

Diseño de Software Orientado a Objetos II

Ingeniería de Software – IIC2143

2-2015

Yadran Eterovic

Resumen hasta ahora

Sept. 3:

- ejemplo de diseño e implementación OO

Sept. 8:

- los diagramas de clases (de software) y de secuencia de UML

Sept. 10:

- ejemplo del proceso de diseño OO y aplicación de algunos principios

Patrones de asignación de responsabilidades

Es posible nombrar y explicar los principios y el razonamiento necesarios para entender lo básico del diseño de objetos,

... asignando responsabilidades a los objetos de una manera metódica, racional, explicable

Este enfoque para entender y usar principios de diseño

... está basado en los **patrones de asignación de responsabilidades**

Patrones

Los desarrolladores experimentados construyen un repertorio, tanto de principios generales, como de soluciones particulares, que los guían en la creación de software

Al darle un nombre y codificar estos principios en un formato estructurado que describe el problema y la solución, los llamamos *patrones*

Un **patrón** es una descripción de un problema y su solución, con un nombre y que puede ser aplicado en nuevos contextos:

- idealmente, nos aconseja cómo aplicar la solución en diversas circunstancias

Cuidado con la idea de un “nuevo patrón”: no hay tal cosa

“Patrón” sugiere algo que se viene repitiendo por mucho tiempo

Los patrones de diseño **no son** nuevas ideas de diseño

Los mejores patrones tratan de codificar conocimiento y principios existentes que han demostrado ser eficaces:

- mientras más antiguo y ampliamente usado, mejor

Los patrones que vamos a estudiar no establecen nuevas ideas:

- por el contrario, **le dan nombre y codifican principios básicos ampliamente usados**
- a un diseñador experto, parecerán familiares

La herramienta de diseño crítica para desarrollo de software es

... una mente bien educada en los principios de diseño

Principios de diseño OO

Vamos a volver sobre ellos más adelante

Encapsular lo que cambia

Codificar con respecto a una interfaz y no a una implementación

Cada clase debe tener sólo una razón para cambiar

Las clases son acerca de comportamiento y funcionalidad

Las clases deben estar abiertas para extenderlas, pero cerradas para modificarlas

Evitar el código duplicado, colocando las cosas que son comunes en un único lugar

Todo objeto debe tener una sola responsabilidad

Las subclases deben ser apropiadas para su clase base

Otra formulación: Los 9 principios GRASP

Experto en Información (o Experto)

Creador

Alta Cohesión

Bajo Acoplamiento

Controlador

Polimorfismo

Indirección

Fabricación Pura

Variaciones Protegidas

El principio *Experto en Información*

Solución: Asignemos una responsabilidad a la clase que tiene la información necesaria para cumplirla —la experta o el experto

Problema: ¿Cuál es un principio general de asignación de responsabilidades a objetos?

Ejemplos:

- en el caso del juego “The Trivium”, ¿a quién le enviamos el mensaje de lanzar el dado?
- en una aplicación de órdenes de compra, alguna clase tiene que saber el monto total de la orden

Para asignar una responsabilidad, hay que establecer claramente la responsabilidad:

- ¿en qué clase ponemos el método que simula el lanzamiento del dado?
- ¿quién debiera ser responsable de saber el monto total de una orden?

Según *Experto*, busquemos una clase que tenga la información necesaria:

- si hay clases relevantes en el modelo del diseño (clases de software), buscamos ahí primero
- de lo contrario, buscamos en el modelo del dominio (clases conceptuales)
- ... y usamos o expandimos sus representaciones para crear las clases de diseño correspondientes

En el caso de “*The Trivium*”, ¿a quién enviamos el mensaje de *lanzar el dado*?*

De acuerdo con el principio *Experto en Información* ...

... a la clase que tiene la información necesaria para lanzar el dado

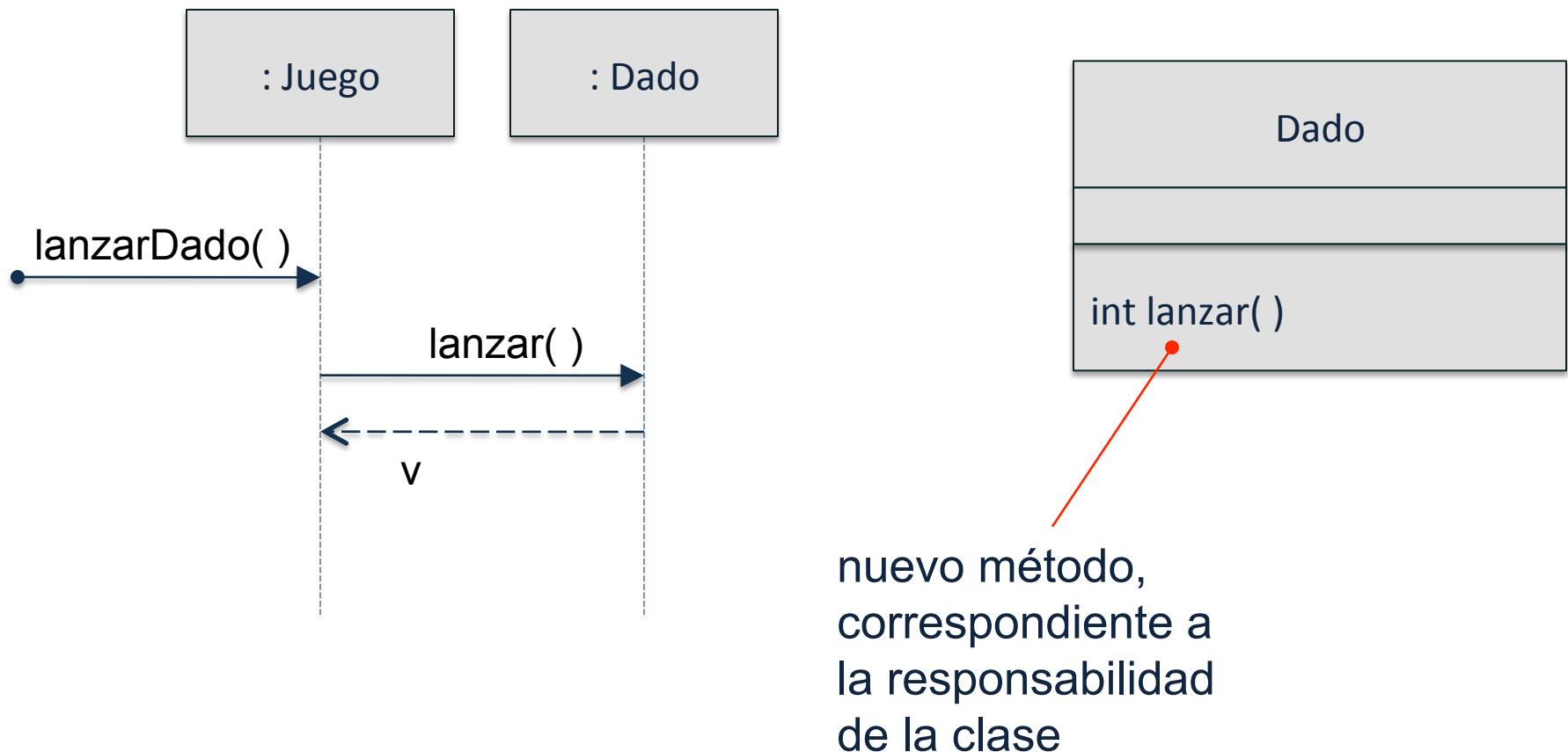
En este caso, Dado

- ... aunque podría ser Juego (o cualquiera otra), porque “lanzar el dado” no requiere ninguna información especial, es decir, no depende del estado de ningún objeto

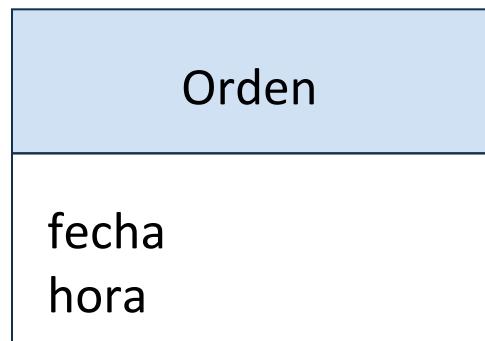
Según el diagrama de clases, el :Juego puede pedirle directamente al :Dado que “se lance”

* Ejemplo visto en más detalle en la clase anterior

Diagrama de secuencia según *Experto*, e implicación sobre el diagrama de clases



Experto: Ejemplo



1

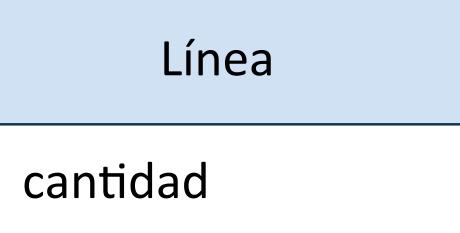
Contiene

1..*

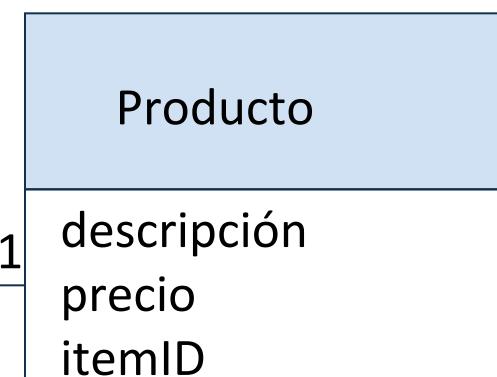
*

Descrito-por

1



cantidad



fecha
hora

descripción
precio
itemID

t = obtTotal()

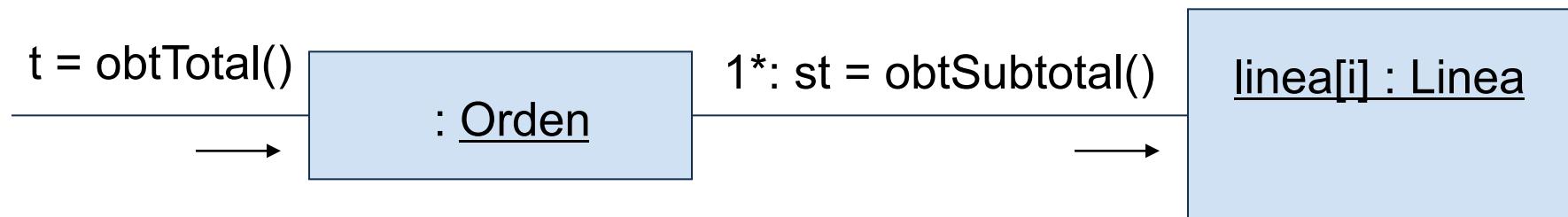
: Orden

fecha
hora

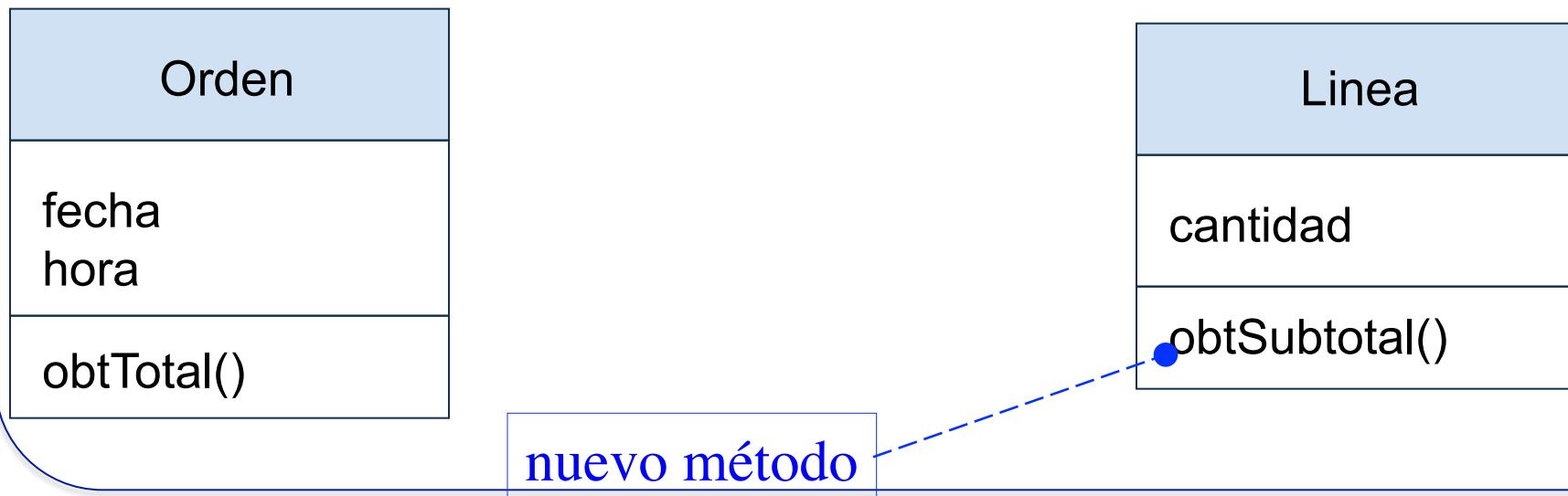
obtTotal()

nuevo método

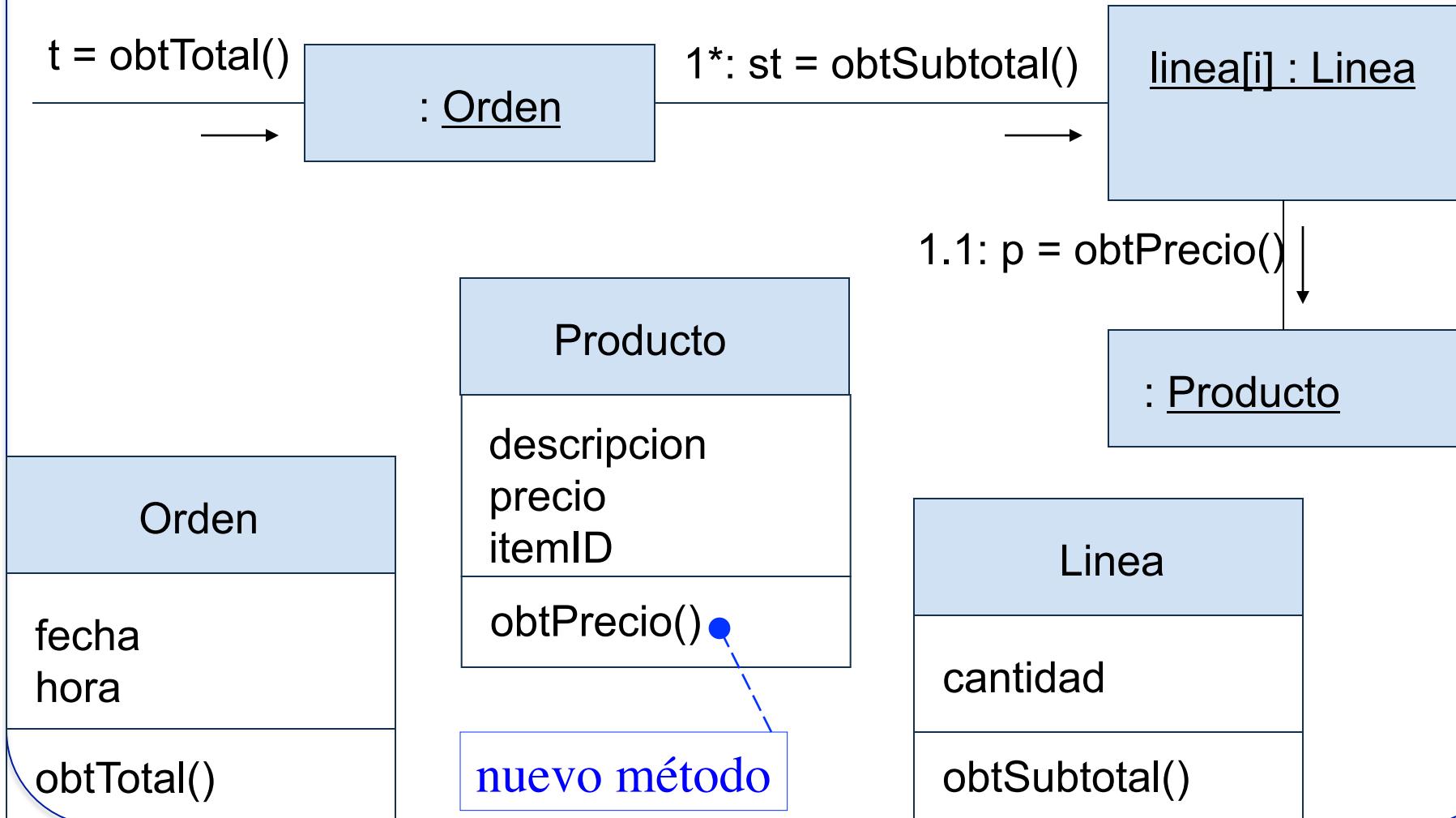
La Orden sabe cuáles Linea's la componen; si cada Linea calcula su propio subtotal, entonces Orden puede calcular su total.



[Diagrama de comunicación (equivale a un diagrama de secuencia): 1* y linea[i] implican iteración sobre todos los elementos de la colección]



Cada Linea conoce el producto que le corresponde (Producto), al que consulta el precio; como la Linea almacena la cantidad de artículos, puede determinar su subtotal.



En resumen, para cumplir la responsabilidad de saber y poder responder el total de la orden, asignamos tres responsabilidades a tres clases del diseño:

Clase del diseño	Responsabilidad
Orden	conoce el total de la orden
Línea	conoce el subtotal del ítem
Producto	conoce el precio del producto

Hacemos esta asignación de responsabilidades mientras dibujamos diagramas de secuencia

El principio *Bajo Acoplamiento*

Solución: Asignar una responsabilidad de modo que el *acoplamiento* permanezca bajo

Problema: ¿Cómo favorecer una baja dependencia, un bajo impacto de cambios, y un mayor reuso?

Ejemplos:

- ¿Cómo obtiene el Juego la pregunta que tiene que hacerle al jugador?
- Considerando las clases Pago, Caja y Orden, ¿quién debería ser responsable de crear una instancia de Pago y asociarla con la Orden?

El **acoplamiento** mide qué tan fuertemente una clase está conectada a, sabe de, o necesita otras clases

Características —más bien negativas— de una clase con alto acoplamiento:

- es afectada por cambios en las clases asociadas
- es más difícil de entender por sí sola y de reusar,
- ... ya que necesita la presencia de las clases de las cuales depende

En el caso de “*The Trivium*”, podemos conectar Juego, Tablero y Casilla de dos maneras*

Una posibilidad:

- conectar Juego con Casilla en el diagrama de clases
- ... luego, hacer que el :Tablero devuelva la :Casilla al :Juego
- ... finalmente, hacer que el :Juego le pida la pregunta a la :Casilla

Otra posibilidad:

- hacer que el :Tablero devuelva la casilla al :Juego
- ... luego, hacer que el :Juego informe al :Jugador de su nueva :Casilla
- ... finalmente, hacer que el :Jugador le pida la pregunta a la :Casilla

* Ejemplo visto en la clase anterior

Las alternativas anteriores tienen distinto efecto sobre el acoplamiento entre clases

Primera posibilidad:

- aumenta el acoplamiento entre las clases, al conectar Juego con Casilla

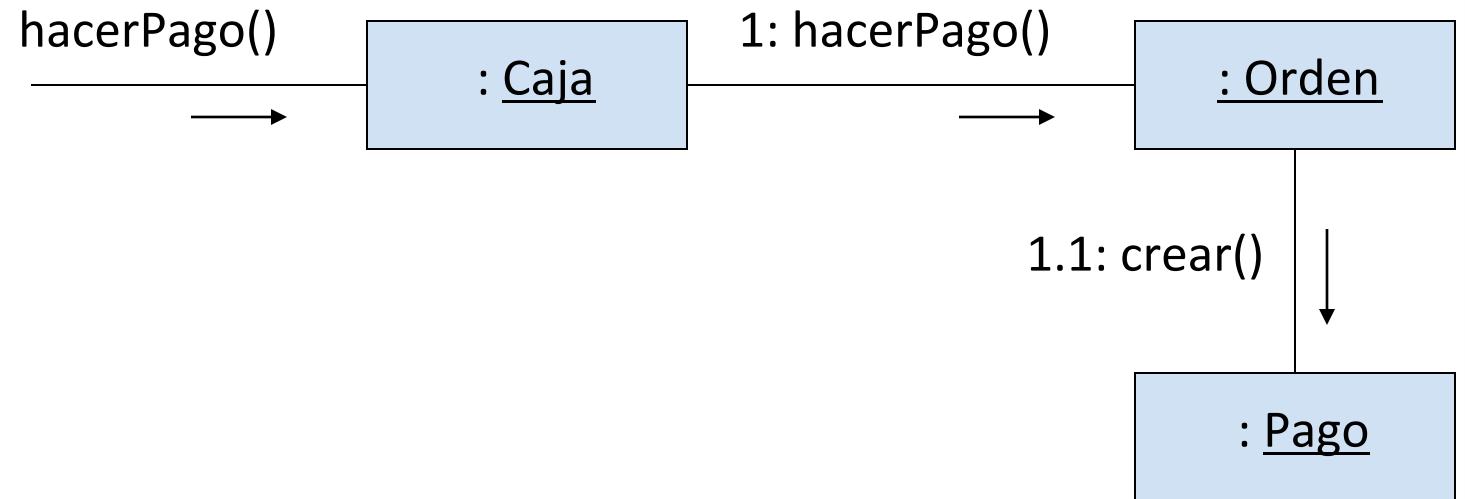
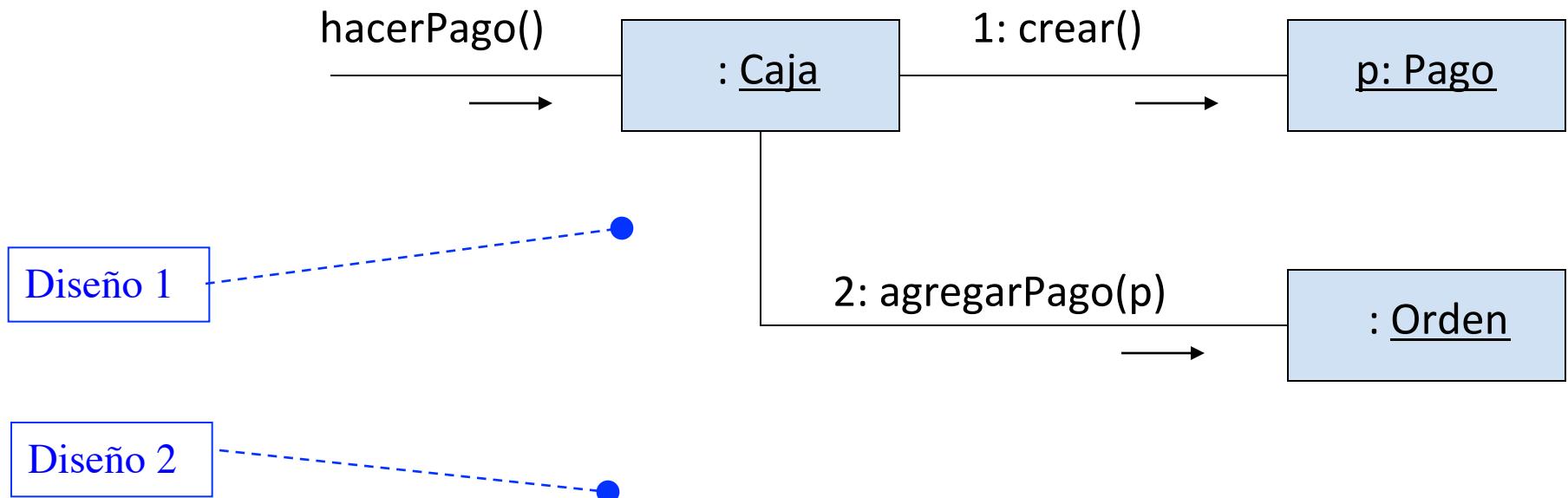
Segunda posibilidad:

- no aumenta el acoplamiento, ya que no requiere nuevas conexiones entre clases

En consecuencia, según el principio *Bajo Acoplamiento*, preferimos la segunda alternativa

Diseño 1. La caja “registra” los pagos en el mundo real —*Creador* sugiere a Caja para crear el Pago:

- Caja envía un mensaje `agregarPago()` a la Orden, junto con el nuevo Pago como parámetro
- esta asignación de responsabilidades acopla la clase Caja a conocimiento acerca de la clase Pago



Diseño 2. La Caja envía un mensaje hacerPago() a la Orden y ésta crea el Pago

En ambos casos la Orden sabe del Pago:

- el diseño 1 agrega el acoplamiento de Caja con Pago
- el diseño 2 no aumenta el acoplamiento

El principio *Alta Cohesión*

Solución: Asignar una responsabilidad de modo que la cohesión permanezca alta

Problema: ¿Cómo mantener los objetos focalizados, fáciles de entender y manejables, y de paso favorecer el Bajo Acoplamiento?

Ejemplo:

- El mismo ejemplo anterior de la Caja, el Pago y la Orden (de compra)

La **cohesión** mide qué tan fuertemente relacionadas o focalizadas son las responsabilidades de una clase

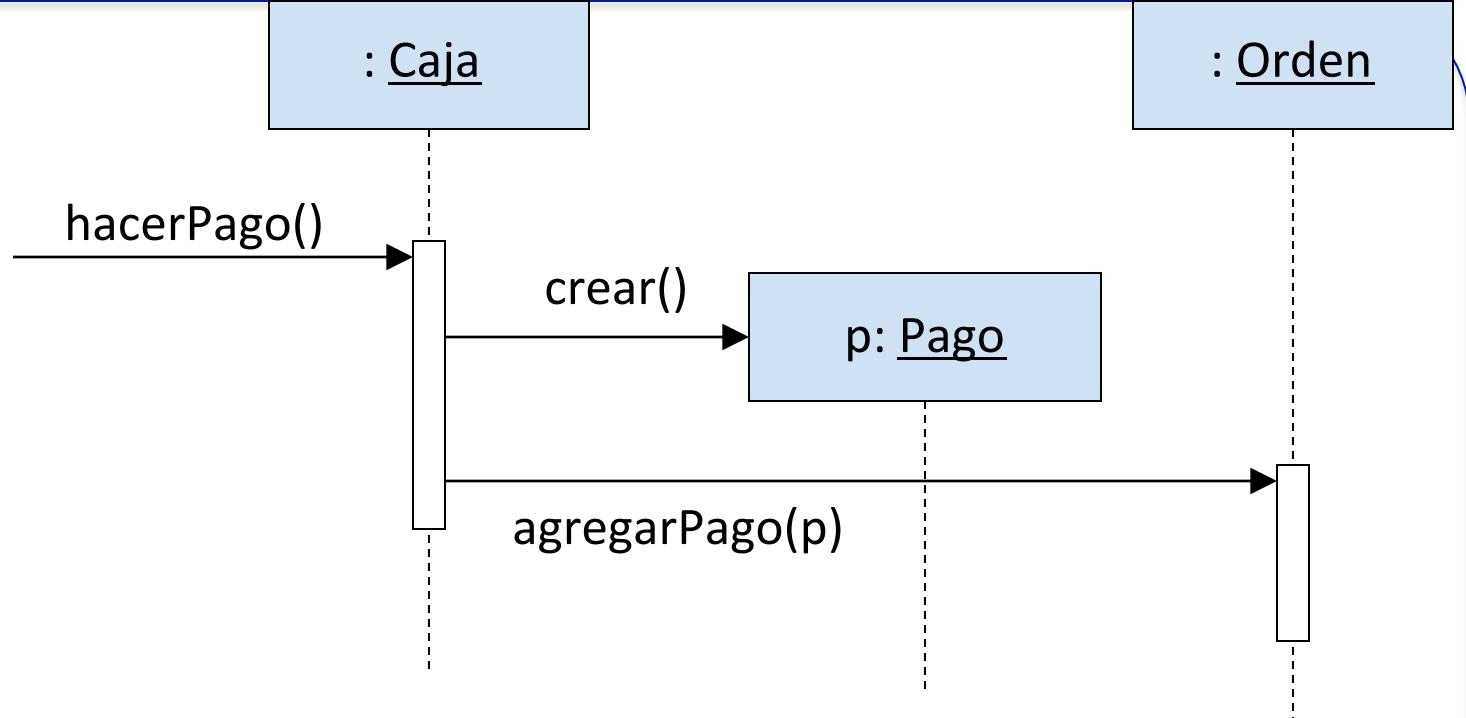
Características de una clase con baja cohesión:

- hace muchas cosas no relacionadas
- hace demasiadas cosas
- es difícil de comprender, reusar y mantener
- se ve constantemente afectada por cambios
- representa una abstracción de “granularidad muy grande”
- ha asumido responsabilidades que debieran ser delegadas a otras clases

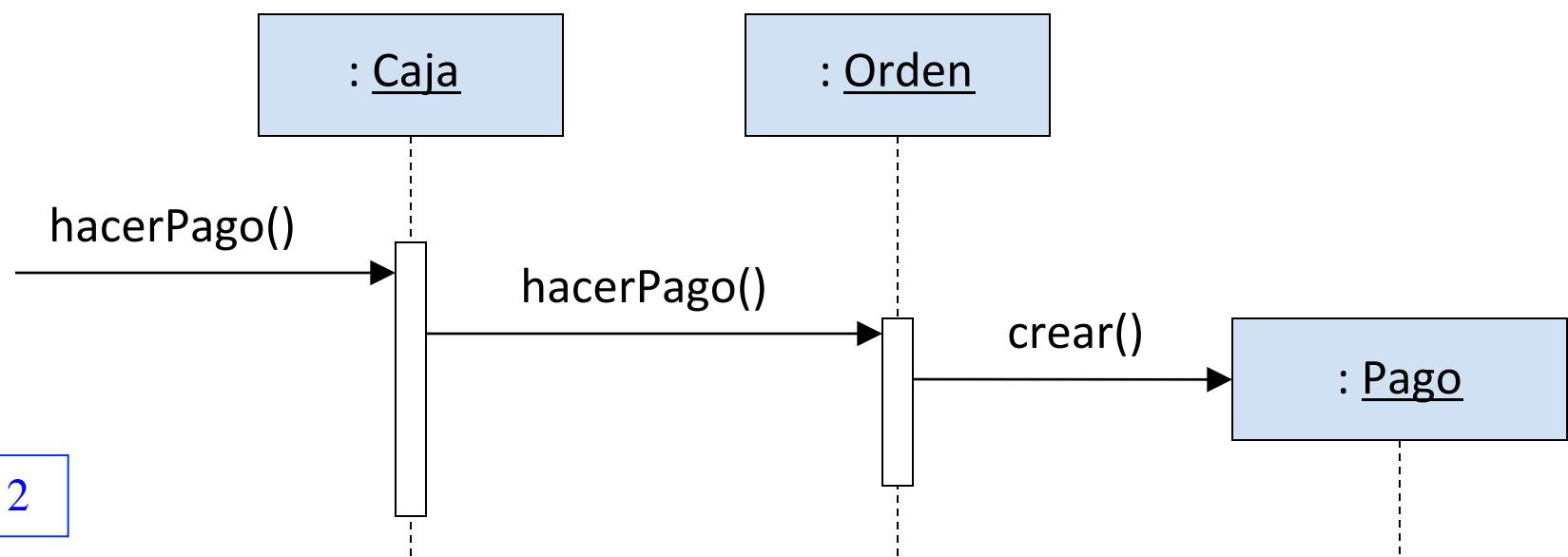
Diseño 1. En el mundo real, la caja registra los pagos —*Creador* sugiere a Caja para crear el Pago:

- la Caja asume parte de la responsabilidad de cumplir con la operación hacerPago() del sistema
- por sí solo, es aceptable,
- ... pero si seguimos haciendo a Caja responsable de hacer operaciones del sistema, terminará cargada con tareas y se volverá “incohesiva”

Diseño 1



Diseño 2



Diseño 2. Delegamos a la Orden la responsabilidad de crear el Pago:

- favorece una mayor cohesión de la Caja
- también favorece bajo acoplamiento —es mejor

Bajo Acoplamiento no es absoluto

Al diseñar, siempre tengamos presente *Bajo Acoplamiento* —es un principio evaluativo

Un acoplamiento alto con elementos estables no es problema:

- p.ej., una aplicación en Java puede acoplarse sin problemas a las bibliotecas de Java

Hay que focalizarse en los puntos de *alta inestabilidad real*:

- p.ej., subsistemas diversos con similares funcionalidades provistos por terceros

Formas comunes de acoplamiento entre dos clases, X e Y

X tiene un atributo que se refiere a una instancia de Y —o a Y propiamente tal

Un objeto X llama a servicios de un objeto Y

X tiene un método que hace referencia a una instancia de Y , o a Y propiamente tal —parámetro, variable local, valor de retorno

X es una subclase directa/indirecta de Y

Y es una interfaz —colección de firmas de operaciones públicas— y X implementa la interfaz

La herencia produce alto acoplamiento

Una subclase está fuertemente acoplada a su superclase:

- hay que considerar cuidadosamente la decisión de衍生 una clase desde una superclase

P.ej., supongamos que hay objetos que necesitan ser almacenados persistentemente:

- es común crear una clase abstracta *ObjetoPersistente* de la cual deriven las otras clases
- ventaja: herencia automática del comportamiento persistente
- desventaja: acopla fuertemente los objetos del dominio a un servicio técnico particular y mezcla diferentes temas arquitectónicos

La ausencia (total) de acoplamiento no es aconsejable

El caso extremo de bajo acoplamiento es cuando no hay acoplamiento entre las clases:

- no es deseable —una metáfora central en OO es un conjunto de objetos que se comunican mediante mensajes
- muy bajo acoplamiento lleva a unos pocos objetos no cohesivos y complejos que hacen todo
- un grado moderado de acoplamiento entre clases es normal y necesario en un sistema en que las tareas son cumplidas mediante una colaboración entre objetos

No hay una medida absoluta de cuándo hay demasiado acoplamiento:

- las clases inherentemente genéricas y con alta probabilidad de reuso debieran tener bajo acoplamiento

Definición práctica de alta cohesión

Siempre tengamos presente *Alta Cohesión*:

- también es un principio evaluativo

Existe **alta cohesión** en los siguientes casos:

- cuando la clase tiene pocos métodos, que no hacen muchas cosas sino trabajan todos juntos para proporcionar un comportamiento bien delimitado
- cuando la clase colabora con otras clases para compartir el esfuerzo si la tarea es muy compleja

Muy baja cohesión:

- una clase es responsable única de funcionalidades diferentes; p.ej., interactuar con bases de datos y manejar llamadas a procedimientos remotos

Baja cohesión:

- una clase es responsable única de una tarea compleja; p.ej., interactuar con bases de datos

Alta cohesión:

- una clase es parcialmente responsable de un área funcional; p.ej., interactuar con bases de datos

Cohesión moderada:

- una clase tiene pocas y únicas responsabilidades en áreas diferentes; p.ej., conocer a sus empleados y conocer su información financiera

Bajo Acoplamiento + Alta Cohesión

El nivel de acoplamiento o el de cohesión no deben ser considerados aisladamente de otros principios

... tales como *Experto* y *Alta Cohesión* o *Bajo Acoplamiento*, respectivamente

Modularidad es la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados

Dónde estamos y para dónde vamos

Hemos revisado tres principios generales de diseño de software:

- Experto en Información
- Bajo Acoplamiento
- Alta Cohesión

Veamos otros, algo más particulares:

- **Creador**
- **Controlador**
- Polimorfismo
- Indirección
- Fabricación Pura
- Variaciones Protegidas

El principio *Creador*

Solución: Asignar a la clase B la responsabilidad de crear una instancia de la clase A —B es un creador de objetos A— si al menos uno de los siguientes es verdadero:

- B contiene o agrega objetos A (preferir esta opción)
- B registra instancias de objetos A
- B usa directamente objetos A
- B tiene los datos de inicialización que se pasará a A

Problema: ¿Quién debiera ser responsable de crear una nueva instancia de alguna clase?

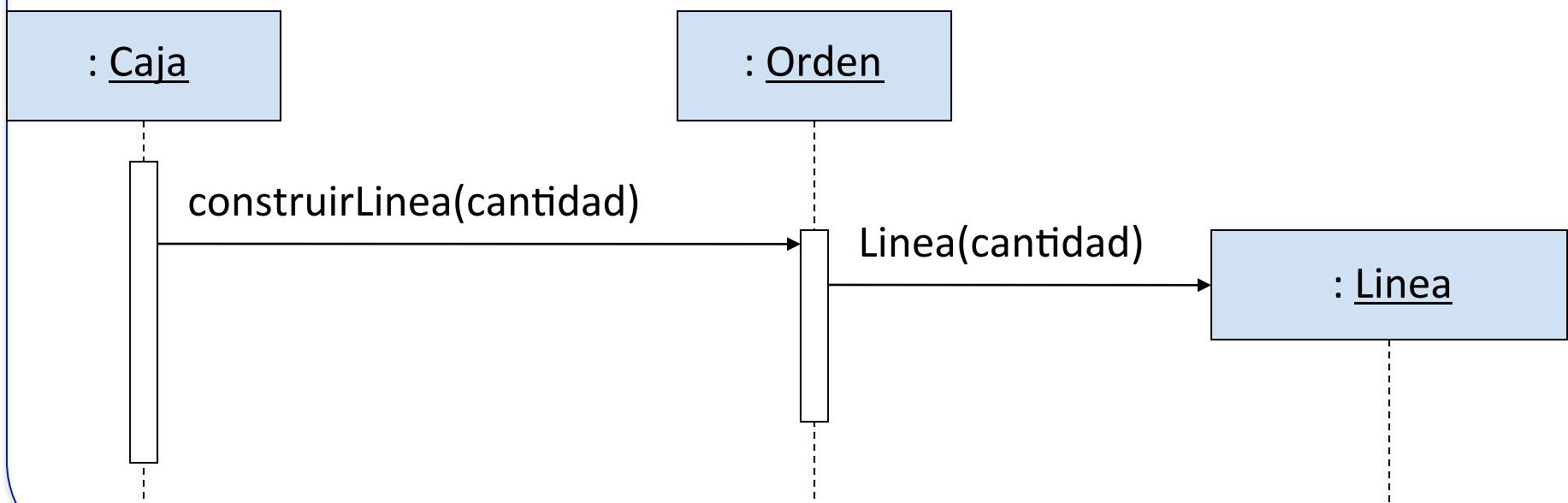
Ejemplos:

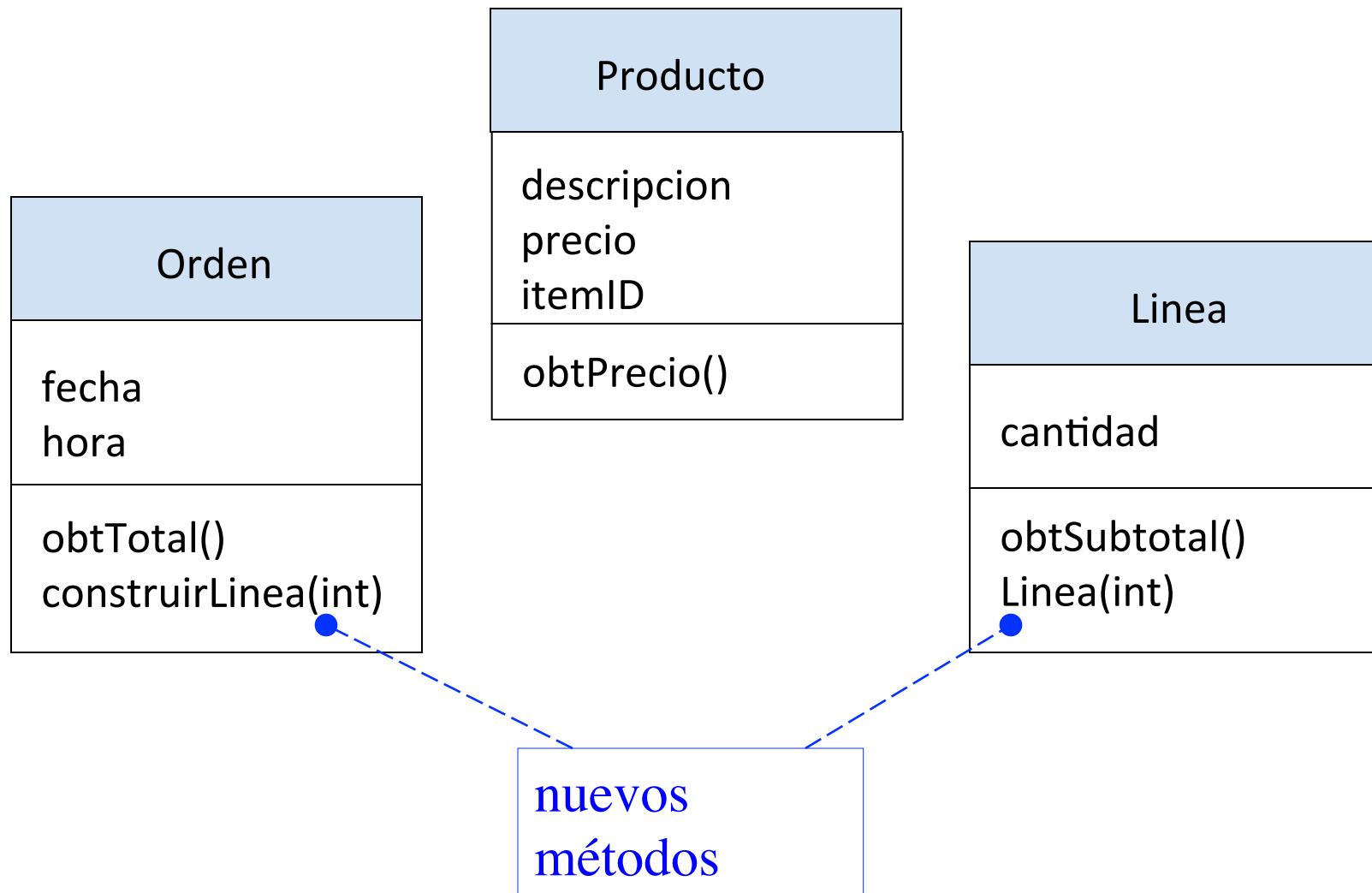
- en la aplicación de órdenes, ¿quién debería crear instancias de Linea?
- en el juego, ¿quién crea los objetos que representan a los jugadores?

En el modelo de dominio de la clase pasada:

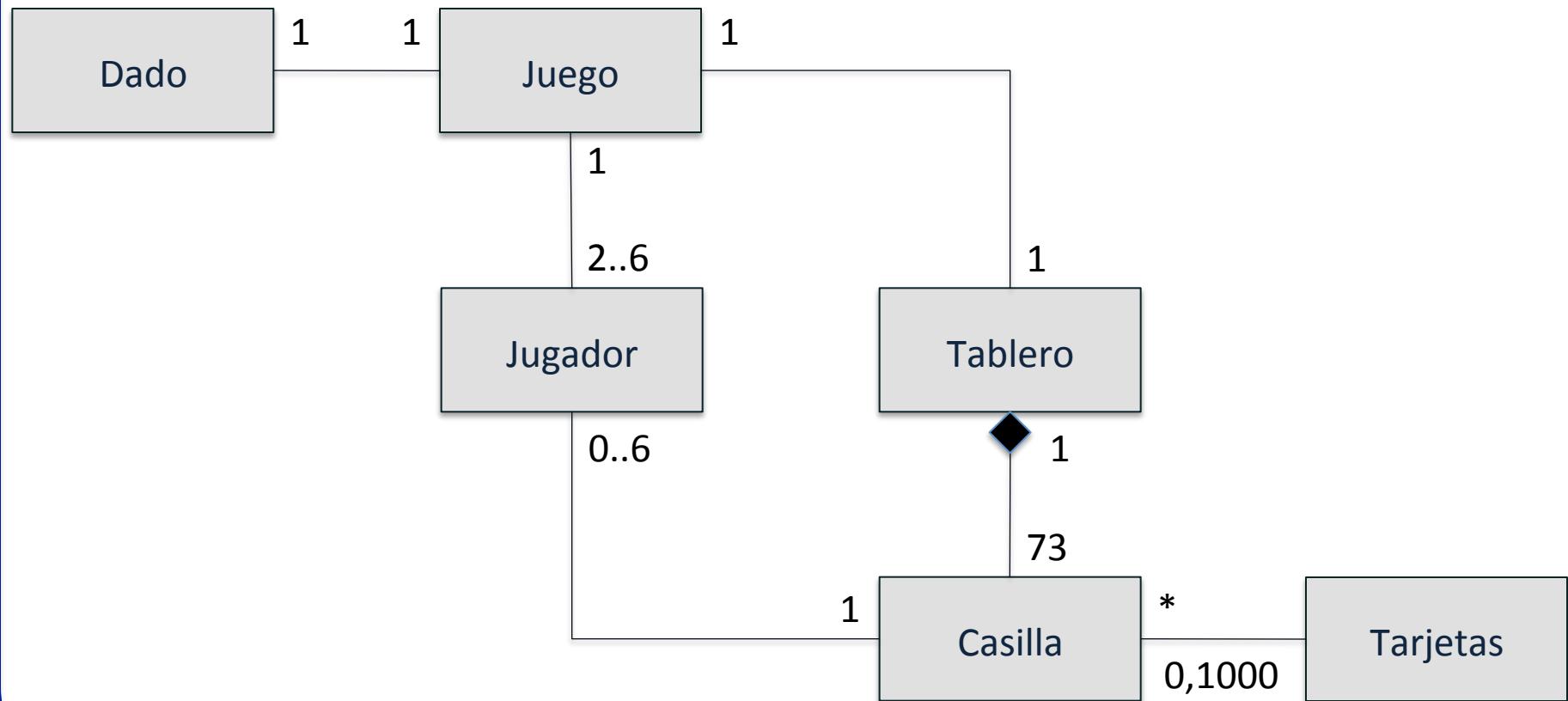
- una Orden contiene muchos objetos de tipo Linea
- *Creador* sugiere que Orden es un buen candidato para tener la responsabilidad de crear instancias de Linea

La asignación de responsabilidades requiere que definamos un método `construirLinea()` en Orden





¿Quién crea a los jugadores? ¿Juego o Casilla?



El principio *Controlador*: Solución y problema

Solución: Asignar la responsabilidad de manejar los mensajes enviados al sistema desde el exterior (p.ej., por el usuario) a una clase que represente uno de los siguientes:

- el sistema, dispositivo en el que corre el software, o subsistema — controlador de fachada
- el caso de uso en el que ocurre el mensaje, típicamente llamado Manejador de <nombredelCasodeUso> —controlador de sesión

Problema: ¿Cuál primer objeto más allá de la gui recibe y coordina los mensajes enviados al sistema desde el exterior?

El principio *Controlador*: Ejemplos

Ejemplos:

- en la aplicación de órdenes de compra, hay varias operaciones del sistema
... ¿quién debe ser el controlador de `ingresarItem()` y `cerrarOrden()`?
- similarmente en el juego, ¿quién debe ser el controlador de `lanzarDado()` y `moverFicha()`?

Un **evento**, o **mensaje de operación**, del sistema es producido por un actor externo, p.ej., el usuario

Estos eventos están asociados con y originan las operaciones (internas) del sistema:

- operaciones del sistema en respuesta a los eventos externos, tal como están asociados mensajes y métodos
- p.ej., cuando un cajero oprime el botón “*Cerrar Orden*”, produce un evento que indica que “*el procesamiento de la orden de compra está terminado*”

Un **controlador** es un objeto, que no es realmente de la GUI, responsable de recibir/manejar eventos:

- el controlador define el método para la operación asociada al evento

¿A quién asignamos la responsabilidad de las operaciones del sistema?

- durante el análisis, pueden ser asignadas a una clase **Sistema**, para indicar que son operaciones del sistema
- esto no significa que una clase del software llamada **Sistema** se haga cargo de ellas en la práctica
- luego, durante el diseño, la responsabilidad de estas operaciones es asignada a una clase controladora

¿Quién debiera ser el controlador para eventos tales como `ingresarItem()` y `cerrarOrden()`?

Según el patrón *Controlador*,

- `Caja`, `SistemadeOrdenes`: representan el sistema o dispositivo global
- `ManejadordeProcesarOrden`, `SesiónProcesarOrden`: representa un receptor o manejador de todos los eventos del sistema para (un escenario de) un caso de uso

La elección depende de otros factores

Durante el diseño, las operaciones del sistema identificadas durante el análisis son asignadas a clases controladoras, tales como `Caja`

Expertos parciales y *animación* de objetos

El cumplimiento de una responsabilidad normalmente necesita información que está dispersa en varias clases:

- varios expertos “parciales” que deben colaborar

Experto lleva a diseños en que un objeto hace **operaciones que en el mundo real se le hacen a la cosa inanimada que representa**:

- en el mundo real, una orden de compra no le dice a uno su total
... alguien calcula el total

La separación de responsabilidades es un principio de diseño básico (*separation of concerns*)

A veces, la solución sugerida por *Experto* no es adecuada —p.ej., produce acoplamiento

P.ej., ¿quién debiera ser responsable de almacenar una :Orden en la base de datos?

- *Experto* dice la clase Orden —tiene los datos de la :Orden
- generalizando, cada clase debiera tener sus propios servicios para almacenar sus instancias en la base de datos —esto crea un problema

Diseñemos separando las responsabilidades principales del sistema:

- mantengamos la lógica de la aplicación en un lugar,
- ... la de la base de datos en otro, etc.

Un creador puede delegar su responsabilidad

Creador intenta encontrar un objeto creador que en cualquier caso necesite ser conectado al objeto creado:

- la clase contenedora o registradora circundante es un buen candidato para la responsabilidad de crear lo contenido o registrado

A veces, la creación es compleja:

- p.ej., hay que reciclar instancias
- p.ej., hay que crear condicionalmente una instancia de una clase de una familia de clases similares
- **conviene delegar la creación a una clase auxiliar —*Fábrica***

Un controlador debe tener la cantidad “justa” de responsabilidad

Un controlador es una entrada a la capa del dominio desde la capa de la GUI:

- no hay que darle muchas responsabilidades
- debe delegar a otros objetos el trabajo que hay que hacer

Si para un caso de uso todos los eventos del sistema son recibidos/delegados por un mismo controlador, éste puede mantener información del estado del caso de uso:

- p.ej., para detectar eventos fuera de secuencia

Cada uno de estos cinco principios tiene sus ventajas

Experto:

- mantiene encapsulación de información
- distribuye el comportamiento del sistema entre varias clases especializadas

Creador:

- apoya el bajo acoplamiento

Bajo Acoplamiento:

- independiza la clase de cambios en otros componentes
- facilita el entendimiento y el reuso de la clase

Alta Cohesión:

- aumenta la claridad del diseño, simplifica el mantenimiento y el mejoramiento
- apoya el bajo acoplamiento y el reuso

Controlador:

- mayor potencial de reuso de la lógica e interfaces intercambiables
- permite razonar acerca del estado del caso de uso