Bergen Center for Computational Science
COMPUTATIONAL BIOLOGY UNIT

Web Services Tutorial

Center for Scientific Computing, Espoo Finland.

Jan Christian Bryne, chrb@ii.uib.no
Anders Lanzen, Anders.Lanzen@bccs.uib.no
Computational Biology Unit, Bergen Center for Computational Science

Contents

1 About this document

The purpose of the document is to provide a brief theoretical overview of SOAP and Web Services technology, as well as describing a number of exercises. While some of the material needed to carry out these exercises are available on a dedicated website, they depend on a preconfigured environment configured for the course. We do not recommend that you try Exercise 2 outside the course lab provided during the course.

Web Services is a relatively recent technology that is rapidly evolving. The recommendations and descriptions in this document were to the best of our knowledge up-to-date in the summer of 2006, but we expect that will have changed within a short time.

2 XML

XML is short for eXtensible Markup Language. Information is structured using *elements* and *attributes*.

Elements are the name of the tags:

```
<Element> ... </Element>
```

Attributes are defined inside an Element tag like this:

```
<Element attribute="..."> ... </Element>
```

It is generally easier for computers to parse information structured with elements instead of as attributes.

2.1 XML Schema

XML Schema is a way to define the structure of XML documents. It can be used to define data types in a data model. An XML document can be an instance of an XML Schema. Such XML documents can be validated against the Schema to verify that it conforms to it.

XML Schema can be compared to class definitions in object oriented programming. Just like a data type in a data model can be expressed in a computer language using class definitions, data types can also be expressed using XML Schema. This enables exchange of data types between different programming languages and platforms. This important feature of XML is utilised heavily in Web Services.

2.2 Namespaces

XML *namespaces* is a method to avoid conflicts between elements with the same name. This type of conflict may arise if an XML document contains elements from several XML Schema and an element name occurs in more than one schema. It can be resolved by defining a *namespace* for each element, such as:

```
<namespace:Element> ... </namespace:Element>
```

To avoid repeating a namespace declaration for each element that belongs to it, a prefix declaration can be made in a parent element. If, for example we have two different elements with the name "element" and want to use two separate namespaces for these, we would write:

```
<ParentElement xmlns:ns1="http://mysite.com/NS1" xmlns:ns2="http://mysite.com/NS2">
    <ns1:Element> ... </ns1:Element>
    <ns2:Element/>
</ParentElement>
```

The first element in the example above belongs to namespace "http://mysite.com/NS1" while the second belongs to "http://mysite.com/NS2". The use of http network addresses for namespace declarations is just a convention. Any string could be used.

If XML concepts such as namespaces and XML Schema seem too abstract or unfamiliar, have a look at the XML Tutorial at W3 Schools at http://www.w3schools.com/xml/.

3 Web Service Technologies

A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. Web services are self-contained, self-describing, modular and platform independent. Other systems interact with the Web Service in a manner prescribed by the service itself using SOAP messages (see Section 3.1) , normally transferred using HTTP.

Thanks to the high level of standardisation that Web Services require, developers will find it easy to integrate and use them. Combined with their widespread use, this makes them a very suitable technology for integration of services. Most popular development platforms and high-level programming languages today provide functionality that makes it easy to access Web Services. You will find an example of this in Section 4.3.

3.1 SOAP

SOAP is a framework for exchange of XML documents. It enjoys the status of a recommendation by the World Wide Web Consortium (W3C), which can be found at http://www.w3.org/TR/soap/.

SOAP introduces some additional structure in XML documents. A *SOAP Message* is an Envelope element that consists of an optional Header element and a required Body element:

```
<Envelope>
   <Header> ... </Header>
   <Body> ... </Body>
</Envelope>
```

This structure allows separation of meta data contained in the Header from the message payload contained in the Body. Such meta data can be routing information, expiration information, or other.

The header is usually not used in Plain Vanilla (ordinary) Web Services, but we will show an example later in the course that will demonstrate how it may be used. The Body element can contain any XML.

SOAP also defines:

- a processing model
- a mechanism for error handling
- an extensibility model
- a mechanism for data representation
- a convention for Remote Procedure Calls (RPC)
- a protocol binding framework

3.1.1 SOAP Engines

A SOAP engine is a framework used in servers and clients that facilitates:

1. Serializing objects from a programming language into SOAP messages; and
2. De-serializing SOAP messages into objects in a programming language, i.e. creating appropriate data types and populating these with the message content.

The SOAP engines we will use in this course are SOAPpy implemented in Python and Axis which is implemented in Java.

In addition to acting as a SOAP engine, Axis is capable of generating server and client code automatically, based on either Java classes or a WSDL file (see Section 3.2). For more information, visit ws.apache.org/axis.

3.2 Web Services Description Language (WSDL)

The interface of a Web Service is defined using the XML format Web Services Description Language (WSDL). The interface is independent of the Web Service implementation. A WSDL document may look unnecessarily complex at first glance, but they are not primarily meant to be read by humans. While providing a high level of flexibility in terms of the Web Service interface, WSDL documents must also ensure that the WSDL-to-SOAP mapping is unambiguous. In other words, the Web Service interface must always be interpreted in the same way by clients accessing it.

The formal specification of WSDL can be found at www.w3.org/TR/wsdl. Version 2.0 is currently a submission to the W3C, i.e. not yet accepted as a recommendation.

A WSDL document defines one or several *operations*, and the messages they receive and return. These operations and messages are first described abstractly and then bound to one or more concrete instances in terms of protocol, message format and network address, known as an *endpoint*. This allows for flexibility and reuse of abstract definitions.

Simple overview of a WSDL document (each element may occur more than once):

```
<definitions>
   <types/>
   <message/> <part/> </message>
   <portType> <operation/> </portType>
   <binding/>
   <service> <port/> </service>
</definitions>
```

As you can see from the overview above, WSDL document consists of the following parts:

`types`: data type declarations using some type system (normally XML Schema). External XML Schema may also be imported.

`message` elements : abstract definitions of what can be communicated, i.e. incoming and outgoing messages, and their data types. Data types are defined in `part` child elements and may refer to types declared in the `types` element described above.

`portType`: a Port Type contains one or more `operation` elements. Each Operation abstractly specifies an action supported by the service and the Messages that may be sent to and from it. The set of Operations in a Port Type is supported by one or more endpoints.

`binding`: a Binding defines message format and protocol details for Operations and messages defined by a Port Type, i.e. the WSDL-to-SOAP mapping of the service (providing that the Binding uses SOAP). The data formatting regulations and encoding are usually RPC/encoded, RPC/literal or document/literal. This is also known as the *style* of the Web Service.

`service`: A collection of endpoints. Each endpoint is represented by a `port` element and each port defines a network address and a name for a Binding, thus defining a Web Service instance.

3.2.1 Exercise 1 : Exploring the ELM Database WSDL

Using a web browser, open (a simplified version of) the WSDL for the ELM database at http://www.bioinfo.no/courses/wscourse/ELMdb.wsdl. Please note that some browsers does not display all information in an XML file. In that case you need to view the file by choosing "view source" or equivalent. ELM is short for Eukaryotic Linear Motifs and is a resource for predicting functional sites in eukaryotic proteins. Putative functional sites are identified by patterns (regular expressions). Have a look at the WSDL in order to get a feeling of the different operations this Web Service offers and what data types they use. Have a closer look at the `getELM` Operation. As you can see in its declaration, it requires a `getELMRequest` as input message. What data type does such a message contain? Browse through the WSDL upwards and look at `getELM` in the portType declaration and then the `getELMRequest` Message and then the `getELM` element in the types declaration.

As you can see, this element contains an `ELMAccessionType` that is not declared in this WSDL. In fact, it is imported from the XML Schema at http://api.bioinfo.no/schema/ELM.xsd as can be seen in the import declaration in the beginning of `types`. Open this XML Schema in a new window and look at the type declaration of the `ELMAccessionType`. Can you deduct anything about the format of the Accession Numbers used in ELM from this? How many characters does such an Accession Number have?

3.2.2 Web services interoperability (WS-I)

Web services interoperability(WS-I) introduces a higher level of standardization than what is outlined by WSDL. This is useful in order to ensure full interoperability of Web Services. Rather than a standard in itself, WS-I is the name of an organisation designed to promote interoperability of Web Services across different platforms and languages. For more information about WS-I, visit www.ws-i.org. The WS-I Basic Profile is an outline of requirements for WSDL and Web service protocol, in terms of SOAP/HTTP traffic. It can be found under Deliverables at the WS-I website.

Of the binding styles discussed above, only Document/literal and RPC/literal complies with WS-I. Document/literal is only compliant when the soap body has one child element. This is the case with Document/literal wrapped (described in Section4.1.1), which is therefore WS-I compliant.

4 Web Service Development

4.1 Development methods

In this section we will look at the inner workings of a Web Service, i.e. how the service works on the server side. In Exercise 2 below (4.2) we will learn how to create our own Web Service.

As you learnt from Section 3.2, a WSDL document is used to describe the interface of a Web Service for clients. A client does not need any knowledge about the Web Service besides what he can find in the WSDL. On the server side, the Web Service is responsible for:

1. de-serializing incoming SOAP request messages,
2. implementing a service,
3. serialising resulting objects into SOAP and
4. sending the result back to the client

A SOAP engine (see 3.1.1) is responsible for the low-level implementation of steps 1) and 3) above. Server code (called the server stub) interacts with the SOAP engine according to the functionality of the Web Service. The server stub is straight-forward and can be generated automatically. There has to be a complete correspondence between the WSDL file and the server stub. since the server stub is the implementation of the interface described in the WSDL file.

The code that implements the Web Server logic (step 2) is usually kept separate from the server stub. We will refer to this code as the *back-end* code from here on. The back-end code can contain anything from a full-scale software system with multiple existing user interfaces, to a few simple lines of code implementing something we want to be able to do through a Web Service interface. See figure.

The requirements listed above can to a large degree be trusted to a tool such as Apache Axis or another Web Service development platform. There are two basic strategies for generating a Web Service:

1. Generating the server stub and a WSDL document from existing back-end code.
2. Generating the server stub from a WSDL file and connecting it manually with the back-end code.

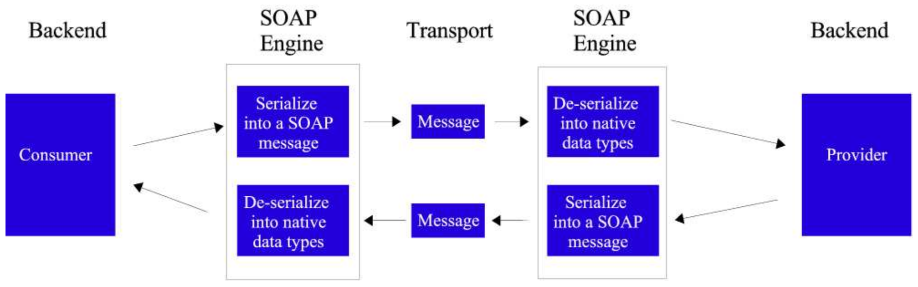

Backend
SOAP Engine
Transport
SOAP Engine
Backend
Consumer
Serialize into a SOAP message
De-serialize into native data types
Message
Message
De-serialize into native data types
Serialize into a SOAP message
Provider

Figure 4.1: Data flow between user and provider (client and server) of a Web Service

Method 1 is the faster strategy and requires little effort from the programmer. However, the resulting WSDL file and server stub is a result of the tool with which they were generated. This leaves little control over the details to the developer. In some cases, the resulting Web Service generated can only be understood by clients using the same tool for de-serialization.

To ensure that the WSDL file and service supports the rapidly evolving standards and best practices, including WS-I (see 3.2.2) , fine-grained control over the design is often necessary. With existing tools, developers are usually forced to operate on a lower level and design the WSDL file manually to ensure proper interoperability. Because of this, our recommendation is to use method 2 in most situations.

4.1.1 Binding styles:

In a WSDL document, a binding style describes the SOAP mapping of the service, i.e. how the data types received and sent by the service should be translated into a SOAP envelope. This is declared in the Binding of WSDL and implemented by the SOAP engine used.

A WSDL SOAP binding may be either Remote Procedure Call (RPC) style or Document style. Further, input and output documents may have either encoded or literal use. Further, there is a style called Document / Literal wrapped that follows some additional conventions for how the service and WSDL is built. All in all, this gives rise to five different styles, although only four are used in practice:

- RPC/encoded
- RPC/literal
- Document/literal
- Document/literal wrapped

The naming conventions of "RPC" vs. "Document" is perhaps unfortunate, as there is nothing wrong in having a Literal style Web Service with operations that behave as remote procedure calls. The term "RPC" simply refers to a convention for how to de-serialize objects to SOAP and nothing more. Our recommendation is to use the Document/literal Wrapped style in most cases. Only this style will be described here, but if you are interested in the other styles and how they differ, have a look at the very well-written guide "Which style of WSDL should I use?" at http://www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/.

The Document/literal wrapped style The Document/literal wrapped style is a special case of the Literal/wrapped style. This style prescribes that any data type declarations are described using an XML Schema. The SOAP messages sent to or from a Web Service following this style can thus be validated against this schema using an XML validator. This is an important and useful feature!

Further, each `message part` used for an input message must refer to an element declared in the data type declaration that has *the same name* as the `operation.` This is called a wrapper element. Thanks to this, the operation name appears in the incoming SOAP message.

If this sounds too abstract, have a look at the following example of a WSDL of a service with an operation called getDNASequence. This method accepts two positions and returns the DNA Sub-sequence from a particular bacterial genome between these two positions, encoded as a string.

```
<types>
  <schema>
   <element name="getDNASequence">
      <complexType><sequence>
  <element name="start" type="xsd:int"/>
  <element name="end" type="xsd:int"/>
       </sequence></complexType>
   </element>
   <element name="getDNASequenceResponse">
     <complexType><sequence>
       <element name="dnaSequence" type="xsd:int"/>
     </sequence></complexType>
   </element>
  </schema>
</types>

<message name="getDNASequenceRequest">
   <part name="parameters" element="getDNASequence"/>
</message>
```

```
<message name="getDNASequenceResponse">
   <part name="parameters" element="getDNASequenceResponse"/>
</message>

<portType name="PT">
  <operation name="getDNASequence">
    <input message="getDNASequenceRequest"/>
    <output message="getDNASequenceResponse"/>
  </operation>
</portType>

<binding ... />
```

A wrapped SOAP message for an incoming request to this operation would look like this:

```
<soap:envelope>
  <soap:body>
    <getDNASequence>
       <start>1</start>
       <end>2500</end>
     </getDNASequence>
  </soap:envelope>
</soap:body>
```

A Document/literal wrapped style Web Service has the following characteristics:

- The input message in the WSDL has a single part that refers to an element.
- This wrapper element has the same name as the operation.
- There are no attributes in the wrapper element.

This style has a number of advantages that, in our opinion, outweighs the disadvantage of the fairly complicated WSDL document:

- No type declarations for variables in SOAP messages (as opposed to the encoded style).
- The body of the SOAP messages is defined by a schema and can be validated against it.
- The method name appears in incoming SOAP messages.
- The style is WS-I compliant.

The ELM database Web Service, from the exercise in 3.2.1 above, uses the Document/literal wrapped style.

4.2 Exercise 2 - Creating a Web Service for the long-orfs program

We will follow an interface oriented development strategy (#2 in the list in above) in this course. This strategy requires the following steps:

1. Express data types in an XML schema.
2. Define the interface in WSDL.
3. Generate server code from the WSDL document.
4. Connect the back-end code with the generated server code
5. Deploy the Web Service

All of the above steps will be performed using the Eclipse Web Tools Platform (WTP).

- Steps 1 and 2 involves using the XML editor in Eclipse WTP.
- Step 3 is performed using Ant and a class library from Axis.
- Step 4 is done in the Eclipse Java editor.
- Step 5 is performed using a Tomcat plug-in and Ant.

4.2.1 About the long-orfs program

The long-orfs program is a command line based tool for identifying long Open Reading Frames (ORFs) in a genomic DNA sequence. It is part of the gene prediction system Glimmer3 and has a number of options such as resolving conflicts between overlapping ORFs etc, that we do not need to concern ourselves with in this tutorial. For more information, visit cbcb.umd.edu/software/glimmer/.

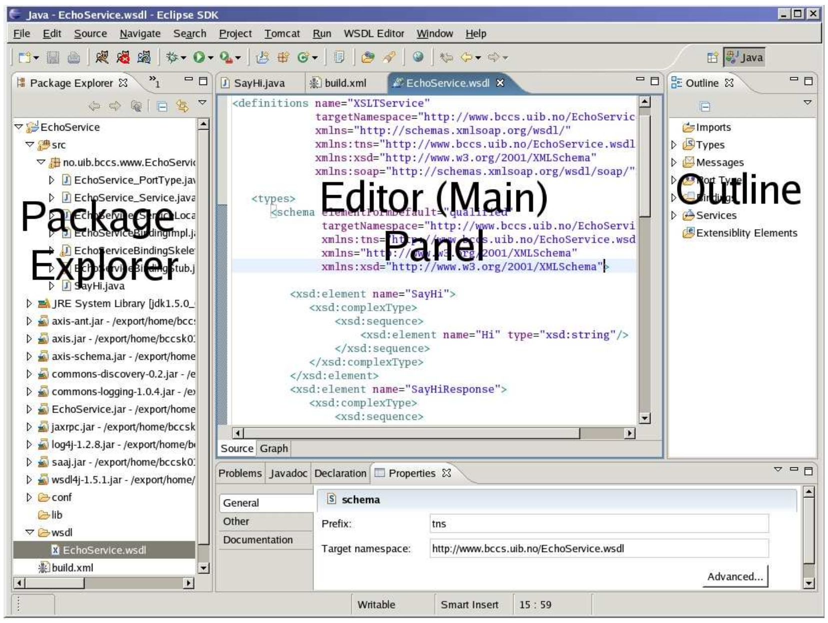
The output from long-orfs is normally sent to the standard output buffer and is in the form of a tab-separated text table. In this tutorial, we will use a Java wrapper for long-orfs. This wrapper makes a system call to the long-orfs program and parses its output into Java objects. It constitutes our *back-end code.*

4.2.2 About Eclipse WTP

Eclipse is an open source community whose projects are focused on providing a vendor-neutral open development platform and application frameworks for building software. The Eclipse Web Tools Platform (Eclipse WTP) project extends the Eclipse platform with tools for developing J2EE Web applications. WTP adds a lot of functionality, but we will use only the XML editor and WS-I validator. For more information, visit www.eclipse.org/webtools/.

NOTE: You are free to use any editors you would like during the course, but we recommend the use of WTP Eclipse because it removes a lot of irrelevant complexity in the development environment.

The main parts of the Java Perspective in the Eclipse Workbench are shown in Figure 4.2.



```
<definitions name="XSLTService"
             targetNamespace="http://www.bccs.uib.no/EchoServic
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:tns="http://www.bccs.uib.no/EchoService.wsdl
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

  <types>
    <schema elementFormDefault="qualified"
             targetNamespace="http://www.bccs.uib.no/EchoServi
             xmlns:tns="http://www.bccs.uib.no/EchoService.wsd
             xmlns="http://www.w3.org/2001/XMLSchema"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">

      <xsd:element name="SayHi">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="Hi" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="SayHiResponse">
        <xsd:complexType>
          <xsd:sequence>
```

schema
Prefix: tns
Target namespace: http://www.bccs.uib.no/EchoService.wsdl

Figure 4.2: The main parts of the Java Perspective in the Eclipse Workbench (default)

4.2.3 Explore the EchoService example

Open Eclipse WTP by clicking on the Eclipse icon on the desktop.

If you are new to Eclipse, start the Introductory tutorial by selecting Help > Welcome and then clicking on Tutorials and then Java Development. This will only take 10 minutes of your time and will teach you how to create and configure projects and use the Java Editor.

In the Package Explorer tab, expand the EchoService project. This is an example of a complete Web Service. Use the same conventions in naming of directories, e.g. using a directory named "WSDL" for your WSDL file in the project you will develop for the long-orfs program.

Explore the contents of the source directory ("src"). In this simple example, `SayHi.java` constitutes the back-end code. All other classes are part of the server stub and has been auto-generated from the WSDL document. `EchoServiceBindingImpl.java` has then been manually modified to call the back-end code in a meaningful way.

Double click on build.xml to open it. This is an Ant Tasks file configured to perform standard tasks such as generating the server stub and deploying the complete Web Service. Note how the available tasks are shown in the OUTLINE tab to the right in the workbench.

The EchoService is already deployed as a Web Service on your local computer and you can have a look at it using the tool SOAPUI.

To launch SOAPUI, use the icon on the desktop. First of all you need to create a new project. This is done using FILE > NEW WSDL PROJECT. Name your project for example EchoService and accept to create the suggested project file. Then, you need to add the external WSDL file. Do this by right-clicking on the project in the list of projects (in the top left panel) and select ADD WSDL FROM URL. When prompted for the address, type "http://localhost:8080/axis/services/EchoService?wsdl". Select yes to the question about creating default requests.

SOAPUI has now created a SOAP envelope template for you in order to send requests to the Web Service. Double-click this request template ("Request 1" under operation `SayHi` in your service) to open it in a new frame. You can now fill in the contents of the element Hi where a question mark has been put. Click the green "play" button in the top left corner of the request frame! You can then see the response SOAP envelope that is sent back to you. The contents of its `HiResponse` Element is the same as you sent it (hence, "EchoService").

Notice how the elements in the request and response elements correspond to the declarations in the WSDL!

4.2.4 Create your Web Service Interface

Now, have a look at the project "LongORFs", that is already available in the PACKAGE EXPLORER. This is a half-finished project for creating the Web Service for long-orfs that we will finish. You can use the EchoService wsdl file as an example. A number of tasks not really related to Web Service development have already been performed. We don't need to worry about these in this tutorial, but for the interested reader, here they are:

- A new project was created
- An Ant tasks file (build.xml) was imported. (In fact, this is an identical copy of the file in "EchoService")
- The configuration directory "conf" was created and a build.properties file with some settings for Ant created.

- A number of jars (class libraries) were made available to the project's build path. This is done by selecting PROJECT > PROPERTIES > JAVA BUILD PATH and then selecting ADD JARS in the LIBRARIES tab.
- The back-end code for accessing the long-orfs program was imported to the source directory ("src"). This code is contained in the package longorfs.
- Created a skeleton WSDL file called `LongORFs.wsdl` inside the wsdl folder.

Now, to complete the Web Service:

- Open `LongORFs.wsdl` in Eclipse WTP's WSDL Editor by double-clicking on it.
- The WSDL Editor enables two views to a WSDL file: source or graph. You can switch between these by clicking on the tabs right underneath the editor window (see Fig. 3.1.1). We recommend always using the source view when writing the WSDL from scratch.
- Start writing the WSDL by creating an XML Schema to define the complex data types you want to return from the Web Service. An ORF prediction data type definition (for the output) and a Genome sequence (for the input) should be enough in this case.
- Continue by defining the single element data types to be sent as input and output from the operations of the Web Service. In this case it is probably enough with one operation, e.g. `getORFs`. Use the Document/literal wrapped style!
- Create your Message Elements. (e.g. `getORFsRequest` and `getORFsResponse`) and connect them to your wrapper data types.
- Create your Port Type Element and its Operation(s). Connect its Input and Output Elements to your message elements
- Create your Binding Element. Specify your binding style (using a `soap:binding` element). Add the Operation(s) from your Port Type. These also need `input` and `output` elements, but there is a subtle difference from how they are specified here compared to in the abstract Binding: They do not have attributes and they need child elements specifying their content in terms of SOAP encoding. In this case, it is enough with a SOAP body, using literal encoding:

  ```
  <soap:body encodingStyle=
  "http://schemas.xmlsoap.org/soap/encoding/" use="literal" />
  ```

- Create your concrete endpoint, i.e. the Service Element and its Port. Put `http://localhost:8080/axis/LongORFs` as your network address.

Hints:

- Instances of the class `ORF` in the `longorfs` package represents a predicted ORF. Review this class when designing your data types. The format of an ORF is quite simple. It has a start and a stop position represented by integers, a number, a strand (+ or -) and a score.
- Use the Document/literal wrapped style. Remember the rules from 4.1.1 above! Each operation you want your Web Service to perform must have a wrapper data type definition for its input using the same name as the operation. For the output it must have a data type with the same name as the operation plus "Response".
- Have a look at the WSDL from EchoService or the ELMdb WSDL from previous exercises and imitate the style.

4.2.5 Validate your WSDL

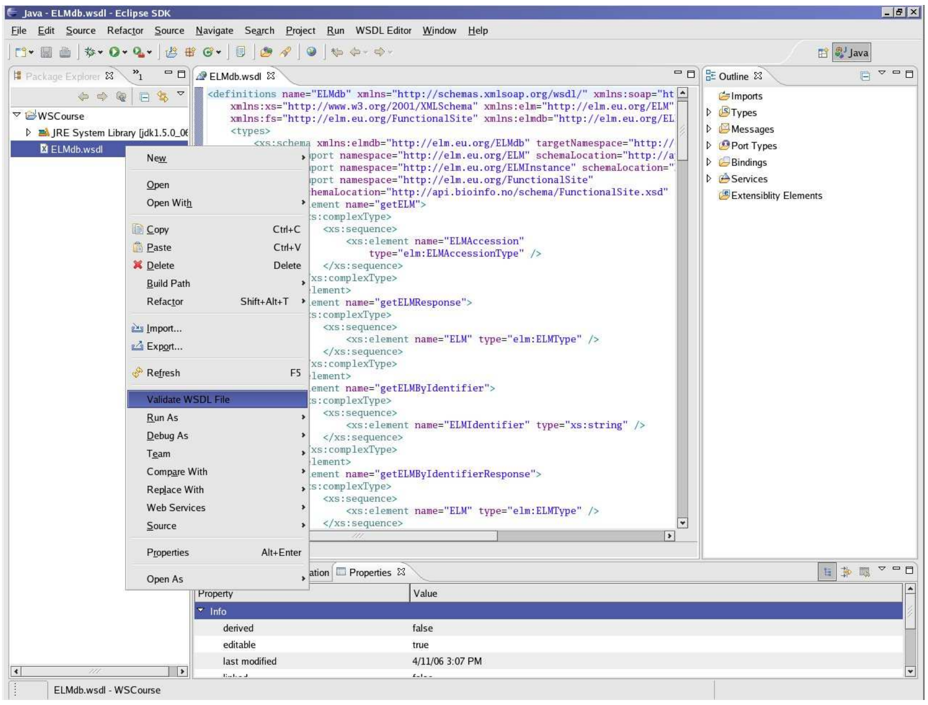
Save your WSDL file and validate it using Eclipse WTP's validation tool. This is done by right-clicking the file and selecting VALIDATE WSDL FILE (see Fig. 4.3 below). Error indications will appear to the left of problematic elements in the editor panel. Read any error messages you get by hovering with the cursor over these. Try to fix your errors one and remember to always save your file and re-validate it each time you have corrected anything! The error indications may not disappear until you do this, even though the problem is fixed.

The WSDL validation ensures that the WSDL fulfills the WSDL standard and the WS-I Basic Profile. The WS-I compliance level can be modified in Eclipse WTP's preferences, but since we want to use the Document/literal wrapped style and to comply with the Basic Profile, we will go with the default settings.

4.2.6 Generate server code from your WSDL

Now that we have a validated WSDL document describing the interface to your Web Service, it is time to let the tool do some work for us by automatically generating your server code from the specifications in the WSDL. The tool we will use for this is Axis. An Ant task that carries out this operation is provided:

- Run the `generateServerStub` task by clicking on the ant icon on the icon bar and select `generateServerStub` (see Fig. 4.4 above). If everything goes right, you should see the output from Ant in the Console Panel that concludes with "BUILD SUCCESSFUL".
- Axis's WSDL2Java tool has now generated a number of new classes in your source and configuration directories. You should now see a new package named `no.uib.bccs.www.LongORFs_wsdl`. This is your auto-generated server stub!



```
<definitions name="ELMdb" xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:soap="ht
    xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:elm="http://elm.eu.org/ELM"
    xmlns:fs="http://elm.eu.org/FunctionalSite" xmlns:elmdb="http://elm.eu.org/EL
    <types>
        <xs:schema xmlns:elmdb="http://elm.eu.org/ELMdb" targetNamespace="http://
port namespace="http://elm.eu.org/ELM" schemaLocation="http://a
port namespace="http://elm.eu.org/ELMInstance" schemaLocation="
port namespace="http://elm.eu.org/FunctionalSite"
hemaLocation="http://api.bioinfo.no/schema/FunctionalSite.xsd"
ement name="getELM">
s:complexType>
   <xs:sequence>
       <xs:element name="ELMAccession"
           type="elm:ELMAccessionType" />
   </xs:sequence>
xs:complexType>
lement>
ement name="getELMResponse">
s:complexType>
   <xs:sequence>
       <xs:element name="ELM" type="elm:ELMType" />
   </xs:sequence>
xs:complexType>
lement>
ement name="getELMByIdentifier">
s:complexType>
   <xs:sequence>
       <xs:element name="ELMIdentifier" type="xs:string" />
   </xs:sequence>
xs:complexType>
lement>
ement name="getELMByIdentifierResponse">
s:complexType>
   <xs:sequence>
       <xs:element name="ELM" type="elm:ELMType" />
   </xs:sequence>
```

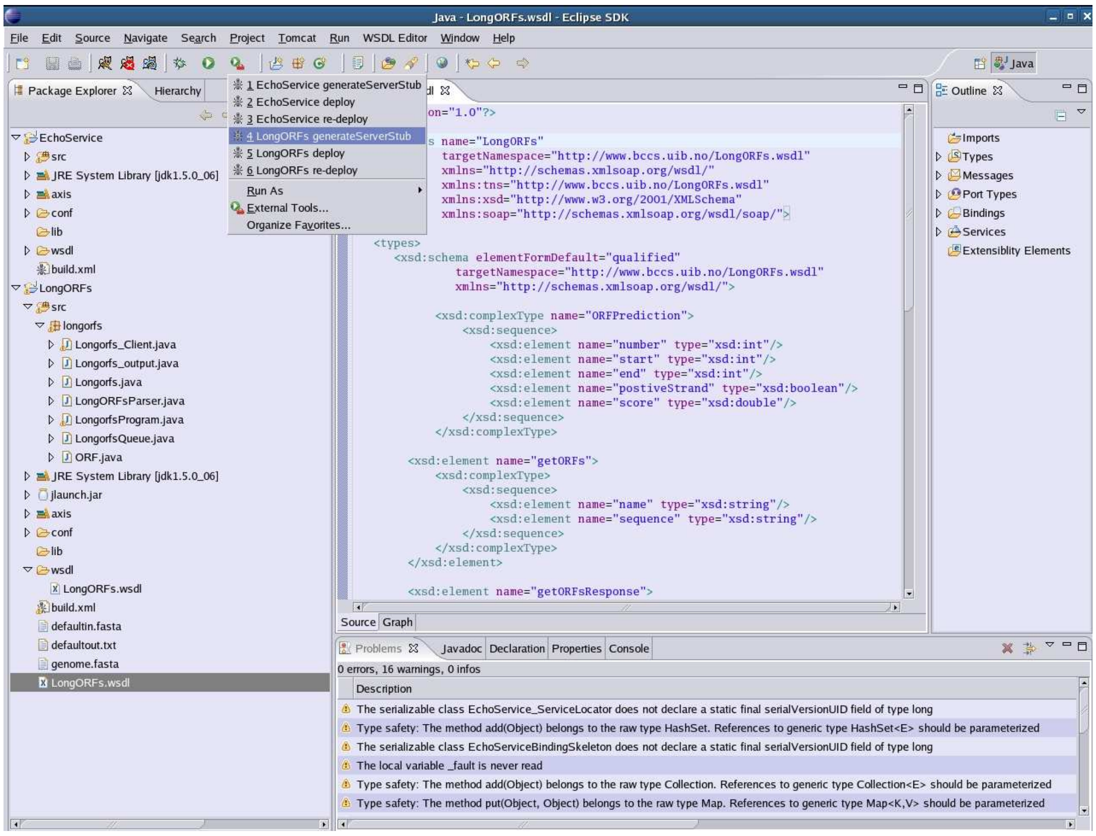
Property	Value
Info	
derived	false
editable	true
last modified	4/11/06 3:07 PM

Figure 4.3: Re-validate your WSDL each time you have corrected an error!

- Besides the server stub, Axis has generated two Axis WSDD files; `deploy.wsdd` and `undeploy.wsdd`. These have been moved to the “conf“ directory by ant. They are configuration files that are needed by Axis to correctly deploy and undeploy your service. More about this later. For now, we will only assure once more that our WSDL follows the Document/literal style. Open the file `deploy.wsdd` and look at the attributes of the `<service>` element in it. It should have the attributes `style="wrapped" use="literal"`.If you get something else, go back to your WSDL and try to find out what does not comply with the Document/literal Wrapped style (see 4.1.1). Then generate the server stub again, until the right attributes appear in `deploy.wsdd`.

4.2.7 Connect back-end code with generated code

- Open the class `LongORFsBindingImpl.java`. If you named your operation “getORFs” the method in this class that answers to that Web Service operation will also be called getORFs. Note how it accepts a genome sequence object and returns an ORF Prediction object, or array of ORF Predictions, depending on how you defined your types. Java classes representing these data types have been auto generated for you. Have a look at them! (A convenient way to navigate in Eclipse is to hold in ctrl



```
on="1.0"?>

s name="LongORFs"
    targetNamespace="http://www.bccs.uib.no/LongORFs.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.bccs.uib.no/LongORFs.wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

  <types>
     <xsd:schema elementFormDefault="qualified"
            targetNamespace="http://www.bccs.uib.no/LongORFs.wsdl"
            xmlns="http://schemas.xmlsoap.org/wsdl/">

         <xsd:complexType name="ORFPrediction">
            <xsd:sequence>
               <xsd:element name="number" type="xsd:int"/>
               <xsd:element name="start" type="xsd:int"/>
               <xsd:element name="end" type="xsd:int"/>
               <xsd:element name="postiveStrand" type="xsd:boolean"/>
               <xsd:element name="score" type="xsd:double"/>
            </xsd:sequence>
         </xsd:complexType>

      <xsd:element name="getORFs">
         <xsd:complexType>
            <xsd:sequence>
               <xsd:element name="name" type="xsd:string"/>
               <xsd:element name="sequence" type="xsd:string"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>

      <xsd:element name="getORFsResponse">
```

```
0 errors, 16 warnings, 0 infos
The serializable class EchoService_ServiceLocator does not declare a static final serialVersionUID field of type long
Type safety: The method add(Object) belongs to the raw type HashSet. References to generic type HashSet<E> should be parameterized
The serializable class EchoServiceBindingSkeleton does not declare a static final serialVersionUID field of type long
The local variable _fault is never read
Type safety: The method add(Object) belongs to the raw type Collection. References to generic type Collection<E> should be parameterized
Type safety: The method put(Object, Object) belongs to the raw type Map. References to generic type Map<K,V> should be parameterized
```

Figure 4.4: Running the `generateServerStub` task using Ant

and click on a class or method to jump to it.)

- Now it's time to connect the server code with the back-end code. This is simply done by modifying the getORFs method (or whatever name you chose for the operation in the WSDL) in `LongORFsBindingImpl`. The only back-end class you need to be concerned with is the `LongORFsParser`. The long-orfs program is called by creating a new instance of `LongORFsParser` and then call its run() method. This will return a `Longorfs_output` instance from which you can get an array of `ORF` objects. These objects are what you should return from your service, but you can not simply return these objects. The reason for this is that the parser has its own class definition of ORFs, and the auto-generated code has a different class definition. You therefore need to translate the list of ORFs from one type to the other. Hint: You can iterate over the list and run orf1.setxxx(orf2.getxxx()) on each variable.

Hints:

- To better understand how to use the `LongORFsParser`, open the `Longorfs_Client`. This also contains a suitable sequence string that you can use for testing.

4.2.8 Deploying your Web Service

Your Web Service is now ready deploy! For this we use the SOAP engine Axis. To use Axis for this, Axis itself needs to be deployed as a web application (servlet) in a servlet container. In the system we will use in this course, Axis is deployed to the servlet container Tomcat. Tomcat is running on port 8080.

- Open localhost:8080/axis in a browser. This is the web-based user interface to the part of Axis that takes care of deployment of Web Services. Click "List". Note how EchoService is already deployed. By clicking it, you can see its WSDL.
- Now go back to Eclipse WTP. Look at the task "deploy" in the Ant tasks file (build.xml). Try to figure out how it works. Then run the task. Your Web Service is now deployed (almost)!
- To properly deploy the Web Service you need to restart tomcat. Conveniently, this can be done from Eclipse WTP simply by clicking the `Restart Tomcat` icon in the tool bar (or `Tomcat > Restart` in the menu).
- In your browser, reload the page listing your deployed Web Services. If nothing has gone completely wrong, your Web Service LongORFs should now appear in the list. That's it. You have now created a Web Service!

4.2.9 Try out your Web Service using soapUI

Use soapUI to try out your Web Service by sending it a genome sequence. This is done in the same manner as in Section 4.2.3. Create a new SOAPUI project for your Web Service. In your home directory, you will find a file named `genome.fasta` as an example of a genome sequence. Copy and paste a sequence from this into the SOAPUI request.

4.3 Exercise 3 - Creating a client using a simple scripting language (Python)

SOAPpy is a Python module for constructing and using Web Services. For more information on SOAPpy, see http://pywebsvcs.sourceforge.net/. In this exercise, you will use the Python Programming language and SOAPpy to create a simple client for the longORFs Web Service from Exercise 2. If you have never used Python before, do not panic. It's quite easy:

First of all, use your favourite text editor to create a new python script. First of all, we need to import a couple of classes from the SOAPpy library. This is done by adding the following lines to your script:

```
from SOAPpy import SOAPProxy
from SOAPpy import Types
```

An instance of SOAPProxy acts as a proxy to the Web Service endpoint of its URL variable. When constructed, it will automatically have the same methods as the Web Service it acts as a proxy for.

To create a SOAPProxy is very easy:

```
url = 'http://www.webservices.com/the-service/endpoint'
namespace = 'my_namespace'
longORFService = SOAPProxy(url, namespace)
```

The response from the Web Service will automatically be serialised into a Python object when such a request is carried out (this object is returned by the method invoked). Print out some information about the response object on the screen.

To print something to standard out in Python, simply use:

```
print 'the text you want to print'.
```

To run the Python script from the command line use:

```
python my_script.py
```

To make a reference to a variable, the same convention as in Java applies, i.e. the variable 'variable' in the data type 'dataType' can be reached by 'dataType.variable'.

The variables will have the same name and type as those in the complex type element returned. As can be seen from the data type declaration of the WSDL of your longORFs service.

Index