UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Optimizing VGI production using status maps requested through web services

Francisco Ruiz López

In this MSc. Thesis, sketching, designing and developing a proof-of-concept of the Request-Status Map Mechanism applied to Volunteer Geographic Information is performed using Web Services.

MSc. Thesis directed by:
Vicente Pelechano Ferragud
Rob Lemmens

Máster Universitario en Ingeniería del Software, Métodos Formales y Sistemas de Información

Universitat Politècnica de València
Faculty of Geo-Information Science and Earth Observation of the University of Twente

August 2013

Dedicatoria

Este trabajo está dedicado a toda mi familia, que me ha apoyado siempre en todos los grandes proyectos de mi vida y a la que estoy muy agradecido por ayudarme a ser quien soy. No puedo olvidar tampoco a mis amigos y a la gente que he conocido durante mi estancia en los Países Bajos. Ha sido una experiencia enriquecedora en todos los sentidos.

Académicamente hablando me gustaría agradecer tanto la supervisión, los consejos y también la comprensión recibida de mi director de tesina el sr. Pelechano, mis supervisores en Enschede (Paises Bajos), el sr. Lemmens y el sr. Zurita, y también de los equipos de investigación en los que he estado involucrado, que han proporcionado siempre una excelente crítica constructiva tan necesaria en este tipo de trabajos.

Summary

The present MSc. Thesis is the result of the research performed within the Erasmus program at the University of Twente (Enschede, The Netherlands). This program allowed me to work closely with a high skilled team of researchers and other PhD candidates and Master's students. The team is focused on Geo-Information and it has been a challenge to put together our efforts to reach a common goal.

At the starting point, this research was fully planned as a research focused Master's Thesis. However, due to external factors the Erasmus grant had to be terminated and the plan was forced to change towards a more practical Thesis.
In this MSc. Thesis, sketching, designing and developing a proof-of-concept of the Request-Status Map Mechanism applied to Volunteer Geographic Information is performed using Web Services.

The first step is focused on describing the main characteristics of the Request-Status Map Mechanism, and justifying how it could help in this new "trending topic" of Volunteered Geographic Information. It also described how it could be applied to the ILWIS software that is being developed and supported by the Faculty where the student was cooperating.

Developing a prototype was requested and, therefore, stages for analysis and design of such prototype were needed. In order to develop the prototype properly, research focused on how the mechanism should work as a whole was performed as well (the architecture, what specific services were ideal, structures and protocols to be used, etc.).

The next step, after deciding the proper characteristics to fit the prototype, the development itself was performed and described along with the tests completed to ensure the prototype tool delivered the expected results. The development has been focused in the Status Map side from a particular user perspective (the planning team users). However, in this MSc. Thesis, the other pieces of the mechanism have been described. These other pieces were not developed because the aim of this MSc. Thesis was focused on the proof-of-concept, not a fully functional environment.

The final step includes the conclusions and the most relevant ideas that could be useful for research and development in the future.

Contents

Chapter 1: Introduction and research identification

1 Motivation

In 2010 in Haiti there was a terrible earthquake that caused lots of casualties and catastrophic damage in a poor region. Suddenly, from all parts of the world, volunteers started to make new maps of the island, trying to help by providing basic information to the authorities to coordinate the response and the aid more efficiently. Before that, other natural disasters like Katrina Hurricane and the Indian tsunami put the first stones to use VGI[1] in disaster management (Goodchild 2007) but their contributions weren't so remarkably, the Haiti earthquake mobilized enough volunteers to create a full cartography of a small country in a couple of months (Neis, Singler and Zipf 2010).

This wonderful act of solidarity is a very good example of volunteered geographic information, where each person commits in producing information that could be spatially referenced. This information will be available for the community through some platforms and technologies, usually web-based and the limits in its purpose is just the human imagination.

Nevertheless, those users are individual users, usually without connections between them, so there is no specific way to manage where or which areas are more important to work in, because they are unorganized, and it causes duplicate (and sometimes inconsistent) information and reduces the efficiency.

Along this MSc. Thesis we will develop a mechanism that could be used to improve management of large datasets based on VGI. This can be applied to long-term planning, like monitoring environmental variables or maintenance works in large scale or applied to prevention sector, like coordination of cartographic production after catastrophic events that require fast action and big number of volunteers.

Developing this mechanism, which is based on web services, will require a big documentation process because in this project several disciplines from diverse areas of knowledge (not only computer science) are interconnected for achieving the goal. It is a good example of how all those knowledge, tools and habits acquired along the Master could be applied to other fields with the suitable analysis and research stages.

1.1 Overview of the proposed mechanism

We propose a tool, the request-status map, which is a mechanism that assigns a working area for producing VGI to a user. The volunteer browses the map, which is classified in sectors depending on its priority and then the volunteer will know where has to start creating or updating data. The status map not only considers the *Priority* attribute, it should include other attributes like last update time, activity or users involved, a measure of the accuracy of the spatial data within that area and some more parameters that could be even derived. It would help to increase the potential applications of such maps. Temporal attributes could be used also for versioning. A very basic examples would be obtaining a map of the noise levels and then create a new map of areas where there's need for measurements of higher quality or knowing in real time the evolution of mapping processes in some place, although our case use will be focused in coordinating volunteers for phenological observations.

[1] VGI is the acronym of Volunteered Geographic Information, a particular subset of spatial information produced by users aimed to create or update information voluntarily; a good example is a mapping party coordinated using social networks.

This mechanism will be developed as an add-on for the Geographic Information Client ILWIS, which is a powerful software for managing geographic information, but nowadays is not capable of use the web services that would make this mechanism work, hence is mandatory to implement such functionality in the main client along this MSc. thesis by developing a prototype. This prototype will be undertaken following the natural path of software engineering, using techniques learnt during theoretic and practical lessons.

The mechanism involves a big number of technologies and applications. The main work is focused in the planning client (because it provides new capabilities to ILWIS software) but along the MSc. Thesis we will come up with new specifications or definitions for new tools as result of the necessary research and documentation process.

A status map is a thematic map that shows the actual state of a variable, it could be based on many different variables (i.e. priority, quality, risk or percentage of main elements digitalised). A request map is produced selecting some sectors (of the status map) depending on its status. Such *status* represents, for instance, priority or completeness percentage. Once the right area to work in is selected by the user or by the system, the features and is attributes will be loaded into the GIS client to let the user update them.

Success in dealing with crisis depends strongly in having a good plan to face difficulties, but such plans must be based on updated information. Then, for knowing which locations require more or faster assistance and rationalize the emergency resources the authorities need to have up-to-date information. In crisis time it's hard to control the VGI production on main priority areas, so that's where the request - status map could be the key, because it can be planned in advance.

If the authorities have planned which areas have more priority, the users can access the system and start producing VGI of the most interesting places (i.e. nuclear plants, or facilities with hazardous materials, hospitals, transport network, etc.) to send the aid in the critical zones and assist basic services first to minimize the casualties and the damages. Usually this task of controlling the cartographic and VGI production has been coordinated by expert cartographers or geographers (McDougall 2009), but for immediate action protocols have to be elaborated and be ready to use, that's why this status-map mechanism seems to be the best choice. Because there's no need to elaborate the production plan, it's already done, It might be slightly updated to adapt it better to the epicentre of the catastrophe and its scope, but anyway much faster than develop a plan from the beginning once the catastrophic event took place.

For dealing with spatial information, the users might need some functions to assist the VGI production, that's why we are going to think about the best natural interaction for modifying features (i.e. double click and dragging vertex, snap tools, etc.) and also if those request-status maps would need spatial topology or not for satisfying the requirements.

There are several options of spatial topology in case it will needed, an extensive period of documentation is mandatory to choose the best option because it has big influence in the final result. The topology can be managed in server side or client side, and could be explicit, implicit or rule-based. Each one has their strong and weak points, that's why we have to choose carefully.

Moreover, considering other faces of the situation, the large improvements in networking and communication services have encouraged the use of web services to access remote data and the use of social networks during last the decade. And it seems the tendency will continue the following years. The location of the volunteers forces us to plan a tool able to communicate them no matter what time it is, what language they speak or what location they are. Then there is no doubt in using Internet in this tool as the channel for all communications between elements.

In addition, new tendencies of software development focus on distributed architectures which entail some advantages, mainly efficiency, load management and scalability. ILWIS software is likely to undergo an extensive re-modularization process in order to de-couple the code and makes it more suitable for new challenges to come.

2 Research identification

2.1 Objectives

2.1.1 Main objective

1. Design and evaluate a request status map mechanism for a client – server GIS environment in order to improve the productivity of VGI based on web services. And produce a prototype in C++ for ILWIS.

2.1.2 Sub objectives

1. Devise a use case to identify user requirements for request-status maps and to implement a proof-of concept according to such requirements.
2. Evaluate the use of OGC Standards to exchange information in the request – status map context.
3. Propose viable alternatives in case the chosen ones won't provide remarkable benefits over the rest.

2.2 Research questions

2.2.1 For main objective:

1. What other extra attributes (besides Priority) would add interesting information and make the request – status map mechanism more powerful and versatile?
2. Which type of information is more suitable in this case, polygons or raster datasets?
3. What will be our policy in sectors with heterogeneous accuracy?
4. What advantages could provide the use of spatial topology in the request – status map mechanism?
5. In case spatial topology would be appropriate, which kind of topology shall we use and how could it be implemented (client side or server side, transparent to the user? Etc.) to ensure the best results?

2.2.2 For sub objective 1:

6. Which web service implementation is the most suitable for implementing the OGC standards and testing the prototype? And why?

2.2.3 For sub objective 2:

7. Which OGC standard is more suitable to fit the requirements? (WMS,WFS…. Etc). And why?

2.2.4 For sub objective 3:

8. What would be the interaction with the user, what does the user needs to use it and what use do we expect for the tool?

2.3 Innovation aimed at

The innovation in this project is to make the VGI production more efficient, through request-status maps which are editable and shareable.

2.4 Related work

Almost every powerful GIS client has its own module for connecting to web services, although not all of them allow editing information through them. ArcGIS, gvSIG, GeoMedia and many more are good examples. Other GIS clients, the same than ILWIS, cannot operate with WFS by default, but can get this functionality using plugins, this is the case of QGis (Weichand 2012).

ERDAS Apollo is a powerful tool that allows retrieving and updating information from web services by using OGC standards (Dietz 2010).

Gaia WFS-T Extenders allows operating with WFS-T to retrieve, create and update geometries from WFS, it has been used to create VGI during Haiti earthquake. This might be the most similar tool to the expected ILWIS module.

There are also many tools that allow VGI production through websites. There was a project in Barbados to involve the local community in the development of a sustainable tourism plan (Ricker, Johnson and Sieber 2013) where they've used My Google Maps for adding metadata and extra information (not only geographic information (like routes or interesting points), also media(images or videos of interesting places)).

There was also a similar project called Osm-matrix, specifically designed for Haiti crisis to show the status of the mapping process using regular grids and classification based on completion parameters. This software, developed by Jan Tappenbeck seems to be no longer maintained or supported.

There is also an updated and evolved version of Osm-matrix, its name is OSM Tasking Manager (Wiki, OpenStreetMap 2012) and it has been developed by the Humanitarian OSM Team with the purpose of coordinate more efficiently the edition of the spatial database of OSM. The underlying idea is similar to our proposal, nonetheless our proposal is not focused only in OSM, and it could be used with other VGI sources in addition to be able of managing the priority and link extra attributes.

More related with phenology, the USA National Phenology Network, which is an association that monitors the influence of the climate on animal, vegetal life and landscapes, gathers field information thanks to the volunteers and inputs into a system that can be accessed through its viewer. Although it contains all kind of information about the topic this viewer doesn't provide coordination functionalities.

Chapter 2: Methodology

Along this chapter we will describe the main idea of how the request – status maps work, by analysing the context of the software and the stakeholders in the process. Moreover, we will gather the requirements for obtaining the full functionality and explain how such requirements configure a set of constraints that will answer some research questions and also define some aspects of the software development.

3 Architecture

The architecture of the system is summarized in the following scheme. It contains related blocks that should be considered to get a clear idea of the scope of the proposed mechanism.

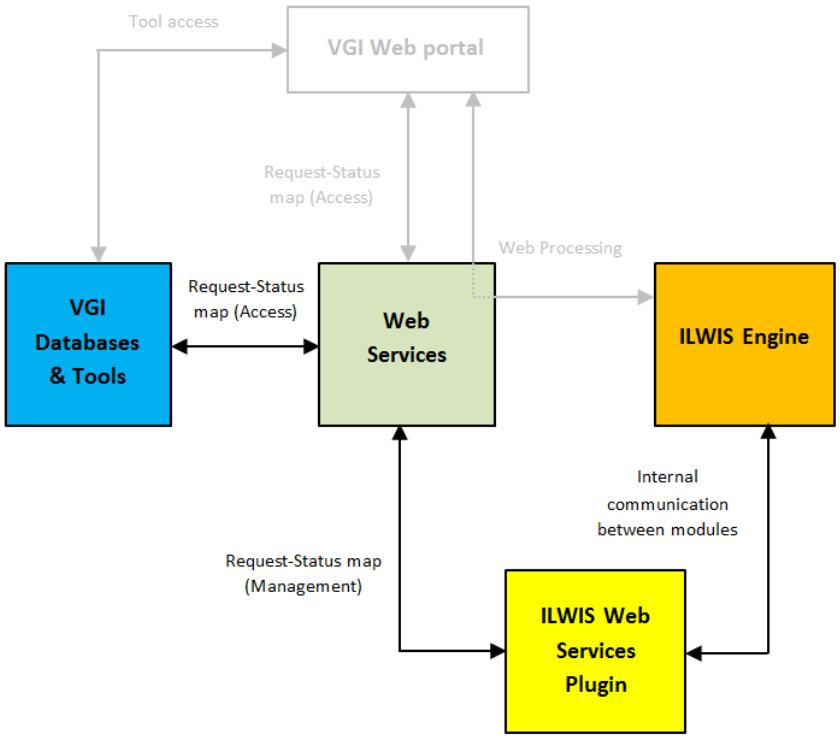


Figure 1. Schema of the main actors in the process of retrieving information through web services applied to VGI using ILWIS software.

This project is related with another MSc. thesis ("Interoperable VGI web applications") done in parallel[2], which has the goal of make request-status maps available for the community by publishing such information within a web portal to facilitate the access to users, connecting to existing VGI sources and tools and provide an infrastructure for using geoprocessing tools for such request – status maps in an easy way. It is expected to connect the ILWIS engine with such web application to achieve geoprocessing capabilities.

Because of the correlation in projects the schema is provided to depict both projects. The coloured items belong to this MSc. thesis proposal and the grey ones to the "Interoperable VGI web applications" thesis. We expect to retrieve information from VGI databases through web services and feed the ILWIS engine.

[2] The project topics are related, but their work set up is not dependent, in order to avoid risks of non-delivery across the two projects.

After that we will use the software to manage the spatial data of the status map and its attributes, finally the spatial database where the status map is stored will be updated. This update process requires the development of a new module or plugin between the engine and the web services capable of managing such request-status maps.

A deeper look into the mechanism produces this other schema. There are many actors involved in the proposed mechanism and they are connected using web services (particularly Web Feature Service defined by the Open Geospatial Consortium). Depending on the context the user can request a map or produce reports to update the status map. The planners update or modify the status maps using a heavy client (as suggestion ILWIS in this MSc. Thesis) by making use of the prototype. And finally some automatic routines could be implemented to avoid planners handle repetitive tasks like recalculating priorities in function of report values.

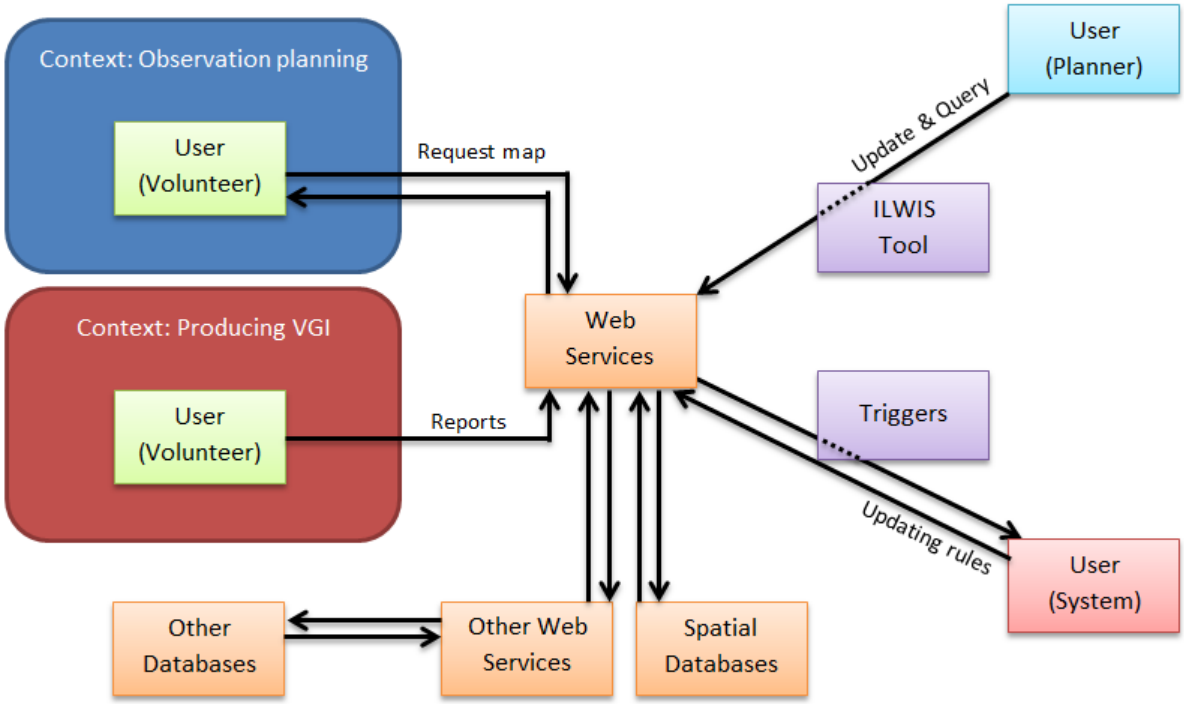


Figure 2. A deeper look into the R-S Map Mechanism, focusing on user profiles and their most common expected requests.

This prototype is focused on ILWIS software, and ILWIS has started a path to update its own architecture to adapt it to the newest paradigms (more web oriented) so a modularization process will be undertaken, with a central engine that will coordinate all the processes and a set of modules (or plugins) that will be loaded when required (i.e. the GUI module or the data module for accessing databases). That's the reason for developing a module instead of adding new code to the actual application.

4 Requirement analysis

The foundations of quality software development are based on good requirements engineering processes, where we have to plan and check if the software does what it's expected, its reliability, efficiency, security, easy-to-use grade, maintenance, flexibility and testing. The following points have been designed to achieve good control of each facet of the proposed tool. They are particularized for the proposed example (phenology) but should be designed generically enough to be used in other applications without too many special changes.

The process followed to define the requirements has been based on iterative meetings from high level requirements to more particular ones along the evolution of the development. There are several kinds of requirements:

- High level requirements:
  - This request-status map mechanism should be able to be used in many contexts.
  - Should be capable of managing status maps based on different departure information.
- Functionality requirements:
  - Should make use of web services.
  - Should be able to update remote spatial databases thorough such web services.
- Implementation requirements:
  - Must be developed under C++.
  - Using Microsoft Visual Studio 2008.
  - Should use OGC standards for exchange information between stakeholders.
  - Should communicate with ILWIS software natively.
- Performance requirements:
  - There are no remarkable performance requirements.
- Usability requirements:
  - There are no remarkable usability requirements.

4.1 Context analysis

According to the Oxford Dictionary, phenology is the study of cyclic and seasonal natural phenomena, especially in relation to climate and plant and animal life. Hence we have to get a general idea of the processes related with these phenological observations, what are they used for and how.

The volunteers have to choose what, where and when are they going to observe, get basic notions of how they have to do it (for example select at least some other species for calibrating their own observations). They can also download datasheets for field work in this site, so they have all they require to start their task.

After getting the field data they login in the platform and input the raw data using the "nature's notebook" tool available at the site, which is the departure point for all data management of the volunteers in the database.

Usually the volunteers select their location of study depending on their location, (i. e. their back yard or parks near their homes), that produces constant information about some places, but for understanding the dynamics of the ecosystems the association needs a full geographical coverage of data, not only on easy-to-access areas. That's why some volunteers ask the coordination office for interesting locations to produce information, and it mainly depends on the specie, the date, the location and the ecosystem.

Hence, the association needs a tool to redirect the volunteers to interesting places based on such variables. The Figure 2 shows schematically the process.

Such information is mixed with other sources in order to analyse it and extract knowledge from this product. Then the biologists can produce patrons of behaviour of the species in their habitats and along different moments in the year.

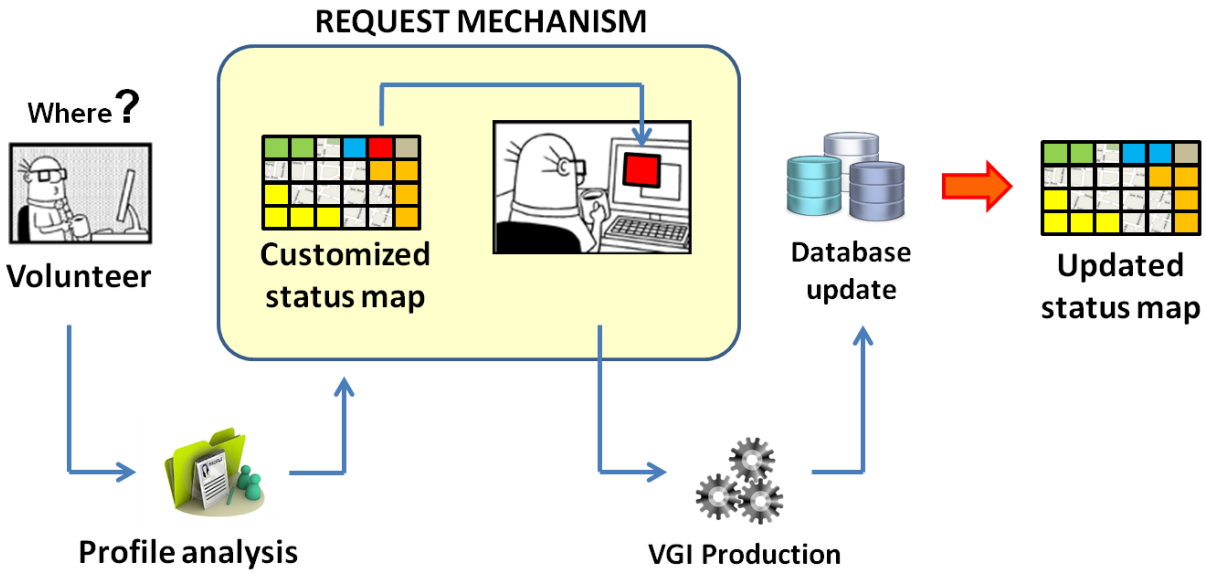


Figure 3. Schema of the request process in VGI production.

4.2 Information analysis

In this step is where we will analyse which of the storage formats, primitives, topologies or technical facets about the data we are going to manage. Because we will know how is it produced and what is needed so we will choose the best formats available to fulfil the requirements.

The status maps show the actual situation of one (or more) parameters based on environmental variables (because we are focusing on the phenology example, if we would be working in the emergency protocols example the input probably would be transport and services networks, urban areas, public services, etc.). The most of these variables would be input to the system using layers of spatial information (i.e. points of observation, habitats, climatic zones, topography, etc.) others require special treatment (i.e. temporal dimension) and these cases are beyond the purpose of this MSc. Thesis because is more related to the status map elaboration methodology itself and the request – status mechanism is non-dependant on the way the status map is created.

The spatial information is managed using layers, which are entities that collect features (usually) from the same family. Depending on the technology used such layers are constrained (or not) to store only one spatial primitive (points (0 dimensions), segments (1 dimension) or polygons (2 dimensions), sometimes even *n*-dimensional features are allowed). Although the usual is separate such features depending on the spatial primitive.

In professional environments like SDI[3] the most usual way to store these layers are databases with spatial extensions, like PostGIS[4] or Oracle Spatial[5], and such layers are tables in the databases. In these tables the geometry columns are interpreted as the spatial description of the feature and the other columns are the associated attributes in case of vector elements, for raster the internal storage depends on the database engine.

[3] SDI is the acronym of Spatial Data Infrastructure. It is a set of tools and technologies that connect spatial information and users through networks to facilitate the use of such information. It could be used d to facilitate the availability and access to spatial data and can be used for decision making and planning (Aalders and Moellering 2001).
[4] PostGIS is the spatial extension for PostgreSQL for supporting geographic objects. It is one of the most advanced open-source databases available nowadays (Research 2012).
[5] Oracle Spatial and Graph is an optional component of Oracle Databases to support geographic data (Oracle 2012).

Such aspects are very important when we are designing the status map because the methodology of its creation depends on the nature of the departure layers. Then we will enumerate the most influencing layers that will be the starting point for configuring our first status map.

- Reports (Point layer)
- Hydrology (Line and Polygon layers)
- Cities and population (Point layer)
- Heights (Raster)
- Counties
- Climatic zones
- Etc.

The power of GIS environments remains in the link between spatial and alphanumerical information in databases. This link allows the planners to store and manage some extra information that could be interesting along the process.

There are two kinds of attributes in status maps, the common ones for all the status maps and the specific ones that contains information about particular facets that are or could be useful in this context but not in a different one.

4.2.1 Common attributes

Attributes related to particular aspects of the areas in the status map.

- Priority based on particular classification schema (cardinality 0 … *) The classification schema should be mentioned and described in the metadata
  - Calculated basing on some criteria or updated manually ()
- Special priority
  - According to some criteria (There can be more than one status for the same feature depending on different classifications) the criteria should be mentioned in the metadata. (Modifies the priority field according to planner's particular criteria)
- Status (Active / Inactive)
- Date of creation (of the feature)
- Creation user
- Number of editions
- Date of last editing (of the feature)
- Last editing user
- …
- Geometry
- Area
- Perimeter

Priority and status can use range domains or tagged values domains.

4.2.2 Specific attributes

Could be input directly or calculated using spatial analysis, (i.e. intersections with ecosystems), and are specific for the purpose of the status map, in our phenological example:

- Ecosystem
- Climatic zone
- Number animal of observations

- Number of plant observations
- Statistics
- Etc.

4.2.3 Data structures

The status map is, in the end, a map of areas which considers several attributes. Then the first task is determine which of the possible architectures is the best depending on their properties, raster or vector, some of them are extracted from (Burrough, et al. 1998).

The main properties of raster are:

- Good capability for continuous variable analysis in the space.
- Easy to compute mathematical operations with raster structures.
- Easy and fast for managing regular grids.
- Big cell size implies low space storage.
- Use of pyramidal structures for increasing the performance.
- Problems of value definition in the boundaries of the cells, sometimes require complex interpolation functions.

The main properties of vector are:

- Vectors allow defining non-regular boundaries.
- For big scales vector layers usually use less storage space in comparison with raster.
- Could use generalization techniques for increase the performance at multi-scale view.
- This structure allows an efficient use of topology.
- The storage for extra attributes is simple and can contain different basic types in the same feature.

For managing the status map there is no big difference between vector and raster, in theory we could use both, besides there are OGC standards available for editing both through web services that have transactional support, WFS for vector and WCS for raster.

Although both have their service for managing the information the feature model of WFS is less complex than WCS. WCS is very flexible and allows using extensions to increase the capabilities of the service, which is powerful, but turns the model in a more complex structure that could require too much building effort for obtaining such capabilities and is less interoperable. Just for the same functionality.

The WFS model contains a set of operations in an easier model, considering the spatial geometries as subclass of "AbstractFeature" class.

In addition for managing the extra attributes associated with the spatial information raster and vector use very different strategies.

A polygonal feature stores the attributes as values in columns, so one polygon can store many attributes and still being managed as a unique entity. In raster structures is not possible to store more than one attribute per band, then, if we need more attributes we need to add new bands to the coverage. It makes more difficult the management, because constraints the formats (not all raster allows multiband managing) makes complex the model and difficult the updates. And also makes difficult mixing attributes of different basic types (like bands of dates, strings and integers), that constraints the type of possible attributes to use, which is a serious handicap.

Another reason that could be important is the source data format. Originally the departure information managed for the planners of the USA-NPN is in vector format (i.e. observations as points and ecosystems as polygons), so each time that the status map (in raster) should be updated we probably should pre-process the departure layers to adapt it (rasterization or vectorization) and be able to perform operations (because most of times is not possible mix both formats in analysis).

We expect certain frequency of update, probably this update would be triggered by some rules related with the VGI production statistics, so each time that the status map requires updates or the departure information or the status map should be rasterized and then perform the spatial operations (intersections, identities, etc.). We expect to have full coverage of the USA-NPN network, so this step, even it's solved since time ago and is a reliable and mature technique (if the parameters that determines the precision of the rasterization are accurate), entails large amount of space and computation, being time-consuming. Then it is not practical to work with different formats when many spatial operations are expected.

We could use raster status maps because it would be possible to perform all the required operations to configure status map using raster datasets, even if the source data is vector (it could be rasterized) the only difference would be the contours of the status map, that couldn't have been irregular, should have been regular grids. The other point which need special attention would have been the size of the raster cells, to be small enough for configure realistic work areas and to be big enough for not to consume extreme storage space. Moreover implementing topology in this case would be difficult.

In the end both structures are capable of satisfying the requirements (technologies, remote edition and update, topology, attribute management, etc.) nonetheless seems that using WFS, although require extra effort for managing the topology, fits better with the extra attribute management, as new columns on the database better than extra bands in raster structures.

Vector	Raster
Requires basic topology for maintaining complete coverage	It is implicit in its structure
The departure datasets are in this format	Requires conversion to perform spatial analysis with departure datasets
Polygons define better discrete areas	Good structure for continuous variables in the space
Polygons store extra attributes easily and could manage diverse types of data as unique feature	Raster must use diverse bands (or sets of rasters) to store extra attribute data. Big difficulty to mix different basic types (strings, dates, integers…)
Good definition of shapes in boundaries	To get good definition the raster size should be smaller, increasing the space and the number of cells
One polygon define one area	If the area is non-regular might be necessary use more than one raster cell to materialize the area. Duplicating information and making more complex the model
When the amount of polygons or vertex is large the management could require big computational effort	Low computational costs because the structure is efficiently managed.

Table 1. Summary table (focused on status maps and areal geometries).

Analysing the previous points and the table, we can conclude that although both are viable options for managing geographical information using web services for this particular purpose (coordination of volunteers in phenology observations) fits better defining the status map using polygons instead of raster cells, mainly because of the complexity using several attributes and the maturity of the feature technology

against the coverage services. As a consequence we are going to develop the tool prototype using the Web Feature Service instead of Web Coverage Service.

4.2.4 Topology

The status map should fulfil some requirements. Due to its definition there is no sense in containing hollow areas. In case one area is not computed or the status is no relevant an attribute should store that information, but having areas with no coverage is not an option. This aspect is solved in raster if we use formats that don not allow hollow areas, and there's no need to implement any topology because of the intrinsic structure of raster cells. But in vector formats we have to care about it. This requirement turns in a must considering topology in this mechanism, but there are many options for managing topological relationships, hence, which one to choose?

Persistent topologies (Mainly there are three options available):

- Implicit topology
- Explicit topology
- Rule-based topology

The implicit topology controls the relationships between features of one or many layers while editing, it allows the shared editing for common elements and such elements will keep their relationship (i.e. shared edges, spatial coincidence, etc.), the basis of this kind of topology is the cluster tolerance, which could be interpreted as an internal grid to snap the coordinates of the vertices of the geometries.

The explicit topology generates tables in the hard disk to store explicitly the relationships, mainly nodes and arc tables, generating polygon tables as association of arcs, and generating such arcs as a result of associations between nodes. Such topology is very powerful but requires lot of maintenance processes of cleaning and updating each time the geometries are updated in order to maintain the consistency.

The rule based topologies are, as their name says, a rule or set of rules of topological relationships that one or more layers should maintain. Using this topologies do not guarantee having a complete correct and topologically consistent dataset, however sometimes trying to filter, clean and build a complete explicit topology would be impossible to undertake, therefore with this kind of topology it's possible to ensure just the topological relationships that will be needed.

Use topology *on-the-fly*:

This is a process where a temporal topology is created just for processing the editing operations and then, once updated the database, the topology is discarded.

Implementing mechanisms for guarantee the required topological relationships is, for the moment, beyond the scope of this MSc. thesis, nevertheless would be interesting to point out the possible strategies to implement (in a nearby future) the topology control in the prototype.

As a consequence of using web services to access the data remotely by definition the use of explicit topology is not possible because the storage management in the service database is already prepared and beyond the possibilities of the user. It lets us the two remaining options, implicit topology and topology *on-the-fly*.

4.2.4.1 Implicit topology alternative

This first strategy is based on the main architecture of ILWIS, which is written in C++. We can make use of the GEOS[6] library that would allow us to manage topology relationships in ILWIS.

Mainly the user would request the geometries and once all such geometries are in the map window the user starts an editing session. Because of the topology rules that must be guaranteed are known (no gaps, no overlapping, etc.) the client can be preconfigured to check in every edition if such relationships are satisfied or not.

And then, after the edition, if all is ok, executing the WFS transaction will update the status map in the database. This way the status map always pass from one consistent status from another consistent status (topologically speaking) each editing session.

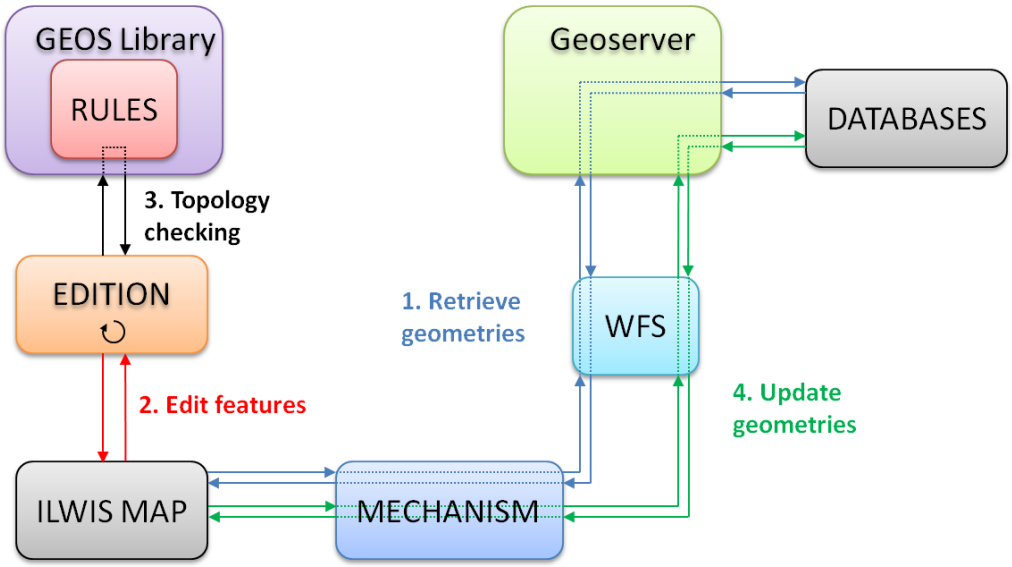


Figure 4. Schema of the proposed topology mechanism using implicit topology.

The main benefit of this alternative is the speed, because the topological analysis of the status map is embedded in the ILWIS client using C++, which is faster than any available option. However this option entails a very rigid system that ensures only the required topological relationships required for the status maps. Besides some of the functions of GEOS that would be used require tolerance values that are sensitive and should be managed carefully.

4.2.4.2 On-the-fly topology alternative

The second alternative is based on make use of services to check the integrity of the topological relationships.

The first steps are the same than the previous alternative, get the features using WFS requests and enable an edition session. After this the user starts editing the features and after all request a topological validation from the server. The request contains all the rules that should be validated and the features to be applied to. Then the server executes the analysis and returns the results to the client, if it's all right then allows the WFS Transaction to update the database, on the contrary shows a message error and guides the user to solve the topological errors.

[6] GEOS is the acronym of Geometry Engine - Open Source. This library is the translation in C++ of the Java Topology Suite (JTS). For more information check the websites of both projects (Foundation 2012), (Solutions 2012).

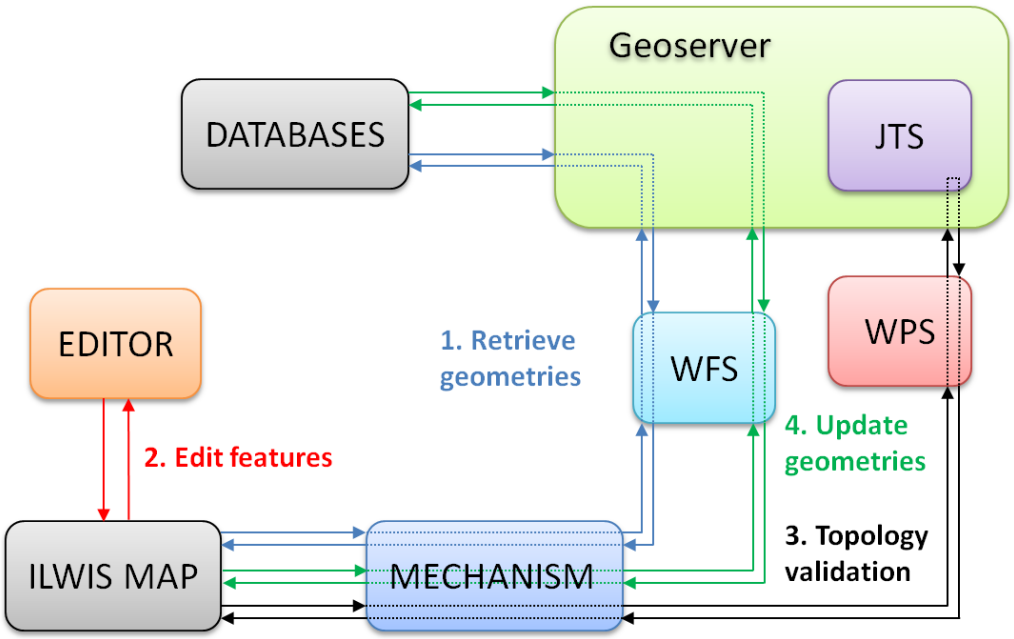


Figure 5. Schema of proposed topology mechanism using Geoserver implemented capabilities.

As a consequence, the alternative depends on the capabilities of the server, some of them allow topological control using WPS[7] (which is still in the first stages of the development and could contain bugs or not be very efficient) but not all of the available servers. Besides, some servers provide its own topological functions in order to complement the JTS functions, so it makes this server-dependant.

Another consequence is that checking the topology remotely could make this process a bit slower than embed the functionality in the GIS client.

For all those reasons seems that the best option in this case would be using the GEOS library and implicit topology, although it would require extra development effort for the ILWIS team. Nonetheless that strategy collides with the perception of web-based conception for the new versions of ILWIS, moreover the improvements in network performance push up the use of WPS for topology checking in this case. Both options are viable, although the conception of ILWIS Next Generation would use the WPS in this case.

After the research stage using topology has turned into a new requirement, nonetheless implement this capability (topology verification) will be delayed to posterior versions of the prototype because of time constraints. The topological control will be, in first stage, just a validation of the basic relationships, no mark the topological errors. The implementation of this topology *on-the-fly* is expected to be done by using the Geoserver Web Processing Service capabilities.

4.1 User analysis

Then we have to analyse the involved stakeholders because there are several kinds of potential users (Hardy 2010). This could include many aspects that might help us to get one or more profiles of users, trying to define it better if it is possible (locations, equipment, background, etc.)

[7] WPS is the acronym of Web Processing Service, one of the OGC standards. It provides rules for standardizing how inputs and outputs (requests and responses) for geospatial processing services (OGC 2012), although this standard is in its early stages yet (version 1.0 at the time of writing this MSc. Thesis).

The main users of the tool will be specialised personnel with big background in environmental sciences and biology. Such users probably will be part of the coordination office of the USA PN. They will have access to all datasets available in the databases and will have permissions to edit and update it. They will be the heads for the elaboration of the first versions of the status maps and will have permission to update manually it whenever they consider necessary. We will refer to them as "Planners" in the future. It's expected that planners will work in desktop computers using ILWIS software.

The volunteers will be in contact with the "*Request*" part of the mechanism, mainly for producing information in the databases. That's why they will be classified depending on their interests (observing plants, animals or both) and depending on which particular specie are they going to observe or if they don't have preferences about any specie. We will refer to them as "Regular users" or "Volunteers" in the future. It's expected that such users use desktop computers or mobile dispositives with Internet access to make requests or upload the observations.

The last kind of user is a special one, the program which analyse the data produced and the existing data in order to update automatically the status map according to pre-established rules. This user will be named "Automatic users". Such "automatic users" will reside in the server and be triggered using schedules and similar methods.

4.2 Interaction analysis

4.2.1 Planners

The interaction for "Planners" is the most complex, it is subdivided in two stages, the first one is connecting to the service and retrieve layers Figure 6. The second one is edit, validate topology and update the server databases Figure 7. The dialogs and particular tools that are going to be implemented will be accessible through buttons in a "WFS-T" toolbar in ILWIS map window, a mock-up of the toolbar can be seen at Figure 8.

The first stage of the interaction takes place on the ILWIS Navigator space, where the user adds a new WFS layer by selecting the WFS root element on the tree and right click "Add WFS server". Then the wizard is launched.

1. The user press adds new WFS server in the navigator tree.
2. The User inputs the URL and the Port of the Web Feature Service in the wizard (that will be stored for future connections to that service as new sub-node of the WFS node).
3. Build a getCapabilities request.
4. Launch the request against the Web Service.
5. Retrieve the answer of the Web Service as XML file.
6. Parse the file and extract the relevant information.
7. Show the capabilities in the "Add WFS Layer" wizard, where the user can select which layers and attributes wants to select and specify filters and other constraints.
8. The user makes the selection and specifies the constraints, and then the prototype builds the getFeature request.
9. The request is launched against the Web Service again.
10. The Web Service processes the request and sends the answer.
11. The prototype captures the answer and produces an XML document.
12. The prototype parses the answer and translates it into features that ILWIS software can manage and opens a new map window with the features retrieved.

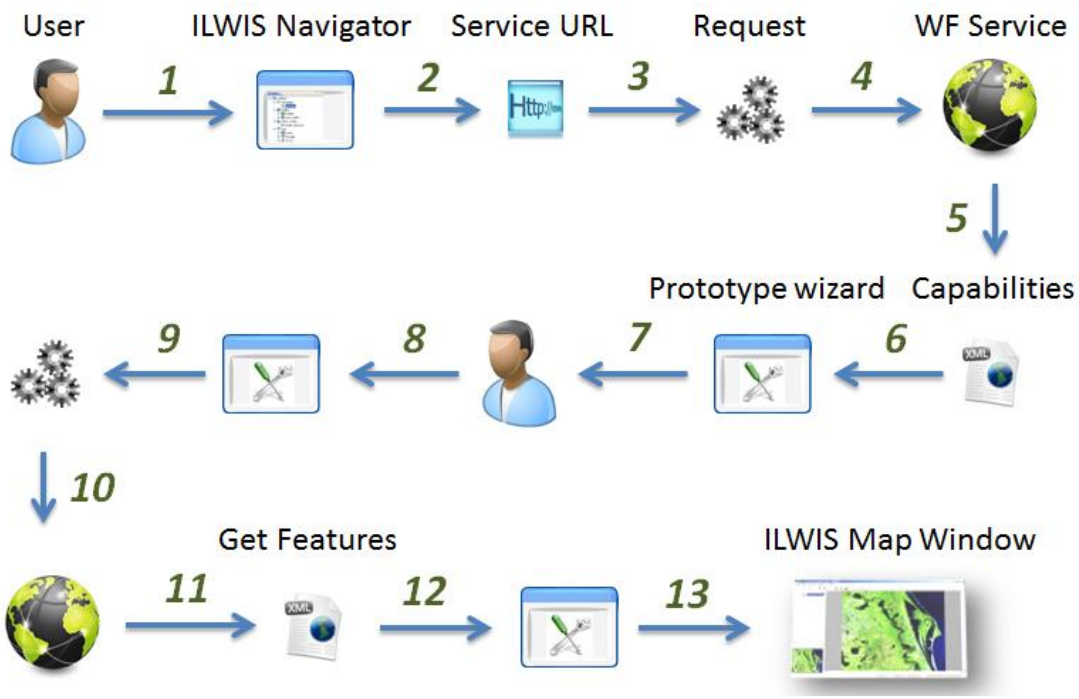


Figure 6. Schema of the proposed workflow to retrieve vector layers in ILWIS using the prototype.

As we can see in the schema the user interacts with a particular GUI, which is a wizard that helps to set the location of the resource and help the user to select those layers that will be interesting for the task, allowing the user to define constraints using SQL queries and other advance features.

If the user decides to add a new WFS layer to an existing map will select the WFS server from the navigator (which stores the URL) but the user will have to specify the parameters of the request using the wizard.

Once the features are loaded in the Map window in ILWIS, the user might want to edit the features, and then the process requires topology control, which is transparent for the user. In this schema we will apply topology *on-the-fly* using Web Processing Service capabilities of Geoserver and the related processes are marked in light yellow.

The Figure 7shows an schematic process of edition, with light yellow background for all steps which have to be done only if validating topology through Web Processing Services are implemented.

1. The User edits the previously retrieved features using ILWIS edition capabilities.
2. Then a WPS query is built by the tool for checking the topology relationships for the modified set of features.
3. The query is wrapped in XML.
4. The query is sent to Geoserver in order to be processed.
5. Geoserver produces an XML with the results of the topology validation.
6. These results are retrieved by the tool.
7. The answer is parsed.
8. Depending on the result the user is informed and asked to update the features to fit the topological constraints or continue to next step.
9. The tool builds a transaction to update the features in the server.
10. The transaction is launched against the Web Service.

The Web Feature Service connects to the database and updates it.

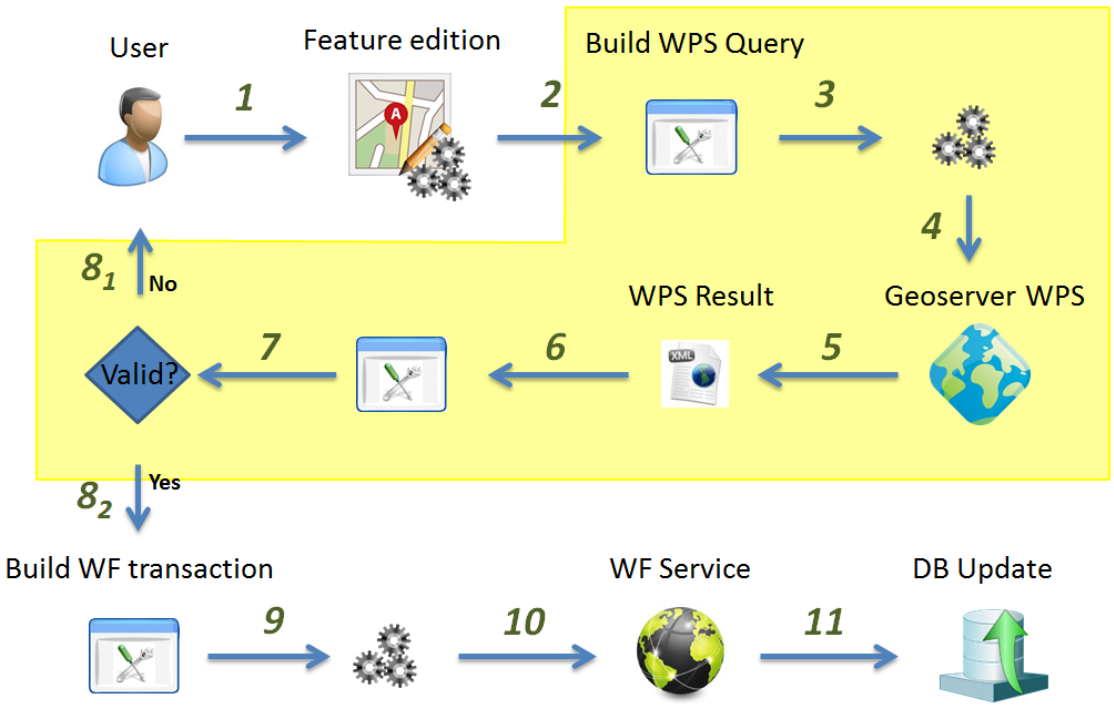


Figure 7. Schema of the proposed workflow to update remotely vector features using ILWIS and the prototype.

The editions will be done using the IWLIS capabilities, but the user will use again a dialog to confirm the topological validation (when it will be implemented) and will receive the status (performed or not) of the transaction through a dialog message.

The user interaction for edit features and attributes should be studied because ILWIS already implements partially editing capabilities, but only attributes in polygon and point geometries, so it is mandatory to propose alternatives to add full editing capabilities.

Nowadays the editions in ILWIS are focused to the attributes by performing double click event on 0D or 2D geometries. That event launches a window with the attributes and its values of the geometry selected.

The user would be interested in edit attributes or geometries, due to the nature of the editions the actual behaviour should be separated in different branches, after analyse the required functionalities and several aspects of human-computer interaction we propose two different alternatives. In those versions that will implement topology control this topology should be considered because could interfere with the regular edition workflow slightly.

4.2.1.1.1 Alternative 1 for managing editions, Click events

We propose an edition environment controlled by mouse click events, where the edition session is opened since the very first moment the map window is opened. Then double left click event over a feature selects the feature (highlighting it with a complementary colour for increasing the perception of the feature in crowded maps, to ensure the user that the selected feature is the one to be edited) and triggers a floating menu with three options:

- Edit attributes
- Edit geometry
- Cancel

Edit attributes option will guide the user to a window to modify the values, it is already implemented in the current version of ILWIS. It would be important continue considering the data domains at this point, this way the editor tool window will adapt the input controls depending on the basic type and the allowed values in the domains that are retrieved for each particular layer. This helps to maintain consistency and attribute accuracy meanwhile editing attributes.

Edit geometry option would de-highlight the feature and mark the vertex of the feature with auxiliary nodes (I.e. blue squares ■), mouse event (right click or left click) will mark the auxiliary vector as select (i.e. in red colour ■ to differentiate between all auxiliary vertex and the one which is being edited) that could be moved by intuitive left click & drag (in case of implicit topology the snapping could be active and help the user to move the vertex to neighbour interesting vertex) or by performing right click a secondary floating menu would show up with advanced edition options like:

- Move to a specific coordinates (A dialog will show up to input the new coordinates)
- Remove vertex (deletes the selected vertex)
- Remove full feature (deletes the geometry and the associated rows in the table)
- Move feature
- Rotate feature
- Cancel edition (restore the feature to last saved place)

In particular case of polylines and polygons, if the user performs right click on one arc of the selected feature instead of the auxiliary vertex the menu should show:

- Add vertex
- Remove full feature
- Move feature
- Rotate feature
- Cancel edition (restore the feature to last saved shape and place)

Once the user has performed a shape edition the feature is still selected with all the auxiliary vertex marked, The feature is still ready for being edited, but when the user finished the shape edition to "update" definitely the shape a left click somewhere else out the feature will end the edition of that feature, cleaning the auxiliary vertex and update the values of the feature in the ILWIS internal storage system.

In case the edited layer would be a WFS one, we would send all the involved features (not only the edited one) with WPS to a topology validation (if the *on-the-fly* topology is used, if implicit topology is used the edition should adapt to the cluster tolerance and make use of snapping techniques to ensure the perfect coincidence of the shapes), and if the validation is correct then the update transaction will be build and executed. If the validation is not correct the user must check the last edition and make sure the topological relationships are fully satisfied in order to be able to save, otherwise the user has to perform right click and select cancel edition to stop the loop.

4.2.1.1.2 Alternative 2 for managing editions, Edition sessions and toolbar

In this alternative the standard ILWIS behaviour (see Figure 9) of editing attributes with double click is disabled. The edition is controlled from a new toolbar "Edition" (See Figure 8 for a mock-up). The concept of editing session is created to specify when is allowed edition or not. This is like a switch activated by one toggle button of such that enables or not the other advanced editing tools. From left to right are:

Figure 8. Proposed toolbar to enable and handling editing sessions using Transactional WFS in the embeeded prototype in ILWIS.

- The enable / disable edition toggle button: Activates or deactivates edition session, only allowed for local layers and those WF Services that allow Transactional support. The other buttons of the toolbar are disabled while the edition session is on. If the button is disabled and have been unsaved modifications the user will be asked to save or discard last changes.
- Save editions: In case of topology validation through WPS sends the validation first and if is valid then creates the transaction and executes against the WF Service that provided the features. Otherwise shows an error message.
- The select feature tool: Selects the feature to be edited. Highlights it and after de-highlight it, showing the auxiliary vertex of the geometry, like in the previous alternative.
- Create a new feature: The software will capture the location of the points input by the user using the mouse and define a new feature. For finish defining the contour of the polygon or the shape of a polyline the user must perform right click and select "Finish sketch". After defining the shape automatically the attribute edition window will show up for input the alphanumeric values associated.
- Edit attributes: This button will be only enabled if there is a feature already selected. Then by pressing the button the attribute edition window will show up. This window is the same than described for the previous alternative.
- Right click event (with feature selected): Like previous alternative the main edition capabilities are performed from the mouse. After this event a floating submenu shows up. This submenu is the same than used in the previous alternative because the editing functions should not depend on the user interface.

4.2.1.1.3 Alternative chosen

In the end both alternatives are functional and could be implemented, providing advanced editing capabilities for vector datasets in ILWIS, both can be used (with slightly modifications) with other sources and applications, even with more geometric types like points or polylines. That would make the ILWIS software more powerful and versatile.

For all those applications that do not require topology it could be interesting make the topological verification optional, using a button that implements the required topological relationships for the request-status map mechanism we would obtain generic edition capabilities for vector features and the possibility of verify the specific topology that a status map should satisfy.

Finally, analysing all the aspects of the interaction in editions we conclude that the second option is more flexible because adding a new button we could obtain a generic mechanism for edition and automate the verification for certain topologies is easier and more intuitive than control the mouse events to trigger topology verifications.

Besides, the verifications will be done under request, not each time geometries are edited. It turns in an increase of speed and release the dependency of availability of WPS services to complete editions. More improvements can be done in future versions, like turning this pre-established topology verifications in a dialog window where we could customize the rules used and save it in profiles to increase the productivity.

Then, the edition toolbar will be implemented adding a new button to check the topology using WPS (in the corresponding version of the prototype).

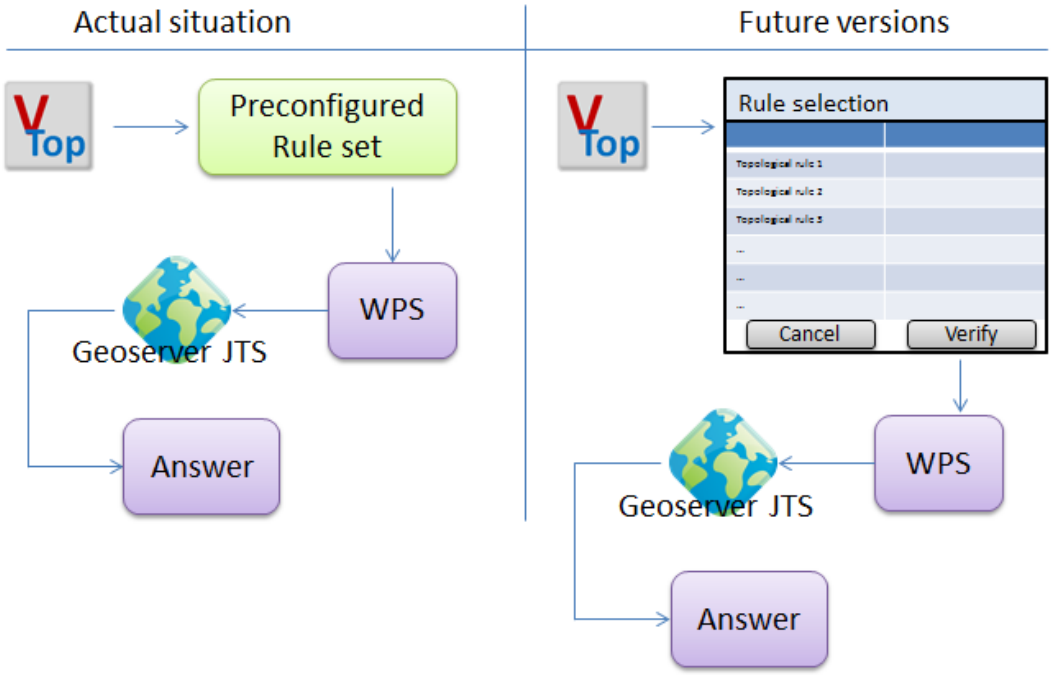


Figure 9. Comparative between actual and proposed editing sessions in ILWIS software (in case of topology rules).

4.2.1 Regular users

Regular users, (the volunteers) will use different hardware configurations for checking which location they shall visit for producing information. They might use GIS clients (Like ILWIS or ArcGIS) in desktop environments (like home or offices) or can use smartphones or pads to access through light web clients to status maps. They can browse the status map using these tools, but at one point they have to ask for possible locations (sectors) to work in. In case of desktop environments the prototype tool is expected to implement the request mechanism for requesting areas from ILWIS software taking profit of the main status map infrastructure already built. In ubiquitous environments (smartphones, tablets or laptops) it will be done by accessing the web application, which will check the preferences of the user and cross it with the status map, for proposing areas to work in.

Then the user will select one and the web application will update the databases. Figure 10 shows a mock-up of how the application would look like.

For tablets and PC's the bigger size of the screen allows integrate all required parameters in two windows, using the ILWIS map window as viewer, avoiding the excessive use of menus.

The combination PC + ILWIS for using the request mechanism could not be the most suitable in first instance, but ILWIS is, in the end, educational software, so it could be interesting provide such capability in the main window. It could be done adding a new button or sub-menu item that launches the wizard. A mock-up could be Figure 11.

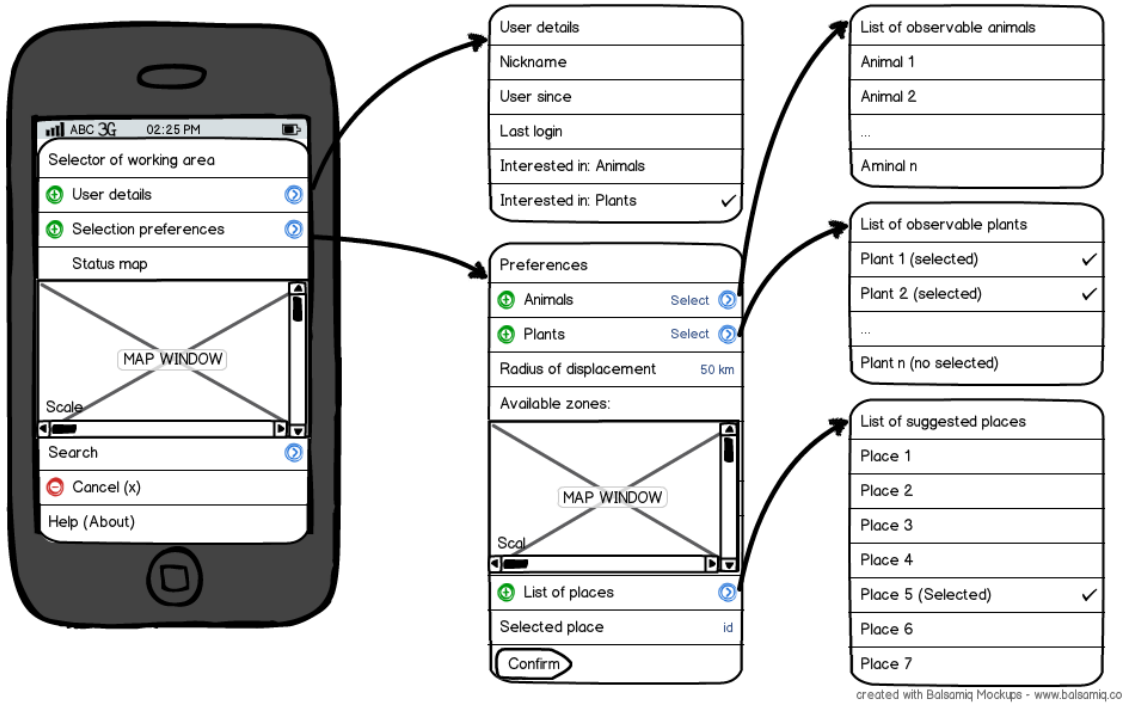


Figure 10. Mock-up of a request mechanism implemented in a mobile device software. Only for demonstrative purposes.

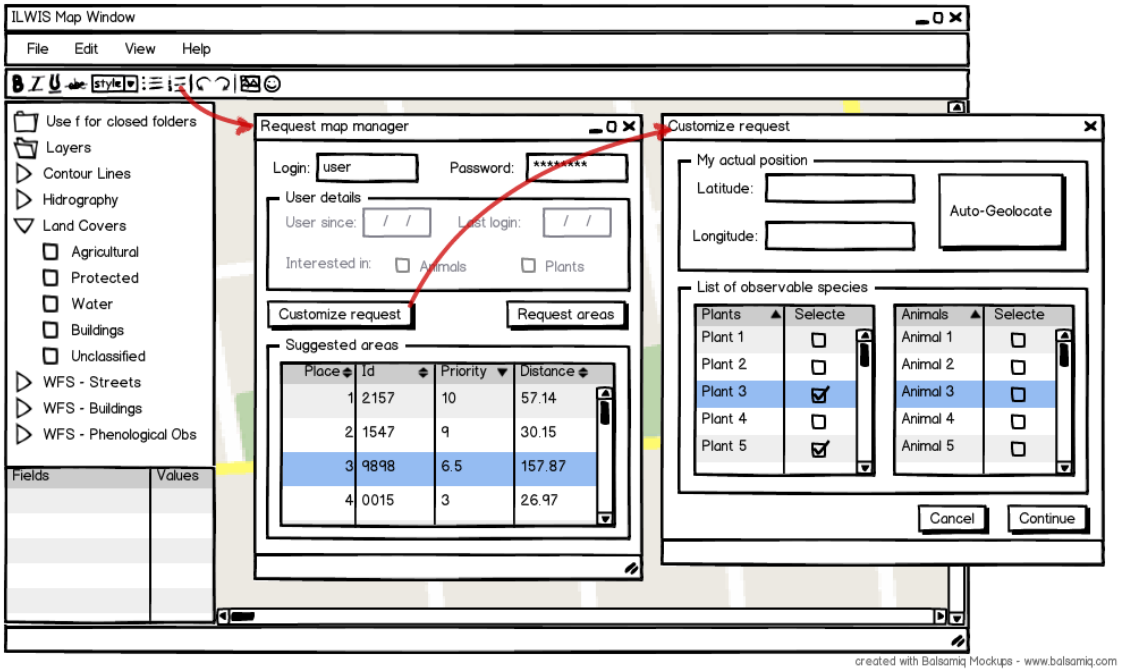


Figure 11. Mock up for ILWIS VGI planning add on.

4.2.1 Automatic users

Automatic users are programs that are triggered by events; the interaction of this user with the tool will be done by scripts and routines that will make use of the implemented routines. There's no need to create GUI's for these users.

5 Prototype roadmap

The development of the prototype will be split in several stages depending on the functionalities that will implement. The roadmap is summarized in Table 2.

Functionality	0.1	0.2	0.3	0.4	0.4	0.5	0.6	0.7	0.8
Explore Service Capabilities in console app	x	x	x	x	x	x	x	x	x
Retrieve Features from WFS in console app		x	x	x	x	x	x	x	x
Implement transactions for attributes in console app			x	x	x	x	x	x	x
Store WFS Servers in ILWIS				x	x	x	x	x	x
Launch the capabilities wizard within ILWIS				x	x	x	x	x	x
Create maps from / with WFS layers in ILWIS				x	x	x	x	x	x
Implement transactions for features in console app					x	x	x	x	x
Implement topology validation in console app						x	x	x	x
Implement save edits and topology validation in ILWIS						x	x	x	x
Create an interface for request mechanism in ILWIS							x	x	x
Mark topological errors in ILWIS map window							x	x	x
Implement automatic re-calculation of status maps								x	x
Create new status maps in remote servers using ILWIS									x

Table 2. Proposed iterations roadmap.

For the development of the request-status map mechanism a new library in C++ will be developed, creating a console application that makes use of such library (and other related) to check the functionality, when the functionality is tested and works as expected then the ILWIS software implements the new commands for making use of libraries to implement the functionality.

Chapter 3: Development

6 General overview

After several meetings with involved parts in the research department we have gathered enough information to start planning the development stage.

The implicit and explicit constraints of the ILWIS project and the R-S maps mechanism, altogether with a quick assessment of our own software development skills and budget capabilities make us consider Visual Studio 2008 as IDE to develop the C++ application.

Actually ILWIS is made of many C++ projects in visual studio (35 projects the day the text was written) and we will contribute to "IlwisDataExchange" project, which contains already the WMS service capabilities and classes for OpenStreetMaps or Raster formats. Although it was designed with educational purposes to make easy students kick off with spatial data management and analysis quickly evolved to integrate new functionalities demanded for other research teams etc., growing in size and capabilities till today.

The project we are working in is composed of many different code files, to fit in the structure the main developer of ILWIS created two files for us, where we will implement the functionalities within their model, the file "WFS.cpp" and its corresponding header, and we will deploy the connector and the log files with their corresponding header.

Besides that some other code files will be generated to manage the graphical user interfaces and wizards required for an intuitive and complete experience with the prototype, although we will not extend ourselves more in that direction because it will be produced by the main developer to get homogeneous behavior and appearance, generating classes and code in "IlwisDataExangeUI".

Nonetheless, some of the functionalities required to the prototype are not fully implementable in actual version of ILWIS because its own intrinsic architecture. Hence the development has been split in two, one stand-alone prototype with all the functionalities and one ILWIS project implementing the operations that nowadays ILWIS can handle within its own data model for status maps. This last one will be based on the stand-alone version and will be carried on by the ILWIS main developer at his own pace, adding gradually new pieces according new capabilities to be implemented natively in ILWIS as the model evolves to support all the options that the prototype can provide.

7 Development

7.1 Prototype design

After the common first steps in every development (requirements gathering, constraints, functionalities, roles, etc., etc.) the next stage was designing a class diagram. After some tests and basic designs we realized the class diagram was turning into something too complex compared to what was expected from a piece of software that satisfies the requirements. This is caused mainly because the big capabilities of WFS-T, which requires implementing transactions, specializing in classes for each one and full of relationships between them, with constraints depending on the geometry and entities. It turned the class diagram in something very difficult and time-consuming to handle and to translate into functional code.

The required granularity level to model all the behaviour and particular aspects of the prototype was too small, producing a big diagram difficult to implement, it made us think in simplify it to the maximum and keep things simple, keeping in mind this is a proof of concept. The research team wanted to have a prototype working as soon as possible.

The initial idea of generating all into the same WFS source file soon turned impossible to handle properly and we decided to add the ConnectorWFS and Log files.

Then we have a class ConnectorWFS that implements the required functions to ask, retrieve and also send information using the Web Feature Service through an interface of public functions well documented. We have also a Log class, implemented using the singleton pattern and it has been the main debugging tool altogether with the Visual Studio IDE.

This class can be ported to other platforms or software to provide WFS-T basic capabilities, and the ILWIS WFS file acts as a bridge between ILWIS internal objects and the data retrieved by the WFS Connector.

7.2 Models and schemas

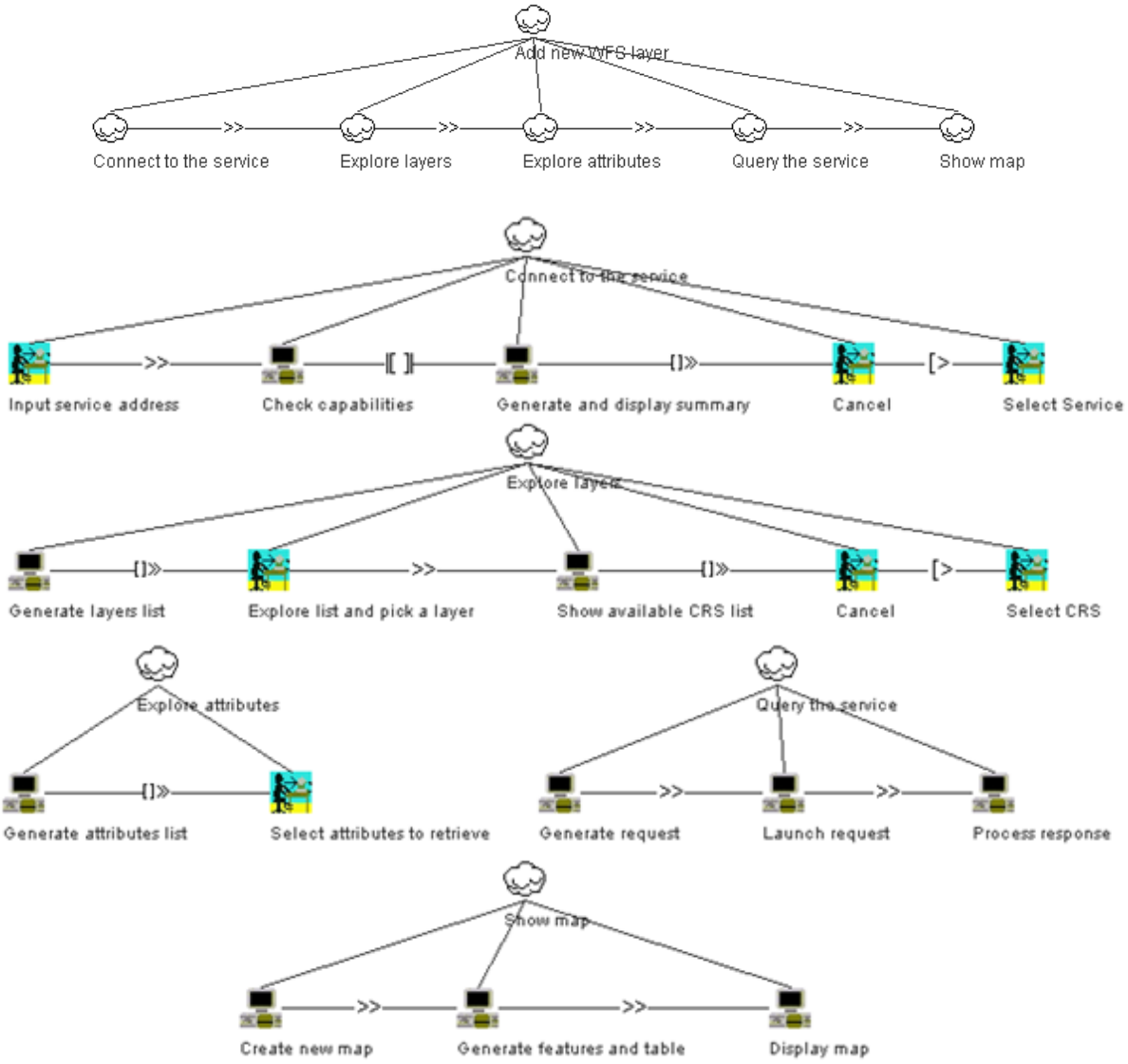


Figure 12. Schemas for HCI in ILWIS wizard (planning perspective).

There is a BPMN diagram for the expected wizard interaction in the Annexes section more detailed than the previous diagram.

7.3 Interfaces (public)

7.3.1 WFSConnector

The connector provides high level methods to handle the operations of the web service:

Besides the destructor and two constructors (depending on the information we have about the service address) users can make use of:

- getAttributeNameList
- getAttributeType
- getFeaturesFromLayer
- getLayerAbstract
- getLayerBoundingBox
- getLayerSRS_FullEPSG
- getLayerSRS_OnlyEPSGCode
- getLayerTitle
- getListLayerNames
- executeTransactionDelete
- executeTransactionInsert
- executeTransactionUpdate

There are some other private methods that helps with intermediate stages of the process, but there is no point on making them public (i.e. getLayerNamespaceURL) due to the profile of the expected users.

Querying a WFS service produces a complex output and then, it is mandatory to set up some environment to ensure a safe execution. That's why the constructor calls "InitializeConnector" method, which ensures the web service exists and it is ready to be queried and also retrieves basic information about it, very useful at wizard GUI.

Once it is built the connector is ready to query the service using the public methods implemented, an usual workflow would be check which are the available layers (getListLayerNames), choose the layer(s) that could be interesting work with and ask for more information (attributes, spatial reference system, spatial extension, abstract, etc.). Finally if the layer is interesting the user can retrieve the features in order to generate maps using "getFeaturesFromLayer".

The connector also allows the user to update the databases remotely throughout the web feature service if it implements transactional support. It is fully functional but there is no point in WFS transactional without authenticate systems. This aspect requires more investigation to find a suitable solution.

Besides the methods in the class the prototype implements a structure (MemoryStruct) and a function (WriteMemoryCallback) to place the information obtained with curl after querying the service.

7.3.2 WFS

WFS.cpp provides high level methods to use the connector with particular functionalities in ILWIS.

- importWFS
- initialize
- generatePointMap

- generateLineMap
- generatePolygonMap
- ParseCoordinatesPoint
- ParseCoordinatesLine
- ParseCoordinatesPolygon

The main point is the method "ImportWFS", which works as entry point for the data retrieved using the connector to feed ILWIS objects. This method parses the WFS answer and redirect to the right function depending on the geometry type of the map the user wants to retrieve from the service (generatePointMap, generateLineMap or generatePolygonMap).

Those "generate" functions create a new ILWIS map object, based on the Spatial Reference System of the information retrieved using the web service (Actually the prototype can handle a large amount of Spatial Reference Systems, all the SRS belonging to EPSG database implemented in the cartographic server used by the web service the user is querying. But we had several problems defining such SRS automatically in ILWIS, and since is a prototype for now we are working in WGS84 in geodetic coordinates until the automatic CRS definition and creation is fixed in ILWIS).

After creating the map object an auxiliary parsing function is called to extract the information retrieved in the service (an XML document full of features and its attributes) and produce an output directly applicable to geometry ILWIS objects (Points, Polylines or Polygons). The control comes back to generate function that creates the geometries, add them to the map, creates the associated attribute table, fill it, link it to the features and finally show the map to the user.

The ConnectorWFS class implements more functionalities than the WFS class manages nowadays, but newer versions of ILWIS software will implement other features that will allow managing WFS-T capabilities prepared in the prototype. If this class is used by other programmers to extend their software they can implement their own transactional extensions in their "equivalent WFS" file.

7.3.3 Log

The log file has been very useful in debugging stages, and since it follows a singleton model the interface is:

- Instance
- printToLog

Where "instance" provides the object and "printToLog" is an overloaded method that produces friendly and readable output for those methods implemented in the connector. It is planned to implement some other methods to increase the versatility of the log by adding new methods to control size and multiple logging levels (depending on the debug level the log file can turn into a very large file).

7.4 External libraries

Before starting witting the code we had to select some libraries to help ourselves and avoid reinventing the wheel by developing everything from scratch. Mainly we needed xml parsing capabilities and connectivity capabilities.

C++ is a very powerful and popular language, there have been many developers working on it and there is a wide offer of libraries available. It took some time explore them and run some tests, because we had many options that apparently would have worked for us.

ILWIS itself is (compared to the capabilities of the software) very small, everything is packed in an installer. And some of the libraries we've looking into (i.e. Xerces) were bigger than the main software package (around 30Mb). Then we looked for light-weight libraries with enough functionality, and finally we selected pugixml[8] and curl[9].

Once the libraries were added and the IDE was set up and ready we started a new console application and started with the basics, the constructor. Once the project was integrated in ILWIS solution we made use of the libraries implemented in ILWIS for managing http requests and xml documents. Luckily the libraries selected for this tasks in ILWIS were the same (it wasn't done on purpose), hence, the only task was redirect the libraries in the includes to make use of the ILWIS already set-up environment.

7.5 Prototype GUI

At the time of writing this chapter the graphical user interface designed to manage the WFS requests from ILWIS is not ready, hence we are forced to use the common import layer. We use that form to launch our piece of code in WFS.cpp to import the layers.

The actual interface is not ready to implement neither the WFS-T authentication nor the transactions to update databases remotely throughout the service. But it is present in the roadmap for future versions.

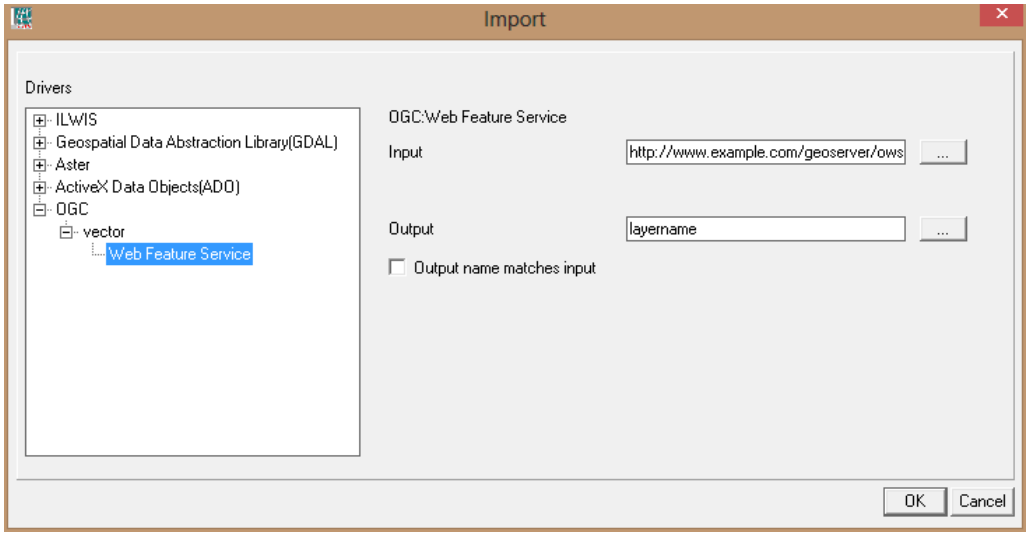


Figure 13. This is the General Import Form in ILWIS. It has been suggested to use the "…"as link to add WFS wizard.

7.6 Debugging

There hasn't been an automatic debugging strategy, because of the scope of the project, its experimental character and the number of people involved in its development we have decided to use "echo debugging" (Silva 2013) and make extensive use of "Log" class (which implements easy-to-read representations of the features retrieved by the service) to ensure it works properly.

7.7 Functionality tests

At the time of writing this chapter the graphical user interface for adding a WFS layer in ILWIS is not ready, it forced us to test the developed mechanism by creating an auxiliary code file which made use of the connector and the bridge, specifying by code processes like creating a map or create a feature remotely using WFS.

[8] For more information about this library check the pugixml official website.
[9] For more information about this library check curl's official website.

We have run several tests, some of them involve only the bridge between ILWIS and the connector, but most of them make use of both. They have been growing in complexity and some of them make use of previously checked functionalities:

- Test GetCapabilities WFS operation
- Test DescribeFeature WFS operation
- Test GetFeature WFS operation
- Retrieve a full layer and all its attributes
- Retrieve a feature and its attributes
- Creating a new map in ILWIS and set up the right SRS
- Retrieve a full point layer in ILWIS
- Retrieve a point layer specifying a query in ILWIS
- Retrieve a multipoint layer (and implicitly transform it into single point layer) in ILWIS
- Retrieve a full line layer in ILWIS
- Retrieve a line layer specifying a query in ILWIS
- Retrieve a multiline layer (and implicitly transform it into single point layer) in ILWIS
- Retrieve a full polygon layer in ILWIS
- Retrieve a polygon layer specifying a query in ILWIS
- Retrieve a polygon layer (and implicitly transform it into single point layer) in ILWIS
- Delete features remotely using WFS-T
- Delete features in a layer specifying a query in WFS-T
- Create new feature remotely using WFS-T
- Update feature remotely using WFS-T

As example some screenshots have been attached to graphically document the testing stage. We can see how the log shows in console the result of querying the service using the connector (Figure 14). Also we can see how a polygon map has been retrieved using the prototype and is shown in an ILWIS map object (see Figure 15). And finally an example of the reply of the web service to a "Delete feature transaction" executed using the connector against WFS (see Figure 16)

Code block: Creating a WFSConnector object

```
// test3.cpp : Defines the entry point for the console application.

#include "stdafx.h"
#include <ConnectorWFS.hpp>
#include <Log.hpp>

int _tmain(int argc, _TCHAR* argv[])
{
	//Init the log object
	//SINGLETON MODEL// From now on use always this structure!!
	Log *log = Log::Instance();

	const std::string str_url = "http://192.168.43.101:8082/geoserver/wfs?";
	ConnectorWFS *conn = new ConnectorWFS(str_url); //Create object in the heap

	delete log; //We call automatically the destructor from here.
	system("PAUSE");
	return 0;
}
```

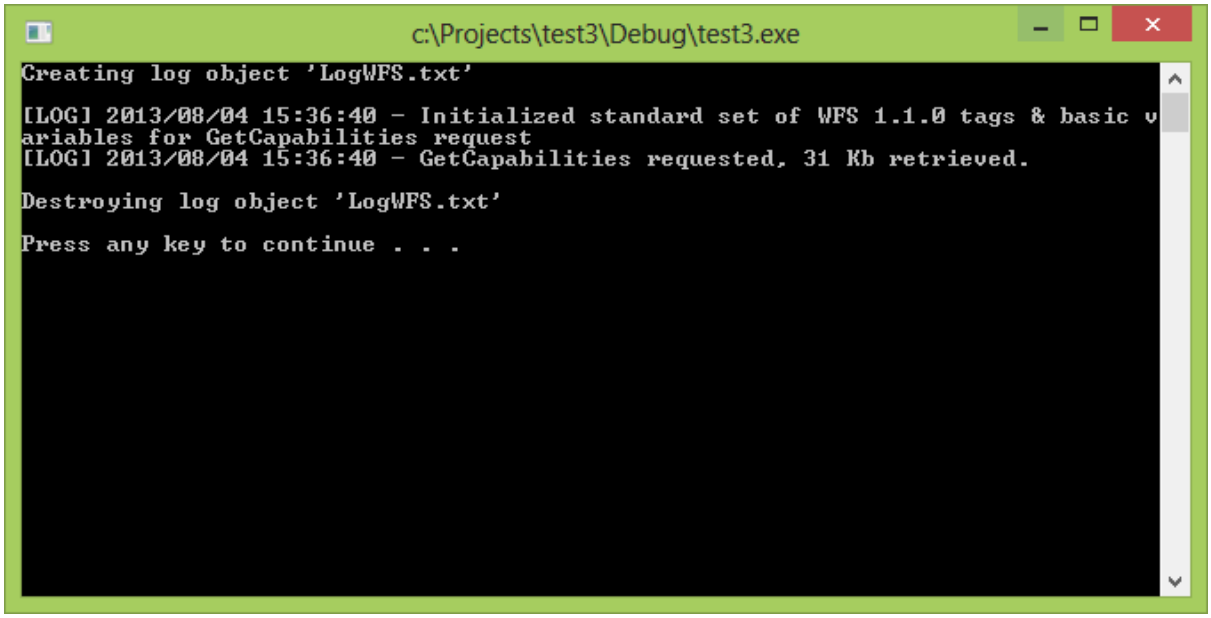


Figure 14. Command window (standard output) resulting of creating a WFSConnector object, where calling "GetCapabilities" operation from WFS is one of the initializing functions after creation.

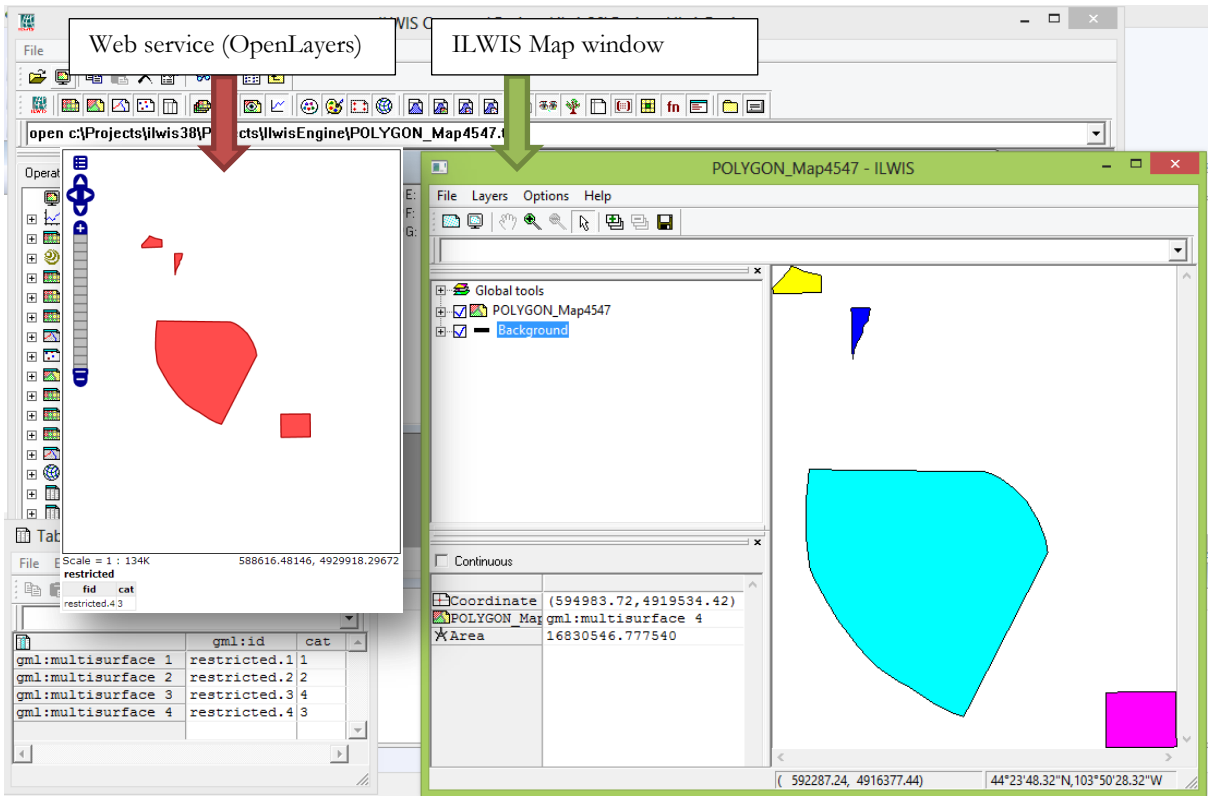


Figure 15. Composition of the real features in the Web Service queried in a Web browser using Open Layers and the corresponding features loaded in a new ILWIS Map window.

The log file contains a summary of the main internal operations used to generate that "simple" map.

Extract from log file creating a polygon map:

```
2013/08/04 16:55:53 - Initialized standard set of WFS 1.1.0 tags & basic variables for
GetCapabilities request
2013/08/04 16:55:53 - GetCapabilities requested, 31 Kb retrieved.
2013/08/04 16:55:53 - Retrieving attribute names...
2013/08/04 16:55:53 - Executing DescribeFeatureType request...
2013/08/04 16:55:53 - DescribeFeatureType requested, 1 Kb retrieved.
2013/08/04 16:55:53 - DescribeFeatureType request executed.
2013/08/04 16:55:53 - Attribute names retrieved.
2013/08/04 16:55:53 - Searching for the geometry column...
2013/08/04 16:55:53 - Geometry column found.
2013/08/04 16:55:53 - Obtaining geometry type for the layer...
2013/08/04 16:55:53 - Retrieving attribute type for the geom ...
2013/08/04 16:55:53 - Attribute type retrieved [gml:MultiSurface].
2013/08/04 16:55:53 - Geometry type obtained (gml:multisurface).
2013/08/04 16:55:53 - full SRS code retrieved
2013/08/04 16:55:53 - Retrieving attribute names...
2013/08/04 16:55:53 - Executing DescribeFeatureType request...
2013/08/04 16:55:53 - DescribeFeatureType requested, 1 Kb retrieved.
2013/08/04 16:55:53 - DescribeFeatureType request executed.
2013/08/04 16:55:53 - Attribute names retrieved.
2013/08/04 16:55:53 - Retrieved 4966 bytes.
2013/08/04 16:55:53 - Retrieved features from [sf:restricted]:
http://192.168.43.101:8082/geoserver/wfs?SERVICE=WFS&VERSION=1.1.0&REQUEST=GETFEATURE&TYPENAME
=sf:restricted
2013/08/04 16:55:53 - Retrieving attribute names...
2013/08/04 16:55:53 - Executing DescribeFeatureType request...
2013/08/04 16:55:53 - DescribeFeatureType requested, 1 Kb retrieved.
2013/08/04 16:55:53 - DescribeFeatureType request executed.
2013/08/04 16:55:53 - Attribute names retrieved.
2013/08/04 16:55:53 - Searching for the geometry column...
2013/08/04 16:55:53 - Geometry column found.
2013/08/04 16:55:53 - Obtaining geometry type for the layer...
2013/08/04 16:55:53 - Retrieving attribute type for the geom ...
2013/08/04 16:55:53 - Attribute type retrieved [gml:MultiSurface].
2013/08/04 16:55:53 - Geometry type obtained (gml:multisurface).
2013/08/04 16:55:53 - Retrieving attribute names...
2013/08/04 16:55:53 - Executing DescribeFeatureType request...
2013/08/04 16:55:53 - DescribeFeatureType requested, 1 Kb retrieved.
2013/08/04 16:55:53 - DescribeFeatureType request executed.
2013/08/04 16:55:53 - Attribute names retrieved.
2013/08/04 16:55:53 - Searching for the geometry column...
2013/08/04 16:55:53 - Geometry column found.
2013/08/04 16:55:53 - WARNING: No <gml:featureMember> tag found, trying with Geoserver buggy
custom tag: <gml:featureMembers>
2013/08/04 16:55:53 - Cantidad de nodos gml:featureMember(s) 1
2013/08/04 16:55:53 - GeometryType is gml:multisurface
2013/08/04 16:55:53 - Number of FEATURE nodes found: 4
2013/08/04 16:55:53 - Number of POLYGON nodes found: 1
2013/08/04 16:55:53 - Number of INNER POLYGON nodes found: 0
2013/08/04 16:55:53 - Vector Lista de coordenadas [size: 3]
2013/08/04 16:55:53        + 591954.3359385637 4925859.483293386 591957.3824433011 …
2013/08/04 16:55:53        + restricted.1
2013/08/04 16:55:53        + 1
2013/08/04 16:55:53 - Number of POLYGON nodes found: 1
2013/08/04 16:55:53 - Number of INNER POLYGON nodes found: 0
2013/08/04 16:55:53 - Vector Lista de coordenadas [size: 3]
2013/08/04 16:55:53        + 593183.6607205212 4923980.52355841 593157.5354776975 …
2013/08/04 16:55:53        + restricted.2
2013/08/04 16:55:53        + 2
2013/08/04 16:55:53 - Number of POLYGON nodes found: 1
2013/08/04 16:55:53 - Number of INNER POLYGON nodes found: 0
2013/08/04 16:55:53 - Vector Lista de coordenadas [size: 3]
2013/08/04 16:55:53        + 598239.5942270659 4917334.785918331 599645.0893316591 …
…
…
```

Code block: Delete transaction: (using query) "Delete all streams of second category"

```
// test3.cpp : Defines the entry point for the console application.

#include "stdafx.h"
#include <ConnectorWFS.hpp>
#include <Log.hpp>

int _tmain(int argc, _TCHAR* argv[])
{
	//Init the log object
	Log *log = Log::Instance();

	const std::string str_url = "http://192.168.43.101:8082/geoserver/wfs?";
	ConnectorWFS *conn = new ConnectorWFS(str_url); //Create object in the heap

	//Select the layer and the element to erase (manually)
	std::string str_lyr = "sf:streams";
	std::string str_field = "sf:cat";
	std::string str_value = "2";

	conn->executeTransactionDelete(str_lyr,    str_field,    str_value,    "WFS",
"1.1.0");

	delete log; //We call automatically the destructor from here.
	system("PAUSE");
	return 0;
}
```

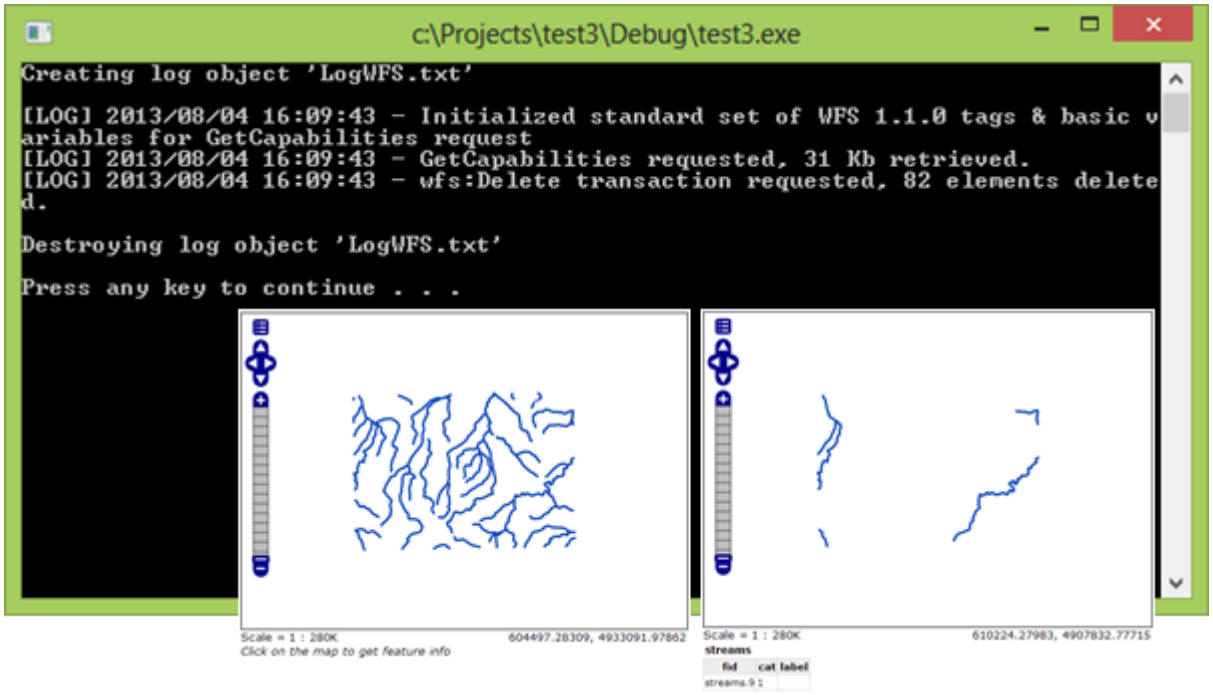


Figure 16. Command window with the summary after removing all the water streams that satisfy a particular query. 82 features have been removed and it is possible to compare "before" and "after".

Chapter 4: Preparing status maps

In this chapter we are going to explain and describe the involved tasks that compose the workflow to obtain status maps. Besides as a result of our research in this field we are going to propose alternatives in some steps of the process when possible. Such alternatives are related with diverse application scenarios, future technologies or functionalities implemented in futures versions of the tool which is being developed.

8 Setting up the common framework

As described in the Methodology chapter our status maps will share a common set of attributes that provides basic information useful in multiple purposes. Besides, another set will contain specific attributes focused on the main purpose of the status map, such attributes will be the base to perform other analysis to derive information and obtain meaningful parameters to be used in the status or priority fields.

But before start creating and calculating fields it's mandatory to establish the data model, a framework composed by a set of constraints, parameters, structures, etc. that will ensure a generic model for our request-status map mechanism. It's very important to set up a generic environment that can be used for all the community, that excludes proprietary software and formats and regional reference systems between other important aspects.

8.1 Information formats

Since the beginning of the Geographic Information Systems a big set of formats have been developed, some of them proprietary software others free. At this point we have many options available, depending on our strategy.

For creating the status map we will need some tools to derive information from our base layers, even though some of them use formats that are not compatible with the cartographic server used in our prototype. In that case we can prepare the status map in our software and export it after all or simply work with compatible formats with the cartographic server since the beginning. It's important to care about the exporting process, because some accuracy problems can appear and infect the involved layers, but if an expert supervises the export operation procedures both datasets will be equally valid.

Format	Considerations
Shapefile	Very old (and extended) exchange format, mostly all software and cartographic servers can handle it. Does not support topology, tolerances nor domains. Not implemented in databases, so there are problems with concurrence and versioning.
Geodatabases	Proprietary format, allows internal structures, domains, tolerances and topological rules. Needs ArcGIS to manage it. Depending on the license uses Microsoft Access engine or other engines to work.
PostGis Tables	Requires a PosgreSQL+PostGIS installation, allows tolerances, domains and topology in version 2.0. There are many free tools that can handle this format and provide the tools required to elaborate status maps.

Table 3. Recommended formats for feeding the service at the Status Map section of the mechanism.

This list of is very short, but those are the main realistic alternatives. There are also WFS sources available or other proprietary formats like Oracle Spatial, which have all those functionalities and also allows versioning with a great level of performance, but it's too expensive and there's no point on use economic

resources in this engine because we can achieve the same results with 0-cost tools that also have good performance.

In our development we are using Geodatabases in ArcGIS 10, it means our cartographic server won't be able to read our status maps if we do not export them to PostGis or shapefile. We do it this way to use the advantages of cluster tolerance, topology rules and the quality of the geoprocessing analysis tools implemented in ArcGis. It means we finally will export our datasets to shapefile to upload them in the service.

Using this workflow ensures a shapefile topologically correct and without information out of our data domains because is direct product of a Geodatabase export.

An alternative is to use PostGIS tables in QGis client to perform the analysis. It requires a basic knowledge of SQL language, but the result is directly uploaded to the service and the quality is ensured, keeping the data domains to the final step of the process which is an advantage to the first methodology, although WFS does not consider data domains it is always one extra security step in attribute quality. Since the analysis operations to be used are basic this second option is also very good and feasible.

8.1 Setting up our base maps

Once decided which format and the workflow to be used we are ready to set up the base map, common element for all status maps. This step can be done by the user in case the proposed one here does not fit the requirements.

Since we are providing a generic tool there's no point in fixing the spatial domain of the status map, it means that we have to use maps capable of represent information in the full planet; it is an implicit constraint of reference system.

It means that we cannot use local ENUs[10], and since the planet shape is approximate an ellipsoid, mathematics tell us there's no way to find a projection that preserves all the magnitudes once the shapes are transformed to a plane. It means that we shouldn't use either a projected coordinate system (UTM is very extended and solves the problem along latitude by splitting the ellipsoid in UTM Zones, even tough, we shouldn't use this projection at high latitudes because of the deformation that implies.

The same way in these cases we could use a polar projection, but then we won't cover the equatorial areas, so from here we deduce that we must use a global reference system based on geodetic coordinates (non-projected coordinates), to avoid projection problems and to get global coverage (there's no point in use cartesian geocentric coordinates, such cartersian geocentric vectors have no intuitive interpretation at ellipsoidal surfaces and could result messy for final users).

We work this way because we want to create an easy-to-use generic mechanism, the user can create its own status map for local areas to avoid this complexity if the scope of the project is well defined as local scope, besides, in our case of study for the prototype we will work with the United States of America, which could be considered continental scope.

The science of geodesy has been working in this field for many years, and nowadays we have very good global systems available, product of the explosion of GNSS[11]. As a consequence we can choose between

[10]ENU: Easting Northing Upping, acronym of local system that considers the planet as plane element. Very used in engineering projects because it simplifies the calculations.
[11] GNSS is the Acronym of Global navigation satellite system. Examples of this are the GPS (USA), GLONASS (Russia) and COMPASS-BEIDOU (China).

many global reference systems. The most known are WGS84 (World geodetic system 1984), PZ90 (Parametry Zemli 1990) and ITRS (International Reference System) the most precise one.

We could use a materialization of ITRS for one epoch, also known as reference frame. This reference frame is a set of spread stations in the earth with coordinates calculated in ITRS for one temporal point (it is so accurate that can detect tectonic movements or tidal oscillations), since ITRS was defined many solutions have been obtained (ITRF[12] 92, 93, 94…, 2008). Such stations can be used as an anchor to transform observed coordinates with our mobile devices in order to feed the service in ITRF coordinates.

Although scientifically is the best option (it could provide the best accuracy in our system and be applicable to many scenarios) this ITRF solution is not the best choice for VGI applications, some of the reasons are:

ITRS is so accurate than needs temporal anchor, it means the coordinates at one point in the space are valid only for that moment, otherwise we will have to use information about velocity and acceleration in that point and the current date to "move" the point to the real ITRS location.

Such displacements are, almost always, of millimetric or centimetric order, but VGI users probably will produce spatial locations using their mobile devices, which usually are not ready to use high accuracy methods in GNSS, achieving much less accurate data than the spatial reference can handle, introducing extra computational cost caused by transforming all the coordinates which usually are in WGS84. We could say using ITRF in VGI is like *"breaking nuts with a sledgehammer*".

Instead of that we can use directly WGS84 reference system, which is the native reference system used by almost all the GNSS devices and has also enough global accuracy to be used in any part of the world. This way the users just provide the coordinates obtained by their devices, no matter where or when and without any extra effort. To identify this reference system is very common use the European Petrol Survey Group code, which is 4326.

The WGS84 also evolves parallel to time, modifying their coordinates to adjust the crust movements and other sources of variations like ITRS does, but this is done automatically by the system, besides the last period between modifications was 10 years and the variation was 6 cm, so we can still use the WGS84 without any problem.

At this point we specified which spatial reference system and which coordinate system we are going to use. The next step is subdividing the globe in sectors that will be used for ease the location of the areas of the status map at global level.

The zonification implies decomposition in east/west and hemispheres of a grid of 2x2 degrees; it works as reference grid for further decompositions for status maps. From now on these 2x2 degrees areas will be named sectors, and each sector is approximately 50.000 km^2 close to equatorial latitudes.

Each sector is also subdivided in a standard grid of 6x6 minutes of latitude and longitude, which is approximately 120km^2. Hence each sector contains 400 areas by default, although they can be generalized (i.e. in deserts or seas) or increased in number (i.e. cities or special areas) depending on the application or the necessities of the planners.

We have to keep in mind that the longitude of a parallel arc depends on the latitude, it means that such grid is not square, is more "trapezoidal-shaped" (See Figure 17). Furthermore the earth is not completely

[12] For mor information about ITRF visit the official website of ITRF (Institut Géographique National (IGN France) 2012) and IERS (Federal Agency for Cartography and Geodesy 2013), the organism in charge of the project

spherical, so the longitude of an arc of meridian isn't the same at different latitudes. Even though the areas are not square shaped and the surface covered will vary depending on the latitude we consider this is the best option because there are no projections that can cover the world or do so with accuracy, and such trick to get universal projections (like UTM) require dividing the globe in zones, which produces jumps and lead to confusions in non-expert users.

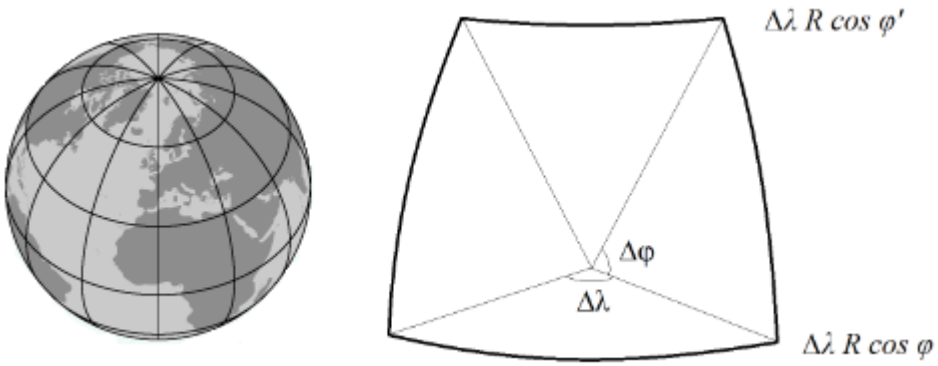


Figure 17. Composition detail of the different length of the spheroid arc depending on the latitude. Source: Map Mathematics.

Hence the most suitable option is using geodetic coordinates in WGS48, the ones provided by default in almost all mobile devices that can be potentially used by volunteers.

9 Obtaining base information

Depending on the goal of the request-status map mechanism the planners will have to get or derive from existing information the required layers that they will use for their analysis.

In the request-status map mechanism the spatial accuracy and precision of the layers involved in the process is not as important as thematic accuracy is. However, it is always good keep in mind that as good practice and try to get spatially consistent datasets (beyond the spatial topology) to ensure a high-level quality product as result, that won't skew the potential and undiscovered yet applications of the mechanism.

In our case of study we will generate the initial status map (taking the planner role) as result of a combination of spatial analysis based on those layers mentioned in Information analysis4.2 Information analysis.

The sources of this spatial information are the AWIPS database (National Oceanic and Atmospheric Administration's (NOAA) 2012), the US National Atlas (U.S. Geological Survey 2013) and Natural Earth (NACIS 2013) mainly. Such resources are free available and cover different areas at different scales and constitute a very good starting point to collect geographic data.

10 Deriving data from base layers

It is not the goal of this MSc. thesis define the spatial analysis that will the planners use for obtaining the layers (such analysis will be different depending on the goal and the scope of the mechanism). But we will describe briefly the methodology for the use case that can be used as basic example for other applications.

First we will configure a simulation of existing reports according to spatial distribution of cities, communications, ecoregions and population distribution. This is just to have departure info ready to use for calculating status maps.

After that we compile all the layers and perform a multivariable spatial analysis, to obtain initial the initial status map, and add a random dispersion in the values to increase the variability (this step is just to make sure the example scenarios are versatile enough to test the prototype).

This spatial analysis include creation of digital elevation models, rasterization and vectorization processes, spatial joins and other overlay analysis tools, slope maps, elevations map, distance maps and a big set of binary map calculations to produce the results.

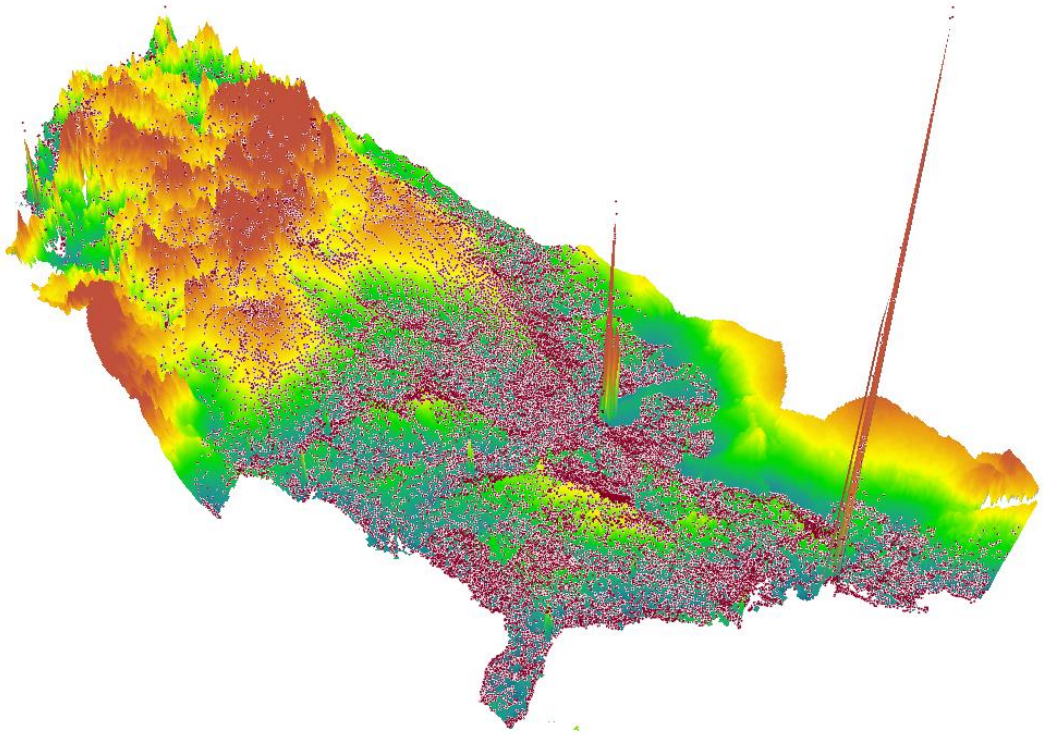


Figure 18. Illustrative caption of one of the intermediate products of the spatial analysis performed to generate the initial Status Map. We can notice two outliers due to low quality of the departure data.

We have simulated the planning stage (further explanations of this processes go beyond the scope of this project) and now we are ready to set up the layers in the server to make them available via WFS service, which will let us to start testing the prototype.

11 Setting up the web service

Initially we arrange access to the research server to set up the web service using advanced infrastructure, but the Erasmus period was over before finishing the tool and the analysis, hence we don't have access to it anymore, so we will set up a virtual environment to test the prototype. It will reduce computational capability and speed, but the functionality is identical.

The web service runs within a Virtual Box machine with a live Linux distribution inside specially focused on cartography and Geographic Information Systems (Open Source Geospatial Foundation 2013), which has easy to use interface to set up the web server (Apache) and the cartographic server (Geoserver), we had to try several Geoserver versions due to incompatibility problems.

Since the project is developed in MSVS2008 we had to connect to that virtual machine from "outside" the sandbox, using a virtual net connector created by the Virtual Machine software, having an internal IP address in Linux and also access to those IPs from windows system being able to debug meanwhile from MSVC++. We ping ourselves to be sure it is ready.

After that we open an admin session and create a new *workspace* to keep all layers grouped. After that we will add the *store,* which is a pointer to the location of the layers (in our case we distinguish between auxiliary data and the R-S map mechanism layers. Finally we add the layers in the right store, not forgetting update the spatial reference system (Geoserver has some problems auto detecting it from the *.prj file) nor calculating the spatial domain before adding them to the layer's list.

Once that step is done we have the layers ready to be queried, we can check directly the recently loaded using Open Layers and Web Map Service in the web browser (http://192.168.43.101:8082/geoserver/fran/wms?service=WMS&version=1.1.0&request=GetMap&layers=fran:WGS84_RSM_Zones_PriorityC_2D&styles=&bbox=-124.90002441419523,24.000122070177554,-67.90002441389547,49.000122070577284&width=752&height=330&srs=EPSG:4326&format=application/openlayers), obtain a *gml*[13] document in the web browser querying the service in WFS or use a GIS client to build and handle a WFS request and its resulting features.
Now the service is up and running, with all the spatial information uploaded and referenced, ready to use. The next step will be launching the prototype and start the battery of tests.

[13] GML is the acronym of Geographic Markup Language. A standard developed by the Open Geospatial Consortium which is a particular XML grammar for geographical features (OGC 2012).

Chapter 5: Preparing request maps

Along this chapter we are going to describe and explain the request map side of the mechanism. There won't be a specific task of creating request maps, they should be automatically created (in server side) depending on some specific parameters and provided to the volunteer through a web service.

Continuing with the phenology example the request map is the answer to a "*Where should I go and observe?*" volunteer question, and the answer is a list of suggested places (zones) the server considers suitable depending on such parameters.

This MSc Thesis is focused on the client side software for planner users in the request map mechanism, although we suggest a rough definition of other sides and user perspectives as starting point for further developments taking profit of the research work performed to sketch the full mechanism.

In order to answer the question the server will perform a set of operations to find the most suitable spots according to user preferences, habits and location. After that the server will generate a map (the request map) and send it to the user.

That answer is the result of a query generated automatically by the software after the user confirms using the request-side mechanism. This will be a web application or a mobile device application which will work as wizard.

12 Generating the request map

The request map is produced each time the user "asks" for locations and is generated *on-the-fly* at server side. In this example we suggest the following workflow:

12.1 Information used

12.1.1 Habits

The system will keep an historic of reports; hence it is possible to use that information to try to define patterns (i.e. user *X* likes going to Louisiana swamps to observe crocodiles and birds because last 6 of last 10 reports with different observation date were performed there and to those species), hence the first step to determine if there are patterns or not for user *X*. This operation could be done in advance, to reduce computational costs at request time, i.e. implementing pattern finding in standard recalculation procedures in users database. The patterns will be over-weighted to "redirect" some results to those locations and make sure the user is suggested to go to their favourite places (zones from the status map) but those ones actives with the higher priority level.

In order to do so we suggest generating a buffer map of several kilometres (real experimental records are needed in order to calibrate the distance of such buffer) and make a selection by location, selecting the zones within that buffer and ordering them by priority.

12.1.1.1 Preferences

The user also fulfils some data relative to his or her observation preferences. It can be used to establish thresholds in spatial searches (i.e. displacements closer than 50km), focus on specific kind (animals or plants), or even define exactly a set of interesting species to be observed. In the request application those fields will be automatically updated with saved user preferences, but they can be modified in the GUI to

customize the query with the personal preferences of the volunteer for that observation day (weather, mood, other volunteers, etc.). Examples are Figure 10 and Figure 11.

12.1.1.2 Location

Finally the server also will perform a buffer centred on user's location to find the most suitable places (logically a Louisiana volunteer probably won't observe in Alaska). Then the zones will be selected and ordered by priority. Unlike habits case this case is not optional.

12.2 Requesting a Request map

The next step is to perform the spatial analysis in the server side to offer the user the results, but it could be done in several ways. In one hand it is possible to implement the analysis by programming such operations in the server but it is hard, complex and time-consuming task. On the other hand it is possible to make use of Web Processing Services that will do that task for us in the cartographic server engine. This has some advantages (mainly versatility, the number of WPS operations defined and the OGC support because it is a standard).

Using the WPS also allows two different paths. We could directly ask for the layer resulting of concatenating the WPS operations (intersection, contains, buffer, etc.) and sorting using a weighted algorithm to consider the other preferences, however it could require WFS support and mobile devices usually are not capable of handling that service because of its complexity, hence it is better to create a temporary layer in the server (there won't be memory or storage problems because such layers will be very small and will contain basic information, meaning that the service will be capable of handling hundreds or thousands without much trouble) and concatenate a Web Map Service request against that temporary layer and retrieve the information using that other OGC web service.

It could be possible to use Web Coverage Service, which also works producing raster files (images) of cartographic layers, but it is slightly more complex service because allows editing such raster files and it is a functionality not to be used in the request perspective of the mechanism.

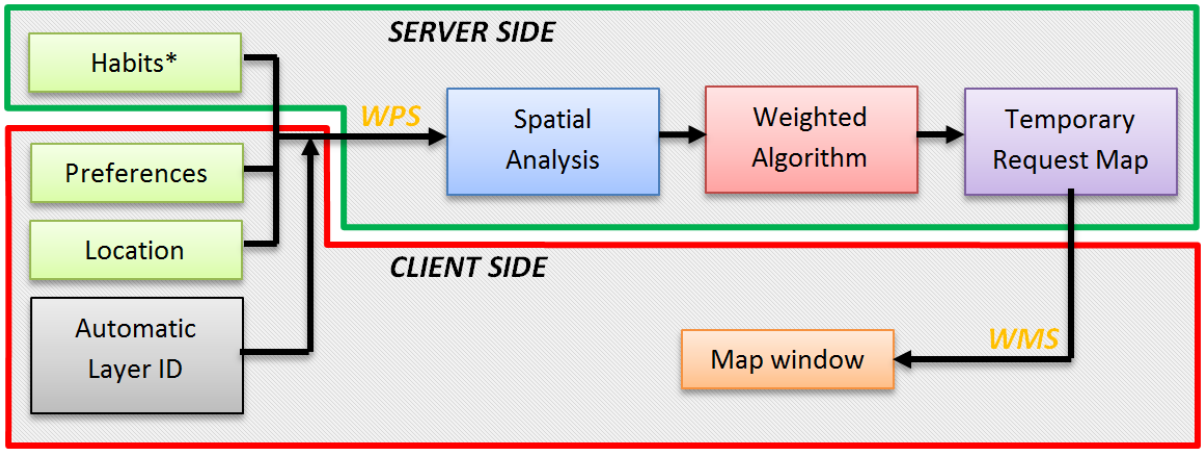


Figure 19. Schema of the processes context in a Request Map request.

12.3 Obtaining the Request Map

Web Map Service produces as standard output a geo-referenced image of the layers that compose the map. It is very flexible and allows us to specify the output style of the map, handling vector or raster layers (even layers from other WMS, WCS or WFS servers) using "*getMap*" operation.

WMS also allows the user click in a location in the image and query the attribute table of the feature using "*getFeatureInfo*" operation. It is possible to create a web application to handle such operations and achieve

a complete environment to query the map by the volunteer. A good example of the use of such WMS operations is the related MSc. Thesis done also in Netherlands mentioned in previous chapters, where using Javascript a full viewer has been prepared to connect to several VGI sources using web services.

There is a complete pre-set of tools that include scale viewer, zooms, pans, pointer coordinates, activate and deactivate layers among others, ready to implement in websites named *OpenLayers*. Openlayers is frequently used in cartographic viewers in light clients.

In case of heavy clients (or mobile applications) the developer will have to manually retrieve the geo-referenced image or other interesting variables like the numeric scale of the view and handle it internally to fit in the GUI.

That's why we recommend using a web application to ask for Request maps in the beginning as prototype and proof-of-concept. Using (i.e.) PHP or ASP rather than HTML because programming languages will allow customized formatting for map elements (i.e. round a scale value to an integer number instead of a floating number), more control of the final results and more professional and smooth appearance.

Although request maps also would allow the user to automatically retrieve the spatial information (layers) related to the selected zone in the request map once "accepted and reserved" for generating VGI. In case of phenology there is no point on it, but (i.e.) mapping parties could be very interesting implementing Request maps in heavy clients or advanced web applications and download the layers to be updated, but not the whole layer, only a section where the user is going to be redirected to control and enhance the VGI production.

Chapter 6: Results

13 Results obtained

As direct result of the work done during the ERASMUS stance and the posterior independent work the student has designed and built a working prototype as proof-of-concept of the R-S mechanism based on OGC Web Feature Service 1.1

The main purpose and structure of the R-S Map mechanism has been clarified and sketched, and there has been an intensive documentation process to justify the faces of the mechanism involved in this MSc. Thesis, proposing alternatives when possible. The full system has not been fully defined and developed; only the WFS-T part of it, but as consequence of the work done further developments associated to other MSc. Thesis or PhD. can finish the other sides of the mechanism, achieving a full working system based on the general definition generated here. A system generic enough to control VGI production or any other purpose involving WFS-T.

The prototype has been embedded in ILWIS software, and is capable of retrieving data from external OGC services and generate the internal ILWIS Domains, Coordinate systems, Maps, Tables and Features objects required to show the contents of remote servers in ILWIS. Although it can be embedded in any other C++ software or ported to other platforms.

Despite ILWIS software was not fully adapted to WFS-Transactional (at the moment of writing) the developed module is capable of handling the 3 operations of the WFS-T 1.1 protocol (create, delete and update feature), and it is possible to alter tables in remote servers using commands of the WFSConnector object.

Chapter 7: Conclusions

14 Answers to the research questions

14.1 Research question 1:

After devising one use case based on phenological observations we realized there is no point on force all the extra information to fit inside the Status Map, because it would require spatial intersections and it would generate an extremely large number of polygonal features combination of all other layers and it does not provide any real interesting benefit. Each particular application will manage its own data and particular attributes, but we encourage to keep attributes to identify features in the related database (ID's), pointers to users, temporal tagging (Creation, modification, last update date) and priority modifier (to recalculate the map in function of planning special requirements). A mandatory field is Status field, (active, inactive, idle, selected or other) to include such fields or not in the requests maps or recalculation routines.

14.2 Research question 2:

After the research stage for Status Maps we encourage to use vector features (polygons in this case), because it is a bit more versatile than raster structures, although for some operations to recalculate Status Maps some rasterizations are used eventually, but only for temporal intermediate stages. It is important to point out that could be possible to achieve similar results using raster datasets although in that case the Web Feature Service wouldn't be suitable, we would have used Web Coverage Service, also defined by the OGC.

For handling the Request Maps it is clearly easier to use non editable rasters, which is clearly terrain of Web Map Service (WMS).

14.3 Research question 3:

There is no point (at this stage) on prepare special routines for sectors with heterogeneous accuracy. The results of the Status Map are highly dependent on the quality of the initial data, and usually the free available data have not high standard quality (We have to keep in mind we are working in VGI). It could be possible to generalize and dissolve zones (or even sectors) but it would consume much more effort to determine (automatically) the quality of the data than skipping it and recomputed all the same way.

14.4 Research question 4:

Topology allows to keep geometric consistency in the Status Map layer, it is mandatory if the Status Map is going to be influenced by spatial analysis tasks (and it is going to happen in updating stages) to ensure unaltered shapes and avoid sliver polygons (which appears after chaining spatial analysis). It is optional to use topology in other layers that will be used to update the Status Map.

14.5 Research question 5:

The actual mechanism does not support topology, it will be implemented in future versions, although everything points to keep the topology in server side transparent to the user because it requires some knowledge that we cannot expect from all VGI users.

14.6 Research question 6:

Status maps will work using WFS, it is possible to use WCS but we consider WFS is slightly more suitable in this case.

14.7 Research question 7:

As commented before WFS for Status maps and WMS for Request Maps in mobile clients. It would be possible to use WFS also in desktop applications and in some special mobile devices, but it is like opening a nut with a sledge hammer. It is not practical and is much easier develop the Request part in using WMS in a web application. Only if the application expects to download automatically the involved information to be updated in the client we could use WFS-T. But in the phenology example is not suitable.

14.8 Research question 8:

There is a section to fully explain this question, actually we will use the existing interaction programmed in ILWIS, but we encourage updating this in future version to a much more intuitive interaction instead of the structured actual behaviour.

15 General considerations

After all the work done we consider there are a few ideas worth to point out about the prototype and the Request Status mechanism, which we have learnt during the development of the MSc. Thesis.

To make the prototype more versatile we will need to implement some functions to handle specific features that make the software crash in some specific contexts:

- UTF-8 encoding support is required. Nowadays only works with basic ASCII table characters.
- Full support for xml namespaces is recommendable. Although it won't provide extra functionality and this aspect is solved in the initial version we recommend to handle automatically this in next versions because otherwise "bad" formed documents are not accepted.
- The source code should be rewritten in order to implement the CURL object as a singleton element or function that provides a string or memory chunk with the http request. It would make the code "smoother" and more professional.
- Implement tags collections for WFS versions non implemented (we have to keep in mind the support is for version 1.1.0)
- Improve support for WFS 2.0.0

About the human-computer interaction there are also some remarkable points:

- It is mandatory to develop an easy-to-use graphical user interface as wizard to explore WFS services in ILWIS.
- Nowadays editing features with ILWIS is not as intuitive task as it should be; a new strategy is required in this case. We encourage a change of direction towards the editing interactions proposed in the MSc. Thesis.
- Pervasive computing can provide extra (automatic) functionality; particularly it could be interesting to consider ubiquity to automatically determine the user's context. (i. e. The dispositive could launch the right application "skin" -request zone mode or reporting mode- depending on the location). Location is not the only information we can use making use from pervasive computing world to improve the volunteer experience. The growing number of sensors in mobile devices increases dramatically the capabilities to extract information, and maybe that information could be applied to the mechanism. It would be too ambitious to get involved in this

field in this MSc. Thesis, but we consider this could be interesting for other works related to this field.

We have prepared a prototype to start setting up a real production software to handle VGI, but this is just the first milestone, there are many more and before continue developing it is important to evaluate somehow the capabilities of this prototype, and check if it improves or not the efficiency from the VGI point of view, comparing the existing production of geographic information to users equipped with this technology. It could be done, mainly with simulations in controlled environments because we still lack the mobile side of the mechanism.

Those simulations could contain a battery of tests to measure the efficiency of the client and the mechanism, not only from pure computing performance (for that purpose could be interesting use *profiling,* extended compilers to measure resources administration and code efficiency).

We have developed a mechanism based on WFS, even though the full Request-Status map mechanism is not forced to fit in that web service. The mobile side of the mechanism seems to fit well in Web Map Service (WMS) to show the request map in the mobile device through web application or mobile application, mainly because it is much more simple and easy to implement and low computing resources required.

Chapter 8: References

Aalders, HJGL, and Harold Moellering. "Spatial data infrastructure." *Proceedings of the 20 th International Cartographic Conference*. Beijing, China, 2001. 2234--2244.

Burrough, Peter A, Rachael McDonnell, Peter A Burrough, and Rachael McDonnell. *Principles of geographical information systems*. Vol. 333. Oxford university press Oxford, 1998.

Dietz, Cynthia. "Implementing geospatial web services: a resource webliography." *Issues in Science and Technology Librarianship*, no. 61 (2010): 6.

Federal Agency for Cartography and Geodesy. *International Earth Rotation and Reference Systems Service official website*. July 2013. http://www.iers.org/ (accessed 2013).

Foundation, Open Source Geospatial. "Geos Official Website." *Geos Official Website*. 2012.

Goodchild, Michael F. "Citizens as sensors: the world of volunteered geography." *GeoJournal* (Springer) 69, no. 4 (2007): 211-221.

Hardy, Darren. *Volunteered Geographic Information in Wikipedia*. ERIC, 2010.

Institut Géographique National (IGN France). *International Terrestrial Reference Frame website*. May 2012. http://itrf.ensg.ign.fr/ (accessed 2013).

McDougall, Kevin. "The potential of citizen volunteered spatial information for building SDI." *University of Southern Queensland, Australian Centre for Sustainable Catchments* (GSDI Association Press), 2009.

NACIS. *Natural Earth*. 2013. http://www.naturalearthdata.com/ (accessed 2013).

National Oceanic and Atmospheric Administration's (NOAA). *National Weather Service*. March 2012. http://www.nws.noaa.gov/geodata/ (accessed 2013).

Neis, Pascal, Peter Singler, and Alexander Zipf. "Collaborative mapping and Emergency Routing for Disaster Logistics-Case studies from the Haiti earthquake and the UN portal for Afrika." *Geospatial Crossroads@ GI_Forum* 10 (2010).

OGC. *Geography Markup Language Official Website - Open Geospatial Consortium*. 2012. http://www.opengeospatial.org/standards/gml (accessed 2013).

—. *Web Processing Service Official Website - Open Geospatial Consortium*. 2012. http://www.opengeospatial.org/standards/wps (accessed 2013).

Open Source Geospatial Foundation. *OSGeo official website*. August 2013. http://www.osgeo.org/ (accessed 2013).

Oracle. "Oracle Spatial and Graph Official Website." *Oracle Spatial and Graph Official Website*. 2012.

Research, Refractions. "Postgis Official Website." *Postgis Official Website*. 2012.

Ricker, Britta A, Peter A Johnson, and Renee E Sieber. "Tourism and environmental change in Barbados: gathering citizen perspectives with volunteered geographic information (VGI)." *Journal of Sustainable Tourism* (Taylor \& Francis) 21, no. 2 (2013): 212-228.

Silva, Josep Francesc. "Técnicas de depuración tradicionales." Universidad Politécnica de Valencia, 2013.

Solutions, Vivid. "Jts Official Website." *Jts Official Website*. 2012.

U.S. Geological Survey. *National Atlas*. July 2013. http://www.nationalatlas.gov/atlasftp.html (accessed 2013).

Weichand, J. "Geoinformatik blog." *Geoinformatik blog*. http://www.weichand.de/2012/06/07/wfs-2-0-client-plugin-for-qgis-english/, September 2012.

Wiki, OpenStreetMap. "OSM Tasking Manager." *OpenStreetMap Wiki*. 2012.

Chapter 9: Annexes

16 Code files

```
//
//
//  MSc. Thesis: Optimizing VGI production using status maps requested through web services
//
//  Official Master's Degree in Software Engineering, Formal Methods and Information Systems
//  Polytechnic University of Valencia, Spain (UPV)
//  Faculty of Geo-Information Science and Earth Observation, The Netherlands (ITC)
//
//  Supervisor UPV: Vicente Pelechano
//  Supervisor ITC: Rob Lemmens
//
//  ---------------------------------------------------------------------------------------
//
//  @ Project : WFS Module for ILWIS
//  @ File Name : ConnectorWFS.hpp
//  @ Date : 30/10/2012
//  @ Author : Francisco Ruiz-Lopez
//
//  ---------------------------------------------------------------------------------------



#if !defined(_CONNECTORWFS_H)
#define _CONNECTORWFS_H

#include <iostream>
#include <string.h>
#include <vector>
#include "Log.hpp"

class ConnectorWFS {
public:
     ConnectorWFS(std::string ip, std::string port, std::string servicePath);
     ConnectorWFS(std::string fullAddress);

     // This function calls getCapabilities (To get an updated answer) and answers a list with the NAMES (not the
     titles).
     std::vector<std::string> ConnectorWFS::getListLayerNames();
     // Retrieve the attribute names of a layer (using the NAME of the layer).
     std::vector<std::string> ConnectorWFS::getAttributeNamesList(std::string layerName);
     // Retrieve the data type of an attribute
     std::string ConnectorWFS::getAttributeType(std::string attributeName);
     // Provides a vector of vectors with the geometries of the layer and its attributes
     std::vector<std::vector<std::string>> getFeaturesFromLayer(std::string layerName, std::string version = "1.1.0"
     , std::string filter = "");

     // Provides the LAYER TITLE
     std::string ConnectorWFS::getLayerTitle(std::string layerName);
     // Provides the LAYER ABSTRACT (If exists, otherwise returns "").
     std::string ConnectorWFS::getLayerAbstract(std::string layerName);
     // Searches in the attribute table for the column that contains the geometry
     std::string ConnectorWFS::getGeometryColumn(std::string layerName);
     // Search for the primitive geometry of the layer
     std::string ConnectorWFS::getGeometryType(std::string layerName);
     // Get the EPSG Code for one layer (The default EPSG Code obtained from capabilities document) THE SAME THAN THE
     SERVER PRODUCES
     std::string ConnectorWFS::getLayerSRS_FullEPSG(std::string layerName);
     // Get the EPSG Code for one layer (The default EPSG Code obtained from capabilities document) ONLY THE NUMERIC
     CODE!
```

```
    std::string ConnectorWFS::getLayerSRS_OnlyEPSGCode(std::string layerName);
    // Get the BoundingBox for one layer (As defined in the Capabilities document)
    std::vector<double> ConnectorWFS::getLayerBoundingBox(std::string layerName);

    int ConnectorWFS::executeTransactionDelete(std::string layerName, std::vector<std::string> vector_gmlIds, std::
    string service = "WFS", std::string version = "1.1.0"); //Return 0 = OK; Return -1 = ERROR;
    int ConnectorWFS::executeTransactionDelete(std::string layerName, std::string str_field, std::string str_value,
     std::string service = "WFS", std::string version = "1.1.0"); //Return 0 = OK; Return -1 = ERROR;
    int ConnectorWFS::executeTransactionInsert(std::string layerName, std::vector<std::vector<std::string>>
    v_attributes, std::string service = "WFS", std::string version = "1.1.0"); //Return 0 = OK; Return -1 = ERROR;
    // Update transaction only updates FIELDS, not geometries (yet).
    int ConnectorWFS::executeTransactionUpdate(std::string layerName, std::string field, std::string value, std::
    vector<std::string> vector_gmlIds = std::vector<std::string>(), std::string service = "WFS", std::string
    version = "1.1.0"); //Return 0 = OK; Return -1 = ERROR; //If there's no gmlId values the changes will be
    applied to all layer.

private:
    ~ConnectorWFS();
    int ConnectorWFS::getCapabilities(); // Return 0 = OK; Return -1 = ERROR;
    // The DescribeFeatureType operation returns a schema description of feature types offered by a WFS instance.
    // The schema descriptions define how a WFS expects feature instances to be encoded on input (via Insert, Update
    and Replace actions) and how feature instances shall be encoded on output (in response to a  GetPropertyValue,
    GetFeature or GetFeatureWithLock operation).
    void ConnectorWFS::describeFeatureType(std::string featureTypeName = "NAMESPACE:FEATURETYPENAME", std::string
    service = "WFS", std::string version = "1.1.0", std::string handle = "");
    // The GetFeature operation returns a selection of features from a data store. A WFS processes a GetFeature
    request and returns a response document to the client that contains zero or more feature instances that satisfy
    the query expressions specified in the request.
    std::string ConnectorWFS::getFeature(std::string featureTypeName, std::string version = "1.1.0", std::string
    queryFilter = "");
    // Get the URL of the namespace for the layer returns "no-url-found" value if not found, IT IS MANDATORY FOR
    INSERT/UPDATE TRANSACTIONS, BUT IF NO WFS TRANSACTIONAL there's no need to control errors.
    std::string ConnectorWFS::getLayerNameSpaceURL(std::string layerName);

    // Initialize variables and environment for the basic capabilities operations and retrieving features.
    int ConnectorWFS::initializeConnector(); //Return 0 = OK; Return -1 = ERROR;

    // Tags and variables for WFS 1.1.0
    int ConnectorWFS::initializeFieldsCapabilities(); //Tags and variables. //Return 0 = OK; Return -1 = ERROR;

};

#endif  //_CONNECTORWFS_H
```

```
//
//
//  MSc. Thesis: Optimizing VGI production using status maps requested through web services
//
//  Official Master's Degree in Software Engineering, Formal Methods and Information Systems
//  Polytechnic University of Valencia, Spain (UPV)
//  Faculty of Geo-Information Science and Earth Observation, The Netherlands (ITC)
//
//  Supervisor UPV: Vicente Pelechano
//  Supervisor ITC: Rob Lemmens
//
//  ---------------------------------------------------------------------------------------
//
//  @ Project : WFS Module for ILWIS
//  @ File Name : ConnectorWFS.cpp
//  @ Date : 30/10/2012
//  @ Author : Francisco Ruiz-Lopez
//
//  ---------------------------------------------------------------------------------------

#include <iostream>
#include <sstream>
#include <string.h>
#include <algorithm> //Used for transform to upper/lowercase
#include "ConnectorWFS.hpp"

//These includes only work if I use the library by myself, ILWIS compiles also those libraries so if I use them there
will be a name conflict.
//#include <pugixml/pugixml.hpp>   //pugi version = 1.2
//#include <curl/curl.h>
#include "Engine/Base/XML/pugixml.hpp"
#include "Engine/DataExchange/curlIncludes/curl.h"

using namespace std;

//Try to create this on the heap instead of the stack (Is it actually created on the stack?)
struct MemoryStruct
{
     char *memory;
     size_t size;
};

//Try to create this on the heap instead of the stack (Is it actually created on the stack?)
static size_t WriteMemoryCallback(void *contents, size_t size, size_t nmemb, void *userp)
{
     size_t realsize = size *nmemb;
     struct MemoryStruct *mem = (struct MemoryStruct *)userp;

     mem->memory = (char*)realloc(mem->memory, mem->size + realsize + 1);
     if (mem->memory == NULL)
     {
          // out of memory!
          printf("not enough memory (realloc returned NULL)\n");
          exit(EXIT_FAILURE);
     }

     memcpy(&(mem->memory[mem->size]), contents, realsize);
     mem->size += realsize;
```

```
    mem->memory[mem->size] = 0;

    //std::cout << "Calling writeMemoryCallback" << std::endl;
    return realsize;
}

// Class Variables:[defined in HEADER FILE!]
std::string serverURL ="";                  // Variable with the main service URL
std::vector<std::string> tagsWFS110;    // Vector with all the tags from WFS 1.1.0
//************************************
// Variables for GetCapabilities Operation
//************************************
//************************************ WFS Connector attributes [Service Details]
std::string serviceTitle               ="";
std::string serviceAbstract            ="";
std::string serviceType                ="";
std::string serviceTypeVersion        ="";
std::string serviceFees                ="";
std::string serviceAccessConstraints ="";
std::vector<std::vector<std::string>> v_serviceNSurls; //Vector to store all the URL addresses of the namespaces in
Geosever.
//************************************ WFS Connector attributes [Service Provider]
std::string serviceProviderName                    ="";
std::string serviceProviderContactIndividualName ="";
std::string serviceProviderContactPositionName    ="";
//************************************ WFS Connector attributes [Operations Metadata] {Supported operations}
std::vector<std::string> v_serviceOperationsMetadataOperations;
//************************************ WFS Connector attributes [Feature Type List]   {List of available layers}
std::vector<std::string> v_serviceFeatureTypeListOperations;     //Not mandatory, could be size 0
std::vector<std::string> v_serviceFeatureTypeListFeatureTypes;   //Here we point to the NAME child node of each
feature!
//************************************ WFS Connector attributes [Filter_Capabilities]
std::vector<std::string> v_serviceFilterSpatialGeometryOperands; //Mandatory
std::vector<std::string> v_serviceFilterSpatialSpatialOperators; //Mandatory
std::vector<std::string> v_serviceFilterScalarLogicalOperators;
std::vector<std::string> v_serviceFilterScalarComparisonOperators;
std::vector<std::string> v_serviceFilterScalarArithmeticOperatorsSimpleAritmetic;
std::vector<std::string> v_serviceFilterScalarArithmeticOperatorsFunctions;
//std::vector<std::string> v_serviceFilterScalarId;
//************************************ Layer's list
std::vector<std::vector<std::string>> vLayers;         // [NAME (id), TITLE (Descriptive title), ABSTRACT,
DEFAULT_SRS, BBox{LOWERcornerLON, LOWERcornerLAT, UPPERcornerLON, UPPERcornerLAT}]
//************************************
// Variables for DescribeFeatureType Operation
//************************************
std::vector< std::vector<std::string> > v_AttributesLayer; //ATTRIBUTE[ NAME, TYPE, NULLABLE*, MINOCC*, MAXOCC*]
*Optionals

//Define the log as singleton element for all the class.
Log *logWFS = Log::Instance();
std::stringstream logSS; // StringStream to redirect complex string combinations to Log File, it is easy and safe.
Don't forget to flush!


ConnectorWFS::ConnectorWFS(string ip, string port, string servicePath) {
    string aux = "http://";
    aux.append(ip).append(":").append(port).append(servicePath);
    logWFS->printToLog(std::string("Creating WFS Connector with address: ").append(aux));
```

```
	ConnectorWFS *c_01 = new ConnectorWFS(aux); //Use the next constructor
}

ConnectorWFS::ConnectorWFS(string fullAddress) {
	//The address MUST end with "?"
	if(fullAddress[fullAddress.size()-1]!='?')
	{
		fullAddress.append("?");
		logWFS->printToLog("The address should end with '?' character to ensure full operational capabilities");
		logWFS->printToLog("Address updated: " + fullAddress);
	}
	serverURL = fullAddress;
	// Initialize the variables for further processes:
	if (initializeConnector()==-1){
		logWFS->printToLog("CRITICAL ERROR INITIALIZING. Ending the module.");
		exit(EXIT_FAILURE);
	}
}

ConnectorWFS::~ConnectorWFS() {
	logWFS->printToLog("Destroying object ConnectorWFS...");
}

int ConnectorWFS::getCapabilities() {
	//Retrieve the relevant details about the Web Service to manage further operations.
	//DETAILS
	int status = -1;

	//BUILD THE URL GET  "http://localhost:8080/geoserver/ows?service=wfs&version=1.1.0&request=GetCapabilities"
	std::string url_GetCapabilities = serverURL;
	url_GetCapabilities.append("SERVICE=WFS&VERSION=1.1.0");              // Service + Version
	url_GetCapabilities.append("&REQUEST=GETCAPABILITIES");                    // Request

	MemoryStruct chunk;
	chunk.memory = (char*)malloc(1);  // will be grown as needed by the realloc above
	chunk.size = 0;                              // no data at this point

	CURL *curl_handle;
	curl_global_init(CURL_GLOBAL_ALL);
	//init the curl session
	curl_handle = curl_easy_init();
	//specify URL to get
	curl_easy_setopt(curl_handle, CURLOPT_URL, url_GetCapabilities.c_str());
	//send all data to this function
	curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, WriteMemoryCallback);
	//we pass our 'chunk' struct to the callback function
	curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk);
	//some servers don't like requests that are made without a user-agent field, so we provide one
	curl_easy_setopt(curl_handle, CURLOPT_USERAGENT, "libcurl-agent/1.0");
	//get it!
	CURLcode err =curl_easy_perform(curl_handle);
	//cleanup curl stuff
	curl_easy_cleanup(curl_handle);
	//we're done with libcurl, so clean it up
	curl_global_cleanup();

	logSS << "GetCapabilities requested, " << (long)chunk.size/1024 << " Kb retrieved.";
	logWFS->printToLog(logSS.str());
```

```
    logSS.str(""); //Clearing the strinStream.
    logSS.flush(); //Synchronizing the buffer

    //logWFS->printToLog(chunk.memory);
    if ((long)chunk.size == 0)
    {
         logWFS->printToLog("ERROR : No data retrieved! Check the service address");
         //Destroy the element. The user should create another object and try to connect. To prevent the String error.
         return status; // Exit from the function, is void, so there's no need to specify value.
    }



    //*********************************************************************************************
    // Before using the Xpath Search functions we should homogeneize the tags from the namespaces!
    // By reading the document and adding the required ones if this exists:
    // All tags that use as default namespace WFS namespace will be added as tags form that namespace. Add also the
    namespaces avoiding dupplicates.

    std::string cap_doc = "";
    cap_doc = chunk.memory;
    free(chunk.memory); //We are not gonna use the original chunk data anymore.
    int index = -2;
    std::string aux_nameSpaceWfs = "xmlns=\"http://www.opengis.net/wfs\"";
    index = cap_doc.find(aux_nameSpaceWfs,0); //index = -1 if there's no coincidence.

    //ATTENTION!: Besides updating the tags we have to add the namespace definition, otherwise we get an error!
    //Avoiding duplicate WFS namespace!! add it only if it does not exsist!!!
    int aux_indexNS = -1;
    aux_indexNS = cap_doc.find("xmlns:wfs=\"http://www.opengis.net/wfs\"",0);
    if (aux_indexNS == -1)
    {
         cap_doc.replace(index, aux_nameSpaceWfs.size(), aux_nameSpaceWfs+"
         xmlns:wfs=\"http://www.opengis.net/wfs\"");
    }
    //std::cout << "Indice: " << index << std::endl;

    pugi::xml_document doc;
    pugi::xml_parse_result result = doc.load(cap_doc.c_str());

    if (index !=-1){ //If the default namespace is wfs then add to all tags without ns "wfs:"
         for ( int i = 0 ; i<tagsWFS110.size(); i++ ){ //Loop to explore all the elements in WFS Schema.
              pugi::xpath_node_set ns;
              std::string expression = std::string("//").append(tagsWFS110[i]);
              ns = doc.select_nodes(expression.c_str()); //We select all the nodes with that exactly name and
              replace them for the wfs:nodename element.
              for(pugi::xpath_node_set::const_iterator it = ns.begin(); it!=ns.end(); ++it){
                   pugi::xpath_node n = *it;
                   std::string newTag = string("wfs:").append(tagsWFS110[i]);
                   n.node().set_name(newTag.c_str());
              }
         }
    }



    // Service Details
    //
    ***********************************************************************************************
```

```
pugi::xpath_node capab_node;
pugi::xpath_node_set capab_node_set;
capab_node = doc.select_single_node("/wfs:WFS_Capabilities/ows:ServiceIdentification/ows:Title");
serviceTitle=capab_node.node().child_value();
capab_node = doc.select_single_node("/wfs:WFS_Capabilities/ows:ServiceIdentification/ows:Abstract");
serviceAbstract=capab_node.node().child_value();
capab_node = doc.select_single_node("/wfs:WFS_Capabilities/ows:ServiceIdentification/ows:ServiceType");
serviceType=capab_node.node().child_value();
capab_node = doc.select_single_node("/wfs:WFS_Capabilities/ows:ServiceIdentification/ows:ServiceTypeVersion");
serviceTypeVersion=capab_node.node().child_value();
capab_node = doc.select_single_node("/wfs:WFS_Capabilities/ows:ServiceIdentification/ows:Fees");
serviceFees=capab_node.node().child_value();
capab_node = doc.select_single_node("/wfs:WFS_Capabilities/ows:ServiceIdentification/ows:AccessConstraints");
serviceAccessConstraints=capab_node.node().child_value();
capab_node =  doc.select_single_node("/wfs:WFS_Capabilities");
for (pugi::xml_attribute_iterator att_it = capab_node.node().attributes_begin(); att_it!=capab_node.node().
attributes_end(); ++att_it)
{
     pugi::xml_attribute at = *att_it;
     std::vector<std::string> v_aux;
     std::string att_name = at.name();
     std::string att_val = at.value();
     size_t found = std::string::npos;
     found=att_name.find("xmlns:");
     if (found!=std::string::npos){
          v_aux.push_back(att_name);
          v_aux.push_back(att_val);
          v_serviceNSurls.push_back(v_aux);
     }
}


// Service Provider
//
*****************************************************************************************************************
capab_node = doc.select_single_node(
"/wfs:WFS_Capabilities/ows:ServiceProvider/ows:ServiceContact/ows:IndividualName");
serviceProviderName=capab_node.node().child_value();
capab_node = doc.select_single_node(
"/wfs:WFS_Capabilities/ows:ServiceProvider/ows:ServiceContact/ows:PositionName");
serviceProviderContactIndividualName=capab_node.node().child_value();


// Operations Metadata {Supported operations}
//
*****************************************************************************************************************
capab_node_set = doc.select_nodes("/wfs:WFS_Capabilities/ows:OperationsMetadata/*");
for (pugi::xpath_node_set::const_iterator it = capab_node_set.begin(); it != capab_node_set.end(); ++it)
{
     pugi::xpath_node node = *it;
     v_serviceOperationsMetadataOperations.push_back(node.node().attribute("name").value());
}

// Feature Type List  {List of available layers}
//
*****************************************************************************************************************
capab_node_set = doc.select_nodes("/wfs:WFS_Capabilities/wfs:FeatureTypeList/wfs:Operations/*");
for (pugi::xpath_node_set::const_iterator it = capab_node_set.begin(); it != capab_node_set.end(); ++it)
```

```
{
     pugi::xpath_node node = *it;
     v_serviceFeatureTypeListOperations.push_back(node.node().child_value());
}
capab_node_set = doc.select_nodes("/wfs:WFS_Capabilities/wfs:FeatureTypeList/wfs:FeatureType");
for (pugi::xpath_node_set::const_iterator it = capab_node_set.begin(); it != capab_node_set.end(); ++it)
{
     pugi::xpath_node node = *it;
     v_serviceFeatureTypeListFeatureTypes.push_back(node.node().child("wfs:Name").child_value());
}


// WFS Connector attributes [Filter_Capabilities]
//
*****************************************************************************************************************

capab_node_set = doc.select_nodes(
"/wfs:WFS_Capabilities/ogc:Filter_Capabilities/ogc:Spatial_Capabilities/ogc:GeometryOperands/*");
for (pugi::xpath_node_set::const_iterator it = capab_node_set.begin(); it != capab_node_set.end(); ++it)
{
     pugi::xpath_node node = *it;
     v_serviceFilterSpatialGeometryOperands.push_back(node.node().child_value());
}
capab_node_set = doc.select_nodes(
"/wfs:WFS_Capabilities/ogc:Filter_Capabilities/ogc:Spatial_Capabilities/ogc:SpatialOperators/*");
for (pugi::xpath_node_set::const_iterator it = capab_node_set.begin(); it != capab_node_set.end(); ++it)
{
     pugi::xpath_node node = *it;
     v_serviceFilterSpatialSpatialOperators.push_back(node.node().attribute("name").value());
}
capab_node_set = doc.select_nodes(
"/wfs:WFS_Capabilities/ogc:Filter_Capabilities/ogc:Scalar_Capabilities/ogc:LogicalOperators/*");
for (pugi::xpath_node_set::const_iterator it = capab_node_set.begin(); it != capab_node_set.end(); ++it)
{
     pugi::xpath_node node = *it;
     v_serviceFilterScalarLogicalOperators.push_back(node.node().child_value());
}
capab_node_set = doc.select_nodes(
"/wfs:WFS_Capabilities/ogc:Filter_Capabilities/ogc:Scalar_Capabilities/ogc:ComparisonOperators/*");
for (pugi::xpath_node_set::const_iterator it = capab_node_set.begin(); it != capab_node_set.end(); ++it)
{
     pugi::xpath_node node = *it;
     v_serviceFilterScalarComparisonOperators.push_back(node.node().child_value());
}
capab_node_set = doc.select_nodes(
"/wfs:WFS_Capabilities/ogc:Filter_Capabilities/ogc:Scalar_Capabilities/ogc:ArithmeticOperators/ogc:SimpleArithmeti
c/*");
for (pugi::xpath_node_set::const_iterator it = capab_node_set.begin(); it != capab_node_set.end(); ++it)
{
     pugi::xpath_node node = *it;
     v_serviceFilterScalarArithmeticOperatorsSimpleAritmetic.push_back(node.node().child_value());
}
capab_node_set = doc.select_nodes(
"/wfs:WFS_Capabilities/ogc:Filter_Capabilities/ogc:Scalar_Capabilities/ogc:ArithmeticOperators/ogc:Functions/ogc:F
unctionNames/*"); //Here we retrieve ONLY the names, but we could also retrieve (from the attributes of the
node) the number of parameters required.
for (pugi::xpath_node_set::const_iterator it = capab_node_set.begin(); it != capab_node_set.end(); ++it)
{
```

```
            pugi::xpath_node node = *it;
            v_serviceFilterScalarArithmeticOperatorsFunctions.push_back(node.node().child_value());
        }


        // Prepare the layer's list.  [NAME (id), TITLE (Descriptive title), ABSTRACT, DEFAULT_SRS, BBox{LOWERcornerLON,
        LOWERcornerLAT, UPPERcornerLON, UPPERcornerLAT}]
        vLayers.clear();
        //capab_node_set = doc.select_nodes("/wfs:WFS_Capabilities/wfs:FeatureTypeList/wfs:FeatureType");
        capab_node_set = doc.select_nodes("//wfs:FeatureType");
        std::string aux_lowerCorner = "";
        std::string aux_upperCorner = "";
        size_t pos_lower=0;
        size_t pos_upper=0;
        for (pugi::xpath_node_set::const_iterator it = capab_node_set.begin(); it != capab_node_set.end(); ++it)
        {
            pugi::xpath_node node = *it;
            std::vector<std::string> aux_vect;
            aux_vect.push_back(node.node().child("wfs:Name").child_value());
            aux_vect.push_back(node.node().child("wfs:Title").child_value());
            aux_vect.push_back(node.node().child("wfs:Abstract").child_value());
            aux_vect.push_back(node.node().child("wfs:DefaultSRS").child_value());
            pugi::xpath_node n1 = node.node().select_single_node("ows:WGS84BoundingBox/ows:LowerCorner |
            WGS84BoundingBox/LowerCorner");
            pugi::xpath_node n2 = node.node().select_single_node("ows:WGS84BoundingBox/ows:UpperCorner |
            WGS84BoundingBox/UpperCorner");
            aux_lowerCorner = n1.node().child_value();
            aux_upperCorner = n2.node().child_value();
            pos_lower = aux_lowerCorner.find(" "); //In the list the coordinates are split by " ".
            pos_upper = aux_upperCorner.find(" "); //In the list the coordinates are split by " ".
            aux_vect.push_back(aux_lowerCorner.substr(0,pos_lower)); //LOWER CORNER LONGITUDE
            aux_vect.push_back(aux_lowerCorner.substr(pos_lower));    //LOWER CORNER LATITUDE
            aux_vect.push_back(aux_upperCorner.substr(0,pos_upper)); //UPPER CORNER LONGITUDE
            aux_vect.push_back(aux_upperCorner.substr(pos_upper));    //UPPER CORNER LATITUDE
            vLayers.push_back(aux_vect);
        }
        // SUMMARIZE THE LAYER LIST
        //for (int i=0; i< vLayers.size(); i++) { logWFS->printToLog("Layer",vLayers[i]);    }

        // Initial value for status = -1, If there's no error it's changed and returned as 0.
        status = 0; // Status = 0 means OK, Status = -1 Means Error
        return status;
}


// ConnectorWFS::describeFeatureType DEPRECATED.
void ConnectorWFS::describeFeatureType(string featureTypeName, string service, string version, string handle) {
        logWFS->printToLog("Executing DescribeFeatureType request...");

        // BUILD THE URL GET
        std::string url_DescribeFeatureType = serverURL;
        url_DescribeFeatureType.append("SERVICE=").append(service);
        url_DescribeFeatureType.append("&VERSION=").append(version);
        url_DescribeFeatureType.append("&REQUEST=DESCRIBEFEATURETYPE");            // Request
        url_DescribeFeatureType.append("&TYPENAME=");                                   // Optional parameter, where we
        define the name of the schema.
        url_DescribeFeatureType.append(featureTypeName);                           // LAYER TO BE DESCRIBED!
```

```
//url_DescribeFeatureType="http://localhost:8080/geoserver/wfs?request=DescribeFeatureType&version=1.1.0&service=WFS&typeName=topp:states";


// RETRIEVE THE INFORMATION
MemoryStruct chunk;
chunk.memory = (char*)malloc(1);  // will be grown as needed by the realloc above
chunk.size = 0;     // no data at this point

CURL *curl_handle;
curl_global_init(CURL_GLOBAL_ALL);
curl_handle = curl_easy_init();
curl_easy_setopt(curl_handle, CURLOPT_URL, url_DescribeFeatureType.c_str());
curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, WriteMemoryCallback);
curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk);
curl_easy_setopt(curl_handle, CURLOPT_USERAGENT, "libcurl-agent/1.0");
curl_easy_perform(curl_handle); //This is the function that makes the process slower.
curl_easy_cleanup(curl_handle);
curl_global_cleanup();


// RETRIEVE THE SCHEMA OF THE LAYER
std::string layer_schema_doc = "";
layer_schema_doc = chunk.memory;

//Let's see if the tag exists:
int index_1 = layer_schema_doc.find("xsd:complexType name=",0); //index = -1 if there's no coincidence.
//Control errors (The service might use namespace xsd (xmlns:xsd="http://www.w3.org/2001/XMLSchema"), in this case we should find the tag (complexType name=)
//If we have to look for the tag withouth xsd namespace we will put it in a variable for preventing error tag not found on next stage.
int aux_useNS = -1;
if (index_1==-1 && layer_schema_doc.find("xmlns:xsd=\"http://www.w3.org/2001/XMLSchema")!=-1){
     index_1 = layer_schema_doc.find("complexType name=");
     aux_useNS = 0;
}
if (index_1 == -1)
{
     logWFS->printToLog("ERROR. THE RETRIEVED ANSWER DOESN'T FIT THE EXPECTED SCHEMA.");
     //If the server does not work well it might be interesting. But if the service is OGC compilant there's no need to do it.

}else
{
     logSS << "DescribeFeatureType requested, " << (long)chunk.size/1024 << " Kb retrieved.";
     logWFS->printToLog(logSS.str());
     logSS.str(""); //Clearing the strinStream.
     logSS.flush(); //Synchronizing the buffer
     pugi::xml_document doc_dft;
     pugi::xml_parse_result result = doc_dft.load_buffer(chunk.memory, chunk.size); //We can access directly to memory. No reparsing needed.
     free(chunk.memory);

     pugi::xpath_node_set dft_node_set;
     if (aux_useNS==-1){
          dft_node_set = doc_dft.select_nodes("//xsd:sequence/*");
     }else{
          dft_node_set = doc_dft.select_nodes("//sequence/*");
```

```
        }
        //std::cout << "ATTRIBUTE NUMBER OF COLUMNS: " << dft_node_set.size() << std::endl;

        //Reset the vector:
        v_AttributesLayer.clear();
        for (pugi::xpath_node_set::const_iterator it = dft_node_set.begin(); it != dft_node_set.end(); ++it)
        {
            pugi::xpath_node dft_node = *it;
            std::vector<std::string> aux_vectAtt;
            //Add attributes to the auxiliar vector.
            std::string aux_attName = "";
            std::string aux_atttype = "";
            std::string aux_attNull = "";
            std::string aux_attMaxO = "";
            std::string aux_attMinO = "";
            aux_attName = dft_node.node().attribute("name").value();
            aux_atttype = dft_node.node().attribute("type").value();
            aux_attNull = dft_node.node().attribute("nillable").value();
            aux_attMaxO = dft_node.node().attribute("maxOccurs").value();
            aux_attMinO = dft_node.node().attribute("minOccurs").value();
            //std::cout << "" << aux_attName << "   \t | TYPE: \t" << aux_atttype << " | NULL   " << aux_attNull
            << " | MAX   " << aux_attMaxO << " | MIN   " << aux_attMinO << std::endl;

            aux_vectAtt.push_back(aux_attName);
            aux_vectAtt.push_back(aux_atttype);
            aux_vectAtt.push_back(aux_attNull);
            aux_vectAtt.push_back(aux_attMaxO);
            aux_vectAtt.push_back(aux_attMinO);

            //Add the auxuliar vector to the main vector of Attributes.
            v_AttributesLayer.push_back(aux_vectAtt);
        }
    }
    logWFS->printToLog("DescribeFeatureType request executed.");
}


//Retrieve features from a layer.
std::string ConnectorWFS::getFeature(string featureTypeName, string version, string queryFilter) {
    std::string aux_response = "";
    // BUILD THE URL GET
    std::string url_GetFeature = serverURL;
    url_GetFeature.append("SERVICE=WFS");
    url_GetFeature.append("&VERSION=").append(version);
    url_GetFeature.append("&REQUEST=GETFEATURE");                                    // Request
    url_GetFeature.append("&TYPENAME=");                                             // Optional parameter, where we
    define the name of the schema.
    url_GetFeature.append(featureTypeName);                                          // LAYER TO BE DESCRIBED!
    if (queryFilter!=""){url_GetFeature.append("&").append(queryFilter);} //In case there is filter we add it.

    // RETRIEVE THE INFORMATION
    MemoryStruct chunk;
    chunk.memory = (char*)malloc(1);  // will be grown as needed by the realloc above
    chunk.size = 0;     // no data at this point

    CURL *curl_handle;
    curl_global_init(CURL_GLOBAL_ALL);
    curl_handle = curl_easy_init();
```

```
	curl_easy_setopt(curl_handle, CURLOPT_URL, url_GetFeature.c_str());
	curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, WriteMemoryCallback);
	curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk);
	curl_easy_setopt(curl_handle, CURLOPT_USERAGENT, "libcurl-agent/1.0");
	curl_easy_perform(curl_handle);
	curl_easy_cleanup(curl_handle);
	curl_global_cleanup();



	logSS << "Retrieved " << chunk.size << " bytes.";
	logWFS->printToLog(logSS.str());
	logSS.str("");
	logSS << "Retrieved features from [" << featureTypeName << "]: " << url_GetFeature;
	logWFS->printToLog(logSS.str());
	logSS.str("");

	// RETRIEVE THE SELECTED FEATURES
	std::string features_doc = "";
	features_doc = chunk.memory;

	//TODO: We could implement filter in future versions to extract a part of the full layer.

	aux_response = features_doc;

	if(aux_response==""){
		logWFS->printToLog("ERROR. The request didn't produce results");
	}
	return aux_response;
}


//Retrieving layer's list:
std::vector<std::string> ConnectorWFS::getListLayerNames(){
	logWFS->printToLog("Retrieving layers...");
	std::vector<std::string> aux_answer;
	//getCapabilities(); //To have up-to-date data. It makes the process much slower, for now we will use
	getCapabilities just creating the object.

	for (int i=0; i<vLayers.size(); i++){
		aux_answer.push_back(vLayers[i][0]); //Retrieve the name.
	}
	logWFS->printToLog("Layers retrieved.");
	//logWFS->printToLog("LAYERS NAMES: ", aux_answer);
	return aux_answer;
}


//Retrieving attributes list
std::vector<std::string> ConnectorWFS::getAttributeNamesList(std::string layerName){
	logWFS->printToLog("Retrieving attribute names...");
	std::vector<std::string> aux_answer;
	//getCapabilities(); //To have up-to-date data. It makes the process much slower, for now we will use
	getCapabilities just creating the object.
	describeFeatureType(layerName);

	for (int i=0; i<v_AttributesLayer.size(); i++){
		aux_answer.push_back(v_AttributesLayer[i][0]);
	}
```

```
	logWFS->printToLog("Attribute names retrieved.");
	//logWFS->printToLog("ATTRIBUTE NAMES: ", aux_answer);
	return aux_answer;
}


//Retrieving attribute type
std::string ConnectorWFS::getAttributeType(std::string attributeName){
	logSS << "Retrieving attribute type for " << attributeName << " ...";
	logWFS->printToLog(logSS.str());
	logSS.str("");
	std::string aux_answer = "";

	for (int i=0; i<v_AttributesLayer.size(); i++){
		if(v_AttributesLayer[i][0]==attributeName){
			aux_answer = v_AttributesLayer[i][1];
		}
	}

	//Remove the last part (PropertyType) of the FeatureType to make it more understandable.
	std::string key ("PropertyType");
	size_t found;
	found=aux_answer.rfind(key);
	if (found!=string::npos){
		aux_answer.replace (found,key.length(),"");
	}
	if(aux_answer==""){
		logWFS->printToLog("ERROR: Attribute name not found. Check the name and try again");
	}
	logSS << "Attribute type retrieved [" << aux_answer << "].";
	logWFS->printToLog(logSS.str());
	logSS.str("");
	return aux_answer;
}


// Auxiliar functions
**********************************************************************************************************
// Provides the LAYER TITLE
std::string ConnectorWFS::getLayerTitle(std::string layerName){
	std::string var_aux = "";
	for (int i=0; i< vLayers.size(); i++){
		if(vLayers[i][0]==layerName){
			var_aux = vLayers[i][1]; //If found then update the value.
		}
	}
	if (var_aux=="")var_aux = "ERROR: Layer name not found on layer's list. Try refreshing the service.";
	return var_aux;
}

// Provides the LAYER ABSTRACT
std::string ConnectorWFS::getLayerAbstract(std::string layerName){
	std::string var_aux = "";
	for (int i=0; i< vLayers.size(); i++){
		if(vLayers[i][0]==layerName){
			var_aux = vLayers[i][2]; //If found then update the value.
		}
	}
```

```
     if (var_aux=="")var_aux = "ERROR: Layer name not found on layer's list. Try refreshing the service.";
     return var_aux;
}

std::string ConnectorWFS::getGeometryColumn(std::string layerName){
     getAttributeNamesList(layerName);
     logWFS->printToLog("Searching for the geometry column...");
     std::string var_aux = "";
     for (int i=0; i< v_AttributesLayer.size(); i++){
          if(v_AttributesLayer[i][1].rfind("gml:")==0){ //starts with gml:
               var_aux = v_AttributesLayer[i][0]; //If found then update the value.
          }
     }
     if (var_aux=="")var_aux = "ERROR: Layer name not found on layer's list. Try refreshing the service.";
     logWFS->printToLog("Geometry column found.");
     return var_aux;
}

std::string ConnectorWFS::getGeometryType(std::string layerName){
     std::string str_gc = getGeometryColumn(layerName);
     logWFS->printToLog("Obtaining geometry type for the layer...");
     std::string var_aux = "";
     var_aux = getAttributeType(str_gc);
     //If the geometry type retrieved is "Geometry" we have to perform a getFeature request to know what kind of
     element it will have.
     //It is the only way because that means the geometry type is not in the schema definition. [Deegree servers!]
     //The geometry type will be considered for all the layer the first basic type that will be found (gml:Point,
     gml:LineString, gml:Polygon).
     //var_aux should be translated to all caps or lowercase to check perfect match and not omit any result.
     std::transform(var_aux.begin(),var_aux.end(),var_aux.begin(), ::tolower); // Function that transforms a string
     to lowercase.
     if (var_aux == "gml:geometry" || var_aux == "geometry"){
          logWFS->printToLog("Unable to determine the geometry type from describeFeatureType operation. Using
          alternative metod...");
          std::string aux_gf = getFeature(layerName);
          size_t found=0;
          std::string key;
          key = ("gml:Point");
          found=aux_gf.find(key);
          if(found != -1){
               var_aux = key;
          }else{
               key = ("gml:LineString");
               found=aux_gf.find(key);
               if(found != -1){
                    var_aux = key;
               }else{
                    key = ("gml:Polygon");
                    found=aux_gf.find(key);
                    if(found != -1){
                         var_aux = key;
                    }else{
                         var_aux = "UNKNOWN TYPE";
                    }
               }
          }
     }
     if (var_aux==""){
```

```
            var_aux = "ERROR: Layer name not found on layer's list. Try refreshing the service.";
            logWFS->printToLog(var_aux);
      }else{
            logSS << "Geometry type obtained (" << var_aux << ").";
            logWFS->printToLog(logSS.str());
            logSS.str("");
      }
      return var_aux;
}

std::vector<double> ConnectorWFS::getLayerBoundingBox(std::string layerName){
      std::vector<double> aux_result;
      for (int i=0; i< vLayers.size(); i++){
            if(vLayers[i][0]==layerName){
                  aux_result.push_back(atof(vLayers[i][4].c_str()));
                  aux_result.push_back(atof(vLayers[i][5].c_str()));
                  aux_result.push_back(atof(vLayers[i][6].c_str()));
                  aux_result.push_back(atof(vLayers[i][7].c_str()));
            }
      }
      if (aux_result.size()==0) logWFS->printToLog("ERROR: Layer name not found on layer's list. Try refreshing the
      service.");
      return aux_result; //LOWERcornerLON, LOWERcornerLAT, UPPERcornerLON, UPPERcornerLAT
}

std::string ConnectorWFS::getLayerNameSpaceURL(std::string layerName){
      //I don't check if the layer exists or not, only the reference to the namespace.
      std::string aux_result = "no-url-found"; //Just in case the URL attribute is not found.
      //Extract the namespace from the layername ---> layerName = namespace:layer.
      size_t found = std::string::npos;
      found = layerName.find(":");
      std::string ns_aux = std::string("xmlns:").append(layerName.substr(0,found));
      if (found==std::string::npos) return aux_result; //If the layername doesn't contain namespace we retunt
      "no-url-found", in further versions we might return the default ns.
      for (int i=0; i< v_serviceNSurls.size(); i++){
            if(v_serviceNSurls[i][0]==ns_aux){
                  aux_result = v_serviceNSurls[i][1].c_str();
            }
      }
      if (aux_result=="no-url-found") logWFS->printToLog("WARNING: Layer namespace URL not found on layer's list,
      default 'no-url-found' value return. It might crash during further transaction operations. Continue at your own
      risk.");
      return aux_result;
}

//Returns a matrix for being able to store MULTIELEMENTS, ALTOUGH IN THIS VERSION WON'T BE IMPLEMENTED FULLY
FUNCTIONAL. For knowing if it's a multi element check the same id for more than one row in the vector.
//IMPORTANT NOTE: The connector doesn't work wit multi-elements, multi-elements are splitted to get simple ones.
std::vector<std::vector<std::string>> ConnectorWFS::getFeaturesFromLayer(std::string layerName, std::string version,
std::string filter){
      std::vector<std::vector<std::string>> aux_fullFeaturesList;
      std::vector<std::string> aux_feature;
      std::string aux_gf = ConnectorWFS::getFeature(layerName, version, filter);
      std::string geometryType = getGeometryType(layerName);
      std::string aux_str = "";
      std::string geom_column_name = "";
      geom_column_name = getGeometryColumn(layerName);
      //We have to check if the feature type contains the namespace (element separated by :) In that case add it to
```

```
    geometry columns:
    std::string ns = "";
    size_t pos_dots = layerName.find(":");
    if(pos_dots!=std::string::npos){
         //This means the layer name contains namespace.
         ns = layerName.substr(0,pos_dots);
         geom_column_name = ns.append(":").append(geom_column_name);
    }

    // Version 1.0.0
    ********************************************************************************************************
    if (version == "1.0.0"){
         logWFS->printToLog("ERROR: Version (1.0.0) not supported yet!");
    }
    // Version 1.1.0
    ********************************************************************************************************
    if (version == "1.1.0"){
         pugi::xml_document doc_gf;
         pugi::xml_parse_result result = doc_gf.load(aux_gf.c_str());
         //doc_gf.print(std::cout);
         pugi::xpath_node_set featureMember_node_set;
         //NOTE: Some servers have implemented this tag no standard mode, so we will search for "gml:featureMembers"
         too.
         featureMember_node_set = doc_gf.select_nodes("//gml:featureMember");
         if(featureMember_node_set.size()==0){
              logWFS->printToLog("WARNING: No <gml:featureMember> tag found, trying with Geoserver buggy custom tag:
              <gml:featureMembers>");
              featureMember_node_set = doc_gf.select_nodes("//gml:featureMembers");
              if(featureMember_node_set.size()==0){
                   logWFS->printToLog("WARNING: No <gml:featureMember> or <gml:featureMembers> tags found. Check the
                   query.");
              }
         }

         logSS << "Cantidad de nodos gml:featureMember(s) " << featureMember_node_set.size();
         logWFS->printToLog(logSS.str());
         logSS.str("");

         std::cout << "Accediendo a la parte de busqueda de geometrias" << std::endl;
         //Determine if the geometry is composed (MULTIelement contains several elementMEMBER {"1"MultiSurface
         ->"n"surfaceMember})
         if (geometryType=="gml:multisurface" || geometryType=="gml:Polygon"){     // MultiSurface could contain n
         polygons, each polygon could contain island polygons.
              logSS << "GeometryType is " << geometryType << std::endl;
              logWFS->printToLog(logSS.str());
              logSS.str("");
              for (pugi::xpath_node_set::const_iterator it = featureMember_node_set.begin(); it !=
              featureMember_node_set.end(); ++it){
                   pugi::xpath_node featureMember_node = *it;
                   pugi::xpath_node_set feature_nodes; //In case there are more than one feature per entity.
                   feature_nodes = featureMember_node.node().select_nodes("*"); //Collection of features, that can
                   be also made of multypolygons
                   //from those nodes hang the different fields, so we can retrieve from here.

                   logSS << "Number of FEATURE nodes found: " << feature_nodes.size();
                   logWFS->printToLog(logSS.str());
                   logSS.str("");
```

```
for(pugi::xpath_node_set::const_iterator it2 = feature_nodes.begin(); it2 != feature_nodes.end
(); ++it2){
     pugi::xpath_node feature_node = *it2;
     //Now we're in each one of the features, it's here where we can look for multipart or
     singlepart and extract it.
     //Each feature can contain several surfaceMember tags, but each surfaceMember can contain
     only ONE polygon,
     //So we can extract polygons directly from features as node_set.
     pugi::xpath_node_set polygon_node_set;
     polygon_node_set = feature_node.node().select_nodes(".//gml:Polygon");
     //-------------------------------------------------------------
     //Store the feature id in the gml file for further operations.
     std::string fid = "";
     fid = feature_node.node().attribute("gml:id").value();
     std::vector<std::string> aux_Attributes; //Vector that contains all the attributes (except
     the geom) in string mode
     aux_Attributes.clear();
     //THE FIRST ELEMENT IN THIS ATT. VECTOR WILL BE THE gml:id ATTRIBUTE, BECAUSE FOR
     TRANSACTIONS WILL BE USEFUL.
     aux_Attributes.push_back(fid);
     //We use the attribute in order, that simplifies the process
     pugi::xpath_node_set poly_attributes_set;
     poly_attributes_set = feature_node.node().select_nodes("./*");
     for(pugi::xpath_node_set::const_iterator att_it = poly_attributes_set.begin(); att_it!=
     poly_attributes_set.end(); ++att_it){
          pugi::xpath_node feature_att_node = *att_it;
          if (feature_att_node.node().name()!=geom_column_name){
               std::string str_field_value = feature_att_node.node().child_value();
               aux_Attributes.push_back(str_field_value);//Add to the vector the values for each
               column except the geometrycolumn.
          }
     }
     //-------------------------------------------------------------

     logSS << "Number of POLYGON nodes found: " << polygon_node_set.size();
     logWFS->printToLog(logSS.str());
     logSS.str("");

     //Now we have to check if the polygon contains "holes".
     //The structure is 1 tag gml:exterior 0..n tags gml:interior (the holes)
     for(pugi::xpath_node_set::const_iterator it3 = polygon_node_set.begin(); it3 !=
     polygon_node_set.end(); ++it3){
          pugi::xpath_node polygon_node = *it3;
          pugi::xpath_node_set interior_polygon_nodes;
          interior_polygon_nodes = polygon_node.node().select_nodes(".//gml:interior");

          logSS << "Number of INNER POLYGON nodes found: " << interior_polygon_nodes.size();
          logWFS->printToLog(logSS.str());
          logSS.str("");

          std::string str_ext = "";
          std::string str_CompletePosList = "";
          str_ext = polygon_node.node().select_single_node(".//gml:posList").node().child_value();
          //std::cout << "LISTA: " << str_ext << std::endl;
          str_CompletePosList.append(str_ext);

          std::vector<std::string> singleFeatureParsed; //Contains in STRING mode the
          information relative to each single feature part.
```

```
                    //Structure [GEOMLIST{EXTERIOR(,interior1, ..., interiorn)}, ATT1, ATT2.....]
                    if(interior_polygon_nodes.size()>0){
                        std::string str_int = "";
                        for (pugi::xpath_node_set::const_iterator it4=interior_polygon_nodes.begin();it4!=
                        interior_polygon_nodes.end();++it4){
                            pugi::xpath_node interior_node = *it4;
                            str_int = interior_node.node().select_single_node(".//gml:posList").node().
                            child_value();
                            str_CompletePosList.append(",").append(str_int);
                        }
                    }
                    singleFeatureParsed.push_back(str_CompletePosList); //FIRST MEMBER OF THE VECTOR.
                    [GEOMETRY LIST] [EXT,int,int...]
                    //singleFeatureParsed.push_back()//Attributes
                    for (int i=0; i< aux_Attributes.size(); i++){
                        singleFeatureParsed.push_back(aux_Attributes[i]);
                    }
                    logWFS->printToLog("Lista de coordenadas",singleFeatureParsed);
                    aux_fullFeaturesList.push_back(singleFeatureParsed);
                }
            }
        }
    }
    if (geometryType=="gml:multilinestring" || geometryType=="gml:LineString"){          // MultiLineString
    could contain n linestrings.
        logWFS->printToLog("GeometryType is gml:MultiLineString");
        for (pugi::xpath_node_set::const_iterator it = featureMember_node_set.begin(); it !=
        featureMember_node_set.end(); ++it){
            pugi::xpath_node featureMember_node = *it;
            pugi::xpath_node_set feature_nodes; //In case there are more than one feature per entity.
            feature_nodes = featureMember_node.node().select_nodes("*");
            //from those nodes hang the different fields, so we can retrieve from here.

            logSS << "Number of FEATURE nodes found: " << feature_nodes.size();
            logWFS->printToLog(logSS.str());
            logSS.str("");

            for(pugi::xpath_node_set::const_iterator it2 = feature_nodes.begin(); it2 != feature_nodes.end
            (); ++it2){
                pugi::xpath_node feature_node = *it2;
                //Now we're in each one of the features, it's here where we can look for multipart or
                singlepart and extract it.
                //Each feature can contain several gml:lineStringMember tags, but each lineStringMember can
                contain only ONE line,
                //So we can extract linestrings directly from features as node_set.
                //IF lineString_node_set.size()>1 then it's a multiline element.
                pugi::xpath_node_set lineString_node_set;
                lineString_node_set = feature_node.node().select_nodes(".//gml:LineString");
                //-------------------------------------------------------------------
                //Store the feature id in the gml file for further operations.
                std::string fid = "";
                fid = feature_node.node().attribute("gml:id").value();
                std::vector<std::string> aux_Attributes; //Vector that contains all the attributes (except
                the geom) in string mode
                aux_Attributes.clear();
                //THE FIRST ELEMENT IN THIS ATT. VECTOR WILL BE THE gml:id ATTRIBUTE, BECAUSE FOR
                TRANSACTIONS WILL BE USEFUL.
                aux_Attributes.push_back(fid);
```

```
                    //We use the attribute in order, that simplifies the process
                    pugi::xpath_node_set line_attributes_set;
                    line_attributes_set = feature_node.node().select_nodes("./*");
                    for(pugi::xpath_node_set::const_iterator att_it = line_attributes_set.begin(); att_it!=
                    line_attributes_set.end(); ++att_it){
                         pugi::xpath_node feature_att_node = *att_it;
                         if (feature_att_node.node().name()!=geom_column_name){
                              std::string str_field_value = feature_att_node.node().child_value();
                              aux_Attributes.push_back(str_field_value);//Add to the vector the values for each
                              column except the geometrycolumn.
                         }
                    }
                    //-----------------------------------------------------------------
                    logSS << "Number of LINESTRING nodes found in each feature: " << lineString_node_set.size();
                    logWFS->printToLog(logSS.str());
                    logSS.str("");

                    for(pugi::xpath_node_set::const_iterator it3=lineString_node_set.begin();it3!=
                    lineString_node_set.end();++it3){
                         pugi::xpath_node single_line_node = *it3;

                         std::vector<std::string> singleFeatureParsed; //Contains in STRING mode the
                         information relative to each single feature part.
                         //Structure [GEOMLIST, ATT1, ATT2.....]
                         std::string str_CompletePosList = single_line_node.node().select_single_node(
                         "./gml:posList").node().child_value();
                         singleFeatureParsed.push_back(str_CompletePosList);
                         //Add the other 'regular' attribute values to this list.
                         for (int i=0; i< aux_Attributes.size(); i++){
                              singleFeatureParsed.push_back(aux_Attributes[i]);
                         }
                         aux_fullFeaturesList.push_back(singleFeatureParsed);
                         logWFS->printToLog("Lista de coordenadas",singleFeatureParsed);
                    }
               }
          }
     }
     if (geometryType=="gml:multipoint" || geometryType=="gml:point"){          // MultiPoint could contain n
     points.
          logWFS->printToLog("GeometryType is gml:MultiPoint");
          for (pugi::xpath_node_set::const_iterator it = featureMember_node_set.begin(); it !=
          featureMember_node_set.end(); ++it){
               pugi::xpath_node featureMember_node = *it;
               pugi::xpath_node_set feature_nodes; //In case there are more than one feature per entity.
               feature_nodes = featureMember_node.node().select_nodes("*");
               //from those nodes hang the different fields, so we can retrieve from here.

               logSS << "Number of FEATURE nodes found: " << feature_nodes.size();
               logWFS->printToLog(logSS.str());
               logSS.str("");

               for(pugi::xpath_node_set::const_iterator it2 = feature_nodes.begin(); it2 != feature_nodes.end
               (); ++it2){
                    pugi::xpath_node feature_node = *it2;
                    //Now we're in each one of the features, it's here where we can look for multipart or
                    singlepart and extract it.
                    //Each feature can contain several gml:pointMember tags, but each pointMember can contain
                    only ONE point,
```

```
                    //So we can extract points directly from features as node_set.
                    //IF point_node_set.size()>1 then it's a multipoint element.
                    pugi::xpath_node_set point_node_set;
                    point_node_set = feature_node.node().select_nodes(".//gml:Point");
                    //-------------------------------------------------------------
                    //Store the feature id in the gml file for further operations.
                    std::string fid = "";
                    fid = feature_node.node().attribute("gml:id").value();
                    std::vector<std::string> aux_Attributes; //Vector that contains all the attributes (except
                    the geom) in string mode
                    aux_Attributes.clear();
                    //THE FIRST ELEMENT IN THIS ATT. VECTOR WILL BE THE gml:id ATTRIBUTE, BECAUSE FOR
                    TRANSACTIONS WILL BE USEFUL.
                    aux_Attributes.push_back(fid);
                    //We use the attribute in order, that simplifies the process
                    pugi::xpath_node_set point_attributes_set;
                    point_attributes_set = feature_node.node().select_nodes("./*");
                    for(pugi::xpath_node_set::const_iterator att_it = point_attributes_set.begin(); att_it!=
                    point_attributes_set.end(); ++att_it){
                         pugi::xpath_node feature_att_node = *att_it;
                         if (feature_att_node.node().name()!=geom_column_name){
                              std::string str_field_value = feature_att_node.node().child_value();
                              aux_Attributes.push_back(str_field_value);//Add to the vector the values for each
                              column except the geometrycolumn.
                         }
                    }
                    //-------------------------------------------------------------

                    logSS << "Number of POINT nodes found in each feature: " << point_node_set.size();
                    logWFS->printToLog(logSS.str());
                    logSS.str("");

                    for(pugi::xpath_node_set::const_iterator it3=point_node_set.begin();it3!=point_node_set.end
                    ();++it3){
                         pugi::xpath_node single_point_node = *it3;

                         std::vector<std::string> singleFeatureParsed; //Contains in STRING mode the
                         information relative to each single feature part.
                         //Structure [GEOMLIST, ATT1, ATT2.....]
                         std::string str_CompletePosList = single_point_node.node().select_single_node(
                         "./gml:pos").node().child_value();
                         singleFeatureParsed.push_back(str_CompletePosList);
                         //Add the other 'regular' attribute values to this list.//We suppose the first column
                         is the geometry column always, the second the gml:id
                         for (int i=0; i< aux_Attributes.size(); i++){
                              singleFeatureParsed.push_back(aux_Attributes[i]);
                         }
                         aux_fullFeaturesList.push_back(singleFeatureParsed);
                         logWFS->printToLog("Lista de coordenadas y atributos:",singleFeatureParsed);
                    }
               }
          }
     }
     //
     //
}
// Version 2.1.0
*****************************************************************************************************
```

```
    if (version == "2.0.0"){
        logWFS->printToLog("ERROR: Version (2.0.0) not supported yet!");
    }

    if (aux_fullFeaturesList.size()==0){
        logWFS->printToLog("WARNING: 0 features retrieved in this request.");
    }
    return aux_fullFeaturesList;
}

// Get the EPSG Code for one layer (The default EPSG Code obtained from capabilities document) THE SAME THAN THE
SERVER PRODUCES
std::string ConnectorWFS::getLayerSRS_FullEPSG(std::string layerName){
    std::string result = "";
    for(int i=0; i<vLayers.size(); i++){
        if(vLayers[i][0]==layerName){
            result = vLayers[i][3];
            //logWFS->printToLog(result);
        }
    }
    if(result==""){
        logWFS->printToLog("ERROR: SRS NOT FOUND.");
    }
    logWFS->printToLog("full SRS code retrieved");
    return result;
}

// Get the EPSG Code for one layer (The default EPSG Code obtained from capabilities document) ONLY THE NUMERIC CODE!
std::string ConnectorWFS::getLayerSRS_OnlyEPSGCode(std::string layerName){
    std::string result = "";
    std::string aux = getLayerSRS_FullEPSG(layerName);
    if (aux==""){
        logWFS->printToLog("ERROR: IMPOSSIBLE EXTRACT EPSG CODE, SRS Not retrieved successfully.");
    }else{
        size_t found=0;
        found = aux.rfind("EPSG:");
        if (found!=string::npos)
            result = aux.substr(found+5,aux.length()-found);//Number 5 is the lenght of "EPSG:" element.
        //logWFS->printToLog("SRS EPSG code retrieved");

    }
    return result;
}

//Initialize, We retrieve the basic information to ensure good performance later.
//In this connector we expect the service capabilities WON'T change while is being used.
int ConnectorWFS::initializeConnector(){
    int init_status = -1; //-1:Errors initializing, 0:No errors

    //Call the functions to fill the main vectors
    if (initializeFieldsCapabilities()==-1){
        logWFS->printToLog("ERROR: INITIALIZING BASIC VARIABLES PROCESS FAILED.");
        return init_status;
    }
    if (getCapabilities()==-1){
        logWFS->printToLog("ERROR: GETCAPABILITIES PROCESS FAILED.");
        return init_status;
    }
```

```
	init_status = 0;
	return init_status;
}

//Execute a transaction against a service. [Transactions use mandatory HTTP-POST.]
int ConnectorWFS::executeTransactionDelete(std::string layerName, std::vector<std::string> vector_ids, std::string
serv, std::string vers){
	int result = -1;
	const std::string str_nswfs = "wfs:";


	// BUILD THE URL POST
	std::string url_executeTransaction = serverURL;

	// PREPARE THE XML DOCUMENT FOR SENDING
	pugi::xml_document doc;
	//pugi::xml_node node = doc.append_child("GetCapabilities"); // Create the main node
	pugi::xml_node node = doc.append_child("wfs:Transaction"); // Create the main node
	// Add the attributes
	node.append_attribute("service")    = serv.c_str();
	node.append_attribute("version")    = vers.c_str(); //GET CAPABILITIES DO NOT USE VERSION PARAMETER.
	node.append_attribute("xmlns")      = "http://www.opengis.net/wfs";
	node.append_attribute("xmlns:xsi")  = "http://www.w3.org/2001/XMLSchema-instance";
	node.append_attribute("xmlns:wfs")  = "http://www.opengis.net/wfs";
	node.append_attribute("xmlns:gml")  = "http://www.opengis.net/gml";
	node.append_attribute("xmlns:ogc")  = "http://www.opengis.net/ogc";
	//node.append_attribute("xsi:schemaLocation") = "http://www.opengis.net/wfs
	http://schemas.opengis.net/wfs/1.1.0/wfs.xsd";


	//We create the OPERATION node for the transaction, in this case DELETE NODE.
	std::string str_operation = "Delete";
	str_operation.insert(0, str_nswfs);
	//node.append_child("asdasd");
	//The layer is defined as attribute in the "Operation" node.
	pugi::xml_node op_node = node.append_child(str_operation.c_str());
	op_node.append_attribute("typeName") = layerName.c_str();
	pugi::xml_node filter_node = op_node.append_child("ogc:Filter");
	//Collection of elemenents to be deleted
	//********************************************************************************** USING THE gml:id
	for (int i=0; i<vector_ids.size(); i++){
		pugi::xml_node node_id = filter_node.append_child("ogc:GmlObjectId");
		node_id.append_attribute("gml:id") = vector_ids[i].c_str();
	}

	//Transform the xml_document in a string that could be managed by curl
	std::stringstream xml_document("");
	doc.save(xml_document);
	xml_document.flush(); //Synchronizing the buffer
	std::string str_xml = xml_document.str();
	//str_xml = "<?xml version=\"1.0\"?><GetCapabilities service=\"WFS\" xmlns=\"http://www.opengis.net/wfs\"
	xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" xsi:schemaLocation=\"http://www.opengis.net/wfs
	http://schemas.opengis.net/wfs/1.1.0/wfs.xsd\"/>";

	doc.save_file("request.xml");
	// RETRIEVE THE INFORMATION
	MemoryStruct chunk;
	chunk.memory = (char*)malloc(1);  // will be grown as needed by the realloc above
```

```
	chunk.size = 0;	// no data at this point

	CURL *curl_handle;
	CURLcode res;
	curl_global_init(CURL_GLOBAL_ALL);// In windows, this will init the winsock stuff
	curl_handle = curl_easy_init();//get a curl handle
	if(curl_handle) {
		curl_easy_setopt(curl_handle, CURLOPT_URL, url_executeTransaction.c_str());	// First set the URL that
		is about to receive our POST.
		//curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDS, "service=wfs&request=getCapabilities"); //This only
		works with the standard curl header
		struct curl_slist *slist = curl_slist_append(NULL, "Content-Type: text/xml; charset=utf-8");
		curl_easy_setopt(curl_handle, CURLOPT_HTTPHEADER, slist);
		curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDS, str_xml.c_str());
		curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDSIZE, str_xml.length());
		curl_easy_setopt(curl_handle, CURLOPT_POST, 1L);
		curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, WriteMemoryCallback);
		curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk);
		res = curl_easy_perform(curl_handle);// Perform the request, res will get the return code
		// Check for errors
		if(res != CURLE_OK){
			fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(res));
		}
		curl_easy_cleanup(curl_handle);// always cleanup
	}
	curl_global_cleanup();

	// RETRIEVE THE ANSWER!
	pugi::xml_document answer_trans;
	answer_trans.load(std::string(chunk.memory).c_str());
	answer_trans.save_file("trans_answer.xml");
	//If the answer document contains tags "ows:Exception" then something went wrong.
	std::string aux_check(chunk.memory);
	size_t found = 0;
	found = aux_check.find("ows:Exception", 0);
	if (found==std::string::npos){
		found = aux_check.find("<Exception>", 0);
	}
	if (found!=std::string::npos){
		logWFS->printToLog("ERROR. TRANSACTION NOT EXECUTED");
		return -1;
	}else{
		pugi::xpath_node node_summary = answer_trans.select_single_node("//wfs:totalDeleted");
		logSS << str_operation << " transaction requested, " << node_summary.node().child_value() << " elements
		deleted.";
		logWFS->printToLog(logSS.str());
		logSS.str(""); //Clearing the strinStream.
		logSS.flush(); //Synchronizing the buffer
	}
	free(chunk.memory);
	result = 0; //If reaches this point without crash the operation was successful.
	return result;
}

int ConnectorWFS::executeTransactionDelete(std::string layerName, std::string str_field, std::string str_value, std
::string serv, std::string vers){
	int result = -1;
	const std::string str_nswfs = "wfs:";
```

```
// BUILD THE URL POST
std::string url_executeTransaction = serverURL;

// PREPARE THE XML DOCUMENT FOR SENDING
pugi::xml_document doc;
//pugi::xml_node node = doc.append_child("GetCapabilities"); // Create the main node
pugi::xml_node node = doc.append_child("wfs:Transaction"); // Create the main node
// Add the attributes
node.append_attribute("service")    = serv.c_str();
node.append_attribute("version")    = vers.c_str(); //GET CAPABILITIES DO NOT USE VERSION PARAMETER.
node.append_attribute("xmlns")      = "http://www.opengis.net/wfs";
node.append_attribute("xmlns:xsi") = "http://www.w3.org/2001/XMLSchema-instance";
node.append_attribute("xmlns:wfs") = "http://www.opengis.net/wfs";
node.append_attribute("xmlns:gml") = "http://www.opengis.net/gml";
node.append_attribute("xmlns:ogc") = "http://www.opengis.net/ogc";
//node.append_attribute("xsi:schemaLocation") = "http://www.opengis.net/wfs
http://schemas.opengis.net/wfs/1.1.0/wfs.xsd";


//We create the OPERATION node for the transaction, in this case DELETE NODE.
std::string str_operation = "Delete";
str_operation.insert(0, str_nswfs);
//node.append_child("asdasd");
//The layer is defined as attribute in the "Operation" node.
pugi::xml_node op_node = node.append_child(str_operation.c_str());
op_node.append_attribute("typeName") = layerName.c_str();
pugi::xml_node filter_node = op_node.append_child("ogc:Filter");
//******************************************************************************** SELECTING BY ONE
ATTRIBUTE VALUE
pugi::xml_node node_property = filter_node.append_child("ogc:PropertyIsEqualTo");
pugi::xml_node node_field = node_property.append_child("ogc:PropertyName");
pugi::xml_node node_literal = node_property.append_child("ogc:Literal");
node_field.append_child(pugi::node_pcdata).set_value(str_field.c_str());
node_literal.append_child(pugi::node_pcdata).set_value(str_value.c_str());

//Transform the xml_document in a string that could be managed by curl
std::stringstream xml_document("");
doc.save(xml_document);
xml_document.flush(); //Synchronizing the buffer
std::string str_xml = xml_document.str();
//doc.save_file("request.xml");

// RETRIEVE THE INFORMATION
MemoryStruct  chunk;
chunk.memory = (char*)malloc(1);   // will be grown as needed by the realloc above
chunk.size = 0;     // no data at this point

CURL *curl_handle;
CURLcode res;
curl_global_init(CURL_GLOBAL_ALL);// In windows, this will init the winsock stuff
curl_handle = curl_easy_init();//get a curl handle
if(curl_handle) {
     curl_easy_setopt(curl_handle, CURLOPT_URL, url_executeTransaction.c_str());     // First set the URL that
     is about to receive our POST.
     //curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDS, "service=wfs&request=getCapabilities"); //This only
     works with the standard curl header
```

```
          struct curl_slist *slist = curl_slist_append(NULL, "Content-Type: text/xml; charset=utf-8");
          curl_easy_setopt(curl_handle, CURLOPT_HTTPHEADER, slist);
          curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDS, str_xml.c_str());
          curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDSIZE, str_xml.length());
          curl_easy_setopt(curl_handle, CURLOPT_POST, 1L);
          curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, WriteMemoryCallback);
          curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk);
          res = curl_easy_perform(curl_handle);// Perform the request, res will get the return code
          // Check for errors
          if(res != CURLE_OK){
               fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(res));
          }
          curl_easy_cleanup(curl_handle);// always cleanup
     }
     curl_global_cleanup();

     // RETRIEVE THE ANSWER!
     pugi::xml_document answer_trans;
     answer_trans.load(std::string(chunk.memory).c_str());
     answer_trans.save_file("trans_answer.xml");
     //If the answer document contains tags "ows:Exception" then something went wrong.
     std::string aux_check(chunk.memory);
     size_t found = 0;
     found = aux_check.find("ows:Exception", 0);
     if (found==std::string::npos){
          found = aux_check.find("<Exception>", 0);
     }
     if (found!=std::string::npos){
          logWFS->printToLog("ERROR. TRANSACTION NOT EXECUTED");
          return -1;
     }else{
          pugi::xpath_node node_summary = answer_trans.select_single_node("//wfs:totalDeleted");
          logSS << str_operation << " transaction requested, " << node_summary.node().child_value() << " elements
          deleted.";
          logWFS->printToLog(logSS.str());
          logSS.str(""); //Clearing the strinStream.
          logSS.flush(); //Synchronizing the buffer
     }
     free(chunk.memory);
     result = 0; //If reaches this point without crash the operation was successful.
     return result;
}

int ConnectorWFS::executeTransactionInsert(std::string layerName, std::vector<std::vector<std::string>> v_attributes,
 std::string serv, std::string vers){
     if (vers != "1.1.0"){
          logWFS->printToLog("ERROR. VERSION NOT SUPPORTED YET.");
          return -1;
     }
     const std::string str_nswfs = "wfs:";

     // BUILD THE URL POST
     std::string url_executeTransaction = serverURL;

     // PREPARE THE XML DOCUMENT FOR SENDING
     pugi::xml_document doc;
     pugi::xml_node node = doc.append_child("wfs:Transaction"); // Create the main node
     // Add the attributes
```

```
node.append_attribute("service")    = serv.c_str();
node.append_attribute("version")    = vers.c_str();
node.append_attribute("xmlns")      = "http://www.opengis.net/wfs";
node.append_attribute("xmlns:xsi") = "http://www.w3.org/2001/XMLSchema-instance";
node.append_attribute("xmlns:wfs") = "http://www.opengis.net/wfs";
node.append_attribute("xmlns:gml") = "http://www.opengis.net/gml";
node.append_attribute("xmlns:ogc") = "http://www.opengis.net/ogc";
node.append_attribute("xsi:schemaLocation") = "xsi:schemaLocation=\"http://www.opengis.net/wfs
http://schemas.opengis.net/wfs/1.0.0/WFS-transaction.xsd\">";


//NAMESPACE OF THE LAYER MUST EXIST!
std::string str_nsLayer = "";
size_t foundns = 0;
foundns = layerName.find(":",0);
if (foundns != std::string::npos){
     str_nsLayer = layerName.substr(0,foundns);
     std::string aux = "xmlns:";
     aux.append(str_nsLayer);
     node.append_attribute(aux.c_str()) = getLayerNameSpaceURL(layerName).c_str();
}

//We create the OPERATION node for the transaction, in this case INSERT NODE.
std::string str_operation = "Insert";
str_operation.insert(0, str_nswfs);
pugi::xml_node op_node = node.append_child(str_operation.c_str());
pugi::xml_node layer_node = op_node.append_child(layerName.c_str());
//layer_node.append_attribute("xmlns:fran") =
"http://localhost:8080/geoserver/fran/wfs?service=WFS&version=1.1.0&request=DescribeFeatureType&typeName=fran:test
points1";

//Which kind of geometry does the layer have? (depending on the geometry we will define different node structure)
std::string lyr_geomType = getGeometryType(layerName); //pasar a minusculas.
std::transform(lyr_geomType.begin(),lyr_geomType.end(),lyr_geomType.begin(), ::tolower); // Function that
transforms a string to lowercase.
if (lyr_geomType == "gml:point" || lyr_geomType == "point" || lyr_geomType == "gml:multilinestring" ||
lyr_geomType == "multilinestring" || lyr_geomType == "gml:linestring" || lyr_geomType == "linestring" ||
lyr_geomType == "gml:polygon" || lyr_geomType == "polygon" || lyr_geomType == "gml:multipolygon" ||
lyr_geomType == "multipolygon" || lyr_geomType == "gml:multisurface" || lyr_geomType == "multisurface"){
     //Geometry detected, building customized transaction xml file
}else{
     logWFS->printToLog("ERROR. GEOMETRY TYPE UNDEFINED, IMPOSSIBLE BUILD TRANSACTION. EXECUTE TRANSACTION
     CANCELED.");
     return -1;
}
//Add the attribute nodes. The vector of information to be added always contains the first column as geometry
and the second as gml:id!
std::string column_geom = getGeometryColumn(layerName);
column_geom.insert(0,":").insert(0,str_nsLayer.c_str());
pugi::xml_node geom_node = layer_node.append_child(column_geom.c_str());
//First the "normal" attributes and then the geometry column.
for (int i=2; i<v_attributes.size(); i++){
     pugi::xml_node att_node = layer_node.append_child(v_attributes[i][0].c_str());
     att_node.append_child(pugi::node_pcdata).set_value(v_attributes[i][1].c_str());
}
if (lyr_geomType == "gml:point" || lyr_geomType == "point"){
     pugi::xml_node point_node = geom_node.append_child("gml:Point");
     pugi::xml_node pos_node = point_node.append_child("gml:pos"); //FOR WFS 1.0.0 pugi::xml_node pos_node =
```

```
        point_node.append_child("gml:coordinates");
        pos_node.append_child(pugi::node_pcdata).set_value(v_attributes[0][1].c_str());//Geometry coordinates list
    }
    //There's no support for "simple" linestring elements yet. only multiLine structures.
    if (lyr_geomType == "gml:linestring" || lyr_geomType == "linestring" || lyr_geomType == "gml:multilinestring"
    || lyr_geomType == "multilinestring"){
        pugi::xml_node multiElement_node = geom_node.append_child("gml:MultiLineString");
        pugi::xml_node member_node = multiElement_node.append_child("gml:lineStringMember");
        pugi::xml_node ls_node = member_node.append_child("gml:LineString");
        pugi::xml_node posl_node = ls_node.append_child("gml:posList");
        posl_node.append_child(pugi::node_pcdata).set_value(v_attributes[0][1].c_str());//Geometry coordinates list
    }
    //There's no support for "simple" linestring elements yet. only multiLine structures. Polygons with holes are
    supported.
    if (lyr_geomType == "gml:polygon" || lyr_geomType == "polygon" || lyr_geomType == "gml:multipolygon" ||
    lyr_geomType == "multipolygon" || lyr_geomType == "gml:multisurface" || lyr_geomType == "multisurface"){
        //Obtain number of interior polygons (Holes), such elements are separated by ",". The first element is the
        exterior one.
        size_t found = 0;
        size_t lastfound = 0;
        std::vector<int> v; //Vector withe the main splitting points, includes head and tail of the sequence.
        v.push_back(0);
        while (found!=std::string::npos){
            found = v_attributes[0][1].find(",",lastfound);
            if (found!=std::string::npos){
                v.push_back(found);
                lastfound=found+1;
            }
        }
        v.push_back(v_attributes[0][1].size());
        std::vector<std::string> v_boundaries;
        for (int i=0; i<v.size()-1; i++){
            if (i==0){ //External boundary
                v_boundaries.push_back(v_attributes[0][1].substr(v[0],v[1]));
            }else{
                int l0 = v[i]+1;
                int l1 = v[i+1];
                v_boundaries.push_back(v_attributes[0][1].substr(l0,l1-l0));
            }
        }
        pugi::xml_node multiElement_node = geom_node.append_child("gml:MultiSurface");
        pugi::xml_node member_node = multiElement_node.append_child("gml:surfaceMember");
        pugi::xml_node polygon_node = member_node.append_child("gml:Polygon");
        pugi::xml_node exterior_node = polygon_node.append_child("gml:exterior"); //-------------------- Exterior
        boundary
        pugi::xml_node linear_ring_node = exterior_node.append_child("gml:LinearRing");
        pugi::xml_node posl_node = linear_ring_node.append_child("gml:posList");
        posl_node.append_child(pugi::node_pcdata).set_value(v_boundaries[0].c_str());//Geometry coordinates list
        for (int i=1; i<v_boundaries.size(); i++){
            pugi::xml_node interior_node = polygon_node.append_child("gml:interior"); //--------------------
            Interior boundaries
            pugi::xml_node linear_ring_node = interior_node.append_child("gml:LinearRing");
            pugi::xml_node posl_node = linear_ring_node.append_child("gml:posList");
            posl_node.append_child(pugi::node_pcdata).set_value(v_boundaries[i].c_str());//Geometry coordinates list
        }
    }
```

```
//now we have to create the children for the geomertry column and for the standard attributes.

//Transform the xml_document in a string that could be managed by curl
std::stringstream xml_document("");
doc.save(xml_document);
xml_document.flush(); //Synchronizing the buffer
std::string str_xml = xml_document.str();
doc.save_file("request.xml");
//str_xml = "<wfs:Transaction service=\"WFS\" version=\"1.0.0\" xmlns:wfs=\"http://www.opengis.net/wfs\"
xmlns:topp=\"http://www.openplans.org/topp\" xmlns:gml=\"http://www.opengis.net/gml\"
xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" xsi:schemaLocation=\"http://www.opengis.net/wfs
http://schemas.opengis.net/wfs/1.0.0/WFS-transaction.xsd http://www.openplans.org/topp
http://localhost:8080/geoserver/wfs/DescribeFeatureType?typename=topp:tasmania_roads\"><wfs:Insert><topp:tasmania_
roads><topp:the_geom><gml:MultiLineString
srsName=\"http://www.opengis.net/gml/srs/epsg.xml#4326\"><gml:lineStringMember><gml:LineString><gml:coordinates
decimal=\".\" cs=\",\" ts=\" \">494475.71056415,5433016.8189323
494982.70115662,5435041.95096618</gml:coordinates></gml:LineString></gml:lineStringMember></gml:MultiLineString></
topp:the_geom><topp:TYPE>xxx</topp:TYPE></topp:tasmania_roads></wfs:Insert></wfs:Transaction>";



// RETRIEVE THE INFORMATION
MemoryStruct chunk;
chunk.memory = (char*)malloc(1);  // will be grown as needed by the realloc above
chunk.size = 0;     // no data at this point

CURL *curl_handle;
CURLcode res;
curl_global_init(CURL_GLOBAL_ALL);// In windows, this will init the winsock stuff
curl_handle = curl_easy_init();//get a curl handle
if(curl_handle) {
     curl_easy_setopt(curl_handle, CURLOPT_URL, url_executeTransaction.c_str());     // First set the URL that
     is about to receive our POST.
     //curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDS, "service=wfs&request=getCapabilities"); //This only
     works with the standard curl header
     struct curl_slist *slist = curl_slist_append(NULL, "Content-Type: text/xml; charset=utf-8");
     curl_easy_setopt(curl_handle, CURLOPT_HTTPHEADER, slist);
     curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDS, str_xml.c_str());
     curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDSIZE, str_xml.length());
     curl_easy_setopt(curl_handle, CURLOPT_POST, 1L);
     curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, WriteMemoryCallback);
     curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk);
     res = curl_easy_perform(curl_handle);// Perform the request, res will get the return code
     // Check for errors
     if(res != CURLE_OK){
          fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(res));
     }
     curl_easy_cleanup(curl_handle);// always cleanup
}
curl_global_cleanup();

// RETRIEVE THE ANSWER!
pugi::xml_document answer_trans;
answer_trans.load(std::string(chunk.memory).c_str());
answer_trans.save_file("trans_answer.xml");
//If the answer document contains tags "ows:Exception" then something went wrong.
std::string aux_check(chunk.memory);
size_t found = 0;
found = aux_check.find("ows:Exception", 0);
```

```
    if (found==std::string::npos){
        found = aux_check.find("<Exception>", 0);
    }
    if (found!=std::string::npos){
        logWFS->printToLog("ERROR. TRANSACTION NOT EXECUTED");
        return -1;
    }else{
        pugi::xpath_node node_summary = answer_trans.select_single_node("//wfs:totalInserted");
        logSS << str_operation << " transaction requested, " << node_summary.node().child_value() << " elements
        inserted.";
        logWFS->printToLog(logSS.str());
        logSS.str(""); //Clearing the strinStream.
        logSS.flush(); //Synchronizing the buffer
    }
    free(chunk.memory);
    return 0;//If reaches this point without crash the operation was successful.
}

int ConnectorWFS::executeTransactionUpdate(std::string layerName, std::string field, std::string value, std::vector<
std::string> vector_ids, std::string serv, std::string vers){
    if (vers != "1.1.0"){
        logWFS->printToLog("ERROR. VERSION NOT SUPPORTED YET.");
        return -1;
    }
    const std::string str_nswfs = "wfs:";

    // BUILD THE URL POST
    std::string url_executeTransaction = serverURL;

    // PREPARE THE XML DOCUMENT FOR SENDING
    pugi::xml_document doc;
    pugi::xml_node node = doc.append_child("wfs:Transaction"); // Create the main node
    // Add the attributes
    node.append_attribute("service")    = serv.c_str();
    node.append_attribute("version")    = vers.c_str();
    node.append_attribute("xmlns")       = "http://www.opengis.net/wfs";
    node.append_attribute("xmlns:xsi") = "http://www.w3.org/2001/XMLSchema-instance";
    node.append_attribute("xmlns:wfs") = "http://www.opengis.net/wfs";
    node.append_attribute("xmlns:gml") = "http://www.opengis.net/gml";
    node.append_attribute("xmlns:ogc") = "http://www.opengis.net/ogc";
    node.append_attribute("xsi:schemaLocation") = "xsi:schemaLocation=\"http://www.opengis.net/wfs
    http://schemas.opengis.net/wfs/1.0.0/WFS-transaction.xsd\">";

    //NAMESPACE OF THE LAYER MUST EXIST!
    std::string str_nsLayer = "";
    size_t foundns = 0;
    foundns = layerName.find(":",0);
    if (foundns != std::string::npos){
        str_nsLayer = layerName.substr(0,foundns);
        std::string aux = "xmlns:";
        aux.append(str_nsLayer);
        node.append_attribute(aux.c_str()) = getLayerNameSpaceURL(layerName).c_str();
    }

    //We create the OPERATION node for the transaction, in this case INSERT NODE.
    std::string str_operation = "Update";
    str_operation.insert(0, str_nswfs);
    pugi::xml_node op_node = node.append_child(str_operation.c_str());
```

```
pugi::xml_attribute op_att_node = op_node.append_attribute("typeName") = layerName.c_str();
//layer_node.append_attribute("xmlns:fran") =
"http://localhost:8080/geoserver/fran/wfs?service=WFS&version=1.1.0&request=DescribeFeatureType&typeName=fran:test
points1";

//We create the Property tags that contain the modifications. [On further versions we can create this as vector,
so we can modify more than one field at once]
pugi::xml_node prop_node = op_node.append_child("wfs:Property");
pugi::xml_node prop_name_node = prop_node.append_child("wfs:Name");
prop_name_node.append_child(pugi::node_pcdata).set_value(field.c_str());
pugi::xml_node prop_val_node = prop_node.append_child("wfs:Value");
prop_val_node.append_child(pugi::node_pcdata).set_value(value.c_str());

//In case the ids Vector is empty there's no filtering and apply to all layer
if(vector_ids.size()>0){
     pugi::xml_node filter_node = op_node.append_child("wfs:Filter");
     for (int i=0; i<vector_ids.size(); i++){
          pugi::xml_node n = filter_node.append_child("ogc:GmlObjectId");
          pugi::xml_attribute a = n.append_attribute("gml:id") = vector_ids[i].c_str();
     }
}



//Transform the xml_document in a string that could be managed by curl
std::stringstream xml_document("");
doc.save(xml_document);
xml_document.flush(); //Synchronizing the buffer
std::string str_xml = xml_document.str();
doc.save_file("request.xml");



// RETRIEVE THE INFORMATION
MemoryStruct chunk;
chunk.memory = (char*)malloc(1);  // will be grown as needed by the realloc above
chunk.size = 0;     // no data at this point

CURL *curl_handle;
CURLcode res;
curl_global_init(CURL_GLOBAL_ALL);// In windows, this will init the winsock stuff
curl_handle = curl_easy_init();//get a curl handle
if(curl_handle) {
     curl_easy_setopt(curl_handle, CURLOPT_URL, url_executeTransaction.c_str());     // First set the URL that
     is about to receive our POST.
     //curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDS, "service=wfs&request=getCapabilities"); //This only
     works with the standard curl header
     struct curl_slist *slist = curl_slist_append(NULL, "Content-Type: text/xml; charset=utf-8");
     curl_easy_setopt(curl_handle, CURLOPT_HTTPHEADER, slist);
     curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDS, str_xml.c_str());
     curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDSIZE, str_xml.length());
     curl_easy_setopt(curl_handle, CURLOPT_POST, 1L);
     curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, WriteMemoryCallback);
     curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk);
     res = curl_easy_perform(curl_handle);// Perform the request, res will get the return code
     // Check for errors
     if(res != CURLE_OK){
          fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(res));
     }
     curl_easy_cleanup(curl_handle);// always cleanup
```

```
	}
	curl_global_cleanup();

	// RETRIEVE THE ANSWER!
	pugi::xml_document answer_trans;
	answer_trans.load(std::string(chunk.memory).c_str());
	answer_trans.save_file("trans_answer.xml");
	//If the answer document contains tags "ows:Exception" then something went wrong.
	std::string aux_check(chunk.memory);
	size_t found = 0;
	found = aux_check.find("ows:Exception", 0);
	if (found==std::string::npos){
		found = aux_check.find("<Exception>", 0);
	}
	if (found!=std::string::npos){
		logWFS->printToLog("ERROR. TRANSACTION NOT EXECUTED");
		return -1;
	}else{
		pugi::xpath_node node_summary = answer_trans.select_single_node("//wfs:totalUpdated");
		logSS << str_operation << " transaction requested, " << node_summary.node().child_value() << " elements
		updated.";
		logWFS->printToLog(logSS.str());
		logSS.str(""); //Clearing the strinStream.
		logSS.flush(); //Synchronizing the buffer
	}
	free(chunk.memory);
	return 0;//If reaches this point without crash the operation was successful.
}


//Tags and variables.
int ConnectorWFS::initializeFieldsCapabilities(){
	int init_status = -1;
	tagsWFS110.clear();
	tagsWFS110.push_back("Abstract");
	tagsWFS110.push_back("Action");
	tagsWFS110.push_back("DefaultSRS");
	tagsWFS110.push_back("Delete");
	tagsWFS110.push_back("DescribeFeatureType");
	tagsWFS110.push_back("Feature");
	tagsWFS110.push_back("FeatureCollection");
	tagsWFS110.push_back("FeatureType");
	tagsWFS110.push_back("FeatureTypeList");
	tagsWFS110.push_back("FeaturesLocked");
	tagsWFS110.push_back("FeaturesNotLocked");
	tagsWFS110.push_back("Format");
	tagsWFS110.push_back("GMLObjectType");
	tagsWFS110.push_back("GetCapabilities");
	tagsWFS110.push_back("GetFeature");
	tagsWFS110.push_back("GetFeatureWithLock");
	tagsWFS110.push_back("GetGmlObject");
	tagsWFS110.push_back("Insert");
	tagsWFS110.push_back("InsertResults");
	tagsWFS110.push_back("Lock");
	tagsWFS110.push_back("LockFeature");
	tagsWFS110.push_back("LockFeatureResponse");
	tagsWFS110.push_back("LockId");
	tagsWFS110.push_back("Message");
```

```
    tagsWFS110.push_back("MetadataURL");
    tagsWFS110.push_back("Name");
    tagsWFS110.push_back("Native");
    tagsWFS110.push_back("NoSRS");
    tagsWFS110.push_back("Operation");
    tagsWFS110.push_back("Operations");
    tagsWFS110.push_back("OtherSRS");
    tagsWFS110.push_back("OutputFormats");
    tagsWFS110.push_back("Property");
    tagsWFS110.push_back("PropertyName");
    tagsWFS110.push_back("Query");
    tagsWFS110.push_back("ServesGMLObjectTypeList");
    tagsWFS110.push_back("SupportsGMLObjectTypeList");
    tagsWFS110.push_back("Title");
    tagsWFS110.push_back("Transaction");
    tagsWFS110.push_back("TransactionResponse");
    tagsWFS110.push_back("TransactionResults");
    tagsWFS110.push_back("TransactionSummary");
    tagsWFS110.push_back("TypeName");
    tagsWFS110.push_back("Update");
    tagsWFS110.push_back("Value");
    tagsWFS110.push_back("WFS_Capabilities");
    tagsWFS110.push_back("XlinkPropertyName");
    tagsWFS110.push_back("totalDeleted");
    tagsWFS110.push_back("totalInserted");
    tagsWFS110.push_back("totalUpdated");
    tagsWFS110.push_back("ActionType");
    tagsWFS110.push_back("BaseRequestType");
    tagsWFS110.push_back("DeleteElementType");
    tagsWFS110.push_back("DescribeFeatureTypeType");
    tagsWFS110.push_back("FeatureCollectionType");
    tagsWFS110.push_back("FeatureTypeListType");
    tagsWFS110.push_back("FeatureTypeType");
    tagsWFS110.push_back("FeaturesLockedType");
    tagsWFS110.push_back("FeaturesNotLockedType");
    tagsWFS110.push_back("GMLObjectTypeListType");
    tagsWFS110.push_back("GMLObjectTypeType");
    tagsWFS110.push_back("GetCapabilitiesType");
    tagsWFS110.push_back("GetFeatureType");
    tagsWFS110.push_back("GetFeatureWithLockType");
    tagsWFS110.push_back("GetGmlObjectType");
    tagsWFS110.push_back("InsertElementType");
    tagsWFS110.push_back("InsertResultType");
    tagsWFS110.push_back("InsertedFeatureType");
    tagsWFS110.push_back("LockFeatureResponseType");
    tagsWFS110.push_back("LockFeatureType");
    tagsWFS110.push_back("LockType");
    tagsWFS110.push_back("MetadataURLType");
    tagsWFS110.push_back("NativeType");
    tagsWFS110.push_back("OperationsType");
    tagsWFS110.push_back("OutputFormatListType");
    tagsWFS110.push_back("PropertyType");
    tagsWFS110.push_back("QueryType");
    tagsWFS110.push_back("TransactionResponseType");
    tagsWFS110.push_back("TransactionResultsType");
    tagsWFS110.push_back("TransactionSummaryType");
    tagsWFS110.push_back("TransactionType");
    tagsWFS110.push_back("UpdateElementType");
```

```
    tagsWFS110.push_back("WFS_CapabilitiesType");
    tagsWFS110.push_back("AllSomeType");
    tagsWFS110.push_back("Base_TypeNameListType");
    tagsWFS110.push_back("IdentifierGenerationOptionType");
    tagsWFS110.push_back("OperationType");
    tagsWFS110.push_back("ResultTypeType");
    tagsWFS110.push_back("TypeNameListType");
    logWFS->printToLog("Initialized standard set of WFS 1.1.0 tags & basic variables for GetCapabilities request");

    init_status = 0;
    return init_status;
}
```

```
void wfsimportlayer(const String& cmd);

//Auxiliar function to initialize the WFS bridge, with standard and common parameters (for WFS 1.1.0)
void initialize();

void generatePointMap();
void generateLineMap();
void generatePolygonMap();

//Reads the coordinate list for points and return a vector with two coords X,Y
std::vector<double> parseCoordinatesPoint(const std::string coordList);

//Reads the coordinate list for points and return a vector with two coords X1,Y1,X2,Y2...
std::vector<std::vector<double>> parseCoordinatesPolyline(const std::string coordList);

//If interior polygons the first vector is the exterior and the next interiros
V1{X1,Y1,X2,Y2...},V2{Interior1},V3{Interior2}...
std::vector<std::vector<std::vector<double>>> parseCoordinatesPolygon(const std::string coordList);
```

```
/***************************************************************
 ILWIS integrates image, vector and thematic data in one unique
 and powerful package on the desktop. ILWIS delivers a wide
 range of feautures including import/export, digitizing, editing,
 analysis and display of data as well as production of
 quality mapsinformation about the sensor mounting platform

 Exclusive rights of use by 52°North Initiative for Geospatial
 Open Source Software GmbH 2007, Germany

 Copyright (C) 2007 by 52°North Initiative for Geospatial
 Open Source Software GmbH

 Author: Jan Hendrikse, Willem Nieuwenhuis,Wim Koolhoven
 Bas Restsios, Martin Schouwenburg, Lichun Wang, Jelle Wind

 Contact: Martin Schouwenburg; schouwenburg@itc.nl;
 tel +31-534874371

 This program is free software; you can redistribute it and/or
 modify it under the terms of the GNU General Public License
 version 2 as published by the Free Software Foundation.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with this program (see gnu-gpl v2.txt); if not, write to
 the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 Boston, MA 02111-1307, USA or visit the web page of the Free
 Software Foundation, http://www.fsf.org.

 Created on: 2007-02-8
 ***************************************************************/

#include "Headers\messages.h"
#include "Headers\toolspch.h"
#include "Engine\SpatialReference\Coordsys.h"
#include "Engine\Table\Col.h"
#include "Engine\Base\DataObjects\valrange.h"
#include "Engine\DataExchange\ForeignFormat.h"
#include "Engine\Map\Point\ilwPoint.h"
#include "Engine\Map\Point\PNT.H"
#include "Engine\Map\Polygon\POL.H"
#include "Engine\Map\Point\PNTSTORE.H"
#include "Engine\Map\Polygon\POLSTORE.H"
#include "Engine\Map\Polygon\POL14.H"
#include "Engine\Base\DataObjects\valrange.h"
#include "Engine\Domain\Dmvalue.h"
#include "Engine\Base\System\Engine.h"
#include "Engine\DataExchange\gdalproxy.h"
#include <cstdlib>
#include <time.h>
#include "wfs.h"
#include "Log.hpp"
#include "ConnectorWFS.hpp"
```

```
struct wfsdata{
     String cmd;
     String dir;
};


Log *logWFS_ILWIS;
std::string str_lyr;
std::string str_GeometryType;
std::string lyr_SRS;
std::vector<double> lyr_BBox;
std::vector<std::string> lyr_ColumnNames;
std::vector<std::vector<std::string>> lyr_geometries;

void importWFS(const String& cmd, const String& dir) {

     initialize();

     const std::string str_url = "http://192.168.43.101:8082/geoserver/wfs?";
     ConnectorWFS *conn = new ConnectorWFS(str_url); //Create object in the heap

     str_lyr = "sf:restricted";
     str_GeometryType = conn->getGeometryType(str_lyr);
     lyr_SRS = conn->getLayerSRS_OnlyEPSGCode(str_lyr);
     lyr_BBox = conn->getLayerBoundingBox(str_lyr);
     lyr_ColumnNames = conn->getAttributeNamesList(str_lyr);
     lyr_geometries = conn->getFeaturesFromLayer(str_lyr);

     //Figure out which kind of map do we have to create
     if (str_GeometryType == "gml:point") generatePointMap();
     if (str_GeometryType == "gml:linestring" || str_GeometryType == "gml:multilinestring") generateLineMap();
     if (str_GeometryType == "gml:polygon" || str_GeometryType == "gml:multisurface") generatePolygonMap();
     if (str_GeometryType != "gml:point" && str_GeometryType != "gml:linestring" && str_GeometryType !=
     "gml:multilinestring" && str_GeometryType != "gml:polygon" && str_GeometryType != "gml:multisurface") {
          logWFS_ILWIS->printToLog("ERROR: GEOMETRY TYPE NOT RECOGNIZED.");
          exit(EXIT_FAILURE);
     }
}

UINT WfsThread(void * data) {
     wfsdata *d = (wfsdata *)data;
     getEngine()->InitThreadLocalVars();
     String cmd = d->cmd;
     String dir = d->dir;
     delete d;
     importWFS(cmd, dir);
     getEngine()->RemoveThreadLocalVars();

     return 1;
}

void wfsimportlayer(const String& cmd) {
     ParmList parms(cmd);
     if ( !parms.fExist("quiet")) {
          wfsdata *d = new wfsdata();
          d->cmd = cmd;
          d->dir = getEngine()->sGetCurDir();
```

```
            AfxBeginThread(WfsThread, (LPVOID)d);
      }
}

void initialize(){
      logWFS_ILWIS = Log::Instance();
      str_lyr;
      str_GeometryType;
      lyr_SRS="";
      lyr_BBox.clear();
      lyr_ColumnNames.clear();
      lyr_geometries.clear();
}

void generatePointMap(){
      srand( (unsigned)time( NULL ) );                //TODO: REMOVE THIS AND SUBSTITUTE IT FROM NAME OBTAINED FROM GUI.
      int random_integer = rand() % (1000000 + 1);
      std::string mapname = "PointMap";
      char random_char[20];
      itoa(random_integer, random_char, 10);
      std::string str_fnMap = mapname.append(random_char).append(".mpp");
      FileName fnMap (str_fnMap);
      //FileName fnMap ("PointMap2.mpp");
      FileName fnDom(fnMap, ".dom");
      FileName fnAttr(fnMap, ".tbt");
      FileName fnBBCsys(fnMap, ".csy");
      Domain dom(fnDom, lyr_geometries.size(), dmtUNIQUEID, "gml:point");
      //   CoordSystem coordSys (fnCsy);
      CoordSystem coordSys = getEngine()->gdal->getCoordSystem(fnMap,atoi(lyr_SRS.c_str())); //Filename, EPSG Code in
      INTEGER!.

      //The boundaries are always in WGS84 in WFS.
      CoordSystem csyWGS84 ("LatlonWGS84.csy");
      //CoordSystem csyWGS84 = getEngine()->gdal->getCoordSystem(fnBBCsys,4326); //Filename, EPSG Code in INTEGER!.
      CoordBounds BBox = CoordBounds(lyr_BBox[0],lyr_BBox[1],lyr_BBox[2],lyr_BBox[3]); //LOWER CORNER LON, LAT, UPPER
      CORNER LON, LAT.
      BBox = csyWGS84->cbConv(csyWGS84, BBox);

      PointMap pm(fnMap, coordSys, BBox, dom); //dm is the linking domain between features and tables.

      //Create the att. table and its columns.
      Table attable(fnAttr, dom);

      //Add the gml:id column.
      Column c = attable->colNew("gml:id",Domain("string"));
      //Add the remaining tables. for now all will be strings.
      for (int i=1; i<lyr_ColumnNames.size(); i++){
            Column c = attable->colNew(lyr_ColumnNames[i], Domain("string"));//Domains for int, double, string, date,
            classes... HOW TO???
            c->SetOwnedByTable(true);
      }

      //Filling the attribute table. Attention: We consider too the gml:id property as a column in the att table!!.
      for (int i=1; i<lyr_ColumnNames.size()+1; i++){//In columnNames we don't have the id column.//Here we suppose
      the geoemtry column is always the first one and that's why we skip the first.
            //Select the column we're gonna fill
            Column c = attable->col(i-1);//Ilwis counts the first column as 0.
            for (int j=0; j<lyr_geometries.size(); j++){
```

```
                c->PutVal(j+1, lyr_geometries[j][i]); //TODO : The value should be the right TYPE, I don't care about
                it here yet.
          }
     }

     //Create the features and its geometries.
     CoordBounds bbox;
     std::vector<double> v_coord;
     long raw_index = 1; //All features in Ilwis have a “raw” value which is basically a long ( starting at index 1,
     0 is reserved for the undefined value).
     //There's a ¿bug?, so I have to add the first one twice!
     v_coord = parseCoordinatesPoint(lyr_geometries[1][0]);
     Coord crd (v_coord[0],v_coord[1],0);
     ILWIS::Point *feature = CPOINT(pm->newFeature());
     feature->setCoord(crd);
     feature->PutVal(raw_index);
     for (int i=0; i<lyr_geometries.size(); i++){
          raw_index++;
          v_coord.clear();
          v_coord = parseCoordinatesPoint(lyr_geometries[i][0]);
          Coord crd (v_coord[0],v_coord[1],0);
          ILWIS::Point *feature = CPOINT(pm->newFeature());
          feature->setCoord(crd);
          feature->PutVal(raw_index);
          //We increase the bbox size depending on the coordinates of the new points. //Alternative way to stablish
          the bbox.
          bbox += crd;
     }
     pm->SetCoordBounds(bbox);

     //Try to show automatically the map once is done.
     String sCmd = "show " + fnMap.sFullNameQuoted(true);
     char* str = sCmd.sVal();
     getEngine()->SendMessage(ILWM_EXECUTE, 0, (LPARAM)str);
}

void generateLineMap(){
     srand( (unsigned)time( NULL ) );
     int random_integer = rand() % (1000000 + 1);
     std::string mapname = "SegmentMap";
     char random_char[20];
     itoa(random_integer, random_char, 10);
     std::string str_fnMap = mapname.append(random_char).append(".mps");
     FileName fnMap (str_fnMap);
     FileName fnDom(fnMap, ".dom");
     FileName fnAttr(fnMap, ".tbt");
     FileName fnBBCsys(fnMap, ".csy");
     Domain dom(fnDom, lyr_geometries.size(), dmtUNIQUEID, "gml:linestring");
     CoordSystem coordSys = getEngine()->gdal->getCoordSystem(fnMap,atoi(lyr_SRS.c_str())); //Filename, EPSG Code in
     INTEGER!.

     //The boundaries are always in WGS84 in WFS.
     CoordSystem csyWGS84 ("LatlonWGS84.csy");
     //CoordSystem csyWGS84 = getEngine()->gdal->getCoordSystem(fnBBCsys,4326); //Filename, EPSG Code in INTEGER!.

     //This way to establish the boundaries DOES NOT work. It will be recalculated adding the geometries.
     CoordBounds BBox = CoordBounds(lyr_BBox[0],lyr_BBox[1],lyr_BBox[2],lyr_BBox[3]); //LOWER CORNER LON, LAT, UPPER
     CORNER LON, LAT.
```

```
BBox = csyWGS84->cbConv(csyWGS84, BBox);

SegmentMap sm(fnMap, coordSys, BBox, dom); //dm is the linking domain between features and tables.

//Create the att. table and its columns.
Table attable(fnAttr, dom);

//Add the gml:id column.
Column c = attable->colNew("gml:id",Domain("string"));
//Add the remaining tables. for now all will be strings.
for (int i=1; i<lyr_ColumnNames.size(); i++){
     Column c = attable->colNew(lyr_ColumnNames[i], Domain("string"));
     c->SetOwnedByTable(true);
}

//Filling the attribute table. Attention: We consider too the gml:id property as a column in the att table!!.
for (int i=1; i<lyr_ColumnNames.size()+1; i++){//In columnNames we don't have the id column.//Here we suppose
the geoemtry column is always the first one and that's why we skip the first.
     //Select the column we're gonna fill
     Column c = attable->col(i-1);//Ilwis counts the first column as 0.
     for (int j=0; j<lyr_geometries.size(); j++){
          c->PutVal(j+1, lyr_geometries[j][i]); //TODO : The value should be the right TYPE, I don't care about
          it here yet.
     }
}

//Create the features and its geometries.
CoordBounds bbox;
//std::vector<double> v_coord;
long raw_index = 1; //All features in Ilwis have a “raw” value which is basically a long ( starting at index 1,
0 is reserved for the undefined value).
//There's a ¿bug?, so I have to add the first one twice!
ILWIS::Segment *feature = CSEGMENT(sm->newFeature());
std::vector<geos::geom::Coordinate> coord_seq;
double xmin=0;
double xmax=0;
double ymin=0;
double ymax=0;
std::vector<std::vector<double>> v_coord = parseCoordinatesPolyline(lyr_geometries[1][0]);
for (int i=0; i<v_coord.size(); i++){
     if (i==0){
          xmin=v_coord[i][0];
          xmax=v_coord[i][0];
          ymin=v_coord[i][1];
          ymax=v_coord[i][1];
     }else{
          if(v_coord[i][0]<xmin) xmin=v_coord[i][0];
          if(v_coord[i][0]>xmax) xmax=v_coord[i][0];
          if(v_coord[i][1]<ymin) ymin=v_coord[i][1];
          if(v_coord[i][1]>ymax) ymax=v_coord[i][1];
     }
     geos::geom::Coordinate c_aux = Coordinate(v_coord[i][0],v_coord[i][1],0);
     coord_seq.push_back(c_aux);
}
geos::geom::CoordinateSequence *seq = feature->getCoordinates(); //Initialize the sequence.
seq->setPoints(coord_seq);
feature->PutCoords(seq);
feature->PutVal(raw_index);
```

```
    for (int count=0; count<lyr_geometries.size(); count++){
        raw_index++;
        v_coord.clear();
        coord_seq.clear();

        v_coord = parseCoordinatesPolyline(lyr_geometries[1][0]);
        for (int i=0; i<v_coord.size(); i++){
            if (i==0){
                xmin=v_coord[i][0];
                xmax=v_coord[i][0];
                ymin=v_coord[i][1];
                ymax=v_coord[i][1];
            }else{
                if(v_coord[i][0]<xmin) xmin=v_coord[i][0];
                if(v_coord[i][0]>xmax) xmax=v_coord[i][0];
                if(v_coord[i][1]<ymin) ymin=v_coord[i][1];
                if(v_coord[i][1]>ymax) ymax=v_coord[i][1];
            }
            geos::geom::Coordinate c_aux = Coordinate(v_coord[i][0],v_coord[i][1],0);
            coord_seq.push_back(c_aux);
            geos::geom::CoordinateSequence *seq = feature->getCoordinates(); //Initialize the sequence.
            seq->setPoints(coord_seq);
            feature->PutCoords(seq);
            feature->PutVal(raw_index);
        }

    }

    //Establish the real value for the bounding box
    geos::geom::Coordinate c_aux1 = Coordinate(xmin,ymin,0);
    geos::geom::Coordinate c_aux2 = Coordinate(xmax,ymax,0);
    bbox+=c_aux1;
    bbox+=c_aux2;
    sm->SetCoordBounds(bbox);

    //Try to show automatically the map once is done.
    String sCmd = "show " + fnMap.sFullNameQuoted(true);
    char* str = sCmd.sVal();
    getEngine()->SendMessage(ILWM_EXECUTE, 0, (LPARAM)str);

}

void generatePolygonMap(){
    srand( (unsigned)time( NULL ) );                //TODO: REMOVE THIS AND SUBSTITUTE IT FROM NAME OBTAINED FROM GUI.
    int random_integer = rand() % (1000000 + 1);
    std::string mapname = "POLYGON_Map";
    char random_char[20];
    itoa(random_integer, random_char, 10);
    std::string str_fnMap = mapname.append(random_char).append(".mpa");
    FileName fnMap (str_fnMap);
    FileName fnDom(fnMap, ".dom");
    FileName fnAttr(fnMap, ".tbt");
    FileName fnBBCsys(fnMap, ".csy");
    Domain dom(fnDom, lyr_geometries.size(), dmtUNIQUEID, "gml:multisurface");
    CoordSystem coordSys = getEngine()->gdal->getCoordSystem(fnMap,atoi(lyr_SRS.c_str())); //Filename, EPSG Code in
    INTEGER!.
```

```
//The boundaries are always in WGS84 in WFS.
CoordSystem csyWGS84 ("LatlonWGS84.csy");
//CoordSystem csyWGS84 = getEngine()->gdal->getCoordSystem(fnBBCsys,4326); //Filename, EPSG Code in INTEGER!.

//This way to establish the boundaries DOES NOT work. It will be recalculated adding the geometries.
CoordBounds BBox = CoordBounds(lyr_BBox[0],lyr_BBox[1],lyr_BBox[2],lyr_BBox[3]); //LOWER CORNER LON, LAT, UPPER CORNER LON, LAT.
BBox = csyWGS84->cbConv(csyWGS84, BBox);

PolygonMap pm(fnMap, coordSys, BBox, dom); //dm is the linking domain between features and tables.

//Create the att. table and its columns.
Table attable(fnAttr, dom);

//Add the gml:id column.
Column c = attable->colNew("gml:id",Domain("string"));
//Add the remaining tables. for now all will be strings.
for (int i=1; i<lyr_ColumnNames.size(); i++){
     Column c = attable->colNew(lyr_ColumnNames[i], Domain("string"));
     c->SetOwnedByTable(true);
}

//Filling the attribute table. Attention: We consider too the gml:id property as a column in the att table!!.
for (int i=1; i<lyr_ColumnNames.size()+1; i++){//In columnNames we don't have the id column.//Here we suppose the geoemtry column is always the first one and that's why we skip the first.
     //Select the column we're gonna fill
     Column c = attable->col(i-1);//Ilwis counts the first column as 0.
     for (int j=0; j<lyr_geometries.size(); j++){
          c->PutVal(j+1, lyr_geometries[j][i]); //TODO : The value should be the right TYPE, I don't care about it here yet.
     }
}

//Create the features and its geometries.
CoordBounds bbox;
//std::vector<double> v_coord;
long raw_index = 1; //All features in Ilwis have a “raw” value which is basically a long ( starting at index 1, 0 is reserved for the undefined value).
//There's a ¿bug?, so I have to add the first one twice!
//ILWIS::Polygon *feature = CPOLYGON(pm->newFeature());
std::vector<geos::geom::Coordinate> coord_seq;
double xmin=0;
double xmax=0;
double ymin=0;
double ymax=0;

//std::vector<std::vector<double>> v_coord = parseCoordinatesPolyline(lyr_geometries[1][0]);
for (int i=0; i<lyr_geometries.size();i++){      //Iterate for all polygons in the layer.
     std::vector<std::vector<std::vector<double>>> v_boundaries = parseCoordinatesPolygon(lyr_geometries[i][0]);

     ILWIS::Polygon *feature = CPOLYGON(pm->newFeature());
     //ILWIS::Polygon *empty_feature = CPOLYGON(pm->newFeature());

     for (int j=0; j<v_boundaries.size(); j++){  //Iterate for all rings in the polygon.
          std::vector<std::vector<double>> v_b = v_boundaries[j];
          geos::geom::CoordinateSequence *seq = new CoordinateArraySequence ();
          //empty_feature->getCoordinates(); //Initialize the sequence, since there's no previous data this initialization implies an empty set.
```

```
                std::vector<geos::geom::Coordinate> coord_seq;

                for (int k=0; k<v_b.size(); k++){      //Iterate for all vertex nodes that define the boundary.
                     std::vector<double> v_coord = v_b[k];
                     geos::geom::Coordinate c_aux = Coordinate(v_coord[0],v_coord[1],0);
                     coord_seq.push_back(c_aux);
                     seq->setPoints(coord_seq);

                     //Bounding box checking ----------------------
                     if (i==0 && k==0){ //Only for the first element!!!
                          xmin=v_coord[0];
                          xmax=v_coord[0];
                          ymin=v_coord[1];
                          ymax=v_coord[1];
                     }else{
                          if(v_coord[0]<xmin) xmin=v_coord[0];
                          if(v_coord[0]>xmax) xmax=v_coord[0];
                          if(v_coord[1]<ymin) ymin=v_coord[1];
                          if(v_coord[1]>ymax) ymax=v_coord[1];
                     }// -----------------------------------------

                }
                LinearRing *ring = new LinearRing(seq, new geos::geom::GeometryFactory());
                if (j==0){
                     //External polygon boundary
                     feature->addBoundary(ring);
                }else{
                     //Internal polygon boundaries
                     feature->addHole(ring);
                }
                feature->PutVal(raw_index);
                raw_index++;
           }

     }

     //Establish the real value for the bounding box
     geos::geom::Coordinate c_aux1 = Coordinate(xmin,ymin,0);
     geos::geom::Coordinate c_aux2 = Coordinate(xmax,ymax,0);
     //geos::geom::Coordinate c_aux1 = Coordinate(lyr_BBox[0],lyr_BBox[1],0);
     //geos::geom::Coordinate c_aux2 = Coordinate(lyr_BBox[2],lyr_BBox[3],0);
     bbox+=c_aux1;
     bbox+=c_aux2;
     pm->SetCoordBounds(bbox);

}

//Reads the coordinate list for points and return a vector with two coords X,Y
std::vector<double> parseCoordinatesPoint(const std::string coordList) {
     std::vector<double> v_res;
     std::string x = "";
     std::string y = "";
     size_t pos = std::string::npos;
     pos = coordList.find(" ",0);
     x = coordList.substr(0,pos);
     y = coordList.substr(pos);
     v_res.push_back(atof(x.c_str()));
     v_res.push_back(atof(y.c_str()));
```

```
     return v_res;
}
//Reads the coordinate list for points and return a vector with two coords X1,Y1,X2,Y2...
std::vector<std::vector<double>> parseCoordinatesPolyline(const std::string coordList) {
     std::vector<std::vector<double>> result;
     std::vector<double> v_aux;

     size_t pos0 = 0;//std::string::npos;
     size_t pos1 = 0;//std::string::npos;
     size_t pos2 = 0;//std::string::npos;

     std::string x = "";
     std::string y = "";

     bool condition=true;
     while(condition==true){
          x="";
          y="";
          v_aux.clear();
          pos1 = coordList.find(" ",pos0);
          pos2 = coordList.find(" ",pos1+1);
          x = coordList.substr(pos0,pos1-pos0);
          y = coordList.substr(pos1+1,pos2-pos1-1);
          pos0=pos2+1;

          v_aux.push_back(atof(x.c_str()));
          v_aux.push_back(atof(y.c_str()));
          result.push_back(v_aux);

          if(coordList.find(" ",pos2)==std::string::npos) condition=false;
     }

     return result;
}

//If interior polygons the first vector is the exterior and the next interiros
V1{X1,Y1,X2,Y2...},V2{Interior1},V3{Interior2}...
std::vector<std::vector<std::vector<double>>> parseCoordinatesPolygon(const std::string coordList){
     std::vector<std::vector<std::vector<double>>> result;
     std::vector<int> loc_int_pol;
     int num_interior = 0;
     size_t pos = 0;
     loc_int_pol.push_back(0);//We add the initial position to parse.
     while(pos!=std::string::npos){
          if (pos>0) pos=pos+1;
          pos = coordList.find(",",pos);
          if (pos!=std::string::npos) {
               num_interior++;
               loc_int_pol.push_back(pos);
          }
     }
     loc_int_pol.push_back(coordList.size()); //We add the last position to parse.

     std::vector<std::vector<double>> v_aux;
     std::string boundary;
     for (int i=0; i<loc_int_pol.size()-1; i++){
          boundary = "";
          if (i!=0){
```

```
                //We solve the parsing problem with second and next X coordinates.
                boundary = coordList.substr(loc_int_pol[i]+1,loc_int_pol[i+1]-loc_int_pol[i]);
          }else{
                boundary = coordList.substr(loc_int_pol[i],loc_int_pol[i+1]-loc_int_pol[i]);
          }
          result.push_back(parseCoordinatesPolyline(boundary));
     }
     return result;
}
```

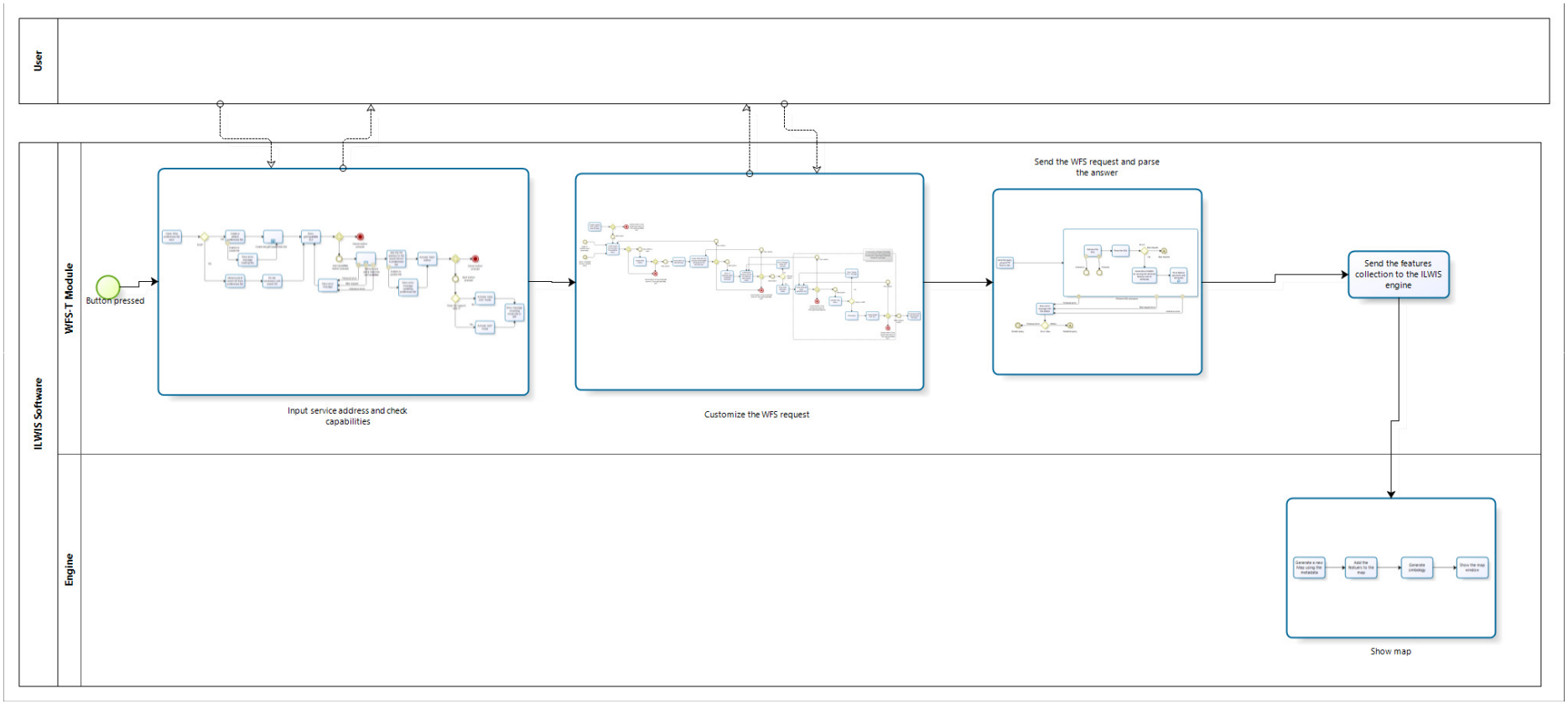
```
//
//
//  MSc. Thesis: Optimizing VGI production using status maps requested through web services
//
//  Official Master's Degree in Software Engineering, Formal Methods and Information Systems
//  Polytechnic University of Valencia, Spain (UPV)
//  Faculty of Geo-Information Science and Earth Observation, The Netherlands (ITC)
//
//  Supervisor UPV: Vicente Pelechano
//  Supervisor ITC: Rob Lemmens
//
//  ---------------------------------------------------------------------------------------
//
//  @ Project : WFS Module for ILWIS
//  @ File Name : Log.hpp
//  @ Date : 30/10/2012
//  @ Author : Francisco Ruiz-Lopez
//  @ Version : 1.1
//
//  ---------------------------------------------------------------------------------------



#if !defined(_LOG_H)
#define _LOG_H

#include <fstream>
#include <vector>

//Implementing Singleton pattern.
class Log {
     public:
          static Log* Instance();
          ~Log();
          void printToLog(const std::string logline);
          void printToLog(const std::string vectorName, const std::vector<std::string> v);
     protected:
          Log(); //Constructor
     private:
          std::ofstream m_stream;
          static Log* pInstance;
};

#endif  //_LOG_H
```

```
//
//
//  MSc. Thesis: Optimizing VGI production using status maps requested through web services
//
//  Official Master's Degree in Software Engineering, Formal Methods and Information Systems
//  Polytechnic University of Valencia, Spain (UPV)
//  Faculty of Geo-Information Science and Earth Observation, The Netherlands (ITC)
//
//  Supervisor UPV: Vicente Pelechano
//  Supervisor ITC: Rob Lemmens
//
//  ---------------------------------------------------------------------------------------
//
//  @ Project : WFS Module for ILWIS
//  @ File Name : Log.cpp
//  @ Date : 30/10/2012
//  @ Author : Francisco Ruiz-Lopez
//  @ Version : 1.1
//
//  ---------------------------------------------------------------------------------------

#include <iostream>
#include <ctime>
#include "Log.hpp"

//Implementing Singleton patron.
Log* Log::pInstance = NULL; //Initialize pointer

Log* Log::Instance() {
     if (pInstance == NULL){ //Is it the first call?
          pInstance = new Log; //Create the instance
     }
     return pInstance; //Return the address of the instance
}

//Constructor
Log::Log() {
     std::cout << "Creating log object 'LogWFS.txt'" << std::endl << std::endl;
}

Log::~Log() {
     std::cout << std::endl << "Destroying log object 'LogWFS.txt'" << std::endl << std::endl;
}

void Log::printToLog(const std::string logline) {
     m_stream.open("LogWFS.txt",std::ios_base::app); //Open file in append mode.
     time_t now = time(NULL);
     struct tm* tmPtr;
     char tmp[40];

     tmPtr = localtime(&now);
     strftime( tmp, 40, "%Y/%m/%d %H:%M:%S", tmPtr );
     //printf( "La hora y fecha locales son: %s\n", tmp );

     //Redirect also to the console, just for development. It could easily changed.
     m_stream << tmp << " - " << logline.c_str() << std::endl;
     m_stream.flush();
     m_stream.close();
```

```
    std::cout << "[LOG] " << tmp << " - " << logline.c_str() << std::endl;
}

void Log::printToLog(const std::string vectorName, const std::vector<std::string> v) {
    m_stream.open("LogWFS.txt",std::ios_base::app); //Open file in append mode.
    time_t now = time(NULL);
    struct tm* tmPtr;
    char tmp[40];

    tmPtr = localtime(&now);
    strftime( tmp, 40, "%Y/%m/%d %H:%M:%S", tmPtr );
    //printf( "La hora y fecha locales son: %s\n", tmp );

    //Redirect also to the console, just for development. It could easily changed.
    m_stream << tmp << " - " << "Vector " << vectorName.c_str() << " [size: " << v.size() << "]" << std::endl
    ;               //File
    std::cout << "[LOG] " << tmp << " - Vector " << vectorName.c_str() << " [size: " << v.size() << "]" << std::
    endl;   //Console
    for (int i=0; i< v.size(); i++)
    {

        m_stream << tmp << "        + " << v[i].c_str() << std::endl;
        m_stream.flush();
        std::cout << "[LOG] " << tmp << "        + " << v[i].c_str() << std::endl;
    }
    m_stream.close();

}
```

17 WFS Layer wizard BPMN diagram

General diagram



Complete diagram A1