1. Introducción a los Servicios web

Los servicios web son un tipo de *middleware* mediante el que pueden comunicarse aplicaciones remotas. En esencia, funciona como cualquier otro tipo de *middleware* (rmi, CORBA…), pero con la diferencia importante de que la los mensajes que se envían y se reciben se adhieren a un protocolo estandarizado llamado SOAP (*Simple Object Access Protocol*). Tanto la llamada al servicio remoto como la respuesta se codifican en SOAP y se transportan, normalmente, mediante *http*.

Todos los protocolos de mensajería pueden verse como en la siguiente figura, que muestra dos máquinas cualesquiera que se comunican: cuando *A* desea enviarle un mensaje a *B*, prepara el mensaje en un formato equivalente al que espera *B*, y lo envía. En la figura, *CA* representa el elemento de *A* que codifica el mensaje, mientras que *EB* representa el elemento de *B* por el que ésta escucha. Este modelo es válido para *rmi*, *CORBA*, servicios web o incluso para un protocolo de mensajería que nos inventemos nosotros y que podríamos implementar mediante *sockets:* lo único que tendríamos que hacer es ponernos de acuerdo en el formato de los mensajes que queremos enviar desde *A* hasta *B* y desde *B* hasta *A*, y luego implementar el mecanismo de codificación y descodificación.

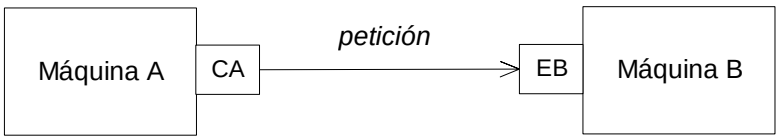

Figura 1. Dos máquinas se comunican mediante algún protocolo de mensajería

En lugar de reinventar la rueda se ha propuesto SOAP, un protocolo de mensajería basado en XML: así, la llamada a una operación ofrecida por un servidor consiste realmente en la transmisión de un mensaje SOAP, el resultado devuelto es otro mensaje SOAP, etc. De este modo, el cliente puede estar construido en Java y el servidor en .NET, pero ambos conseguirán comunicarse gracias a la estructura de los mensajes que intercambian.

Por otro lado, los servidores ofrecen, a sus posibles clientes, una lista con los servicios web que ofrecen, describiéndolos también un lenguaje estandarizado llamado WSDL (*Web Services Description Language*), que es una representación en XML de los servicios ofrecidos. Así, un cliente puede conocer los métodos ofrecidos por el servidor,

sus parámetros con sus tipos, etc., simplemente consultando el correspondiente documento WSDL.

1.01 WSDL

Supongamos que un sistema de gestión bancario utiliza, para validar las operaciones realizadas con tarjeta de crédito, el siguiente método remoto:

public boolean validar(String numeroDeTarjeta, double importe)

Si este método es accesible como un servicio web, debe estar descrito en WSDL, por ejemplo, del siguiente modo:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <!-- Generated by the Oracle9i JDeveloper Web Services WSDL Generator -->
- <definitions name="VisaWS" targetNamespace="http://visa/dominio/Visa.wsdl"
      xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://visa/dominio/Visa.wsdl"
      xmlns:ns1="http://visa.dominio/IVisaWS.xsd">
  - <types>
      <schema targetNamespace="http://visa.dominio/IVisaWS.xsd"
          xmlns="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-
          ENC="http://schemas.xmlsoap.org/soap/encoding/" />
    </types>
  - <message name="validar0Request">
      <part name="numeroDeTarjeta" type="xsd:string" />
      <part name="importe" type="xsd:double" />
    </message>
  - <message name="validar0Response">
      <part name="return" type="xsd:boolean" />
    </message>
  - <portType name="VisaPortType">
    - <operation name="validar">
        <input name="validar0Request" message="tns:validar0Request" />
        <output name="validar0Response" message="tns:validar0Response" />
      </operation>
    </portType>
  - <binding name="VisaBinding" type="tns:VisaPortType">
      <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
    - <operation name="validar">
        <soap:operation soapAction="" style="rpc" />
      - <input name="validar0Request">
          <soap:body use="encoded" namespace="VisaWS"
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
        </input>
      - <output name="validar0Response">
          <soap:body use="encoded" namespace="VisaWS"
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
        </output>
      </operation>
    </binding>
  - <service name="VisaWS">
    - <port name="VisaPort" binding="tns:VisaBinding">
        <soap:address location="" />
      </port>
    </service>
</definitions>
```

Figura 2. Descripción en WSDL del método de validación anterior

De la figura anterior merece la pena destacar algunos elementos:

- <message name="**validar0Request**"> <part name="**numeroDeTarjeta**" type="**xsd:string**" /> <part name="**importe**" type="**xsd:double**" /> </message>	Nombre del método accesible de forma remota, nombres y tipos de los parámetros. El postfijo *Request* denota el formato en que debe enviarse la solicitud al servidor. Cuando el cliente invoca el servicio, envía un mensaje *validar0Request.*
- <message name="**validar0Response**"> <part name="**return**" type="**xsd:boolean**" /> </message>	Tipo del resultado devuelto por el método. El postfijo *Response* se refiere precisamente a que es el tipo devuelto lo que se está representando.
- <portType name="**VisaPortType**"> - <operation name="**validar**"> <input name="**validar0Request**" message="**tns:validar0Request**" /> <output name="**validar0Response**" message="**tns:validar0Response**" /> </operation> </portType>	Operaciones que conforman la interfaz del servicio *validar*, que se corresponden con los dos *messages* anteriores.

Figura 3. Significado de algunos elementos del WSDL mostrado en la figura anterior

Los entornos de desarrollo recientes incluyen los *add-ins* necesarios para generar la especificación WSDL de una clase.

1.02 Escritura de un cliente que acceda a un servicio web

El cliente que utiliza el servicio web necesita una clase que actúe como *proxy* entre él mismo y el servicio web ofertado por el servidor (en la Figura 1, correspondería al elemento *CA*). Cuando el *proxy* recibe del cliente una solicitud de llamada al servicio web, el *proxy* la traduce a un mensaje SOAP, que envía al servidor; éste, entonces, lo ejecuta, y devuelve un mensaje SOAP al *proxy*; éste, entonces, traduce el mensaje a objetos Java, .NET, etc. y entrega el resultado al cliente que efectuó la petición.

La siguiente figura muestra la relación entre el *proxy*, el servicio web y la clase que lo implementa: el servicio web (en el centro) ofrece a los clientes acceso a la operación *validar(String, Double);* la operación se encuentra realmente implementada en la clase situada abajo (*Visa*), y podría incluir llamadas a otros métodos de otras clases, acceso a una base de datos, acceso a otros servicios web, etc. El elemento de la izquierda (*VisaWSStub*) es la clase que actúa de proxy entre los clientes y el servicio web. Nótese que esta clase incluye, además de otras, la operación *validar(String, Double)*. El proxy mostrado se ha obtenido de forma automática con un entorno de desarrollo, por lo que sus miembros pueden variar de unos casos a otros. El atributo *_endpoint* representa la URL en la que se encuentra publicado el servicio web.

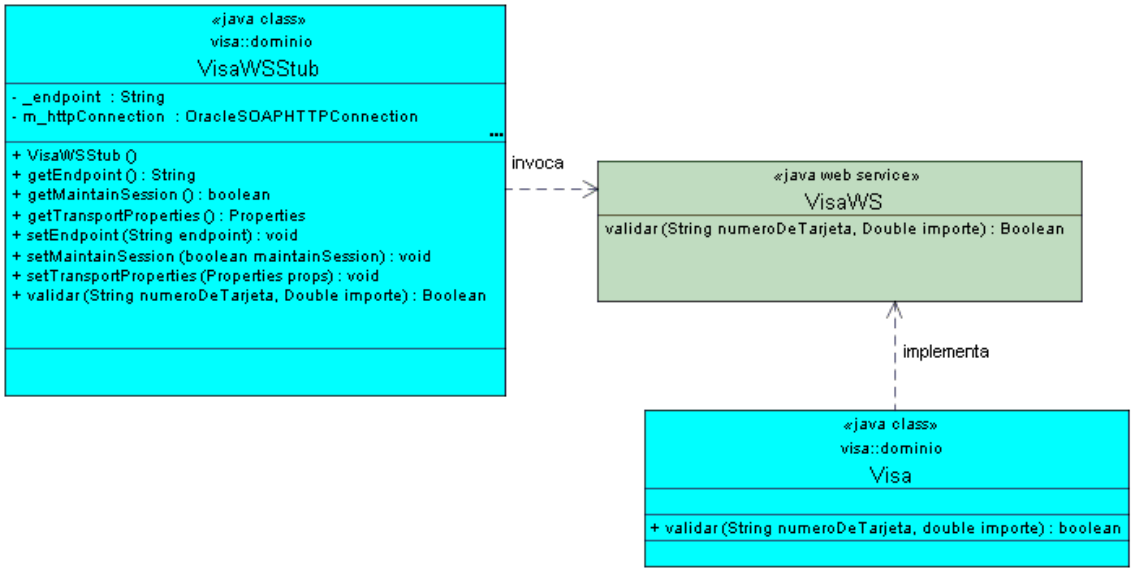

Figura 4. El proxy, el servicio web y la clase que lo implementa

La aplicación cliente hace entonces uso del proxy para acceder al servicio web, por ejemplo con un trozo de código como el que sigue:

```
protected void validarDisponibilidadDeCredito(double importe)
                                                    throws Exception
{
     // Se intancia el proxy
    VisaWSStub stub = new VisaWSStub();
     // Se le dice al proxy dónde puede encontrar el servicio web
    stub.setEndpoint("http://161.67.27.108:8988/soap/servlet/soaprouter");
     // Se llama al método ofertado por el proxy
    if (!stub.validar("", new Double(5.0)).booleanValue())
      throw new Exception("Operación no admitida");
}
```

Obsérvese que el tipo del segundo parámetro de la llamada a *validar* es de tipo *Double* (con mayúsculas) y no *double* (con minúsculas), a pesar de que el método toma un *double* en la clase en la que se encuentra implementado. Esto es así porque el generador del proxy ha optado por incluir aquel tipo de dato en lugar de éste.

Cuando el tipo de un parámetro o del resultado es un tipo complejo, es preciso realizar una descripción del tipo (o de la clase) en el documento WSDL. Supongamos que la clase *Visa* ofrece la siguiente operación:

public dominio.Tarjeta getTarjetaConMayorCredito()

La clase *dominio.Tarjeta* es un tipo complejo y debe especificarse en el documento que describe el servicio web. El entorno de desarrollo es capaz de añadir al WSDL la descripción estandarizada de esta clase:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <!--
Generated by the Oracle9i JDeveloper Web Services WSDL Generator
-->
```

```
- <!--
Date Created: Thu Apr 22 18:00:42 CEST 2004
  -->
- <definitions name="VisaWS" targetNamespace="http://visa/dominio/Visa.wsdl"
      xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://visa/dominio/Visa.wsdl"
      xmlns:ns1="http://visa.dominio/IVisaWS.xsd">
  - <types>
    - <schema targetNamespace="http://visa.dominio/IVisaWS.xsd"
          xmlns="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-
          ENC="http://schemas.xmlsoap.org/soap/encoding/">
      - <complexType name="dominio_Tarjeta" jdev:packageName="dominio"
            xmlns:jdev="http://xmlns.oracle.com/jdeveloper/webservices">
        - <all>
            <element name="numero" type="string" />
            <element name="titular" type="string" />
            <element name="validaDesde" type="string" />
            <element name="validaHasta" type="string" />
          </all>
        </complexType>
      </schema>
    </types>
    <message name="getTarjetaConMayorCredito0Request" />
  - <message name="getTarjetaConMayorCredito0Response">
      <part name="return" type="ns1:dominio_Tarjeta" />
    </message>
```

Figura 5. El WSDL incluye ahora la descripción del tipo complejo "Tarjeta"

1.03 Elementos necesarios para el desarrollo en Eclipse de SW

Para desarrollar y probar servicios web en Eclipse necesitamos, además del propio Eclipse y del JDK que corresponda, un servidor web como el Tomcat de Apache, un motor de ejecución de servicios web (como Axis2, también de Apache) y dos plugins de Axis2 para Eclipse (Tabla 1).

Elemento	URL	Descripción
Tomcat	http://tomcat.apache.org/	Servidor web
Axis2	http://ws.apache.org/axis2/download.cgi	Motor de ejecución de WS
Axis2 Service Archive Wizard	http://ws.apache.org/axis2/tools/index.html	Genera el desplegable de la aplicación
Axis2 Code Generator Wizard	http://ws.apache.org/axis2/tools/index.html	Generador del WSDL

Tabla 1. Elementos de los que conviene disponer

Una vez descargados e instalados el Tomcat y Axis2 (habiendo establecido además las variables CATALINA_HOME y AXIS2_HOME a los directorios en los que hemos instalado, respectivamente, ambas herramientas), es necesario darlos de alta en nuestro entorno Eclipse, como se muestra en las siguientes figuras:

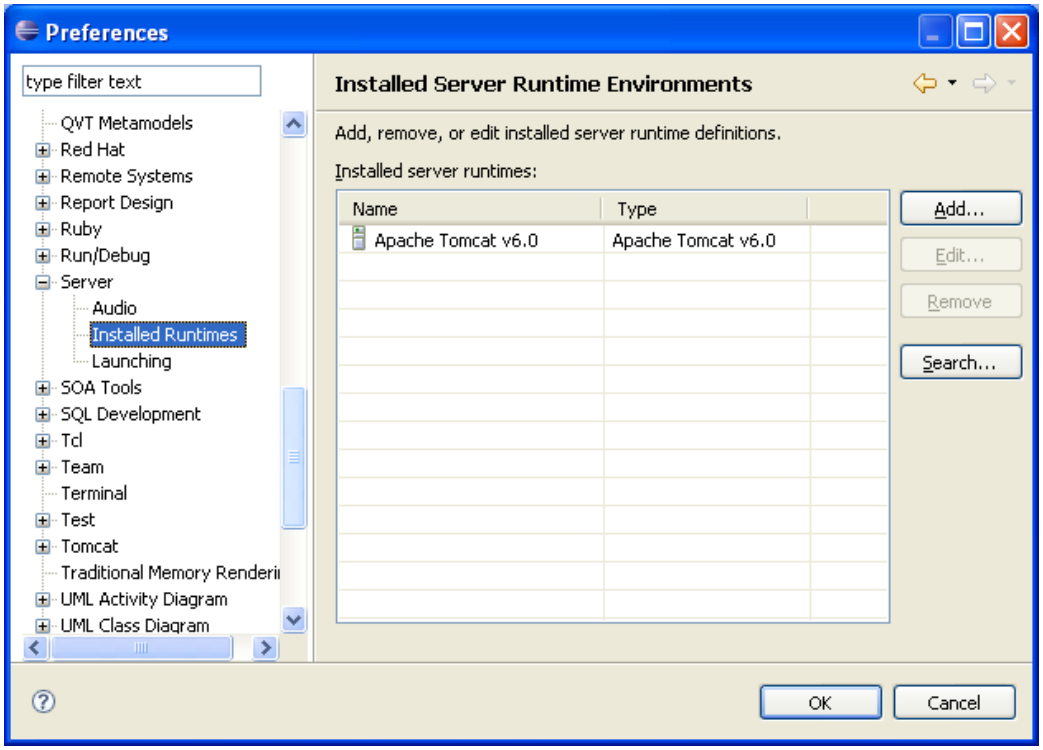

Figura 6. Alta de Tomcat en Eclipse

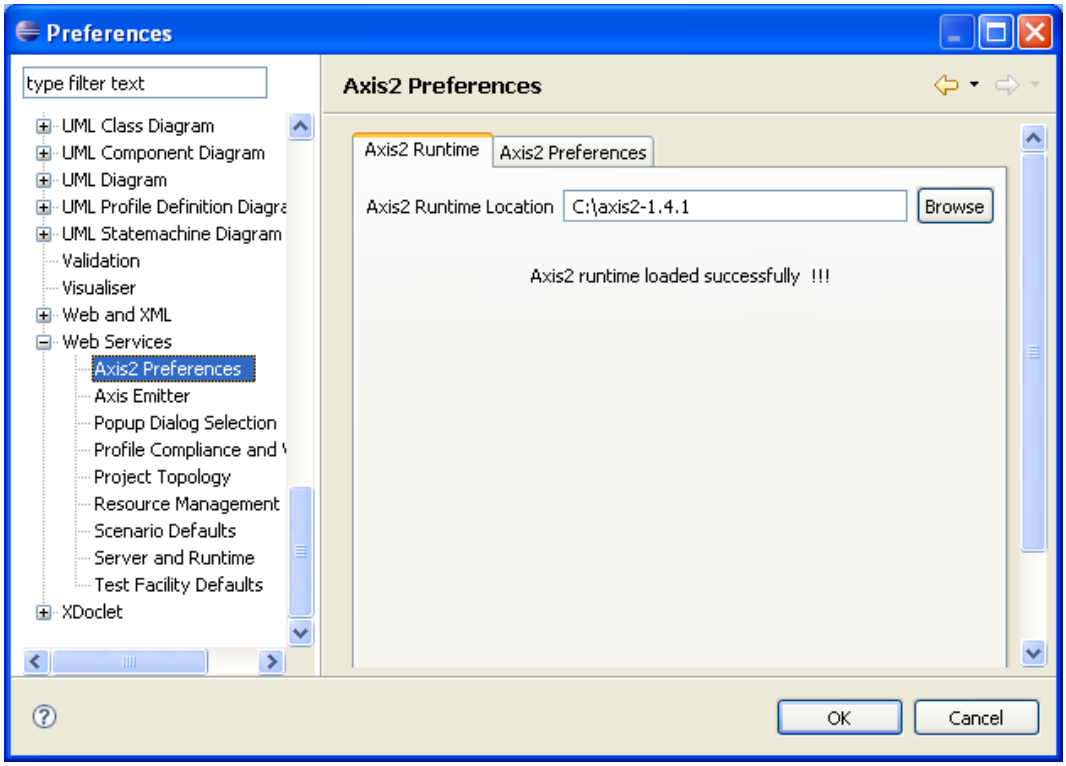

Figura 7. Alta de Axis2 en Eclipse

Por otro lado, para que Tomcat sea capaz de procesar los mensajes SOAP y reencaminarlos adecuadamente a la implementación de los servicios ofrecidos, copiamos al directorio webapps de Tomcat el fichero *axis2.war* de axis2.

Para instalar los dos plugins de Axis2 para Eclipse basta con descomprimir los dos archivos en el directorio *plugins* de Eclipse (Figura 8).

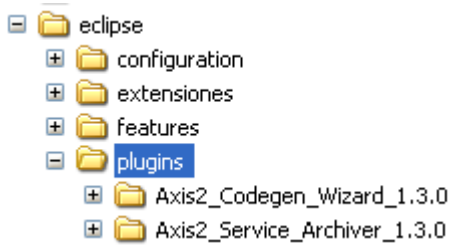

Figura 8. Instalación de los dos plugins en Eclipse

1.04 Publicación de servicios web

Un servicio web expone una serie de operaciones que se encuentran implementadas en alguna aplicación.

En nuestro caso, ofreceremos a nuestros clientes la operación de validación de una operación con tarjeta de crédito, contenida en la clase mostrada en la Figura 9, que se encuentra incluida en un proyecto Java de Eclipse:

```
package banco;

import java.util.Random;

public class Visa {
      public boolean validar(String numeroDeTarjeta, double importe) {
            Random r=new Random();
            return r.nextDouble()>0.4;
      }
}
```

Figura 9. Sencilla clase con una operación que se expondrá como un servicio web

A continuación, seleccionamos la clase *Visa* en el árbol del proyecto y elegimos *New*, *Other*, y seleccionamos *Axis2 Service Archiver*, que corresponde a uno de los plugins que hemos instalado:

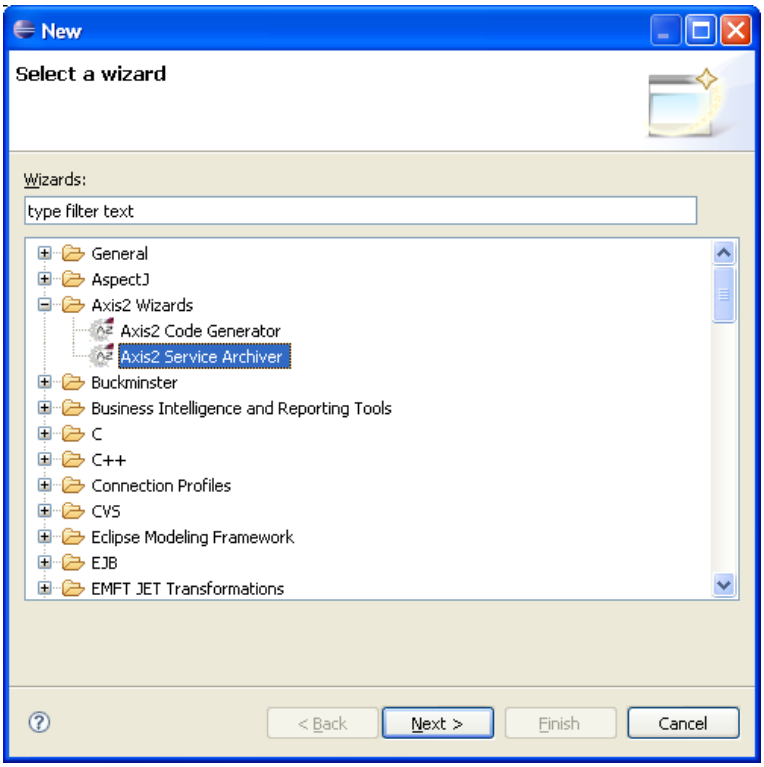

Figura 10

Tras pulsar *Next*, indicamos dónde se encuentra la clase que posee las operaciones que queremos exponer como servicios, como se muestra en la Figura 11:

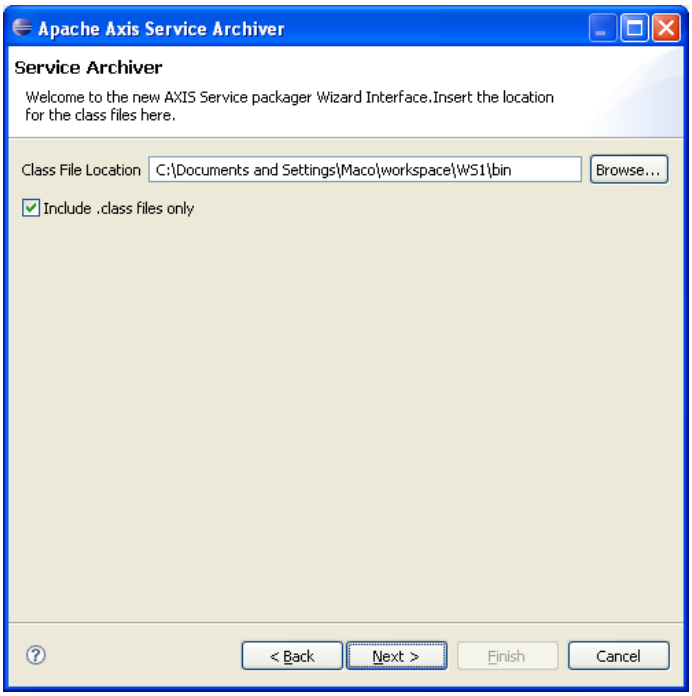

Figura 11

Como se ha comentado, el servidor web ofrecerá una descripción de las operaciones ofrecidas en un documento WSDL. En el siguiente paso del asistente,

podemos especificar un documento WSDL que hayamos generado nosotros, o bien dejar que se genere por nosotros. Para esto último, elegimos la opción “Skip WSDL”:

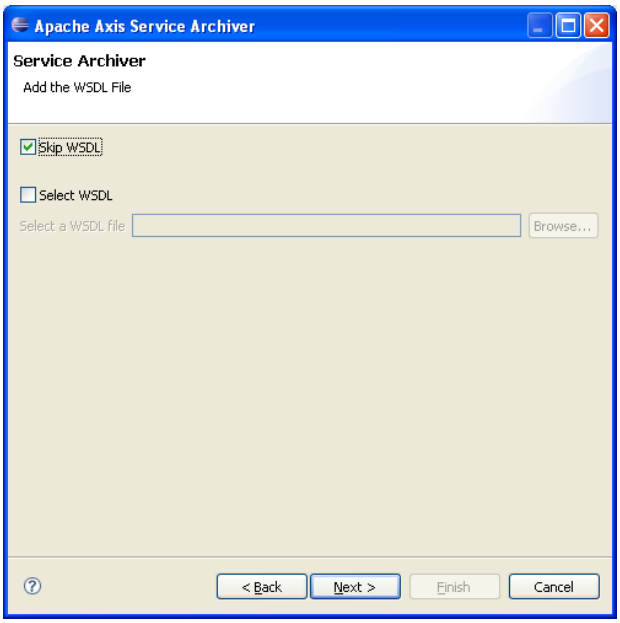

Figura 12

Es posible que la aplicación que posee las operaciones que vamos a ofrecer como servicios web requiera librerías adicionales (p.ej., si accede a una base de datos, requerirá las librerías con los drivers necesarios para acceder a dicha base de datos). Estas librerías las indicamos en el siguiente paso del asistente (Figura 13), aunque en este sencillo ejemplo no es necesario especificar ninguna.

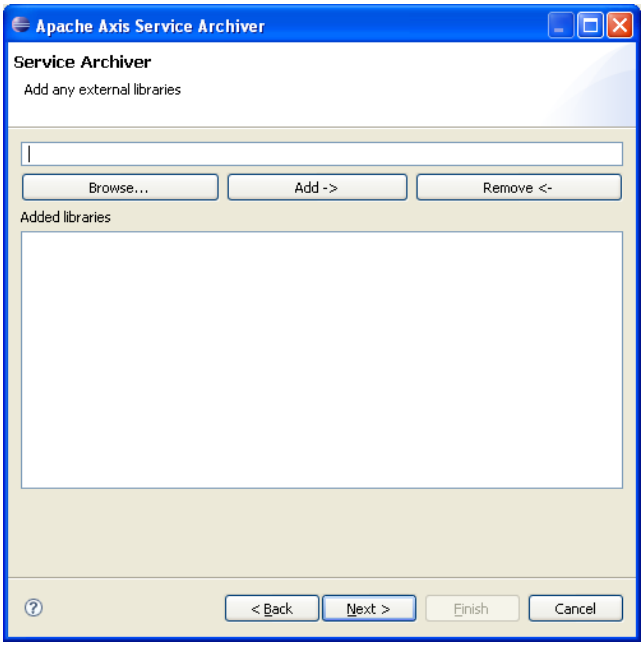

Figura 13

En el siguiente paso dejamos marcada la opción “Generate the service xml automatically”, que añadirá al archivo con la aplicación de servicios web un fichero XML con la identificación del servicio (Figura 14).

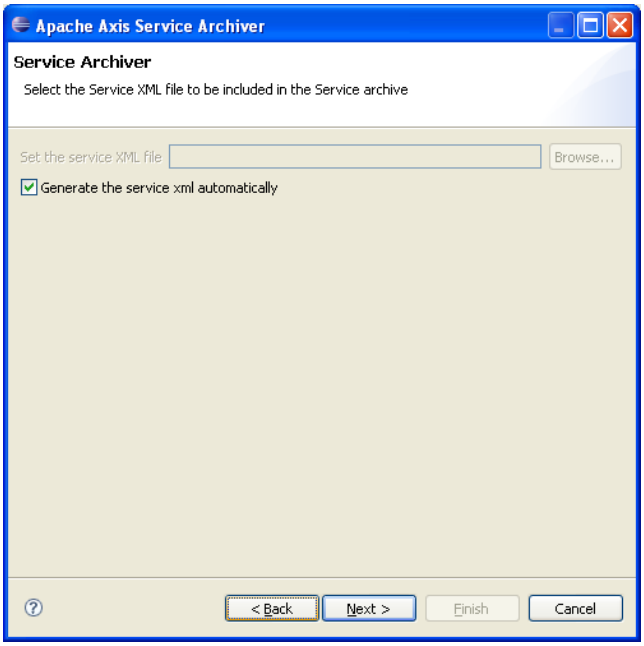

Figura 14

A continuación, especificamos el nombre con el que publicaremos el servicio web (realmente, el servicio web expone como servicios una lista de operaciones) e indicamos el nombre completo (indicando el paquete) de la clase que implementa las operaciones, y seleccionamos aquéllas que deseamos ofrecer como servicios (Figura 15).

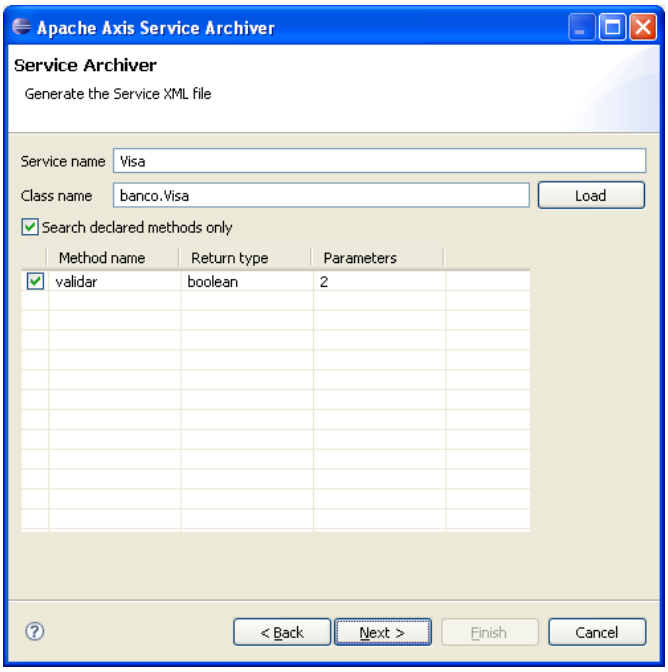

Figura 15

Por último, indicamos el lugar en el que queremos dejar el fichero con la aplicación web, que tendrá extensión *.aar* (en este ejemplo, mostrado en la Figura 16, el fichero se llamará *banco.aar*). Podemos elegir directamente la carpeta en la que Tomcat guarda las aplicaciones de servicios web (*webapps\axis2\WEB-INF\services)* del directorio de instalación de Tomcat.

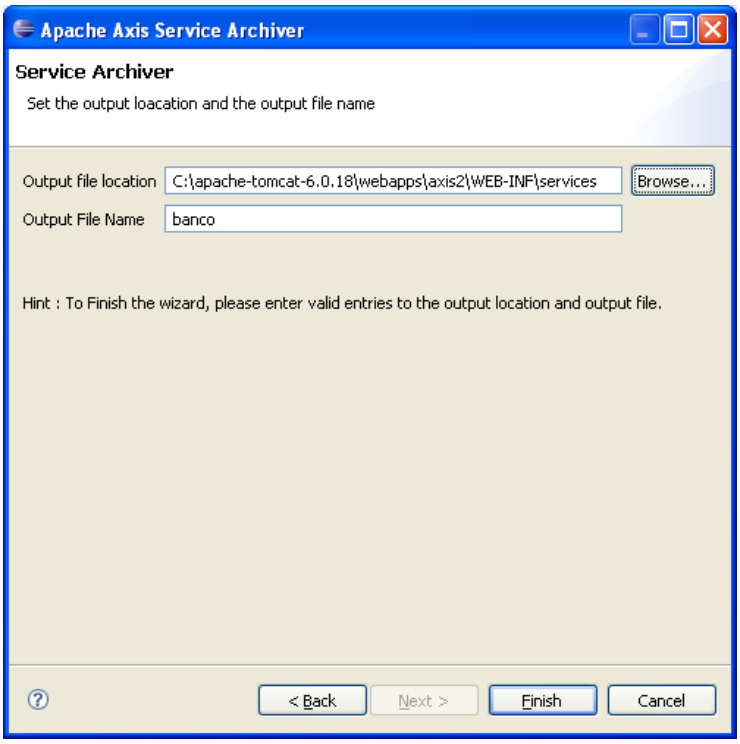

Figura 16

Si todo ha ido bien, el *.aar* se habrá generado correctamente y veremos el siguiente cuadro de diálogo:

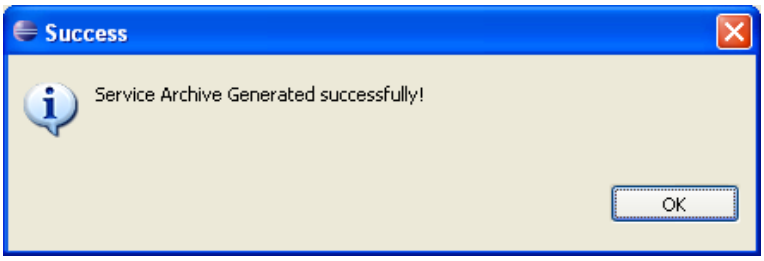

Figura 17

A continuación, y para comprobar que efectivamente el servicio web está publicado y en ejecución, arrancamos Tomcat y tecleamos la URL de publicación en un navegador (http://localhost:8080/axis2/services/listServices):

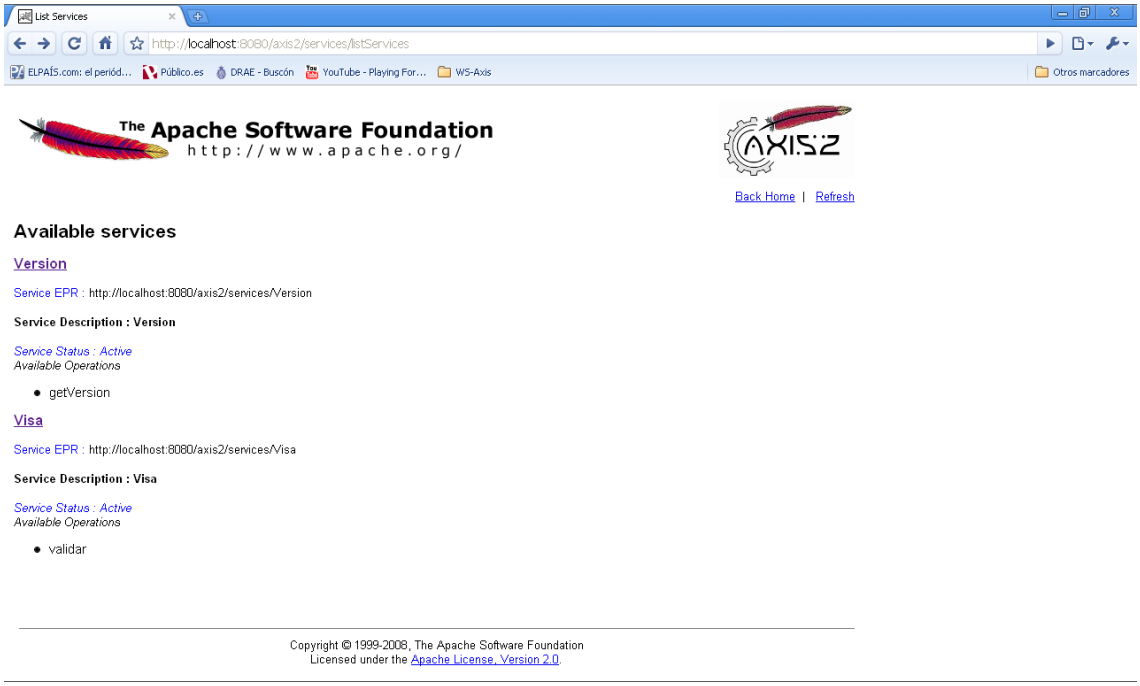

Figura 18. Lista de servicios publicados

Si pinchamos en el servicio que nos interesa (*Visa)*, y dependiendo del navegador que utilicemos, podremos ver el contenido del documento WSDL que se ha generado con la descripción del servicio. Si usamos Internet Explorer, tenemos lo siguiente:

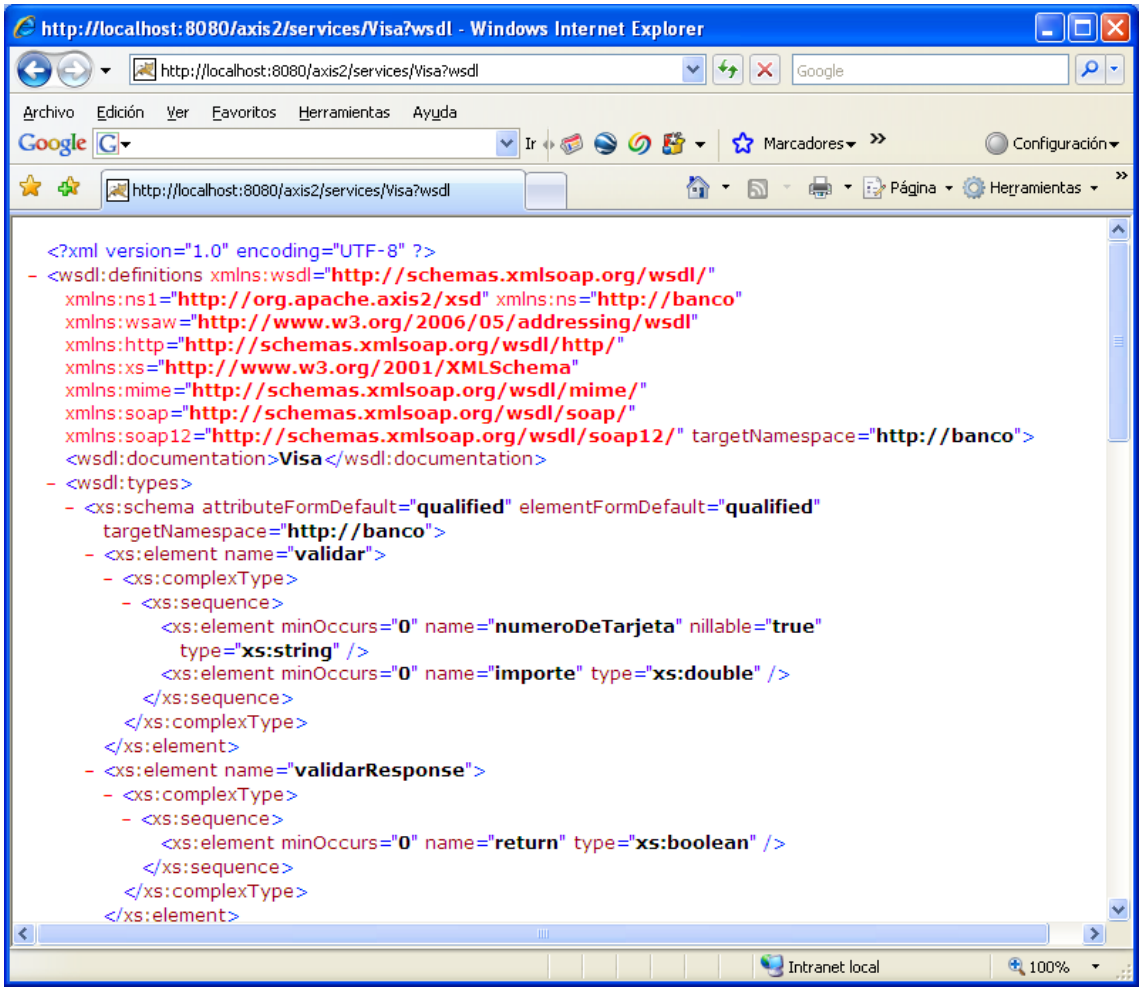

Figura 19. Documento WSDL con la descripción del servicio que acabamos de publicar

1.05 Acceso a servicios web

Hasta ahora hemos visto cómo ofrecer operaciones mediante servicios web. A continuación veremos cómo acceder a esas operaciones: es decir, aprenderemos a construir aplicaciones que hagan uso de las operaciones ofrecidas mediante el servicio web.

El resultado que obtengamos dependerá del entorno de desarrollo que utilicemos y de las herramientas y plugins seleccionadas: en nuestro caso, Eclipse con Axis2 y un par de plugins de Axis2 para Eclipse.

En primer lugar, creamos un proyecto nuevo que representará un cliente del servidor de servicios web. En el árbol del proyecto, elegimos *New*, *Other*, *Axis2 Code Generator* (Figura 20).

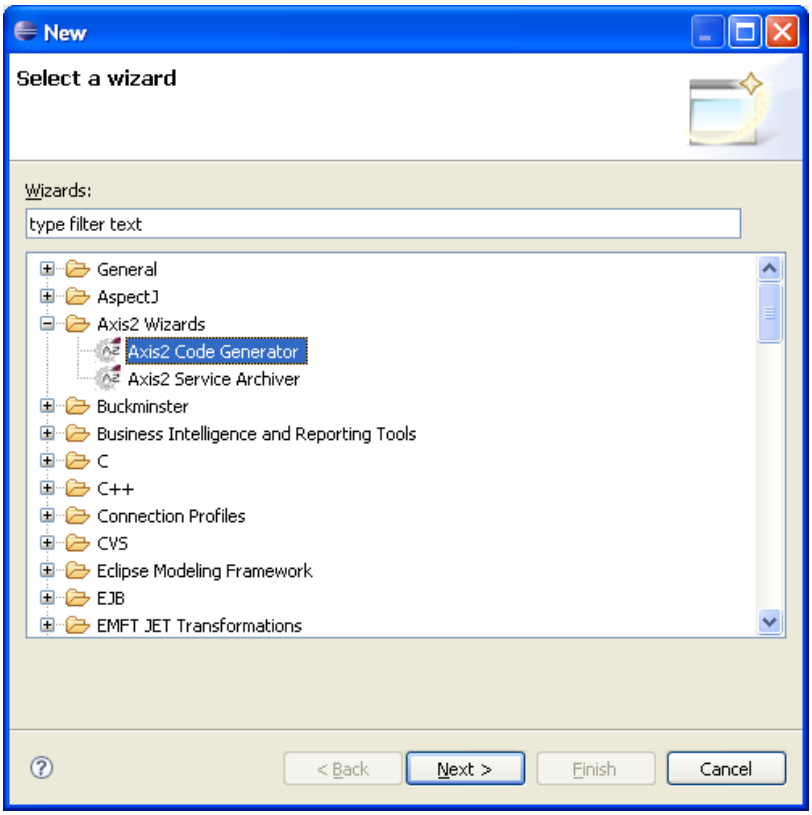

Figura 20

Tras pulsar *Next*, marcamos la primera opción que se nos muestra (generar código Java a partir de un WSDL):

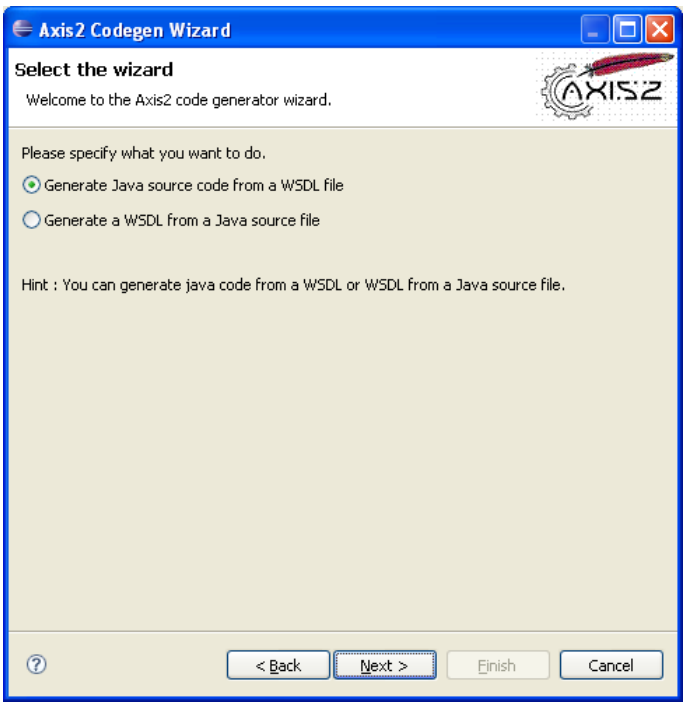

Figura 21

En el siguiente paso indicamos la ubicación del documento WSDL, que puede ser la URL publicada en el servidor web, y que tenemos en la barra de direcciones del navegador mostrado en la Figura 19:

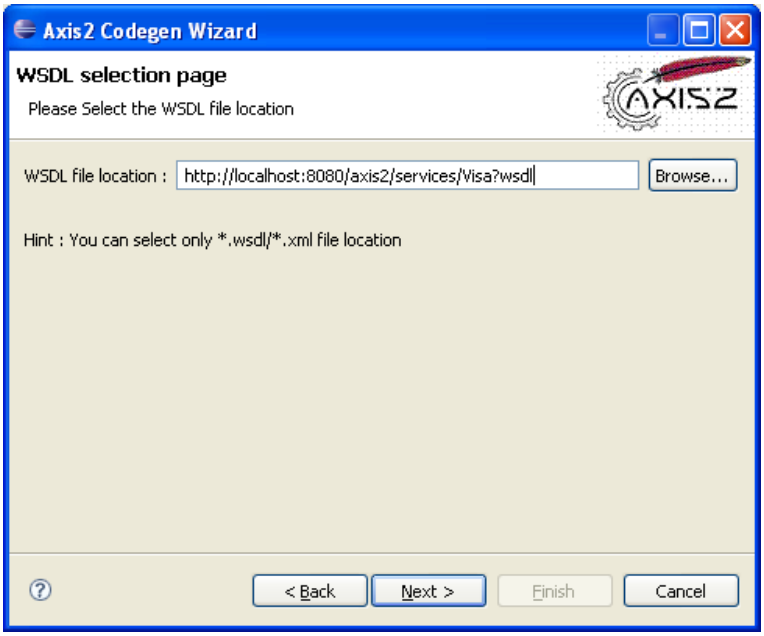

Figura 22

En el siguiente paso, dejamos la configuración por defecto (Figura 23) e indicamos, en la última ventana del asistente (Figura 24), el proyecto cliente que acabamos de crear.

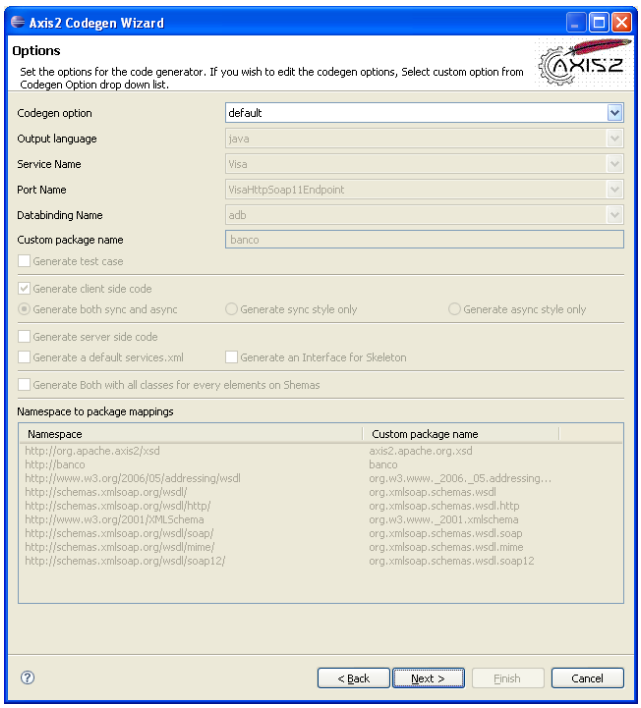

Figura 23

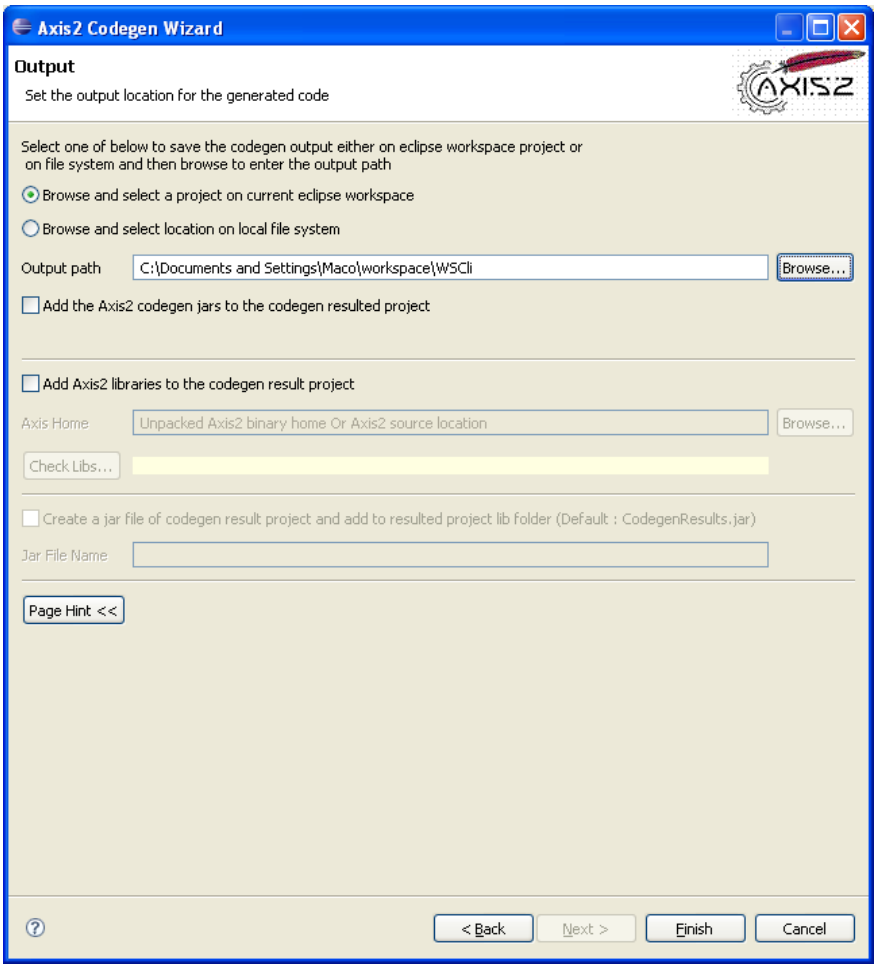

Figura 24

Tras pulsar Finish, el asistente nos habrá generado un conjunto de clases, que ya podemos utilizar en nuestro proyecto para acceder al servicio web. En este ejemplo, el conjunto de clases generado es el siguiente:

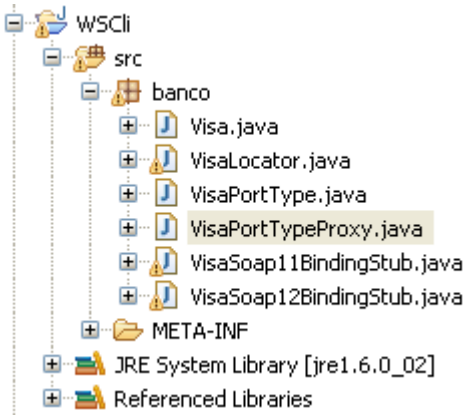

Figura 25. Conjunto de clases generado por el asistente

De todas éstas, la clase que más nos interesa es *VisaPortTypeProxy*, que representa el *proxy* (véase sección 1.02, página 3) que permite a la aplicación cliente el acceso a la aplicación web.

El ejemplo que hemos venido desarrollando es muy sencillo, pero nos sirve como ilustración: creamos una clase nueva en la que ubicamos el método *main* que se muestra a continuación:

```
public static void main(String[] args) throws ServiceException, RemoteException {
      VisaPortTypeProxy vp=new VisaPortTypeProxy();
      boolean x=vp.validar("1234", 68.0);
      System.out.print(x);
}
```

Figura 26. Código que accede a la operación *validar* a través del *proxy*.

1.06 Tipos de datos complejos

Supongamos que añadimos al servicio web una nueva operación (*public boolean validarTarjeta(Tarjeta tarjeta, double importe)*) que también deseamos ofrecer como un servicio web. Como se ve, el primer parámetro es del tipo complejo *Tarjeta*; supongamos que las operaciones de esta clase, sin que nos importe la forma en que están implementadas, son las siguientes:

public Tarjeta()

public String getNumero()

Para publicar las dos operaciones (la *validar* antigua y esta nueva versión, *validarTarjeta*), seguimos los pasos descritos en la sección 1.04: Axis2 se habrá encargado de generar el documento WSDL correspondiente, con la especificación de las dos operaciones y, también, del fragmento del tipo complejo *Tarjeta* que interese (obsérvese la definición del tipo complejo en el WSDL en la Figura 27):

```
- <xs:complexType name="Tarjeta">
  - <xs:sequence>
      <xs:element minOccurs="0" name="numero" nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figura 27. Definición WSDL del tipo complejo *Tarjeta*

La operación ya está accesible a partir del servicio web. Para generar el *proxy* y el resto de clases auxiliares, seguimos los pasos indicados en la sección 1.05 y observamos que se crea un subpaquete que contiene la especificación Java de los tipos de datos complejos, según la definición dada en el WSDL (Figura 28). Nótese que el código de la clase *Tarjeta* añadida al proyecto cliente no coincide con el código del mismo tipo en

el servidor, ya que la *Tarjeta* del cliente contiene el código mínimo necesario para que el cliente pueda manipular instancias de tipo *Tarjeta* con las que comunicarse con el servidor.

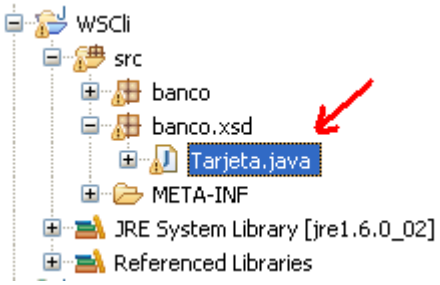

Figura 28. Código Java generado en el cliente a partir del tipo *Tarjeta* del servidor

El nuevo código de utilización del *proxy* puede ser el que sigue:

```
public static void main(String[] args) throws RemoteException  {
     VisaPortTypeProxy vp=new VisaPortTypeProxy();
     boolean x=vp.validar("1234", 68.0);
     System.out.print(x);

     banco.xsd.Tarjeta t=new banco.xsd.Tarjeta("5678");
     x=vp.validarTarjeta(t, 55.0);
     System.out.println(x);
}
```

Figura 29. Código de utilización del *proxy*, que usa ahora el tipo complejo *Tarjea*