UDE
UNIVERSIDAD DE LA EMPRESA

Bases de Datos 3

- Docentes: Federico Gómez, Diego Siri
- Carreras: Licenciatura & Ingeniería en Informática
- Año de la carrera: 3º

UDE
UNIVERSIDAD DE LA EMPRESA

Capítulo 4:

Acceso a BD en Arquitecturas de 2 Capas

UDE
UNIVERSIDAD DE LA EMPRESA

Introducción:

En este capítulo estudiaremos los fundamentos para el desarrollo de aplicaciones que utilizan ***bases de datos*** como mecanismo de ***persistencia*** en ***arquitecturas lógicas de 2 capas***.

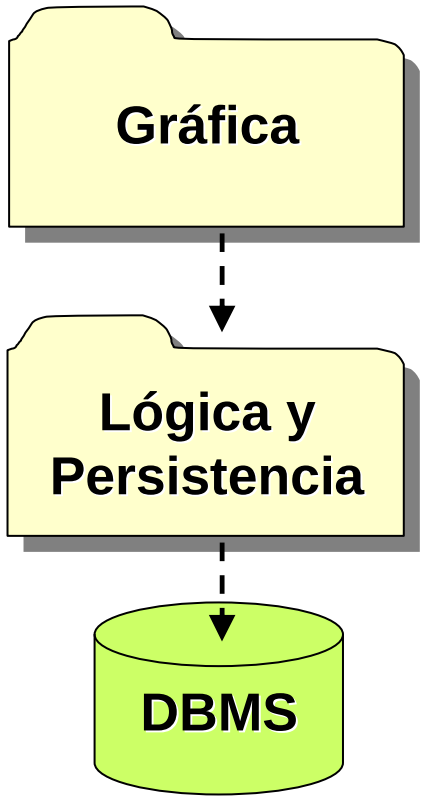
De acuerdo a lo visto en el capítulo 1, los sistemas desarrollados siguiendo arquitecturas lógicas de 2 capas son tales que separan la interfaz de usuario y mantienen unificadas lógica y persistencia.

Recordar que no existe relación específica entre la ***arquitectura lógica*** y la ***arquitectura física*** de una aplicación. Pueden existir aplicaciones de 2 capas tanto ***standalone*** como ***distribuidas***. Lo que las distingue es el hecho de que fusionan lógica y persistencia, separando la presentación, sin importar la cantidad de equipos.

Características de las aplicaciones en 2 capas:

Las aplicaciones desarrolladas según una arquitectura lógica de 2 capas poseen las siguientes características:

- Ya **no** se accede más directamente a la persistencia desde la interfaz de usuario.
- La lógica de los requerimientos todavía sigue siendo resuelta en forma conjunta con el acceso a la persistencia.
- A nivel de código, ahora existe un límite claro entre presentación de datos y el resto de la aplicación, pero aún sigue siendo difuso el límite entre la lógica de requerimientos y el acceso a la persistencia.



UNIVERSIDAD DE LA EMPRESA

Ventajas de las aplicaciones en 2 capas:

- Permiten migrar a una nueva capa de presentación ***sin*** tener que modificar código fuente en la lógica y persistencia.
- ***Liberan*** a la capa gráfica de la responsabilidad de participar en los procesos de resolución de requerimientos y almacenamiento de información.
- Útiles en proyectos donde ya se cuenta con ***stored procedures*** que resuelven los requerimientos y se desea reutilizarlos. Es posible programar una nueva interfaz de usuario para ellos en forma ***independiente*** de cómo resuelven los requerimientos.

Las ***aplicaciones en 2 capas*** fueron ganando popularidad ante el auge de los nuevos tipos de interfaz de usuario (ventanas, páginas web, interfaces de celulares, etc.). Se buscó adaptar aplicaciones ya existentes a las nuevas formas de presentación.

UDE
UNIVERSIDAD DE LA EMPRESA

Desventajas de las aplicaciones en 2 capas:

- Siguen teniendo baja ***mantenibilidad*** (aunque mejoran respecto a las aplicaciones en 1 capa). En caso de usar SQL ***propietario*** en el DBMS, migrar la aplicación a un segundo DBMS implicaría **re- escribir** casi todo el código fuente de la lógica y persistencia.
- Sigue habiendo baja ***portabilidad*** (aunque también mejor en relación a 1 capa). La lógica de requerimientos está ***fusionada*** con el acceso a la persistencia. Si se desea ***mantener*** la lógica y ***cambiar*** la forma de persistencia (por ejemplo: de BD a XML) el costo de separarlas puede ser muy alto.

Las ***aplicaciones en 2 capas*** mejoran varios aspectos en relación a las de 1 capa. No obstante, siguen teniendo desventajas respecto a las de 3 capas, en términos de las ***cualidades del software***.

UDE
UNIVERSIDAD DE LA EMPRESA

Estrategias para separación en 2 capas:

Conforme las aplicaciones fueron evolucionando, se han propuesto varios mecanismos de diseño para separar la capa de presentación de la capa lógica y de persistencia.

Un mecanismo muy difundido es el uso de ***patrones de diseño*** en la aplicación. Podemos pensar en un patrón de diseño como en una "plantilla general de diseño" que podemos adaptar y contextualizar en nuestros propios diseños.

Un patrón es una "***Descripción de la comunicación entre objetos y clases que está adaptada para resolver un problema general de diseño en un contexto particular***". [***Gamma, et al***]. Un patrón nombra, motiva y explica un diseño general al cual recurrir en un problema específico en un desarrollo orientado a objetos.

UNIVERSIDAD DE LA EMPRESA

Estrategias para separación en 2 capas (continuación):

Si bien los patrones de diseño han surgido fundamentalmente en el contexto de los lenguajes ***orientados a objetos***, varias de las ideas que aplican pueden ser trasladables también a ***otros paradigmas***.

En general, un patrón de diseño describe una solución ***ingeniosa*** a un problema de diseño. Muchas veces, la solución no es trivial o no es fácil de deducir mediante las técnicas de diseño tradicionales.

A la fecha, se han creado y documentado decenas (quizás cientos) de patrones. Cada patrón tiene un ***nombre*** que lo identifica, una descripción del ***problema*** que resuelve, un detalle de la ***solución*** propuesta (normalmente dada mediante ***diagramas de UML***) y una lista de las ***consecuencias*** de su aplicación (costos y beneficios).

UDE
UNIVERSIDAD DE LA EMPRESA

Estrategias para separación en 2 capas (continuación):

En este curso no haremos un estudio exhaustivo de patrones. Sólo estudiaremos aquellos orientados a ***separar en capas*** nuestras aplicaciones. Los patrones que estudiaremos para separar la capa gráfica de la capa lógica y de persistencia son:

- ***Value Object***
- ***Facade***
- ***Model-View-Controller***

Cada uno de ellos ataca una problemática puntual de la separación en capas. Cuando se combinan los tres, se consigue una ***completa*** separación entre la capa gráfica y la capa lógica y de persistencia.

Observación: También son llamados ***patrones de arquitectura*** pues refieren a aspectos de ***arquitectura lógica*** de la aplicación.

Value Object:

Este patrón apunta a resolver la transferencia de información entre las dos capas.

Problema:

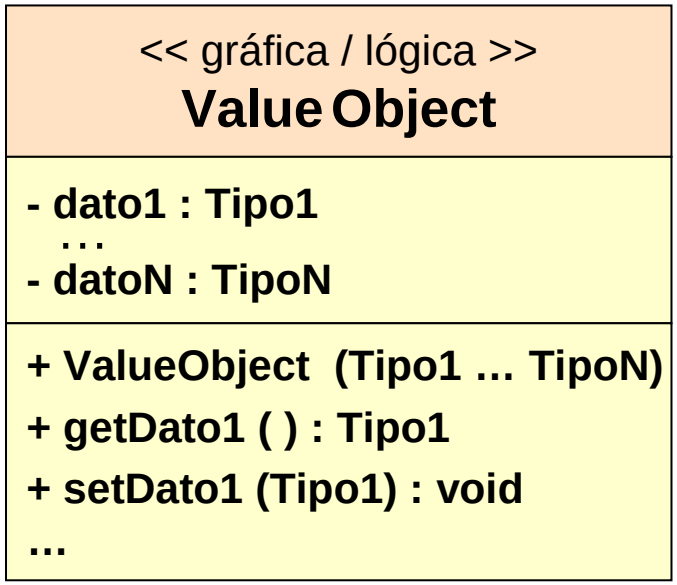
- La capa gráfica necesita intercambiar datos con la capa lógica y de persistencia.
- Se quiere garantizar el intercambio exclusivo de aquellos datos que son estrictamente necesarios.
- Las dos capas residen potencialmente en equipos diferentes, por lo que se quiere realizar el intercambio de todos los datos en forma atómica, a efectos de reducir tiempos en la red.

Observación: También se lo conoce como ***Transfer Object***.

Value Object (continuación):

Solución:

- Crear una clase denominada **Value Object** que tenga como atributos los datos a intercambiar y como únicos métodos los correspondientes constructores, selectores y/o modificadores.



Value Object (continuación):

Consecuencias:

- Solamente se transfieren los datos ***estrictamente*** necesarios.
- Si se desea transferir ***múltiples*** juegos de datos, hay que crear ***tantas*** clases Value Object como conjuntos de datos se tengan.
- Puede producir ***replicación*** de código si los conjuntos ***no*** son disjuntos. Una posible estrategia para evitarlo es mantener una jerarquía de ***herencia*** entre las distintas clases Value Object.

Facade:

Este patrón apunta a resolver el establecimiento de un límite entre las dos capas.

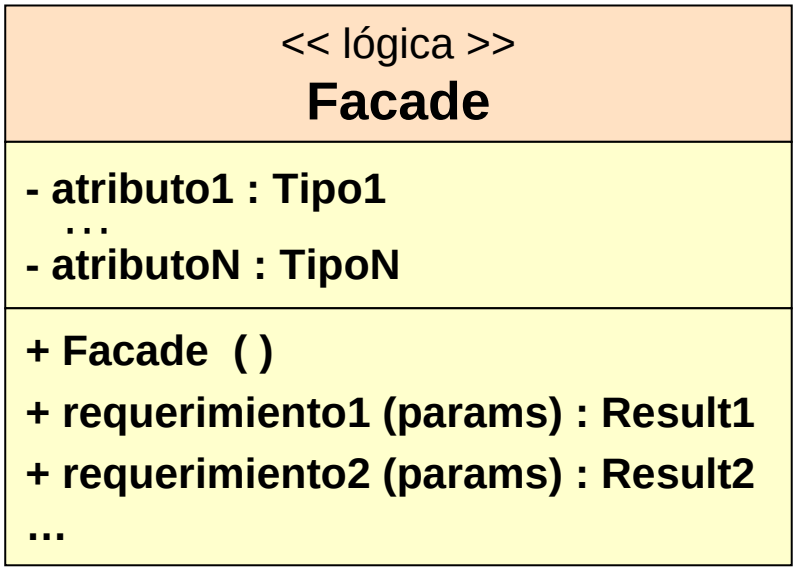
Problema:

- Se desea tener una interfaz de métodos que sirva como ***punto de acceso*** a un subsistema.
- Se desea bajar el acoplamiento entre las clases que utilizan el subsistema (normalmente las de la capa gráfica) y las clases internas al mismo.
- Se desea que cada método de la interfaz permita resolver una funcionalidad (requerimiento funcional) que posee un propósito específico en el contexto de la aplicación.

Facade (continuación):

Solución:

- Crear una clase denominada **Facade** que provea los métodos necesarios para las funcionalidades del subsistema y que encapsule el comportamiento y los recursos internos utilizados.



UNIVERSIDAD DE LA EMPRESA

Facade (continuación):

Consecuencias:

- Se reduce el número de clases que los clientes del subsistema deben conocer, promoviendo así el ***bajo acoplamiento*** entre éstos y las clases del subsistema.
- Se reducen las ***dependencias*** entre las clases externas y las clases internas al subsistema.
- Si se combina con el patrón ***Value Object***, se logra separar ***completamente*** a las clases del exterior (generalmente la capa gráfica) del interior del subsistema (generalmente la capa lógica y de persistencia).

UDE
UNIVERSIDAD DE LA EMPRESA

Model-View-Controller:

Este patrón apunta a desacoplar la presentación de los datos del procesamiento y la transferencia de los mismos.

Problema:

- Se desea mostrar distintas presentaciones gráficas (***vistas***) de un mismo conjunto de datos (***modelo***). Por ejemplo, interfaces de consola, ventanas de escritorio, páginas web, etc.
- Se desea reducir al máximo la toma de decisiones por parte de las vistas y que éstas se encarguen ***exclusivamente*** de mostrar y/o cargar datos.

UNIVERSIDAD DE LA EMPRESA

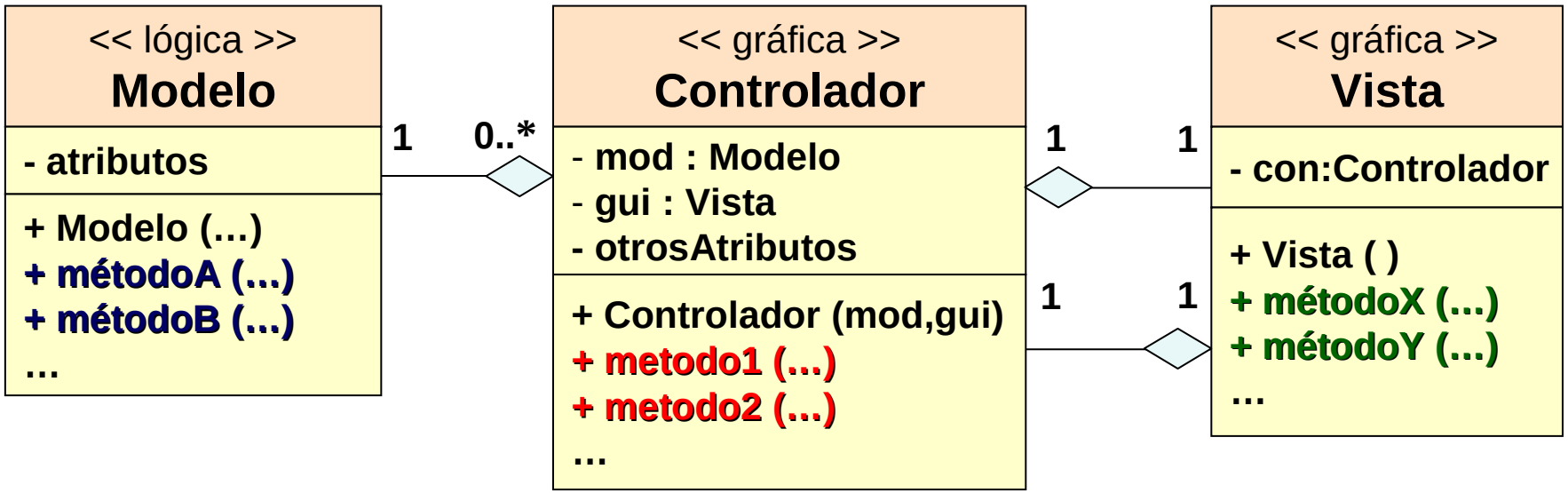
Model-View-Controller (continuación):

Solución:

- Crear una clase denominada ***Modelo*** que encapsule el manejo de los datos y (eventualmente) la persistencia de los mismos.
- Tener una o más clases denominadas ***Vistas*** que presenten los datos al usuario y/o permitan ingresarlos.
- Proveer a cada Vista de un objeto denominado ***Controlador*** que tome la mayoría de las decisiones relativas a la Vista y oficie de intermediario entre ella y el Modelo.

Observación: Las vistas y controladores no necesariamente han de estar implementados mediante clases. Pueden estar dados por módulos de consola, páginas web, etc. Incluso el modelo no ha de ser una clase si se trabaja en lenguajes ***no*** orientados a objetos.

Model-View-Controller (continuación):



- Los ***métodos*** del Modelo son invocados desde los Controladores.
- Los ***métodos*** del Controlador son invocados desde la Vista.
- Los ***métodos*** de la Vista son invocados desde el Controlador.

UNIVERSIDAD DE LA EMPRESA

Model-View-Controller (continuación):

Consecuencias:

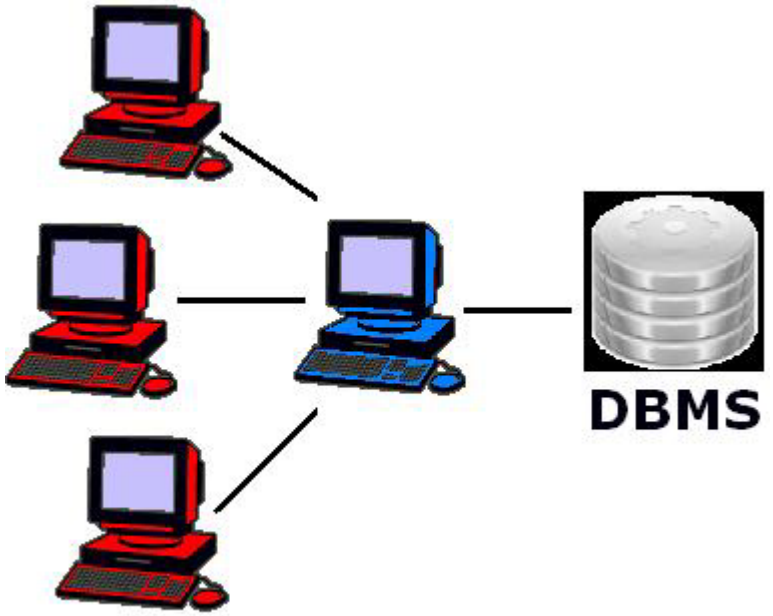
- Se desacopla ***totalmente*** el procesamiento de los datos de sus posibles presentaciones gráficas.
- La toma de decisiones relativas al comportamiento de las Vistas y su interacción con el Modelo es ***delegada*** a los Controladores, dejando a la Vista la ***única*** responsabilidad de mostrar y cargar datos (excelente distribución de responsabilidades).
- Si se combina con los patrones ***Facade*** y ***Value Object*** (siendo el ***Modelo*** quien cumple el rol de la ***Fachada*** y los datos de entrada/salida de sus métodos encapsulados en ***Value Objects***) se consigue una ***completa*** separación entre ambas capas.

Ejemplo de una aplicación en 2 capas:

Veremos un ejemplo de una aplicación en 2 capas desarrollada en **Java** que usa ***bases de datos*** como mecanismo de persistencia, interactuando a través de **JDBC**.

Se trata de un sistema contable con una arquitectura lógica de ***2 capas*** y una arquitectura física ***3-Tier***. Será utilizado por los empleados en forma ***concurrente*** en una red LAN.

Existen puestos de trabajo para los usuarios y un equipo servidor que brinda solución a los requerimientos funcionales y accede a la base de datos (que reside en otro equipo).



UNIVERSIDAD DE LA EMPRESA

Ejemplo de una aplicación en 2 capas (continuación):

La aplicación resolverá requerimientos de gestión contable (nómina de empleados, cálculo de sueldos, pago de facturas, etc.) y aplicará los patrones de diseño: ***Facade***, ***Value Object*** y ***MVC*** vistos antes:

- En el equipo de cada usuario residirá la ***capa gráfica***, donde se ejecutarán **vistas** y **controladores**.
- En el equipo servidor residirá la ***capa lógica y de persistencia***. Tendrá la **fachada** que resolverá los requerimientos, accediendo internamente al equipo que alberga al DBMS.
- Los controladores invocarán remotamente a los métodos de la fachada pasándoles **value objects** como parámetros de entrada y/o recibiéndolos como resultados de las operaciones.

Ejemplo de una aplicación en 2 capas (continuación):


```
public void actionPerformed (...) {
    obtener datos de la ventana
    controlador.altaEmpleado (datos)
}
```

```
public class Controlador {
    ...
    public void altaEmpleado (datos)
    {   crear V.O con datos empleado
        fachada.altaEmpleado (V.O)
        ventana.setMsg (éxito/error)
    }
}
```

```
public class Fachada {
    ...
    public void altaEmpleado (V.O)
    {   obtener datos del  V.O
        chequear existencia en la BD
        si no existe entonces
            insertar empleado en la BD
    }
}
```

UDE
UNIVERSIDAD DE LA EMPRESA

Ejemplo de una aplicación en 2 capas (continuación):

Creamos una clase **Consultas** en el Servidor que define el texto de cada consulta a realizar:

```
public class Consultas {
    public String existsCedula() {
        String query = "SELECT cedula FROM " +
                "Empleados WHERE cedula = ?";
        return query;
    }
    public String insertEmpleado() {
        String insert = "INSERT INTO Empleados " +
         "(cedula, nombre, fecha) VALUES (?,?,?)";
        return insert;
    }
}
```

UNIVERSIDAD DE LA EMPRESA

Ejemplo de una aplicación en 2 capas (continuación):

Creamos una clase **AccesoBD** en el Servidor que encapsula el acceso a la BD:

```
public class AccesoBD {
   public boolean existsCedula
      (Connection con, String ced)
                     throws PersistenciaException
   {  /* verifico la cédula en la BD */  }

   public void insertEmpleado
      (Connection con, String ced, String nom,
                       Date fnac)
                     throws PersistenciaException
   {  /* inserto el registro en la BD */ }
}
```

Ejemplo de una aplicación en 2 capas (continuación):

```
public boolean existsCedula (...) throws ... {
    boolean existeCedula = false;
    try {
        Consultas consultas = new Consultas();
        String query = consultas.existsCedula();
        PreparedStatement pstmt1 =
                con.prepareStatement(query);
        pstmt1.setString(1, ced);
        ResultSet rs = pstmt1.executeQuery();
        if (rs.next()) existeCedula = true;
        rs.close();  pstmt1.close();
    } catch (SQLException e)
        throw new PersistenciaException(...);
    return existeCedula;
}
```

UNIVERSIDAD DE LA EMPRESA

Ejemplo de una aplicación en 2 capas (continuación):

```
public void insertEmpleado (...) throws ... {
    try {
        Consultas consultas = new Consultas();
        String ins = consultas.insertEmpleado();
        PreparedStatement pstmt2 =
                con.prepareStatement(ins);
        pstmt2.setString(1, ced);
        pstmt2.setString(2, nom);
        pstmt2.setDate(3, fnac);
        pstmt2.executeUpdate();
        pstmt2.close();
    } catch (SQLException e)
        throw new PersistenciaException(...);
}
```

UNIVERSIDAD DE LA EMPRESA

Ejemplo de una aplicación en 2 capas (continuación):

Implementamos el método **altaEmpleado** de la clase **Fachada**:

```
public void altaEmpleado (VOEmpleado vo) throws
            EmpleadoException, PersistenciaException
{  try
   {
     Connection con = DriverManager.getConnection
                          (url, user, password);
     con.setTransactionIsolation
           (Connection.TRANSACTION_SERIALIZABLE);
     con.setAutoCommit (false);
     String ced = vo.getCedula ();
     String nom = vo.getNombre ();
     Date fnac = vo.getFechaNacimiento ();
     ...
```

UNIVERSIDAD DE LA EMPRESA

Ejemplo de una aplicación en 2 capas (continuación):

```
    ...
    AccesoBD abd = new AccesoBD ();
    boolean existCed = abd.existsCedula (con, ced);
    if (!existCed)
       abd.insertEmpleado (con, ced, hora ,fnac);
    else
       msgError = “empleado ya registrado”;
    con.commit ();
}
catch (Exception e) {
    con.rollback ();
    errorPersistencia = true;
    msgError = “error de acceso a los datos”;
}
...
```

Ejemplo de una aplicación en 2 capas (continuación):

```
    ...
    finally
    {
      con.close();
      if (existCed)
         throw new EmpleadoException (msgError);
      if (errorPersistencia)
         throw new PersistenciaException (msgError);
    }
}
```

Observación: La capa gráfica permanece completamente ***ajena*** al acceso a la base de datos. Solamente le pasa **value objects** a la capa lógica y recibe **excepciones** en caso de fallar la ejecución.

UNIVERSIDAD DE LA EMPRESA

Pool de conexiones:

En el ejemplo anterior, los usuarios acceden en forma ***concurrente*** a la base de datos. A cada usuario que ejecuta un requerimiento se le asigna su propia ***conexión*** y se manejan ***transacciones***.

Las conexiones a la BD son un ***recurso caro***. Normalmente, existe un número ***limitado*** de conexiones que el DBMS puede mantener abiertas. Dicho número depende de su potencia, memoria, etc.

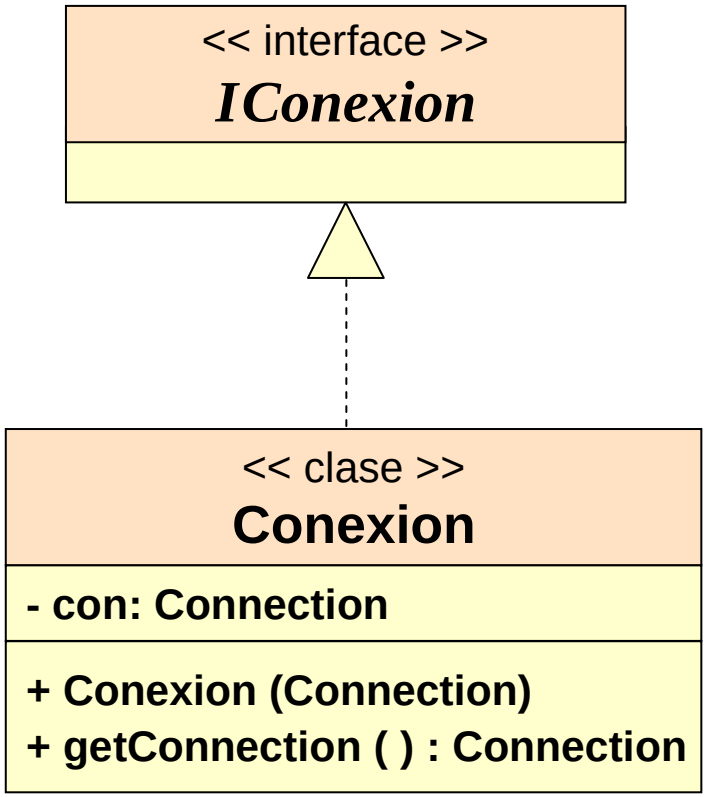
Ninguna aplicación debería sobrecargar la cantidad de conexiones que utiliza. En general, cada aplicación suele mantener abierto un conjunto ***acotado*** de conexiones que reparte entre sus usuarios.

Dicho conjunto se suele llamar ***pool de conexiones***. Hay lenguajes y frameworks que ya los traen predefinidos en sus librerías, aunque también es posible implementar nuestro propio pool de conexiones.

Pool de conexiones (continuación):

Existen ***múltiples*** implementaciones posibles para un ***pool de conexiones***. En este capítulo propondremos una primera implementación sencilla.

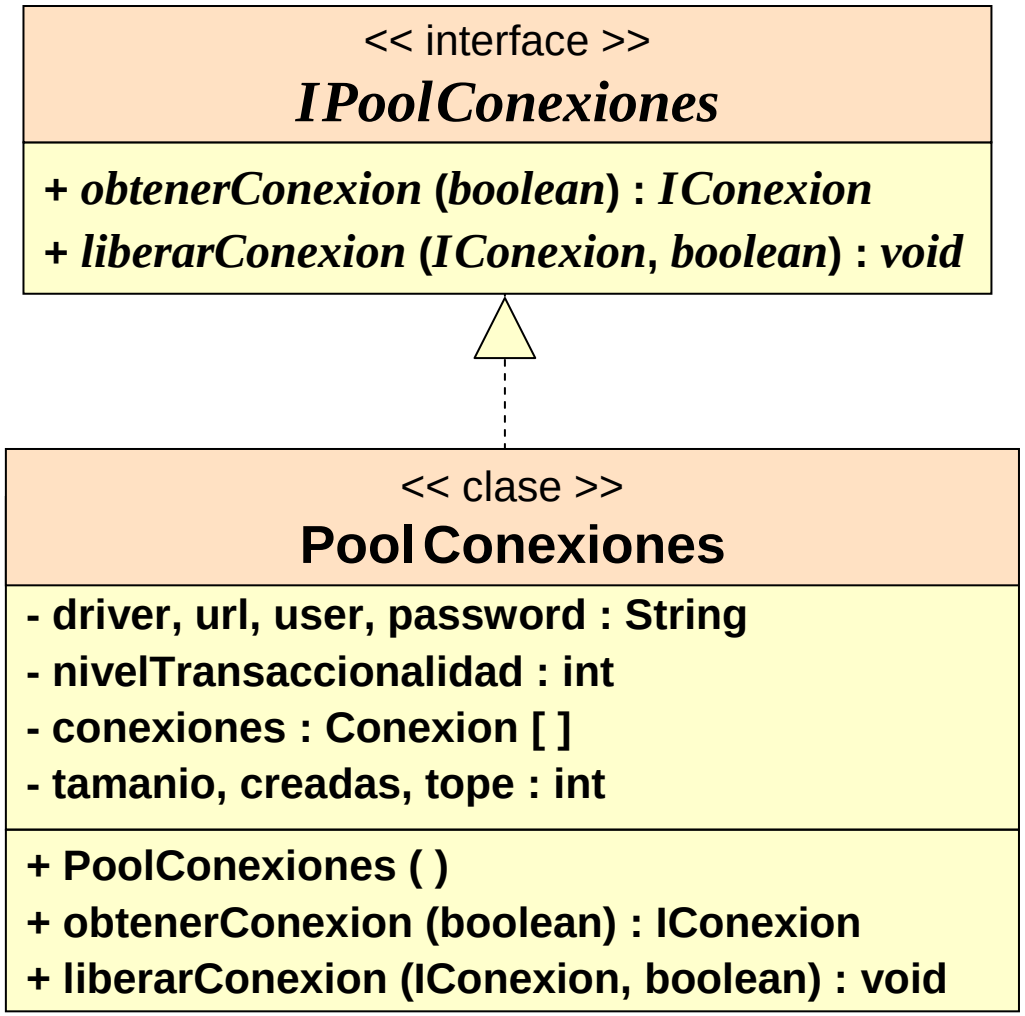
El siguiente diagrama de clases en UML corresponde a una interface que representa una ***conexión*** abstracta junto con una clase concreta que implementa dicha conexión usando internamente una ***Connection*** concreta de JDBC.



Pool de conexiones (continuación):

El siguiente diagrama de clases en UML corresponde a una interface que define un ***pool de conexiones*** a nivel abstracto.

Conjuntamente, se propone una Implementación concreta que maneja una estructura de ***arreglo con tope*** para ir almacenando las conexiones concretas.



UNIVERSIDAD DE LA EMPRESA

Pool de conexiones (continuación):

La interface ***IPoolConexiones*** define los siguientes ***métodos***:

- ***IConexion obtenerConexion (boolean modifica)***

  Solicita una conexión (abstracta) al pool. El parámetro **`modifica`** indica si la conexión que se va a pedir es para realizar una transacción con posibles sentencias de modificación o no.

- ***void liberarConexion (IConexion con, boolean ok)***

  Devuelve una conexión al pool indicando si la transacción fue exitosa o no. Si **ok** vale true, significa que lo fue, si vale false, significa que no lo fue.

Observación: Definir pool de conexiones mediante una ***interface*** tiene la ventaja de que abre la posibilidad de programar ***distintas*** implementaciones para el pool (en este ejemplo se hará con un arreglo con tope, pero podrían haber ***otras*** implementaciones).

UNIVERSIDAD DE LA EMPRESA

Pool de conexiones (continuación):

La clase **PoolConexiones** posee los siguientes ***atributos***:

- **driver, url, user, password**: Para establecer la carga del driver y la url, usuario y password de la BD con la que se establecerán las conexiones.
- **nivelTransaccionalidad**: Nivel de transaccionalidad a ser usado por el pool de conexiones.
- **conexiones**: El arreglo donde se almacenarán las conexiones.
- **tamanio, creadas, tope**: Tamaño del arreglo, cantidad total de conexiones creadas hasta el momento y cantidad actual de conexiones almacenadas en el arreglo, respectivamente.

UDE
UNIVERSIDAD DE LA EMPRESA

Pool de conexiones (continuación):

La clase **PoolConexiones** implementa los siguientes ***métodos***:

- **PoolConexiones ( )**: Constructor de la clase. Realiza la carga del driver, solicita memoria para el arreglo con tope e inicializa los distintos atributos.
- **IConexion obtenerConexion (boolean modifica)**: Solicita una conexión al pool. En caso de que todas estén actualmente en uso, bloqueará al usuario hasta que otro usuario libere alguna.
- **void liberarConexion (IConexion con, boolean ok)**: Devuelve una conexión al pool y avisa a posibles usuarios bloqueados. Si **ok** vale true, hará ***commit*** al devolverla, sino hará ***rollback***.

Observación: Si bien **no** es frecuente, el pool podría usarse en un contexto que **no** es transaccional. En tal caso, al liberar la conexión simplemente se la retorna al pool **sin** hacer ***commit*** ni ***rollback***.

UNIVERSIDAD DE LA EMPRESA

Pool de conexiones (continuación):

Modificamos el método **altaEmpleado** de la **Fachada** del ejemplo anterior para incorporarle el uso del ***pool de conexiones***:

```
public void altaEmpleado (VOEmpleado vo) throws
            EmpleadoException, PersistenciaException
{  try
   {
     /* la fachada tendrá ahora el siguiente
        atributo: IPoolConexiones pool; */
     IConexion icon = pool.obtenerConexion(true);
     String ced = vo.getCedula();
     String nom = vo.getNombre();
     Date fnac = vo.getFechaNacimiento();
     ...
```

UNIVERSIDAD DE LA EMPRESA

Pool de conexiones (continuación):

```
    ...
    AccesoBD abd = new AccesoBD ();
    boolean existCed = abd.existsCedula (icon, ced);
    if (!existCed)
       abd.insertEmpleado (icon, ced, hora ,fnac);
    else
       msgError = "empleado ya registrado";
    pool.liberarConexion (icon, true);
  }
  catch (Exception e) {
    pool.liberarConexion (icon, false);
    errorPersistencia = true;
    msgError = "error de acceso a los datos";
  }
  ...
```

Pool de conexiones (continuación):

```
    ...
    finally {
      if (existCed)
         throw new EmpleadoException (msgError);
      if (errorPersistencia)
         throw new PersistenciaException (msgError);
    }
}
```

Observación: A diferencia del primer ejemplo, la fachada ***no*** cierra la conexión, ***solamente*** la devuelve al pool. Ahora es el pool quien internamente se encarga de abrir y cerrar las conexiones.

Nótese que la **Fachada** maneja ahora conexiones ***abstractas***. Las conexiones ***concretas*** son manipuladas dentro de los métodos de la clase **AccesoBD**.