Geomajas GWT face

Geomajas Developers and Geosparc

Geomajas GWT face

by Geomajas Developers and Geosparc

1.9.0

Table of Contents

1. Introduction 1
2. GWT face architecture 2
    1. Client-server communication 2
    2. The map's model 2
        2.1. MapModel 2
        2.2. MapView 3
        2.3. Workflow 3
        2.4. Selection of features 4
    3. The spatial package 5
        3.1. Geometry definitions 6
        3.2. Editing geometries 6
        3.3. Snapping 6
    4. Graphics & rendering 7
        4.1. GFX interfaces 8
        4.2. Rendering manual 12
    5. User interaction 14
        5.1. Toolbar: ToolbarAction 14
        5.2. Toolbar: ToolbarModalAction 15
        5.3. LayerTree: LayerTreeAction 16
        5.4. LayerTree: LayerTreeModalAction 17
        5.5. Menu: MenuAction 18
        5.6. Controllers on the map 18
        5.7. Listeners on the map 21
    6. Internationalization in Geomajas 21
    7. Unit testing GWT widgets 22
    8. Creating (custom) feature forms 22
        8.1. Using custom form items within a FeatureForm 23
        8.2. Using custom form items for association attributes 24
        8.3. Creating a custom FeatureForm 25
3. Configuration 26
    1. Dependencies 26
    2. web.xml 27
    3. Build steps 27
4. GWT widgets 29
    1. GraphicsWidget 29
    2. MapWidget 29
    3. OverviewMap 30
    4. Toolbar 31
    5. LayerTree 33
    6. Legend 34
    7. FeatureListGrid 35
    8. FeatureAttributeWindow 36
    9. ActivityMonitor 37
    10. ScaleSelect 37
    11. FeatureSearch 38
5. How-to 39
    1. How to avoid error messages when changing the current window location ? 39

List of Figures

2.1. Workflow API 4
2.2. Selection events and handlers 5
2.3. Main context interfaces from the GFX package 9
2.4. GraphicsContext interface 10
2.5. General graphics interfaces 11
4.1. MapWidget example 30
4.2. OverviewMap example 31
4.3. Tool bar example 33
4.4. LayerTree example 33
4.5. LayerTree with selected layer 34
4.6. Legend 35
4.7. FeatureListGrid example 36
4.8. FeatureAttributeWIndow, editing allowed but not enabled 37
4.9. ActivityMonitor example 37
4.10. ScaleSelect example 38
4.11. FeatureSearch example 38

List of Examples

2.1. Example use of executing a command. 2
2.2. GeometryOperation interface 6
2.3. ToolbarAction 15
2.4. ToolbarModalAction 16
2.5. LayerTreeAction 17
2.6. MenuAction 18
2.7. GraphicsController 19
2.8. Extract from AbstractGraphicsController 20
2.9. FeatureFormFactory interface 22
3.1. Include GWT client dependency 26
3.2. GWT dependencies 26
3.3. Including geomajas-dep for module versions 26
3.4. DependencyManagement with geomajas-dep and a snapshot GWT face 27
3.5. GWT servlet definition 27
3.6. GWT compilation 28

Chapter 1. Introduction

The GWT face allows you to use Java code for development of your AJAX based GIS web applications. It uses Google Web Toolkit and the SmartGWT widget library to have powerful widget which look nice.

GWT is a toolkit from Google which is used amongst others for the development of Google AdWords and Google Wave. It compiles your Java code to JavaScript to allow the code to run on the client side, in the browser. While doing so, it takes care of many browser idiosyncrasies to alleviate the need to consider browser compatibility issues. GWT tries to make the user experience as comfortable as possible, to a large extent by focusing on responsiveness of the user interface. It combines all code and aggressively compresses that to reduce loading delays and allow caching. When the script file gets large, you still have the option to split it to reduce load time (the other parts are then loaded asynchronously. It also contains support for building internationalized applications, for combining resources etc.

For the developer, there is the ability to run your applications in development mode. Though this is slower than production mode, it does remove the need for the full compilation before application startup (a big improvement as GWT produces different code per browser and language pair), and what's more, it allows debugging your application in Java!

Chapter 2. GWT face architecture

1. Client-server communication

Client-server communication in Geomajas is executed through a series of commands instantiated on the client and executed by the server. In the GWT face, all such commands are executed by `org.geomajas.gwt.client.command.GwtCommandDispatcher`. This class has one single method for instantiating such commands and handling the result: the `execute` method.

The command, required by the execute method (`GwtCommand`), is in fact a wrapper around a `CommandRequest` object (see architecture), wherein the name of the requested command is found. Each command expects a certain implementation of the `CommandRequest` and `CommandResponse` objects. Note that these request and response objects must also be known server-side, and thus will not be packaged within the GWT client packages. As additional parameters for the execute method, `CommandCallback` objects can be passed. These can contain methods to call after successful or unsuccessful completion of the command.

The execution will immediately return a `Deferred` object, that can be used to add more callback methods, both for successful as for an erroneous result of the command's execution.

Example 2.1. Example use of executing a command.

```
MySuperDoItRequest commandRequest = new MySuperDoItRequest();
// .... add parameters to the request.

// Create the command, with the correct Spring bean name:
GwtCommand command = new GwtCommand("command.MySuperDoIt");
command.setCommandRequest(commandRequest);

// Execute the command, and do something with the response:
GwtCommandDispatcher.getInstance().execute(command, new CommandCallback

    public void execute(CommandResponse response) {
        // Command returned successfully. Do something with the result.
    }
});
```

2. The map's model

As with any GIS framework or application, the most crucial of all entities is the Map. A map is represented by the `org.geomajas.gwt.client.widget.MapWidget`, but is set up using the standard model-view-controller paradigm. The widget is the actual view of a map, with `Painter` objects building the display. The user interaction is handled using controllers, represented by the `GraphicsController` interface. The third aspect, the model, is represented by the `org.geomajas.gwt.client.map.MapModel`.

2.1. MapModel

The definition of the model behind a map (`org.geomajas.gwt.client.map.MapModel`). This object stores all map metadata and layers, and has an extensive arsenal of methods to operate on the layers and the features.

Another aspect of the `MapModel` is the list of events that it fires. Handlers for the following events can be added to the `MapModel`:

- *MapModelEvent*: this event is fired when the `MapModel` has been properly initialized. When a `MapWidget` is added to the HTML page, it automatically triggers an initialization method. This will ask the server to supply it with the correct metadata, so that the `MapModel` can actually build it's layers, etc. Once this initialization process is done, a `MapModelEvent` will be fired. Add a `org.geomajas.gwt.client.map.event.MapModelHandler` to use this.

- *FeatureSelectedEvent*: this event is fired every time a feature within one of this model's vector layers is selected. Add a `org.geomajas.gwt.client.map.event.FeatureSelectionHandler` to use this.

- *FeatureDeselectedEvent*: this event is fired every time a feature within one of this model's vector layers is deselected. Add a `org.geomajas.gwt.client.map.event.FeatureSelectionHandler` to use this.

- *LayerSelectedEvent*: this event is fired every time a layer within this model is selected. Add a `org.geomajas.gwt.client.map.event.LayerSelectionHandler` to use this.

- *LayerDeselectedEvent*: this event is fired every time a layer within this model is deselected. Add a `org.geomajas.gwt.client.map.event.LayerSelectionHandler` to use this.

2.2. MapView

Part of a map is this `org.geomajas.gwt.client.map.MapView` object, which determines and influences what area is currently visible. Internally the `MapView` has a `Camera` object that you can think of a a satellite that floats above the map. This `Camera` floats at a certain height, on a certain position, and this will determine what part of the map is shown.

The map contains several coordinate spaces.

- world space : coordinates use the crs of the map.

- screen space : pixel coordinates.

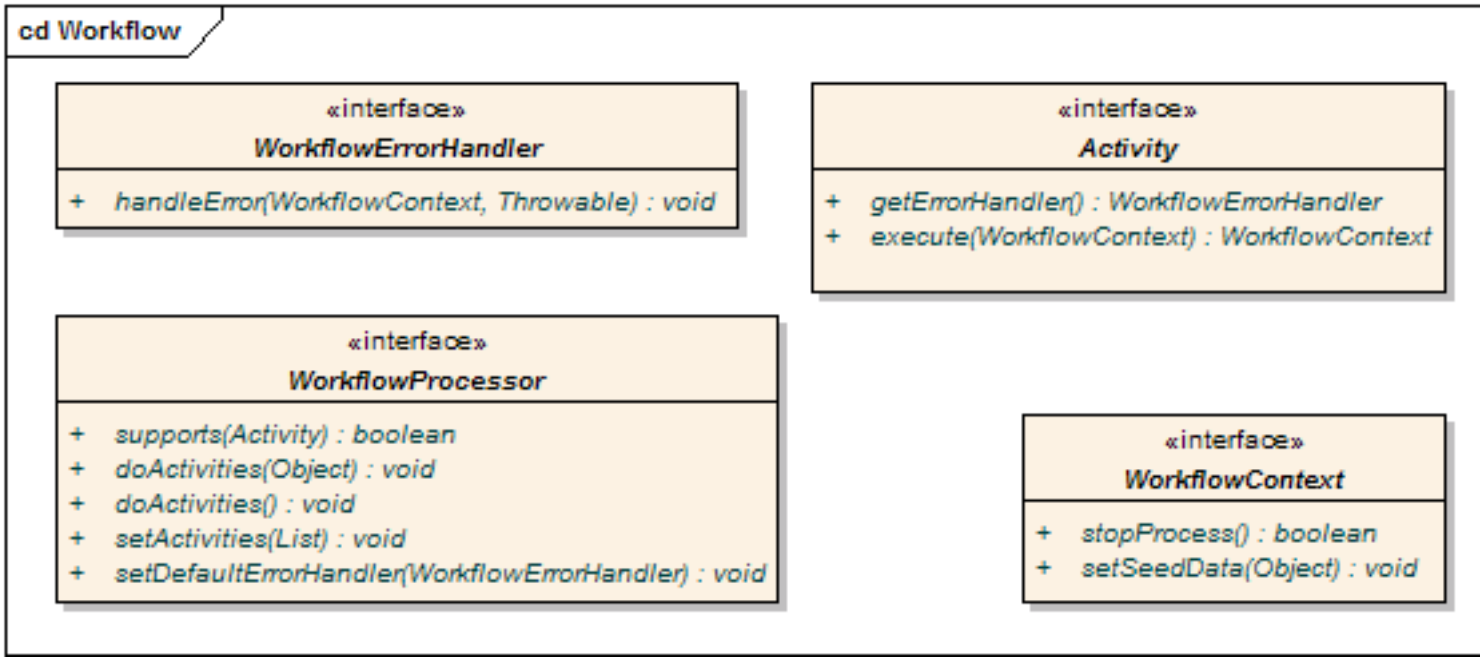
- pan space : objects have already been scaled to the map's current scale, but the translation transformation still needs to occur. This is space is used for rendering the layers and should normally not be used for other purposes.

The `MapView` provides conversions of coordinates between these spaces. Inside the MapView is a transformer (`org.geomajas.gwt.client.spatial.WorldViewTransformer`) that is able to transform coordinates, geometries and bounding boxes from screen space to world space and back.

2.3. Workflow

A work flow in the GWT face determines how editing should be handled. It can consist of several steps, called activities. When a work flow finishes, changes will typically be persisted. The package is: `org.geomajas.gwt.client.map.workflow`

Figure 2.1. Workflow API



Activity:

- *execute*: this method is called by the encompassing processor to execute the activity.
- *getErrorHandler*: get the error handler that is specifically tuned for the activity.

WorkflowErrorHandler:

- *handleError*: Executed when an activity throws an exception during execution. The `WorkflowProcessor` must make sure this method is executed.

WorkflowContext:

- *stopProcess*: informs the `WorkflowProcessor` to stop the processing of activities. It is the `WorkflowProcessors` responsibility to ask for this, and execute no more activities when "`true`" is returned.
- *setSeedData*: provide some seed information to the context. This is usually provided at the time of work flow kickoff.

WorkflowProcessor:

- *supports*: ensure that each activity configured in this process is supported. This method should be called by implemented subclasses for each activity that is part of the process.
- *doActivities*: this method kicks off the processing of work flow activities.
- *setActivities*: set a list of activities to be executed in the process. This would also be a good time to check if activities are supported.
- *setDefaultErrorHandler*: set a default error handler, which is invoked when the activity throws an uncaught exception.

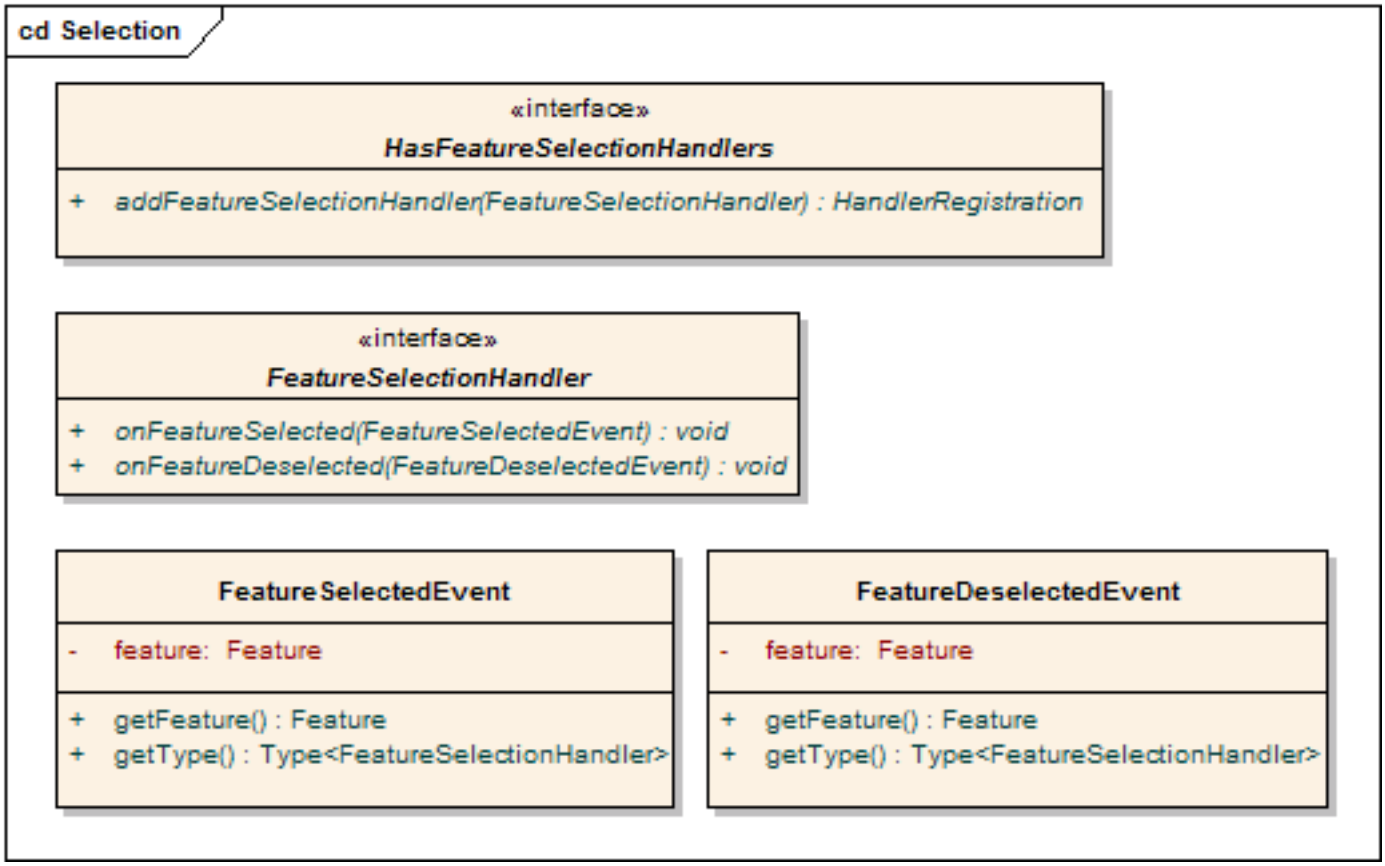
2.4. Selection of features

A feature in a vector layer has the possibility to be selected. Selection usually changes the colour of the feature on the map or in a table to make it stand out. The actual selecting handled by the `MapModel`. In the map model, the selected features are stored. After all, it is possible that by moving away from

a certain area, the selected features are no longer in sight, and perhaps no longer in the client layer-cache. Still their contents may be needed for specific tasks.

The `MapModel` fires events when changes in selection occur. It will fire either `FeatureSelectedEvent` or `FeatureDeselectedEvent`. The `MapModel` implements the `HasFeatureSelectionHandler` interface so other components can register with the `MapModel` as `FeatureSelectionHandler`. For example the `MapWidget` will register itself as a handler to know when a feature is selected or deselected and redraw it accordingly.

Figure 2.2. Selection events and handlers



When writing code that needs to react upon the selection of features, implement the `FeatureSelectionHandler` interface, and register yourself with the `MapModel`.

Note

Individual vector layers can also fire selection events. Actually the `MapModel` propagates the events from the individual vector layers, so that the user need only install his handlers in one place.

3. The spatial package

The spatial package (`org.geomajas.gwt.client.spatial`) contains a collection of math and geometry related classes and utilities to provide all the client-side calculations one should need. If really complex calculations need to be performed, it's best to let the server (probably using JTS) handle it anyway. The root of the package contains the general mathematical definitions of a Vector, Matrix, LineSegment, and so on. It also provides a general math library and the `org.geomajas.gwt.client.spatial.WorldViewTransformer`.

The `WorldViewTransformer` in particular can be a very valuable tool. It allows you to transform coordinates, bounding boxes and geometries between the 3 pre-defined spaces (world, view, pan).

3.1. Geometry definitions

All `Geometry` definitions in the GWT face are based on geometries from JTS, the Java Topology Suite, and the OGC simple feature specification. The classes can be found in the `org.geomajas.gwt.client.spatial.geometry` package. Supported geometries are:

- `Point`: a geometry representation of a single coordinate.
- `MultiPoint`: a geometry containing multiple `Point` geometries.
- `LineString`: a list of connected coordinates. Sometimes also called a polyline.
- `LinearRing`: an extension of the `LineString` geometry that expects the last coordinate to be equal to the first coordinate. In other words, a `LinearRing` is a closed `LineString`.
- `MultiLineString`: a geometry containing multiple `LineString` geometries.
- `Polygon`: a `Polygon` is a two-part geometry, consisting of an exterior `LinearRing` and a list of interior `LinearRings`. The exterior `LinearRing`, also called the shell, is the outer hull of the geometry, while the interior rings can be seen as holes in the exterior ring's surface area.
- `MultiPolygon`: a geometry containing multiple `Polygon` geometries.

3.2. Editing geometries

The `Geometry` implementations themselves do not have any setters methods. Instead the editing of geometries is done through a series of operations, all implementing the `org.geomajas.gwt.client.spatial.geometry.operation.GeometryOperation` interface. This interface consists of only one method, accepting a geometry, and returning the result as a new geometry.

Geometries have no public constructors, so the creation of new geometries is done using a factory, `org.geomajas.gwt.client.spatial.geometry.GeometryFactory`. A `GeometryFactory` can be created using a spatial reference id and a certain precision, but it can also be retrieved from any geometry instance (where srid and precision are automatically correct).

`GeometryOperation` interface:

Example 2.2. GeometryOperation interface

```
@Api(allMethods = true)
public interface GeometryOperation {

    /**
     * The main edit function. It is passed a geometry object. If other values a
     * constructor, or via setters.
     *
     * @param geometry
     *            The {@link Geometry} object to be adjusted.
     * @return Returns the resulting geometry, leaving the original unharmed.
     */
    Geometry execute(Geometry geometry);
}
```

3.3. Snapping

Snapping in Geomajas, is handled by a single manager class called the `org.geomajas.gwt.client.spatial.snapping.Snapper`. It is the main handler for

snapping to coordinates. It supports different modes of operation and different algorithms for the actual snapping. The different algorithms to use are defined in the vector layer configuration files, while the modes are defined by the different implementations of the `SnappingMode` class. Let us first start with the different modes:

- *ALL_GEOMETRIES_EQUAL*: this snapping mode considers all geometries equal when it comes to determining where to snap to. Depending on the snapping algorithm used, it will simply consider all nearby geometries.
- *PRIORITY_TO_INTERSECTING_GEOMETRIES*: this snapping mode tries to snap to intersecting geometries before trying the general approach. When searching a snapping coordinate for a given point, this mode will first search for intersecting geometries and try to get a snap to that. If no snapping point can be found, it will consider all nearby geometries (like ALL_GEOMETRIES_EQUAL).

The snapping rules themselves are defined in the server-side configuration. Each vector layer can have many snapping rules. For each rule, 3 fields must be filled:

- *layer*: the target layer to snap to.
- *distance*: the distance over which to snap. This distance must be expressed in the map's coordinate system.
- *type*: the snapping algorithm to use. At the moment 2 types of snapping algorithms are supported: to the nearest point (type=1), and to the nearest edge (type=2). For nearest point snapping can only occur to any coordinate which is a end-point for a geometry, for nearest edge that can be any coordinate on the edge of the geometry. Needless to say, the nearest edge requires more calculating power than the nearest point.

Snapping on the map

When a `GraphicsController` for the map needs to use snapping (i.e. editing controllers), they should extend the `org.geomajas.gwt.client.controller.AbstractSnappingController` class. This class extends the `GraphicsController` class (the base class for all Geomajas map controllers), and overwrites the `getScreenPosition` and `getWorldPosition` methods to assure the points are snapped. The `AbstractSnappingController` also supports the on-the-fly activation and deactivation of snapping.

4. Graphics & rendering

In the GWT face, the main `render` method can be found in `MapWidget`. The render method requires three parameters, a paintable object, a target group to paint in and a status. The paintable object is the actual object that needs to be painted. The target group (`org.geomajas.gwt.client.widget.MapWidget.RenderGroup`) specifies where in the DOM to draw. The usual choices here are the SCREEN or the WORLD groups. The rendering status (`org.geomajas.gwt.client.widget.MapWidget.RenderStatus`) determines what drawing action to take.

RenderStatus

The render status can be one of the following:

- *ALL*: completely render or re-render the paintable object. If the paintable object contains other paintable object, go through them recursively (a map will paint layers, who in turn will paint tiles,...)
- *UPDATE*: update the paintable object in question, but do not update recursively.
- *DELETE*: delete the paintable object from the map.

While rendering, the map uses a visitor to visit the paintable objects recursively and search for painters for each object or sub-object. The "ALL" status will paint recursively while the "UPDATE" status will not go deeper then the given paintable object. Of course, if a given paintable object has no recursive paintable objects, then the difference between "ALL" and "UPDATE" is irrelevant.

RenderGroup

The render group that needs to be specified when calling the map's render method, represents the logical place on the map to draw the paintable object. There are four choices, each having a huge impact.

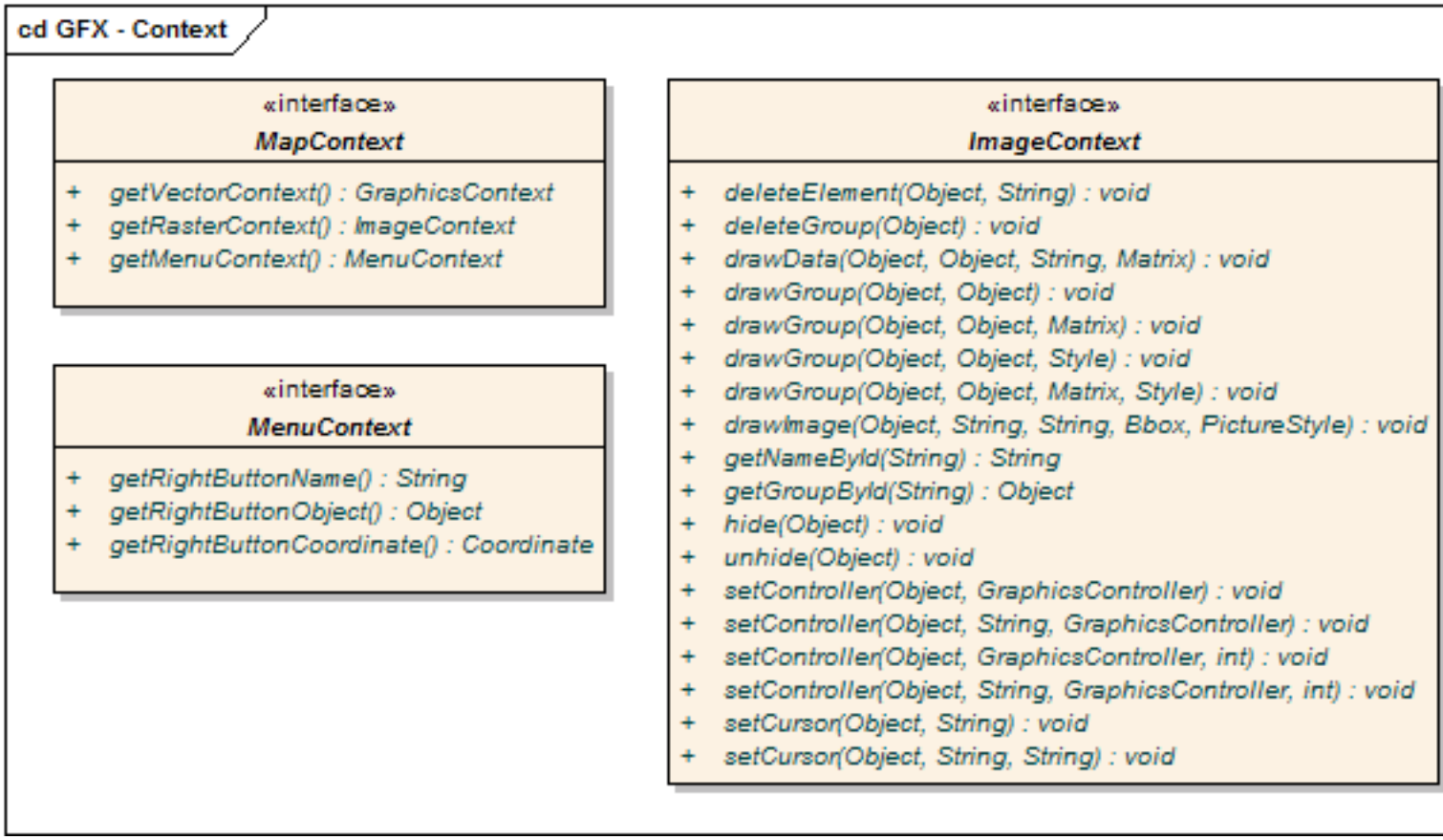
- *WORLD*: drawing should be done in world coordinates. World coordinates means that the map coordinate system should be used. The advantage of rendering objects in the world group, is that when the user moves the map around, the objects will move with it.
- *SCREEN*: drawing should be done in screen coordinates. Screen coordinates are expressed in pixels, starting from the top left corner of the map. When rendering objects in the screen group they will always appear at a fixed position, even when the user moves the map about.
- *RASTER*: drawing should be done in pan coordinates. All raster layers are drawn in this group. In essence this means that the coordinates are expected to have been scaled for the current scale before drawing, and that only the translation still needs to occur. For advanced use only.
- *VECTOR*: drawing should be done in pan coordinates. All vector layers, their selection and their labels are drawn in this group. In essence this means that the coordinates are expected to have been scaled for the current scale before drawing, and that only the translation still needs to occur. For advanced use only.

4.1. GFX interfaces

As will be explained in more detail in the "rendering manual", there are 2 ways of drawing on the map: directly using some rendering context, or indirectly using `Paintable` objects, `Painter`s and the `MapWidget`'s `render` method (as explained above). When using the direct approach, one has to call the methods of one of the different rendering contexts. A `MapWidget` contains a `MapContext` implementation, which in turn contains 3 different contexts:

- *MenuContext*: used for keeping details about right mouse clicks. Not used for rendering.
- *ImageContext*: used for rendering images in HTML. All raster layers use this context.
- *GraphicsContext*: the main vector graphics renderer. Can also render images, but uses SVG or VML to do so. When rendering shapes, circles, rectangle, etc. you will always be using this context.

Figure 2.3. Main context interfaces from the GFX package

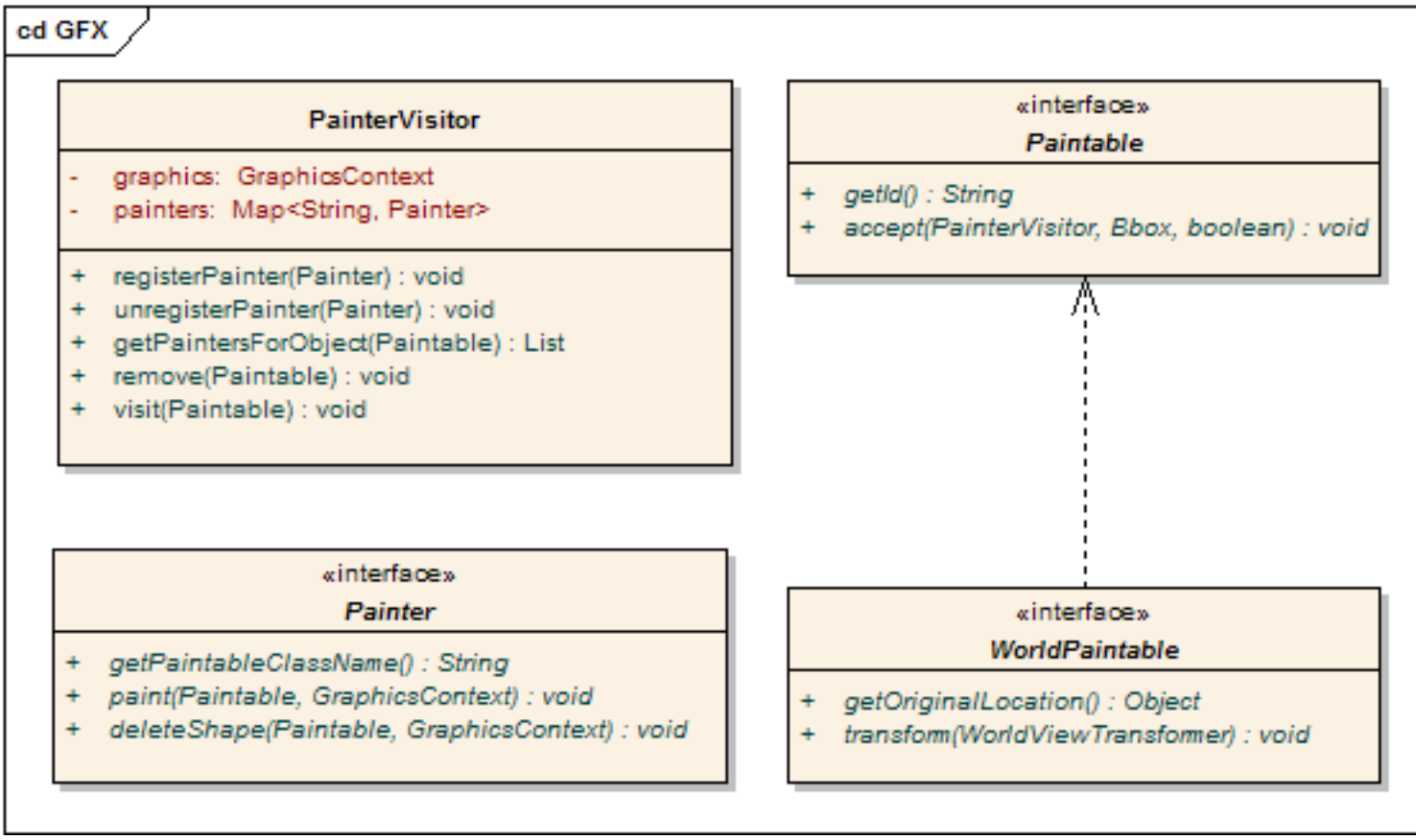


The `GraphicsContext` is the main vector drawing context. It has two implementations: one for SVG and one for VML.

Figure 2.4. GraphicsContext interface

Every object that appears on a Geomajas map, has to implement the `Paintable` interface. This interface marks types of objects that can be painted. For each type/class of paintable object, an accompanying `Painter` must be defined as well. The painter will ultimately decide exactly how a paintable object should be rendered. The painter will render objects using it's `paint` method, or delete objects from the map using it's `deleteShape` method.

Figure 2.5. General graphics interfaces



- *GraphicsContext*: this is the basic drawing interface. Different implementations will draw in different technologies (i.e. `SvgGraphicsContext` and `VmlGraphicsContext`). The whole idea of this `GraphicsContext` and the painters, was inspired by the Java AWT library. This context will draw basic shapes, according to their id. Since we are using web technologies, all implementations (be they SVG or VML) will use DOM elements to create their drawings.

  There are two different kinds of DOM elements we like to distinguish: group elements and non-group elements.

  Group elements are container elements that have no particular representation of themselves but are used to group other elements and create hierarchical dependency. In SVG, they are represented by `<g>` tags, in VML these are `<group>` tags. *Group elements or groups* can be drawn by passing an arbitrary Java object to the `drawGroup()`method. This object can be used for later reference to the group. To position the group in the hierarchy, a parent group object can be passed as a second parameter to the `drawGroup()` method.

  Non-group elements (all other elements like circle, rectangle, symbol, etc.) can be drawn by passing a group element - the parent - and a name to the element-specific drawing method (`drawCircle()`, `drawRectangle()`, etc.). These elements will be attached to their parent group in the DOM. Each non-group element in the `GraphicsContext` DOM tree will therefore have a name and a parent group object. This combination of name and parent can later be used to update or delete the object.

  In essence, the drawing methods will result in changes in the visuals of the map, and the painters that will call these methods.

- *Paintable*: the basic definition of an object that can be painted onto the map. For each `Paintable` class, an accompanying `Painter` class must be defined. The `Paintable` interface has only two methods. The `getId` method returns the `Paintable` objects id, which is it's key in the DOM tree

within the parent group. While the accept method will be traversed by the `PainterVisitor`, and is used to have the object passed to the correct `Painter`, which will draw the object.

- *WorldPaintable*: extension of the `Paintable` interface for objects that support being rendered in world space. This means that it should be possible to transform the object's geometry/location/coordinate/bbox.

- *Painter*: A `Painter` knows how to paint a specific kind of `Paintable` object. Exactly what class of `Paintable` objects it can draw, must be made clear by it's `getPaintableClassName` method. Furthermore, the `Painter` has two methods to paint or remove `Paintable` objects on or from the given `GraphicsContext`. Basically, the `Painter` translates the fields and parameters of the `Paintable` object into calls to the `GraphicsContext`.

- *PainterVisitor*: Geomajas uses a visitor algorithm for it's client side rendering process. The `MapWidget` uses a `PainterVisitor` to recursively traverse the tree of `Paintable` objects, calling the `accept` method on each node.

  Of course this recursive system of searching for the correct `Painter`, can only work when the `PainterVisitor` has all the necessary painters registered. When registering a `Painter` with the `MapWidget`, it will actually pass it along to this `PainterVisitor` instance.

  An example of the recursive painting, can be found in the `MapModel`, which calls the `accept` methods of it's layers, which call the `accept` methods of the visible tiles, which contain features.

4.2. Rendering manual

There are two ways to render objects onto the map. One uses `Paintable` objects and `Painter`s, the other is by using the `GraphicsContext` directly. There are still some general notions that one must know before attempting to draw on the map. Since HTML, SVG and VML are all markup languages which use a DOM tree as basic model, rendering basically is the adding and removing of nodes within this tree. As parent nodes have styling information or other attributes that reflect their child nodes, it is very important to add nodes to the correct parent when drawing.

The GraphicsContext reflects this, by requiring a parent object as first parameters in all drawing methods. Associated with every node is an object that represents it. Given this object, the `GraphicsContext` can find the correct node. When using the `GraphicsContext` directly, it is important to be aware of the necessity of using the correct parent group when drawing.

4.2.1. The simple approach - indirect drawing

4.2.1.1. Drawing in screen space

This approach uses `MapWidget`'s render method, which requires three parameters: a `Paintable` object, a target `RenderGroup`, and a `RenderStatus`. If you are unfamiliar with these, visit the beginning of this chapter for a detailed description.

Let us start with an example, where we draw a rectangle in screen space (=pixel coordinates). The code would look something like this:

```
Rectangle rectangle = new Rectangle("myRectangle");
rectangle.setBounds(new Bbox(10, 10, 200, 200));
rectangle.setStyle(new ShapeStyle("#FF0000", .8f, "#0000FF", .6f, 2));

map.render(rectangle, RenderGroup.SCREEN, RenderStatus.ALL);
```

This code snippet would draw a rectangle (which implements `Paintable`), called "myRectangle" in screen space (10 pixels from the top, 10 pixels from the left, and with a width and height of both 200), using the defined style (red interior with a blue border). To delete the rectangle again, you would have to do something like this:

```
map.render(rectangle, RenderGroup.SCREEN, RenderStatus.DELETE);
```

4.2.1.2. Drawing in world space

Remember that there was also an extension of the `Paintable` interface, designed for rendering objects in world space. Rendering in world space means that objects are drawn in the coordinate system of the map. This also means that when the user moves about on the map, the object will move with it (keeping the same location in map coordinates). Only objects that implement the `WorldPaintable` interface can be drawn in world space.

Let us, for the next example, assume that the map has been defined using EPSG:4326 (lonlat) as coordinate system, and we would apply the following code snippet:

```
Rectangle rectangle = new Rectangle("myRectangle");
rectangle.setBounds(new Bbox(-60, -60, 120, 120));
rectangle.setStyle(new ShapeStyle("#FF0000", .8f, "#0000FF", .6f, 2));

// Register the rectangle to the map, so that it gets redrawn
// automatically when the user navigates on the map.
map.registerWorldPaintable(rectangle);
```

Starting from -60, -60 and using a width and height of 120, we would have a rectangle that encompasses a huge part of the world. Note that drawing objects in world space requires you to register them with the map. This is necessary to have the map automatically update the object's position when the user navigates. When registering the `WorldPaintable` rectangle, it is automatically drawn on the map.

4.2.2. The advanced approach - using the GraphicsContext directly

When more flexibility is required from the rendering system, the map's render method might sometimes not be enough. If for example we want to render more than one object within a specific parent group. The following code snippet shows how to render a specific parent in screen space and then render a circle within this parent group:

```
// Create a parent group within screen space:
Composite parent = new Composite("myParent");
map.getVectorContext().drawGroup(map.getGroup(RenderGroup.SCREEN), parent);

// Draw a circle at (20, 20) with radius 10 pixels within parent group:
Coordinate pos = new Coordinate(20, 20);
ShapeStyle style = new ShapeStyle("#FF0000", .8f, "#0000FF", .6f, 2);
map.getVectorContext().drawCircle(parent, "myCircle", pos, 10, style);
```

Since no special parameters are added to the "myParent" node, the circle is drawn as if it were in the screen group itself. But thanks to the extra parent group, we now have the ability for apply specific styling or transformations on that parent group, and thus altering all children within it. Let us for example move the circle 100 pixels to the right:

```
// Translate the parent group 100 pixels to the right:
Matrix m = new Matrix(1, 0, 0, 1, 100, 0);
map.getVectorContext().drawGroup(map.getGroup(RenderGroup.SCREEN), parent, m);
```

Warning

Do not try to render objects in world space by directly accessing the `GraphicsContext`. Behind the screens of the `MapWidget`, the world space objects are actually transformed and rendered in *vector* space. This is done to avoid scaling in the DOM tree (as this is not possible cross browsers).

5. User interaction

This section covers the many interfaces regarding buttons, menu items and such that make up the user interface. The specific Geomajas widgets (i.e. `LayerTree`) require a specific way of doing things. We will cover the interfaces for the `Toolbar`, `LayerTree`, map controllers and context menus.

- *Toolbar*: the tool bar has two types of default actions one can add to it (there is always the `addChild` method, which can add any widget, but we are now talking about Geomajas specific possibilities): the `ToolbarAction` and the `ToolbarModalAction`. The `ToolbarAction` is used for actions that need immediate response upon clicking, while the `ToolbarModalAction` is used for enabling or disabling a certain state.

- *LayerTree*: the `LayerTree` has the possibility to add buttons to it's tool bar that usually act upon the selected layer within the `LayerTree`. Again two types of actions can be added: the `LayerTreeAction` and the `LayerTreeModalAction`. The same difference as with the tool bar applies: the `LayerTreeAction` is a base abstract class for actions that execute immediately upon clicking, while the `LayerTreeModalAction` is used for enabling or disabling a certain state (for example: toggle the layer's visibility).

- *Menu*: each item in a context menu should extend the `MenuAction` base class. This is your basic starting point for easily creating new menu items or context menu items.

- *Controllers on the map*: for controllers listening to mouse events on a map, there is the `GraphicsController` interface, or an abstraction under the name of `AbstractGraphicsController`.

Note

For buttons in the `Toolbar` or `LayerTree` it is possible to add them to the `org.geomajas.gwt.client.action.toolbar.ToolbarRegistry` or `org.geomajas.gwt.client.action.layertree.LayerTreeRegistry` upon application startup (before `MapWidget` initialisation!). This allows you to add new buttons which can be included in the map configuration.

5.1. Toolbar: ToolbarAction

The `ToolbarAction` is your basic abstract class for building tool bar buttons that are executed immediately when clicked. The class implements the `ClickHandler` interface and requires your to specify an icon and a tool tip on creation.

`ToolbarAction` classes need to be registered in the `ToolbarRegistry` class. This allows you to get an instance of the widget to put in the tool bar. The tools which are part of the GWT face are statically defined in the class. Other tools can be added (or overwritten) at runtime before the map is initialised.

When a `ToolbarAction` is configurable (XML configuration), it should implement the `ConfigurableAction` interface. This contains a "`configure()`" method which will be called for each of the parameters which are defined in the tool configuration.

Example 2.3. ToolbarAction

```
/**
 * Abstract class that serves as a template for building tool bar actions. A to
 * executed immediately when the tool bar button is clicked. If you want a sele
 * the {@link ToolbarModalAction} class.
 *
 * @author Pieter De Graef
 * @since 1.6.0
 */
@Api(allMethods = true)
public abstract class ToolbarAction extends ToolbarBaseAction implements ClickH

    public ToolbarAction(String icon, String tooltip) {
        super(icon, tooltip);
    }
}
```

5.2. Toolbar: ToolbarModalAction

The `ToolbarModalAction` is the basic template for creating selectable tool bar buttons. Usually they enable and disable a certain state on the map when selected or deselected. Many of the implementations that come with Geomajas set a new controller on the map when they are selected.

Note that only one of these `ToolbarModalActions` can be selected at any given time. In that sense they act as radio buttons.

`ToolbarModalAction` classes need to be registered in the `ToolbarRegistry` class. This allows you to get an instance of the widget to put in the tool bar. The tools which are always part of the GWT face are statically defined in the class. Other tools can be added (or overwritten) at runtime before the map is initialised.

When a `ToolbarAction` is configurable, it should implement the `ConfigurableAction` interface. This contains a `configure()` method which will be called for each of the parameters which are defined in the tool configuration.

Example 2.4. ToolbarModalAction

```
/**
 * Abstract class which serves as a template for selectable buttons in a tool ba
 * selected and deselected. With each of these actions a different method is exe
 * button is used to set a new controller onto the {@link org.geomajas.gwt.clie
 * for an action that should be executed immediately when clicking on it, have a
 * {@link org.geomajas.gwt.client.action.ToolbarAction} class.
 *
 * @author Pieter De Graef
 * @since 1.6.0
 */
@Api(allMethods = true)
public abstract class ToolbarModalAction extends ToolbarBaseAction {

    public ToolbarModalAction(String icon, String tooltip) {
        super(icon, tooltip);
    }

    // Class specific actions:

    /**
     * When the tool bar button is selected, this method will be called.
     */
    public abstract void onSelect(ClickEvent event);

    /**
     * When the tool bar button is deselected, this method will be called.
     */
    public abstract void onDeselect(ClickEvent event);
}
```

5.3. LayerTree: LayerTreeAction

The `LayerTreeAction` is your basic abstract class for building layer tree buttons that are executed immediately when clicked. The `onClick()` method needs to be implemented and it also requires you to specify an icon, a tool tip and a disabled icon. Note that the `onClick()` has the selected layer within the `LayerTree` as a parameter.

`LayerTreeAction` classes need to be registered in the `LayerTreeRegistry` class. The tools which are always part of the GWT face are statically defined in the class. Other tools can be added (or overwritten) at runtime before the map is initialised.

Example 2.5. LayerTreeAction

```
public abstract class LayerTreeAction extends ToolbarBaseAction {

    private String disabledIcon;

    /**
     * Constructor setting all values.
     *
     * @param icon The default icon for the button.
     * @param tooltip The default tooltip for the button.
     * @param disabledIcon The icon used when the button is disabled.
     */
    public LayerTreeAction(String icon, String tooltip, String disabledIcon) {
        super(icon, tooltip);
        this.disabledIcon = disabledIcon;
    }

    /**
     * This method will be called when the user clicks on the button.
     *
     * @param layer The currently selected layer.
     */
    public abstract void onClick(Layer<?> layer);

    /**
     * Is the this action enabled for the layer?
     *
     * @param layer layer to test
     * @return enabled status of action for layer
     */
    public abstract boolean isEnabled(Layer<?> layer);

    /**
     * Set icon to display when button is disabled.
     *
     * @return icon shown when the button is disabled
     */
    public String getDisabledIcon() {
        return disabledIcon;
    }

    /**
     * Set icon for disabled state.
     *
     * @param disabledIcon icon for disabled state
     */
    public void setDisabledIcon(String disabledIcon) {
        this.disabledIcon = disabledIcon;
    }
}
```

5.4. LayerTree: LayerTreeModalAction

The `LayerTreeModalAction` is the basic template for creating selectable layer tree buttons. Usually they enable and disable a certain state for the selected layer within the layer tree (for example that layer's visibility).

`LayerTreeModalAction` classes need to be registered in the `LayerTreeRegistry` class. The tools which are always part of the GWT face are statically defined in the class. Other tools can be added (or overwritten) at runtime before the map is initialised.

5.5. Menu: MenuAction

To create menu items or context menu items, Geomajas provides a base which extends from SmartGWT's `MenuItem` class. It requires you to set a title and icon. It also implements the `ClickHandler` interface for defining the `onClick()` execution function.

Example 2.6. MenuAction

```
/**
 * General definition of a <code>MenuAction</code>. All Geomajas actions in too
 * this class.
 *
 * @author Pieter De Graef
 * @since 1.6.0
 */
@Api(allMethods = true)
public abstract class MenuAction extends MenuItem implements ClickHandler {

    /**
     * Constructor that expects you to immediately fill in the title and the ic
     *
     * @param title
     *            The textual title of the menu item.
     * @param icon
     *            A picture to be used as icon for the menu item.
     */
    protected MenuAction(String title, String icon) {
        super(title, icon);
        addClickHandler(this);
    }
}
```

5.6. Controllers on the map

5.6.1. GraphicsController

For interactive mouse controllers on the map there is a general interface, `GraphicsController`. To write a custom controller, you should always extend `AbstractGraphicsController`.

Caution

The `GraphicsController` interface does NOT use SmartGWT events as they provide no way of getting the target DOM element from the mouse events. So the list of handlers that the `GraphicsController` extends, are all basic GWT event handlers. A separate widget (`GraphicsWidget`) has been created to catch the events, while the normal `MapWidget` (which encapsulates the `GraphicsWidget`) can still handle SmartGWT events.

On top of all the event handling methods that come from the different handlers, the interface also has `onActivate()` and an `onDeactivate()` methods. The `onActivate()` is called before the controller is actually applied on the `GraphicsWidget`. This is usually used to apply a new context menu on the map and such. The `onDeactivate()` method is called when the controller is removed from the `GraphicsWidget`. This is usually used for cleaning up.

```
 * </p>
 *
 * @author Pieter De Graef
 * @since 1.6.0
 */
@Api(allMethods = true)
public interface GraphicsController extends MouseDownHandler, MouseUpHandler, M
        MouseOverHandler, MouseWheelHandler, DoubleClickHandler {

    /**
     * Function executed when the controller instance is applied on the map.
     */
    void onActivate();

    /**
     * Function executed when the controller instance is removed from the map.
     */
    void onDeactivate();

    /**
     * An offset along the X-axis expressed in pixels for event coordinates. Us
     * specific elements that have such an offset as compared to the origin of
     * X,Y coordinates relative from their own position, but need this extra of
     * correct screen and world position.
     *
     * @since 1.8.0
     */
    int getOffsetX();

    /**
     * An offset along the X-axis expressed in pixels for event coordinates. Us
     * specific elements that have such an offset as compared to the origin of
     * X,Y coordinates relative from their own position, but need this extra of
     * correct screen and world position.
     *
     * @param offsetX
     *            Set the actual offset value in pixels.
     *
     * @since 1.8.0
     */
    void setOffsetX(int offsetX);

    /**
     * An offset along the Y-axis expressed in pixels for event coordinates. Us
     * specific elements that have such an offset as compared to the origin of
     * X,Y coordinates relative from their own position, but need this extra of
     * correct screen and world position.
     *
     * @since 1.8.0
     */
    int getOffsetY();

    /**
     * An offset along the Y-axis expressed in pixels for event coordinates. Us
     * specific elements that have such an offset as compared to the origin of
     * X,Y coordinates relative from their own position, but need this extra of
     * correct screen and world position.
     *
     * @param offsetY
     *            Set the actual offset value in pixels.
     * @since 1.8.0
     */
    void setOffsetY(int offsetY);
}
```

Example 2.7 GraphicsController

You should never directly implement `GraphicsController` (not that it does not have the "`@UserImplemented`" annotation), you should always extend `AbstractGraphicsController`. This abstract class implements all methods as empty methods so you don't have to clutter your code with empty methods (often only a few of the mouse event methods are actually used). It also has some extra methods for return useful information for the mouse events, such as the position (expressed in screen coordinates) or the target DOM element.

Small extract from the `AbstractGraphicsController` class:

Example 2.8. Extract from AbstractGraphicsController

```
protected Coordinate getScreenPosition(MouseEvent<?> event) {
    return GwtEventUtil.getPosition(event, offsetX, offsetY);
}

protected Coordinate getPanPosition(MouseEvent<?> event) {
    return getTransformer().viewToPan(GwtEventUtil.getPosition(event, offse
}

protected Coordinate getWorldPosition(MouseEvent<?> event) {
    return getTransformer().viewToWorld(GwtEventUtil.getPosition(event, off
}

protected Element getTarget(MouseEvent<?> event) {
    return GwtEventUtil.getTarget(event);
}

protected String getTargetId(MouseEvent<?> event) {
    return GwtEventUtil.getTargetId(event);
}
```

Now that you have your controller you can set it;

```
mapWidget.setController(new MeasureDistanceController(mapWidget));
```

Note

There are more abstractions than just the `AbstractGraphicsController`:

- *`AbstractRectangleController`* : abstract controller that handles drawing a rectangle by dragging the mouse on the map.
- *`AbstractSnappingController`* : abstract controller that allows snapping to be enabled and disabled. When enabled, the returned points when asking `getPosition()`, are snapped (depending on the configured snapping rules).

In order to disable the active controller, set a "null" value as controller:

```
mapWidget.setController(null);
```

5.6.2. Active and fallback controllers

Only one `GraphicsController` can be active at any one time on a map. This is done deliberately in order for controllers not to interfere with each other. By default the MapWidget uses a "fallback controller" that is activated when no explicit controller is set (a navigation controller by default). So when no explicit controller is set, or `mapWidget.setController(null)` is used, the map will turn to it's fallback controller.

It is possible to replace this fallback controller by another than the default, by calling:

```
mapWidget.setFallbackController(<the new controller>);
```

In order to completely disable the use of a fallback controller, set "null" as the new fallback controller.

5.7. Listeners on the map

As an alternative to the interactive controllers on a map, there are passive `Listeners` as well. These listener have an interface very much like the `GraphicsController` interface, but they never receive the real mouse events. As such they are meant to be passive observers, that simply receive notifications of mouse events. As opposed to the `GraphicsControllers`, these listeners are never allowed to interfere.

As a result, multiple listeners can be registered on the map, while only one `GraphicsController` can be active at any given time. Instead of the real mouse events, listeners receive placeholder events of the type `ListenerEvent`. These `ListenerEvent` objects resemble the GWT mouse events a bit, and contain most of the information of the real mouse events.

In order to add a listener to the map, or remove a listener from the map, your code should be something like this:

```
// Define a new Listener, starting from the AbstractListener:
Listener myListener = new AbstractListener() {
    public void onMouseMove(ListenerEvent event) {
        // Do something...
    }
};

// Add the Listener to the map:
mapWidget.addListener(myListener);

// Remove the Listener from the map:
mapWidget.removeListener(myListener);
```

Note

When creating a new Listener, often it is easiest to start from the `AbstractListener` class, which has all empty methods for catching the mouse events. In this case you simply override the methods for the events you are interested in.

6. Internationalization in Geomajas

For internationalization, Geomajas uses the default GWT i18n implementation. For Geomajas specifically, the i18n is used in several places, each having it's own list of messages. Basically all i18n message definitions are located in the package `org.geomajas.gwt.client.i18n`, as are the properties files containing the translations.

Several separate definitions have been created:

- *MenuMessages*: the `MenuMessages` defines parameterized string values that are used in the titles of `MenuAction` classes. Examples are the editing context menus.
- *ToolbarConstants*: this defines strings that are used as tool tips when hovering over the buttons in the tool bar. This list of values is used by the `ToolbarAction` and `ToolbarSelectAction` classes. Note that for tool tips the " " character is used instead of the default space.
- *AttributeMessages*: .....
- *GlobalMessages*: .....

- *LayerTreeMessages*: .....
- *SearchMessages*: .....

To avoid multiple instantiations of the constants and messages classes and have a central access point for all internationalization concerns, the `I18nProvider` class has been created. This class has static methods for accessing the constants and messages classes. Usage is as follows:

```
String dist = I18nProvider.getMenu().getMeasureDistanceString(totalDistance, ra
setContents( "<div><b>" + I18nProvider.getMenu().distance() + "</b>:</div><div
```

7. Unit testing GWT widgets

A GWT unit test should inherit from the `GWTTestcase` base class and should be named `GwtTestXxx.java`. GWT unit tests are run inside a development mode environment and can refer to most of the GWT API. To run a GWT test case, run the Maven command **gwt:test** or execute the integration test phase.

8. Creating (custom) feature forms

This functionality has been added to the GWT client API in version 1.9.0.

When working with vector layers, one often has a need for editing not only the vector component of the features, but also the alpha-numerical attributes. To this end, special factories have been created that can be passed to the editing widgets (FeatureAttirbuteEditor and FeatureAttributeWindow) to create such forms.

All interfaces and classes in this section can be found in the following package: `org.geomajas.gwt.client.widget.attribute`

Central in all this, is the `FeatureForm` and the `FeatureFormFactory`. The `FeatureFormFactory` is an interface that defines a single method for the creation of a `FeatureForm` instance.

Example 2.9. FeatureFormFactory interface

```
public interface FeatureFormFactory {

   /**
    * Creates a form using the specified attribute information.
    *
    * @param infos
    *           List of attribute definitions. Normally taken from a {@link Vec
    * @return An attribute form that allows for editing of it's values.
    */
    FeatureForm createFeatureForm(VectorLayer layer);
}
```

The default implementation of this factory has been implemented as `DefaultFeatureFormFactory`. This factory returns a default implementation for the feature form for each layer: `DefaultFeatureForm`. When the default layout of a feature form is sufficient use the `DefaultFeatureFormFactory` as is, otherwise feel free to implement your own factory that returns custom instances of a `FeatureForm`.

The `FeatureForm` is a form definition that has been tuned towards the Geomajas GWT feature definition, and is used in the central editing widgets (`FeatureAttributeEditor` and `FeatureAttributeWindow`). Note that those editing widgets have constructors to accept a `FeatureFormFactory`, so you have to pass them a custom factory that returns your custom form.

Of course implementing your own `FeatureForm` might be a bit overwhelming and fortunately is usually not necessary as you can easily extend the `DefaultFeatureForm` to suit your needs. The `DefaultFeatureForm` has several hooks to allow you to influence its layout:

- The `createField(AttributeInfo)` and/or `createItem(AttributeInfo)` methods can be overridden to customize form items or create your own.
- The `isIncluded(AttributeInfo)` method can be overridden to omit certain attributes.
- The `prepareForm(FormItemList, DataSource)` method can be overridden to set the layout properties of the form and add additional items such as spacers or headers.

Adding new form item types can also be done in a more global way. Therefore, an extra utility class has been added to create the individual form items that represent the feature's attributes. This utility class is the `AttributeFormFieldRegistry`. This registry keeps a pre-defined set of form item definitions and data source fields (both are necessary to successfully create forms in SmartGWT) that map onto attribute types. For example, it will map a DateItem and DataSourceDateField onto all attributes of type "DATE". This registry can now be used to create the individual items within the form, or to register custom form items types to use in the forms.

Note

Internally, the `org.geomajas.gwt.client.widget.attribute.DefaultFeatureForm` makes use of the AttributeFormFieldRegistry. This means that if you register custom form items for certain attribute types, these custom items will also appear in the default feature forms, without you having to implement a custom FeatureForm.

In the following sections, the details of registering custom form items and creating custom forms will be explained.

8.1. Using custom form items within a FeatureForm

In order to use custom form items within a FeatureForm, there are 2 ways: override one the default for a certain attribute type, or define a completely new type.

In any case, you will need to register a custom type with the AttributeFormFieldRegistry. Let us start with a coding example:

```
// We define the custom type "myType" in the AttributeFormItemFactory:
        AttributeFormFieldRegistry.registerCustomFormItem("myType", new DataSou

            public DataSourceField create() {
                return new DataSourceIntegerField();
            }
        }, new FormItemFactory() {

            public FormItem create() {
                return new SliderItem();
            }
        }, null);
```

This short piece of code will register a "SliderItem" for all attributes of the type "myType". Let us first go over the code in more detail. In order to register a new type, 4 arguments are required:

- The key associated with the given `FormItemFactory` and `DataSourceFieldFactory`. This key is either the name of an attribute type (i.e. `PrimitiveType.DATE.name()`) to overwrite the default definitions, or a completely new type which can be configured in the attribute

definitions with the `formInputType` field (see further in this section). This key is a unique identifier. In the example above: "myType".

- A `DataSourceFieldFactory` definition. This factory will create a SmartGWT `DataSourceField` that represents the underlying data-source in the form.
- A `FormItemFactory` definition. This factory will create a SmartGWT FormItem that represents the actual widget by which the attribute is presented in the form. In the case above, the attribute will be represented by a `SliderItem`.
- A list of validators that can be attached to the underlying `DataSourceField`. Any time the user changes values within a FeatureForm, the associated validators will be executed. In the example above no extra validators are attached to the `DataSourceIntegerField (value null)`.

So the parameter explanation already told us how to overwrite the default attribute types: by registering a new type for an attribute type (i.e. `PrimitiveType.DATE.name()`). The other option, to register completely new types, is done by using a new key (i.e."myType"). The only thing left to do than is to associate attributes with this key. This can be done within the vector layer XML configuration, where a field by the name of "`formInputType`" can be added to an attribute configuration. For more information on attribute configuration, visit the configuration part in the Geomajas developer guide [http://files.geomajas.org/maven/trunk/geomajas/docbook-devuserguide/html/master.html#conf-vectorInfo].

8.2. Using custom form items for association attributes

Association attributes (one-to-many and many-to-one) require a specific treatment because they are too complex to be mapped to a single-valued form item. For these attributes, special interfaces have been constructed that provide full control over the way association value data travels between the feature and it's form representation:

- The `ManyToOneItem` interface defines the contract for a custom form item that represents a many-to-one assocation
- he `OneToManyItem` interface defines the contract for a custom form item that represents a one-to-many assocation

Both interfaces contain the following methods:

- `getItem()`: this methods returns the actual form item to be used in the form. This can either be an item that holds the actual value or a button or link that opens a more complex editor.
- `fromItem(), toItem()` : these methods govern the actual transfer of data between the form and the association value. This may include transfer from and to a more complex editor.
- `init(AssociationAttributeInfo attributeInfo, AttributeProvider attributeProvider)`: this method provides a way to initialize the custom form item and/or editor. It passes information on the nested attributes of the association value and an attribute provider interface. The provider interface allows to query a list of possible attribute values in case the user has to select the value from a predefined list.
- `clearValue()` : this method allows custom clearance of the form item and editor.

The default implementation of `ManyToOneItem` is a simple combobox.

The default implementation of `OneToManyItem` is a "`More...`" link that pops up a master-detail editor configuration for editing a list of association values.

These are in line with our default interpretation of a many-to-one attribute as being an association by reference to an externally managed list and a one-to-many item as being a parent-child relationship.

Custom behavior can of course be implemented by registering a different implementation of the interfaces.

8.3. Creating a custom FeatureForm

The next step is to create a custom `FeatureForm`. Let us again start with a short code example:

```
public class AttributeCustomForm extends DefaultFeatureForm {

 public AttributeCustomForm(VectorLayer vectorLayer) {
  super(vectorLayer);
 }

 @Override
 protected FormItem createItem(AttributeInfo info) {
  FormItem formItem = super.createItem(info); // call super to create the defau
  formItem.setWidth("*");
  if ("dateAttr".equals(info.getName())) {
   // The date attribute will span all 4 columns:
   formItem.setColSpan(4);
  }
  return formItem;
 }

 @Override
 protected void prepareForm(FormItemList formItems, DataSource source) {
  // Quickly insert a row spacer before the 'stringAttr' item (which is the tex
  formItems.insertBefore("stringAttr", new RowSpacerItem()); // inserting an it
  getWidget().setNumCols(4);
  getWidget().setWidth(450);
  getWidget().setColWidths(100, 180, 20, 150);
  getWidget().setGroupTitle("Custom Attribute Form");
  getWidget().setIsGroup(true);
 }
}
```

The `createItem(AttributeInfo info)` method is overridden to set the item width and make the date attribute span 4 columns. The `prepareForm(FormItemList formItems, DataSource source)` method is overridden to insert an extra spacer item before the string attribute and set the general column layout. The form that results from this will not differ all too much from the default implementation, but it should get you on your way.

Chapter 3. Configuration

To use the GWT face, you have to include the relevant dependencies, the required build steps, and make sure the GWT dispatcher servlet is included in your web.xml.

1. Dependencies

You have to include the GWT face (client) module. As this is a GWT specific jar, it already includes the source (in many other cases, you would need to explicitly add the sources as well).

Example 3.1. Include GWT client dependency

```
<dependency>
    <groupId>org.geomajas</groupId>
    <artifactId>geomajas-gwt-client</artifactId>
</dependency>
```

You also include the GWT dependencies themselves. Even though these dependencies are already used by the geomajas-gwt-client module, they need to be redefined because of the scopes.

Example 3.2. GWT dependencies

```
<dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-user</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-servlet</artifactId>
    <scope>runtime</scope>
</dependency>
```

These snippets don't include any version info. It is assumed that you use the geomajas-dep module in a recent incarnation to assure these versions are set.

Example 3.3. Including geomajas-dep for module versions

```
<dependencyManagement>
        <dependency>
            <groupId>org.geomajas</groupId>
            <artifactId>geomajas-dep</artifactId>
            <version>1.10.32</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

If you want to use a snapshot version of the GWT face, you should also included a dependency on the specific snapshot in the dependencyManagement section of your pom.

Example 3.4. DependencyManagement with geomajas-dep and a snapshot GWT face

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.geomajas</groupId>
            <artifactId>geomajas-face-gwt</artifactId>
            <version>1.9.0-SNAPS</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.geomajas</groupId>
            <artifactId>geomajas-dep</artifactId>
            <version>1.10.32</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2. web.xml

You need to include the Geomajas GWT service to assure the GWT files can be found. The servlet mapping has to be edited to match your module class's fully qualified name (example here from the geomajas-gwt-simple module).

Example 3.5. GWT servlet definition

```
<servlet>
    <servlet-name>GeomajasServiceServlet</servlet-name>
    <servlet-class>org.geomajas.gwt.server.GeomajasServiceImpl</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>GeomajasServiceServlet</servlet-name>
    <url-pattern>/mypackage.GeomajasSimple/geomajasService</url-pattern>
</servlet-mapping>
```

3. Build steps

The GWT compilation needs to be added as one of the build steps. You will need to update the module class and the i18nConstantsWithLookupBundle to match your application.

Example 3.6. GWT compilation

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>gwt-maven-plugin</artifactId>
    <version>1.2</version>
    <configuration>
        <inplace>true</inplace>
        <module>mypackage.GeomajasSimple</module>
        <runTarget>index.html</runTarget>
        <warSourceDirectory>war</warSourceDirectory>
        <disableCastChecking>true</disableCastChecking>
        <disableClassMetadata>true</disableClassMetadata>
        <extraJvmArgs>-Xmx512M -Xss1024k</extraJvmArgs>
        <i18nConstantsWithLookupBundle>
            mypackage.client.i18n.Simple
        </i18nConstantsWithLookupBundle>
    </configuration>
    <executions>
        <execution>
            <id>test</id>
            <goals>
                <goal>clean</goal>
                <goal>compile</goal>
                <goal>generateAsync</goal>
                <goal>test</goal>
            </goals>
        </execution>
        <execution>
            <id>i18n</id>
            <phase>generate-resources</phase>
            <goals>
                <goal>i18n</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Chapter 4. GWT widgets

This second chapter of the GWT face describes all the widgets that Geomajas has added on top of the SmartGWT widget list. Each widget will be handled in detail so that developers might get a better understanding of what they are here for, and how to use them. Know that many of these widgets are closely connected, either through configuration or coding.

1. GraphicsWidget

The `GraphicsWidget` is the basic widget that allows drawing onto a `GraphicsContext`, and catches mouse events at the same time. It implements the `MapContext` interface and provides it's own `MenuContext` implementation. As for the `VectorContext`, it delegates to a browser specific implementation (`VmlGraphicsContext` or `SvgGraphicsContext`). It is also responsible for handling `GraphicsController`s (only one global controller at a time!). The reason to place the controller handling here, is because we needed a default GWT widget to handle the events, not a SmartGWT widget. The SmartGWT events do not contain the actual DOM elements for MouseEvents, while the default GWT events do - for some functionality it is absolutely vital that it is well-known which DOM node was the target of an event.

Using the MenuContext, this widget always has the coordinates of the latest right mouse click. Usually the right mouse button is used for drawing context menus. But sometimes it is necessary to have the DOM element onto which the context menu was clicked, to influence this menu. That is why this widget always stores this latest event (or at least it's DOM element id, and screen location).

This widget is the bridge between the internal Svg or Vml rendering in GWT and the SmartGWT widget library. It is used internally in the MapWidget, but is not meant to be used directly by developers.

2. MapWidget

The main map for any Geomajas application using the GWT face. This widget controls the `MapModel`, the `MapView` objects, has an internal `GraphicsWidget` for the actual rendering, and much more. Being the most central of all widget, the MapWidget has quite a few responsibilities and options.

Map - initialization

A first responsibility of the map is the correct initialization of it's model and all layers from the configuration. When the MapWidget is added to the HTML (onDraw), it will automatically fetch the configuration from the server, and than initialize itself (more precisely, build the `MapModel`). When this is done, the `MapModel` will fire a `MapModelEvent`. Many other widgets wait for this moment to initialize themselves, as they often require the `MapModel`'s contents.

View - rendering

A second responsibility lies in the ability to render shapes. The `render()` method uses a `PainterVisitor` to recursively go through `Paintable` objects and look for the correct `Painter`. All `Painter` definitions must be registered in the `MapWidget`, by means of the `registerPainter()` and `unregisterPainter()` methods. Also the full list of `WorldPaintables` is stored within the `MapWidget`. For more information regarding the rendering, using the render method, visit the rendering manual.

As an addition of the `Paintable` objects in screen-space, the definition of a `MapAddon` has been created as well. `MapAddons` are self regulating pieces of software that are visible at a certain location on the map (in screen space!), and optionally have attached behaviour. Examples are the Navigation buttons and the scale bar.

Controller

To add interactivity to a map, you can add two types of controllers: the `GraphicsController` and the GWT `MouseWheelHandler`. For both it is possible to apply a single instance using the `setController()` and `setMouseWheelController()` methods.

Options

On top of the previous list of responsibilities, the map also has a few options that allow certain functionality to be present or not. The following options are standard:

- *navigationAddonEnabled*: this option can be configured from within the configuration, and determines whether or not the navigation `MapAddon` is visible. This `MapAddon` is placed in the upper left corner of the map and allows the user to pan, zoom in and out, and zoom to maximum extent.
- *scaleBarEnabled*: this option can be configured from within the configuration, and determines whether or not the scale bar is visible. This shows you the scale of the map by means of a bar of certain length, expressed in the preferred unit type (metric versus English).
- *zoomOnScrollEnabled*: this option determines whether or not the `ZoomOnScrollController` is active by default. This allows zooming in and out on the map using the mouse wheel.

Figure 4.1. MapWidget example

3. OverviewMap

The overview map is an extension of the `MapWidget`, which keeps the overview of a target `MapWidget`. It keeps track of the target map's view, and reacts whenever that target map changes it's view. The `OverViewMap` implements the `MapViewChangedHandler` to track the changes of it's target map. As it is an extension of a normal `MapWidget`, it has all the functionality of a normal map. So you can configure layers for an overview map, just as you would for a normal map.

Figure 4.2. OverviewMap example

What can be tricky is to properly configure the bounds of the overview map. In the image above, this is slightly bigger than the world (which in this case is probably not what you want). The bounds for the overview map normally uses the initial bounds as defined on the map. However, when you set the useTargetMapMaxExtent parameter to true when creating the overview map, following locations are checked, using the first which was configured:

1. targetMap.maxBounds

2. union of layer extents

The bounds are automatically extended by a percentage. The default is 5%, but this can be configured when creating the overview map.

4. Toolbar

The Geomajas tool bar is an extension of the SmartGWT `ToolStrip` widget, and allows for many different widgets to be added to it. A tool bar must be initialized with an instance of the `MapWidget` it is related to. When the `MapWidget` has successfully initialized itself, it's `MapModel` will fire the `MapModelEvent` saying so. The tool bar reacts on this event by searching in the map configuration for the correct list of tool bar buttons. The map configuration can contain tool ids to indicate the tools which need to be added, together with optional parameters. Using the `ToolbarRegistry`, which contains the mappings between these ids and the relevant `ToolbarAction` or `ToolbarModalAction` classes, the tool bar will initialize itself (for more info; see User Interaction).

Existing tools which can be defined include:

- *EditMode*: a ToolbarModalAction for editing on the map. Allows the user to create new objects in the selected layer, and allows updating and deleting of selected objects.

- *MeasureDistanceMode*: a ToolbarModalAction which allows the user to measure distances on the map.

- *SelectionMode*: allow selecting features either by clicking on them, or by dragging a rectangle, thus selecting the features which are inside the rectangle. You need an active (vector) layer for the selection to work. The right click menu allows clearing the selected features and toggling selection at the current position. Press shift or control while selecting to add the selection to the previously selected features. Possible parameters:

    - *clickTimeout*: when the button it released in less than the number of milliseconds specified here, then the selection is treated as a click. When it takes longer, it is treated as dragging. Default is "500" (ms).

    - *coverageRatio*: ratio of the feature which needs to be inside the selected area for the feature to be selected. When this is "1.0" then the entire feature needs to be inside the selection rectangle. Default is "0.7".

    - *priorityToSelectedLayer*: when this is "true" selection will first check the selected layer, and use default behaviour only if nothing is found in that layer. Default behaviour is to try all visible layers, from front to back.

    - *pixelTolerance*: number of pixels of tolerance allowed when trying to select features. The default pixel tolerance is 5.

- *ZoomIn*: zoom in to the map at the location clicked (will be centered), using the zoom factor which is configured.

    - *delta*: zoom in factor, should be >1 to effectively zoom in.

- *ZoomOut*: zoom out of the map at the location clicked (will be centered), using the zoom factor which is configured.

    - *delta*: zoom in factor, should be in the]0,1[range to effectively zoom out

- *PanMode*: this action allows you to pan the screen by dragging. When keeping either the shift or control key down, it is also possible to indicate an area to zoom into (like *ZoomToRectangleMode*).

- *ZoomToRectangleMode*: you can indicate a rectangle (by dragging) and it will zoom to make the selected area as big as possible while still entirely inside the map widget.

- *ZoomToSelection*: first select some items on the map. After clicking on the the zoomToSelection button the map will be zoomed so that all selected items will fit nicely on the screen.

- *panToSelection*: first select some items on the map. After clicking on the panToSelection button the map will be panned in such a way that the center of the selected items is in the center of the screen.

- *ZoomPrevious*: go back to the previous zoom level (and location).

- *ZoomNext*: go forward again, cancelling a click on *ZoomPrevious*.

Figure 4.3. Tool bar example

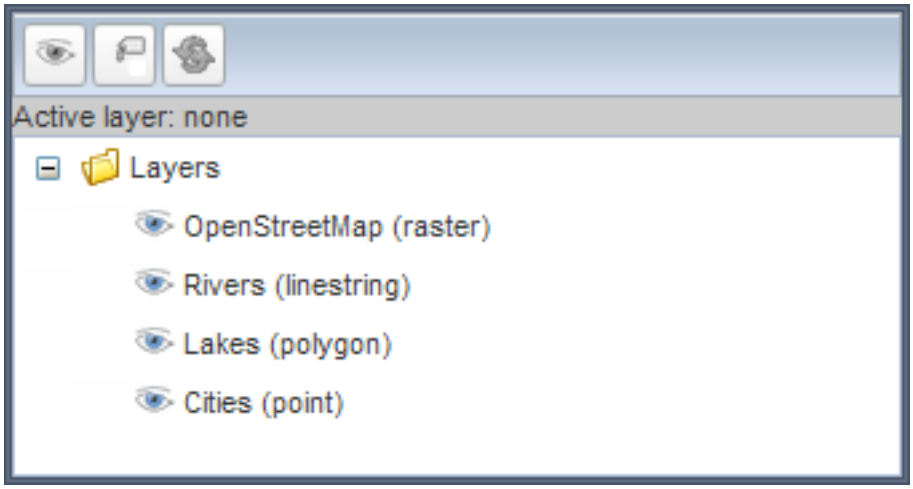
5. LayerTree

This widget represents a view on the map model which is focused on layers. You see the map layers in a tree, just as they are configured. Accompanied with this view, there are buttons that define certain actions on these layers. Originally there are no buttons in this widget, so they have to be added manually or through configuration. These buttons can either be single actions or selectable buttons (similar to the `Toolbar` widget - see User Interaction).

Just like the tool bar, the `LayerTree` waits for the `MapModel` to be initialized, and also reacts to the `MapModelEvent`. The layer tree configuration is contained in the map configuration. When the `MapModelEvent` is fired, the `LayerTree` will read configuration to know the layer tree structure, which buttons to include,...

Below you see a screen shot of a simple `LayerTree` where no layer has been selected (and thus all the buttons are disabled):

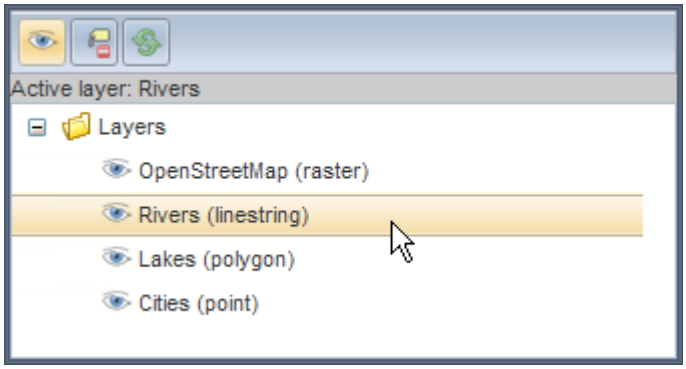
Figure 4.4. LayerTree example



Once a layer is selected, the `LayerTree` will ask all buttons whether they should be enabled for that layer. For example, the

`org.geomajas.gwt.client.action.layertree.LabelAction`, which toggles a layer's labels, is only applicable on vector layers, so if the selected layer is a raster layer, that button will remain disabled. The same `LayerTree` with a selected layer looks like this:

Figure 4.5. LayerTree with selected layer



The `LayerTree` has few public methods, but it does quite a lot behind the screen. The tree is a SmartGWT `TreeGrid`, where the `LayerTree` adds handlers to the nodes and leaves (using `LeafClickHandler` and `FolderClickHandler`), which trigger layer selection in the `MapModel`. The `LayerTree` also listens to layer selection events, to adjust it's own appearance. For example, when a layer is selected, the proper node has to be selected and all the buttons updated.

The `LayerTreeAction` and `LayerTreeModalAction` are also specifically designed to cope with the different stages that they should be able to display. The `LayerTreeModalAction` can be disabled, enabled and selected or enabled and deselected. For each it is imperative that clear markings are given. This means that different icons are usually used for the different stages. These different icons should be given to the actions at construction time.

Currently the following actions are defined:

- *org.geomajas.gwt.client.action.layertree.VisibleAction*: a `LayerTreeModalAction` that switches the visible flag on the selected layer.
- *org.geomajas.gwt.client.action.layertree.LabelAction*: a `LayerTreeModalAction` that switches the display of labels for the selected layer.
- *org.geomajas.gwt.client.action.layertree.LayerRefreshAction*: a `LayerTreeAction` that refreshes the selected layer on click.

6. Legend

The `Legend` is a graphical widget that shows all styles for the currently visible vector layers. In that sense it is another view on the map model that only shows the style definitions that are currently relevant. Just like the map widget, the legend is built on `GraphicsWidget`. It reads the available layers from the map model and draws a legend to match the style of these layers.

Figure 4.6. Legend

7. FeatureListGrid

The `FeatureListGrid` is a table listing the attributes of features within a single vector layer. Each feature is represented by a row in the grid, with at the top a header that shows the attribute label, as configured in the configuration. As only vector layers can contain features, the grid will be empty for raster layers.

The `FeatureListGrid` has a few options that determine it's behaviour and appearance:

- *selectionEnabled*: enables or disables selection of features when selecting a row in the table. When this is enabled, the table will keep feature selection consistent with the map model. If the user selects a row, the feature will also be selected on the map.

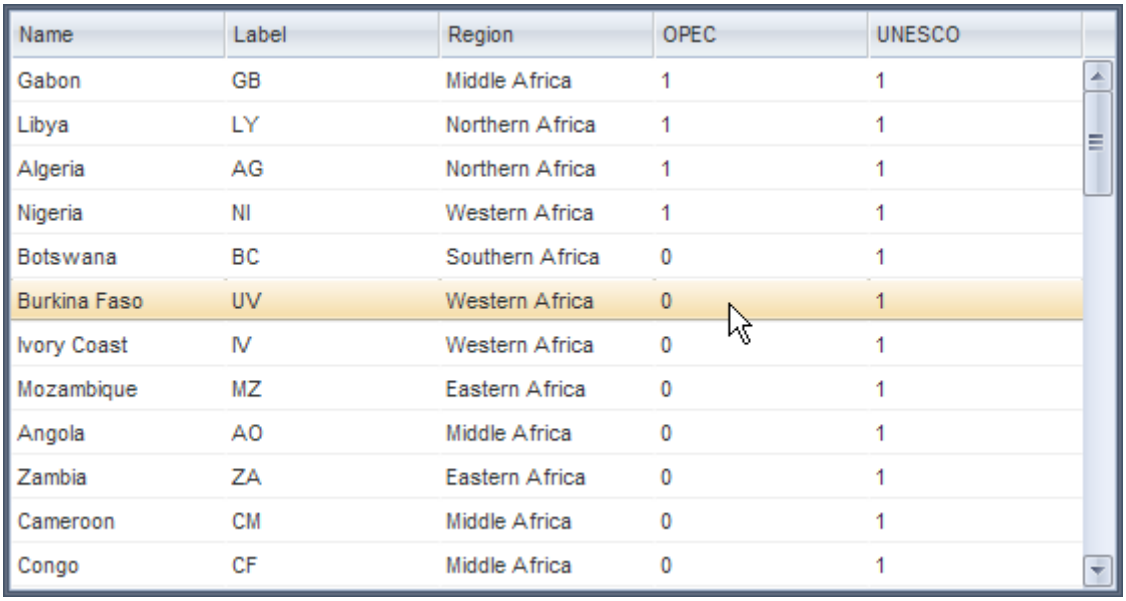
- *allAttributesDisplayed*: show all attributes (true) or only the 'identifying' attributes (false)? Attributes can be configured as "identifying" in the configuration. This difference allows for a select list of attributes to be visible in the grid, keeping overview. The user can always ask more details by double clicking the line.

- *editingEnabled*: determines whether or not editing the attributes is allowed. When double clicking a row in the table, a `FeatureAttributeWindow` will appear, containing the feature of the selected row. This setting determines if that window allows editing or not.

- *idInTable*: show the feature's id in the table. This is false by default, and should not really be necessary.

This table is an extension of the SmartGWT `ListGrid` widget. It automatically has grouping, filtering and sorting abilities (and much more...). The `FeatureListGrid` makes it possible to easily display features. You have to set the layer from which to display features. Then you can add features one by one. If no layer is set, then then "`addFeature`" method will not add any rows to the table. Setting the layer will automatically create the grid header, using the layer's attribute definitions.

Figure 4.7. FeatureListGrid example



Name	Label	Region	OPEC	UNESCO
Gabon	GB	Middle Africa	1	1
Libya	LY	Northern Africa	1	1
Algeria	AG	Northern Africa	1	1
Nigeria	NI	Western Africa	1	1
Botswana	BC	Southern Africa	0	1
Burkina Faso	UV	Western Africa	0	1
Ivory Coast	IV	Western Africa	0	1
Mozambique	MZ	Eastern Africa	0	1
Angola	AO	Middle Africa	0	1
Zambia	ZA	Eastern Africa	0	1
Cameroon	CM	Middle Africa	0	1
Congo	CF	Middle Africa	0	1

8. FeatureAttributeWindow

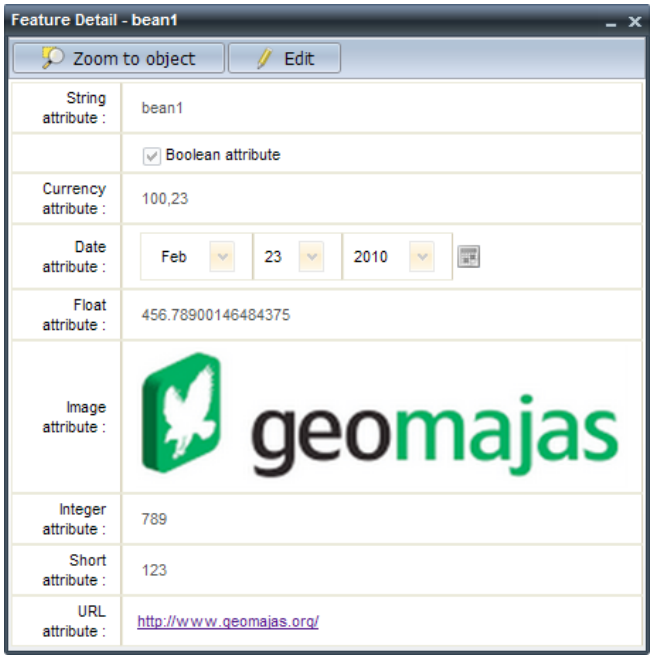
The `FeatureAttributeWindow` is a floating window to display and enable editing of a feature's attributes and persist these changes. This widget is a `FeatureAttributeEditor` with some extra buttons like "save". When setting a feature, it first makes a clone so you are not editing the feature directly and changes are only applied when the save is clicked. This widget also checks whether or not all fields are valid, and will not allow saving when at least one of the fields is not valid.

The `FeatureAttributeWindow` has the following options:

- editingAllowed: is editing allowed? This must be set *before* the widget is actually drawn.
- editingEnabled: is editing currently enabled or not? This widget can toggle this value on the fly. When editing is enabled, it will display an editable attribute form with save, cancel and reset buttons. When editing is not enabled, these buttons will disappear, and a simple attribute form is shown that displays the attribute values, but does not allow editing. This effect only works when *editingAllowed* is true.

Below is a screen shot of a `FeatureAttributeWindow` with editing allowed but not enabled. Without the editing allowed, there would be now "edit" button. The button itself triggers setting editing enabled.

Figure 4.8. FeatureAttributeWIndow, editing allowed but not enabled



9. ActivityMonitor

This widget monitors client-server communication traffic, and displays that activity to the end-user. It's purpose is inform the user that a server request is in progress. For example, when the user zooms in, it can sometimes take a few seconds before everything is redrawn. This widget displays that traffic by listening to the `GwtCommandDispatcher` events. On the screen shot below, you can see the difference between it being passive and active:

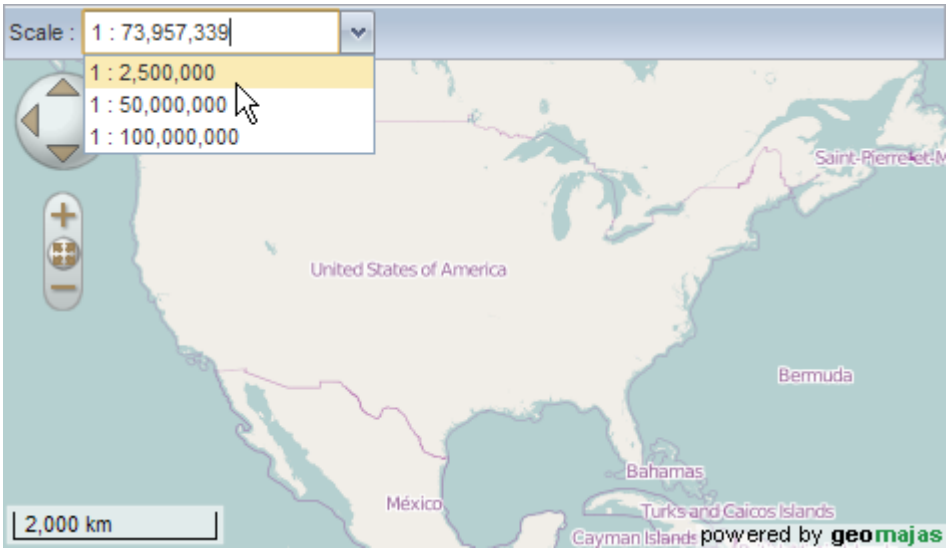
Figure 4.9. ActivityMonitor example

10. ScaleSelect

Combo box for changing the scale on the map which can be added to a tool bar. It displays a list of possible scales to choose from, but also allows the user to type a specific scale. The scale select is constructed with a `MapView` as parameter. If this `MapView` contains pre-configured resolutions (zoom-steps - these can be set in the configuration), than the scale select will allow selection from these scales. If no resolutions are present, the scale select will automatically choose relevant scales to choose from.

Using the `setScales()` method, one can always override the list of scales in the widget.

Figure 4.10. ScaleSelect example



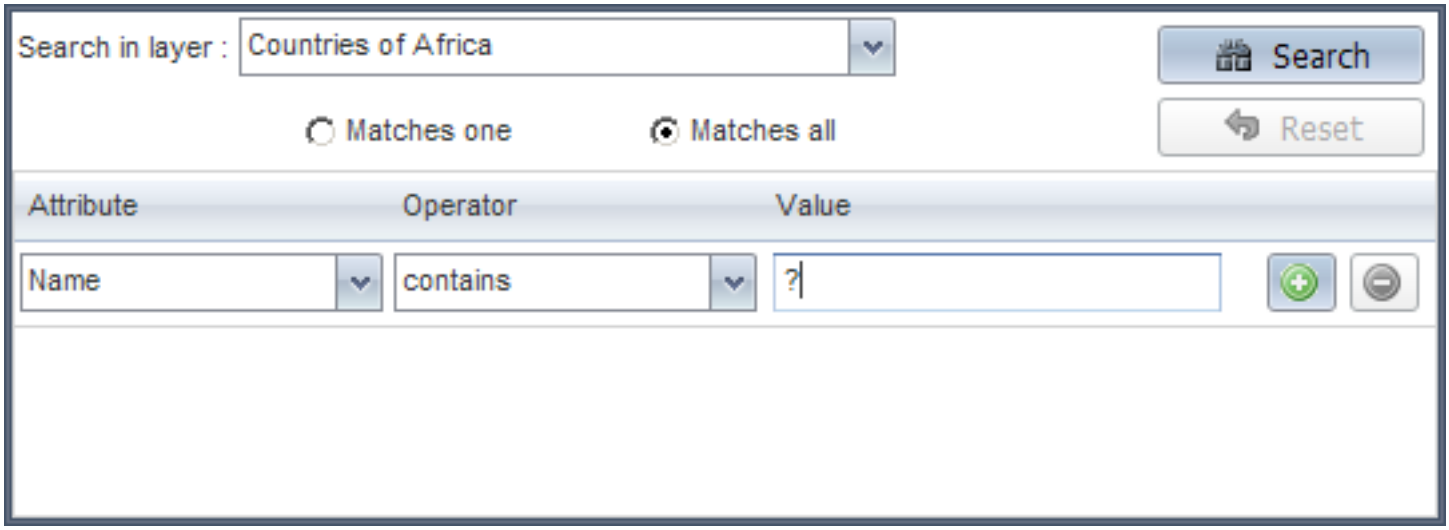
11. FeatureSearch

Widget that supports searching for features on the attributes. Requires a value for `manualLayerSelection` at construction time. If true, a select box will be shown so the user can select what layer to search in. The possible list of layers consists of all the vector layers that are present in the map model. If false, this widget will search in the currently selected layer.

When the "search" button is indicated, the search will be performed server-side. When the result returns, a `SearchEvent` is fired. This event holds a reference to the `VectorLayer` in which the search took place, and a list of all the features that were found. In order to do something with the results (such as displaying in a `FeatureListGrid`), add a `SearchHandler`. For the specific case of displaying the feature in a `FeatureListGrid`, there is a `DefaultSearchHandler` that already does this.

Note that there is a limit to the number of features that are returned. By default this limit is set to 100. Modify `maximumResultSize` to change this. Note that a high limit can have a serious impact on performance and memory consumption!

Figure 4.11. FeatureSearch example



Chapter 5. How-to

1. How to avoid error messages when changing the current window location ?

Sometimes you will want to redirect the current browser window to another location (different URL). This is the expected behaviour when a user clicks on an external link, for example. When the client is still waiting for outstanding command responses, this can cause warning messages to pop up (default Geomajas behaviour). To avoid such messages, use the following code:

```
WindowUtil.setLocation("<put your external url here>");
```