Geomajas Editing plug-in documentation

Geomajas Developers and Geosparc

Geomajas Editing plug-in documentation

by Geomajas Developers and Geosparc

Version 2.4.1

Copyright © 2010-2015 Geosparc nv

Table of Contents

1. Introduction 1
  1. What does this plugin do? 1
  2. Maven configuration 2
2. Editing API 4
  1. Getting started 4
  2. Geometry editing 4
    2.1. GeometryEditor behind the screens 4
    2.2. General workflow 5
    2.3. The GeometryIndex concept 5
    2.4. The central editing services 6
    2.5. The editing state 6
    2.6. Examples 7
  3. Using snapping while editing 8
    3.1. Snapping options 8
    3.2. Snapping rules 8
  4. Merging geometries 9
  5. Splitting geometries 10
3. How-to 11
  1. How to create a new geometry 11
  2. How to add an interior ring 11
  3. How to delete an interior ring 12

List of Tables

2.1. Geometry index samples 6

Chapter 1. Introduction

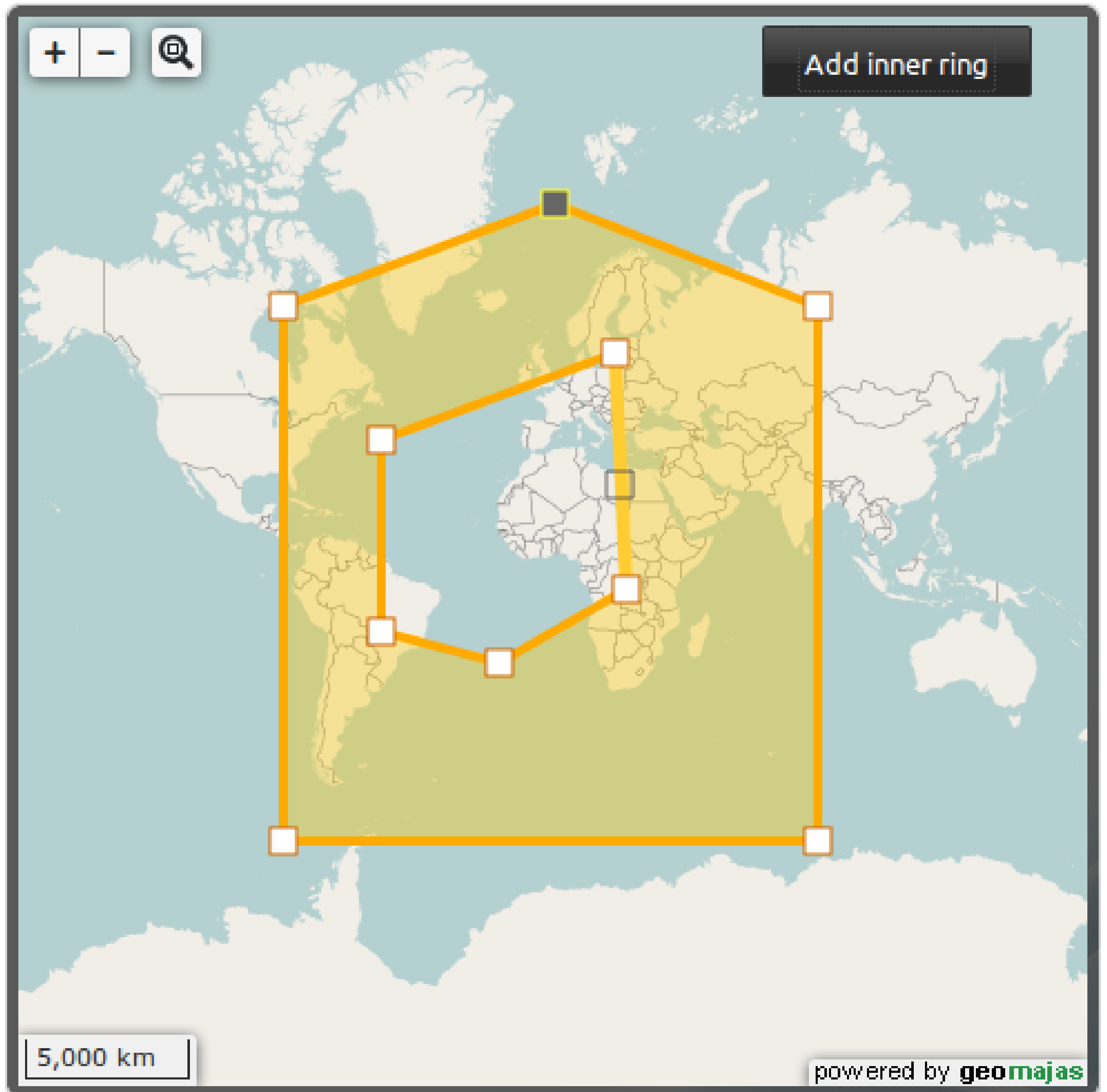
1. What does this plugin do?

In short, this plugin provides a set of services and utilities for editing and manipulating geometries. If at any time you need to create new geometries, split existing geometries, or simple calculate buffers, etc. this plugin does just that.

A short overview of functionalities currently within this plugin:

- Visually manipulating the shape of a geometry on the map. This allows the user interact with the map to create new geometries or change the shape of existing geometries. This is often used to change the geometries of features on the map, or for redlining, or ....
- Snapping: While editing a geometry, it is possible to apply snapping rules that help the user during geometry editing. This snapping rules are fully customizable.
- Splitting geometries: A separate service has been created where the user can draw a line to split a geometry into multiple parts. (requires a dependency on the Geomajas server project)
- Merging geometries: Using this functionality it is possible to create the union of several geometries. (requires a dependency on the Geomajas server project)

Here is a screenshot of what the default geometry editing within this plugin could look like:



2. Maven configuration

In order to use this plug-in in combination with the GWT face, the following Maven dependency is required:

```
<dependency>
    <groupId>org.geomajas.plugin</groupId>
    <artifactId>geomajas-client-gwt2-plugin-editing-impl</artifactId>
    <version>2.4.1</version>
</dependency>
```

If you want splitting or merging support, you will have to use the following dependency:

```
<dependency>
    <groupId>org.geomajas.plugin</groupId>
```

```
    <artifactId>geomajas-client-gwt2-plugin-editing-server-extension</artifactI
    <version>2.4.1</version>
</dependency>
```

This in turn includes a dependency on the Geomajas server project.

Chapter 2. Editing API

1. Getting started

As with most Geomajas plugins, the editing plugin has a single class that provides access to all the services and functionalities within the plugin:

```
org.geomajas.gwt2.plugin.editing.client.Editing
```

From here on, it's a simple matter of getting the required services and using them. This class provides the following services:

- *GeometryEditor*: The GeometryEditor is a map specific entity for managing the geometry editing process. It is used to visually create a new geometry on the map, or manipulating an existing geometry on the map. Note that this is a visual process. The GeometryEditor will automatically make sure that the special controllers are in place to ensure the correct user interaction with the map, and it will also install a renderer to visually aid the editing process.
- *GeometrySplitService*: This service assists in splitting an existing geometry into multiple parts. It will allow the user to draw a "splitting line" by which the splitting process is calculated.
- *GeometryMergeService*: This service assists in merging multiple geometries into a single union.

In the sections that follow, we will go over these services one by one, starting with the most important one: the basic geometry editing.

2. Geometry editing

This section describes how to use the basic geometry editor. The GeometryEditor is used to let the user create new geometries, or change the shape of existing geometries. Since this process works through user interaction on a map, a GeometryEditor is always bound to a single map. The following piece of code create a new GeometryEditor:

```
GeometryEditor editor = Editing.getInstance().createGeometryEditor(mapPresenter
```

Once you have a GeometryEditor, you can immediately start editing geometries. But before you do, allow us to better explain how exactly it works.

2.1. GeometryEditor behind the screens

In essence the GeometryEditor uses the tried and tested Model-View-Controller principle:

- *Model*: The `GeometryEditService`. It keep track of the geometry editing process. It provides methods for starting and stopping the editing process, as well as altering the shape of the geometry. Furthermore, it even provides undo/redo methods. This is the central geometry editing service.
- *View*: The `GeometryRenderer`. While manipulating the geometry through the `GeometryEditService`, it fires events. These events are then caught be the `GeometryRenderer` who renders the geometry accordingly.
- *Controller*: While rendering the geometry on the map, the `GeometryRenderer` installs a series of controllers/handlers on the different vertices and edges that are drawn. These controllers allow the user to i.e. drag vertices to different locations, by making calls to the `GeometryEditService`.

Note

The renderer is one example of a listener to the many events that the `GeometryEditService` fires, but essentially anyone can listen to those events. If you need to react to any change in a geometry's shape, just add the correct handler.

2.2. General workflow

The central service determining the editing workflow is the GeometryEditService. It can be acquired as such:

```
GeometryEditService editService = editor.getEditService();
```

Editing a geometry comes down to starting the editing process, applying some operations and then stopping the process again. Starting and stopping happens through the `start` and `stop` methods:

```
// Start the editing process on some geometry:
editService.start(geometry);

// The user now performs several operations on the map (dragging point, insertin
// ....

// Finally through some button the user decides the geometry editing is ready, s
editService.stop();
```

The editing process happens through user interaction. Usually there will be some buttons or other widgets that help in that process, such as a "save" button, or undo/redo buttons. Such buttons are not provided out-of-the-box. It is up to you to determine how the editing process should be implemented in your applications.

Note

Know that operations onto the geometry really do apply on the same geometry that was passed with the `start` method. In other words, this service changes the original geometry. If you want to support some roll-back functionality within your code, make sure to create a clone of the geometry before starting this edit service.

2.3. The GeometryIndex concept

Before trying to figure out how the `GeometryEditService` works, it is important to understand the `GeometryIndex` concept. A `GeometryIndex` is an index within a geometry that points to a single vertex, edge or sub-geometry. All operations within the `GeometryEditService` operate on a list of such `GeometryIndices`.

Take for example the "move" operation. This operation will move the given `GeometryIndex` to the specified location. This operation is used when the user drags a vertex around on the map, but this operation could also be used to let the user drag an interior ring within a polygon, or an entire LineString within a MultiLineString, or event the whole geometry.

The GeometryIndex is based on the internal structure of the geometry, which may contain 4 or more levels:

1. Geometry collection level: this is the highest structural level for geometry collections: multipolygon, multilinestring or arbitrary geometry collections. In theory a geometry collection may contain other geometry collections, but this is rarely encountered.

2. Geometry level: this is the level of a basic geometry like polygon, linestring or point

3. Ring level: for a polygon, this is the level of the linear rings. There is usually an exterior ring (boundary), but there may also be additional interior rings (holes)

4. Vertex or edge level: this is the level of the individual vertices and edges. A single edge connects 2 vertices.

The elements at each level have a fixed ordering, which makes it possible to uniquely determine such an element by its order at each level of the structural tree. This combination of order numbers, together

with a type to distinguish between edges, vertices or higher level structures (which we generally call geometries) forms the `GeometryIndex`.

Lets give some examples to clarify this. The following table shows at the left column a geometry in WKT format with a highlighted section and the corresponding `GeometryIndex`. The GeometryIndex is an array of integers combined with a type. For edge, the type is `edge`, for vertex it is `vertex` and for all other structures it is `geometry`. The last row contains a multipolygon with 2 polygons. The highlighted section is a couple of points that determines an edge of the interior ring of the first polygon. The indices are 0 (for the first polygon), 1 (for the interior ring) and 2 for being the 3rd edge of this ring (counting starts with index 0 in all cases).

Table 2.1. Geometry index samples

WKT	Geometry index
POINT(**0 0**)	[0], type = vertex
LINESTRING (30 10, **10 30**, 40 40)	[1], type = vertex
LINESTRING (**30 10**, 10 30, **40 40**)	[2], type = edge
POLYGON ((35 10, 10 20, 15 40, 45 45, 35 10), (**20 30, 35 35, 30 20, 20 30**))	[1], type = geometry
POLYGON ((35 10, 10 20, 15 40, 45 45, 35 10), (20 30, 35 35, **30 20**, 20 30))	[1,2], type = vertex
MULTIPOLYGON (((35 10, 10 20, 15 40, 45 45, 35 10), (20 30, 35 35, **30 20, 20 30**)),((35 10, 10 20, 15 40, 45 45, 35 10)))	[0,1,2], type = edge

2.4. The central editing services

There are 3 central services that help in the editing process. All three have a very distinct responsibility:

- *GeometryEditService*: Defines the editing workflow and the basic operations (with undo/redo) that are supported. Also allows to add handlers to all events.
- *GeometryIndexService*: This service defines operations for creating and manipulating GeometryIndices. It also supports retrieving information based upon a certain geometry and index. For example what are the adjacent vertices to a certain edge within a given geometry?
- *GeometryIndexStateService*: Keeps track of the state of all indices that make up the geometry being edited. It allows for selecting/deselecting, enabling/disabling, highlighting, etc any vertices/edges/sub-geometries during the editing process. This state can then be used by the controllers. For example, a controller could allow only selected vertices to be dragged by the user.

There are more services then the 3 mentioned above such as a `SnapService`, `GeometryMergeService` and `GeometrySplitService`, but those just add more functionality to the basic set that the 3 above already provide. They will be discussed later.

2.5. The editing state

At any time during the editing process, the `GeometryEditService` has a general state that tells you what's going on. This state is defined in the `GeometryEditState`. Currently there are 3 possible states for the editing process to be in:

- *IDLE*: The default state. The editing process always reverts to this state when the user is not interacting with the controllers on the map or the handlers on the edges and vertices.
- *INSERTING*: The user is currently inserting new points into the geometry. The `GeometryEditService` has an "insertIndex" (of the type `GeometryIndex`), that points to

the next suggested insert location. The controllers pick up on this index to insert points (or edges, or geometries).

- *DRAGGING*: The user is currently dragging a part of the geometry. The `GeometryIndexStateService` can select vertices/edges/sub-geometries, which can then be dragged around.

As you may have noticed from the descriptions, the GeometryEditState is used mainly within the controllers that operate on the map. An insert controller will only be active when the edit state is "INSERTING". Likewise a drag controller will only be active when the edit state is "DRAGGING".

2.6. Examples

Now that the basic concepts have been described, it is time to show by example.

2.6.1. Editing an existing geometry

Let's start of with the most basic of examples: how to let the user edit a certain geometry. Let's say this geometry comes from a feature within a FeaturesSupported layer:

```
GeometryEditor editor = Editing.getInstance().createGeometryEditor(mapPresenter
GeometryEditService editService = editor.getEditService();

Geometry geometry = feature.getGeometry();
editService.start(geometry);
```

And that's it. The geometry editing process has been started on the features geometry.

2.6.2. Creating a new geometry

This example shows how to create a new LineString geometry using the editing process. The idea is that the user just clicks on the map to determine where to add/insert additional points into the geometry. Therefore we start the editing process using an empty geometry and set the editing state to "INSERTING":

```
// Create an empty point geometry. It has no coordinates yet. That is up to the
Geometry line = new Geometry(Geometry.LINE_STRING, 0, -1);
editService.start(line);

// Set the editing service in "INSERTING" mode.

// Make sure it starts inserting in the correct index: the first vertex within t
GeometryIndex index = editService.getIndexService().create(GeometryIndexType.TY
editService.setEditingState(GeometryEditState.INSERTING);
editService.setInsertIndex(index);

// Et voila, the use may now click on the map...
```

Perhaps some extra clarification is in order here. We want the user to start adding points to our empty LineString. In order to do this, we change the editing state to "INSERTING". The problem is that the service does not yet know where in the geometry it should insert points, so we must provide that information. For a LineString this may seem overly complex, but consider a MultiPolygon where we want to add a new inner ring in one of the Polygons. We better pick the correct Polygon to insert an inner ring.

In any case, when switching to "INSERTING" state, you need to provide an "InsertIndex". We use the `GeometryIndexService` to create such an index. In this case, the first vertex within the LineString geometry.

3. Using snapping while editing

3.1. Snapping options

The editing plug-in has support for snapping while inserting or dragging. The controllers are equipped with a `SnapService` which can convert the mouse event locations into snapped locations, before they are passed to the `GeometryEditService` for operation execution. This `SnapService` is actually provided by the `GeometryEditor`.

In order to activate snapping for both inserting and dragging state, use the following code:

```
editor.setSnapOnDrag(true);
editor.setSnapOnInsert(true);
```

3.2. Snapping rules

The `SnapService` works through a series of rules that need to be active. Without any snapping rules, the `SnapService` will not snap. Adding snapping rules, goes through the "`addSnappingRule`" method, and requires the following parameters:

- *algorithm*: The snapping algorithm to be used. For example, snap to end-points only, or also to edges, or...
- *sourceProvider*: The provider of target geometries where to snap. For example, snap to features of a layer, or to a grid, or...
- *distance*: The maximum distance to bridge during snapping. Expressed in the unit of the map CRS.
- *highPriority*: High priority means that this rule will always be executed. Low priority means that if a previous snapping algorithm has found a snapping candidate, this algorithm will not be executed anymore.

3.2.1. SnapSourceProvider

Let us start with the `SnapSourceProvider`. It's task is to provide sources (geometries) for the `SnapService` to snap to. An often used source of geometries to snap to, are the features of some `FeaturesSupported` layer. The SnapSourceProvider has only 2 methods:

- *update*: This method is called every time the map navigates. It provides the current location on the map. This is used so that the `SnapSourceProvider` may better choose which geometries to provide for the snapping process.
- *getSnappingSources*: This method is always called after the `update` method. This is when the `SnapSourceProvider` is asked for a list of geometries for the `SnapService` to use.

Let's consider the case of snapping to the features of a "Countries" layer. The countries layer contains all the countries in the world. When zooming in to North America, it's no use for the `SnapSourceProvider` to also provide the countries of Europe or Asia. By knowing the map extent, the `SnapSourceProvider` may provide a shorter list of geometries, making the snapping process more efficient.

Note

Right now, there are no default implementations of the SnapSourceProvider. You will have to implement this yourself.

3.2.2. Snapping algorithms

Next are the snapping algorithms. There are multiple ways of snapping. The editing plugin provides multiple algorithms out of the box:

- *NearestVertexSnapAlgorithm*: This algorithm snaps only to the vertices of a geometry.
- *NearestEdgeSnapAlgorithm*: This snapping algorithm snaps not only to the vertices of a geometry but also to any point along the edges. Of course this snapping algorithm takes more processing time than the `NearestVertexSnapAlgorithm`, but gives a far smoother snapping experience.

3.2.3. Defining snapping rules

Now let us proceed to actually show how to add snapping rules. First note that is it possible to add multiple snapping rules. Each rule may provide it's own `SnapSourceProvider` and `SnappingAlgorithm`. Through the "highPriority" setting, they `SnapService` will determine order and whether or not they should always be used.

More about that later, let us first demonstrate how to add a single snapping rule. In this example we assume that we have created a `SnapSourceProvider` implementation called `CountriesLayerProvider`. This implementation can actually be found within the examples that come with this plugin (see showcase).

```
SnapAlgorithm snapAlgorithm = new NearestVertexSnapAlgorithm();
SnapSourceProvider sourceProvider = new CountriesLayerProvider();
SnappingRule rule = new SnappingRule(snapAlgorithm, sourceProvider, 1000.0);
editor.getSnappingService().addSnappingRule(rule);
```

If the editing process should start now, it would make use of this snapping rule while inserting points or dragging points (assuming we have activated both).

4. Merging geometries

Next to the default geometry editing, this plugin also provides a service for merging geometries. Just as the `GeometryEditService`, this service is meant to be used visually, on the map. The `GeometryMergeService` can be acquired as such:

```
GeometryMergeService mergeService = Editing.getInstance().getGeometryMergeServi
```

Just like the `GeometryEditService`, the `GeometryMergeService` uses a `start` and a `stop` method to start and stop the merging process. In between the `start` and `stop` calls, it is possible to add (or remove again) geometries to the service. The `stop` method will then execute the merge on the list of geometries added to the `GeometryMergeService`.

Here is a simple example:

```
GeometryMergeService mergeService = Editing.getInstance().getGeometryMergeServi
mergeService.start();
mergeService.addGeometry(feature1.getGeometry());
mergeService.addGeometry(feature2.getGeometry());
mergeService.stop(new GeometryFunction() {

    public void execute(Geometry geometry) {
        // Do something with the resulting geometry ...
    }
});
```

Every method in the GeometryMergeService fires an appropriate event, so it is always possible to have multiple listeners to the geometry merging process:

- *GeometryMergeStartEvent*: fired when the start method is called. This signals the start of the geometry merging process.
- *GeometryMergeStopEvent*: signals the end of the geometry merging process. It is fired at the very end, after the result has been processed.

- *GeometryMergeAddedEvent*: fired when a geometry has been added to the `GeometryMergeService`.
- *GeometryMergeRemovedEvent*: fired when a geometry has been removed again from the `GeometryMergeService`.

5. Splitting geometries

Of course, if it's possible to merge geometries, it must also be possible to split geometries. This can be achieved through the `GeometrySplitService`. The `GeometrySplitService` lets the user draw a "splitting line", that is then used to split up the original geometry into multiple parts, as indicated by the splitting line. The splitting line is drawn through a `GeometryEditService` that lets the user create a new `LineString` geometry. This `LineString` is then used to split the original geometry.

A GeometrySplitService can be acquired as follows:

```
GeometryEditor editor = Editing.getInstance().createGeometryEditor(mapPresenter
GeometryEditService editService = editor.getEditService();
GeometrySplitService splitService = Editing.getInstance().createGeometrySplitSe
```

Analogous with the previous services, this service too uses a `start` and a `stop` method. The start method requires you to pass the original geometry. It will automatically invoke the `GeometryEditService` to let the user create a new `LineString` geometry (the splitting line). Once it's decided that the splitting line is sufficient, the stop method on the `GeometrySplitService` can be called.

This is how to start the service:

```
splitService.start(feature.getGeometry());
```

Usually some button will be foreseen that calls the stop method. Stopping goes as follows:

```
splitService.stop(new GeometryArrayFunction() {

    public void execute(Geometry[] geometries) {
        // Do something with the resulting geometries ...
    }
});
```

The `GeometrySplitService` too will fire events on the start and stop methods:

- *GeometrySplitStartEvent*: Fired when the splitting process is started.
- *GeometrySplitStopEvent*: Fired when the splitting process has stopped. It is fired at the very end, after the resulting geometries have been processed.

Note

While the user is busy drawing the splitting line, the `GeometryEditService` will fires it's normal events. That means it's also possible to listen to those events as well during the drawing phase.

Chapter 3. How-to

This chapter shows how to perform some of the most common editing operations.

1. How to create a new geometry

This section describes how to let the user draw a new geometry of some pre-defined type. The idea is that the user can click on the map to insert vertices into the geometry. This requires three steps:

1. Set up a `GeometryEditor` for the map. The editor is responsible for drawing the edited geometry and setting the correct event handler for capturing user events
2. Prepare an initial (empty) geometry for the editor. The editor always operates on a single geometry, which has to be set programmatically.
3. Prepare the intial state of editing. During the editing phase, the editor is in one of the 3 main states: idle (waiting for the user to select), inserting (inserting vertices) or dragging (dragging a part of the geometry). To start drawing on an empty geometry, the inserting state has to be activated and the insert index (index of the vertex that will be inserted) should be set.

The following code has to be executed:

```
GeometryEditor editor = new GeometryEditorImpl(map); // (1)
Geometry polygon = new Geometry(Geometry.POLYGON, 0, 0);  // (2)
editor.getEditService().start(polygon); // (2)
try {
    GeometryIndex index = editor.getEditService().addEmptyChild(); // (3)
    editor.getEditService().setInsertIndex(editor.getEditService().getIndexServ
    editor.getEditService().setEditingState(GeometryEditState.INSERTING); // (3
} catch (GeometryOperationFailedException e) {
    editor.getEditService().stop();
    Window.alert("Exception during editing: " + e.getMessage());
}
```

From there on, the user can take over. Depending on the use case, the editing service could be stopped by letting the user click outside the finished geometry

```
editor.getBaseController().setClickToStop(true);
```

or by explicitly stopping the service:

```
editor.getEditService().stop();
```

The editing happens on the same object that was originally passed to the service.

2. How to add an interior ring

The following steps have to be taken:

1. Add an extra empty ring to the polygon
2. Prepare the editing state to start inserting at the first child index of the ring

The following code assumes that the polygon being edited already has an exterior ring:

```
try {
     GeometryEditService service = editor.getEditService();
     GeometryIndex ringIndex = service.addEmptyChild(); // (1)
```

```
        // Free drawing means inserting mode. First create a new empty child, th
        // index to the child's first vertex:
        service.setInsertIndex(service.getIndexService().addChildren(ringIndex,G
        service.setEditingState(GeometryEditState.INSERTING);// (2)
    } catch (GeometryOperationFailedException e) {
        Window.alert("Error during editing: " + e.getMessage());
    }
```

3. How to delete an interior ring

The following steps have to be taken:

1. Find the correct index for the ring

2. Call the service to remove the ring

The following code deletes the first interior ring if the polygon has one:

```
GeometryEditService service = editor.getEditService();
Geometry geometry = service.getGeometry();
if(geometry.getGeometries().length > 1) {
    GeometryIndex ringIndex = service.create(GeometryIndexType.TYPE_GEOMETRY, 1
    service.remove(Collections.singletonList(ringIndex)); // (2)
}
```