Laboratorio de Ingeniería del Software

El ciclo de vida de los objetos del dominio

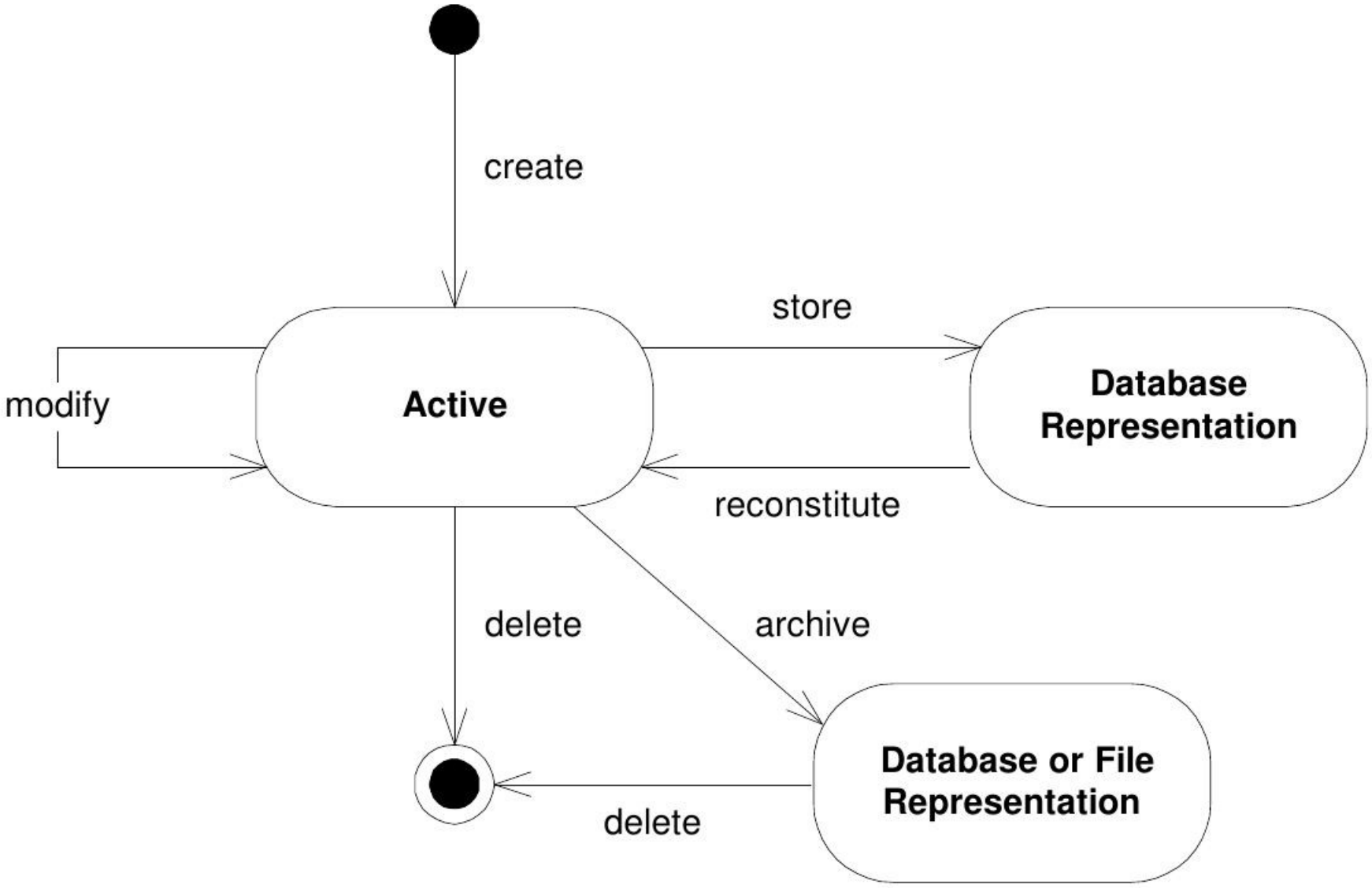
Contenidos

- El ciclo de vida de los objetos
- Agregados
- Factorías
- Repositorios
- Persistencia en BD
- Spring Data JPA

El ciclo de vida de los objetos

El ciclo de vida de los objetos

- Todos los objetos tienen un ciclo de vida
  - Se crean, pasan por diversos estados y en algún momento son borrados de memoria y/o archivados en otro medio
- Hay objetos cuyo ciclo de vida es muy simple, y no hay que complicarlo
  - P.ej.: se instancia a través del constructor, se usa, y se deja para que el *garbage collector* lo reclame
- Pero hay otros que tienen vidas más largas y no pasan todo su tiempo en memoria
  - Gestionar estos ciclos de vida es más complicado



El ciclo de vida de los objetos

- El problema tiene dos partes
  - Mantener la integridad de los objetos a lo largo de su ciclo de vida...
  - ...y evitar que el modelo se complique demasiado mientras intentamos conseguirlo
- Veremos tres patrones para abordar este problema
  - Agregados: para evitar tener marañas indistintas de objetos interrelacionados
  - Factorías: para crear, y reconstituir, objetos complejos y agregados
  - Repositorios: para encontrar y recuperar objetos persistentes

Agregados

Agregados

- Simplificar al máximo las asociaciones entre entidades y objetos valor ayuda a tener modelos de dominio manejables
- Sin embargo, muchos dominios tienen tantas interconexiones entre sus elementos que aún así acabamos teniendo largas cadenas de referencias entre objetos
  - Es un reflejo de la realidad, pero un problema para nosotros
- Por ejemplo, si quiero borrar un objeto Persona de mi sistema, ¿borro el objeto Dirección correspondiente?
  - Si tengo más personas viviendo en esa dirección, igual no debo...
  - El problema es peor en un sistema con actualizaciones concurrentes, donde tengo que tener cuidado si actualizo simultáneamente dos objetos que tienen alguna relación

Agregados

- Es difícil garantizar la consistencia de los cambios realizados sobre objetos relacionados
  - Sobre todo si estas relaciones son complejas
- Hay invariantes que se deben mantener sobre grupos de objetos relacionados, no solo sobre objetos individuales
- Y si queremos un sistema concurrente que funcione bien, querremos evitar bloqueos excesivos
  - P.ej., bloquear todo el sistema mientras hago una actualización evitará estos problemas, pero mis prestaciones serán muy malas

Agregados

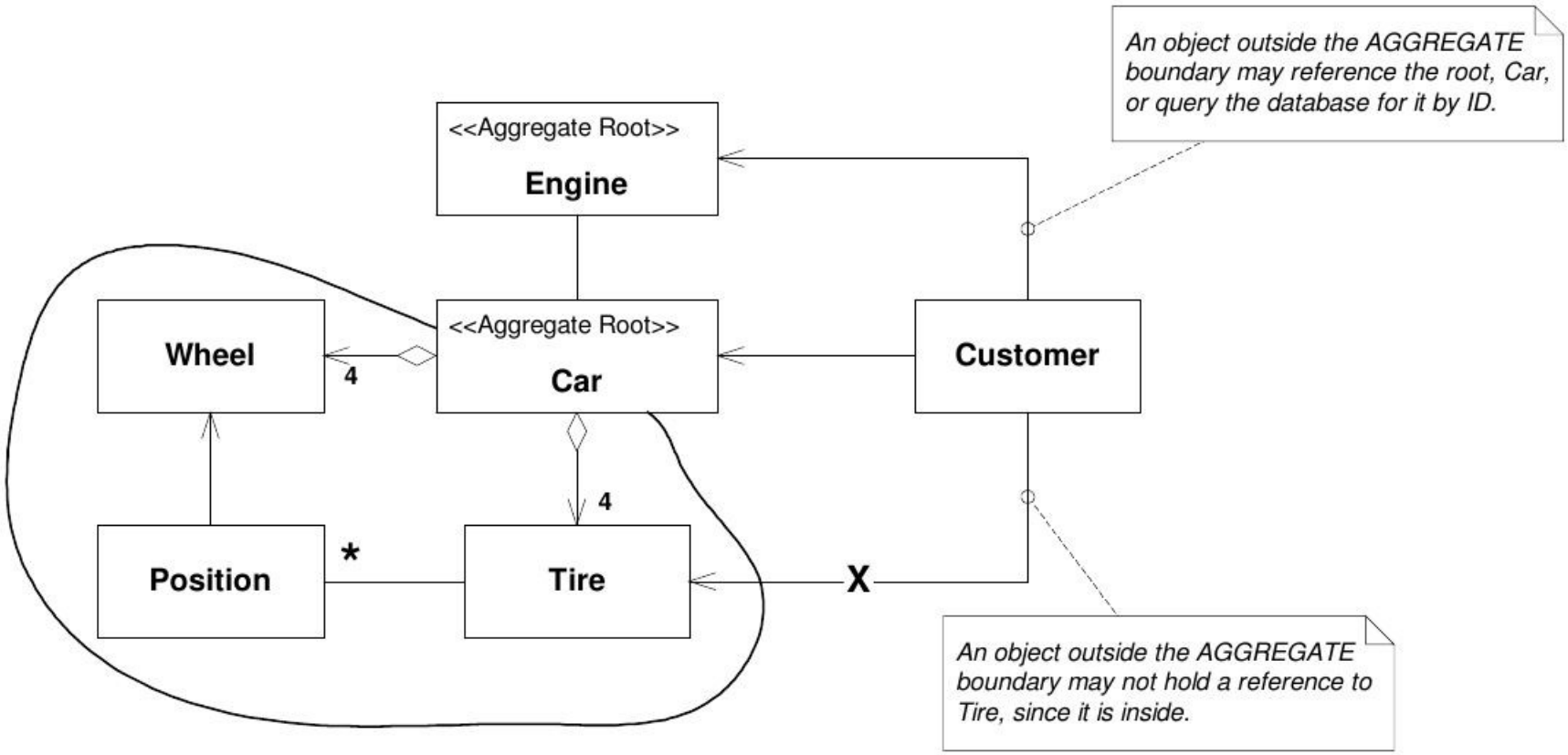
- Una solución habitual ha sido encapsular en consultas de BD (típicamente SQL) parte de, o casi toda, la lógica del dominio
  - Actualizaciones, inserciones y borrados abarcan las tablas necesarias en una única consulta, o en una serie de consultas dentro de una transacción
    - Con o sin apoyo de un ORM, pero en cualquier caso abordado como un problema de datos, no de objetos
  - En parte porque este tipo de problemas son en parte técnicos y propios de la base de datos/ORM que usemos
- Pero el origen de este problema está en el modelo del dominio
  - Queremos una solución determinada por nuestro conocimiento del dominio, y expresada en los objetos del dominio
    - Que sea independiente de la tecnología de persistencia

Agregados

- Un **agregado** es un grupo de objetos asociados que tratamos como **una unidad a la hora de hacer cambios** en los datos
  - Tiene una raíz (es parte del agregado) y una frontera
- La raíz es una entidad
  - Es el único miembro del agregado del que objetos externos pueden mantener referencias, aunque querremos limitar estas referencias todo lo posible
- Los objetos dentro de la frontera pueden mantener referencias unos a otros
- Si el agregado tiene otras entidades, la identidad de estas solo necesita ser única dentro del agregado
  - Porque ningún objeto externo va a ver estas entidades fuera del contexto del agregado
- Consideraremos que una entidad por sí misma es un agregado
  - Y en general la mayor parte de los agregados de nuestro sistema tendrán una sola entidad
    - Normalmente con algunos objetos valor asociados

Ejemplo: aplicación para talleres de coches

- Necesitamos guardar información de cada coche
  - Su identidad es importante => Coche será una entidad
- Necesitamos guardar información de cada neumático
  - Alguna es genérica (p.ej. fabricante y modelo) pero nos interesa tener una historia (por ejemplo, en qué eje lo hemos tenido y cuánto tiempo ha estado ahí) => Neumático será una entidad
  - Sin embargo, una vez lo quitemos del coche (por ejemplo para reciclarlo) esa historia deja de tener interés
  - Incluso mientras esté en el coche, no nos interesa buscar neumáticos concretos en nuestro sistema de información
    - Buscamos coches, y de ahí ya sacamos el historial de sus ruedas
- El coche será la entidad raíz de un agregado cuya frontera incluye sus neumáticos



Agregados

- Los invariantes, reglas de consistencia que se deben mantener bajo cualquier cambio de los datos, involucrarán relaciones entre los miembros de un agregado
  - Como máximo, dejarán de ser ciertas mientras se está completando una transacción, pero volverán a serlo en cuanto esta se complete
- Las reglas que involucren miembros de varios agregados no serán invariantes
  - Normalmente habrá mecanismos para que tarde o temprano se resuelvan posibles inconsistencias y estas reglas se cumplan, pero no habrá garantías de cuándo (*eventual consistency*)
    - Pero solo las que involucren a los miembros de un mismo agregado se podrán hacer cumplir en cada transacción
- En general buscaremos que en cada transacción se modifique una sola instancia de un solo agregado

Recordatorio: Transacciones

- Una transacción es un proceso indivisible que implica cambios en algunos datos y que normalmente tiene estas propiedades (ACID):
- **A**tómica
  - Los cambios en el estado son atómicos: u ocurren todos, o no ocurre ninguno (cambios en la base de datos, mensajes etc.)
- **C**onsistente
  - La transformación del estado es correcta: la transacción no viola ninguna restricción de integridad asociada con el estado que cambia
- a**I**slada
  - Aunque ocurran concurrentemente, lo que cada transacción T puede ver es que cualquier otra transacción T' se ha ejecutado o bien antes o bien después
- **D**uradera
  - Cuando una transacción termina exitosamente (*commit*) sus cambios al estado sobreviven a posibles fallos

Implementar agregados (1)

- La entidad raíz tiene identidad global y es la responsable de chequear los invariantes
- Las entidades dentro de la frontera tienen identidad local
  - Sus identificadores solo tienen que ser únicos dentro de la frontera
- Nada fuera de la frontera puede guardar una referencia a nada de dentro excepto a la entidad raíz
  - La entidad raíz puede pasar referencias a otras entidades del agregado a objetos externos, pero estos solo pueden usarlas de manera transitoria (dentro de una sola operación) y no guardárselas
  - La entidad raíz puede pasar copias de objetos valor del agregado a objetos externos sin peligro

Implementar agregados (2)

- Solo las raíces de los agregados pueden obtenerse haciendo consultas directamente a la base de datos
  - Los demás objetos se obtendrán navegando asociaciones
- Los objetos de un agregado pueden tener referencias a raíces de otros agregados
  - Aunque muchas veces querremos que sean solo indirectas, p.ej. guardando simplemente sus identificadores
- Una operación de borrado eliminará todo lo que haya dentro de la frontera de un agregado de una vez
  - Puesto que nadie externo guarda referencias a nada del agregado excepto a su raíz, si tenemos los objetos en memoria y un *garbage collector* bastará con eliminar la raíz
- Al hacer *commit* de cualquier cambio en cualquier objeto de un agregado, todos los invariantes del agregado tienen que satisfacerse

Factorías

Factorías

- Los objetos que expresan conceptos del dominio no deberían tener ninguna funcionalidad que no soporte su papel en ese dominio
  - Asignar a un objeto complejo la responsabilidad de su creación puede dificultar esto
- El ensamblaje de un objeto complejo es una tarea compleja que no tiene que ver con su funcionalidad
  - Podemos (debemos) dar esta responsabilidad a otros
  - Pero no podemos dársela a los objetos que usarán a este, porque tendrían que conocer cómo es por dentro y eso violaría su encapsulación

Factorías

- La creación de objetos complejos es una responsabilidad de la capa del dominio, pero no pertenece a los objetos que expresan el modelo de dominio
  - No tiene sentido en el dominio, pero es necesario para la implementación
- Tenemos que añadir elementos al diseño del dominio que no son entidades, objetos valor ni servicios
  - **No corresponden con el modelo del dominio, pero aún así son parte de las responsabilidades de la capa del dominio**
- Una factoría es un elemento del programa cuya responsabilidad es la creación de objetos

Factorías

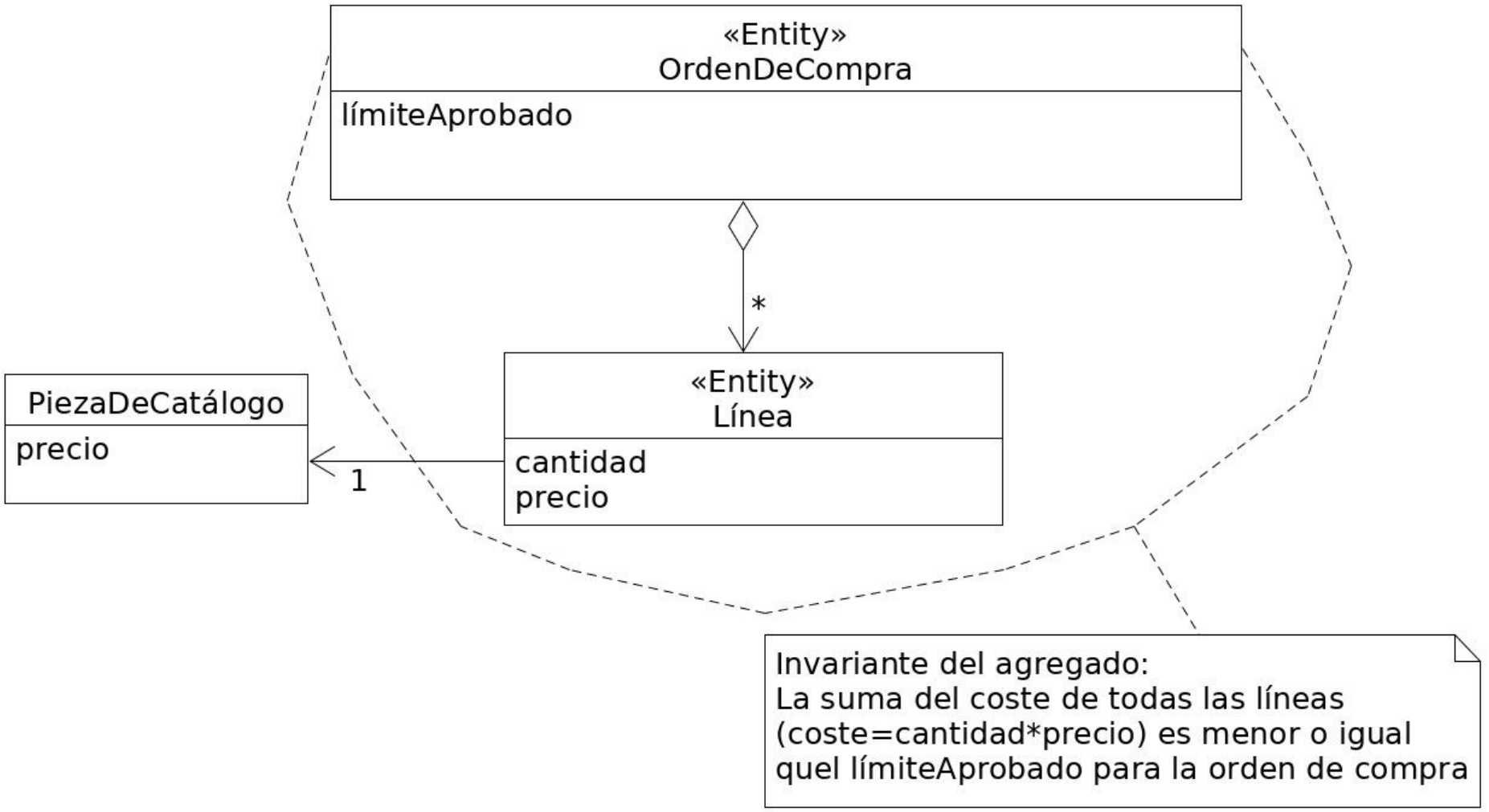
- La interfaz de un objeto debería encapsular su implementación, permitiendo a otro objeto usarlo sin saber cómo funciona por dentro
- De la misma forma una factoría encapsula el conocimiento necesario para crear un objeto complejo (o un agregado)
- Daremos la responsabilidad de crear instancias de objetos complejos y agregados a un objeto separado que
  - Puede no tener una responsabilidad en el modelo del dominio pero aún así es parte de la capa del dominio
  - Proporciona una interfaz que encapsula las complejidades del montaje del objeto o agregado
  - Crea agregados enteros de una pieza, asegurándose de que se cumplen sus invariantes

Requisitos básicos para las factorías

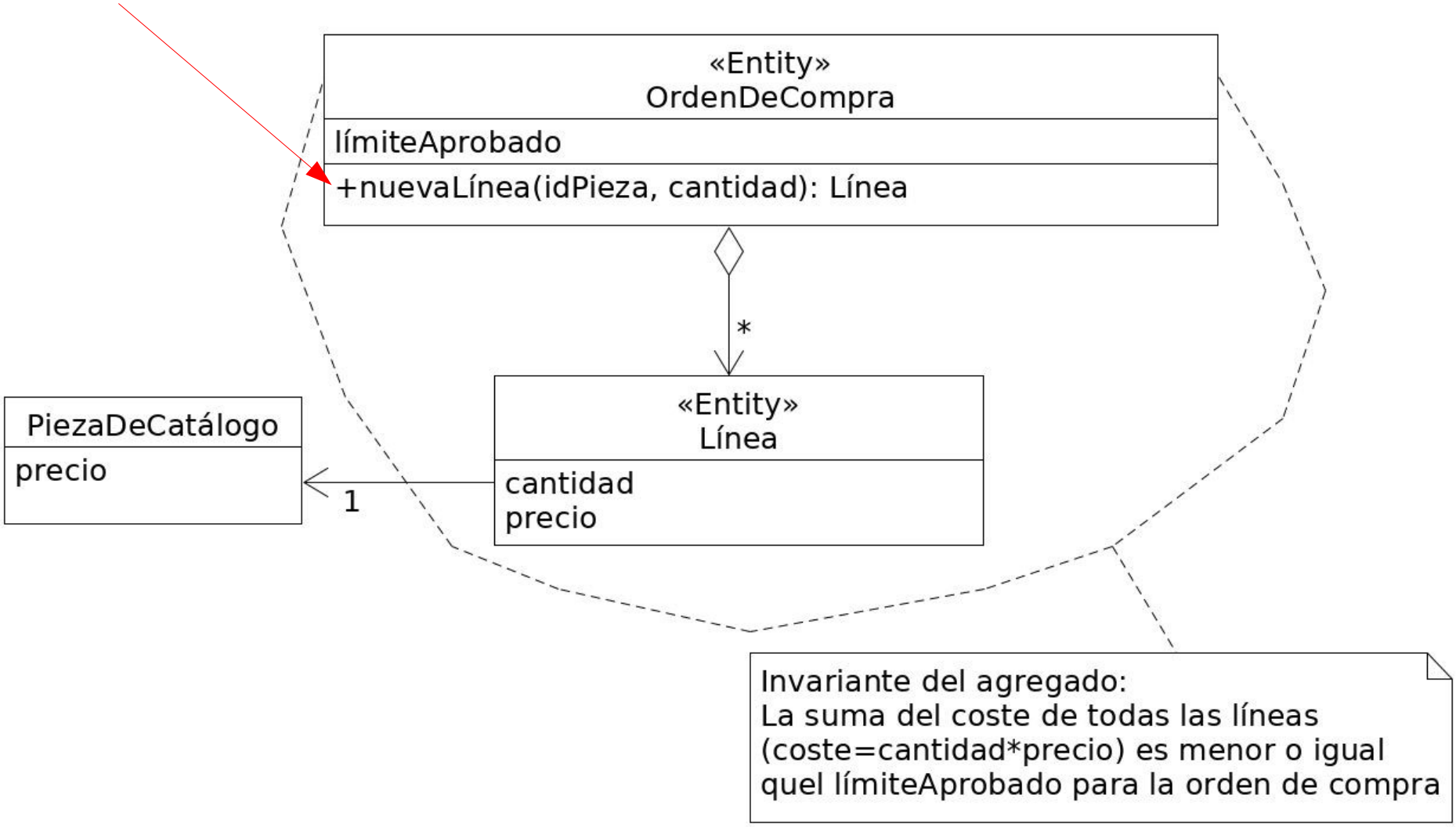
- Cada método de creación es atómico y hace cumplir todos los invariantes del objeto o agregado que crea
  - Solo puede producir objetos o agregados en estados consistentes
  - Sí que puede dejar algunos elementos opcionales para que se añadan más adelante
- La factoría debería declarar que devuelve objetos de tipo abstracto
  - Tendrá que devolver objetos de una clase concreta, que son los que se pueden instanciar, pero el objeto que llama a las operaciones de la factoría no debería necesitar conocer qué clase concreta será devuelta
  - Los patrones de diseño *factory method, abstract factory* y *builder* ayudan con este requisito

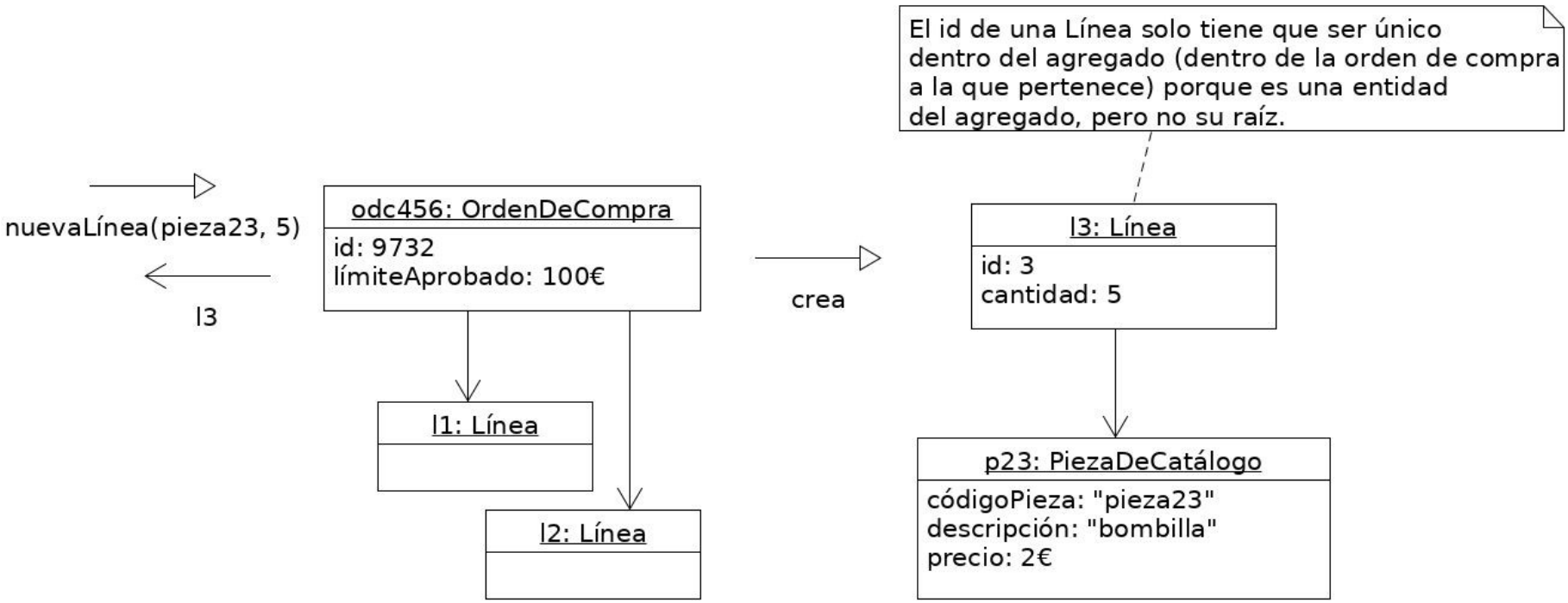
Elegir factorías y su sitio

- Las factorías se crean para construir algo cuyos detalles se quieren ocultar
  - Y en el sitio donde se quiere dar el control de esto
- Para añadir elementos a un agregado preexistente, puedes poner un *factory method* en la raíz del agregado
  - La implementación del resto del agregado queda oculta
  - La raíz tiene la responsabilidad de asegurarse de la integridad del agregado conforme se añaden elementos


«Entity»
OrdenDeCompra
límiteAprobado
*
«Entity»
Línea
cantidad
precio
PiezaDeCatálogo
precio
1
Invariante del agregado:
La suma del coste de todas las líneas
(coste=cantidad*precio) es menor o igual
quel límiteAprobado para la orden de compra

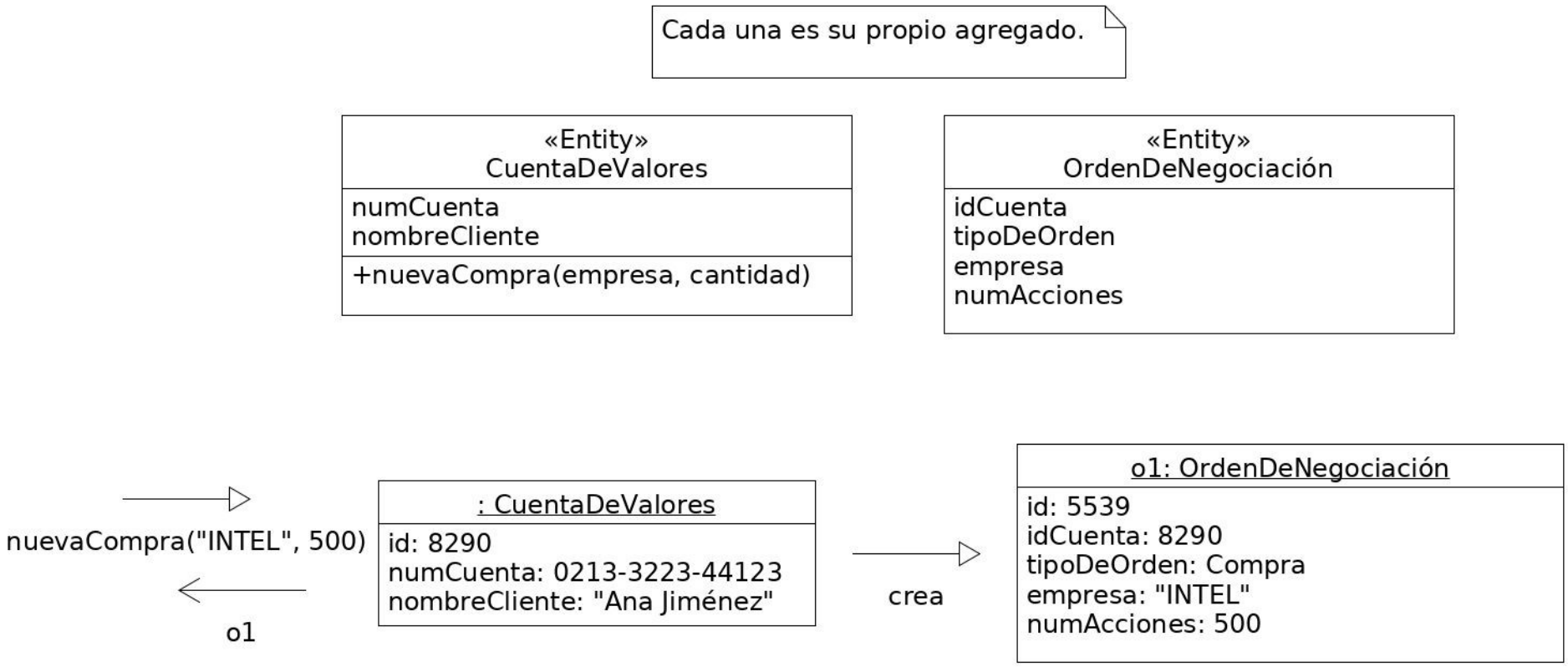
Factory method en la raíz del agregado




El id de una Línea solo tiene que ser único dentro del agregado (dentro de la orden de compra a la que pertenece) porque es una entidad del agregado, pero no su raíz.
nuevaLínea(pieza23, 5)
l3
odc456: OrdenDeCompra
id: 9732
límiteAprobado: 100€
l1: Línea
l2: Línea
crea
l3: Línea
id: 3
cantidad: 5
p23: PiezaDeCatálogo
códigoPieza: "pieza23"
descripción: "bombilla"
precio: 2€

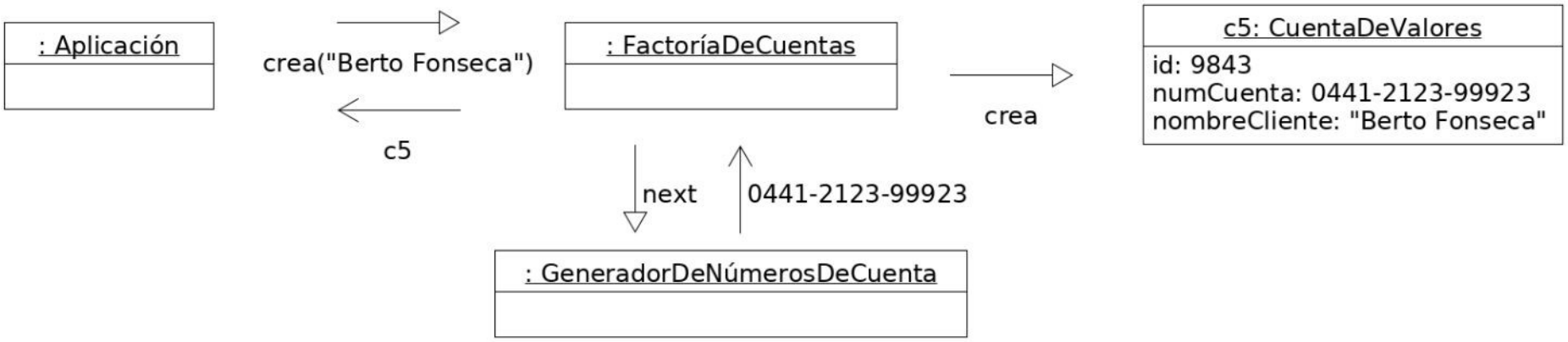
Elegir factorías y su sitio

- A veces nos interesa poner un *factory method* en un objeto A que es importante en la creación de otro objeto B, aunque ni siquiera pertenezcan al mismo agregado
  - Si ese objeto A es importante para la creación de B, si fuéramos a crear B en otro sitio (p.ej. una factoría específica), ese sitio tendría que consultar a A
    - Para eso, directamente podemos poner el *factory method* en A
  - También es una forma de hacer más clara en el modelo la estrecha relación entre estos dos objetos


Cada una es su propio agregado.
«Entity»
CuentaDeValores
numCuenta
nombreCliente
+nuevaCompra(empresa, cantidad)
«Entity»
OrdenDeNegociación
idCuenta
tipoDeOrden
empresa
numAcciones
nuevaCompra("INTEL", 500)
o1
: CuentaDeValores
id: 8290
numCuenta: 0213-3223-44123
nombreCliente: "Ana Jiménez"
crea
o1: OrdenDeNegociación
id: 5539
idCuenta: 8290
tipoDeOrden: Compra
empresa: "INTEL"
numAcciones: 500

Elegir factorías y su sitio

- A veces no hay un objeto natural donde poner una factoría, pero aún así hay algo cuya construcción es lo bastante compleja como para necesitar una
- La solución es crear un objeto (o servicio) cuyo único papel sea hacer de factoría
- Normalmente usamos estas factorías dedicadas para crear raíces de algún agregado
  - La factoría se asegura de que se cumplen los invariantes del agregado
  - Si un objeto que pertenece a un agregado necesita una factoría, y la raíz del agregado no nos parece un buen sitio, también se puede construir en una factoría dedicada
    - Con un ojo en las reglas de lo que se puede hacer y no con un objeto interno de un agregado


: Aplicación
crea("Berto Fonseca")
c5
: FactoríaDeCuentas
crea
c5: CuentaDeValores
id: 9843
numCuenta: 0441-2123-99923
nombreCliente: "Berto Fonseca"
next
0441-2123-99923
: GeneradorDeNúmerosDeCuenta

¿Cuándo es suficiente con el constructor?

- Para objetos cuya clase no pertenece a una jerarquía de generalización-especialización
- Cuando no queremos ocultar los detalles de implementación del objeto a aquellos que lo quieran instanciar
- Cuando el que quiere instanciar el objeto ya conoce todos los atributos que tiene que tener ese objeto
- Cuando la construcción no es complicada
- Un constructor público tiene que seguir las mismas reglas que una factoría
  - Ser atómico y asegurar que se satisfacen todos los invariantes del objeto creado
- Ojo con llamar a unos constructores dentro de otros salvo que sean muy simples
  - Si resulta que tienes unos cuantos objetos relacionados que se construyen juntos, quizás necesitas una factoría

La interfaz de una factoría

- Todas sus operaciones deben ser atómicas
  - Todo lo necesario para instanciar el objeto se le tiene que pasar a la factoría de una sola vez
  - Hay que decidir qué se hace cuando falla la construcción
    - Devolver un objeto nulo, lanzar una excepción...
- La factoría estará muy acoplada a los parámetros de sus operaciones y esto puede crear muchas dependencias
  - Si simplemente los coge y se los pasa sin leerlos al objeto que está construyendo, es una dependencia leve
  - Si se cogen partes de los parámetros o se usan algunas de sus operaciones, la dependencia es más fuerte
  - Suele ser buena idea que los parámetros sean clases abstractas y no concretas cuando esto sea posible

¿Dónde va la lógica de los invariantes?

- Una factoría es responsable de asegurar que se cumplen todos los invariantes del objeto o agregado que crea
  - Pero suele ser bueno que las reglas que se aplican a un objeto estén dentro del objeto
- **Normalmente, lo mejor es que la factoría delegue el chequeo de invariantes en el producto que esté construyendo**
  - La factoría sigue responsabilizándose de que se haga este chequeo, pero el chequeo lo hace el producto
- Pero dada la estrecha relación de una factoría con los productos que crea, a veces esta lógica de chequeo de invariantes puede ir en la factoría
  - Muy interesante con reglas de agregados (porque abarcan varios objetos) y bastante problemático con *factory methods* que están en otros objetos del dominio
- Cuando algo no puede cambiar después de ser construido, puede no tener sentido que el objeto correspondiente lleve consigo, durante toda su vida, invariantes que solo se comprueban durante su creación
  - Este tipo de reglas también podrían ir a una factoría

Factorías de entidades y de objetos valor

- Los objetos valor son inmutables, así que una vez construidos tenemos el objeto completo
  - Las **factorías de los objetos valor** tienen que permitir que les pasemos **una descripción completa** del producto a construir
- Las **factorías de entidades y agregados** tienden a requerir **solo los atributos esenciales** para crear algo válido
  - Los detalles se suelen añadir después
- Las entidades tienen identidad (y por tanto identificadores)
  - Si el identificador es externo, la operación de construcción en la factoría debe permitir que se lo pasemos
  - Si el identificador es asignado automáticamente, la factoría es un buen sitio para controlar esto
    - Quizás pidiéndoselo a un objeto de tipo "generador de secuencias" o algo parecido de la capa de infraestructura

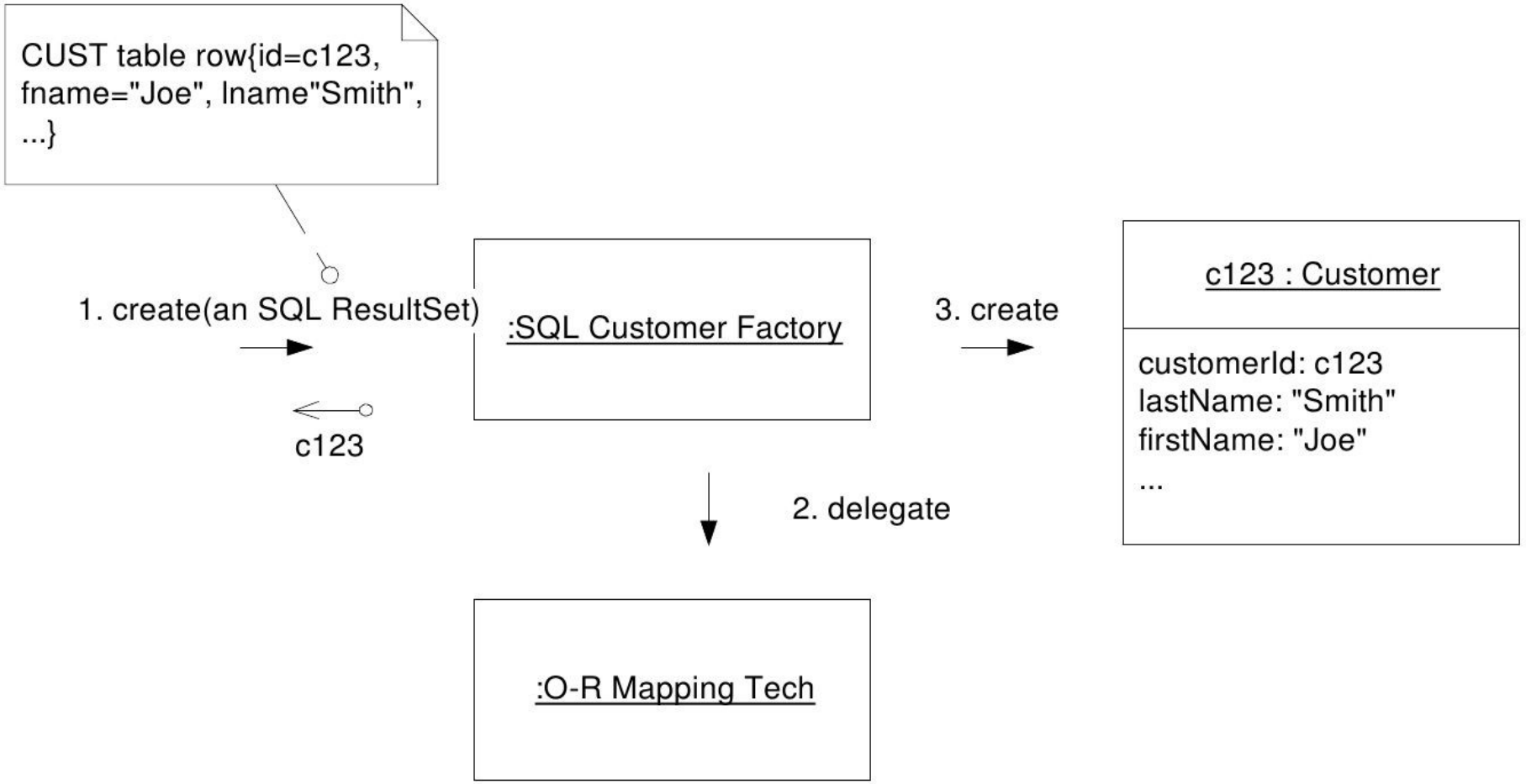
Reconstitución de objetos almacenados

- Las factorías juegan un papel esencial en el comienzo del ciclo de vida de algunos objetos
  - Los crean
- En algún momento, la mayoría de los objetos son almacenados en disco/base de datos o enviados a través de una red
  - Normalmente perdiendo el carácter de objeto
- Cuando se recuperan, hay que volver a "ensamblar las piezas" en un objeto vivo
- Llamaremos reconstitución a la acción de crear una instancia de un objeto a partir de datos almacenados (en BD relacional, en XML...)
  - No es un término universalmente aceptado para referirse a esto, es un forma de recordarnos que es algo que se produce en mitad del ciclo de vida de los objetos

Factorías de reconstitución

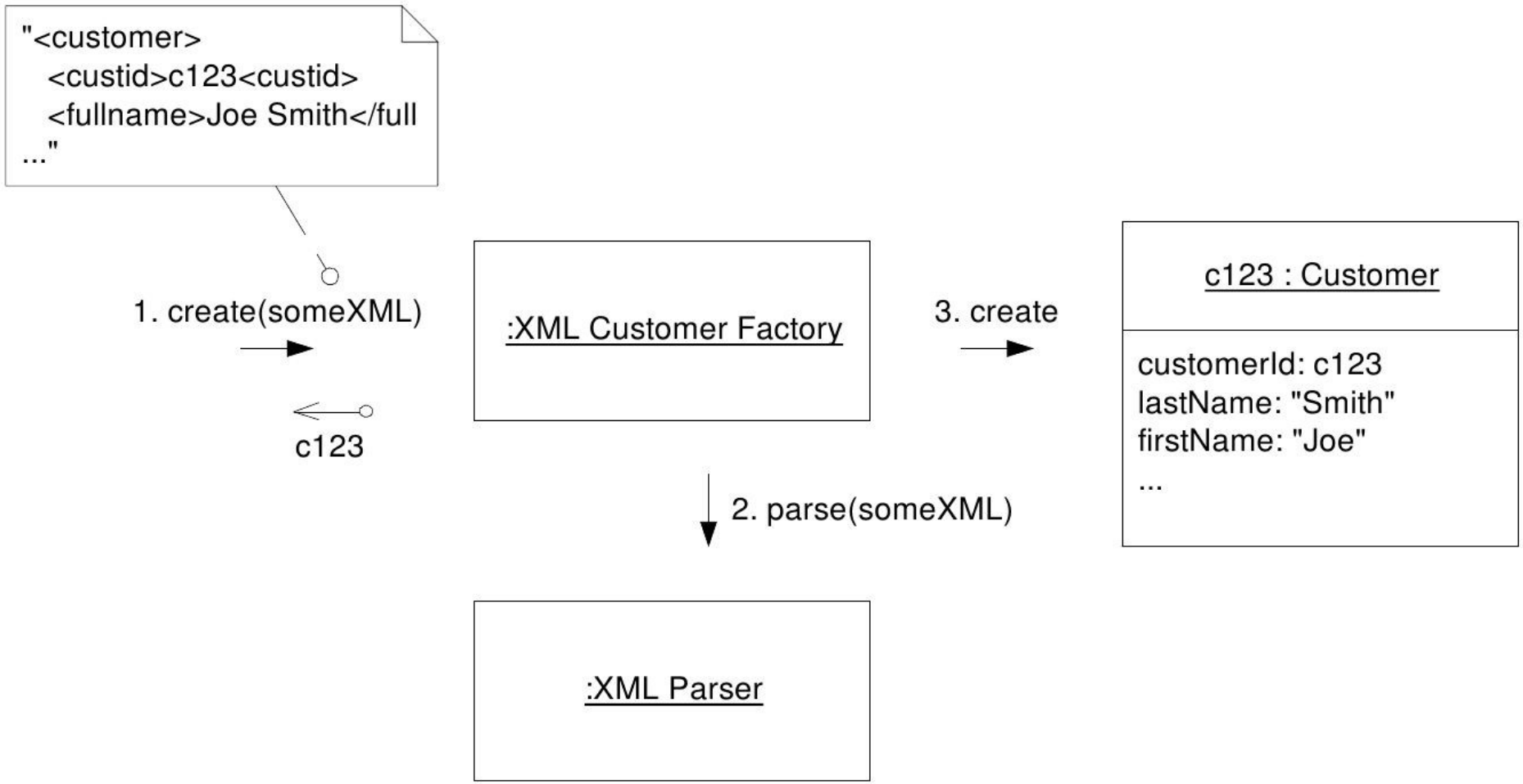
- Una factoría de reconstitución es como una de creación excepto que
  - Una factoría de reconstitución de entidades no asigna un nuevo id a estas
    - El id de la entidad a reconstituir debe ser siempre un parámetro de entrada
  - Una factoría de reconstitución de objetos puede gestionar la violación de un invariante de forma diferente
    - Lo que en la creación de un objeto es un error, en la reconstitución puede que haya que tolerarlo porque ese objeto "problemático" estaba en la BD o ha llegado por red
      - Ni podemos ignorarlo, ni tampoco ignorar que se viola un invariante
      - Habrá que pensar una estrategia para abordar este tipo de inconsistencias (dependerá del dominio)
- Un *framework* de mapeo entre objetos y BD relacionales puede proporcionar este tipo de servicios de forma bastante transparente

Reconstituir una entidad sacada de una base de datos relacional



Eric Evans
Domain-Driven Design...
Figure 6.10

Reconstituir una entidad a partir de XML



Eric Evans
Domain-Driven Design...
Figure 6.11

Repositorios

Referencias a los objetos

- Para usar un objeto necesitas una referencia al mismo. ¿Cómo se consiguen estas referencias?
  - Creando objetos
    - La operación de creación devuelve una referencia al objeto recién creado
  - Recorriendo asociaciones
    - A partir de un objeto que conoces vas pidiendo los objetos asociados
  - Ejecutando una consulta, a partir de alguno de sus atributos, en una base de datos y reconstituyendo el objeto
    - O en un servicio web que permita hacer búsquedas o algo similar
- La tercera estrategia (consulta, típicamente a una BD) permite obtener una referencia a cualquier objeto almacenado

Reconstitución de objetos de dominio

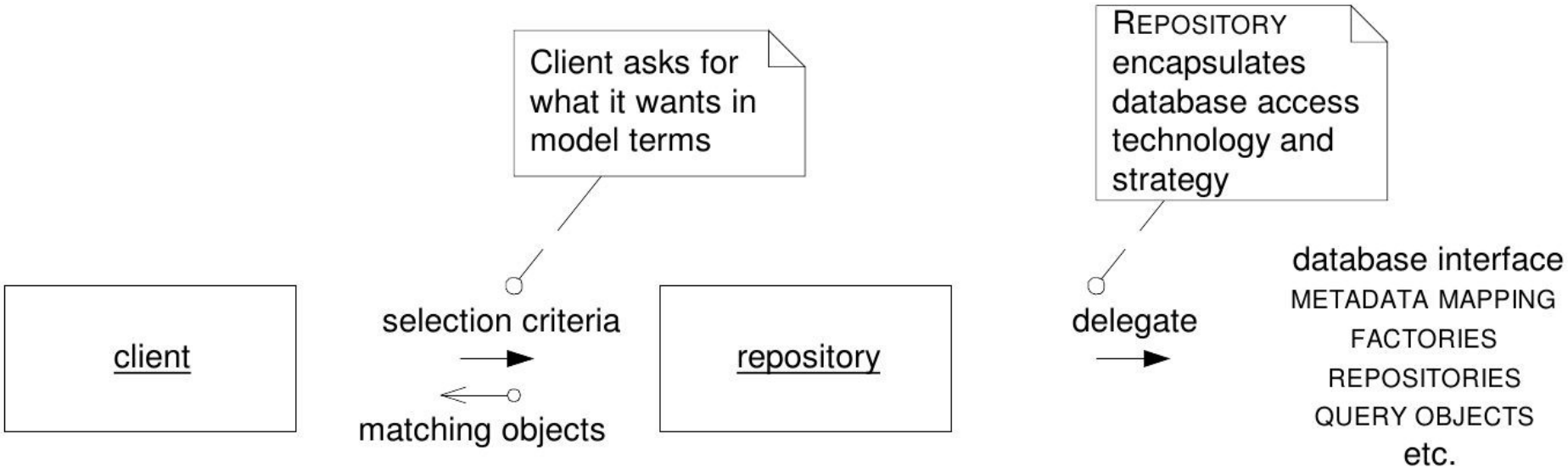
- Hay patrones para tratar con los desafíos del acceso a BD
  - Encapsular consultas SQL en *Query Objects*, traducir entre objetos y tablas con *Metadata Mapping*, usar Factorías para reconstituir objetos almacenados etc.
  - Son el tipo de patrones que implementa un ORM como p.ej. Hibernate
- Pero todos son patrones técnicos, independientes del dominio de problema y que se pueden usar tanto en modelos de dominio ricos como en modelos anémicos
  - Un modelo anémico se caracteriza por una capa de objetos muy simples entre la aplicación y la BD, que esencialmente se limita a mapear servicios de aplicación con consultas de esta BD, con muy poca lógica adicional en los objetos
    - Patrones típicos de un modelo anémico son los Data Access Objects (DAO), también llamados Table Data Gateways, asociados a Data Transfer Objects (DTO)
  - Para el DDD, necesitamos algo que nos de **un nivel de abstracción más alto**

Repositorios

- Un **repositorio** representa al conjunto de todos los objetos de un cierto tipo
- Actúa como una colección con capacidades de consulta/búsqueda complejas
- Los objetos del tipo adecuado se añaden y quitan del repositorio, y este se encarga de insertarlos en, o borrarlos de, la base de datos
  - Desacoplan el diseño de la aplicación de la tecnología de persistencia
- Proporciona acceso a las raíces de los agregados durante casi todo el ciclo de vida de los mismos
  - Son una forma sencilla de obtener objetos persistentes y gestionar su ciclo de vida

Repositorios

- Pedimos objetos al repositorio usando sus métodos de consulta, que seleccionan objetos basándose en los criterios que especifiquemos
  - Normalmente esos criterios son el valor de ciertos atributos
- El repositorio obtiene el objeto solicitado, encapsulando la tecnología de persistencia
  - Consultas SQL, mapeador objeto-relacional, driver de BD NoSQL...
- Un repositorio puede soportar distintos tipos de consultas y también devolver cierta información resumida
  - P.ej. cuántos objetos en el repositorio cumplen ciertos criterios
  - Incluso devolver algunos cálculos de resumen
    - Por ejemplo, la suma total de cierto atributo numérico en los objetos que encajan en cierta consulta
- Los repositorios son un punto de acceso a una interfaz sencilla que permite pedir los objetos que se necesitan en términos del modelo del dominio
  - El repositorio pude tener cierta complejidad, pero su interfaz es simple y se entiende en términos de los conceptos del dominio



Repositorios

- En general, crea un repositorio para cada tipo de entidad de tu dominio que sea raíz de un agregado
  - A otras entidades y objetos valor llegaremos siempre navegando asociaciones desde las entidades que hemos sacado de los repositorios
    - Incluso es posible que para algunas entidades raíz de agregados descubras que no te hace falta que tengan repositorio propio
- Cada repositorio debe ser accesible para todo el mundo (respetando las capas)
  - Distintos servicios de la capa de aplicación, y quizás algunos objetos y servicios del dominio, necesitarán acceder a los repositorios
- Proporciona métodos para añadir y quitar objetos a los repositorios, que encapsularán la inserción y borrado en la BD
- Proporciona métodos para seleccionar objetos en base a ciertos criterios y devuelve objetos plenamente instanciados, o colecciones de objetos que cumplen con esos criterios (encapsulando así parte de la tecnología de consulta y almacenamiento)
- Toda la tecnología de almacenamiento debe quedar encapsulada en los repositorios

Consultar a un repositorio

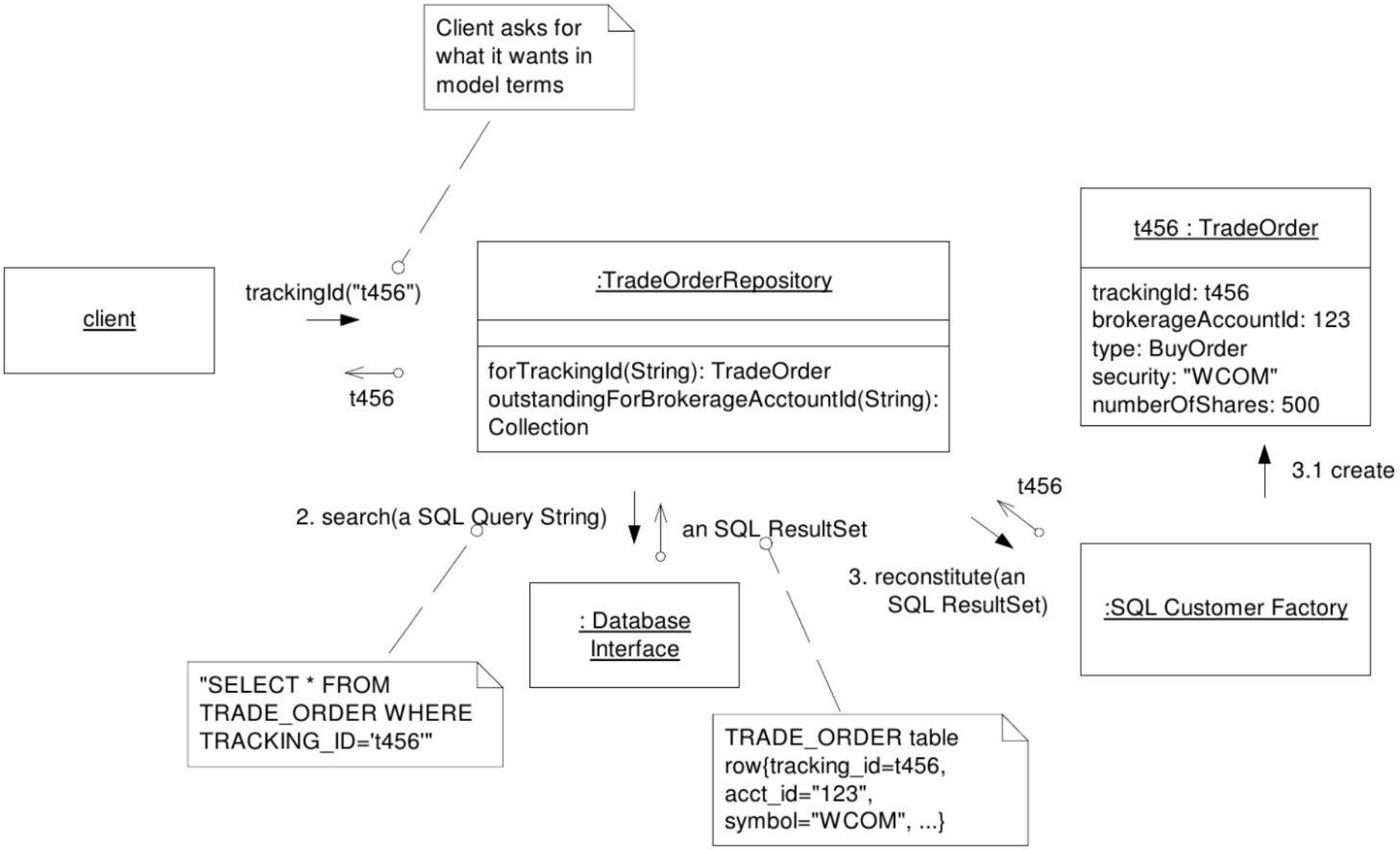
- Los repositorios más simples ofrecen en su interfaz consultas prefijadas con parámetros variables
  - Obtener una entidad pasando un id
  - Obtener una colección de objetos que tienen cierto valor para uno de sus atributos, o para alguna combinación de atributos, o algún rango de valores...
  - También pueden devolver algunos cálculos resumen
- Este tipo de consultas son fáciles de implementar con cualquier tecnología que haya debajo
- En proyectos más complejos y con recursos suficientes, los repositorios pueden ofrecer capacidades de consulta más flexibles
  - Una aproximación que funciona bien en diseño dirigido por el dominio son las consultas basadas en especificaciones

Prestaciones de los Repositorios

- El código que usa las operaciones de un repositorio debe ser independiente de la tecnología de persistencia en la que éste se basa
- Quien escribe ese código sí que tiene que tener cuidado con las implicaciones de lo que está pasando
  - La diferencia de prestaciones entre acceder a memoria o a disco es extrema
  - En los tests puede que no aparezcan estos problemas si no se hacen pruebas con conjuntos de datos de tamaño realista

Implementación de un repositorio

- La implementación es muy dependiente de la tecnología de persistencia que se use, y la infraestructura que se tenga
- Encapsular los mecanismos de almacenamiento, recuperación y consulta es lo mínimo que un repositorio tiene que implementar
- Normalmente se establece un *framework* en la capa de infraestructura para soportar la implementación de repositorios
  - Superclase repositorio, relación con la infraestructura de persistencia y quizás algunas consultas básicas
- Las posibilidades son muy variadas, veremos un ejemplo concreto con Spring Data el final de este tema



Implementación de un repositorio: consideraciones

- Un repositorio da acceso a todas las instancias de un tipo, pero no hace falta tener un repositorio para cada clase
  - El tipo que contiene el repositorio debería ser la clase abstracta o interfaz que está arriba del todo en la jerarquía de clases
  - Puede ser una clase concreta si no hay ninguna jerarquía significativa
- Se puede aprovechar que el repositorio desacopla el código cliente de la tecnología de almacenamiento
  - La implementación del repositorio puede cambiar con más libertad
  - Para optimización de prestaciones, diferentes estrategias de consulta, caché de objetos, cambio de estrategia de persistencia en cualquier momento...
  - Para facilitar las pruebas con un repositorio *dummy* que almacene todo en memoria

Repositorios y transacciones

- Los repositorios pertenecen a la capa del dominio, pero las transacciones de BD no
  - Por una parte, tienen una parte que es técnica (p.ej. la durabilidad)
  - Por otra parte las operaciones de los objetos del dominio normalmente son de grano demasiado fino como para gestionar transacciones o saber el papel que las transacciones toman en su vida
- La aproximación típica es controlarlas en la capa de aplicación
  - El repositorio inserta y borra de la base de datos, pero que esas operaciones vayan o no en una transacción, y donde empieza y termina esta, es algo que debe determinar el código que lo usa
    - El repositorio, como parte de la tarea de encapsular la tecnología de persistencia, sí que puede ofrecer operaciones para controlar las transacciones

Ejercicio

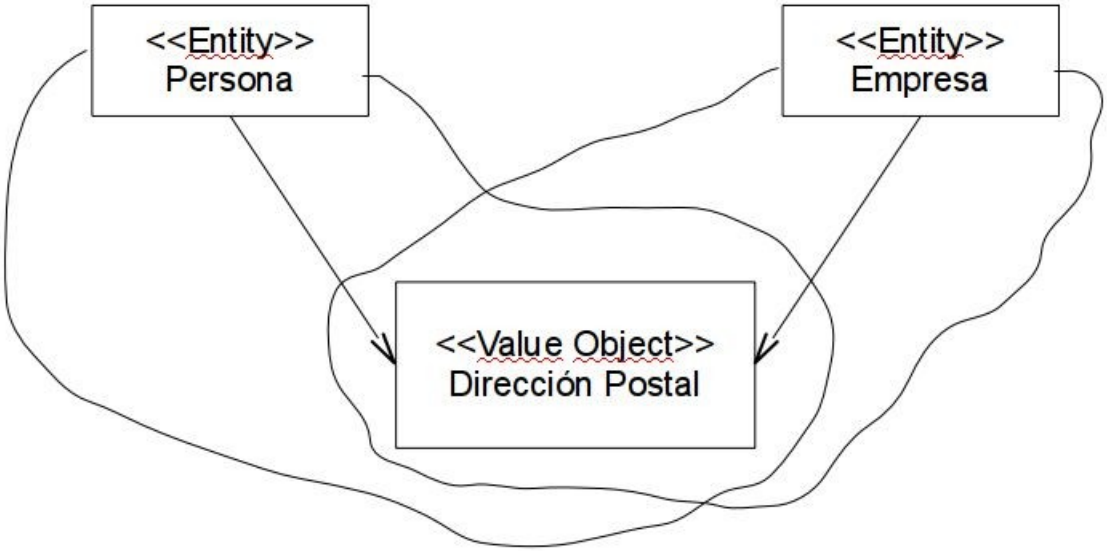
- Tenemos un servicio de dominio que calcula la compatibilidad entre dos entidades `PersonaSolitaria` para un sitio de citas online

```
public class CalculadorCompatibilidad {
  // No tiene atributos, no guarda estado, es un Servicio de Dominio
  public TasaDeCompatibilidad CalculaCompatibilidad(PersonaSolitaria persona1,
PersonaSolitaria persona2) {
    // saca datos de persona 1 y persona 2, compara genera una tasa de compatibilidad y la
devuelve
  }
}
```

- Hay una historia de usuario en el sistema que dice: "Como persona solitaria, quiero poder ver todas las otras personas solitarias de mi ciudad ordenadas por mi compatibilidad con ellas para poder iniciar un chat con las más compatibles"
- Asume que la entidad `PersonaSolitaria` tiene la información que necesites (por ejemplo, cada `PersonaSolitaria` tiene una ciudad de residencia)
- Diseña cómo sería la interacción entre los objetos que proporcionarían la funcionalidad de esta historia. Asume que la GUI ya está diseñada, y que el mecanismo de persistencia de la capa de infraestructura es el que prefieras. Tienes que considerar al menos:
  - Un servicio de aplicación (servicio de la capa de aplicación) para esta historia de usuario
  - El repositorio o repositorios que puedas necesitar
  - Un diagrama de secuencia que "conecte las piezas"

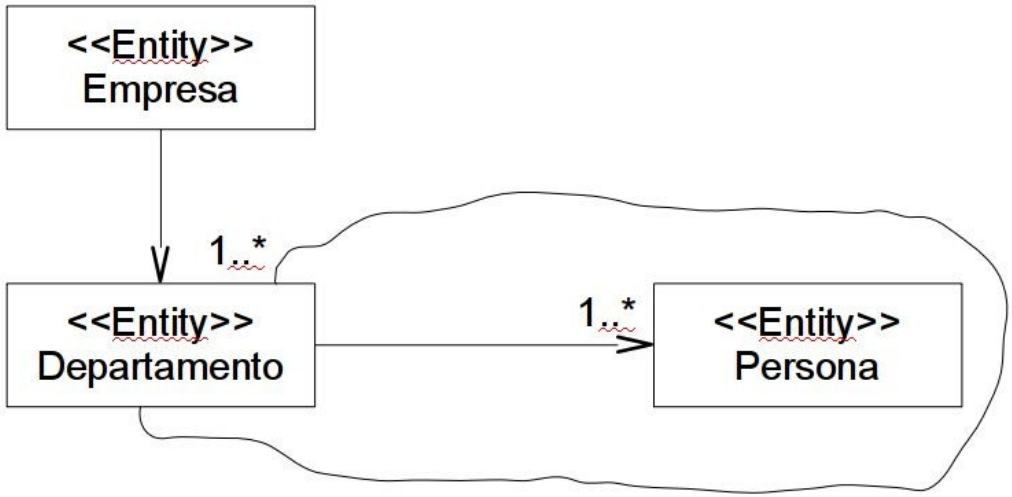
Pregunta

- Tenemos dos agregados, uno Persona y otro Empresa
- Regla de agregados: "nada fuera de la frontera puede guardar una referencia a nada de dentro excepto a la entidad raíz"
  - Pero Empresa guarda una referencia a Dirección Postal que está dentro del agregado de Persona (y lo mismo podemos decir de Persona)
- ¿Qué está pasando aquí?



Pregunta

- Tenemos en nuestro dominio estos dos agregados, Empresa y Departamento
- Tenemos un repositorio por cada agregado (lo normal)
- Desde RRHH nos piden una aplicación para poder generar informes con todas las personas de la empresa, o con aquellas personas que cumplan ciertos criterios que especifiquen desde la aplicación, indicando para cada una a qué departamento pertenecen
- ¿Cómo la hacemos?



Pregunta

- ¿Qué opción de las 3 preferimos para acceder al Cliente que realizó un determinado Pedido?

```
public class Pedido extends BaseEntity {
  // ...
  private Cliente cliente;
  public Cliente getCliente() {
    return cliente;
  }}

public class Pedido extends BaseEntity {
  // ...
  private ID clienteId;
  public ID getClienteId() {
    return clienteId;
  }}

public class Pedido extends BaseEntity {
  // ...
  private ID clienteId;
  public Cliente getCliente() {
    return repositorioClientes.findCliente(clienteId);
  }}
```

Persistencia en BD

Tecnología de persistencia

- Tenemos al menos dos opciones generales para la base de datos de nuestro sistema
  - Relacionales (SQL) o NoSQL
    - Dentro de las NoSQL hay muchas variantes (documentos, clave-valor, columnas...)
- Con cualquier base de datos que elijamos, un framework como Spring Data o Hibernate puede hacer más fácil gestionar la persistencia de nuestros objetos
- Si hemos decidido usar Diseño Dirigido por el Dominio, deberíamos elegir tipo de BD y framework con eso en mente
  - Y cuando no podamos elegir BD y/o framework, debemos tener claro donde puede haber problemas, y como minimizarlos

Base de datos

- Para DDD las bases de datos NoSQL de documentos (como MongoDB) funcionan muy bien
  - Estas BD almacenan documentos, p.ej. en BSON (JSON binario), dentro de colecciones
- Un documento de p.ej. MongoDB es algo muy cercano a un agregado de DDD
- Si tienes un agregado con una entidad raíz, asociada a varios objetos valor y quizás a alguna entidad interna, lo más natural es almacenar todo eso junto en un mismo documento
  - Los enlaces entre la entidad raíz de un agregado y las entidades raíz de otros agregados los implementas guardando sus identificadores (estilo clave ajena)
- Los documentos de MongoDB se crean, leen, modifican y borran en una única transacción
  - Esto es lo que hacemos normalmente con los agregados, que por definición son una unidad desde el punto de vista de los cambios en los datos

Base de datos

- En muchas ocasiones tienes que usar una BD SQL
  - Estás modificando un sistema existente, la BD se va a usar desde más de un sistema, hay funcionalidad en la BD SQL que no está disponible en otras NoSQL...
- El mapeo entre objetos y bases de datos relacionales es complejo (*impedance mismatch*)
  - Pero es un problema común y hay herramientas razonablemente refinadas para facilitarlo
- Hay tres casos habituales
  - La BD SQL se ha diseñado para este sistema, pero como simple almacén para los objetos
  - La BD SQL se diseñó para otro sistema
  - La BD SQL se ha diseñado para este sistema, pero tiene un papel mayor que el de simple almacén de objetos

BD SQL como simple almacén

- Si estamos creando una nueva BD SQL para nuestro sistema y solo la queremos como almacén de objetos se puede sacrificar algo de expresividad en el modelo SQL para que el mapeo sea simple
  - Que solo sea un almacén significa que solo queremos poder guardar y recuperar los objetos, pero que no necesitamos consultas que requieran cruzar tablas etc.
  - Es decir, que podemos tener un esquema SQL muy "tonto", incluso no muy normalizado, porque la "inteligencia" va a estar en nuestros objetos y nos interesa que la BD SQL sea muy simple
- Ningún proceso fuera de nuestro sistema debería acceder (salvo quizás solo en lectura) a una BD SQL configurada así
  - Se podrían violar invariantes y restricciones de integridad que solo nuestros objetos garantizan que se cumplen
    - Recuerda que nuestro esquema SQL es muy "tonto", esto implica que las restricciones de integridad no se expresan ni garantizan a nivel de BD, sino en el código de nuestros objetos
- Un ORM generalmente será la solución más fácil para implementar esta estrategia
- Si no usamos un ORM, la implementación tampoco tiene que ser complicada
  - Una fila de una tabla contiene un objeto, quizás junto con el resto de objetos subsidiarios si es la raíz de un agregado y una clave ajena en la tabla se traduce en una referencia a otro objeto entidad

BD SQL existente con su modelo propio

- Si la BD SQL se creó para otro sistema, en realidad tenemos dos modelos de dominio coexistiendo en el mismo sistema
  - El del sistema original (que está implícito en las tablas de esa BD) y el del que estamos creando nosotros
  - Puede tener sentido adaptar nuestros objetos al modelo implícito en el sistema original, o puede ser mejor hacerlo totalmente distinto y mapear nuestros objetos en las tablas existentes como veamos
- Si ajustamos nuestros objetos al modelo SQL existente, un ORM puede ser una buena opción, aunque habrá que ajustar cosas a mano
  - Como mínimo nombres y tipos de campos/columnas, pero también restricciones de integridad, claves...
- Si preferimos un modelo de objetos más independiente, el ORM puede no ser la mejor solución

BD SQL diseñada para nuestro sistema, pero no como simple almacén

- Si la BD SQL se crea nueva para nuestro sistema, y queremos aprovechar la potencia de las consultas SQL, podemos considerar tener un esquema bastante distinto del modelo de objetos
  - Si descartamos la idea "simple almacén de objetos", tenemos que aceptar que el mapeo objetos-tablas ya no será tan simple
  - Podemos crear un esquema SQL más limpio y más independiente de nuestro modelo de objetos
  - La BD SQL puede ser usada por otros sistemas sin tener que pasar por nuestros objetos
    - Porque ese mejor esquema SQL va a ir acompañado de sus restricciones de integridad, y quizás hasta de triggers etc.
  - Puede ser más fácil mantener el código, porque los objetos los podemos cambiar sin que eso se traduzca necesariamente en cambios en las tablas de la BD
- Esta opción es complicada si ponemos un ORM de por medio, porque el trabajo del ORM es mantener sincronizados objetos y tablas automáticamente y con esta opción esa sincronización no va a ser tan fácil de automatizar
  - Si vas a ir por este camino, es probable que un ORM no sea la mejor solución

Spring Data JPA

Framework: Spring Data JPA

- La Java Persistence API (JPA) es una API estandarizada para la persistencia en Java, que soporta ORM (mapeo objeto-relacional) y tiene su propio lenguaje de consultas (JPQL)
  - Hay bastantes implementaciones, como Hibernate y ObjectDB
  - JPA define un ORM, es decir, que es para cuando tenemos una BD SQL
- Spring Data JPA añade algunas cosas, especialmente repositorios, pero no implementa JPA, sino que da acceso transparante a otras implementaciones (Hibernate)
  - A pesar de que en teoría esos repositorios están inspirados por el concepto de DDD, la realidad es un poco más complicada como ahora veremos
- La documentación de Hibernate es muy completa con respecto al estándar JPA (además de con las cosas adicionales que tiene Hibernate) y la documentación de Spring Data JPA es imprescindible para ver cómo se usan sus repositorios
  - La de ObjectDB puede venir bien para complementar a la de Hibernate
- Vamos a ver algunos elementos esenciales de JPA, especialmente en su relación con los conceptos de DDD, pero la documentación os resultará imprescindible si lo vais a usar

Spring Data JPA - Entidades

- El concepto de Entidad de DDD es similar al de JPA, al menos en lo que se refiere a su persistencia, así que las podemos expresar como POJOs anotados con @Entity en lugar de definir nosotros una clase abstracta / interfaz para las Entidades
  - Normalmente con un identificador anotado con @Id
  - Es muy posible que aún usando @Entity sigamos queriendo una clase base abstracta y/o interfaz para Entidades (para poder explotar el polimorfismo OO en nuestro código)
  - Sugerencia: explorad las anotaciones @GeneratedValue y @GenericGenerator para que los id se creen automáticamente

```
@Entity
public class Person {
   @Id
   private Long id;
   ...
}
```

Spring Data JPA - Entidades

- Si tenemos una jerarquía de entidades, hay varias opciones para gestionarla
- La más sencilla, y adecuada para DDD, es @MappedSuperclass donde la herencia se gestiona a nivel de objetos pero no a nivel de BD
  - En la BD cada subclase se mapea a una tabla distinto que incluye tanto sus atributos como los de su clase madre

```
@MappedSuperclass
public abstract class Animal {
    @Id private Long id;
}
@Entity
public class Person extends Animal {
    // el id lo hereda
    ...
}
```

Spring Data JPA – Objetos Valor

- Aparte de las entidades, JPA da persistencia a los tipos de Java simples (tipos primitivos, Strings, Dates…), las Colecciones, Mapas y Arrays, los Enum Types y alguna otra cosa
  - Desde el punto de vista de DDD, estos son objetos valor
- Para los objetos valor de clases definidas por nosotros mismos, los definimos con @Embeddable y los referenciamos con @Embedded
  - Atención: es necesario que tengan un constructor vacío y los atributos no pueden ser `final`. Tenemos que sacrificar la inmutabilidad que nos gustaría tener en nuestros objetos valor para que JPA les de persistencia

```
@Entity
public class Persona {
    @Embedded
    private Dirección dir;
}
```

```
@Embeddable
public class Dirección {
    private String calle;
    private int num;
...}
```

Spring Data JPA – Tipo de Acceso

- En JPA podemos anotar atributos de las clases (*field access*), o podemos anotar los getters de esos atributos (*property access*)
  - Pero debemos ser consistentes o fallará
  - Lo que hagamos con la anotación @Id determinará el tipo de acceso elegido para esa entidad
    - Sus objetos @Embeddable deberán hacerlo igual
- La diferencia es que con *field access* se guardan y pueblan los atributos directamente, mientras que con *property access*, el ORM lo hace a través de los getters y setters
- Para DDD el *field access* resulta en general un poco más conveniente
  - Es muy normal que queramos añadir más métodos que simples getters y setters, y si usáramos *property access* habría que acordarse de anotarlos como @Transient
  - También queremos evitar usar getters y setters cuando no sean estrictamente necesarios, así que es posible que algún atributo persistente no pudiera ser anotado en el getter porque no lo tendría

Spring Data JPA – Objetos Valor

- Las colecciones de objetos valor, ya sean básicos o definidos por nosotros con @Embeddable, las tenemos que etiquetar como @ElementCollection
  - Se les trata como objetos valor: su existencia y persistencia van ligadas totalmente a la entidad a la que pertenecen
  - Una clase @Embeddable que pertenece a una @ElementCollection no puede tener a su vez otras @ElementCollection

```
@Entity
public class Persona {
   @Id
   private Long id;

  @ElementCollection(fetch = FetchType.EAGER)
  private List<String> apodos = new ArrayList<>();
```

Spring Data JPA – Agregados

- Un agregado de DDD es un grupo de objetos relacionados que se trata como una unidad desde el punto de vista de la persistencia
- JPA no tiene un concepto similar, así que hay que decidir la mejor forma de usarlo para los agregados
- La entidad raíz, y si hay alguna otra entidad en el agregado, las anotaremos con @Entity con su @Id correspondiente
  - Las que no son raíz realmente no necesitan un @Id globalmente único, aunque no pasa nada porque lo tengan
    - Pero podéis explorar las anotaciones @MapsId y @EmbeddedId si realmente necesitáis identificadores derivados para estas
- Las relaciones que no podamos gestionar como @ElementCollection, por ejemplo porque son entre entidades o porque queremos que tengan un reflejo en tablas más flexible, las anotaremos adecuadamente
  - @OneToOne, @ManyToOne, @OneToMany y, debería ser poco habitual, @ManyToMany

Spring Data JPA – Agregados

- Con los parámetros fetch y cascade de esas relaciones anotadas vamos a delimitar la frontera del agregado. Una recomendación básica general (pero con un modelo complicado o problemas de prestaciones podría no ser suficiente) es:
  - fetch = LAZY y sin cascade (o cascade={}) para asociaciones que cruzan fronteras de agregados
  - fetch = EAGER, cascade = ALL y orphanRemoval = true para asociaciones dentro de agregados
- Es decir, cuando reconstituyamos una entidad raíz de una agregado, queremos que venga con todos los objetos que son parte de ese agregado (fetch = EAGER) pero no con las entidades raíz de otros agregados con los que pueda estar relacionada (fetch = LAZY)
- Y cuando borremos o modifiquemos cualquier cosa dentro de un agregado, ese cambio afecta a todo el agregado (cascade = ALL y orphanRemoval = true), pero no a otros agregados

Spring Data JPA – Agregados

- Realmente para las asociaciones entre objetos de distintos agregados hay una opción mejor
  - Recordad que las reglas de los agregados solo permiten guardar referencias a un único objeto de otro agregado, que es su entidad raíz
- Esta opción es no tener asociaciones navegables entre objetos de distintos agregados
  - Sí que podemos tener asociaciones entre ellos, pero de manera indirecta, guardando simplemente el id de la raíz del otro agregado
  - Y que sea el servicio que necesite acceder a los dos agregados el que se encargue de ir a los repositorios correspondientes y sacar los objetos de allí y luego guardarlos si ha habido cambios
- Esto facilita mucho evitar errores en la gestión de las fronteras de los agregados, y nos ayuda a seguir la recomendación de que en cada transacción solo debería modificarse una instancia de un agregado
  - Con este planteamiento hay que hacerlo con intención y explícitamente si realmente quieres transacciones que involucren a dos o más agregados

Spring Data JPA – Agregados

```
@Embeddable public class IdCliente {
    @Column(name = "id_cliente")
    private Long idCliente;
    ...
}
@Entity public class Pedido extends RootEntity {
    @Embedded
    private IdCliente idCliente;

    // En lugar de tener una asociación navegable como esta:
    // @ManyToOne(fetch=FetchType.LAZY, cascade={})
    // private Customer customer;
    ...
}
// Si un servicio necesita navegar del Pedido al Cliente, lo haría a
// través del clienteRepository:
Cliente c =  clienteRepository.retrieve(pedido.getIdCliente());
```

Spring Data JPA – Repositorios

- Spring Data, que no el estándar JPA, tiene repositorios similares a los de DDD
  - Con algún matiz que vemos ahora
- Spring Data define una interfaz `Repository`, pero la que normalmente se usa como base es esta:

```
public interface CrudRepository<T, ID extends Serializable>
  extends Repository<T, ID> {
  <S extends T> S save(S entity);
  Optional<T> findById(ID primaryKey);
  Iterable<T> findAll();
  long count();
  void delete(T entity);
  boolean existsById(ID primaryKey);
  ...
}
```

Spring Data JPA – Repositorios

- Lo que se hace es extender esa interfaz para las entidades que lo necesiten (añadiendo o no algunos métodos extra)
  - En DDD generalmente habrá un repositorio por cada entidad que sea raíz de un agregado

```
interface PersonaRepository extends CrudRepository<Persona,
Long> {
  long cuentaPorApellido(String apellido);
  long borraPorApellido(String apellido);
  List<Persona> encuentraPorApellido(String apellido);
  ...
}
```

Spring Data JPA – Repositorios

- Si los métodos que añades siguen ciertas convenciones de nombrado, se implementarán las consultas adecuadas contra la BD automágicamente
  - P.ej. Un método findXByZ devolverá entidades de tipo X buscando por su atributo Z sin que lo tengas que implementar tú
- Si tienes que hacer consultas más complicadas que lo que se soporta, puedes añadir la consulta JPQL en una Anotación @NamedQuery a tus @Entity o bien puedes ponerlo en una anotación @Query directamente en la operación de tu repositorio

```
public interface UserRepository extends CrudRepository<User,
Long> {

  @Query("select u from User u where u.emailAddress = ?1")
  User findByEmailAddress(String emailAddress);
}
```

Spring Data JPA – Repositorios

- Luego los declaras y los usas donde sea necesario
  - En DDD generalmente en servicios de la capa de aplicación y, ocasionalmente, en algún servicio u objeto de la capa de dominio

```
@Service
public class UnServicio {
    private final PersonaRepository personaRepository;

    public UnServicio(PersonaRepository pR) {
        this.personaRepository = pR;
    }

    public void prueba() {
        Persona amanda = new Persona("Amanda");
        personaRepository.save(amanda);
        ...
```

Spring Data JPA – Repositorios

- Si tienes una jerarquía de clases y usas la anotación @MappedSuperclass, puedes definir un repositorio genérico para la clase madre, quizás con consultas especializadas, y luego repositorios específicos para las hijas que la extiendan

```
@MappedSuperclass
public abstract class Person {
   @Id Long id;
   String name;
   ...
}

@Entity
public class Worker extends
Person {…}
```

```
@NoRepositoryBean
public interface PRepo<T extends Person>
extends CrudRepository <T, Long> {

@Query(“select t from #{#entityName} t
where t.name = ?1”)
List<T> findByName(String name);
}

public interface WRepo extends
Prepo<Worker> {...}
```

Spring Data JPA – Transacciones

- ¿Dónde están las diferencias con los repositorios DDD?
- En los CrudRepository de Spring Data cada operación va por defecto en su propia transacción
  - Esencialmente cada operación está anotada con @Transactional
  - Pero en DDD normalmente queremos que el límite de las transacciones se decida en la capa de aplicación, no en los repositorios que están en la capa de dominio
- Si hace falta puedes redefinir los métodos que necesites en tus extensiones de CrudRepository y alterar su comportamiento transaccional
- Pero lo más sencillo y directo para encajar DDD es definir operaciones @Transactional en servicios que usen uno o más repositorios
  - Para que esto funcione tenemos que haber anotado explícitamente con @EnableTransactionManagement la clase anotada con @Config (o con @SpringBootApplication) en nuestra aplicación
  - La anotación @Transactional solo funcionará en beans de Spring, p.ej. clases anotadas con @Service que será lo más frecuente en DDD
  - El comportamiento con respecto a las transacciones de las operaciones de los repositorios que se llamen desde estas operaciones @Transactional es anulado automáticamente

Spring Data JPA – Transacciones

```
@Service
public class UnServicio {
    private final PersonaRepository personaRepository;
    private final RolRepository rolRepository;
    public UnServicio(PersonaRepository pR, RolRepository rR) {
        this.personaRepository = pR;
        this.rolRepository = rR;
    }

    @Transactional
    public void añadeUnRolATodos(String nombreRol) {
        Rol rol = rolRepository.findByName(nombreRol);
        for (Persona p : personaRepository.findAll()) {
            p.addRole(rol);
            personaRepository.save(p);
        }
    }
```

Spring Data JPA – Transacciones

- El @Transactional se puede poner a una clase y se aplica a todos sus métodos públicos, o bien solo a los métodos que se quiera
  - El @Transactional solo funcionará en métodos públicos
    - Las llamadas a los protected/private/package no pasan a través del proxy que crea Spring para inyectar el código de gestión de la transacción y por tanto no serían transaccionales
  - Aunque el @Transactional esté en un método público, si lo llamamos desde dentro de su misma clase no será transaccional
    - Por la misma razón: las llamadas desde dentro de una misma clase a sus propios métodos no pasan por el proxy y por tanto no tienen oportunidad de ejecutar el código que gestiona la transacción
- Su funcionamiento depende en parte de la implementación que use Spring por debajo
  - Puede ser JDBC, JPA, Hibernate, JTA (Java Transaction API) dependiendo del entorno (servidor de aplicaciones), dependencias (p.ej. si declaramos Spring Data JPA como dependencia) y configuración explícita que hagamos

Spring Data JPA – Transacciones

- Cada @Transactional puede configurarse con algunos atributos. Por defecto:
  - Las transacciones se propagan (propagation), lo que significa que si varios métodos @Transactional se llaman en el contexto de una transacción, todos se ejecutan dentro de la misma
  - El nivel de aislamiento (isolation) de la transacción es el que tenga por defecto la implementación subyacente
    - Para JDBC o JPA, serán transacciones ACID "duras", es decir, totalmente aisladas unas de otras
  - Son de lectura-escritura
  - El timeout es el que tenga por defecto la implementación subyacente
    - En el caso de JPA no hay timeouts
- La configuración por defecto con JPA es razonable para DDD si no necesitamos transacciones globales
  - Es decir, que involucren a BD distintas, o incluso a otros componentes como p.ej. un broker de mensajes
    - Si necesitamos esto, necesitaremos una implementación subyacente que las soporte, como JTA

Spring Data JPA – Transacciones

- ¿Cuándo se hace un rollback de una transacción anotada con @Transactional?
- Por defecto, cuando el código de un método @Transactional lanza cualquier excepción *unchecked*
  - RuntimeException o Error
- Pero puedes configurar explícitamente que se haga el rollback cuando se lance una excepción de cualquier tipo, incluso uno definido por ti
  - Propiedad rollbackFor de @Transactional, por ejemplo `@Transactional(rollbackFor = Exception.class)`
  - Notad que si esa excepción se captura dentro del método, entonces no se llegará a hacer el rollback

Persistencia en DDD con MongoDB

- Hemos visto antes que las BD SQL no son la opción más fácil para hacer DDD
  - Aunque si vamos a usar una, escribiremos mucho menos código repetitivo con un ORM como Spring Data JPA
- Si podemos elegir trabajar con una NoSQL, las de documentos como MongoDB encajan muy bien, podemos implementar los elementos que necesitamos de manera bastante sencilla
  - Un mecanismo para serializar/deserializar objetos en BSON/JSON
  - Algo para que MongoDB asigne ids únicos a las entidades que son raíz de sus agregados
  - Una referencia al nodo/clúster de MongoDB
  - Una colección para cada tipo de agregado
    - Recuerda que un repositorio esencialmente aparenta ser una colección de objetos de cierto tipo que encapsula un mecanismo de persistencia

Persistencia en DDD con Spring Data MongoDB

- Si no queremos implementar esas cosas nosotros mismos con el driver de MongoDB, podemos usar algo como Spring Data MongoDB
  - No necesitas anotar tus clases con @Entity, @Id, @Embeddable…
    - Por convención, si un atributo de tu clase se llama id, esa es la @Id y la clase es una @Entity
    - Si lo prefieres, puedes marcar con @Id el atributo que quieras (en cualquier caso tiene que ser String, ObjectId o BigInteger)
  - Tienes los repositorios de Spring Data que funcionan esencialmente igual que para el caso de Spring Data JPA
    - Necesitas activarlos con @EnableMongoRepositories en tu clase @Configuration (o @SpringBootApplication)
    - Puedes tener consultas automágicas a partir de convenciones de nombre de las mismas, o puedes definirlas en el JSON de MongoDB o con alguna alternativa más
  - El control de transacciones es menos claro que con JPA
    - El soporte para transacciones de las BD NoSQL es más limitado que en las SQL

Bibliografía

- Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*
  - Capítulo 6
- Vaughn Vernon. *Implementing Domain-Driven Design. Addison Wesley*, 2013
  - Capítulo 12