IIC2143 - Ingeniería de Software

Introducción al diseño orientado a objetos y a UML

2-2015

Yadran Eterovic (yadran@ing.puc.cl)

UML es un lenguaje visual para especificar, diseñar y documentar software OO

UML es una "familia" de 13 diagramas:

los lenguajes de programación no tienen un nivel de abstracción que facilite discusiones sobre diseño

Es un estándar relativamente abierto del OMG:

consorcio abierto de compañías formado para definir estándares para la interoperabilidad de sistemas OO

Nació en 1997:

de la unificación de varios lenguajes de modelado gráfico OO del período 1985 – 1995

UML **no es** un método de modelamiento

UML sólo proporciona una sintaxis visual:

aunque, naturalmente, hay algunos aspectos metodológicos implícitos en los elementos que forman un modelo UML

UML **no es** un método de desarrollo de sistemas

UML no está vinculado a ningún método de desarrollo

… o ciclo de vida específico:

- puede ser usado con cualquiera de los métodos existentes

Hay dos formas habituales de usar UML

1) Para dibujar *bosquejos exploratorios* —la más habitual:

- diagramas informales o incompletos, hechos para explorar las partes difíciles del problema o del espacio de soluciones

2) Para dibujar *planos de análisis* y *planos de diseño* (más definitivos):

- diagramas suficientemente detallados y completos
- para generar partes del código sin muchos problemas
- para visualizar y entender código existente (ingeniería reversa)

"Podemos modelar software como colecciones de objetos interactuantes"

Esta premisa básica del UML concuerda con los lenguajes y sistemas de software OO,

- … y también funciona para procesos de negocios, procesos productivos, y otros tipos de aplicaciones

Un objeto es una agrupación cohesiva de dos tipos de propiedades:

- datos —los objetos contienen información;
- comportamiento —los objetos pueden realizar funciones, responder consultas, cumplir responsabilidades

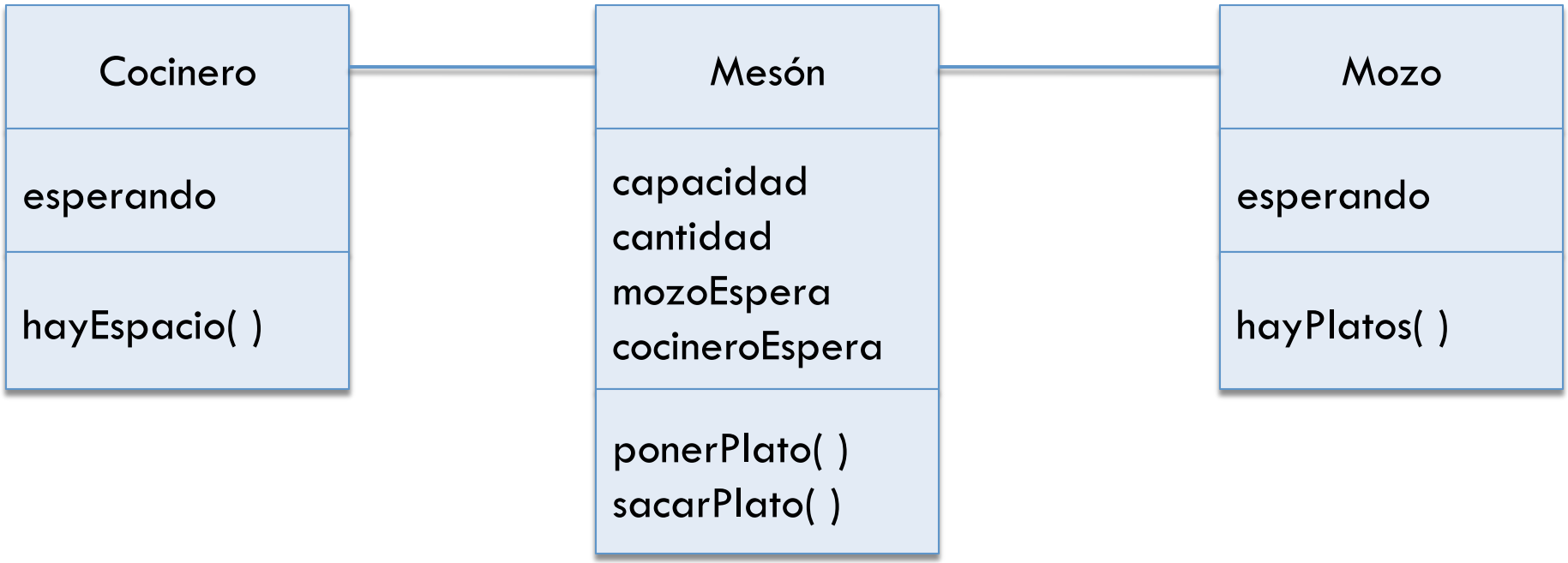
Un objeto es una agrupación cohesiva de datos y comportamiento

Un modelo en UML tiene dos aspectos casi inseparables, y uno no está completo sin el otro:

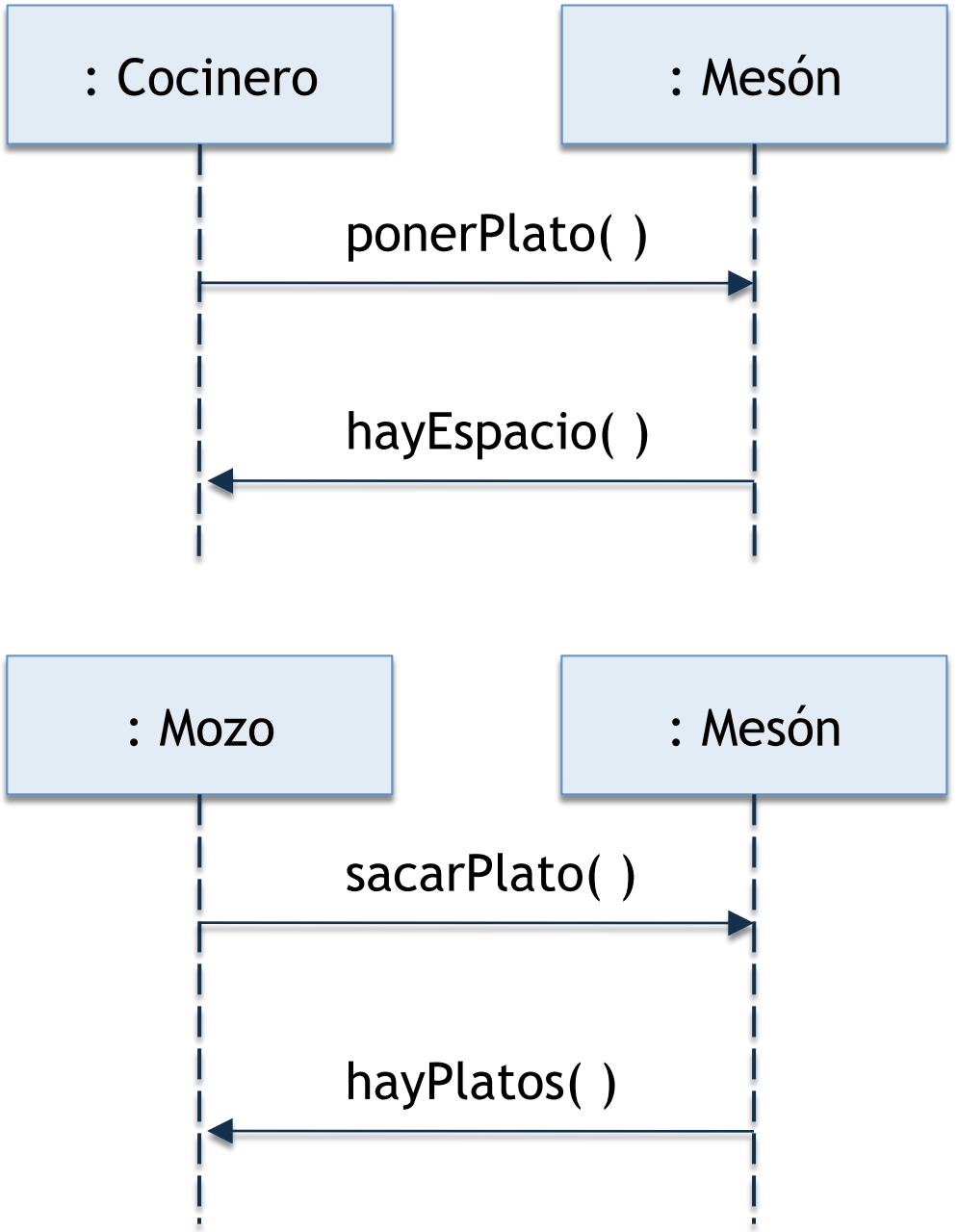
- la estructura estática describe qué tipos de objetos son importantes para modelar el sistema y cómo están relacionados —ver próxima diap.
- el comportamiento dinámico describe cómo los objetos interactúan entre ellos para entregar funcionalidad —ver diap. #9

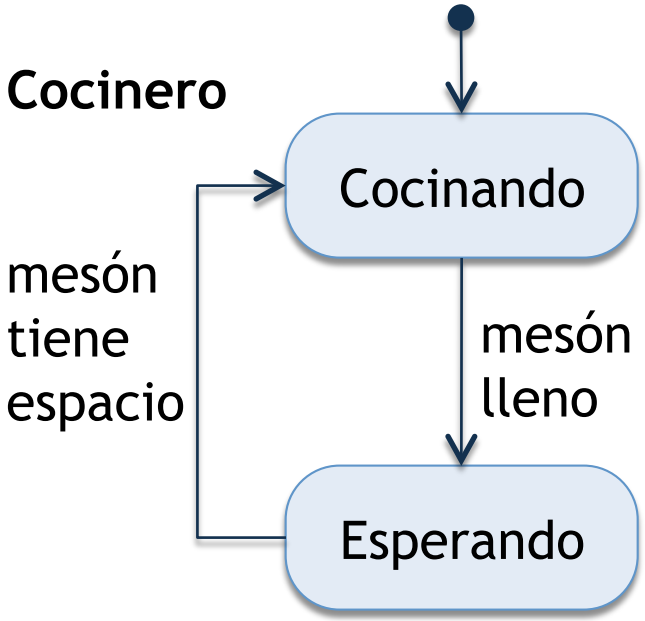
  … y los ciclos de vida de los objetos a medida que cumplen sus responsabilidades —ver diap. #10

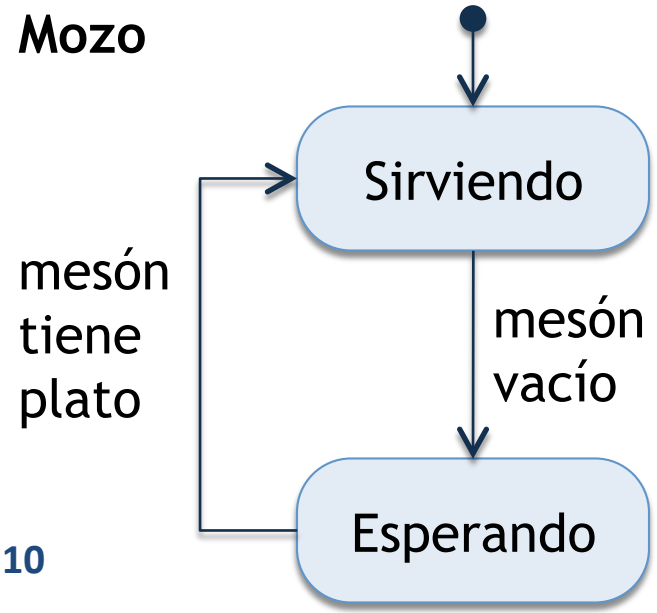
Ejemplo de un diagrama de clases

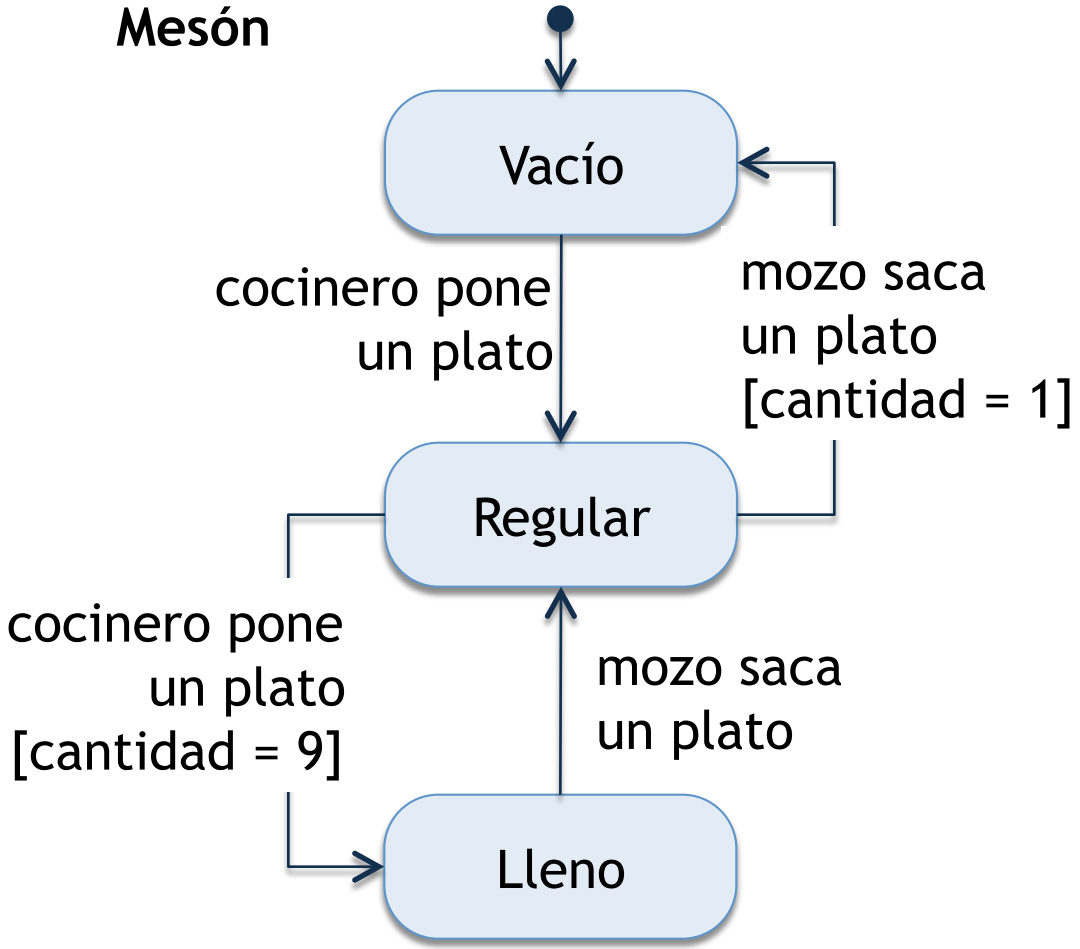


Ejemplos de diagramas de secuencia









Ejemplos de diagramas de estado

UML 2.0 [2005]: Colección de 13 diagramas

UML permite modelar dos aspectos de un sistema :

- aspectos estructurales o estáticos;
- aspectos de comportamiento o dinámicos.

Diagramas estructurales:

clases, estructuras compuestas, objetos, componentes, instalación, paquetes.

Diagramas de comportamiento:

casos de uso, actividades, máquinas de estados, comunicación, secuencia, tiempo, resumen de interacción

Los diagramas más usados:
Casos de uso, *clases* y *secuencia*

Diagrama de casos de uso:

identifica los usos que los usuarios —actores— pueden hacer de un sistema

Diagrama de clases:

describe los tipos de objetos —clases— de un sistema (ya sea computacional o no), sus propiedades y relaciones permanentes

Diagrama de secuencia:

describe las interacciones entre los objetos de un sistema, en tiempo de ejecución

El *diseño* pone el énfasis en una solución conceptual que satisfaga los requisitos

Podemos describir, p.ej., el esquema de una base de datos

Podemos hacer, p.ej., diseño orientado a objetos, diseño de la base de datos

La solución es *conceptual* porque no incluye detalles de "bajo nivel"; p.ej.,

- elección de las estructuras de datos
- elección del lenguaje de programación

(*Diseño* también es el proceso que lleva a esta solución conceptual)

El software es un producto de ingeniería

… como los rascacielos, los aviones, los puentes colgantes, y los teléfonos celulares

La producción de estos productos sigue un ciclo de vida:

- primero, se especifica
- luego, se diseña
- finalmente, se fabrica o construye

Pero el ciclo de vida del software es distinto:

- se especifica, aunque apenas;
- luego, se implementa (o se trata de implementar), normalmente sin pasar por el diseño

El diseño es importante por varios motivos

¿Cómo sé si las especificaciones son técnicamente realistas?

¿Cómo puedo decidir entre diversas opciones de implementación?

¿Cómo puedo dividir la implementación de un sistema complejo entre varios programadores?

¿Cómo puedo facilitar la evolución de un sistema y su adaptación a nuevas necesidaes?

¿Qué es *diseño orientado a objetos*?

En **diseño OO**, ponemos el énfasis en definir objetos de software

… y cómo ellos colaboran para satisfacer los requisitos:

p.ej., en un sistema de reserva de vuelos, un objeto de software *Reserva* puede tener un atributo *confirmada* y un método *verDetalles( )*

Ahora veremos cómo representar diseños OO en UML

El diseño OO tiene **tres objetivos principales**

Identificar las **clases**

Identificar las **responsabilidades** de estas clases:

- **atributos**
- **métodos (operaciones)**

Identificar las **relaciones** entre estas clases

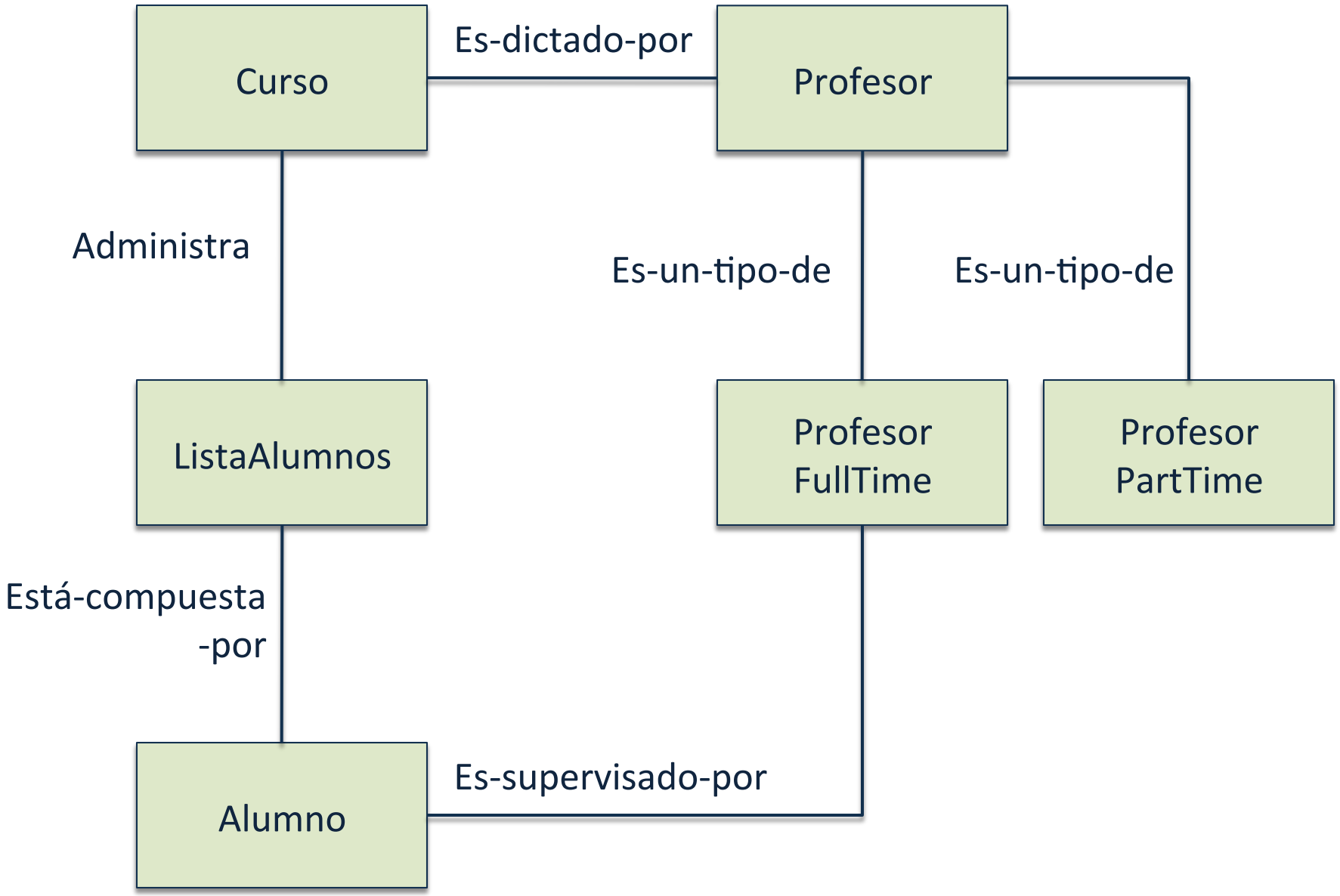
El proceso de diseño es iterativo

Los anteriores son *objetivos*, no pasos:

- típicamente, no es posible encontrar todas las clases primero, luego sus responsabilidades, y finalmente sus relaciones
- la identificación de una propiedad de una clase puede llevar al descubrimiento de otras clases o puede forzar cambios en otras clases

Los *diagramas de clases* pueden incluir diversos niveles de detalle

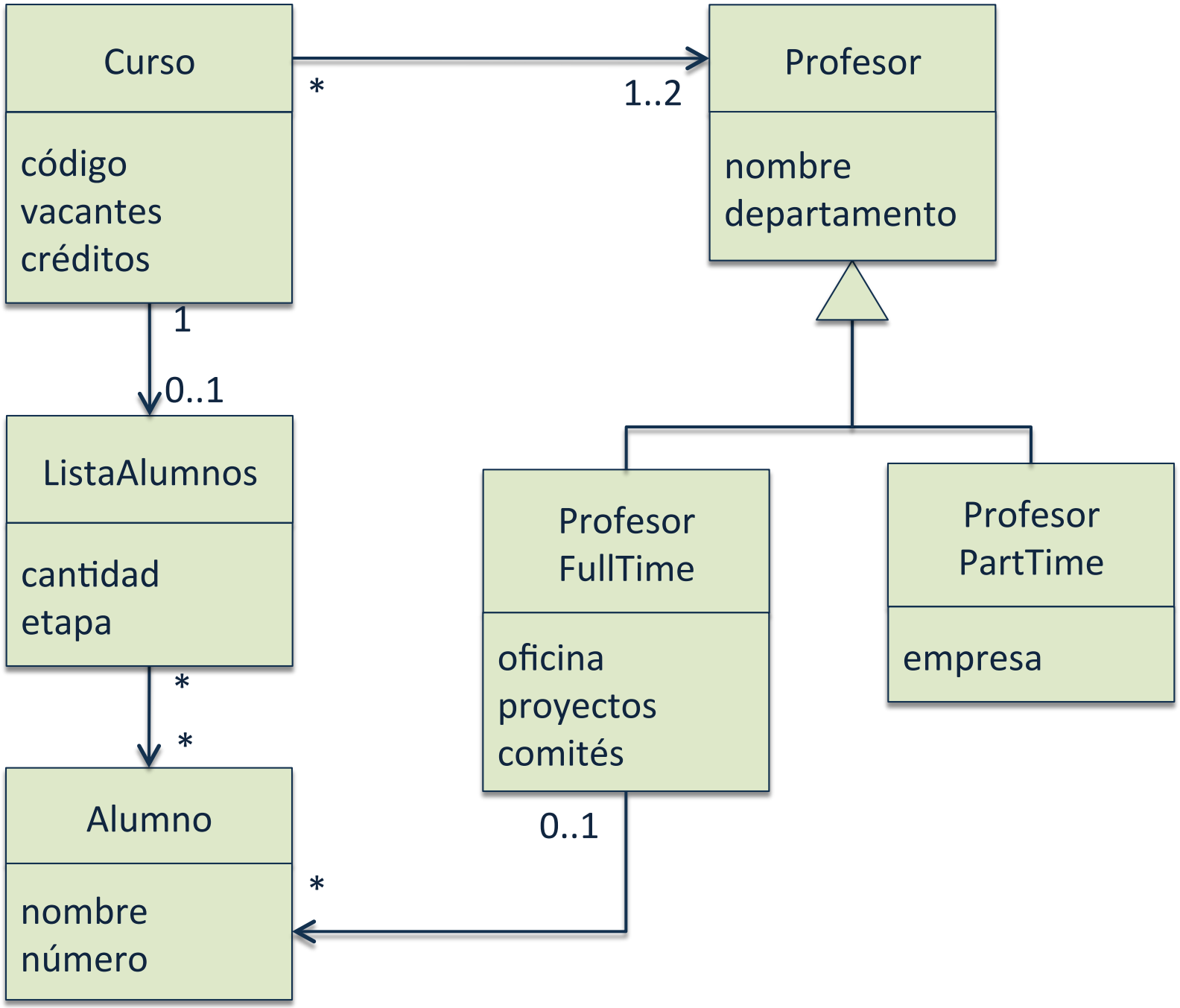
P.ej., la próxima diapositiva muestra un *diagrama de clases* que representa un *modelo del dominio* —producto de la fase de análisis— de cursos, alumnos y profesores en una universidad


Curso
Es-dictado-por
Profesor
Administra
Es-un-tipo-de
Es-un-tipo-de
ListaAlumnos
Profesor
FullTime
Profesor
PartTime
Está-compuesta
-por
Alumno
Es-supervisado-por

Un diseño de software se representa mediante un *diagrama de clases del diseño* (DCD)

P.ej., la próxima diapositiva muestra un DCD basado en el modelo de dominio anterior:

- muestra los atributos de las clases (también pueden mostrarse en el modelo de dominio)
- muestra las multiplicidades de las asociaciones (también pueden mostrarse en el modelo de dominio)
- muestra la navegabilidad de las asociaciones
- típicamente, no muestra los nombres de las asociaciones


Curso
código
vacantes
créditos
*
1..2
Profesor
nombre
departamento
1
0..1
ListaAlumnos
cantidad
etapa
*
*
Alumno
nombre
número
Profesor
FullTime
oficina
proyectos
comités
0..1
*
Profesor
PartTime
empresa

Los DCD's más detallados incluyen propiedades adicionales de las clases y las asociaciones

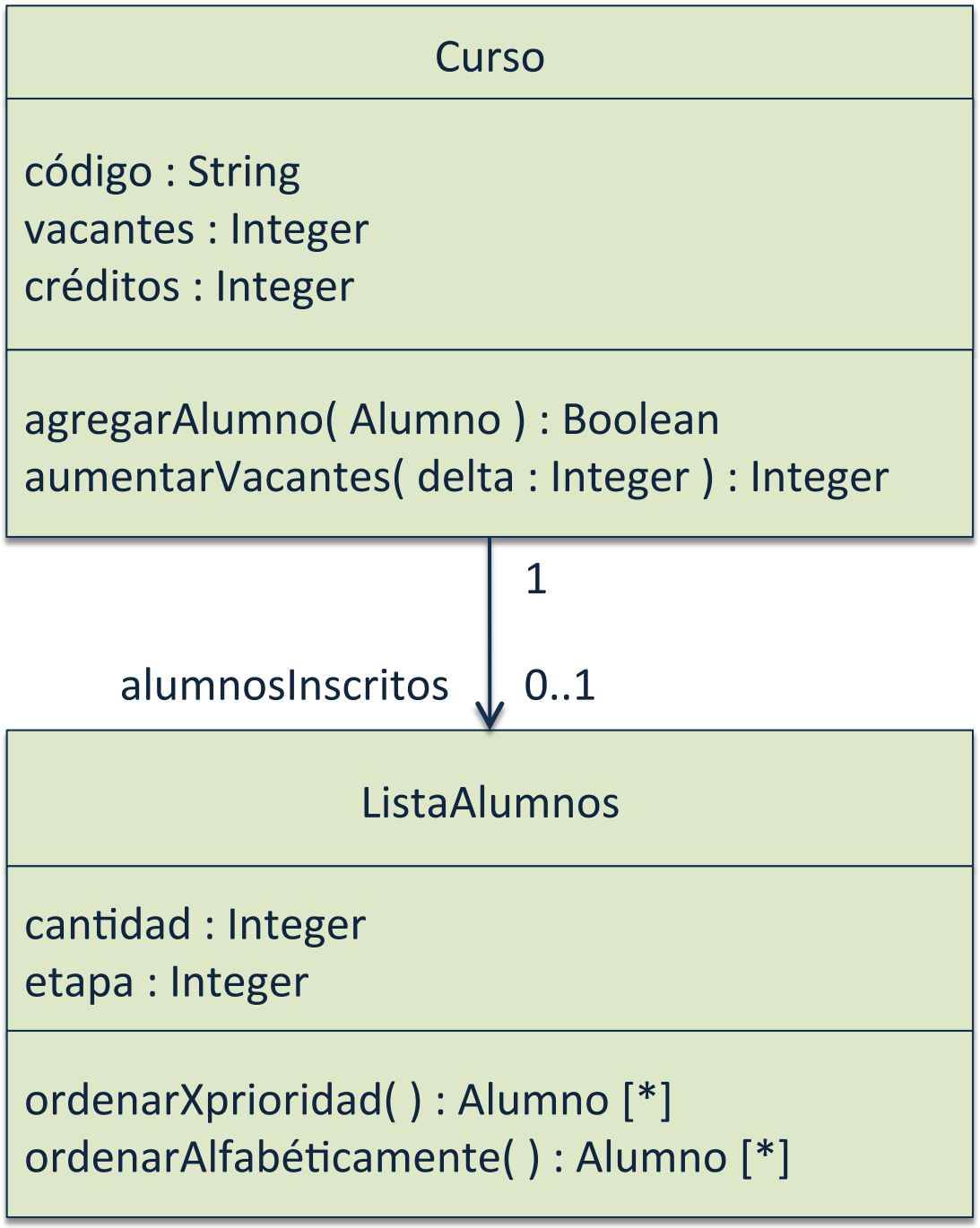
Las *operaciones* (o *métodos*) de las clases —ya sea sólo los nombres, o las firmas completas

Los *nombres* y las *multiplicidades* de los *roles* que desempeñan las clases en cada extremo de una asociación con respecto a la clase en el otro extremo

… (en lugar de los nombres de las asociaciones que van en el modelo del dominio)

P.ej., la próxima diap. muestra estas propiedades para una porción del DCD anterior:

- muestra las operaciones *agregarAlumno*( ), *aumentarVacantes*( ), *ordenarXprioridad*( ), *ordenarAlfabéticamente*( )
- muestra que la clase *ListaAlumnos* desempeña el rol de *alumnos inscritos* con respecto a la clase *Curso*


Curso
código : String
vacantes : Integer
créditos : Integer
agregarAlumno(Alumno) : Boolean
aumentarVacantes(delta : Integer) : Integer
1
alumnosInscritos
0..1
ListaAlumnos
cantidad : Integer
etapa : Integer
ordenarXprioridad( ) : Alumno [*]
ordenarAlfabéticamente( ) : Alumno [*]

Las operaciones pueden incluir información adicional al nombre

Tipo de dato del resultado producido por la operación

Nombre y *tipo de dato* de los parámetros de la operación —si se especifica los parámetros, los nombres son, nuevamente, opcionales

p.ej., *agregarAlumno*(*Alumno*) : *Boolean*

agregarAlumno(*a* : *Alumno*) : *Boolean*

En un DCD hay dos grandes tipos de relaciones entre las clases

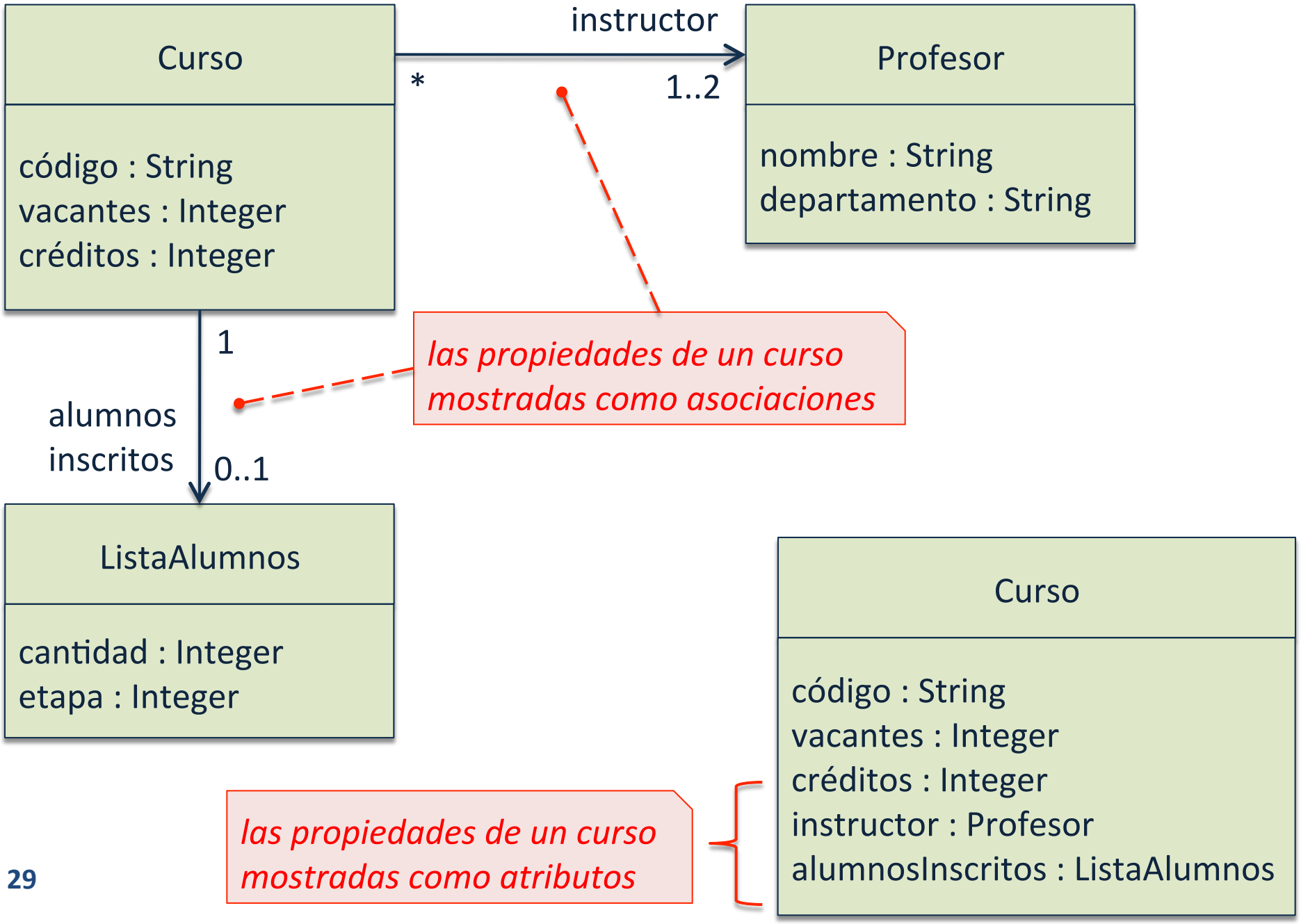
Asociaciones —denotadas por flechas cuyos extremos pueden indicar roles y multiplicidades

Generalizaciones (o **herencia**) —denotadas por un triángulo (Δ), en cuyo vértice superior descansa la clase más general

Importante: atributos y asociaciones son **dos formas de representar lo mismo**

En UML, *atributos* y *asociaciones* son dos formas de representar las propiedades cuyos valores están almacenados (no son métodos ni operaciones) en los objetos:

- los **atributos** son las propiedades cuyos valores están almacenados mediante *subobjetos* —en general, son de tipos simples: *Integer*, *String*, etc;
- las **asociaciones** son las propiedades cuyos valores están almacenados mediante *referencias* —en general, son otros objetos


Curso
código : String
vacantes : Integer
créditos : Integer
instructor
*
1..2
Profesor
nombre : String
departamento : String
1
alumnos
inscritos
0..1
las propiedades de un curso
mostradas como asociaciones
ListaAlumnos
cantidad : Integer
etapa : Integer
Curso
código : String
vacantes : Integer
créditos : Integer
instructor : Profesor
alumnosInscritos : ListaAlumnos
las propiedades de un curso
mostradas como atributos

Las propiedades —atributos y asociaciones— tienen nombres

El nombre de los atributos es obligatorio:

- opcionalmente, se puede mostrar el tipo de datos (o clase), la multiplicidad, etc.

El nombre de las asociaciones es opcional:

- en un DCD, normalmente se indica el *rol* que desempeñan las instancias de la clase en un extremo de la asociación con respecto a las instancias de la clase en el otro extremo
- p.ej., un *Profesor* es el *instructor* de un *Curso*, y una *ListaAlumnos* contiene los *alumnos inscritos* en un *Curso*

Las asociaciones pueden refinarse

Relación de **agregación** (*Universidad - Profesor*):

- es un tipo de relación algo vaga del todo con sus partes
- el todo usa los servicios de sus partes
- una parte puede ser compartida por varios “todos”

Relación de **composición** (*Universidad - Facultad*):

- también relaciona al todo con sus partes, pero de manera más estrecha y precisa
- las partes no tienen vida independiente fuera del todo
- cada parte pertenece a lo más a un todo

Agregación: Una Universidad es una agregación de (está formada por) varios profesores que trabajan para la universidad; los profesores pueden trabajar para otras universidades.

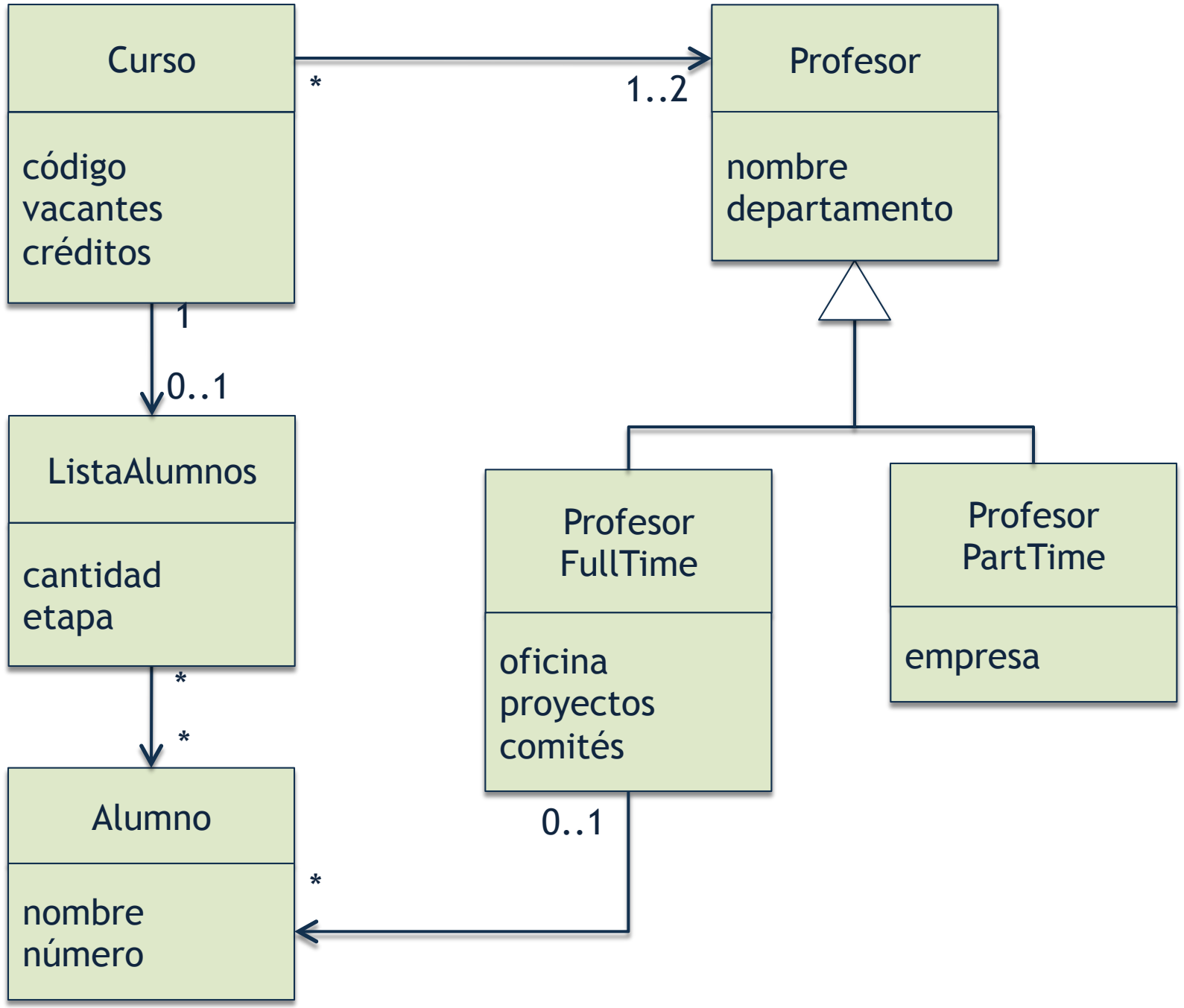
Composición: Una Universidad es una composición de varias facultades que son parte de la universidad; las facultades no existen fuera de la universidad.

Las asociaciones en un DCD son unidireccionales

Sólo importa el rol que desempeña la clase en el extremo con la punta de flecha con respecto a la clase de partida, y no vice versa

P.ej., en la próxima diapositiva:

- para un curso es importante el profesor que lo dicta y la lista de alumnos inscritos —*Profesor* y *ListaAlumnos* son propiedades de *Curso*
- para un profesor, o una lista de alumnos, el curso no está representado como una propiedad — *Curso* no es una propiedad de *Profesor* ni de *ListaAlumnos*


Curso
código
vacantes
créditos
*
1..2
Profesor
nombre
departamento
1
0..1
ListaAlumnos
cantidad
etapa
*
*
Alumno
nombre
número
*
Profesor
FullTime
oficina
proyectos
comités
0..1
Profesor
PartTime
empresa

Las *operaciones* son las acciones que una clase sabe llevar a cabo

Corresponden a las responsabilidades de hacer que tiene una clase:

- en OO, las llamamos **métodos**

La especificación de una operación en un DCD incluye lo siguiente (diaps. # 25 y 37):

- *nombre* (obligatorio)
- *lista de parámetros* (opcional)
- *tipo de datos del valor de retorno* (opcional)

Usamos *generalización* (y *herencia*) cuando hay diferencias pero también similitudes

Propiedades tanto de profesores full time como part time (próxima diap.):

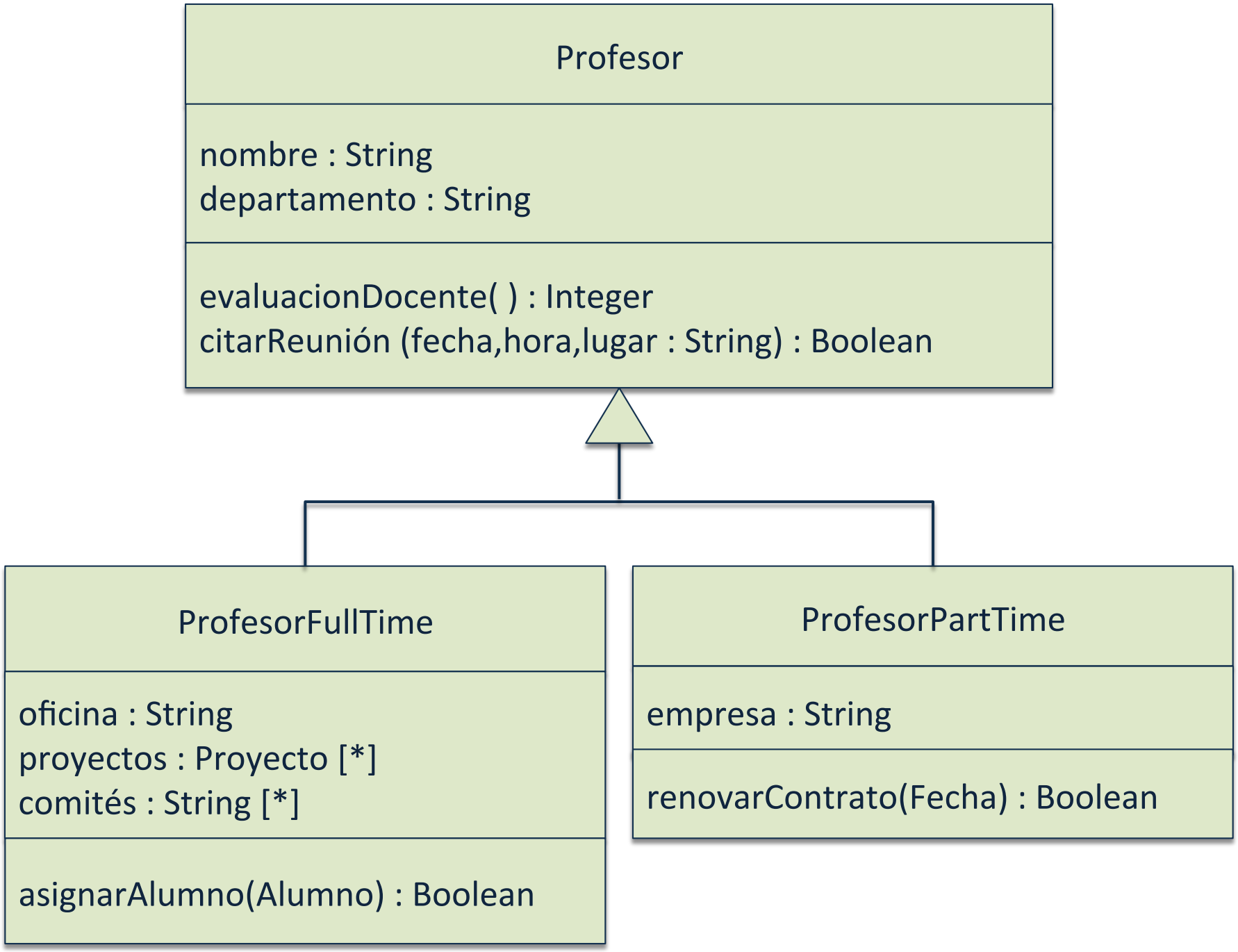
nombre, departamento, y formas para obtener su calificación docente y citarlo a reunión

Propiedades sólo de profesores part time:

la empresa en que trabajan full time, y una forma para renovarles el contrato

Propiedades sólo de profesores full time:

oficina que ocupan, proyectos y comisiones en que participan, y una forma para asignarles alumnos


Profesor
nombre : String
departamento : String
evaluacionDocente( ) : Integer
citarReunión (fecha,hora,lugar : String) : Boolean
ProfesorFullTime
oficina : String
proyectos : Proyecto [*]
comités : String [*]
asignarAlumno(Alumno) : Boolean
ProfesorPartTime
empresa : String
renovarContrato(Fecha) : Boolean

Incluimos las similitudes en una (*super*)clase general …

Propiedades de la (super)clase *Profesor*:

- los atributos nombre y departamento
- la operaciones *evaluacionDocente* y *citarReunión*

… y definimos *subclases* (o clases *herederas*)

Las subclases tienen sus propias propiedades,

… pero también tienen las propiedades de la superclase:

- toda instancia de *ProfesorFulltime* es también, por definición, una instancia de *Profesor*
- un *ProfesorFulltime* es un tipo especial de *Profesor*
- todo lo que es válido para *Profesor* —asociaciones, atributos, métodos— también lo es para *ProfesorFulltime*

Al usar generalización o herencia, observemos el *principio de sustituibilidad*

Debería poder sustituir un *ProfesorFulltime*, o un *ProfesorParttime*, en cualquier fragmento de código que requiera un *Profesor*, y todo debería funcionar bien

Si escribo código suponiendo que tengo un *Profesor*, puedo usar libremente cualquier subclase de *Profesor*

El *ProfesorFulltime* puede responder a ciertos métodos diferentemente de otro *Profesor*, por polimorfismo, pero el que hace la llamada no debería tener que preocuparse por la diferencia

Un *diagrama de secuencia* (DS) describe cómo colaboran un grupo de objetos …

… para implementar la funcionalidad de un sistema

Es un diagrama de objetos, no de clases

P.ej., tenemos una orden de compra y llamamos a una de sus operaciones —*calcPrecio( )*— para calcular su precio (próxima diap.):

- primero, la orden necesita determinar el precio de cada una de sus líneas —*calcPrecioBase( )*— que depende de la cantidad de productos —*obtCantidad( )*— y del precio unitario del producto de cada línea —*obtProducto( )* y *obtPrecio( )*
- luego, la orden debe calcular el descuento, que depende del cliente —*obtInfoDescuento( )*

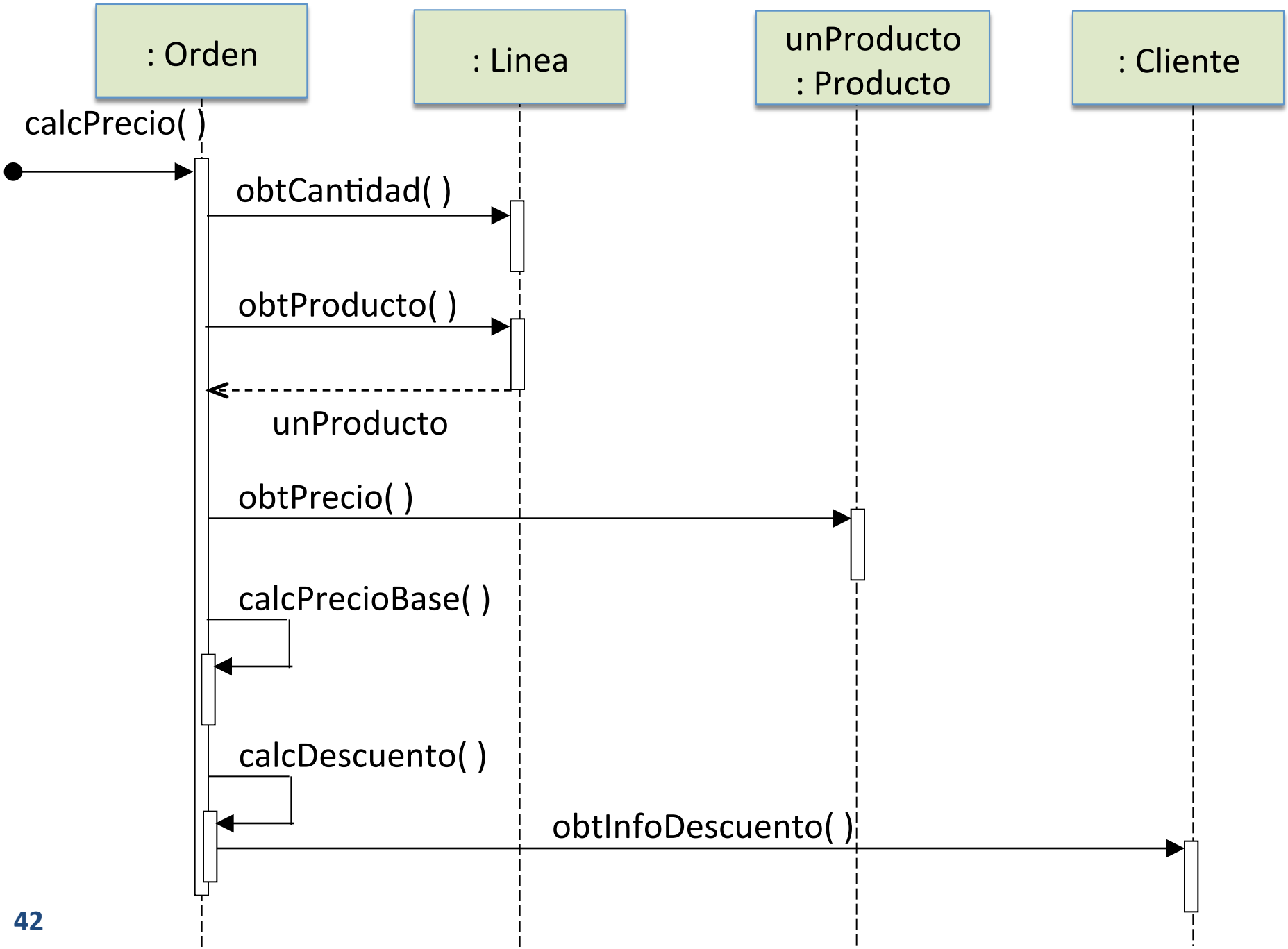
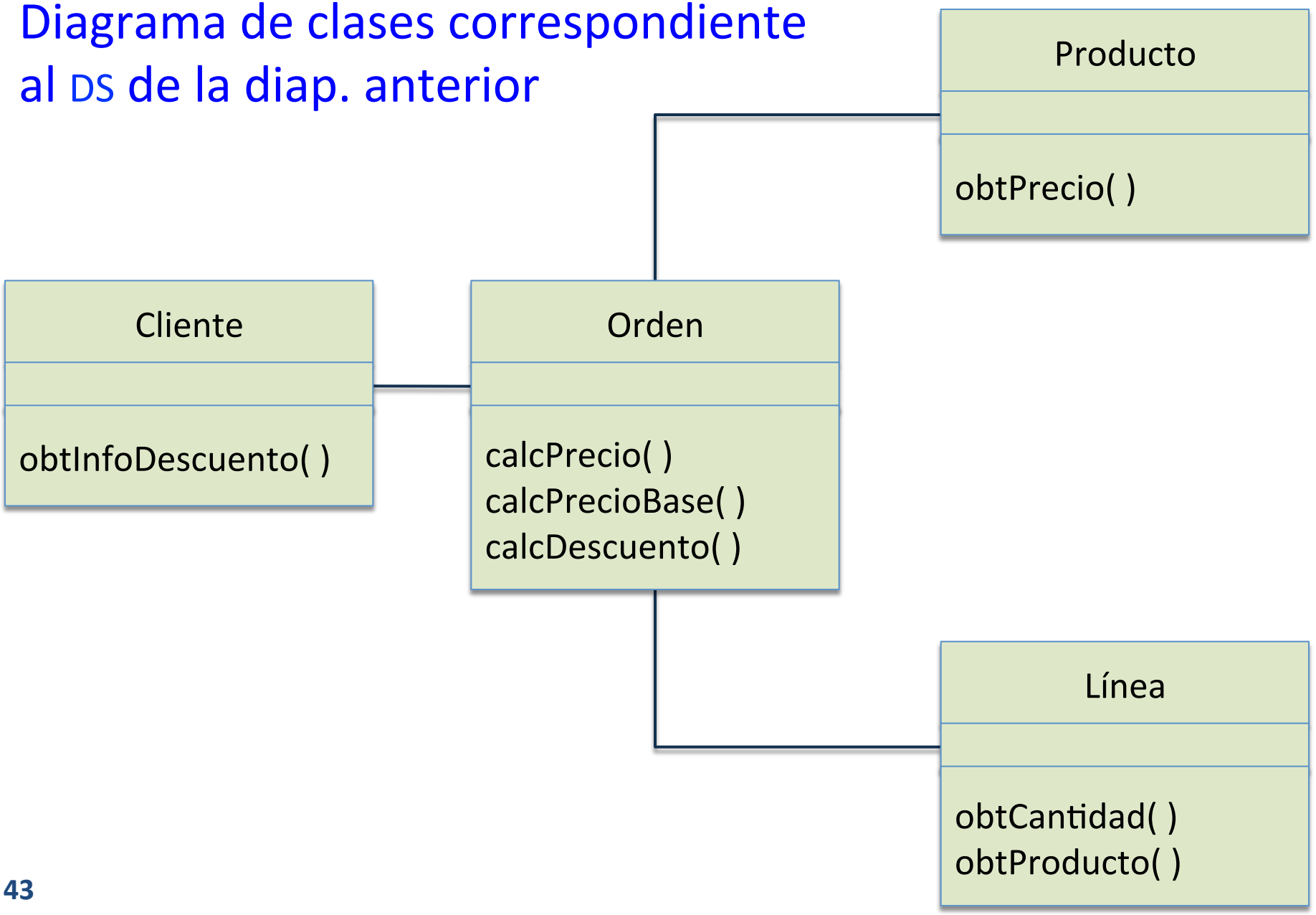

: Orden
: Linea
unProducto
: Producto
: Cliente
calcPrecio( )
obtCantidad( )
obtProducto( )
unProducto
obtPrecio( )
calcPrecioBase( )
calcDescuento( )
obtInfoDescuento( )

Diagrama de clases correspondiente al DS de la diap. anterior



Un DS (básico) casi no necesita explicación

El diagrama anterior muestra lo siguiente:

- cada *objeto* participante, con una *línea de vida* vertical —el tiempo avanza bajando por la línea
- las *operaciones* llamadas —mensajes enviados de unos objetos a otros
- el orden de los mensajes —leyendo el diagrama de arriba a abajo

Los mensajes enviados a un objeto deben corresponder a las operaciones definidas en la clase del objeto

En el DS anterior es fácil ver lo siguiente

Una instancia de *Orden* envía los mensajes *obtCantidad* y *obtProducto* a una instancia de *Linea*

La *Orden* llama a un método de sí misma, *calcDescuento*, y éste envía el mensaje *obtInfoDescuento* a una instancia de *Cliente*

Los DS's muestran el nombre y la actividad de los objetos participantes

Cada línea de vida tiene una *barra de activación* (o varias), opcional:

- muestra cuándo el objeto participante está activo
- corresponde a la ejecución de una operación del objeto

Dar nombres a los objetos permite correlacionarlos

- p.ej., el mensaje *obtProducto( )* retorna *unProducto* —el mismo nombre, y por lo tanto el mismo participante, que el objeto *unProducto* al que se le envía el mensaje *obtPrecio( )*

Los DS's pueden mostrar valores de retorno y mensajes sin una fuente

Las respuestas a los mensajes enviados son opcionales:

- incluirlos cuando agreguen información
- p.ej., el diagrama muestra sólo la respuesta al mensaje *obtProducto( )*, para explicitar la correspondencia entre el producto retornado, *unProducto*, y el producto al que se le envía a continuación el mensaje *obtPrecio*

El primer mensaje, *calcPrecio( )*, no tiene un participante que lo haya enviado:

- proviene de una fuente indeterminada
- se conoce como un *mensaje encontrado* (*found message*)

Los objetos podrían colaborar más descentralizadamente

En la próxima diap., la orden le pide a cada línea que calcule su propio precio

La línea, a su vez, le traspasa el problema del cálculo al producto mismo:

- le pasa como parámetro la cantidad de unidades

Para calcular el descuento, la orden le envía un mensaje al cliente:

- como el cliente necesita información de la orden, envía un mensaje de vuelta, *obtValorBase( )*, a la orden

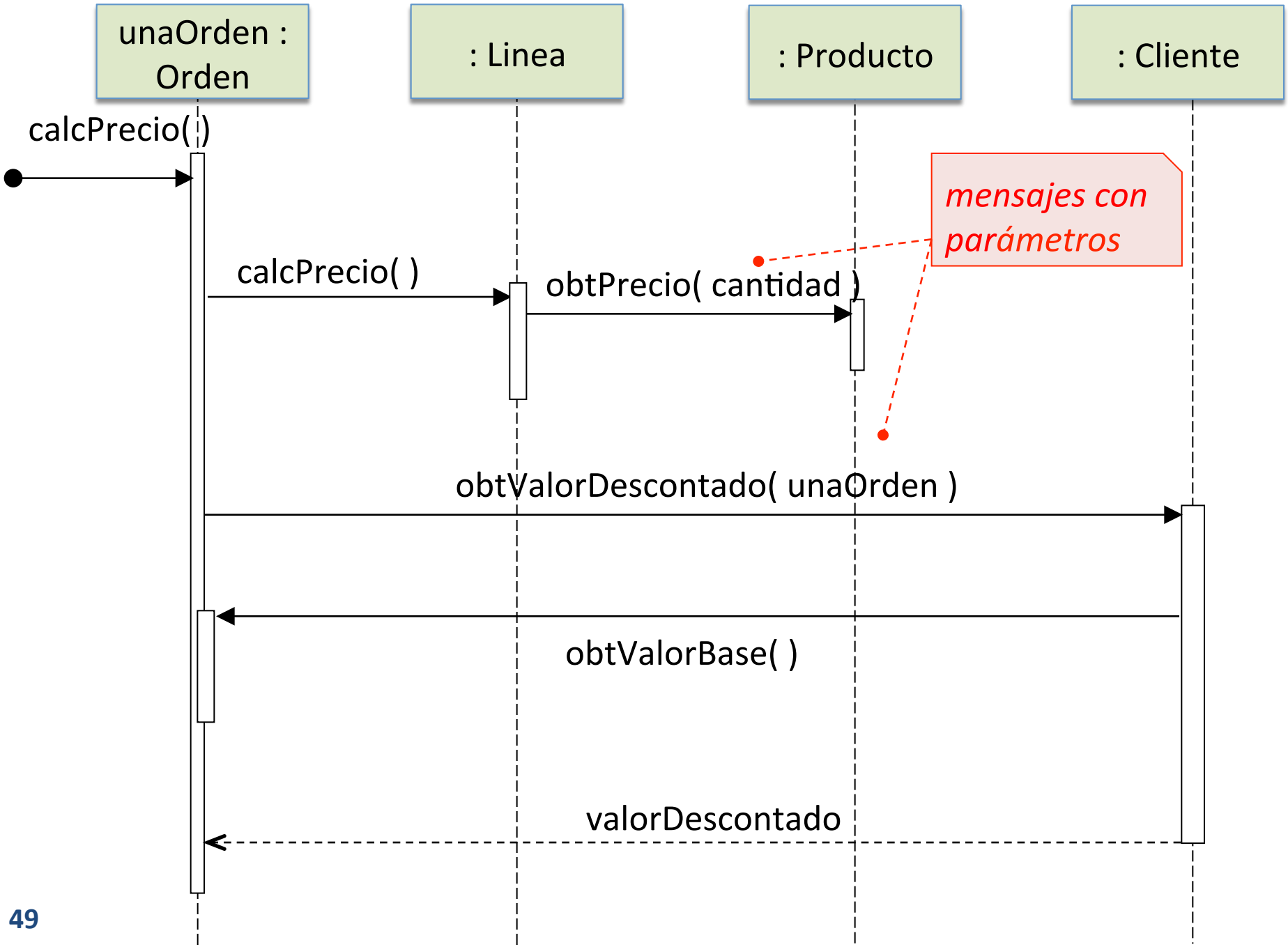
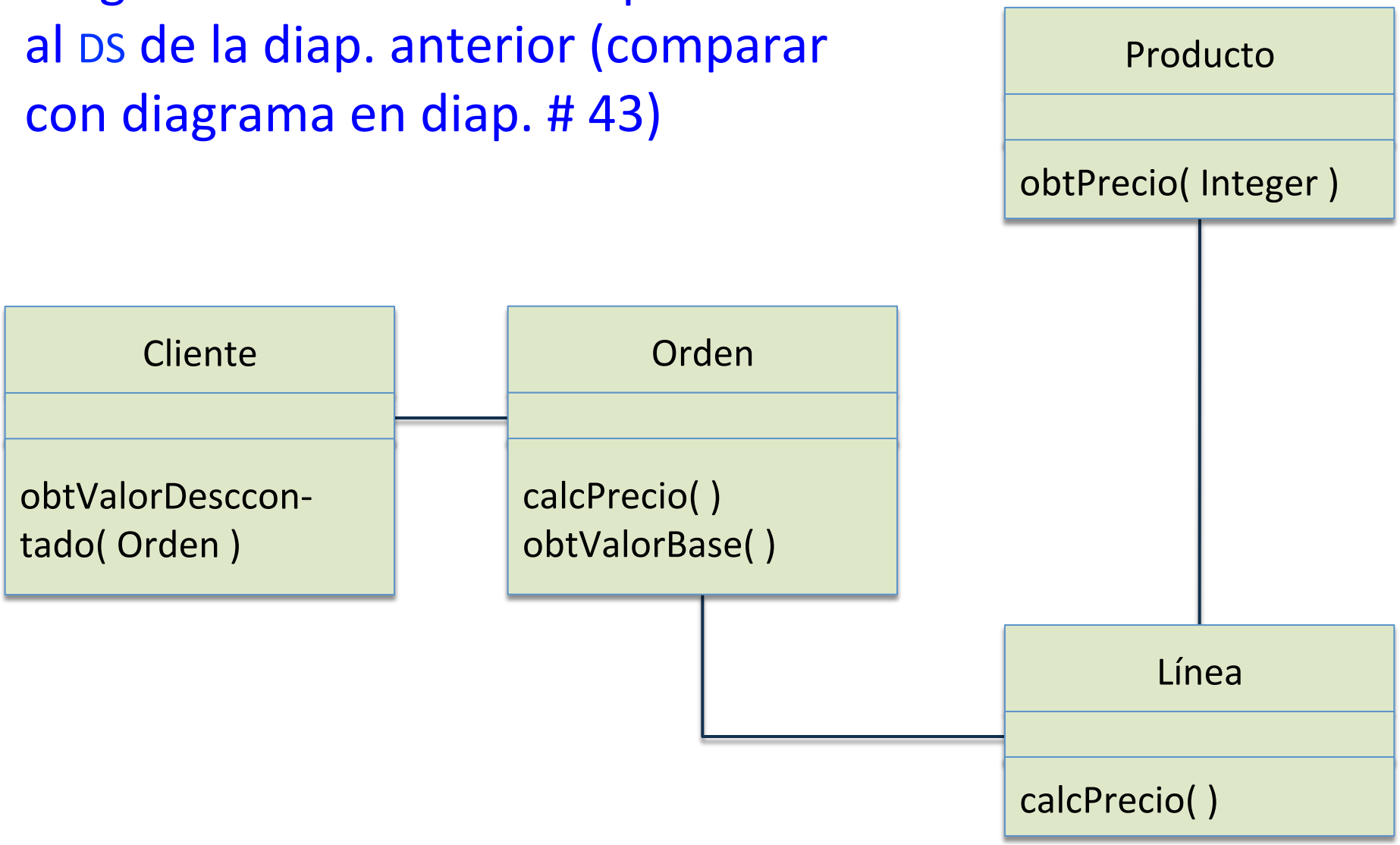

unaOrden : Orden
: Linea
: Producto
: Cliente
calcPrecio( )
calcPrecio( )
obtPrecio(cantidad)
mensajes con parámetros
obtValorDescontado(unaOrden)
obtValorBase( )
valorDescontado

Diagrama de clases correspondiente al DS de la diap. anterior (comparar con diagrama en diap. # 43)



Los DS's muestran claramente cómo interactúan los participantes

Los diagramas de las diaps. # 42 y # 49 muestran dos formas diferentes de interacción entre los mismos cuatro objetos participantes

Ésta es su fortaleza más importante:

- los mensajes entre objetos se ven claramente
- dan una buena idea de cuáles objetos hacen qué

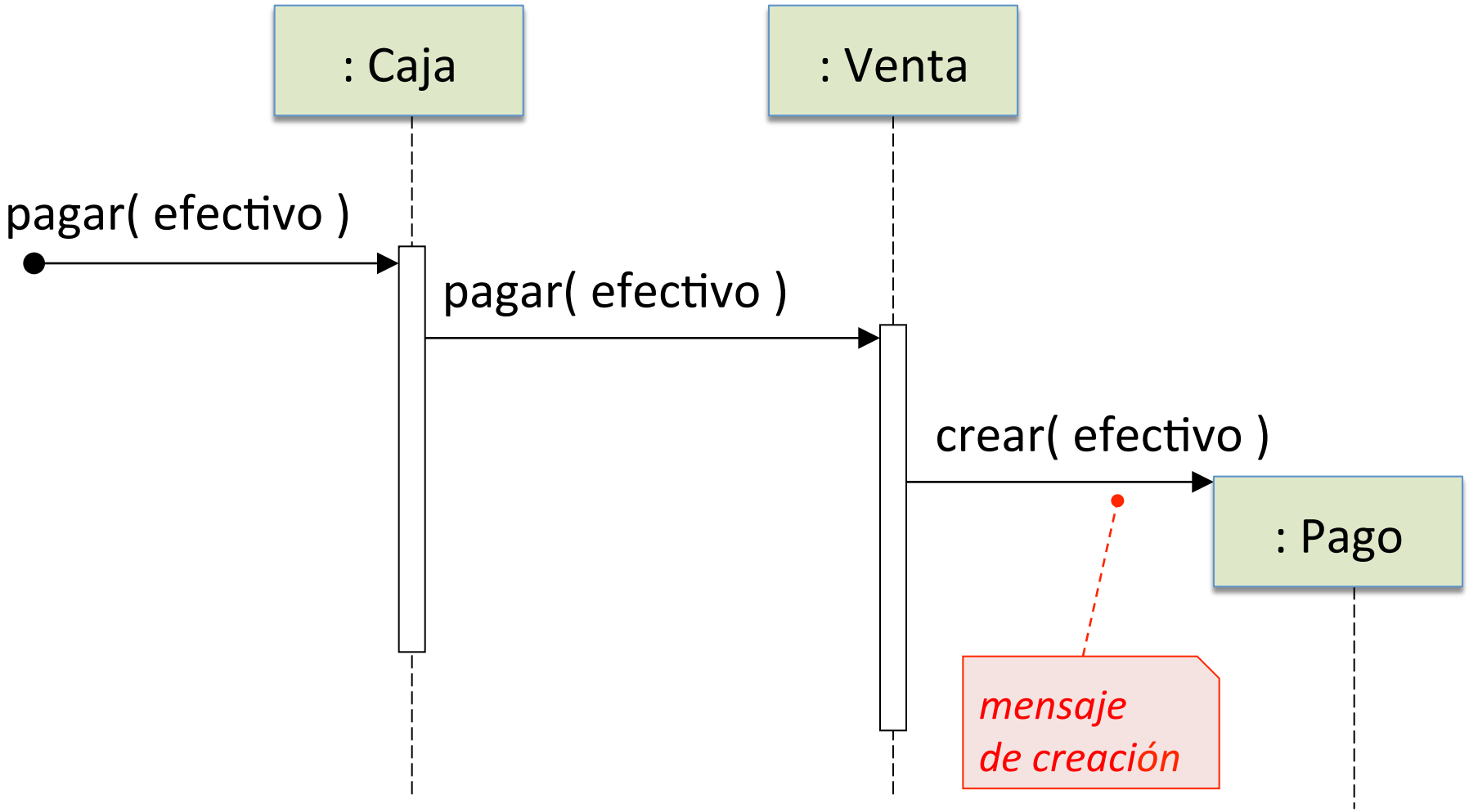
Los DS's no son tan buenos para mostrar detalles de algoritmos, ya que hay que agregar figuras adicionales:

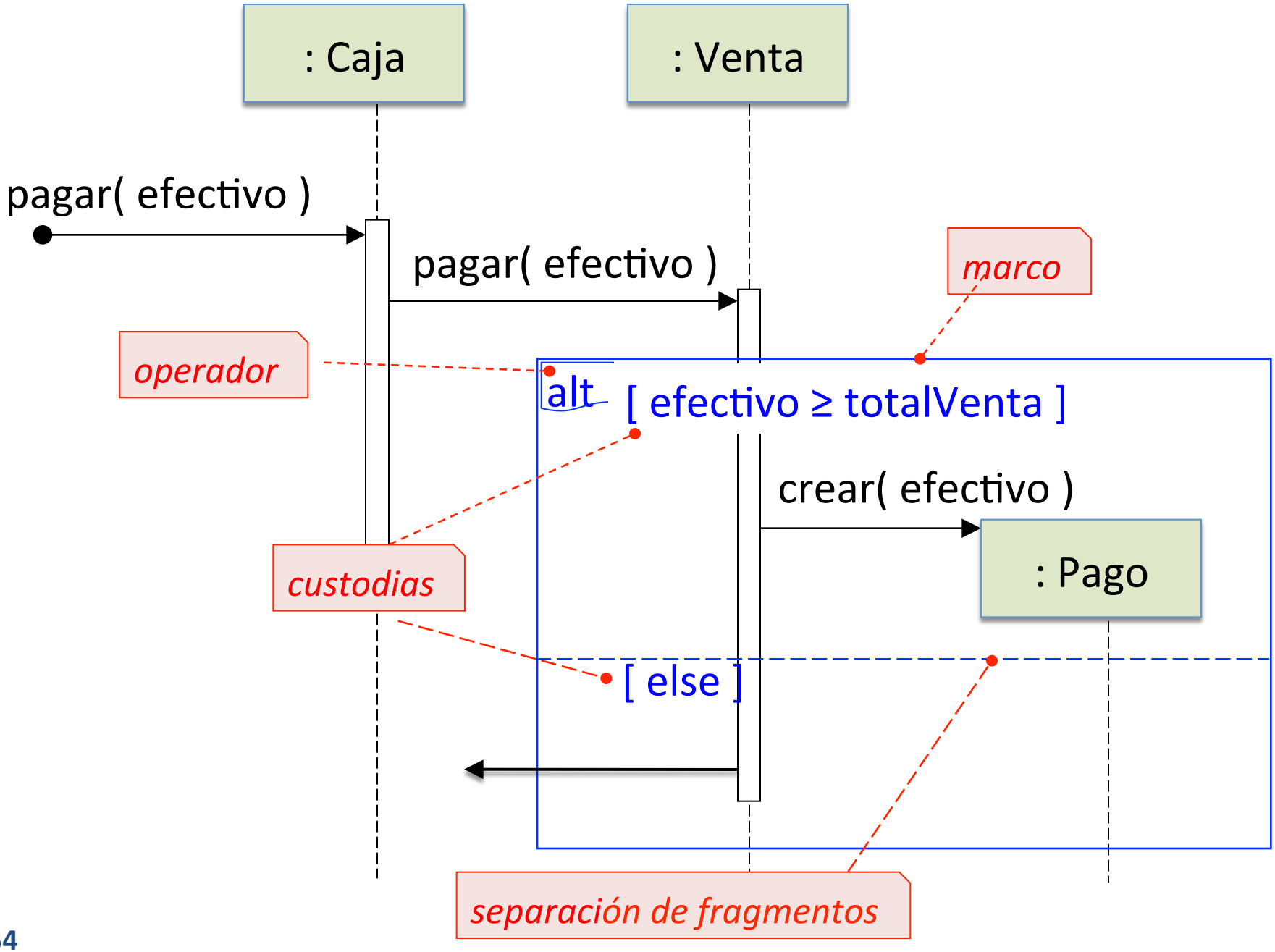
- p.ej., comportamiento condicional o repetitivo

Un DS puede mostrar *creación de objetos* y *comportamiento condicional*

Las próximas diaps. muestran el (mini) flujo de *Pagar en efectivo* para un sistema de PdV, en que participan instancias de *Caja*, *Venta* y *Pago*:

- la instancia de *Pago* es creada durante la ejecución del escenario —creación de participantes
- la creación de un pago se puede hacer depender de si el efectivo (entregado por el cliente) es mayor o igual al total de la venta —comportamiento condicional


: Caja
: Venta
pagar(efectivo)
pagar(efectivo)
crear(efectivo)
: Pago
mensaje
de creación


: Caja
: Venta
pagar(efectivo)
pagar(efectivo)
marco
operador
alt
[efectivo ≥ totalVenta]
crear(efectivo)
: Pago
custodias
[else]
separación de fragmentos

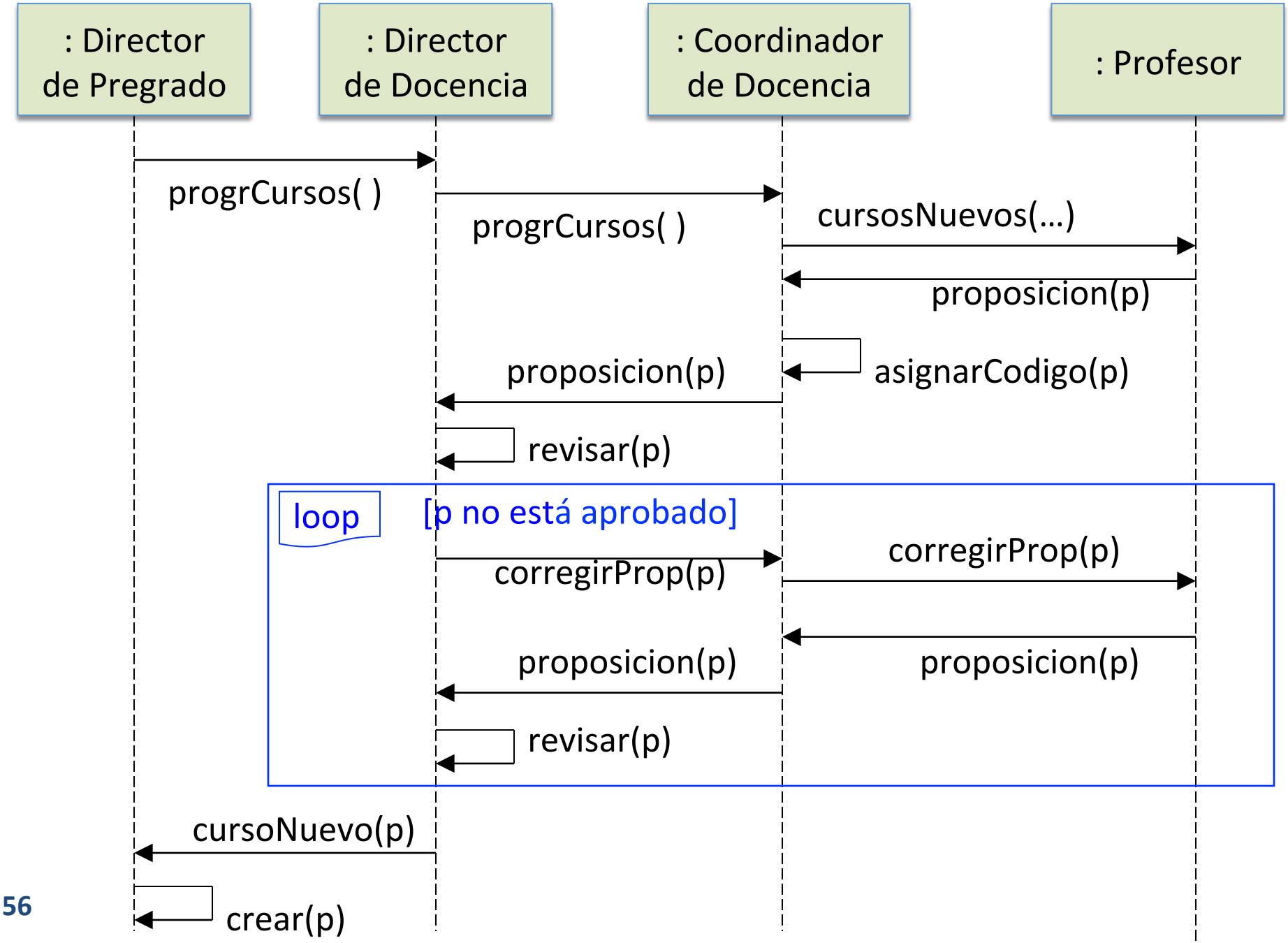
Los *marcos de interacción* son formas de destacar una región de un DS

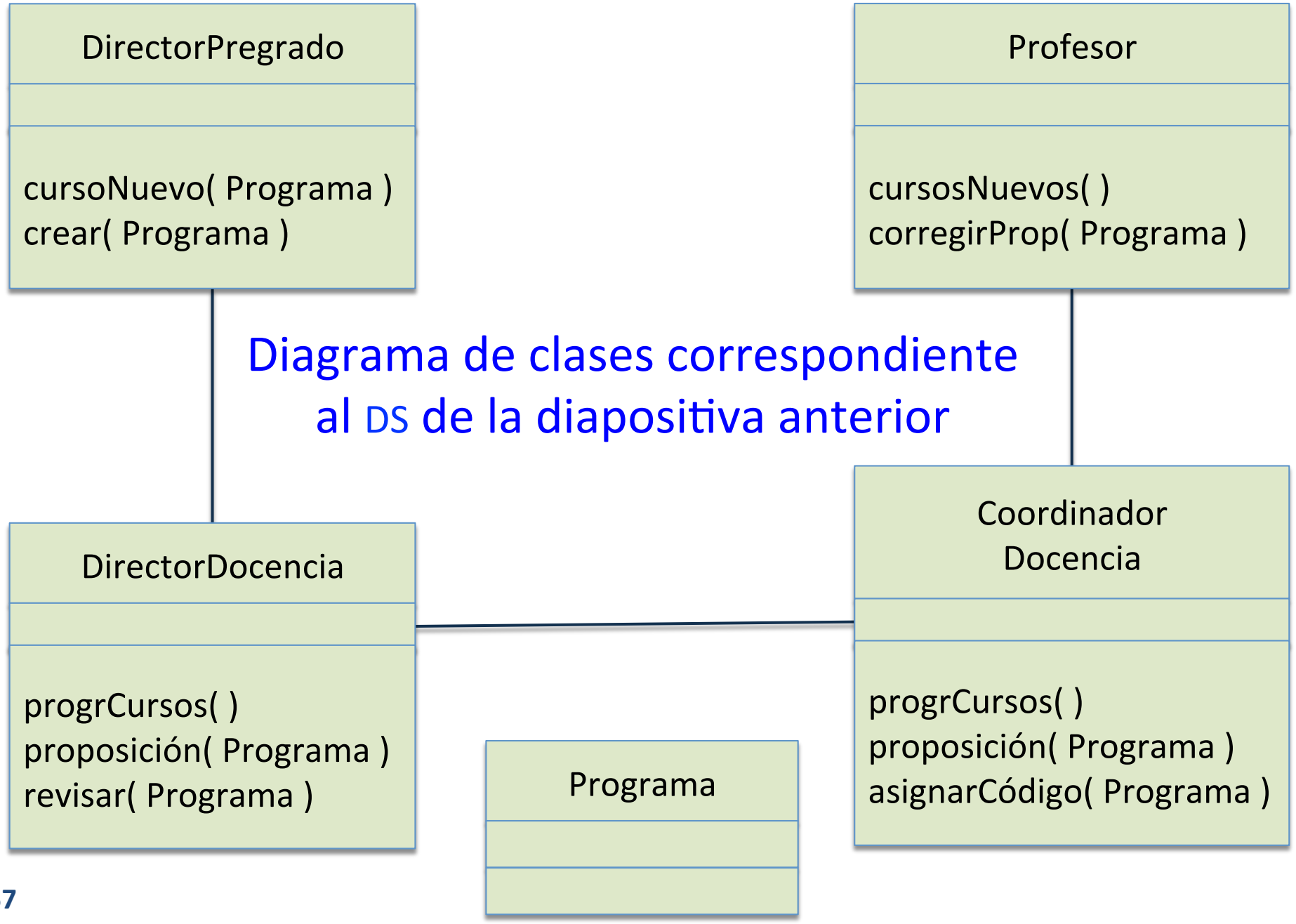
La región es dividida en fragmentos:

- cada fragmento tiene un operador y puede tener una **custodia** (una condición)

P.ej.,

- para mostrar repetición, usamos el operador **loop** con un fragmento y la custodia es la condición de la repetición
- para mostrar comportamiento condicional, usamos el operador **alt** con dos o más fragmentos, y ponemos condiciones mutuamente excluyentes en cada custodia —sólo una puede ser verdadera y sólo el fragmento correspondiente se ejecuta


: Director de Pregrado
: Director de Docencia
: Coordinador de Docencia
: Profesor
progrCursos( )
progrCursos( )
cursosNuevos(...)
proposicion(p)
proposicion(p)
asignarCodigo(p)
revisar(p)
loop
[p no está aprobado]
corregirProp(p)
corregirProp(p)
proposicion(p)
proposicion(p)
revisar(p)
cursoNuevo(p)
crear(p)


DirectorPregrado
cursoNuevo(Programa)
crear(Programa)
Profesor
cursosNuevos( )
corregirProp(Programa)
Diagrama de clases correspondiente
al DS de la diapositiva anterior
DirectorDocencia
progrCursos( )
proposición(Programa)
revisar(Programa)
Coordinador
Docencia
progrCursos( )
proposición(Programa)
asignarCódigo(Programa)
Programa

alt : Múltiples fragmentos alternativos; sólo aquél cuya custodia es verdadera se ejecutará

opt : Opcional; el fragmento se ejecuta sólo si la custodia es verdadera —equivale a alt con una alternativa

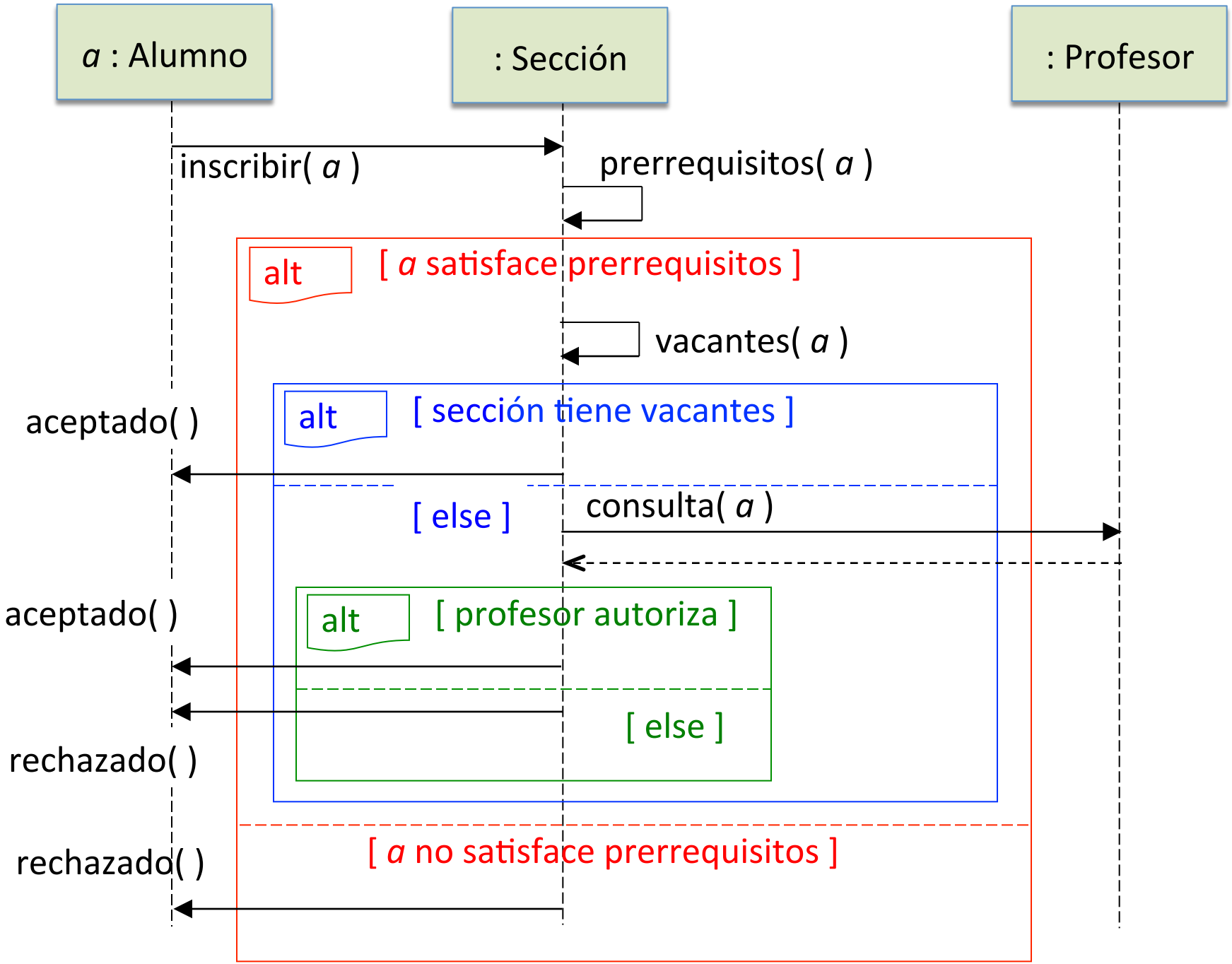
loop : Ciclo; el fragmento puede ejecutarse varias veces y la custodia indica la condición de la repetición

par : Paralelismo; cada fragmento corre en paralelo

ref : Referencia a un fragmento que aparece en otro diagrama

strict : Los mensajes en el fragmento están totalmente ordenados —sólo hay una traza de ejecución consistente con el fragmento

criticalRegion : El fragmento debe ser tratado como atómico y no puede ser intercalado con otras ocurrencias de eventos


a : Alumno
: Sección
: Profesor
inscribir(a)
prerrequisitos(a)
alt
[a satisface prerrequisitos]
vacantes(a)
alt
[sección tiene vacantes]
aceptado( )
[else]
consulta(a)
alt
[profesor autoriza]
aceptado( )
[else]
rechazado( )
[a no satisface prerrequisitos]
rechazado( )

Es posible anidar unos marcos de interacción dentro de otros

La diap. anterior muestra el DS del proceso de inscripción de un alumno en una sección de un curso:

- el alumno, *a*, solicita la inscripción a la sección, y ésta verifica el cumplimiento de los prerrequisitos
- si el alumno tiene los prerrequisitos, entonces la sección verifica la disponibilidad de vacantes
- si hay vacantes, el alumno es admitido; de lo contrario, se consulta al profesor, que puede autorizar o no
- si el alumno no tiene los prerrequisitos, es rechazado