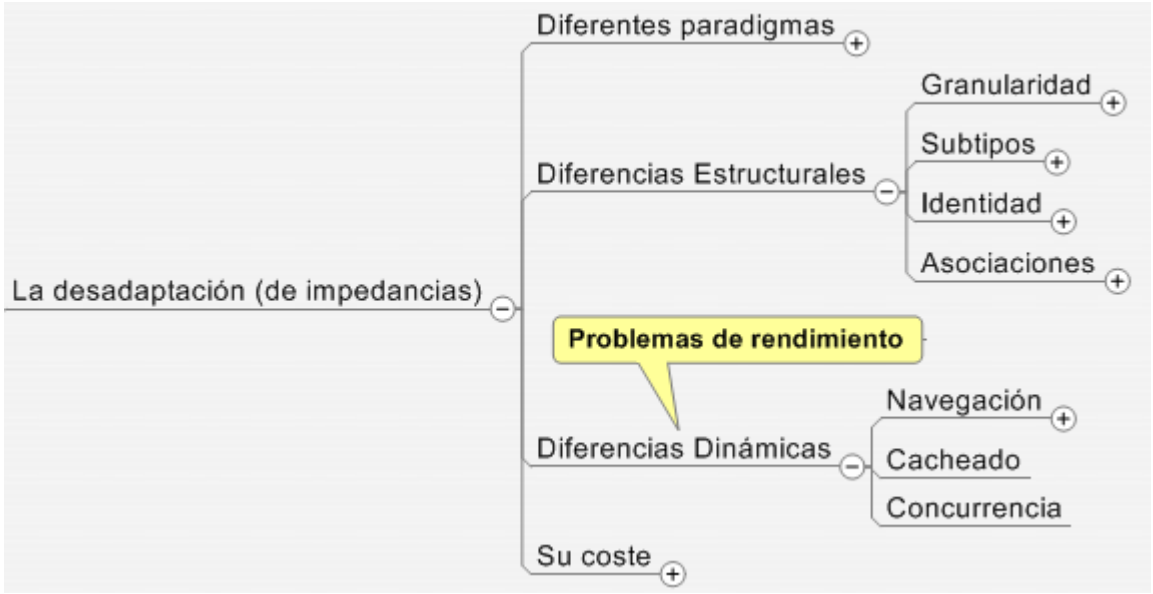
Desadaptación objeto-relacional



Diferentes paradigmas

- Mundo OO
  - Los objetos se relacionan entre sí formando grafos
  - Navegación por referencias
  - No hay modelo formal
- Mundo Relacional
  - Los datos están en tablas con integridad referencial
  - Operaciones con semántica formal definidas por el algebra relacional
  - Operaciones siempre dan tablas (conjuntos)
  - No hay navegación, hay joins entre tablas

Ejemplo

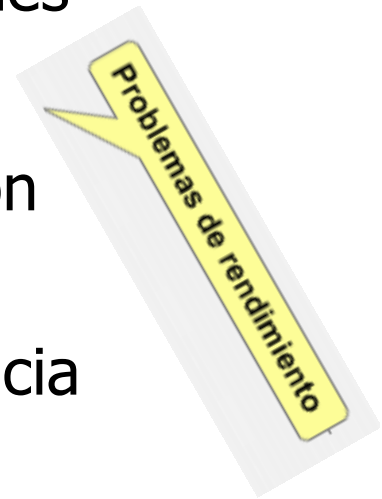
- Calcular el importe de la mano de obra de una avería

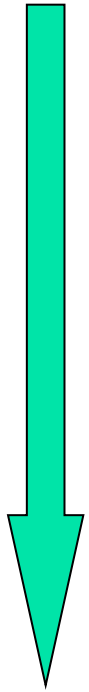
```
total += minutos * vehiculo.getTipoVehiculo().getPrecioHora();
```

```
select a.minutos * tv.preciohora
from TAverias a, TVehiculos v, TTipoVehiculo tv
where a.vehiculo_id = v.id
  and v.tipo_id = tv.id
  and a.id = ?
```

Diferencias

- Estructurales
  - Granularidad
  - Identidad
  - Subtipado
  - Asociaciones
- Dinámicas
  - Navegación
  - Cacheado
  - Concurrencia



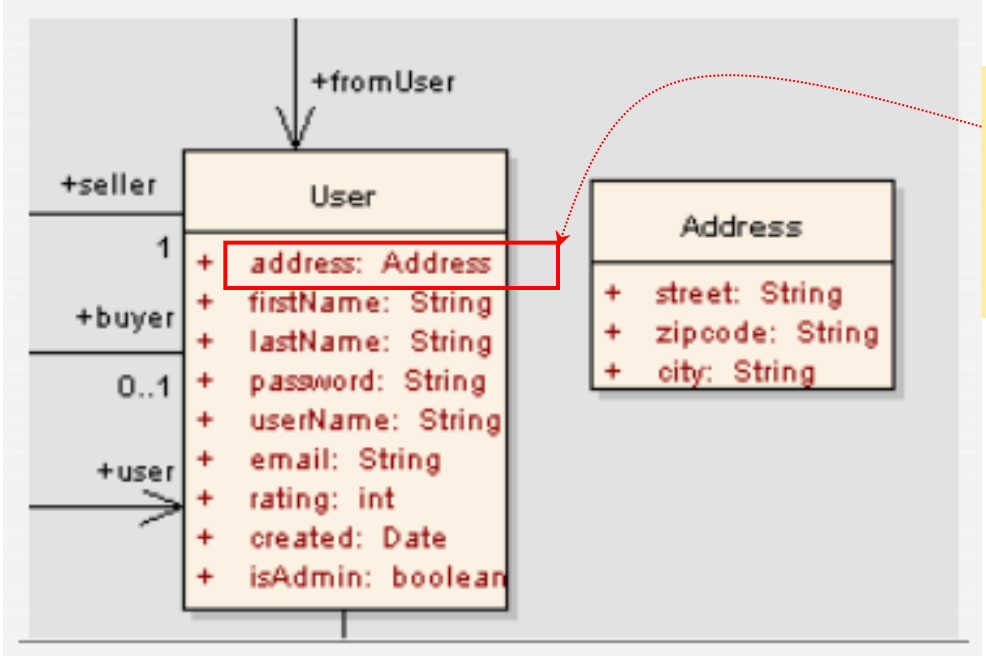


Poca

Grado de dificultad

Mucha

Granularidad: Ejemplo



Address en un atributo de una clase e implementado como un una clase …

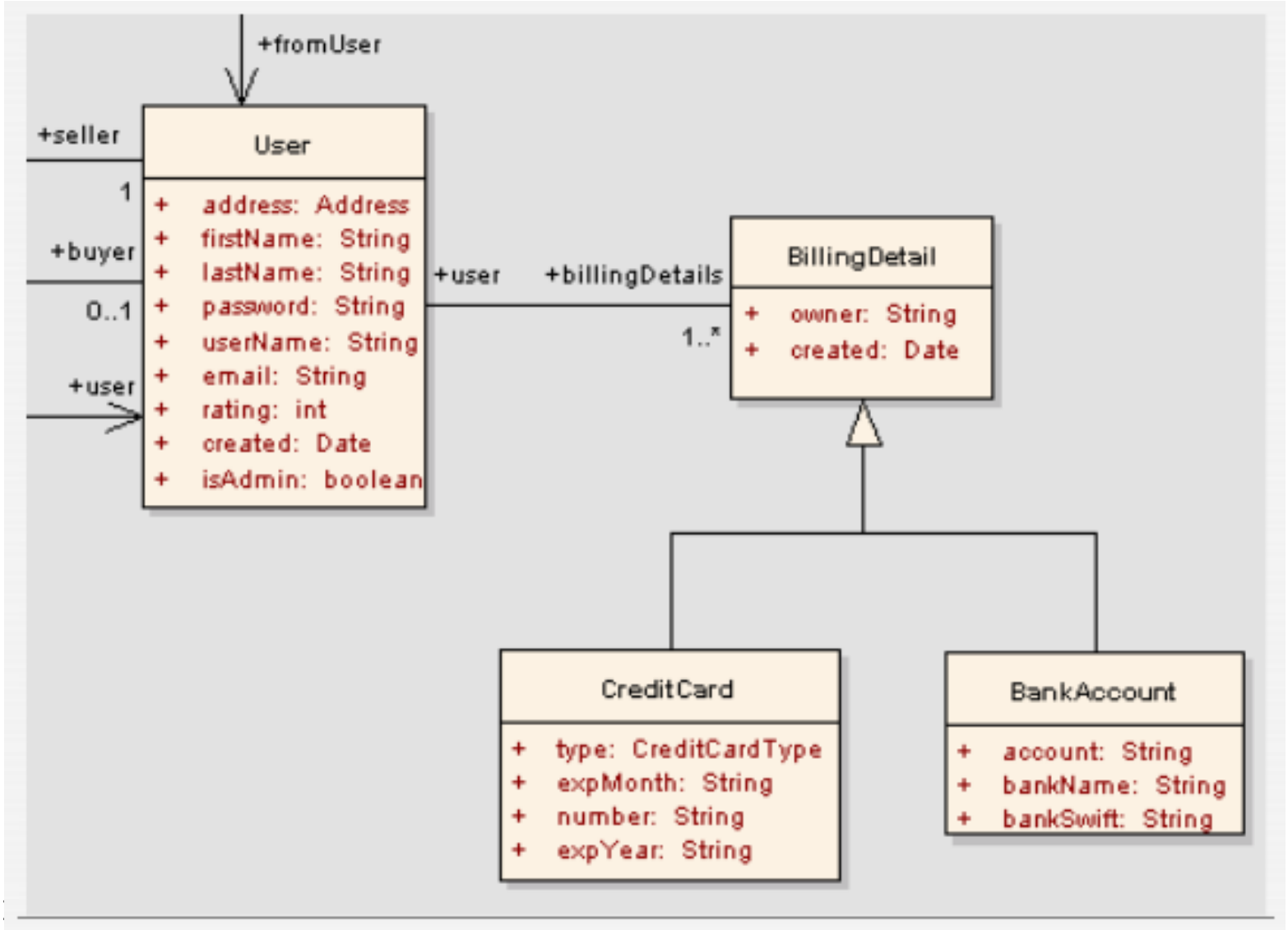
… en una tabla son varios atributos

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    NAME varchar(50) not null,
    ADDRESS_STREET varchar(50),
    ADDRESS_CITY varchar(15),
    ADDRESS_ZIPCODE varchar(5),
```

Granularidad

- En un modelo de dominio OO hay varios tipos de clases
- Entidades (Entities)
  - Tienen identidad propia y participan en asociaciones
- Value Types
  - No se necesita conocer su identidad, sólo su valor
    - String, Date, Time, Money, Integer, Complex…
  - No participan en relaciones, son atributos
  - Semántica de composición
    - Su ciclo de vida está ligado al de la clase

Subtipos: Ejemplo



Subtipos

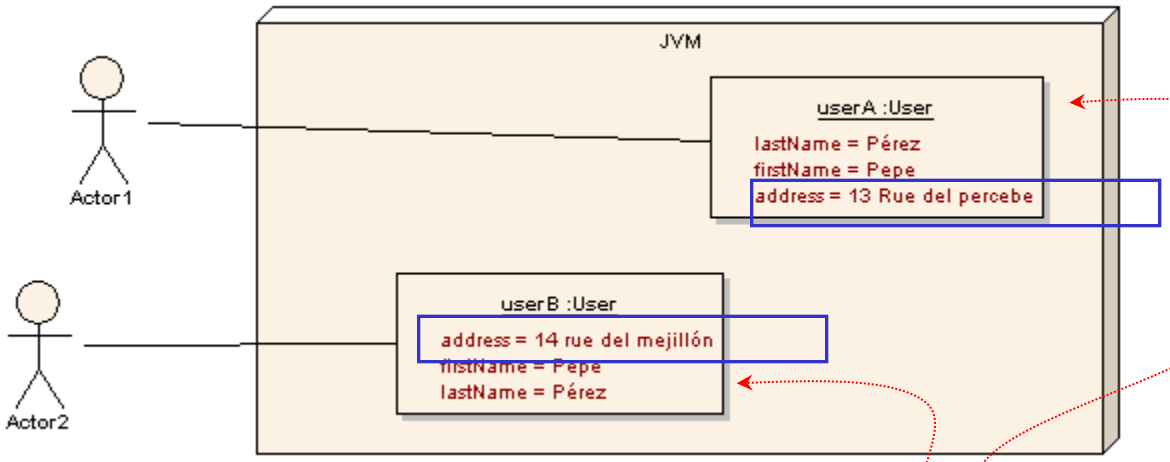
- En los objetos está en el lenguaje
- En el modelo relacional:
  - No existe en la mayoría de las BBDD
  - En algunas existe pero no es estándar
  - El programador lo simula con varias estrategias
  - No existe el polimorfismo
    - Una clave se refiere únicamente a una tabla y no a varias

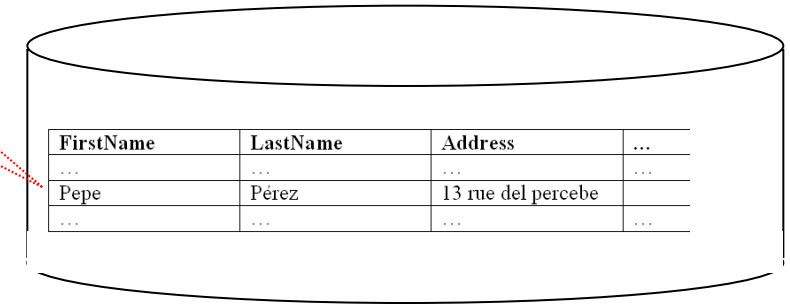
Identidad

- En java
  - Identidad (a == b)
    - dos referencias que apuntan al mismo objeto
  - Equivalencia (a.equals(b))
    - dos objetos que contienen los mismos valores
- En BDD relacional
  - Dos filas son la misma *si tienen la misma clave*

3 identidades

En este caso userA y userB:
- No son iguales
- No son equivalentes
- **¡Pero son la misma fila!**



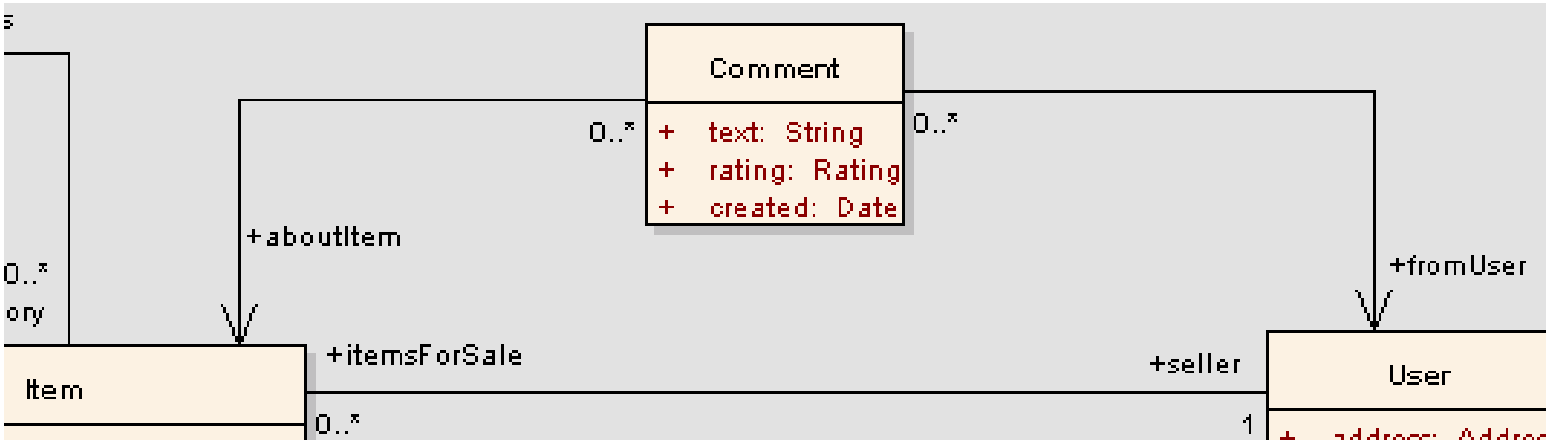


FirstName	LastName	Address	...
...	...	...	...
Pepe	Pérez	13 rue del percebe	
...	...	...	...

```
userA == userB → false
userA.equals(userB) → false
userA.getKey().equals(userB.getKey()) → true
```

Identidad

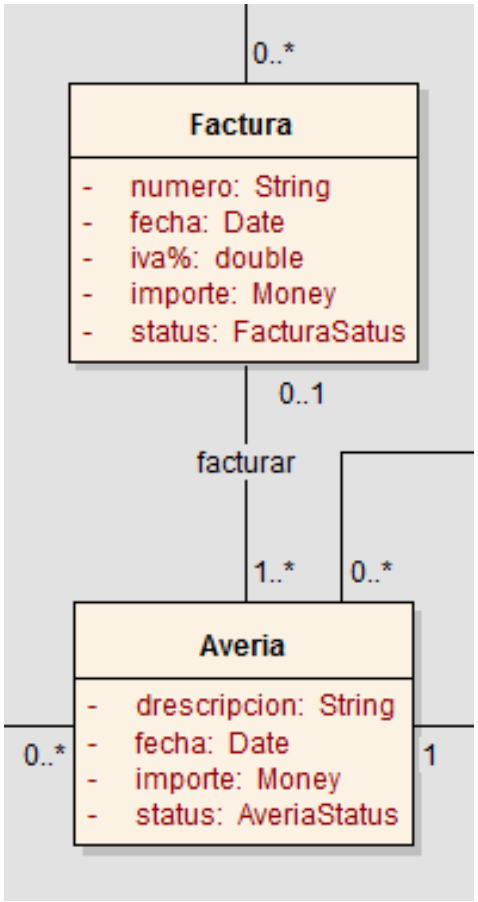
- Se necesita asociar la identidad Java con la identidad en la Tabla
- equals() debe definirse sobre los datos del objeto
  - Varios hilos concurrentes modificando datos pueden hacer que equals() sea distinto
- Problemas con los java.util.Set



Asociaciones

- Navegabilidad
  - Unidireccional
  - Bidireccional
- Cardinalidad
  - Uno a uno
  - Uno a muchos
  - Muchos a muchos

Java: asociaciones como referencias




```
public class Factura {
    ...
    private Set<Averia> averias;
    ...
}
```

```
public class Averia {
    ...
    private Factura factura;
    ...
}
```

Relacional: asociaciones como claves ajenas

- Mediante claves ajenas
- No tienen dirección, no hay navegación
  - Se recuperan datos con consultas que hacen joins

```
USER_ID bigint foreign key references USERS
```

```
select *
    from USERS u
    left outer join BILLING_DETAILS bd on bd.USER_ID = u.USER_ID
    where u.USER_ID = 123
```

Diferencias dinámicas

- Generan pérdidas de eficiencia
  - Navegación
  - Cacheado
  - Concurrencia

Navegación: Ejemplo

- OO

```
aUser.getBillingDetails().getAccountNumber()
```

- SQL

```
select *
    from USERS u
    left outer join BILLING_DETAILS bd on bd.USER_ID = u.USER_ID
    where u.USER_ID = 123
```

Navegación

- En java se recorre un grafo libremente usando las referencias entre objetos
  - Todos los objetos en memoria, los limites son los del grafo
- En SQL se indica qué JOINS hacer
  - Implica conocer de antemano que recorrido vamos a hacer

Navegación

- Se busca crear la ilusión de que todos los objetos ya están en memoria

```
Factura f = Mapper.load(123);
// f está cargado en memoria, pero no sus averias

Set<Averia> averias = f.getAverias();
// ahora las averías están cargadas
```

- Alternativas:
  - Precargar: eager loading
  - Bajo demanda: lazy loading

Navegación

- Eager loading: Se carga un objeto y sus asociados
  - Puede cargar más objetos de la cuenta
  - Riesgo de producto cartesiano
- Lazy loading: Se carga al necesitarlo
  - Puede genera demasiadas SELECT * FROM
  - El problema de las n+1 consultas

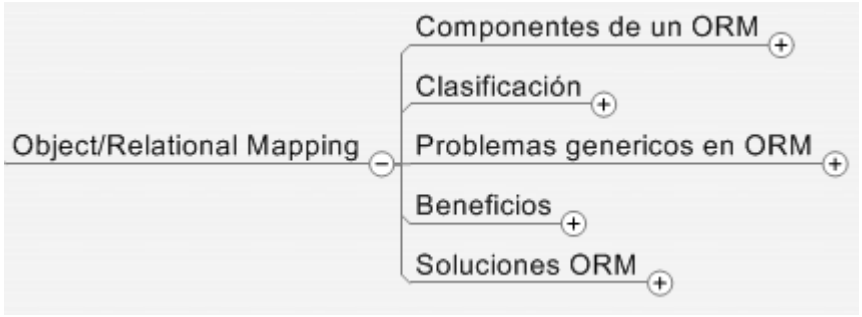
Cacheado

- Optimiza el rendimiento al reducir el trasiego con la BBDD
- Permite hacer optimizaciones
  - Write-behind delayed
  - Batch load/update
- ¿Caché por proceso?, ¿por hilo?, ¿por cluster?

Concurrencia

- Varios hilos de ejecución (usuarios) trabajando sobre los mismos datos…
  - Que pueden estar en caché…
  - ¿Como se controlan las transacciones ACID?
  - Si ya la base de datos lo sabe hacer

Object/Relational Mapping



Aspectos de un ORM

- API para CRUD
- Portabilidad entre BBDD
- Lenguaje o API para hacer consultas
- Metadatos
- Técnicas/políticas configurables
  - Cacheado
  - Precarga
  - Transacciones

Clasificación de los ORM

- Pure relational
  - Procedimientos almacenados
  - Gestión del SQL directa
  - Apto para pequeñas aplicaciones (pocas tablas)
  - Problemas de mantenibilidad
- Light object mapping
  - Clases mapeadas directamente a tablas
  - SQL se oculta tras patrones (DAO) o en clases abstractas
- Medium
  - Aplicación diseñada alrededor de un modelo de objetos
  - SQL generado o soportado por un framework en runtime
  - Se soportan asociaciones y lenguajes de consultas OO
  - Puede ser hecho a mano

Clasificación de los ORM(2)

- Full
  - Semántica de Composición
  - Herencia
  - Polimorfismo
  - Persistencia por alcanzabilidad
  - Persistencia transparente
  - Estrategias de fetching
  - Muy complejos para hacerlos a mano

Curiosidad: Hibernate 80.000 LOC + 25.000 LOC test

Beneficios de usar un ORM

- Productividad
  - Se escribe menos código, con menos errores
- Mantenibilidad
  - < LOC
  - Modelos del dominio son OO, se piensa en objetos
- Rendimiento
  - Bastante eficiente, muy optimizado
  - Posibilidad de ejecutar código malo de cualquier forma
- Independencia de la BBDD

Java Persistence API (JPA)

- JPA es una especificación
  - Como lo es JDBC
  - Trabajaremos con la versión 2.0 (JSR 317)
- Se necesita una implementación
  - Hibernate, EclipseLink, TopLink, CocoBase, OpenJPA, Kodo, DataNucleus, Amber, …
- Interfaz natural para BBDD OO también
  - ObjectDB, Versant ODB, Intersystems C, …