An Introduction to Development with TextGrid

Christoph Ludwig   Thorsten Vitt

August 3, 2009
SVN Revision 4008

Contents

1 Introduction

TextGrid[1] provides a workbench for scholars in text-oriented disciplines. It supports collaboration and resource sharing by leveraging Grid and Web service technologies at the infrastructure and service level.

However, such a workbench can never be complete; there will always be special tools and features that may be essential for a particular research project but don't exist within TextGrid yet. It was therefore one of the architectural design objectives of TextGrid that third parties can easily extend the workbench.

This tutorial gives developers a short introduction how they can extend TextGrid with their own tools. Sect. 2 explains the most important aspects when developing a Web service that accesses the TextGrid middleware. Similarly, Sect. 3 shows how one can develop additional features and views for the Eclipse-based user interface TextGridLab.

Please note, though, that we presume the reader has some experience with Web service development and / or Eclipse plug-in development. We aim to provide a step-by-step explanation of what we do, but for any background material the reader has to consult specialized tutorials or books covering the relevant topics.

It is not practical to reproduce all source code in full length in a tutorial. But the complete code is available online in TextGrid's subversion repository. The folders underneath https://develop.sub.uni-goettingen.de/repos/textgrid/tags/DevelopersWorkshop_2009-01-22/ hold snapshots of the service and plug-ins developed in the following so you can easily follow the tutorial.

For a more comprehensive and reference-oriented documentation of TextGrid's APIs see [Tex09]. If you have more questions, contact us at info@textgrid.de.

2 Service Development

This section's objective is to develop an admittedly very simple SOAP based web service that accesses the TextGrid infrastructure. The service operation will be called with the URI of a TextGrid object that is presumed to contain a TEI-encoded document. The service reads this object and returns all names therein that are marked by the TEI tag `<tei:persName>`.

We will implement the service in Java and make use of Axis2. However, if you are familiar with any other web service framework, no matter which programming language, then this tutorial should provide you with sufficient information to implement this service in the framework of your choice as well.

Please note that we don't intend to provide an Axis2 tutorial – there are many books and online resources you can consult. If you don't have much experience with SOAP based web services and you want to get up and running quickly so you can follow this tutorial in Eclipse, then [TKI08] provides the necessary information (but admittedly not

[1]TextGrid (http://www.textgrid.de/) was partially funded by the German Federal Ministry of Education and Research (BMBF) under the D-Grid initiative by agreement 07TG01A-H; the responsibility for this publication rests with its authors.

much more than that). [Jay08] gives a good overview over Axis2's architecture, over issues like the lifetime of service objects, and over service modules. [FTW07] is much more comprehensive than the aforementioned books. However, it's emphasis is more on the use of Axiom and the Axiom Data Binding (ADB) that ships with Axis2. Of course, there is also plenty of information on Axis2 on the Web; besides the Axis2 homepage [Axi], the WSO2 portal [WSO] should be one of your first ports of call for information on Axis2.

2.1 Development Environment

As usual, many roads lead to Rome - that's particular true when it comes to the configuration of your development environment. However, if you intend to use the project snapshots we provide in TextGrid's Subversion repository, then we strongly recommend that you follow the steps below very closely since, e. g., library and Eclipse path variable names appear verbatim in the project configuration files in the SVN repository.

2.1.1 Axis2 Setup

You need to download the most recent release of Apache Axis2[2] from [Axi] and unpack it in a directory we will refer to in the following as `$AXIS2_HOME`.

Axis2 ships with a standalone Web server. It is not intended for production use, but it is very convenient for testing your services locally on your development machine. We will therefore restrict ourselves to running our services in this Web server; we don't cover the necessary configuration changes if you run Axis2 inside an application server like, say, Tomcat.

We presume you enabled Axis2's hot update mechanism and you made the Axis2 Web server listens on TCP port 8180. For this you have to make sure that the relevant elements in `$AXIS2_HOME/conf/axis.xml` have the values shown in Listing 1.

Nobody is perfect - it is therefore unlikely that your service code is bug free the very first time you deploy it. Fortunately, you can attach a Java debugger to your Axis2 process and observe what's happening in your service using the comfortable Eclipse environment. However, you must instruct the Java VM when it starts Axis2 to open a port for the debugger where it can connect. The necessary options are quite unwieldy whence we recommend you create a separate start script `$AXIS2_HOME/bin/debugServer.sh` with the content shown in Listing 2. This start script is merely an extension of the script provided by Apache in `$AXIS2_HOME/bin/axis2server.sh`. The Windows Batch file `$AXIS2_HOME/bin/axis2server.bat` can be extended in a similar way.

2.1.2 Eclipse Setup

In this tutorial we assume you use Eclipse 3.4 (Ganymede) as your development environment. Furthermore, we presume that you have recent versions of Subclipse[3], the XML

[2]We tested our code with Axis2 1.4.2.
[3]http://subclipse.tigris.org/

Listing 1: Axis2 configuration assumed throughout this tutorial

```
<axisconfig name="AxisJava2.0">
    <!-- ================================ -->
    <!-- Parameters -->
    <!-- ================================ -->
    <parameter name="hotdeployment">true</parameter>
    <parameter name="hotupdate">true</parameter>

    <!-- ... many more parameters and options ... -->

    <!-- ================================ -->
    <!-- Transport Ins -->
    <!-- ================================ -->
    <transportReceiver name="http"
         class="org.apache.axis2.transport.http.SimpleHTTPServer">
        <parameter name="port">8180</parameter>
    </transportReceiver>
</axisconfig>
```

Listing 2: Start Script for the Axis2 Web Server in Debug Mode

```
#!/bin/sh

# Get the context and from that find the location of setenv.sh
. `dirname $0`/setenv.sh

JAVA_OPTS='-Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n
    '
export JAVA_OPTS

echo "␣Using␣JAVA_OPTS:␣$JAVA_OPTS"

java -classpath "$AXIS2_CLASSPATH" $JAVA_OPTS \
  org.apache.axis2.transport.SimpleAxis2Server \
  -repo "$AXIS2_HOME"/repository \
  -conf "$AXIS2_HOME"/conf/axis2.xml $*
```

Schema Definition Model[4], and the Web services tools[5] installed.

Axis2 User Library. Axis2 ships with a large number of jar-files; most of them need to be in the classpath when you develop a new service. Since it is very cumbersome and error-prone to add them to every service project you start in Eclipse, you should collect them in an Eclipse user library. For this go to the "Preferences" dialog (available in the menu bar under "Window" or "Eclipse", depending on your operating system) in the screen "Java" / "Build Path" / "User Libraries". Click the "New" button and enter the name *Axis2*. (If you follow this naming convention, then the services you check out from TextGrid's Subversion repository will work out of the box as well.) After "OK", click "Add JARs" and select all jar-files in Axis2's lib folder.

Axis2 Service Folder. Before we set up an Eclipse project for our service, you need to create a folder for it in the Axis2 repository's service folder. Unless you moved the repository, you have to create `$AXIS2_HOME/repository/services/NameService` using the command line or a file system explorer of your choice.

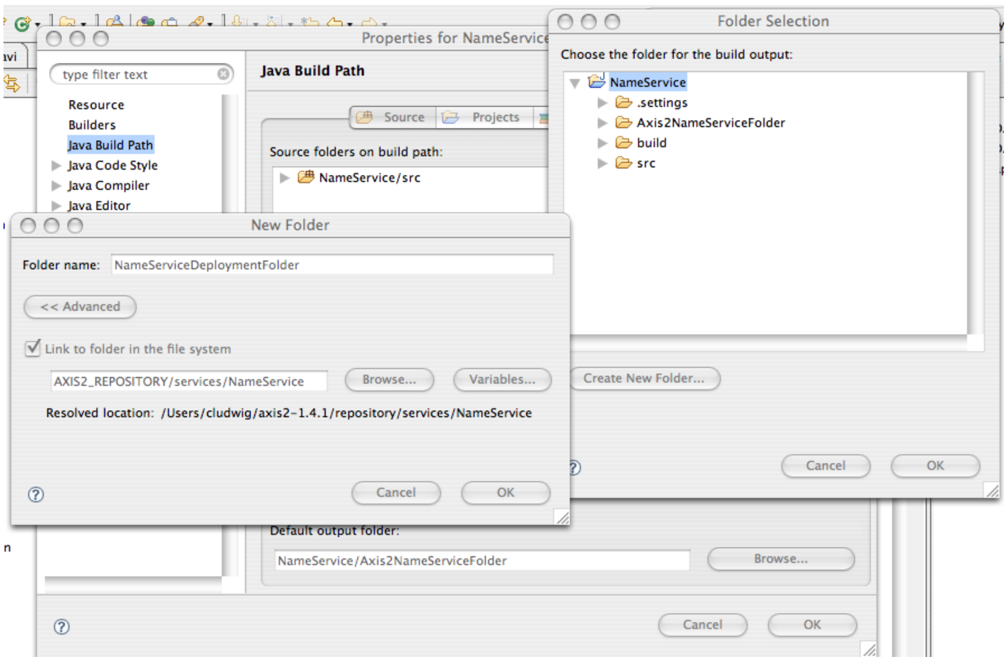
Eclipse Project. Back in Eclipse, select "File" / "New" / "Java Project", enter the project name *NameService*, and click "Next". In the following dialog we make two adjustments: We instruct Eclipse to write the service's class files directly into Axis2's service repository and we add the Axis2 user library to the project's build path.

On the tab named "Source" click on "Browse" next to the input field for the default output folder. Select the top level project folder and then click on the button that lets you create a new output folder. In the dialog that follows, click on "Advanced" and check the box "Link to folder in the file system". Now you can click on "Variables" and select (or, if necessary, create) the Eclipse variable "AXIS2_REPOSITORY". This variable has to point to the repository folder in your Axis2 installation, typically found in `$AXIS2_HOME/repository`. After that, you append `/services/NameService` to the folder location and change the (Eclipse-internal) folder name to *NameServiceDeploymentFolder*, cf. Fig. 1.

Finally, you need to add the previously created user library *Axis2* to the project's build path. On the tab "Libraries" click on "Add Library", select "User Library", and then check the button next to *Axis2* before closing the dialog. Once you pressed the wizard's "Finish" button, the new project appears in your workspace.

Axis2 Debug Configuration. In the previous subsection you created a script that starts the Axis2 Web server in debug mode. If you want to connect to the server with the Java debugger in Eclipse, then you need to create a suitable Eclipse debug configuration first. Open the dialog "Run" / "Debug Configurations..." and select "Remote Java Application" in the list on the left hand side. After a click on the *New* button in the top, fill in the configuration data as seen in Figure 2.

[4]http://www.eclipse.org/modeling/mdt/?project=xsd#xsd
[5]http://www.eclipse.org/webtools/ws/


Properties for NameService

Java Build Path

Source folders on build path:

NameService/src

Default output folder:

NameService/Axis2NameServiceFolder

Folder Selection

Choose the folder for the build output:

NameService
- .settings
- Axis2NameServiceFolder
- build
- src

New Folder

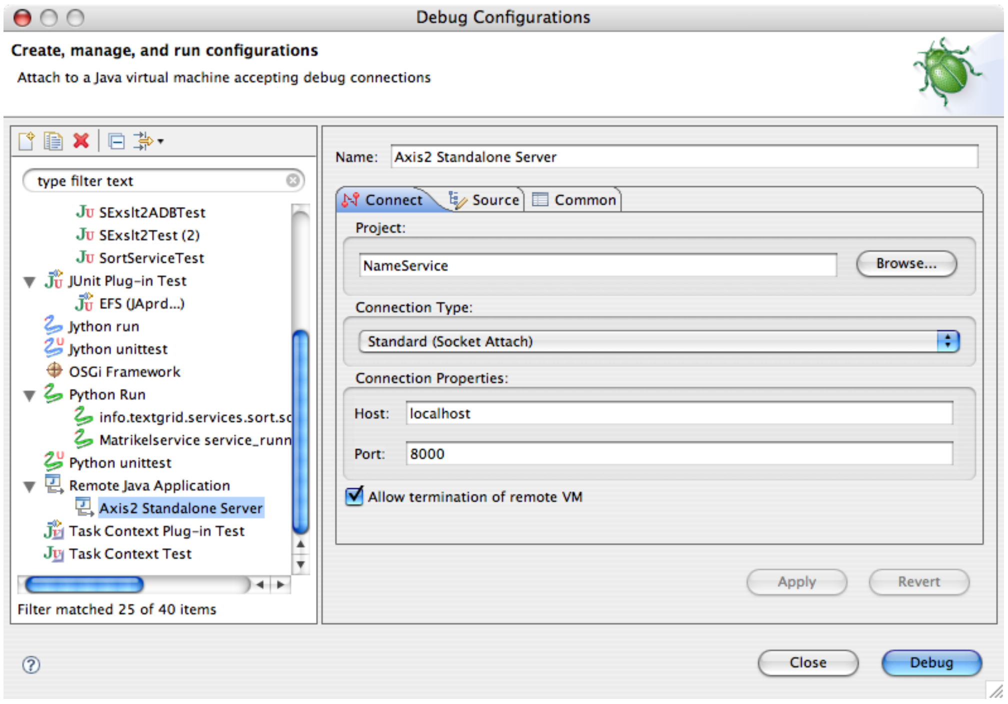
Folder name: NameServiceDeploymentFolder

Link to folder in the file system

AXIS2_REPOSITORY/services/NameService

Resolved location: /Users/cludwig/axis2-1.4.1/repository/services/NameService

Figure 1: Service Project Output Folder Configuration


Debug Configurations

Create, manage, and run configurations

Attach to a Java virtual machine accepting debug connections

Name: Axis2 Standalone Server

Project:

NameService

Connection Type:

Standard (Socket Attach)

Connection Properties:

Host: localhost

Port: 8000

Allow termination of remote VM

- SExslt2ADBTest
- SExslt2Test (2)
- SortServiceTest
- JUnit Plug-in Test
  - EFS (JAprd...)
- Jython run
- Jython unittest
- OSGi Framework
- Python Run
  - info.textgrid.services.sort.so
  - Matrikelservice service_runn
- Python unittest
- Remote Java Application
  - Axis2 Standalone Server
- Task Context Plug-in Test
- Task Context Test

Filter matched 25 of 40 items

Figure 2: The Configuration for Debugging Axis2 Web Services

2.2 WSDL

Web service clients obviously need to know where they can find the service, which operations it provides, what parameters the operations expect and how to transfer these parameters and potential return values. This formal interface description is typically bundled together in an XML document according to the WSDL specification by the W3C. WSDL 1.1 [CCMW01] is still in wide use while WSDL 2.0 [BL07, CMRW07] is gaining more and more support.

There are two approaches to Web service development: Code First and Contract First. With the Code First approach you don't need to worry about the service's public interface in the beginning. You develop the service's core in your favourite programming language and, once it's finished, you run a tool that puts the service binding on top and generates the corresponding WSDL. However, we found that these tools often carry programming language artifacts over into the service description in the way they, say, map array types to XML Schema types.

In a heterogeneous environment like TextGrid it is therefore much easier to avoid non-interoperable constructs if you follow the Contract First approach: Here you first define the service interface and the XML Schema types used to transport parameters and results. Then you use a tool that generates from the WSDL a service skeleton into which you then fill in the service implementation.

As mentioned above, our service *NameService* will offer just one operation *extractNames*. This operation requires the URI of a TextGrid object (XML Schema type anyURI) and returns a list of all names (as strings) it finds in this object. Per convention, all TextGrid services also take two optional parameters: A session ID required for access to non-public TextGrid objects and a logging parameter. Both parameters are encoded as strings. We will see in the following subsections how these parameters are used.

Of course, we must also prepare for error conditions. Our service knows three types of faults (more or less the Web service equivalent of Java exceptions): One that signals an error while retrieving the TextGrid object via the TextGrid middleware, one for errors while accessing TextGrid's log mechanism, and one that is returned if the TextGrid object does not contain well formed XML or there is any other problem with the XPath evaluation.

Fortunately, Eclipse offers a WSDL editor that spares you from most of the syntactical details of WSDL and XML Schema. Under "File" / "New" / "Other" select "WSDL" from the folder "Web Services", select the project *NameService* and enter the file name *NameService.wsdl*. On the next screen, enter http://textgrid.info/namespaces/examples/NameService as target namespace. Keep all other options' default values, i. e., we want Eclipse to generate a WSDL skeleton for a SOAP based service with document/literal binding. The skeleton is created and opened in the WSDL editor as soon as you click "Finish".

The context menu of the port *NameServiceSOAP* in the left hand side of the diagram lets you open the properties view. There you change the service's endpoint to http://localhost:8180/axis2/services/NameService.

After a click on the only operation (still named *NewOperation*) of the service shown on

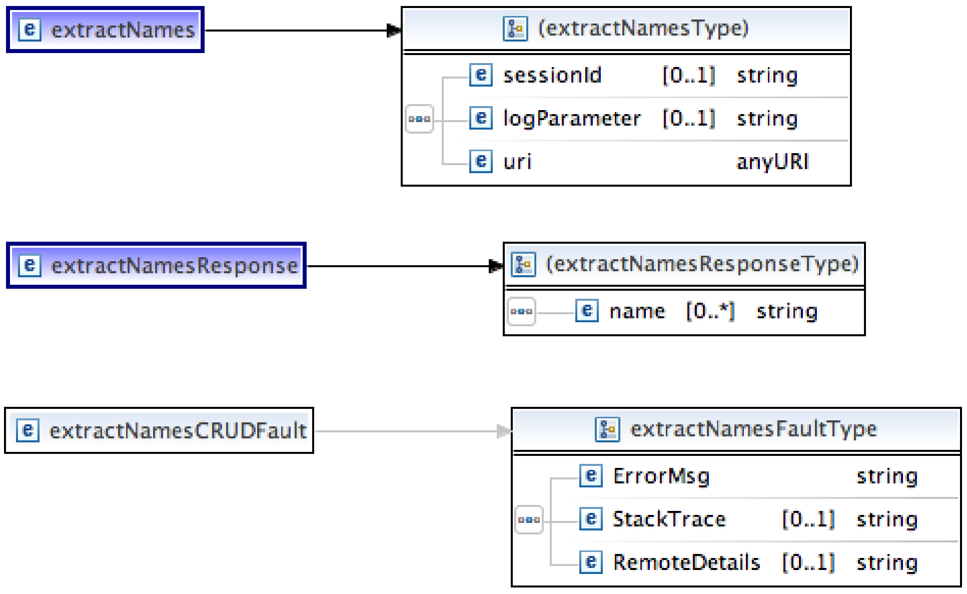


Figure 3: XML Schema Types for the extractNames Operation Messages

the right hand side you can change its name in the properties view into *extractNames*. Add faults *CRUDFault*, *LogFault*, and *DataFault* to the operation using its context menu. In the faults' property views change the message names to *extractNamesCRUDFaultMsg*, *extractNamesLogFaultMsg*, and *extractNamesDataFaultMsg*, respectively.

Now we are ready to specify the bodies of the SOAP messages that are exchanged between our service and its clients. A click on the arrow to the right of the input message's element opens an inline XML Schema editor. You can see that the XML element ⟨*extractNames*⟩ is of an anonymous complex type (signaled by the brackets around *extractNamesType*) that currently consists of a sequence (the icon with three dots in it) with exactly one element ⟨*in*⟩. Using the sequence's context menu you need to add two further elements and then to change their names, types, and cardinalities in the properties view. The result should be as shown in Fig. 3. The definition of the anonymous type for ⟨*extractNamesResponse*⟩ is similar except that you need to enter "unbounded" as maximum occurrence of the sequence's only element ⟨*name*⟩.

Finally, we change the element names of fault messages into ⟨*extractNamesLogFault*⟩, ⟨*extractNamesCRUDFault*⟩, and ⟨*extractNamesDataFault*⟩, respectively, and assign all these elements a new named XML Schema Type *extractNamesFaultType*. To do so we select in the properties view of one Fault element "New" from the type's drop down list and set the new type up as indicated in Fig. 3. We can then assign this type to the remaining fault elements by choosing "Browse" in the drop down list and selecting *extractNamesFaultType*.

After this exercise you should have a WSDL file that can be used as basis for the following steps. If you are unsure, then you can also import our demo project from TextGrid's subversion repository: Select the *NameService* project in the Eclipse's Package Explorer and call "Import" from its context menu. Select "SVN" / "Checkout Project from SVN", open the repository https://develop.sub.uni-goettingen.de/repos/textgrid in the following screen, and select then `tags/DevelopersWorkshop_2009-01-22/NameService_Step1`. After some further confirmation dialogs, your project will be replaced by our demo project.

2.3 The Service Skeleton

The WSDL we created in the previous subsection is purely declarative. It is about time that we generate code that actually realizes the service. Axis2 and / or the application server it is embedded in handle the incoming HTTP requests and do all the necessary pre- and postprocessing of the SOAP messages exchanged. However, if we want to access the SOAP body we still see only raw XML (or, more precisely, an Axiom representation of the XML). It sometimes is indeed useful to handle the SOAP payload at this level; but most of the time it is more convenient to map the SOAP body to Java classes that provide more immediate access to the data elements. Axis2 provides several automatically generated databindings that implement this mapping. We will make use of the Axiom Databinding (ADB).

There is an Eclipse wizard for the automatic code generation; however, in our experience, this wizard is often unstable and throws errors for no obvious reason. We therefore prefer the command line tool provided by Axis2.

When you call `$AXIS2_HOME/bin/wsdl2java.sh --help` in a shell, you get a long list of available options and their descriptions. Fortunately, we need only few of them:

-o: The top level directory where the generated files are placed. In our case this has to be the project folder in our workspace. So the output directory is `$ECLIPSE_WORKSPACE/NameService` presuming your workspace is located in the directory `$ECLIPSE_WORKSPACE`.

-S: The output directory for the generated source code. The argument is interpreted relative to the top level directory specified with `-o` whence we have to pass `-S src`.

-R: The output directory for the generated or copied resource files (i. e. configuration and WSDL files). Axis2 expects these files in a subdirectory named `META-INF` inside the service directory in the Axis2 repository. Since we instructed Eclipse in Sect. 2.1.2 to use this directory as its output folder and any folder we place in Eclipse's source folder will be copied over, it is sufficient if we pass `-R src/META-INF` to `wsdl2java.sh`. (The argument is again interpreted relative to the top level directory specified with `-o`.)

-ss: Generate a service skeleton. Without this option, `wsdl2java.sh` generates codes for service clients only.

-sd: Generate a service descriptor. This is an XML file that contains various Axis2 specific configuration options.

-d: The databinding to use. We will ask for an ADB binding, but alternatives are XMLBeans, jibx and jaxbri.

-uri: The file system path or the URL of the WSDL file.

Putting all this together, we have to run the following command in the shell:[6]

```
$AXIS2_HOME/bin/wsdl2java.sh \
  -o $ECLIPSE_WORKSPACE/NameService/ -S src \
  -R src/META-INF -ss -sd -d adb \
  -uri $ECLIPSE_WORKSPACE/NameService.wsdl
```

If you now select the NameService project in Eclipse's package explorer and press the F5-key to refresh the view, then you will find a java package info.textgrid.namespaces.examples.nameservice alongside a META-INF directory in the project's source folder. Most of the classes inside this package are part of the databinding, i. e. they represent the operation's arguments, return values and faults. Their names correspond directly to the element and type names found in the WSDL.

The class NameServiceSkeleton is of more immediate interest. It contains for each service operation a corresponding method, which means in our case there is a sole method ExtractNamesResponse extractNames(ExtractNames). The automatically generated stub unconditionally throws an UnsupportedOperationException; it is up to us to replace this method with an implementation of the desired functionality.

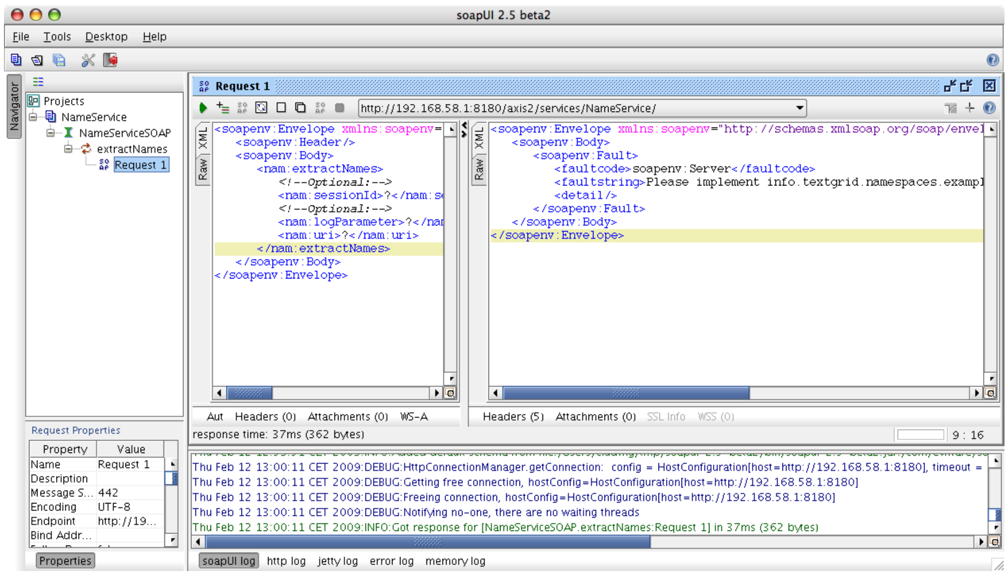
SoapUI. At this point, we have a complete Web service. Admittedly, it does not do anything remotely useful yet - in fact, it will always return a fault caused by the exception thrown in NameServiceSkeleton. However, we can already convince ourselves that the service was successfully deployed and can be executed by Axis2. The first step for this is, of course, to start the Axis2 Web server by running the script given in Listing 2. In the Web server's console output you should find messages that NameService was deployed and that the server is listening on port 8180.

Since we don't have any programmatic NameService client yet, we will resort to an extremely helpful tool: SoapUI [evi]. Among other features, this application generates SOAP messages for calls to the service operations in a given WSDL. You can fill in your own data in these SOAP messages, submit them to the service, and observe the response.

Start SoapUI and choose "File" / "New soapUI Project". Specify a project name of your choice and enter http://localhost:8180/axis2/services/NameService?wsdl as the WSDL location. (A file system path to the NameService's path should work just as well.) When you close the dialog, SoapUI will parse the WSDL and automatically create sample requests for all service operations.

After expanding the project in the explorer to the left completely, a double-click on Request1 shows you in the left half of a new window the SOAP request to be sent. All data is replaced by question marks as placeholders; but since we don't expect the service

[6]Alternatively, you can select "Replace with" / "Branch/Tag" from the NameService's context menu and overwrite your project with a copy of https://develop.sub.uni-goettingen.de/repos/textgrid/tags/DevelopersWorkshop_2009-01-22/NameService_Step2.



```
http://192.168.58.1:8180/axis2/services/NameService/
```

```
<soapenv:Envelope xmlns:soapenv=
   <soapenv:Header/>
   <soapenv:Body>
      <nam:extractNames>
         <!--Optional:-->
         <nam:sessionId>?</nam:se
         <!--Optional:-->
         <nam:logParameter>?</nam
         <nam:uri>?</nam:uri>
      </nam:extractNames>
   </soapenv:Body>
</soapenv:Envelope>
```

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envel
   <soapenv:Body>
      <soapenv:Fault>
         <faultcode>soapenv:Server</faultcode>
         <faultstring>Please implement info.textgrid.namespaces.exampl
         <detail/>
      </soapenv:Fault>
   </soapenv:Body>
</soapenv:Envelope>
```

Property	Value
Name	Request 1
Description	
Message S...	442
Encoding	UTF-8
Endpoint	http://19...
Bind Addr...	

```
response time: 37ms (362 bytes)
```

```
Thu Feb 12 13:00:11 CET 2009:DEBUG:HttpConnectionManager.getConnection:  config = HostConfiguration[host=http://192.168.58.1:8180], timeout =
Thu Feb 12 13:00:11 CET 2009:DEBUG:Getting free connection, hostConfig=HostConfiguration[host=http://192.168.58.1:8180]
Thu Feb 12 13:00:11 CET 2009:DEBUG:Freeing connection, hostConfig=HostConfiguration[host=http://192.168.58.1:8180]
Thu Feb 12 13:00:11 CET 2009:DEBUG:Notifying no-one, there are no waiting threads
Thu Feb 12 13:00:11 CET 2009:INFO:Got response for [NameServiceSOAP.extractNames:Request 1] in 37ms (362 bytes)
```

Figure 4: SoapUI – checking the service deployment

yet to evaluate the data, this does not matter. So simply click on the “send” button (the green arrow in the top left corner of the request window) and SoapUI will show you the service’s response in the right half of the request window. As expected, it is a fault message telling you to implement the extractNames operation, cf. Fig. 4. But this tells you at least that your service is correctly deployed and is being served by the Axis2 Web server.

2.4 TextGrid Middleware Configuration

Our NameService has to access the TextGrid middleware for reading the TextGrid object and for logging. TextGrid reserves the right to move its middleware to new servers without prior notice – so where can you get the current endpoints from? That’s the raison d’être of TextGrid’s ConfService; it will provide you with up-to-date configuration information for TextGrid’s middleware services and is the only service whose endpoint you can presume to be stable.

To use the ConfService, you could either fetch it’s WSDL from https://textgridlab.org/axis2/services/confserv?wsdl. You’d then create your own Web service client using, say, the tool `wsdl2java.sh` from the previous subsection again but this time with the option –s to generate client code. However, if you want to make life easier for yourself then you all you need to do is download https://develop.sub.uni-goettingen.de/repos/textgrid/trunk/middleware/confclient/confclient.jar.

`confclient.jar` exports the package info.textgrid.middleware.confclient and therein exist two convenience classes: ConfservClient’s method getall()returns a java.util.HashMap<String,String> that holds for all middleware services the respective endpoints. You can find the keys into this map in ConfservClientConstants as final attributes.

Listing 3: Return TG-Crud's service endpoint from NameService.extractNames

```java
public static final String DEFAULT_TEXTGRID_CONFIGURATION_ENDPOINT = "https://
    textgridlab.org/axis2/services/confserv";

HashMap<String, String> getConfiguration(String configurationServiceEndpoint) {
    try {
        ConfservClient confservClient = new
        ConfservClient(configurationServiceEndpoint);
        return confservClient.getAll();
    } catch (Exception ex) {
        return null;
    }
}

public ExtractNamesResponse extractNames(ExtractNames extractNames)
         throws ExtractNamesLogFaultMsg,
                ExtractNamesCRUDFaultMsg,
                ExtractNamesDataFaultMsg {
    getConfiguration(DEFAULT_TEXTGRID_CONFIGURATION_ENDPOINT)
    ExtractNamesResponse response = new ExtractNamesResponse();
    response.addName("CRUD␣available␣at:␣"
                    + config.get(ConfservClientConstants.TG_CRUD));
    return response;
}
```

For instance, ConfservClientConstants.TG_CRUD is the key for TG-Crud's endpoint and ConfservClientConstants.LOG_SERVICE is the key for TextGrid's logging service's endpoint.

It is therefore very easy to fetch, say, TG-Crud's endpoint: You only need to instantiate ConfservClient, call the getAll() method and retrieve the endpoint with the key ConfservClientConstants.TG_CRUD. In Listing 3 we actually "abuse" the response message of extractNames and return the so determined endpoint instead of a name. If you modify `NameServiceSkeleton.java` accordingly and call the service operation again from SoapUI, then you should see an URL in the response.

However, we don't want to leave it like this for two reasons:

1. Web service calls are expensive – they slow computationally inexpensive services significantly down. It is therefore preferable to cache the response from ConfService rather than querying it every time your service is invoked for information that is very unlikely to change between two calls.

2. You may wish to access an experimental release of TextGrid's infrastructure, e. g. in preparation for an announced TextGrid upgrade. In that case you have to connect to an alternative instance of ConfService, using a different endpoint. We'd therefore like to keep the endpoint of ConfService configurable by the administrator responsible for our service's deployment.

Listing 4: Service Parametrisation and Information Caching

```java
public void destroy(ServiceContext serviceContext) {
    configurationMap = null;
}

public void init(ServiceContext serviceContext) throws AxisFault {
    String configurationServiceEndpoint = null;
    try {
        MessageContext msgContext = MessageContext
                       .getCurrentMessageContext();
        Parameter myParameter = msgContext
                       .getParameter(TEXTGRID_CONFIGURATION_ENDPOINT);
        configurationServiceEndpoint = myParameter.getParameterElement()
                       .getText();
    } catch (Exception ex) {
        configurationServiceEndpoint = DEFAULT_TEXTGRID_CONFIGURATION_ENDPOINT;
    }
    configurationMap = getConfiguration(configurationServiceEndpoint);
    if (configurationMap == null
                && !DEFAULT_TEXTGRID_CONFIGURATION_ENDPOINT
                               .equals(configurationServiceEndpoint)) {
        configurationMap = getConfiguration(
            DEFAULT_TEXTGRID_CONFIGURATION_ENDPOINT);
    }
}
```

Neither desideratum is hard to achieve, but you probably need a basic understanding of Axis2's architecture to comprehend Listing 4. Jayasinghe [Jay08] provides a thorough but still accessible introduction into this topic.

Unless you instruct Axis2 otherwise, the lifetime of an instance of the service class may or may not extend an individual operation invocation. Axis2 typically keeps some instances alive that are re-used for further invocations, though.

Since we want to cache the configuration map in a member variable of the service objects, we need to ensure that the data is at some point refreshed. For simplicity's sake we clear the cache each time a service session ends. If we'd want the service sessions to actually last longer than a single service operation then we'd have to take further configuration steps. Since our NameService serves demonstration purposes only, we skip this configuration, though.

Each time a service session commences or ends, Axis2 checks by reflexion if the service class has a method init(ServiceContext) or destroy(ServiceContext), respectively, and calls them. We take advantage of this behaviour to bind the lifetime of the information about TextGrid's infrastructure to the service sessions' lifetime: In NameServiceSkeleton.init(ServiceContext), we store it in the field NameServiceSkeleton.configurationMap; in NameServiceSkeleton.destroy(ServiceContext), we set this reference back to null.

Listing 5: Configuring the ConfService Endpoint used by NameService

```xml
<?xml version="1.0" encoding="UTF-8"?>
<serviceGroup>
  <service name="NameService">
    <messageReceivers>
      <messageReceiver
        mep="http://www.w3.org/ns/wsdl/in-out"
         class="info.textgrid.namespaces.examples.nameservice.
             NameServiceMessageReceiverInOut"/>
    </messageReceivers>
    <parameter name="ServiceClass">
      info.textgrid.namespaces.examples.nameservice.NameServiceSkeleton
    </parameter>
    <parameter name="TextGridConfigurationEndpoint">
      https://textgridlab.org/axis2/services/confserv
    </parameter>
    <!-- ... more parameters ... -->
  </service>
</serviceGroup>
```

The static configuration of a service in its `META-INF/service.xml` (or of the enclosing service group or of the whole Axis2 service container) is reflected by Axis2 at runtime in so called *contexts*. These contexts form a hierarchy; if you look up some information in a context and it is not found there, then Axis2 goes up in this hierarchy until it finds the information or determines at the root context that the information is not there. Therefore, if you look for a particular value in, say, a service context but the parameter was actually set at the level of the enclosing service group, then the value is automatically retrieved from the corresponding service group context.

In addition to the contexts corresponding to the various configuration levels, there is an additionally message context that holds information about the current invocation's request message. The message context is in the aforementioned hierarchy under the operation context. You can therefore access any information stored in any context along this hierarchy by querying the current message context. This is demonstrated in NameServiceSkeleton.init(ServiceContext).[7] Now we can easily change the endpoint used for invoking TextGrid's ConfService: All we have to do is modify a parameter entry in NameService's `META-INF/service.xml` as shown in Listing 5.

2.5 Invoking TG-CRUD

Our NameService requires read access to the TextGrid object from which it is supposed to extract the names. TextGrid offers a single point of access to its objects which is the

[7]You can get the code we developed so far by selecting "Replace with" / "Branch/Tag" from the project NameService's context menu and overwriting its content with https://develop.sub.uni-goettingen.de/repos/textgrid/tags/DevelopersWorkshop_2009-01-22/NameService_Step3.

service TG-CRUD. “CRUD” is an acronym for *Create, Read, Update, and Delete* which more or less sums up the functionality provided by TG-CRUD.

Each TextGrid object has a set of metadata associated with it – both descriptive metadata that provides information about the object’s content like who is its author, when was it originally written etc. as well as administrative metadata that covers technical details like when was the object created and last modified, in which relations is it to other objects and so on.[8] For performance’s sake, it is possible to read and update only the metadata without transmitting the potentially large content. Since the metadata is small compared to the content of typical objects, the *read* operation always returns both metadata and content. For the purpose of NameService we are not interested in the metadata, however, and will simply ignore it. TG-CRUD also supports transmitting large files more efficiently by using the Message Transmission Optimization Mechanism (MTOM), but this is beyond the scope of this tutorial.

As in the case of ConfService, you can fetch TG-CRUD’s WSDL[9] and build your own client or you can simply download the JAR file provided by TextGrid[10] The JAR file contains a service stub `info.textgrid.middleware.tgcrud.client.adb.TGCrudServiceStub` built with ADB.

Listing 6 shows how to access the content of a TextGrid object as a `java.io.InputStream`:

1. We instantiate the service stub and pass the service endpoint we got from ConfService to its constructor.

2. We prepare the request message. As mentioned previously, ADB represents all XML elements and XML Schema Types in the WSDL by their own classes. We therefore have to instantiate the class `Read` (a class defined inside `TGCrudServiceStub`) as the top level element of the request message. However, its child elements are defined in its schema type, so we also need to instantiate `ReadType` that offers setters for the actual parameters. The `ReadType` object is then enclosed in the `Read` object by the setter `Read.setRead(ReadType)`.

3. We invoke TG-CRUD’s read operation by a call of the stub’s method of the same name and passing the request object. The method returns the response object; if the service sends a fault, then the stub will throw an exception. Here it is most convenient to work with an IDE like Eclipse that can automatically add the necessary throw statement to the current function’s head. (The digits at the end of the generated exception classes’ names are an artifact of ADB.)

4. We extract the relevant information from the response object that is similarly nested as the request object. The object’s content is transferred as Base64 encoded binary data that ADB automatically wraps in a `javax.activation.DataHandler`. In turn,

[8] The metadata’s schema is defined and documented in https://develop.sub.uni-goettingen.de/repos/textgrid/trunk/middleware/metadata/resources/schemas/textgrid-metadata_2008-07-24.xsd.

[9] https://textgridlab.org/axis2/services/TGCrudService?wsdl

[10] https://develop.sub.uni-goettingen.de/repos/textgrid/tags/DevelopersWorkshop_2009-01-22/NameService_Step4/src/lib/tgcrudclient.jar

Listing 6: Reading TextGrid Objects

```java
InputStream readTextGridObject(URI uri, String logParam, String sessionParam)
            throws ObjectNotFoundFaultException0,
                   AuthFaultException1,
                   IoFaultException2,
                   MetadataParseFaultException3,
                   IOException {
    // instantiate a CRUD stub
    String tgcrudEndpoint = configurationMap
                    .get(ConfservClientConstants.TG_CRUD);
    TGCrudServiceStub crud = new TGCrudServiceStub(tgcrudEndpoint);

    // prepare the CRUD read request
    ReadType readType = new ReadType();
    if (logParam != null) {
            readType.setLogParameter(logParam);
    }
    if (sessionParam != null) {
            readType.setSessionId(sessionParam);
    }
    readType.setUri(uri);
    Read readRequest = new Read();
    readRequest.setRead(readType);

    // execute the actual service call
    ReadResponse response = crud.read(readRequest);

    // retrieve the object's content and turn it into an InputStream
    ReadResponseType responseType = response.getReadResponse();
    DataHandler datahandler = responseType.getData();
    return datahandler.getInputStream();
}
```

DataHandler provides a conversion method into an InputStream whence we get the object's content from the response object with as much as three lines of code.

Step 4 of the NameService project[11] again "abuses" extractNames's response message and returns the complete content of the TextGrid object in lieu of a name. Since the argument extractNames of NameServiceSkeleton.extractNames(ExtractNames) is also of an ADB generated type, the URI of the requested object, the logging parameter, and the session ID required for the invocation of TG-CRUD can be extracted by simple getter methods. In case the request message's optional elements are omitted, the getters return null.

2.6 Name Extraction

Now that we have access to the TextGrid object's content, the remaining steps to implement NameService's core functionality are by no means TextGrid specific, cf. Listing 7. We thus sketch them only; for details on Axiom we recommend you refer to [FTW07].

NameService presumes the requested TextGrid object contains a valid TEI document [BB09] in which names are enclosed in ⟨*tei:persName xmlns:tei="http://www.tei-c.org/ns/1.0"/*⟩. We can therefore find all names in the TextGrid object by evaluating the XPath expression `"//tei:persName"`.

1. The code we developed in Sect. 2.5 returns the requested TextGrid object's content as an InputStream. In order to parse it with Axiom, we need to create a javax.xml.stream.XMLStreamReader that in turn expects a java.io.InputReader.
2. The actual parser is a StAXOMBuilder that is initialized with an Axiom factory and the previously generated reader. The parser's method getDocument() provides access to the TextGrid object's content as an XML document.
3. Evaluating the aforementioned XPath on this document returns all ⟨*tei:persName*⟩ elements in a java.util.List. We extract their text contents and replace all whitespace sequences by a single blank before we add them to the response message.

As usual, the code is available as a complete Eclipse project in TextGrid's subversion repository[12].

2.7 Error Handling

Our code still lacks any proper error handling. We turn any exception we catch into a generic java.lang.RuntimeException. In turn, Axis2 reacts by sending a generic Axis2 service fault. Therefore, "Something went wrong!" is all the client learns; it receives hardly any information on which it could base its fault handling.

[11] https://develop.sub.uni-goettingen.de/repos/textgrid/tags/DevelopersWorkshop_2009-01-22/NameService_Step4

[12] https://develop.sub.uni-goettingen.de/repos/textgrid/tags/DevelopersWorkshop_2009-01-22/NameService_Step5

Listing 7: Extracting Names by Means of XPath

```java
ExtractNamesResponse response = new ExtractNamesResponse();
try {
    InputStream inputStream = readTextGridObject(uri, logParam, sid);
    InputStreamReader reader = new InputStreamReader(inputStream);
    XMLStreamReader xmlReader = XMLInputFactory
                    .newInstance()
                    .createXMLStreamReader(reader);

    OMFactory factory = OMAbstractFactory.getOMFactory();
    StAXOMBuilder builder = new StAXOMBuilder(factory, xmlReader);
    OMDocument document = builder.getDocument();
    AXIOMXPath xpathExpr = new AXIOMXPath("//tei:persName");
    xpathExpr.addNamespace("tei", "http://www.tei-c.org/ns/1.0");

    List persNameElems = xpathExpr.selectNodes(document
                    .getOMDocumentElement());
    for (Object elem : persNameElems) {
        if (OMElement.class.isInstance(elem)) {
            OMElement nameElem = (OMElement) elem;
            String name = nameElem.getText()
                    .replaceAll("\\p{javaWhitespace}+", "␣");
            response.addName(name);
        }
    }
} catch (Exception ex) {
    throw new RuntimeException(
                   "Data␣error␣-␣needs␣to␣be␣handled␣properly", ex);
}
```

However, we already defined three different fault types and the element type of the fault messages in NameService's WSDL. So let's make NameService actually generate proper faults.

If you check the NameService WSDL, then you see in the port type definition specifies for each fault the extractNames operation may produce a corresponding message. `wsdl2java.sh` generated for each fault an exception type where the exception is named after the corresponding message, respectively. So in our case, we have the exception types ExtractNamesCRUDFaultMsg, ExtractNamesDataFaultMsg, and ExtractNamesLogFaultMsg; these are also the exception types listed in the throws specifier of NameServiceSkeleton.extractNames(ExtractNames).

The generated fault message exception types have the constructors that are common for Java exception types; i. e., you can construct new exceptions with no argument, a single String argument, or a String and another Throwable as arguments. In addition, there are methods setFaultMessage and getFaultMessage that allow to set and retrieve the XML element that the WSDL specifies as the faults' parameters, i. e. objects of the ADB types ExtractNamesCRUDFault, ExtractNamesDataFault, and ExtractNamesLogFault, respectively.

From this point on, we have the usual structure of ADB classes: The XML Schema type of the fault elements is represented as its own class ExtractNamesFaultType that encapsulates the actual fault details error message, stack trace, and remote details (a string that is supposed to store detailed information about the problem that caused a service NameService is building on to fail).

We therefore can factor out the construction of ExtractNamesFaultType objects into a separate function but define separate functions that turn an exception we encountered into one of the ADB exception types. Listing 8 shows this for the CRUD faults, the other faults are done similarly.

Now, whenever we encounter an exception (or any other error state that for which we then construct an exception), we call the appropriate method that encapsulates this exception in a the fault exception and throw its return value. You can find all thus resulting modifications to extractNames(ExtractNames) in the TextGrid Subversion repository[13].

2.8 Logging

For any non-trivial service you probably want to write log messages that allow the end-user to check the progress and, in case of an error, where things went wrong. However, the typical user has no access to your server's local file system, so messages logged into a local file may be helpful for your administrator, but not for the user. Therefore, the log messages must be sent sent where the user can read them. That is particularly true if your service is used as part of a workflow. Then the log messages from all nodes of this workflow should be aggregated in one place that, of course, is only accessible by the user

[13] https://develop.sub.uni-goettingen.de/repos/textgrid/tags/DevelopersWorkshop_2009-01-22/NameService_Step6

Listing 8: Constructing a CRUD Fault from an Exception

```java
ExtractNamesFaultType createExtractNamesFaultType(Exception ex) {
    ExtractNamesFaultType param = new ExtractNamesFaultType();
    param.setErrorMsg(ex.getMessage());
    StringWriter stringWriter = new StringWriter();
    PrintWriter writer = new PrintWriter(stringWriter);
    ex.printStackTrace(writer);
    param.setStackTrace(stringWriter.toString());
    Throwable cause = ex.getCause();
    if (cause != null) {
        param.setRemoteDetails(cause.getMessage());
    }
    return param;
}

ExtractNamesCRUDFaultMsg createCRUDFaultMsg(Exception ex) {
    ExtractNamesFaultType param = createExtractNamesFaultType(ex);
    ExtractNamesCRUDFault fault = new ExtractNamesCRUDFault();
    fault.setExtractNamesCRUDFault(param);
    ExtractNamesCRUDFaultMsg msg = new ExtractNamesCRUDFaultMsg("CRUD␣Fault:␣" +
         param.getErrorMsg(), ex);
    msg.setFaultMessage(fault);
    msg.setStackTrace(ex.getStackTrace());
    return msg;
}
```

who initiated the workflow.

TextGrid's log service provides such a "log message sink"; there is also a view in the TextGridLab that displays and filters the log messages in the current log session. In our last step, we are going to extend NameService so it makes use of this facility.

It is up to your client to initiate a log session and provide you with its ID. This is the purpose of the logging parameter that is expected by any TextGrid service. This parameter holds even more information, though: It specifies a log level that is used to determine which messages are actually submitted to the log server and the logging service's endpoint. That means that you don't have to refer to the ConfService to find the logging service; you rather rely on the value provided by your client.

As usual, you are free to download the WSDL of TextGrid's Logging Service[14] and develop your own client. However, if you develop your service in Java, then we recommend you take advantage of the client package provided by TextGrid[15].

This package encapsulates the logging service in info.textgrid.middleware.textgridlogger.TextgridLogger that is initialized with the logging parameter provided by your client (or an empty string that suppresses all logging). Whenever you submit a log message, then you are also supposed to provide a unique log message source ID; that way messages from different sources can be distinguished in the common log. Fortunately, Axis2 provides an API to generate such an ID for each service invocation:

```
MessageContext msgContext = MessageContext.getCurrentMessageContext();
String logID = msgContext.getLogIDString();
```

It is primarily intended for Axis2's own logging mechanism, but there is no reason not to use it for TextGrid's purposes as well.

Besides the message and this logging ID, TextgridLogger.log(String,String,int) expects an integer indicating the importance of the log message. The supported values are 0 through 4 where messages at level 0 are never logged, messages at level 1 only if the threshold of the current logging session is set to "error" or lower, messages at level 2 is the threshold is "warning" or lower and so forth. (The remaining levels are "info" and "debug".)

Listing 9 shows an excerpt from the final NameService's implementation that demonstrates both the initialization of the logger object and its usage. As usual, the complete code is available from the TextGrid Subversion repository[16].

3 TextGridLab Client

In this section we will develop a simple user interface for the service created in section 1. We will add a context menu item to every context menu of single TextGrid objects (as they appear in the navigator and search results view) which will result in the display from

[14]https://textgridlab.org/axis2/services/textlog?wsdl
[15]https://develop.sub.uni-goettingen.de/repos/textgrid/trunk/middleware/textgridlogger/TextgridLogger.jar
[16]https://develop.sub.uni-goettingen.de/repos/textgrid/tags/DevelopersWorkshop_2009-01-22/NameService_Step7

Listing 9: Logger Initialization and Usage

```java
String logParam = extractNames.getLogParameter();
URI uri = extractNames.getUri();
TextgridLogger logger = new TextgridLogger(logParam == null
                ? "" : logParam);
MessageContext msgContext = MessageContext.getCurrentMessageContext();
String logID = msgContext.getLogIDString();

try {
    logger.log("ExtractNames␣called␣for␣object␣" +
                    uri.toString(),
                    logID, LogLevel.INFO);
} catch (RemoteException ex) {
    throw createLogFaultMsg(ex);
}
```

figure 5: The selected TextGridObject will be opened in an editor, the names retrieved by the service appear in a view and when you select a name there the corresponding Wikipedia page will be opened in a view at the bottom of the page.

TextGridLab is based on the *Eclipse Rich Client Platform* framework[17]. Every bit of code of the application lives in a *plugin*, and so your client will be a plugin, as well.

3.0 Setting up the development environment

Eclipse and Java installation. The current version of the TextGridLab is based on Eclipse 3.5 (Galileo)[18]. We suggest you download the version for Plug-in Development.

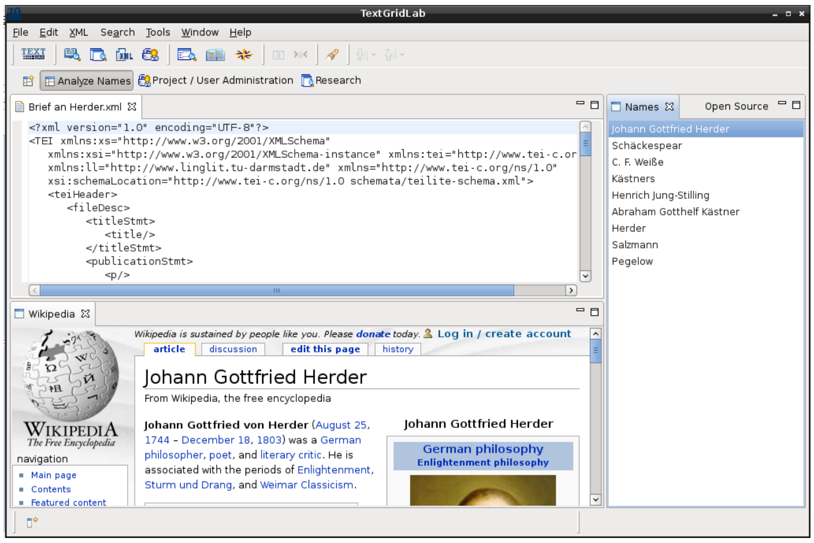
Additionally, please select Java 1.5 as compiler. This is necessary since there is no version of Java 6 for some still popular versions of MacOS X.

You will need to install a SVN client (we're using Subclipse[19]), as well.

Checkout TextGridLab. To get the current version of TextGridLab's source code, please import the Team Project File from https://develop.sub.uni-goettingen.de/repos/textgrid/trunk/lab/build/textgridlab-used.psf using File / Import / Team / Team Project Set. Doing so will check out all TextGridLab plugins of our main development tree as projects in your workspace.

Setup a run configuration. Now you should try to run TextGridLab from within Eclipse. Open the file `textgridlab.product` from the plugin `info.textgrid.lab.core.application`. Click *Launch an Eclipse application* from the Overview page. This should

[17]See [Ani08b] for an introduction
[18]previous versions were based on Eclipse 3.3
[19]http://subclipse.tigris.org/



```
<?xml version="1.0" encoding="UTF-8"?>
<TEI xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:tei="http://www.tei-c.or
   xmlns:ll="http://www.linglit.tu-darmstadt.de" xmlns="http://www.tei-c.org/ns/1.0"
   xsi:schemaLocation="http://www.tei-c.org/ns/1.0 schemata/teilite-schema.xml">
   <teiHeader>
      <fileDesc>
         <titleStmt>
            <title/>
         </titleStmt>
         <publicationStmt>
            <p/>
```

Johann Gottfried Herder
Schäckespear
C. F. Weiße
Kästners
Henrich Jung-Stilling
Abraham Gotthelf Kästner
Herder
Salzmann
Pegelow

Wikipedia is sustained by people like you. Please donate today. Log in / create account

article | discussion | edit this page | history

Johann Gottfried Herder

From Wikipedia, the free encyclopedia

Johann Gottfried von Herder (August 25, 1744 – December 18, 1803) was a German philosopher, poet, and literary critic. He is associated with the periods of Enlightenment, Sturm und Drang, and Weimar Classicism.

Johann Gottfried Herder

German philosophy
Enlightenment philosophy

WIKIPEDIA
The Free Encyclopedia
navigation
- Main page
- Contents
- Featured content

Figure 5: Our goal: The Analyze Names Perspective.

launch the lab, and it should create a run configuration that you can find in the drop-down menu of the Run and Debug buttons in your Eclipse window as well as in the Run Configurations dialog that you can open from the Run menu.

3.1 Creating plugin and menu item

Start by creating a new plugin project (using File / New). After entering some meaningful metadata for the plugin Eclipse press Finish, and Eclipse will open the plugin manifest in a specialized editor. To learn more about plugins and their manifest, see, e. g., [Ani08a].

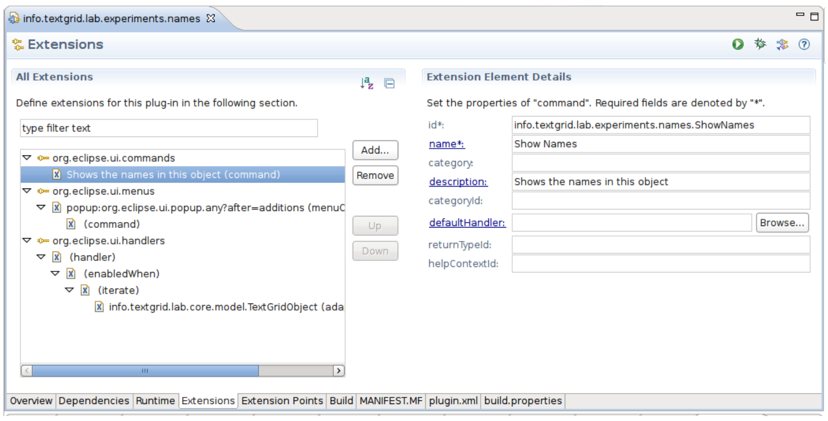
For this tutorial, you will need the Dependencies and the Extensions pages: On the *Dependencies* page, you add the plugins your plugin requires (and thus add their code to your classpath). *Extensions* are basically an XML structure that is used to declaratively specify contributions to Eclipse applications. E.g., you will declare user interface elements like menu items here with everything that is needed to represent them, and your implementation classes will only be loaded when the menu item is clicked for the first time.

3.1.1 Declaring the menu contribution

To add menu items, we will use the newer of Eclipse's command contribution mechanisms: The *Command Framework* [WB$^+$], [Plu, section *Plugging into the Workbench / Basic workbench extension points using commands*].

Add the following extension points, and use the context menu and the form on the right part of the page to add subelements and attributes:

- A *command* using the extension point org.eclipse.ui.commands. This is basically



Define extensions for this plug-in in the following section.

type filter text

```
▽ org.eclipse.ui.commands
      Shows the names in this object (command)
▽ org.eclipse.ui.menus
   ▽ popup:org.eclipse.ui.popup.any?after=additions (menuC
         (command)
▽ org.eclipse.ui.handlers
   ▽ (handler)
      ▽ (enabledWhen)
         ▽ (iterate)
               info.textgrid.lab.core.model.TextGridObject (ada
```

Extension Element Details

Set the properties of "command". Required fields are denoted by "*".

id*:	info.textgrid.lab.experiments.names.ShowNames
name*:	Show Names
category:	
description:	Shows the names in this object
categoryId:	
defaultHandler:	
returnTypeId:	
helpContextId:	

Figure 6: The Extensions page after adding the contributions for your menu item.

an abstract representation that defines an ID and holds information like the (localizable) name and description that would by default be displayed in a menu.

- A *menu contribution* using the extension point org.eclipse.ui.menus. You can use the special locationURI popup:org.eclipse.ui.popup.any?after=additions to contribute to any context menu. Add a command, you only need to specify the command id (you declared above).

- A *handler* – declared using the extension point org.eclipse.ui.handlers – associates the actual implementation with the command. Click on the *class* link to create the handler implementation (it is discussed in Sec. 3.1.2).

  Additionally, you may add an *enabledWhen* expression to the handler declaration. The example from Listing 10, ll. 21ff, declares that the handler should only be enabled when all selected objects can be adapted to a TextGridObject. TextGridObject is the class representing one document from the repository in TextGridLab.

Listing 10: The XML representation of the extensions from Fig. 6. This can be found on the *plugin.xml* page.

```
1   <extension
2         point="org.eclipse.ui.commands">
3      <command
4            description="Shows␣the␣names␣in␣this␣object"
5            id="info.textgrid.lab.experiments.names.ShowNames"
6            name="Show␣Names">
7      </command>
8   </extension>
9   <extension
10        point="org.eclipse.ui.menus">
11     <menuContribution
12           locationURI="popup:org.eclipse.ui.popup.any?after=additions">
```

```
            <command
                  commandId="info.textgrid.lab.experiments.names.ShowNames"
                  style="push">
            </command>
         </menuContribution>
      </extension>
      <extension
            point="org.eclipse.ui.handlers">
         <handler
               class="info.textgrid.lab.experiments.names.ShowNamesHandler"
               commandId="info.textgrid.lab.experiments.names.ShowNames">
            <enabledWhen>
               <iterate
                     ifEmpty="false"
                     operator="and">
                  <adapt
                        type="info.textgrid.lab.core.model.TextGridObject">
                  </adapt>
               </iterate>
            </enabledWhen>
         </handler>
      </extension>
```

3.1.2 First implementation for the menu contribution

Clicking on the *class* link in the virgin handler form lets you create a new class that implements the actual functionality for the command. We suggest that you let your new class inherit from AbstractHandler and simply override the execute method with your implementation instead of implementing the IHandler interface by hand.

Our first handler implementation will simply display some metadata of the first selected TextGridObject in a dialog box.

Handlers' execute methods get passed an event from which they can extract relevant information using static methods of the HandlerUtil class. In the first steps, we'll see if the selection is a structured selection[20], and try to adapt its contents to TextGridObject:

```
@Override
public Object execute(ExecutionEvent event) throws ExecutionException {
  ISelection currentSelection = HandlerUtil.getCurrentSelection(event);
  if (currentSelection instanceof IStructuredSelection) {
    IStructuredSelection selection = (IStructuredSelection) currentSelection;
    TextGridObject object = AdapterUtils.getAdapter(selection
                        .getFirstElement(), TextGridObject.class);
```

The adapter pattern [Bea08] is used throughout Eclipse and TextGridLab to convert between not directly related classes. Items like navigator entries and search results that represent an object in TextGrid's repository can always be adapted to TextGridObject.

[20]To learn more about selections, see [Hof06]

getAdapter may always return **null** if nobody could fulfill the conversion request, but if it does not, the result is of the requested class. We can use TextGridObject's methods to retrieve metadata for the object:

```
    if (object != null) {
      try {
        MessageDialog.openInformation(HandlerUtil
          .getActiveShell(event), "Names", object.getTitle());
      } catch (CoreException e) {
        StatusManager.getManager().handle(e.getStatus(),
               StatusManager.SHOW | StatusManager.LOG);
      }
    }
  }
  return null;
}
```

TextGridObject methods that may access the grid usually throw some descendant of CoreException, which always contain Eclipse's Status objects that can be displayed or shown in the Error Log (Ctrl+3 Error RET). In this case we log the error and display it in a message dialog.

3.2 Calling the service

In this section, you will access the service created in Section 2 to retrieve a list of names from the selected document.

3.2.1 Generating the client stubs

First, you'll have to generate java stubs for the clients. As the corresponding Eclipse plugins are currently broken, we use the command line. So, open a terminal in the project directory of your plugin and issue the command:

```
wsdl2java.sh -uri ../NameService.wsdl
```

This will generate the stubs classes in the `src` subdirectory. After switching back to Eclipse, select the project in the Package Explorer and hit F5 or choose Refresh from the context menu in order to make Eclipse realize something has happened on your disk.

3.2.2 Modelling names

To represent a name in our plugin, we implement a very simple wrapper Name that simply aggregates the string returned by the service:

```
public class Name {
  private String name;

  public Name(String string) {
    this.setName(string);
  }
  public void setName(String name) {
    this.name = name;
  }

  @Override
  public String toString() {
    return name;
  }

  public String getName() {
    return name;
  }
}
```

3.2.3 Calling the service

Now we'll add a static method extractNames to the Names class that implements the actual service call to this class. First we create the stub and the ADB binding for the service call argument. For the stub, we simply pass in the absolute URL for now:

```
public static Name[] extractNames(TextGridObject object)
  throws CoreException {

try {
  NameServiceStub nameService = new NameServiceStub("http://localhost:8180/axis2
      /services/NameService");

  ExtractNames extractNames = new ExtractNames();
```

We'll have to fill out three arguments: The session ID for authentication, the log argument and the URI of the object to work on. For the session ID, there is a singleton class in TextGridLab's authorization plugin to use. So, add info.textgrid.lab.authn and info.textgrid.lab.log to your plugin's dependencies to be able to write

```
  extractNames.setSessionId(RBACSession.getInstance().getSID(false));
```

Similarly, retrieve the log info string using

```
  extractNames.setLogParameter(logsession.getInstance().getloginfo());
```

Finally, we set the URI argument to the URI retrieved from the TextGridObject and call the service. Unfortunately, ADB uses its own URI class instead of the standard java.net.URI ...

```
  extractNames.setUri(new org.apache.axis2.databinding.types.URI(object.getURI()
      .toASCIIString()));
  ExtractNamesResponse response = nameService
      .extractNames(extractNames);
```

From the response, let's generate the model objects:

```
  Name[] result = new Name[response.getName().length];
  for (int i = 0; i < result.length; i++) {
    result[i] = new Name(response.getName()[i]);
  }
  return result;
```

3.2.4 Error handling

As you will have noticed, NameServiceStub.extractNames can throw quite a lot of exceptions. It is usually a good idea to encapsulate these original exceptions in Eclipse's CoreExceptions or subclasses thereof. CoreExceptions contain a IStatus object that further describes the problem and that can be passed, e.g., to StatusHandler in order to be logged or displayed to the user. You can find a wrapping implementation example for this in the source code in the SVN repository.

3.2.5 Calling extractNames

Finally, we modify our handler from Section 3.1.2 to display the extractNames result instead of the object's title:

```
  Name[] names = Name.extractNames(object);
  MessageDialog.openInformation(HandlerUtil.getActiveShell(event),
      "Extracted␣Names",
      MessageFormat.format("Extracted␣names␣for␣{0}:\n{1}", object,
          Arrays.toString(names)));
```

3.3 A View for names

Instead of in a modal dialog, we would like to display the names in a *View* that can remain on the screen as long as our user wants.

This step handles mainly general Eclipse topics, so we'll hurry through it.

To add a view, we'll have to declare it first by adding a corresponding extension to the plugin manifest. This is the excerpt of plugin.xml generated:

```
   <extension
         point="org.eclipse.ui.views">
      <view
            class="info.textgrid.lab.experiments.names.NamesView"
            id="info.textgrid.lab.experiments.names.NamesView"
            name="Names">
      </view>
   </extension>
```

We'll implement two interesting methods: First, createPartControl defines what the view contains. In our case, simply a ListViewer:

```
 @Override
  public void createPartControl(Composite parent) {
```

```java
    // JFace viewers are wrappers around standard widgets like Lists
    viewer = new ListViewer(parent, SWT.SINGLE);
    // A Content Provider translates the input you provide to (model)
    // objects that should actually displayed in the viewer:
    viewer.setContentProvider(new ArrayContentProvider());
    // A Label Provider translates the model objects from the content
    // provider to strings and images to display
    viewer.setLabelProvider(new LabelProvider());
  }
```

Second, we write a setTextGridObject method that initializes the view with the results for the given object:

```java
  public void setTextGridObject(TextGridObject object) {
    this.textGridObject = object;
    if (viewer != null && !viewer.getControl().isDisposed()) {
      try {
        // The viewer's input is whatever your content provider can
        // handle. Usually, setInput triggers the content provider.
        viewer.setInput(Name.extractNames(object));
      } catch (CoreException e) {
        StatusManager.getManager().handle(e.getStatus(),
            StatusManager.SHOW | StatusManager.LOG);
      }
    }
  }
```

To learn more about viewers, see [Plu, Section JFace UI Framework].

Finally, we modify our handler again to open this view and call the setTextGridObject method:

```java
      // You open other workbench parts by digging through to the
      // component that has a showView method. It's the active
      // workbench window's current page ...
      NamesView view = (NamesView) HandlerUtil
          .getActiveWorkbenchWindow(event).getActivePage()
          .showView("info.textgrid.lab.experiments.names.NamesView");
      view.setTextGridObject(object);
```

3.4 The Wikipedia View

The Wikipedia view will contain a browser widget that loads the corresponding Wikipedia page whenever a name is selected.

Again, the Wikipedia view is mainly an Eclipse topic that covers selection handling as one of the main means of communication inside Eclipse. To learn more about the selection service, see [Hof06].

Again, declare the view in the plugin manifest as in Sec. 3.3 and create its class. In the createControl method, we add a browser control:

```java
  @Override
  public void createPartControl(Composite parent) {
    browser = new Browser(parent, SWT.NONE);
    browser.setLayoutData(new GridData(SWT.FILL, SWT.FILL, true, true));
    browser.setText("Select␣a␣name,␣please");
```

To react on name selection changes, we have to register a selection listener with the workbench window's selection service:

```java
    ISelectionService selectionService = getSite().getWorkbenchWindow()
        .getSelectionService();
    selectionService.addSelectionListener(this);
  }
```

In the handler method, we craft a Wikipedia URL and ask the browser to visit it:

```java
public void selectionChanged(IWorkbenchPart part, ISelection selection) {
  if (!selection.isEmpty() && selection instanceof IStructuredSelection) {
    IStructuredSelection sel = (IStructuredSelection) selection;
    if (sel.getFirstElement() instanceof Name) {
      Name name = (Name) sel.getFirstElement();
      try {
        String url = new URI("http", "de.wikipedia.org", "/wiki",
            "go=Go&search=" + name.getName(), null)
            .toASCIIString();
        browser.setUrl(url);
      } catch (URISyntaxException e) {
        // TODO
 Auto-generated catch block
        e.printStackTrace();
      }
    }
  }
}
```

As a counterpart, the Names view needs to relay its viewer's selection to the workbench. The following code needs to be added to the Names View's createPartControl method (cf. Sec. 3.3):

```java
    // A part's "site" is responsible for all communication with the
    // part's environment. In this case, we register our viewer as the
    // selection provider, so if you click something here, the whole
    // workbench may know.
    getSite().setSelectionProvider(viewer);
```

3.5 Open the object in an editor, create a perspective

In this workshop's final step, you will open the TextGridObject in an editor and assemble editor, names and Wikipedia view to a perspective.

3.5.1 Access (and edit) a TextGridObject's contents

As mentioned before, objects in the TextGrid repository are represented in TextGridLab by instances of the TextGridObject class. However, TextGridObjects *contain* only the metadata. In order to access an TextGridObject's contents, you need to adapt the object to the org.eclipse.resources.IFile interface (which is Eclipse's representation of files in the workspace). IFile has methods setContent and getContent to read from and write to the files.

To open a TextGridObject in a specific editor, it's usually enough to

1. adapt the TextGridObject to IFile,
2. wrap the IFile in a FileEditorInput (the latter is in the plugin org.eclipse.ui.ide) and
3. ask the workbench page to open the editor with the FileEditorInput.

And that's what we do in the handler now:

```
// remember: object is our TextGridObject
IFile file = (IFile) object.getAdapter(IFile.class);
if (file != null) {
  workbenchPage.openEditor(
          new FileEditorInput(file),
          "org.eclipse.ui.DefaultTextEditor");
  // you could use the Open command to try to find the
  // default editor for the specific object ...
}
```

3.5.2 Grouping GUI elements in a perspective

Additionally, we add a *perspective* for the purpose of analyzing names. A perspective arranges a set of user interface elements to a specific (customizable) layout – our perspective will arrange the editor, the names and the Wikipedia view like in Fig. 5 on page 23.

As with views and menu items, a perspective must be declared in the plugin manifest:

```
<extension
     point="org.eclipse.ui.perspectives">
  <perspective
        class="info.textgrid.lab.experiments.names.NamesPerspective"
        id="info.textgrid.lab.experiments.names.NamesPerspective"
        name="Analyze␣Names">
  </perspective>
</extension>
```

The implementing class is really simple, it just implements a createInitialLayout method that defines how the perspective should look like:

```
public void createInitialLayout(IPageLayout layout) {
  layout.setEditorAreaVisible(true); // default
  layout.addView("info.textgrid.lab.experiments.names.NamesView", IPageLayout.
      RIGHT, 0.75f, layout.getEditorArea());
  layout.addView("info.textgrid.lab.experiments.names.WikipediaView",
      IPageLayout.BOTTOM, 0.5f, layout.getEditorArea());
}
```

The arguments are explained in IPageLayouts javadocs.

The only thing remaining open is to switch to the perspective in the handler:

```
  workbenchWindow.getWorkbench().showPerspective(
         "info.textgrid.lab.experiments.names.NamesPerspective",
         workbenchWindow);
```

4 Ways of Improvement

The service and the plug-ins developed so far are fully functional. However, they were consciously kept simplistic; the development of production-ready extensions that take advantage of all possible optimizations and provide all desirable features would have gone beyond the scope of this tutorial. We'd nevertheless like to point some of the most obvious improvements; their implementation is a recommended exercise.

Our NameService implementation caches the TextGrid configuration in a member variable in the service object. This works because once created the data is never modified. In general, though, any data that is supposed to exist beyond a single service operation should be stored within an Axis2 context where it then can be accessed by all service object instances. Since the TextGrid configuration is required by all TextGrid services, it would be most efficient to combine all TextGrid services deployed on a server in one Axis2 service group and to store the configuration in the service group's context. The cache may then be refreshed once a day or whenever access to any middleware service fails.

TG-CRUD supports the more efficient delivery of large objects as MTOM attachments if the client indicates that it can handle MTOM. By default, Axis2 client code does not request MTOM response messages, so we can reduce some of the CPU and network load caused by NameService by explicitly enabling this option.

We should also reduce the default timeout for TG-CRUD (and ConfServ) invocations. Left at its default value, it is very likely that NameService's client times out before any TG-CRUD time-out can be properly reported by a CRUDFault whence the client does not learn the cause of the problem.

Currently, NameService blocks while log messages are submitted to the server. Since Web service calls are expensive (performance-wise), this can significantly hurt the performance of NameService. On the other hand, few users will mind if problems during the log message submissions are silently swallowed; the failure will be apparent from the missing log file entries anyway. The performance can therefore easily be improved if the log messages are not submitted directly but rather added to a message queue that is consumed in a dedicated logging thread that submits all messages.

Finally, the functionality of NameService's extractNames operation is overly restrictive. To be useful in real-life research projects, it should support additional markups or even general XPath expressions.

The first thing to improve in the TextGridLab frontend would be to make the GUI more responsive by moving all network access methods to a background job. Eclipse offers a rather comfortable mechanism for this which includes automatted management of a thread pool, progress reporting and cancellation, see [Val04] for an introduction. And note that you have to do all access to GUI elements in the UI thread, e. g. by scheduling a UIJob from your background job.

Another improvement would be to open the default editor for the opened file's content type which can be determined by combining TextGrid's content type field with Eclipse's editor registry.

Generally, it would have been possible to use TG-search with a specialized XPath query for the NameService's task (but in this case, you would not have learned how to deal with the other utilities).

References

[Ani08a] Chris Aniszczyk. Plug-in development 101, part 1: The fundamentals. *IBM developerWorks*, Apr 2008. http://www.ibm.com/developerworks/library/os-eclipse-plugindev1/.

[Ani08b] Chris Aniszczyk. Plug-in development 101, part 2: Introducing rich-client applications. *IBM developerWorks*, Apr 2008. http://www.ibm.com/developerworks/opensource/library/os-eclipse-plugindev2/.

[Axi] Apache Axis2. http://ws.apache.org/axis2/ (2009-02-09).

[BB09] Lou Burnard and Syd Bauman. *TEI P5: Guidelines for Electronic Text Encoding and Interchange*. TEI Consortium, February 2009. Version 1.3.0 http://www.tei-c.org/release/doc/tei-p5-doc/en/Guidelines.pdf (2009-02-15).

[Bea08] Wayne Beaton. Adapters. Eclipse Corner Article, online at http://www.eclipse.org/articles/article.php?file=Article-Adapters/index.html, Jun 2008.

[BL07] David Booth and Canyang Kevin Liu. Web services description language (WSDL) version 2.0 part 0: Primer. W3C recommendation, W3C, 2007. http://www.w3.org/TR/wsdl20-primer/ (2009-02-10).

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. W3C note, W3C, 2001. http://www.w3.org/TR/wsdl (2009-02-10).

[CMRW07] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. W3C recommendation, W3C, 2007. http://www.w3.org/TR/wsdl20/ (2009-02-10).

[evi] eviware. SoapUI 2.5. http://www.soapui.org (2009-02-11).

[FTW07] Thilo Frotscher, Marc Teufel, and Dapeng Wang. *Java Web Service mit Apache Axis2*. entwickler.press, 2007. (German).

[Hof06] Marc R. Hoffmann. Eclipse workbench: Using the selection service. online at http://www.eclipse.org/articles/Article-WorkbenchSelections/article.html, Apr 2006.

[Jay08] Deepal Jayasinghe. *Quickstart Apache Axis2*. Packt Publishing, 2008.

[Plu] Eclipse Foundation. *Eclipse Platform Plug-in Developer Guide, Programmer's Guide*, version 3.3 edition. http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/int.htm.

[Tex09] TextGrid. *TextGrid Manual: Tool Development*. *TextGrid Report 3.5*, Feb 2009. http://www.textgrid.de/fileadmin/TextGrid/reports/R3_5-manual-tools.pdf.

[TKI08] Kent Tong Ka Iok. *Developing Web Services with Apache Axis2*. Tip Tec Development, 2nd edition, 2008.

[Val04] Michael Valenta. On the job: The eclipse jobs API. http://www.eclipse.org/articles/Article-Concurrency/jobs-api.html, Sep 2004.

[WB$^+$] Paul Webster, John J. Barton, et al. Platform command framework. Eclipse Wiki, http://wiki.eclipse.org/index.php/Platform_Command_Framework.

[WSO] WSO2 Oxygen Tank – the developer portal for open source SOA web services and middleware. http://wso2.org/ (2009-02-09).