Pain Free GeoTools

This article is in response to an excellent observation made by Martin Fowler here:

- Humane Interface - *design the interface so that it's really easy to do the common case*
- Minimal Interface - *to the smallest reasonable set of methods that will do the job*

This articles covers how to use Builders and Utility classes to make geotools easy.

Minimal Interfaces for Interoperability

As discussed elsewhere, geotools makes use of Interfaces and Factories **everywhere**. To ease the burden on implementors we try for the minimal set of methods that will do the job. We also aggressively take this approach with standards, fostering off definition (and associated careful documentation duties) on the GeoAPI project whenever we can.

GeoAPI:

```
public interface Crosses extends BinarySpatialOperator {
}
public interface BinarySpatialOperator extends SpatialOperator {
    Expression getExpression1();
    Expression getExpression2();
}
public interface SpatialOperator extends Filter {
}
public interface Filter {
    boolean evaluate(Feature feature);
    Object accept(FilterVisitor visitor, Object extraData);
}
```

So they don't line up yet, gives us something to work towards (or away from).

GeoTools:

```
public interface GeometryFilter extends Filter {
    void addRightGeometry(Expression rightGeometry) throws IllegalFilterException;
    void addLeftGeometry(Expression leftGeometry) throws IllegalFilterException;
    boolean contains(Feature feature);
    Expression getRightGeometry();
    Expression getLeftGeometry();
}
public interface Filter extends FilterType {
    boolean contains(Feature feature);
    short getFilterType();
    void accept(FilterVisitor visitor);
    Filter and(Filter filter); // funny!
    Filter or(Filter filter);  // funny!
    Filter not();              // funny!
}
public interface FilterType {
    public static final short GEOMETRY_CROSSES = 9;
}
```

Violating Minimal Interfaces

What happens when you step away from the minimal interface? Let's consider the ones marked as "Funny" 🙂

To start with these methods involve creation, they are out of place in what is otherwise a API focused on functionality (ie capturing what is needed for interoperability). This places a burden on implemententors that is rather heavy - creation is a difficult problem!

To wit: the implementation will need to keep the Factory that created it - so it can implement these methods.

But what about Usability

It is true that the resulting interfaces are sometimes crazy to use (try making a CRS by hand, or changing something in a FeatureType). You will go crazy tring to do it once, you would be crazy to do it twice.

To address this need we have our utility classes:

```
SLD.setSymbolizer( symbolizer, Color.Red );
```

```
CoordinateReferenceSystem crs = CRS.decode("EPSG:4326");
```

```
JTS.toGrographic( bbox, crs );
```

```
FeatureType type = DataUtilities.createType( "road",
"name:String,id:0,geom:MultiLineString" );
FeatureType subtype = DataUtilities.createType( type, "geom" );

FeatureReader reader = DataUtilities.reader( new Feature[]{ feature1, feature2} );
```

Right now most of these utility classes consist of **static final** methods that do the right thing.

What about Builders?

The use of Builders is closely related. These, by definition, are classes that put a pretty face on creation. As such they often provide a humane interface beyond that available through a the minimal API required for interoperability.

Here is how Builders are different: they are used specifically to ease the creation process. Often they appear as a pretty **mutable** face on things that are hard to construct or immutable. StringBuffer is a builder of Strings, FeatureTypeBuilder is a builder of FeatureTypes. Builders that construct complicated content in this manner are **stateful**.

```
ExpressionBuilder().parser( "[name = \"Douglas\"]" );
```

Going forward with Utility classes

One of the things that is happening as we fix up the toolkit and make it consistent is that these classes are moving away from being full of static final methods....

```
ExpressionBuilder expr = ExpressionBuilder();
Filter filter = (Filter) expr.parser( "[name = \"Douglas\"]" );
```

Why you ask? Because it turns out **most** of the methods actually need to do something and to do something you need to be connected up with GeoTools factories, a possible open-ended set of factories...

```
ExpressionBuilder expr = ExpressionBuilder( myFilterFactory );
Filter filter = (Filter) expr.parser( "[name = \"Douglas\"]" );
```

Before

So lets look at the SLD utility class as an example.

```
public class SLD {
    /** <code>NOTFOUND</code> indicates int value was unavailable */
    public static final int NOTFOUND = -1;
    public static final StyleBuilder builder = new StyleBuilder();
    /**
     * Sets the Colour for the given Line symbolizer
     *
     * @param symbolizer
     * @param colour
     */
    public static void setLineColour(LineSymbolizer symbolizer, Color colour) {
        if (symbolizer == null) {
            return;
        }

        Stroke stroke = symbolizer.getStroke();

        if (stroke == null) {
            stroke = builder.createStroke(colour);
            symbolizer.setStroke(stroke);
        }

        if (colour != null) {
            stroke.setColor(builder.colorExpression(colour));
        }
    }
}
```

This is a good example because:

- It shows using a Builder to aid in construction of something difficult
- StyleBuilder itself makes use of two Factories: StyleFactory and FilterFactory

This is a bad example because:

- StyleBuilder does not actually hold state (it only consists of "nicer" create methods)
- as such it is probably misnamed

Chaining utility classes together like this allows us to minimize the need for explicitly showing a cascade of factory injection.

The practice makes no difference to whoever is setting us up, and even represents less work for them.

After

To make SLD play nice with others we need to ensure that others can play with SLD. Playing is called dependency injection (SLD will depend on them to do some work after all). Setting this up is done in two ways.

```
public class SLD {
    /** <code>NOTFOUND</code> indicates int value was unavailable */
    public static final int NOTFOUND = -1;
    public StyleBuilder builder;

    public class SLD(){
        this( new StyleBuilder() );
    }
    /** Constructor injection */
    public SLD( StyleBuilder builder ){
        this.builder = builder;
    }
    /** Setter injection */
    public void setStyleBuilder( StyleBuilder builder ){
        this.builder = builder;
    }

    /**
     * Sets the Colour for the given Line symbolizer
     *
     * @param symbolizer
     * @param colour
     */
    public void setLineColour(LineSymbolizer symbolizer, Color colour) {
        if (symbolizer == null) {
            return;
        }

        Stroke stroke = symbolizer.getStroke();
        if (stroke == null) {
            stroke = builder.createStroke(colour);
            symbolizer.setStroke(stroke);
        }

        if (colour != null) {
            stroke.setColor(builder.colorExpression(colour));
        }
    }
}
```

Oops!

You may have noticed that the above would of broken a lot of code - here is the missing link:

```
static final SLD sld = new SLD(); // configured w/ default factories

/**
 * Configure default SLD for toolkit.
 * @deprecated Force default SLD, warning you make the tookit unstable!
 */
public static void setSLD( SLD sld ){
    this.sld = sld;
}
/**
 * Sets the Colour for the given Line symbolizer using "default" factories.
 *
 * @depricated Please use new SLD().setLineColor( symbolizer, color )
 * @param symbolizer
 * @param colour
 */
public static final void setLineColour(LineSymbolizer symbolizer, Color colour) {
    return sld.setLineColour( symbolizer, colour );
}
```

This preserves backwards compatibility, it is deprecated (so it will be removed in a future release). It does allow a hook for the default to be forced by client application code, but since this cross cuts the entire VM it is not recommended.

Summing Up

GIS is a tough nut to crack, often the task is to make things work at all. If you get a chance try and provide assistance to those using the toolkit. This article illustrates how this can be done without consequence to interoptability.

*An alternative approach would be to put all the fun convenience methods into the implementations; this would be dangerous **if** we accidently started using them internally - and it would have the negative consequence of locking users into a specific implementation.*

The relationship between minimal interfaces and usability is a difficult balance, the use of utility classes can act as a wrapper around the problem. Wrap up an implementation with a utility class for ease of use, just be sure to allow for the network of required factories.

In the explicit case of creation Builders often allow for a more Humane (and sometimes the only) take on the situtation.