A Design Concept for Implementing Interoperable Cartographic Services based on reusable GIS Components

Hugentobler, M, Iosifescu-Enescu, I., Hurni, L., Institute of Cartography, ETH Zurich
{hugentobler, iosifescu, hurni}@karto.baug.ethz.ch

Abstract
The design concept introduced in this paper provides an efficient and flexible pattern to create Web map services from existing GIS backends. The components of the concept are described and discussed. The collaboration and interactions between the components are explained and illustrated with class and sequence diagrams. Finally, the application of this concept in the QGIS mapserver project is shown.

1. Introduction

In information technology, a trend from monolithic applications towards service based architecture can be observed. In service oriented architecture (SOA), software is organized into independent and distributed service module communicating over a network (e.g. the internet) [Papazoglou 2003]. Popular transport protocols for Web services include HTTP GET/POST and SOAP. The trend towards distributed and service oriented architectures is also reflected in the geographic and cartographic domain, where several OGC standards allow interoperability between different map servers and map clients. Important standards for Web mapping include Web Map Service (WMS) [OGC 2004], Styled Layer Descriptor (SLD) [OGC 2002] and Filter Encoding Specification [OGC 2005a].

The IST - 6th Framework Integrated Project ORCHESTRA (Open Architecture and Spatial Data Infrastructure for Risk Management) tries to improve the efficiency in dealing with risks by developing an open service-oriented architecture for risk management, based on de-facto and de-jure standards. Special attention is paid to an integrated service and data approach including both their spatial, temporal and thematic characteristics. The Institute of Cartography is one of 14 institutions involved in the development of ORCHESTRA and mainly responsible for the service description and implementation of the mapping service.
The specifications of the ORCHESTRA Map and Diagram Service are implemented under the GPL license in the QGIS mapserver project. QGIS mapserver is a contribution of ORCHESTRA to the open source community [QGIS mapserver 2007].

The requirement to be distributed and accessible as software component over the internet is thus crucial for many existing mapping software. This paper presents an extensible design concept and a methodology for an efficient implementation of interoperable cartographic enriched interfaces. It can be used both for creating new OGC compliant cartographic Web services and for turning existing desktop mapping applications into mapservers. It is also well suited to understand the design of the new QGIS mapserver software [QGIS mapserver 2007].

Section 2 is about motivation and applicability of the concept. Section 3 describes the fundamentals and the consequences of the design concept, both in an abstract manner and with focuses on OGC service standards. Section 4 finally shows the implementation of the concept in the opensource project ‘QGIS mapserver’.

2. Motivation and applicability

Many desktop mapping software has been designed to be used locally on one computer and to get input from and draw output to screen. However, with the trend towards distributed mapping systems described in the section 1, these desktop mapping systems are more and more faced to concurrency from distributed software systems.

This paper shows that in such cases, there is no point to drop the existing desktop mapping software and replace it with a mapserver. Rather, the desktop software should be enhanced such that it is accessible as a service over the internet, thus becoming a mapserver itself. The desktop software should be changed in a way that it is able to create maps according to web service requests, draw the maps into off-screen buffers instead on screen and send the maps back to the clients byte by byte. It is an advantage for the software if it is compatible with the OGC web services such that it is interoperable with existing clients. Thus, you may want to apply the design concept proposed in this paper if you have a desktop mapping application and want to make it a service component in a distributed system. Already existing mapserver applications may still use selected parts from the design concept to enhance design flexibility and extensibility.

The proposed design concept has several interesting characteristics. One of them is the property of being flexible to changes in the transport protocol. This is important for OGC
compliant map servers because there are discussions to also have support for SOAP messages [OGC 2005b] besides HTTP/GET and HTTP/POST, which are already established for OGC Web services. Also, the design concept allows for map styles which can be predefined on the server or sent with the client request, e.g. described by the OGC standard styled layer descriptor (SLD) [OGC2002].

3. Design concept

3.1 Participants

Fig. 1. Design concept - participants

3.1.1 Webserver
An incoming request starts at the webserver (e.g. Apache). The Web server receives a request from a Web client, e.g. a Web browser, a client side script-based application or a desktop GIS like QGIS, uDig or JUMP. The webserver then calls the real mapserver application, whereas the calling mechanism is dependant on the used server technology and not subject of this paper.

3.1.2 Server main
This is the entry point into the mapserver software, usually called from the webserver for each request. Depending on the used technology, this may be the main method of the server executable (e.g. CGI), a loop (e.g. FastCGI) or a callback function in the mapserver.
3.1.3 Request handler (e.g. GetRequestHandler, PostRequestHandler, SOAPRequestHandler)

This interface hides the details of the individual transport protocols from the other parts of the server application. Examples of subclasses would be handlers for HTTP GET, HTTP POST or SOAP. The request handler has a method `parseInput()` which extracts the parameters out of the request specific format into a keyword/value map containing all the request parameters. The request handler also has methods to create responses with the specific transport protocol. There is a method for each supported operation. E.g. for WMS, there would be methods to send the output of GetCapabilities, GetMap, GetFeatureInfo or to send a service exception. The advantage of the request handler interface is that it is straightforward to add support for new service protocols. E.g. an OGC compliant mapserver just needs a new request handler subclass to have support for SOAP Webservices.

3.1.4 Server class (WMS server)

The server class receives the parameter map from the request handler and is responsible for the request independent execution of the server methods. For a server implementing the methods of WMS, this means that the server class has the methods `GetCapabilities()`, `GetMap()` and optionally `GetFeatureInfo()`. In these methods the creation of objects specific to the mapping backend is initiated according to the request parameters, e.g. map layers, symbologies, coordinate systems. More specific, those objects are created by the configuration parser (see section 3.1.5). The server class arranges the objects and passes them to the mapping backend.

3.1.5 Configuration parser (SLD parser)

This class analyses configuration data containing information about available layers, data sources and styles. The configuration parser is able to create objects specific to the mapping backend. Configuration data may come to a mapserver either as administrator data located on the server (e.g. mapfile in UMN mapserver) or as user data in the request (SLD). Thus, mapserver software usually needs two classes to parse the configuration. The approach of this design concept is to use the same configuration language for administrator configuration and for user defined configuration. For WMS compliant services, this means that both hosted administrator configuration and sent configuration are in SLD. Using this approach, it is only necessary to have one class for configuration parsing. This
considerably reduces the number of code lines in a mapserver project and enhances maintainability of code.

Sent configuration data may refer to administrator configuration data. E.g. symbologies may be sent for layers defined in the administrator configuration data. Therefore configuration parsers may be linked and a member of the configuration parser class is a pointer to another parser object (fallback parser, may also be 0). So if one parser representing user defined configuration does not have enough information to successfully perform a method, it tries to get this information from the second parser object which represents the hosted administrator configuration.

3.1.6 Mapping backend

The mapping backend does the real work of reading data and rendering a symbolized map to an image format. Most commonly, it supports the concepts of raster and vector layers and coordinate systems and has a specific representation of vector and raster symbologies. Ideally, it is able to render the requested map into an off-screen buffer such that the mapserver may send back the content of this buffer to the client. Popular examples of such backends are the open source projects GeoTools (Java) [GeoTools 2007] or the core library of QGIS (C++) [QGIS 2007].

Some existing desktop mapping systems may just draw the generated maps on screen and not offering the functionality of rendering into an off-screen buffer. In these cases, the core of the software needs to be adapted. This adaptation is usually not very complex as most GUI libraries have an abstract base class for a paint device such that painting on screen and in memory can be done with the same method calls [Sun 2007, Trolltech2007].

An interesting question is how it would be possible to extend this design concept by the possibility to easily exchange the mapping backend, e.g. to make performance comparisons or maybe even to decide at runtime which backend to use. Code specific to the backend is used in the server class and in the symbology parser. The main method and the request handlers don't know anything about the specifics of the backends used. Therefore, introducing an abstract interface for the server class would be enough to exchange mapping backends in a flexible way. Each server class would then need a symbology parser specific to the mapping backend used. These specific parsers could be derived from an SLD parser class which contains the parts that are only related to SLD and not to a specific backend.
3.2 Collaboration
The collaboration and interactions between the server components are presented for the two WMS requests GetCapabilities and GetMap.

3.2.1 GetCapabilities
Figure 2 shows the sequence diagram of a GetCapabilities request. The main method of the server executable creates an appropriate request handler object and calls its method parseInput(). The return value of this function is a parameter / value map where the main method can see that the request type is GetCapabilities. It creates a WMS server object, passes the parameter map to it and calls the method 'GetCapabilities'. The WMS server object creates the general parts of the capabilities document and creates an SLD parser object for the parts of the capabilities document that are layer specific (e.g. layer names, available styles, bounding boxes of the layers). The WMS server object assembles the general and layer specific parts and returns the whole document to the main method. As a last step, the request handler sends back the document using the corresponding transport protocol.

![Sequence diagram for a WMS GetCapabilities request]

Figure 1: Sequence diagram for a WMS GetCapabilities request
3.2.2 GetMap

Figure 3 shows the sequence diagram of a getMap() request with user defined configuration sent in SLD format. The main method of the server executable decides, which concrete subclass of request handler is used to parse the incoming request. This class then parses the request parameters and stores them in a map. This map is passed to the constructor of the WMS server class. The server class looks for the configuration file hosted on the server and creates a first instance of the SLD parser class. Then, in the method getMap() of the WMS server class, the parameters are examined and it turns out, that SLD symbolisation is used in this request. Therefore, the WMS server creates a second SLD parser for the user configuration. The parser for the hosted data is now the fallback object. The example in figure 3 shows how one parser forwards the request for a map layer to the second parser object. The WMS server then collects the map layers together with the other settings and passes them to the mapping backend. It advises the backend to render the map and passes the map image back to the main method. Then, the map image is sent with the request handler using the transport protocol corresponding to the used subclass of request handler.

![Sequence diagram for a WMS GetMap request with user defined map configuration (SLD)](image)

Figure 2: Sequence diagram for a WMS GetMap request with user defined map configuration (SLD)
3.3 Consequences
The proposed design concept has several important consequences. The usage of the request handler interface makes the mapserver flexible with respect to changes in the transport protocol. The main executable may even do tests at runtime to guess the protocol of the request and use an appropriate subclass of the request handler. This flexibility is important for OGC compliant servers as the requirement of supporting SOAP additionally to HTTP GET and POST is more and more important.

A further advantage is the property to have only one class for configuration parsing. Such parser classes may get quite large and the possibility to save an implementation means a significant save of development and bug fix time.

Extensions to the existing standards can be made by adding additional contents to the configuration format. In case of SLD, support for additional tags may be implemented and the server is still able to collaborate with clients supporting standard SLD.

4. Implementation in the QGIS mapserver project
The proposed design concept has been implemented in the opensource project 'QGIS mapserver' [QGIS mapserver 2007]. This mapserver uses the core library of the QGIS project [QGIS 2007] as a mapping backend. QGIS mapserver can be used as a CGI or FastCGI application and is written in C++, ensuring portability of the code by using the Qt library [Trolltech 2007], which is available for most operating systems and computer architectures.

The classes of QGIS mapserver, QgsRequestHandler, QgsWMSServer, QgsSLDParser, correspond to the ones in section 5. It has a request handler subclass for HTTP GET and one for SOAP over HTTP POST.

In order to use an SLD file for server configuration, tags not contained in standard SLD are used, e.g. to describe the location of data hosted on the server.

QGIS mapserver also implements extensions of the SLD standards (e.g. support for the <DiagramSymbolizer>) as specified in the ORCHESTRA Map and Diagram Service Specifications and the documented extensions [Iosifescu et al. 2007]. Currently, bar and pie charts are supported.
5. Conclusions

The proposed design concept allows for an efficient creation of Web map services based on existing mapping libraries. An important property is the request handler interface which hides the details of individual transport protocols from the rest of the server code. Another interesting characteristic is the symbology parser, which is capable of parsing user defined symbology and hosted administrator symbology in a uniform way. As shown in section 3.1.6, the presented concept can easily be extended to support multiple mapping backends.

References: