Data Access Library

desde la perspectiva del cliente de la librería

gvSIG: Avanzando Juntos

http://www.gvsig.gva.es
http://www.gvsig.org

Jose Manuel Vivó Arnal (Chema)
Desarrollador v2.0
josemanuel.vivo@iver.es

Objetivos de la presentación:

Introducción a la Arquitectura DAL.

Conocer los servicios DAL para fuentes tabulares:

Enumeración de servicios

Introducción a los principales servicios.

¿Dónde encontrar documentación?

http://www.gvsig.org/web/docdev/gvsig_desktop_2_0/org.gvsig.fmap.dal/

Índice de la presentación:

1. Introducción y Contexto de la librería.
2. Descripción general de la arquitectura.
3. Descripción de la arquitectura para datos tabulares.
4. *Preguntas.*
5. Acceso a *Fenómenos* (Básico, Filtrado, Ordenación, Contexto).
6. *Preguntas.*
7. Modificación y escritura de un almacén.
8. *Preguntas.*
9. Consultar la estructura de un almacén.
10. Creación de un almacén nuevo.

Índice de la presentación:

¿Qué es DAL?

Data Access Library: denominación de la capa de accesos a dato de gvSIG.

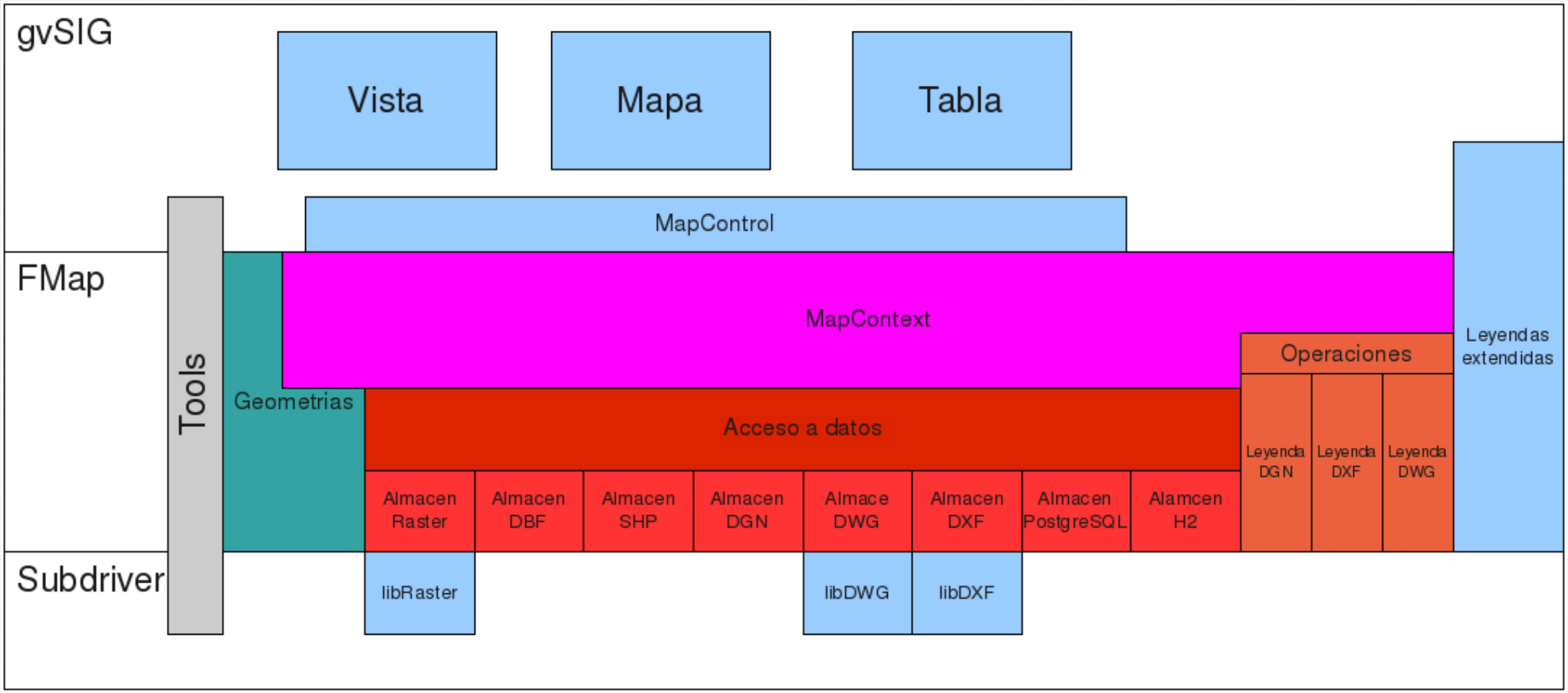
Pretende dotar de una capa de abstracción homogénea a las fuentes de datos para el núcleo de la aplicación.

Arquitectura pensada para ser flexible y robusta:

Desacoplamiento.

Trazabilidad.

Contexto de DAL



Descripción general de la arquitectura.

Bloques que componen la librería:

Application Programming Interface (API):

Servicios que se ofrecen al *consumidor* de la librería.

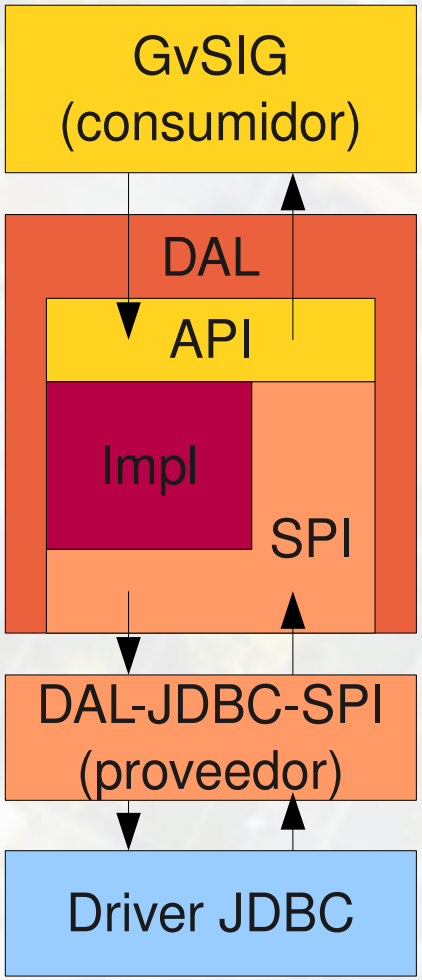
Service Provider Interface (SPI):

Servicios que deben ofrecer los *proveedores* de datos a la librería.

Implementación:

Parte que procesa la peticiones del *consumidor* usando los servicios de *proveedor*.

Descripción general de la arquitectura.



Como *consumidor* gvSIG sólo ve las interfaces de API.

La parte de la *implementación* aisla al *consumidor* de los detalles de la fuente de datos.

La *Implementación* prepara las peticiones al *proveedor* de forma estándar.

El *proveedor* se encarga de hacer las operaciones necesarias para procesar la petición y realizar la consulta de los datos a la *fuente de datos*, devolviéndolos de forma estándar.

La *implementación* ofrece los resultados de forma idéntica, independientemente de las características de la fuente.

Visión general del API

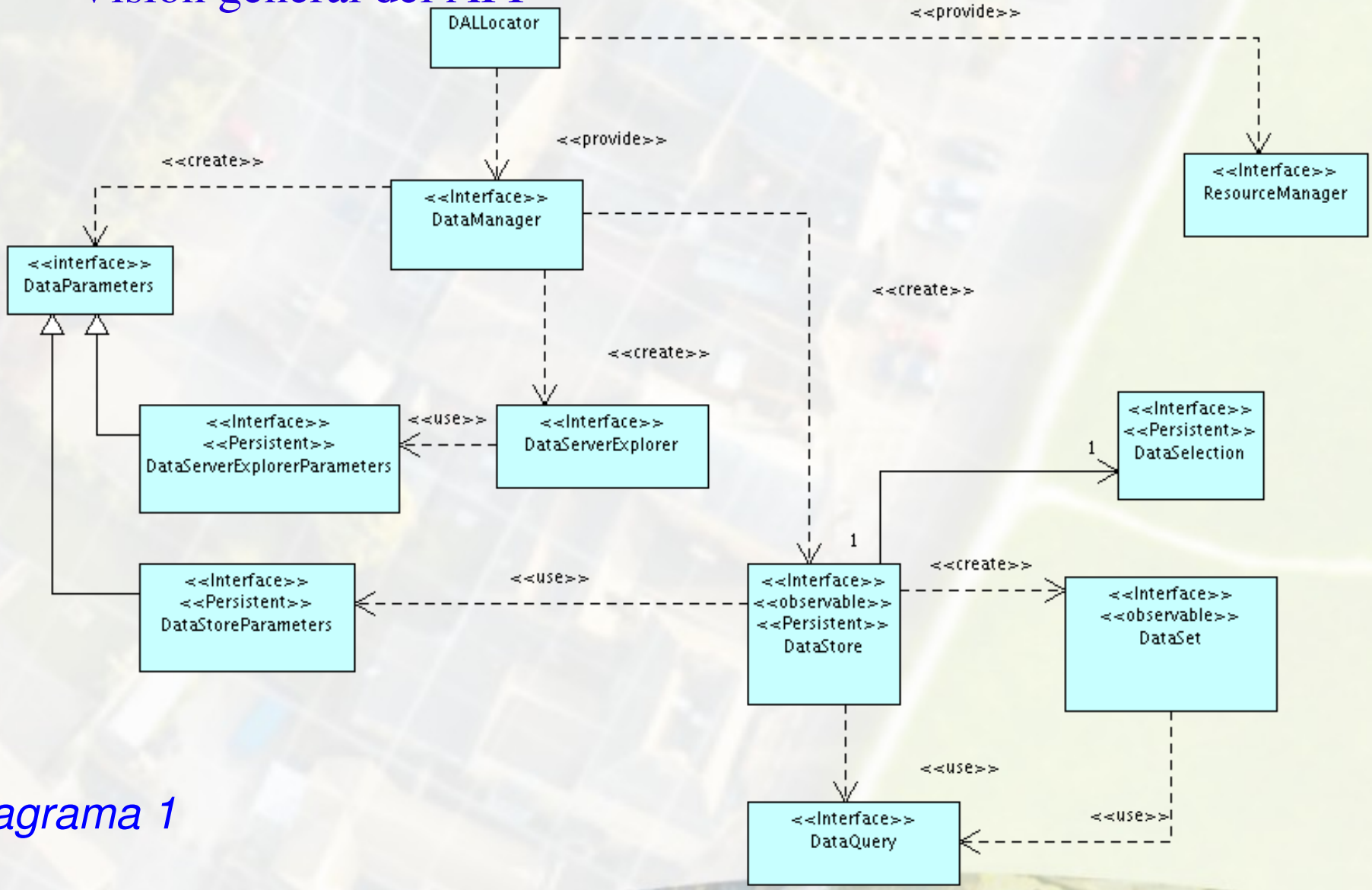


Diagrama 1

gvSIG_des_2.x_d: Curso de desarrolladores de gvSIG Desktop v 2.x

DAL desde la perspectiva del cliente de la librería.

Visión general de API para datos Tabulares

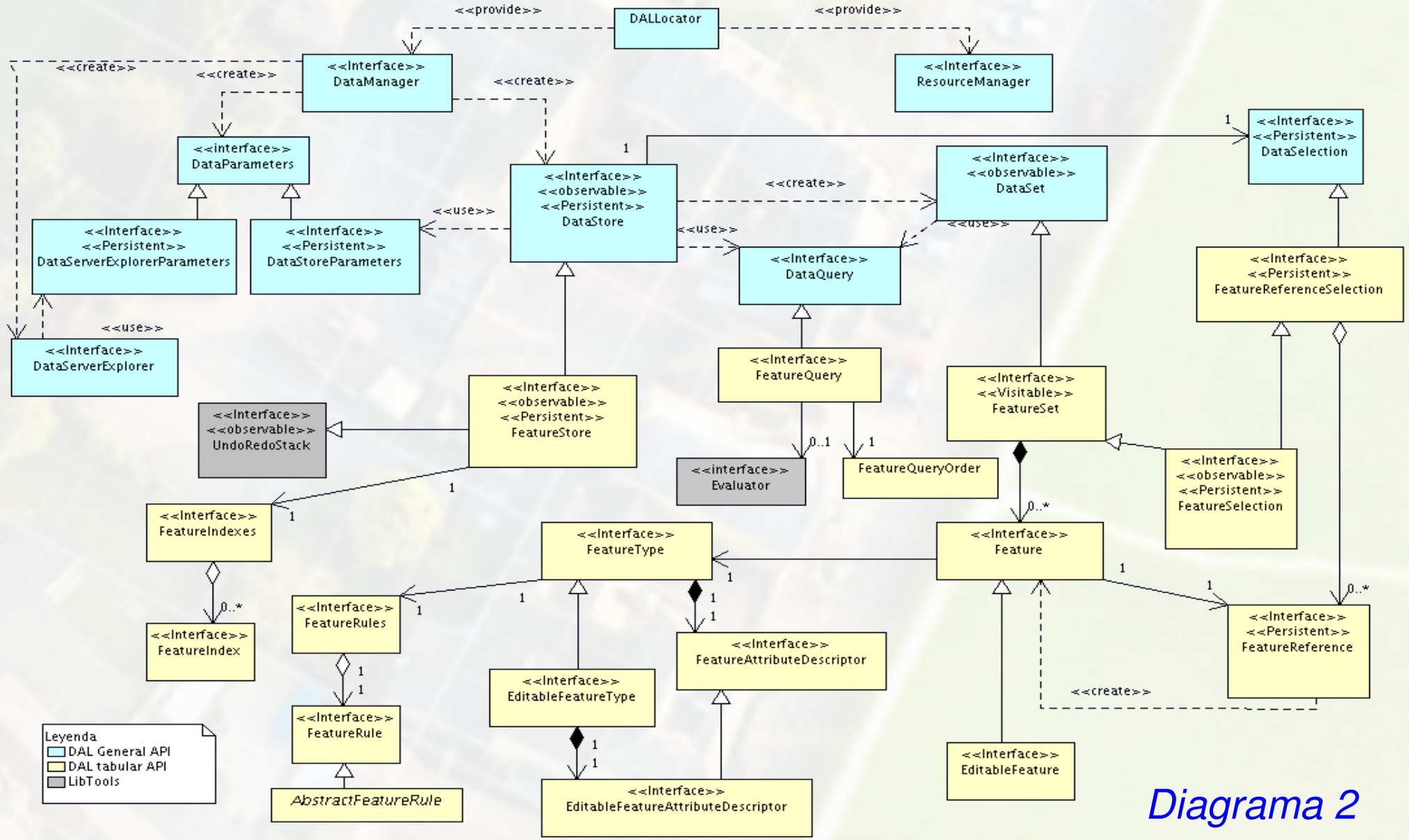


Diagrama 2

Ejemplo de acceso a un Shape y a una tabla PostgreSQL:

```
DataManager manager;
StoreParameters params;
FeatureStore store;
FeatureSet features;
Feature feature;

manager = DALLocator.getDataManager();
params = manager.createStoreParameters("Shape");
params.setDynValue("shpfilename","data/prueba.shp");

store = (FeatureStore)manager.createStore(params);
features = store.getFeatureSet();

DisposableIterator it = features.iterator();
while( it.hasNext() ) {
  feature = (Feature)it.next();
  System.out.println(feature.getString("NOMBRE"));
}

it.dispose();
features.dispose();
store.dispose();
```

Código fuente 1

```
DataManager manager;
StoreParameters params;
FeatureStore store;
FeatureSet features;
Feature feature;

manager = DALLocator.getDataManager();
params = manager.createStoreParameters("PostgreSQL");
params.setDynValue("host", SERVER_IP);
params.setDynValue("dbuser",SERVER_USER);
params.setDynValue("password",SERVER_PASWD);
params.setDynValue("dbname",SERVER_DBNAME);
params.setDynValue("table","prueba");

store = (FeatureStore)manager.createStore(params);
features = store.getFeatureSet();

DisposableIterator it = features.iterator();
while( it.hasNext() ) {
  feature = (Feature)it.next();
  System.out.println(feature.getString("nombre"));
}

it.dispose();
features.dispose();
store.dispose();
```

Código fuente 2

Descripción Arquitectura DAL

Preguntas

¿ ?

Acceso a Fenómenos: Básico

Explicación de ejemplo (Código Fuente 1):

1. Accedemos al *Manager* a través del *Locator*.

```
7  manager = DALLocator.getDataManager();
```

2. Creamos una instancia de los parámetros.

```
8  params = manager.createStoreParameters("Shape");
```

3. Rellenamos los datos necesarios para acceder al almacén.

```
9  params.setDynValue("shpfilename","data/prueba.shp");
```

4. Creamos el almacén.

```
10 store = (FeatureStore)manager.createStore(params);
```

5. Accedemos al conjunto (*Set)* de los *Fenómenos*.

```
11 features = store.getFeatureSet();
```

Acceso a Fenómenos: Básico

6. Pedimos un *Iterador* al conjunto de *Fenómenos*.

```
13 DisposableIterator it = features.iterator();
```

7. Iteramos.

```
14 while (it.hasNext()){
15   Feature feature = (Feature)it.next();
     ...
17 }
```

8. Liberamos los recursos que hemos usado.

```
18 it.dispose();
19 features.dispose();
20 store.dispose();
```

Acceso a Fenómenos: Básico

FeatureStore:

- *getFeatureTypes, getDefaultFeatureType*: aceso a la definición de la estructura del almacén.

- *getFeatureSet*: Crea un subconjunto de fenómenos.

- *getFeatureCount*: Cantidad total de fenómenos.

- *getParameters*: Parámetros del almacén.

- *dispose*: Libera recursos utilizados y prepara para su eliminación.

Acceso a Fenómenos: Básico

FeatureSet:

- *getFeatureTypes, getDefaultFeatureType*

- *getSize*: Cantidad de fenómenos en el subconjunto.

- *isEmpty*: Informa si hay fenómenos.

- *iterator*: Crea un nuevo iterador sobre el subconjunto.

- *fastIterator*: Crea un nuevo iterador sobre el subconjunto usando la misma instancia de *Feature*.

Acceso a Fenómenos: Básico

Feature:

- *getType*: devuelve la estructura del fenómeno.

- *get*: devuelve el valor de un atributo (por indicie o nombre).

- *getInt, getString, getBoolean, getDouble...*: devuelve el valor de un atributo usando el tipo especificado.

- *getArray*: devuelve el valor de un atributo de tipo Array.

- *getGeometry*: devuelve el valor de un atributo de tipo geométrico.

- *getDefaultGeometry*: devuelve el valor del atributo de tipo geométrico seleccionado por defecto.

Acceso a Fenómenos: Filtrado

Filtrado por tipo:

```
Iterator typeIt = store
   .getFeatureTypes().iterator();

while( typeIt.hasNext() ) {
  FeatureQuery query = store
     .createFeatureQuery();
  FeatureType type = (FeatureType)typeIt.next();
  query.setFeatureType(type);
  features = store.getFeatureSet(query);

  DisposableIterator it = features.iterator();
  while( it.hasNext() ) {
    feature = (Feature)it.next();
    System.out.println(
          feature.getString("NOMBRE"));
  }
  it.dispose();
  features.dispose();
}
```

Código fuente 3

Filtrado por expresión:

```
FeatureQuery query = store
   .createFeatureQuery();
Evaluator filter = manager
   .createExpresion("NOMBRE like 'a%'");
query.setFilter(filter);
features = store.getFeatureSet(query);

DisposableIterator it = features.iterator();
while( it.hasNext() ) {
  feature = (Feature)it.next();
  System.out.println(
        feature.getString("NOMBRE"));
}
it.dispose();
features.dispose();
```

Código fuente 4

Acceso a Fenómenos: Filtrado

Explicación de ejemplo (Código Fuente 4):

1. Creamos un *FeatureQuery* a través del *store*.

```
1  FeatureQuery query = store
2      .createFeatureQuery();
```

2. Creamos un *Evaluator* basado en una expresión.

```
3  Evaluato filter = manager
4      .createExpresion("NOMBRE like 'a%'");
```

3. Establecemos el filtro en la consulta.

```
5  query.setFilter(filter);
```

5. Accedemos al conjunto (*Set)* de los *Fenómenos* usando la definición de consulta que hemos construido.

```
6 features = store.getFeatureSet(query);
```

Acceso a Fenómenos: Filtrado

FeatureStore:

- *getFeatureSet(FeatureQuery)*: Crear un subconjunto de fenómenos según la consulta recibida.

- *getFeatureSet(FeatureQuery,Observer)*: Crear un subconjunto de fenómenos según la consulta recibida en segundo plano.

- *createFeatureQuery()*: Crear una instancia de definición de consulta.

FeatureQuery:

- *setFeatureType(FeatureType)*: Establece/Selecciona el tipo a usar.

- *get/setFeatureTypeId(String)*: *id* del tipo a usar.

- *get/setAttributeNames(String[])*: Lista de atributos a usar del tipo especificado.

- *get/setFilter(Evaluator)*: Filtro a aplicar.

- *hasFilter()*: Informa si se ha aplicado un filtro.

Acceso a Fenómenos: Ordenación

Ordenación por nombre:

```
FeatureQuery query = store
   .createFeatureQuery();
query.getOrder().add("CODIGO",true);
query.getOrder().add("NOMBRE",false);
features = store.getFeatureSet(query);

DisposableIterator it = features.iterator();
while( it.hasNext() ) {
  feature = (Feature)it.next();
  System.out.println(
       feature.getString("NOMBRE"));
}
it.dispose();
features.dispose();
```

Código fuente 5

Ordenación por expresión:

```
FeatureQuery query = store
   .createFeatureQuery();
Evaluator nombreToLower = manager
   .createExpresion("toLower(NOMBRE)");
query.getOrder().add("CODIGO",true);
query.getOrder().add(nombreToLower,false);
features = store.getFeatureSet(query);

DisposableIterator it = features.iterator();
while( it.hasNext() ) {
  feature = (Feature)it.next();
  System.out.println(
       feature.getString("NOMBRE"));
}
it.dispose();
features.dispose();
```

Código fuente 6

Acceso a Fenómenos: Ordenación

Explicación de ejemplo (Código Fuente 6):

1. Creamos un *FeatureQuery* a través del *store*.

```
1  FeatureQuery query = store
2      .createFeatureQuery();
```

2. Creamos un *Evaluator* basado en una expresión que transforma el atributo *NOMBRE* a minúsculas.

```
3  Evaluato nombreToLower = manager
4      .createExpresion("toLower(NOMBRE)");
```

3. Establecemos el orden en la consulta.

```
5  query.getOrder().add("CODIGO",true);
6  query.getOrder().add(nombreToLower,false);
```

4. Creamos un conjunto (*Set)* de *Fenómenos* usando la definición de consulta que hemos construido.

```
7 features = store.getFeatureSet(query);
```

Acceso a Fenómenos: Ordenación

FeatureQuery:

- *get/setOrder(FeatureQueryOrder)*: Establece un orden a partir de uno existente.

- *hasOrder()*: Informa si hay establecido un orden.

FeatureQueryOrder:

- *add(String, boolean)*: Añade un orden a partir de un nombre de atributo.

- *add(Evaluator, boolean)*: Añade un orden a partir de una expresión.

- *clear()*: Limpia el orden establecido.

- *remove*: Elimina un elemento del orden.

Acceso a Fenómenos: Contexto

El objeto FeatureQuery admite añadir parámetros de contexto para que el proveedor de datos pueda *optimizar* los datos resultantes dependiendo de la operación. Un ejemplo sería la *escala de visualización* en el pintado de la capa.

FeatureQuery:

- *get/setQueryParameter(String,Object)*: Devuelve/establece el valor de un parámetro de contexto.

- *get/setScale(double)*: Devuelve/establece el valor del parámetro *escala de visualización*.

Acceso a Fenómenos

Preguntas

¿?

Modificación y escritura de un almacén.

En este apartado veremos:

- Introducción.
- Servicios que ofrece el almacén para la edición.
- Servicios de modificación de un fenómeno.
- Inserción de nuevos fenómenos.
- Modificación de fenómenos ya existentes.
- Eliminación de fenómenos.
- Reglas de validación.

Modificación y escritura de un almacén: Introducción.

Un almacén tiene los siguientes modos:

MODE_QUERY: Modo de Consulta o estándar. No admite operaciones de edición.

MODE_FULLEDIT: Modo de edición completa. Admite operaciones de deshacer y rehacer. Admite consulta de datos. Los cambios se aplican en memoria y no persisten hasta finalizar. Soportado para todos los almacenes. Si se intenta persistir y el almacén es de sólo lectura habrá una excepción.

MODE_APPEND: Modo en *sólo inserción* de nuevo fenómenos. No admite deshacer/rehacer. No admite consultas. Los cambios se aplican *directamente en el proveedor*. No soportado por todos los almacenes.

Modificación y escritura de un almacén: Servicios del almacén para edición.

FeatureStore:

- *allowWrite*: Informa si es capaz de persistir cambios.
- *edit*: Entra en modo edición (Full o Append) (Full por omisión).
- *cancelEditing*: Finaliza edición cancelando los cambios.
- *finishEditing*: Finaliza edición persistiendo los cambios.
- *IsEditing/isAppending*: Informa del modo actual.
- *undo/redo*: deshacer/rehacer cambios.
- *insert/update/delete*: inserción/actualización/eliminación de fenómenos
- *createNewFeature*: creación de un nuevo fenómeno para rellenar.
- *IsAppendModeSupported*: Informa del soporte del modo *Append* en el almacén
- *begin/endEditingGroup*: gestionan agrupaciones de modificaciones (para undo/redo).

Modificación y escritura de un almacén: Servicios de modificación de un fenómeno.

El objeto Feature no da servicios de modificación, para ello debemos usar EditableFeature, el cual se hace a través de *FeatureStore.createNewFeature* o del *Feature.getEditable*

EditableFeature:

set: Estable el valor de un atributo.

setInt,setString,setDouble: Establece el valor de un atributo.

setDefaultGeometry: Establece el valor del campo marcado como geometría por defecto.

getSource: Devuelve la Feature original de la que se ha sacado la copia con el *Feature.getEditable*.

copyFrom: Copia los valore de los atributos usando los de otra Feature.

Modificación y escritura de un almacén: Validez de los Set y sus servicios de edición.

La modificación de un almacén provoca que los *FeatureSet* que estuviesen activos hasta el momento queden invalidados. Para evitar que el *FeatureSet* con el que trabajamos se invalide, debemos usar sus propios servicios de actualización

FeatureSet:

insert/update/delete: Realiza un cambio en el almacén origen de los datos, teniendo en cuenta los cambios en el propio *Set*.

Iterator:

remove: (*de un FeatureSet)* Elimina del almacén origen el último fenómeno accedido, teniendo en cuenta los cambios en el *Set* de origen.

Modificación y escritura de un almacén: Inserción de nuevos fenómenos.

```
store.edit();

EditableFeature feature = store
    .createNewFeature();
feature.set("NOMBRE", "Burjasot");
feature.set("TIPO", "MUNICIPIO");

store.insert(feature);
store.finishEditing();
```

Código fuente 7

Explicación del ejemplo:

1. Entramos en modo *Fullediting*.

```
1  store.edit();
```

2. Creamos un nuevo fenómeno.

```
3  EditableFeature feature =
store.createNewFeature();
```

3. Rellenamos los datos del fenómeno.

```
4  feature.set("NOMBRE","Burjasot");
```

4. Insertamos en el almacén.

```
7  store.insert(feature);
```

4. Finalizamos edición guardando los cambios.

```
8  store.finishEditing();
```

Modificación y escritura de un almacén: Modificación de fenómenos existentes.

```
store.edit();

EditableFeature feature;
FeatureQuery query = store.createFeatureQuery();
query.setFilter(manager.createExpresion("NOMBRE = 'Burjasot'"));
features = store.getFeatureSet(query);

Iterator it = features.iterator();
while( it.hasNext() ) {
  feature = ((Feature)it.next()).getEditable();
  feature.set("TIPO", "Municipio");
  features.update(feature);
}
features.dispose();

store.finishEditing();
```

Código fuente 8

Modificación y escritura de un almacén: Modificación de fenómenos existentes.

Explicación de ejemplo (Código Fuente 8):

1. Entramos en modo edición.

```
1  store.edit();
```

2. Creamos un *FeatureSet* con los fenómenos que queremos modificar.

```
6  features = store.getFeatureSet(query);
```

3. Pedimos al fenómeno su *Editable* y actualizamos el valor.

```
11 feature = ((Feature)it.next()).getEditable();
12 feature.set("TIPO", "Municipio");
```

4. Actualizamos el fenómeno en el *Set*.

```
13 features.update(feature);
```

5. Finalizamos edición guardando los datos.

```
17 store.finishEditing();
```

Modificación y escritura de un almacén: Eliminación de fenómenos.

```
store.edit();

FeatureQuery query = store.createFeatureQuery();
query.setFilter(manager.createExpresion("NOMBRE = 'Burjasot'"));
features = store.getFeatureSet(query);

Iterator it = features.iterator();
while( it.hasNext() ) {
  it.next();
  it.remove();
}
features.dispose();

store.finishEditing();
```

Código fuente 9

Modificación y escritura de un almacén: Eliminación de fenómenos.

Explicación de ejemplo (Código Fuente 9):

1. Entramos en modo edición.

```
1  store.edit();
```

2. Creamos un *FeatureSet* con los fenómenos que queremos modificar.

```
5  features = store.getFeatureSet(query);
```

3. Iteramos al siguiente elemento.

```
10 it.next();
```

4. Eliminamos el fenómeno actual desde el iterador.

```
11 it.remove();
```

5. Finalizamos edición guardando los datos.

```
13 store.finishEditing();
```

Modificación y escritura de un almacén: Reglas de validación.

La librería soporta la aplicación de reglas que validen la integridad de los datos de los almacenes.

Dichas reglas pueden ser aplicadas:

En la inserción o modificación de un fenómeno.

Al finalizar la edición.

En ambos casos.

La definición de una Regla de validación pasa por implementar el interface *FeatureRule*.

Las reglas se aplican sobre los *FeatureType*.

Modificación y escritura de un almacén.

Preguntas

¿ ?

Consultar la estructura de un almacén. Introducción.

Un almacén admite múltiples estructuras de datos.

Un almacén, por comodidad, declara una estructura a usar por defecto.

Las estructuras se definen por la clase *FeatureType*.

Un atributos de los que compone un *FeatureType* lo define un *FeatureAttributeDescriptor.*

Un *FeatureType* puede tener definido un atributo de tipo geometría para usar por defecto.

Consultar la estructura de un almacén. Principales servicios.

FeatureType:

- *size*: Número de atributos.
- *getIndex*: Índice de atributo por su nombre.
- *getDefaultGeometryAttributeName/Index:* atributo geométrico por defecto.
- *Iterator*: Iterador sobre los atributos.
- *get/getAttribute*: devuelve el atributo (por nombre o índice).
- *getId*: devuelve el identificador de la estructura.

Consultar la estructura de un almacén. Principales servicios.

FeatureAttributeDescriptor:

- *getName*: Nombre del atributo.
- *getDataType/getDataTypeName*: tipo del atributo.
- *getDefaultValue:* valor por defecto.
- *getIndex*: índice que ocupa.
- *getSize*: tamaño.
- *getEvaluator*: devuelve el evaluador que se usará para calcular un atributo calculado.
- *allowNull*: informa si admite el valor nulo.
- *isPrimaryKey*: informa si es parte de la clave primaria.
- *isReadOnly*: informa si es de sólo lectura.

Consultar la estructura de un almacén. Ejemplo.

```
FeatureType featureType = store.getDefaultFeatureType();
Iterator it = featureType.iterator();
while( it.hasNext() ) {
  attribute = (FeatureAttributeDescriptor)it.next();
  System.out.print(attribute.getDataTypeName();
  if( attribute.getSize() > 1 ) {
    System.out.print("["+attribute.getSize()+"]");
  }
  System.out.print(" " + attribute.getName() );
  Evaluator eval = attribute.getEvaluator();
  if( eval != null ) {
    System.out.print(
       attribute.getName()+
       ", "+
       attribute.getDataTypeName()+
       " -- Calculado "+ eval.getCQL());
  }
  System.out.println();
}
```

Código fuente 10

Creación de un almacén nuevo. Introducción a los exploradores.

Los servicios de consulta, creación y eliminación de almacenes de la librería ofrecen a través de los *ServerExplorer*.

Los *ServerExplorer* van más unidos al *soporte* de los datos que al tipo (*FileSystem, DB, Web)*.

Los *ServerExplorer* requieren sus propios parámetros, que pueden ser los mismo o un subconjunto de los necesarios para el almacén.

Creación de un almacén nuevo. Principales servicios de los exploradores.

DataServerExplorer:

- *canAdd*: informa si es posible la creación de almacenes.

- *canAdd(tipo)*: informa si es posible la creación de almacenes de un tipo en concreto (si soporta varios tipos).

- *list*: devuelve una lista de *DataParameters* de los almacenes disponibles en el servidor.

- *add*: Crea un nuevo almacén a partir de los parámetros recibidos

- *remove*: Elimina un almacén del servidor

- *getAddParameters*: crea un objeto para rellenar los parámetros necesarios para la creación de un almacén.

Creación de un almacén nuevo. Servicios definición de estructura.

EditableFeatureType:

- *add*: añade un nuevo atributo en base a los parámetros.
- *remove*: elimina un atributo.
- *setDefaultGeometryAttributeName*: establece el nombre del atributo geométrico que se va a usar por defecto.

EditableFeatureAttributeDescriptor:

- *setSize*: establece el tamaño del atributo.
- *setPrecision*: estable la precisión del atributo para los tipo decimales.
- *setPrimaryKey*: establece que forma parte de la clave primaria.
- *setAllowNull*: establece si el atributo admitirá valores nulos.

Creación de un almacén nuevo. Ejemplo.

```
DataExplorerParameters eparams = manager
    .createServerExplorerParameters("FilesystemExplorer");
eparams.setDynValue("initialpath","/data");
DataServerExplorer serverExplorer =
manager.createServerExplorer(eparams);

NewFeatureStoreParameters sparams =  (NewFeatureStoreParameters)
    serverExplorer.getAddParameters("DBF");
sparams.setDynValue("dbffilename","prueba.dbf");

EditableFeatureType featureType = (EditableFeatureType)
    sparams.getDefaultFeatureType();
featureType.add("NOMBRE", DataTypes.STRING,100);
featureType.add("MUNICIPIO", DataTypes.STRING,100);
featureType.add("POBLACION", DataTypes.LONG);
featureType.add("AREA", DataTypes.DOUBLE);

serverExplorer.add(sparams);
```

Código fuente 11

Creación de un almacén nuevo. Ejemplo.

Explicación de ejemplo (Código Fuente 11):

1. Creamos los parámetro para un explorador de ficheros.

```
1  DataExplorerParameters eparams = manager
2     .createServerExplorerParameters("FilesystemExplorer");
3  eparams.setDynValue("initialpath","/data");
```

2. Creamos un explorador de ficheros.

```
4  DataServerExplorer serverExplorer = manager
         .createServerExplorer(eparams);
```

3. Pedimos los parámetros al explorador para crear un almacén de tipo DBF.

```
6  NewFeatureStoreParameters sparams =  (NewFeatureStoreParameters)
7     serverExplorer.getAddParameters("DBF");
```

4. Rellenamos el parámetro de la ruta al nuevo fichero.

```
8  sparams.setDynValue("dbffilename","prueba.dbf");
```

Creación de un almacén nuevo. Ejemplo.

5. Definimos la estructura de los datos del almacén.

```
11 EditableFeatureType featureType = (EditableFeatureType)
12     sparams.getDefaultFeatureType();
13 featureType.add("NOMBRE", DataTypes.STRING,100);
14 featureType.add("MUNICIPIO", DataTypes.STRING,100);
   ...
```

6. Solicitamos al explorador que cree el almacén.

```
18 serverExplorer.add(sparams);
```

Modificación de la estructura de un almacén. Ejemplo

```
store.edit();
FeatureType featureType = store.getDefaultFeatureType();
EditableFeatureType editableFeatureType = featureType.getEditable();
editableFeatureType.getAttributeDescriptor("MUNICIPIO").setSize(50);
editableFeatureType.getAttributeDescriptor("NOMBRE").setSize(50);
store.update(editableFeatureType);
store.finishEdition();
```

Código fuente 12

Modificación de la estructura de un almacén. Ejemplo.

Explicación de ejemplo (Código Fuente 12):

1. Entramos en edición.

```
1  store.edit();
```

2. Accedemos al *FeatureType* por defecto.

```
2  FeatureType featureType = store.getDefaultFeatureType();
```

3. Accedemos a una copia *Editable* de la estructura.

```
3  EditableFeatureType editableFeatureType = featureType
       .getEditable();
```

4. Hacemos las modificaciones.

```
4  editableFeatureType.getAttributeDescriptor("MUNICIPIO")
       .setSize(50);
```

4. Actualizamos en el almacén.

```
6  store.update(editableFeatureType);
```

4. Finalizamos edición guardando los cambios.

```
7  store.finishEdition();
```

Crear almacén nuevo. Consulta y modificación de estructura del almacén.

Preguntas

¿?

Manejo de la selección de Fenómenos. Introducción.

Un almacén ofrece una selección de fenómenos.

La selección es una especialización de un *FeatureSet*.

Los *FeatureSelection* por defecto almacenan *FeatureReference*, pero los proveedores pueden ofrecer otra implementación.

Manejo de la selección de Fenómenos. Principales servicios.

FeatureStore:

- *getFeatureSelection*: selección actual.
- *createFeatureSelection*: crea un *set* de selección vacío.
- *setSelection:* estable una selección sustituyendo al actual.

FeatureReference:

- *getFeature*: devuelve la feature a la que hace referencia.

Feature:

- *getReference*: devuelve una instancia de *FeatureRefrence*.

Manejo de la selección de Fenómenos. Principales servicios.

FeatureSelection:

- *de/select*: de/selecciona un *Feature* o un *FeatureSet*.
- *isSelected*: informa si una *Feature* está seleccionada.
- *de/selectAll*: de/selecciona todos los elementos.
- *reverse*: invierte la selección.

Bloqueo de Fenómenos en el servidor.

La librería ofrece un interface para el bloqueo de fenómenos en el servidor.

Este servicio se delega en el proveedor de datos.

Los proveedores pueden no soportar este servicio.

Esta parte de la librería no está desarrollada y es susceptible a cambios.

Acceso a operaciones específicas del proveedor.

Los proveedores de datos puede ofrecer servicios específicos que no estén contemplados en el API.

A estos servicios se accede mediante los *DynMethods* a través del almacén.

Un ejemplo serían las leyendas estándar de los proveedores de CAD.

Se pueden explorar que métodos específicos del proveedor que tiene un almacén usando su *DynClass*.

Soporte para atributos calculados.

La librería ofrece la posibilidad de definir y añadir campos calculados al cualquier almacén.

Estos campos no persisten dentro de soporte del almacén.

Se calculan bajo demanda, pero sólo la primera vez.

Se usa una instancia de *Evaluator* para definirlo.

Se añaden en el almacén usando la edición.

Pueden añadirse en almacenes de sólo lectura.

Soporte para atributos calculados. Ejemplo

```
store.edit();
FeatureType featureType = store.getDefaultFeatureType();
EditableFeatureType editableFeatureType = featureType.getEditable();
editableFeatureType.add("DENSIDAD_POBLACION",
        DataTypes.DOUBLE,
        manager.createExpresion("POBLACION / AREA")
);
store.update(editableFeatureType);
store.finishEditing();
```

Código fuente 13

Soporte para atributos calculados. Ejemplo

Explicación de ejemplo (Código Fuente 13):

1. Entramos en edición.

```
1  store.edit();
```

2. Accedemos al *FeatureType* por defecto.

```
2  FeatureType featureType = store.getDefaultFeatureType();
```

3. Accedemos a una copia *Editable* de la estructura.

```
3  EditableFeatureType editableFeatureType = featureType
       .getEditable();
```

4. Añadimos el atributo basado en una expresión.

```
4  editableFeatureType.add("DENSIDAD_POBLACION",
5          DataTypes.DOUBLE,
6          manager.createExpresion("POBLACION / AREA")
7  );
```

4. Actualizamos en el almacén.

```
6  store.update(editableFeatureType);
```

4. Finalizamos edición guardando los cambios.

```
7  store.finishEdition();
```

Soporte para atributos calculados. Ejemplo con atributo personalizado.

```
class XY2Geometry extends AbstractEvaluator {
     private String xname;
     private String yname;
     private GeometryManager geomManager;

     public XY2Geometry initialize(String xname, String yname) {
        this.xname = xname;
        this.yname = yname;
        geomManager = GeometryLocator.getGeometryManager();
        return this;
    }
```

Continua...

Código fuente 14

Soporte para atributos calculados. Ejemplo con atributo personalizado.

Continua...

```
    public Object evaluate(EvaluatorData data) throws
            EvaluatorException {
        Double x = (Double) data.getDataValue(this.xname);
        Double y = (Double) data.getDataValue(this.yname);
        Geometry geom;
        try {
            geom = geomManager.createPoint(x.doubleValue(), y
                .doubleValue(), Geometry.SUBTYPES.GEOM2D);
        } catch (CreateGeometryException e) {
            throw new EvaluatorException(e);
        }
        return geom;
    }

    public String getName() {
        return "XY2Geometry";
    }
}
```

Código fuente 14

Soporte para atributos calculados. Ejemplo con atributo personalizado.

Para aplicar el atributo que hemos definido, usaremos lo siguiente:

```
store.edit();
FeatureType featureType = store.getDefaultFeatureType();
EditableFeatureType editableFeatureType = featureType.getEditable();
editableFeatureType.add("GEOM",
  DataTypes.GEOMETRY,
  new XY2Geometry().initialize("X", "Y")
);
store.update(editableFeatureType);
store.finishEditing();
```

Código fuente 15

Soporte para atributos calculados.

Preguntas

¿ ?

Pila de comandos de edición.

En el modo de edición *FULLEDITING* hay soporte para deshacer y rehacer las modificaciones que se aplican al almacén.

Estas acciones se almacenan en forma de comandos en una pila.

La información de esta pila está accesible a través del almacén.

Puede que existan acciones en las que no sea posible deshacer.

Soporte para índices.

La librería ofrece el API y SPI para el uso e implementación de índices de datos *locales*.

Actualmente hay varias implementaciones de índices para atributos de tipo Geométrico.

Las implementaciones de índices pueden ser en memoria, disco....

Las búsquedas y ordenaciones usan los índices de forma automática, si son aplicables, una vez generados.

Un almacén puede tener creados uno o varios índices sobre cada una de las estructuras de almacén.

Transformación de un almacén.

Transformación es un algoritmo que a partir de un fenómeno de un tipo produce otro fenómeno de otro tipo distinto, sin modificar el original.

Se pueden *apilar* varias transformaciones sobre un almacén.

Un almacén *transformados* es de sólo lectura.

Las transformaciones deben implementar el interface *FeatureStoreTransform*.

Transformación de un almacén. Ejemplo.

```
public class MyTransform extends AbstractFeatureStoreTransform {
    private String geomName;
    private String xname;
    private String yname;
    private GeometryManager geomManager;

    public MyTransform() {
      this.geomManager = GeometryLocator.getGeometryManager();
    }

    public MyTransform initialize(FeatureStore store,
                  String geomName,
                  String xname, String yname) throws DataException {
      setFeatureStore(store);
      this.geomName = geomName;
      this.xname = xname;
      this.yname = yname;
      EditableFeatureType type = getFeatureStore()
                .getDefaultFeatureType().getEditable();
```

Código fuente 16

Continua...

Transformación de un almacén. Ejemplo.

Continua...

```
        type.add(geomName, DataTypes.GEOMETRY);
        List list = new ArrayList(1);
        list.add(type.getNotEditableCopy());
        setFeatureTypes(list, (FeatureType) list.get(0));
        return this;
    }

    public void applyTransform(Feature source,
                    EditableFeature target) throws DataException {
        target.copyFrom(source);
        Geometry geom;
        try {
            geom = geomManager.createPoint(
                    source.getDouble(xname),source.getDouble(yname),
                    Geometry.SUBTYPES.GEOM2D);
        } catch (CreateGeometryException e) {
            throw new ReadException("XYTranform", e);
        }
```

Código fuente 16

Continua...

Transformación de un almacén. Ejemplo.

Continua...

```
        target.setGeometry(this.geomName, geom);
    }

    public void saveToState(PersistentState state) throws
                    PersistenceException {
        state.set("xname", xname);
        state.set("yname", yname);
        state.set("geomName", geomName);
    }

    public void loadFromState(PersistentState state)
                    throws PersistenceException {
        this.xname = state.getString("xname");
        this.yname = state.getString("yname");
        this.geomName = state.getString("geomName");
  }
```

Continua...

Código fuente 16

Transformación de un almacén. Ejemplo.

Continua...

```
    public FeatureType getSourceFeatureTypeFrom(
                        FeatureType targetFeatureType) {
            EditableFeatureType ed = targetFeatureType.getEditable();
            ed.remove(this.geomName);
            return ed.getNotEditableCopy();
    }

    public boolean isTransformsOriginalValues() {
            return false;
    }
}
```

Código fuente 16

Soporte para atributos calculados. Ejemplo con atributo personalizado.

Para aplicar la transformación anterior, usaremos lo siguiente:

```
FeatureStoreTransform transform = new MyTransform()
        .initialize(store,"geom","x","y");
store.getTransforms().add(transform);
```

Código fuente 17

Preguntas

¿ ?

Gracias por su atención.

gvSIG. Geographic Information System of the Valencian Government