Object Relational Mapping de Java

- Java La especificación de interfaz de programación Persistence API o **JPA**
- correspondencia entre objetos y bases de datos relacionales
- “O/R mapping” (“object–relational mapping”) u ORM en la literatura.
- Trabajo de grupo experto JSR 220 del denominado *Java Community Process*.
- JPA 2.0 ha sido el resultado del trabajo del grupo experto JSR 317.

Características de JPA

La persistencia que nos proporciona JPA cubre tres importantes ámbitos:

- el propio API definido en el paquete `javax.persistence` del lenguaje Java
- el lenguaje de consulta *Java Persistence Query Language* (JPQL)
- los metadatos de objetos y relacionales

Características de JPA II

- JPA puede crear automáticamente tablas de una base de datos, directamente a partir de las relaciones entre los *objetos de negocio*.
- JPA automáticamente realiza la correspondencia entre los objetos del diseño de la aplicación y las filas de una tabla de una base de datos relacional.
- JPA puede realizar automáticamente uniones ("joins") para satisfacer las relaciones entre los objetos anteriores.
- JPA se ejecuta encima de JDBC y, por tanto, es compatible con cualquier base de datos que posea un driver JDBC.
- El uso de JPA evita al programador de aplicaciones Java tener que escribir código JDBC y SQL.

Anotaciones que propone JPA para las *clases de negocio*

- Una clase Java anotada con `javax.persistence.entity` pasa a ser una *entidad*
- Anotación `@GeneratedValue`, permite generar una clave primaria en la base de datos.
- Anotación: `@Table(name="NEWTABLENAME")`
- Los campos pertenecientes a una *entidad* se han de salvar en la base de datos.
- JPA puede utilizar variables de instancia de las clases o los métodos `getter` y `setter` pero nunca mezclarlos
- Por defecto, JPA convierte en persistentes a todos los campos de una *entidad*: usa `@Transient` para deshacer

Anotaciones JPA relacionadas con la persistencia de campos

`@Id`	Identifica el único ID de la entrada en la base de datos
`@GeneratedValue`	Junto con ID sirve para definir que este valor se genera
`@Transient`	Este campo no será salvado en la base de datos

Relaciones entre entidades

- JPA permite definir relaciones entre clases
- Relaciones del tipo: uno–a–uno, muchos–a–uno y muchos–a–muchos.
- RElaciones unidireccionales y bidireccionales
- En una relación *bidireccional* necesitamos especificar el lado propietario de dicha relación (`@ManyToMany(mappedBy=`“atributoDeLaClaseAdquirida”))

Anotaciones relacionadas con relaciones entre entidades

- `@OneToOne`
- `@OneToMany`
- `@ManyToOne`
- `@ManyToMany`

Ejemplo

```
1  @Entity
2  public class Familia {
3  @Id
4  @GeneratedValue(strategy = GenerationType.TABLE)
5  private int id;
6  private String descripcion;
7  @OneToMany(mappedBy = "familia")
8  private final List<Persona> miembros= new ArrayList<Persona>();
9  public int getId() {return id;}
10 public void setId(int id) { this.id = id;}
11 public String getDescripcion() {return descripcion;}
12 public void setDescripcion(String descripcion){
13           this.descripcion = descripcion;}
14 public List<Persona> getMiembros() {return miembros;}
15 public void setMiembros(Persona persona){miembros.add(persona);}
16 }
```

La entidad `Persona`

```
1  @Entity
2  public class Persona {
3  @Id
4  @GeneratedValue(strategy = GenerationType.TABLE)
5  private String id; private String nombre;
6  private String apellidos; private Familia familia;
7  private List<Empleo> listaEmpleos = new ArrayList<Empleo>();
8  public String getId() {return id;}
9  public void setId(String Id) {this.id = Id;}
10    ...
11 @ManyToOne public Familia getFamilia() {return familia;}
12 public void setFamilia(Familia familia){this.familia=familia}
13 @Transient public String getCampoVacio() {return campoVacio;}
14    public void setCampoVacio(String campoVacio) {
15                this.campoVacio = campoVacio;}
16 @OneToMany public List<Empleo> listaEmpleos(){
17                return this.listaEmpleos;}
18 public void setListaEmpleos(List<Empleo> nombre){
19            this.listaEmpleos = nombre;}
20 }
```

La entidad Empleo

```
1  @Entity
2  public class Empleo {
3  @Id
4  @GeneratedValue(strategy = GenerationType.TABLE)
5  private int id; private double salario;
6  private String descripcionTrabajo;
7  public int getId() {return id;}
8  public void setId(int id){this.id = id;}
9  public double getSalario(){return salario;}
10 public void setSalario(double salario) {
11             this.salario = salario;}
12 public String getDescripcionTrabajo() {
13             return descripcionTrabajo;}
14 public void setDescripcionTrabajo(String descripcionTrabajo) {
15          this.descripcionTrabajo = descripcionTrabajo;
16    }
17 }
```

Modelado

Enunciado del problema

Utilizando JPA, se pide programar una aplicación para crear Listas de Correo que utilizará un canal (`DBUsuario`) para escribir los datos de los usuarios de una Lista de Correo en una base de datos relacional.

* La aplicación ha de utilizar un "connection pool" para permitir conectar rápidamente las hebras de usuarios a la base de datos a través de un Servlet (del tipo `HttpServlet` de Java).
* Ejecutar la aplicación como un proyecto Java de Eclipse y utilizarla para añadir usuarios a la lista de correo.
* Crear una aplicación de "*Administración de Usuarios*", que permita: visualizar todos los usuarios incluídos en la BD, actualizar los usuarios existentes y eliminar los usuarios almacenados en la mencionada tabla *usuario*.

Utilizar *Squirrel* o una herramienta similar para ver las tablas de la BD y comprobar que incluye una tabla llamada "usuario", con columnas que se corresponderán con los campos de la clase `Usuario`.

Diseño arquitectónico de la aplicación

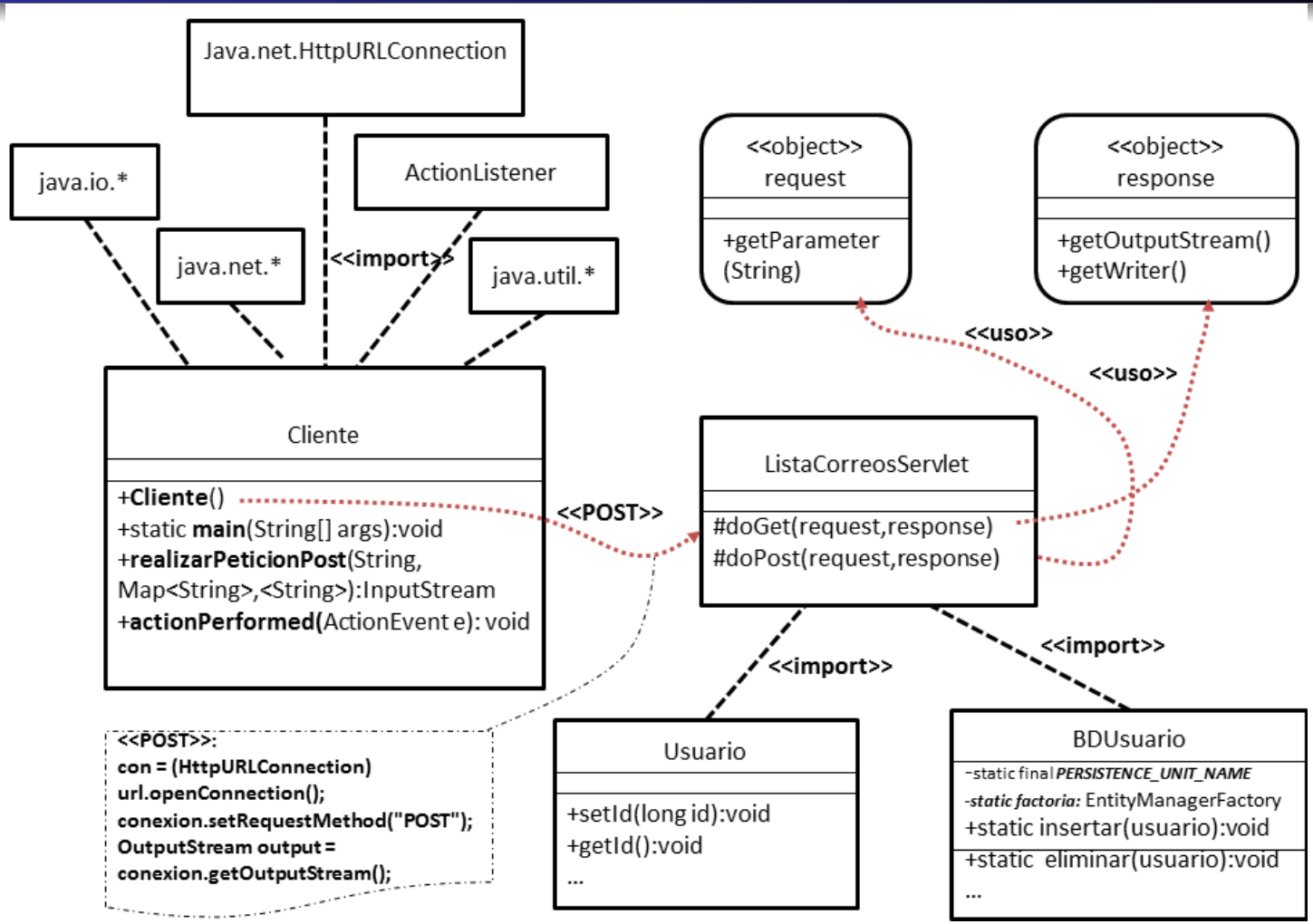


Figura: Componentes y conectores de la aplicación *Administración de Usuarios*

Partes de la aplicación

- package modelo
  - Clase `Usuario`
  - Clase `BDUsuario`
- package comunicacion
  - Clase ListaCorreosServlet
- package interfaz
  - Clase principal `Cliente`
  - Clase `TablaUsuarios`
  - Clase `ButtonColumn`

Package modelo

```
1  import javax.persistence.EntityManager;... EntityTransaction;
2  import javax.persistence.NoResultException;
3  import javax.persistence.TypedQuery; import Usuario;
4  public class BDUsuario {
5  private static final String PERSISTENCE_UNIT_NAME = "usuario";
6  private static EntityManagerFactory factoria = Persistence.
       createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
7  //Insertar un usuario; si ya existe, no tiene efecto.
8      public static void insertar(Usuario usuario) { //...}
9      //Actualizar los datos de un usuario en la base de datos
10     public static void actualizar(Usuario usuario) { //... }
11 //Eliminar un usuario de la base de datos
12     public static void eliminar(Usuario usuario) { //... }
13 //Recuperar un usuario desde la base de datos
14     public static User seleccionarUsuario(String email) {//...}
15     //Comprobar que existe el usuario cuyo email pasamos como
           argumento
16     public static boolean existeEmail(String email) { //...}
17     //Listar los usuarios de la base de datos
18     public static List<Usuario> listarUsuarios(){//...}
19 }
```

Package modelo

```
1  import java.io.Serializable;import javax.persistence.Id;
2  import javax.persistence.Entity;...GeneratedValue;...
       GenerationType;
3  @Entity
4  public class Usuario implements Serializable {
5  private static final long serialVersionUID = 1L;
6  @Id
7  @GeneratedValue(strategy = GenerationType.TABLE)
8    private long id; //Identificador numerico del usuario
9    private String nombre; //nombre del usuario
10   private String apellido;//appellidos del usuario
11   private String email;//su direccion de correo electronico
12   public Usuario() {}//Para construir un usuario vacio
13   public Usuario(Usuario us){ ...}
14   public long getId(){...}
15   public void setId(long id){... }
16   //... @Override
17   public String toString() {
18     return nombre + "␣" + apellido + "␣-␣" + email;}
19 }
```

Implementación de `ListaCorreosServlet`

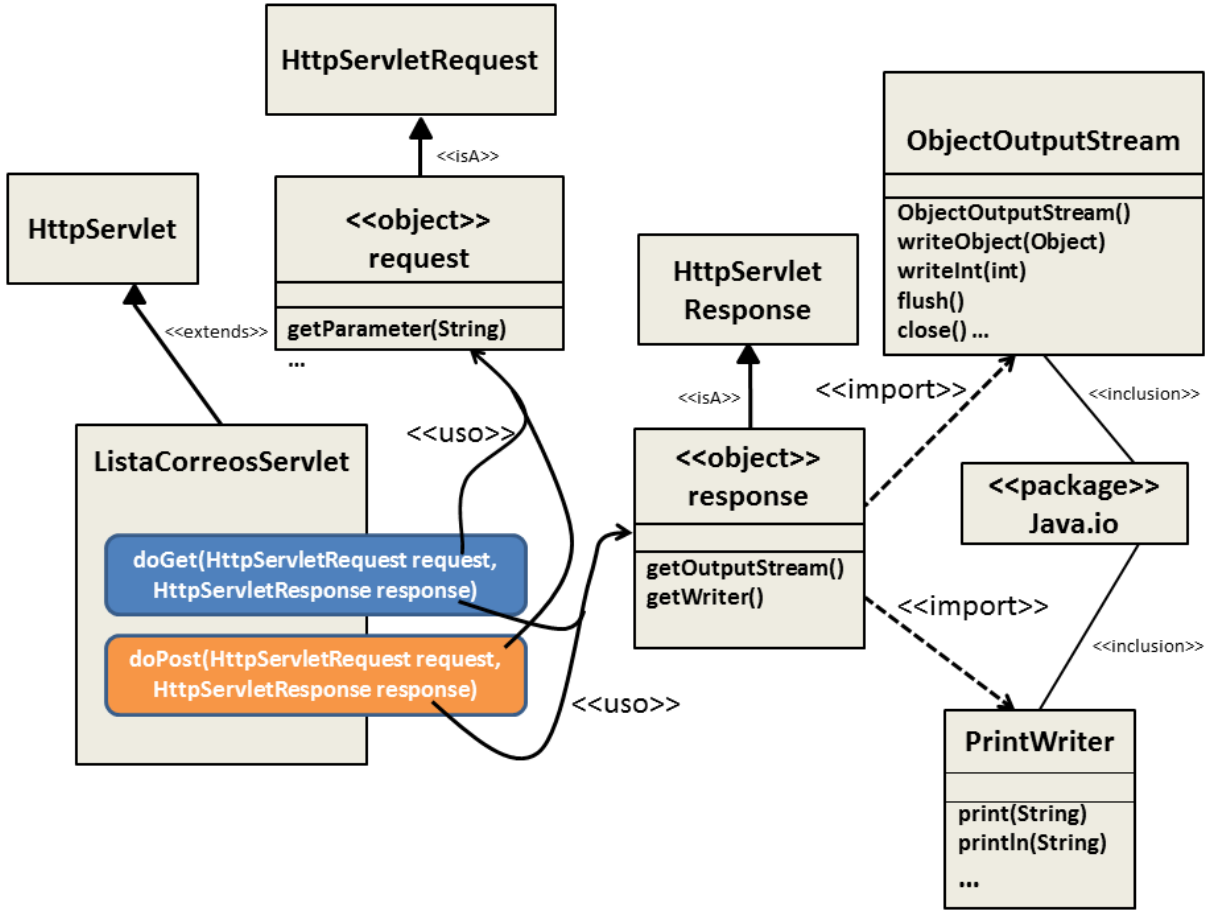


Figura: Diagrama de clases que muestra las relaciones de la clase

Package `interfaz`

Clase Principal `Cliente.java`

- Comienza presentando la tabla con los usuarios de la BD en pantalla (p.e.: una ventana *Swing*)
- Se invoca el método para obtener la *lista de usuarios*:

```
ObjectInputStream
 respuesta= new ObjectInputStream(
 realizarPeticionPost(urlString, parametros));
```

- Espera la respuesta del *Servlet*, que leerá como objetos del *InputStream*

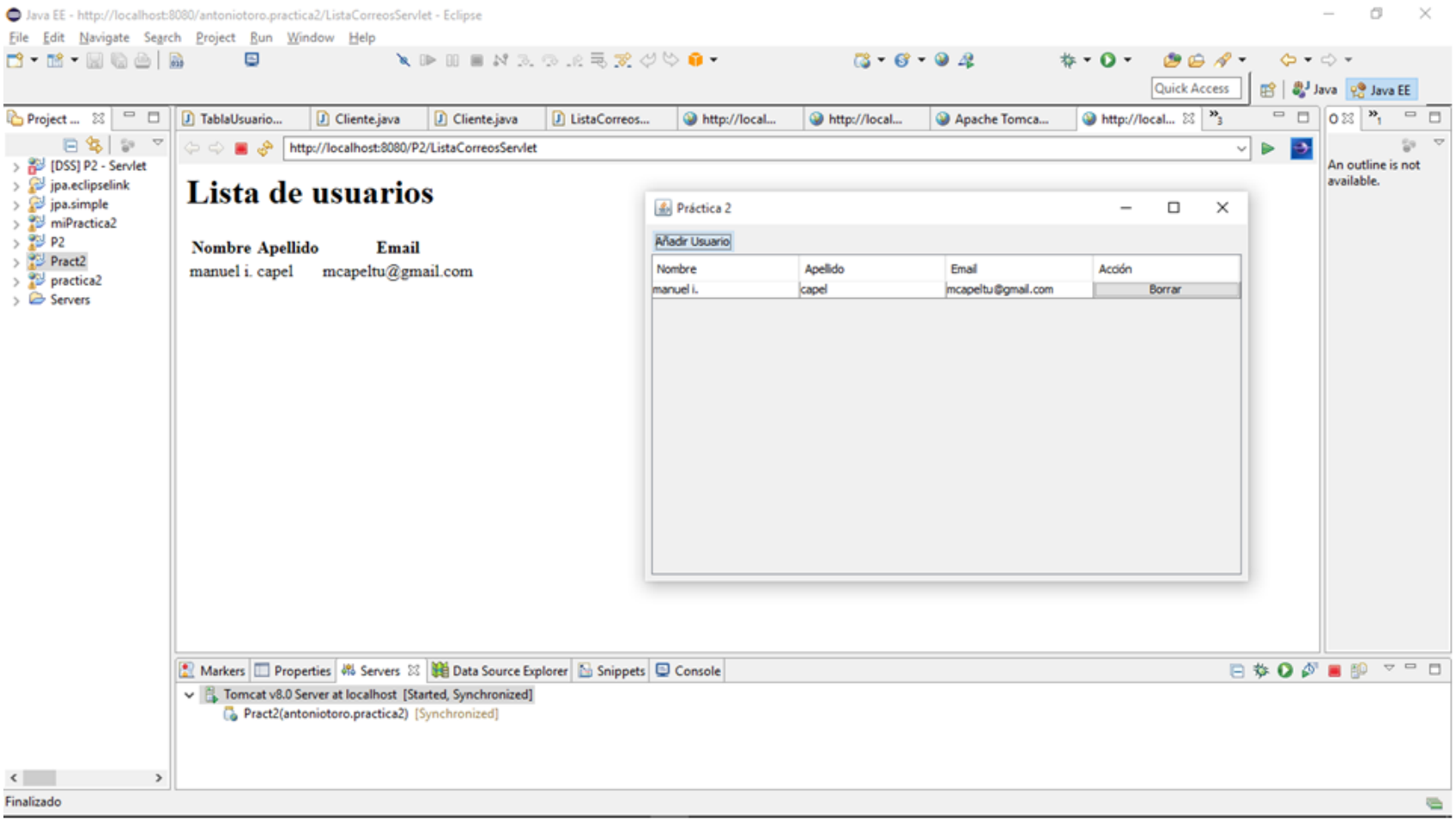
Package `interfaz` II

Clase Principal `Cliente.java`

- La conexión con el *Servlet* devuelve un objeto *InputStream* al volver el método: `getInputStream()`:

```
1 conexion = (HttpURLConnection) url.openConnection();
2 conexion.setUseCaches(false);
3 conexion.setRequestMethod("POST");
4 conexion.setDoOutput(true);
5 OutputStream output = conexion.getOutputStream();
6 output.write(cadenaParametros.getBytes());
7 output.flush();
8 output.close();
9 return conexion.getInputStream();
```

Ventana de *Swing* con la tabla de usuarios



Package `interfaz` III

Clase Principal `Cliente.java`

- A continuación, devuelve un objeto `List<Usuario>` con el que se contruye la tabla de usuarios, que se muestra en la ventana (Swing)
- Por último, se programa la acción para eliminar a un usuario: se programa como cuarta columna de la tabla, para eliminar una fila completa

Bibliografía Fundamental


Bass, L., Clements, P., and Kazman, R. (2012).
Software Architecture In Practice.
Addison-Wesley, Boston, Masssachussets, third edition.


Booch, G. (2008).
Handbook of Software Architecture.
`http://www.booch.com/systems.jsp`.


Buschmann, F. and et al. (2004).
Pattern-oriented software architecture, volume I.
Wiley, Chichester, England.


Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., and Stafford, J. (2010).
Documenting Software Architectures: Views and Beyond.