

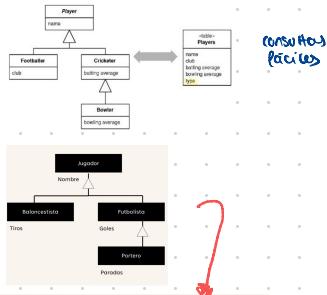


# T7 : PATRONES II

## Object-Relational Structural Patterns

### Single Table Inheritance

Representar herencia → 1 tabla



```
-- Crear la tabla "jugador"
CREATE TABLE IF NOT EXISTS jugador (
  nombre VARCHAR(255) PRIMARY KEY NOT NULL,
  tiros INT,
  goles INT,
  paradas INT,
  tipo VARCHAR(50)
);

INSERT INTO jugador(nombre, goles, tipo) VALUES ('Carlos', 40, 'futbolista');
INSERT INTO jugador(nombre, tiros, tipo) VALUES ('Palomo', 33, 'futbolista');
INSERT INTO jugador(nombre, paradas, tipo) VALUES ('Orianna', 123, 'portero');
```

	nombre	tiros	goles	paradas	tipo
1	Carlos	NULL	40	NULL	futbolista
2	Palomo	33	NULL	NULL	baloncestista
3	Orianna	NULL	1	123	portero

#### Ventajas

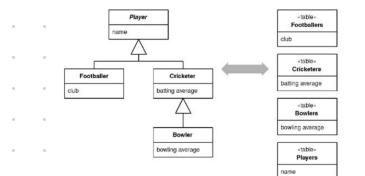
- ④ Solo una tabla
- ④ No hay uniones
- ④ Fácil cambiar jerarquía

#### Desventajas

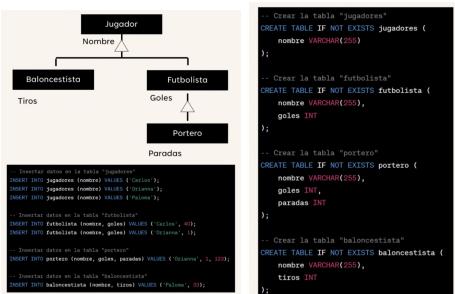
- ④ Campos irrelevantes
- ④ Espacio desperdiciado
- ④ 1 sola tabla → Tabla muy grande
- ④ Espacio de nombres únicos
- se confunde

### Class Table Inheritance

Representar herencia → 1 tabla por clase



un poco de joins



#### Ventajas

- ④ Fáciles de entender
- ④ Campos son relevantes
- ④ Relación modelo de dominio → BD ex<sup>cada</sup>-directa
- ④ Evita uniones complejas

#### Desventajas

- ④ Cambio en la estructura → cambios BDD
- ④ Problemas de integridad referencial  
(claves primarias & foráneas deben ser consistentes)
- ④ ++ normalización → dificultar comprensión consultas ad hoc

### Concrete Class Table Inheritance

Representar jerarquía → cada clase concreta tiene su propia tabla



```
-- Crear la tabla "jugador"
CREATE TABLE IF NOT EXISTS jugador (
  nombre VARCHAR(255) PRIMARY KEY NOT NULL
);

-- Crear la tabla "futbolista"
CREATE TABLE IF NOT EXISTS futbolista (
  nombre VARCHAR(255),
  goles INT
);

-- Crear la tabla "baloncestista"
CREATE TABLE IF NOT EXISTS baloncestista (
  nombre VARCHAR(255),
  tiros INT
);

-- Insertar datos en la tabla "futbolista"
INSERT INTO futbolista (nombre, goles) VALUES ('Carlos', 40);
INSERT INTO futbolista (nombre, goles) VALUES ('Orianna', 1);
INSERT INTO futbolista (nombre, goles) VALUES ('Paloma', 33);

-- Insertar datos en la tabla "baloncestista"
INSERT INTO baloncestista (nombre, tiros) VALUES ('Paloma', 33);
INSERT INTO baloncestista (nombre, tiros) VALUES ('Orianna', 1);

-- Insertar datos en la tabla "portero"
INSERT INTO portero (nombre, goles, paradas) VALUES ('Universo', 1, 123);
```

```
-- Crear la tabla "portero"
CREATE TABLE IF NOT EXISTS portero (
  nombre VARCHAR(255),
  goles INT,
  paradas INT
);
```

#### Ventajas

- ④ Cada tabla es independiente
- ④ Fácil de entender
- ④ No desperdicia espacio
- ④ Cada Clase → solo accede → A su propia tabla
- ④ No se requieren joins → si quieres ver Futbolista → SELECT \* FROM futbolista NO JOIN con jugador (?)

#### Desventajas

- ④ Claves primarias pueden ser complicadas de manejar
- ④ Redundancia de datos → Tabla Futbolista & Tabla Portero
- ④ Consultas q abarquen todas las clases → complicadas

joins y subconsultas

# T7 : PATRONES II

## Object-Relational Structural Patterns

### Identity Field

objetivo: Proporcionar identificador único exclusivo para cada fila/registro

Person
id: long

#### Ventajas

- ① Asociación con claves de BD
- ② Gestión de claves en Objeto en Memoria
- Asociando clave primaria de lo «objeto» → localizar en los campos de obj en item
- ③ Resolución problemas asociados a claves secundarias → significativas? empleo?

→ PQ UTILIZADO?

Ej: usar identificado → DNI

- ④ Es un dato q. t. a la persona, luego puede cambiar (cambio físico?)
- ⑤ Pertenecer a la semántica

Id → no pertenece a la semántica ✓

#### PROBLEMA

"Id" está en muchas tablas y BBDD si se querieren unir puede haber problema de identidad

SOLUCIÓN

UUID → identif único universal  
↳ es largo pero cualquier lo soporta

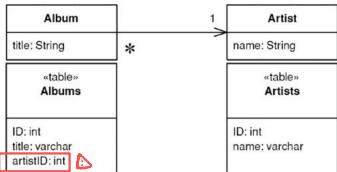
#### Id Field

↳ tipo de objeto q. entraña la clave primaria en uno de sus attr

DNI UUID

### Foreign Key Mapper

Garantizar integridad datos de relaciones 1-N



Cada tabla de la BD tiene 1 Clave Primaria Única a lo que se hace referencia → 1-N

#### Ventajas

- ④ Mantiene integridad referencial
- ⑤ Simplifica consultas relacionales
- ⑥ Evita redundancia de datos

#### CREACIÓN DE LAS TABLAS:

```

CREATE TABLE Autores (
    autor_id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(255) NOT NULL
);
  
```

#### INserCIÓN DE VALORES:

```

INSERT INTO Autores (nombre) VALUES ('Nombre del Autor');
INSERT INTO Libros (titulo, autor_id) VALUES ('Titulo del Libro', 1);
  
```

```

CREATE TABLE Libros (
    libro_id INT AUTO_INCREMENT PRIMARY KEY,
    titulo VARCHAR(255) NOT NULL,
    autor_id INT,
    FOREIGN KEY (autor_id) REFERENCES Autores(autor_id)
);
  
```

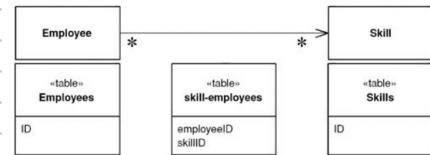
#### SELECCIÓN DE VALORES:

```

SELECT Libros.titulo, Autores.nombre
FROM Libros
INNER JOIN Autores ON Libros.autor_id = Autores.autor_id;
  
```

### Association Table Mapping

Garantizar integridad de datos de relaciones (n-n) tabla ≠



#### Ventajas

- ④ Mantiene integridad referencial
- ⑤ Simplifica consultas relacionales
- ⑥ Evita redundancia de datos

#### CREACIÓN DE TABLAS:

```

CREATE TABLE Estudiantes (
    EstudianteID INT PRIMARY KEY,
    Nombre VARCHAR(50)
);
  
```

```

CREATE TABLE Cursos (
    CursoID INT PRIMARY KEY,
    Nombre VARCHAR(50)
);
  
```

#### Inserción:

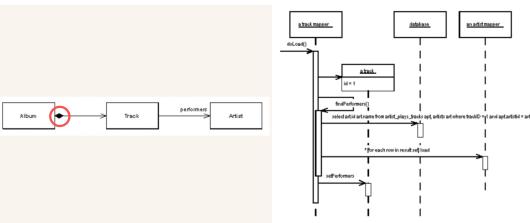
```

INSERT INTO Inscripciones (EstudianteID, CursoID) VALUES (1, 101);
  
```

#### SELECCIÓN:

```

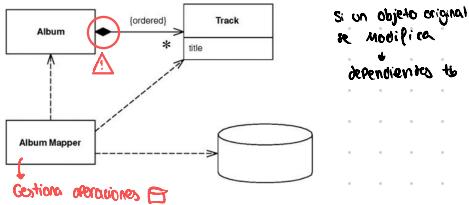
SELECT Cursos.Nombre
FROM Cursos
JOIN Inscripciones ON Cursos.CursoID = Inscripciones.CursoID
WHERE Inscripciones.EstudanteID = 1;
  
```



# T7 : PATRONES II

## Dependent Mapping

Proporciona una gestión coherente de las relaciones entre obj.



```
CREATE TABLE albums (
    album_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    release_date DATE NOT NULL
);

CREATE TABLE tracks (
    track_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    duration TIME NOT NULL,
    album_id INT NOT NULL,
    FOREIGN KEY (album_id) REFERENCES albums(album_id)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
```

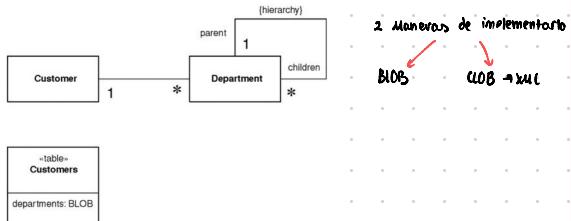
## Inheritance Mappers

ORM → como implementar BD cuando el herencia  
cada clase Dominio → 1 MAPPER guarda datos

## Object-Relational Structural Patterns

### Serialized LOB

Guarda un grafo de objetos serializados en un único objeto grande (LOB)  
que se almacena en un campo



```
CREATE TABLE Customer (
    id UUID PRIMARY KEY,
    nombre VARCHAR(255),
    apellido VARCHAR(255),
    departamento_id INTEGER,
    FOREIGN KEY (departamento_id) REFERENCES Departamento(departamento_id)
);

CREATE TABLE Departamento (
    departamento_id INTEGER PRIMARY KEY,
    nombre_departamento VARCHAR(255),
    id_departamento_pade INTEGER,
    FOREIGN KEY (id_departamento_pade) REFERENCES Departamento(departamento_id)
);
```

```
METH RECURSIVE Subdepartamentos AS
SELECT d.departamento_id, nombre_departamento, id_departamento_pade, 1 AS nivel
FROM Departamento d
WHERE id_departamento_pade = id_departamento_pade
UNION ALL
SELECT d.departamento_id, nombre_departamento, d.id_departamento_pade, s.nivel + 1
FROM Departamento d
JOIN Subdepartamentos s ON d.id_departamento_pade = s.departamento_id
        ) AS
SELECT * FROM Subdepartamentos
WHERE nivel = $;
```

Este patrón propone dos maneras de serializar el grafo:

#### BLOB

Algunas Ventajas

Muchos más simples de implementar y programar

Muchas plataformas serializan gratis el BLOB

#### CLOB

Algunas Desventajas

Es fácil de leer e interpretar ya que está en formato de texto.

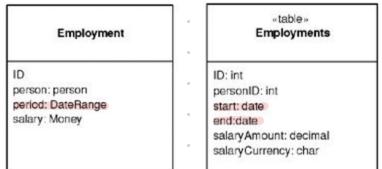
Sus desventajas se ven resueltas al implementar XML

No todos los sistemas lo soportan

Tiende a ocupar mucho espacio. Puede ser mucho más lento, e incluso requerir un parser específico

### Embedded value

Mapa un objeto a varios campos de una tabla.



Representar y manipular objetos que contienen otros objetos como parte de sus atributos

✓ Evitar q los datos se dispersen por x partes del código  
y queremos enrutar esos datos → ~~estructura convencional~~ ~~reutilizable~~

```
CREATE TABLE Money (
    id UUID PRIMARY KEY,
    cantidad INT,
    tipo_moneda VARCHAR(255)
);
```

```
CREATE TABLE Account (
    id_account UUID PRIMARY KEY,
    id_owner UUID,
    cantidad INT,
    tipo_moneda VARCHAR(255),
    FOREIGN KEY (id_owner) REFERENCES User(id_user)
);
```

```
import uuid

class Money():
    def __init__(self, cantidad, tipo_moneda):
        self.id = uuid.uuid4()
        self.id_owner = id_owner
        self.cantidad = cantidad
        self.tipo_moneda = tipo_moneda

class Account():
    def __init__(self, id, id_owner, cantidad, tipo_moneda):
        self.id_account = id
        self.id_owner = id_owner
        self.cantidad = cantidad
        self.tipo_moneda = tipo_moneda

    def depositar(self, cantidad, tipo_moneda):
        if self.cantidad > cantidad:
            self.cantidad += cantidad
            self.tipo_moneda = tipo_moneda
        else:
            print("Saldo insuficiente")
```

TE cargas la 1º forma Normal