

FeatureVisitor

Now that we have a decent FeatureCollection interface (well not indecent, you could take it home to meet mom), and more importantly a FeatureCollection interface that is **used** - it's time to have some fun.

There are two ways to go about hacking at features these days: FeatureReader (and Iterator) or as an entire FeatureCollection (formally FeatureResults)

What is the Point

I would like to set up a FeatureVisitor extension for making summary calculations. I am going to have to define the API for uDig developers anyways, they already want to know the number "distinct" values for an attribute, or min/max information etc ...

This is needed to make SLD documents (I mean SymbologyEncoding documents) and that involves giving the users a bit of a summary of what is going on - histogram, unique values, ranges, statistical breakdown - or something.

Note: James already has the problem licked for GeoVista

The bad old days

In the bad old days we would need to write a for loop to do something like "count" the number of features.

```

URL url =
SampleTest.class.getResource("bc_elections_2
000.shp");
DataStore shapefile = new
ShapefileDataStore( url );
String typeName =
shapefile.getTypeNames()[0];
FeatureSource source =
shapefile.getFeatureSource( typeName );

FeatureCollection collection =
source.getFeatures();
int count = 0;
Iterator iterator;
try {
    for( iterator = collection.iterator();
iterator.hasNext(); ){
        Feature feature = (Feature)
iterator.next();
        count++;
    }
} finally {
    collection.close( iterator );
}
System.out.println("Count:"+count);

```

There are a couple things wrong with this picture:

- it prescribes an order (sequential or otherwise) through the data

- it gives us no chance to turn that lovely "count" into naked SQL for speed and profit

In a sense it depends on the internal structure of the FeatureCollection and by showing us too much of the guts, takes away the FeatureCollection implementors freedom to do magic stuff.

Ah Magic Stuff

What kind of magic stuff?

- like splitting the collection in two and processing half on each processor
- picking up the visitor and processing over on the server side
- abusing the internal structure of the collection (perhaps there is an index?)

The amount of magic you can do really depends on the abilities of your FeatureCollection, different magic is available for an indexed shapefile vs a local HSQL Database. You may need to balance the amount of data collected against any distribution that may be in play (consider a geometric "buffer" operation performed on a remote PostGIS).

So yeah Visitor Pattern

Normally when you want to hide internal structure you break out the visitor pattern ... see GOF.

```
interface FeatureVisitor {  
    public void visit( Feature feature )  
}
```

★ Yes that really is just the inside of the for loop}

Sample use:

```
aFeatureCollection.accepts( new  
FeatureVisitor() {  
    public void visit( Feature feature ){  
System.out.println( feature ); }  
});
```

- Note: To try this out we will need `FeatureCollections.visit(FeatureCollection, FeatureVisitor)`

Now there is a small wrinkle - when breaking apart work, and merging it together again you should use a "Collector" - the one most people are familiar with are `TestResults` from the JUnit infrastructure. You can see this idea already in the `ValidationResults` (if you want an example in `GeoTools`).

By isolating the traversal of the datastructure and reporting of results, it buys us a lot of freedom. So much freedom we can just pretend to visit, as long as we produce the right results nobody will know we generated some SQL on the fly.

Where is this Going?

From email:

```
interface FeatureCalculate extends
FeatureVisitor {
    Object getValue();
}
class Sum implements FeatureCalculate {
    Expression expr;
    int total=0;
    public Sum( String attribute ){
        this(
FeatureTypeFactory.getDefault().createAttrib
uteExpression( attribute ) );
    }
    public Sum( Expression expression ){
        expr= expression;
    }
    public void visit( Feature feature ){
        Number number = ((Number)feature.get(
xpath ));
        total += number.intValue();
    }
    public int getTotal(){ return total; }
    public Object getValue(){ return new
```

```
Integer( total ); }  
}  
class Count implements FeatureCalculate {  
    int count=0;  
    public void visit( Feature feature ){  
count++; }  
    public int getCount(){ return count; }  
}
```

```
public Object getValue(){ return new
Integer( count ); }
}
```

...

```
FilterFactory ff =
FilterFactory.createFilterFactory();
Expression attrExpression =
ff.createAttributeExpression("cities",
aFeatureType);
FeatureVisitor uniqueVisitor = new
UniqueVisitor(attrExpression);
aFeatureCollection.accepts(uniqueVisitor);
Set value1 =
uniqueVisitor.getResult().toSet();
```

or alternatively:

```
Expression attrExpression =
ff.createAttributeExpression(null,
"featureMember/*/foo");
FunctionExpression funcSum =
ff.createFunctionExpression("Collection_Sum"
);
funcSum.setArgs(new Expression[] {
attrExpression });
CalcResult sumResult = (CalcResult)
funcSum.getValue(aFeatureCollection);
double sum = sumResult.toDouble();
System.out.println( "Sum = " + sum );
```

or even better:

```
ExpressionBuilder eb = new
ExpressionBuilder();
FunctionExpression expr =
(FunctionExpression)
eb.parser("Collection_Sum(foo)");
double sum = ((Number)
expr.getValue(aFeatureCollection)).doubleValue();
System.out.println( "Sum = " + sum );
```

and we can do more fun stuff too:

```
...
CalcResult sumResult = (CalcResult)
sumCalc.getValue(aFeatureCollection);
CalcResult countResult = (CalcResult)
countCalc.getValue(aFeatureCollection);
CalcResult averageResult =
sumResult.merge(countResult);
System.out.println( "Average = " +
averageResult.toDouble() );
```

Other stuff that Jody wrote that hasn't been implemented yet:

```
Sum length = new Sum( "length" );
Count count = new Count() ;
aFeatureCollection.accepts( new
FeatureVisitor[] { length, count } );
System.out.println( "Average length:" +
length.getTotal()/count.getCount() );
```

Pros:

- replaces FeatureResults getCount() aFeatureSource.features(Filter filter).visit(new Count())
- replace getBounds() can do the same kind of thing
- Visitor is also easier to optimize if you have multiple processors ...

Cons:

- hard to optimize into SQL