SOGETI

XDFL programming guide

Guillaume Baurand
Benjamin Garrigues

XDFL Language

Summary

1. Introduction

Technical considerations
The main characteristics of the XDFL language are:

- *Macroscopic approach*
  One of XDFL's main goals is to prevent the developer from digging into technical issues that have already been implemented a thousand times before.

- *Blocs Assembly*
  Technical issues are implemented once for all in an efficient way and grouped in ready to use blocs, called "nodes". This enables the developer to code using only high-level primitives. XDFL language is the way you make those nodes communicate.

- *"Data stream" approach*
  XDFL nodes are data stream processing units: they take bits of data as input, process them, and send the result to the next node as soon as they are done processing. This enables the data to be processed as a stream. Nodes don't wait for the whole input processing to be done in order to send results to the next node. This way of processing data is low memory consuming (you don't need to store the whole input data in memory between each node, but only bits of it) and particularly adapted for processing huge records of XML data.

- *Extensibility*
  XDFL interpreter is meant to be extensible and is designed as a plug-in architecture. This means that you can code and add your own nodes and integrate them to the XDFL language as new XDFL tags (see below). Your new libraries of nodes will be dynamically loaded at execution time just like the standard ones. This enables you to deal with issues not previously dealt by the engine, and thus extend it. It is also possible to create new nodes made only out of original XDFL code.

Syntax considerations
XDFL is an interpreted language derived from XML. XDFL syntax is a "specialization" of XML grammar and thus inherits from all its features:

- *Strong syntax rules:*
  XML syntax standard describes precise rules for valid ("well-formed") documents. Respect of those simple rules gives a strong structured document, readable by a wide range of actors: humans, text editors, and software using XML parser (almost every languages and every OS has one).

- *Tree-structure*
  XML standard structures documents as trees. Elements of an XML documents are parent or/and child, highest element being the document itself.
  XDFL code takes the shape of a tree in which data streams from leafs to root.

- *XML for data*
  XDFL code syntax is XML, and so is XDFL data. Nodes communicate with each other using XML. They take XML stream as input, and produce XML stream as output.

Note: It is possible to mark streams as raw data stream that should not be interpreted.

1. Basics

1.1. XDFL script

A script is a XML document describing chains of processing nodes through which streams data.

Structure of a script is tree-like: it is composed by nodes, each one of them having one unique parent node, and child nodes.

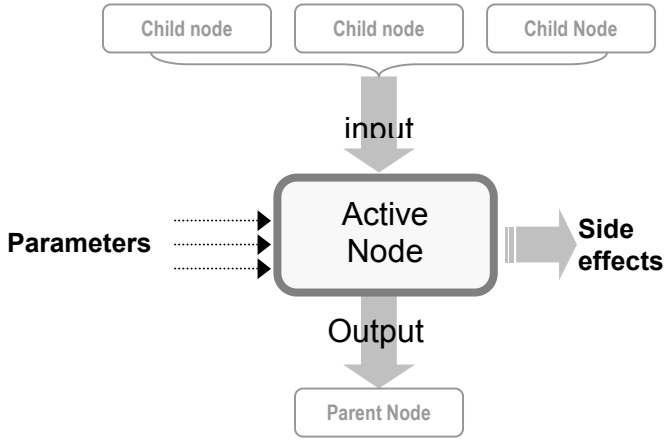
There are two types of nodes:

- **Passive nodes**, corresponding to static XML content, not processed by the interpreter. They are used “as is” by parents nodes.
- **Active nodes**, interpreted by the interpreter to manipulate and process data. Actives nodes are stored in nodes libraries that the interpreter can read.

Active nodes are structured as a tree, and process data, each parent processing data send by the children, and send the result to its own parent. Data flows from leafs to the root.

1.2. Actives Nodes

You can describe an active node as a processing bloc, in the most generic sense of term : it takes parameters, process input data, possibly modify the global state of the system as a side effect and output processing results.



Parameters

Active node parameters control its behavior. They can be scalar values, lists, or references to data stored elsewhere in the code.

Input

Input consists of an arbitrary number of child nodes, either actives or passives or both.

Output

Output of an active node is unique and consists of the result of processed data.

Side effects

Examples of side effects are: file being written on disk, data sent to DBM…
Side effects do not necessarily have impact on output data.

1.3. Syntax

According to the XML standard, a typical node in an XDFL script will look like this:

```
<parent>
        …
        <node_name attribute1= 'value1' attribute2='value2' .. >
                <child1 ..> …</child1 >
                <child2 ..> … </child2 >
…
                <childn ..> … </childn >
        </node_name >
        …
</parent>
```

Passive nodes
All XML nodes except active nodes are considered passive. The only rules that apply to passive nodes are those of the XML standard.
Passive nodes are usually considered as data.

Active nodes
Actives nodes are processing blocs from the XDFL library. A node is considered as an active node if it belongs to the **'xdfl'** namespace.
The name of an active node identifies the processing bloc. It has to be one of the names stored in the nodes library.
XML attribute of an active node give the parameters for the processing. What parameters the node will use depends solely on the node. Order of the parameters doesn't matter.

The node's children give input data. From the node's point of view, input data is the addition of passive children nodes and XML output of active children nodes.
Output data then replace current node's XML from its parent point of view.

Example:

```
<xml xmlns:xdfl='xdfl'>

   <xdfl:F param='val'>
      <a>…</a>
   </xdfl:F>

   <xdfl:G>
      <b>
         <xdfl:H>
            <c>
                …
            </c>
         </xdfl:H>
      </b>
   </xdfl:G>

</xml>
```

***XDFL** Namespace declaration*

***F**,**G** and **H** are the active nodes*

F node is active : it process the input data **<a> … ** and has one parameter **"param"** set to **"val".**

G node is active : its input data is the output of the **H** process inserted between the XML tags ..

Parsed expression
Expressions provide you with a way to insert computed values within the XML stream. The parsed expression is replaced by its value by the XDFL interpreter. General syntax of a parsed expression is :
@@[PREFIX:argument]@@

@@[delimiter marks the start of the parsed expression
@@] delimiter marks the end of the parsed expression

PREFIX **:** stands for the parsed expression's type.
Argument **:** the arguments of the parsed expression.

Example :

`@@[DATE:file]@@`

- **DATE** tells that the expression's type is DATE. Those type of expression return the current date. The format is specified in the argument section.
- **file** tells that the date format has to be adapted to the creation of file names.

This expression thus evaluates to the value : **« 20040429_111903 »**

Parsed expressions are located and evaluated in attribute's values and text nodes of parsed XDFL scripts.

1.4. Execution-time entities

Variables
Variables are scalar values associated to a name. You use the name to retrieve the value. There is no limit to the number of variables you can create, set and get.

Buffers
Buffers provide a way to stock data stream from a specific location inside an XDFL script and retrieve the data from another location in the script. You can get the data stored in the buffer using its name.

Others
Some specific XDFL functionality may use specific entities. For example, XSLT transformation functionality requires to store compiled style sheets.

1.5. Modules, classes and objects

Modules
Modules are equivalent to what most languages call functions :
They are reusable blocs of code that you can call from different places in the XDFL script. Modules work the same way as active nodes do : they take input data from their children, parameters, and return output data to their parent's nodes.

Classes
Classes are meant to file set of modules (those modules becoming the class' methods). All methods of a class are public and can be called in a static way (you don't need to instantiate an object of the class). XDFL offers inheritance mechanism that enables one to derive a class from another and inherit from its functionalities as well as an overloading mechanism to redefine inherited methods.

Objects
Objects in XDFL are instances of XDFL classes : they are entities with their own data and all the classes' methods to process them.

1.6. Execution context

The execution of XDFL scripts relies on **execution context .** An execution context stores the current state of execution for the script, as well as objects used by the script. In short, it stores all kinds of data involved in the script execution. It is created when the script starts, is used during all its execution, then it is destroyed as well as all the data it contains when the script ends.

1.6.1. Execution context stacking

The execution of an XDFL script may involve more than one context. Those contexts are linked to each other by a container/contained relationship: each context may contain other contexts. As a consequence, the visibility and the lifetime of an entity depends on where it stands in this stack of contexts. The lifetime of an entity is the same as its context lifetime. An entity is visible from its context and every context it contains (directly or indirectly).
The choice for the context in which the developer wants to put an entity depends on the use she wants to make of the entity:

- ***'APPLICATION' context***

This context is the XDFL interpreter's one. It is the root context that contains all other contexts. This context is only destroyed when the XDFL interpreter/processor stops.
It is meant to store the static state of the application, such as "global" variables.

- ***'SCRIPT' context***

The context of the script currently being executed: It is created at the start of the script and destroyed at its end. Just like every other context, 'the SCRIPT' context is contained by the 'APPLICATION' context.

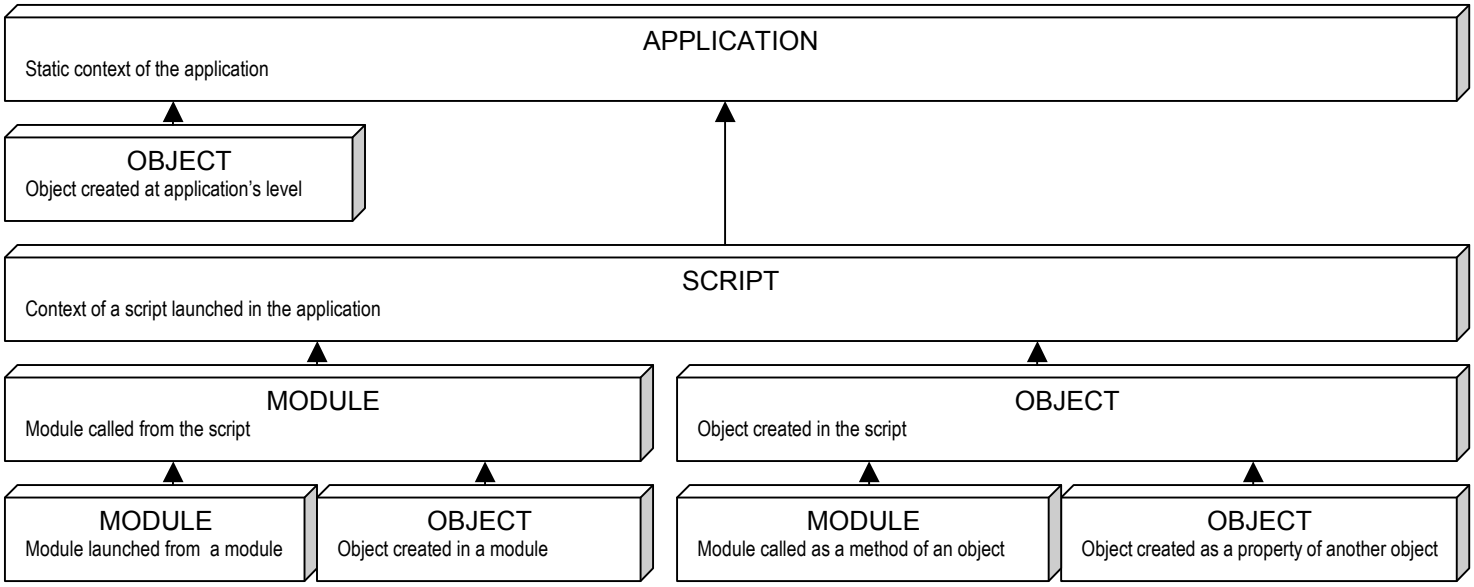
- ***'MODULE' context***

The 'MODULE' context is created every time a module is called and is used by this module as its execution context. It is destroyed when the module execution stops.
If a module is launched from a script, the 'MODULE' context is contained in the 'SCRIPT' context of the calling script.
If a module is launched from another module, then its context is include into the 'MODULE' context of the calling module.
If a module is launched as a method of an object, then its context is contained in the 'OBJECT' context of the object (see below).

- ***'OBJET' context***

An 'OBJET' context is created for every object created. It exists during the entire object's lifetime and is destroyed when the object is destroyed. The context of an object contains all its data.
If the object is created in a script, its context is contained in the 'SCRIPT' context of the script.
If it is created in a module, its context is in the 'MODULE' context of the module.
An object can also be created as a property of another object. Its context is then contained in this other object's context.

Context stacking example



2. Using XDFL

2.1. Launching XDFL script

Command line
You can launch an XDFL using command line shell by calling the XDFLrun utility:
```
XDFLrun script=<xdfl script file> init=<init script file> input=<input file > output=<output file> <arg1>=<val1> ... <argn>=<valn>
```

The only required argument is **script**. It is the name of the file containing the XDFL script you want to launch.
The **init** parameter is used to launch an initialization script that well be launched first. This initialization script usually contains constants, external libraries loading instruction, as well as modules and classes compilation instruction.
Every argument given in the command-line via command-line parameters are available as variables during the script execution.
When a script is launched using command-line, input data are either read from the standard input (default behavior), or from the file given in the **input** parameter. Output data are written on standard output by default, or in the file given in the **file** parameter.

Web application
Launching an XDFL script on a web server is done just like on every other dynamic web platform: an HTTP request invokes the script on the server:
```
http://www.myserver.com/xdfl/<script_xdfl>?<arg1>=<val1>&<argn>=<valn>
```

The request launches the script on the server and returns the result of its execution to the client.
All arguments given by the URL are available during the script's execution as variables.
When a script is launched this way, input data are data found in the HTTP header (POST) sent by the client, and output data will be stored in the web page sent to the client.
Initialization script is not given by the URL but is part of the web server configuration. Elements created in the initialization script will be available throughout every scripts of the web application.

2.2. Creating XDFL script

An XDFL script and XML file. It is editable by any text editors.
In order for the interpreter to identify actives nodes, you have to define the **'xdfl'** namespace and assign a prefix to it (this prefix is usually simply 'xdfl'). This way, every active node will by formed this way: `<xdfl:XDFL_TAG_NAME … />`
For example, the `<xdfl:OUTPUT>…</xdfl:OUTPUT>` tag writes input data on the script's output.

It is now time to show you how the famous *Hello, world !* script looks like in its XDFL version:

```
<?xml version="1.0" encoding="iso-8859-1"?> <!–XML declaration, along with the char set -->

<XDFL xmlns:xdfl="xdfl"> <!–script's root, with namespace declaration -->
        <xdfl:OUTPUT >Hello, world! </xdfl:OUTPUT >
</XDFL>
```

```
Hello, world!
```

2.3. Base language functionalities

2.3.1. Input, Output and log (INPUT, OUTPUT, LOG tags) :

`<xdfl:INPUT />` tag returns input data of the script. It is generally used only once in a script. It doesn't require child node. In fact, data send to this node via its children is simply ignored (and blocked).
`<xdfl:OUTPUT>…</xdfl:OUTPUT>` tag writes its input data into the script's output.
`<xdfl:LOG >…</xdfl:LOG >` tag write its input data to the log file. It is used to visualize data flow at some point in the XDFL script (useful for debugging).

The following script writes input data on the log file then on the output.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <xdfl:OUTPUT > <!– writing to the output -->
                <xdfl:LOG > <!–writing to the log -->
Input:
                        <xdfl:INPUT /> <!–reading input data-->
                </xdfl:LOG >
        </xdfl:OUTPUT >

</XDFL>
```

⬇

```
Input:
Input data
```

Notice the "Input : " before input data. It has been simply written in the source file before the <xdfl:INPUT /> tag, and is reproduced "as is" by the <xdfl:LOG> and <xdfl:OUTPUT> tags.

2.3.2. Variables (VAL tag)

Variables are created and assigned with the `<xdfl:VAL name='name1' value='value' />` XDFL tag. This tag associates the value named 'name1' to the value 'value'. Variables can be assigned as many times as wanted.
Getting the value of a variable can be done in two ways :

- `<xdfl:GET val='name1' />` tag sends the value of the variable *'name1'.*
- Parsed expression `@@[VAL:name1]@@` is replaced by the value of the *name1* variable.

The second way is most used one. It provides you with a way to use variables as XDFL tags parameters.

The following script assign a first variable, copies its content into a second variable. It then outputs the content of the two variables using both possible syntaxes :

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">
        <!-- assignment of 'myvar1' with  value 'val1' -->
        <xdfl:VAL name="myvar1" value="val1"/>
        <!-- assignment of 'myvar2' with value of 'myvar1' -->
        <xdfl:VAL name="myvar2" value="@@[VAL:myvar1]@@"/>

        <xdfl:OUTPUT >  <!-- outputs values of both variables -->
myvar1 : <xdfl:GET val="myvar2"/>
myvar2 : @@[VAL:myvar2]@@
        </xdfl:OUTPUT >

</XDFL>
```

⬇

```
myvar1 : val1
myvar2 : val1
```

It is also possible to assign values using XML stream. This is done by giving the XML stream as input to the <xdfl:VAL> tag. For every text node or attribute found in the XML stream, a variable is automatically created and named after the parent nodes :

```
<xdfl:VAL>
        <a>
                <b>val1</b>
                <c attr='val2'>
                        val3
                </c>
        </a>
</xdfl:VAL>
```

creates the following set of variables : `a.b = val1; a.c.@attr = valeur2; a.c = valeur3`

Example :

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <!-- assignment of variables from a stream -->
        <xdfl:VAL >
                <a>
                        <b>val1</b>
                        <c attr='val2'>
                                val3
                        </c>
                </a>
        </xdfl:VAL >

        <!-- outputs of variables values-->
        <xdfl:OUTPUT >
a.b : @@[VAL:a.b]@@
a.c.@attr : @@[VAL:a.c.@attr]@@
a.c : @@[VAL:a.c]@@
        </xdfl:OUTPUT >

</XDFL>
```

⬇

```
a.b : val1
a.c.@attr : val2
a.c : val3
```

Input Parameters (ARG)

Input parameters given when invoking the XDFL script are available through the use of variables named ARG.*name_of_input_parameter*

Reading those parameters is done the same way other variables are read.

The following script outputs value of 'param' input parameter on the output using both possible syntaxes for reading variable values:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <!-- writes value of parameter 'param' to the output -->
        <xdfl:OUTPUT >
param : <xdfl:GET val="ARG.param"/>
param : @@[VAL:ARG.param]@@
        </xdfl:OUTPUT >

</XDFL>
```

⬇

```
param : value of parameter 'param'
param : value of parameter 'param'
```

2.3.3. Buffers (BUF_SET and BUF_GET tags)

Buffers are used to store data stream at some point of the script in order to use this stream elsewhere in the script. It can also be used as a second output to an active node.
Writing to a buffer is done with the tag :
<xdfl:BUF_SET name='*name_of_buffer*'>…</xdfl :BUF_SET>
If the buffer is used for the very first time, or if you want to initialize/empty it, you must set the parameter 'create' to 1 :
<xdfl:BUF_SET name='*name_of_buffer*' create='1'>…</xdfl :BUF_SET>
Reading the buffer is done with the tag :
<xdfl:BUF_GET name='*name_of_buffer*'/>

Note : It is possible for every XDFL node to redirect its output to a buffer instead of sending it to its parent node. In order to do that, you have to set the parameter 'output' to the name of the buffer : output='*name_of_buffer*'.

Example of how to have some fun with buffers :

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <!-- create a first buffer -->
        <xdfl:BUF_SET name="buffer1" create="1">
                <data attribute="value">
                        <more_xml_data/>
                </data>
        </xdfl:BUF_SET >

        <!-- writes content of the first buffer to the output -->
        <xdfl:OUTPUT >
buffer1:
<xdfl:BUF_GET name="buffer1"/>
        </xdfl:OUTPUT >

        <!-- create a second buffer -->
        <xdfl:BUF_SET name="buffer2" create="1">
                <XML>
                        <!-- writes content of the first buffer into the second -->
                        <xdfl:BUF_GET name="buffer1"/>
                </XML>
        </xdfl:BUF_SET >

        <!-- writes more data  into the first buffer -->
        <xdfl:BUF_SET name="buffer1" >
                <even_more_xml/>
        </xdfl:BUF_SET >

        <!-- create a third buffer -->
        <xdfl:BUF_SET name="buffer3" create="1"/>

        <!-- copies content of the first buffer into the third -->
        <xdfl:BUF_GET name="buffer1" output="buffer3"/>

        <!-- Output content of the last two buffers -->
        <xdfl:OUTPUT >
buffer2:
<xdfl:BUF_GET name="buffer2"/>
buffer3:
<xdfl:BUF_GET name="buffer3"/>
        </xdfl:OUTPUT >

</XDFL>
```

```
buffer1:
                <data attribute="value">
                        <more_xml_data></more_xml_data>
                </data>
buffer2:

                <XML>
                <data attribute="value">
                        <more_xml_data></more_xml_data>
                </data>
                </XML>
buffer3:
                <data attribute="value">
                        <more_xml_data></more_xml_data>
                </data>
```

```
            <even_more_xml></even_more_xml>
```

2.3.4. Passive tags and conditions (PASSIVE tag and IF tag):

-Passive tag

It is possible to prevent a branch of the tree from being compiled by using the tag : <xdfl:PASSIVE >..</xdfl:PASSIVE >.
This tag is taken into account at compile time and forces the interpreter to consider its child nodes as passive XML content (even if some nodes are using the xdfl namespace). The output XML stream of a PASSIVE node/tag is the XDFL code (and not the result of its potential execution).
This tag is mostly used when defining modules, because you want the compile tag to receive the XDFL code to compile, not its result.
<xdfl:PASSIVE /> tag can be seen as a compilation directive. It is the only one in XDFL language.

-Conditions (IF tag)

It is possible to set a condition for the execution of a branch using the tag :
<xdfl:IF val1='value1' val2='value2' op='operator'>..</xdfl:IF >.This tag compares *value1* to *value2* using operator *operator*. Depending on the result of this comparison, the child nodes are executed or not. No stream is sent as output when condition is unmet. Otherwise, the IF tag forwards data sent by its child nodes.

Example

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

	<xdfl:OUTPUT >

		<xdfl:PASSIVE >
			<some_xml>
				<and_some_more />

				<!-- Variable will not be taken into account -->
				<xdfl:VAL name="onevar1" value="data1"/>

			<!-This bloc doesn't compile yet no error is thrown -->
				<xdfl:THIS_XDFL_TAG_DOES_NOT_EXIST>
				</xdfl:THIS_XDFL_TAG_DOES_NOT_EXIST>

			</some_xml>
		</xdfl:PASSIVE >

		<!-- Condition (never met) -->
		<xdfl:IF val1="1" val2="0" op="eq">
			<will_not_be_displayed >
				<xdfl:VAL name="onevar2" value="data2"/>
			</will_not_be_displayed>
		</xdfl:IF >

		<!-- Condition (always met) -->
		<xdfl:IF val1="1" val2="1" op="eq">
			<will_be_displayed >
				<xdfl:VAL name="onevar3" value="data3"/>
			</will_be_displayed >
		</xdfl:IF >

onevar1 : @@[VAL:onevar1]@@
onevar2 : @@[VAL:onevar2]@@
onevar3 : @@[VAL:onevar3]@@

	</xdfl:OUTPUT >

</XDFL>
```

```
			<some_xml>
				<and_some_more></and_some_more>
				<xdfl:VAL name="onevar1" value="data1"></xdfl:VAL>
				<xdfl:THIS_XDFL_TAG_DOES_NOT_EXIST>
				</xdfl:THIS_XDFL_TAG_DOES_NOT_EXIST>
			</some_xml>
			<will_be_displayed>
			</will_be_displayed>
onevar1 :
onevar2 :
onevar3 : data3
```

2.3.5. Doing nothing (NULL tag) :

The <xdfl:NULL>..</xdfl:NULL> tag does nothing : This tag just forwards input data to its output : It is useful in cases where you only want to use a functionality common to every nodes (such as output parameter). In other cases you may want to use the NULL tag to ensure correct XML syntax without interfering with data.

Example

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <xdfl:BUF_SET name="buffer" create="1" />

        <xdfl:OUTPUT >

                <xdfl:NULL >
                                <some_xml>
                                        <blah/>...<blah/>...<blah/>
                                </some_xml>
                </xdfl:NULL >

                <!-- Write into the buffer -->
                <xdfl:NULL output="buffer">
                                <some_more_xml>
                                        <data/>
                                </some_more_xml>
                </xdfl:NULL >
buffer:
                <xdfl:BUF_GET name="buffer" />
        </xdfl:OUTPUT >

</XDFL>
```

```
                                <some_xml>
                                        <blah></blah>...<blah></blah>...<blah></blah>
                                </some_xml>
buffer:
                                <some_more_xml>
                                        <data></data>
                                </some_more_xml>
```

2.3.6. Errors – raising and handling them (RAISE_ERROR tag)

Whenever an active node errs, it produces an XML message describing the error. If the error is not handled the script is stopped.
The `<xdfl:RAISE_ERROR error_number='errno' message='msg' />` tag raises an error with error number *errno* and error message *msg*.

Example

```
<?xml version="1.0" encoding="iso8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <!-- Raise an error-->
        <xdfl:RAISE_ERROR number="666" message="Catastrophic failure" />

</XDFL>
```

This scripts stops with message error (printed in log file) being :

```
ERROR #666 in RaiseErrorStreamer::initStream : Catastrophic failure ()
```

Catching an error thrown by a node is made by possible by setting the "`error_fatal`" parameter to "0" on one of its parents. The node with parameter "`error_fatal`" set to "0" will then be responsible for handling the error because the error will no longer propagate to parent nodes (see below).
XML message for describing an error is like this :

```
<ERROR>
        <number> error number </number>
        <message> message </message>
        <localisation> localisation in the interpreter </localisation>
        <context> complementary message </context>
        <cause> complementary message </cause>
</ERROR>
```

The message can be stored in a buffer using the `output_error='buffer_name'` parameter on the node that raised the error or the node that catches it. The XML error message/stream is redirected to the buffer *buffer_name*.

Example

```
<?xml version="1.0" encoding="iso8859-1"?>

<XDFL xmlns:xdfl="xdfl">
        <xdfl:BUF_SET name="error_buffer" create="1" />

        <!-- NULL is used as an error-catching node -->
        <xdfl:NULL error_fatal="0" output_error="error_buffer">

                <!-- VAL expects a DOCUMENT for input -->
                <!-- by finding two roots it will raise an error -->
                <xdfl:VAL >
                        <docroot1/>
                        <docroot2/>
                </xdfl:VAL >
        </xdfl:NULL >

        <!-- Decomposing the error stream into variables -->
        <xdfl:VAL >
                <xdfl:BUF_GET name="error_buffer" />
        </xdfl:VAL >

        <xdfl:OUTPUT >
                <xdfl:BUF_GET name="error_buffer" />
Error No:@@[VAL:ERROR.number]@@ reason : @@[VAL:ERROR.message]@@
        </xdfl:OUTPUT >

</XDFL>
```

```
<ERROR>
        <number>5</number>
        <message><![CDATA[Fatal XML Error at (3,4):
Expected comment or processing instruction
]]></message>
        <localisation><![CDATA[XDFLSaxErrorReporter]]></localisation>
        <context><![CDATA[SaxStreamer]]></context>
        <cause><![CDATA[]]></cause>
</ERROR>
```

```
Error No:5 reason : Fatal XML Error at (3,4):
Expected comment or processing instruction
```

2.3.7. System commands call (SYSTEM tag)

The <xdfl:SYSTEM> …</xdfl:SYSTEM> tag execute system commands contained in the XML input stream. It extracts text nodes of the input XML stream and executes them as system commands. Structure of the input XML stream doesn't matter, all text nodes will be executed.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <xdfl:SYSTEM >
                <commands>
                        <!-- print system version -->
                        <command>ver</command>

                        <!-- current date -->
                        <command>date /T</command>
                </commands>
        </xdfl:SYSTEM >

</XDFL>
```

```
Microsoft Windows 2000 [Version 5.00.2195]
ven. 02/04/2004
```

2.3.8. Loading external libraries (_MODULE_LOAD , _ALIAS tags)

XDFL interpreter can load external libraries (or plug-ins) in order to add new tags and new functionalities. External libraries are dynamic libraries (.dll under Windows, .so under UNIX)

Loading dynamic libraries is done in the initialization script using tag `<xdfl:_MODULE_LOAD path='library' />`

It is also possible to create alias for XDFL tags :
`<xdfl:_ALIAS name='original_name' alias='alias'/>`

The following script loads the examples external library and the XDFL SAMPLE tag (that does nothing). It then creates an alias for this tag called SAMPLE2.
Initialization script (in the initialization file):

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <!-- loading library -->
        <xdfl:_MODULE_LOAD path="../../build/win32.1.0/sampleModuleStreamer_win32.1.0.dll" />

        <!-- creating alias -->
        <xdfl:_ALIAS name="SAMPLE" alias="SAMPLE2" />

</XDFL>
```

script

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <!-- XDFL tag imported from the library-->
        <xdfl:SAMPLE />

        <!-- XDFL tag aliased from SAMPLE -->
        <xdfl:SAMPLE2/>

</XDFL>
```

2.4. Modules, classes and objects

2.4.1. Modules (_MODULE_COMPILE, MODULE_EXEC tags)

An XDFL module is the equivalent of a function. Modules work the same way a active nodes does : it contains blocs of codes that receive input data and parameters and return processed data as output. Input data of a module is accessed using the <xdfl :INPUT/> tag. Output is the root of the module : all data reaching the root of the module are written on its output.

2.4.2. Compiling

The `<xdfl:_MODULE_COMPILE name='name_of_module'> </xdfl:MODULE_COMPILE>` tag compile the input data (in this case XDFL code) as a module and associate this module to the name *name_of_module*.
In order to prevent the interpreter from trying to execute the source code of the module before trying to compile it, it is most of times surrounded by `<xdfl:PASSIVE>..</xdfl:PASSIVE>` tag.
An important note is that the XDFL code of the module has to be like every other XDFL script : it has only one root, and must declare the xdfl namespace. That is why the XDFL code of the module is also surrounded by `<xdfl:NULL xmlns:xdfl='xdfl'>..</xdfl:NULL>`

2.4.3. Executing

The `<xdf:MODULE_EXEC name='name_of_module'>…</xdfl:MODULE_EXEC>` tag invoke the *name_of_module* module. Output of this tag is the output of the invoked module. Input data is given the same way as for every other active node.

Arguments can be transmitted to the module by using additional attributes to the MODULE_EXEC tag. Those arguments will be accessible from within the module through the ARG.*argument_name* variable (just like in a regular script).

Example

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <!-- Compilation of module -->
        <xdfl:_MODULE_COMPILE name="gen_envelope">
                <xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl">

                        <xdfl:IF val1="@@[VAL:ARG.type]@@" val2="XML">
                                <XML >
                                        <xdfl:INPUT />
                                </XML>
                        </xdfl:IF >

                        <xdfl:IF val1="@@[VAL:ARG.type]@@" val2="DATA">
                                <DATA >
                                        <xdfl:INPUT />
                                </DATA >
                        </xdfl:IF >

                </xdfl:NULL ></xdfl:PASSIVE >
        </xdfl:_MODULE_COMPILE >

        <!-- execution -->
        <xdfl:OUTPUT >
                <xdfl:MODULE_EXEC name="gen_envelope" type="DATA">
                        <xdfl:MODULE_EXEC name="gen_envelope" type="XML">
                                <some_xml>
                                        <data/>
                                </some_xml>
                        </xdfl:MODULE_EXEC >
                </xdfl:MODULE_EXEC >
        </xdfl:OUTPUT >

</XDFL>
```

```
<DATA>
<XML>
```

```
<some_xml>
        <data></data>
</some_xml>
</XML>
</DATA>
```

2.4.4. Classes (_CLASS_DECLARE tag)

Classes are basically set of modules. XDFL classes only have methods and do not allow you to define properties :
A class declaration looks like this :
`<xdfl:_CLASS_DECLARE class='myclass' extends='parent' />`
Parameter extends='parent' makes the class inherit from all the modules in the 'parent' class.

In order to add modules/methods to a class, one just need to compile modules using the 'class' parameter :
`<xdfl:_MODULE_COMPILE class='myclass' name='method'></xdfl:MODULE_COMPILE>`

Calling methods of a class from a script or a module not belonging to this class or a derived class is done by giving the name of the class in the 'class' parameter :
`<xdf:MODULE_EXEC class='myclass' name='method'>…</xdfl:MODULE_EXEC>`

In order to call methods of a class from a module that belongs to this class, or calling a parent's class method, you need to use the **ancestor** parameter (this parameter is not compulsory if you're already in the class):
`<xdf:MODULE_EXEC ancestor='parent' name='method'>…</xdfl:MODULE_EXEC>`

All methods are public. A class that inherits from another one inherits from all its methods.

Classes are "static" : you don't need to instantiate an object to invoke the classes' methods.

Example :

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

	<!-- Class 1 -->
	<xdfl:_CLASS_DECLARE class="class1" />

		<xdfl:_MODULE_COMPILE class="class1" name="moduleA">
			<xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl">

				<xdfl:MODULE_EXEC name="moduleB" >
					<xdfl:INPUT />
				</xdfl:MODULE_EXEC >

			</xdfl:NULL ></xdfl:PASSIVE >
		</xdfl:_MODULE_COMPILE >

		<xdfl:_MODULE_COMPILE class="class1" name="moduleB">
			<xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl">

				<class1><xdfl:INPUT /></class1>

			</xdfl:NULL ></xdfl:PASSIVE >
		</xdfl:_MODULE_COMPILE >


	<!-- Class 2 -->
	<xdfl:_CLASS_DECLARE class="class2" extends="class1"/>

		<xdfl:_MODULE_COMPILE class="class2" name="moduleB">
			<xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl">

				<class2><xdfl:INPUT /></class2>

			</xdfl:NULL ></xdfl:PASSIVE >
		</xdfl:_MODULE_COMPILE >


		<xdfl:_MODULE_COMPILE class="class2" name="moduleC">
			<xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl">

				<xdfl:MODULE_EXEC ancestor="class1" name="moduleB" >
					<xdfl:INPUT />
				</xdfl:MODULE_EXEC >

			</xdfl:NULL ></xdfl:PASSIVE >
		</xdfl:_MODULE_COMPILE >

	<!-- Execution -->
	<xdfl:OUTPUT >

		<xdfl:MODULE_EXEC class="class1" name="moduleA" >
			<class1_module_a/>
		</xdfl:MODULE_EXEC >

		<xdfl:MODULE_EXEC class="class2" name="moduleA" >
			<class2_module_a/>
		</xdfl:MODULE_EXEC >

		<xdfl:MODULE_EXEC class="class2" name="moduleC" >
			<class2_module_c/>
		</xdfl:MODULE_EXEC >

	</xdfl:OUTPUT >

</XDFL>
```

```
<class1>
	<class1_module_a></class1_module_a>
</class1>

<class2>
	<class2_module_a></class2_module_a>
</class2>

<class1>
	<class2_module_c></class2_module_c>
</class1>
```

2.4.5. Objects (CLASS_CREATEOBJECT tag) :

XDFL classes can be instantiates to create objects. Object creation allocates a context containing data for the object. Methods of the class can access this context in order to manipulate the object's data.
An object can contains every type of data a context can store, no matter what the class is.

Instantiation of a class *myclass* to create an object is done using the XDFL tag :
```
<xdfl:CLASS_CREATEOBJECT class="class" object="object" />
```
When the object is created, the init method of the class is called with all arguments given in the CLASS_CREATE_OBJECT tag. The same way a constructor is called in traditional object programming.

Method invocation on an object is done using **object** parameter on the MODULE_EXEC tag :
```
<xdf:MODULE_EXEC object='object' name='method'>…</xdfl:MODULE_EXEC>
```
Note : You don't need this parameter if you're calling the method from within the object.

Changing the context to the object's context from inside a method is done using the **context** parameter (this parameter works with all XDFL tags). See below for an example of how to use this parameter.

Example

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

	<!-- Class 1 -->

	<xdfl:_CLASS_DECLARE class="class1" />

		<xdfl:_MODULE_COMPILE class="class1" name="init">
			<xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl">

				<xdfl:BUF_SET context="OBJECT" create="1" name="object_buffer">
					<xdfl:INPUT />
				</xdfl:BUF_SET >

				<xdfl:VAL context="OBJECT" name="prop" value="(NULL)"/>

			</xdfl:NULL ></xdfl:PASSIVE >
		</xdfl:_MODULE_COMPILE >

		<xdfl:_MODULE_COMPILE class="class1" name="set_prop">
			<xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl">

			<xdfl:VAL context="OBJECT" name="prop" value="@@[VAL:ARG.val]@@"/>

			</xdfl:NULL ></xdfl:PASSIVE >
		</xdfl:_MODULE_COMPILE >

		<xdfl:_MODULE_COMPILE class="class1" name="get_data">
			<xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl">

				<xdfl:BUF_GET name="object_buffer"/>

			</xdfl:NULL ></xdfl:PASSIVE >
		</xdfl:_MODULE_COMPILE >

		<xdfl:_MODULE_COMPILE class="class1" name="get_xml">
			<xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl">

				<object_class1 prop="@@[VAL:prop]@@">
					<xdfl:MODULE_EXEC name="get_data"/>
				</object_class1>

			</xdfl:NULL ></xdfl:PASSIVE >
		</xdfl:_MODULE_COMPILE >

	<!-- object1 -->

	<xdfl:CLASS_CREATEOBJECT class="class1" object="object1">
		<object1>
			<data/>
		</object1>
	</xdfl:CLASS_CREATEOBJECT >

	<xdfl:MODULE_EXEC object="object1" name="set_prop" val="1"/>

	<!-- Execution -->

	<xdfl:OUTPUT >

		<xdfl:MODULE_EXEC object="object1" name="get_xml" />

	</xdfl:OUTPUT >

</XDFL>
```

```
<object_class1 prop="1">
	<object1>
		<data></data>
	</object1>
</object_class1>
```

2.5. Working with files

2.5.1. File Reading / Writing (FS_GET , FS_SET tags)

`<xdfl:FS_GET target= 'file_path' />` tag outputs content of the indicated file.
`<xdfl:FS_SET target='file path'>…</xdfl :FS_SET>` tag writes input data to the indicated file. It also forwards input data unchanged to its output.

The following script copies the file given in the **org** parameter into the file given in the **dest** parameter :

```
<?xml version="1.0" encoding="iso-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <!-- Read from org, writes to dest -->
<xdfl:FS_SET target="@@[VAL:ARG.dest]@@"><xdfl:FS_GET target="@@[VAL:ARG.org]@@"/></xdfl:FS_SET>

        <xdfl:OUTPUT >File @@[VAL:ARG.org]@@ copied to @@[VAL:ARG.dest]@@</xdfl:OUTPUT >

</XDFL>
```

2.5.2. Directory browsing (FS_FIND tag)

The `<XDFL:FS_FIND path='path'/>` tag returns an XML stream describing the directory or file indicated in the **path** parameter. Setting parameter **recurse** to **'1'**, makes the find recursive.

The next script browse the directory indicated by the path parameter using the expression given in the **glob** parameter as a filter (glob unix).

```
<?xml version="1.0" encoding="iso8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <xdfl:OUTPUT >

             <xdfl:FS_FIND path="@@[VAL:ARG.path]@@" recurse="1" glob="@@[VAL:ARG.match]@@"/>

        </xdfl:OUTPUT >

</XDFL>
```

```
<directory path=".." fullname="11-fs_find" name="11-fs_find">
        <file path="..\11-fs_find" fullname="fs_find.bat" name="fs_find" ext="bat"/>
        <file path="..\11-fs_find" fullname="fs_find.log" name="fs_find" ext="log"/>
        <file path="..\11-fs_find" fullname="fs_find.xdfl" name="fs_find" ext="xdfl"/>
</directory>
```

(in this example, no filter was provided)

2.6. XPATH/XSLT functionalities

2.6.1. XPATH Requests

XPATH is a language standardized by the W3C made for selecting nodes in an XML document.
The XDFL tag `<xdfl:XPATH_SELECT path='xpath_request'>…</xdfl :XPATH_SELECT>` returns nodes selected by the xpath_request request in the input XML document/stream.

The following script execute the xpath request on the input of the script :

```
<?xml version="1.0" encoding="iso8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <xdfl:OUTPUT >
                <xdfl:XPATH_SELECT path="@@[VAL:ARG.path]@@" >
                        <xdfl:INPUT/>
                </xdfl:XPATH_SELECT>
        </xdfl:OUTPUT >

</XDFL>
```

```
<order_entry fk_order="3" fk_product="2">
    <quantity>3</quantity>
    <product id="2">
     <name>Computer</name>
     <description>A small computer</description>
     <price>1000</price>
    </product>
</order_entry>
```

(xpath request being : DATA/client/order/order_entry[quantity > 2])

2.6.2. XSLT Transform (XSL_COMPILE, XSL_TRANSFORM tags)

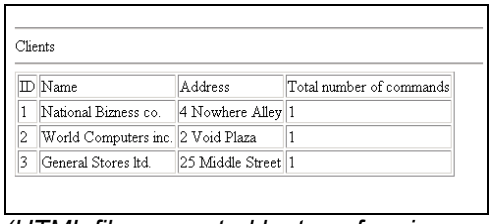
XSLT is a standard language used to transform XML streams.

The XDFL tag `<xdfl:XSL_COMPILE name='name_of_xsl'>…</xsl:COMPILE>` takes an XML document for input representing and XSL sheet, compile it, and makes it available in the current context under the name *name_of_xsl*.

The XDFL tag `<xdfl:XSL_TRANSFORM name='name_of_xsl'>…<xdfl:XSL_TRANSFORM>` takes a XML document for input, transform it using the compiled XSL sheet called *name_of_xsl* and return the result of the transformation.

The following example transforms the document given in the script's input using the XSL sheet in the file **xsl**. It then returns the result of the transformation to the script's output.

```
<?xml version="1.0" encoding="iso8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <!-- Read XSL file and compile it -->
<xdfl:XSL_COMPILE name="xsl_sheet"><xdfl:FS_GET target="@@[VAL:ARG.xsl]@@"/></xdfl:XSL_COMPILE>

        <!-- Transform the input, send result to output -->
        <xdfl:OUTPUT >

                <xdfl:XSL_TRANSFORM name="xsl_sheet" ><xdfl:INPUT/></xdfl:XSL_TRANSFORM >

        </xdfl:OUTPUT >

</XDFL>
```



Clients

ID	Name	Address	Total number of commands
1	National Bizness co.	4 Nowhere Alley	1
2	World Computers inc.	2 Void Plaza	1
3	General Stores ltd.	25 Middle Street	1

(HTML file generated by transforming example data with example XSL sheet)

2.7. Working with javascript

2.7.1. Using javascript (JS_COMPILE, JS_EXEC tags)

XDFL engine makes it possible for the developer to integrate javascript code in an XDFL script.

The `<xdfl:JS_COMPILE>…</xdfl :JS_COMPILE>` tag takes a text stream containing the javascript for input and compile it. Functions contained in the javascript code are then available in the current context.
The `<xdfl:JS_EXEC call='function_js(args)' >…</xdfl :JS_EXEC>` tag executes the *function_js(args)* javascript function previously compiled with JS_COMPILE on its input stream.

In order to make it possible for javascript to interact with the XDFLengine, a javascript object called **XDFL**, is accessible in the javascript code. Its functionalities are :

- **XDFL.input()** : returns input document as a string.
- **XDFL.output(text)** : writes string 'text' on the output of the calling JS_EXEC node.
- **XDFL.arg(argname)** : returns the value of the *argname* parameter given in the JS_EXEC calling tag.
- **XDFL.getValue(valname)** : returns the value of the XDFL variable called *valname*.
- **XDFL.setValue(valname, value)** : assign *value* to the XDFL variable *valname*.
- **XDFL.log(text)** : writes *text* in the log file.

Example

```
<?xml version="1.0" encoding="iso8859-1"?>

<XDFL xmlns:xdfl="xdfl">

	<!-- JS script compilation -->
	<xdfl:JS_COMPILE >
		<![CDATA[
		function makeNodes()
		{
			OB = String.fromCharCode("60")
			CB = String.fromCharCode("62")
			XDFL.output( OB+"nodes" + CB+ "\n")
			for(i=0; XDFL.arg("num")!=i; i++)
			{
				XDFL.output("\t"+OB+"node"+i+"/"+CB+"\n");
			}
			XDFL.output( OB+"/nodes" + CB+ "\n")
		}

		function getStreamLength()
		{
			XDFL.setValue("length", XDFL.input().length)
			XDFL.output(XDFL.input())
		}
		]]>
	</xdfl:JS_COMPILE >

	<!-- Execution ... -->
	<xdfl:OUTPUT >
		<XML>
		<xdfl:JS_EXEC call="getStreamLength()">
			<xdfl:JS_EXEC call="makeNodes()" num="10"/>
		</xdfl:JS_EXEC >
		<length>@@[VAL:length]@@</length>
		</XML>
	</xdfl:OUTPUT >

</XDFL>
```

```
<XML>
    <nodes>
       <node0/><node1/><node2/><node3/><node4/><node5/><node6/><node7/><node8/><node9/>
    </nodes>
    <length>107</length>
</XML>
```

2.7.2. Using the jsMicroDOM model

Parse_dom parameter of the `<xdfl:JS_EXEC parse_dom='1'>` tag says whether the input document is available for the js script as a DOM (*Document Object Model)* document. In this case, the input document is parsed and made accessible using a subset of the DOM (jsMicroDom) and XPATH (jsMicroXpath).

If **parse_dom** parameter is set to 1, more methods of the XDFL object are available :

- **XDFL.XMLinput()** : returns input document as a jsMicroDOM document.
- **XDFL.outputXML(node)** : writes the XML stream corresponding to the jsMicroDOM node *node* and its child nodes on the output of the calling JS_EXEC node.
- **XDFL.select(path)** : executes the jsMicroXpath *path* request on the input and returns the value of the node.
- **XDFL.selectSingleNode(node, path)** : returns the first node corresponding to the *path* jsMicroXpath request starting at the *node* node.
- **XDFL.selectNodes(node, path) :** returns a collection of nodes corresponding to the *path* jsMicroXpath request starting at the *node* node.

The jsMicroDOM model represents a XML document as an XMLDOMElement objects tree. A XMLDOMElement node may have a XMLDOMElement parent node, a collection of XMLDOMElement child nodes, a collection of named attributes, and text content.
XMLDOMElement classes has the following attributes and functions:

- **XMLDOMElement.nodeName** : name of the node.
- **XMLDOMElement.parentNode** : XMLDOMElement parent node.
- **XMLDOMElement.attributes** : collection of named attributes.
- **XMLDOMElement.childNodes** : collection of child XMLDOMElement nodes.
- **XMLDOMElement.nodeValue** : content of the text node.
- **XMLDOMElement.appendChild(node)** : append the child node *node* and returns this node.
- **XMLDOMElement.removeChild(node)** : remove the child node *node* and returns this node.
- **XMLDOMElement.replaceChild(newnode ,node)** : replace *node* by *newnode* and return this node.
- **XMLDOMElement.getAttribute(name)** : return *name* attribute.
- **XMLDOMElement.setAttribute(name, value)** : set value of attribute *name* to *value*.
- **XMLDOMElement.removeAttribute(name)** : remove attribute *name*.

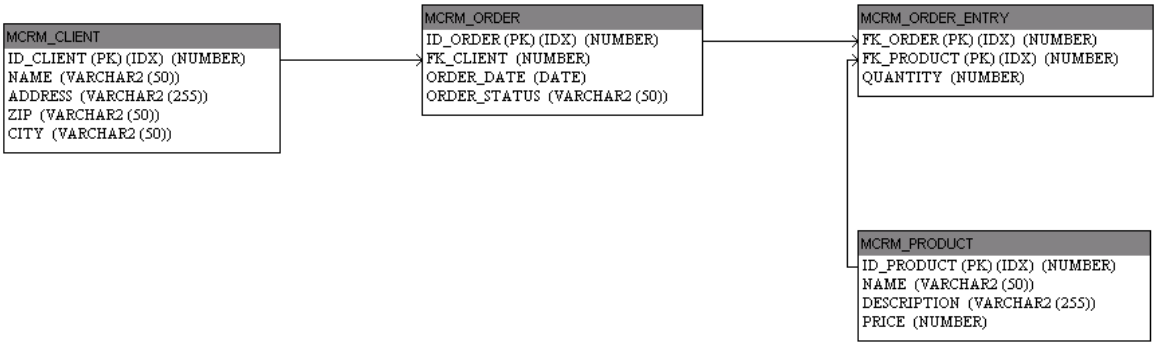
Example

```
<?xml version="1.0" encoding="iso8859-1"?>

<XDFL xmlns:xdfl="xdfl">

	<!-- compilation of the JS script-->
	<xdfl:JS_COMPILE >
		<![CDATA[
function sumOrders()
{
	try{
	xmlOrderEntries = XDFL.selectNodes( XDFL.XMLinput(),"client/order/order_entry")

	intNumOrders = xmlOrderEntries.length
	intTotalPrice = 0

	for(i=0; i != intNumOrders-1; i++)
	{
		xmlOrderEntry = xmlOrderEntries[i]
		XDFL.log(xmlOrderEntry.getAttribute("fk_order"))
		intPrice = XDFL.selectSingleNode(xmlOrderEntry, "product/price").nodeValue

		intQuant = XDFL.selectSingleNode(xmlOrderEntry, "quantity").nodeValue
		intTotalPrice += intPrice*intQuant
	}

	intAvgPrice = intTotalPrice / intNumOrders

	xmlOutputTotal=new XMLDOMElement("XML")
	xmlOutputTotal.appendChild(new XMLDOMElement("total")).nodeValue = intTotalPrice
	xmlOutputTotal.appendChild(new XMLDOMElement("number")).nodeValue = intNumOrders
	xmlOutputTotal.appendChild(new XMLDOMElement("average")).nodeValue = intAvgPrice

	XDFL.outputXML( xmlOutputTotal)
	} catch(e){XDFL.log(e)}
}
		]]>
	</xdfl:JS_COMPILE >

	<xdfl:OUTPUT >
		<!-- Execution ... -->
		<xdfl:JS_EXEC call="sumOrders()" parse_dom="1">
			<xdfl:FS_GET target="../sample_data.xml"/>
		</xdfl:JS_EXEC >
	</xdfl:OUTPUT >

</XDFL>
```

```
<XML>
<total><![CDATA[698000]]></total>
<number><![CDATA[9]]></number>
<average><![CDATA[77555.55555555556]]></average></XML>
```

2.8. Working with Database in record mode

Examples given in this section are based on the following data model:



2.8.1. Loading database access tags from libraries

Database access tags are imported by loading external libraries. Choice of the library depends on the database access API (or “connector”) and the operating system :
For example :
OCI/Oracle 8i for Win32 → *dbModuleStreamer_ora8i_win32.1.0.dll*
ODBC for Win32 → *dbModuleStreamer_odbc_win32.1.0.dll*
OCI/Oracle 8 for HP-UX 11.00 → *libdbModuleStreamer_ora8i_HP-UXB.11.00.1.0.sl*
ODBC-UNIX for Linux → *libdbModuleStreamer_odbcx_Linux_All.1.0.so*

Database access libraries take two parameters when loaded :
connection_life_time : lifetime of a connection in the connection pool (in milliseconds)
sql_syntax : SQL syntax (ORA, MSSQL, MYSQL)

Each library imports the same set of XDFL tags (*SQL*, *DB_GET*, *DB_SET, DB_AUTODESC, _DBDEF_COMPILE*) but with a different suffix .
For example, with the OCI / Oracle8i library, those tags will be *SQL_ORA8I, DB_GET_ORA8I, DB_SET_ORA8I, DB_AUTODESC_ORA8I, _DBDEF_COMPILE_ORA8I*.
If you want your code to be independent from the name of the database library used, you have to create alias for each and every specific XDFL tag :
Example :
Loading libraries and alias creation are usually made in the initialization script. This example will be used as the initialization script for all the following examples.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

       <xdfl:VAL context="APPLICATION" name="BINS" value="../../build/win32.1.0/" />

       <xdfl:_MODULE_LOAD path=" @@[VAL:BINS]@@/dbModuleStreamer_ora8i_win32.1.0.dll"
                                          connection_life_time="360000" sql_syntax="ORA"/>

       <xdfl:_ALIAS name="SQL_ORA8I" alias="SQL"/>
       <xdfl:_ALIAS name="DB_GET_ORA8I" alias="DB_GET"/>
       <xdfl:_ALIAS name="DB_SET_ORA8I" alias="DB_SET"/>
       <xdfl:_ALIAS name="DB_AUTODESC_ORA8I" alias="DB_AUTODESC"/>
       <xdfl:_ALIAS name="_DBDEF_COMPILE_ORA8I" alias="_DBDEF_COMPILE"/>

</XDFL>
```

2.8.2. Executing SQL request for extraction (SQL tag)

Access to database in record-set mod is done via tabular XML streams :

The following table

ID	Nom	Address	ZIP	City
1	National Bizness co.	3 Nowhere Alley	94034	St Cloud
2	World Computers inc.	2 Void Plaza	94334	Paris
3	General Stores ltd.	25 Middle Street	34400	St Cloud

Is represented by the following tabular XML stream :

```
<DATA>
        <ROW>
                        <ID_CLIENT>1</ID_CLIENT>
                        <NAME>National Bizness co.</NAME>
                        <ADDRESS>3 Nowhere Alley</ADDRESS>
                        <ZIP>94034</ZIP>
                        <CITY>St Cloud</CITY>
        </ROW>
        <ROW>
                        <ID_CLIENT>2</ID_CLIENT>
                        <NAME>World Computers inc.</NAME>
                        <ADDRESS>2 Void Plaza</ADDRESS>
                        <ZIP>94334</ZIP>
                        <CITY>Paris</CITY>
        </ROW>
        <ROW>
                        <ID_CLIENT>3</ID_CLIENT>
                        <NAME>General Stores ltd.</NAME>
                        <ADDRESS>25 Middle Street</ADDRESS>
                        <ZIP>34400</ZIP>
                        <CITY>St Cloud</CITY>
        </ROW>
</DATA>
```

`<xdfl:SQL connection='dsn' statement='sql_statement'>…</xdfl :SQL>` tag executes the *sql_statement* SQL statement on the database using *dsn* as the connection string.
enclose_record parameter is the name of the tag surrounding every record.
streamrecords parameter sets the number of records sent at the same time.

You can customize the output XML stream structure by giving the following prefixes for the extracted columns in the “as” section of the SQL request :
XML_ATTR_*attribute* → creates attribute *attribute*.
XML_OPEN_*tag* → opens a tag named *tag*
XML_CLOSE → closes the last opened tag
XML_TEXT → creates a text node
XML_CDATA → creates a CDATA section

Example : the following script executes *sql* SQL requests on the *dsn* database and sends resulting records as XML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <xdfl:OUTPUT >
                <DATA>
                <xdfl:SQL connection="@@[VAL:ARG.dsn]@@" statement="@@[VAL:ARG.sql]@@"
                                                        streamrecords="100" enclose_record="ROW"/>
                </DATA>
        </xdfl:OUTPUT >

</XDFL>
```

```
<ROW>
        <ID_CLIENT>1</ID_CLIENT>
        <NAME><![CDATA[National Bizness co.]]></NAME>
        <ADDRESS><![CDATA[3 Nowhere Alley]]></ADDRESS>
        <ZIP><![CDATA[94034]]></ZIP>
        <CITY><![CDATA[St Cloud]]></CITY></ROW>
<ROW>
        <ID_CLIENT>2</ID_CLIENT>
        <NAME><![CDATA[World Computers inc.]]></NAME>
        <ADDRESS><![CDATA[2 Void Plaza]]></ADDRESS>
        <ZIP><![CDATA[94334]]></ZIP>
        <CITY><![CDATA[Paris]]></CITY></ROW>
<ROW>
        <ID_CLIENT>3</ID_CLIENT>
        <NAME><![CDATA[General Stores ltd.]]></NAME>
        <ADDRESS><![CDATA[25 Middle Street]]></ADDRESS>
        <ZIP><![CDATA[34400]]></ZIP>
        <CITY><![CDATA[St Cloud]]></CITY></ROW>
```

*(with sql=select * from MCRM_CLIENT)*

```
<ROW>
        <CLIENT id="1"><![CDATA[National Bizness co.]]></CLIENT>
</ROW>

<ROW>
        <CLIENT id="2"><![CDATA[World Computers inc.]]></CLIENT>
</ROW>

<ROW>
        <CLIENT id="3"><![CDATA[General Stores ltd.]]></CLIENT>
</ROW>
```

(with sql=select 0 as XML_OPEN_CLIENT ,id_client as XML_ATTR_id, NAME as XML_CDATA from MCRM_CLIENT)

2.8.3. Using SQL tag for writing

<xdfl :SQL ../> tag can also writes data if given an INSERT,UPDATE,DELETE type of SQL request (or even a call to a stored procedure).
You can also use linked variables that will be replaced by input stream at execution time.
This solution is more efficient when you have to execute huge numbers of inserts. It requires the use of an input XML stream having a tabular structure. Each “line” of the input provides the request with a set of data for execution.
Syntax for linked variable is **:***name*<*type*> (double dot is part of the syntax)

The following script insert input data in the *dns* database by executing the *sql* SQL request :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <xdfl:OUTPUT >
                <DATA>
                <xdfl:SQL connection="@@[VAL:ARG.dsn]@@" statement="@@[VAL:ARG.sql]@@" >
                        <xdfl:INPUT />
                </xdfl:SQL >
                </DATA>
        </xdfl:OUTPUT >

</XDFL>
```

With input data being :

```
<DATA>
<ROW>
        <NAME><![CDATA[National Bizness co.]]></NAME>
        <ADDRESS><![CDATA[3 Nowhere Alley]]></ADDRESS>
        <ZIP><![CDATA[94034]]></ZIP>
        <CITY><![CDATA[St-Cloud]]></CITY></ROW>
<ROW>
        <NAME><![CDATA[World Computers inc.]]></NAME>
        <ADDRESS><![CDATA[2 Void Plaza]]></ADDRESS>
        <ZIP><![CDATA[94334]]></ZIP>
        <CITY><![CDATA[Paris]]></CITY></ROW>
<ROW>
        <NAME><![CDATA[General Stores ltd.]]></NAME>
        <ADDRESS><![CDATA[25 Middle Street]]></ADDRESS>
        <ZIP><![CDATA[34400]]></ZIP>
        <CITY><![CDATA[St-Cloud]]></CITY></ROW>
</DATA>
```

and sql request :

```
sql = INSERT INTO MCRM_CLIENT ( ID_CLIENT, NAME, ADDRESS, ZIP, CITY )
VALUES( MCRM_SEQ_CLIENT.nextVal, :NAME<char[50]>, :ADDESSE<char[255]>, :ZIP<char[50]>,
:CITY<char[50]> )
```

Insertion of 3 records in the MRCM_CLIENT table

2.9. Working with database in object mode

2.9.1. Introduction to database object

A database object is composed by a set of data stored on database tables and linked to each other by the relational model. Defining the database object is done by isolating the part of the data model that you want to manipulate as a compact entity. This way of structuring data is made to facilitate handling of complex data structures.

An object is an XML tree, and is assimilated to the root-node of the tree. Children-nodes of the tree will be referred to as sub-objects.

Structure of an object is function of the way data is stored in the database. A node of the tree usually corresponds to a table or a view whereas leafs are usually meant to represent columns of those table. Links between nodes represent database references (those references may not be explicit).

Defining the structure of the object is made using *node* XML nodes for tables and *field* XML nodes for columns. Once the structure is defined, explicit linkage to the database tables and columns is made through the use of *Dbtable* and *Dbfield* attributes. Attribute *name* sets the name of the XML tag (node or field) that will be used in the object's XML stream.
Prefixing a field's name with @ makes the column being represented as an attribute rather than a child node in the object's XML stream.
Node nodes contain *field* node and other *node* nodes ; *field* nodes being leafs, they cannot contain anything.

Example :
The following definition

```
<node name="client" DBtable="MCRM_CLIENT"  .... >

  <field name="@id" DBfield="id_client" .../>
  <field name="name" DBfield="name".../>

  <node name="client_order" DBtable="MCRM_ORDER" >

    <field name="@id_client" DBfield="fk_client".../>

  </node>
</node>
```

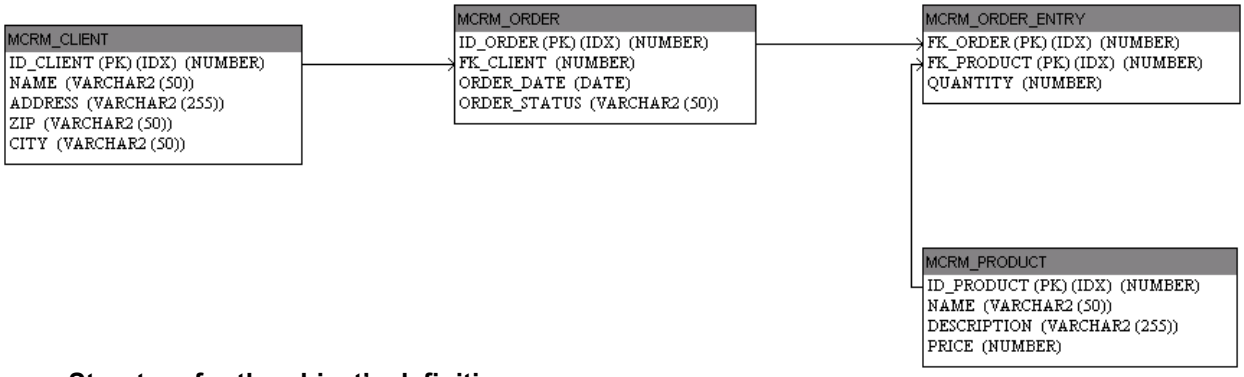
is understood as an object mapped on two tables, `MCRM_CLIENT(id_client, name)` and `MCRM_ORDER(fk_client)`. Its corresponding XML stream will be

```
<client id="<value of MCRM_CLIENT.id_client>" >

        <name><value of MCRM_CLIENT.name></name>

        <client_order id_client="<value of MCRM_ORDER.fk_client>">
        </client_order>

</client>
```

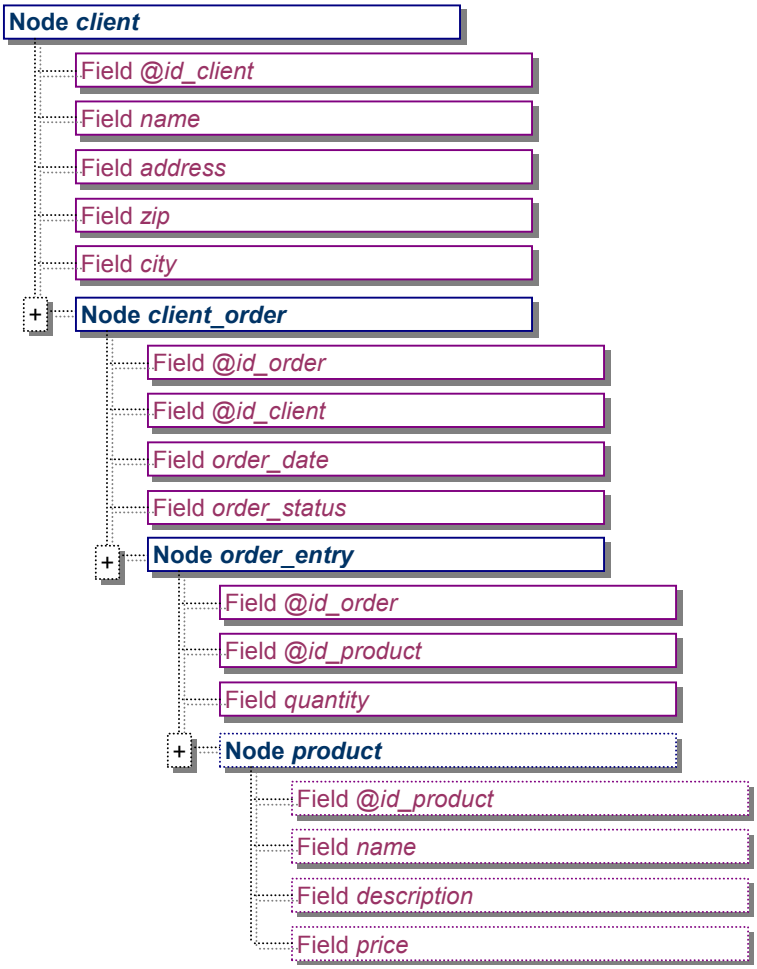
Example : defining an object for clients.

We want to define and instantiate an object representing a client, its orders, entries for those orders, and the referenced products. All those data are stored in four tables: MCRM_CLIENT, MCRM_ORDER, MCRM_ORDER_ENTRY and MCRM_PRODUCT.

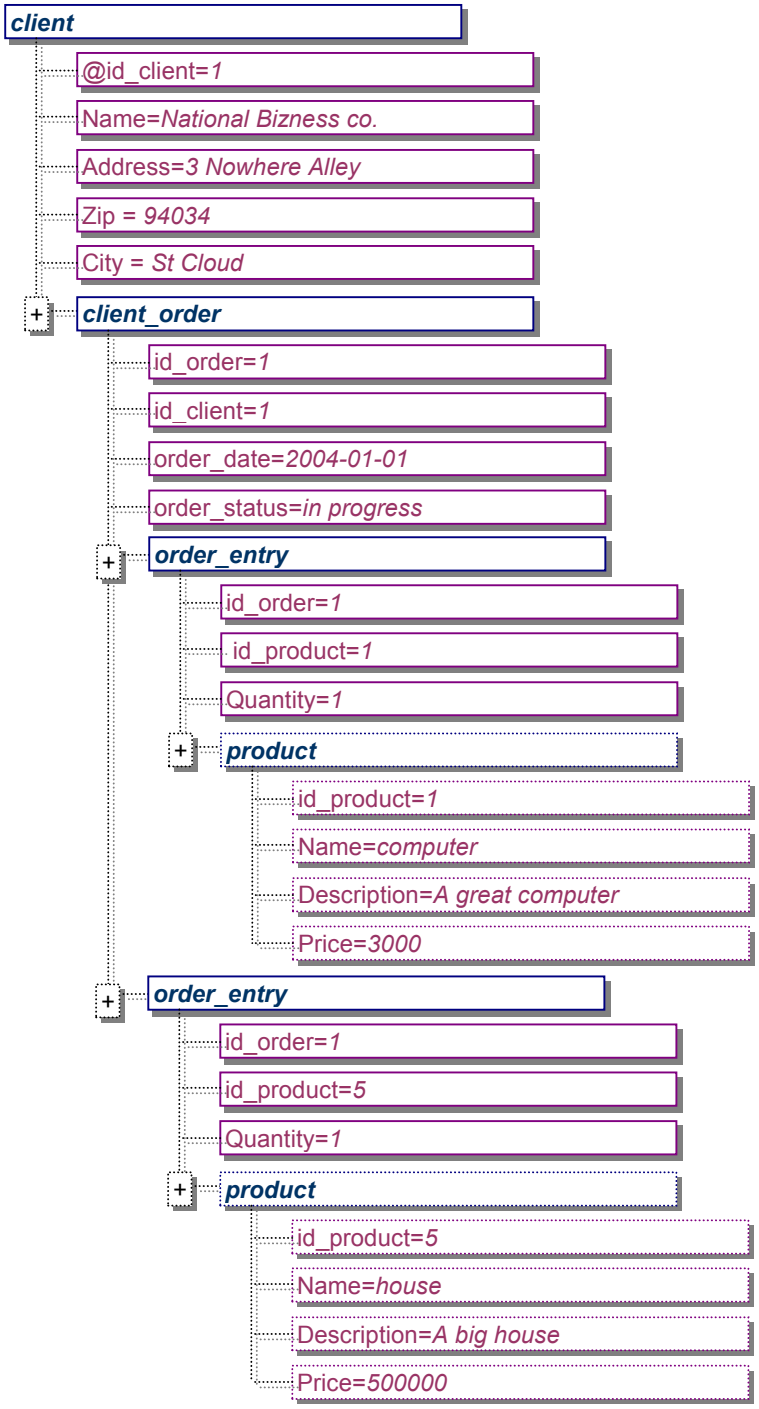
Database model:



Structure for the object's definition



Example of object generated with this definition



The client object can then be manipulated as a “bloc” for current database commands (INSERT, UPDATE, DELETE, SELECT). For example, the default behavior when deleting a client object is to delete the corresponding record in the MCRM_CLIENT table, but also in the MCRM_ORDER, MCRM_ORDER_ENTRY and MCRM_PRODUCT tables.

2.9.2. Steps for working with objects

Process of designing and using database object is pretty straightforward. Here are the steps you will probably take every time you will be working with database objects.

- ***Step 1: definition***

Structure
First step is to build an XML tree representing the structure of the object and the database mapping information (**Dbtable** and **Dbfield** parameters).

Here, *mcrm_product* node has a **lock** parameter set to 1 (true): this means that the node is in read-only mode. We don't want the product's entry to be modified when the client is modified (or even worse, deleted) so we set **lock** to 1. When locked, a node simply ignores update, delete and insert actions. As a side effect, locked nodes are not auto-completed either (see below).
Important note: **lock** parameter also locks sub-nodes of the branch. In this example, locking the "client_order" would result in locking both "client_order", "order_entry" and "product" nodes and applying all effects of the lock.

```
<node name="client" DBtable="MCRM_CLIENT" >

  <field name="@id" DBfield="id_client" DBmap="num"/>
  <field name="name" DBfield="name" DBmap="string" />
  <field name="address" DBfield="address" DBmap="string" />
  <field name="zip" DBfield="zip" DBmap="string" />
  <field name="city" DBfield="city" DBmap="string" />

  <node name="client_order" DBtable="MCRM_ORDER" >

    <field name="@id" DBfield="id_order" DBmap="num" />
    <field name="@id_client" DBfield="fk_client" DBmap="num"  />
    <field name="order_date" DBfield="order_date" DBmap="date"  />
    <field name="order_status" DBfield="order_status" DBmap="string" />

    <node name="order_entry" DBtable="MCRM_ORDER_ENTRY" >

      <field name="@id_order" DBfield="fk_order" DBmap="num" />
      <field name="@id_product" DBfield="fk_product" DBmap="num" />
      <field name="quantity" DBfield="quantity" DBmap="num" />

      <node name="product" DBtable="MCRM_PRODUCT" lock="1">

        <field name="@id" DBfield="id_product" DBmap="num" />
        <field name="name" DBfield="name" DBmap="string" />
        <field name="description" DBfield="description" DBmap="string" />
        <field name="price" DBfield="price" DBmap="num" />

      </node>

    </node>

  </node>

</node>
```

Joints

Next step is adding joints information: they tell to the engine what fields have to be used to link a node to its parent when extracting or inserting data. You can set joints by applying the following parameters on the node :

- **pkey** (*required)* Primary key associated to the node.
- **inkey** : Internal key for linkage to the parent node. Multiple names can be given if the link to the parent node involves more than one column.
- **outkey** : External key for linkage to the parent node. The number of columns in the *outkey* is equal to the number of columns of the inkey. Columns for this key are the parent's columns involved in the link.

If *inkey='col1,col2'* and *outkey='col3,col4'* then the joint is
nœud.col1=parent.col3 AND nœud.col2=parent.col4

Note : this joint will be found in the WHERE part of the SQL statement.

```
<node name="client" DBtable="MCRM_CLIENT" pkey="@id" >

  <field name="@id" DBfield="id_client" DBmap="num" />
  <field name="name" DBfield="name" DBmap="string" />
  <field name="address" DBfield="address" DBmap="string" />
  <field name="zip" DBfield="zip" DBmap="string" />
  <field name="city" DBfield="city" DBmap="string" />

  <node name="client_order" DBtable="MCRM_ORDER" pkey="@id" inkey="@id_client" outkey="@id" >

    <field name="@id" DBfield="id_order" DBmap="num" />
    <field name="@id_client" DBfield="fk_client" DBmap="num" value="@@[XPATH:../@id]@@"  />
    <field name="order_date" DBfield="order_date" DBmap="date"  />
    <field name="order_status" DBfield="order_status" DBmap="string" />

    <node name="order_entry" DBtable="MCRM_ORDER_ENTRY"
      pkey="@id_order,@id_product" inkey="@id_order" outkey="@id" >

      <field name="@id_order" DBfield="fk_order" DBmap="num" />
      <field name="@id_product" DBfield="fk_product" DBmap="num/>
      <field name="quantity" DBfield="quantity" DBmap="num" />

      <node name="product" DBtable="MCRM_PRODUCT"
        pkey="@id" inkey="@id" outkey="@id_product" lock="1" >

        <field name="@id" DBfield="id_product" DBmap="num" />
        <field name="name" DBfield="name" DBmap="string" />
        <field name="description" DBfield="description" DBmap="string" />
        <field name="price" DBfield="price" DBmap="num" />

      </node>

    </node>

  </node>

</node>
```

Valuation
Last step in the object's creation is to insert valuation parameters of primary keys and joint keys : typical valuation parameters won't be directly set by the user but instead generated either from the database (using sequences for example) or calculated from values elsewhere in the object (like in the case of a joint, you want to ensure that the keys linked together have the same value). In order to do that you use parsed values :

- `@@[SQL:SELECT sequence.NEXTVAL FROM DUAL]@@` uses an Oracle sequence to return a value (useful for index-type columns where values have to be different on every insertion)
- `@@[XPATH:../@id]@@` uses XPATH path, to value a field from the value of the id attribute of the parent node (in this case).

Field valuation can be activated on different actions : update (using parameter **uvalue**), insert (using parameter **ivalue**), delete (parameter **dvalue**) or for all actions (using parameter **value**).

In our example :
Since a sequence is only useful when inserting an object the ivalue parameter will be used (we suppose that an oracle sequence is associated for each table).
Joints keys have to be valuated at all time : value parameter will be used.
The locked nodes don't have to be valuated since they will never be involved when writing on the database.

```
<node name="client" DBtable="MCRM_CLIENT" pkey="@id">

  <field name="@id" DBfield="id_client" DBmap="num"
    ivalue="@@[SQL:SELECT MCRM_SEQ_CLIENT.nextVal from DUAL]@@" />
  <field name="name" DBfield="name" DBmap="string" />
  <field name="address" DBfield="address" DBmap="string" />
  <field name="zip" DBfield="zip" DBmap="string" />
  <field name="city" DBfield="city" DBmap="string" />

  <node name="client_order" DBtable="MCRM_ORDER" pkey="@id" inkey="@id_client" outkey="@id" >

    <field name="@id" DBfield="id_order" DBmap="num"
      ivalue="@@[SQL:SELECT MCRM_SEQ_ORDER.nextVal from DUAL]@@"  />
    <field name="@id_client" DBfield="fk_client" DBmap="num" value="@@[XPATH:../@id]@@"  />
    <field name="order_date" DBfield="order_date" DBmap="date"  />
    <field name="order_status" DBfield="order_status" DBmap="string" />

    <node name="order_entry" DBtable="MCRM_ORDER_ENTRY"
      pkey="@id_order,@id_product" inkey="@id_order" outkey="@id" >

      <field name="@id_order" DBfield="fk_order" DBmap="num" value="@@[XPATH:../@id]@@" />
      <field name="@id_product" DBfield="fk_product" DBmap="num"
        value="@@[XPATH:mcrm_product/@id]@@" />
      <field name="quantity" DBfield="quantity" DBmap="num" />

      <node name="product" DBtable="MCRM_PRODUCT"
        pkey="@id" inkey="@id" outkey="@id_product" lock="1">

        <field name="@id" DBfield="id_product" DBmap="num"  />
        <field name="name" DBfield="name" DBmap="string" />
        <field name="description" DBfield="description" DBmap="string" />
        <field name="price" DBfield="price" DBmap="num" />

      </node>

    </node>

  </node>

</node>
```

The definition is now complete. In the next paragraph we will explore the automatic description feature that will allow us to write shorter descriptions and let the engine complete the description by automatically adding missing fields. **Using the auto complete feature is the best and fastest method for defining objects**.

Automatic description (DB_AUTODESC tag)
Automatic description feature let the XDFLengine add missing information to the object definition.
Those missing information can be :

- *field* nodes that are not part of a key (pkey, inkey or outkey). Note that if you want to value the field, you have to write it on the object definition.
- Data types (Dbmap attribute).

- Date format (for date field)

Autodescription also adds specific information depending on the database used in a *Dbspec* attribute in *field* nodes. This specific information is used for optimization.

With automatic description, you only need to define structural information of the object.
Using this feature is done by giving the object's definition to complete as input to a `<xdfl:DB_AUTODESC ..>` node. The output of this node will be the fully completed description.
Example of autodesc usage is given at the next step.

- ***Step 2 : compilation (_DBDEF_COMPILE tag)***

Compiling the object
Using an object definition requires to compile it and associate it with a name. The **<xdfl :_DBDEF_COMPILE name='*name_of_definition*'…>** tag compiles the input definition and associates the compiled object definition to the name *name_of_definition* .

Example
Following is the partial definition of an object. It will be auto-completed and compiled as "mydbobject" in the next script.

```
<node name="client" DBtable="MCRM_CLIENT" pkey="@id" lock="0">

    <field name="@id" DBfield="id_client"
     ivalue="@@[SQL:SELECT MCRM_SEQ_CLIENT.nextVal from DUAL]@@"  />

    <node name="client_order" DBtable="MCRM_ORDER" pkey="@id" inkey="@id_client" outkey="@id">

        <field name="@id" DBfield="id_order"
         ivalue="@@[SQL:SELECT MCRM_SEQ_ORDER.nextVal from DUAL]@@"  />

        <field name="@id_client" DBfield="fk_client" value="@@[XPATH:../@id]@@"/>

        <node name="order_entry" DBtable="MCRM_ORDER_ENTRY"
         pkey="@id_order,@id_product"  inkey="@id_order" outkey="@id">

            <field name="@id_order" DBfield="fk_order" value="@@[XPATH:../@id]@@"/>

            <field name="@id_product" DBfield="fk_product"
             value="@@[XPATH:mcrm_product/@id]@@"/>

            <node name="product" DBtable="MCRM_PRODUCT"
             pkey="@id" inkey="@id" outkey="@id_product" lock="1">

                <field name="@id" DBfield="id_product" />
                <field name="description" DBfield="description" DBmap="string" />
            </node>
        </node>
    </node>
</node >
```

script :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<XDFL xmlns:xdfl="xdfl">
       <xdfl:OUTPUT >
              <xdfl:_DBDEF_COMPILE name="mydbobject"  >
                     <xdfl:DB_AUTODESC connection="@@[VAL:ARG.dsn]@@" expand_field="0">
                            <xdfl:INPUT />
                     </xdfl:DB_AUTODESC >
              </xdfl:_DBDEF_COMPILE >
       </xdfl:OUTPUT >
</XDFL>
```

```
<node name="client" DBtable="MCRM_CLIENT"
      restrict="" check_pre="(none)" check_post="(none)"
      oninsert="" onupdate="" ondelete="" lock="0" distinct="0"
      pkey="@id" inkey="" outkey="" >

       <field name="@id" DBfield="id_client" DBmap="num"
         ivalue="@@[SQL:SELECT MCRM_SEQ_CLIENT.nextVal from DUAL]@@"
         hidden="0" DBspec="2,2,22,0,0,0" />

       <field name="name" DBfield="name" DBmap="string"
         hidden="0" DBspec="1,1,50,0,0,0" />

       <field name="address" DBfield="address" DBmap="string"
         hidden="0" DBspec="1,1,255,0,0,1" />

       <field name="zip" DBfield="zip" DBmap="string"
         hidden="0" DBspec="1,1,50,0,0,1" />

       <field name="city" DBfield="city" DBmap="string"
         hidden="0" DBspec="1,1,50,0,0,1" />

       <node name="client_order" DBtable="MCRM_ORDER"
             restrict="" check_pre="(none)" check_post="(none)"
             oninsert="" onupdate="" ondelete="" lock="0" distinct="0"
             pkey="@id" inkey="@id_client" outkey="@id" >

             <field name="@id" DBfield="id_order" DBmap="num"
              ivalue="@@[SQL:SELECT MCRM_SEQ_ORDER.nextVal from DUAL]@@"
              hidden="0" DBspec="2,2,22,0,0,0" />

             <field name="@id_client" DBfield="fk_client" DBmap="num"
              value="@@[XPATH:../@id]@@"
              hidden="0" DBspec="2,2,22,0,0,0" />

             <field name="order_date" DBfield="order_date" DBmap="date"
              date_format="YYYY-MM-DD HH24:MI:SS"
              hidden="0" DBspec="8,12,7,0,0,1" />

             <field name="order_status" DBfield="order_status" DBmap="string"
              hidden="0" DBspec="1,1,50,0,0,1" />

             <node name="order_entry" DBtable="MCRM_ORDER_ENTRY"
              restrict="" check_pre="(none)" check_post="(none)"
              oninsert="" onupdate="" ondelete="" lock="0" distinct="0"
              pkey="@id_order,@id_product" inkey="@id_order" outkey="@id" >

                    <field name="@id_order" DBfield="fk_order" DBmap="num"
                     value="@@[XPATH:../@id]@@" hidden="0" DBspec="2,2,22,0,0,0" />

                    <field name="@id_product" DBfield="fk_product" DBmap="num"
                     value="@@[XPATH:mcrm_product/@id]@@" hidden="0" DBspec="2,2,22,0,0,0" />

                    <field name="quantity" DBfield="quantity" DBmap="num"
                     hidden="0" DBspec="2,2,22,0,0,0" />


                    <node name="product" DBtable="MCRM_PRODUCT" restrict=""
                     check_pre="(none)" check_post="(none)"
                     oninsert="" onupdate="" ondelete="" lock="1" distinct="0"
                     pkey="@id" inkey="@id" outkey="@id_product" >

                           <field name="@id" DBfield="id_product" DBmap="num"
                            hidden="0" DBspec="2,2,22,0,0,0" />
                    </node>
             </node>
       </node>
</node>
```

- ***Step 3: extracting objects from the database (DB_GET tag)***

Once the definition is compiled, it is ready to be used to extract XML stream from the database.
Extracting data using an object definition (in fact, we should say extracting an object) is done via the XDFL tag:

```
<xdfl:DB_GET connection='dsn' target='name_of_definition' >
      <DB_GET request>
</xdfl:DB_GET>
```

This tag extracts data defined by the *name_of_definition* compiled object definition on the *dsn* database. Additional filters can be set using an input request.
The format for a DB_GET request is :

```
<request>
      <filter field='field' op='filtering_operator'> value </filter>
      <order field='filter' op='sorting_operator'>
      <sub_node>
            <filter field='field' op='filtering_operator'> value </filter>
            <order field='field' op='sorting_operator'>
      </sub_node>
</request>
```

Every filter acts as another restriction for the resulting set of extracted data.
Attributes of a filter are :
field : the name of the field involved.
op : name of the restriction operator.
Operators are :

- ***eq*** (equals – default operator),
- ***lt*** (lesser than),
- ***lte*** (lesser or equals than),
- ***gt*** (greater than),
- ***gte*** (greater or equals than),
- ***ne*** (not equal),
- ***in*** (in),
- ***search*** (non case sensitive search ending with a * wildcard) ,
- ***find*** (case sensitive search starting and ending with a * wildcard)
- ***like*** (SQL LIKE).

Value for the filter is provided as input of the *filter* node.
Filters can also filter child nodes. In order to do that, you have to write the tag of the child node in the request and insert the filter in the tag.

Providing no filter means extracting all data corresponding to the object's definition.

order node gives sorting information on the way data are returned. Attributes of an order node are :
field : name of the field used to sort data
op : either **ASC** for ascending values or **DESC**. Default is ASC.

Example
The following script uses autodesc and compiles an object's definition (*def*) and uses this definition to extract all objects having field @*id* equal to the value of the *id* argument (the object obviously needs to have a field named @*id* in its root node).

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

	<xdfl:_DBDEF_COMPILE name="dbobject"  >

		<xdfl:DB_AUTODESC connection="@@[VAL:ARG.dsn]@@" expand_field="1">

			<xdfl:FS_GET target="@@[VAL:ARG.def]@@"/>

		</xdfl:DB_AUTODESC >

	</xdfl:_DBDEF_COMPILE >

	<xdfl:OUTPUT >

		<xdfl:DB_GET connection="@@[VAL:ARG.dsn]@@" target="dbobject"  enclose="DATA">

			<request>
				<filter field="@id">@@[VAL:ARG.id]@@</filter>
			</request>

		</xdfl:DB_GET >

	</xdfl:OUTPUT >

</XDFL>
```

```
<DATA>
<client id="1" >
	<name><![CDATA[National Bizness co.]]></name>
	<address><![CDATA[3 Nowhere Alley]]></address>
	<zip><![CDATA[94034]]></zip>
	<city><![CDATA[St Cloud]]></city>
<client_order id="1" id_client="1" >
	<order_date><![CDATA[2004-01-01 00:00:00]]></order_date>
	<order_status><![CDATA[in progress]]></order_status>
<order_entry id_order="1" id_product="1" >
	<quantity>1</quantity>
<product id="1" >
	<description><![CDATA[A great computer]]></description></product>
</order_entry>

<order_entry id_order="1" id_product="5" >
	<quantity>1</quantity>
<product id="5" >
	<description><![CDATA[A big house]]></description></product>
</order_entry>
</client_order>
</client>
</DATA>
```

(using previous model data and definition)

- ***Step 4: Submitting object in the database (DB_SET tag)***

Submitting an object can be done in various ways:

- Insertion: object does not exist and will be created by the submission.
- Update: object exists and its data are updated.
- Deletion: object exists and its data will be deleted.

Submission is done using the tag

```
<xdfl:DB_SET connection='dsn'
             target='name_of_definition'
             action='submit_mode'>
      <object … >
                …
      </object >
</Xdfl:DB_SET>
```

The tag submits the object given as input and defined by the *name_of_definition* compiled definition to the *dsn* database for submit mode *submit_mode.*
Output data for this tag are data of the submitted object. Some values may have been generated by the submission (ivalue, uvalue, dvalue, value clauses of the definition).

Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <xdfl:_DBDEF_COMPILE name="dbobject"  >

                <xdfl:DB_AUTODESC connection="@@[VAL:ARG.dsn]@@" expand_field="1">

                        <xdfl:FS_GET target="@@[VAL:ARG.def]@@"/>

                </xdfl:DB_AUTODESC >

        </xdfl:_DBDEF_COMPILE >

        <xdfl:OUTPUT >

                <xdfl:DB_SET connection="@@[VAL:ARG.dsn]@@"
                             target="dbobject"
                             action="@@[VAL:ARG.action]@@">

                        <xdfl:INPUT />

                </xdfl:DB_SET >

        </xdfl:OUTPUT >

</XDFL>
```

```
<client map_status='insert' id="107" >
<name><![CDATA[National Bizness co.]]></name>
<address><![CDATA[3 Nowhere Alley]]></address>
<zip><![CDATA[94034]]></zip>
<city><![CDATA[St Cloud]]></city>

<client_order map_status='insert' id="79" id_client="107" >
<order_date><![CDATA[2004-01-01 00:00:00]]></order_date>
<order_status><![CDATA[in progress]]></order_status>

<order_entry map_status='insert' id_order="79" id_product="" >
<quantity>1</quantity>

<product map_status='insert' id="1" >
<description><![CDATA[A great computer]]></description>
</product>
</order_entry>

<order_entry map_status='insert' id_order="79" id_product="" >
<quantity>1</quantity>

<product map_status='insert' id="5" >
<description><![CDATA[A big house]]></description>
</product>
</order_entry>
</client_order>
</client>
```