wscltest developer tutorial

Anton Lluis (Tonlluis) Fontgivell Mas
UOC - Universitat Oberta de Catalunya

tonlluis@uoc.edu
tonlluis@gmail.com

Copyright © 2009 Anton Lluis Fontgivell Mas

2009-02-04

1. Introduction

wscltest is a Comand Line tool to easy test webservices. It uses a working directory where a configuration file resides in along with xml request files. The configuration file defines the service's url and the operations to be called in sequential order. Read the user tutorial to learn how to use it.

wscltest is free software, licensed under GPL v. 2. You can find it at Source Forge (http://sourceforge.net/projects/wscltest)

This document is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported License. To view a copy of this license, you can click here (http://creativecommons.org/licenses/by-sa/3.0/) or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

2. Platforms and development tools

wscltest is implemented in C# for the Mono platform. MonoDevelop IDE (v. 1.0) is used to develop it, so the files that you'll find in the project's subversion repository are those of a MonoDevelop solution.

In windows platforms wscltest can run on .NET framework (but it's not free software, I suggest you to use Mono); these of you who want to develop in that platform can use SharpDevelop. There is a option in MonoDevelop to export a project to SharpDevelop v. 1, but the current version of SharpDevelop is 2.2 and the exported projects don't work in it. I recomend you to create a new solution and replace all its .cs files with these of wscltest; but in doing so you'll find a little problem with the *AssemblyInfo.cs* file: it will prevent you to compile the solution. To solve this problem you'll need to edit that file and replace the following line:

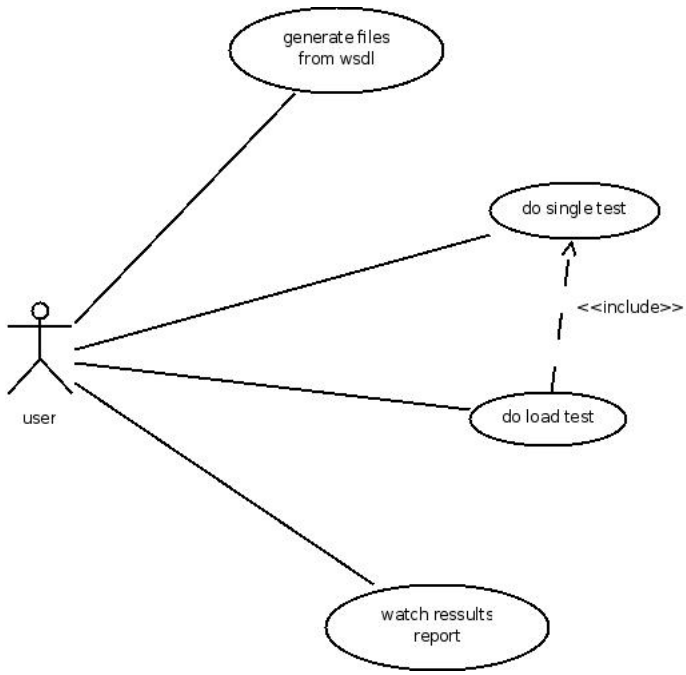
```
[assembly: AssemblyVersion("1.0.*.*")]
```

for:

```
[assembly: AssemblyVersion("1.0.*")]
```

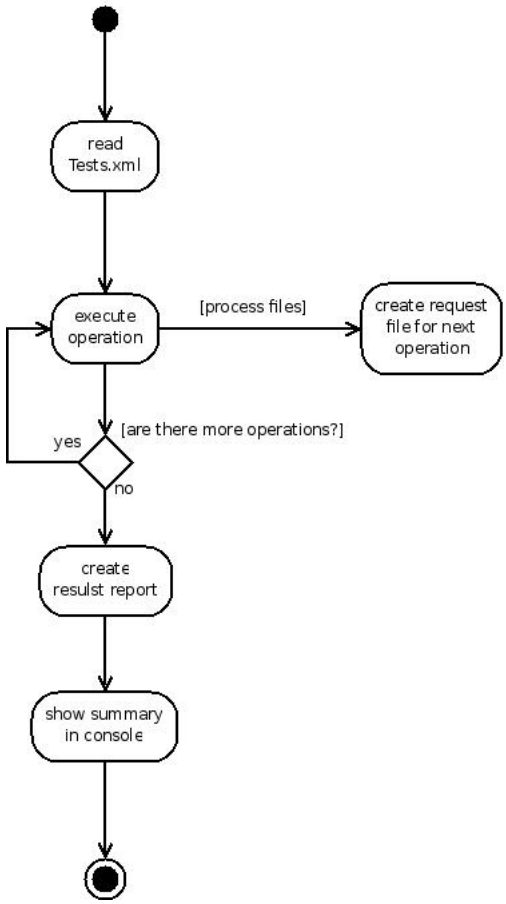
3. Use case diagram

Let's see the use case diagram to introduce the following explanations.

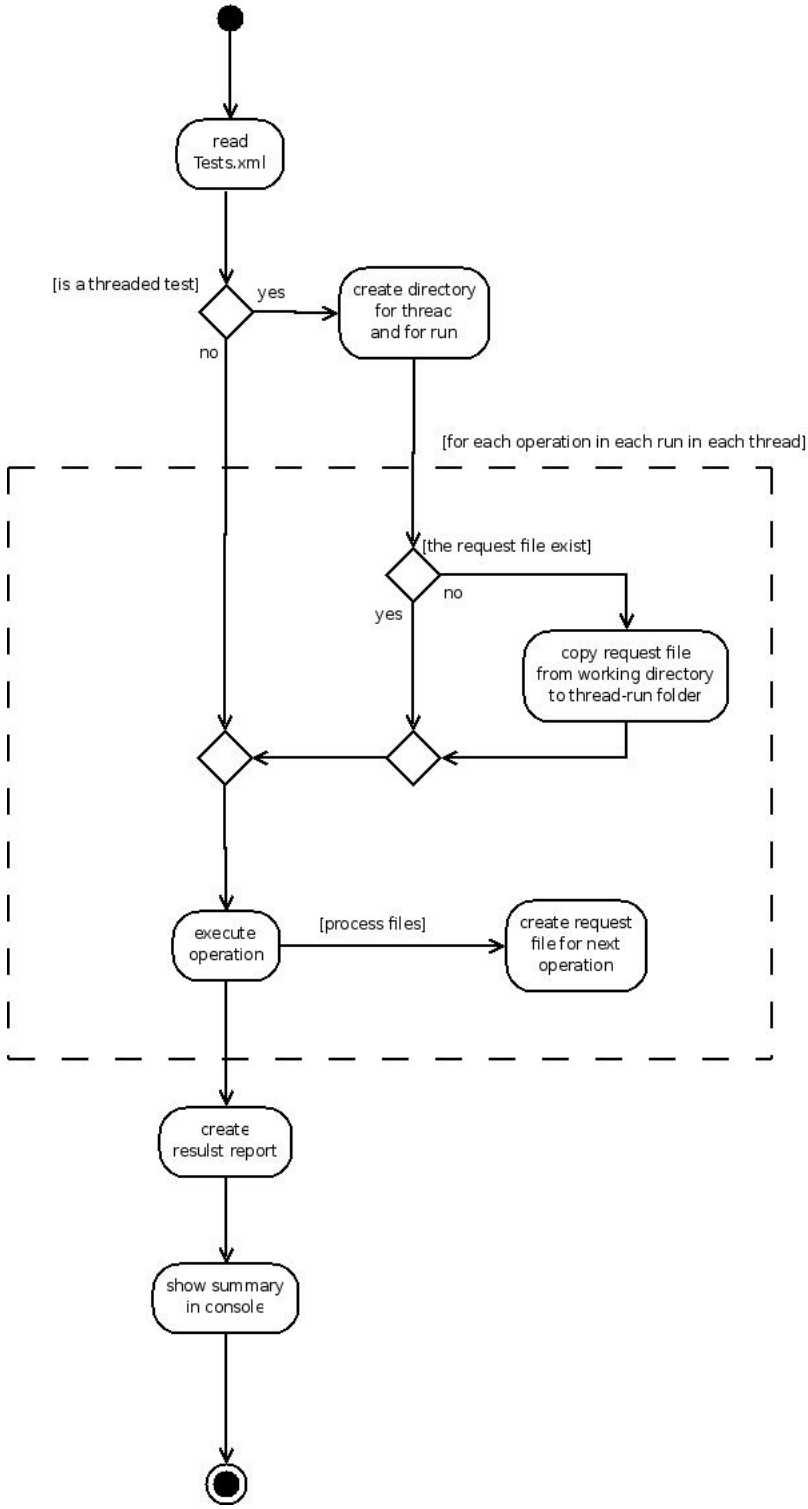


4. doing a test, single or threaded

Let's begin with the activity diagram for a single test:

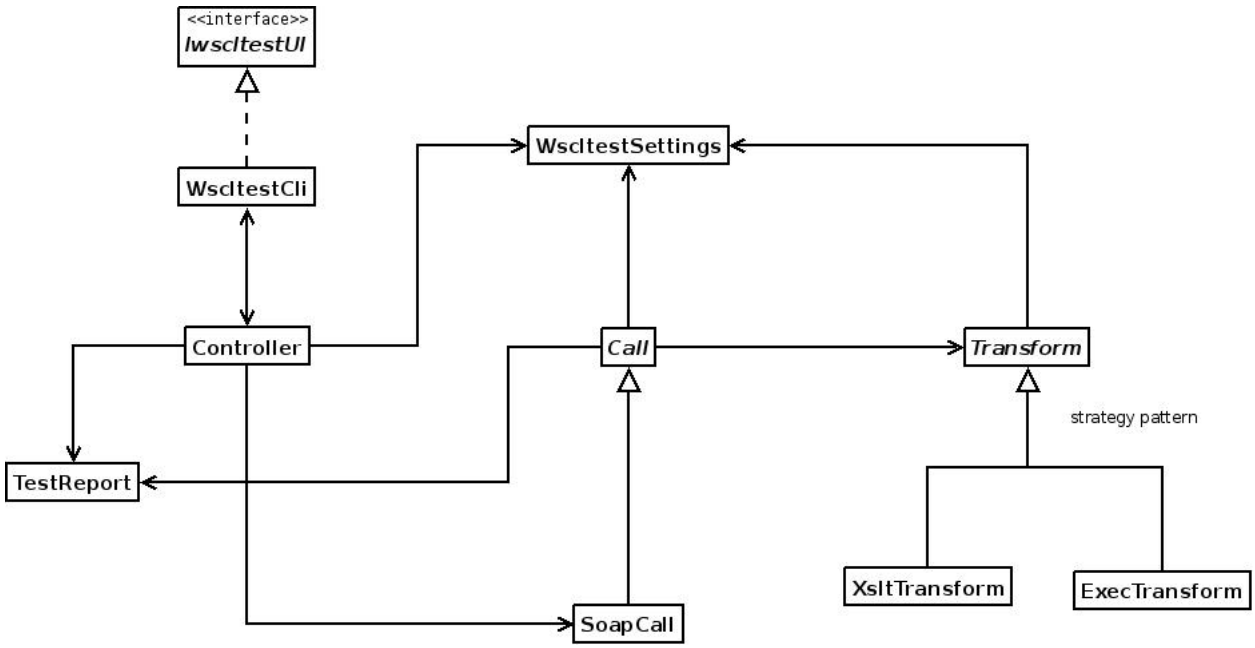


As you can see it's very simple: read the configuration file, execute every operation, create request files if specified, create results report and show summary in the console. When we consider the possibility of doing threaded tests the diagram gets more sophisticated, but not much:



The basic changes from the former diagram are that if it's a threaded test a folder for each run inside a folder for each thread are created; and if the request file for the operation is not generated (so it won't

exist in the run folder) it is copied to that folder from the working directory. Now let's see the class diagram that implements this functionality:

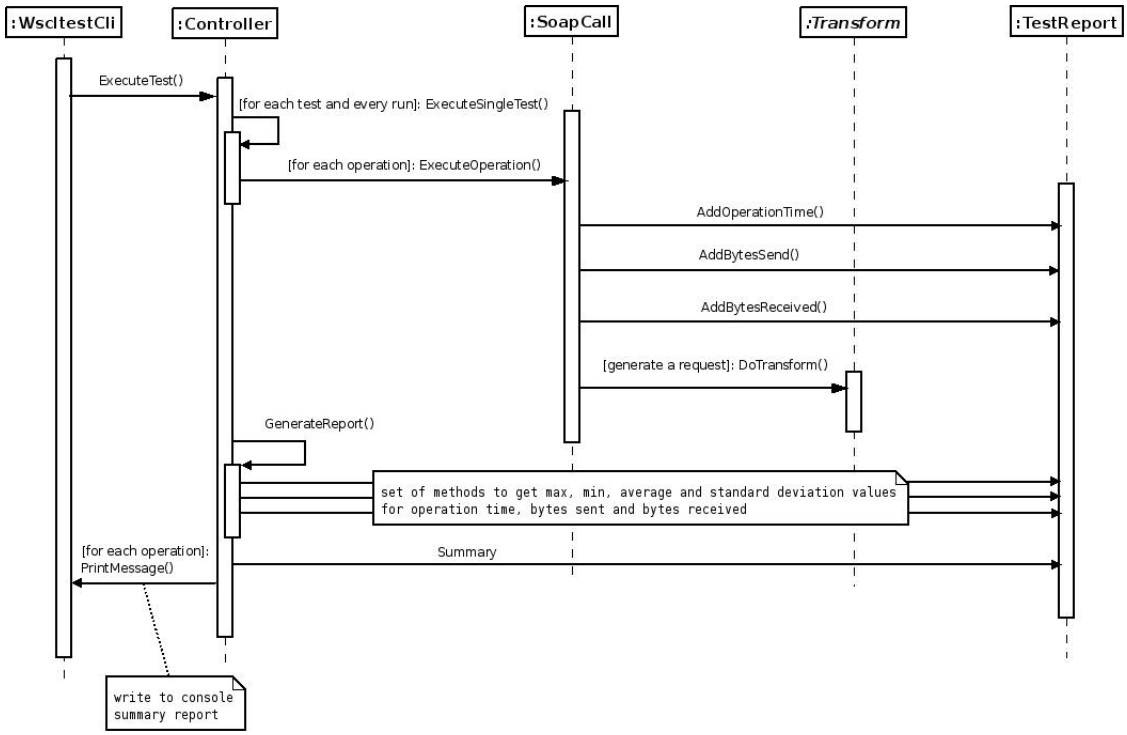


The Controller class, as its name suggest, is the controller. The Main method is located at WscltestCli; it represents the user interface (the command line in this case), it realizes the UI interface and instantiates the Controller, passing itself as an argument in the constructor; thus the Controller has visibility to the UI object too. At this moment this arquitecture maybe has little utility, but in the future a GUI may be incorporated and it seemed to me that it made sense regarding that possibility.

WscltestSettings is used to hold data that is needed in the process (like the name of the configuration file, the name of the log file, the value of the -wd parameter, etc).

At this moment wscltest accepts only one type of webservice: the SOAP standard's one. But it's planned to be added RESTful and xml-rpc ones too. For this reason there is an abstract class *Call*, from which the SoapCall extends, to apply a strategy pattern. Such pattern is also used to make "transformations" (that is, to generate new request files dynamically) if the element *ProcessFile* is present.

To facilitate its understanding here is a sequence diagram:



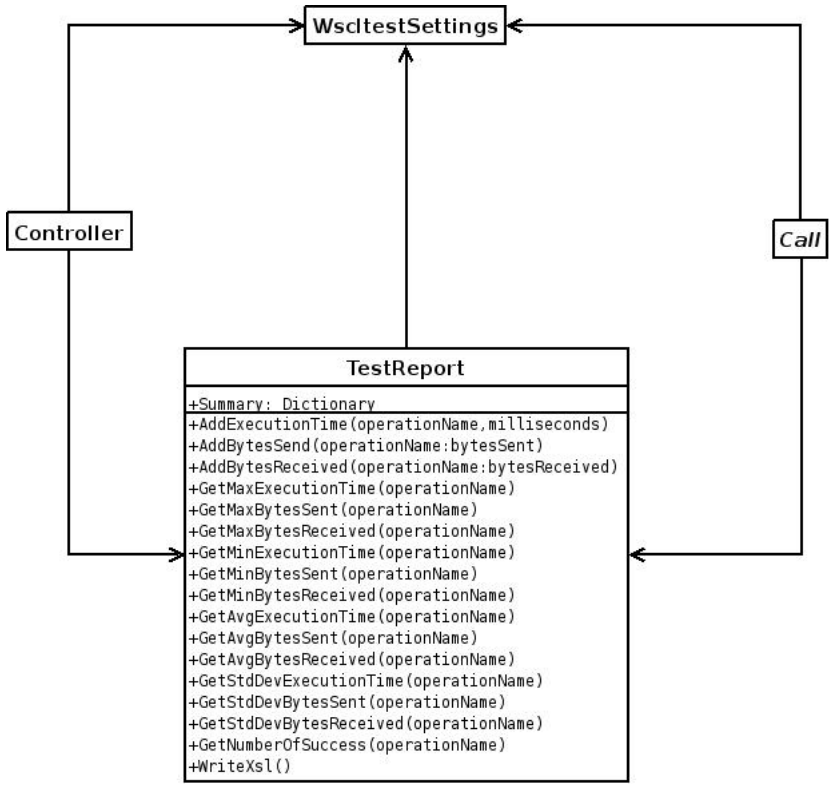
Besides there are some classes that are not present in the class diagram. They are the ones that are used to read the configuration file (Tests.xml). This file is readed by XML Serialization and the classes are generated from the the schema Tests.xsd by the xsd tool with the following command:

```
xsd Tests.xsd /c /n:wscltest.core.Tests
```

(the /c option is to generate the classes, they will be created in the namespace specified with the /n option)

5. Results report

The bulk of the work is done by TestReport class:



TestReport is a static class. It appears in the sequence diagram in the previous section, so you can see its relations with other classes in a dynamic view. Every time an operation is executed its execution time, bytes sent and bytes received are added to the respective internal dictionaries by the methods "Addxxx()". Due to the multithreading nature of wscltest a locking on the access of these dictionaries is performed. When all the operations (in all runs in all threads) are done we can obtain the max, min, average and standard deviation values for every operation calling the "Getxxx()" methods, and its number of success too (the number of executions failed or not executed is a derived value, obtained by substracting the number of success from total number of runs).

As was the case with the Test.xml, that was readed by XML serialization (see previous section), so too is the case with Results.xml. Only that in this case the XML serialization is used to generate the file, not to read it. There is in the project a Results.xsd schema file that is used to generate the classes needed for this situation; they are generated with the following command:

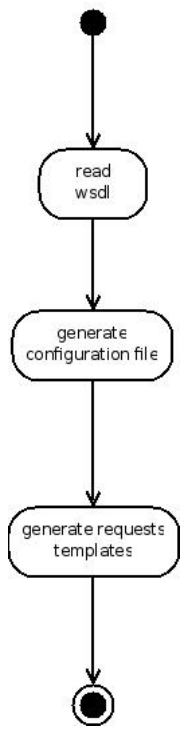
```
xsd Results.xsd /c /n:wscltest.core.Results
```

At the moment of doing the xml serialization we must add at the obtained xml a reference to Results.xsl. And after that we must generate this latter file by WriteXsl() method in TestReport, but only in the case that this file doesn't already exist in the working directory.

Apart from the Results.xml file, a summary has to be printed in the console, with the average execution time for every operation (or absolute values in the case of a single test). We obtain a Dictionary with these values from the Summary property of TestReport.

6. Generating support files

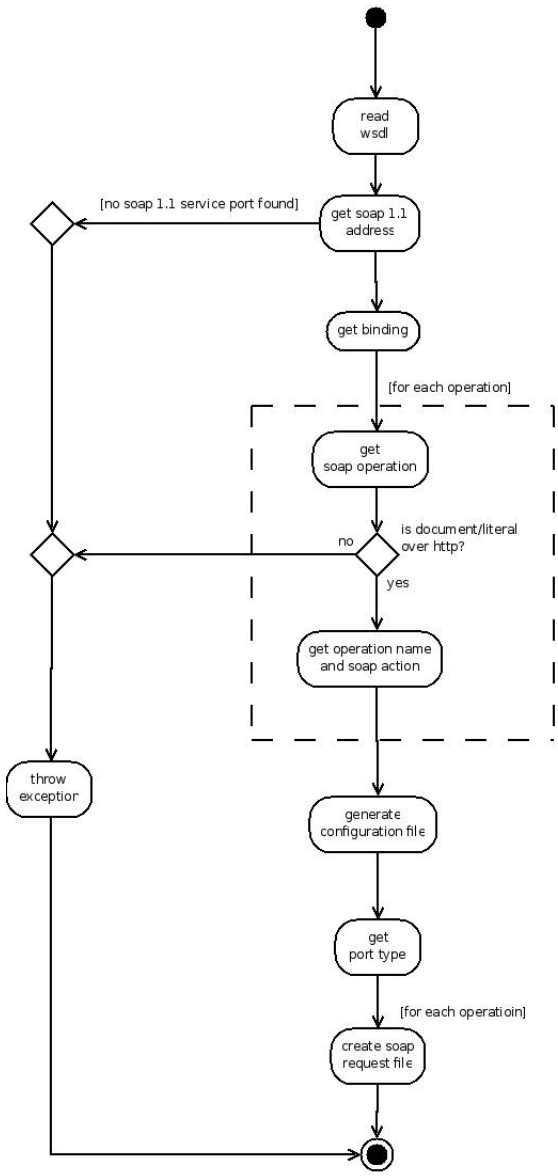
An important feature of wscltest is its option to generate the configuration file and templates for the operations of a webservice from its wsdl file. The next activity diagram summarizes this option:



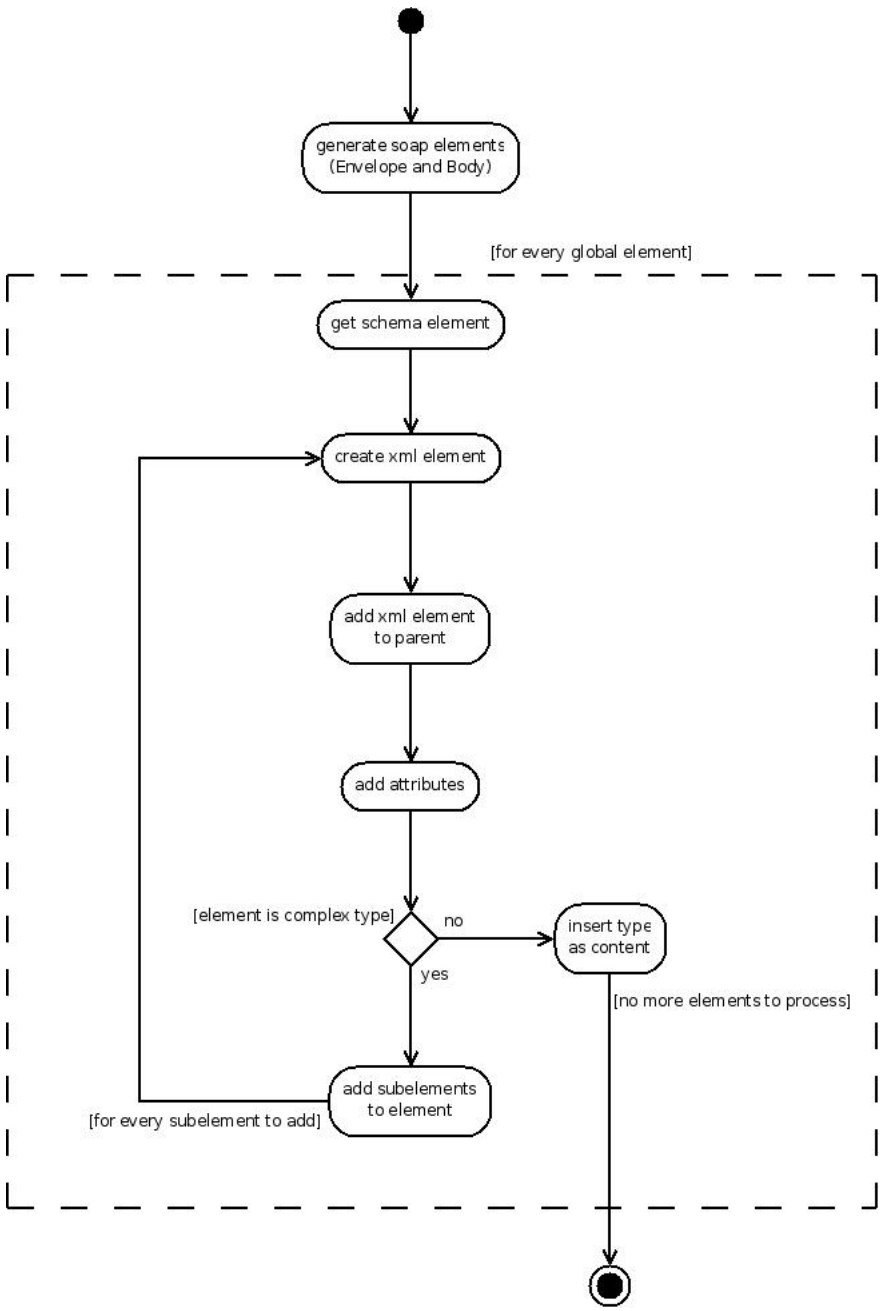
As you can see, the concept is very simple, but the details get more complicated. To understand them you need a basic understanding of the structrure of a wsdl file and a xml schema file, and SOAP protocol too. A wsdl can be split in two parts: a concrete part and an abstract one. Oversimplifying a little we can say that the concrete part defines where the webservice is located and what operations it has, and the abstract

part defines the internal structure for every operation. So we have to generate the configuration file from the concrete part and the request files from the abstract one.

Let's see the activity diagram for the concrete part:



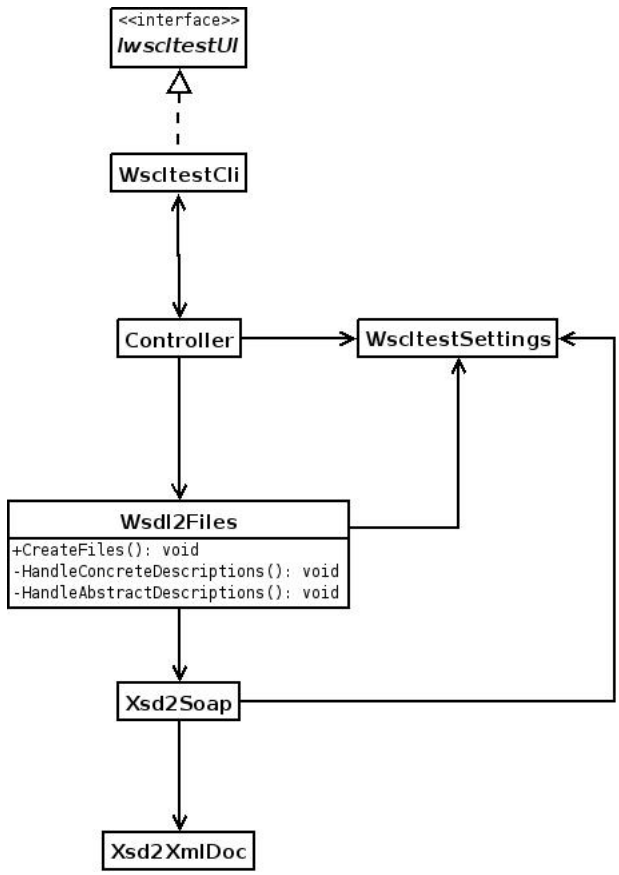
Well, as I've said, you need a basic understanding of wsdl to get it. From this diagram you can see an important thing of this option: it only deals with document/literals over http SOAP 1.1 messages. The last step ("create soap request file") is in fact the process of the abstract part, its corresponding diagram being the next one:



This diagram summarizes the process of generating a SOAP request message and, inside its body, the structure of a xml message inferred from the *Types* element of the wsdl (that corresponds to xml schema definitions for the diferent messages).

> **Important:** this part is perhaps the weakest of the entire application and it needs revission to contemplate all the details of a schema, as only the most important ones are contemplated. Besides, at this moment, this option will not generate the structure of the header part of a SOAP message, even if it's included in the wsdl.

It's the moment to see the class diagram:

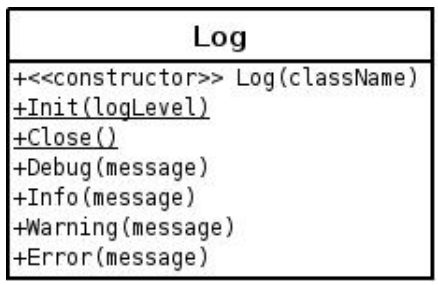


The process begins with the call to the CreateFiles() method of Wsdl2Files from Controller. Inside this method respective calls to the HandleConcreteDescription() and HandleAbstractDescriptions() methods are done. The concrete part is implemented in the Wsdl2Files class, and the abstract one in the Xsd2Soap

and Xsd2XmlDoc (note that this latter class could be used to create xml files from xsd independently of the application via the GetExampleDocument() method, without arguments).

7. Logging

A single class is in charge of logging: Log.



To use this class it is compulsory to Init and Close it before instantiating any object. There are two static methods that serve this purpose: *Init(logLevel)* and *Close()*, so they must be called at the Main() method. After that we can instantiate an object of it as a static member of each class that we need to do some logging on, passing as an argument to the constructor the name of the class.

In the Init() method the logLevel is defined (if it's null the default level, INFO, is used) and a StreamWriter to the log file is opened. At the Close() method that StreamWriter is closed. As the application is threaded we must lock the access to the log file every time a log line is to be writen in it.

SOURCEFORGE.NET