ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Modelo Entidad-Relación: Q.E.P.D.: La persistencia en los nuevos *frameworks*

María Consuelo Franky

ConsueloFranky@cincosoft.com

CincoSOFT Ltda.

http://www.cincosoft.com

Tel: (1)6230180

Bogotá

XXVII SALÓN DE INFORMÁTICA

"Una Ingeniería de Software para un mundo cada vez más complejo"

SEPTIEMBRE 19, 20 Y 21 DE 2007 • BIBLIOTECA LUIS ÁNGEL ARANGO • BOGOTÁ

http://www.acis.org.co

ACIS

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Introducción

- *Java EE 5 reemplazó a J2EE hace 1 año: cambio profundo de modelo conceptual, arquitectura y estrategia de desarrollo*
- *El reto es volver a aprender a hacer aplicaciones de una manera completamente distinta:*
  - *pantallas que exponen directamente las entidades del Modelo*
  - *se modela en términos de entidades persistentes y no de tablas de la base de datos*
  - *consecuencia: aplicaciones concisas y eficientes y eliminación de patrones burocráticos*
- *Objetivo de la conferencia: visión general del modelaje unificado de datos en términos de entidades persistentes y no de tablas de la base de datos (Modelaje E-R: Q.E.P.D.)*
- *Aplicable a Java EE 5 pero también a .NET*

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

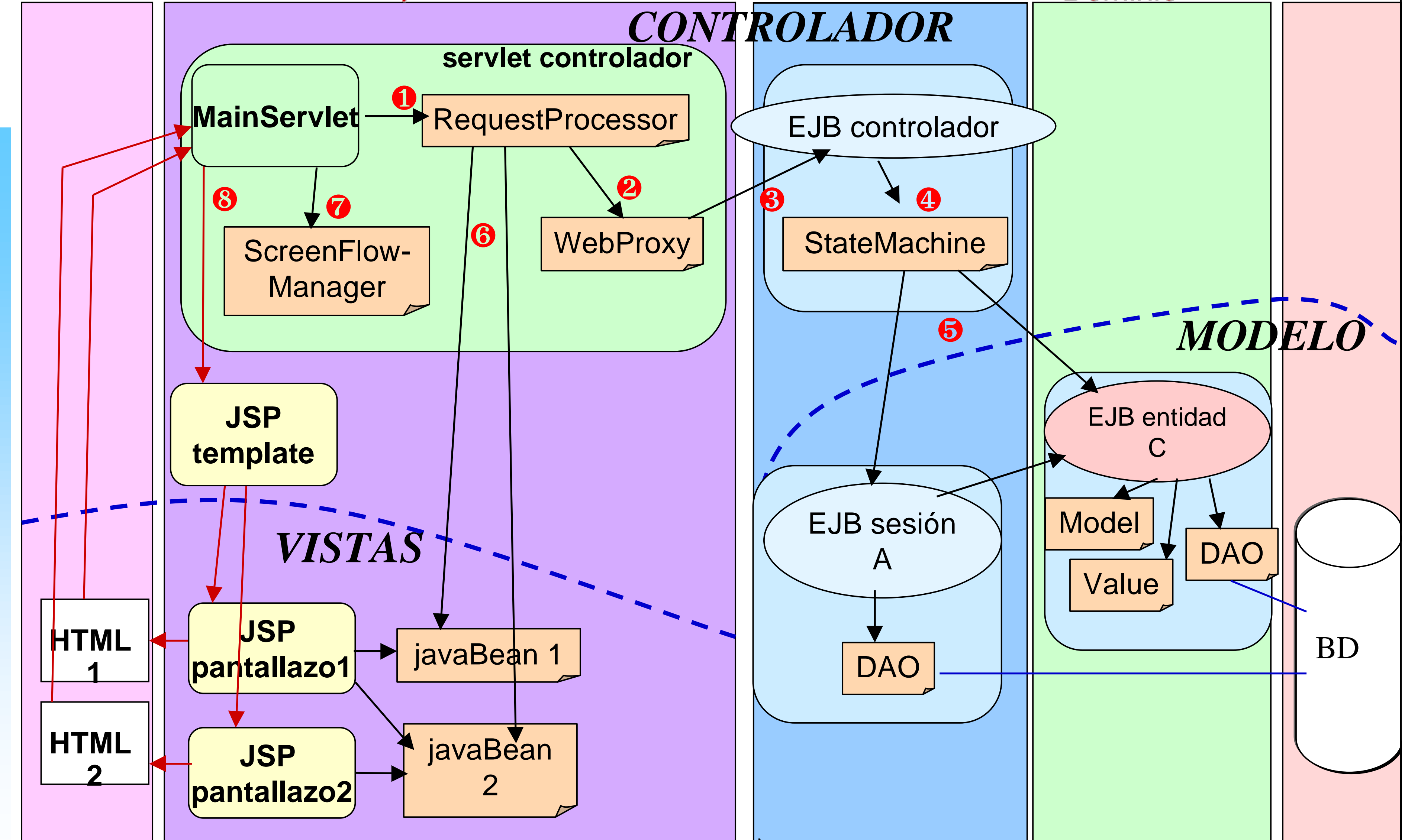
Temario

- Arquitectura de una aplicación Java EE 5
- Facilidades de JPQL para navegar sobre entidades persistentes
- Manejo de relaciones entre entidades
- Herencia entre entidades
- Control opcional en la transformación Objetos - Relacional
- Consultas en SQL nativo

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

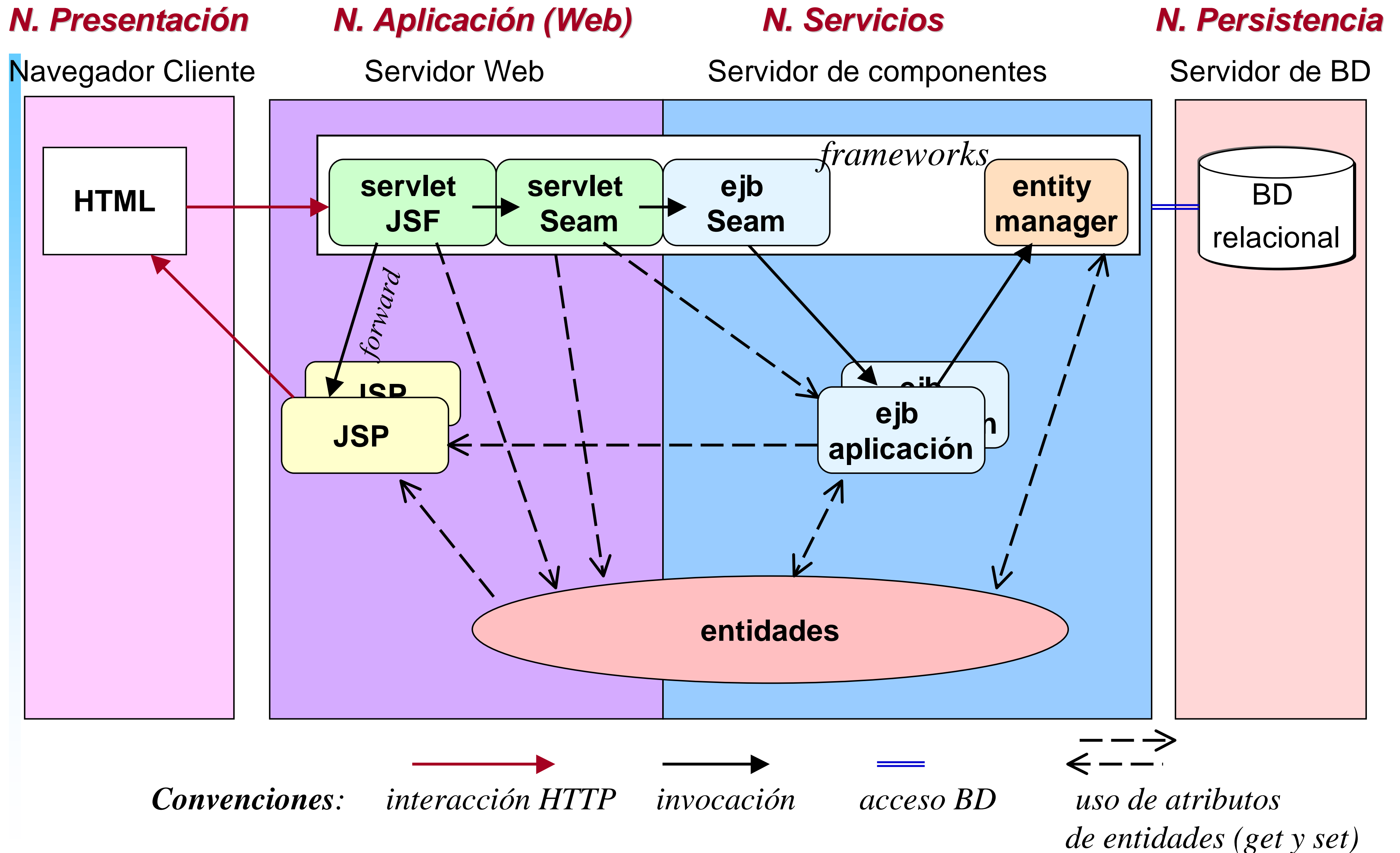
Arquitectura de una aplicación Java EE 5

J2EE: Arquitectura MVC de una aplicación



ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Java EE 5: Arquitectura de una aplicación utilizando frameworks JSF, EJB 3.0 y Seam



Nivel web basado en el framework JSF

- Pantallas se contruyen con componentes gráficos UIComponent que reaccionan a eventos:
  - son componentes de alto nivel que encapsulan elementos HTML y tienen comportamiento asociado
  - muestran y actualizan valores del Modelo contenidos en clases java ("backing beans")
  - como reacción a eventos (por ej: oprimir un botón) invocan directamente métodos de los "backing beans"
- Aspectos de validación de los datos del usuario:
  - Validación automática asociada a cada componente gráfico
  - Cuando la validación falla se vuelve a mostrar la pantalla junto con los mensajes de error
  - Procesamiento de eventos solo cuando la validación es exitosa

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

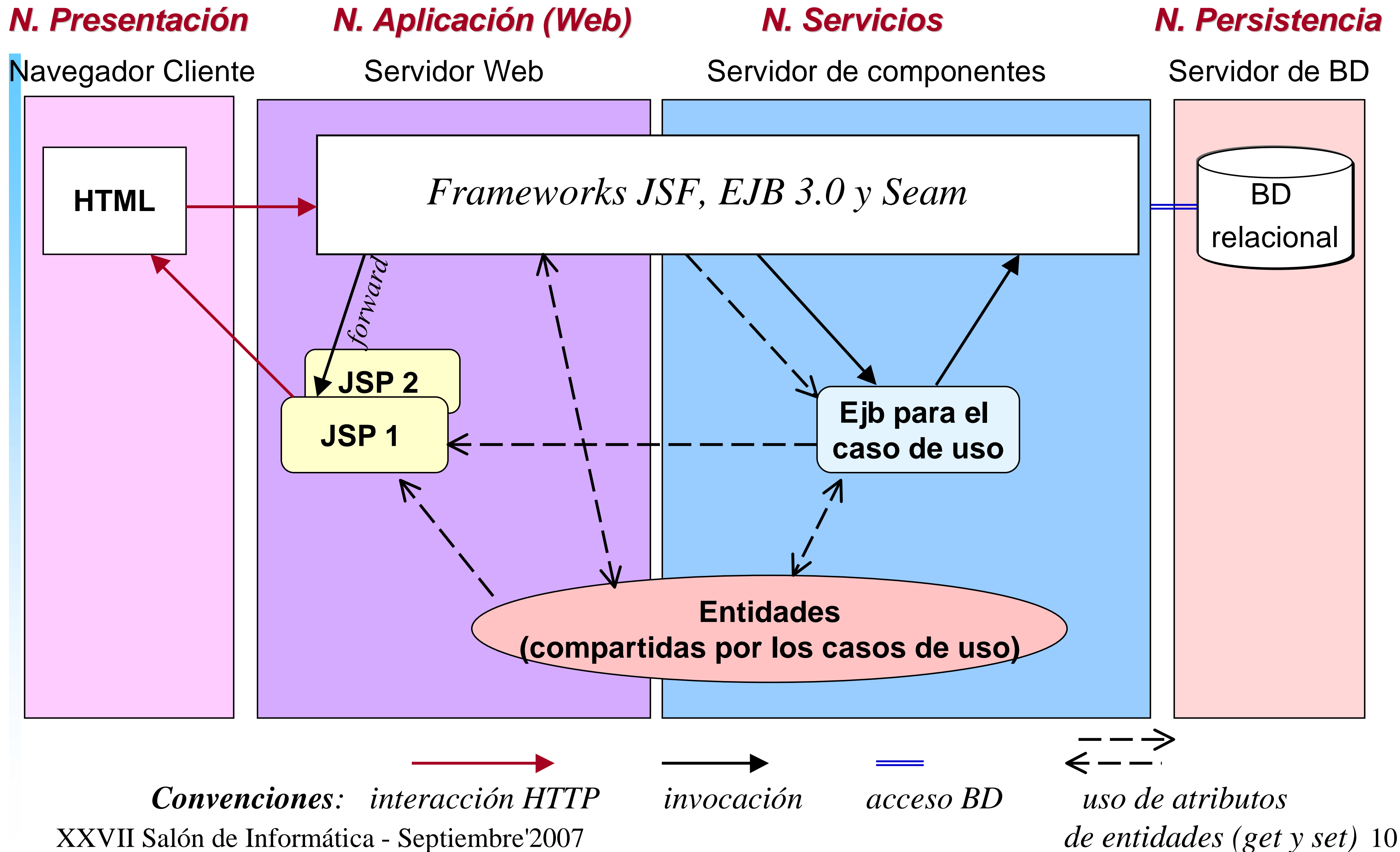
Niveles de Servicio y Dominio basados en el framework EJB 3.0

- Concepto de EJB de entidad
  - modela una entidad de negocio persistente
- Concepto de EJB de sesión
  - componente de negocio: implementa servicios ofrecidos por la aplicación
- Simplificación de Java EE 5 respecto a J2EE
  - clases simples para implementar entidades y componentes de negocio:
    - se regresa a un ***POJO***: "Plain Old Java Objects"
  - descriptores ya no son obligatorios (pero ahora hay ***@anotaciones*** no obligatorias)
    - más trabajo para el Contenedor, menos para el desarrollador
  - persistencia automática: desaparece SQL burocrático
  - ahora sí se programa en términos de objetos (entidades) y no en términos de tablas de la BD => gran reducción de código y una mayor robustez
- Framework EJB 3.0 recoge experiencias de **JDO**, **HIBERNATE**, **TOPLINK**, **SPRING**

Acople entre niveles utilizando el framework JBoss Seam

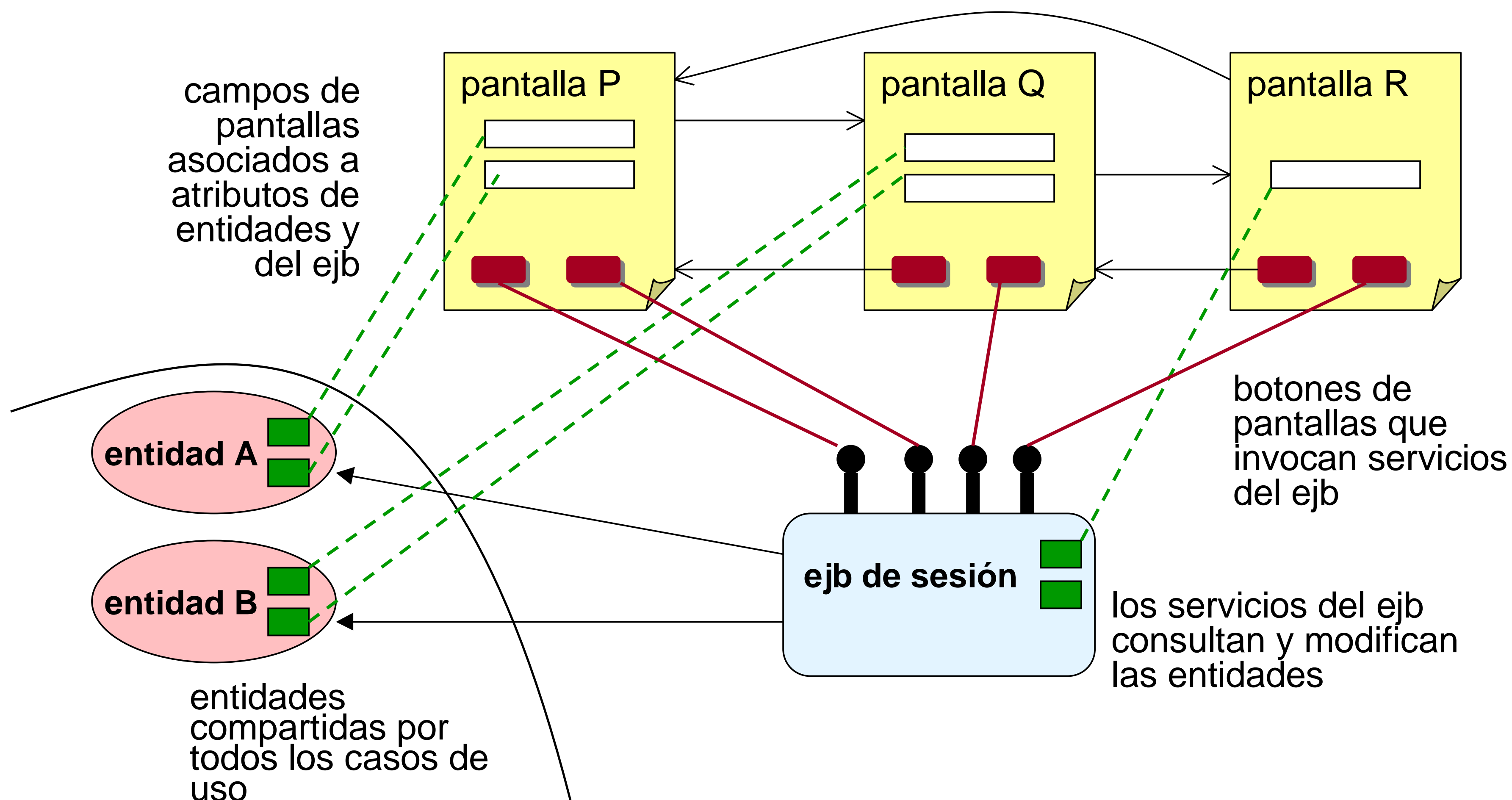
- Seam logra que las entidades estén asociadas directamente a las pantallas
  - las pantallas muestran valores de atributos de entidades
  - el usuario modifica o suministra valores para estos atributos
- Procesamiento de eventos de pantallas es realizado directamente por los componentes EJB de sesión
- Se eliminan intermediarios => muchos patrones se vuelven innecesarios
- Portabilidad de Seam a cualquier servidor que soporte JSF y EJB 3.0
- Seam fue aprobado por unanimidad en Junio 2006 como parte de la próxima versión del estándar Java EE (especificación "**Web Beans**")

Java EE 5: Arquitectura para soportar un caso de uso

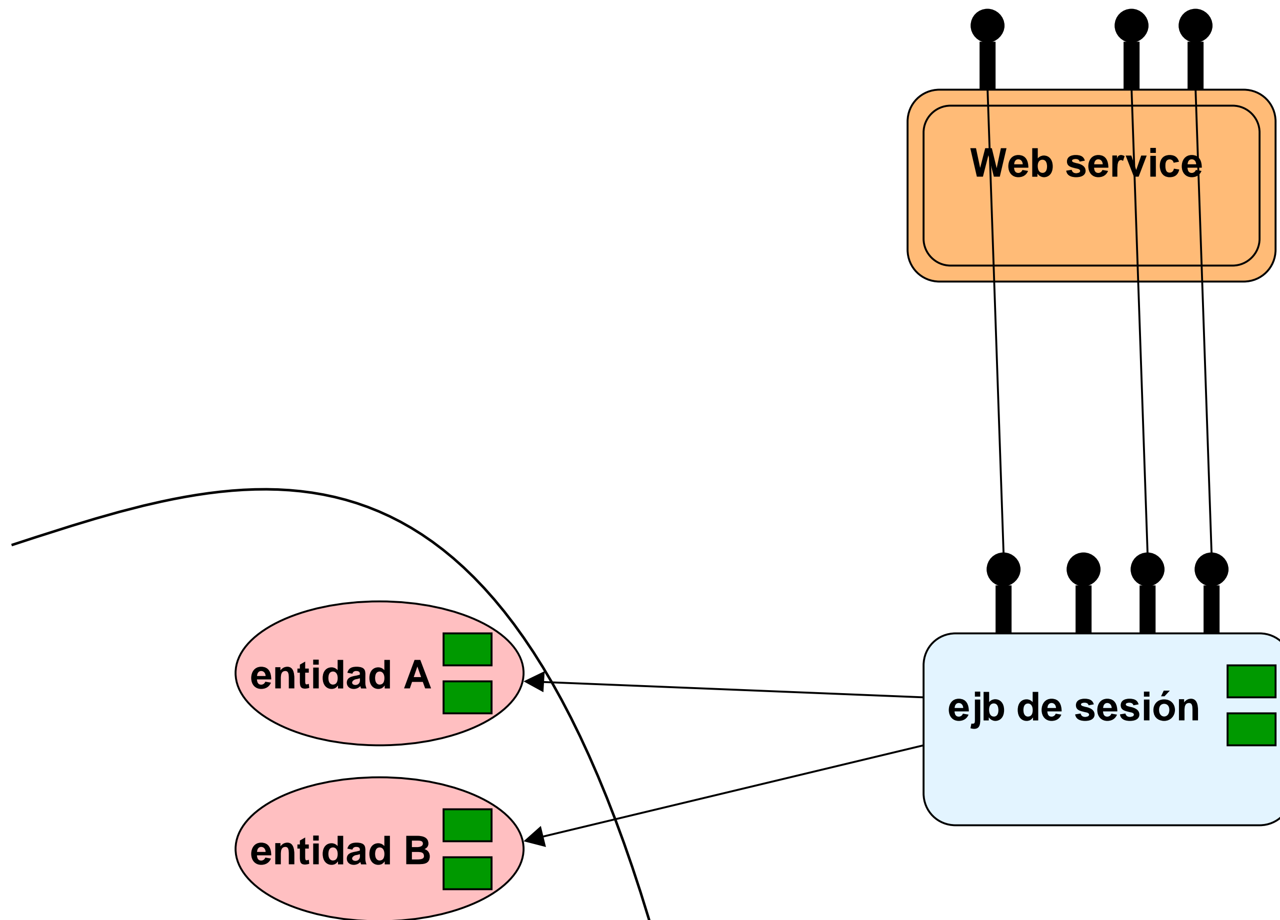


Concepto de un caso de uso

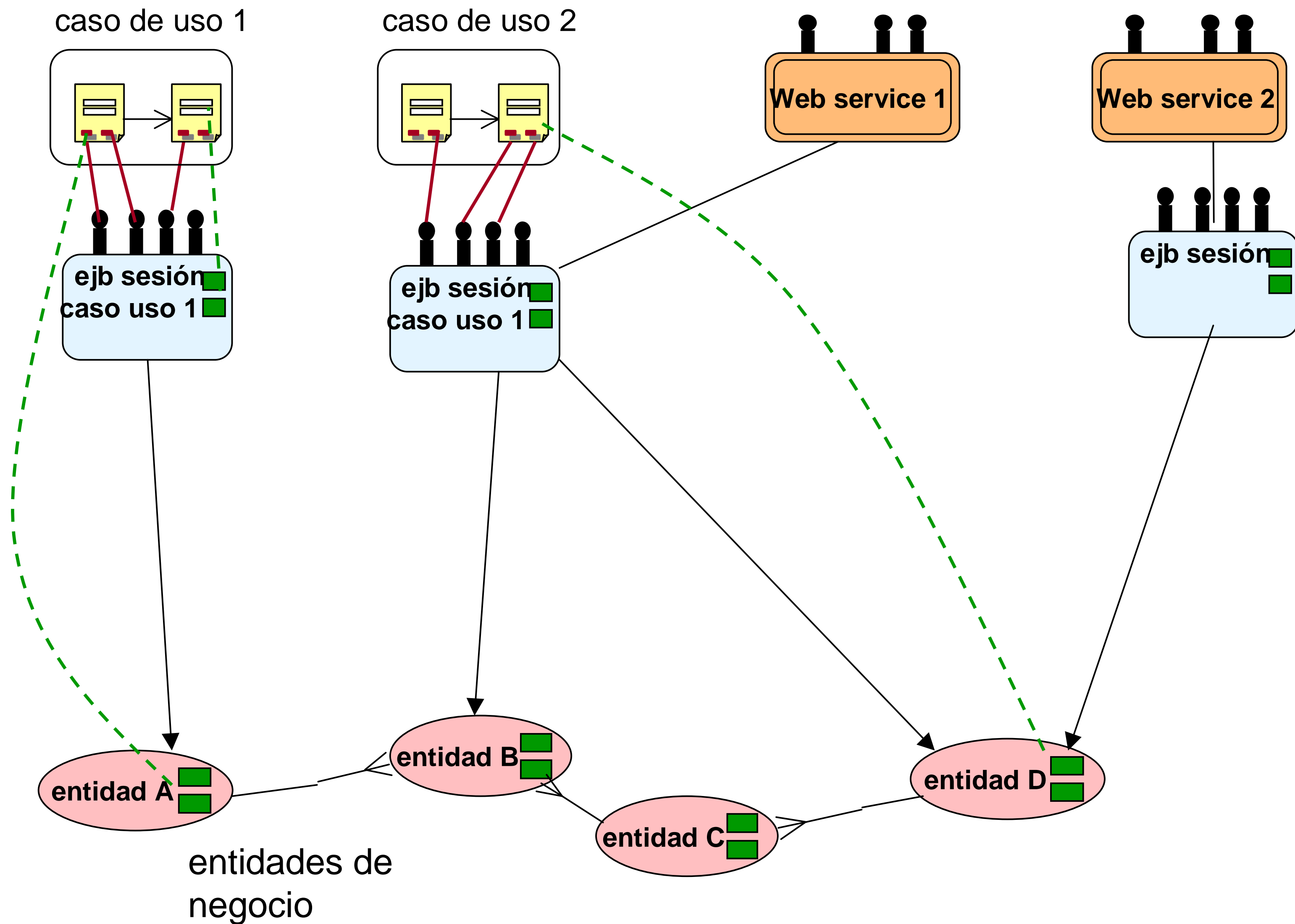
- *Def: una capacidad del sistema para entregar un resultado útil e indivisible al usuario*
- Conjunto de servicios (acciones) sobre entidades de negocio, implementado por un ejb de sesión y por un conjunto de pantallas desde donde se solicitan estos servicios



- Un ejb de sesión que soporta un caso de uso puede exponerse como Web service ofreciendo públicamente un subconjunto de sus servicios

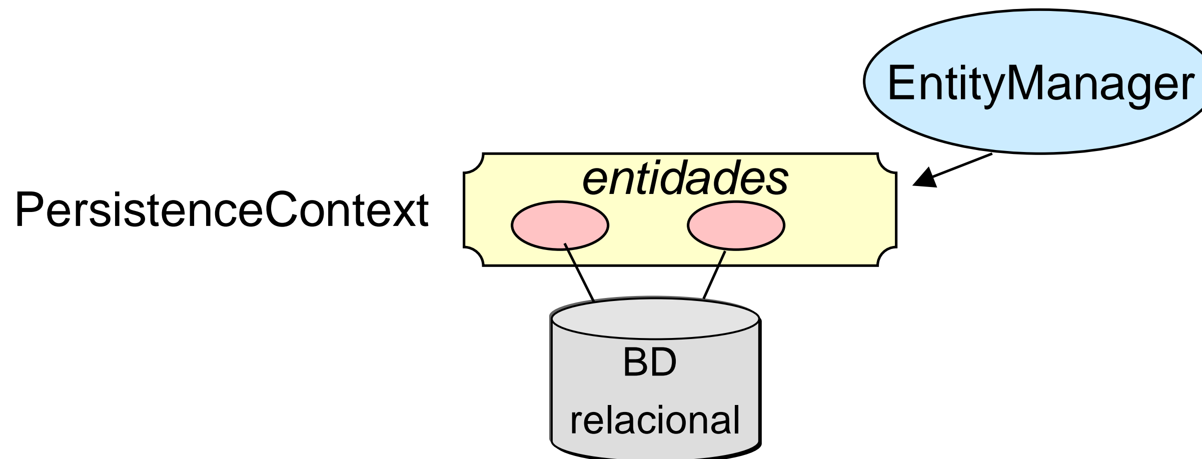


Aplicación en términos de procesos = entidades + ejbs + web services + pantallas



EntityManager: Manejador de entidades persistentes ofrecido por el framework EJB3.0

- El EntityManager maneja la persistencia de entidades de un contexto de persistencia



- Obtener la referencia al EntityManager, por inyección:
  - Obtener la referencia al EntityManager por inyección:

```
@PersistenceContext
private EntityManager em;
```

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Servicios de persistencia del EntityManager

- Servicios para obtener entidades manejadas:
  - persist: *insert* BD de una nueva entidad :
```
em.persist(empleado); // el objeto adquiere PK
```
  - find: *select* BD de una entidad existente, conociendo su PK:
```
Empleado empleado= em.find(Empleado.class, valorId);
// retorna null si no lo encuentra
```
  - merge: mezcla para una entidad de su versión en memoria con su versión en la BD, prevaleciendo su versión en memoria:
```
empleado = em.merge(empleado);
```
- Servicios sobre entidades manejadas:
  - refresh: *select* BD de una entidad existente, cancelando posibles modificaciones en memoria durante la transacción
```
em.refresh(empleado);
```
  - remove: *delete* BD de una entidad existente:
```
em.remove(empleado);
```
  - flush: aplicar a la BD modificaciones de entidades, acumuladas durante la transacción (pone candados y se somete al commit/rollback):
```
em.flush();
//efecto visto por siguiente consulta de la trans.
//ocurre automáticamente en el commit de la trans.
```

Ilustración de caso de uso: Ingresar empleado

- Pantalla para ingresar un Empleado:

entidad

Empleado
<<PK>> id: int nombre: String salario: long

```
<s:decorate template="edit.xhtml">
   <ui:define name="label">Nombre:</ui:define>
   <h:inputText value="#{empleado.nombre}" required="true" />
</s:decorate>

<s:decorate template="edit.xhtml">
   <ui:define name="label">Salario:</ui:define>
   <h:inputText value="#{empleado.salario}" required="true" />
</s:decorate>

<div class="buttonBox">
   <h:commandButton value="Proceder" styleClass="button"
      action="#{inscripcionEmpleado.verificarNuevoEmpleado}" />
   <s:button value="Cancelar" styleClass="button"
      action="#{inscripcionEmpleado.cancelar}"   />
</div>
```

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

- Entidad maestra: empleado:

```
@Entity
@Name("empleado")
public class Empleado implements Serializable {
   private int id;
   private String nombre;
   private long salario;

   public Empleado() {}

   @Id @GeneratedValue
   public Long getId() { return id;}
   public void setId(Long id) {this.id = id;}

   @NotNull
   @Length (min = 1, max = 15, message =
    "nombre requerido de maximo {max} caracteres")
   public String getNombre() { return nombre;}
   public void setNombre(String nombre) {
      this.nombre= nombre;
   }
}
```

  - Observar:
    - nombre de la entidad en el contexto de variables
    - anotación indicando el atributo llave
    - anotación de atributo obligatorio
    - anotación de validación de un atributo
    - en lugar de los *get* se pueden anotar los atributos

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

- Ejb de sesión para el caso de uso: inscripcionEmpleado
  - Nombre del componente en el contexto de variables
  - Atributos:
    - atributos que representan los frameworks de control
    - atributo que representa entidad Empleado
    - atributo que indica el resultado de validar la nueva entidad Empleado
    - inyección/extracción de atributos (el EJB también se extrae)

```
@Stateful
@Name("inscripcionEmpleado")
public class InscripcionEmpleadoAction
             implements InscripcionEmpleado {
   @PersistenceContext
   private EntityManager em;

   @In
   private FacesMessages facesMessages;

   @In(required=false)
   @Out(required=false)
   private Empleado empleado;

   private boolean empleadoValido;
   public boolean isEmpleadoValido(){
     return empleadoValido;
   }

   @Remove
   public void destroy() {}
```

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

- Acciones transaccionales (servicios que ofrece el caso de uso):
  - acciones que inician o terminan conversación de inscribir un empleado
  - acción que valida el nuevo empleado
  - acción que ingresa el nuevo empleado a la BD (usando servicio de persistencia)

```
    @Begin
    public void nuevoEmpleado() {
      empleado = new Empleado();
    }

    public void verificarNuevoEmpleado() {
      //... chequeos que involucran varios campos ...

      empleadoValido = true;  // o false
    }

    @End
    public void confirmar() {
      em.persist(empleado); // INSERTs en EMPLEADO y TELEFONO
      facesMessages.add("Gracias, su inscripcion "
      + " de #{empleado.nombre} tiene id #{empleado.id}");
    }

    @End
    public void cancelar(){}
}
```

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Facilidades de JPQL para navegar sobre entidades persistentes

(*Java Persistence Query Language*)

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Ventajas de JPQL respecto a SQL

- JPQL = Java Persistence Query Language
  - es un lenguaje para consultar o modificar entidades y no tablas
  - es una evolución de EJB QL
- Ofrece portabilidad
  - se traduce a la mayoría de dialectos SQL eliminando la dependencia de un motor de base de datos específico
- Permite manipular entidades sin tener que conocer su implementación como tablas
  - diseño OBJETUAL del subsistema Modelo de la aplicación, definiendo las entidades de negocio y sus relaciones
  - un solo diseño del Modelo en términos de entidades volviendo innecesario el diseño del Modelo en términos de tablas
- Mayor robustez de las aplicaciones
  - el framework de persistencia asegura la transformación objeto-relacional y no la aplicación

Consultas o actualizaciones expresadas en JPQL (Java Persistence Query Language)

- createQuery(): consulta o actualización de entidades según diversos criterios:
  - *select* una entidad:

```
Empleado empleado = (Empleado )em.createQuery
("select e from Empleado e where e.nombre =:nombre")
    .setParameter("nombre", "Juan Porras")
    .getSingleResult();
```

  - *select* una lista de entidades:

```
List<Empleado> empleados = em.createQuery
    ( " select e from Empleado e   " +
      " where e.salario > :salario")
    .setParameter("salario ", 500000)
    .getResultList();
```

  - *update* una o varias entidades:

```
  em.createQuery
     ( " update Empleado e                     " +
       " set e.salario = e.salario + :aumento ")
     .setParameter("aumento", 15000).executeUpdate();
```

  - *delete* una o varias entidades:

```
  em.createQuery
    ("delete from Empleado e where e.salario  < 0")
    .executeUpdate();
```

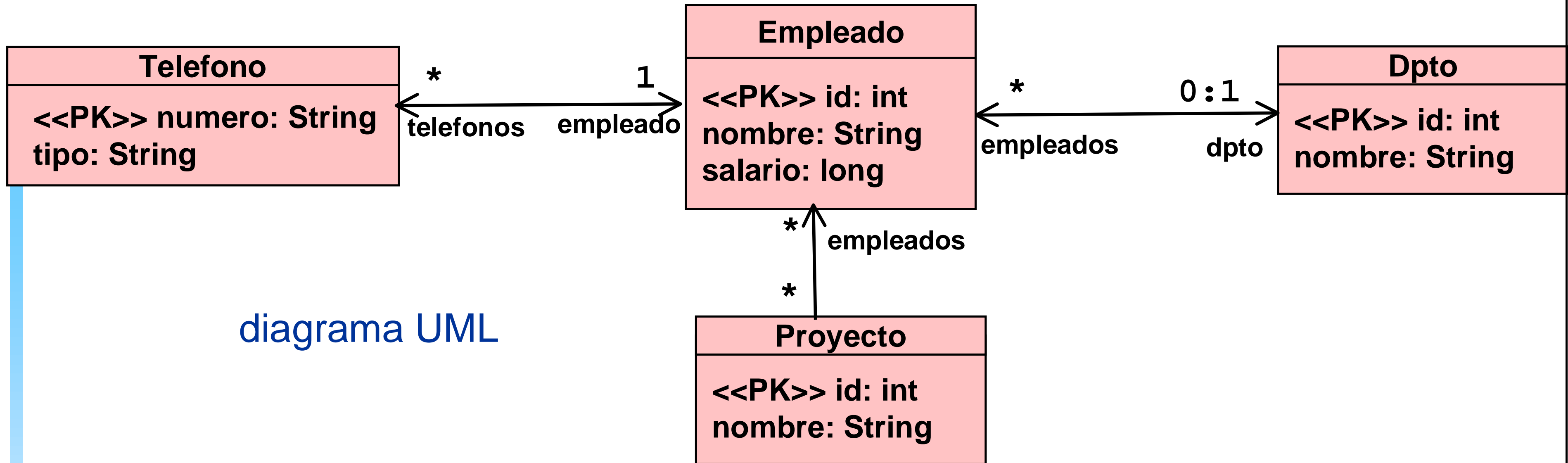
- Consulta con paginación:

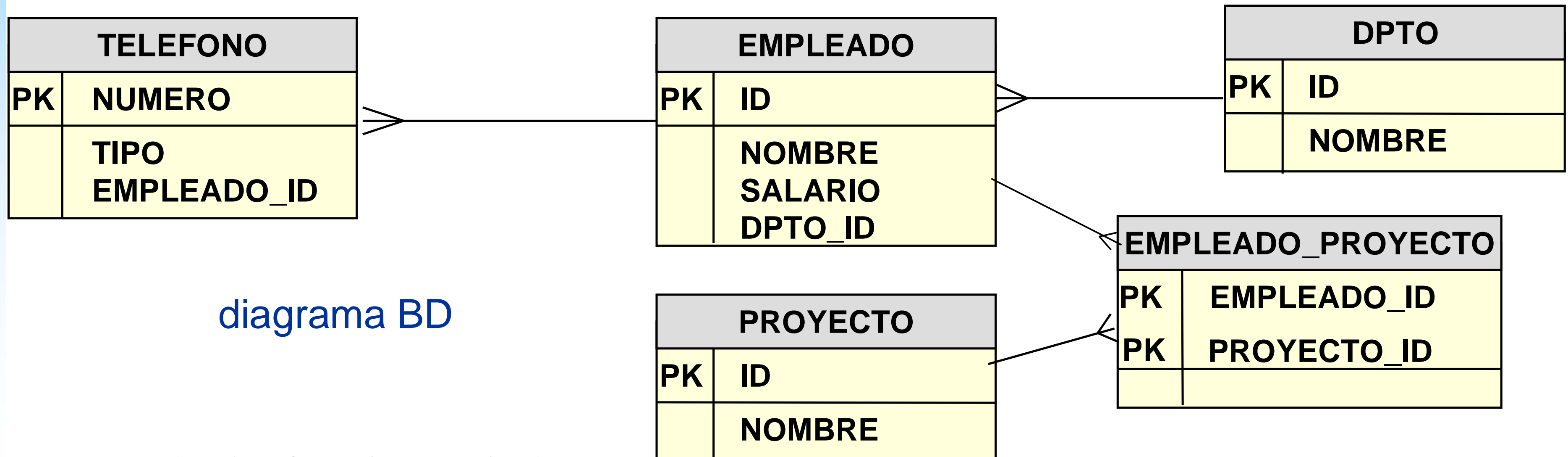
```
@Out
private List<Empleado> empleados;
...
public void findEmpleados() {
   empleados = em.createQuery
    ("select e from Empleado e        " +
     "where lower(name) like :search "   )
      .setParameter("search", searchPattern)
      .setFirstResult( page * pageSize )
      .setMaxResults(pageSize)
      .getResultList();
}
```

  - la lista obtenida puede ser expuesta directamente en una pantalla para mostrarla en una tabla con navegación por lotes:

```
    <h:dataTable var="emp" value="#{empleados}" >
       ...
    </h:dataTable>
```

Lo importante es el Modelo de entidades y no el Modelo de tablas





ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

- Navegación entre entidades recorre el Modelo de entidades
  - ej.: obtener la lista de nombres de los dptos en donde hay empleados:
    ```
    select DISTINCT e.dpto.nombre
    from Empleado e
    ```
    - navega de la entidad Empleado hacia la entidad Dpto asociado
    - join implícito entre las dos entidades
  - ej.: obtener lista de los dptos en donde hay empleados vinculados a proyectos:
    ```
    select DISTINCT d
    from Proyecto p JOIN p.empleados e JOIN e.dpto d
    ```
    - número arbitrario de entidades combinadas por joins
    - expresión clara y concisa comparada con su equivalente en SQL
- Expresiones de navegación se construyen con un número variable de elementos
  - navegación posible en el SELECT, FROM o WHERE
  - tipo de resultado de una expresión de navegación:
    - atributo de estado (ej: `e.dpto.nombre`)
    - asociacion univaluada: entidad (`e.dpto`)
    - asociación multi-valuada: colección (`p.empleados`) (restricción: no puede ir en el SELECT)

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

- Filtro de resultados
  - en el WHERE pueden usarse los mismos operadores de SQL pero aplicados a entidades:
    AND, OR,..., IN, LIKE, BETWEEN, LIKE,..., IS EMPTY, MEMBER OF
    - ejemplo: obtener el empleado asociado a un determinado teléfono:

```
select e
from Empleado e
where :telefono MEMBER OF e.telefonos
```

  - funciones disponibles: **abs, size, lower, substring, length, concat, current_date, ...**
  - Subqueries en el WHERE:
    - ejemplo: obtener todos los empleados que tienen celular:

```
select e
from Empleado e
where EXISTS (select t
              from e.telefonos t
              where t.tipo = 'celular')
```

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Definición de consultas con nombre

- Una consulta frecuente sobre una entidad debería definirse con nombre en la clase que define la entidad, por ej:

```
@Entity
@NamedQuery(name="findEmpleados",
 query="select e from Empleado e order by e.nombre")
public class Empleado implements Serializable {
 ...
}
```

- Uso de una consulta con nombre (por ej, desde EJB de sesión):

```
private List<Empleados> empleados;
...
public void getEmpleados() {
  empleados = em.createNamedQuery("findEmpleados")
                 .getResultList();
}
```

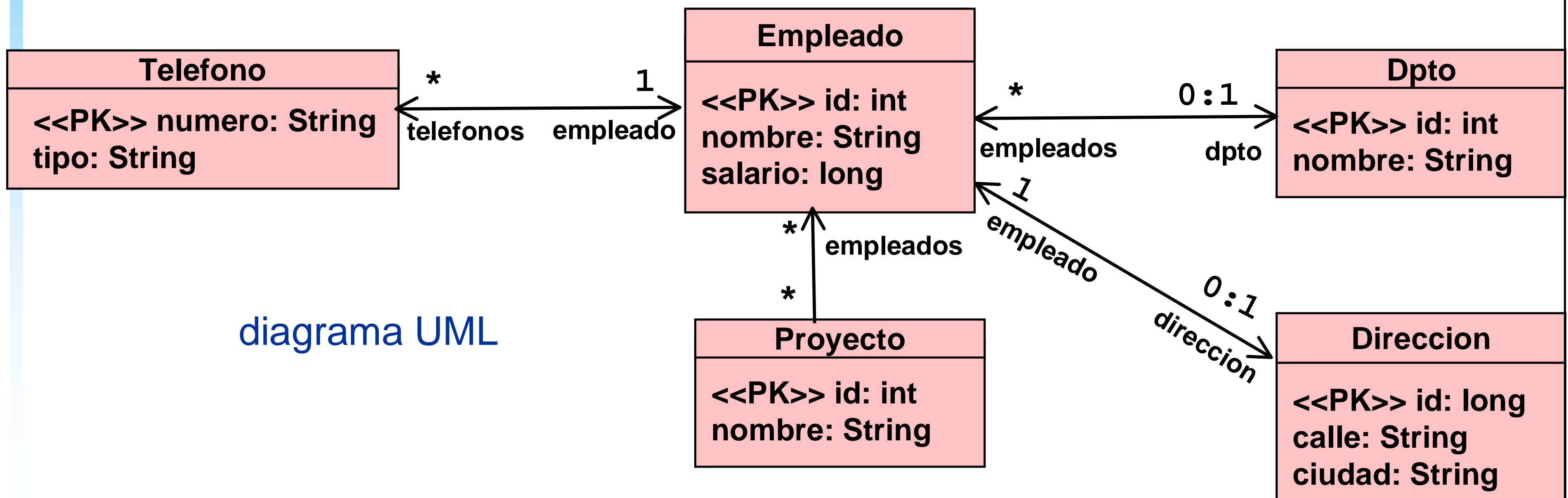
- Ventajas de las consultas con nombre:
  - definir una sola vez y en un solo sitio la consulta frecuente
  - es precompilada => mayor eficiencia
  - seguridad ante intrusos: la consulta no puede ser alterada en ejecución

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Manejo de relaciones entre entidades

Características de una relación

- Cardinalidad y direccionalidad
  - cardinalidad: cuantas instancias pueden existir en cada lado de una relación entre dos entidades: 0..1, 1, *
  - direccionalidad: cómo se navega entre dos entidades que mantienen una relación:
    - unidireccional: si solo una entidad referencia a la otra
    - bidireccional: si cada entidad referencia a la otra
- Ejemplo: entidades de una empresa



ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Anotaciones para expresar relaciones

- A nivel atributo
  - **@OneToMany** : atributo que es lista de detalles: indica que la entidad mantiene con este atributo una relación maestro-detalles
  - **@ManyToOne** : en un atributo de una entidad detalle, indica que el atributo representa a la entidad maestra
  - **@OneToOne**: atributo que representa entidad con la cual se mantiene relación 1-1
  - **@ManyToMany**: atributo lista de entidades con las cuales se mantiene relación N-N
- Opcionalmente
  - **(mappedBy="atribMaestro",cascade=CascadeType.ALL)** : indica atributo maestro en la entidad detalle y si hay efecto cascada
  - **@OrderBy("description asc")** : para un atributo lista de detalles indica el orden en que deben ser obtenidos sus miembros

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Ilustración de relaciones

- Entidad Empleado:
  relaciones con Telefono, Dpto, Direccion y Proyecto

```
@Entity
@Name("empleado")
public class Empleado implements Serializable {
   @Id @GeneratedValue
   private int id;
   private String nombre;
   private long salario;

   @OneToMany(mappedBy="empleado")
   private Set<Telefono> telefonos;
   @ManyToOne
   private Dpto dpto;
   @OneToOne(mappedBy="empleado")
   private Direccion direccion;
   ...
}
```

   - Empleado es la entidad ***inversa*** en su relaciones con Telefono y Direccion: contiene el elemento mappedBy (el FK lo implementan las otras entidades)
   - No indica relación N-N de Empleado con Proyecto porque es unidireccional

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

- Entidad Direccion: tiene relación 1-1 con Empleado y es la entidad ***propietaria*** de la relación (implementa el FK)

```
@Entity
public class Direccion implements Serializable {
   . . .
   private long   id;
   private String calle;
   private String ciudad;

   @OneToOne
   private Empleado empleado;
   ...
}
```

- Entidad Proyecto: tiene relación N-N con Empleado:

```
@Entity
public class Proyecto implements Serializable {
   . . .
   private long   id;
   private String nombre;

   @ManyToMany
   private List<Empleado> empleados;
   ...
}
```

Evaluación temprana o perezosa

- Evaluación temprana (***EAGER***) o perezosa (***LAZY***) de entidades relacionadas
  - **@OneToOne** y **@ManyToOne** implican por omisión evaluación temprana; se puede cambiar a perezosa, por ej en entidad Empleado:

```
@OneToOne(mappedBy="empleado", fetch=FetchType.LAZY)
private Direccion direccion;
```

    - LAZY: al obtener de la BD un empleado **e** no obtiene su direccion
    - posteriormente la expresión **e.direccion** obliga al EntityManager a obtener la direccion
  - **@OneToMany** y **@ManyToMany** implican por omisión evaluación perezosa; se puede cambiar a temprana, por ej en entidad Empleado:

```
@OneToMany(mappedBy="empleado",fetch=FetchType.EAGER)
private Set<Telefono> telefonos;
```

    - EAGER: al obtener de la BD un empleado **e** se obtienen también sus telefonos
    - si fuera LAZY solo se obtendrían los telefonos de **e** al recorrer el conjunto **e.telefonos**
  - Establecer evaluación perezosa o evaluación temprana es una decisión de diseño con impacto en el overhead y eficiencia

Efectos en cascada

- Tipos de cascada:
  - indicados generalmente en la entidad maestra para referirse a la lista de detalles: `cascade=CascadeType.XX`
    - `XX` puede ser: `PERSIST`, `REFRESH`, `REMOVE`, `MERGE`
  - para indicar que se aplican todos los tipos de cascada: `cascade=CascadeType.ALL`
  - el efecto cascada es unidireccional, generalmente del maestro hacia los detalles
    - al invocar la operación de persistencia sobre la entidad maestra se invocará automáticamente a cada instancia de la entidad detalle
    - ej: en entidad Empleado:

```
@OneToMany(mappedBy="empleado",
           cascade=CascadeType.REMOVE)
private Set<Telefono> telefonos;
```

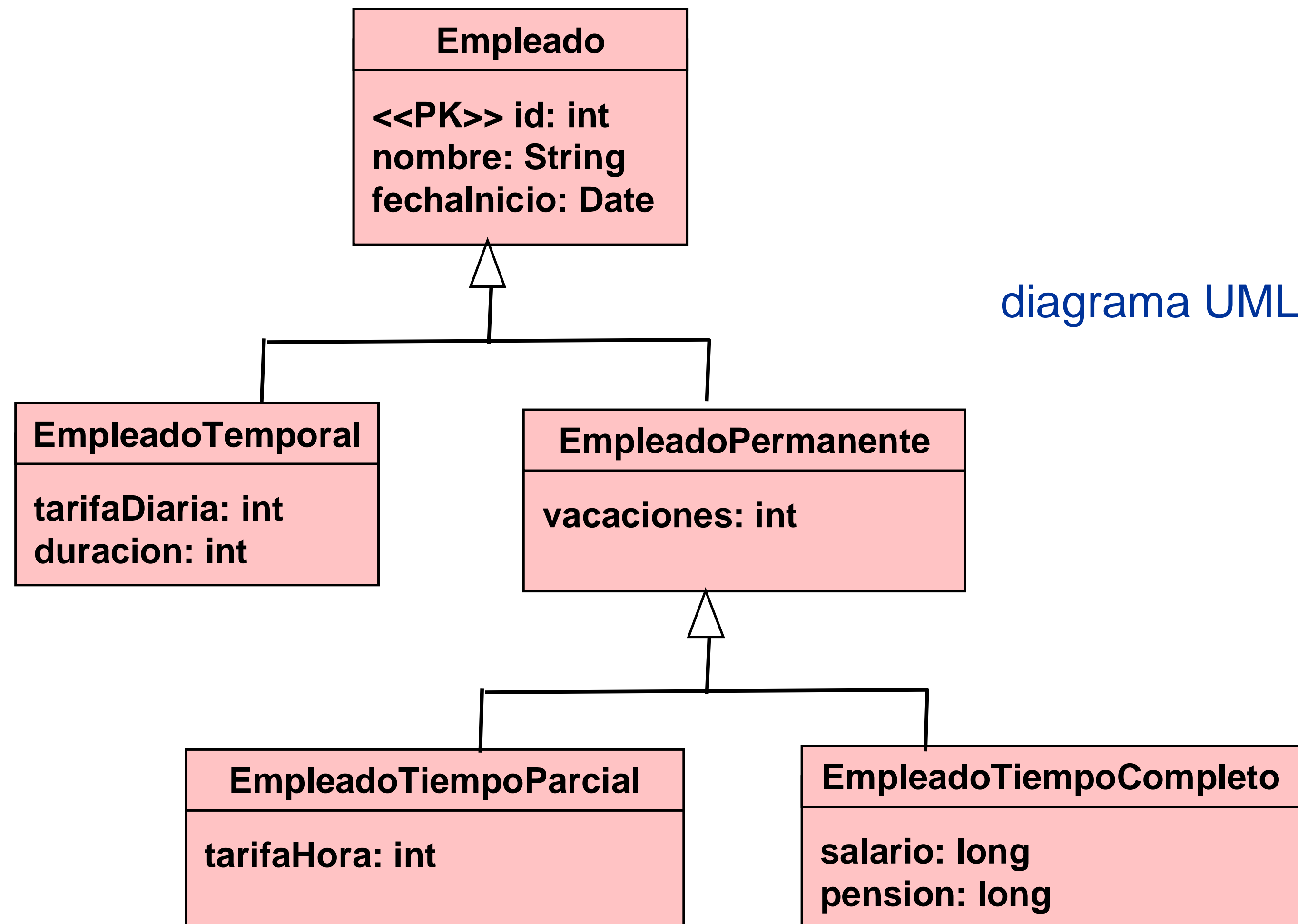
    - al invocar `em.remove(empleado);` se elimina el `empleado` y todos los `telefonos` asociados

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Herencia entre entidades

Una entidad puede heredar estado y comportamiento de otra

- la entidad que es subclase agrega sus propios atributos de estado
- ejemplo de una jerarquía de herencia entre entidades empleados



Tipos de clases en una jerarquía de herencia entre entidades

- clases terminales son **entidades concretas**
- clase no terminal es **abstracta** (no instancias) y define estado/comportamiento que heredan las subclases:
  - **entidad abstracta** que define estado persistente y que puede participar en los queries y en relaciones con entidades
  - **clase abstracta anotada** `@MappedSuperclass` : define estado persistente; no participa en queries, ni en relaciones con entidades
- definición de las clases que intervienen en la jerarquía:
  - ej:clase raiz `Empleado` es una entidad abstracta que define atributos persistentes `id,nombre,fechaInicio` de un empleado:

```
@Entity
public abstract class Empleado {
  @Id private int id;
  private String nombre;
  @Column(name="S_DATE")
  private Date fechaInicio;
  // ...
}
```

  - ej consulta: `select e from Empleado e where e.fechaInicio > :fecha`

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

- clase terminal **EmpleadoTemporal** representa empleados por contrato temporal: es una entidad concreta que extiende **Empleado**

```
@Entity
public class EmpleadoTemporal extends Empleado{
  @Column(name="D_RATE")
  private int tarifaDiaria;
  private int duracion;
  // ...
}
```

- clase no terminal **EmpleadoPermanente** representa empleados vinculados a término indefinido: es una clase abstracta anotada **@MappedSuperclass** que define el atributo persistente **vacaciones:**

```
@MappedSuperclass
public abstract class EmpleadoPermanente
                                extends Empleado{
  private int vacaciones;
  // ...
}
```

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

- clases terminales **EmpleadoTiempoParcial** y **EmpleadoTiempoCompleto** representan empleados a término indefinido de tiempo parcial o tiempo completo: son entidades concretas que extienden **EmpleadoPermanente**

```
@Entity
public class EmpleadoTiempoParcial
                    extends EmpleadoPermanente {
   @Column(name="H_RATE")
   private int tarifaHora;
   // ...
}
```

```
@Entity
public class EmpleadoTiempoCompleto
                    extends EmpleadoPermanente {
   private long salario;
   private long pension;
   // ...
}
```

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Implantación de herencia en la BD mediante la estrategia "Single Table"

- Una sola tabla para guardar instancias de todas las entidades concretas, diferenciándolas por columna "discriminadora"
  - requiere llave primaria definida una sola vez en la entidad raiz
  - solución eficiente porque no requiere join
  - cada entidad usa solo columnas correspondientes a sus atributos, (desperdicio de memoria)
- Ejemplo: diagrama BD y datos
  - columna discriminadora: EMP_TIPO

EMPLEADO	
PK	ID
	NOMBRE S_DATE D_RATE DURACION VACACIONES H_RATE SALARIO PENSION EMP_TIPO

ID	NOMBRE	S_DATE	D_DATE	DURACION	VACACIONES	H_RATE	SALARIO	PENSION	EMP_TIPO
1	Juan	02012001	80000	12					Temp
2	Pablo	02052001	20000	24					Temp
3	Ana	01012002			15		1000000	600000	Completo
4	Carlos	01012004			20		800000	400000	Completo
5	Clara	01062005			20	10000			Parcial
6	Rafael	01062003			15	12000			Parcial

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

- Anotaciones requeridas:
  - En la entidad raiz indicar: estrategia implantación y columna discriminadora:

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="EMP_TIPO")
public abstract class Empleado { ... }
```

  - En c/entidad concreta indicar valor discriminador:

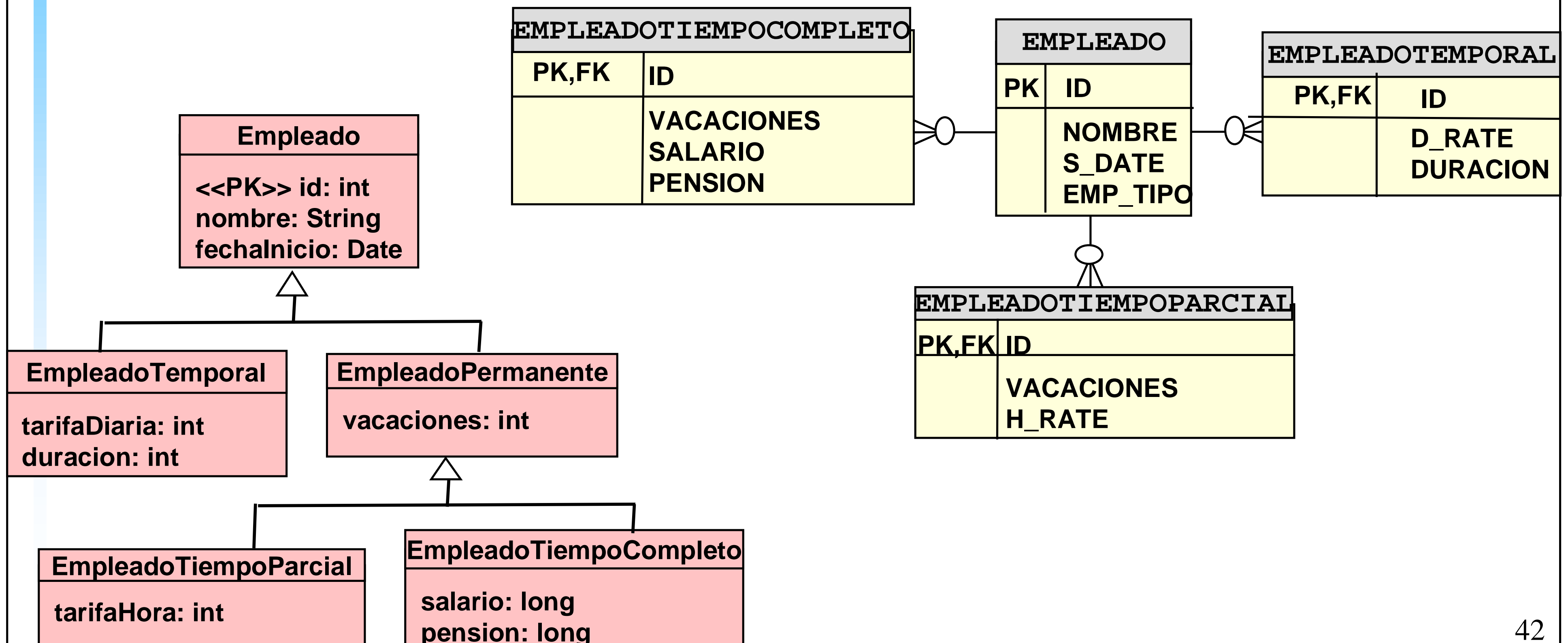
```
@Entity @DiscriminatorValue("Temp")
public class ContractEmployee extends Employee { ... }
```

```
@Entity @DiscriminatorValue("Completo")
public class FullTimeEmployee extends CompanyEmployee
{...}
```

```
@Entity @DiscriminatorValue("Parcial")
public class PartTimeEmployee extends CompanyEmployee
{...}
```

Implantación de herencia en la BD mediante la estrategia "Joined"

- Cada **entidad** de la jerarquía tiene una tabla asociada
  - refleja de manera natural la jerarquía de objetos
  - guarda los datos en forma normalizada sin desperdiciar espacio
  - una instancia de entidad se guarda en múltiples tablas
  - requiere joins para obtener una instancia de una entidad concreta
- Ejemplo: diagrama UML - diagrama BD



ACIS ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Update y Delete cuando hay herencia de entidades

- Query de delete
  - el WHERE filtra entidades que se quieren eliminar
    - ej: eliminar empleados que no están en ningún proyecto:

```
delete from Empleado e
where e.dpto IS NULL
```

  - el query de delete:
    - elimina entidades que son subclases de la entidad nombrada en el FROM y que cumplen el WHERE
    - no elimina en cascada a otras entidades aún si hay anotaciones de cascada con esas entidades
    - la eliminacíón en cascada solo ocurre con la operación:

```
em.remove(entidadMaestra);
```

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Control opcional en la transformación Objetos - Relacional

ACIS ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Creación de índices mediante anotaciones

Direccion
<<PK>> id: long calle: String ciudad: String

- Anotación de índice sobre una sola columna :

```
@Entity
public class Direccion implements Serializable {
   . . .
   private long   id;
   private String calle;

   @Index(name = "IDX_ciudad")
   private String ciudad;
   ...
}
```

- Anotación de índice sobre varias columnas

```
@Entity
@org.hibernate.annotations.Table
  (appliesTo="DIRECCION", indexes =
     {@Index(name="IDX_localizacion",
             columnNames={"ciudad", "calle"})})
public class Direccion implements Serializable {
   ...
}
```

Restricciones de unicidad

- La anotación @Column puede indicar una restricción de unicidad simple

```
@Entity
public class Empleado {
  @Id private int id;
  @Column(unique=true)
  private String nombre;
  // ...
}
```

- La anotación @Table puede indicar restricciones de unicidad simples o compuestas

```
@Entity
@Table(name="EMPLEADO", uniqueConstraints=
   @UniqueConstraint(columnNames={"NOMBRE"}))
public class Employee {
  @Id private int id;
  private String nombre;
  // ...
}
```

ACIS ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Alternativas en la generación de identificadores únicos (llave primaria)

- Generación automática con estrategia por omisión

```
@Id @GeneratedValue private int id;
```

- Generación mediante una tabla
  - tabla global creada por el proveedor:

Empleado_Gen	0
Direccion_Gen	0

  - se pueden diferenciar generadores separados por entidad:
    - entidad Empleado:

```
@TableGenerator(name="Empleado_Gen")
@Id @GeneratedValue(generator="Empleado_Gen")
private int id;
...
```

    - entidad Direccion: utiliza `"Direccion_Gen"`

- Generación mediante un secuenciador de BD
  - secuenciador global creado por el proveedor:

```
@Id @GeneratedValue(strategy=GenerationType.SEQUENCE)
public Long getId(){return id;}
```

  - se podría indicar un secuenciador específico

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Control al generar una columna

- Columna para valores decimales
  - se puede indicar la precisión y escala
  - ejemplo:

```
@Entity
public class EmpleadoTiempoParcial {
   @Column(precision=8, scale=2)
   private float tarifaHora;
   // ...
}
```

- Definición SQL de una columna
  - control de las características de la columna, ejemplo:

```
@Column(name="S_DATE",
   columnDefinition="DATE DEFAULT SYSDATE")
private java.sql.Date fechaInicio;
```

  - en la definición hay dependencia del motor de BD

Entidad implantada en múltiples tablas

- Por razones de modularidad en la BD, una entidad puede implantarse en 2 o más tablas
  - ejemplo: en el Modelo de entidades se quiere ver una entidad `Empleado` que incluye datos de dirección en lugar de modelar la entidad `Direccion`
  - entidad `Empleado` se quiere implantar en una tabla principal con los datos de `id`, `nombre`, `salario` más una tabla secundaria con los datos de dirección del empleado:

diagrama UML

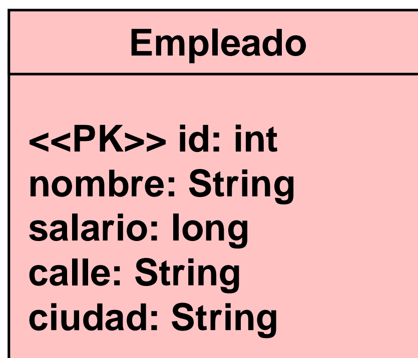
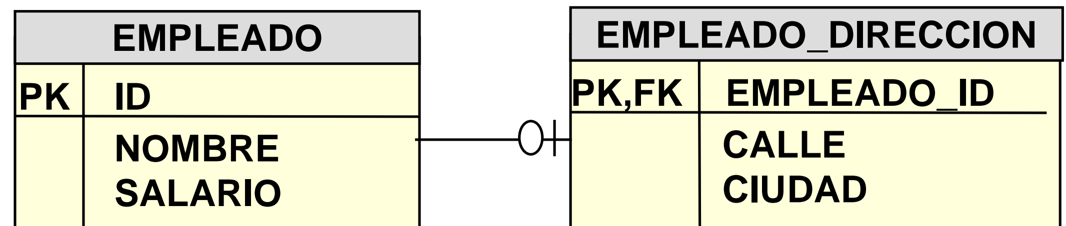


diagrama BD



ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

- Definición de la entidad **Empleado** con anotaciones para indicar tabla seundaria y atributos que van en la tabla secundaria:

```
@Entity
@SecondaryTable(name="EMPLEADO_DIRECCION",pkJoinColumns=
               @PrimaryKeyJoinColumn(name="EMPLEADO_ID"))
public class Empleado {
  @Id private int id;
  private String nombre;
  private long salario;
  @Column(table="EMPLEADO_DIRECCION")
  private String calle;
  @Column(table="EMPLEADO_DIRECCION")
  private String ciudad;
  // ...
}
```

Llaves primarias compuestas

- Requiere definir clase que represente la llave primaria
  - ej: la llave primaria de entidad **Empleado** podría estar compuesta de: país e identificador del empleado dentro de su país

diagrama UML

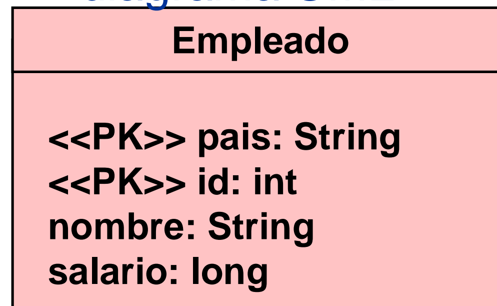


diagrama BD

  - Definición de la entidad **Empleado** que usa llave primaria compuestä:

```
@Entity
@IdClass(EmpleadoId.class)
public class Empleado {
  @Id private String pais;
  @Id  @Column(name="EMPLEADO_ID")
  private int id;

  private String nombre;
  private long salario;
   ...
}
```

- Clase que define llave primaria compuesta (**EmpleadoId**)
  - define atributos que componen la llave (ej: pais, id)
  - no contiene métodos modificadores set
  - es serializable y tiene un constructor vacio
  - debe redefinir métodos **equals**() y **hashCode**() para que el Servidor pueda indexar

- Uso de la llave primaria compuesta en la operación find:

```
em.find(Empleado.class, new EmpleadoId(pais, id));
```

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Consultas en SQL nativo

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

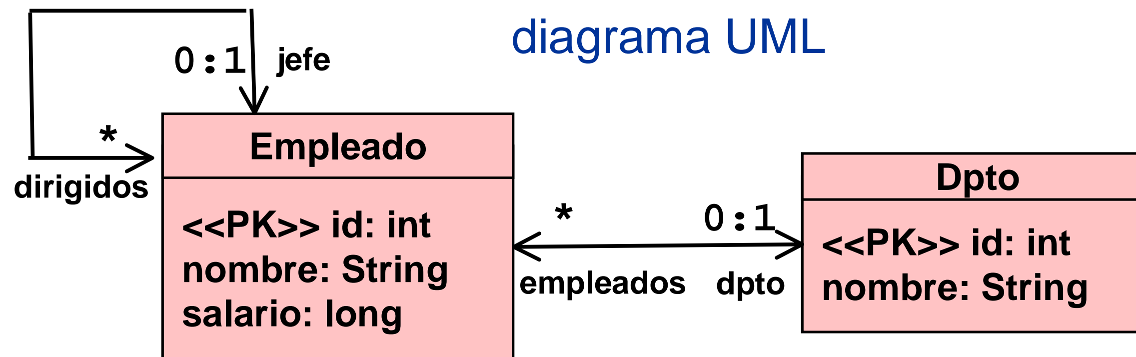
Razones para usar SQL nativo

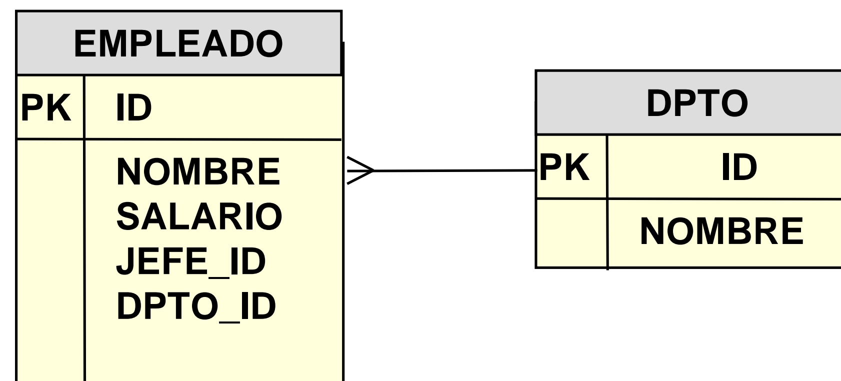
- JPQL es ideal para manipular entidades pero todavía no cubre todas las facilidades de SQL estándar:
  - JPQL no contempla:
    - subqueries definidos en el FROM
    - invocación a procedimientos almacenados
- La portabilidad de JPQL impide que cubra ciertas facilidades de un motor específico de BD
  - JPQL no contempla facilidades específicas de Oracle como:
    - consultas jerárquicas
    - expresiones funcionales para manipular valores de tiempo
- En consultas voluminosas su versión en SQL nativo puede ser más eficiente que la traducción de JPQL
- Recomendaciones:
  - usar SQL nativo solo cuando es estrictamente necesario: para no perder la portabilidad y navegabilidad que ofrece JPQL
  - cuando se decida usar SQL nativo: no usar JDBC sino "**Queries nativos**" que permiten convertir los resultados en entidades (útil para migración)

Query nativo dinámico para obtener una entidad

- Mediante la operación de persistencia `createNativeQuery` indicando tipo de entidad resultado

  - ej: encontrar todos los empleados subalternos directos o indirectos de un determinado empleado jefe:





ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

- en Oracle se aprovecha el soporte de consultas jerárquicas:

```
String subalternos =
"SELECT id,nombre,salario,jefe_id,dpto_id "+
"FROM EMPLEADO " +
"START WITH jefe_id = ? " +
"CONNECT BY PRIOR id = jefe_id";
```

- Query nativo: obtiene una lista de entidades **Empleado** (que se vuelven entidades manejadas para persistencia):

```
EntityManager em;
...
List<Empleado> = em.createNativeQuery
   (subalternos, Empleado.class)//tipo entidad resultado
   .setParameter(1, jefeId)
   .getResultList();
```

- Otras entidades asociadas se podrán evaluar después con métodos get(), por ej **getJefe()**, **getDpto()**
- Restricciones:
  - SELECT debe obtener todas las columnas de la tabla asociada a la entidad
  - parámetros solo posicionales y no parámetros con nombre
  - una entidad no puede asignarse a un parámetro

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Conclusiones

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Conclusiones

- Atractivos de Java EE 5:
  - Conciso y eficiente; reduce el código a la mitad o menos
  - Desaparece JDBC (código burocrático SQL) y varios patrones de intermediarios
  - Modelaje unificado de datos en términos de entidades persistentes y no de tablas de la base de datos (Modelaje E-R: Q.E.P.D.)
  - Programación por Objetos y no procedimental
  - Pantallas JSF con elementos ricos de interfaz y relativa facilidad de uso
  - Integración de múltiples frameworks que ofrecen muchas facilidades: persistencia, seguridad, logs, internacionalización, BPM, *testing*, tareas asincrónicas, *web services*, ...

- Dificultades planteadas por Java EE 5:
  - Comenzar de nuevo ….
  - El manejo de la persistencia de entidades es delicado
  - Pérdida del control a un nivel fino de detalle porque los frameworks actúan como cajas negras => dificultad de depurar
  - Solo recientemente empezó a ser soportado por los proveedores de la tecnología Java
- Disponibilidad de herramientas generadoras:
  - del modelo UML de entidades generan entidades Java (y viceversa)
  - del modelo UML de entidades generan su representación en XML adecuada para participar en *WebServices*
  - de la base de datos generan entidades Java
  - ...
- Modelaje de entidades persistentes también disponible para .NET: framework Nhibernate

ACIS
ASOCIACIÓN COLOMBIANA DE INGENIEROS DE SISTEMAS

Referencias

- Libros sobre Java EE 5:
  - “Pro EJB 3: Java Persistence API”, Apress, 2006.
  - “JavaServer Faces: The Complete Reference”, Mc Graw Hill, 2007.

- Fuentes en internet:
  - http://java.sun.com/javaee/5/docs/tutorial/doc: "The Java EE 5 Tutorial" es el tutorial oficial de Sun (versiones HTML y PDF)
  - http://java.sun.com/javaee/technologies/javaee5.jsp: “Get Started with the Java EE 5 SDK": son tutoriales y demos para comenzar
  - http://java.sun.com/reference/blueprints: Patrones y guías para el desarrollo de aplicaciones Java EE en general
  - http://labs.jboss.com/portal/jbossseam/docs: Manuales de referencia de JBoss Seam