INGENIERÍA DE SOFTWARE I

UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Unidad V

Tema: RUP Ágil.
Modelo de Diseño en RUP Ágil

UNTDF – 2020

RUP Agil. Modelo de Diseño

- Creación del Modelo de Diseño
- Realización de Casos de uso con Patrones GRASP
- Visibilidad
  - Tipos de visibilidad
- Navegabilidad
- Diagrama de clases del Diseño

RUP Agil. Dónde estamos???

Fase de Elaboración. Iteración I

Modelo de Diseño

Visibilidad

Visibilidad

- La visibilidad es la capacidad de un objeto de ver o tener una referencia a otro.
- Para que un objeto emisor envíe un mensaje al receptor, éste debe ser visible al emisor >>> debe tener referencia o puntero al receptor.

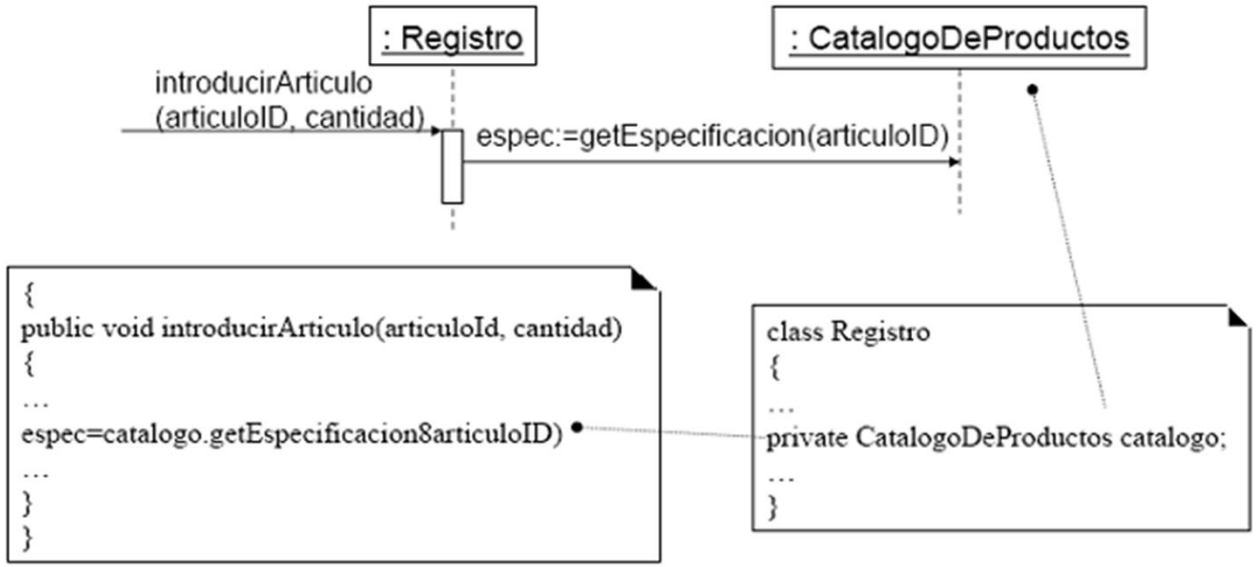
- Hay 4 formas de visibilidad:
  - Visibilidad de atributo
  - Visibilidad de parámetro
  - Visibilidad local
  - Visibilidad global

Visibilidad de atributo

- La visibilidad de atributo desde A a B existe cuando B es un atributo de A.
- Es relativamente permanente porque existe mientras existan A y B.
- Una instancia de Registro tendría visibilidad de una instancia de Catalogo, ya que es un atributo del Registro

```
public class Registro
{
   ......
   private Catalogo catalogo;
}
```

Ejemplo en un Diagrama de Interacción

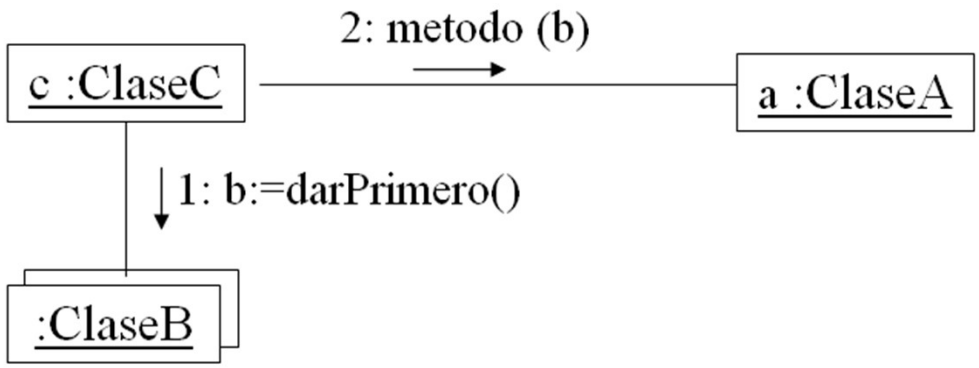


Visibilidad de Parámetro

- La visibilidad de parámetro de A a B existe cuando B se pasa como parámetro a un método de A.
- Es relativamente temporal porque existe sólo en el alcance del método
- Es habitual transformar esta visibilidad en visibilidad de atributo.

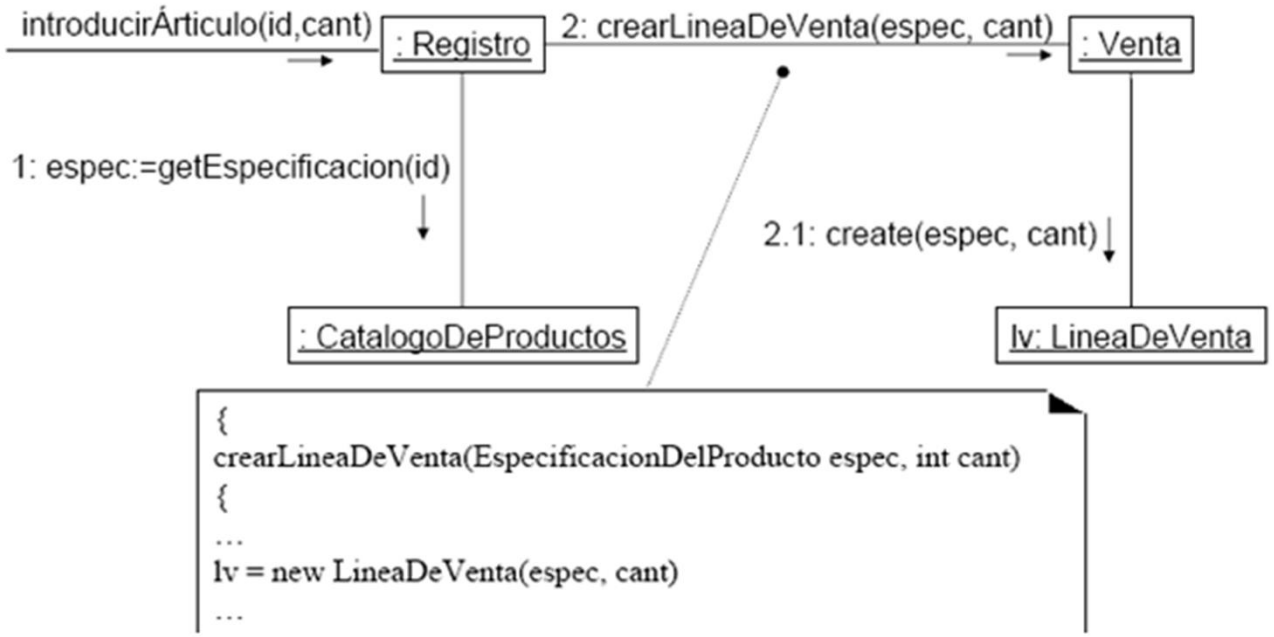
```
public class TipoA
{
...
public void metodo(TipoB objetoB)
{
...
}
...
}
```

Ejemplo de uso en un Diagrama de Interacción



Visibilidad de Parámetro

- En el ejemplo vemos que cuando se envía el mensaje crearLineaDeVenta a la instancia de Venta, se pasa como parámetro una instancia de EspecificacionDeProducto.>>> La Venta tiene visibilidad de parámetro de una EspecificacionDeProducto



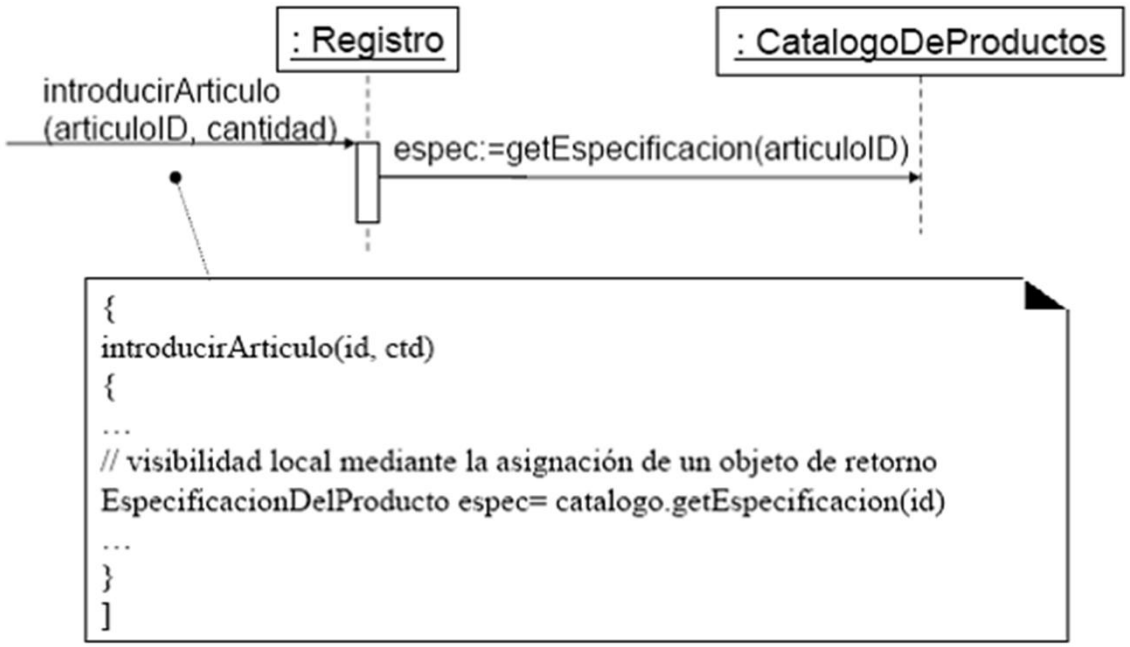
Visibilidad local

- La visibilidad local desde A a B existe cuando B se declara como un objeto local en un método de A.
- Es relativamente temporal porque sólo persiste en el alcance del método
- Dos medios comunes de alcanzar la visibilidad local son:
  - El objeto se puede crear dentro del propio método
  - El objeto puede ser recibido como valor de retorno en la llamada a otro método.

```
class TipoA
{
private TipoD objetoD;
...
public void metodo1()
{
objetoB = new TipoB();
TipoC objetoC = objetoD.metodoX();
...
}
...
}
```

Visibilidad local

- Ejemplo en un Diagrama de Interacción

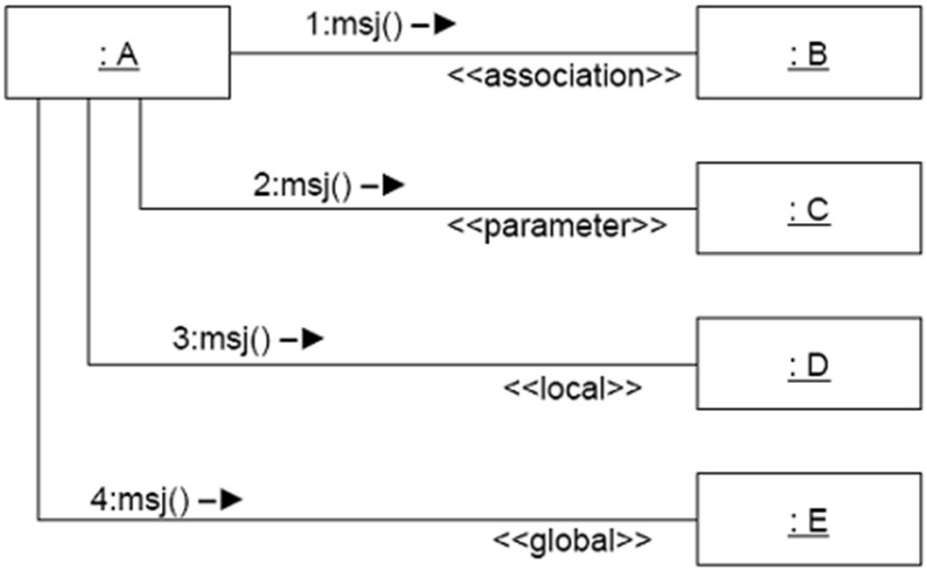


Visibilidad global

- El objeto A tiene visibilidad global sobre el objeto B cuando B es visible globalmente. Es una visibilidad permanente. Se puede conseguir de dos formas:
  - asignando B a una variable global
  - haciendo que B sea un *singleton*

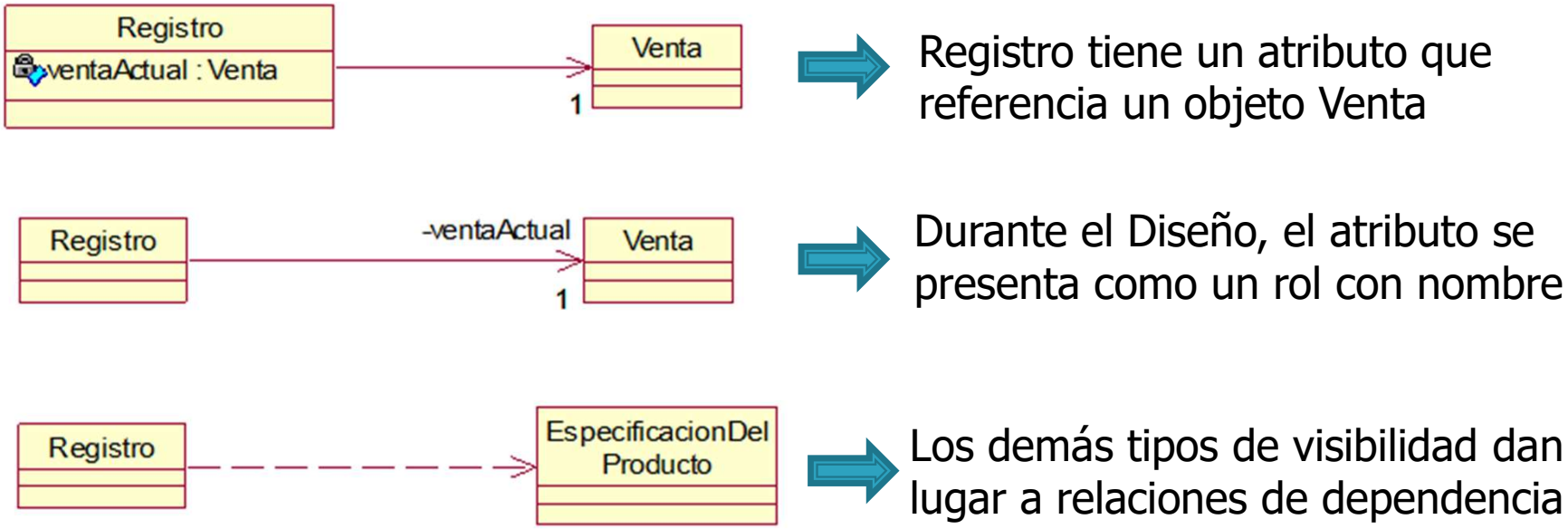
Visibilidad en los Diagramas de Comunicación UML

- UML permite representar, opcionalmente, el tipo de visibilidad sobre el que se basa un enlace en un diagrama de colaboración.



Representación de visibilidad en DCD

- La Navegabilidad es una propiedad del rol que indica que es posible navegar en la dirección de la flecha desde clase origen a destino.
- La navegabilidad en una asociación del diagrama de clases implica visibilidad de atributos.



RUP Agil. Dónde estamos???

Fase de Elaboración. Iteración I

Modelo de Diseño
Creación del Diagrama de Clases del Diseño

Modelo de Diseño
Creación de Diagrama de clases

- Una vez terminados los Diagramas de Interacción para la realización de los casos de uso, es posible realizar la especificación de las clases del software.
- Se les añaden métodos y atributos a las clases del Modelo del Dominio o a clases conceptuales del Análisis.
- Asimismo durante esta etapa aparecen nuevas clases no identificadas durante la etapa conceptual.
- Normalmente se van creando los Diagramas de clases en paralelo con los de interacciones.
- Se utilizan como fuente las clases del Modelo del Dominio y análisis y los Diagramas de Interacción de la realización de los casos de uso)

Modelo de Diseño
El DC en el Modelo de Diseño en RUP

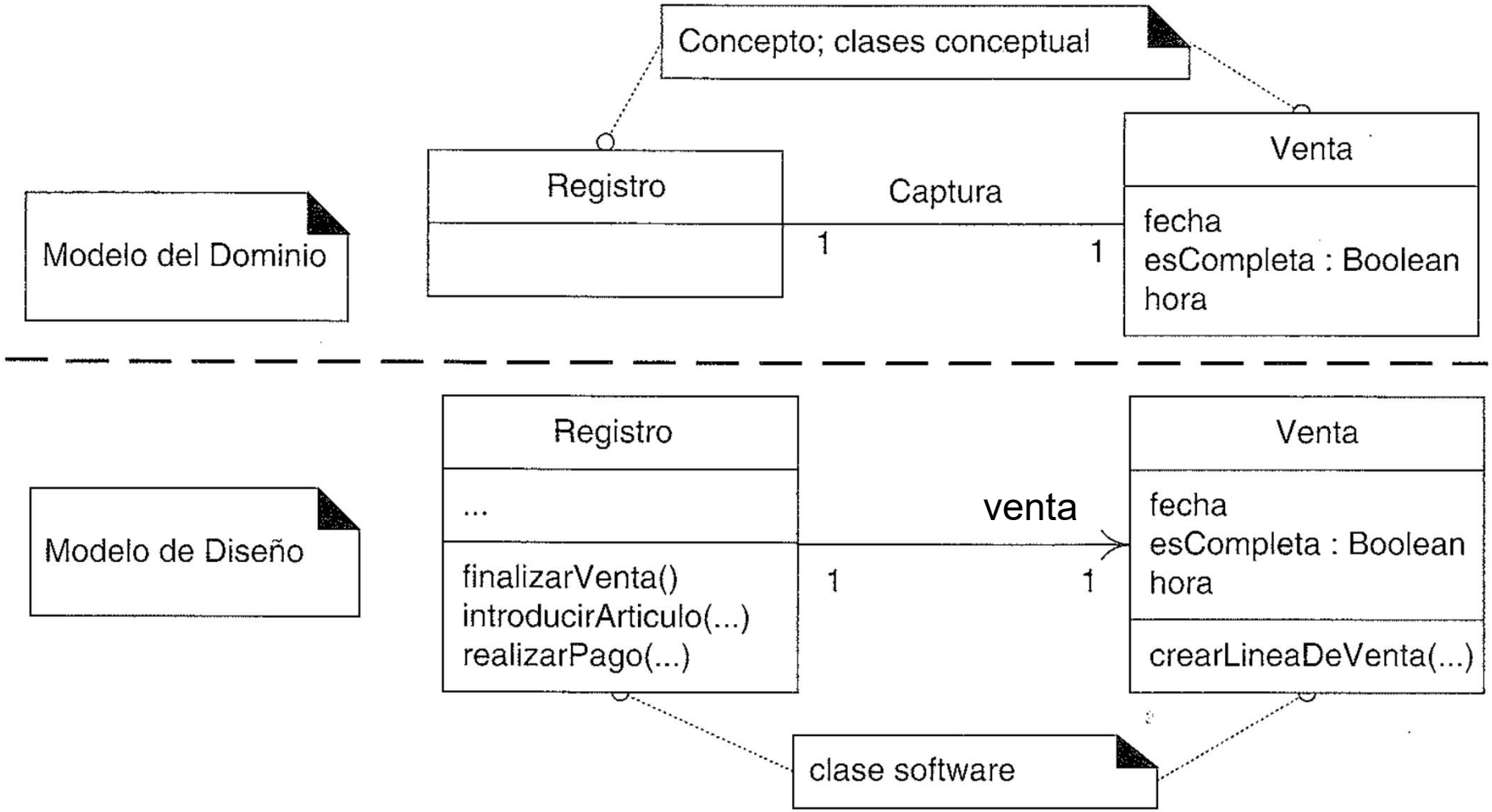
- RUP no define un Artefacto llamado "Diagrama de clases del Diseño" sino un "Modelo de Diseño".
- El Modelo de Diseño comprende además del Diagrama de clases, otros Diagramas: de interacciones, de paquetes, etc.

- El Diagrama de clases del Modelo de Diseño contiene "clases del Diseño"; se utiliza para modelar la estructura del sistema durante el diseño.
  - A diferencia de las clases conceptuales del Modelo de Dominio, las clases del Modelo de Diseño muestran las definiciones de clases software en lugar de conceptos del mundo real

Modelo de Diseño
El DCD de RUP

- Un DCD representa la especificación de las clases e interfaces software en una aplicación. Muestra:
  - Clases, asociaciones y atributos
  - Interfaces con sus operaciones
  - Métodos
  - Información acerca del tipo de los atributos
  - Navegabilidad
  - Dependencias

Clases conceptuales vs. Clases del Diseño



DCD en el Modelo de Diseño de RUP

- Durante esta etapa se identifica lo siguiente:
  - ✓ Qué clases serán implementadas? >>> Se utilizan como guía las clases del dominio y del análisis.
  - ✓ Qué estructura de datos utilizará cada clase?
  - ✓ Qué operaciones ofrecerá como servicio cada clase y cuáles serán sus métodos?
  - ✓ Cómo se implementarán las jerarquías de herencia y composición detectadas en el análisis?

- En esta etapa el Diagrama se amplía presentando (además de asociaciones y atributos) métodos de clase, información de atributos, dirección de navegación, nombres de roles
- Pueden crearse en paralelo con los Diagramas de Interacción

Clases SW para el ejemplo PDV

Registro

CatalogoDeProductos

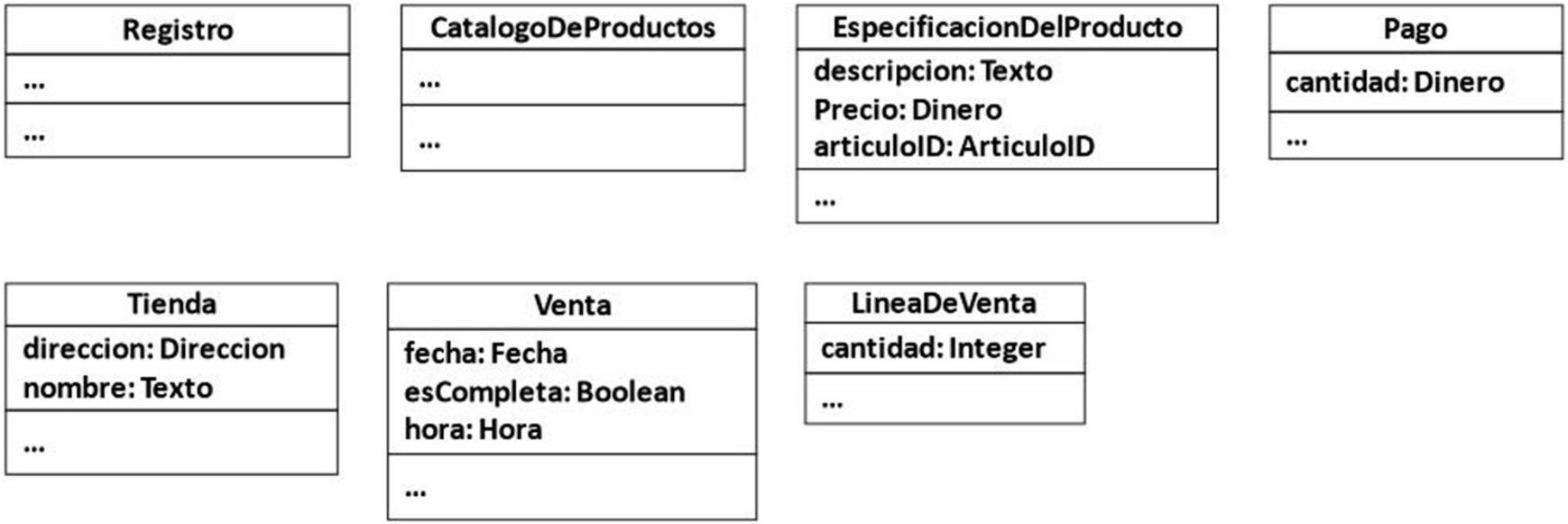
Tienda

Pago

Venta

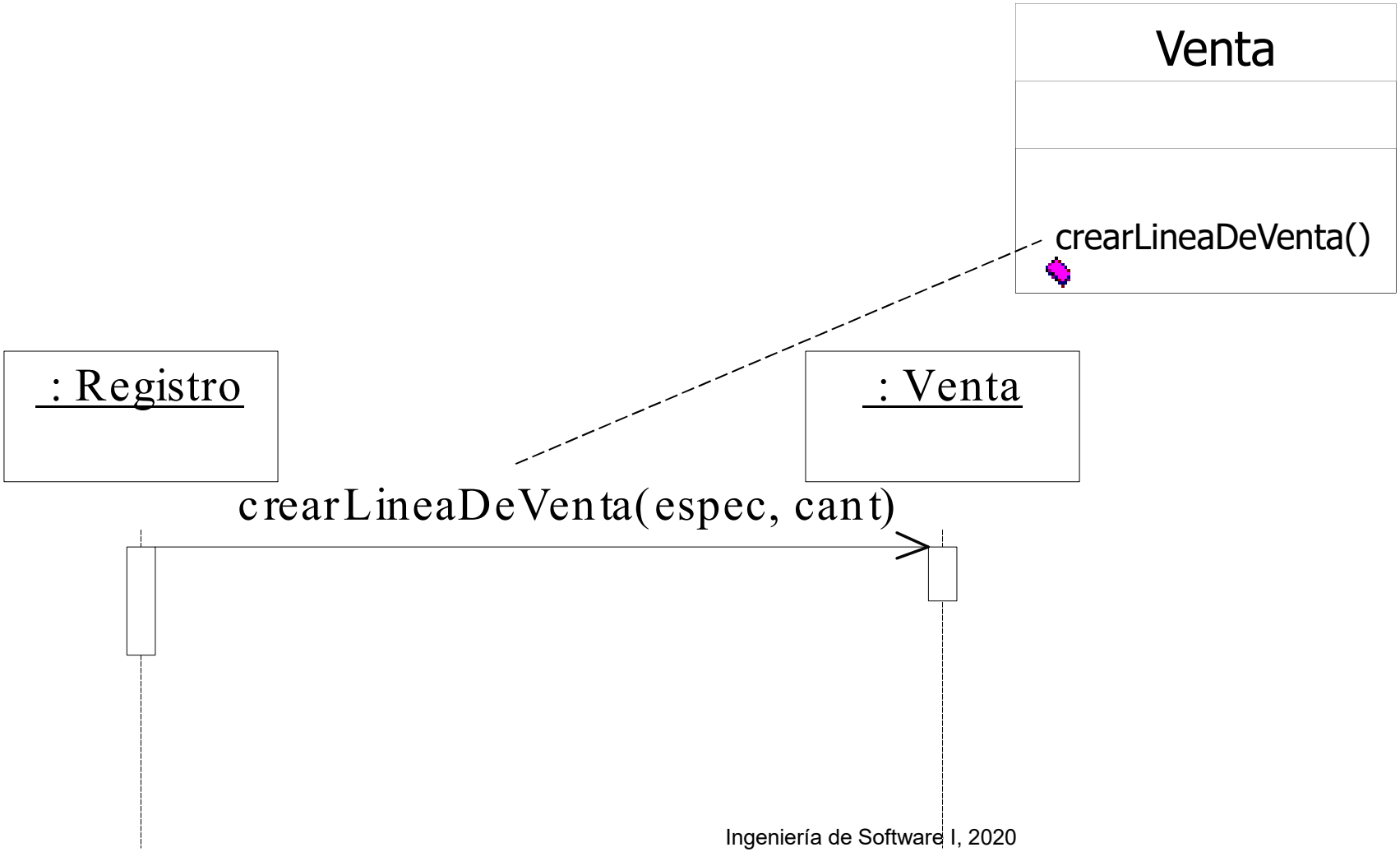
EspecificacionDelProducto

LineaDeVenta



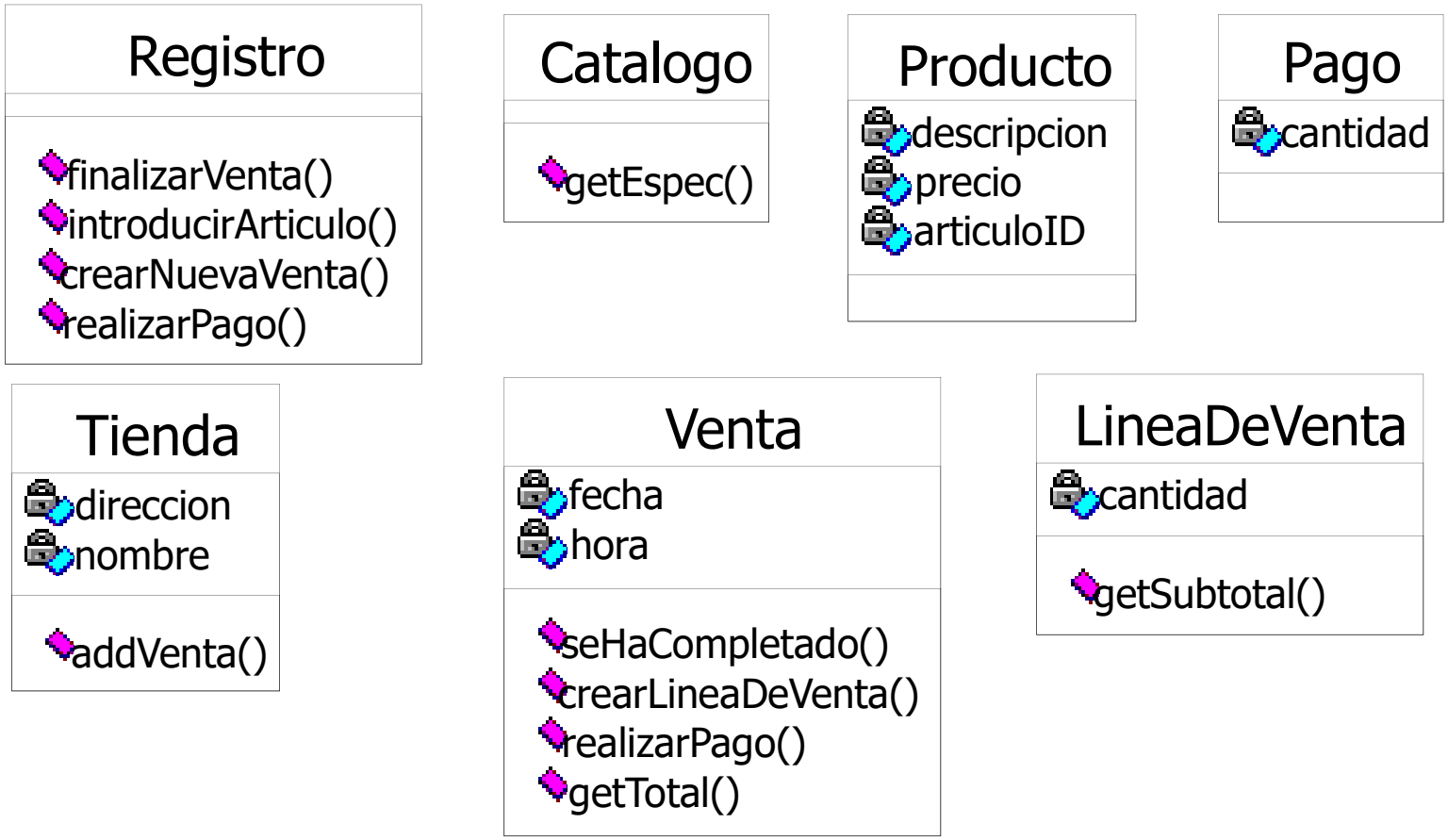
Nombres de los métodos

- Se pueden identificar los nombres de los métodos analizando los diagramas de interacción



Clases del Diseño con operaciones

- La inspección de todos los DI daría lugar a la asignación de operaciones que se muestran:



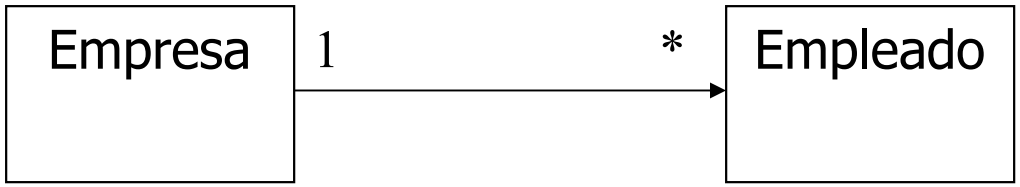
Nombres de los métodos

- El conjunto de todos los mensajes enviados a una clase X a través de todos los DI indican la mayoría de los métodos que debe tener la clase.

- Se debe tener en cuenta >>>
  - ✓Interpretación del mensaje *create.*
  - ✓Descripción de los métodos de acceso.
  - ✓Interpretación de mensajes a multiobjetos.
  - ✓Sintaxis dependiente del Lenguaje.

Construcción de instancias. Create

- Cuando una clase construye sus instancias, asigna espacio para su almacenamiento y configura los valores iniciales de los atributos o campos del objeto.
- Al crear el objeto se garantiza que se cumplan todos los requisitos, es decir que las asociaciones necesarias para su existencia, estén garantizadas.
- En el ejemplo siguiente, se debe garantizar que cada instancia de Empleado, tenga una asociación con un objeto de Empresa.

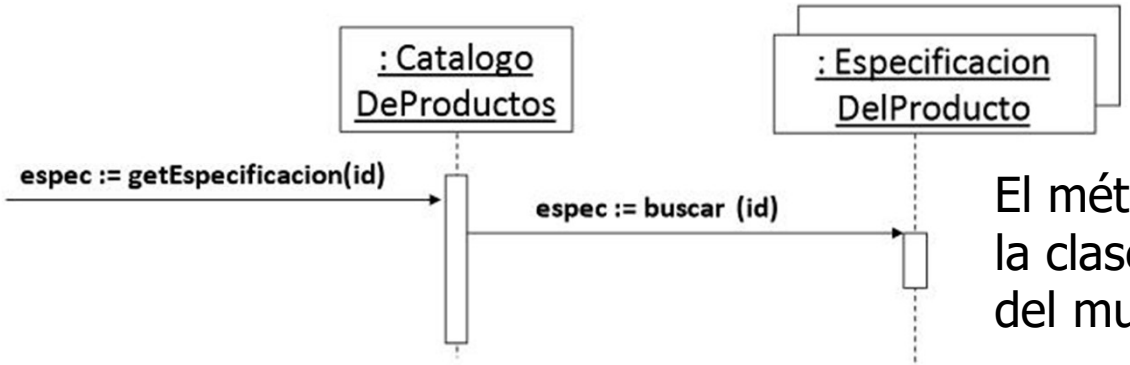


Construcción de instancias. Create

- El mensaje *create()* es una forma de indicar instanciación e inicialización.
- Al traducirlo a un Lenguaje OO, se debe expresar en función de su estilo para instanciación e inicialización.
- Por ejemplo en Java implica la invocación del operador new, seguido de una llamada al Constructor.
- Sin embargo, se recomienda diseñar la construcción de los objetos de la clase con dos operaciones para cada clase:
  - ✓ Crear: asigna espacio para el objeto de la clase
  - ✓ Iniciar: construye el objeto de acuerdo con las especificaciones y restricciones de la clase.
- Estas operaciones se suelen omitir en DCD por simplicidad

Operaciones fundamentales

- Los atributos deben contar con métodos de acceso para recuperar o establecer el valor de los atributos.
- Se les llama método de obtención (*accessor*) y método de cambio (*mutator*).
- Se suelen omitir estos métodos en el Diagrama de Clases.
- Un mensaje a un multiobjeto (colección, contenedor) se interpreta como un mensaje al propio objeto contenedor/colección >>> Por ejemplo un List o Map de Java o una Collection de Smalltalk. Por ej., el mensaje buscar(id)



El método buscar no forma parte de la clase Espec… sino de la interfaz del multiobjeto (Map, List,…)

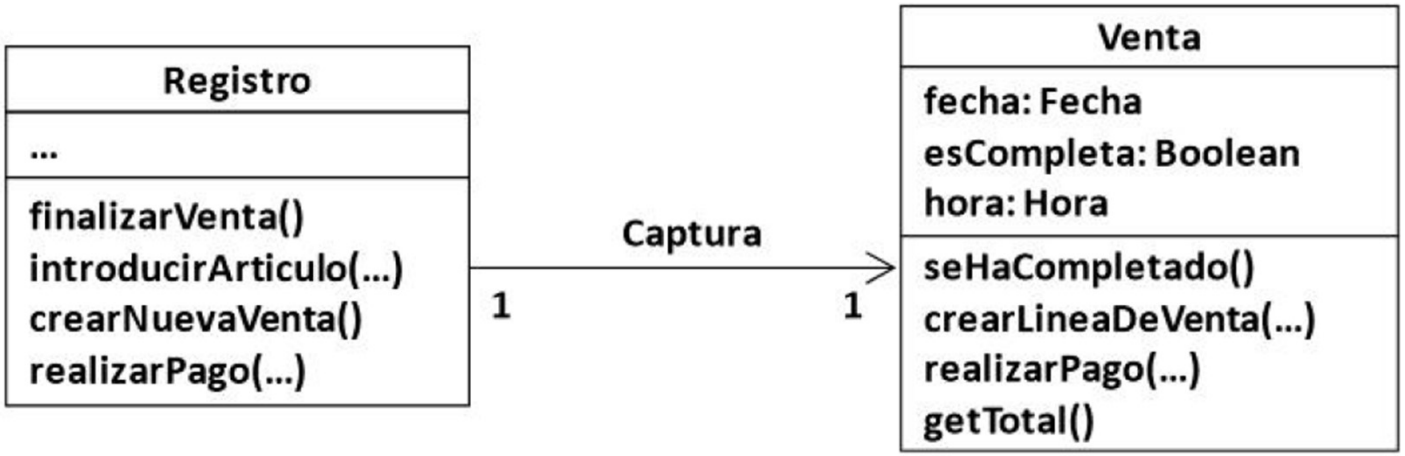
Asociaciones

- Casa extremo de la asociación se denomina rol y en los DCD el rol se completa con una flecha de navegabilidad.
- La navegabilidad implica visibilidad
- La interpretación de una asociación con una flecha de visibilidad es la visibilidad de atributo desde una clase origen a otra destino.
- Durante la implementación en un lenguaje OO una asociación básica se representa como un atributo en la clase origen que hace referencia a la clase destino.

- Por ejemplo la clase Registro definirá un atributo que referencia una instancia de Venta

Asociaciones

- La clase Registro deberá definir un atributo que referencia una instancia de Venta



Asociaciones

- Se agregan roles y navegabilidad >>> propiedad del rol que indica que es posible navegar undireccionalmente en la asociación.
- La visibilidad y las asociaciones requeridas entre las clases se dan a conocer mediante los DI.

- Situaciones comunes que sugieren definir una asociación con un adorno de visibilidad de A a B:
- A envía un mensaje a B
- A crea una instancia de B
- A necesita mantener una conexión a B.

Asociaciones

- Se identifica una conexión permanente entre Tienda con Registro y CatalogoDeProducto (los crea). Idem Catalogo con la colección Especificación de Producto

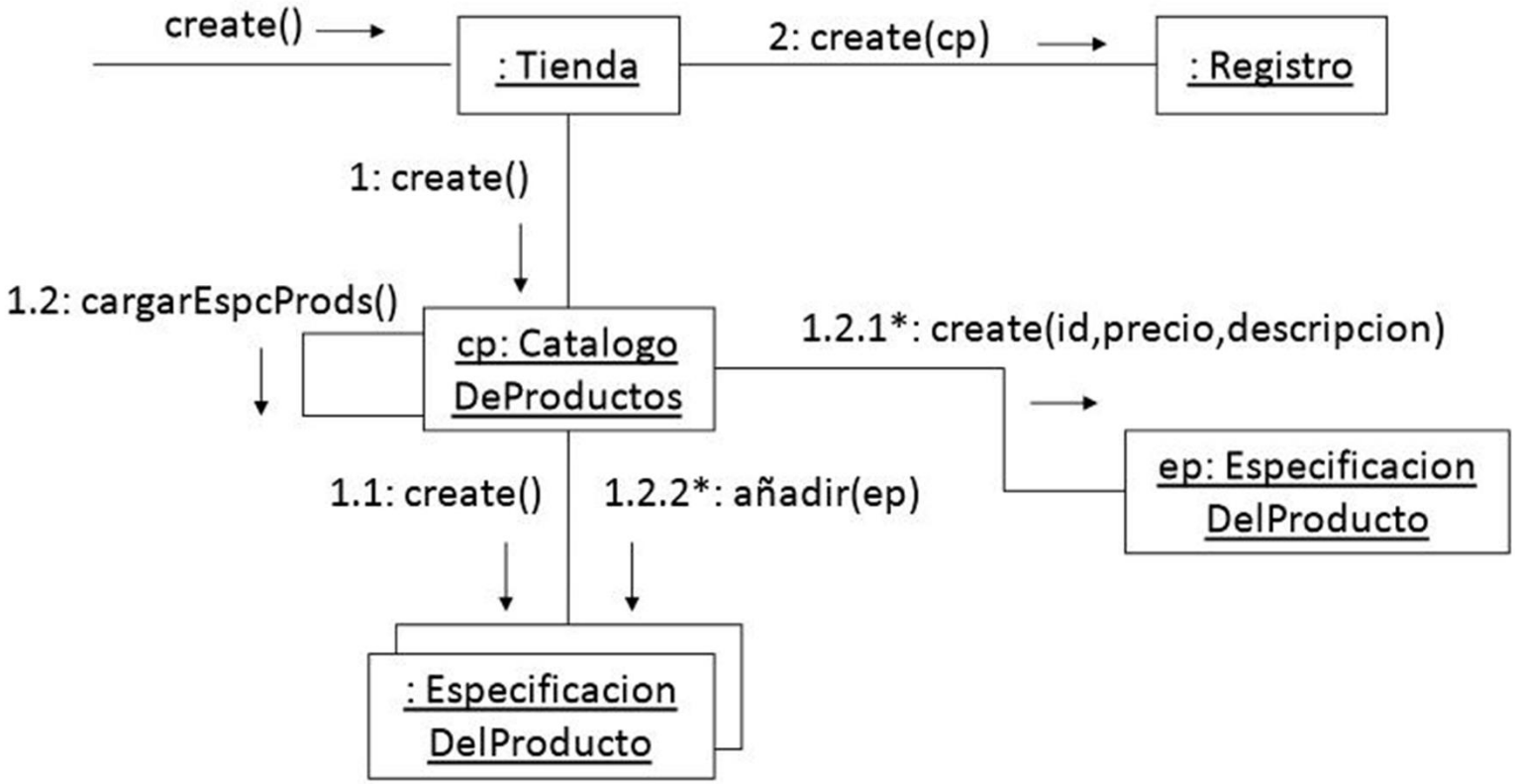
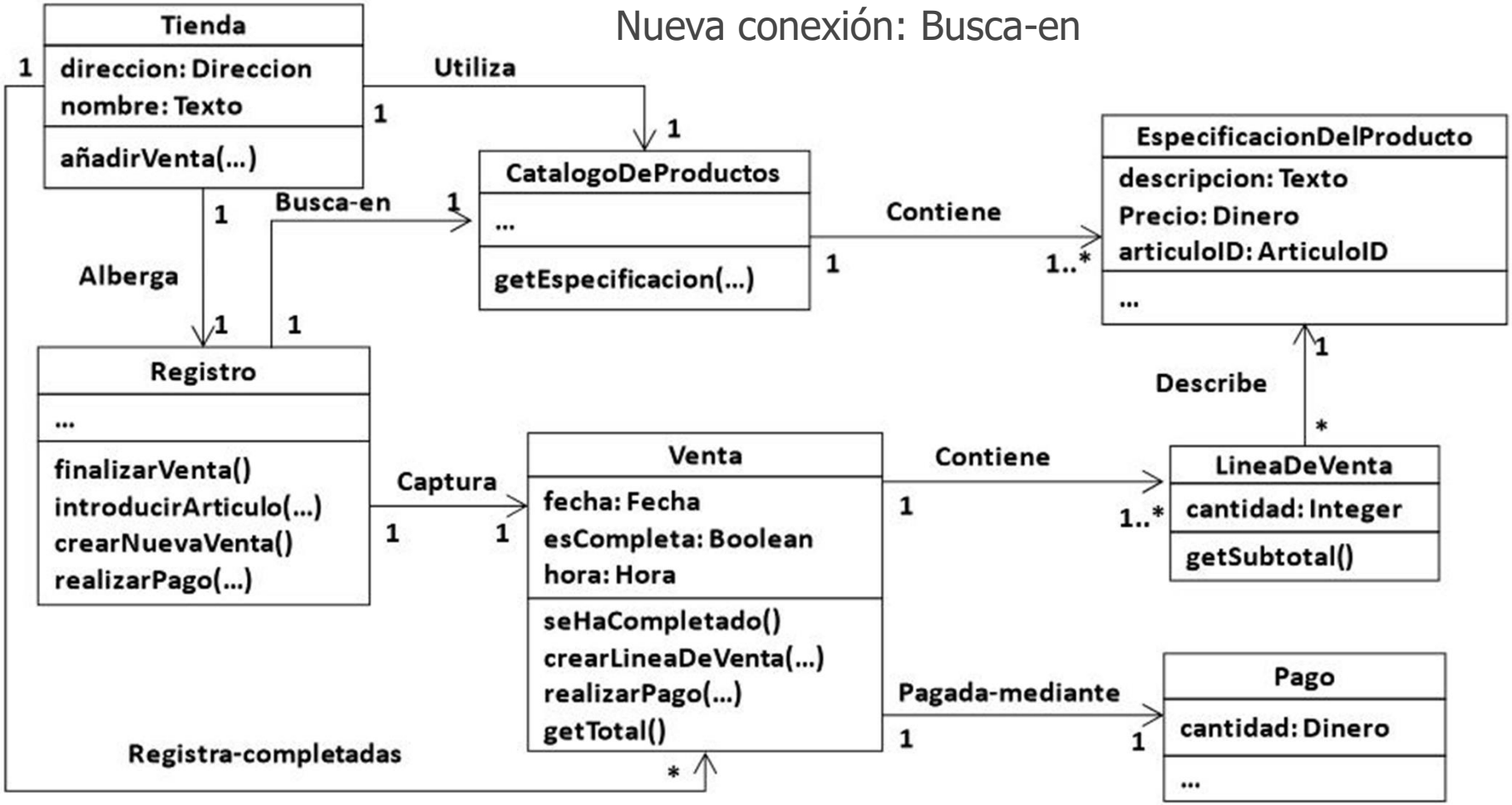
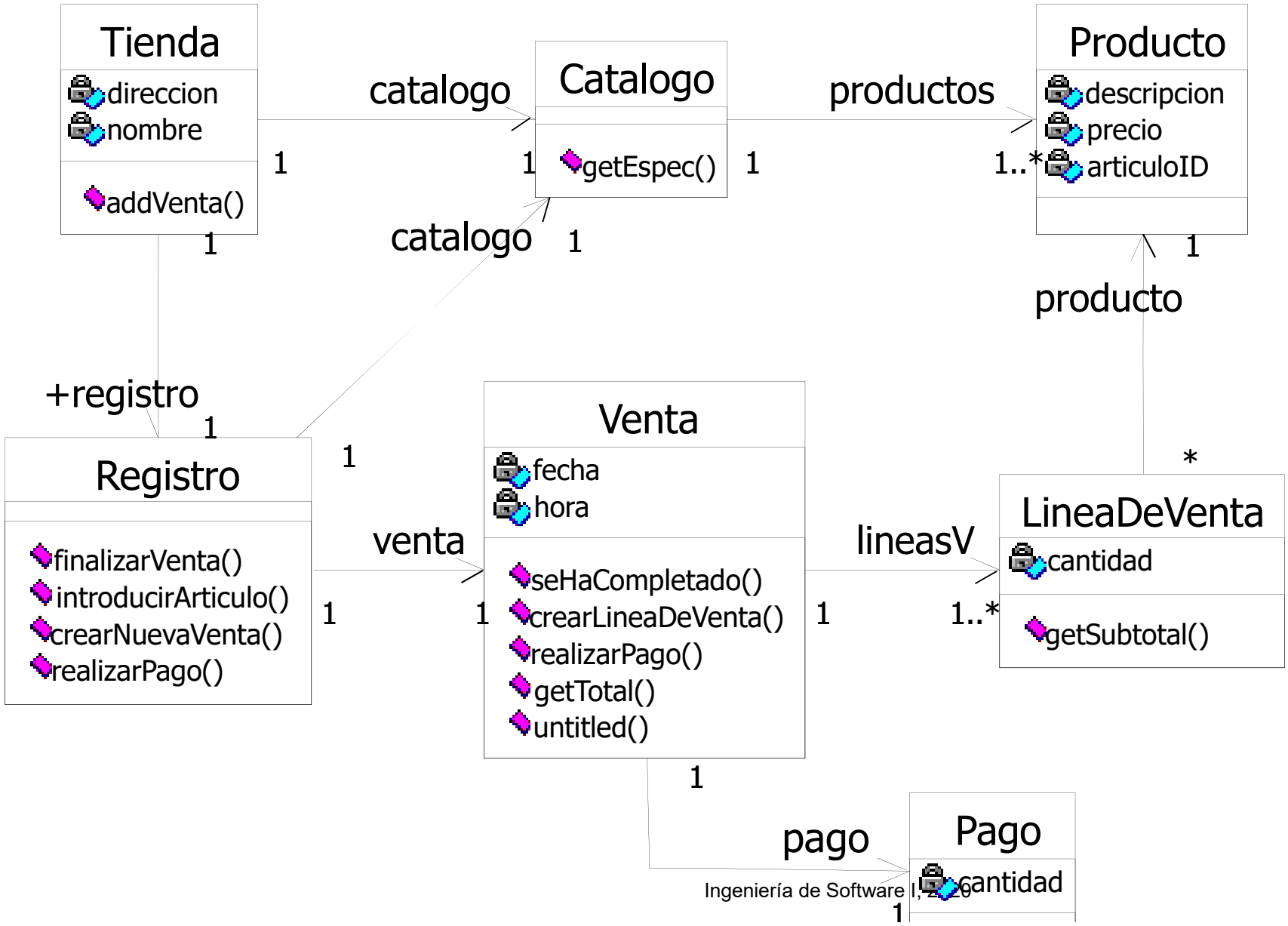


Diagrama de Clases del Diseño



DCD con nombres de rol



RUP Agil. Dónde estamos???

Fase de Elaboración. Iteración I

Transformación de los Diseños en Código

Referencias

- Los artefactos creados durante el Diseño: Diagramas de Interacción y Diagramas de Clases del Diseño, se utilizan como entradas en el proceso de generación de código.
- El RUP define el Modelo de Implementación >>> contiene artefactos como: código fuente, definiciones de bases de datos, etc.
- Los resultados del Diseño son un primer paso incompleto, durante la codificación y prueba se realizan gran cantidad de cambios.
- La implementación en un Lenguaje OO requiere de la escritura de código fuente para:
  - ✓Las definiciones de las clases e interfaces
  - ✓Las definiciones de los métodos.

Creación de las clases de Implementación a partir de los Diagramas de clases del Diseño

Clase con atributos simples

```
  public class LineaDeVenta
{
 private int cantidad;

   public LineaDeVenta(Producto espec, int cant) {...}

 public Dinero getSubtotal() {...}
...
}
```

Clase con atributos simples

- Se incorpora en el Código fuente el constructor LineaDeVenta().
- Se deriva de un mensaje create(espec, cant) a la LineaDeVenta en un Diagrama de Interacción *introducirArticulo*
- Esto indica en Java que se requiere de un constructor que soporte estos parámetros.
- A menudo se excluye el método *create* de los Diagramas de Clases.

Clase con atributos referencia

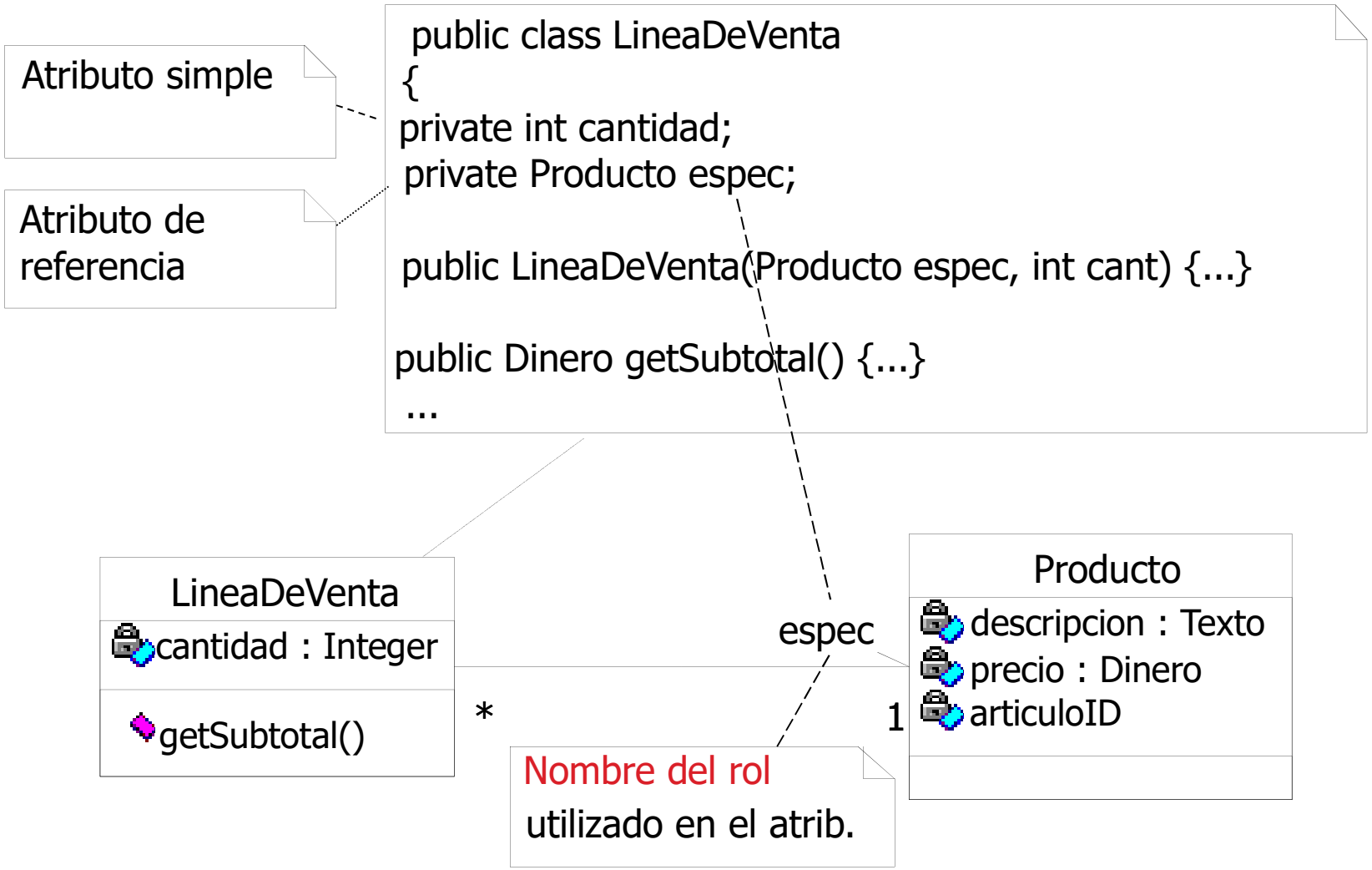
- Un atributo de referencia es un atributo que referencia a otro objeto complejo, no a un tipo primitivo.
- *Los atributos de referencia de una clase se deducen de las asociaciones y la navegabilidad en un Diagrama de Clases*.
- En el ejemplo anterior, dado el sentido de la navegabilidad, existe un atrtibuto de referencia en la clase *LineaDeVenta* que hace referencia a la instancia de *Producto.*
- Los atrributos de referencia de una clase a menudo están implícitos en un DCD, no se declaran como atributos en la sección de atributos de la clase.
- Durante el diseño, las referencias se observan sobre la línea de asociación como nombre de rol.

Clase con atributos referencia
Roles

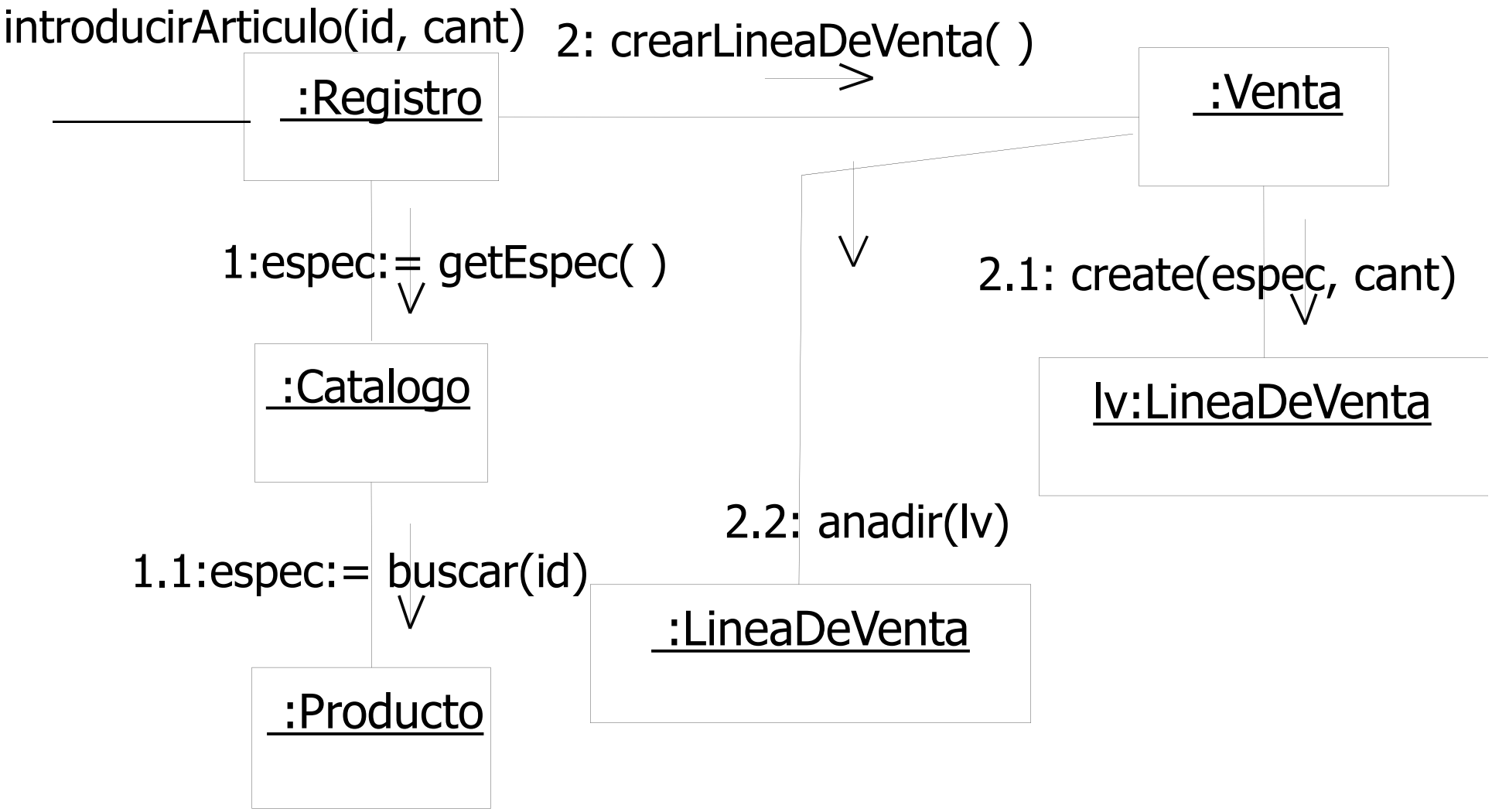
- En un Diagrama estático de clases, cada extremo de la asociación se denomina *rol*

- *Si se coloca el nombre de rol en el DCD, se utiliza como base para el nombre del atributo de referencia durante la generación de código.*

Clase con atributos referencia



Clase con atributos referencia

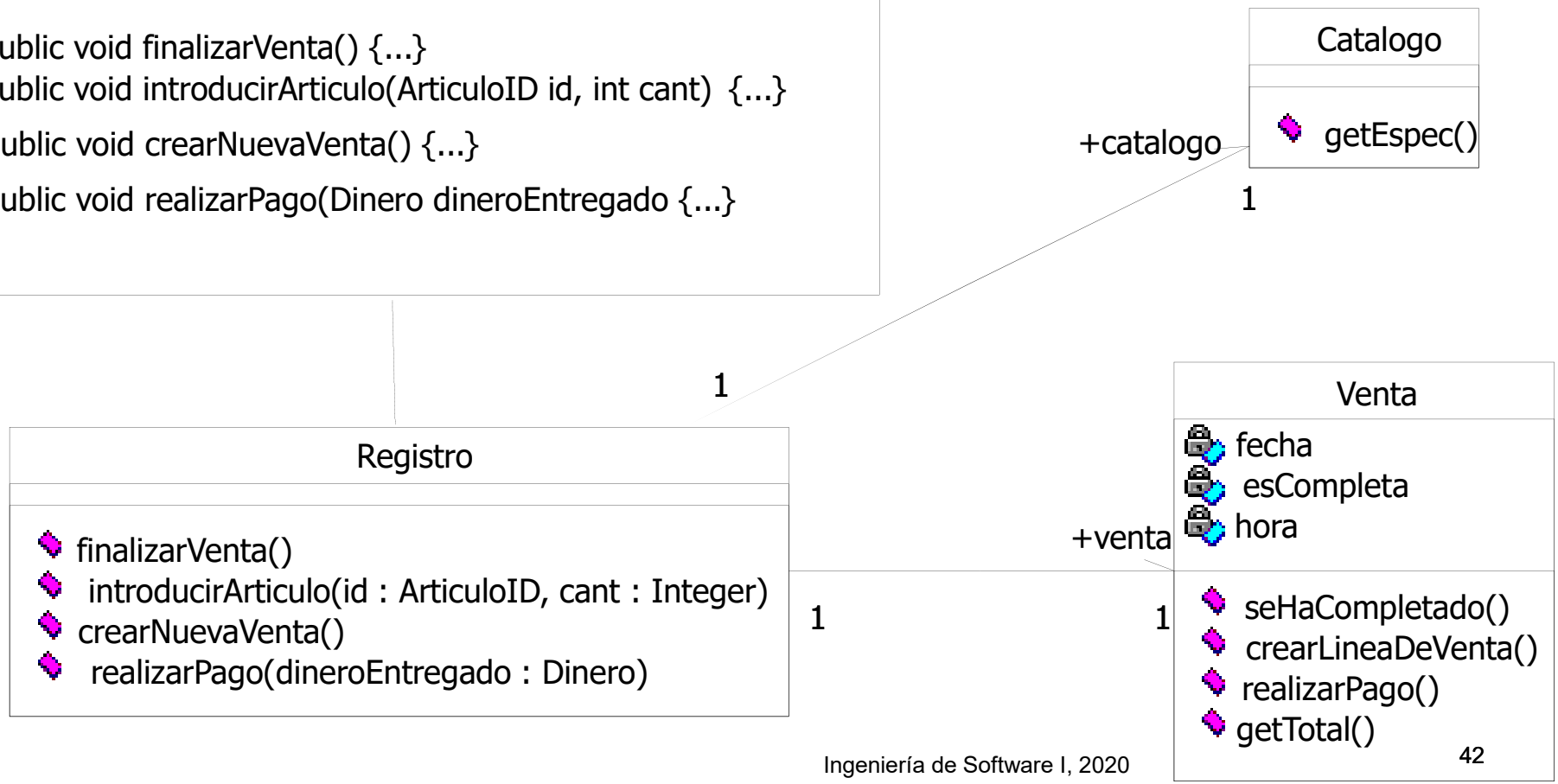


Métodos a partir de los DI

```
 public class Registro
{
 private Catalogo catalogo;
 private Venta venta;

 public Registro(Catalogo cat) {...}

 public void finalizarVenta() {...}
 public void introducirArticulo(ArticuloID id, int cant)  {...}
 public void crearNuevaVenta() {...}
 public void realizarPago(Dinero dineroEntregado {...}
}
```



El método IntroducirArtículo

Se envía el mensaje introducirArticulo a la clase Registro; por lo tanto el método se define en dicha clase.

public void introducirArticulo(ArticuloID art, int ctd)

Mensaje 1: Se envía getEspec al Catalogo para recuperar una especificación del Producto.

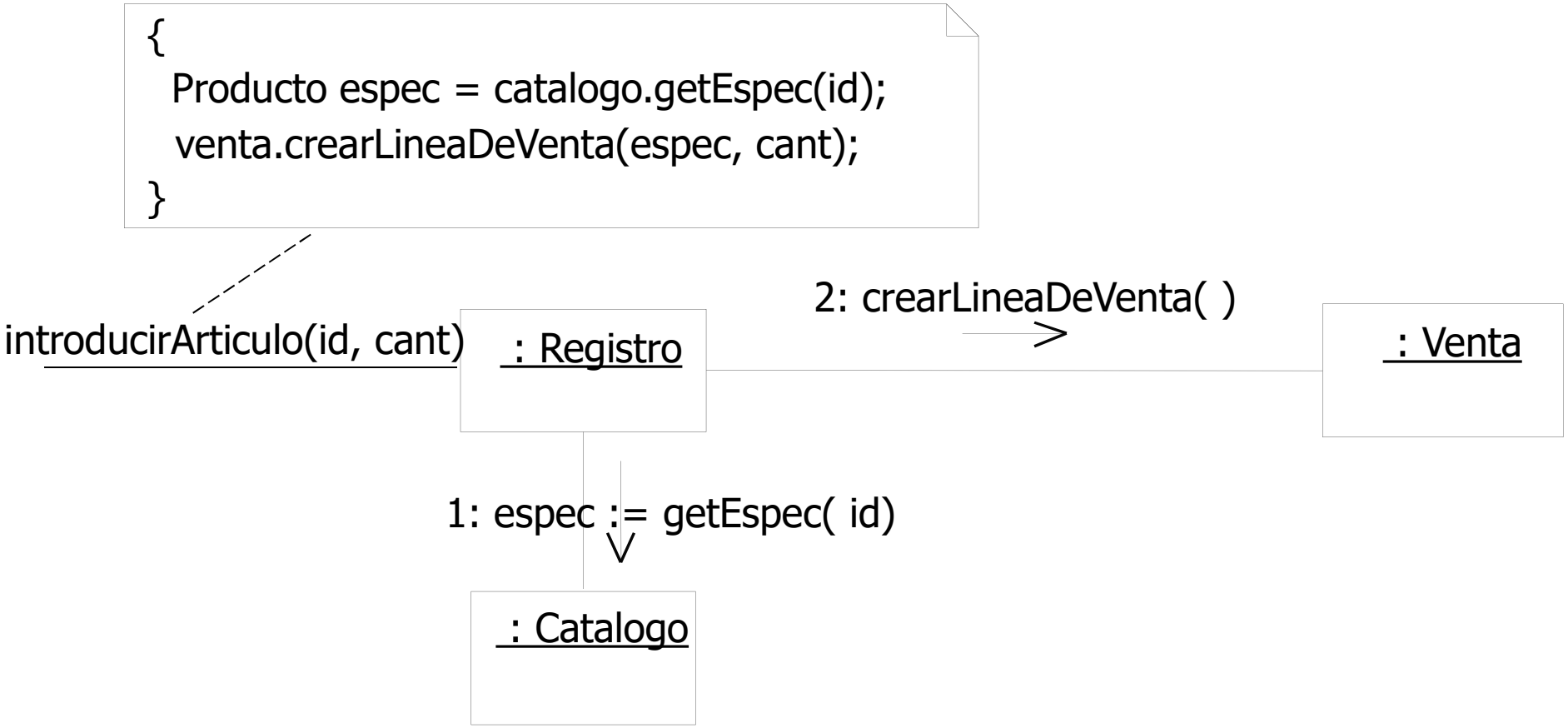
Producto espec = catalogo.getEspec(articuloID)

Mensaje 2: Se envía crearNuevaVenta a la Venta.

venta.crearLineaVenta(espec, cant)

Cada mensaje de la secuencia del método que se muestra en el Diagrama de Interacción se transforma en una sentencia Java

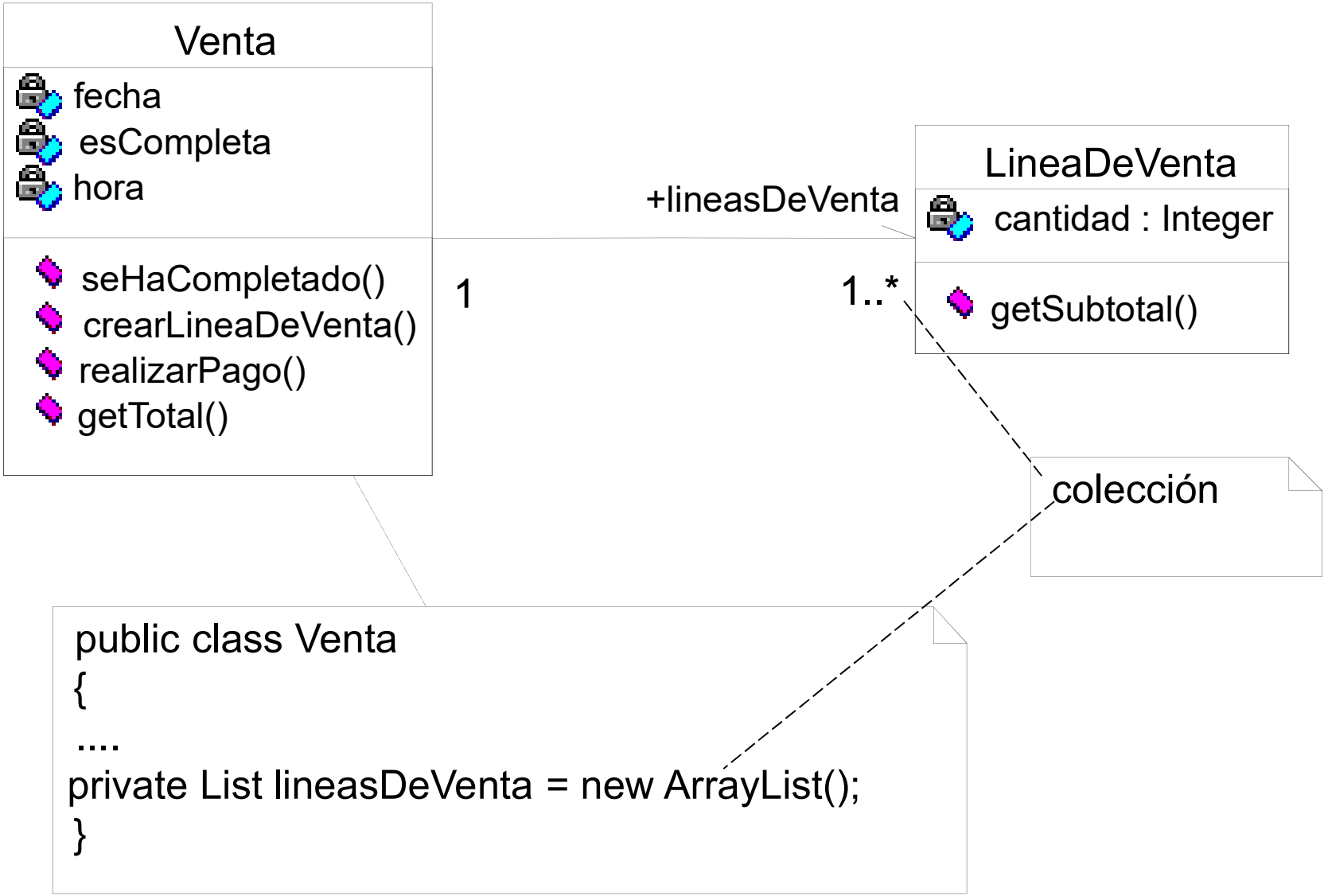
El Metodo introducirArticulo y su relación con el DI



Clases contenedoras

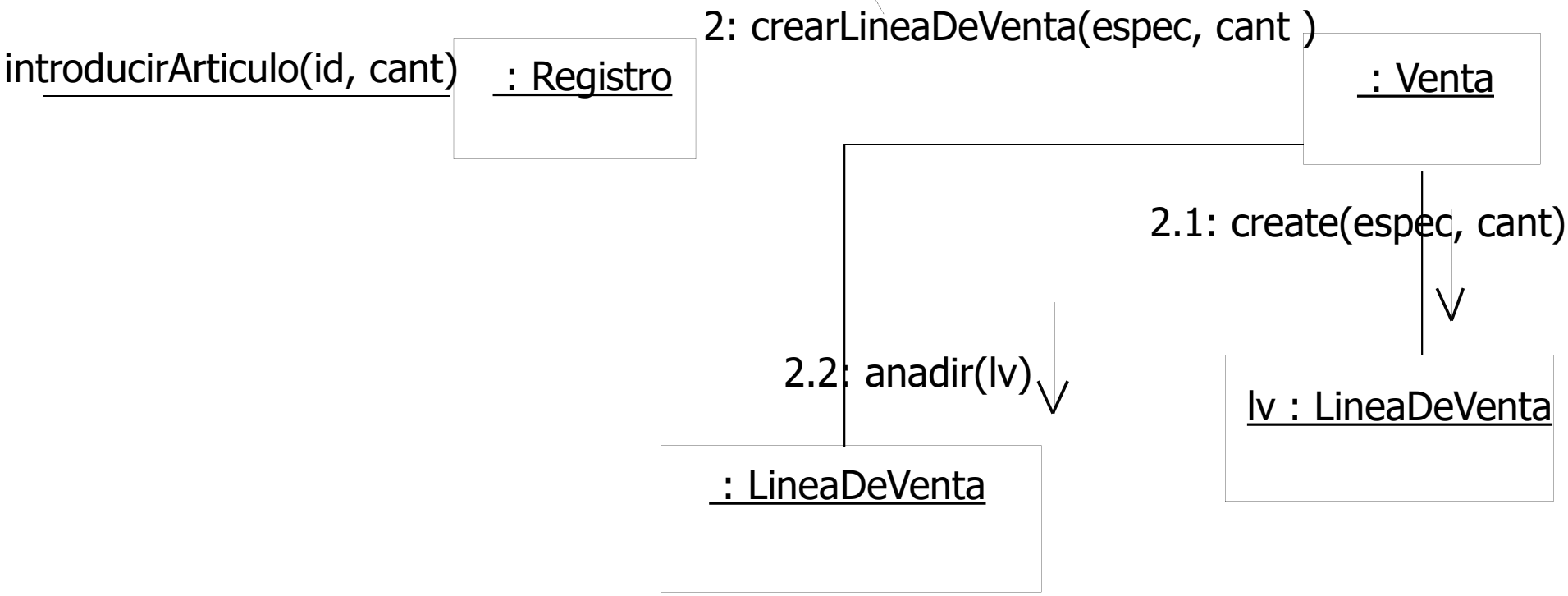
- A menudo es necesario que un objeto mantenga visibilidad a un grupo de otros objetos.
- *Esto se hace evidente a partir de la multiciplidad en el Diagrama de clases (cuando es mayor que uno)*
- Por ej., una venta debe mantener visibilidad a un conjunto de líneas de Venta.
- En los lenguajes OO, normalmente se implementan estas relaciones mediante un contenedor o colección (dependiendo del Lenguaje).
- La clase del lado 1 de la relación contiene un atributo referencia que apunta a la instancia de contenedor o colección del lado muchos.
- Los requerimientos deteminan el tipo de clase contenedor que seleccione

Clases Contenedoras



El método crearLineaDeVenta

```
{
   lineasDeVenta añadir(new LineaDeVenta(espec, cant);
}
```



Clases contenedoras

- Es necesario implementar las clases desde la menos a la más acoplada
- En el siguiente ejemplo >>>
  - ✓Se pueden implementar primero Pago o Producto.
  - ✓Luego se pueden implementar las clases que dependen de la implementación anterior: Catálogo o LineaDeVenta.

- Una buena práctica es *Programar Probando Primero >>>*
- *Se escribe el código de las pruebas de unidad antes del código que se va a probar*

Clases contenedoras

- Antes de programar la clase Venta, se escribe un método de Prueba de unidad en la clase PruebaVenta que hace lo siguiente:

- Crea una nueva venta

- Le añade algunas líneas de venta

- Solicita el total y comprueba si tiene el valor esperado.

Programar probando. Ejemplo

```
public class PruebaVenta extends TestCase
{
   //...
   public void pruebaTotal()
   {
         //inicializa la prueba
         Dinero total = new Dinero(7.5);
         Dinero precio = new Dinero(2.5);
         ArticuloID id = new ArticuloID (1);
         Producto espec;
         espec = new Producto(id, precio, "producto 1");
         Venta venta = new Venta();
         //añade artículos
         Venta.crearLineaDeVenta(espec, 1);
         Venta.crearLineaDeVenta(espec, 2);

         //comprobar que el total es 7.5
         assertEquals(venta.getTotal(), total);
   }
}
```

Clase Pago

```
public class Pago
{
   private Dinero cantidad;
   public Pago(Dinero dineroEntregado) {cantidad =
   dineroEntregado;}
   public Dinero getCantidad() { return cantidad;}
}
```

Clase Catalogo

```
public class Catalogo
{
   private Map productos = new HashMap();
   public Catalogo()
   {
         //datos Ejemplo
         ArticuloID id1 = new ArticuloID(100);
         ArticuloID id2 = new ArticuloID(200);
         Dinero precio = new Dinero(3);
         Producto ep;
         ep = new Producto(id1, precio, "producto 1");
         productos.put(id1, ep);
         ep = new Producto(id2, precio, "producto 2");
         productos.put(id2, ep);
   }
   public Producto getEspec(ArticuloID id) {
          return (Producto) productos.get(id);
}
```

Clase Registro

```
public class Registro
{
   private Catalogo catalogo;
   private Venta venta;
   public Registro(Catalogo catalogo) {        this.catalogo =
   catalogo;}
   public void finalizarVenta() {venta.seHaCompletado();}
   public void introducirArticulo(ArticuloID id, int cantidad) {
         Producto espec = catalogo.getEspec(id);
        venta.crearLineaDeVenta(espec, cantidad);
   }
   public void crearNuevaVenta() {
        venta = new Venta();
   }
   public void realizarPago(Dinero dineroEntregado) {
        venta.realizarPago(dineroEntregado);
   }
}
```

Clase Producto

```
public class Producto
{
   private ArticuloID;
   private Dinero precio;
   private String descripcion;

   public Producto(ArticuloID id, Dinero precio, Strting
   descripcion) {        this.id = id;
         this.precio = precio;
         this.descricion = deescripcion;
   }
   public ArticuloID getArticuloID() {return id;}
   public Dinero getPrecio() {return precio;}
   public String getDescripcion() {return descripcion;}
}
```

Clase Venta

```
public class Venta
{
   private List lineasDeVenta = new ArrayList();
   private Date fecha = new Date();
   private Boolean esCompleta() = false;
   private Pago pago;

   public Dinero getDevolucion()
   {
         return pago.getCantidad().minus(getTotal() );
   }
   public void seHaCompletado() {esCompleta = true; }
   public void esCompleta() {return esCompleta; }
   public void crearLineaDeVenta(Producto espec, int cantidad) {
         lineasDeVenta.add(new LineaDeVenta(espec, cantidad);
   }
```

>>>>>

Clase Venta

```
public class Venta
{
   public Dinero getTotal()
   {
         Dinero total = new Dinero();
         Iterator i = lineasDeVenta.iterator();
         while(i.hasNext())
         {
                  LineaDeVenta ldv = (LineaDeVenta) i.next();
                  total.add(ldv.getSubtotal() );
         }
         return total;
   }
   public void realizarPago(Dinero dineroEntregado)
   {
         pago = new Pago (dineroEntregado);
    }
```

Clase LineaDeVenta

```
public class LineaDeVenta
{
   private int cantidad;
   private Producto espec;

   public LineaDeVental(Producto espec, int cantidad)
   {
         this.espec = espec;
         this.cantidad = cantidad;
   }
   public Dinero getSubtotal()
   {
         return espec.getPrecio().times(cantidad);
    }
}
```

Clase Tienda

```
public class Tienda
{
   private Catalogo catalogo = new Catalogo();
   private Registro registro = new Registro(catalogo);

   public Registro getRegistro() { return registro; }
}
```

Referencias

- El Proceso Unificado de Desarrollo de Software. Jacobson, Booch, Rumbaugh. Addison Wesley, 2000
- Applying UML and Patterns 3ª ed. Larman. Addison Wesley, 2004