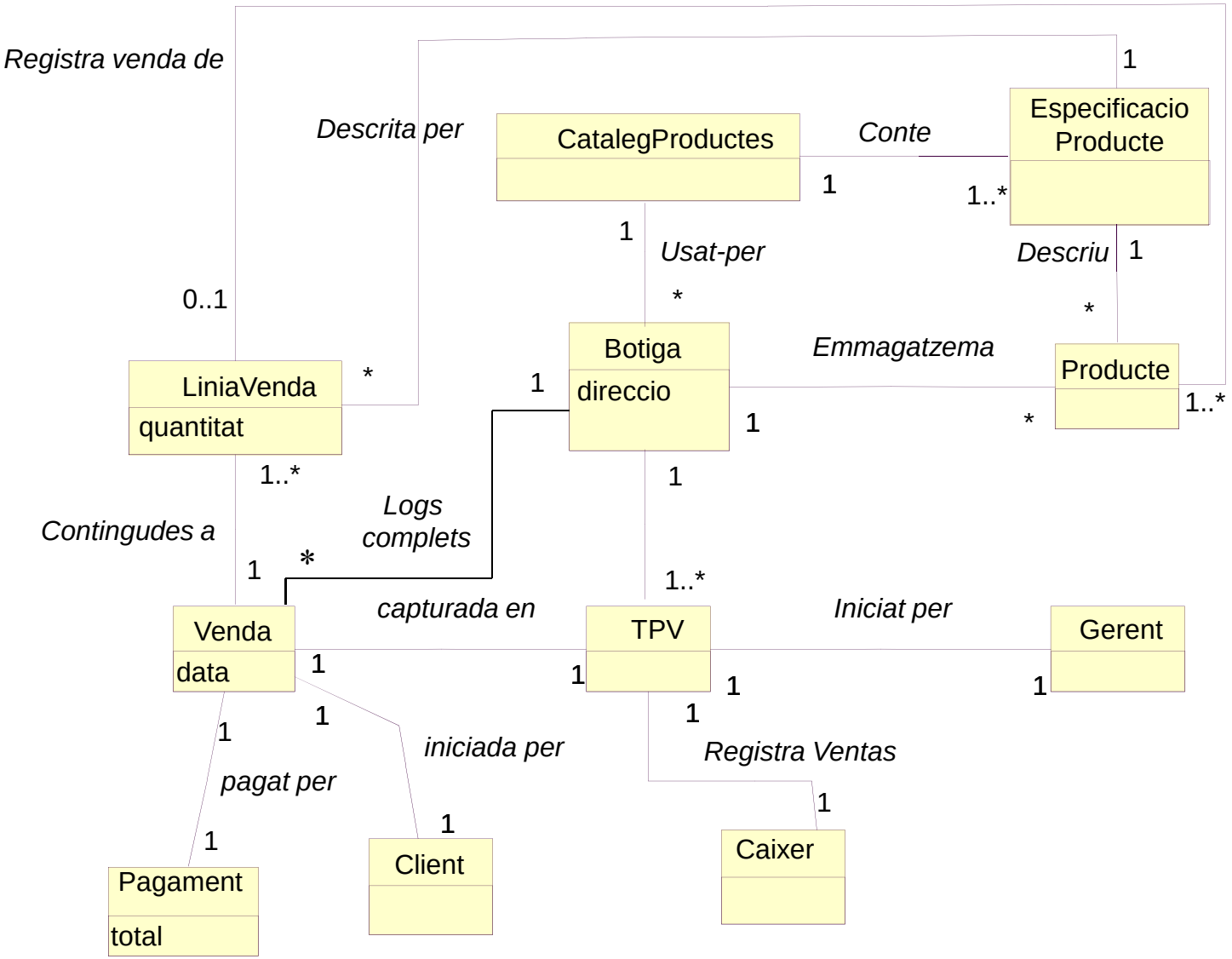
- **Descripció del problema**: Quin pot ser el principi bàsic per l'assignació de responsabilitats a objectes?
  - Una bona assignació facilita el manteniment, la eficiència, la comprensió, ...
- **Solució**: assignar una responsabilitat a *l'Expert en la informació*: la classe que té la informació necessària per fer la responsabilitat

1. Expert en la informació

- Cada objecte és responsable de mantenir la seva propia informació (principi d'encapsulament)
  - Coneix i pot informar el valor dels seus atributs
  - Pot modificar el valor dels seus atributs

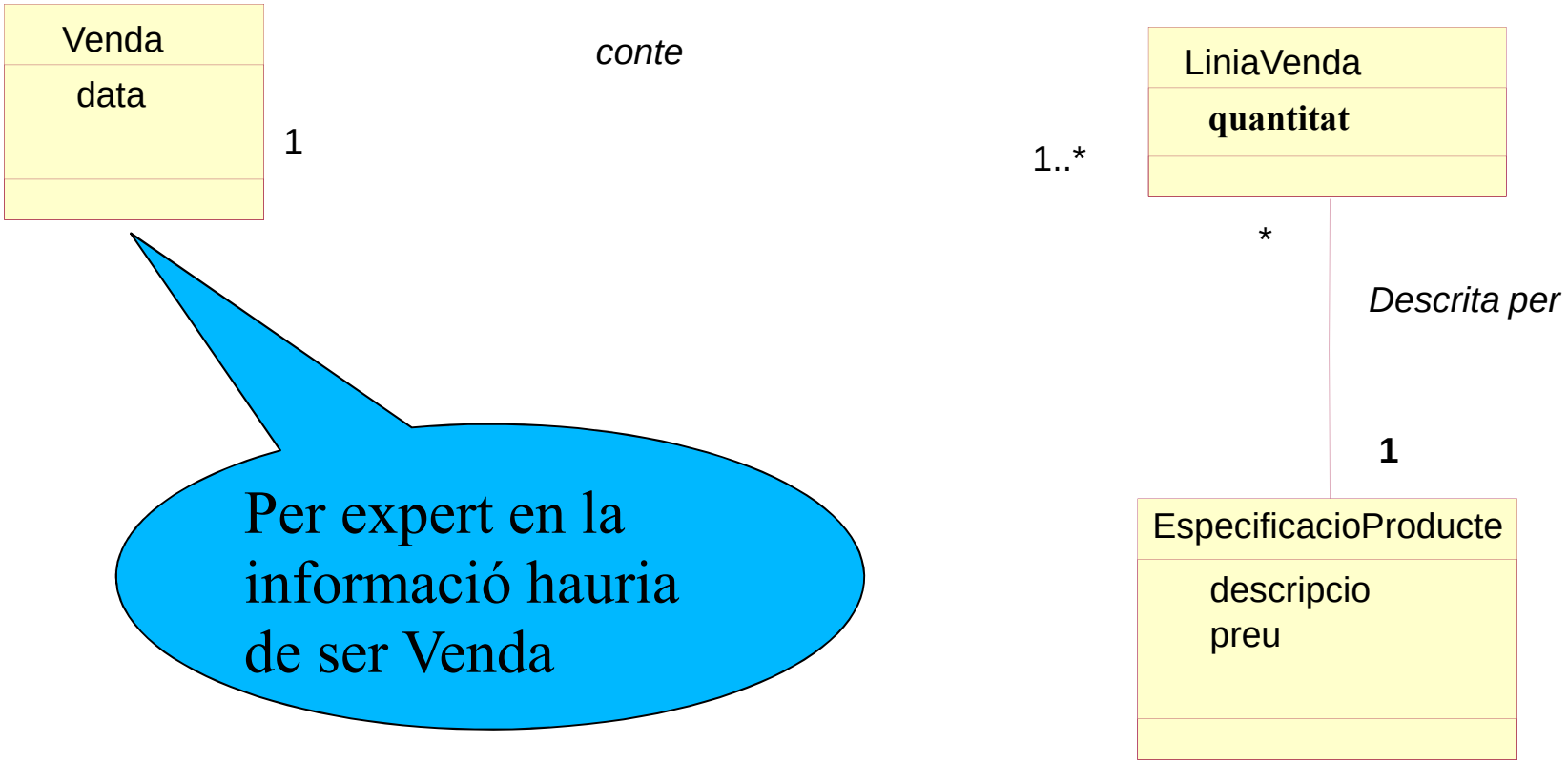
- Si té relacions de composició (agregació forta) amb altres objectes (les seves parts) també serà el responsable de conèixer la informació d'ells, de crear-los (**patró creador**) i de delegar-li les seves operacions

Exemple model domini TPV



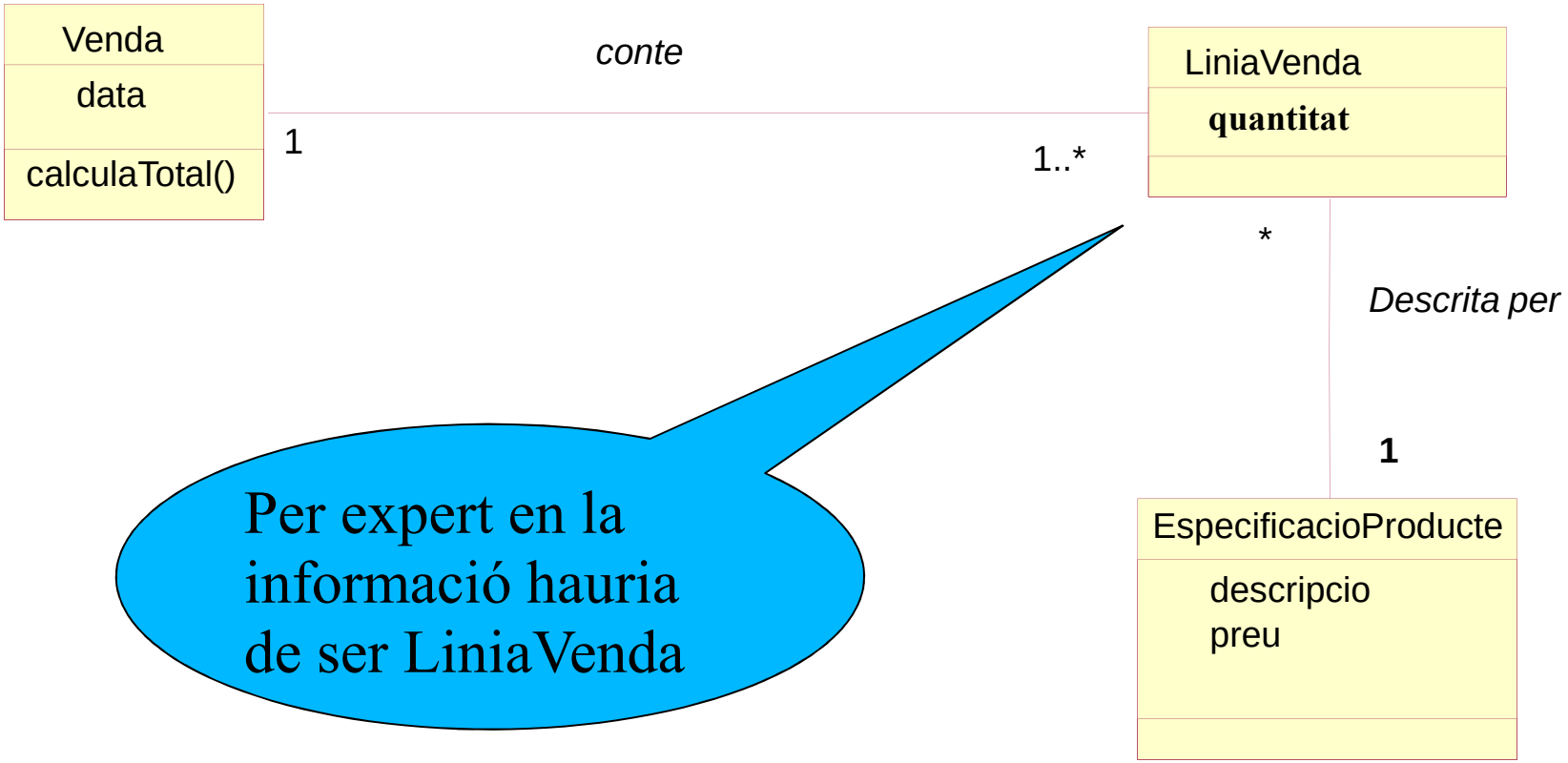
Exemple d'aplicació

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



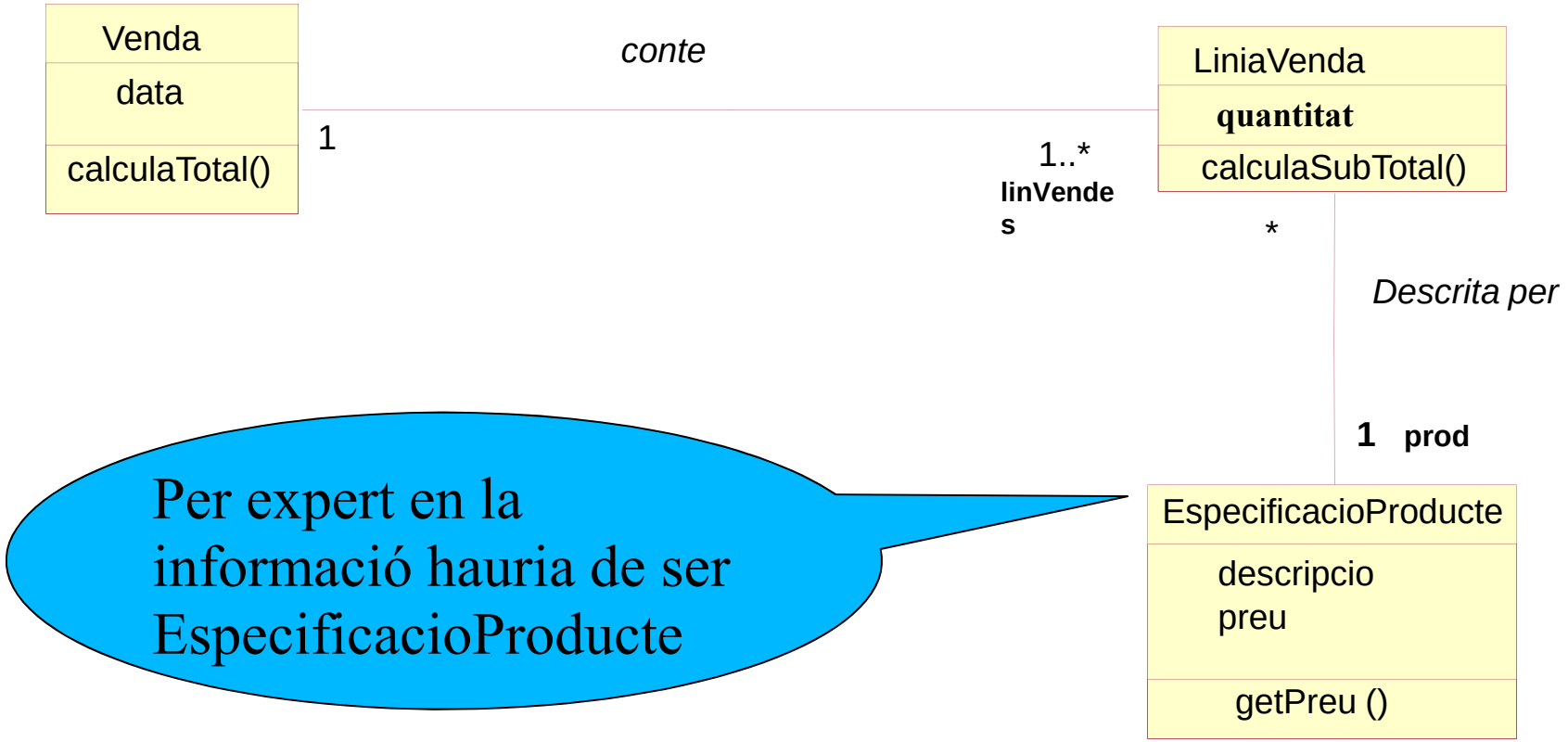
Exemple d'aplicació

- 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?



Exemple vist des d'un DS

(es farà a classe)

Exemple no seguint l'expert en la informació

(es farà a classe)

Contraindicacions

- Hi ha situacions en que l'aplicació d'*Expert en la informació* no porta a bons dissenys,
  - usualment perquè augmenta l'acoblament i redueix la cohesió. Es trenquen principis arquitectònics (p.e., arquitectura a 3 capes)
- Exemples:
  - Interfície d'usuari: imprimir 7 llistats de vendes. Mostrar les línies de venda en una finestra
  - Persistència: Emmagatzemar una venda a la base de dades

Beneficis i patrons relacionats

Beneficis:

- Es manté **l'encapsulació**, ja que els objectes usen la seva pròpia informació per a realitzar les tasques.
  - Habitualment, això també proporciona **baix acoblament**, produint sistemes més **robustos** i mantenibles
- El comportament es distribueix al llarg de les classes que tenen la informació requerida
  - proporcionant definicions de classes "lleugeres" (amb major cohesió), que són més fàcils d'entendre i mantenir.

- **Patrons relacionats**: Baix acoblament, alta cohesió

2. Creador

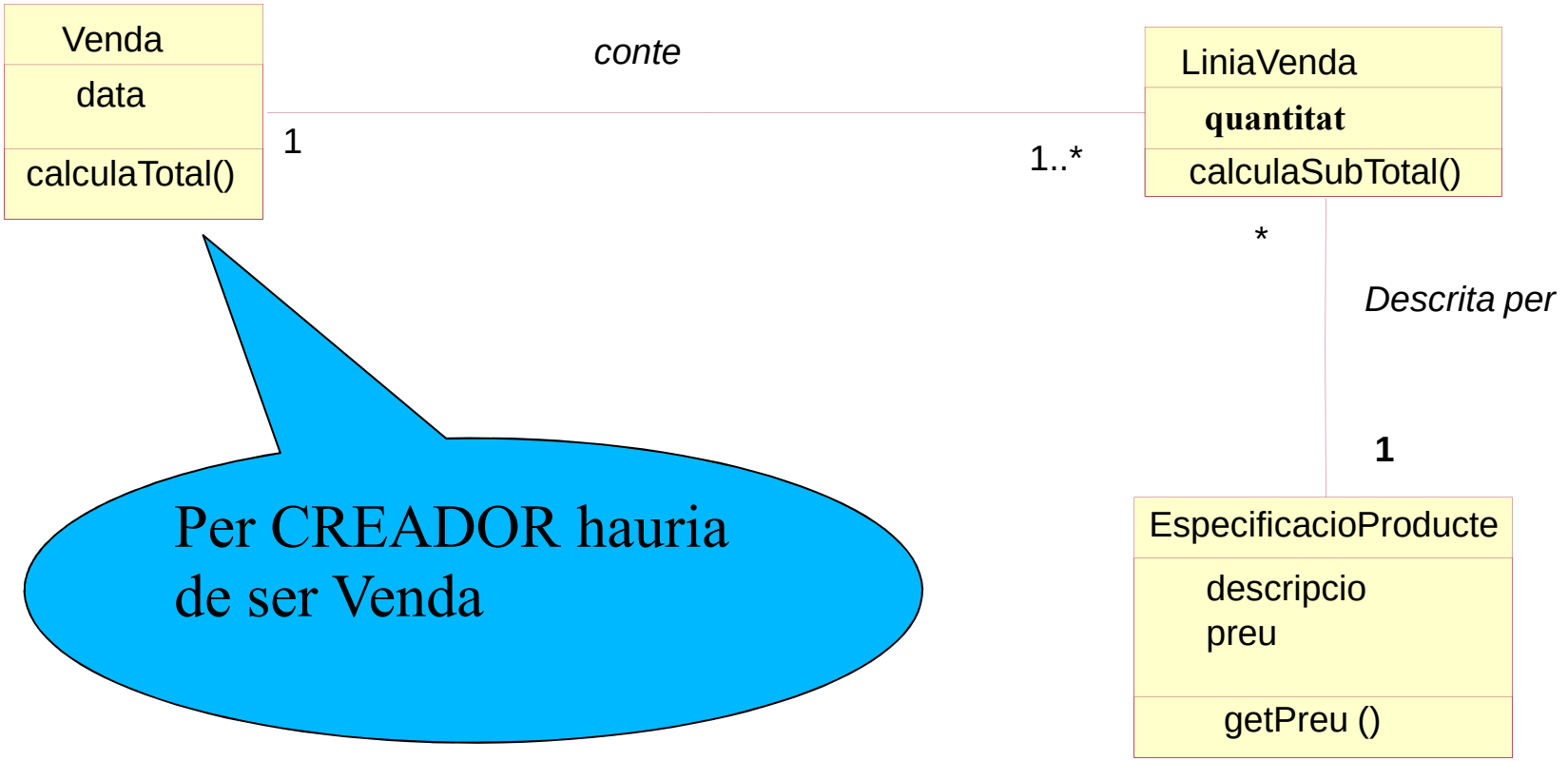
- **Descripció del problema**: Qui hauria de ser el responsable de crear una nova instància d'una classe?
- **Solució**: assignar a una classe **B** la responsabilitat de crear una instància d'**A** si es compleixen **una o més** de les següents condicions:
  - **B** *agrega* objectes de classe **A**
  - **B** *conté* objectes de classe **A**
  - **B** *manté un registre* d'objectes de classe **A**
  - **B** *fa un us exhaustiu* objectes de classe **A**
  - **B** *conté les dades d'inicialització* que es passaran a **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**

B és un creador dels objectes A

Exemple d'aplicació

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



Exemple d'aplicació (DS)

(es farà a classe)

Exemple no seguint el creador (DS)

(es farà a classe)

Contraindicacions, beneficis i patrons relacionats

- **Contraindicacions**: 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 la qual s'anomenarà *Factoria.*
- **Beneficis**: Afavoreix el baix acoblament, ja que la classe ja serà visible pel *creador* degut a les associacions existents que van motivar la seva elecció.
  - Conseqüentment, menors costos de manteniment i majors oportunitats de reusabilitat
- **Patrons relacionats**: baix acoblament, alta cohesió, factoria

3. Baix acoblament

- **L'acoblament** és una mesura de quant fort una classe està connectada (té coneixement de) altres classes.

  Un acoblament elevat produeix els següents problemes:

  – Canvis de classes relacionades obliguen a canviar la classe
  – És més complex entendre la classe de forma aïllada
  – És més difícil reusar, perquè requereix la presència de les classes de las que és depenent (classes que té acoblades)

3. Baix acoblament

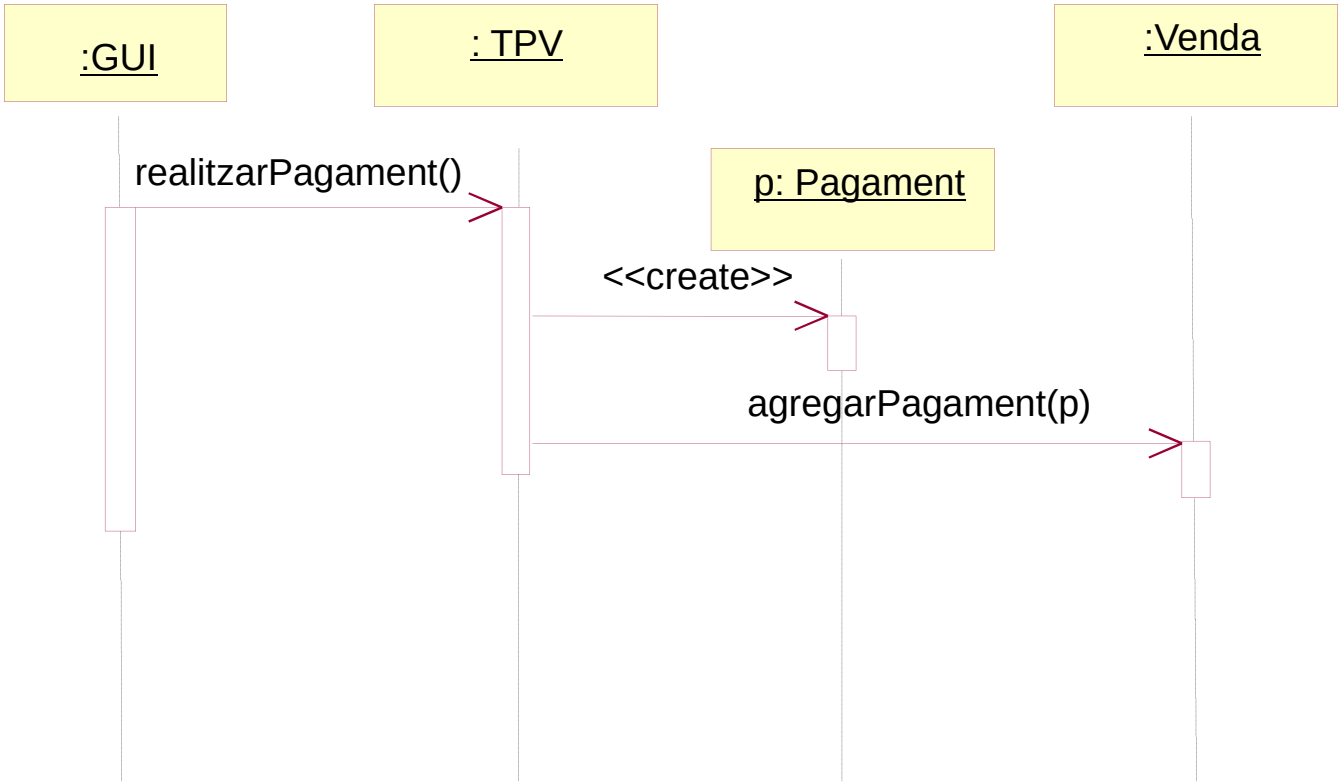
- **Descripció**: Com crear dissenys amb poques dependències, que es vegin pocs impactats pels canvis i que potenciïn la reutilització?

- **Solució**: Assignar una responsabilitat de forma que l'acoblament sigui baix

- És un patró avaluatiu: un baix acoplament permet que el disseny de classes sigui més independent.
  - Redueix l'impacte dels canvis i augmenta la reutilització
  - L'aplica el dissenyar cada cop que ha d'avaluar una decisió de disseny
- No s'ha de considerar ailladament. Pot no ser important si la reutilització no és un objectiu

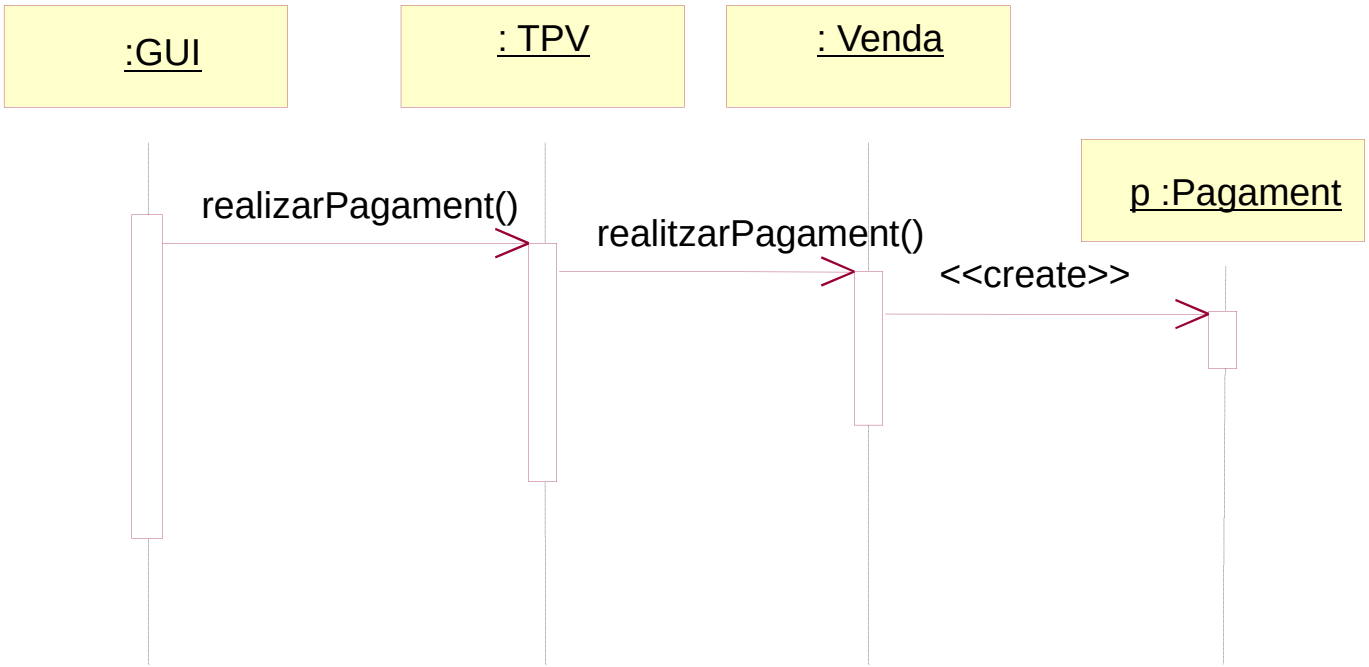
Exemple d'aplicació

- Quin d'aquests dissenys produeix un menor acoblament? Quin suggeriria *creador*?



Exemple d'aplicació

- Quin d'aquests dissenys produeix un menor acoblament? Quin suggeriria *creador*?



Comentaris

- **A la pràctica,** el nivell d'acoblament pot ser considerat de forma aïllada, sinó en paral·lel com *expert en la informació* o *alta cohesió.*
  - En qualsevol cas, és un dels factors a considerar per millorar un disseny.
- Tipus d'acoblament entre 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

Més comentaris i contraindicacions

- No existeix una mesura absoluta de quan el nivell d'acoblament és massa alt (tot i que existeixen mètriques i suggerències)
- Les classes molt genèriques i amb alta probabilitat de ser reutilitzades han de mantenir un nivell d'acoblament més baix
- S'ha de forçar el baix acoblament especialment en parts del sistema que se sap que són propicies a canviar, inestables o punts d'extensió del sistema
- **Contraindicacions:** L'acoblament a elements estables de la plataforma de desenvolupament no pot constituir un problema

Beneficis i patrons relacionats

- Redueix l'impacte de canvis en terceres classes
- Produeix una classe més simple d'entendre de forma aïllada
- Facilita la reusabilitat
- **Patrons relacionats**: Variacions protegides, alta cohesió

4. Alta cohesió

- La **cohesió (funcional)** és una mesura de quan fortament relacionades i focalitzades estan les responsabilitats d'un element.
- Una cohesió baixa produeix els següents problemes:
  - Dificulta la comprensió
  - Dificulta la reusabilitat
  - Dificulta el manteniment
  - Produeix un sistema delicat constantment afectat pel canvi
- **Descripció del problema**: Com mantenir la complexitat manejable?
- **Solució**: Assignar una responsabilitat de forma que la cohesió es mantingui alta

Alta cohesió

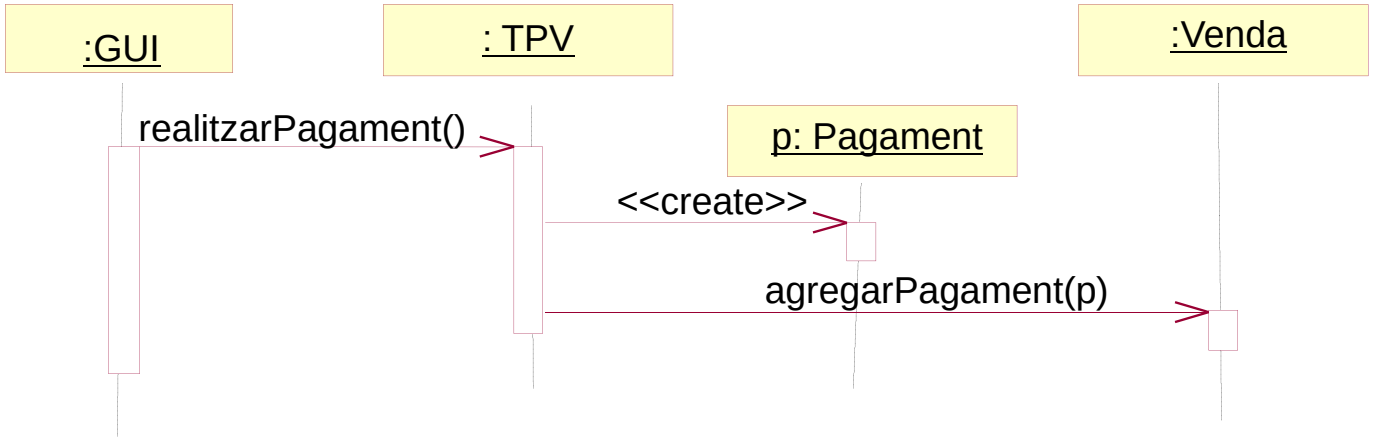
- Una classe amb alta cohesió:
  - no té molts mètodes, que estan molt relacionats funcionament
  - no realitza molt de treball
  - col·labora amb altres classes

- **Beneficis**
  - Millora la reutilització
  - Classes més clares, es poden comprendre millor
  - Millora el manteniment

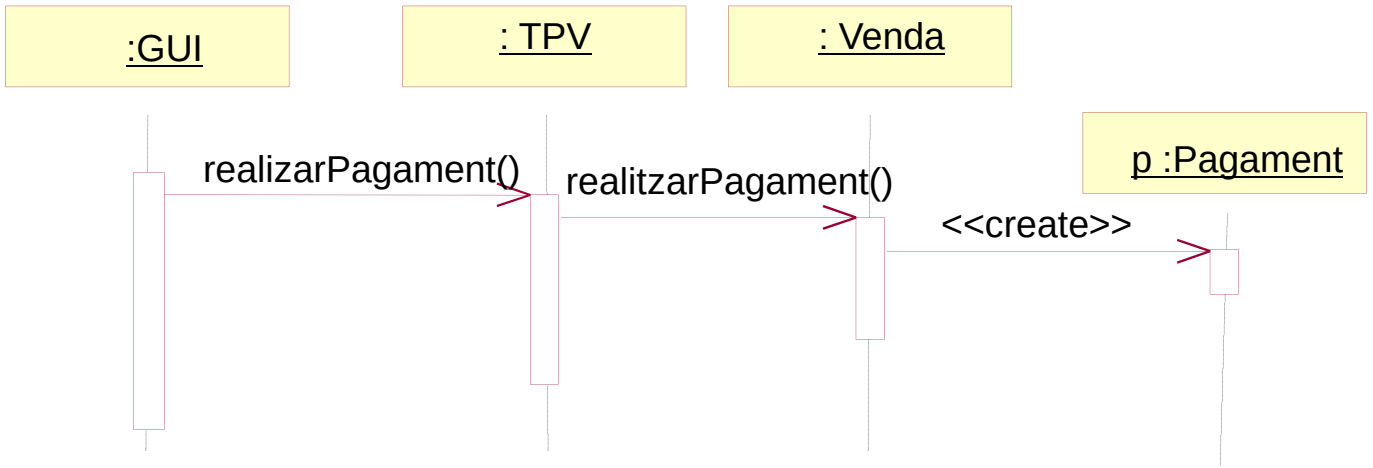
Exemple d'aplicació

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

(a)



(b)



Alta cohesió: Graus

- **Molt baixa**: Una classe és responsable única de moltes coses diferents en àrees funcionals. Exemple: WebServices-Database-UI-Manager
- **Baixa**: Una classe és responsable única d'una tasca complexa en l'àrea funcional. Exemple: DatabaseManager únic
- **Moderada**: Una classe té responsabilitats lleugeres en unes quantes àrees diferents, relacionades amb el concepte de la classe però no entre elles. Exemple: Companyia coneix els seus empleats i la seva informació financera
- **Alta**: Una classe té responsabilitats moderades en una àrea funcional i col·labora amb altres per satisfer tasques. Exemple: DatabaseManager ajudat d'altres 10 classes
- En general, una classe cohesionada conté pocs mètodes que la seva funcionalitat estigui molt relacionada i no realitza molt de treball. Delega en d'altres objectes part de les seves responsabilitats. Per tant, és més fàcil mantenir, reusar i entendre

Contraindicacions i Beneficis

- **Contraindicacions**
  - Quan facilita el manteniment per una persona. Exemple: Diccionari de queries
  - Quan els costos de comunicació s'incrementen, com per exemple en la programació amb objectes distribuïts, poden sorgir objectes pocs cohesius per reduir els costos de comunicació
- **Beneficis**
  - S'incrementa la claredat i facilitat de comprensió del disseny
  - Es simplifiquen els costos de manteniment
  - Sovint es produeixen dissenys amb baix acoblament
  - S'incrementen les possibilitats de reutilització (més fàcil reusar una classe cohesionada que una amb molta funcionalitat i variada)
- **Patrons relacionats:** Baix acoplament

Universitat de Barcelona

5. Controller

Descripció: Qui ha de ser el responsable de gestionar l'entrada d'un event de sistema?

Solució: Assignar una responsabilitat de rebre i gestionar un event de sistema a una classe que representa una de les següents opcions:

- El sistema, dispositiu o subsistema complet
- Un escenari de cas d'ús en el qual l'esdeveniment (event) de sistema ocorre, sovint nomenat <UseCaseName>Handler o <UseCaseName>Coordinator

S'hauria d'usar el mateix controlador per tots els events de sistema en el mateix escenari de cas d'ús. No han de fer-ho les classes de la interfície d'usuari (Window, Application, JFrame,...)

- Un **controller** és qualsevol classe responsable de rebre o manegar events de sistema

Model Vista Controlador (MVC)

- Qualsevol arquitectura fa la distinció entre capa de presentació i capa de domini

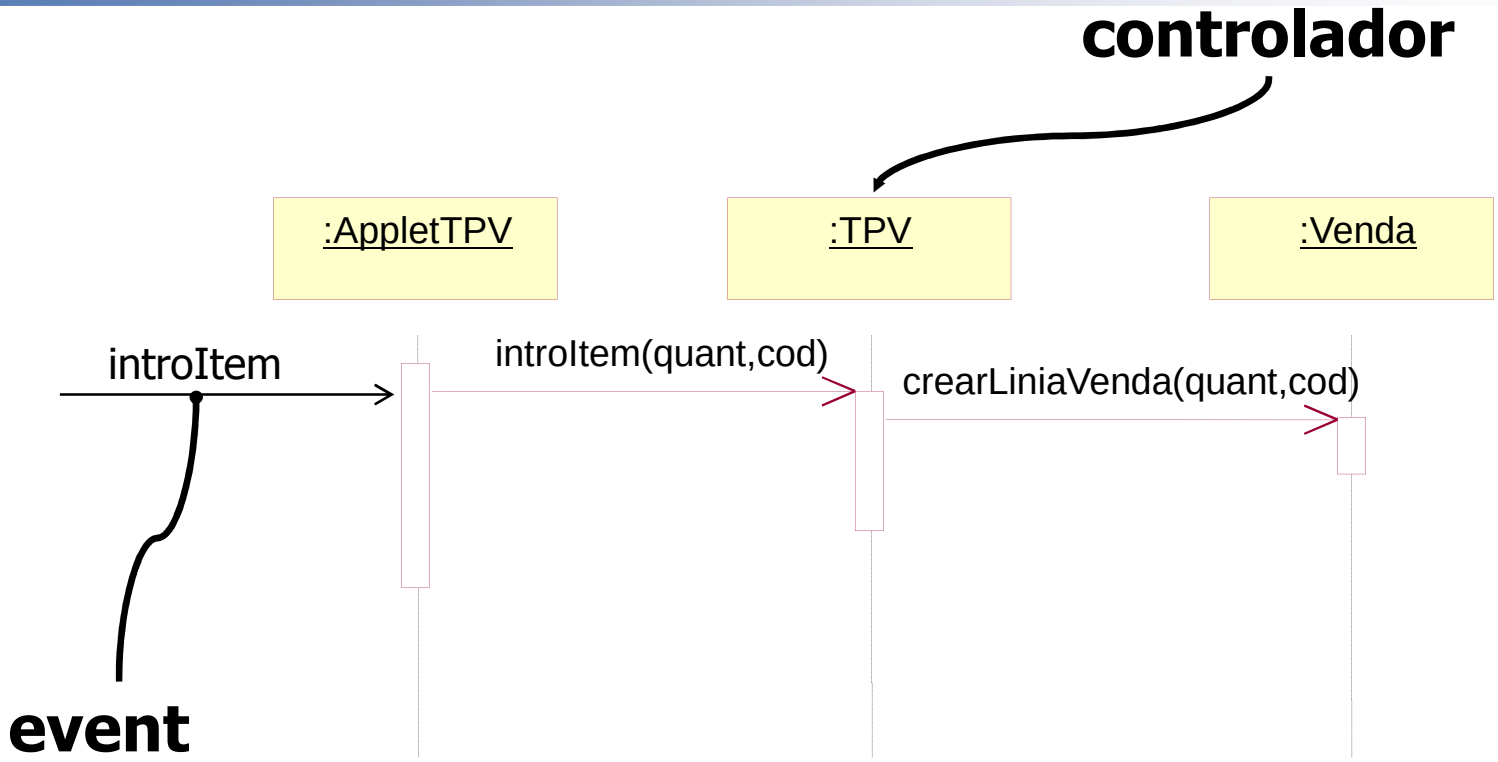
- Un sistema orientat a objecte típic es divideix en tres capes arquitectòniques:
  - **Interfície d'usuari**. Interfície gràfica. Finestres. Deleguen les tasques a un controlador.
  - **Lògica d'aplicació i objectes del domini.** Objectes que representen conceptes del domini i tenen com a missió satisfer els requisits
  - **Serveis tècnics**. Objectes de propòsit general i subsistemes que proveeixen de suport tècnic

Què és un controlador?

- Un controlador és un objecte que no pertany a la capa de presentació, s'encarrega de rebre i manegar un event del sistema que procedeix normalment de la interfície gràfica
- Una classe controlador inclou un mètode per a cada operació del sistema que manega
- Un controlador NO és un objecte de la interfície

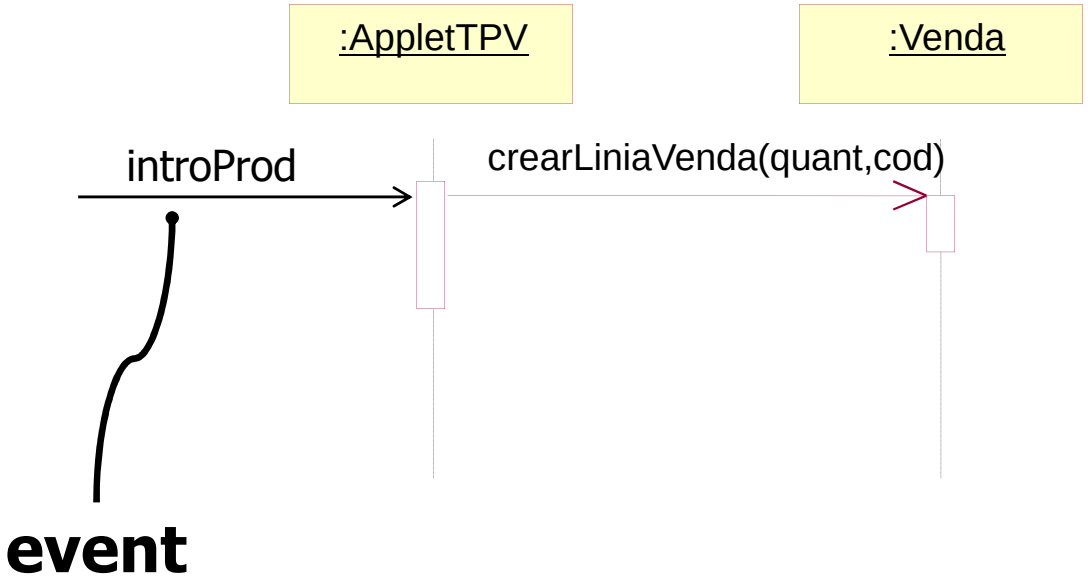
- Un controlador o varis?
  - Usar els patrons d'avaluació (cohesió/acoblament) per decidir. En particular, si hi ha molts events, un sol objecte pot ser poc cohesiu
  - Si es necessari guardar o assegurar una seqüència d'events és millor un controlador per cada cas d'ús

Exemple d'aplicació



Acoblament adequat de la capa presentació amb la capa del domini

No segueix el patró controlador



Acoblament inadequat de la capa presentació amb la capa del domini

Universitat de Barcelona

Comentaris

- Un mateix *controller* hauria d'agrupar tots els events de sistema d'un cas d'ús de forma que controli el seu estat
- Un *controller* ha de delegar a altres objectes el treball que s'ha de realitzar
- Habitualment es comença amb un *controller* per a tots els events de sistema i quan es fa massa complex (poc cohesionat i altament acoblat) es divideix

Més comentaris i Beneficis

- **Comentaris**
  - Diferents variacions del patró s'apliquen en el cas d'aplicacions amb GUI, aplicacions web i aplicacions client servidor, però en general és vàlid per als tres casos. En el cas de sistemes de missatges s'usen patrons *Command* i *CommandProcessor*

- **Beneficis**
  - Major potencial de reusalibitat i interfícies d'usuari intercanviables
  - Control de l'estat del cas d'ús. Important per a seqüències d'events que s'ha de donar en un cert ordre

Controladors inflats

- **Signes**:
  - Hi ha un únic controller rebent tots els events del sistema i són molts
  - El controlador realitza ell mateix gran part de les tasques enlloc de delegar
  - Un controlador manté atributs i informació que haurien de ser distribuïdes a d'altres objectes o duplica la informació que ja es troba en altres llocs
- **Solucions**:
  - Afegir més controladors
  - Delegar responsabilitats a altres objectes

Una altra classificació dels patrons

- Segons el seu propòsit:
  - **De creació**: fan referència al procés de creació d'objectes
  - **D'estructura**: tracten la composició de classes i/o objectes
  - **De comportament**: caracteritzen les formes en les que interactuen i reparteixen responsabilitats les diferents classes o objectes

Classificació dels patrons (GoF)

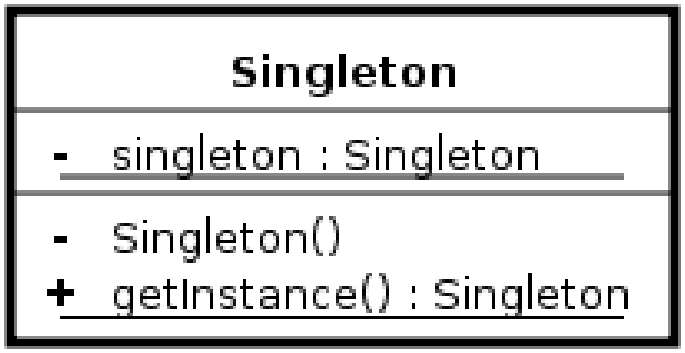
Àmbit/Propòsit	Creació	Estructura	Comportament
Classe	Factory Method	Adapter	Interpreter Template method
Objecte	Astract Factory Builder Prototype **Singleton**	Adapter Bridge Composite **Facade** FlyWeight Proxy	Chain of Responsability Iterator Mediator Memento Observer State Strategy Visitor

Patró SINGLETON

- **Utilitat**
  - Assegurar que una classe només té una sola instància i proporciona un punt d'accés global a ella
- **Avantatges**
  - És necessari quan hi ha classes que han de gestionar de manera centralitzada un recurs
  - Una variable global no garantitza que només s'instancii una sola vegada

Patró SINGLETON

- El constructor de la clase ha de ser **PRIVAT**
- Es proporciona un **mètode estàtic** a la classe que retorna la única instància de la classe: getInstance()



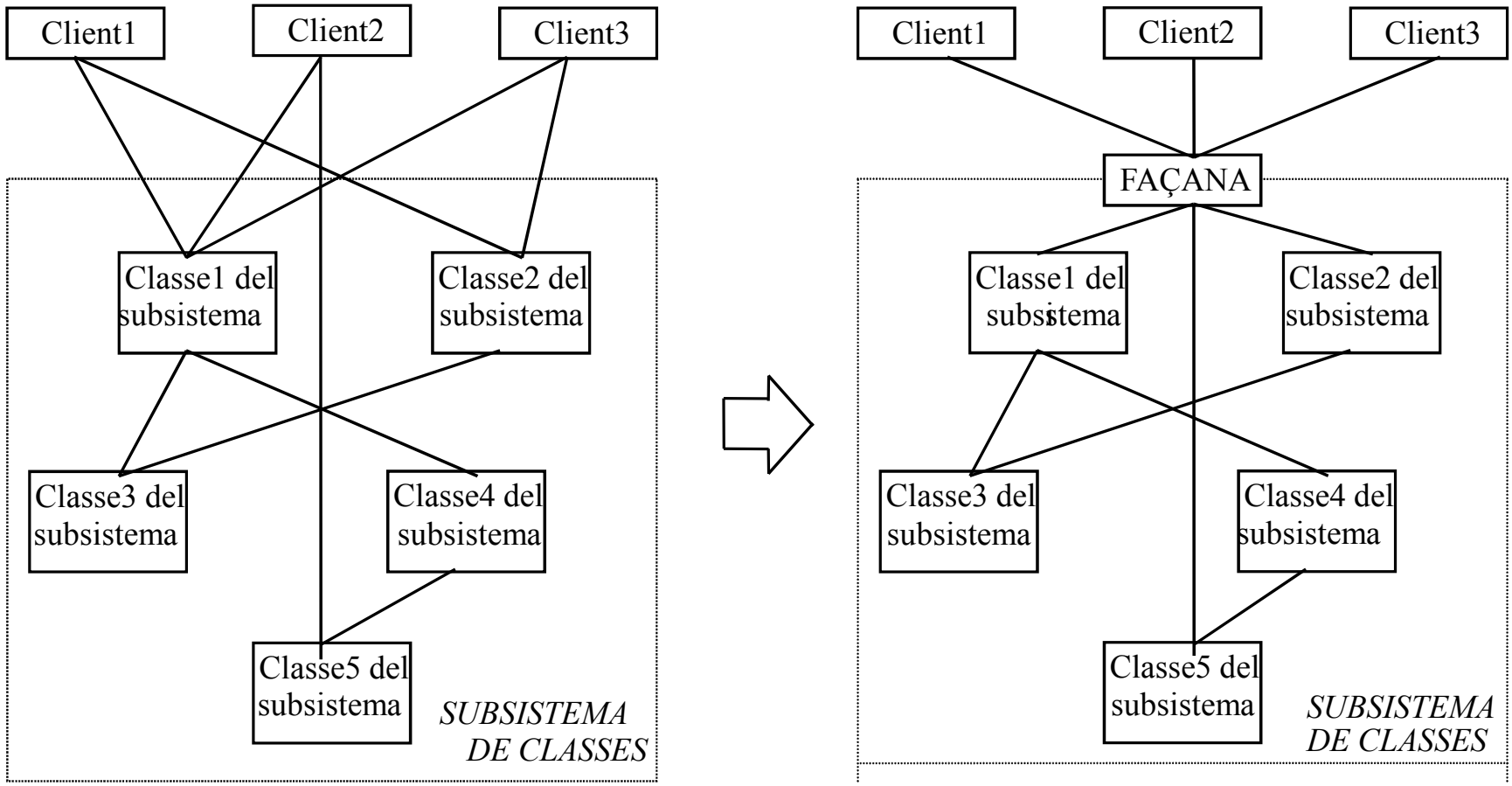
Patró SINGLETON

```
class Singleton {
  private static Singleton ejemplar = null;
 public static Singleton getEjemplar() {
  if ( ejemplar == null )
            ejemplar = new Singleton();
  return ejemplar;
 }
 protected Singleton() {
  // lo que sea necesario
 }
 public void metodo() {...}
}
```

Patró FACADE o FAÇADE

- Utilitat:
  - Simplifica l'accés a un conjunt d'objectes proporcionant un que tots els clients puguin usar per comunicar-se amb el conjunt
- Motivació
  - Minimitzar les comunicacions i dependències entre subsistemes

Patró FAÇADE



Patró FAÇADE

- Per proporcionar una interfície senzilla a un subsistema complex
  - A mesura que un subsistema evoluciona va tenint més classes, més petites, més flexibles i configurables
  - Hi ha clients que no necessiten tanta flexibilitat i que volen una visió més simple del subsistema
  - Només els clients que necessitin detalls a més baix nivell accediran a les classes que hi ha darrera la façana
- Quan hi ha moltes dependències entre els clients i les classes d'implementació d'una abstracció
  - La façana desacobla el subsistema dels clients i d'altres subsistemes
  - Això millora la independència dels subsistemes i la portabilitat
- Per estructurar un sistema en capes
  - La façana permet un punt d'entrada a cada nivell
  - Es poden simplificar les dependències obligant als subsistemes a comunicar-se unicament a través de les façanes

Patró FAÇADE

- **Esquema, participants i col·laboracions**
  - Els clients es comuniquen amb el subsistema fent peticions a la façana, que les envia als objectes del subsistema apropiats (la façana també podria traduir la seva interfície a les interfícies del subsistema)
  - Els clients que usan la façana no han d'accedir als objectesdel subsistema directament

Patró FAÇADE

- **Conseqüències**
  - Oculta als clients els components del subsistema
    - Redueix el nombre d'objectes que han de tractar els clients
  - Disminueix l'acoblament entre un subsistema i els seus clients
    - Un menor acoblament facilita el canvi dels components del subsistema sense afectar als clients
    - Les façanes permeten estructurar el sistema en capes
    - Redueix les dependències de compilació
  - No evita que les aplicacions puguin usar les classes del subsistema si les necessiten
    - Es pot escollir entre facilitat d'ús o generalització

Patró FAÇADE

- Implementació i patrons relacionats
- Reducció de l'acoblament client-subsistema
  - Es pot reduir més l'acoblament fent la façana una classe abstracta amb subclasses concretes per a les diferents implementacions del subsistema. Els clients es comuniquen amb el subsistema usant la interfície de la classe façana abstracta
  - Una altra possibilitat és configurar l'objecte façana amb diferents objectes del subsistema. Per personalitzar la façana n'hi ha prou amb reemplaçar un o varis dels seus objectes del subsistema
  - La factoria abstracta es pot utilitzar al costat de la façana per crear objectes del subsistema de manera independent al subsistema
- Classes del subsistema privades i públiques
  - En Java es poden usar els paquets per determinar les classes que són visibles fora o no. En C++ els namespaces no suporten l'ocultació de classes
- Normalment només fa falta un objecte façana, per tant, se sol implementar com un singleton