Getting SmartTools And VisualStudio.NET to talk to each other using SOAP and web services.

A summer internship report by
Joseph George Variamparambil

Under the supervision of
Dr. Didier Parigot

At INRIA, Sophia-Antipolis

July, 2001.

Acknowledgement

I would like to express my deepest appreciation and sincere gratitude to Dr. Isabelle Attali and Dr. Didier Parigot for giving me the opportunity to come to INRIA, Sophia-Antipolis and work with the SmartTools team. I would also like to extend special thanks to Pascal Degenne and Dr. Didier Parigot for explaining the ins and outs of SmartTools and to Abbondanza Thierry for working with me and providing the SOAP message filters in SmartTools.

Joseph George Variamparambil,
INRIA, Sophia-Antipolis,
July, 2001.

joseph@cse.iitk.ac.in

1 Introduction

The main objective of this project was to investigate and explore the possibility of communication between the .NET framework and SmartTools using web services technologies like SOAP, XML and HTTP.

The first section of the report introduces and describes the various tools and technologies that were used in the project. The second section details how the project work was carried out phase by phase. The report then ends in a conclusion and the references and resources used during the project work.

1.1 SOAP and Web Services

SOAP is the Simple Object Access Protocol. The current version is 1.1, and the actual specification can be found at `www.w3.org/tr/soap`. SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses. SOAP is based on XML and describes a messaging format for machine-to-machine communication.

Here is a typical SOAP request (including the HTTP headers) for an RPC method call named `echoString`, which takes a string as a parameter:

```
POST /test/simple.asmx HTTP/1.1
Host: 131.107.72.13
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://soapinterop.org/echoString"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http://soapinterop.org/"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <tns:echoString>
      <inputString>string</inputString>
    </tns:echoString>
  </soap:Body>
```

Listing 1: An example of a SOAP request.

And the corresponding response:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http://soapinterop.org/"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <tns:echoStringResponse>
      <Return>string</Return>
    </tns:echoStringResponse>
  </soap:Body>
```

Listing 2: An example of a SOAP response.

SOAP is a simple protocol that defines how to access services, objects, and servers in a platform-independent manner using HTTP and XML.

A Web Service is a unit of application logic providing data and services to other applications. Applications access Web Services via ubiquitous Web protocols and data formats such as HTTP, XML, and SOAP, with no need to worry about how each Web Service is implemented. Web Services combine the best aspects of component-based development and the Web.

1.2 SmartTools

SmartTools is a development environment generator that provides a structure editor and semantic tools as its main features. SmartTools is easy to use with its graphical user interface. It is based on Java and XML technologies and offers all the features of SmartTools to any defined language. The main goal of this tool is to provide help and support for designing software development environments for programming languages as well as domain-specific languages that are defined by XML technologies. SmartTools consists of a number of independent software components that communicate with each other through an asynchronous messaging system. The messages are transported in the XML format. The purpose of this project is to extend the messaging system to include remote messaging via SOAP requests and responses with clients, one of them being the VisualStudio.NET programming IDE.

1.3 .NET Framework and VisualStudio.NET

The .NET Framework is a new computing platform designed to simplify application development in the highly distributed environment of the Internet. The .NET Framework has two main components: the common language runtime and the .NET Framework class library.

The common language runtime is the foundation of the .NET Framework. The runtime can be thought of as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict safety and accuracy of the code. The .NET Framework class library is a comprehensive, object-oriented collection of reusable classes that you can be used to develop applications ranging from traditional command-line or graphical user interface applications to applications based on the latest innovations provided by ASP.NET and Web Services.

Visual Studio .NET is Microsoft's newest version of its Visual Studio tools . Built on the Microsoft .NET Framework, Visual Studio .NET provides a complete development environment for building XML Web services and applications. And, one of the key enhancements of VisualStudio.NET is the new C# programming language.

2 Work Done

In order to investigate the communication between VisualStudio.NET and SmartTools using SOAP, the project went through a number of phases. Starting with a simple java client and web service and finally moving on to a more complex .NET client and a SmartTools web service. Apache SOAP was chosen as the SOAP implementation to be used with SmartTools since it had the best interoperability with the .NET framework.

2.1 Phase 1

Playing around with SmartTools and getting familiar with web services technologies: SOAP.

The first phase of this project involved the investigation and understanding of how the SmartTools messaging system worked and getting familiar with SOAP. SmartTools is made of several independent software components that communicate with each other through an asynchronous messaging system. The information carried in the messages are serialized in XML format. The message controller is in charge of managing the flow of messages and delivering them to their destinations. In this project, it was experimented if these messages could be posted remotely and by different clients.

2.2 Phase 2

A simple web service and a java client.

In this phase of the work, a simple web service was implemented in java called `exampleTry`. It doesn't do much but it served as a starting point for using the Apache SOAP implementation. The code for the web service is given in the listing below:

```
public class exampleTry {

          /* A Simple Web Service */
          public String echoString(String i){
                    return "Hello "+i+"!";
          }
}
```

Listing 3: `exampleTry` – a simple java web service.

The above java program has a method called `echoString` that takes a string as an argument and returns the same string with "Hello" appended to the front of it. In order for this program to behave as a web service, it has to be deployed in the Apache SOAP Admin client. This is shown in figure 1.

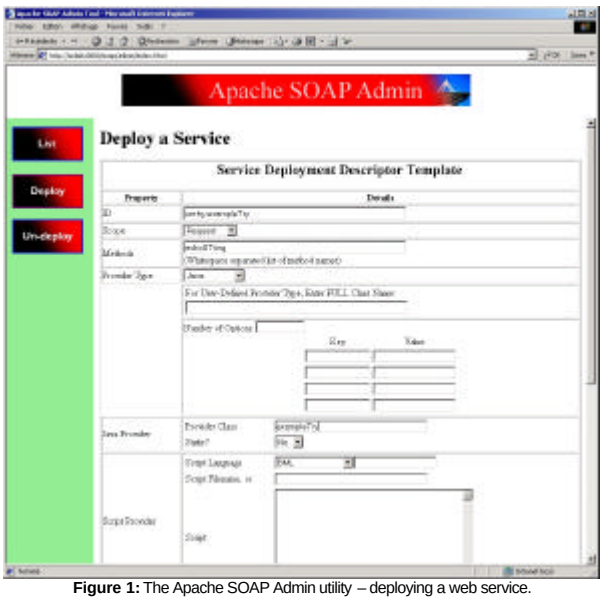

Figure 1: The Apache SOAP Admin utility – deploying a web service.

Once the `exampleTry` web service was up and running, the java client to consume the web service was implemented. Writing clients to access SOAP web services was fairly straight forward. Apache SOAP provides a client-side API to assist in the construction of the SOAP request and in assisting in interpreting the response. The code for the java client is listed below:

```
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class Client{
        public static void main(String[] args) throws Exception{
                URL url=new URL("http://localhost:8080/soap/servlet/rpcrouter");
                String urn="urn:try:exampleTry";
                Call call=new Call();
                call.setTargetObjectURI(urn);
                call.setMethodName("echoString");
                call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
                Vector param=new Vector();
                param.addElement(new Parameter("i",String.class,"World",null));
                call.setParams(param);
                try{
                        System.out.println("Client.java: invoking service...\n");
                        System.out.println("URL="+url+"\nURN="+urn);
                        Response response=call.invoke(url,"");

                        if(!response.generatedFault()){
                                Parameter result=response.getReturnValue();
                                System.out.println("Result="+result.getValue());
                        }
                        else{
                                Fault f=response.getFault();
                                System.err.println("Client.java: Fault Occured!\n");
                                System.err.println("FaultCode="+f.getFaultCode());
                                System.err.println("FaultString="+f.getFaultString());
                        }
                }
                catch(SOAPException e){
                        System.err.println("Client.java: SOAPException caught!");
                        System.err.println("FaultCode="+e.getFaultCode());
```

Listing 4: The java client that consumes the `exampleTry` web service.

When the client ran, it gives the following output shown in figure 2.

Figure 2: Output of the java client consuming the `exampleTry` web service.

2.3 Phase 3

The SmartTools web service and a java client.

After understanding how the Apache SOAP API works and how SOAP messages are transmitted between machines for communication, a more complex web service was implemented in which a java client sends a SOAP `SelectMsg` to SmartTools. A `SelectMsg` is one of the numerous messages that is used in SmartTools. It is posted when a SmartTools user selects a block of code in the SmartTools interface and the SmartTools UI responds by highlighting the portion of code that was selected. Hence, in this phase the goal was to send a `SelectMsg` remotely using SOAP and for SmartTools to highlight it. The portion of code that is to be selected is specified by treepath of a node in the syntax tree of the program being analyzed in SmartTools.

For a client to communicate with SmartTools via SOAP messages, it is required that both Apache Tomcat and SmartTools be in the same JVM instance. In order to do that, the static method `start()` of `StartTomcat.java` is called from `Smart.java`:

```
package fr.smarttools;
import org.apache.tomcat.startup.Tomcat;

public class StartTomcat{


public static void start(){
        String[] argy={"-h","/u/solida/0/oasis/jvariamp/lib/tomcat-3.2.2"};
        Tomcat.main(argy);
}
```

Listing 5: `StartTomcat.java` – A class to start the Tomcat servlet engine from within SmartTools.

Another change that had to be made to SmartTools for the web service to work was to make the method `getMsgControler()` of the class `StModule` static so that `SOAPWebService` could get a refernce to the bus and post messages to it. The listing below shows the code to implement the web service. And, as in Phase 2, the `SOAPWebService` has to be deployed in the Apache SOAP Admin utility before it can be used.

```
package fr.smarttools.webService;

import fr.smarttools.msg.SoapMsg;
import fr.smarttools.comm.MsgControler;
import fr.smarttools.StModule;

public class SOAPWebService implements SOAPWebServiceInterface{

        public SOAPWebService(){
        }

        public void recieveSOAPMsg(String XMLString){
                SoapMsg soapmsg=new SoapMsg(XMLString);
                System.out.println("SOAPWebService - Posting...\n"+XMLString);
                MsgControler bus=StModule.getMsgControler();
                bus.post(soapmsg);
        }
```

Listing 6: The SmartTools web service.

Listing 7 shows the code for the client that consumes the `SOAPWebService` web service.

```
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import java.io.*;

public class TestSmarttoolsSOAP{
        public static void main(String[] args) throws Exception{
                URL url=new URL("http://localhost:8080/soap/servlet/rpcrouter");
                String urn="urn:SmartTools:SOAPWebService";
                StringBuffer XMLString=new StringBuffer();

                try{
                       FileInputStream fs=new FileInputStream("SelectMsg.xml");
                       BufferedInputStream in=new BufferedInputStream(fs);
                       while(in.available()!=0){
                            XMLString.append((char)in.read());
                       }
                        in.close();
                }catch(Exception e){}

               Call call=new Call();
                call.setTargetObjectURI(urn);
                call.setMethodName("recieveSOAPMsg");
                call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
                Vector param=new Vector();
                param.addElement(new Parameter("XMLString",String.class,XMLString.toString(),null));
                call.setParams(param);
                try{
                        System.out.println("TestSmarttoolsSOAP.java: invoking service...\n");
                        System.out.println("URL="+url+"\nURN="+urn);
                        Response response=call.invoke(url,"");

                        if(!response.generatedFault()){
                                System.out.println("TestSmarttoolsSOAP.java: SOAP msg sent successfully!");
                        }
                        else{
                                Fault f=response.getFault();
                                System.err.println("TestSmarttoolsSOAP.java: Fault Occured!\n");
                                System.err.println("FaultCode="+f.getFaultCode());
                                System.err.println("FaultString="+f.getFaultString());
                        }
                }
                catch(SOAPException e){
```

Listing 7: The java client that consumes the SmartTools web service.

The method `recieveSOAPMsg` of `SOAPWebService` is called remotely by the java client above. `recieveSOAPMsg` takes as an argument a string which is in XML format, an example of which is shown in listing 8. It then proceeds to post the `SelectMsg` to the SmartTools bus after the string has been transformed into the appropriate class using the message filters of SmartTools. In the SOAP request that is sent by the client, several things are specified such as the type of message (in this case, a `SelectMsg`), the destination and source identities for the message and the the treepath.

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
<SOAP-ENV:Header>
 <classe xsi:type="xsd:string">fr.smarttools.msg.SelectMsg</classe>
 <destId xsi:type="xsd:int">7</destId>
 <srcId xsi:type="xsd:int">8</srcId>
 </SOAP-ENV:Header>
<SOAP-ENV:Body>
<treePath xsi:type="xsd:string">1.1.1</treePath>
</SOAP-ENV:Body>
```

Listing 8: The SOAP `SelectMsg`.

When the client is executed, it consumes the `SOAPWebService` web service and passes the above XML as an argument to `recieveSOAPMsg` and the SmartTools UI responds by highlighting the code as specified by the treepath in the SOAP request as shown in figure 3.

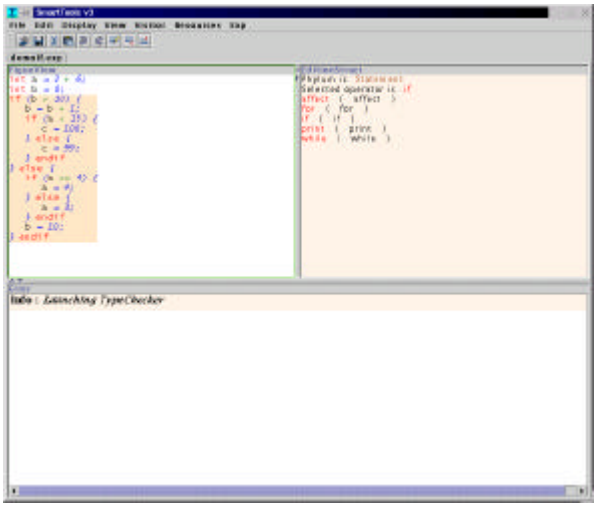

Figure 3: The SmartTools UI after the `SelectMsg` has been posted.

2.4 Phase 4

Consuming the SmartTools web service with a .NET C# client

In this phase of the project, the same SmartTools web service is consumed by a .NET client (a web page client implemented in C#). The client sends a SOAP `SelectMsg` and a portion of the code is highlighted specified by the treepath in the SOAP request. The steps that were followed to achieve this are given below:

?? **Step 1:** To generate the SDL(Service Description Language) document of the SmartTools webservice.

.NET (beta 1) requires the SDL document of a web service in order for clients to communicate with that web service. And the simplest way of doing this was to create a web service in .NET that was identical to the SmartTools web service except that the `reciveSOAPMsg` method was left empty. This was the easiest way of getting the SDL description of the SmartTools web service. Listing 9 shows the code for the web service in .NET which is identical to the java SmartTools web service.

Listing 9: .NET web service identical to the SmartTools web service.

```
namespace SOAPWebService
{
    using System;
    using System.Collections;
    using System.Configuration;
    using System.ComponentModel;
    using System.Data;
    using System.Diagnostics;
    using System.Web;
    using System.Web.Services;

    public class WebService1 : System.Web.Services.WebService
    {
        public WebService1()
        {
                InitializeComponent();
        }

        private void InitializeComponent()
        {
        }

        public override void Dispose()
        {
        }

        // This webservice is identical to the SmartTools web service except that
        // the receiveSOAPMsg method is empty
        [WebMethod]
        public void recieveSOAPMsg(string XMLString)
            {
            }
    }
}
```

And the generated SDL after modifying the namespaces to fit the SmartTools web service is shown in Listing 10 below:

```
<?xml version="1.0"?>
<serviceDescription xmlns:s0="urn:SmartTools:SOAPWebService" name="SOAPWebService"
targetNamespace="urn:SmartTools:SOAPWebService" xmlns="urn:schemas-xmlsoap-org:sdl.2000-01-25">
  <soap xmlns="urn:schemas-xmlsoap-org:soap-sdl-2000-01-25">
    <service>
      <addresses>
        <address uri="http://solida:8070/soap/servlet/rpcrouter"/>
      </addresses>
      <requestResponse name="recieveSOAPMsg" soapAction="">
        <request ref="s0:recieveSOAPMsg"/>
        <response ref="s0:recieveSOAPMsgResult"/>
      </requestResponse>
    </service>
  </soap>
  <httppost xmlns="urn:schemas-xmlsoap-org:post-sdl-2000-01-25">
    <service>
      <requestResponse name="recieveSOAPMsg" href="http://solida:8070/soap/servlet/rpcrouter">
        <request>
          <form>
            <input name="XMLString"/>
          </form>
        </request>
        <response/>
      </requestResponse>
    </service>
  </httppost>
  <httpget xmlns="urn:schemas-xmlsoap-org:get-sdl-2000-01-25">
    <service>
      <requestResponse name="recieveSOAPMsg" href="http://solida:8070/soap/servlet/rpcrouter">
        <request>
          <param name="XMLString"/>
        </request>
        <response/>
      </requestResponse>
    </service>
  </httpget>
  <schema targetNamespace="urn:SmartTools:SOAPWebService" attributeFormDefault="qualified"
elementFormDefault="qualified" xmlns="http://www.w3.org/1999/XMLSchema">
    <element name="recieveSOAPMsg">
      <complexType>
        <all>
          <element name="XMLString" xmlns:q1="http://www.w3.org/1999/XMLSchema" type="q1:string" nullable="true"/>
        </all>
      </complexType>
    </element>
    <element name="recieveSOAPMsgResult">
      <complexType/>
    </element>
  </schema>
</serviceDescription>
```

Listing 10: SDL document describing the SmartTools web service.

?? **Step 2:** Creating a proxy web service object.

Figure 4 below shows how a .NET client may communicate with a web service using a proxy object.

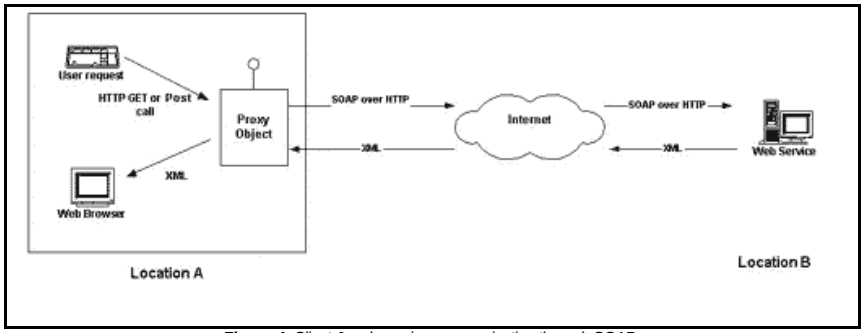

Figure 4: Client & web service communication through SOAP.
(from Building Client Interfaces for .NET Web Services By Chris Peiris at www.15seconds.com)

The functionality of the web service at Location B is replicated at Location A. A proxy object is created to act on behalf of the original web service. This is done in the .NET framework by using the utility called "`WebServiceUtil.exe`".

Figure 5 below show how the SmartTools proxy object was created in MS-DOS using the generated SDL in step 1.

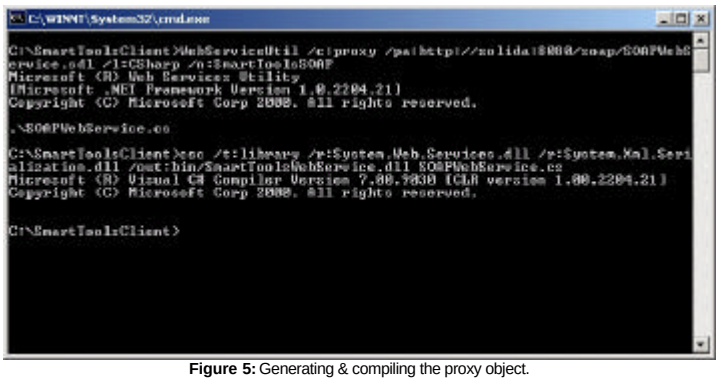

Figure 5: Generating & compiling the proxy object.

The first command creates the `SOAPWebService.cs` file in the current directory. And the next command is to compile the C# class to generate a DLL to link to the client.

?? **Step 3:** Building the web application.

Building the web application and adding the reference to the DLL that was generated is pretty straight forward in VisualStudio.NET and is shown in figure 6.

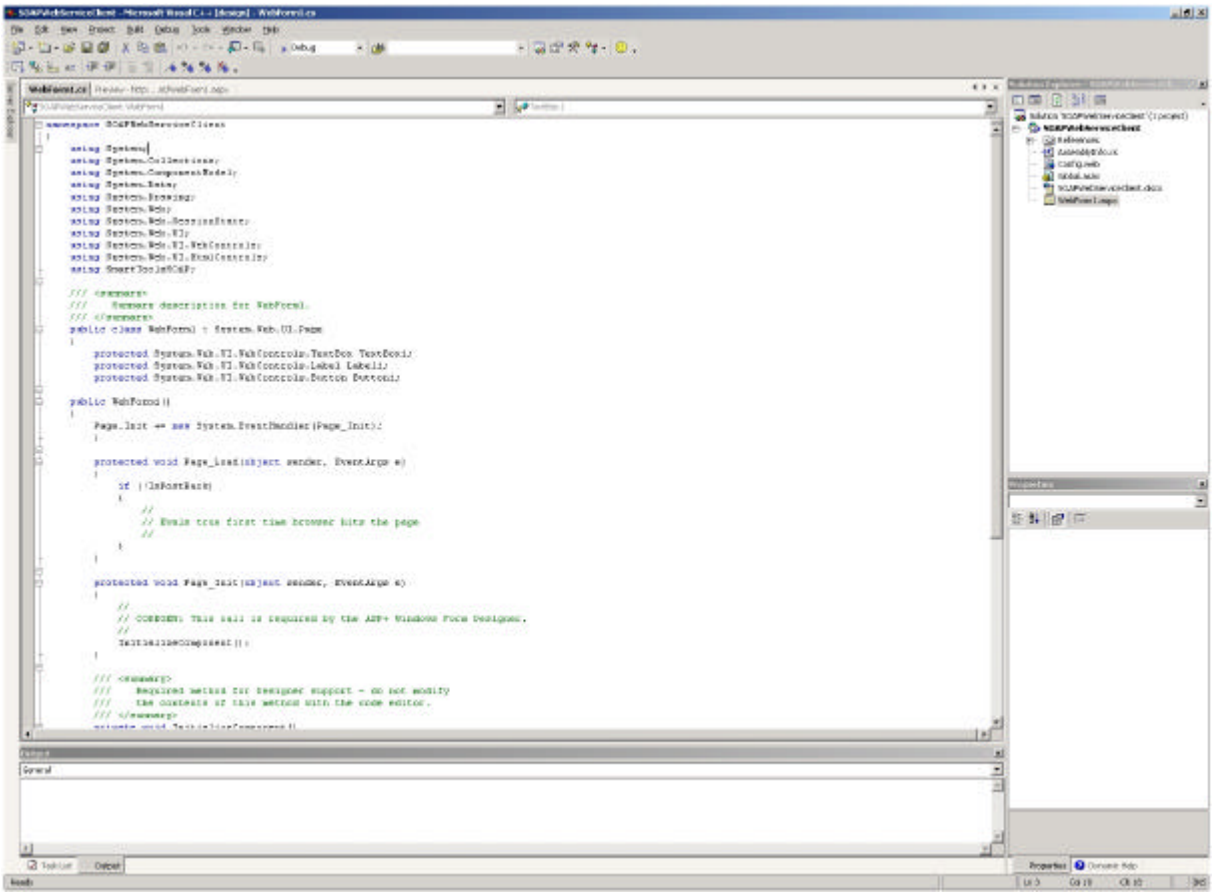

Figure 6: The web application in Visual Studio.NET Beta 1

and the code for the client is given in the listing below:

```
namespace SOAPWebServiceClient
{
    using System;
    using System.Collections;
    using System.ComponentModel;
    using System.Data;
    using System.Drawing;
    using System.Web;
    using System.Web.SessionState;
    using System.Web.UI;
    using System.Web.UI.WebControls;
    using System.Web.UI.HtmlControls;
         using SmartToolsSOAP;

    public class WebForm1 : System.Web.UI.Page
    {
                protected System.Web.UI.WebControls.TextBox TextBox1;
                protected System.Web.UI.WebControls.Label Label1;
                protected System.Web.UI.WebControls.Button Button1;

         public WebForm1()
         {
             Page.Init += new System.EventHandler(Page_Init);
        }

        protected void Page_Load(object sender, EventArgs e)
        {
            if (!IsPostBack)
            {
                // Evals true first time browser hits the page
            }
        }

        protected void Page_Init(object sender, EventArgs e)
        {
            InitializeComponent();
        }

        private void InitializeComponent()
        {
          Button1.Click += new System.EventHandler (this.Button1_Click);
          TextBox1.TextChanged += new System.EventHandler (this.TextBox1_TextChanged);
          this.Load += new System.EventHandler (this.Page_Load);
        }

        public void TextBox1_TextChanged (object sender, System.EventArgs e)
        {
        }

        public void Button1_Click (object sender, System.EventArgs e)
        {
          SOAPWebService wservice = new SOAPWebService();
```

Listing 11: .NET web application to consume the SmartTools web service.

After the above code has been built, the result can be viewed in a browser shown below with the SOAP `SelectMsg` to be sent filled in the textbox:

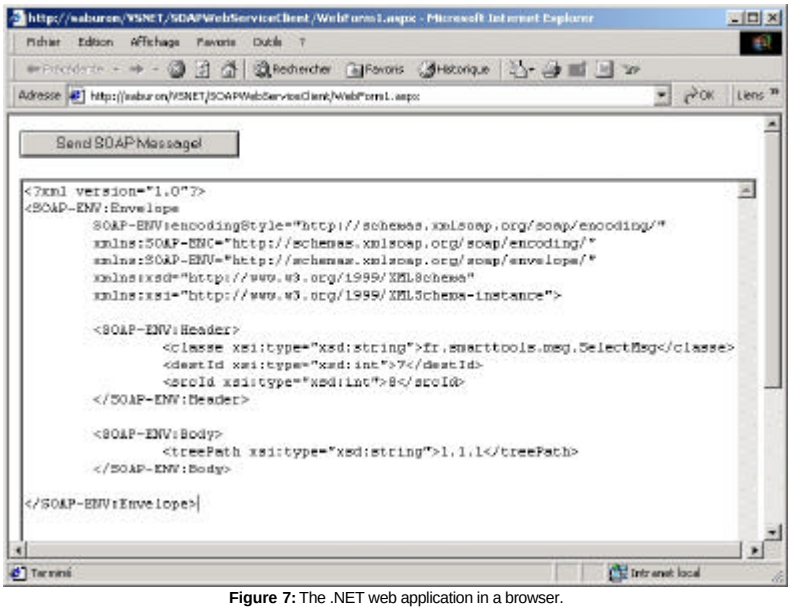

Figure 7: The .NET web application in a browser.

?? **Step 4:** Solving interoperability issues – the problem of the *xsi:type* attribute

According to the SOAP specification, the various elements contained in an envelope may optionally use the *xsi:type* attribute to identify the type of data that the element contains. If the provider and requester have some other means of communicating this information, then it needn't be included in the envelope. The *xsi:type* attribute was only be used if these data types can not be communicated by any other means. To solve this problem, Microsoft built a dependency on an external service description document, that described the data types and could be accessed by both the requester and provider. Apache required that the *xsi:type* attribute be included at all times. Both approaches are "legal SOAP", but they were incompatible with each other. Apache SOAP did not understand the SDL(Service Description Language) used by Microsoft.

In order for Apache to remove the restriction that requires the *xsi:type* attribute to be present, the SmartTools web service had to be redeployed to allow Apache SOAP to work without the *xsi:type* attribute being present. This was done manually and not through the Apache SOAP Admin utility. The deployment descriptor for the SmartTools web service is shown below:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:SmartTools:SOAPWebService">
  <isd:provider type="java"
                scope="Request"
                methods="recieveSOAPMsg">
    <isd:java class="fr.smarttools.webService.SOAPWebService" static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>

  <isd:mappings>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:x="urn:SmartTools:SOAPWebService" qname="x:XMLString"
             xml2JavaClassName="org.apache.soap.encoding.soapenc.StringDeserializer"/>
  </isd:mappings>
```

Listing 12: The Apache SOAP deployment descriptor file for the SmartTools web service.

The above deployment descriptor explicitly specifies the type of the parameter (`XMLString`) of a method call so that Apache SOAP has no problems when deserializing it and no longer has to rely on the *xsi:type* attribute. And, the web service is deployed manually as shown below:

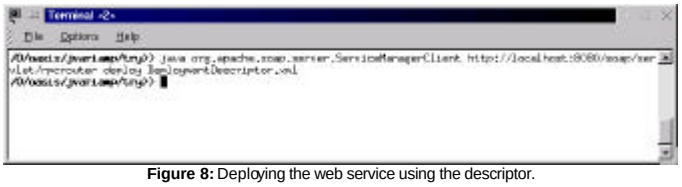

Figure 8: Deploying the web service using the descriptor.

?? **Step 5:** Running the web application

With the *xsi:type* attribute requirement removed from Apache SOAP, the web application was run and the following figure shows the output:

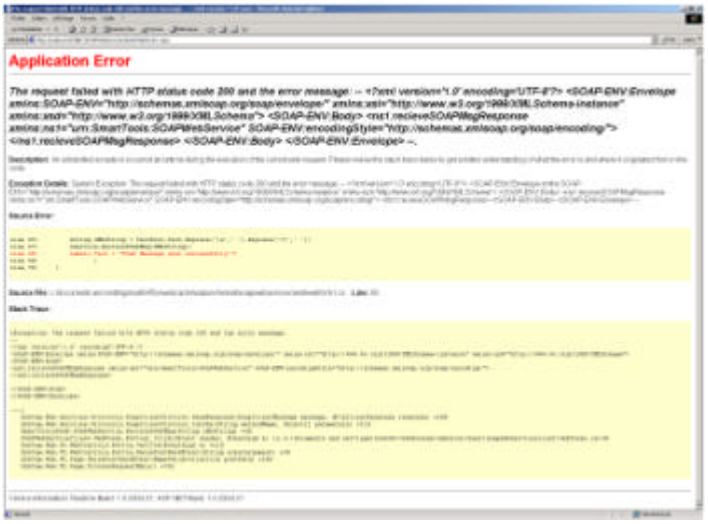

Figure 9: The output of the .NET application.

Although the application responds with an error as described in the above figure, the SOAP request did get sent successfully and code was highlighted in SmartTools. The reason for the above error is due to the non-interoperability between the Apache SOAP and .NET SOAP implementations. The error arises due to the fact that the client doesn't "understand" what Apache SOAP has sent back in the response and it expects something else. Hence, the application flagged that an error has occurred, even thought the communication took place. This is described more in detail below about what .NET expects and what it receives instead.

```
.NET Request :

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <soap:Body>
    <recieveSOAPMsg xmlns="urn:SmartTools:SOAPWebService">
      <XMLString>Hello Joseph</XMLString>
    </recieveSOAPMsg>
  </soap:Body>
</soap:Envelope>


What .NET expects :

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http://soapinterop.org/"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
          <tns:recieveSOAPMsgResponse>
                    <Return></Return>
          </tns:recieveSOAPMsgResponse>
</soap:Body>
</soap:Envelope>


What Apache SOAP sends back :

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>

<ns1:recieveSOAPMsgResponse xmlns:ns1="urn:SmartTools:SOAPWebService" SOAP-
```

Listing 13: .NET Beta 1 and Apache SOAP interop.

.Net expects that the response from the method invocation be between `<Return>` tags but instead Apache SOAP puts the response in between `<recieveSOAPMsgResponse>` tags and this is why the application reports that an error has occurred.

3 Conclusion

The main objective of this project was to investigate the possibility of communication between VisualStudio.NET and SmartTools using web services technologies like SOAP through HTTP. This objective was satisfied but due to the problems of SOAP interoperability between .NET and Apache SOAP, the .NET client complained of errors. The SOAP `SelectMsg` was sent successfully by the .NET client, received by Apache SOAP, the SmartTools web service was invoked and the SmartTools UI responding by highlighting the specified portion of code.

The interoperability issues have been solved in the Beta 2 version of VisualStudio.NET with a new and very flexible SOAP implementation which includes better support and lower-level control over the composition of the SOAP Envelope and with support for WSDL. The result has been two tools, .NET and Apache SOAP, that can communicate freely with one another.

4 References

- Web services and SOAP articles at IBM DeveloperWorks - www-106.ibm.com/developerworks/
- Building Client Interfaces for .NET Web Services by Chris Peiris - http://www.15seconds.com/Issue/010530.htm
- Resources and articles at www.xmethods.com
- .NET, XML & Web Services at msdn.microsoft.com
- The Apache SOAP user mailing list
- VisualStudio.NET help documents