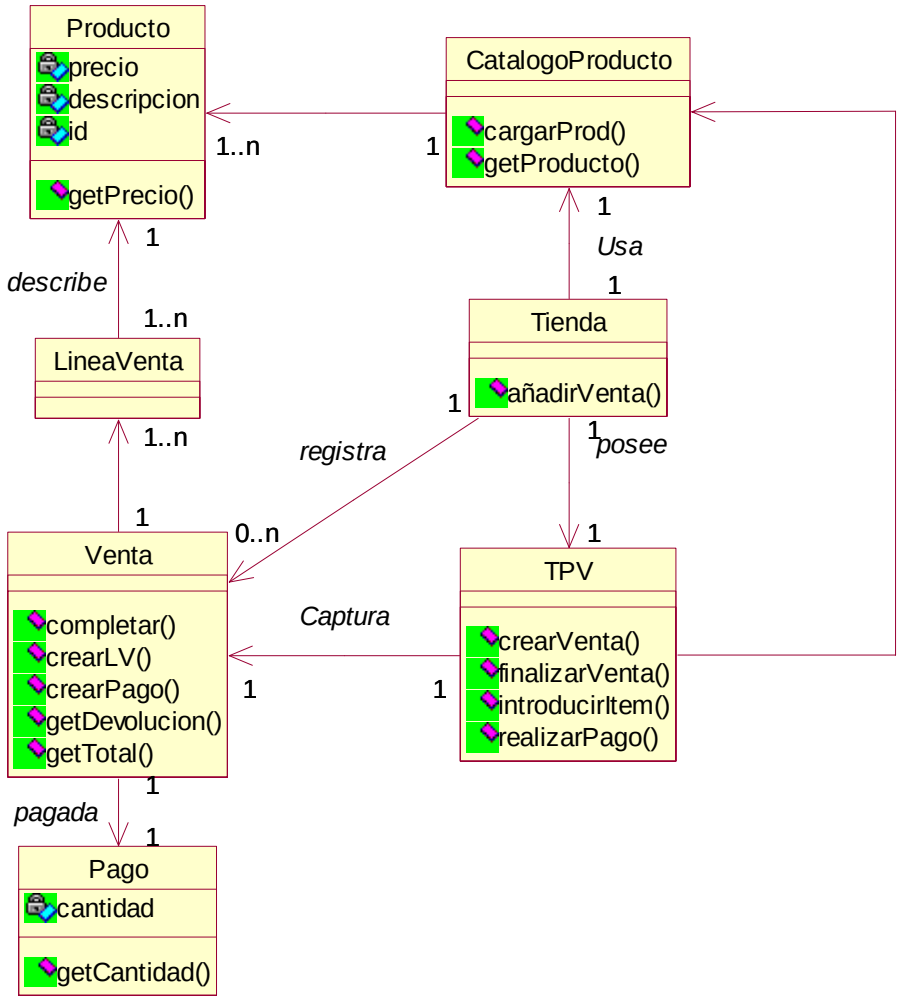
EXEMPLE D'UTILITZACIÓ DE PATRONS GRASP [1]

Es vol desenvolupar el software necessari per a fer funcionar un terminal punt de venda (TPV)

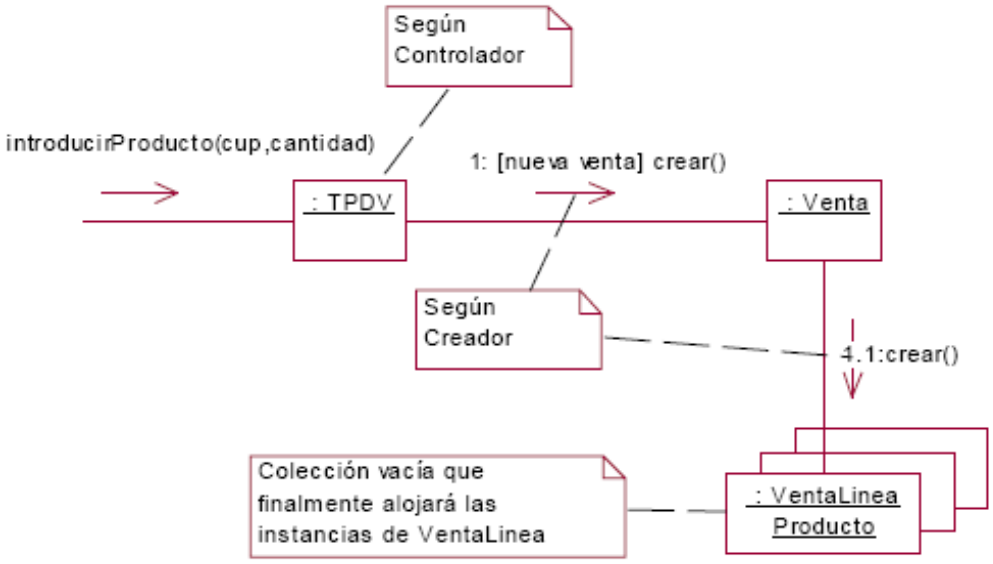
Farem servir el següent diagrama de classes:



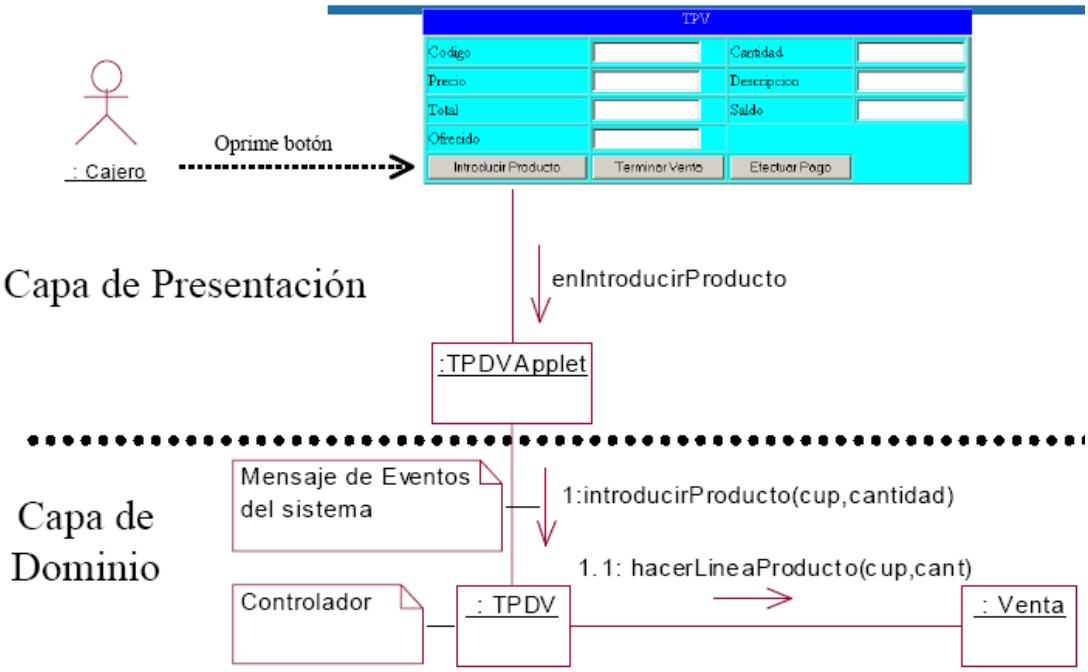
[1] Exemple extret del llibre: Craig Larman. *UM Ly Patrones*. 2nd Edición. Prentice Hall, 2002.

La classe controlador TPV representa el "sistema" global.

Exemple de crear una venta:



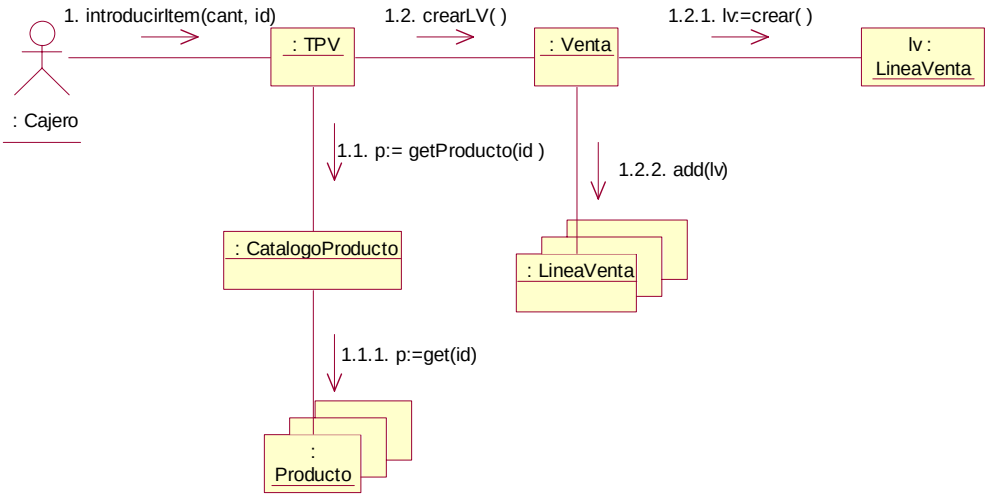
Exemple d’afegir una linia de venta:



Anem a analitzar amb una mica més de detall l'exemple anterior: qui tindrà la responsabilitat de crear una instancia de *LineaVenta ?*

Ja que *Venta* conté (agrega) molts objectes *LineaVenta*, el patró Creador suggereix que la responsabilitat ha de ser de *Venta*.

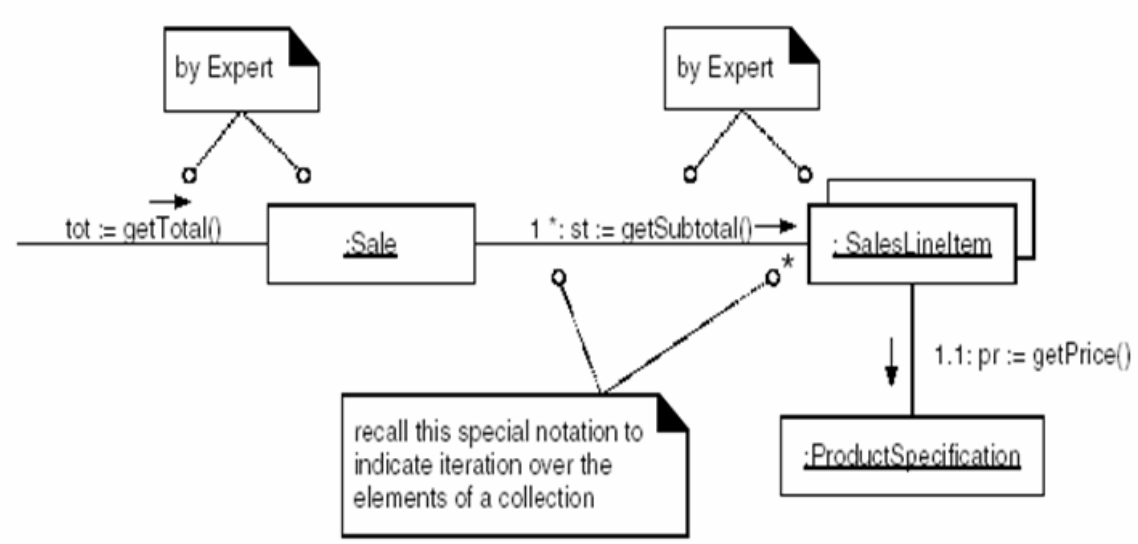
A continuació presentem el diagrama de col·laboració resultant:



Suposem que volem conèixer l'import *total* d'una venda.

A la vista del diagrama de classes es dedueix que qui té coneixement de *LineaVenta* és *Venta*. Per tant el mètode ge*tTotal()* ha de ser de *Venta.*

Per a calcular el *subtotal* d'una línia ens cal conèixer la *cantidad* del producte i el seu *precio*. L'expert d'ambdos és *LineaVenta*.



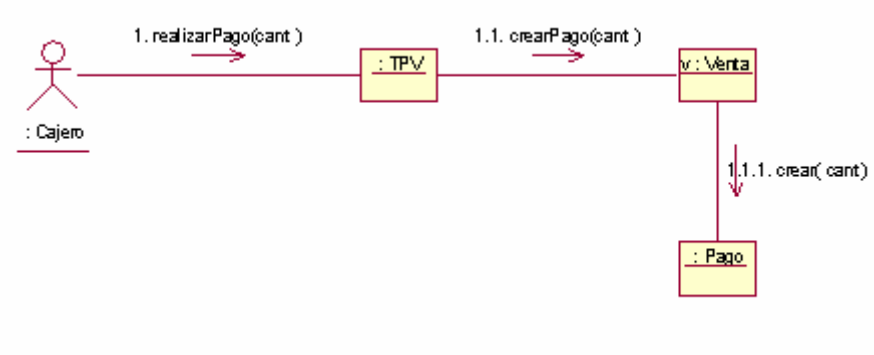
Per tant *Venta* necessita enviar missatges de subtotal a cada *LineaVenta* i sumar els resultats. D'altra banda, *LineaVenta* ha d'enviar un missatge *getPrecio* a *Producto.*

Cal crear una instància de *Pago* i associar-la amb *Venta*. Quina classe haurà de tenir aquesta responsibilitat ?

En el mon real, *PTV* enregistra un *Pago*, per tant el patró Creador suggereix que *PTV* ha de crear un objecte *Pago*. La instància de *PTV* haurà d'enviar un missatge *crearPago* a *Pago*, passant el nou *Pago* com a paràmetre.

D'aquest forma la classe *PTV* quedaria acoblada amb la classe *Pago.*

Podem considerar una solució alternativa consistent en crear el *Pago* i associar-lo amb *Venta*.



Ara *Venta* fa la creació de *Pago* de manera que no s'incrementa l'acoblament.

Observis que, a més amés, amb la primera solució *PTV* podria rebre moltes operacions del sistema i per tant es convertiria en un objecte amb molt baixa cohesió. Per contra, amb la segona solució *PTV* té alta cohesió.

Còdi Java

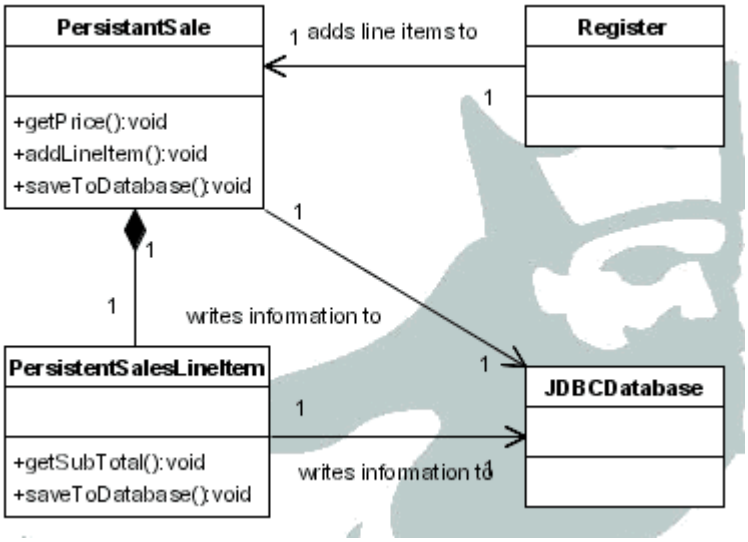
```
public class Pago {
        private Dinero cantEntregada;

        public Pago (Dinero cantidad) { cantEntregada = cantidad; }
        public Dinero getCantEntregada () { return cantEntregada; }
}


public class CatalogoProducto {
        private Map productos = new HashMap ();

        public CatalogoProductos () {
                ItemId id1 = new ItemID(100);
                ItemId id2 = new ItemID(200);
                Dinero precio1 = new Dinero (3);
                Dinero precio2 = new Dinero (5);
                Producto p;
                p:=  new Producto (id1, precio1, “producto 1”);
                productos.put(id1, p);) }
                p = new Producto (id2, precio2, “producto 2”);
                productos.put(id2, p);) }
        public Producto getProducto (ItemId id) {
                return (Producto) productos.get(id); }
}


public class TPV {
        private CatalogoProducto catalogo;
        private Venta venta;

        public TPV(CatalogoProducto cp) { catalogo = cp; }
        public void crearNuevaVenta () {venta = new Venta();}
        public void finalizarVenta () { venta.completar(); }
        public void introducirItem (ItemId id, int cant) {
                Producto p = catalogo.getProducto (id);
                Venta.crearLineaVenta(p, cant); }
        public void realizarPago() { venta.crearPago(cant)}
}


public class Producto {
        private itemID id;
        private Dinero precio;
        private String descripcion;

        public Producto(ItemID id, Dinero precio, String desc) {
                this.id = id;
                this.precio = precio;
                this.descripcion = desc;}
        public ItemId getId() { return id; }
        public Dinero getPrecio() { return precio; }
        public String getDescripcion() { return descripcion; }
}
```

```
public class Venta {
	private List lineaVentas = new ArrayList();
	private Date fecha = new Date();
	private boolean esCompleta;
	private Pago pago;

	public Dinero getDevolucion() { return pago.getCantEntregada().
minus(getTotal() ); }
	public void completar() { esCompleta = true; }
	public void crearLineaVenta(Producto p, int cant) {
		lineaVentas.add(new LineaVenta(p,cant)); }

	public Dinero getTotal() {
		Dinero total = new Dinero();
		Iterator i = lineaVentas.iterator();
		while (i.hasNext()) {
			LineaVenta lv = (LineaVenta) i.next();
			total.add(lv.getSubtotal()); }
		return total;
	}
	public void crearPago (Dinero cantEntregada) { pago = new
Pago(cantEntregada); }
}


public class LineaVenta {
	private int cantidad;
	private Producto producto;

	public LineaVenta(Producto p, int cant) {
		this.producto = p;
		this.cantidad = cant; }
	public Dinero getSubtotal () { return producto.getPrecio().times(cantidad); }
}


public class Tienda {
	private CatalogoProducto catalogo;
	private TPV tpv;

	public TPV getTPV {return TPV; }
}
```

Exemple del Patró Fabricació Pura

Qui se n'ha d'encarregar de guardar la informació de Sale a la Base de Dades?

El patró Expert en informació suggereix que ho faci Sale, ja que coneix tota l'informació (price, items):

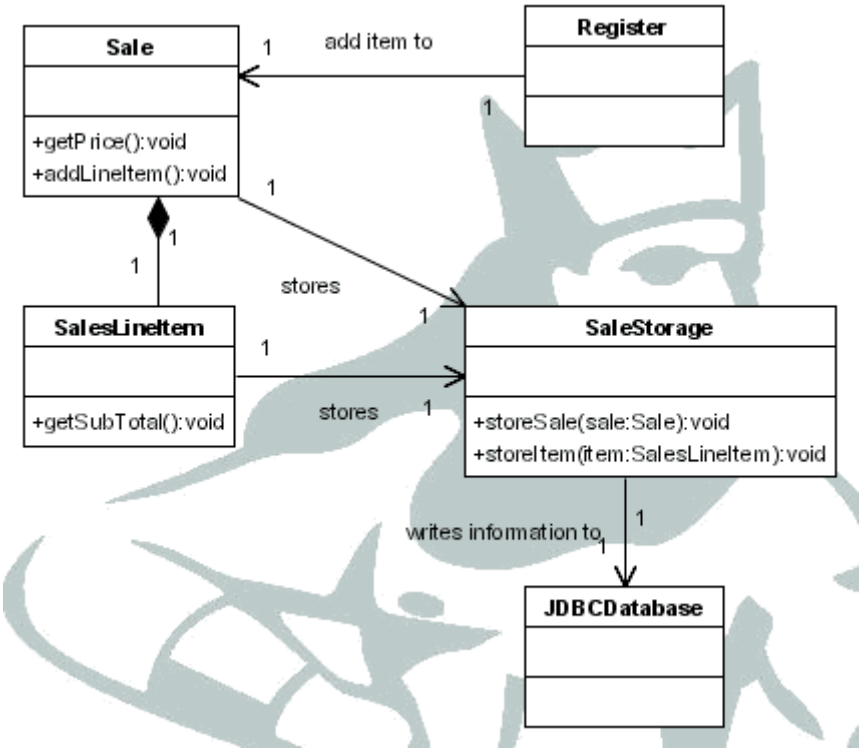


Problema:

Mala Cohesió: Sale fa dues coses diferents !

Solució:

Crear una nova classe: SaleStorage:



- Bona Cohesió: Sale només coneix informació de "sale" i la classe *Fabricada* SalesStorage només guarda informació de Sale a la Base de Dades.

- L'acoblament no és pitjor: Sale segueix "parlant" només amb una altra classe; Register només "parla" amb Sale.

- L'acoblament és millor: Sale és independent de *com* es guarda (JDBC, SQLServer, etc.).