

Introducción al UML

2-2018

Yadran Eterovic (yadran@ing.puc.cl)

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

2

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 ni de desarrollo de sistemas

3

UML sólo proporciona una sintaxis visual:

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

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

4

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

- diagramas informales o incompletos —típicamente de clases y a veces también de secuencia— que sirven para explorar las partes difíciles del problema o del espacio de soluciones

2) Para dibujar *planos de análisis* y *planos de diseño*:

- diagramas formales, 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”

5

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 modelo en UML tiene dos aspectos complementarios y casi inseparables

6

La **estructura estática** describe qué tipos (clases) de objetos son importantes para modelar el sistema

... y cómo están relacionados —ver próxima diap.

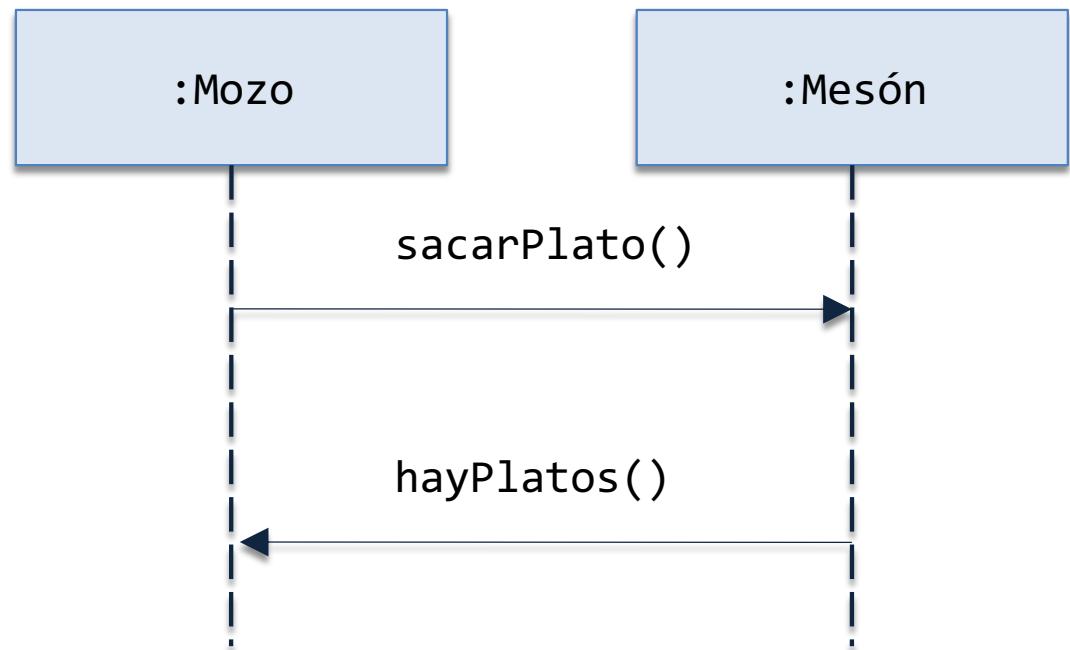
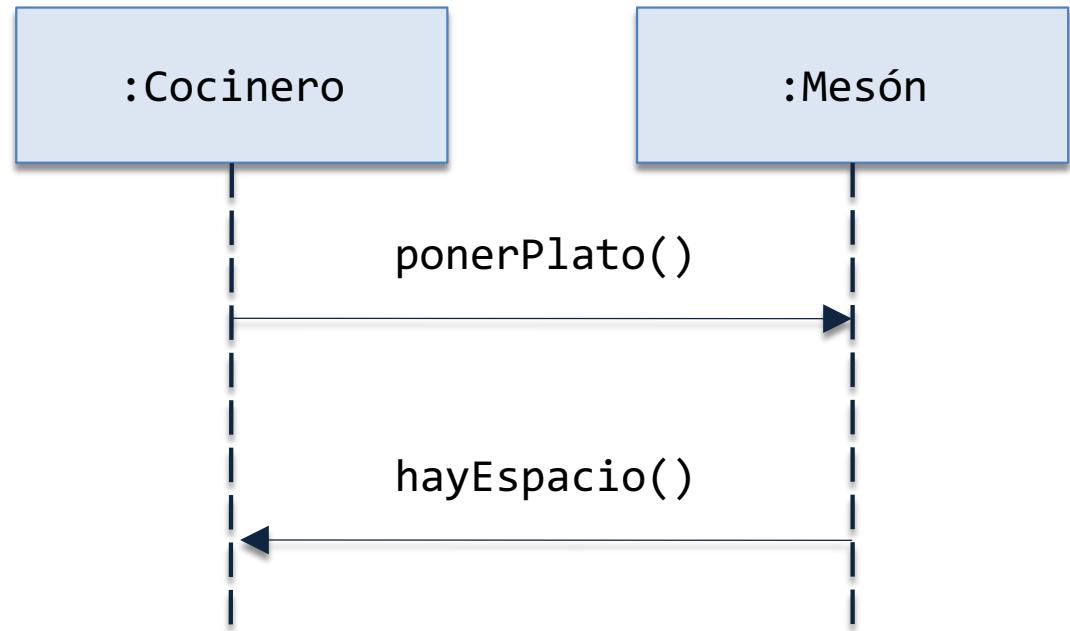
El **comportamiento dinámico** describe cómo los objetos interactúan entre ellos para entregar funcionalidad —ver diap. #8

... y los ciclos de vida de los objetos a medida que cumplen sus responsabilidades —ver diap. #9

Ejemplo de un diagrama de clases

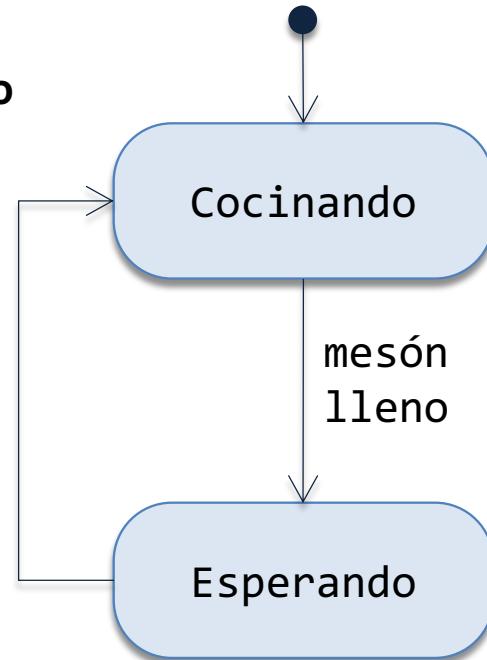


Ejemplos de diagramas de secuencia



Cocinero

mesón
tiene
espacio



Mesón

Vacío

Regular

Lleno

cocinero pone
un plato
[cantidad = 9]

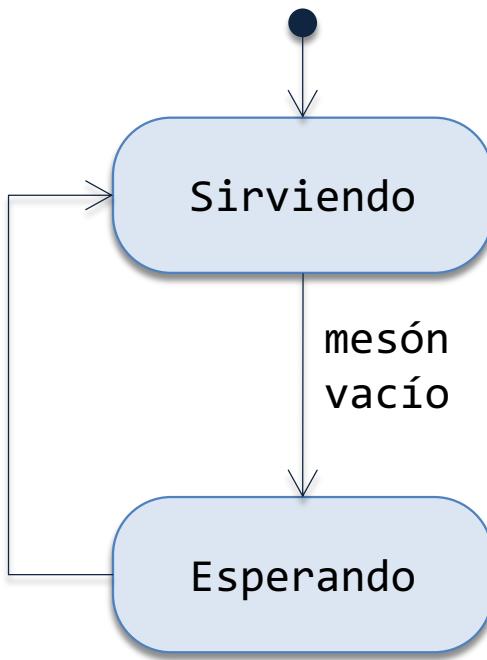
mozo saca
un plato
[cantidad = 1]

mozo saca
un plato

Ejemplos de diagramas
de estado

Mozo

mesón
tiene
plato



UML 2.0 [2005]: Colección de 13 diagramas

10

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:

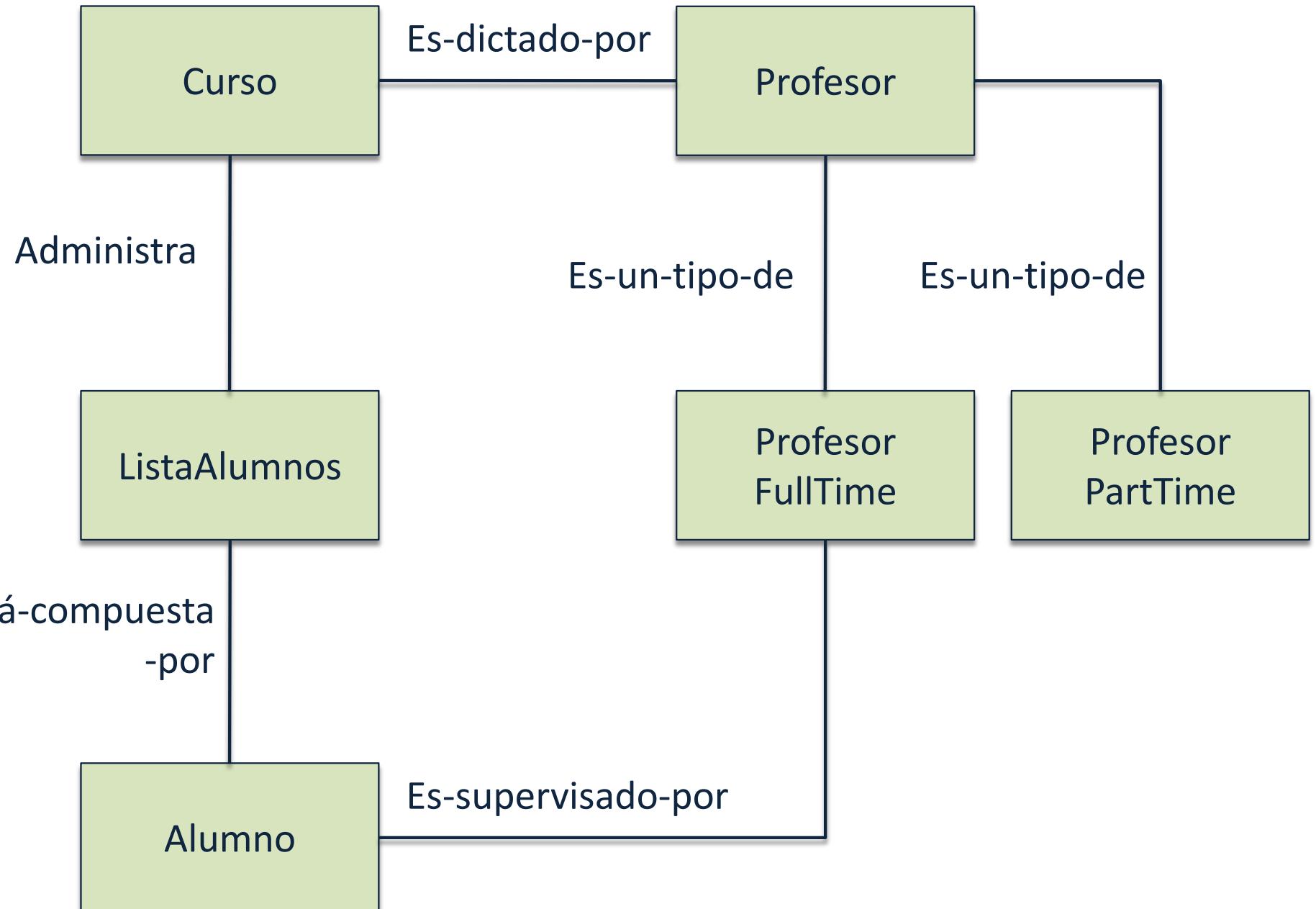
casos de uso, actividades, máquinas de estados, comunicación, secuencia, tiempo, resumen de interacción

Los diagramas de clases pueden incluir diversos niveles de detalle

11

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:

- el diagrama de clases también podría incluir los atributos de las clases y las multiplicidades en los extremos de las asociaciones

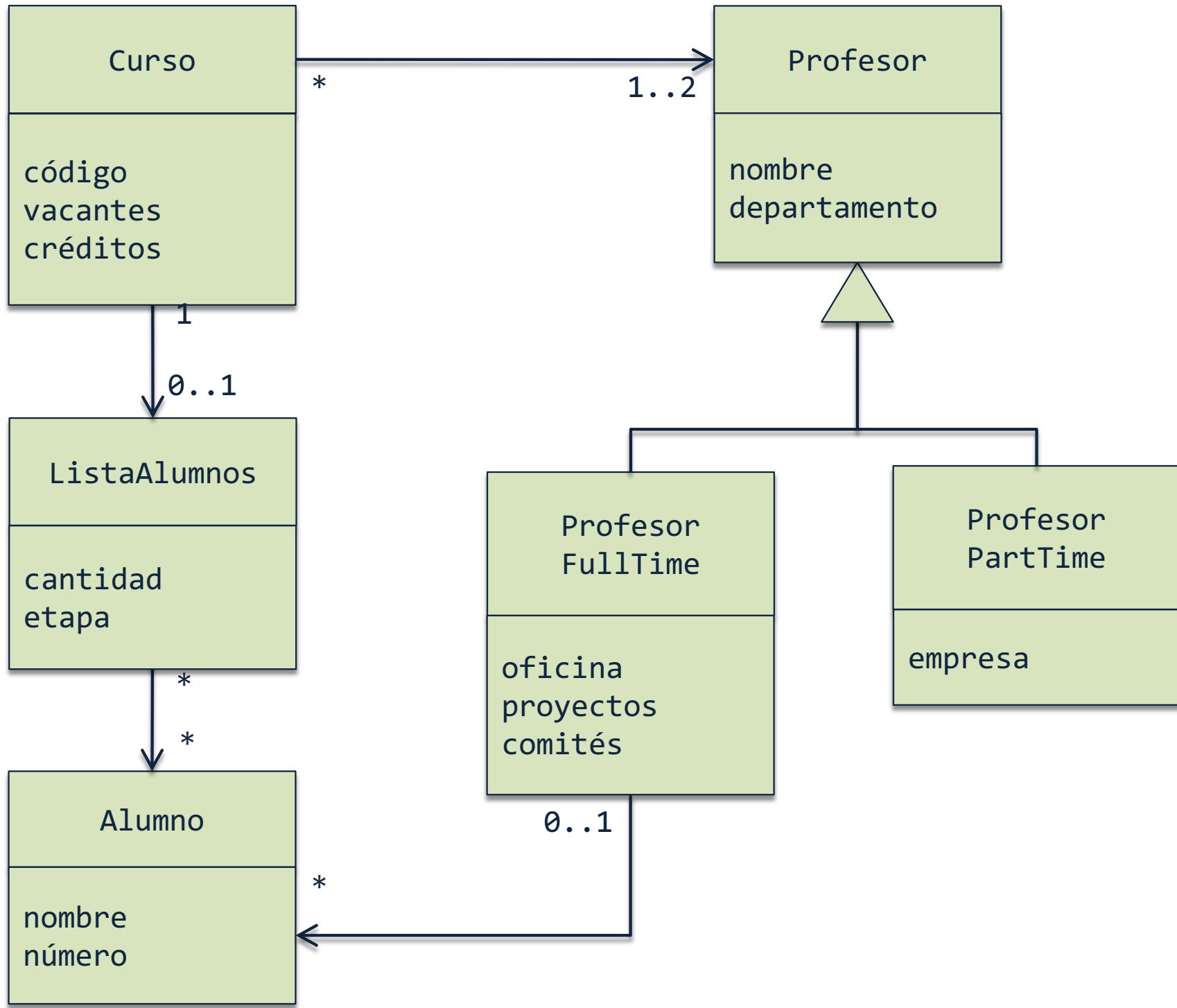


Un diseño de software se representa mediante un *diagrama de clases del diseño* (DCD)

13

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

- muestra los atributos de las clases
- muestra las multiplicidades de las asociaciones
- muestra la navegabilidad de las asociaciones
- típicamente, no muestra los nombres de las asociaciones



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

15

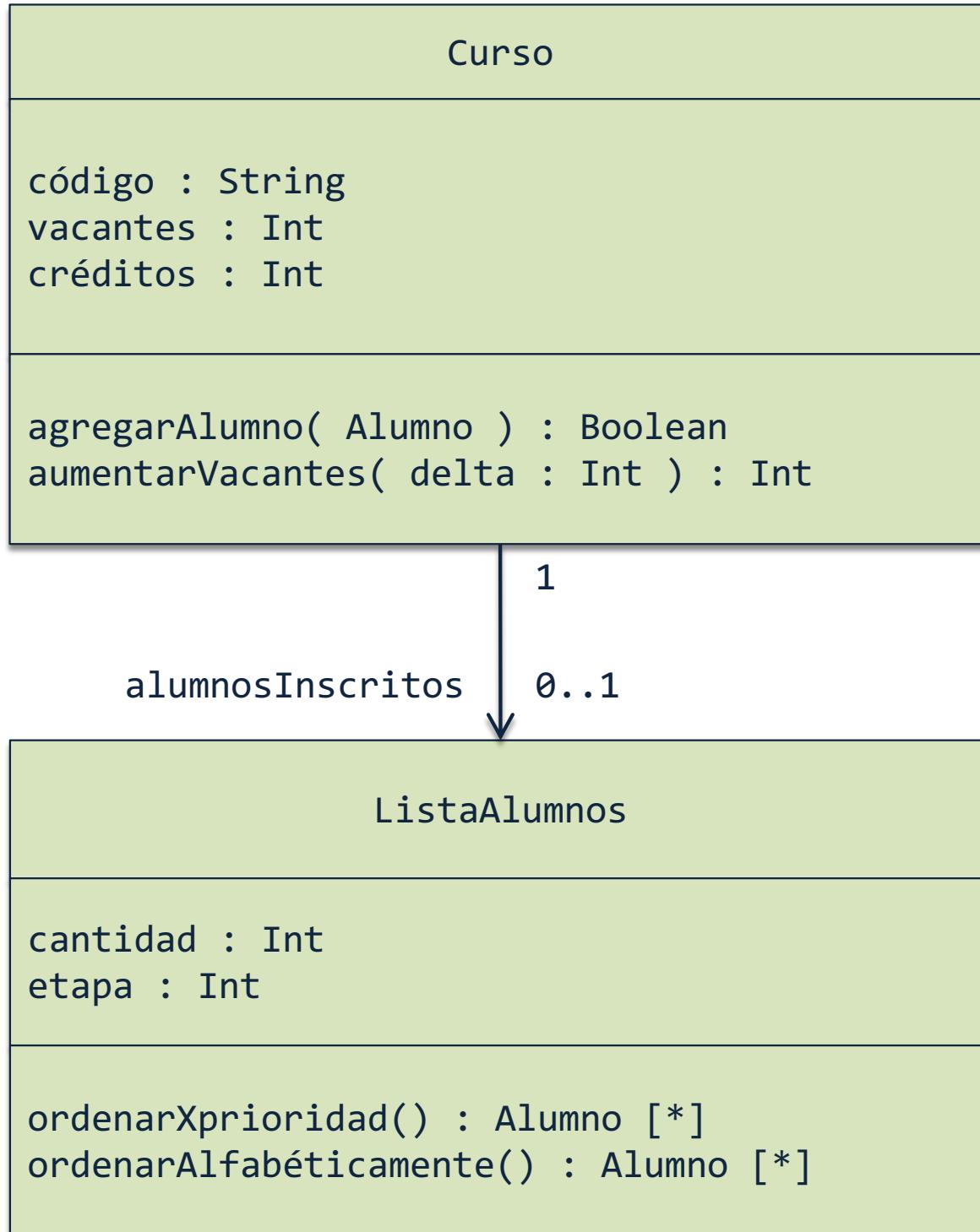
Las *operaciones* (o *métodos*) de las clases —ya sea solo los nombres, o las firmas completas

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*



Las operaciones pueden incluir información adicional al nombre

18

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, de nuevo, opcionales

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

agregarAlumno(a : Alumno) : Boolean

En un DCD hay dos tipos de relaciones entre las clases

19

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

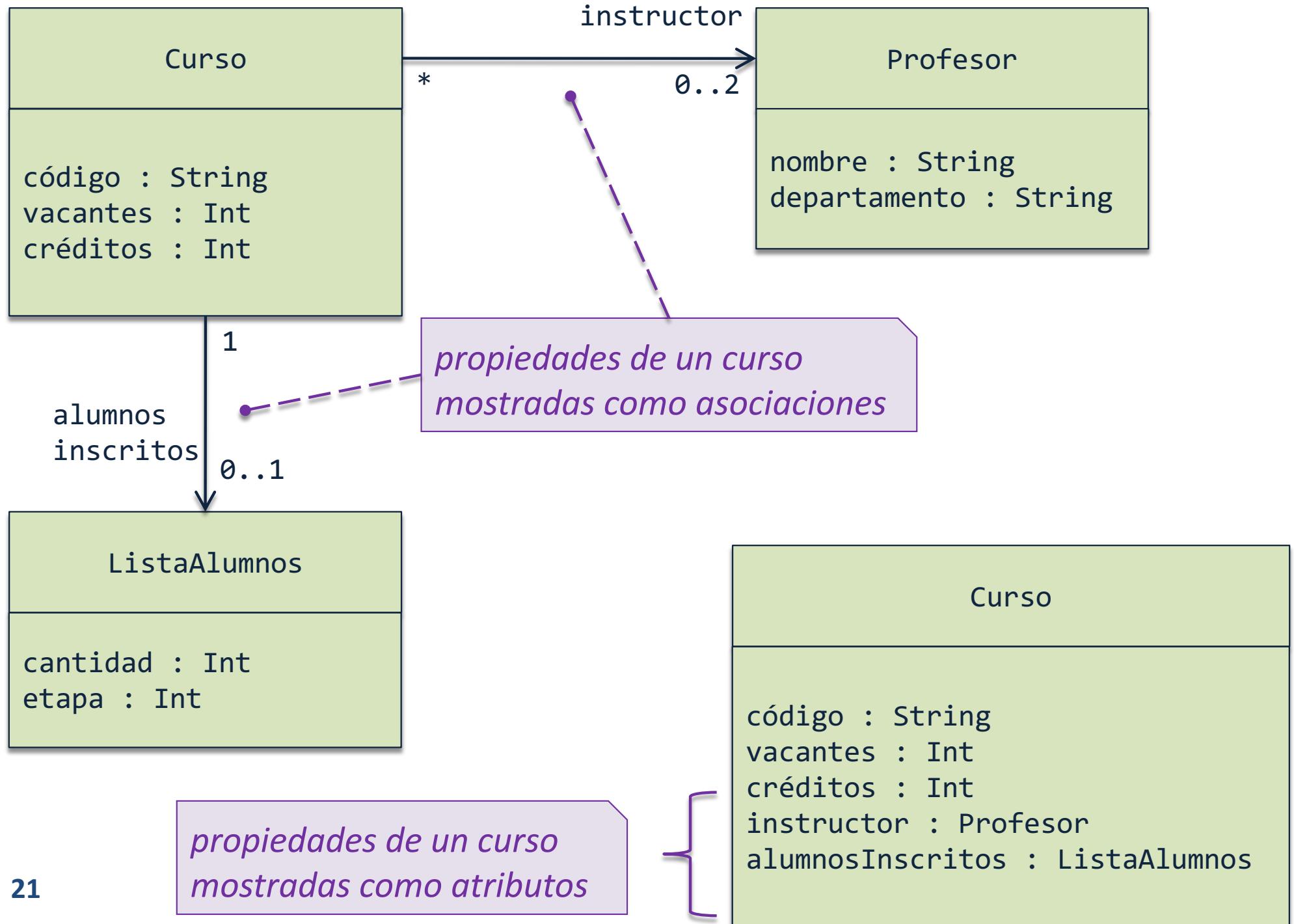
Generalizaciones (o herencia) —denotadas por un triángulo, en cuyo vértice superior descansa la clase más general

Atributos y asociaciones son dos formas de representar lo mismo

20

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: *Int*, *String*, etc;
- las **asociaciones** son las propiedades cuyos valores están almacenados mediante *referencias* —en general, son otros objetos, instancias de otras clases



Las propiedades —atributos y asociaciones— tienen nombres

22

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

23

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

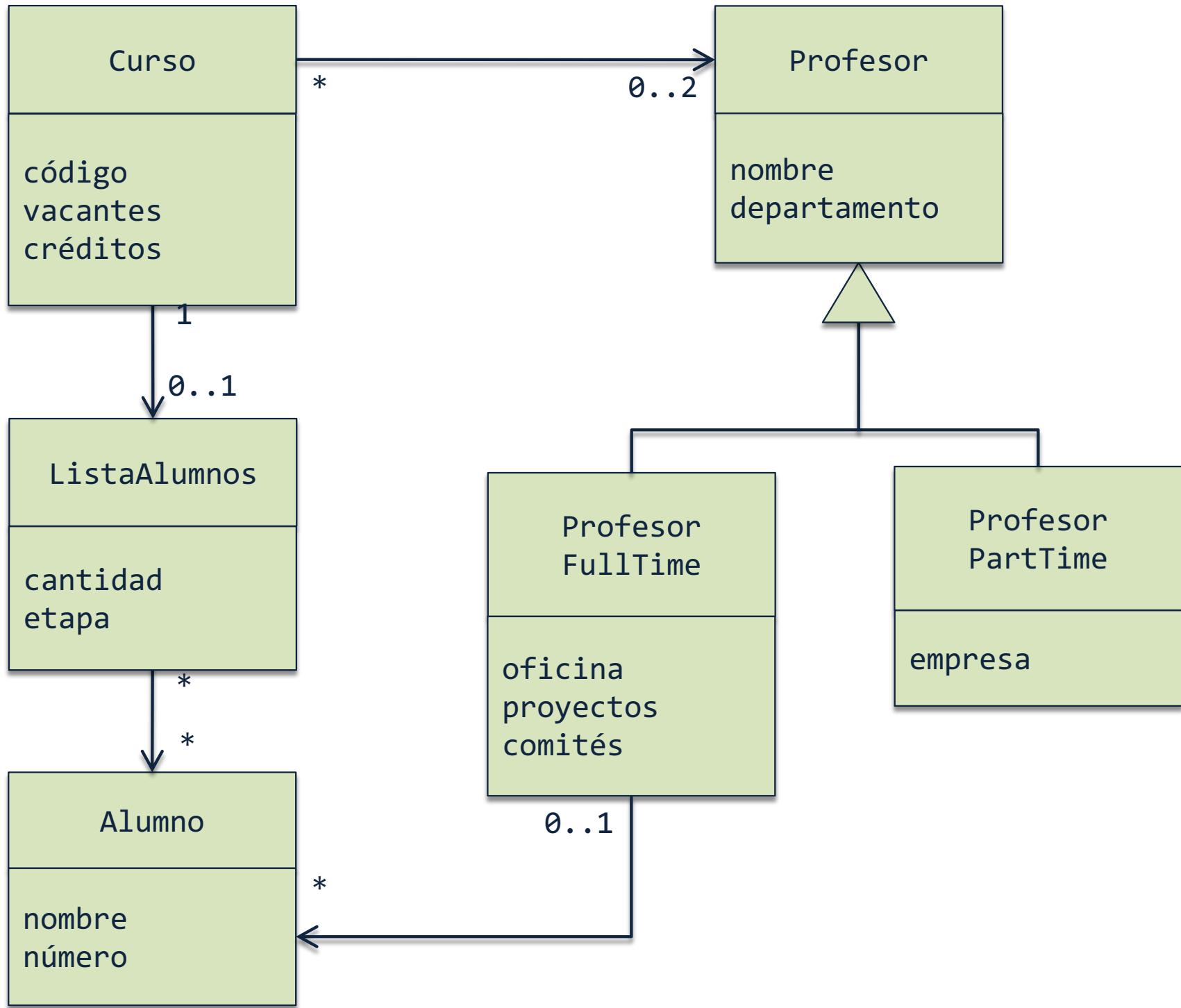
25

Sólo importa el rol que desempeña la clase en el extremo con la punta de flecha con respecto a la clase en el otro extremo,

... 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*



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

27

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. #17 y 29):

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

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

28

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

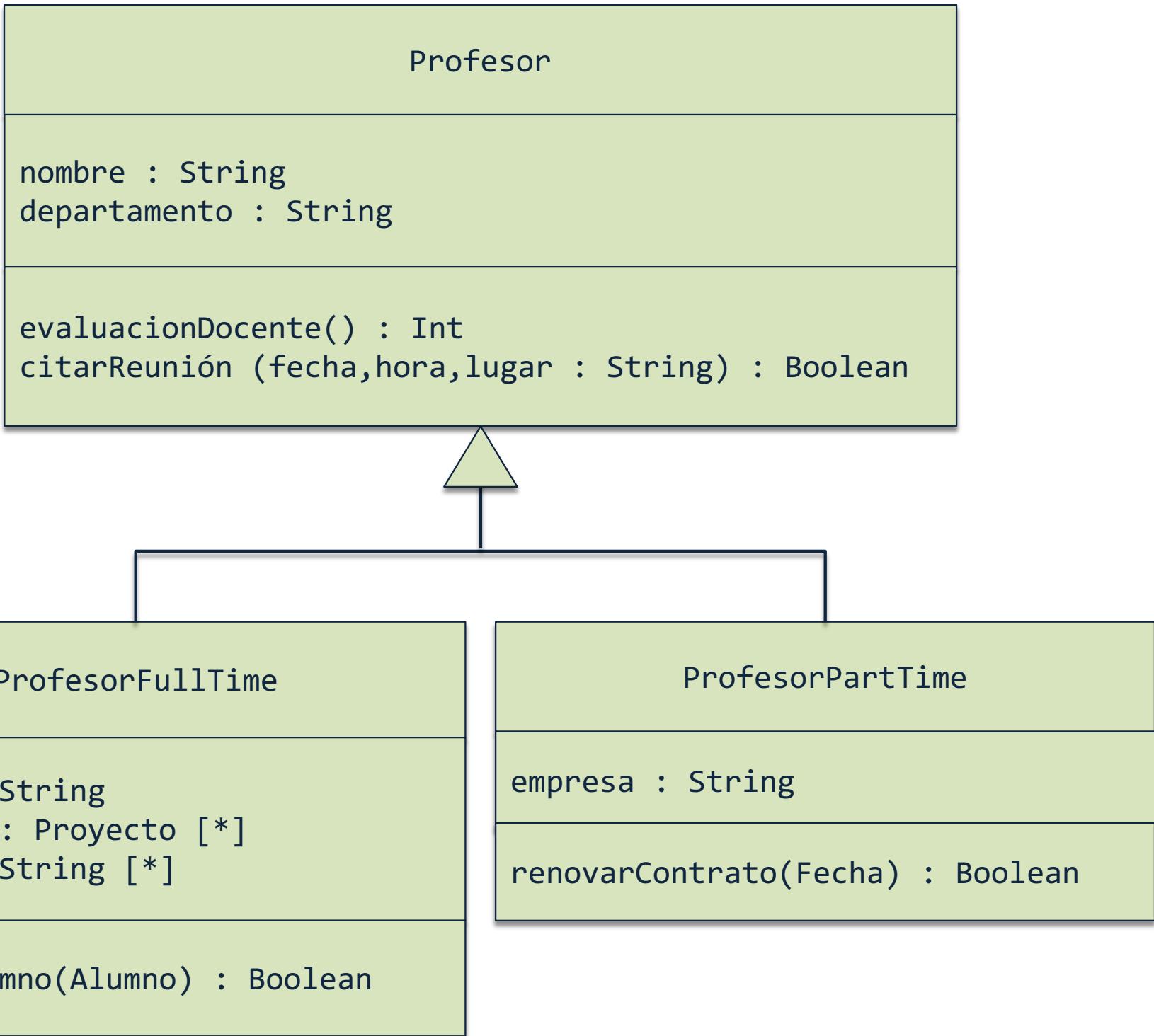
nombre, departamento, y formas para obtener su calificación docente y citarlo a reunión

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



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

30

Propiedades de la (*super*)clase *Profesor*:

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

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

31

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*

32

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 differently de otro *Profesor*, por *polimorfismo*, pero quien hace la llamada no debería tener que preocuparse por la diferencia

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

33

... 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()*

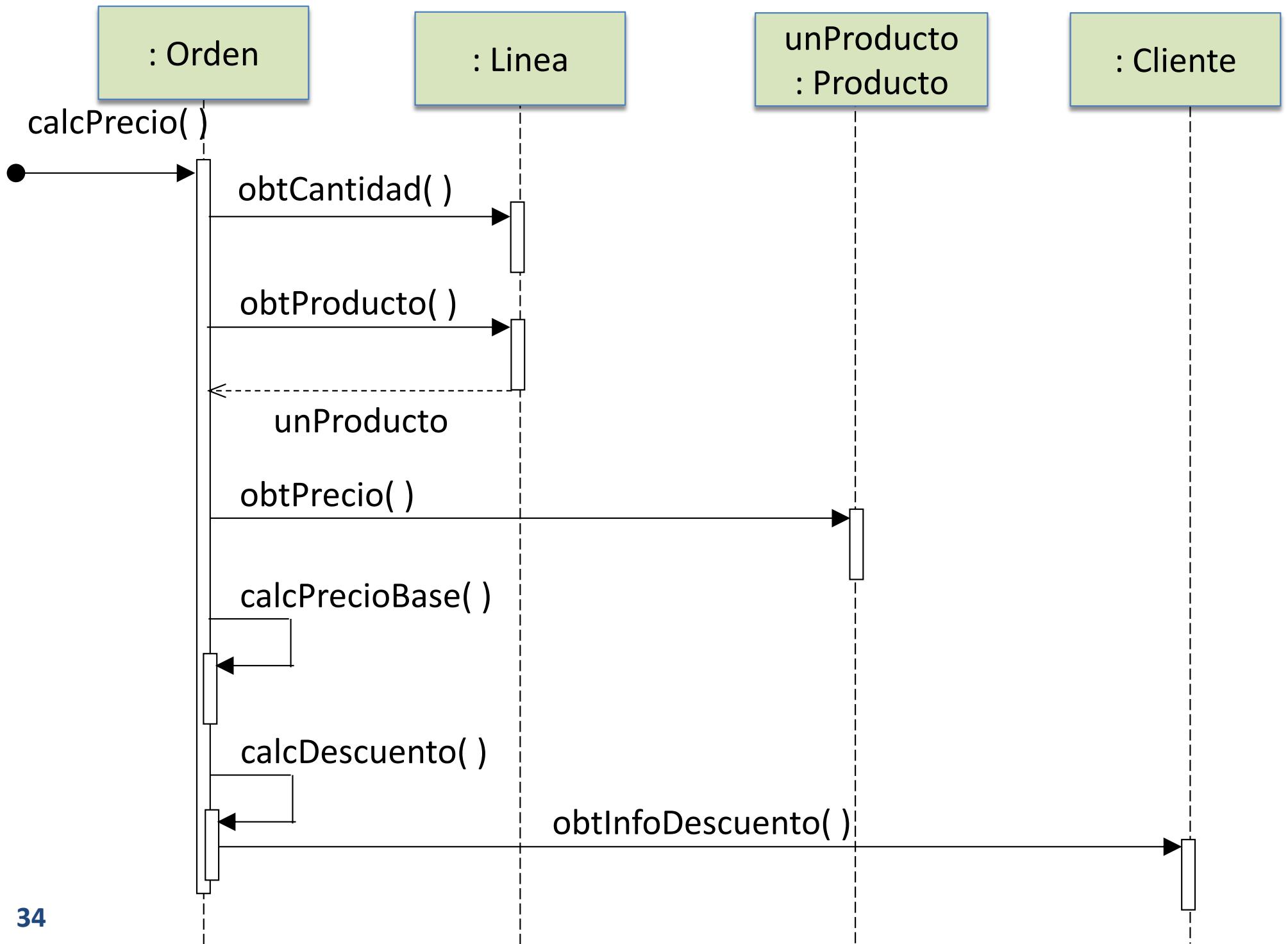
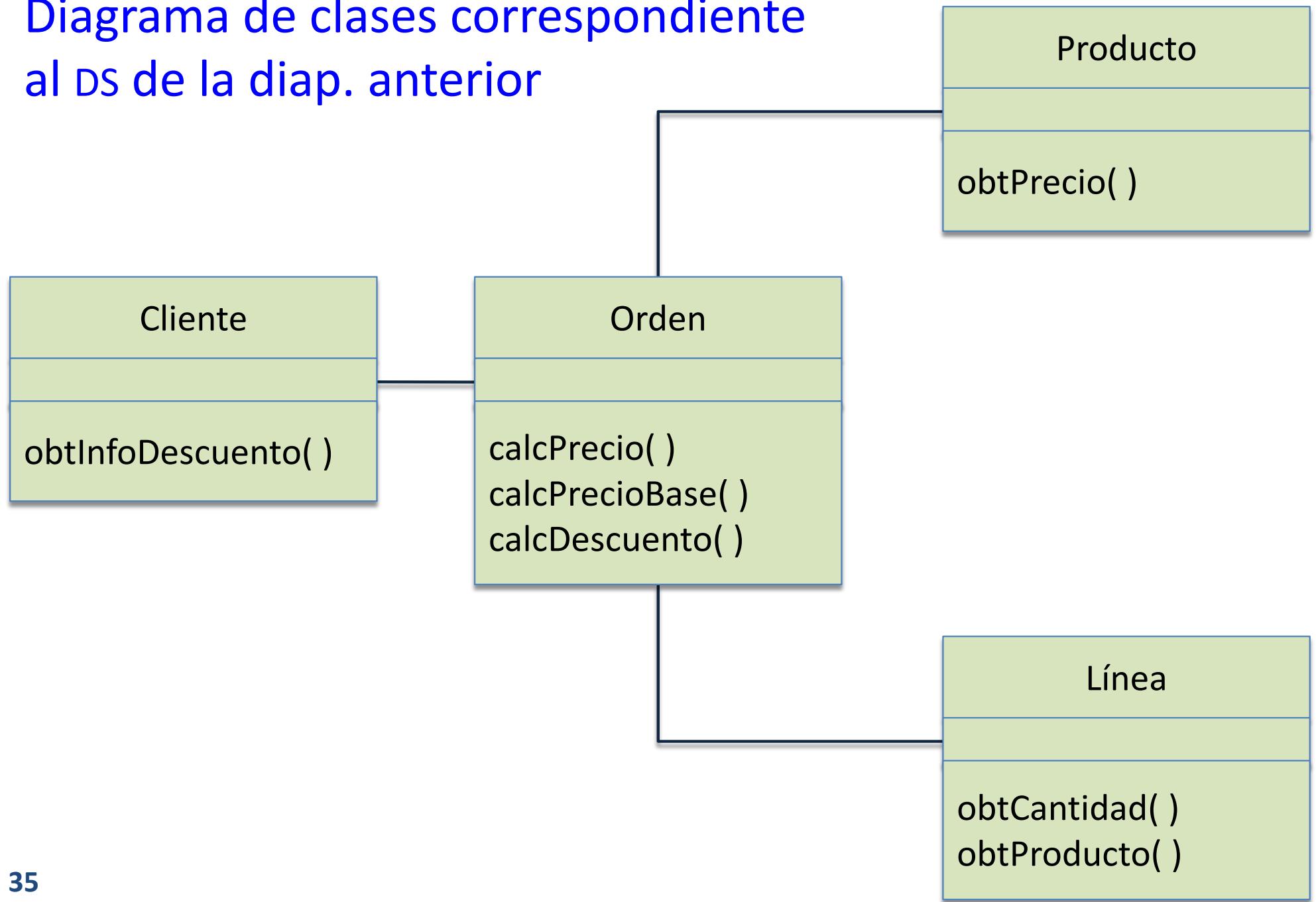


Diagrama de clases correspondiente al DS de la diap. anterior



Un DS (básico) casi no necesita explicación

36

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

37

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

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 la clase, el nombre (opcional) y la actividad de los objetos participantes

38

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

39

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 returnedo, *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 de otra manera, p.ej., más descentralizadamente

40

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

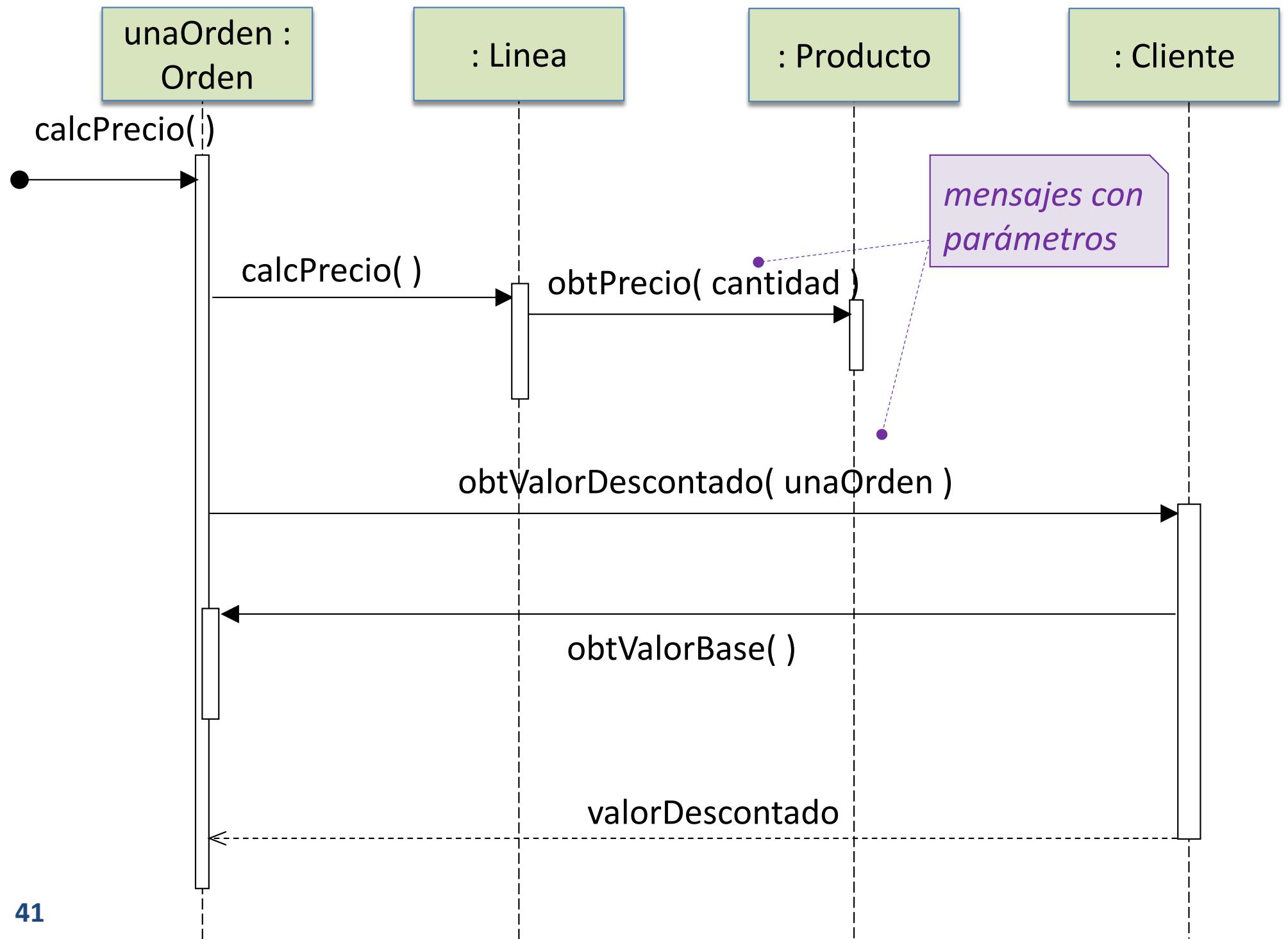
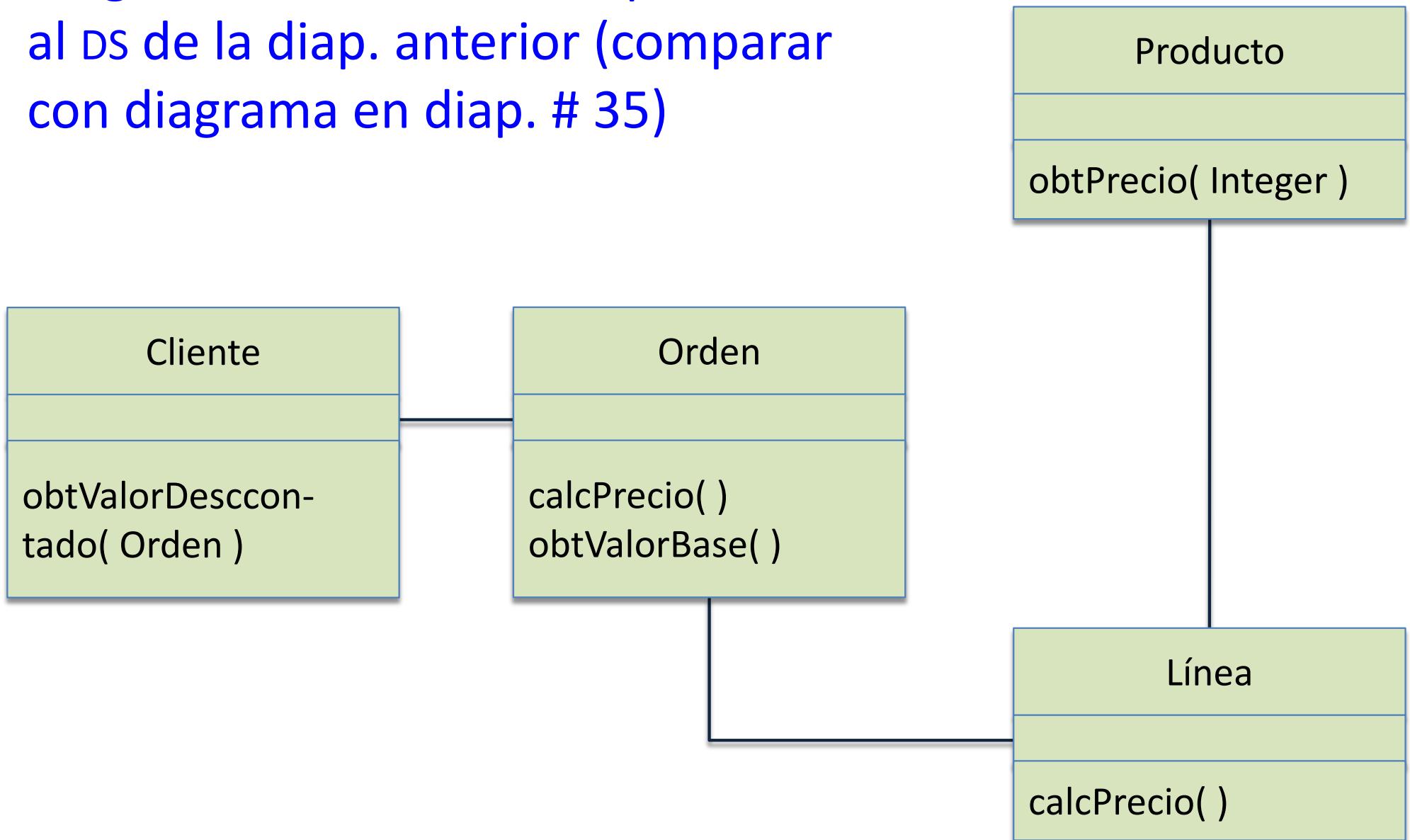


Diagrama de clases correspondiente al DS de la diap. anterior (comparar con diagrama en diap. # 35)



Los DS's muestran claramente cómo interactúan los participantes

43

Los diagramas de las diaps. #34 y #41 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:

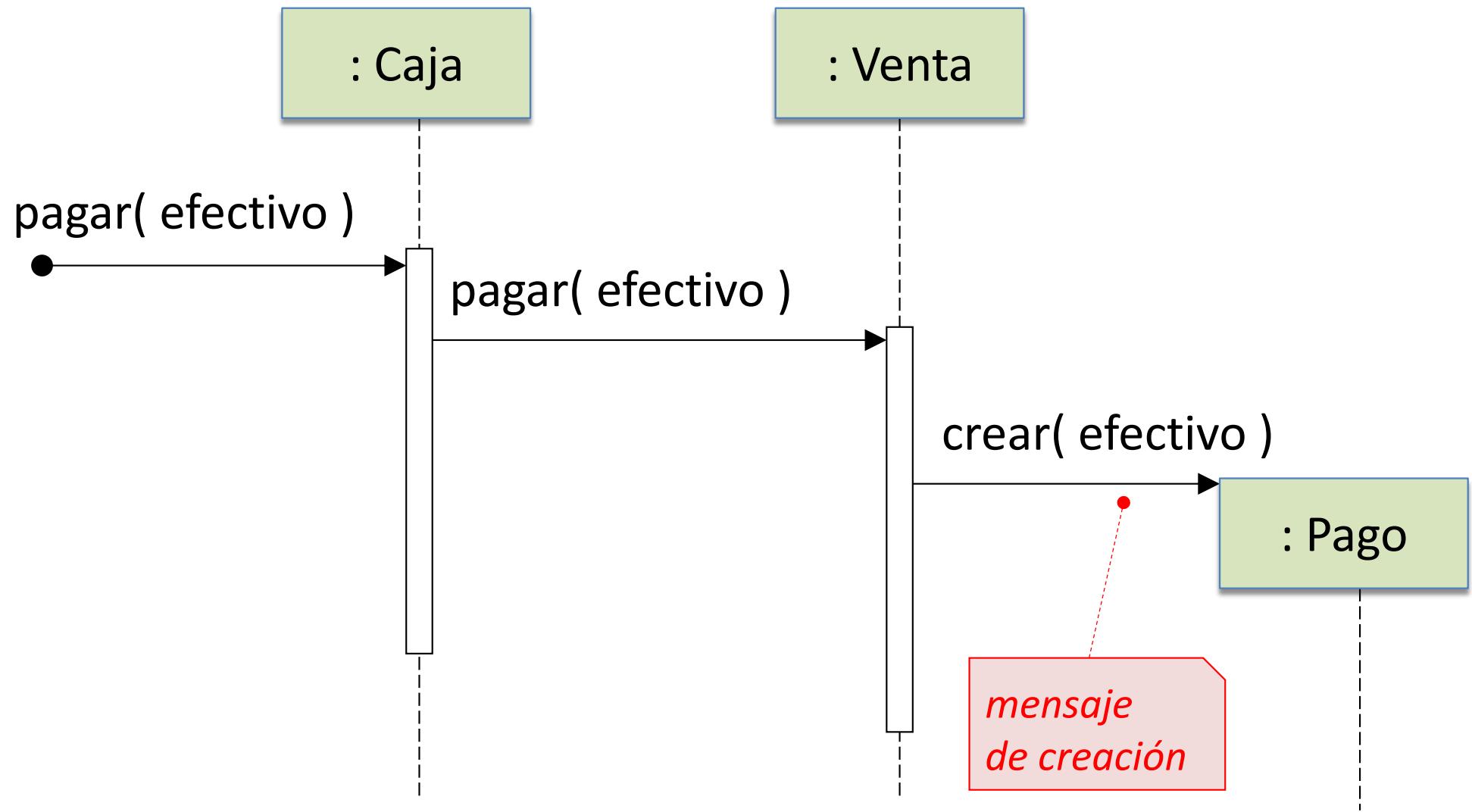
- p.ej., comportamiento condicional o repetitivo (diaps. #46 en adelante)

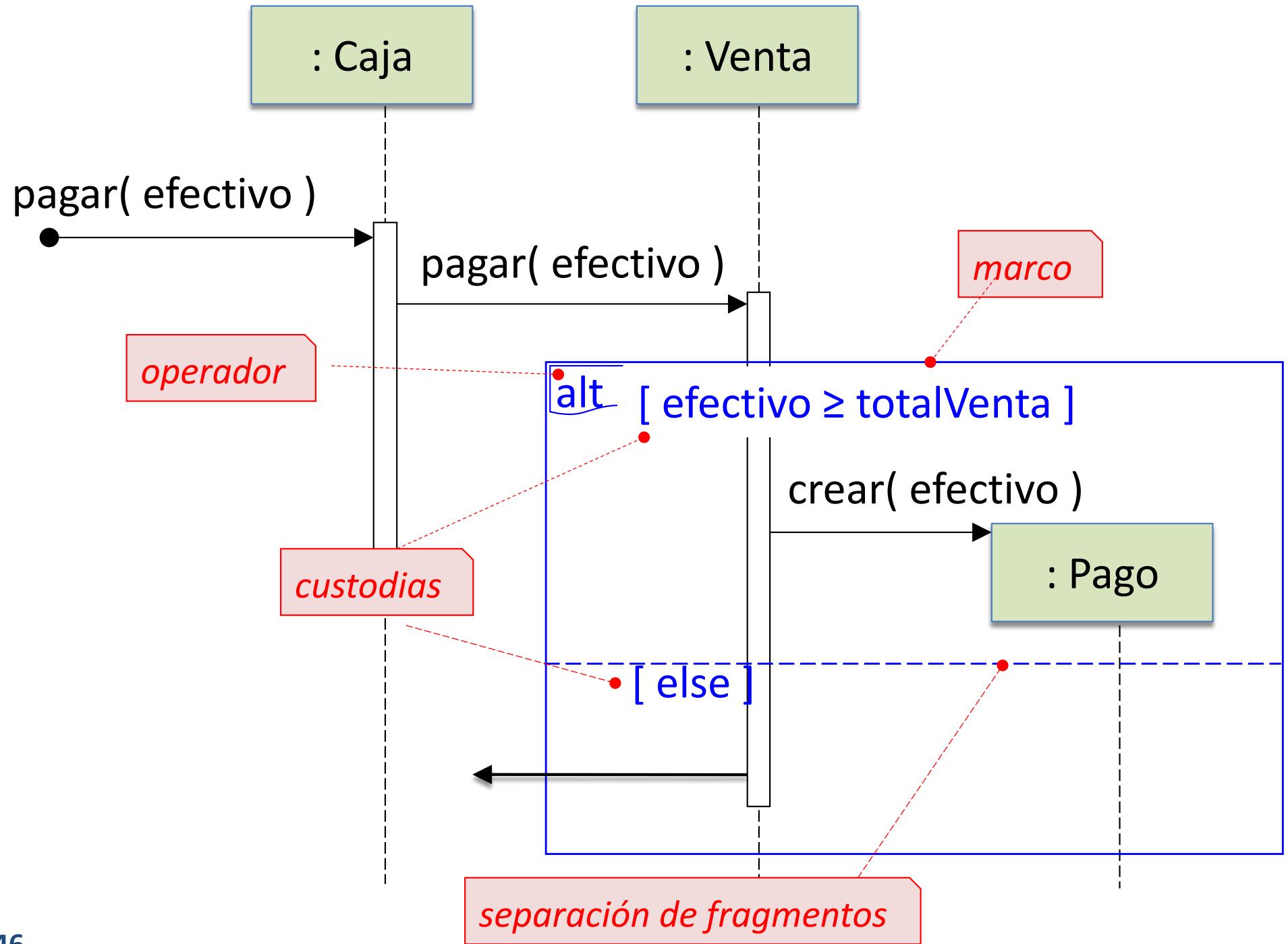
Un DS puede mostrar *creación de objetos* y *comportamiento condicional*

44

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





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

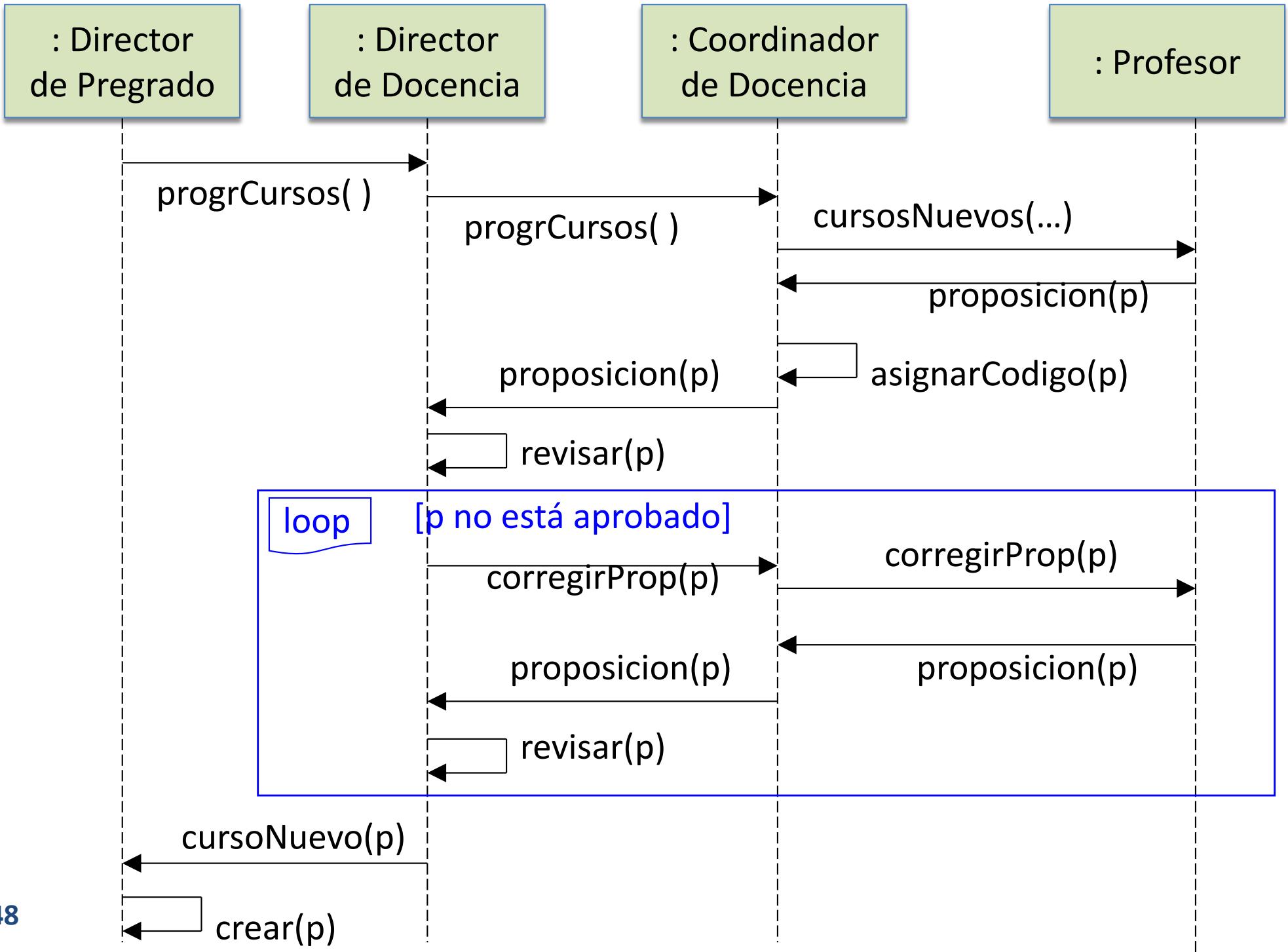
47

La región es dividida en *fragmentos*:

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

P.ej.,

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



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

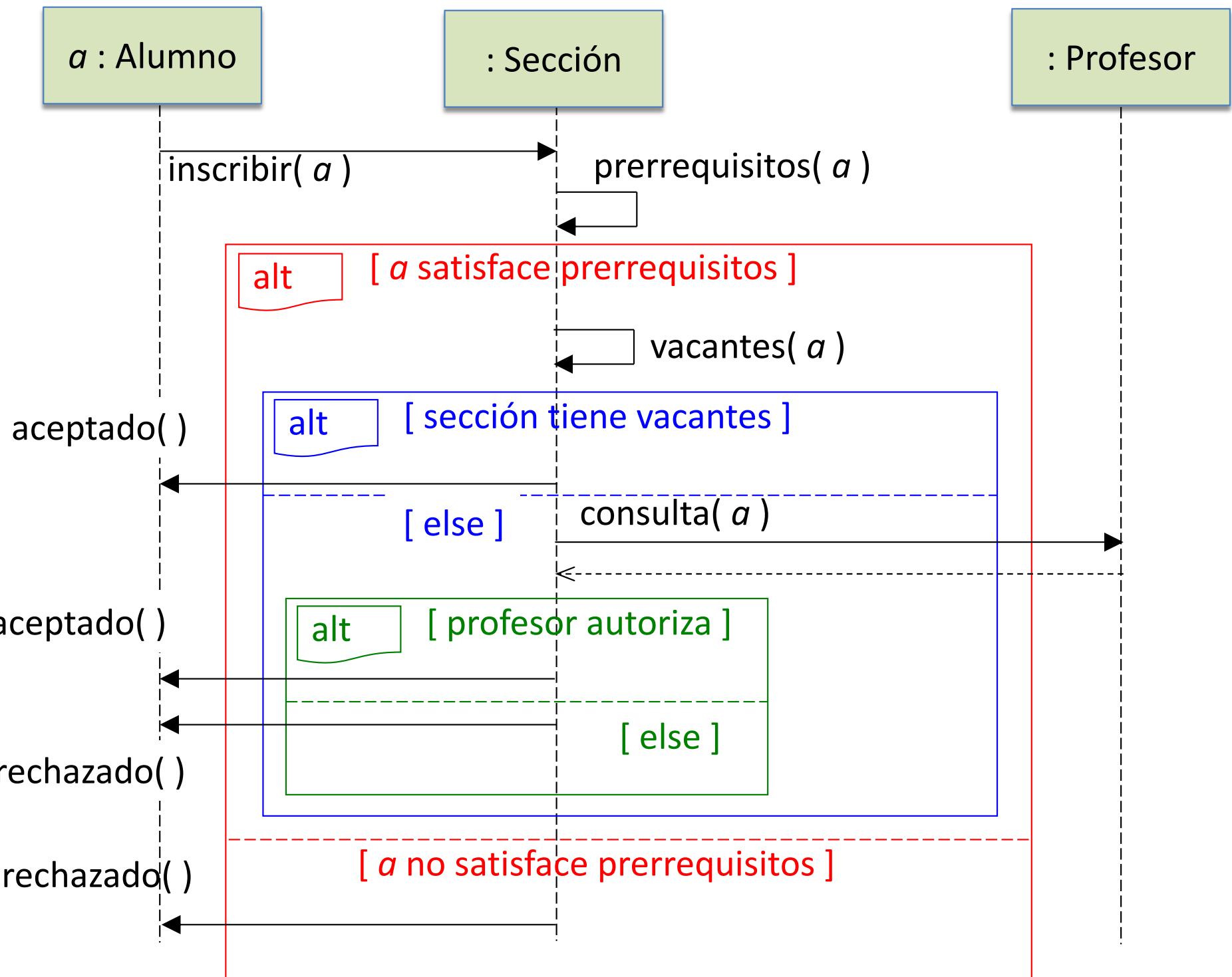
loop : Ciclo; el fragmento puede ejecutarse varias veces y la custodia indica la condición de la repetición

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



Es posible anidar unos marcos de interacción dentro de otros

52

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

El diagrama de máquina de estados

53

Para modelar el ciclo de vida de un objeto —comportamiento dinámico de clases— que puede estar en uno de un número finito de estados y que transita de un estado a otro

Estado:

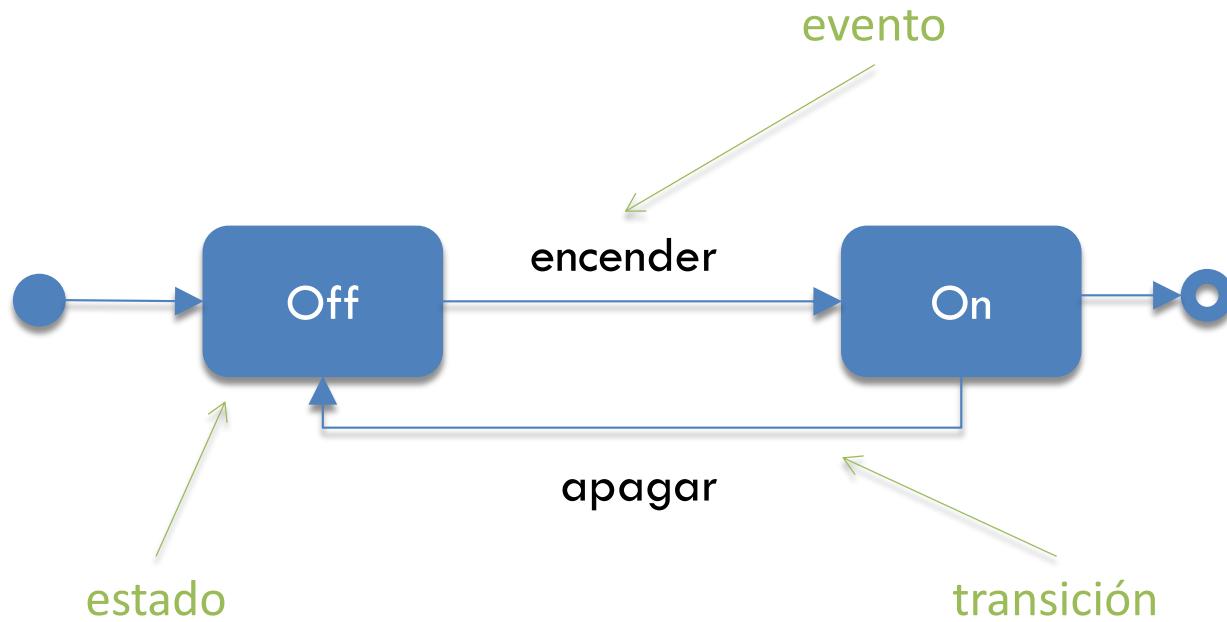
- una condición o situación durante la vida de un objeto durante la cual satisface alguna condición, ejecuta alguna actividad, o espera que ocurra algún evento

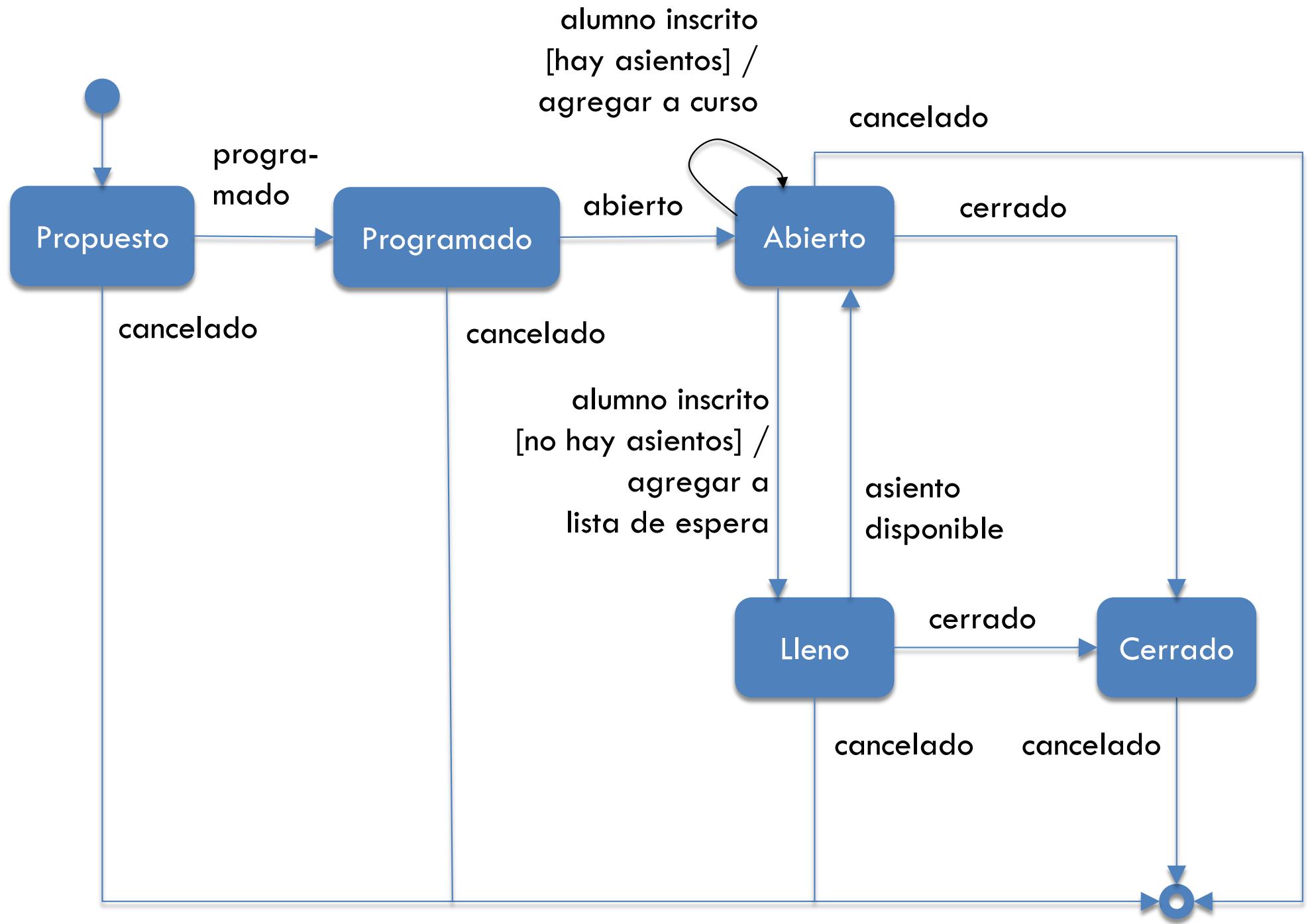
Evento:

- especificación de una ocurrencia significativa en tiempo y espacio

Transición:

- cambio de estado a otro en respuesta a un evento





El estado de un objeto es determinado por

- los valores de los atributos del objeto
- las relaciones que tiene con otros objetos
- las actividades que está ejecutando

Los objetos se envían mensajes unos a otros —eventos que pueden producir cambios en el estado del objeto

Identifiquemos los estados que hacen una diferencia para nuestro sistema:

- tiene que haber una diferencia semántica que hace la diferencia entre estados para que valga la pena modelarlos en una máquina de estados

La sintaxis para una transición es

evento1, evento2 [custodia] / acción

- los eventos (0 o más) especifican ocurrencias externas o internas que pueden gatillar la transición
- la custodia (0 o 1) es una condición que debe ser verdadera antes de que la transición pueda ocurrir
- las acciones (0 o más) es un trabajo que ocurre cuando ocurre la transición

El diagrama de actividad permite modelar los “pasos” en un proceso computacional

58

Modelamiento de los **aspectos dinámicos** de un sistema

Modelamiento de pasos secuenciales (y posiblemente concurrentes)

También, modelamiento del flujo de valores entre “pasos”

Ya sea, la **dinámica de un grupo de objetos**

... o bien el **flujo de control de una operación**

Es más usado en los siguientes casos

59

En el análisis:

- el flujo de un caso de uso
- el flujo entre casos de uso

En el diseño:

- los detalles de una operación
- los detalles de un algoritmo

En modelado de dominio:

- un proceso de negocios

Una *actividad* es una red de nodos conectados por aristas

60

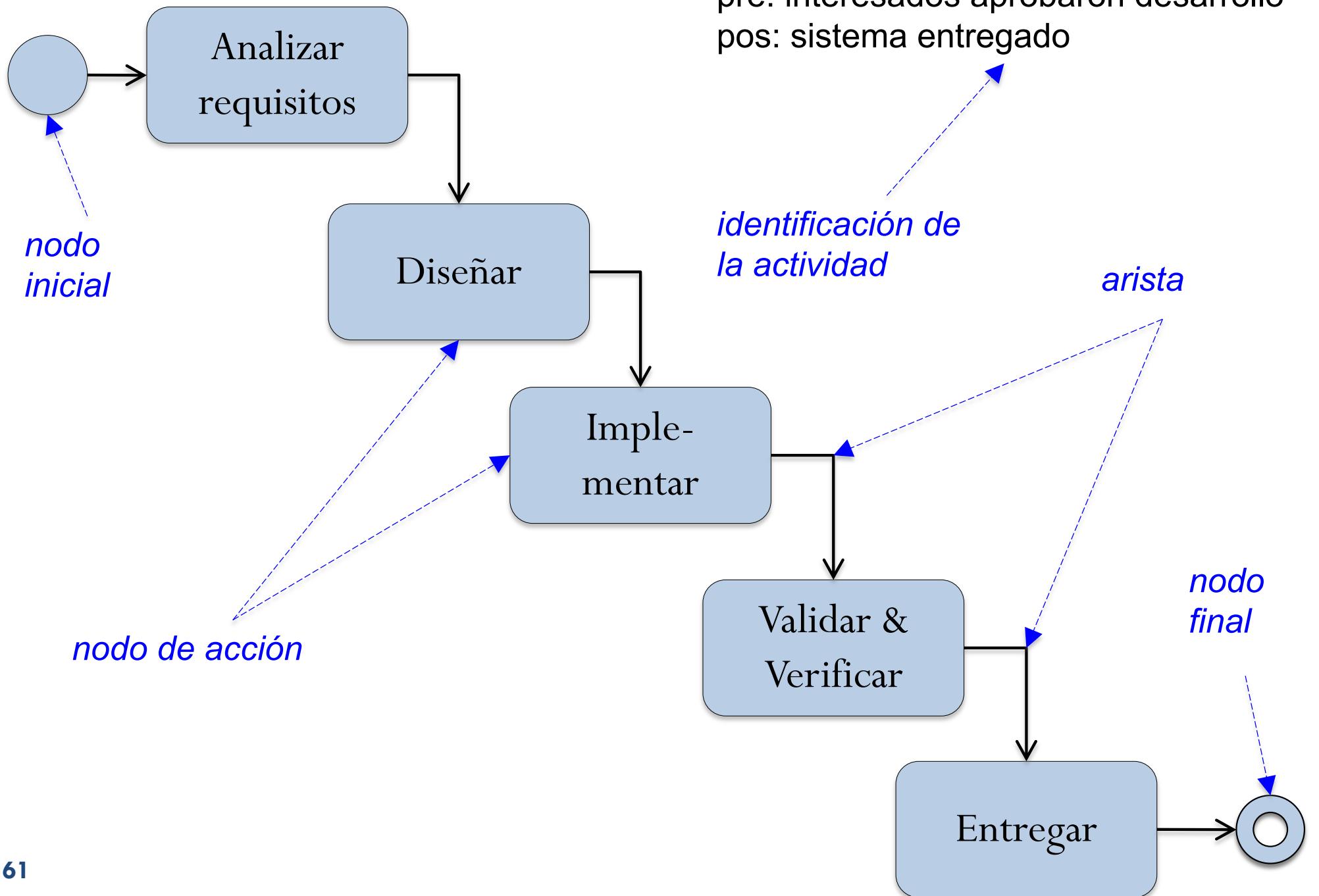
Hay **tres tipos de nodos**:

- acción —unidades de trabajo atómicas
- control —controlan el flujo
- objeto —objetos usados en la actividad

Las aristas representan flujos a través de las actividades

Hay **dos tipos de aristas**:

- flujos de control
- flujos de objetos



Una actividad tiene un nombre y puede tener una pre y una poscondición

Una actividad comienza y termina con un nodo de control —el nodo inicial y uno o más nodos finales

La actividad empieza en el nodo inicial, luego el control transita a los nodos de acción a través de las aristas — Analizar requisitos, Diseñar, etc.— hasta llegar a un nodo final

(Una acción también puede tener una pre y una poscondición)

En la próxima diapositiva
hay seis tipos de *nodos de control*

63

nodo inicial —inicia una actividad

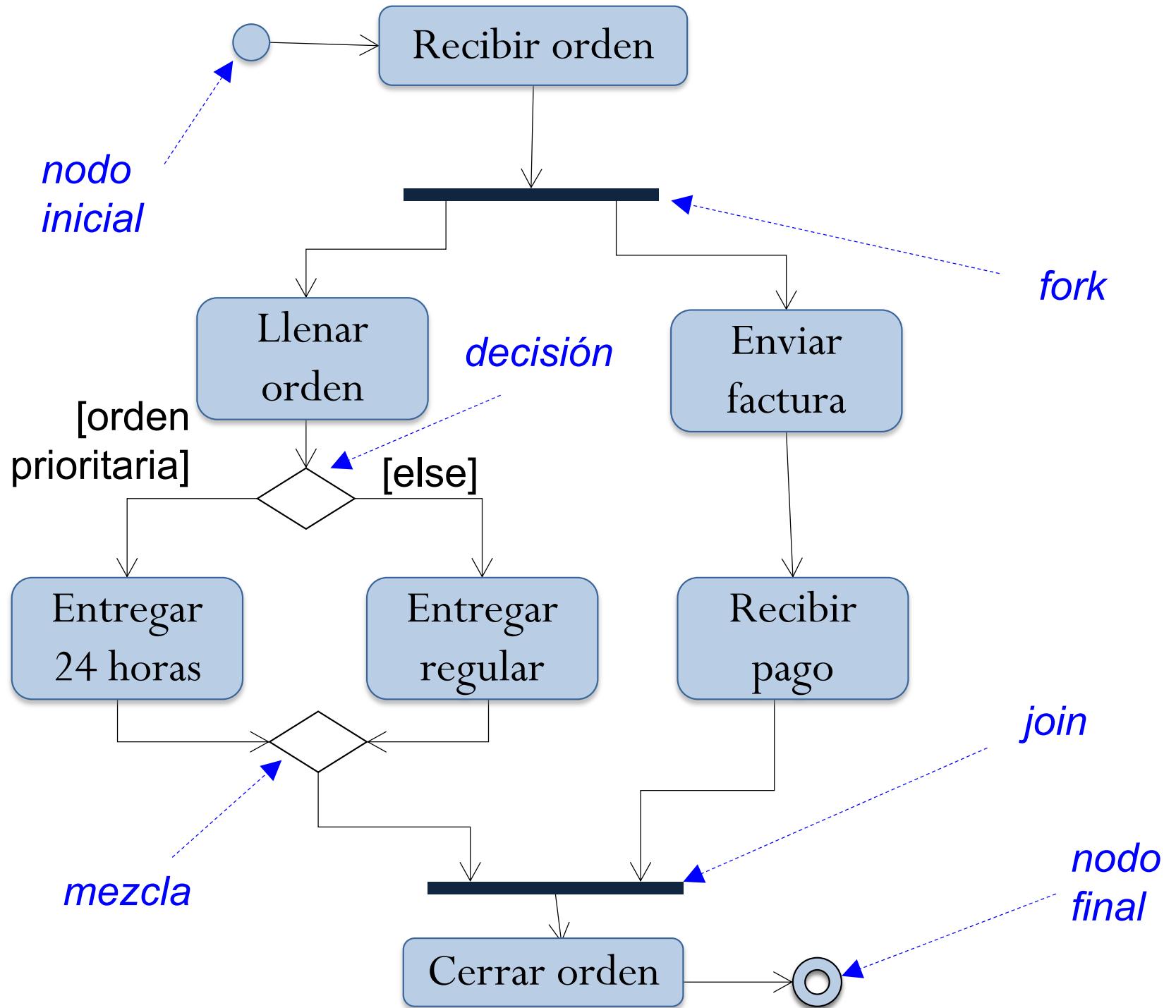
nodo final —termina una actividad

fork —diap. #60

join —diap. #61

decisión —diap. #62

mezcla —diap. #63



Un *fork* tiene un flujo de entrada y varios flujos de salida concurrentes

65

El diagrama sólo establece las reglas de secuenciación esenciales que hay que seguir:

- lo que importa es la secuencia entre las acciones de un mismo flujo de salida
- la secuencia entre las acciones de los diferentes flujos de salida es irrelevante

P.ej., en la diap. anterior,

- uno puede llenar la orden, enviar la factura, entregar y luego recibir el pago
- o también, enviar la factura, recibir el pago, llenar la orden y luego entregar

Un *join* tiene varios flujos de entrada y un flujo de salida

66

Como tenemos paralelismo, necesitamos sincronización —
join:

- en un join, el flujo de salida es tomado sólo cuando todos los flujos de entrada llegan al join
- p.ej., no cerramos una orden hasta que haya sido entregada y está pagada —usamos el join antes de la acción Cerrar Orden

Una acción puede tener varios flujos de entrada —hay un join implícito entre ellos:

- la acción ejecuta sólo cuando todos los flujos han sido tomados

El comportamiento condicional es especificado mediante *decisiones* ...

67

Una **decisión** tiene un único flujo de entrada y varios flujos de salida *custodiados*:

- cada vez que se llega a una decisión, se puede tomar sólo uno de los flujos de salida —las *custodias* deben ser mutuamente excluyentes
- una custodia [*else*] indica que ese flujo debe tomarse si todas las otras custodias son falsas

... y mezclas

68

Una **mezcla** tiene varios flujos de entrada y sólo un flujo de salida:

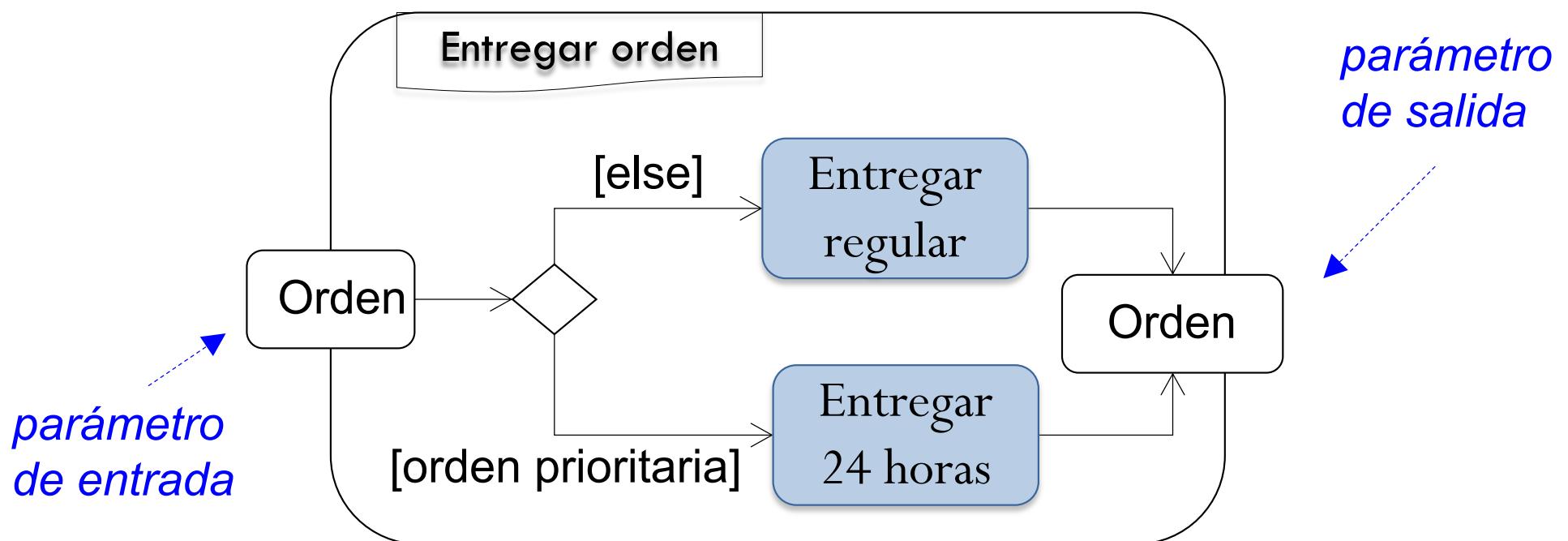
- simplemente, marca el fin de un comportamiento condicional iniciado por una decisión

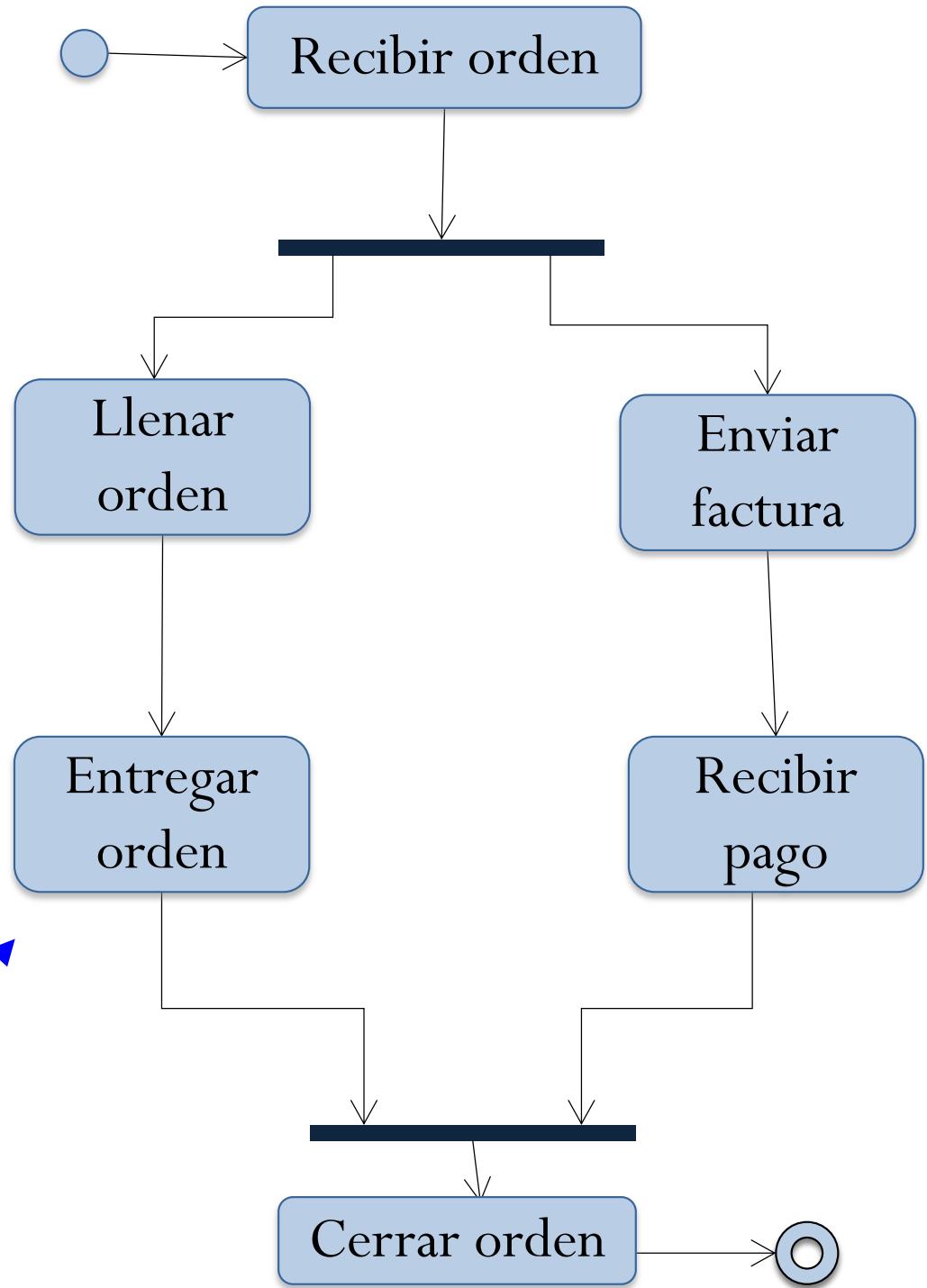
Las acciones pueden ser compuestas / descompuestas en (sub)actividades

69

P.ej.,

- podemos tomar la lógica de entrega de una orden y definirla como su propia actividad
- luego, podemos *llamarla*, como a una acción





esta acción corresponde
al diagrama de la diap.
anterior

Podemos mostrar quién ejecuta las acciones usando *particiones*

71

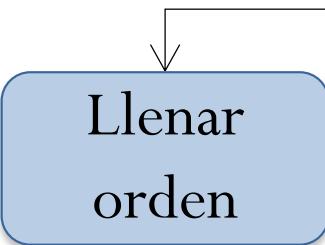
Los diagramas de actividad dicen lo que pasa, pero no quién hace qué:

- el diagrama no muestra cuál clase (u objeto) es responsable de cada acción

Podemos —opcionalmente— dividir el diagrama en **particiones**, que muestran cuáles acciones son llevadas a cabo por una clase:

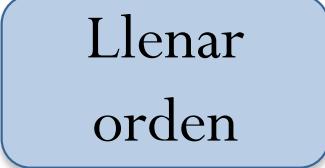
- se puede particionar a lo largo de una o dos dimensiones — columnas (carriles de natación), o columnas y filas
- cada dimensión puede ser dividida a su vez

Departamento
de Despachos

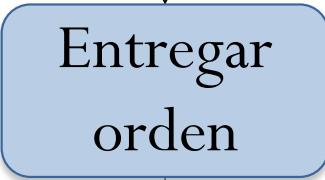


Recibir
orden

Departamento
de Servicio al
Cliente



Enviar
factura



Recibir
pago



Cerrar
orden

Departamento
de Finanzas