J2EE 1.4 Web Services

Object Computing, Inc.
Mark Volkmann, Partner
11/21/03

Fundamental Web Service Specs.

- Simple Object Access Protocol (SOAP)
  - specifies the structure of XML-based request and response messages used by web services
  - specifies how certain data types can be encoded (called SOAP encoding)
- Web Services Description Language (WSDL)
  - describes all aspects of a web service in an XML document
    - message structure (typically defined using XML Schema)
    - style (RPC or document)
    - encoding details (SOAP, literal, …)
    - transport details (HTTP, SMTP, JMS, …)
    - service location (endpoint URL)
  - often used by tools to generate client proxies/stubs that invoke the web service

Fundamental Web Service Specs. (Cont'd)

- Universal Description, Discovery and Integration (UDDI)
  - most common type of XML registry
  - supports querying and updating via web service operations
  - provides information about companies and services
    - not necessarily web services
  - run-time selection?
    - a goal of XML registries was to allow applications to select web service implementations at run-time
    - this idea has not been embraced
    - today, XML registries are used to select web services at design-time

WS-I Basic Profile

- Specification from
  Web Services Interoperability (WS-I) organization
  - http://www.ws-i.org
- Clarifies ambiguities in
  XML, SOAP, WSDL and UDDI specs.
- Recommends how to increase interoperability
  when using them
  - what features to use and avoid
- Web service APIs in J2EE 1.4 support
  features recommended by Basic Profile
  - in addition to some that are not such as
    attachments and RPC/encoded messages

Web Services in J2EE 1.4

- From the J2EE 1.4 spec.
  - "The primary focus of J2EE 1.4 is support for web services."
- Java-specific web service APIs supported
  - Java API for XML Remote Procedure Calls (JAX-RPC)
  - SOAP with Attachments API for Java (SAAJ)
  - Java API for XML Registries (JAXR)

detail on these is coming up

JAX-RPC

- Supports invoking web services in three ways
  - generated stubs (from WSDL)
    - can be used with both RPC and document style services
  - dynamic proxies
    - benefit of this approach is questionable so it won't be covered
  - Dynamic Invocation Interface (DII)
    - used internally by generated stubs to invoke web service operations
    - similar to calling methods using reflection
    - can be used with both RPC and document style services
- Supports implementing web services in two ways
  - plain Java classes
    - called Java Service Endpoint (JSE)
  - EJB stateless session beans
    - can utilize transactional capabilities of EJBs

When using **generated stubs with document-style services**, the return type of all operations is a SAAJ SOAPElement.

When using **DII with document-style services**, custom serializers must be generated at build-time for non-primitive parameter/return types (defined using XML schema in WSDL). If a tool must be run at build time, why not run the tool that generates a client stub instead?

SAAJ

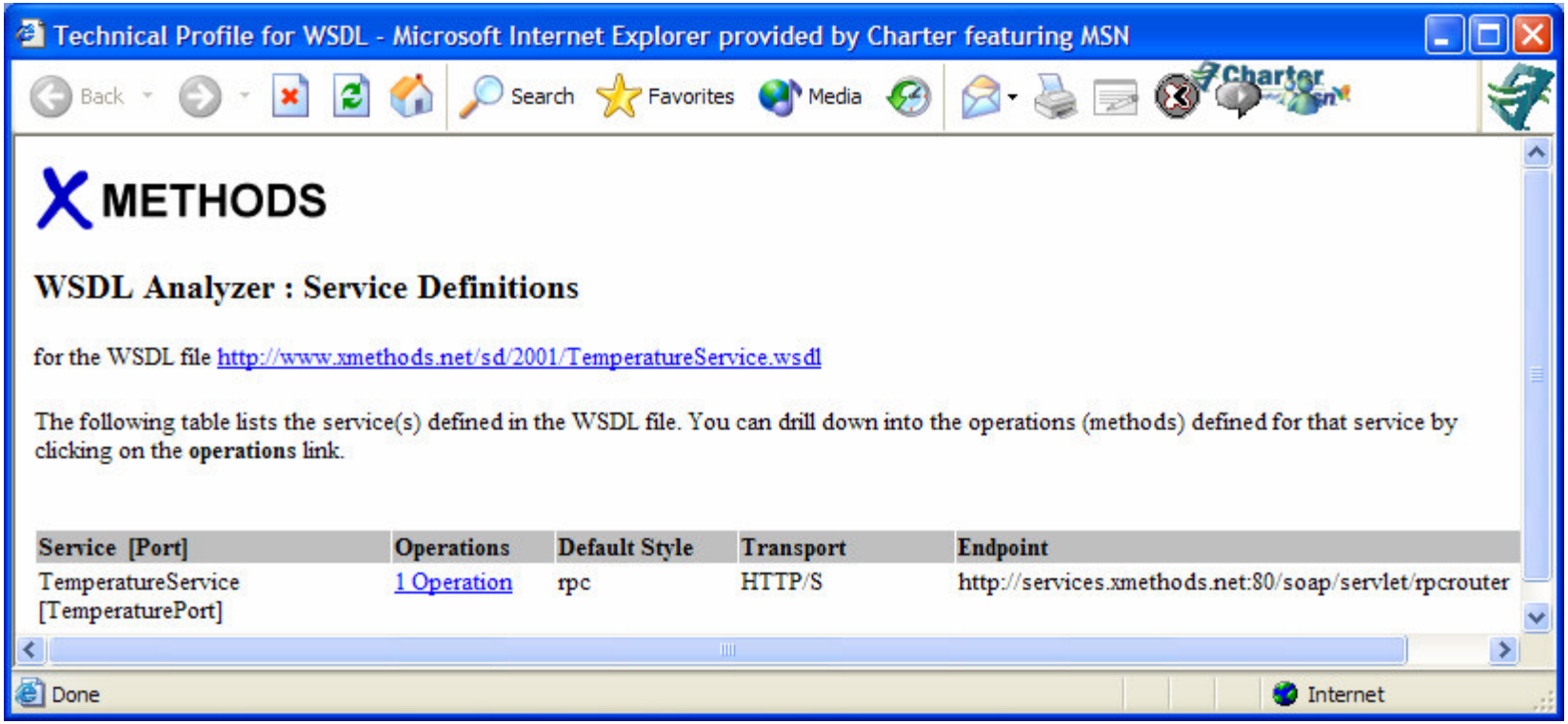
- Pronounced “sage”
- Provides classes that model SOAP messages
- Used by JAX-RPC
- Can be used by itself to write SOAP clients
  - provides maximum control of building requests and processing responses
  - ideal for document-style services, but works with RPC-style too
- Useful even when attachments aren’t being used
- Relationship to the Java API for XML Messaging (JAXM)
  - the contents of the SAAJ spec. used to be part of the JAXM spec.
    - JAXM and JAX-RPC now both depend on SAAJ
  - JAXM also defines capabilities similar to JMS for web services
    - asynchronous, guaranteed messaging
  - support for JAXM is waning and it may not survive

JAXR

- Provides a Java API for querying and updating XML registries such as UDDI
- Hides details of
  - creating and sending SOAP requests to registry operations
  - receiving and parsing SOAP responses from registry operations

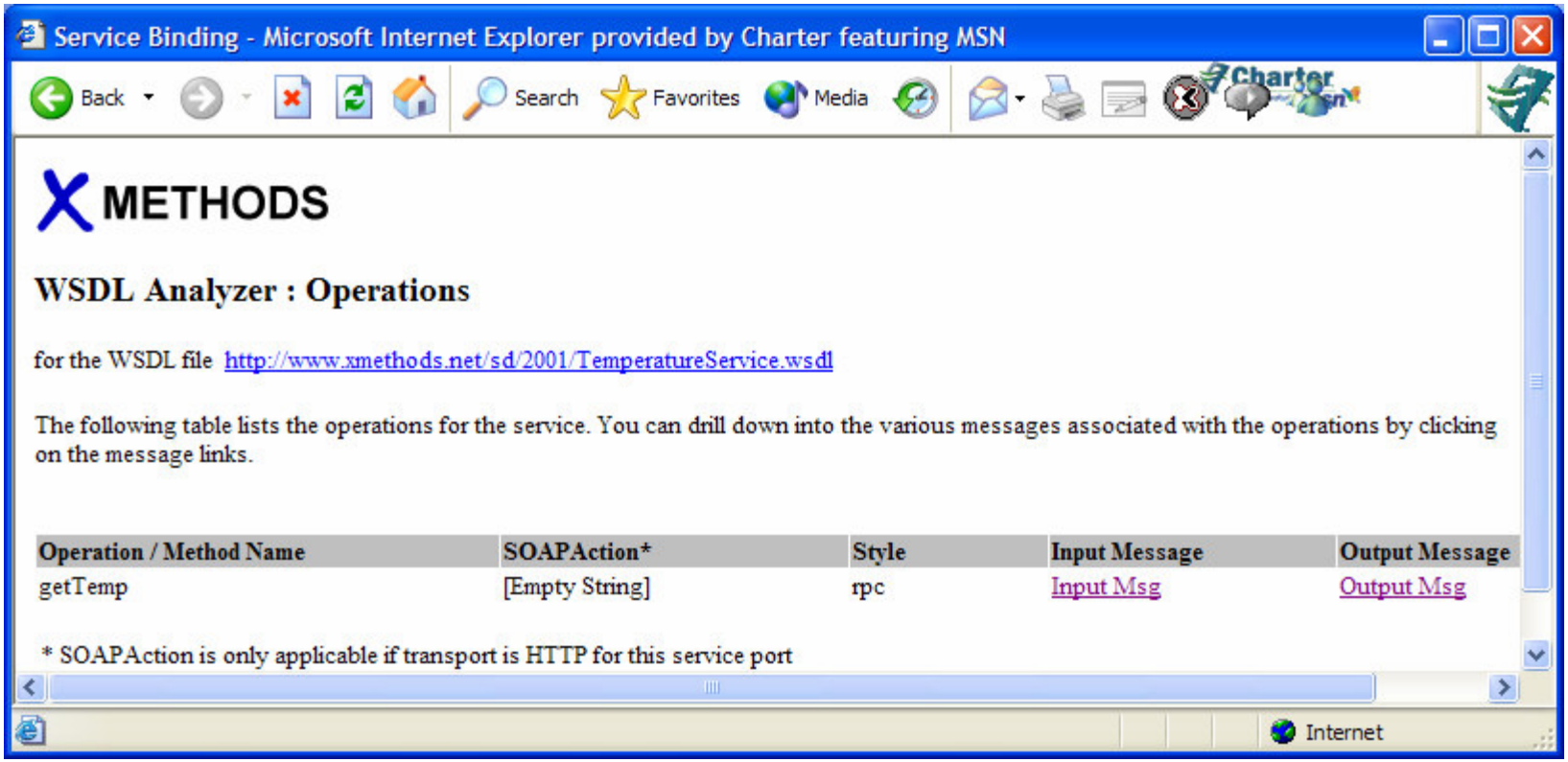
Demonstration Web Service

- "Weather - Temperature" service at http://xmethods.com
- Clicking "Analyze WSDL" link displays the following



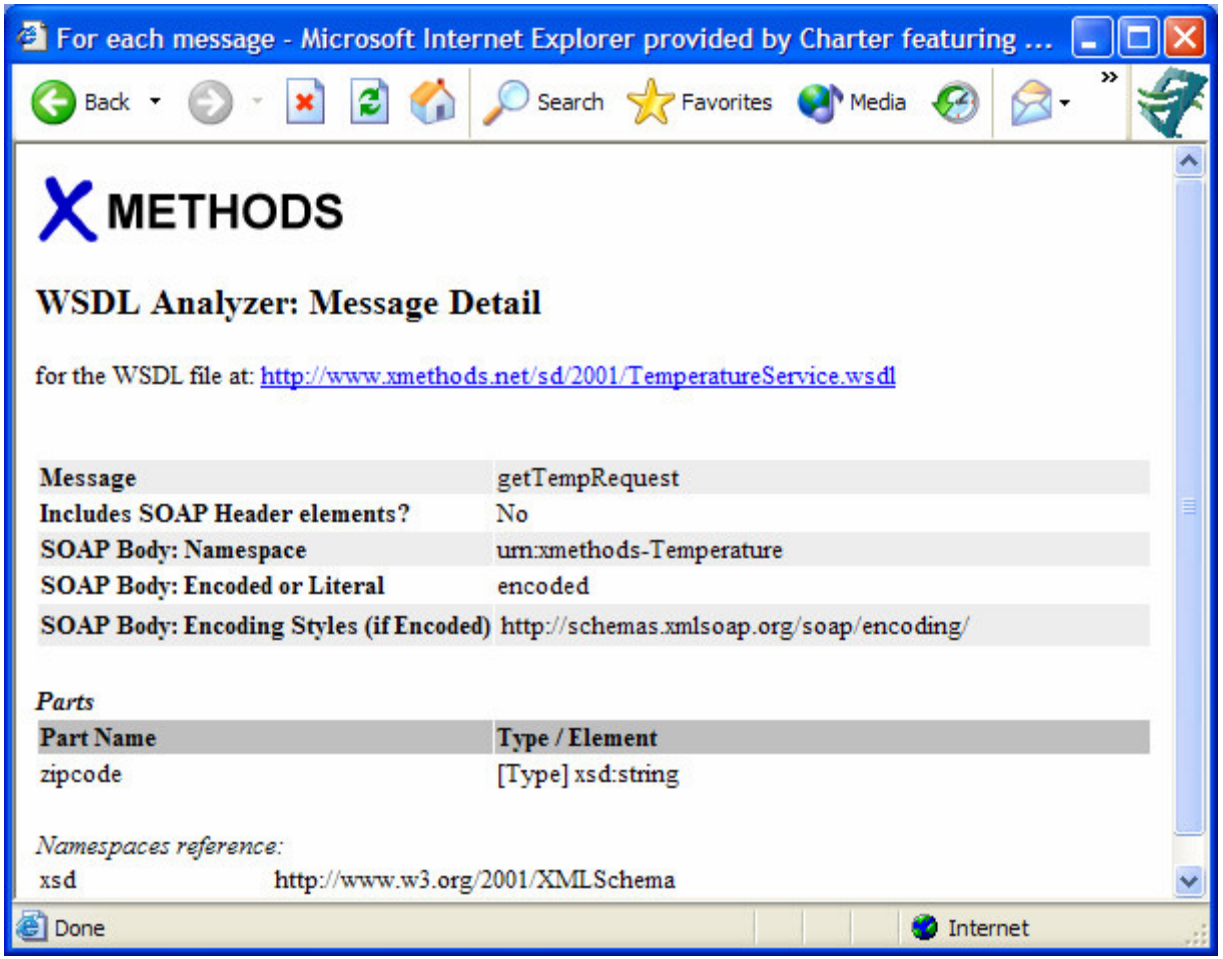
Demonstration Web Service (Cont'd)

- "Operation" page



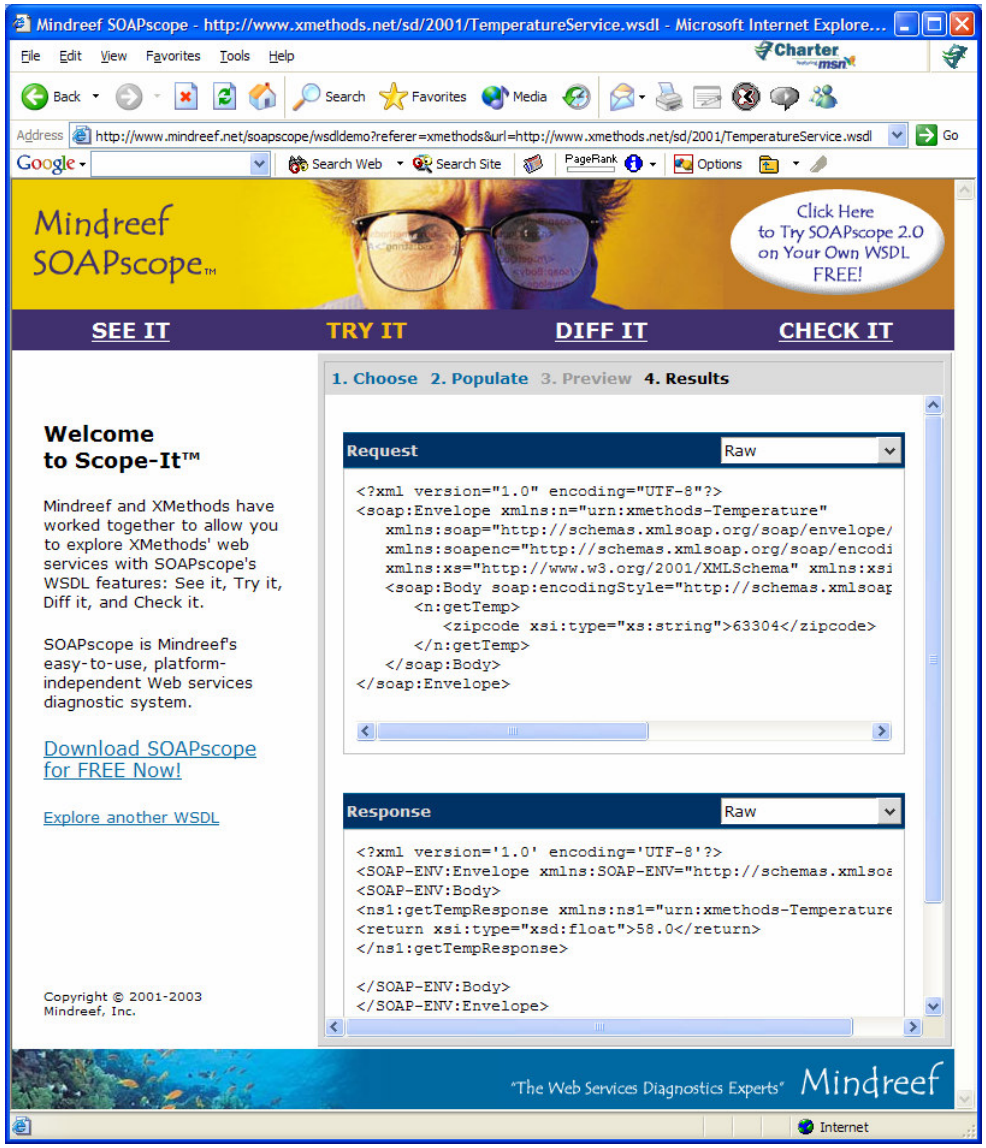
Demonstration Web Service (Cont'd)

- "Input Msg" page



Demonstration Web Service (Cont'd)

- "Try It" page



Temperature Service WSDL

```
<?xml version="1.0"?>
<definitions name="TemperatureService"
  targetNamespace="http://www.xmethods.net/sd/TemperatureService.wsdl"
  xmlns:tns="http://www.xmethods.net/sd/TemperatureService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="getTempRequest">
    <part name="zipcode" type="xsd:string"/>
  </message>
  <message name="getTempResponse">
    <part name="return" type="xsd:float"/>
  </message>
  <portType name="TemperaturePortType">
    <operation name="getTemp">
      <input message="tns:getTempRequest"/>
      <output message="tns:getTempResponse"/>
    </operation>
  </portType>
```

Temperature Service WSDL (Cont'd)

```
<binding name="TemperatureBinding" type="tns:TemperaturePortType">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getTemp">
    <soap:operation soapAction=""/>
    <input>
      <soap:body use="encoded" namespace="urn:xmethods-Temperature"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <soap:body use="encoded" namespace="urn:xmethods-Temperature"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>
```

Temperature Service WSDL (Cont'd)

```
 <service name="TemperatureService">
   <documentation>
     Returns current temperature in a given U.S. zipcode
   </documentation>
   <port name="TemperaturePort" binding="tns:TemperatureBinding">
     <soap:address
       location="http://services.xmethods.net:80/soap/servlet/rpcrouter"/>
    </port>
  </service>
</definitions>
```

Temperature Service Request

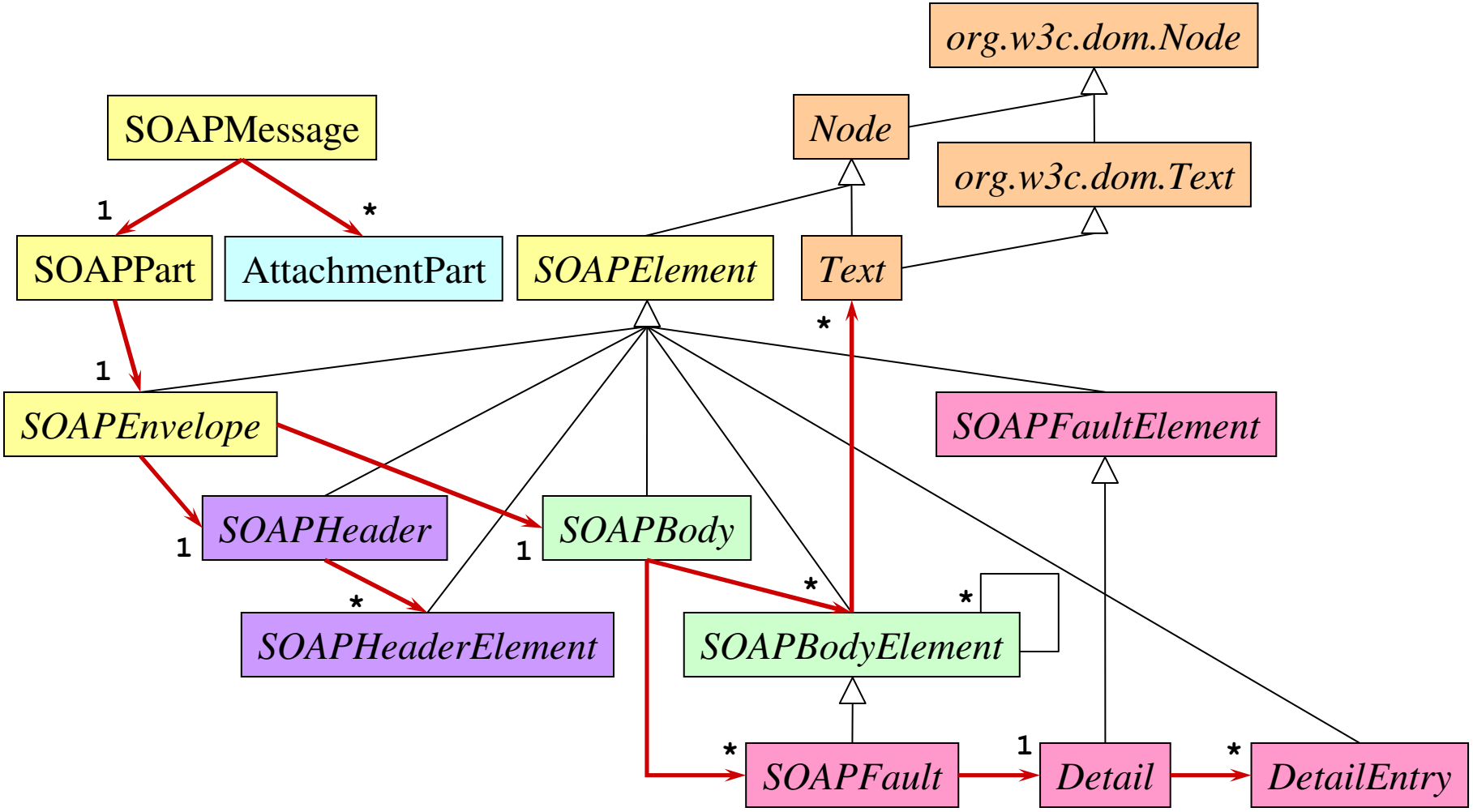
```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:n="urn:xmethods-Temperature"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema”
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body
    soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <n:getTemp>
      <zipcode xsi:type="xs:string">63304</zipcode>
    </n:getTemp>
  </soap:Body>
</soap:Envelope>
```

Temperature Service Response

```
<?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/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:getTempResponse
      xmlns:ns1="urn:xmethods-Temperature"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:float">58.0</return>
    </ns1:getTempResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SAAJ API

(in javax.xml.soap package)



SAAJ Web Service Client

```
package com.ociweb.temperature;

import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.soap.*;
import org.w3c.dom.Node;

public class Client {
  private static final String ENDPOINT =
    "http://services.xmethods.net:80/soap/servlet/rpcrouter";
  private static final String NAMESPACE = "urn:xmethods-Temperature";
  private static final String OPERATION = "getTemp";
  private static final String SOAP_ENCODING_NS =
    "http://schemas.xmlsoap.org/soap/encoding/";
  private static final String SOAP_ENVELOPE_NS =
    "http://schemas.xmlsoap.org/soap/envelope/";
```

SAAJ Web Service Client (Cont'd)

```
private SOAPElement zipElement;
private SOAPMessage request;

public static void main(String[] args) throws Exception {
  Client client = new Client();
  String zip = "63304";
  System.out.println("temperature in " + zip +
    " is " + client.getTemperature(zip));
}
```

SAAJ Web Service Client (Cont'd)

```
public Client() throws MalformedURLException, SOAPException {
  MessageFactory mf = MessageFactory.newInstance();
  request = mf.createMessage();
  request.getSOAPHeader().detachNode(); // not using SOAP headers
  SOAPBody body = request.getSOAPBody();

  // Specify that the SOAP encoding style is being used.
  SOAPFactory soapFactory = SOAPFactory.newInstance();
  Name name = soapFactory.createName
    ("encodingStyle", "SOAP-ENV", SOAP_ENVELOPE_NS);
  body.addAttribute(name, SOAP_ENCODING_NS);

  SOAPElement operationElement =
    body.addChildElement(OPERATION, "n", NAMESPACE);
  zipElement = operationElement.addChildElement("zipcode");
}
```

builds request like on p. 16

SAAJ Web Service Client (Cont'd)

```
public float getTemperature(String zipCode)
throws IOException, SOAPException {
   // Populate request message with parameter values.
   zipElement.addTextNode(zipCode);
   dumpMessage("request", request); // for debugging

   // Make the call.
   SOAPConnectionFactory scf = SOAPConnectionFactory.newInstance();
   SOAPConnection connection = scf.createConnection();
   SOAPMessage response = connection.call(request, new URL(ENDPOINT));
   connection.close();
   dumpMessage("response", response); // for debugging
```

SAAJ Web Service Client (Cont'd)

```
    // Get result out of response message using DOM.
    SOAPBody body = response.getSOAPBody();
    SOAPElement responseElement =
      getFirstChild(body, OPERATION + "Response");
    SOAPElement returnElement =
      getFirstChild(responseElement, "return");
    String value = returnElement.getValue();

    zipElement.removeContents(); // prepare for future calls

    return new Float(value).floatValue();
}
```

parses response like on p. 17

SAAJ Web Service Client (Cont'd)

```
  private static void dumpMessage(String name, SOAPMessage message)
  throws IOException, SOAPException {
    System.out.println(name + " message is");
    message.writeTo(System.out);
    System.out.println();
  }

  private static SOAPElement getFirstChild
    (Node parent, String localName) {
    Node child = parent.getFirstChild();
    while (child != null) {
      if (localName.equals(child.getLocalName())) break;
      child = child.getNextSibling();
    }
    return (SOAPElement) child;
  }
}
```

Creating and using a class containing SAAJ-related utility methods would simplify this code!

JAX-RPC DII Web Service Client

```
package com.ociweb.temperature;

import java.rmi.RemoteException;
import javax.xml.namespace.QName;
import javax.xml.rpc.*;

public class Client {
  private static final String ENDPOINT =
    "http://services.xmethods.net:80/soap/servlet/rpcrouter";
  private static final String NAMESPACE = "urn:xmethods-Temperature";
  private static final String OPERATION = "getTemp";
  private static final String PORT = "TemperaturePort";
  private static final String SERVICE = "TemperatureService";

  private Call call;
```

JAX-RPC DII Web Service Client (Cont'd)

```
public static void main(String[] args) throws Exception {
  Client client = new Client();
  String zip = "63304";
  System.out.println("temperature in " + zip +
    " is " + client.getTemperature(zip));
}

public float getTemperature(String zipCode) throws RemoteException {
  Float temperature = (Float) call.invoke(new Object[] {zipCode});
  return temperature.floatValue();
}
```

JAX-RPC DII Web Service Client (Cont'd)

```
public Client() throws ServiceException {
  ServiceFactory factory = ServiceFactory.newInstance();
  Service service = factory.createService(new QName(SERVICE));

  QName port = new QName(NAMESPACE, PORT);
  QName operation = new QName(NAMESPACE, OPERATION);
  call = service.createCall(port, operation);
  call.setTargetEndpointAddress(ENDPOINT);
  call.addParameter("zipcode", XMLType.XSD_STRING, ParameterMode.IN);
  call.setReturnType(XMLType.XSD_FLOAT);
  call.setProperty(Call.ENCODINGSTYLE_URI_PROPERTY,
    NamespaceConstants.NSURI_SOAP_ENCODING);

  // Some services require setting the SOAPAction HTTP header,
  // but this one doesn't.
  //call.setProperty(Call.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
  //call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
 }
}
```

The code for getting the Service object in a J2EE client such as a servlet or an EJB uses JNDI.

JAX-RPC Generated Stub Web Service Clients

- Supplied tool reads WSDL and generates stubs
  - web service toolkits such as Axis, JWSDP and WebLogic provide such a tool
  - details differ
    - Axis provides `wsdl2java`
    - Java Web Service Developer Pack (JWSDP) provides `wscompile`
    - WebLogic provides `clientgen`
  - also generates data holder classes for types defined in WSDL
    - defined using XML Schema

JAX-RPC Generated Stub Web Service Clients

- JWSDP includes a script to generate stubs
  - `${jwsdp.home}/jaxrpc/bin/wscompile.bat` or `.sh`
  - generates several source files and compiles them
- Generating stub classes using JWSDP and Ant

```
<exec executable="${jwsdp.home}/jaxrpc/bin/wscompile.bat">
  <arg line="-classpath ${classes.dir}"/>
  <arg line="-gen:client"/>
  <arg line="-keep"/>
  <arg line="-d ${classes.dir}"/>
  <arg line="config.xml"/>
</exec>
```

to keep generated source files

This is also a custom Ant task now. Use that instead of the exec task.

- config.xml (JWSDP-specific)

```
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl location="http://www.xmethods.net/sd/TemperatureService.wsdl"
        packageName="com.ociweb.temperature"/>
</configuration>
```

can be a local file

JAX-RPC Generated Stub Web Service Client

```
package com.ociweb.temperature;

public class Client {

  public static void main(String[] args) throws Exception {
    // Get the stub.
    ServiceFactory sf = ServiceFactory.newInstance();
    TemperatureService service = (TemperatureService)
      sf.loadService(TemperatureService.class);
    TemperaturePortType stub = service.getTemperaturePort();

    // Use the stub.
    String zip = "63304";
    float temperature = stub.getTemp(zip);
    System.out.println("temperature in " + zip + " is " + temperature);
  }
}
```

The code for getting the Service object in a J2EE client such as a servlet or an EJB uses JNDI.

no casting or conversion of the response is needed

Summary

- Clearly using generated stubs is easier than SAAJ and DII
- SAAJ and DII are useful when
  - WSDL isn't available
    - but it should always be available
  - web service to be invoked isn't known until runtime
    - not a likely scenario
- SAAJ provides maximum control over
  - building request messages
  - processing response messages
- DII is still necessary since it is used by generated stubs
- SAAJ can be used by DII implementations

What About Ruby?

- Web services in Ruby are supported by SOAP4R
- SOAP4R includes wsdl4ruby.rb script
  - parses WSDL
  - generates Ruby class that invokes operations described in WSDL
  - generates sample Ruby client class
- Example
  - ```
    wsdl2ruby.rb \
    --wsdl http://www.xmethods.net/sd/TemperatureService.wsdl \
    --type client
    ```
    - generates TemperatureServiceDriver.rb and TemperatureServiceClient.rb
  - code in generated client is similar to the following

```
require 'TemperatureServiceDriver.rb'
stub = TemperaturePortType.new() # can pass endpoint URL
zipcode = '63304'
temperature = stub.getTemp(zipcode)
puts "temperature in #{zipcode} is #{temperature}"
```
```