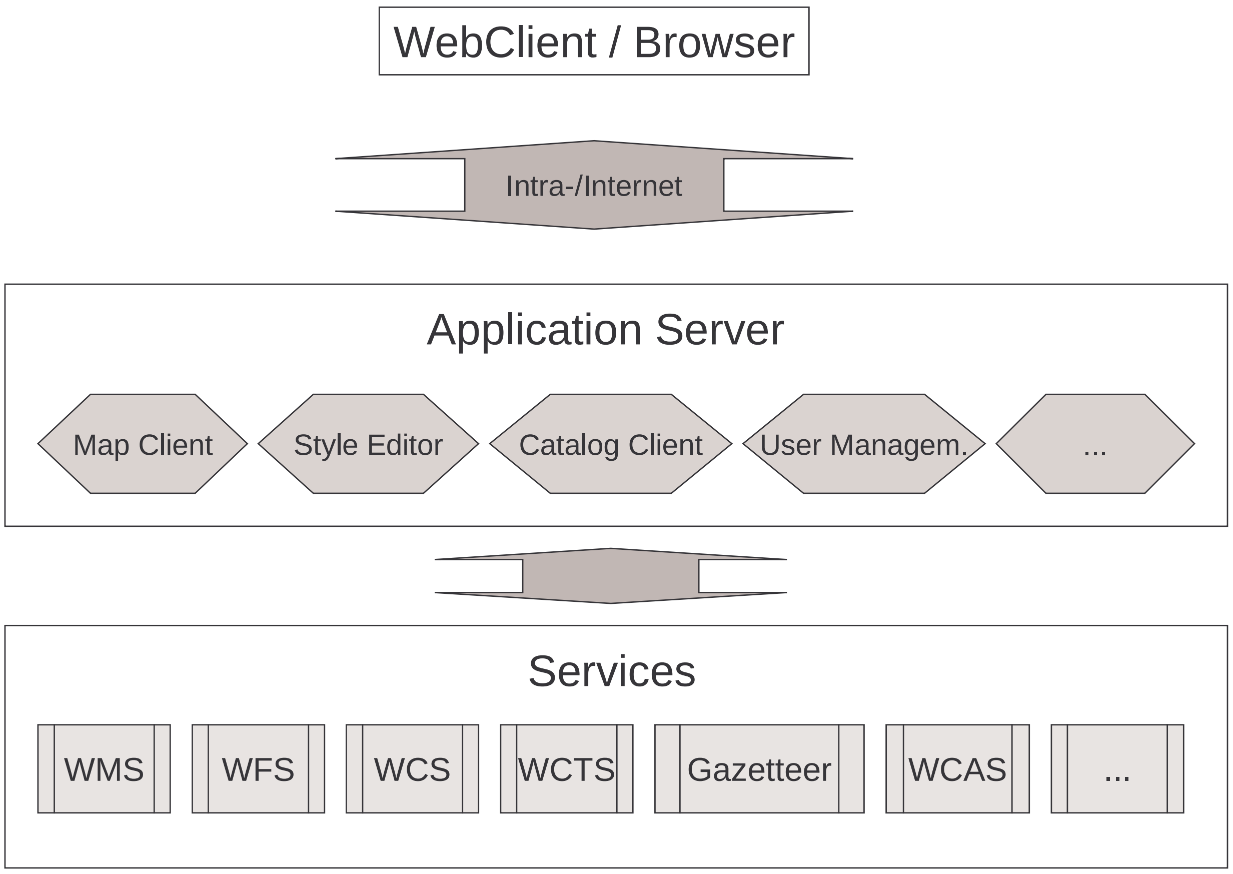
Style Editor / deegree Web Application Client Framework

The basic idea is that we have an application server in front of OGC Web Services (OWS). The application server is though to be a collection of more or less indepent moduls that interact with the web frontend (web browser) on the one side and are connected to the OWS on the other. Because the moduls should be applicable on their own on the one hand it will be possible to develop the different moduls independly and on the other it will be possible to create web applications with different capabilities just by plug-in different moduls.

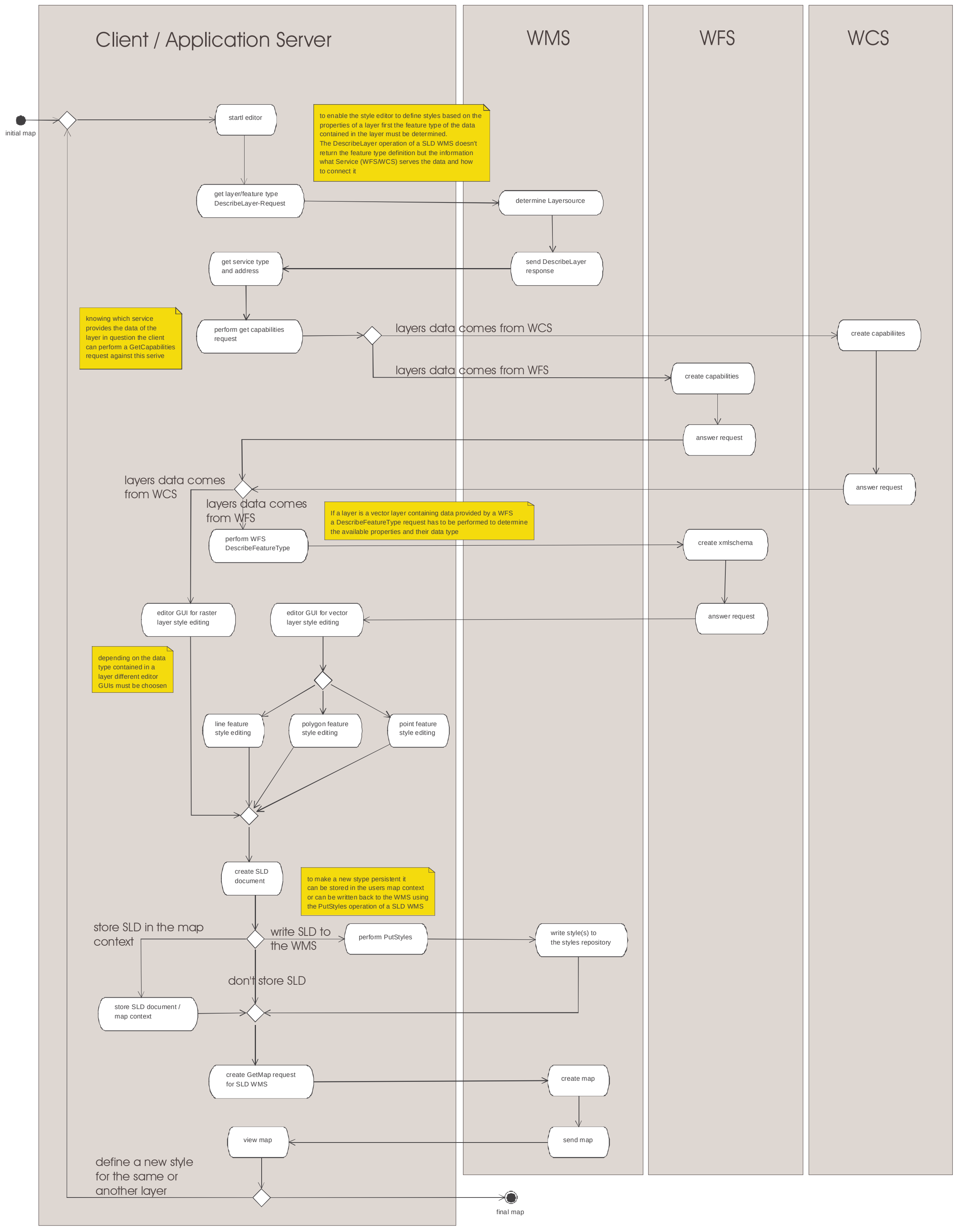


Each modul should/can be implemented on basis on the deegree web client application framework. The responsible classes are located in org.deegree.enterprise.control (interfaces) and org.deegree_impl.enterprise.control (classes). The central idea behind the framework is that each modul uses a servlet as central access point for web requests. Each servlet acts as proxy to the functions that a modul offers (proxy pattern). The functions of a modul are implemented as Listeners that are called by the proxy/dispatcher. Each web request is assigned to a defined event. Depending on the event the responsible listener will be called (observer pattern).

The idea is similar to that of the application server itself. Functions can added to a modul by registering new listeners to it. Because the listeners, known by a modul, and the events they are assigned to are defined within a configuration file, they can be set/added without changing the source code of a modul (will be explaind in more details below).

Comming to the style editor in detail. In our opinion the greatest problem for implementing a web based style editor is not the editing and creation of a style itself but accessing/getting the informations that are required for it. Suggest you like to create a style that colors the countries of the world depending on the population density. The question is what property of the feature collection behind the layer of the map stores the population density. This information is required to create the filter expression for the SLD. Per definition the map produced by a WMS is an image that doesn’t contain any accessable informations about the data behind it. So the map client application first has to determine what properties the data (feature collection) behind a layer offers.

This is a bit difficult because even the WMS as producer of the map doesn't know it directly. So the client application at first has to ask the WMS what OWS is behind of a layer. This can be done using the SLD WMS DescribeLayer request. Assuming the data behind a layer are vector data the result returns the address of the WFS that delivers the data (or the address of a WCS if it's a rasterdata layer). Using this information the client application can perform a get DescribeFeatureType request against the responsible WFS. The returned XML-schema definition of the feature type contains the required informations. (Even if a style editor is implemented as stand alone application it has to be determined what properties of what data type are availabe for a layer, but it will be not so complicated.)


Client / Application Server
WMS
WFS
WCS
initial map
startl editor
get layer/feature type DescribeLayer-Request
determine Layersource
send DescribeLayer response
get service type and address
perform get capabilities request
layers data comes from WCS
layers data comes from WFS
create capabiliites
create capabilities
answer request
answer request
layers data comes from WCS
layers data comes from WFS
perform WFS DescribeFeatureType
create xmlschema
answer request
editor GUI for raster layer style editing
editor GUI for vector layer style editing
line feature style editing
polygon feature style editing
point feature style editing
create SLD document
store SLD in the map context
write SLD to the WMS
perform PutStyles
write style(s) to the styles repository
don't store SLD
store SLD document / map context
create GetMap request for SLD WMS
create map
send map
view map
define a new style for the same or another layer
final map

to enable the style editor to define styles based on the properties of a layer first the feature type of the data contained in the layer must be determined. The DescribeLayer operation of a SLD WMS doesn't return the feature type definition but the information what Service (WFS/WCS) serves the data and how to connect it

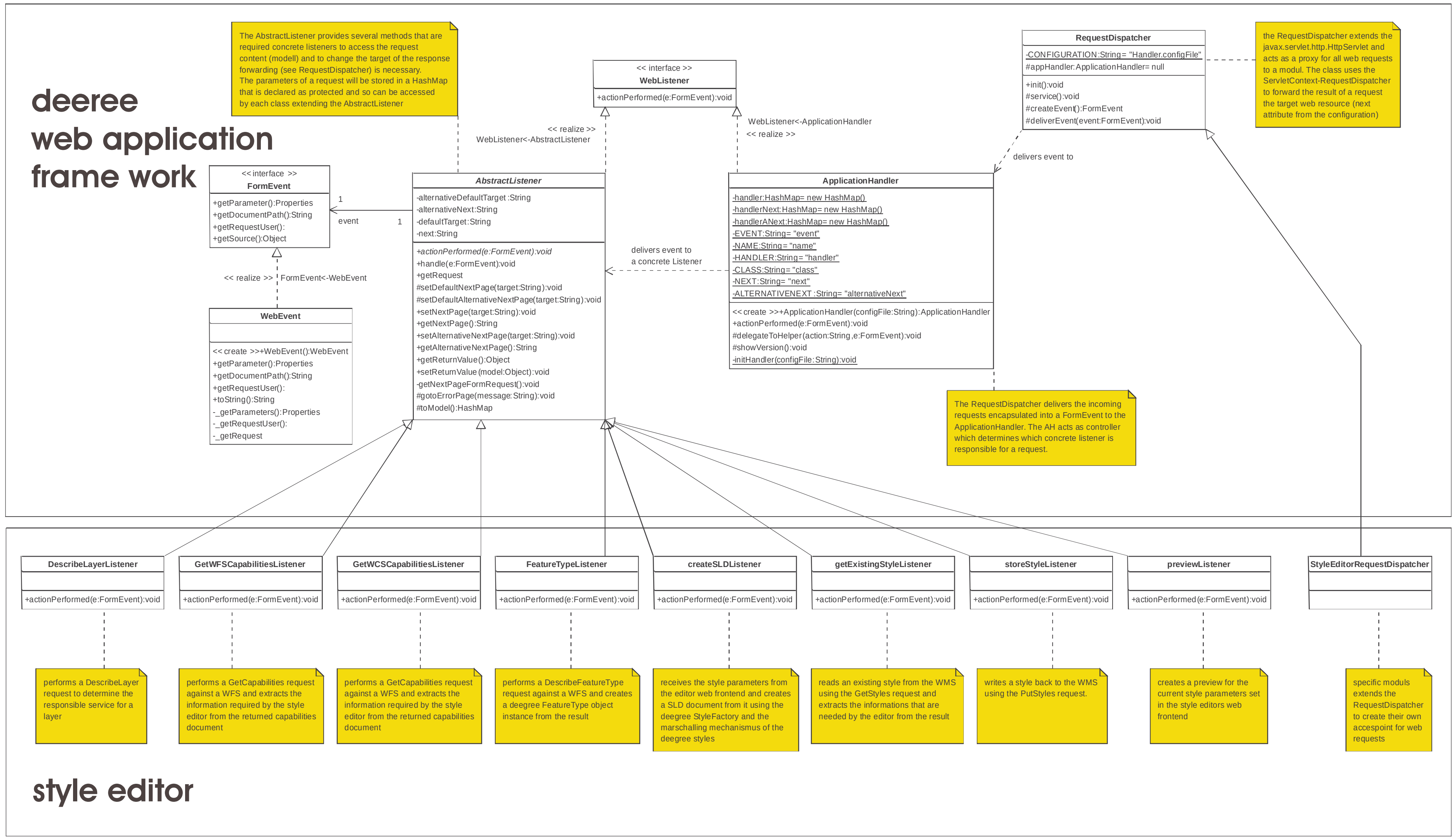
knowing which service provides the data of the layer in question the client can perform a GetCapabilities request against this serive

If a layer is a vector layer containing data provided by a WFS a DescribeFeatureType request has to be performed to determine the available properties and their data type

depending on the data type contained in a layer different editor GUIs must be choosen

to make a new stype persistent it can be stored in the users map context or can be written back to the WMS using the PutStyles operation of a SLD WMS

So the functions that a style editor needs not only targets the creation of styles (SLD) itself but also accessing the required feature type informations and, if required, the storage of the created styles. As mentioned above a web based style editor may be implemented basedon the deegree web application framework with a listener for each desured function:


deeree
web application
frame work
<< interface >>
FormEvent
+getParameter():Properties
+getDocumentPath():String
+getRequestUser():
+getSource():Object
1
event
1
<< realize >>
FormEvent<-WebEvent
WebEvent
<< create >>+WebEvent():WebEvent
+getParameter():Properties
+getDocumentPath():String
+getRequestUser():
+toString():String
-_getParameters():Properties
-_getRequestUser():
-_getRequest
<< interface >>
WebListener
+actionPerformed(e:FormEvent):void
<< realize >>
WebListener<-AbstractListener
WebListener<-ApplicationHandler
<< realize >>
AbstractListener
-alternativeDefaultTarget :String
-alternativeNext:String
-defaultTarget:String
-next:String
+actionPerformed(e:FormEvent):void
+handle(e:FormEvent):void
+getRequest
#setDefaultNextPage(target:String):void
#setDefaultAlternativeNextPage(target:String):void
+setNextPage(target:String):void
+getNextPage():String
+setAlternativeNextPage(target:String):void
+getAlternativeNextPage():String
+getReturnValue():Object
+setReturnValue(model:Object):void
-getNextPageFormRequest():void
#gotoErrorPage(message:String):void
#toModel():HashMap
delivers event to
a concrete Listener
ApplicationHandler
-handler:HashMap= new HashMap()
-handlerNext:HashMap= new HashMap()
-handlerANext:HashMap= new HashMap()
-EVENT:String= "event"
-NAME:String= "name"
-HANDLER:String= "handler"
-CLASS:String= "class"
-NEXT:String= "next"
-ALTERNATIVENEXT :String= "alternativeNext"
<< create >>+ApplicationHandler(configFile:String):ApplicationHandler
+actionPerformed(e:FormEvent):void
#delegateToHelper(action:String,e:FormEvent):void
#showVersion():void
-initHandler(configFile:String):void
delivers event to
RequestDispatcher
-CONFIGURATION:String= "Handler.configFile"
#appHandler:ApplicationHandler= null
+init():void
#service():void
#createEvent():FormEvent
#deliverEvent(event:FormEvent):void
DescribeLayerListener
+actionPerformed(e:FormEvent):void
GetWFSCapabilitiesListener
+actionPerformed(e:FormEvent):void
GetWCSCapabilitiesListener
+actionPerformed(e:FormEvent):void
FeatureTypeListener
+actionPerformed(e:FormEvent):void
createSLDListener
+actionPerformed(e:FormEvent):void
getExistingStyleListener
+actionPerformed(e:FormEvent):void
storeStyleListener
+actionPerformed(e:FormEvent):void
previewListener
+actionPerformed(e:FormEvent):void
StyleEditorRequestDispatcher
style editor

The AbstractListener provides several methods that are required concrete listeners to access the request content (modell) and to change the target of the response forwarding (see RequestDispatcher) is necessary.
The parameters of a request will be stored in a HashMap that is declared as protected and so can be accessed by each class extending the AbstractListener

the RequestDispatcher extends the javax.servlet.http.HttpServlet and acts as a proxy for all web requests to a modul. The class uses the ServletContext-RequestDispatcher to forward the result of a request the target web resource (next attribute from the configuration)

The RequestDispatcher delivers the incoming requests encapsulated into a FormEvent to the ApplicationHandler. The AH acts as controller which determines which concrete listener is responsible for a request.

performs a DescribeLayer request to determine the responsible service for a layer

performs a GetCapabilities request against a WFS and extracts the information required by the style editor from the returned capabilities document

performs a GetCapabilities request against a WFS and extracts the information required by the style editor from the returned capabilities document

performs a DescribeFeatureType request against a WFS and creates a deegree FeatureType object instance from the result

receives the style parameters from the editor web frontend and creates a SLD document from it using the deegree StyleFactory and the marschalling mechanismus of the deegree styles

reads an existing style from the WMS using the GetStyles request and extracts the informations that are needed by the editor from the result

writes a style back to the WMS using the PutStyles request.

creates a preview for the current style parameters set in the style editors web frontend

specific moduls extends the RequestDispatcher to create their own accespoint for web requests

Within the project we are doing we had implemented several moduls based on the this architecture. So we now can say it is flexible, stable and robust. The first modul we published in the deegree framework was a basic web map client which demostrates how to create modules and listeners and how to configure them. For usual we use two configuration files for each module. The first one (for usual named controller.xml) defines the association between web events and listeners:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<controller>
    <event name="init" class="org.deegree_impl.clients.wmsclient.control.InitListener"
next="map.jsp"/>
    <event name="REFRESH" class="org.deegree_impl.clients.wmsclient.control.RefreshListener"
next="map.jsp"/>
    <event name="ZOOMIN" class="org.deegree_impl.clients.wmsclient.control.ZoomInListener"
next="map.jsp"/>
    <event name="ZOOMOUT" class="org.deegree_impl.clients.wmsclient.control.ZoomOutListener"
next="map.jsp"/>
    <event name="RECENTER" class="org.deegree_impl.clients.wmsclient.control.RecenterListener"
next="map.jsp"/>
    <event name="RESET" class="org.deegree_impl.clients.wmsclient.control.ResetListener"
next="map.jsp"/>
    <event name="PAN" class="org.deegree_impl.clients.wmsclient.control.PanListener"
next="map.jsp"/>
    <event name="INFO" class="org.deegree_impl.clients.wmsclient.control.InfoListener"
next="map.jsp"/>
    <event name="SELECTSTYLE"
class="org.deegree_impl.clients.wmsclient.control.SelectStyleListener" next="selectStyle.jsp"/>
</controller>
```

It always has the same structure. Web event are send to the HTTP-Get by setting a parameter name 'action'.

http://localhost:8080/client/control?action=init&param1=XXX&param2=YYY …

The second configuration file has its own structure for each module and contains all informations that are required; see deegree web map client configuration.xml for example.