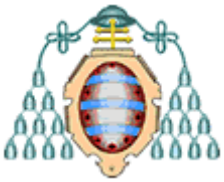
PATRONES

DAO

Ingeniería del Software

José Ramón de Diego



Contenido

- DATA ACCESS OBJECT

DAO:

Patrón definido por SUN e incluido en los J2EE Patterns

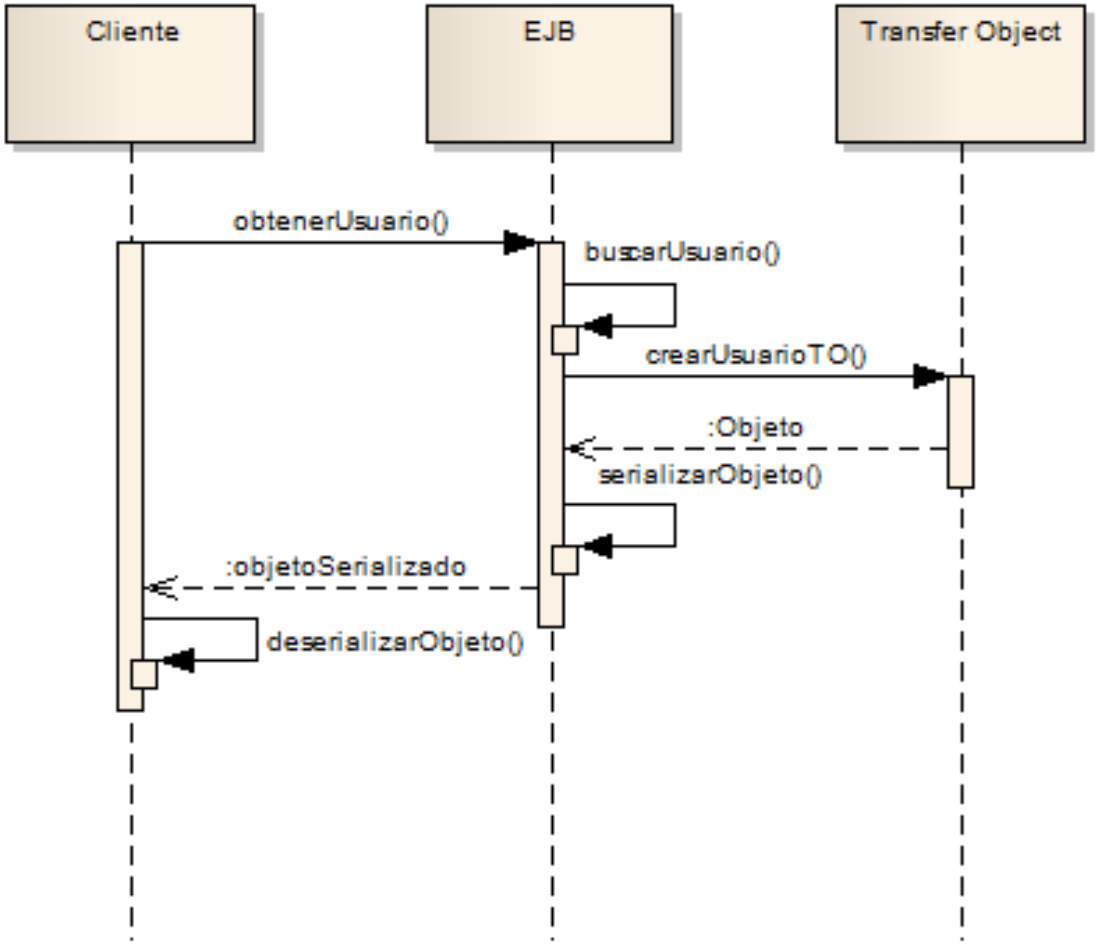
EJB:

J2EE Enterprise JavaBeans

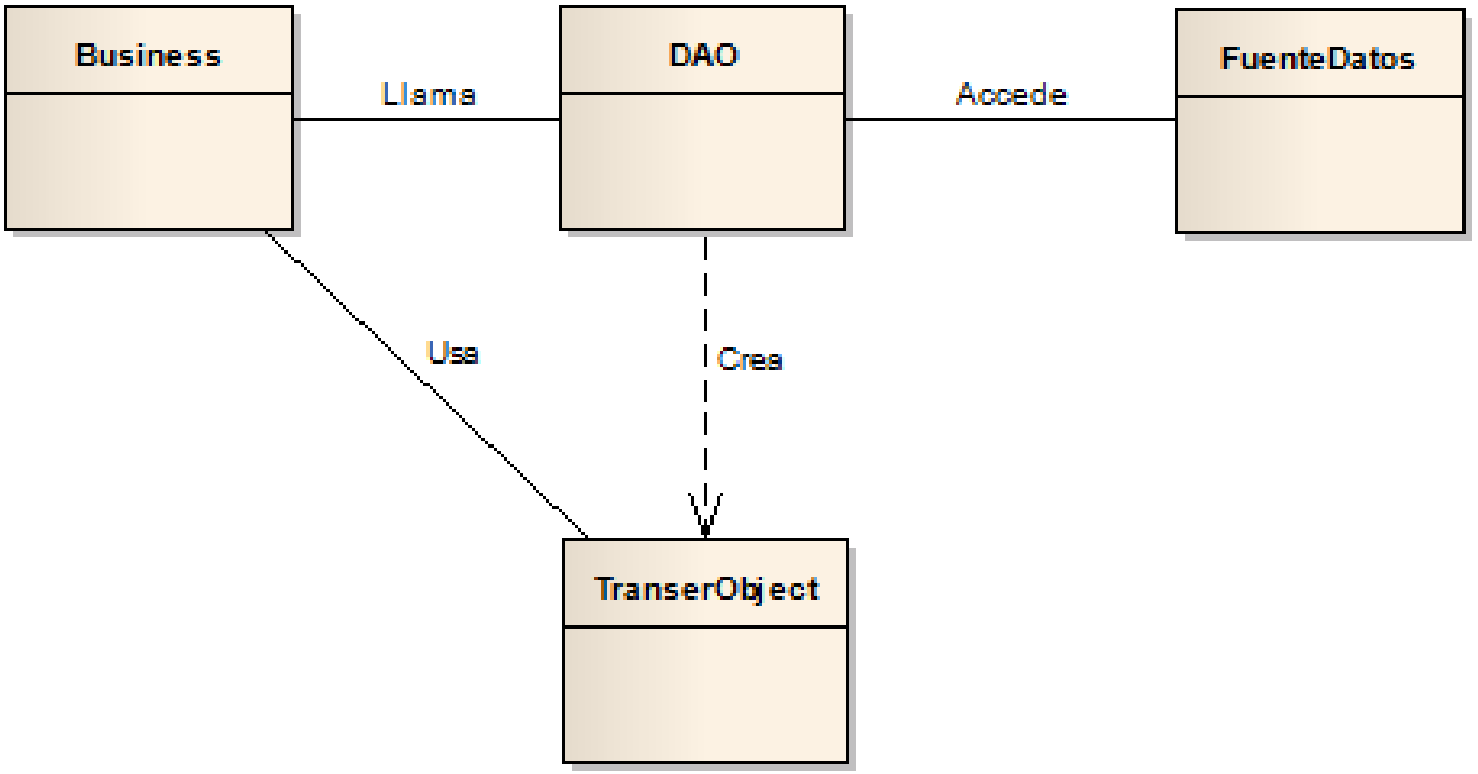
DTO:

Objeto para la transferencia de información

DAO:


Cliente
EJB
Transfer Object
obtenerUsuario()
buscarUsuario()
crearUsuarioTO()
:Objeto
serializarObjeto()
:objetoSerializado
deserializarObjeto()

DAO:


Business
Llama
DAO
Accede
FuenteDatos
Usa
Crea
TranserObject

DAO:

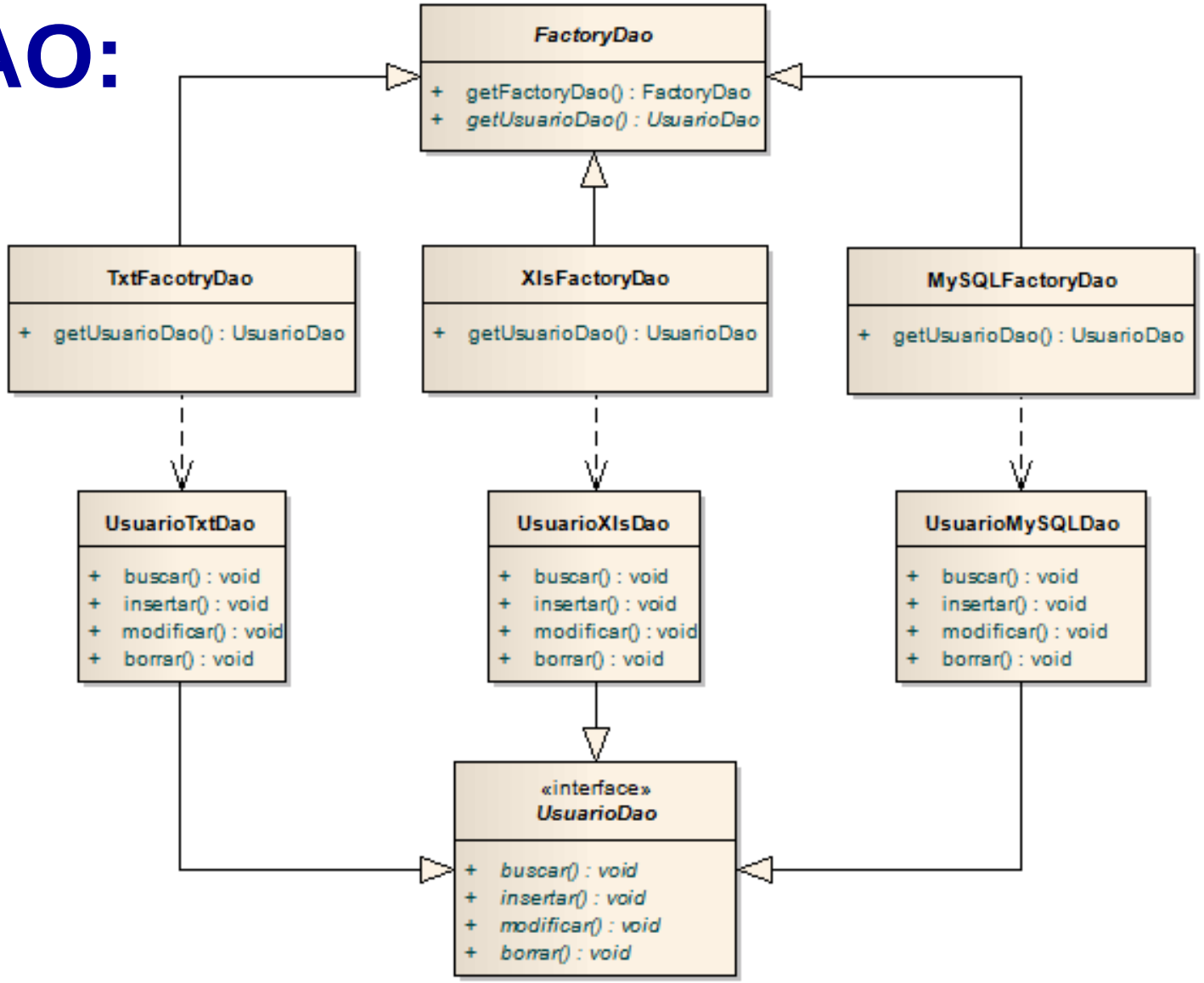
- Objetivo es almacenar información de alguna manera
- Para cada mecanismo de almacenamiento, existen múltiples formas de acceder a la fuente física
  - Al negocio no le debe afectar -> se debe de abstraer todo este conocimiento y envolverlo en una capa.
- DAO envuelve ese conocimiento, la explotación de los datos se hará mediante objetos DAO.
- La información encapsulada a un objeto de tipo TransferObject.
- Negocio solo recibe por medio de objetos la información que necesita

Múltiples Orígenes:

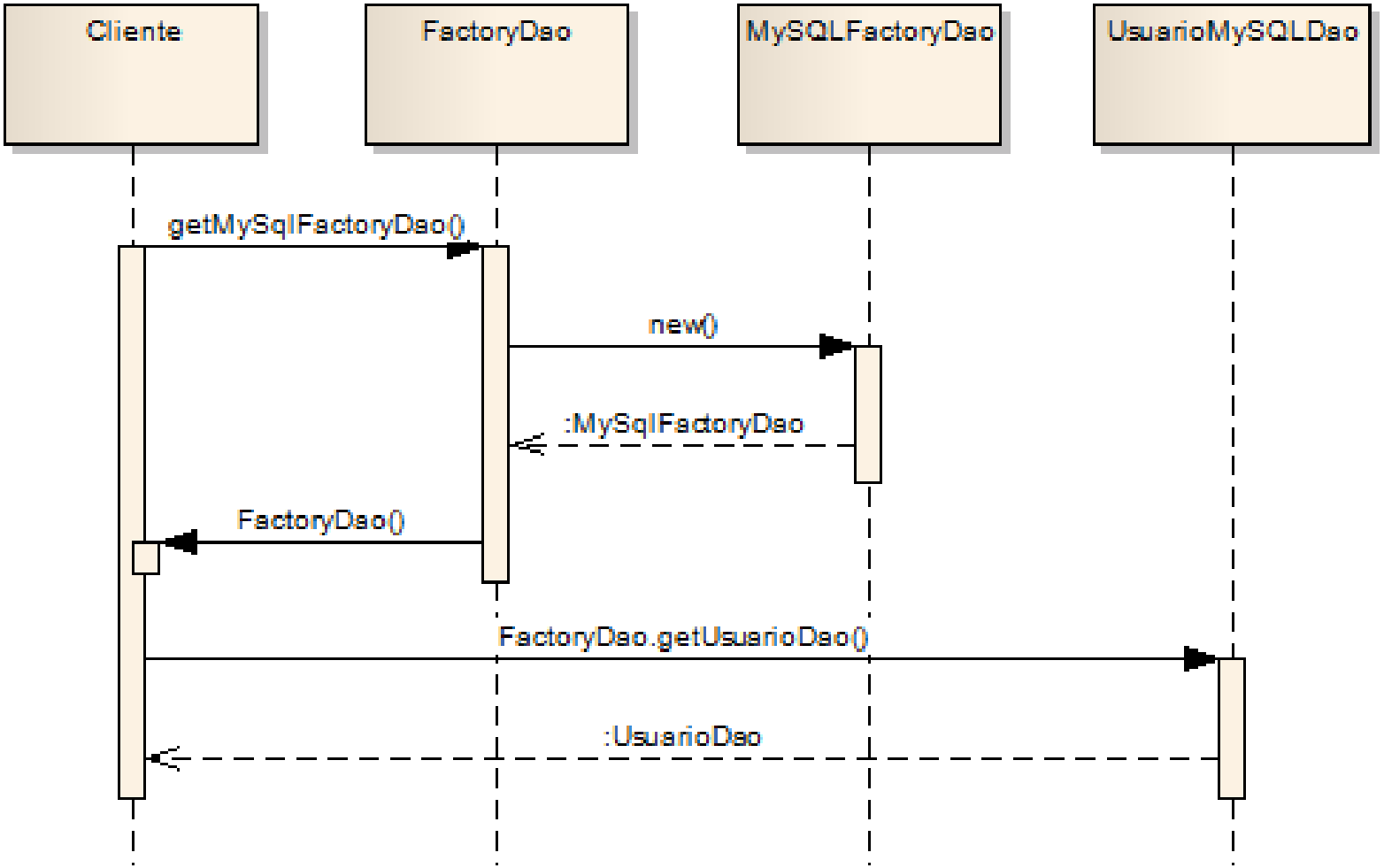
- Muchas fuentes de datos -> varios mecanismos para obtener la información
- Pequeño problema -> ¿Como accedo?

- La capa de negocio no debe enterarse de como se obtiene la información.
- El patrón Factory Method permite solucionar este problema
- Negocio solo sabe que debe recibir un TO que modela un Usuario y de donde quiere sacar la información. Hace esta elección por medio del FactoryDao

DAO:


FactoryDao
+ getFactoryDao() : FactoryDao
+ getUsuarioDao() : UsuarioDao
TxtFacotryDao
+ getUsuarioDao() : UsuarioDao
XlsFactoryDao
+ getUsuarioDao() : UsuarioDao
MySQLFactoryDao
+ getUsuarioDao() : UsuarioDao
UsuarioTxtDao
+ buscar() : void
+ insertar() : void
+ modificar() : void
+ borrar() : void
UsuarioXlsDao
+ buscar() : void
+ insertar() : void
+ modificar() : void
+ borrar() : void
UsuarioMySQLDao
+ buscar() : void
+ insertar() : void
+ modificar() : void
+ borrar() : void
«interface»
UsuarioDao
+ buscar() : void
+ insertar() : void
+ modificar() : void
+ borrar() : void

DAO:


Cliente
FactoryDao
MySQLFactoryDao
UsuarioMySQLDao
getMySqlFactoryDao()
new()
:MySqlFactoryDao
FactoryDao()
FactoryDao.getUsuarioDao()
:UsuarioDao

DAO:

La clase factory se encarga generar los objetos FactoryDao para cada una de las fuentes de datos

Esto lo hace mediante el método estático getFactory(int) y obliga a quien lo implemente a sobreescribir el método getUsuarioDao

```
public abstract class FactoryDao {

    public static final int TXT_FACTORY = 1;
    public static final int MYSQL_FACTORY = 2;

    public abstract UsuarioDao getUsuarioDao();

    public static FactoryDao getFactory(int claveFactory){
        switch(claveFactory){
            case TXT_FACTORY:
                return new TxtFactoryDao();
            case MYSQL_FACTORY:
                return new MySqlFactoryDao();
            default:
                throw new IllegalArgumentException();
        }
    }
}
```

DAO:

```
public interface UsuarioDao {

    UsuarioTO buscarUsuario(String nombre);
    void insertarUsuario(UsuarioTO usuario);
    void modificarUsuario(UsuarioTO usuario);

}
```

DAO:

```
public class MySqlFactoryDao extends FactoryDao {

    @Override
    public UsuarioDao getUsuarioDao() {
        return new UsuarioMySqlFactoryDao();
    }

}
```

Extiende la clase FactoryDao e implementa el metodo getUsuarioDao, generando un objeto que consuma el data source que necesita, en esta caso una base de datos MySQL

```
public class TxtFactoryDao extends FactoryDao{

    @Override
    public UsuarioDao getUsuarioDao() {
        return new UsuarioTXTFactoryDao();
    }
}
```

Extiende la clase FactoryDao e implementa el metodo getUsuarioDao, generando un objeto que consuma el data source que necesita, en esta caso una base de datos TXT

DAO:

```
public class UsuarioMySqlFactoryDao implements UsuarioDao {

    private ConnectionDao cn;

    public UsuarioMySqlFactoryDao() {
        cn = ConnectionDao.getInstance();
    }

    public UsuarioTO buscarUsuario(String nombre) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    public void insertarUsuario(UsuarioTO usuario) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    public void modificarUsuario(UsuarioTO usuario) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

Las operaciones que esta clase hace son dirigidas hacia una sola fuente de datos (Una Base de Datos en MySQL

DAO:

```
public class UsuarioTXTFactoryDao implements UsuarioDao {

    public UsuarioTXTFactoryDao() {

    }

    public UsuarioTO buscarUsuario(String nombre) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    public void insertarUsuario(UsuarioTO usuario) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    public void modificarUsuario(UsuarioTO usuario) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

Las operaciones que esta clase hace son dirigidas hacia una sola fuente de datos (Una Base de Datos en TXT

DAO:

```
public class UsuarioTO {

    private String nombre;
    private String apellido;
    private int edad;


    //Métodos set y get
}
```

DAO:

```
public class Cliente {

    private FactoryDao txtFactory =
        FactoryDao.getFactory(FactoryDao.TXT_FACTORY);

    private UsuarioDao usuarioDao = txtFactory.getUsuarioDao();

    public UsuarioTO buscarUsuario(String nombre){
        return usuarioDao.buscarUsuario(nombre);
    }

    public void insertarUsuario(String nombre, String apellido, int edad){
        UsuarioTO usuario = new UsuarioTO();
        usuario.setNombre(nombre);
        usuario.setApellido(apellido);
        usuario.setEdad(edad);
        usuarioDao.insertarUsuario(usuario);
    }

}
```

Cliente decide que fuente de datos usar por medio del método getFactory(), de el cual se puede obtener un objeto que implemente la interfaz UsuarioDao. El resto de las operaciones sobre UsuarioDao son muy transparentes para la clase cliente