

Información cambios Java:

He utilizado una capa DAO (Data Acces Object) para aislar la implementación de la base de datos que utilicemos. De esta forma, cumplimos el principio de “single responsibility”, y conseguimos un código con menos acoplamiento y a la vez, mas cohesionado ya que cada clase DAO tendrá los métodos para gestionar sus respectivas clases concretas, y no tendremos todas las funciones en una sola clase de aplicacionBD.

```
public abstract class DAOFactory {

    /** numero para identificar que queremos usar una conexion Mysql */
    public static final int MYSQL = 0;

    /** Abstract method para el JocAlternatiuDAO. */
    public abstract JocAlternatiuDAO getJocAlternatiuDAO();
    /** Abstract method para el JocDAO. */
    public abstract JocDAO getJocDAO();
    /** Abstract method para el JocTaulaDAO. */
    public abstract JocTaulaDAO getJocTaulaDAO();
    /** Abstract method para el PartidaDAO. */
    public abstract PartidaDAO getPartidaDAO();
    /** Abstract method para el UsuarioDAO. */
    public abstract UsuarioDAO getUsuarioDAO();

    /** Aqui podremos añadir tantas bases de datos diferentes como queramos,
     * desde el main solo haria falta cambiar el numero de la base de datos a
     * utilizar y adaptariamos todo**
     */

    public static DAOFactory getDAOFactory(int databaseType) {
        switch(databaseType) {
            case MYSQL:
                return new MysqlDAOFactory();

            default:
                return null;
        }
    }
}
```

La clase DAOFactory lo único que hará será crear objetos DAO (objetos que accedan a la base de datos y hagan operaciones con ella) con funciones abstractas para retornar un objeto de tipo interfaz (general) de cada tipo de objeto sin tener en cuenta si usan Mysql, Oracle o cualquier otra base de datos y un método para especificar que tipo de base de datos queremos usar, a través de este método en el main elegimos que base de datos queremos usar:

```
public class Main {

    public static void main(String[] args) throws SQLException {

        DAOFactory mysqlFactory = DAOFactory.getDAOFactory(DAOFactory.MYSQL);
        // ...
    }
}
```

Simplemente debemos pasarle por parámetro el número de la base de datos a usar (en este caso he definido un int estatico en la propia clase para usarlo desde cualquier lado, llamado igual que la base de datos a usar), y **todo nuestro código estará adaptado sin tocar nada más en el código.**

```

public MysqlDAOFactory() {};

/** Usaremos este metodo cada vez que queramos una conexion de tipo mysql */
public static Connection crearConexion() {
    Connection conn = null;
    MysqlDataSource dataSource;
    dataSource = new MysqlDataSource();
    dataSource.setUser("root");
    dataSource.setServerName("127.0.0.1");
    dataSource.setPassword("");
    dataSource.setDatabaseName("AQueJugamos");

    try {
        conn = dataSource.getConnection();
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return conn;
}

```

En la clase MysqlDAOFactory implementamos el método estático que retorna una conexión de tipo Mysql.

```

@Override
public JocAlternatiuDAO getJocAlternatiuDAO() {
    return new JocAlternatiuDAOMysqlImp();
}

```

Y los métodos que hacen override a los de la clase DAOFactory, para devolver una conexión de tipo Mysql.

Cada objeto DAO (usuario, juego, partida, etc) tiene una interfaz para aislar la implementación de cada uno de estos, y de esta manera poder tratar por igual un DAO de mysql, que uno de oracle o cualquier otro acceso a datos. Esto da una gran escalabilidad y flexibilidad.

```

public interface UsuarioDAO {

    public void carregarUsuaris();
    public boolean comprobarUsuari(String mail, String password);
    public void afegirLlistaPreferits(int idJoc, int idUsuari);
    public void consultaDadesUsuari();
    public void insertarDadesUsuari(String nomUsuari, String password, String mail, String fe
    public boolean comprovarUsuariLogejat();
    public void logout();
    public List<Usuari> getLlistaUsuaris();

}

```

(Ejemplo de interfaz, estos métodos los tendrán todos los objetos DAO sea cual sea su implementación)

Por último queda implementar cada DAO según su base de datos (están en el paquete com.aquejugamos.DAOImplementation, en estas clases podemos trabajar como siempre hemos hecho, no tenemos que cambiar nada mas), nosotros por el momento solo necesitamos la de Mysql, así que hacemos la implementación para Mysql (ya la teníamos hecha, solo he separado por clases las funciones que ya teníamos).

```
public class UsuarioDAOMySQLImp implements UsuarioDAO{

    /** Logger */
    private static Logger logger = LoggerFactory.getLogger(UsuarioDAOMySQLImp.class);

    private List<Usuari> llistaUsuaris = new ArrayList<Usuari>();

    public void carregarUsuaris() {
        Connection conn = MySQLDAOFactory.crearConexio();
        try {
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT U.idUsuario, U.Email, U.Nombre, U.password, U.Ac

            while(rs.next())
            {
                int idUsuario=rs.getInt(1);
                String email=rs.getString(2);
                String nombre = rs.getString(3);
                String password=rs.getString(4);
                int activo = rs.getInt(5);
                int bloqueado = rs.getInt(6);
                int isAdmin = rs.getInt(7);
                String grupo = rs.getString(8);
                String fecha = rs.getString(9);
                String provincia = rs.getString(10);
                Usuari usuari = new Usuari(idUsuario,nombre,password,email,activo,bloqueado, isAdmin,

                llistaUsuaris.add(usuari);
            }

            stmt.close();
            rs.close();
        }
    }
}
```

En el main solo hace falta instanciar las clases:

```
public class Main {

    public static void main(String[] args) throws SQLException {

        DAOFactory mysqlFactory = DAOFactory.getDAOFactory(DAOFactory.MYSQL);
        UsuarioDAO usuariDAO = mysqlFactory.getUsuarioDAO();
        JocAlternatiuDAO jocAlternatiuDAO = mysqlFactory.getJocAlternatiuDAO();
        JocTaulaDAO jocTaulaDAO = mysqlFactory.getJocTaulaDAO();
        PartidaDAO partidaDAO = mysqlFactory.getPartidaDAO();
        JocDAO jocDAO = mysqlFactory.getJocDAO();

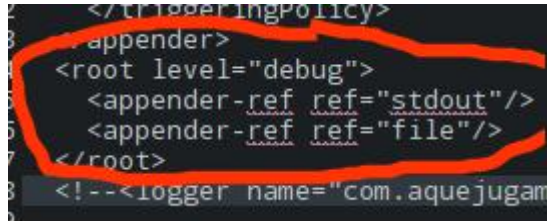
        jocDAO.mostrarJocs();
        jocAlternatiuDAO.carregarJocsAlternatius();
        jocTaulaDAO.carregarJocsTaula();
        usuariDAO.carregarUsuaris();
    }
}
```

(UsuarioDAO tendrá los métodos de base de datos relacionados con usuario: login, registrar, ver favoritos usuario, etc...), el de JocDAO lo relacionado con Joc, etc...)

Si necesitamos añadir una nueva función solo hace falta añadirla en la clase a la que pertenezca y ya.

Sistema de logs:

También he añadido las librerías necesarias para poder utilizar logs y he configurado el archivo XML para personalizarlo. En este archivo he puesto que los logs se guarden en un fichero de texto y que salgan por consola.

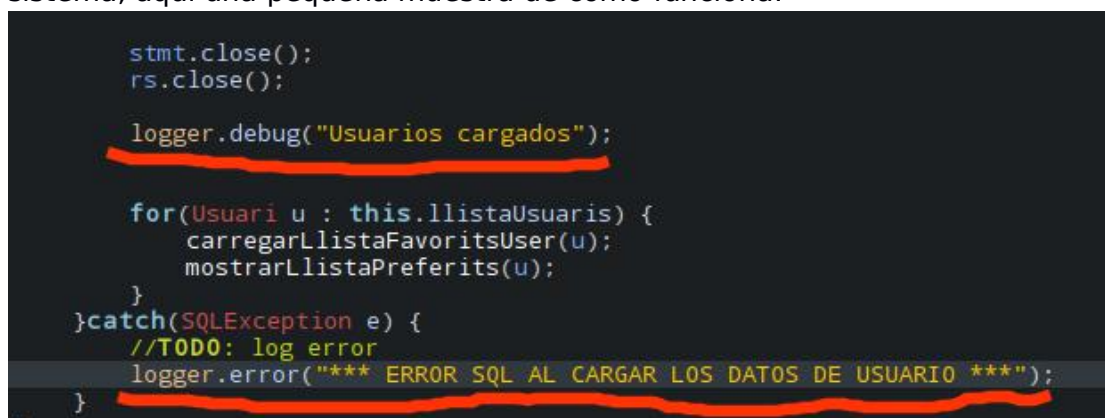
A screenshot of an XML configuration file for logback. The root element is <configuration>. Inside, there is a <root level="debug"> element. This element has two children: <appender-ref ref="stdout"/> and <appender-ref ref="file"/>. The entire configuration block is circled in red. Below the root element, there is a comment: <!--<logger name="com.aquejugam

Solo debemos modificar esos valores, si queremos solo escribirlos en fichero podemos comentar la línea que tiene un appender "stdout" (que es la que específica que se muestren por consola).

Estos son los niveles de log que nos proporciona la librería de logback:

level of request p	effective level q					
	TRACE	DEBUG	INFO	WARN	ERROR	OFF
TRACE	YES	NO	NO	NO	NO	NO
DEBUG	YES	YES	NO	NO	NO	NO
INFO	YES	YES	YES	NO	NO	NO
WARN	YES	YES	YES	YES	NO	NO
ERROR	YES	YES	YES	YES	YES	NO

Se puede especificar en el XML (donde sale root level="debug") que nivel de logs queremos mostrar/guardar, si solo nos interesan los errores, pondremos el nivel a "ERROR". Aun no he puesto casi ningún log, solo he implementado el sistema, aquí una pequeña muestra de como funciona:

A screenshot of Java code. It shows a database connection being closed, then a debug log message "Usuarios cargados" is printed. Then, a loop iterates over a list of users, loading their favorites and preferences. Finally, there is a catch block for SQLException, which contains a TODO comment and an error log message "*** ERROR SQL AL CARGAR LOS DATOS DE USUARIO ***". The log messages are circled in red.

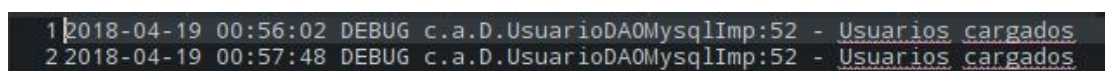
```
stmt.close();
rs.close();

logger.debug("Usuarios cargados");

for(Usuari u : this.llistaUsuaris) {
    cargarLlistaFavoritsUser(u);
    mostrarLlistaPreferits(u);
}
}catch(SQLException e) {
    //TODO: log error
    logger.error("*** ERROR SQL AL CARGAR LOS DATOS DE USUARIO ***");
}
```

Ponemos logger.nivel-de-importancia-del-log("mensaje");

En el fichero log4j-application.log podemos ver los logs guardados.

A screenshot of a log file named log4j-application.log. It shows two log entries. The first is a debug message from c.a.D.UsuarioDAOmysqlImp:52 saying "Usuarios cargados". The second is another debug message from the same source saying "Usuarios cargados".

```
1 2018-04-19 00:56:02 DEBUG c.a.D.UsuarioDAOmysqlImp:52 - Usuarios cargados
2 2018-04-19 00:57:48 DEBUG c.a.D.UsuarioDAOmysqlImp:52 - Usuarios cargados
```


Cambios 20/04/2018

He comenzado a hacer algun test y he tenido que hacer algunos cambios. He pensado bastante sobre como testear los DAO ya que sus funciones se basan en llamadas a la base de datos, pero no tienen mucha mas lógica, por lo que no tenia sentido hacer un mockObject (el cual simula un retorno de datos y se aplica la lógica sobre estos datos). He mirado diferentes opciones por internet pero parece ser que es un tema complicado, hay gente que dice que lo referente a la base de datos se suele testear en las pruebas de integración y otros que tambien se debe hacer unit testing.

Al final he optado por lo siguiente:

-Tener una base de datos en local que sea una copia de la original (no en datos, solo en estructura), donde podamos hacer las pruebas sabiendo los datos que hay en ella y sin que puedan haber efectos colaterales en la base de datos original.

Para ello, he creado una clase que se encargará de crear conexiones Mysql a la base de datos de testeo:

```
public class MysqlDAOFactoryWithTestDB extends DAOFactory{
    /** Logger */
    private static Logger logger = LoggerFactory.getLogger(MysqlDAOFactoryWithTestDB.class);

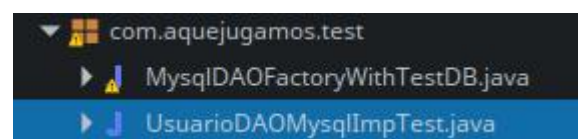
    public MysqlDAOFactoryWithTestDB() {};

    /** Usaremos este metodo cada vez que queramos una conexion de tipo mysql */
    public static Connection crearConexion() {
        Connection conn = null;
        MysqlDataSource dataSource;
        dataSource = new MysqlDataSource();
        dataSource.setUser("root");
        dataSource.setServerName("127.0.0.1");
        dataSource.setPassword("");
        dataSource.setDatabaseName("aquejugamostest"); //cambiamos a una base de datos local de prueba

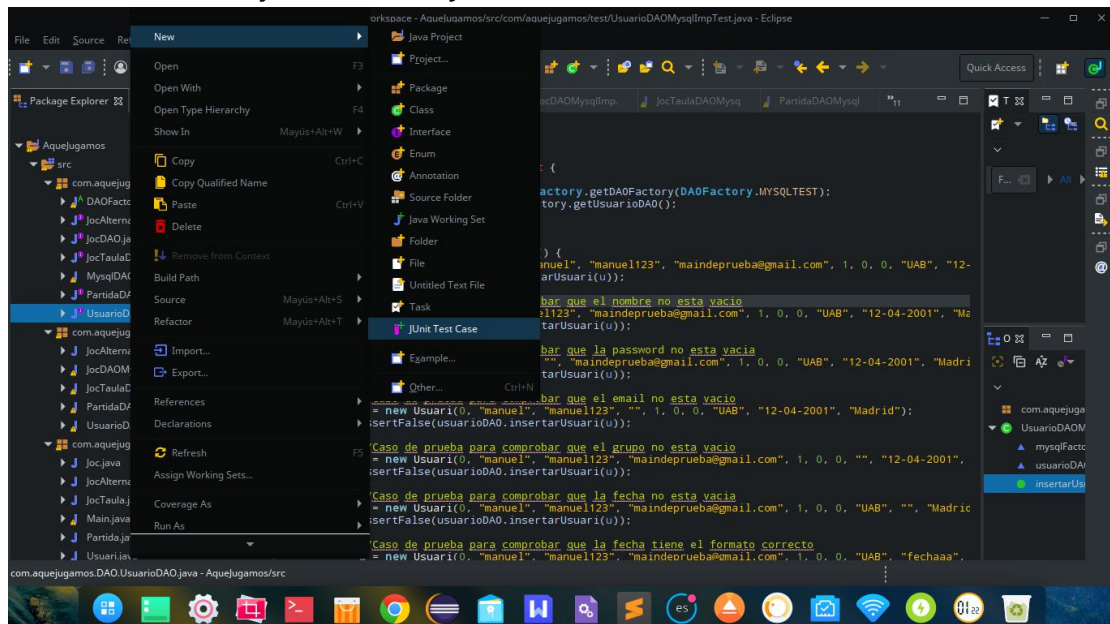
        try {
            conn = dataSource.getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return conn;
    }
}
```

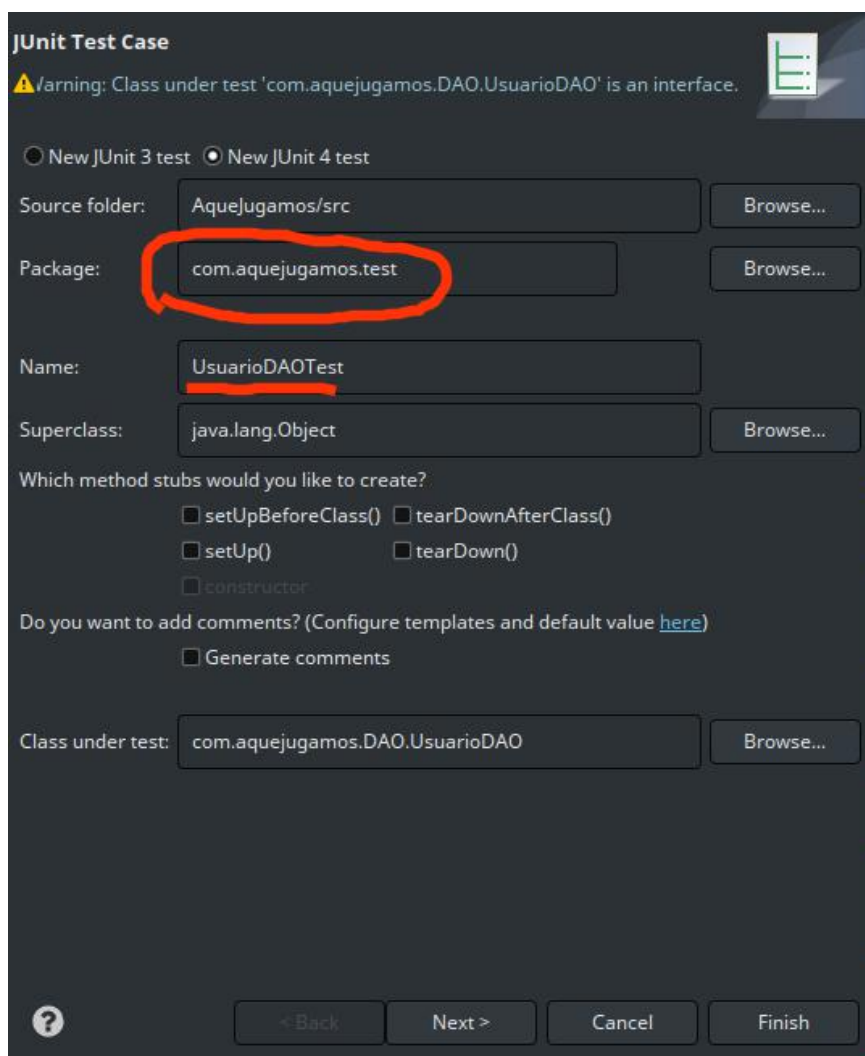
Y para cada clase DAO de implementación (las que contienen Imp) haremos un fichero de testing con Junit (de momento solo he comenzado el de Usuari)



Para crear los ficheros de testing debemos seleccionar con click derecho sobre la clase a testear y seleccionar Junit Test Case:



Y nos saldrá una ventana como esta:

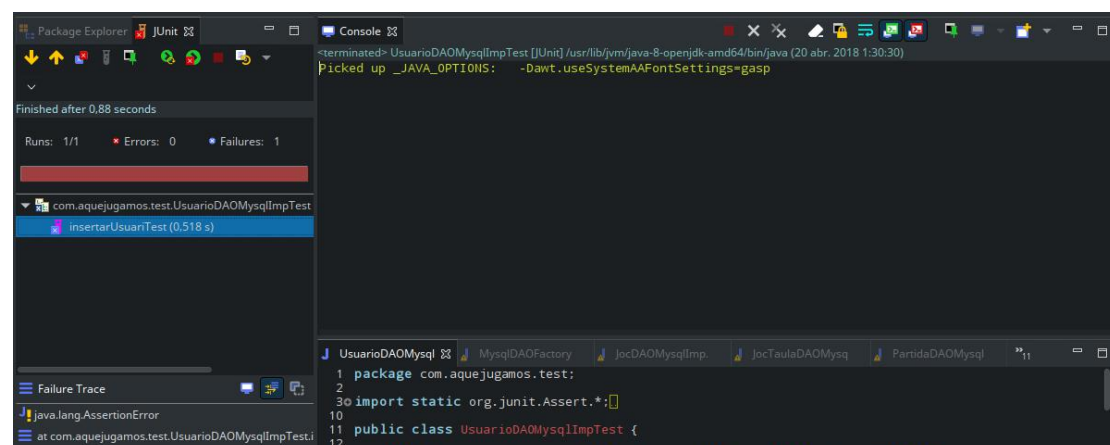


Debemos seleccionar que lo pondremos en el paquete de test, y como convención podemos llamar a los ficheros de test igual que el fichero a testear añadiendo un "Test" al final.

Finalmente ya podemos implementar los test case:

```
1 package com.aquejugamos.test;
2
3 import static org.junit.Assert.*;
4
5 public class UsuarioDAOMysqlImpTest {
6
7     DAOFactory mysqlFactory = DAOFactory.getDAOFactory(DAOFactory.MYSQLTEST);
8     UsuarioDAO usuarioDAO = mysqlFactory.getUsuarioDAO();
9
10    @Test
11    public void insertarUsuariTest() {
12        Usuario u = new Usuario(0, "manuel", "manuel123", "maindeprueba@gmail.com", 1, 0, 0, "UAB", "12-04-2001", "Madrid");
13        assertTrue(usuarioDAO.insertarUsuario(u));
14
15        //Caso de prueba para comprobar que el nombre no esta vacio
16        u = new Usuario(0, "", "manuel123", "maindeprueba@gmail.com", 1, 0, 0, "UAB", "12-04-2001", "Madrid");
17        assertFalse(usuarioDAO.insertarUsuario(u));
18
19        //Caso de prueba para comprobar que la password no esta vacia
20        u = new Usuario(0, "manuel", "", "maindeprueba@gmail.com", 1, 0, 0, "UAB", "12-04-2001", "Madrid");
21        assertFalse(usuarioDAO.insertarUsuario(u));
22
23        //Caso de prueba para comprobar que el email no esta vacio
24        u = new Usuario(0, "manuel", "manuel123", "", 1, 0, 0, "UAB", "12-04-2001", "Madrid");
25        assertFalse(usuarioDAO.insertarUsuario(u));
26
27        //Caso de prueba para comprobar que el grupo no esta vacio
28        u = new Usuario(0, "manuel", "manuel123", "maindeprueba@gmail.com", 1, 0, 0, "", "12-04-2001", "Madrid");
29        assertFalse(usuarioDAO.insertarUsuario(u));
30
31        //Caso de prueba para comprobar que la fecha no esta vacia
32        u = new Usuario(0, "manuel", "manuel123", "maindeprueba@gmail.com", 1, 0, 0, "UAB", "", "Madrid");
33        assertFalse(usuarioDAO.insertarUsuario(u));
34    }
35}
```

Tenemos que indicar que usaremos la base de datos de test, y para cada función de la clase que testeamos, creamos una función de test, donde pondremos todos los casos a cubrir, y compararemos si los datos que retorna concuerdan con lo que debería retornar. Si hacemos click derecho sobre el fichero de test, y le damos a Run as < JUnit Test, se ejecutará el test.



En este caso sale en rojo, indicando que no ha pasado el test. Esto es debido a que en el código aun no tenemos implementadas las validaciones de los datos que introduce el usuario, por lo que de momento el test encuentra los errores.

Por último he hecho unos pequeños cambios en el sistema de logs:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
    <Target>System.out</Target>
    <encoder>
      <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n</pattern>
    </encoder>
  </appender>
  <appender name="file" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <!-- See also http://logback.qos.ch/manual/appenders.html#RollingFileAppender -->
    <!-- <File>log4j-application.log</File> -->
    <encoder>
      <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n</pattern>
    </encoder>
    <!--<rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
      <maxIndex>10</maxIndex>
      <FileNamePattern>log4j-application.log.%i</FileNamePattern>
    </rollingPolicy>-->
    <!--<file>${USER_HOME}/logfile.log</file>-->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <fileNamePattern>logs/aquejugamosLOG.%d.log</fileNamePattern>
    </rollingPolicy>
    <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
      <MaxFileSize>5MB</MaxFileSize>
    </triggeringPolicy>
  </appender>
  <root level="debug">
    <!--<appender-ref ref="stdout"/>-->
    <appender-ref ref="file"/>
  </root>
```

He modificado el XML para guardar un archivo log en una carpeta llamada logs, donde se creará un archivo de log diferente para cada día, y cada fichero tendrá como nombre aquejugamosLOG"fechaDeCuandoSeCreo".log, por lo que podremos buscar los logs por día mas facilmente.