SAP Scripting Tool How to Write a Scripting Generator

Applies To

SAP Scripting Tool 0.8.0; Eclipse 3.1.2; Java 1.4; Python; Java Emitter Templates

Summary

This article gives an introduction in developing custom scripting generators for the SAP Scripting Tool. The latter may generate scripting code utilizing backend access in any scripting language or even pass the service connection data to any kind of text file or data consumer.

Author(s): Vasil Bachvarov

Company: SAP

Created on: 05 October 2006

Version: 1.0.2

Author Bio

Vasil Bachvarov (vasil.bachvarov@sap.com)

has been working for SAP AG since January, 2005. He has developed demo applications and tests for the Composite Applications Framework project, Guided Procedures and does research in the area of scripting language integration with SAP software. Currently considered languages are PHP, Ruby, Perl and Perl.

Vasil is a graduate student at the Technical University of Darmstadt, Germany, major Informational and Communicational Technology. He has about 6 years experience in software development.

Abstract

SAP Scripting Tool is an innovative technology that enables scripting developers to rapidly integrate SAP service calls into their scripts. The product is based on the Eclipse platform and therefore inherits its native flexibility and extensibility. The main goals of SAP Scripting Tool are:

- Repository browsing for common SAP backend systems (R/3 BAPIs, R/3 web services, ESR, SAP-XI, etc.);
- Generation of scripting code in any scripting language (the variety of languages is limited only by what scripting generators are installed);
- Plug-in based extensibility enabling developers to write their own code generators for other languages or connectors for other backend systems;
- Support for concurrent backend service connectors for the same language (e.g. PHP);
- Achieving user friendly approach to download and use out of the box the tool as well as easy installation of new features or updates of the current ones. This is possible thanks to the Eclipse update mechanism;
- Integrated documentation, links and sample applications, consuming the generated code, installable per one click from the update site.

This article helps you to learn **creating custom code generators** for SAP Scripting Tool and **understand its extension API**. As example we will take the Python scripting language and we will write a Python code generator and integrate it into SAP Scripting Tool.

Target audience are scripting developers or any person with Java knowledge, willing to develop custom generators for the Tool.

Although Eclipse plug-in development knowledge is essential for understanding the concepts of extending SAP Scripting Tool, this article tries to make possible even for people that do not know Eclipse architecture in details to develop custom generators. Knowledge of XML is a prerequisite for understanding the text.

Table of Contents

Preparation

Before starting with the code generator development one must have an Eclipse 3.1 environment with JDT, PDE and EMF plug-ins (if you are confused just download the full Eclipse SDK package) and obtain the binaries of SAP Scripting Tool. Eclipse can be downloaded from its web site http://www.eclipse.org (please follow the standard Eclipse installation instructions) and SAP Scripting Tool is available on the SAP Developer Network site http://sdn.sap.com. For installation instructions, please refer to Frederic Ahring's article **SAP Scripting Tool: an Introductions How to Use It** (https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/docs/library/uuid/591c31cf-0d01-0010-8ab7-a1f7d032a66c).

Now the workspace should be ready to start developing the new scripting generator.

Basic Concept of Extending Eclipse and Particularly SAP Scripting Tool

If you have experience with Eclipse development you can skip the following two sections.

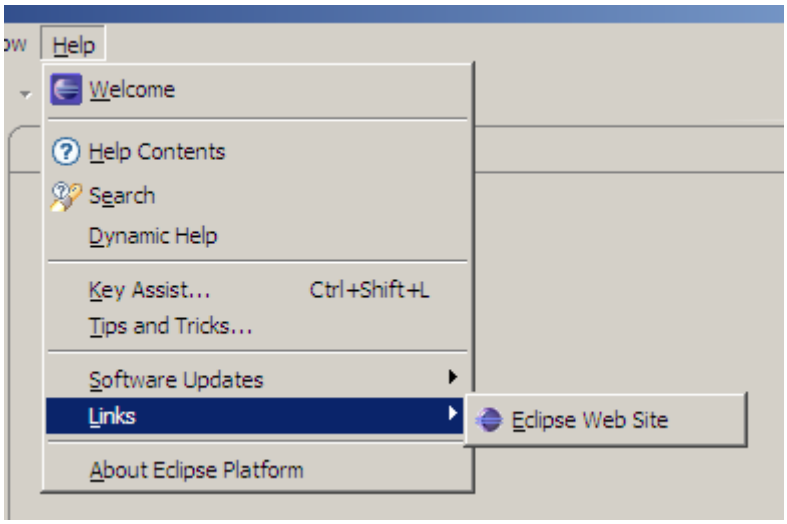
Eclipse Plug-In Mechanism

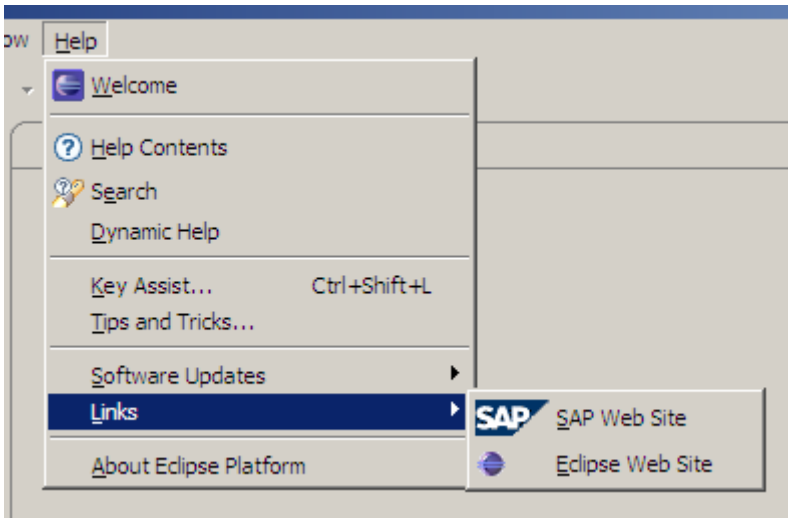
Every contribution to the SAP Scripting Tool and Eclipse itself is done in the form of plug-ins. A plug-in is simply a set of java classes and a plug-in descriptor, distributed to the files **plugin.xml** and **META-INF/Manifest.mf**. When Eclipse starts, it reads the contents of its **plugins** subdirectory and for each plug-in it loads the plug-in description file without loading the java classes that belong to the plug-in. The classes are loaded only when the plug-in functionality is requested, but for displaying its contribution in the environment loading is still not necessary. The contributions of the plug-in are described in the plug-in descriptor and loaded on start up. The approach is also known as lazy initialization. It enables the workbench to have potentially hundreds of active plug-ins and at the same time start in a reasonable time. Besides that one does not use all the plug-ins on each start of the workbench. Sometimes one uses C Development, sometimes - Java Development, sometimes - PHP Development.

Extension Points

Another aspect of Eclipse's architecture is the use of extension points. This is the mean of a plug-in to define how it can be extended by other plug-ins. Each of them contributes some extended functionality to the extension points of the considered plug-in. Here is a short example.

Let someone have developed a plug-in that adds the submenu **Links** to the **Help** menu of Eclipse and let **Links** has a single menu item – **Eclipse Web Site**. By clicking on this menu item the user opens a web browser and displays in it http://www.eclipse.org. The standard approach when defining such menus is to define an extension point for additional menu items there. If another developer decides to add another link there (for example **SAP Web Site**), the only thing he should do is to define an extension to this extension point. This is done in the **plugin.xml** of his plug-in.





In the extension definition he can specify the caption of the menu item and the address it points to. It is even not necessary to write Java code for opening the browser, since this task is accomplished by the first developer's Java code. It has a list with all plug-ins that contribute to this extension point and from their properties (caption and URI) it has all the necessary information.
Furthermore it is a standard approach for each menu to specify a special menu separator called **additions** so all plug-ins that want to insert additional menu items in a specific menu can do this in **additions**.

SAP Scripting Tool Extension Points

Now let us see how the extension point concept can be applied to the SAP Scripting Tool and how can it help us to develop easily custom scripting generators.

SAP Scripting Tool provides a number of extension points for extending its functionality with new backend service repository connectors and scripting generators. Thus it is fairly easy to make the tool generate scripts in any scripting language or connect to any backend system, provided that one writes the necessary plug-ins. These extension points are located in the **com.sap.scripting.core** plug-in:

languages – for describing which languages are present and what are the default extensions for script files in them;

generators – for implementing custom code generators;

connectorTypes – for specifying which types of backend systems can be connected to;

connectors – for custom backend connectors.

In this document we will concentrate on the first two ones.

Writing a Custom Scripting Generator

Creating the Python Language Plug-in

First we have to register the Python language in the SAP Scripting Tool because:

- The SAP Scripting Tool must be aware of what languages are present in the environment. Every scripting generator is coupled to a specific language.
- A set of language properties is common for every generator for this language (language name, default script file extension, etc). These properties are located in the specific scripting language plug-in of SAP Scripting Tool.

This task is straightforward and must be accomplished by creating the **com.sap.scripting.languages.python** plug-in.

*The latter is necessary only in case that the Python language is still not registered in SAP Scripting Tool (the Python language plug-in is not installed). If this is not the case you can simply skip this section and use the existing **com.sap.scripting.languages.python** plug-in.*

First create a new plug-in project in Eclipse (**File » New » Project » Plug-in Project**). Name it **Language.Python** and click **Next**. Provide the following details for the project: ID – **com.sap.scripting.languages.python**, version – **0.8.0**, name - **SAP Scripting Python definition**, provider - **www.sap.com**.

Here is how the **plugin.xml**, **META-INF/Manifest.mf** and **build.properties** look like:

plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<!--
   Plugin.xml for the Python language plug-in.
   Copyright 2006 SAP AG
-->
<plugin>
   <extension
         point="com.sap.scripting.core.languages">
      <language
            default_extension="py"
            display_name="Python"
            id="com.sap.scripting.languages.python"/>
   </extension>
   <extension
         point="org.eclipse.ui.preferencePages">
```



```
      </extension>
   </plugin>
```

META-INF/Manifest.mf

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: SAP Scripting Python definition
Bundle-SymbolicName: com.sap.scripting.languages.python; singleton:=true
Bundle-Version: 0.8.0
Bundle-Vendor: www.sap.com
Bundle-Localization: plugin
Require-Bundle: com.sap.scripting.core,
 org.eclipse.core.runtime,
 org.eclipse.ui,
 com.sap.scripting.common.ui
Export-Package: com.sap.scripting.languages.python.preferences
```

build.properties

```
bin.includes = META-INF/,\
               plugin.xml
```

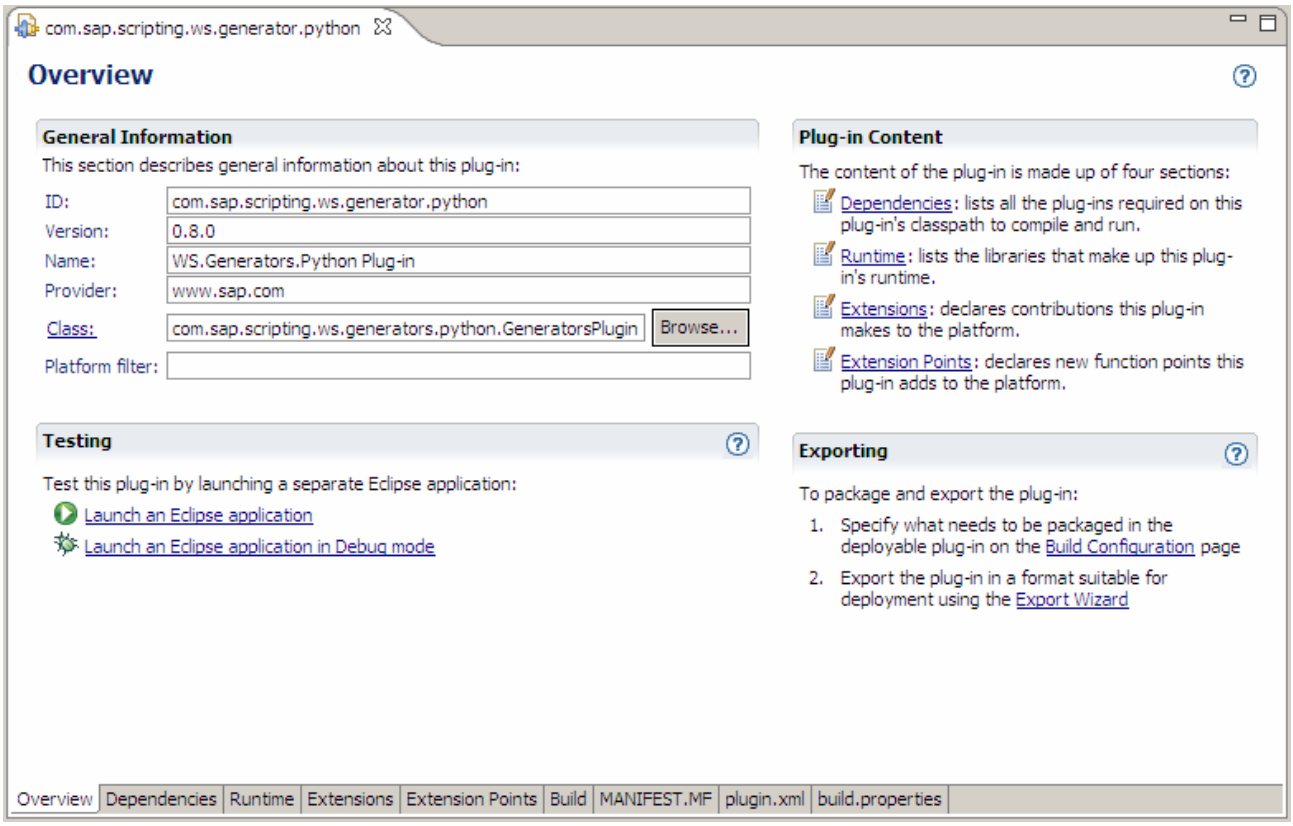
Regarding the preference page declarations and the supplementary Java classes in the **Language.Python** sample project, preference pages (displayed in the Eclipse Preferences window) are not obligatory or relevant to the generator and will not be discussed here. You can remove them or create preference pages. If you would like to learn more about Eclipse preference pages, please consult the Java Developer's Guide to Eclipse or other reference.

Creating the Plug-in Project for the Generator and Basic Plug-in Settings

Now we are ready to start with the development of the custom scripting generator. From the **File** menu select **New » Project** and then select **Plug-in Project** from the **Plug-in Development** category **» Next**. Give a project name (for this example we use **WS.Generators.Python**) and leave the other options without changes **» Next**. Provide the plug-in details **ID**, **Version**, **Name**, **Provider** (ID is **com.sap.scripting.ws.generator.python**, version is **0.8.0**, description is **WS.Generators.Python Plug-in** and the provider is **www.sap.com**) **» Finish**.

Open the **Runtime** tab and click on the **Add…** button. Type in **com.sap.scripting.ws.generators.python** and click **OK**. This will export this package for access from other clients.

You should be able to see a screen like the shown on the next figure. The only difference should be the name of the plug-in class and the lack of **plugin.xml** tab.



First we will do some refactoring, so that the new plug-in complies the SAP Scripting Tool naming convention. Open the project folder in Eclipse and rename the package of the plug-in to **com.sap.scripting.ws.generators.python**. Rename the plug-in class to **GeneratorsPlugin**.

At this point we have to convert the project to a JET (Java Emitter Templates) project. Right-click the project and from the menu select **New » Other…** and then from the category **Java Emitter Templates** select **Convert Projects to JET Projects**. Then check only the combo box of our **WS.Generators.Python** project and click on **Finish**. This will add the **JET** nature to our project, which will allow Eclipse to automatically generate the necessary java source from our python file template and we do not have to care much about writing the Java code for the python files generation. We have to set up the template and the source folder for the JET files. Right click the project and select **Properties**. In the property page **JET Settings** enter **templates** in the **Template Containers** box and **src** in the **Source Container** box. Now every template file that we save in the **templates** subfolder of the project will cause automatic generation of the Java source code which generates files from this template.

Now we are ready to start writing the source code of the generator.

Extending the Generators Extension Point

It is necessary to extend the **generators** extension point in order to display our generator in the available generators list in the tool. For this purpose we have to edit **plugin.xml** and add the declarations for the extension. Here is how our **plugin.xml** should look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<!--
   Plugin.xml for the Python scripting generator.
   Author: Vasil Bachvarov
```



```
    Date: 20 Sept. 2006
    Copyright 2006 SAP AG
-->
<plugin>
   <extension
         point="com.sap.scripting.core.generators">
      <generator
            language="com.sap.scripting.languages.python"
            class="com.sap.scripting.ws.generators.python.Generator"
            id="com.sap.scripting.ws.generators.python.Generator"
            name="SAP Web Service for Python Generator">
         <connector
id="com.sap.scripting.ws.connector.connectorImp.WebServiceConnector"/>
         <source_object>
            <module/>
         </source_object>
         <ui_providers>
            <ui_provider
class="com.sap.scripting.common.ui.fileoutput.DefaultFileNameUIProvider"/>
         </ui_providers>
      </generator>
   </extension>
</plugin>
```

Now let us describe the elements in this XML file. The **extension** element defines the extension point that we extend: **com.sap.scripting.core.generators**, which was already discussed. The **generator** element defines the properties of our generator: language; class; id; name. The **connector** element shows that our generator will generate code for connecting to web services. The **ui_provider** element defines the default generator UI provider, which provides the file name selection dialog. UI providers are used to enable a generator to provide a custom user interface. This may be necessary for a generic scripting generator, for we would like to achieve maximal flexibility and interoperability with wide variety of scripting languages and technologies. For example a scripting generator may want to display a page with log-in information, in which the log-in data is not user name and password, but bank branch code, account number and PIN number.

The file **META-INF/Manifest.mf** contains the rest of the plug-in description:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: WS.Generators.Python Plug-in
Bundle-SymbolicName: com.sap.scripting.ws.generator.python; singleton:=true
Bundle-Version: 0.8.0
Bundle-ClassPath: .
Bundle-Activator: com.sap.scripting.ws.generators.python.GeneratorsPlugin
Bundle-Localization: plugin
```



```
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime,
 com.sap.scripting.common.ui,
 com.sap.scripting.core,
 com.sap.scripting.metadata,
 com.sap.scripting.ws.connector,
 com.sap.scripting.languages.python
Export-Package: com.sap.scripting.ws.generators.python
Bundle-Vendor: www.sap.com
```

After creating **plugin.xml** and **Manifest.mf** we have to create the class with the generator **com.sap.scripting.ws.generators.python.Generator** and its supplementary classes.

Creating the JET Template

The JET template contains the layout of a generated file. It has some special tags (similar to the server-side tags in PHP or JSP) which contain Java code for inserting dynamic content. This code will call some functions of the **Generator** class, which will be implemented in a bit.

Here is how the Python JET template looks like:

Please, note that some lines have been broken. Python is a language that is line driven and new lines do matter. In order to have a correct view of the code you can copy and paste the source in another text editor or look at the supplementing sample archive.

```
<%@ jet package="com.sap.scripting.ws.generators.python.templates"
imports="com.sap.scripting.ws.generators.python.* java.util.Date
java.text.SimpleDateFormat" class="CodeTemplate" %><%
  Data data = (Data)argument;
    SimpleDateFormat dateFormatter= new SimpleDateFormat("dd.MM.yyyy
HH:mm");
%>
#  Stub functions for <%=data.getServiceName()%>: <%=data.getMethod()%>
#  generated by SAP Scripting Tool for Eclipse: Web Service calls generator
for Python.
#  Generation time: <%= dateFormatter.format(new Date()) %>
#
#  Data source: Web Service
#               SOAP entry at <%=data.getPortLocation()%>
#  Connector:   Python 2.4.2-SOAPpy (http://pywebsvcs.sourceforge.net/)
#  Language:    Python 2.4.2

from SOAPpy import SOAPProxy

#  <%=data.getMethod()%>
#  <%= data.getMethodDescription().replaceAll("\n", "\n#  ") %>
```



```
def
<%=data.getServiceName()%>_<%=data.getMethod()%>(<%=data.getInputParametersP
ython()%>):
  proxy_<%=data.getServiceName()%> = SOAPProxy('<%= data.getPortLocation()
%>', '<%= data.getNamespace() %>')
  result = proxy_<%=data.getServiceName()%>.<%= data.getMethod()
%>(<%=data.getInputParametersWS()%>)
  return result
```

The template contains a comment box and the generated method. The name of the method and the parameters are defined at runtime from the SAP Scripting Tool classes and the **Generator** class that will be created in the next sections.

The JET syntax is very similar to JSP tags. For more information refer to the JET documentation.

Our JET template will generate Python files that access web services via the SOAPpy extension of Python.

Creating the Java Classes for the Generator

In the package **com.sap.scripting.ws.generators.python** we will create first a class, called **Data**. It encapsulates the properties for connecting to the web service. These properties are populated from the metadata that is passed to the generator.

Create a new class **Data** in **com.sap.scripting.ws.generators.python**. Create the following private properties:

```
public class Data {

    // WSDL URL
  private String url = null;

  // Service Name
    private String serviceName = null;

    // Method Name
    private String method = null;

    // Method Description - it will be put into the generated comments
    private String methodDescription = null;

    // Input Parameters (for the python call)
    private String inputParametersPython = null;

    // Input Parameters (for the WS call)
    private String inputParametersWS = null;

    // Port Location (SOAP Port)
```



```
        private String portLocation = null;

        // Name Space
        private String namespace = null;

        // Target Name Space
        private String targetNamespace = null;
    }
```

Then use the built-in Eclipse function for generating getters and setters for the properties. (The full source can be found in **Appendix A**).

The **Data** class is ready.

Now in the same package create a class, called **GeneratorsPlugin**. This is the Eclipse plug-in class for the Python generator. Every Eclipse plug-in has such a class.

Here is how the plug-in class looks like:

```
package com.sap.scripting.ws.generators.python;

import org.eclipse.ui.plugin.*;
import org.eclipse.jface.resource.ImageDescriptor;
import org.osgi.framework.BundleContext;

/**
 * The main plugin class to be used in the desktop.
 */
public class GeneratorsPlugin extends AbstractUIPlugin {

  //The shared instance.
  private static GeneratorsPlugin plugin;

  /**
   * The constructor.
   */
  public GeneratorsPlugin() {
        plugin = this;
  }

  /**
   * This method is called upon plug-in activation
   */
```



```
    public void start(BundleContext context) throws Exception {
        super.start(context);
    }

    /**
     * This method is called when the plug-in is stopped
     */
    public void stop(BundleContext context) throws Exception {
        super.stop(context);
        plugin = null;
    }

    /**
     * Returns the shared instance.
     */
    public static GeneratorsPlugin getDefault() {
        return plugin;
    }

    /**
     * Returns an image descriptor for the image file at the given
     * plug-in relative path.
     *
     * @param path the path
     * @return the image descriptor
     */
    public static ImageDescriptor getImageDescriptor(String path) {
        return
AbstractUIPlugin.imageDescriptorFromPlugin("WS.Generators.Python", path);
    }
}
```

An Eclipse plug-in class inherits **AbstractUIPlugin**, which contains the necessary functionality. The methods are standard for a plug-in class. For more information, please refer to Eclipse Developer's Guide or other Eclipse development reference.

Customizing the Generator UI

We do not have to implement a generator UI provider, for our generator will be fairly simple. It can use the default UI provider for the File Name and Destination dialog. In a more sophisticated generator one may want to embed additional pages of configuration parameters like security, etc. In this case one must implement the **com.sap.scripting.core.generators.IGeneratorUIProvider** interface, add the declaration in the **plugin.xml** of the generator plug-in (it is exactly the same as for the default UI provider) and fill the SWT

composite object with the contents of the page. For more information on this topic, please refer to the SWT developer's guide.

Writing the Script Generation Code

Now we are almost ready with the generator. The only thing to be done is to create the **Generator** class and write the code that generates scripting code.

Create a class **Generator** in the **com.sap.scripting.ws.generators.python** package. Let it implement the **com.sap.scripting.core.generators.IGenerator** interface.

Add the private properties for storing intermediate results:

```
public class Generator implements IGenerator {
  private StringBuffer inputParametersPython = null;
  private StringBuffer inputParametersWS = null;
  private Data webServiceData = null;
  private int paramCounter;
…
```

The constructor remains empty. But it may content some generator-specific initializations.

Create a method **generateCode()**:

```
public boolean generateCode(IGeneratorContext context) {
…
        // The file name for the destination file.
        String scriptFileName =
context.getOption(IGeneratorOptions.DESTINATION_FILE_NAME);
        // Is the destination folder a file system folder or an Eclipse
resource.
        int kind =
Integer.valueOf(context.getOption(IGeneratorOptions.DESTINATION_FOLDER_KIND)
).intValue();
        String path =
context.getOption(IGeneratorOptions.DESTINATION_FOLDER);


        // Initialize the output objects.
        IGeneratorOutput generatorOutput = context.getOutput();
        generatorOutput.clear();


        IOutputRoot outputRoot = generatorOutput.addRoot(scriptFileName);
        outputRoot.setKind(kind);
        outputRoot.setPath(path);


        IOutputFile outputFile = outputRoot.addFile(scriptFileName);
…
```


Let us take a closer look at this code fragment. First we get the destination file name. Then a **generatorOutput** object is created, which provides a generator with the ability to abstractly describe logical structure of its output. A generator can generate multiple files in different folders. Then we set the script file name as a root of this generator output and its kind and path. There are two kinds of outputs: **FILE_SYSTEM** and **WORKSPACE_CONTAINER**. Both constants are located in the **com.sap.scripting.core.generators.output.IOutputRootKind** interface. An output file object is created.

The next step is to create and populate a web service data object, where we can store the web service metadata and use it by the generation. The next long code fragment does exactly this and proves in the beginning that the selected metadata object in the context is **MiModule**, which is the only one supported by this generator.

```
          // Create a web service data object to copy the metadata there
(make it handier).
        webServiceData = new Data();


          /* If in the context is selected something else from MiModule
output an error message
         * for unsupported type.
         */
        if(!(context.getSelectedObject() instanceof MiModule)){
         outputFile.setContent(
                    "This generator doesn't support code generation for
object of this type: " +
                    context.getSelectedObject().getClass().toString());
         return true;


        }


        // Get the metadata object.
        MiModule miModule = (MiModule) context.getSelectedObject();


        // Set the Data object properties from the metadata.

webServiceData.setURL(miModule.getAttribute(IWSParams.URL).getValue());
        webServiceData.setMethod(miModule.
                getAttribute(IWSParams.METHOD).getValue());


        MiAttribute serviceNameAttr = miModule
                .getAttribute(IWSAttributes.LOCAL_SERVICE_NAME);
        if (serviceNameAttr != null) {
            webServiceData.setServiceName(serviceNameAttr.getValue());
        }
```



```
            MiElement root = miModule.getInput();
            String methodDescription = (miModule
                    .hasAttribute(IWSAttributes.DESCRIPTION)) ? miModule
                    .getAttribute(IWSAttributes.DESCRIPTION).getValue() : "";

            // Prepare the parameter strings.
            inputParametersPython = new StringBuffer();
            inputParametersWS = new StringBuffer();
            paramCounter = 0;

            readParameters(root, "", 0);


    webServiceData.setInputParametersPython(inputParametersPython.toString());
            webServiceData.setInputParametersWS(inputParametersWS.toString());
            webServiceData.setMethodDescription(methodDescription);

    webServiceData.setPortLocation(miModule.getAttribute(IWSParams.PORT_LOCATION
    ).getValue());

    webServiceData.setNamespace(miModule.getAttribute(IWSParams.NAMESPACE).getVa
    lue());
```

In general taking different attributes of the metadata object **miModule** we populate the web service data object. The only thing that should be commented is the call of the method **readParameters()**. It will walk through the structure with the web service input parameters and generate the strings, containing valid Python parameter lists. These strings will be embedded in the generated scripting code.

```
            // Generate the code with the template.
            CodeTemplate codeTemplate = new CodeTemplate();
            String scriptText = codeTemplate.generate(webServiceData);
            outputFile.setContent(scriptText);
```

The last three lines create the JET template object and let it generate scripting code with the web service data, holding the necessary properties.

```
            return true;
        }
```

The last thing to be done is the **readParameters()** method:

```
        /**
         * Create the strings with parameters for the script calls.
         *
         * @param _node
         * @param _prefix
```



```
     * @param _paramLevel
     */
    private void readParameters(MiElement _node, String _prefix, int
_paramLevel) {
        String paramNumber;
        String name = WSAttributesUtil.getLocalName(_node);
        String paramName = "param_" + name + "_";

        MiType elementType = _node.getType();

        // Check the type of node with the input from the metadata.
        if (elementType instanceof MiSimpleType) {
            // If the input is a simple type - write a single parameter to
the string.
         paramNumber = String.valueOf(paramCounter);
            paramCounter++;
            inputParametersWS.append(name + "=" + paramName + paramNumber);
            inputParametersPython.append(paramName + paramNumber);
        } else if (elementType instanceof MiComplexType) {
         // If the input is a complex type - walk recursively and create
structure in the output string.
         // NOTE: There is a limitation for this generator. It supports only
flat structures.
         EList elementList = ((MiComplexType) elementType).getElementList();

            for (int i = 0; i < elementList.size(); i++) {
                MiElement node = (MiElement) elementList.get(i);
                readParameters(node, _prefix + "." + name, _paramLevel + 1);

                if (i + 1 < elementList.size()) {
                    inputParametersWS.append(", ");
                    inputParametersPython.append(", ");
                }
            }
        }
    }
}
```

We start processing the web service data from the root element. In general it is a tree structure and the method is applied recursively.

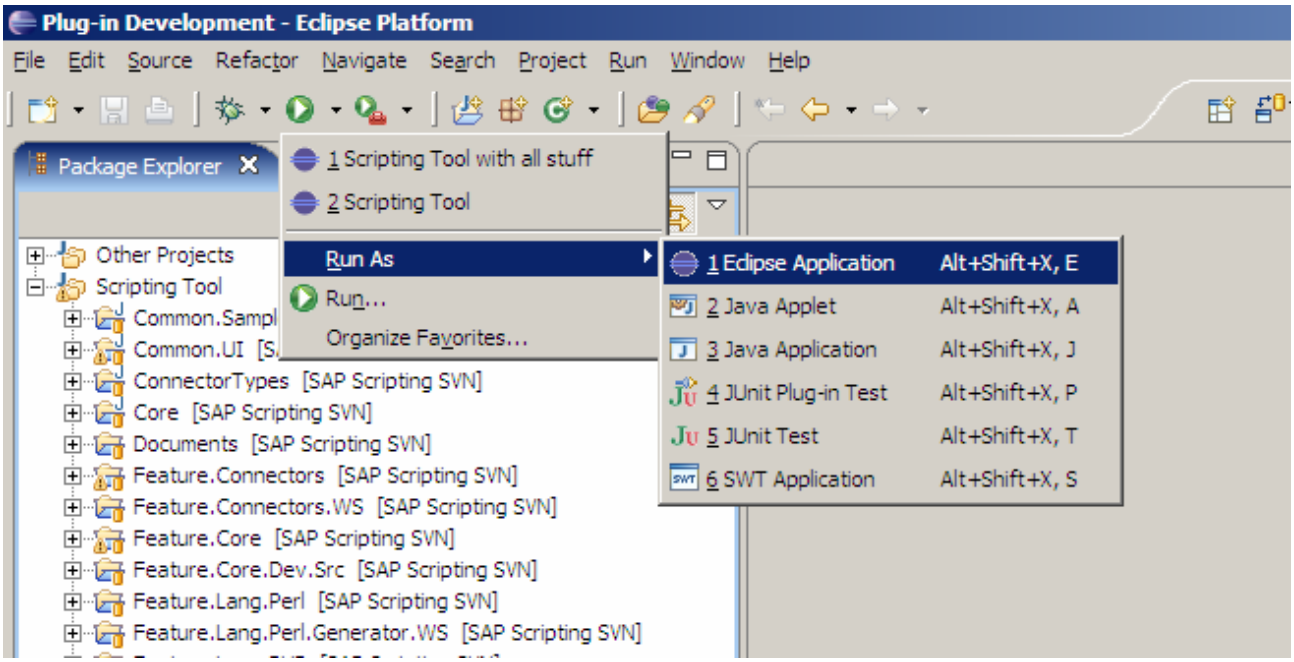
If the current element is of a simple type, then output to the parameter string **param_name=<param_name_number>**, which will generate a string like „**employeeName=param_employeeName_1**“. This is done so we distinguish between **employeeName**, which is the name of the web service parameter, and **param_employeeName_1**, which is the Python variable we use to store the parameter. Parameter names are not absolutely collision free in this sample, but if desired, the generator must take care of it.

In case that the current element is of a complex type (composite object) we first get the list with its elements and for each element we call recursively the same method. In this way we walk through the tree structure and create a flat list with all parameters. This implies that our Python generator supports only flat structures of the input parameters (a list). If nested structure must be considered, one have to use more sophisticated processing procedure. Such algorithms are outside the scope of the article, which main focus is to explain the technology of creating a custom scripting generator.

The Python generator is ready. Complete source code can be found in Appendices A and B, as well as in the two supplementing sample projects.

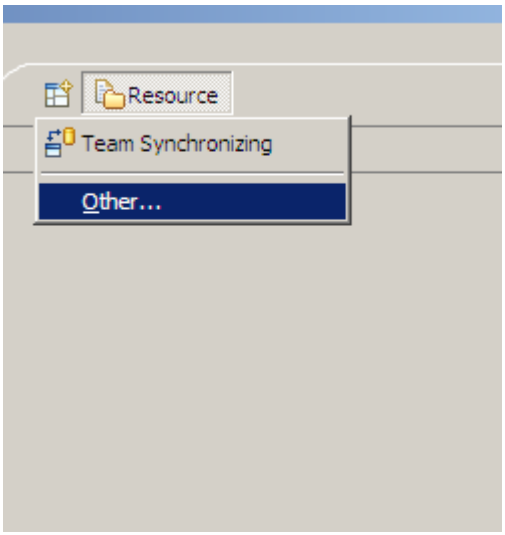
Testing the Python Generator

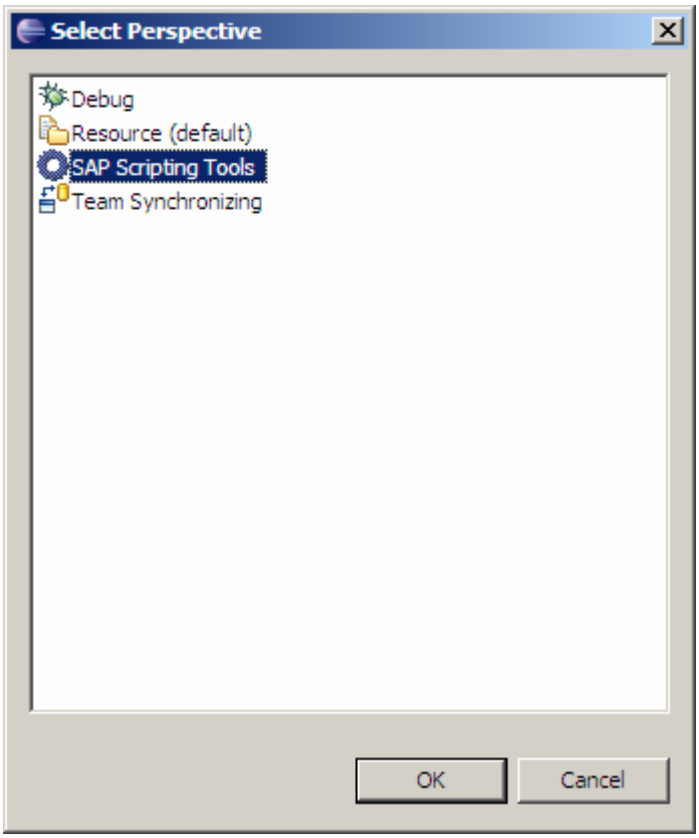
Testing the generator is a very simple task. First you have to open all projects from the SAP Scripting Tool source code **and** your new **WS.Generators.Python** project. Then select from the toolbar or from the **Run** menu **Run As… » Eclipse Application**.



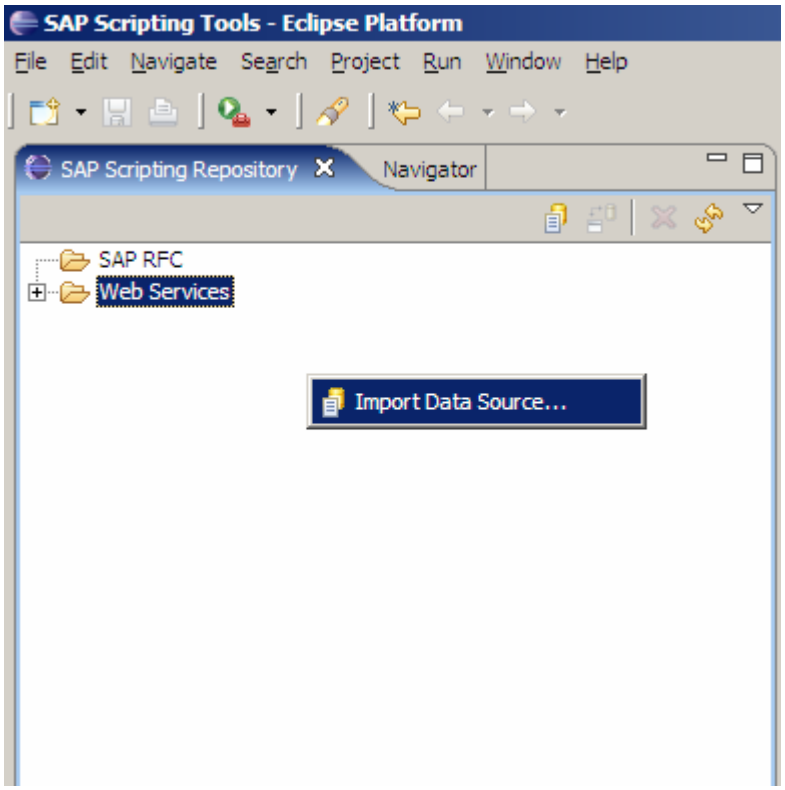
This will start a copy of Eclipse with all active plug-ins and the runtimes of all plug-in projects.

Then open the **SAP Scripting Tools** perspective.

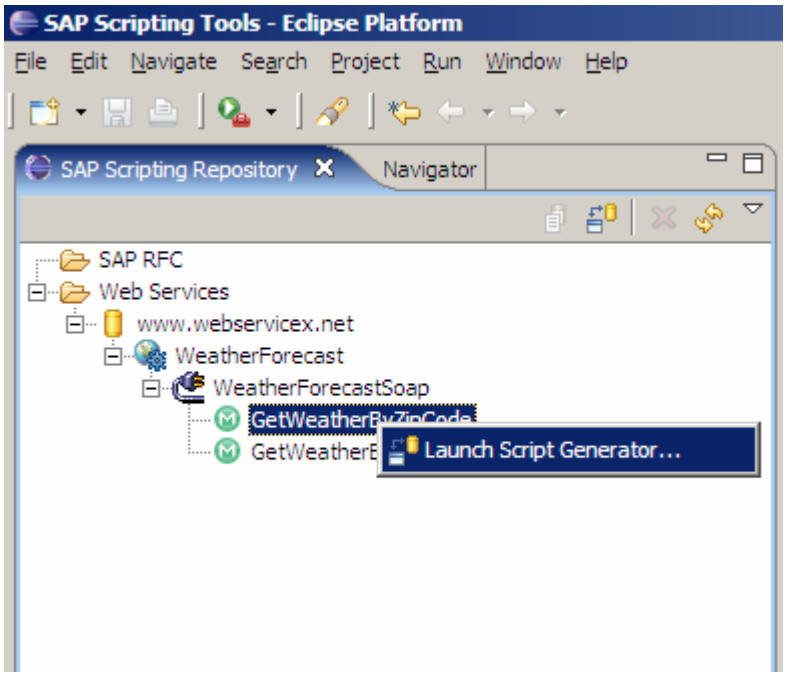


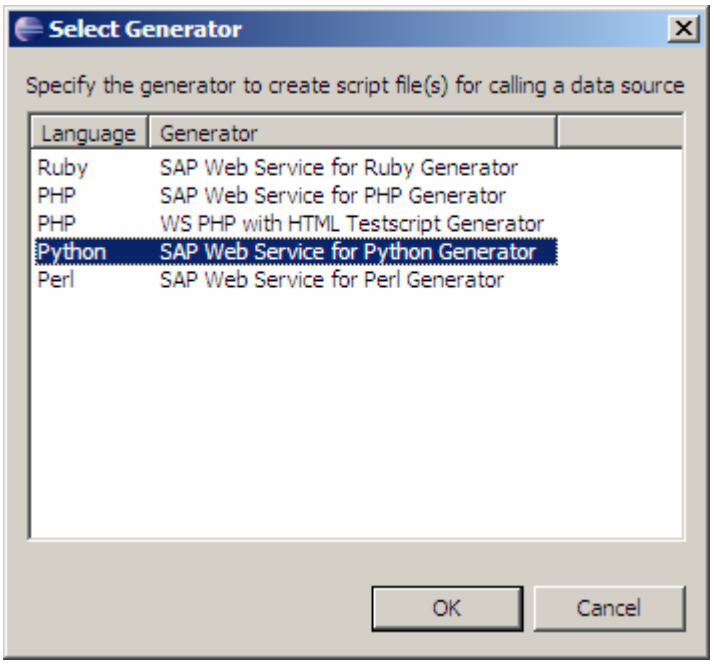


In the **SAP Scripting Repository** view import a backend system.



When the backend system is added to the repository its metadata is saved locally.
Now launch the Python scripting generator you created:





The rest is pretty straightforward. If everything passed fine the new generated Python file should be loaded in an Eclipse editor.

Packaging

The new generator can be exported for other users from **File » Export » Deployable Plug-ins and Fragments**.

Conclusion

In this article we have considered creating a custom scripting generator for the SAP Scripting Tool. Our sample generator was designed to generate Python scripts for accessing web services with the SOAPpy web service connector for Python.

In the beginning we have considered shortly the Eclipse architecture and how plug-ins and extension points are used in the SAP Scripting Tool.

Our main task has been accomplished in a couple of steps:

- Creating the **Language.Python** project that registers the Python language in the SAP Scripting Tool.
- Creating the Python generator project and basic plug-in settings.
- JET Template for the generated files.
- Implementing the code that generates the Python strings, customizing the JET template.

These concepts are applicable to any scripting or other language and any backend connector like web service or BAPI connector.

Appendix A. Python Generator Source Code

plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<!--
  Plugin.xml for the Python scripting generator.
  Author: Vasil Bachvarov
  Date: 20 Sept. 2006
  Copyright 2006 SAP AG
-->
<plugin>
   <extension
         point="com.sap.scripting.core.generators">
      <generator
            language="com.sap.scripting.languages.python"
            class="com.sap.scripting.ws.generators.python.Generator"
            id="com.sap.scripting.ws.generators.python.Generator"
            name="SAP Web Service for Python Generator">
         <connector
id="com.sap.scripting.ws.connector.connectorImp.WebServiceConnector"/>
         <source_object>
            <module/>
         </source_object>
         <ui_providers>
            <ui_provider
class="com.sap.scripting.common.ui.fileoutput.DefaultFileNameUIProvider"/>
         </ui_providers>
      </generator>
   </extension>
</plugin>
```

META-INF/Manifest.mf

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: WS.Generators.Python Plug-in
Bundle-SymbolicName: com.sap.scripting.ws.generator.python; singleton:=true
Bundle-Version: 0.8.0
Bundle-ClassPath: .
```



```
Bundle-Activator: com.sap.scripting.ws.generators.python.GeneratorsPlugin
Bundle-Localization: plugin
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime,
 com.sap.scripting.common.ui,
 com.sap.scripting.core,
 com.sap.scripting.metadata,
 com.sap.scripting.ws.connector,
 com.sap.scripting.languages.python
Export-Package: com.sap.scripting.ws.generators.python
Bundle-Vendor: www.sap.com
```

build.properties

```
source.. = src/
output.. = bin/
bin.includes = META-INF/,\
               .,\
               plugin.xml
```

com.sap.scripting.ws.generators.python.Data

```
/**
 * <copyright>
 * Copyright 2006 SAP AG
 * </copyright>
 */

package com.sap.scripting.ws.generators.python;

/**
 * <!-- begin-user-doc -->
 *
 * <b>Data</b>
 *
 * Data capsulation class for the Python scripting generator.
 * Encapsulates the properties URL (WSDL), Service Name, Method Name, Method
Description,
 * Input Parameters (for python call), Input Parameters (for WS call), Port
Location (SOAP Port),
 * Name Space, Target Name Space.
 *
 * <!-- end-user-doc -->
```



```
 *
 * @author Vasil Bachvarov
 */
public class Data {

    // WSDL URL
  private String url = null;

  // Service Name
    private String serviceName = null;

    // Method Name
    private String method = null;

    // Method Description - it will be put into the generated comments
    private String methodDescription = null;

    // Input Parameters (for the python call)
    private String inputParametersPython = null;

    // Input Parameters (for the WS call)
    private String inputParametersWS = null;

    // Port Location (SOAP Port)
    private String portLocation = null;

    // Name Space
    private String namespace = null;

    // Target Name Space
    private String targetNamespace = null;

    /**
     * @return Returns the inputParametersPython property.
     */
    public String getInputParametersPython() {
        return inputParametersPython;
    }
```



```
    /**
     * Python procedure parameters
     *
     * @param _inputParametersPython The inputParametersPython property to
set.
     */
    public void setInputParametersPython(String _inputParametersPython) {
        inputParametersPython = _inputParametersPython;
    }

    /**
     * Web service parameters
     *
     * @return Returns the inputParametersWS.
     */
    public String getInputParametersWS() {
        return inputParametersWS;
    }

    /**
     * @param _inputParametersWS The inputParametersWS to set.
     */
    public void setInputParametersWS(String _inputParametersWS) {
        inputParametersWS = _inputParametersWS;
    }

    /**
     * @return Returns the name of web service.
     */
    public String getServiceName() {
        return serviceName;
    }

    /**
     * Web service name
     *
     * @param value The name of web service to set.
     */
    public void setServiceName(String value) {
```



```
        serviceName = value;
    }

    /**
     * @return Returns the method.
     */
    public String getMethod() {
        return method;
    }

    /**
     * Web service method
     *
     * @param _method The method to set.
     */
    public void setMethod(String _method) {
        method = _method;
    }

    /**
     * @return Returns the methodDescription property.
     */
    public String getMethodDescription() {
        return methodDescription;
    }

    /**
     * Web service method description
     *
     * @param _methodDescription The methodDescription property to set.
     */
    public void setMethodDescription(String _methodDescription) {
        methodDescription = _methodDescription;
    }

    /**
     * @return Returns the WSDL URL.
     */
    public String getURL() {
```



```
        return url;
    }

    /**
     * URL for WSDL file
     *
     * @param _url The WSDL URL to set.
     */
    public void setURL(String _url) {
        url = _url;
    }

    /**
     * @return Returns the namespace.
     */
    public String getNamespace() {
        return namespace;
    }

    /**
     * Namespace in the WSDL
     *
     * @param namespace The namespace to set.
     */
    public void setNamespace(String namespace) {
        this.namespace = namespace;
    }

    /**
     * @return Returns the SOAP entry point.
     */
    public String getPortLocation() {
        return portLocation;
    }

    /**
     * The SOAP entry point
     *
     * @param portLocation The port location to set.
```



```
        */
      public void setPortLocation(String portLocation) {
            this.portLocation = portLocation;
      }

      /**
         * @return Returns the target name space in WSDL.
         */
      public String getTargetNamespace() {
            return targetNamespace;
      }

      /**
         * The target name space
         *
         * @param targetNamespace The target name space in the WSDL.
         */
      public void setTargetNamespace(String targetNamespace) {
            this.targetNamespace = targetNamespace;
      }

   }
```

com.sap.scripting.ws.generators.python.Generator

```
   /**
    * <copyright>
    * Copyright 2006 SAP AG
    * </copyright>
    */

   package com.sap.scripting.ws.generators.python;

   import org.eclipse.emf.common.util.EList;

   import com.sap.scripting.core.generators.IGenerator;
   import com.sap.scripting.core.generators.IGeneratorContext;
   import com.sap.scripting.core.generators.IGeneratorOptions;
   import com.sap.scripting.core.generators.output.IGeneratorOutput;
   import com.sap.scripting.core.generators.output.IOutputFile;
```



```
import com.sap.scripting.core.generators.output.IOutputRoot;
import com.sap.scripting.metadata.MiAttribute;
import com.sap.scripting.metadata.MiComplexType;
import com.sap.scripting.metadata.MiElement;
import com.sap.scripting.metadata.MiModule;
import com.sap.scripting.metadata.MiSimpleType;
import com.sap.scripting.metadata.MiType;
import com.sap.scripting.ws.connector.wsdl.IWSAttributes;
import com.sap.scripting.ws.connector.wsdl.IWSParams;
import com.sap.scripting.ws.connector.wsdl.WSAttributesUtil;
import com.sap.scripting.ws.generators.python.templates.CodeTemplate;


/**
 *
 * <!-- begin-user-doc -->
 *
 * <b>Generator</b>
 *
 * Main generator class for the Python scripting generator. It uses the Data
class to store
 * its parameter properties. Use the method <b>generateCode()</b> to
generate a Python script
 * from the template.
 *
 * <!-- end-user-doc -->
 *
 * @author Vasil Bachvarov
 *
 */
public class Generator implements IGenerator {


  // Private properties for storing intermediate results.
  private StringBuffer inputParametersPython = null;
    private StringBuffer inputParametersWS = null;
    private Data webServiceData = null;
    private int paramCounter;

    /**
     * Constructor
```



```
     */
    public Generator() {
   }


    /**
     * Method to generate the script. It takes the web service metadata and
from it generates
     * python script for accessing the service.
     *
     * @param context - Generator context.
     * @return TRUE on success.
     */
    public boolean generateCode(IGeneratorContext context) {
        // The file name for the destination file.
        String scriptFileName =
context.getOption(IGeneratorOptions.DESTINATION_FILE_NAME);


        /* This can be used to generate automatically a test script. Notice:
Before that
         * you should create a JET template for the test file.
         */
        // String scriptTestFileName =
context.getOption(IGeneratorOptions.DESTINATION_TEST_FILE_NAME);


        // Is the destination folder a file system folder or an Eclipse
resource.
        int kind =
Integer.valueOf(context.getOption(IGeneratorOptions.DESTINATION_FOLDER_KIND)
).intValue();
        String path =
context.getOption(IGeneratorOptions.DESTINATION_FOLDER);


        // Initialize the output objects.
        IGeneratorOutput generatorOutput = context.getOutput();
        generatorOutput.clear();


        IOutputRoot outputRoot = generatorOutput.addRoot(scriptFileName);
        outputRoot.setKind(kind);
        outputRoot.setPath(path);


        IOutputFile outputFile = outputRoot.addFile(scriptFileName);
```



```
        // IOutputFile outputTestFile =
outputRoot.addFile(scriptTestFileName);


        // Create a web service data object to copy the metadata there (make
it handier).
        webServiceData = new Data();


        /* If in the context is selected something else from MiModule output
an error message
         * for unsupported type.
         */
        if(!(context.getSelectedObject() instanceof MiModule)){
        outputFile.setContent(
                    "This generator doesn't support code generation for
object of this type: " +
                    context.getSelectedObject().getClass().toString());
        return true;


        }


        // Get the metadata object.
        MiModule miModule = (MiModule) context.getSelectedObject();


        // Set the Data object properties from the metadata.

webServiceData.setURL(miModule.getAttribute(IWSParams.URL).getValue());
        webServiceData.setMethod(miModule.
                getAttribute(IWSParams.METHOD).getValue());


        MiAttribute serviceNameAttr = miModule
                .getAttribute(IWSAttributes.LOCAL_SERVICE_NAME);
        if (serviceNameAttr != null) {
            webServiceData.setServiceName(serviceNameAttr.getValue());
        }


        MiElement root = miModule.getInput();
        String methodDescription = (miModule
                .hasAttribute(IWSAttributes.DESCRIPTION)) ? miModule
                .getAttribute(IWSAttributes.DESCRIPTION).getValue() : "";
```



```
        // Prepare the parameter strings.
        inputParametersPython = new StringBuffer();
        inputParametersWS = new StringBuffer();
        paramCounter = 0;

        readParameters(root, "", 0);


webServiceData.setInputParametersPython(inputParametersPython.toString());
        webServiceData.setInputParametersWS(inputParametersWS.toString());
        webServiceData.setMethodDescription(methodDescription);

webServiceData.setPortLocation(miModule.getAttribute(IWSParams.PORT_LOCATION
).getValue());

webServiceData.setNamespace(miModule.getAttribute(IWSParams.NAMESPACE).getVa
lue());


        // Generate the code with the template.
        CodeTemplate codeTemplate = new CodeTemplate();
        String scriptText = codeTemplate.generate(webServiceData);
        outputFile.setContent(scriptText);


        /* You can use these lines to generate test script.
         */
        // TestTemplate testTemplate = new TestTemplate();
        // String testScriptText = testTemplate.generate(webServiceData);
        // outputTestFile.setContent(testScriptText);


        /* You can execute the Python interpreter with the generated script
to ensure that
         * it has been properly generated or just to test it.
         */
        // StringBuffer pythonCommand = new
StringBuffer(UIPlugin.getDefault()
        //
.getPreferenceStore().getString(IPreferencesConstants.EXECUTABLE_PATH));
        // pythonCommand.append(" ");


        return true;
    }
```



```
    /**
     * Create the strings with parameters for the script calls.
     *
     * @param _node
     * @param _prefix
     * @param _paramLevel
     */
    private void readParameters(MiElement _node, String _prefix, int
_paramLevel) {
        String paramNumber;
        String name = WSAttributesUtil.getLocalName(_node);
        String paramName = "param_" + name + "_";

        MiType elementType = _node.getType();

        // Check the type of node with the input from the metadata.
        if (elementType instanceof MiSimpleType) {
            // If the input is a simple type - write a single parameter to
the string.
         paramNumber = String.valueOf(paramCounter);
            paramCounter++;
            inputParametersWS.append(name + "=" + paramName + paramNumber);
            inputParametersPython.append(paramName + paramNumber);
        } else if (elementType instanceof MiComplexType) {
        // If the input is a complex type - walk recursively and create
structure in the output string.
        // NOTE: There is a limitation for this generator. It supports only
flat structures.
        EList elementList = ((MiComplexType) elementType).getElementList();

            for (int i = 0; i < elementList.size(); i++) {
                MiElement node = (MiElement) elementList.get(i);
                readParameters(node, _prefix + "." + name, _paramLevel + 1);

                if (i + 1 < elementList.size()) {
                    inputParametersWS.append(", ");
                    inputParametersPython.append(", ");
                }
```



```
            }
        }
    }
}
```

com.sap.scripting.ws.generators.python.GeneratorsPlugin

```
/**
 * <copyright>
 * Copyright 2006 SAP AG
 * </copyright>
 */

package com.sap.scripting.ws.generators.python;

import org.eclipse.ui.plugin.*;
import org.eclipse.jface.resource.ImageDescriptor;
import org.osgi.framework.BundleContext;

/**
 * The main plugin class to be used in the desktop.
 */
public class GeneratorsPlugin extends AbstractUIPlugin {

  //The shared instance.
  private static GeneratorsPlugin plugin;

  /**
   * The constructor.
   */
  public GeneratorsPlugin() {
        plugin = this;
  }

  /**
   * This method is called upon plug-in activation
   */
  public void start(BundleContext context) throws Exception {
        super.start(context);
  }
```



```
    /**
     * This method is called when the plug-in is stopped
     */
    public void stop(BundleContext context) throws Exception {
          super.stop(context);
          plugin = null;
    }

    /**
     * Returns the shared instance.
     */
    public static GeneratorsPlugin getDefault() {
          return plugin;
    }

    /**
     * Returns an image descriptor for the image file at the given
     * plug-in relative path.
     *
     * @param path the path
     * @return the image descriptor
     */
    public static ImageDescriptor getImageDescriptor(String path) {
          return
AbstractUIPlugin.imageDescriptorFromPlugin("WS.Generators.Python", path);
    }
}
```

Appendix B. Python Language Plug-in for SAP Scripting Tool Source Code

plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<!--
   Plugin.xml for the Python language plug-in.
   Copyright 2006 SAP AG
-->
<plugin>
   <extension
```



```
            point="com.sap.scripting.core.languages">
         <language
               default_extension="py"
               display_name="Python"
               id="com.sap.scripting.languages.python"/>
      </extension>
      <extension
            point="org.eclipse.ui.preferencePages">
      </extension>
   </plugin>
```

META-INF/Manifest.mf

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: SAP Scripting Python definition
Bundle-SymbolicName: com.sap.scripting.languages.python; singleton:=true
Bundle-Version: 0.8.0
Bundle-Vendor: www.sap.com
Bundle-Localization: plugin
Require-Bundle: com.sap.scripting.core,
 org.eclipse.core.runtime,
 org.eclipse.ui,
 com.sap.scripting.common.ui
Export-Package: com.sap.scripting.languages.python.preferences
```

build.properties

```
bin.includes = META-INF/,\
               plugin.xml
```

com.sap.scripting.languages.python.LanguagePythonMessages

```
/**
 * <copyright>
 * Copyright 2006 SAP AG
 * </copyright>
 */

package com.sap.scripting.languages.python;

import java.util.MissingResourceException;
import java.util.ResourceBundle;
```



```
/**
 *
 * <!-- begin-user-doc -->
 *
 * <b>LanguagePythonMessages</b>
 *
 * Utility class for handling messages.
 *
 * <!-- end-user-doc -->
 *
 */
public class LanguagePythonMessages {
  private static final String BUNDLE_NAME =
"com.sap.scripting.languages.python.messages"; //$NON-NLS-1$

  private static final ResourceBundle RESOURCE_BUNDLE = ResourceBundle
              .getBundle(BUNDLE_NAME);

  private LanguagePythonMessages() {
  }

  public static String getString(String key) {
        try {
              return RESOURCE_BUNDLE.getString(key);
        } catch (MissingResourceException e) {
              return '!' + key + '!';
        }
  }
}
```

com.sap.scripting.languages.python/messages.properties

```
Page_description=Python runtime options.
Page_runtimePathFieldLabel=Path to Python runtime:
Page_iniPathFieldLabel=Path to INI file:
```

com.sap.scripting.languages.python.preferences.IPreferencesConstants

```
/**
 * <copyright>
 * Copyright 2006 SAP AG
 * </copyright>
```



```
 */

package com.sap.scripting.languages.python.preferences;


/**
 *
 * <!-- begin-user-doc -->
 *
 * <b>IPreferencesConstants</b>
 *
 * Miscelanous constants regarding the Python interpreter and language.
 *
 * <!-- end-user-doc -->
 *
 */
public interface IPreferencesConstants {
  public static final String EXECUTABLE_PATH =
"com.sap.scripting.languages.python.preferences.exe_path";
  public static final String CONFIG_FILE_PATH =
"com.sap.scripting.languages.python.preferences.cfg_path";
}
```

com.sap.scripting.languages.python.preferences.Page

```
/**
 * <copyright>
 * Copyright 2006 SAP AG
 * </copyright>
 */


package com.sap.scripting.languages.python.preferences;

import org.eclipse.jface.preference.FieldEditorPreferencePage;
import org.eclipse.jface.preference.FileFieldEditor;
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.IWorkbenchPreferencePage;

import com.sap.scripting.common.plugin.UIPlugin;
import com.sap.scripting.languages.python.LanguagePythonMessages;
import com.sap.scripting.languages.python.preferences.IPreferencesConstants;
```



```
/**
 *
 * <!-- begin-user-doc -->
 *
 * <b>Page</b>
 *
 * Preference page for the Python language in SAP Scripting Tool.
 *
 * <!-- end-user-doc -->
 *
 */
public class Page extends FieldEditorPreferencePage implements
  IWorkbenchPreferencePage {


  public Page() {
        super(GRID);
        setPreferenceStore(UIPlugin.getDefault().getPreferenceStore());

  setDescription(LanguagePythonMessages.getString("Page_description"));
//$NON-NLS-1$
  }


  protected void createFieldEditors() {
        addField(new FileFieldEditor(IPreferencesConstants.EXECUTABLE_PATH,

  LanguagePythonMessages.getString("Page_runtimePathFieldLabel"),
getFieldEditorParent())); //$NON-NLS-1$
        addField(new FileFieldEditor(IPreferencesConstants.CONFIG_FILE_PATH,

  LanguagePythonMessages.getString("Page_iniPathFieldLabel"),
getFieldEditorParent())); //$NON-NLS-1$
        }


  public void init(IWorkbench workbench) {
  }
}
```


Related Content

Scripting Languages Support for SAP Services

I. SAP Developer Network (http://sdn.sap.com).

II. SAP Scripting Tool Overview (https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/4405).

III. Introduction to SAP Scripting Tool (https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/docs/library/uuid/591c31cf-0d01-0010-8ab7-a1f7d032a66c).

Eclipse Plug-in Development

IV. PDE Does Plug-ins (http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html).

V. PDE Developer's Guide (http://help.eclipse.org/help30/index.jsp?noscript=1).

VI. Java Developer's Guide to Eclipse, Second Edition (http://www.jdg2e.com/).

Python Specific

VII. Python Language (http://www.python.org/).

VIII. SOAPpy – Python extension for accessing web services (http://pywebsvcs.sourceforge.net/).

Disclaimer and Liability Notice

This document may discuss sample coding or other information that does not include SAP official interfaces and therefore is not supported by SAP. Changes made based on this information are not supported and can be overwritten during an upgrade.

SAP will not be held liable for any damages caused by using or misusing the information, code or methods suggested in this document, and anyone using these methods does so at his/her own risk.

SAP offers no guarantees and assumes no responsibility or liability of any type with respect to the content of this technical article or code sample, including any liability resulting from incompatibility between the content within this document and the materials and services offered by SAP. You agree that you will not hold, or seek to hold, SAP responsible or liable with respect to the content of this document.

