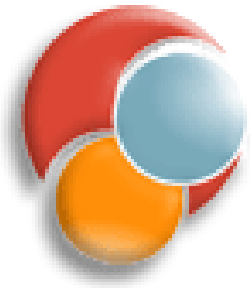


Java y Herramientas de Desarrollo

Sesión 10: Transacciones, optimización, y patrones de datos

Puntos a tratar

- Optimización de consultas
- Transacciones
- Patrón DAO

Sentencias preparadas

- Cuando ejecutamos una sentencia SQL, esta se compila y se envía al SGBD
- Si la vamos a ejecutar muchas veces, es preferible dejar la sentencia preparada (precompilada) para aumentar la eficiencia
- Disponemos de una clase, `PreparedStatement`, que nos permite realizar la precompilación

```
PreparedStatement ps = con.prepareStatement(
          "UPDATE FROM alumnos
                SET sexo = 'H'
                WHERE exp>1200 AND exp<1300");
```

Sentencias preparadas

- Cuando creamos el objeto ya se le pasa la sentencia a ejecutar. En ese momento la precompila
- Esta clase nos va a permitir parametrizar la sentencia:

```
PreparedStatement ps = con.prepareStatement(
          "UPDATE FROM alumnos
                SET sexo = 'H'
                WHERE exp > ? AND exp < ?");
```

- Los ‘?’ son parámetros que podremos asignar más tarde

Asignación de parámetros

- Para dar valor a los parámetros usaremos los métodos `setXXXX` al igual que anteriormente
- Estos métodos reciben dos parámetros, el número de orden del parámetro y el valor a asignar

```
ps.setInt(1,1200);
ps.setInt(2,1300);
```

- Como la sentencia era de actualización, para ejecutarla llamaremos al método `executeUpdate`, pero ahora sin parámetros

```
int n = ps.executeUpdate();
```

- Si la sentencia fuera de consulta usaríamos `executeQuery` y también disponemos de `execute`

Restricciones en las sentencias preparadas

- Los parámetros sólo pueden ser de datos, es decir, no podemos usar como parámetros el nombre de la tabla, por ejemplo
- Una vez asignados los parámetros, estos no desaparecen
- Podemos llamar a `clearParameters`

SQL Injection

- Se trata de “inyectar” código SQL malicioso dentro de una sentencia SQL
- Tenemos el código:

```
String s="SELECT * FROM usuarios WHERE
  nombre='"+nombre+"';";
```

- *Nombre* es una variable con valor de un campo introducido por el usuario. Si `nombre=Miguel`

```
SELECT * FROM usuarios WHERE
  nombre=‘Miguel';
```

Código malicioso

- Si en vez de eso, el usuario mete el siguiente código:

```
Miguel'; drop table usuarios; select *
  from usuarios; grant all privileges ...
```

- El resultado sería:

```
SELECT * FROM usuarios WHERE
  nombre='Miguel'; drop table usuarios;
  select * from usuarios; grant all
  privileges ...
```

Para evitar la inyección de SQL

- Usar sentencias preparadas:

```
PreparedStatement ps =
con.prepareStatement("SELECT * FROM
usuarios WHERE nombre=?");
ps.setString(nombre);
```

- Ahora, con el código de antes, el SQL queda:

```
SELECT * FROM usuarios WHERE
nombre="Miguel'; drop table usuarios;
select * from usuarios; grant all
privileges ...";
```

Transacciones

- En determinadas aplicaciones, cuando tenemos que realizar varias acciones, se requiere que o bien se hagan todas a la vez o bien ninguna
- Reserva de vuelo de Alicante a Osaka:
  - Vuelo Alicante-Madrid
  - Vuelo Madrid-Amsterdam
  - Vuelo Amsterdam-Osaka
- Si alguna de estas reservas no se produce no queremos que se reserve ninguna

Transacción

- Tratamos un conjunto de acciones como una unidad
- Las transacciones hacen que la BD pase de un estado consistente al siguiente
- La mayoría de la BD incorporan un sistema llamado *commit* (“confirmación”)
- Por defecto, se asume que cada vez que realicemos una acción se realiza el *commit* (*autocommit*)

Utilización de *commit*

- Primero debemos decir que no realice el *autocommit* en cada acción
- Luego realizamos cada una de las acciones en esta transacción
- Por último realizamos *commit* para hacer los cambios persistentes
- Si se produjo un error (podemos capturar las excepciones) llamaremos a *rollback* para que elimine los cambios realizados

Ejemplo

```
try {
  con.setAutoCommit(false);
  stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen,
   destino) VALUES('Paquito', 'Alicante', 'Madrid')");
  stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen,
   destino) VALUES('Paquito', 'Madrid', 'Amsterdam')");
  stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen,
   destino) VALUES('Paquito', 'Amsterdam', 'Osaka')");

  // Hemos podido reservar los tres vuelos correctamente
  con.commit();
} catch(SQLException e) {
  // Alguno de los tres ha fallado, deshacemos los cambios
  try {
    con.rollback();
  } catch(SQLException e) {};
}
```

Concurrencia en el acceso

- ¿Qué sucede cuando varias aplicaciones acceden a la vez a los mismo datos (para modificarlos)?
- Una aplicación abre una transacción y modifica un dato de una tabla. Justo en ese momento otra aplicación lee y/o modifica ese dato …
- El manejo de la concurrencia es gestionada de distinta manera por los distintos SGBD

Obtención de información

- Podemos saber cuál es el nivel de concurrencia de la BD, llamando a

```
int con.getTransactionIsolation();
```

- Este método devolverá:
  - `Connection.TRANSACTION_NONE`
    No soporta transacciones
  - `Connection.TRANSACTION_READ_UNCOMMITTED`
    Soporta transacciones, pero sólo en lectura (MySQL)
  - `Connection.TRANSACTION_READ_COMMITTED`
    Si una aplicación ha escrito en un registro, bloquea el registro y ninguna otra aplicación puede escribir en él hasta que la primera no haga *commit* (PostGres)

Interbloqueo

- Un interbloqueo se produce cuando:
  - Aplicación 1: modifica registro A y después B
  - Aplicación 2: modifica registro B y después A
- Se ejecutan las dos a la vez y quedan esperando a que se libere el registro que se quiere modificar
- MySQL no detecta esta situación (no se produce)
- En PostGres se produce, pero detecta la situación y produce una excepción

Patrón *Data Access Object: DAO*

- El acceso a los datos varía dependiendo de la fuente de los datos que estemos usando, incluso es posible que provenga de distintas fuentes de datos (BD, LDAP, SIE, fichero de texto, ...) que a menudo implica el uso de códigos propietarios.

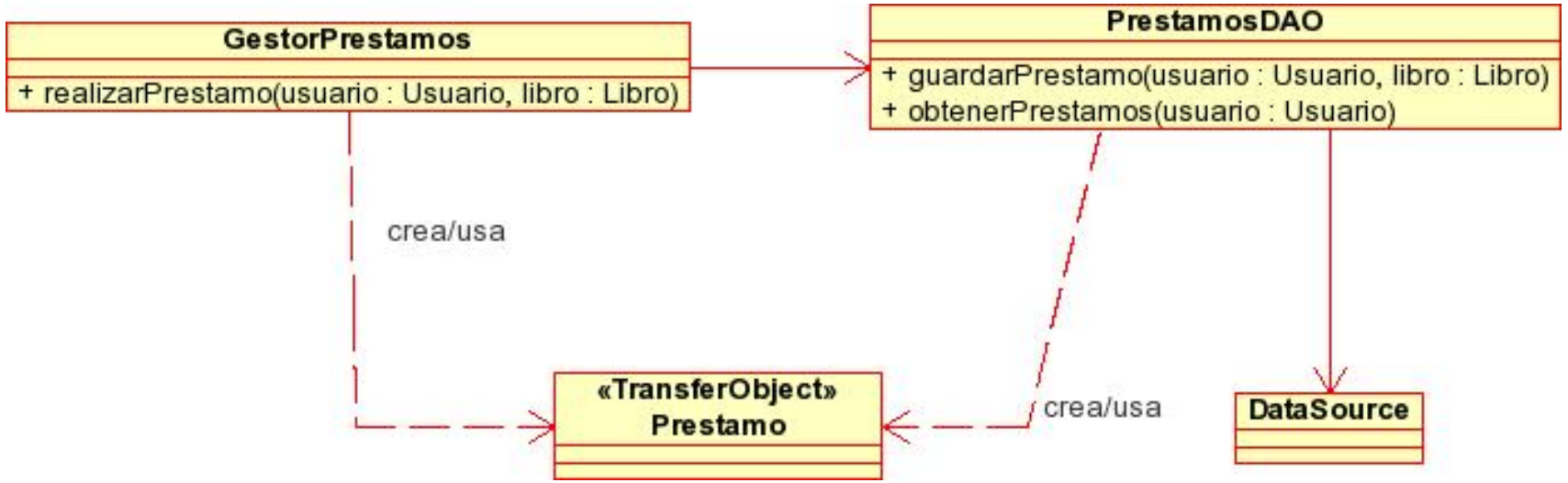
- Por otro lado, podemos tener una clase que además de acceder a la base de datos tiene otras responsabilidades

Ejemplo incorrecto

- Ejemplo: `GestorPrestamos` de una biblioteca, implementa `realizarPrestamo`:
  - **Lógica de negocio**: número de días según tipo de usuario, no prestar si ya es moroso, etc.
  - **Acceso a datos:** crear un registro en la tabla "préstamos"

- **Solución**: crear una clase que abstraiga y se encargue **solo del acceso a datos**

Diagrama de clases del DAO



Beneficios básicos del DAO

- Separar el **acceso a datos** del resto de **funciones**

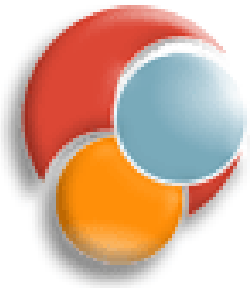
Separation of concerns: una clase no debe tener dos responsabilidades distintas

- **Independencia** del almacén de datos: sea base de datos relacional, fichero XML, `.properties`,... (con la ayuda del patrón *Factory*)

Hay que separar lo que permanece fijo de lo que puede variar en una aplicación

Discusión de algunos aspectos

- El DAO no tiene por qué implementar todas las operaciones CRUD (*Create-Read-Update-Delete*)
- En general por cada objeto de negocio crearemos un DAO distinto (Libro→`LibroDAO`, Usuario→`UsuarioDAO`,...). Añade una capa
- El paso de información se encapsula en *Transfer Objects*



¿Preguntas...?