GeoTools


# Style Workbook

*FOSS4G 2009 Geospatial for Java Tutorials*

27 September 2009

*Jody Garnett*
*Michael Bedward*

# Table of Contents

# 1 Welcome

Welcome to Geospatial for Java -this workbook is aimed at Java developers who are new to geospatial and would like to get started.

You should be sure to have a GeoTools development environment set up and ready to go. For those using maven we will start off each section with the dependencies required.

This workbook practises a “Code First” idea – give the code examples a try; and feel free to read the background information if you have any questions. This workbook covers how to create a Style object by hand as a programmer; how to parse a Style object from an SLD file.

This workbook is part of the FOSS4G 2009 conference proceedings.

**Jody Garnett**

Jody Garnett is the lead architect for the uDig project; and on the steering committee for GeoTools; GeoServer and uDig. Taking the roll of geospatial consultant a bit too literally Jody has presented workshops and training courses in every continent (except Antarctica). Jody Garnett is an employee of LISAsoft.

**Michael Bedward**

Michael Bedward is a researcher with the NSW Department of Environment and Climate Change and an active contributor to the GeoTools users' list. He has a particularly wide grasp of all the possible mistakes one can make using GeoTools.

# 2 Style Lab

To start out with we are going to get something on screen; so we can look at what a Style is then we can work on creating them by hand.

1. To start out with you will need to be sure to include the following dependencies.

```
<dependencies>
    <dependency>
        <groupId>org.geotools</groupId>
        <artifactId>gt-main</artifactId>
        <version>${geotools.version}</version>
    </dependency>
    <dependency>
        <groupId>org.geotools</groupId>
        <artifactId>gt-shapefile</artifactId>
        <version>${geotools.version}</version>
    </dependency>
    <dependency>
        <groupId>org.geotools</groupId>
        <artifactId>gt-epsg-hsql</artifactId>
        <version>${geotools.version}</version>
    </dependency>
    <dependency>
        <groupId>org.geotools</groupId>
        <artifactId>gt-swing</artifactId>
        <version>${geotools.version}</version>
        <exclusions>
          <exclusion>
            <groupId>org.apache.xmlgraphics</groupId>
            <artifactId>batik-transcoder</artifactId>
          </exclusion>
        </exclusions>
    </dependency>
</dependencies>S
```

2. We can now create the StyleLab class and refer to these initial imports. Please keep in mind that your IDE may add these for you as you type.

```
package org.geotools.demo;

import java.awt.Color;
import java.io.File;

import org.geotools.data.FeatureSource;
import org.geotools.data.FileDataStore;
import org.geotools.data.FileDataStoreFinder;
import org.geotools.factory.CommonFactoryFinder;
import org.geotools.map.DefaultMapContext;
import org.geotools.map.MapContext;
import org.geotools.styling.*;
import org.geotools.styling.Fill;
import org.geotools.styling.Graphic;
import org.geotools.styling.LineSymbolizer;
import org.geotools.styling.Mark;
import org.geotools.styling.PointSymbolizer;
import org.geotools.styling.PolygonSymbolizer;
import org.geotools.styling.Rule;
import org.geotools.styling.SLDParser;
import org.geotools.styling.Stroke;
import org.geotools.styling.Style;
import org.geotools.styling.StyleFactory;
import org.geotools.swing.ExceptionMonitor;
import org.geotools.swing.JMapFrame;
import org.geotools.swing.data.JFileDataStoreChooser;
import org.geotools.swing.styling.JSimpleStyleDialog;
```

```
import org.opengis.feature.simple.SimpleFeatureType;
import org.opengis.filter.FilterFactory;

import com.vividsolutions.jts.geom.LineString;
import com.vividsolutions.jts.geom.MultiLineString;
import com.vividsolutions.jts.geom.MultiPolygon;
import com.vividsolutions.jts.geom.Polygon;

public class StyleLab {

}
```

3. The Style objects we work with are represented as interfaces; the specific implementation is created for us by a StyleFactory. We will use the CommonFactoryFinder to hunt down an appropriate implementation.

```
    static StyleFactory styleFactory = CommonFactoryFinder.getStyleFactory(null);
    static FilterFactory filterFactory = CommonFactoryFinder.getFilterFactory(null);

    public static void main(String[] args) throws Exception {
        StyleLab me = new StyleLab();
        me.displayShapefile();
    }
```

4. Next we want to display a Shapefile. When we did this step in the Quickstart we let GeoTools create a default Style for us. This time we are going to make our own; and use it when we add a map layer.

```
    private void displayShapefile() throws Exception {
        File file = JFileDataStoreChooser.showOpenFile("shp", null);
        if (file == null) {
            return;
        }

        FileDataStore store = FileDataStoreFinder.getDataStore(file);
        FeatureSource featureSource = store.getFeatureSource();

        // Create a map context and add our shapefile to it
        MapContext map = new DefaultMapContext();
        map.setTitle("StyleLab");

        // Create a basic Style to render the features
        Style style = createStyle(file, featureSource);

        // Add the features and the associated Style object to
        // the MapContext as a new MapLayer
        map.addLayer(featureSource, style);

        // Now display the map
        JMapFrame.showMap(map);
    }
```

5. To start out with we are going to use JSimpleStyleDialog to create a quick style based on user input. We will also check if there is an SLD file associated with the provided shapefile and use that if it exists.

```
private Style createStyle(File file, FeatureSource featureSource) {
        File sld = toSLDFile(file);
        if (sld != null) {
            return createFromSLD(sld);
        }
        SimpleFeatureType schema = (SimpleFeatureType)featureSource.getSchema();
        return JSimpleStyleDialog.showDialog(null, schema);
    }
```

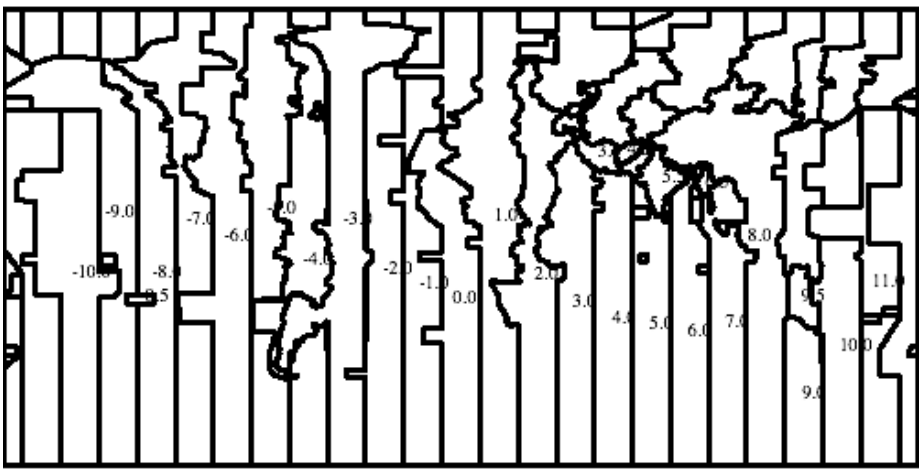
6. The check for an SLD file is straightforward.

```
public File toSLDFile(File file)  {
    String path = file.getAbsolutePath();
    String base = path.substring(0,path.length()-4);
    String newPath = base + ".sld";
    File sld = new File( newPath );
    if( sld.exists() ){
        return sld;
    }
    newPath = base + ".SLD";
    sld = new File( newPath );
    if( sld.exists() ){
        return sld;
    }
    return null;
}
```

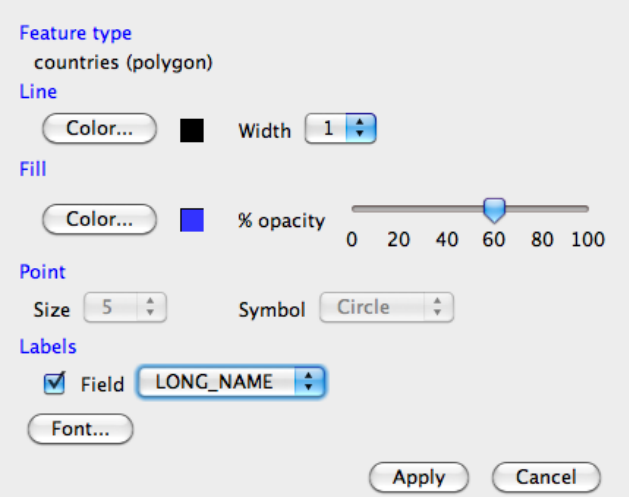
7. To actually read in and process the file GeoTools provides an SLDParser.

```
private Style createFromSLD(File sld) {
    try {
        SLDParser stylereader = new SLDParser(styleFactory, sld.toURI().toURL());
        Style[] style = stylereader.readXML();
        return style[0];

    } catch (Exception e) {
        ExceptionMonitor.show(null, e, "Problem creating style");
    }
    return null;
}
```

8. The program can now be run; if you choose **timezone.shp** it will read in the SLD sidecar file for the shapefile and display the map with the style defined in the SLD file.



9. If you open up the countries.shp file (made of polygons) you will be presented with a simple dialog to choose a few of the most common options.




10. Try with running the application a couple of times with:

- cities.shp in order to style point geometry; and
- bc_border for a line geometry.

## 2.1 Creating a Style by Hand

The methods that we've looked at so far are all we really need in this simple application. But now let's look at how to create a style programmatically. This illustrates some of what is happening behind the scenes in the previous code. It also introduces you to StyleFactory and FilterFactory which provide a huge amount of flexibility in the styles that you can create.

1. We are going to rewrite our createStyle method to work out what type of geometry we have in our shapefile: points, lines or polygons.

```
private Style createStyle2(FeatureSource featureSource) {
        SimpleFeatureType schema = (SimpleFeatureType)featureSource.getSchema();
        Class geomType = schema.getGeometryDescriptor().getType().getBinding();

        if (Polygon.class.isAssignableFrom(geomType)
                || MultiPolygon.class.isAssignableFrom(geomType)) {
            return createPolygonStyle();

        } else if (LineString.class.isAssignableFrom(geomType)
                || MultiLineString.class.isAssignableFrom(geomType)) {
            return createLineStyle();

        } else {
            return createPointStyle();
        }
    }
```

2. We can now go through an example of creating a Style for Polygons.

```
private Style createPolygonStyle() {

        // create a partially opaque outline stroke
        Stroke stroke = styleFactory.createStroke(
                filterFactory.literal(Color.BLUE),
                filterFactory.literal(1),
                filterFactory.literal(0.5));

        // create a partial opaque fill
        Fill fill = styleFactory.createFill(
                filterFactory.literal(Color.CYAN),
                filterFactory.literal(0.5));

        /*
         * Setting the geometryPropertyName arg to null signals that we want to
         * draw the default geomettry of features
         */
        PolygonSymbolizer sym = styleFactory.createPolygonSymbolizer(stroke, fill, null);

        Rule rule = styleFactory.createRule();
        rule.symbolizers().add(sym);
        FeatureTypeStyle fts = styleFactory.createFeatureTypeStyle(new Rule[]{rule});
        Style style = styleFactory.createStyle();
        style.featureTypeStyles().add(fts);

        return style;
    }
```

3. Creating a Line style is done in a similar fashion; defining the Stroke that goes into a LineSymbolizer and wrapping it up in the appropriate Rule.

```
 private Style createLineStyle() {
        Stroke stroke = styleFactory.createStroke(
                filterFactory.literal(Color.BLUE),
                filterFactory.literal(1));
        /*
         * Setting the geometryPropertyName arg to null signals that we want to
         * draw the default geomettry of features
         */
        LineSymbolizer sym = styleFactory.createLineSymbolizer(stroke, null);

        Rule rule = styleFactory.createRule();
        rule.symbolizers().add(sym);
        FeatureTypeStyle fts = styleFactory.createFeatureTypeStyle(new Rule[]{rule});
        Style style = styleFactory.createStyle();
        style.featureTypeStyles().add(fts);

        return style;
    }
```

4. Creating a style for points is also interesting; there are a couple of kinds of “marks” internal marks such as circle or triangle; and external marks such as a PNG or SVG icons.

```
private Style createPointStyle() {
        Graphic gr = styleFactory.createDefaultGraphic();

        Mark mark = styleFactory.getCircleMark();

        mark.setStroke(styleFactory.createStroke(
                filterFactory.literal(Color.BLUE), filterFactory.literal(1)));

        mark.setFill(styleFactory.createFill(filterFactory.literal(Color.CYAN)));

        mark.setSize(filterFactory.literal(3));

        gr.graphicalSymbols().clear();
        gr.graphicalSymbols().add(mark);

        /*
         * Setting the geometryPropertyName arg to null signals that we want to
         * draw the default geomettry of features
         */
        PointSymbolizer sym = styleFactory.createPointSymbolizer(gr, null);

        Rule rule = styleFactory.createRule();
        rule.symbolizers().add(sym);
        FeatureTypeStyle fts = styleFactory.createFeatureTypeStyle(new Rule[]{rule});
        Style style = styleFactory.createStyle();
        style.featureTypeStyles().add(fts);

        return style;
    }
```

## 2.2 Things to Try

Here are a couple of ideas on how to modify the above example:

- Create a rule with both a PolygonSymbolizer and a PointSymbolizer – what happens?
- Create a style with multiple rules; see if you can can draw large cities with a bigger PointSymbolizer.i

# 3 Style

Style is all about looking good – and this section is a box of crayons – learning how to make a map look good is the practice of cartography.

Actually cartography is focused on using a map to communicate, choosing what information to include, being strict about removing information that is off topic and so on.

Occasionally organizations will have “cartographic standards” that must be followed – how thick lines must be exactly, what shade of blue to use for water.  Having a cartographic standard is a great time saver – the rest of us are going to have to be creative.

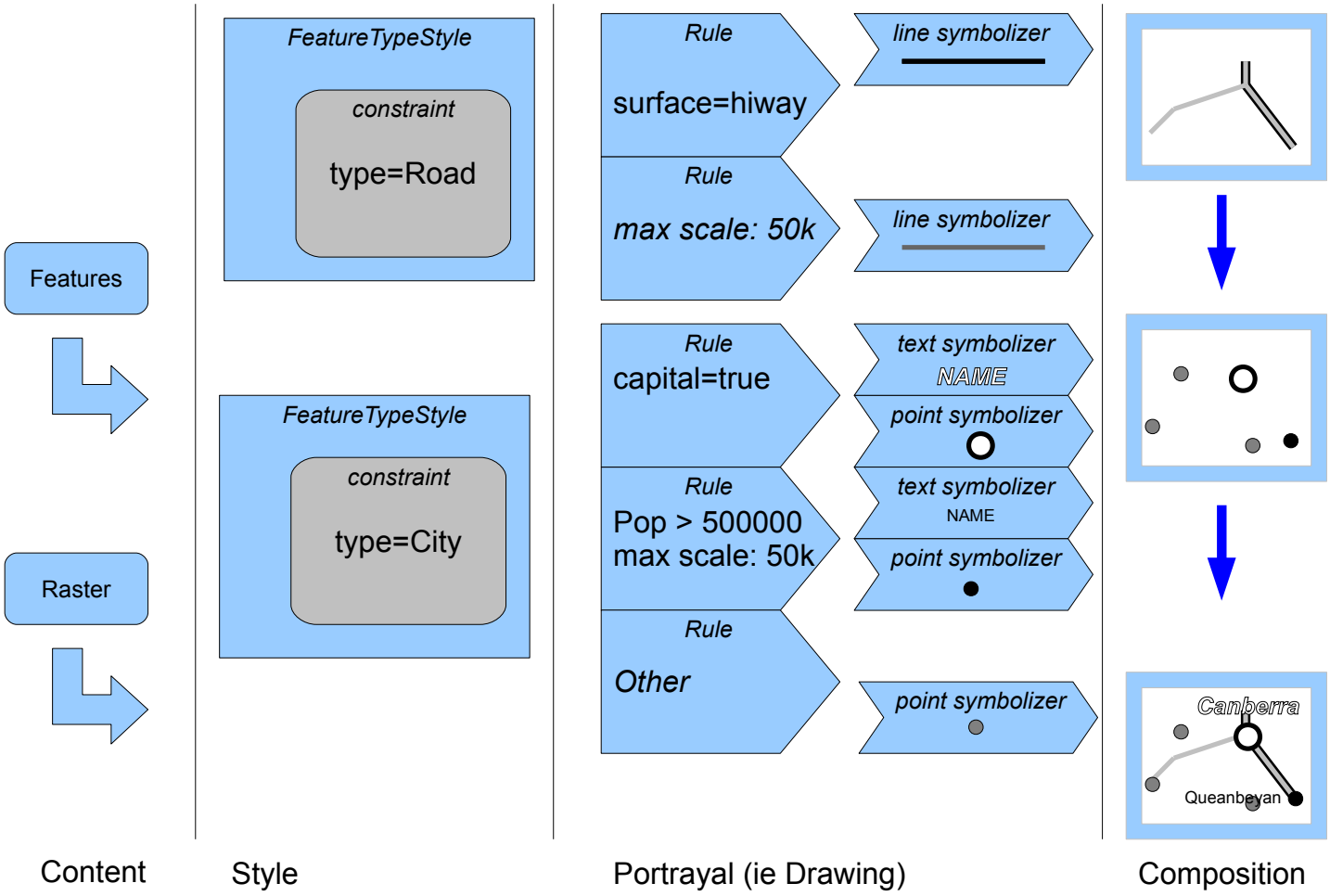
We do have one kind of standard to help us though: the Styled Layer Descriptor (SLD) standard – this document defines a nice data structure for Style which we have captured in the form of Java objects. If you get stuck at any point please review the SLD specification as it defines all the ideas we are going to work with today.

At its heart it focuses on two things:

- Style Layer Descriptor – covers the definition of “layers” or presentations of feature content.
- Symbology Encoding – covers portrayal – or how to draw the features

## 3.1 Controlling the Rendering Process

This is the heart of map making with GeoTools (or indeed with open standards).




It helps if you imagine a big funnel throwing all the features at your map at once. This is going to work kind of like those machines for sorting coins – early stages of the machine are going to select a feature; once we are sure what kind of feature we have we are going to use the feature to control actual drawing onto different bitmaps. Finally we will gather up all the different bitmaps (and slap some labels on top) to produce a final image.

Rendering occurs in the following stages:

- Content Selection – selecting and filtering
- Portrayal – actual drawing
- Composition – putting everything together

The first line of defence is “FeatureTypeStyle”, it makes use of a constraint to select what FeatureType you want to work with. If you don't care use the

FeatureType with the name “Feature” as kind of a wild card (since everything extends “Feature”.

Next up we have Rules. Rules actually use Filter (from the Filter tutorial) to perform strict checks about what is going to get drawn. In addition to checking feature attributes with Filter, a Rule is able to check the current scale of the map. Finally there is an “Other” rule to catch any features left over from earlier Rules.

Now that a Rule has selected features for us to work with we can get down to drawing in the Portrayal step. The renderer will go through a list of symbolizers (for a Rule) and draw the results. The symbolizers are just a list of draw instructions to be done in order. The symbolizers use expressions to define width and color – allowing you to dynamically generate the appearance on a feature by feature basis!

The only symbolizer which is not drawn in order is TextSymbolizer which gathers up text labels for the next step.

Finally in the composition step – will take all the content drawn during portrayal and squish them together into a final image. The icing on the cake is the text labels (produced from any and all TextSymbolizers) which are drizzled on top taking care not to have any overlaps.

# 4 Generating a Style based on Selection

The previous section showed how to style geospatial content using Style Layer Descriptor ideas. The styles are able to use the attributes of individual features to decide which symbolizers to apply via the use of a Filter. The attributes of individual features can also be directly used by symbolizers to pull out text for labels, or colors for theming.

We are going to put these ideas together and make a dynamic style defined by the user at run time.

## 4.1 SelectionLab

We are going to dynamically make a style based on where a user clicks.

This tutorial will combine **everything** we have done so far in one example – hold on to your hats!

- When the user clicks on the screen we are going to construct a rectangle covering 5 x 5 pixels.
- We transform the rectangle in screen coordinates into a bounding box in world (geographic) coordinates for our MapContext.
- We construct a Filter using the rectangle
- We use the Filter to create a Rule.
- We configure the Rule with Symbolizer so that any features that match the filter are drawn in yellow.

## 4.2 The application

1. Please create a new class, **SelectionLab**, in your current project and copy and paste in the following code:

```
package org.geotools.demo;

import com.vividsolutions.jts.geom.LineString;
import com.vividsolutions.jts.geom.MultiLineString;
import com.vividsolutions.jts.geom.MultiPolygon;
import com.vividsolutions.jts.geom.Polygon;
import java.awt.Color;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
```

```
import java.io.File;
import java.util.HashSet;
import java.util.Set;
import javax.swing.JButton;
import javax.swing.JToolBar;
import org.geotools.data.FeatureSource;
import org.geotools.data.FileDataStore;
import org.geotools.data.FileDataStoreFinder;
import org.geotools.factory.CommonFactoryFinder;
import org.geotools.feature.FeatureCollection;
import org.geotools.feature.FeatureIterator;
import org.geotools.geometry.jts.ReferencedEnvelope;
import org.geotools.map.DefaultMapContext;
import org.geotools.map.MapContext;
import org.geotools.styling.FeatureTypeStyle;
import org.geotools.styling.Fill;
import org.geotools.styling.Graphic;
import org.geotools.styling.Mark;
import org.geotools.styling.Rule;
import org.geotools.styling.Stroke;
import org.geotools.styling.Style;
import org.geotools.styling.StyleFactory;
import org.geotools.styling.Symbolizer;
import org.geotools.swing.JMapFrame;
import org.geotools.swing.data.JFileDataStoreChooser;
import org.geotools.swing.event.MapMouseEvent;
import org.geotools.swing.tool.CursorTool;
import org.opengis.feature.simple.SimpleFeature;
import org.opengis.feature.simple.SimpleFeatureType;
import org.opengis.feature.type.GeometryDescriptor;
import org.opengis.filter.Filter;
import org.opengis.filter.FilterFactory2;
import org.opengis.filter.identity.FeatureId;

public class SelectionLab {

    /*
     * Factories that we will use to create style and filter objects
     */
    private StyleFactory sf = CommonFactoryFinder.getStyleFactory(null);
    private FilterFactory2 ff = CommonFactoryFinder.getFilterFactory2(null);

    /*
     * Convenient constants for the type of feature geometry in the shapefile
     */
    private enum GeomType { POINT, LINE, POLYGON };

    /*
     * Some default style variables
     */
    private static final Color LINE_COLOUR = Color.BLUE;
    private static final Color FILL_COLOUR = Color.CYAN;
    private static final Color SELECTED_COLOUR = Color.YELLOW;
    private static final float OPACITY = 1.0f;
    private static final float LINE_WIDTH = 1.0f;
    private static final float POINT_SIZE = 10.0f;

    private JMapFrame mapFrame;
    private FeatureSource<SimpleFeatureType, SimpleFeature> featureSource;

    private String geometryAttributeName;
    private GeomType geometryType;

    /*
     * The application method
     */
    public static void main(String[] args) throws Exception {
        SelectionLab me = new SelectionLab();

        File file = JFileDataStoreChooser.showOpenFile("shp", null);
        if (file == null) {
            return;
        }

        me.displayShapefile(file);
    }
```

Much of this should look familiar to you from the StyleLab class. We've added some constants and class variables that we'll use when creating styles.

A subtle difference is that we are now using FilterFactory2 instead of FilterFactory. This class adds additional methods, one of which we'll need when selecting features based on a mouse click.

2. Next we add the displayShapefile method which is very similar to the one that we used in StyleLab:

```
    public void displayShapefile(File file) throws Exception {
        FileDataStore store = FileDataStoreFinder.getDataStore(file);
        featureSource = store.getFeatureSource();
        setGeometry();

        /*
         * Create the JMapFrame and set it to display the shapefile's features
         * with a default line and colour style
         */
        MapContext map = new DefaultMapContext();
        map.setTitle("Feature selection tool example");
        Style style = createDefaultStyle();
        map.addLayer(featureSource, style);
        mapFrame = new JMapFrame(map);
        mapFrame.enableToolBar(true);
        mapFrame.enableStatusBar(true);

        /*
         * Before making the map frame visible we add a new button to its
         * toolbar for our custom feature selection tool
         */
        JToolBar toolBar = mapFrame.getToolBar();
        JButton btn = new JButton("Select");
        toolBar.addSeparator();
        toolBar.add(btn);
        JButton btn = new JButton("Select");
        toolBar.addSeparator();
        toolBar.add(btn);

        btn.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e) {
                mapFrame.getMapPane().setCursorTool(
                        new CursorTool() {

                            @Override
                            public void onMouseClicked(MapMouseEvent ev) {
                                selectFeatures(ev);
                            }
                        });
            }
        });

        /**
         * Finally, we display the map frame. When it is closed
         * this application will exit.
         */
        mapFrame.setSize(600, 600);
        mapFrame.setVisible(true);
    }
/
```

Note that we're customizing the JMapFrame by adding a button to the toolbar. When the user clicks this button a new **CursorTool** is set for the map window. This tool has just one method that responds to a mouse click in the map area.

3. Next we'll add the method that is called when the user is in selection mode (our custom toolbar button has been clicked) and has clicked somewhere on the map.

   The method first creates a 5x5 pixel wide rectangle around the mouse position to make it easier to select point and line features. This is transformed from pixel coordinates to world coordinates and used to create a **Filter** to identify features that intersect the rectangle:

```
    void selectFeatures(MapMouseEvent ev) {

        System.out.println("Mouse click at: " + ev.getMapPosition());

        /*
         * Construct a 5x5 pixel rectangle centred on the mouse click position
         */
        Point screenPos = ev.getPoint();
        Rectangle screenRect = new Rectangle(screenPos.x-2, screenPos.y-2, 5, 5);

        /*
         * Transform the screen rectangle into bounding box in the coordinate
         * reference system of our map context. Note: we are using a naive method
         * here but GeoTools also offers other, more accurate methods.
         */
        AffineTransform screenToWorld = mapFrame.getMapPane().getScreenToWorldTransform();
        Rectangle2D worldRect = screenToWorld.createTransformedShape(screenRect).getBounds2D();
        ReferencedEnvelope bbox = new ReferencedEnvelope(
                worldRect,
                mapFrame.getMapContext().getCoordinateReferenceSystem());

        /*
         * Create a Filter to select features that intersect with
         * the bounding box
         */
        Filter filter = ff.bbox(ff.property(geometryAttributeName), bbox);

        /*
         * Use the filter to identify the selected features
         */
        try {
            FeatureCollection<SimpleFeatureType, SimpleFeature> selectedFeatures =
                    featureSource.getFeatures(filter);

            FeatureIterator<SimpleFeature> iter = selectedFeatures.features();
            Set<FeatureId> IDs = new HashSet<FeatureId>();
            try {
                while (iter.hasNext()) {
                    SimpleFeature feature = iter.next();
                    IDs.add(feature.getIdentifier());

                    System.out.println("   " + feature.getIdentifier());
                }

            } finally {
                iter.close();
            }

            if (IDs.isEmpty()) {
                System.out.println("   no feature selected");
            }

            displaySelectedFeatures(IDs);

        } catch (Exception ex) {
            ex.printStackTrace();
            return;
        }
    }
```

4. Once the method above has worked out which features were selected, if any, it passes their **FeatureIds** to the **displaySelected** method:

```
public void displaySelectedFeatures(Set<FeatureId> IDs) {
    Style style;

    if (IDs.isEmpty()) {
        style = createDefaultStyle();

    } else {
        style = createSelectedStyle(IDs);
    }

    mapFrame.getMapContext().getLayer(0).setStyle(style);
    mapFrame.getMapPane().repaint();
}
```

As you can see this method just checks if there are any selected features and then uses either the createSelectedStyle method or the createDefaultStyle method to create a new Style for our shapefile. Let's have a look at these two methods:

```
private Style createDefaultStyle() {
    Rule rule = createRule(LINE_COLOUR, FILL_COLOUR);

    FeatureTypeStyle fts = sf.createFeatureTypeStyle();
    fts.rules().add(rule);

    Style style = sf.createStyle();
    style.featureTypeStyles().add(fts);
    return style;
}

private Style createStyle(Set<FeatureId> IDs) {
    Rule selectedRule = createRule(SELECTED_COLOUR, SELECTED_COLOUR);
    selectedRule.setFilter(ff.id(IDs));

    Rule otherRule = createRule(LINE_COLOUR, FILL_COLOUR);
    otherRule.setElseFilter(true);

    FeatureTypeStyle fts = sf.createFeatureTypeStyle();
    fts.rules().add(selectedRule);
    fts.rules().add(otherRule);

    Style style = sf.createStyle();
    style.featureTypeStyles().add(fts);
    return style;
}
```

Note the difference between the two methods: the first creates a Style with a single **Rule** for all features; the second creates a Style with two **Rules**, one of which filters for the selected feature Ids.

5. OK, we're nearly at the end !

   The Style creating methods above both use the following **createRule** method. This is where the Symbolizer is created:

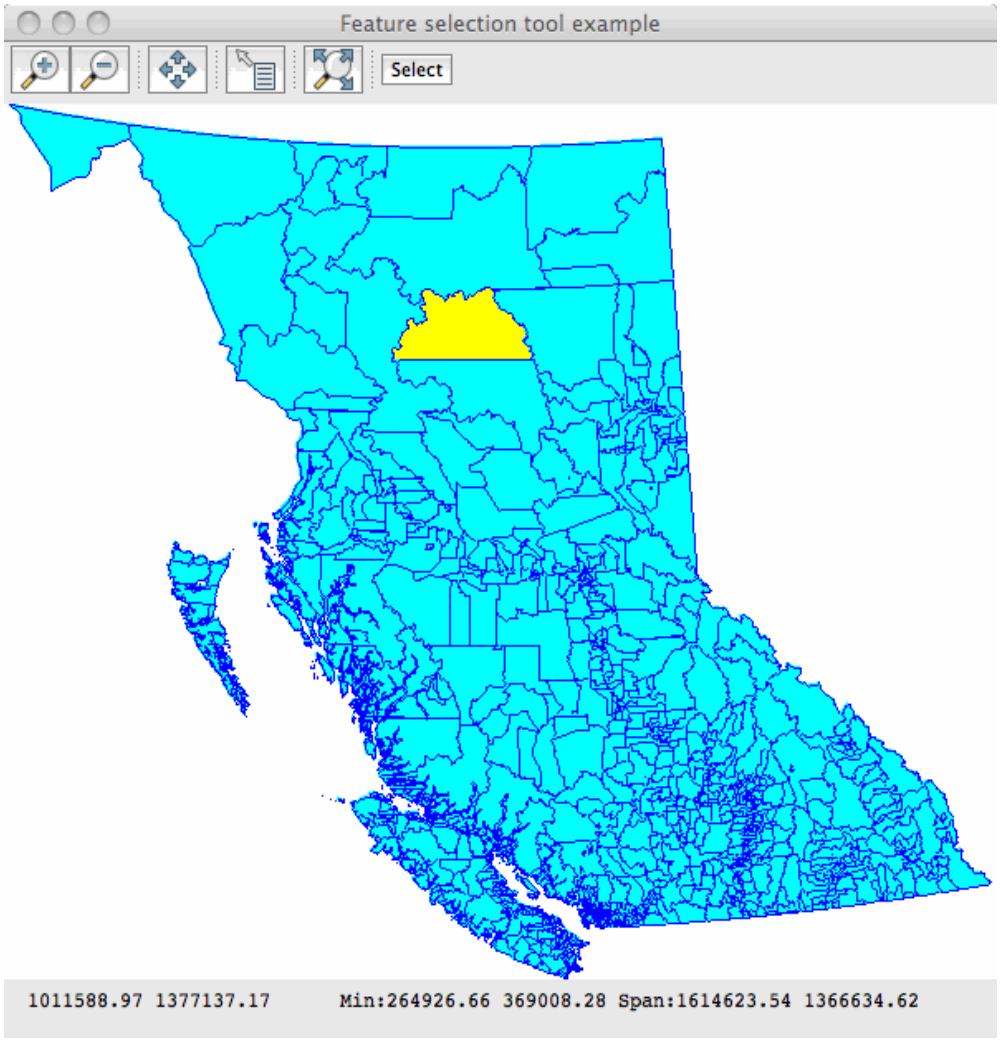
```
private Rule createRule(Color outlineColor, Color fillColor) {
    Symbolizer symbolizer = null;
    Fill fill = null;
    Stroke stroke = sf.createStroke(ff.literal(outlineColor), ff.literal(LINE_WIDTH));

    switch (geometryType) {
        case POLYGON:
            fill = sf.createFill(ff.literal(fillColor), ff.literal(OPACITY));
            symbolizer = sf.createPolygonSymbolizer(stroke, fill, geometryAttributeName);
            break;

        case LINE:
```

```
                symbolizer = sf.createLineSymbolizer(stroke, geometryAttributeName);
                break;

            case POINT:
                fill = sf.createFill(ff.literal(fillColor), ff.literal(OPACITY));

                Mark mark = sf.getCircleMark();
                mark.setFill(fill);
                mark.setStroke(stroke);

                Graphic graphic = sf.createDefaultGraphic();
                graphic.graphicalSymbols().clear();
                graphic.graphicalSymbols().add(mark);
                graphic.setSize(ff.literal(POINT_SIZE));

                symbolizer = sf.createPointSymbolizer(graphic, geometryAttributeName);
        }

        Rule rule = sf.createRule();
        rule.symbolizers().add(symbolizer);
        return rule;
    }
```

6. Finally (yes, really) the createRule method needed to know what sort of feature geometry we are dealing with so that it could create the appropriate class of Symbolizer. Here is the method that works that out:

```
    private void setGeometry() {
        GeometryDescriptor geomDesc = featureSource.getSchema().getGeometryDescriptor();
        geometryAttributeName = geomDesc.getLocalName();

        Class<?> clazz = geomDesc.getType().getBinding();

        if (Polygon.class.isAssignableFrom(clazz) ||
                MultiPolygon.class.isAssignableFrom(clazz)) {
            geometryType = GeomType.POLYGON;

        } else if (LineString.class.isAssignableFrom(clazz) ||
                MultiLineString.class.isAssignableFrom(clazz)) {

            geometryType = GeomType.LINE;

        } else {
            geometryType = GeomType.POINT;
        }
    }
}
```

Here is the application displaying the **bc_voting_areas** shapefile with one feature (polygon) selected:




## 4.3 Things to Try

- Do you remember the CRS Lab where we changed the Coordinate Reference System of your MapContext? Try that out now with one of the "bc" data sets – change the CRS to "EPSG:4326" and then try the selection tool. It does not work anymore!

  See if you can you figure out why and try to fix it.

- There is actually some amazing style generation code included with GeoTools. Try adding a dependency on **gt-brewer** and having a look at the color bewer utility class. The class works by first asking you to calculate a "categorization" using one of the categorization functions on a feature collection; you can then pass the resulting categorization on to color brewer and it will generate a style for you based predefined palettes.

  For more information visit: http://colorbrewer2.org/