Desadaptación objeto-relacional

Object-relational impedance mismatch

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

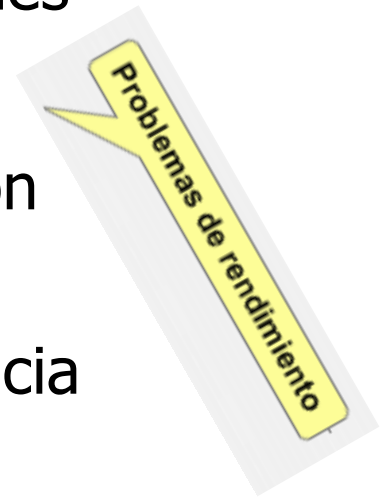
Calcular el importe de la mano de obra de una avería

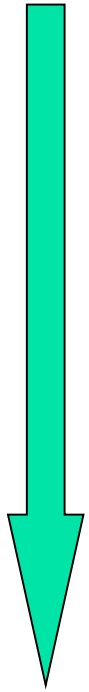
```
total += minutes
    * workOrder.getVehicle().getVehicleType().getPricePerHour()
    / 60.0;
```

```
select i.minutes * vt.pricePerHour / 60.0
from TInterventions i
    join TWorkOrders wo on i.workOrder_id = wo.id
    join TVehicles v on wo.vehicle_id = v.id
    join TVehicleTypes vt on v.vehicleType_id = vt.id
where
    wo.id = ?
```

Diferencias

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





Poca

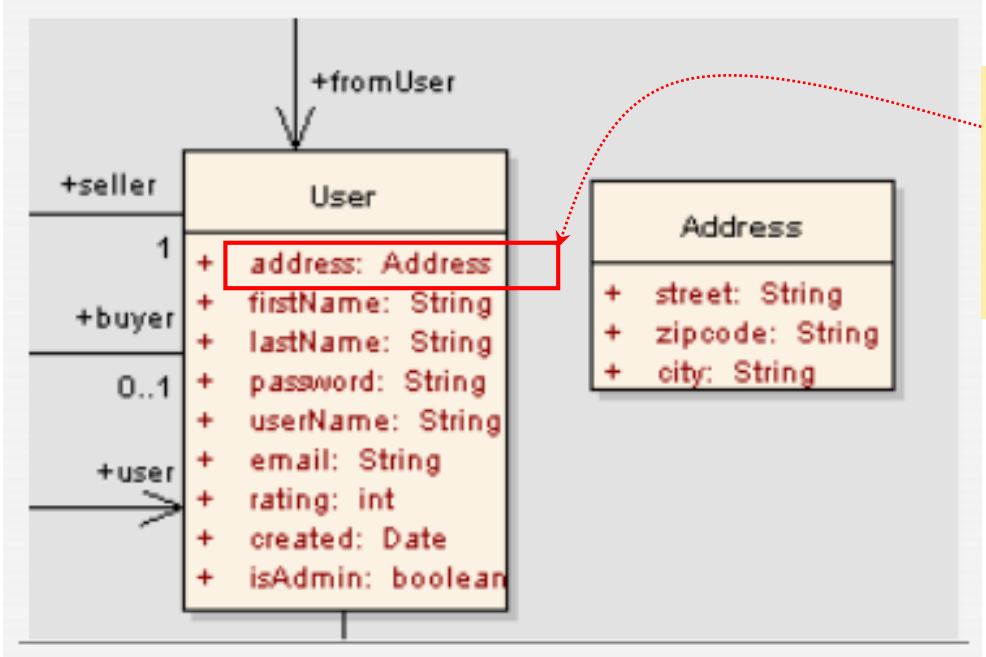
Grado de dificultad

Mucha

Diferencias estructurales

- Afectan al diseño, en ambos mundos
  - Clases en Java
  - Tablas en la base de datos

Granularidad: Ejemplo



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

… en una tabla son varios atributos

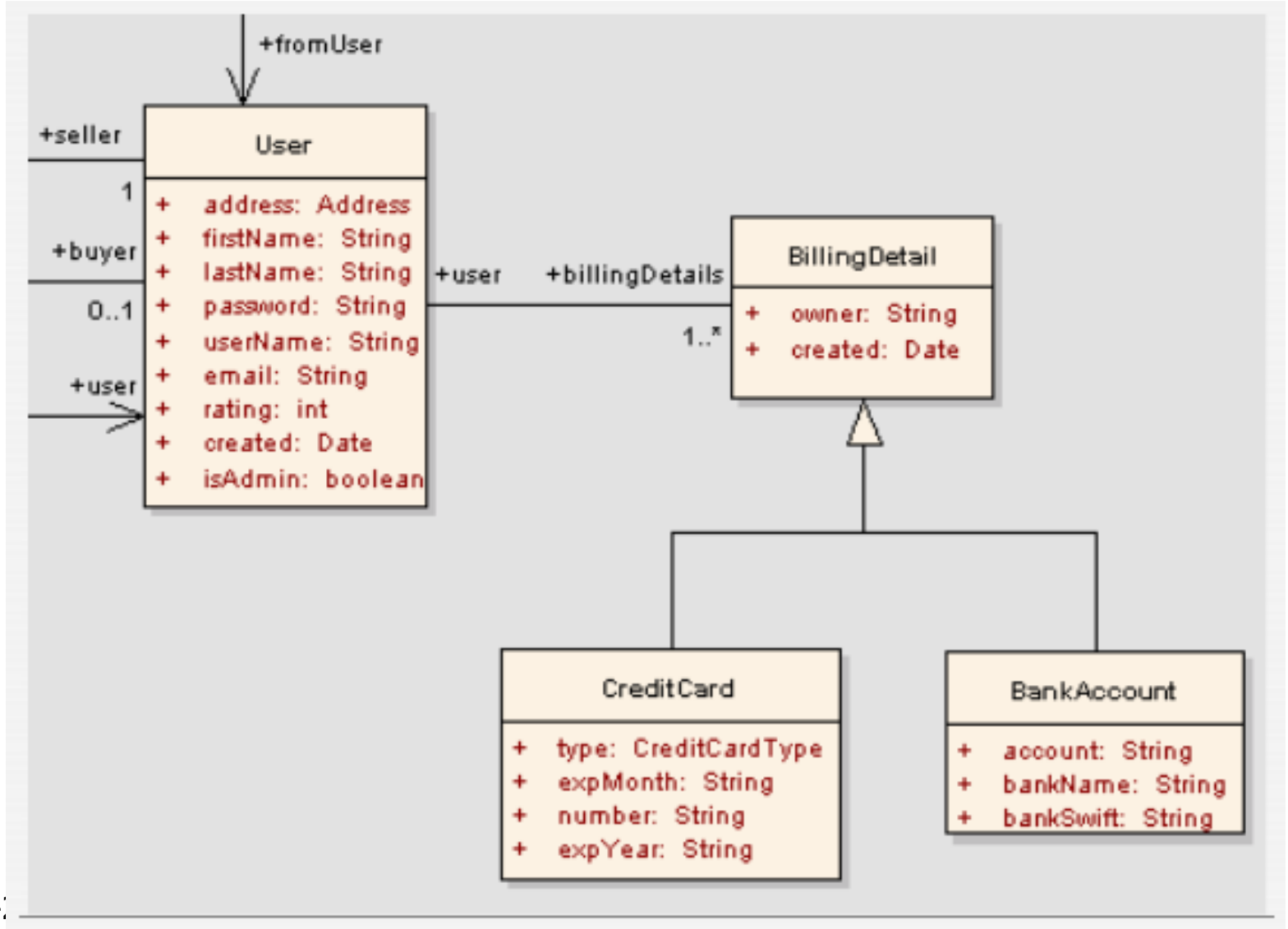
```
create table USERS (
    USERNAME varchar(15) not null primary key,
    NAME varchar(50) not null,
    ADDRESS_STREET varchar(50),
    ADDRESS_CITY varchar(15),
    ADDRESS_ZIPCODE varchar(5),
```

Granularidad

En un modelo de dominio OO hay varios tipos de clases

- Entidades (Entities)
  - Tienen identidad propia, evolucionan a lo largo del tiempo y participan en asociaciones
- Value Types
  - No se necesita conocer su identidad, sólo su valor
    - String, Date, Time, Money, Integer, Complex…
  - Son atributos de entidades, composición
  - Su ciclo de vida está ligado al de la clase

Subtipos: Ejemplo



Subtipos

- En OO es natural
- En el modelo relacional:
  - No existe en la mayoría de las BBDD
    - En algunas existe pero no es estándar
  - Se puede simular 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 representan la misma cosa (bajo cierta definición de equivalencia)

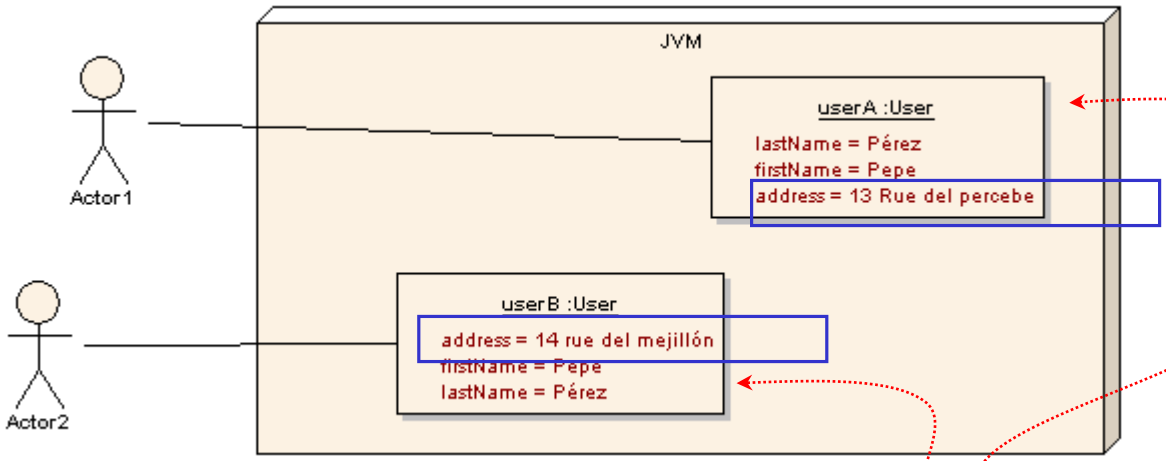
En BDD relacional

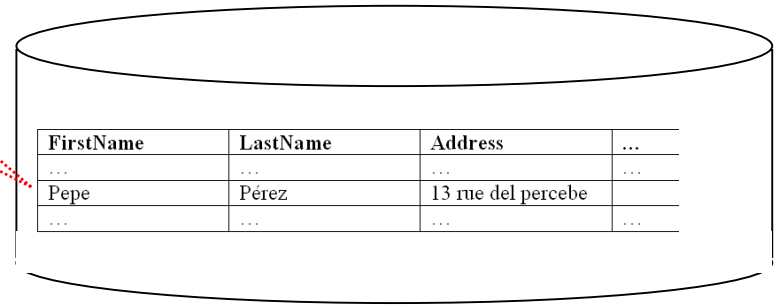
- Clave primaria define la identidad: no hay dos filas con la misma clave

3 identidades

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

Suponiendo equals() and hashCode() definidos sobre todos los atributos





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

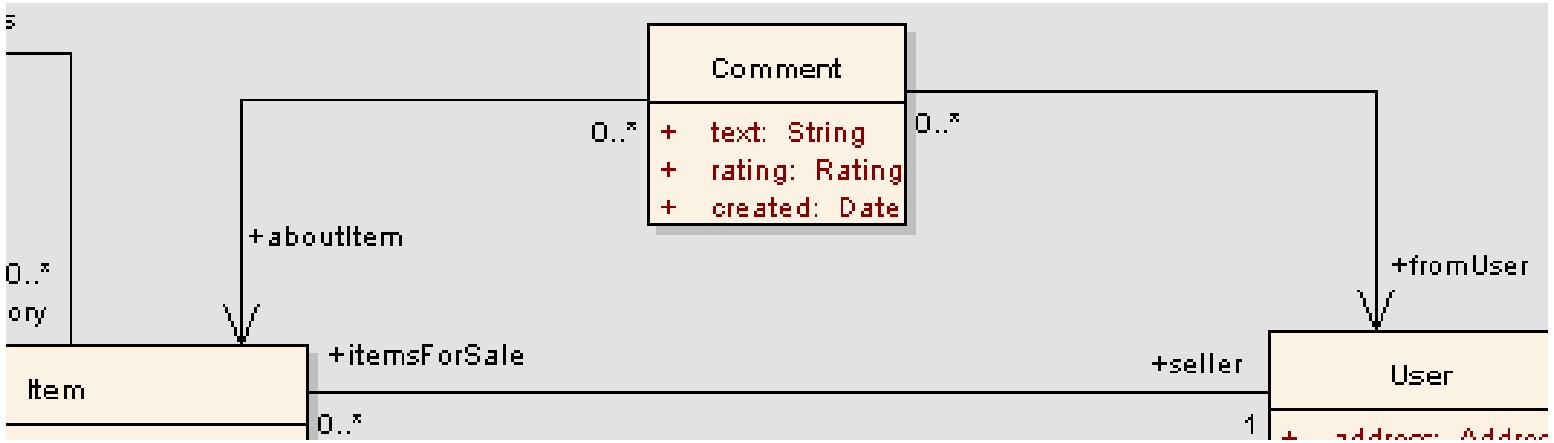
```
userA == userB → false
userA.equals(userB) → false
userA.getKey().equals(userB.getKey()) → true
```

Identidad

¿Cómo vincular la identidad de la entidad con la identidad Java y la clave primaria?

A través de equals() y hashCode(), que deben definirse sobre los datos del objeto

- Problemas con los java.util.Set y Map

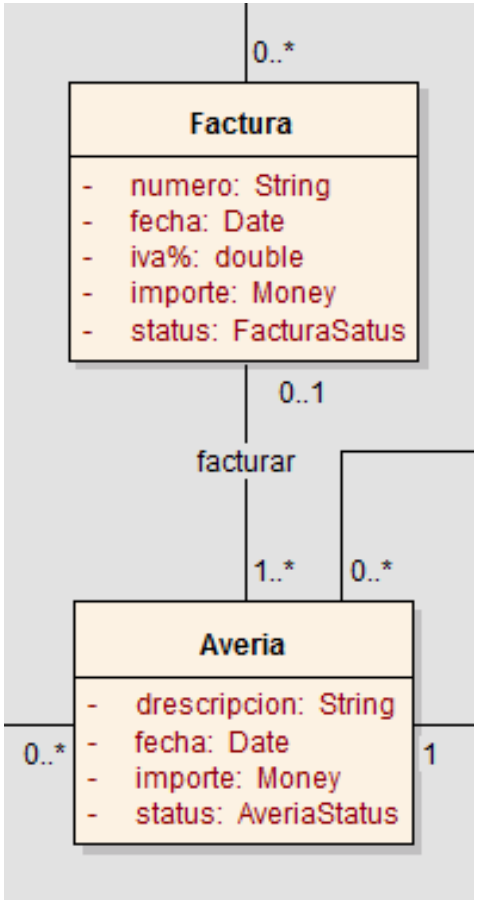


Asociaciones

En concepto OO (UML)

- 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: 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 problemas de pérdida de eficiencia en tiempo de ejecución
  - 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 RAM, 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

```
Invoice i = orm.load(123);                          // Just invoice on memory
Set<WorkOrder> workOrders = i.getWorkOrders();  // Now all the workorders
```

Alternativas:

- Precargar: eager loading
- Bajo demanda: lazy loading

Navegación

- Eager loading: Se carga un objeto y sus asociados
  - Puede cargar (muchos) 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

Cacheado

Diferentes alcances de caché:

- ¿hilo?
- ¿proceso?
- ¿cluster?
- ¿una combinación?

- Gran impacto en concurrencia

Concurrencia

Varios hilos de ejecución (usuarios) trabajando sobre los mismos datos…

Que pueden estar en caché…

¿Como se controlan las transacciones ACID?

Solución más sencilla:

- Caché por hilo
- La BDD gestiona las transacciones

Object/Relational Mapping

Un adaptador entre ambos mundos

Aspectos de un ORM

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

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 2.1 (JSR 338), julio 2017

Con varias implementaciones disponibles

- Hibernate, EclipseLink, TopLink, CocoBase, OpenJPA, Kodo, DataNucleus, Amber, …

Varias BBDD OO puras usan esta especificación para su interfaz

- ObjectDB, Versant ODB, Intersystems C, …