Developing Java Services for the SAP Business Connector

Thomas G. Schuessler

Thomas G. Schuessler is the founder of ARAsoft, a company offering products, consulting, custom development, and training to customers worldwide, specializing in integration between SAP and non-SAP components and applications. Thomas is the author of SAP's BIT525 and BIT526 classes. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years.

(complete bio appears on page 119)

The trouble with having an open mind, of course,
is that people will insist on coming along and trying to put things in it.
– Terry Pratchett, Diggers

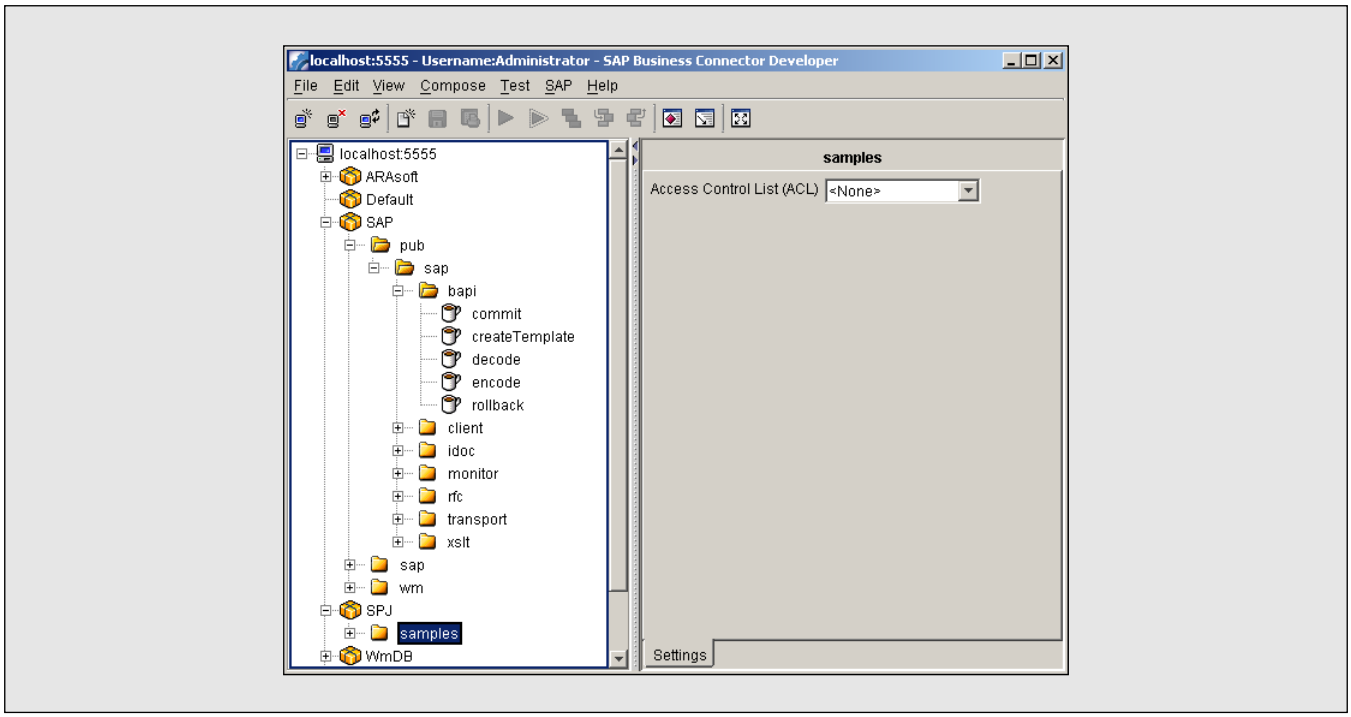
The SAP Business Connector[1] (SBC) is a great tool for XML-enabling ABAP-based SAP components like R/3. SBC is used primarily to connect SAP systems with business partner systems via the Internet. SBC uses a service-based architecture and delivers many ready-to-use services. Customers can build additional services by using SBC's graphical design environment (*flow services*) or by writing Java code (*Java services*).

While a lot can be accomplished with flow services, Java services offer distinct advantages in some cases:

- Since flow services are interpreted at runtime, the performance of Java services is better.
- Java services allow more flexible exception handling.
- Java services can use data types not supported in flow services, e.g., the classes of the Java Collection Framework.
- Java services can utilize functionality in existing Java libraries.
- Java services can easily perform operations that are awkward to

[1] For an excellent introduction to using the SAP Business Connector to exchange IDocs with a partner system, read Robert Chu's articles in the March/April 2003 issue of this publication.

Figure 1 Packages, Folders, and Services



express or impossible in flow services, e.g., numerical operations and string manipulation.

Since flow services can invoke Java services and vice versa, you can freely choose the approach best suited for a particular task.

This article is for Java programmers who are already a little bit familiar with SBC, but have only created flow services so far or want to extend their knowledge of Java services.

Creating Java Services in the SAP Business Connector

All development objects (flow services, Java services, etc.) in SBC are organized in folder hierarchies (cf. **Figure 1**). At the top of the hierarchy are packages. Packages are the basic units of software distribution for SBC.

In Figure 1, I have already added a package called `SPJ` and a folder called `samples`. If you want to follow the step-by-step description of creating a Java service, please do the same in your SBC Developer now. My development took place in SBC 4.7, but release 4.6 can be used as well (and probably older releases, but I have never used them).

The service we are going to build is a simple one. It accepts an input string and removes trailing blanks. SBC already has a trim service (*pub.string:trim* in package `WmPublic`), but it removes leading *and* trailing blanks.

Right-click on the `samples` folder and select "New…" from the pop-up menu. You will now see **Figure 2**. Select "Java Service" and click on the "Next" button.

On the subsequent window (**Figure 3**) enter a name for your service (I have called mine "rtrim") and make sure that the `samples` folder is selected as the

Figure 2 Selecting a Java Service

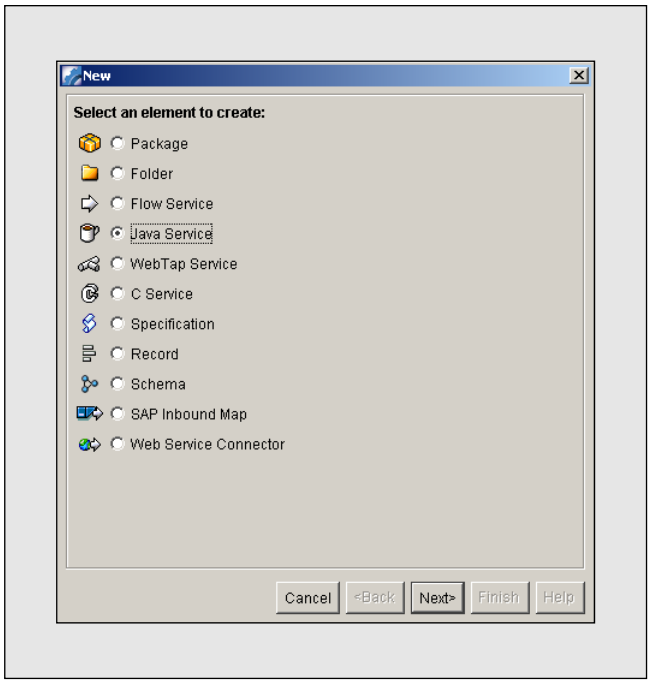
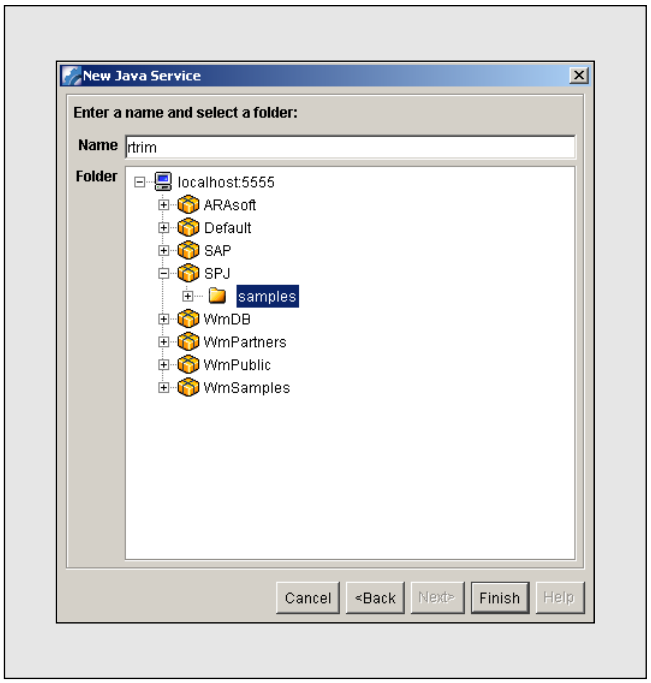


Figure 3 Entering a Name for the Java Service



destination for the service. Click the "Finish" button to add the new Java service.

In **Figure 4** you can see that SBC has generated the service signature. This signature cannot be

Figure 4 An Empty Java Service

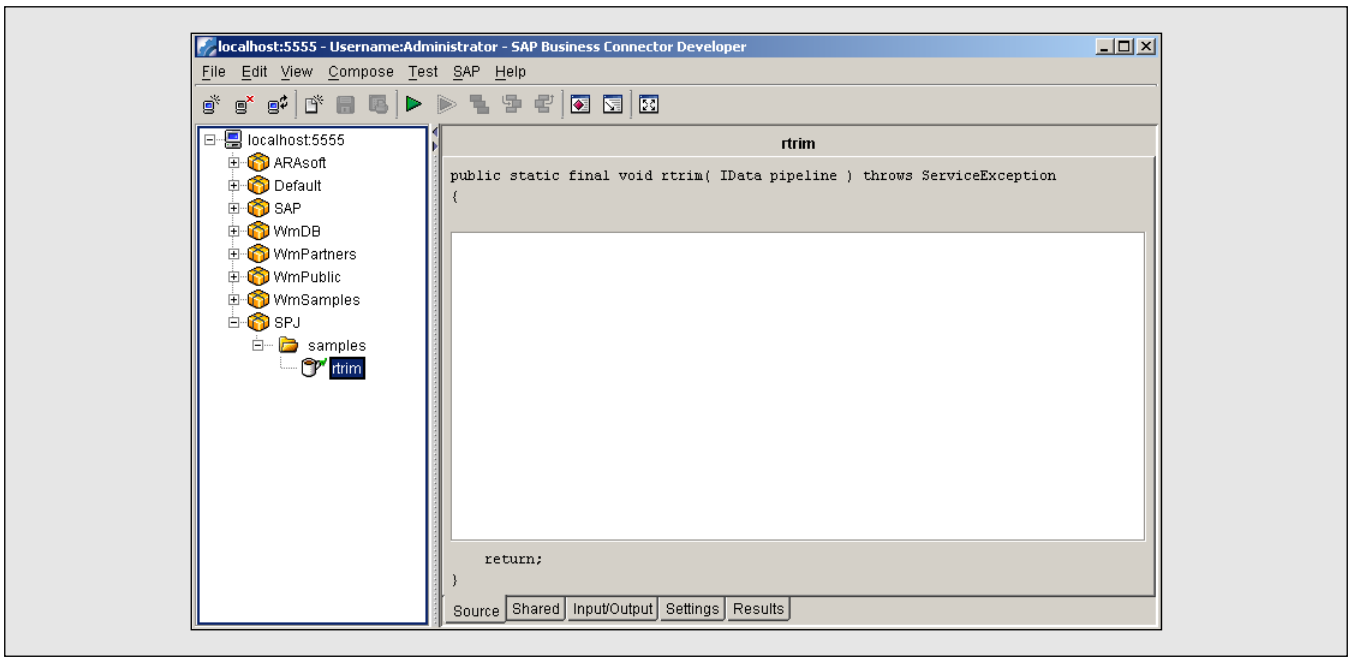
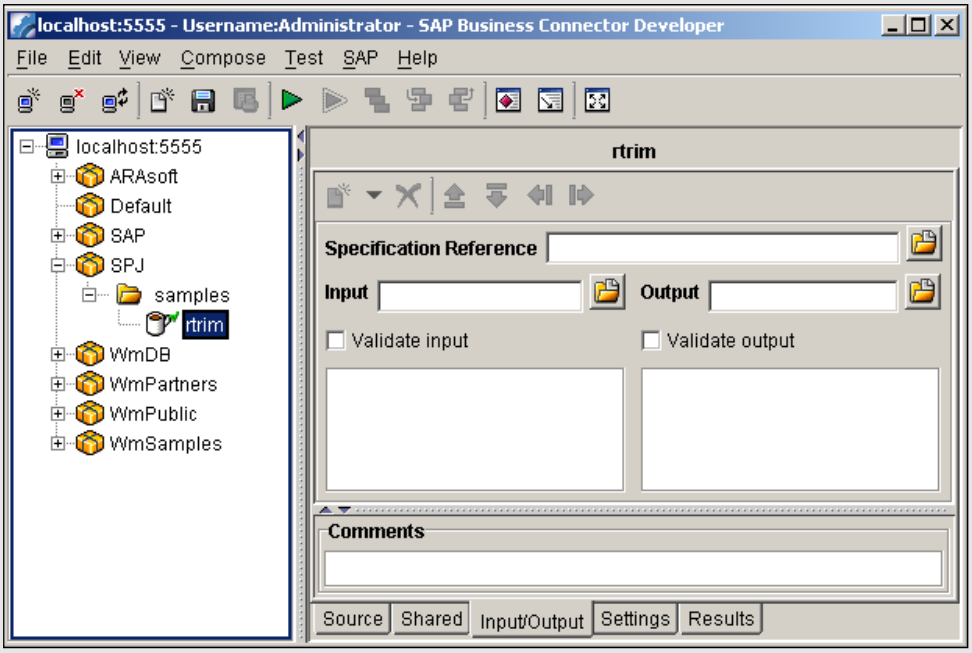


Figure 5 Defining Input and Output Parameters



changed. All SBC Java services are passed a reference to the pipeline[2], so that input parameters can be accessed and output parameters created. Note the `throws` clause of the signature. Java services must only throw exceptions of type *ServiceException* (or subclasses thereof).

Services do not have to specify which input parameters they expect and which output parameters they create, but of course it is highly recommended that you specify them. Otherwise prospective users of your service will have a hard time trying to figure out how to use your service. Click on the "Input/Output" tab to define your parameters (**Figure 5**).

Our service will have one input parameter (`inputString`) and one output parameter (`outputString`). To add an input parameter, click on the white area under the checkbox "Validate input" and click the insert icon () in the application toolbar (see **Figure 6**).

Select "String" from the list of data types and enter `inputString` as the parameter name. Define an output parameter called `outputString` in the same fashion. The result should look similar to **Figure 7**.

Our service will obviously require some code to retrieve the input parameter and create the output parameter. SBC offers the capability to generate such code for us. Once you are more experienced in writing SBC Java services, you will probably simply write that code yourself or copy it from another service, but the first few times it is a good idea to take advantage of the code-generation feature. Select

[2] The pipeline is a container, similar to a Java *Hashtable*, for the input and output variables of services.

Figure 6 Adding a String Parameter

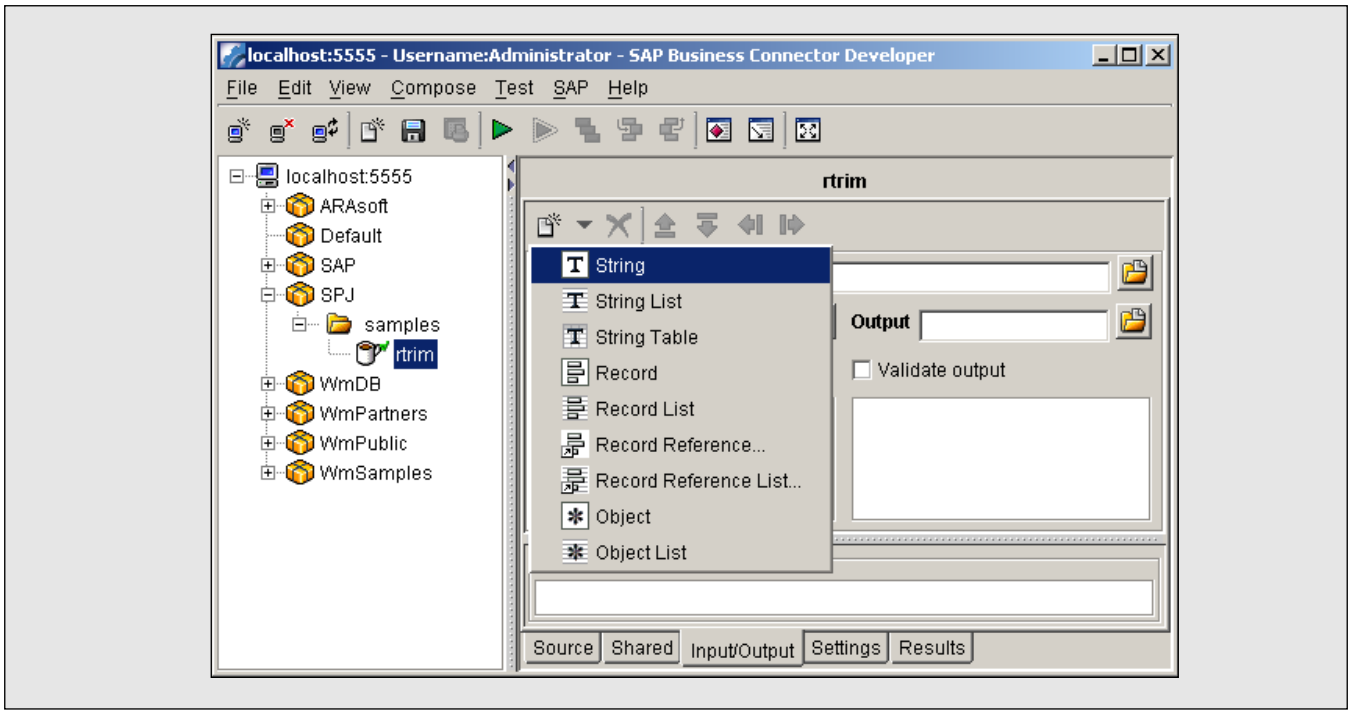


Figure 7 All Parameters Are Defined

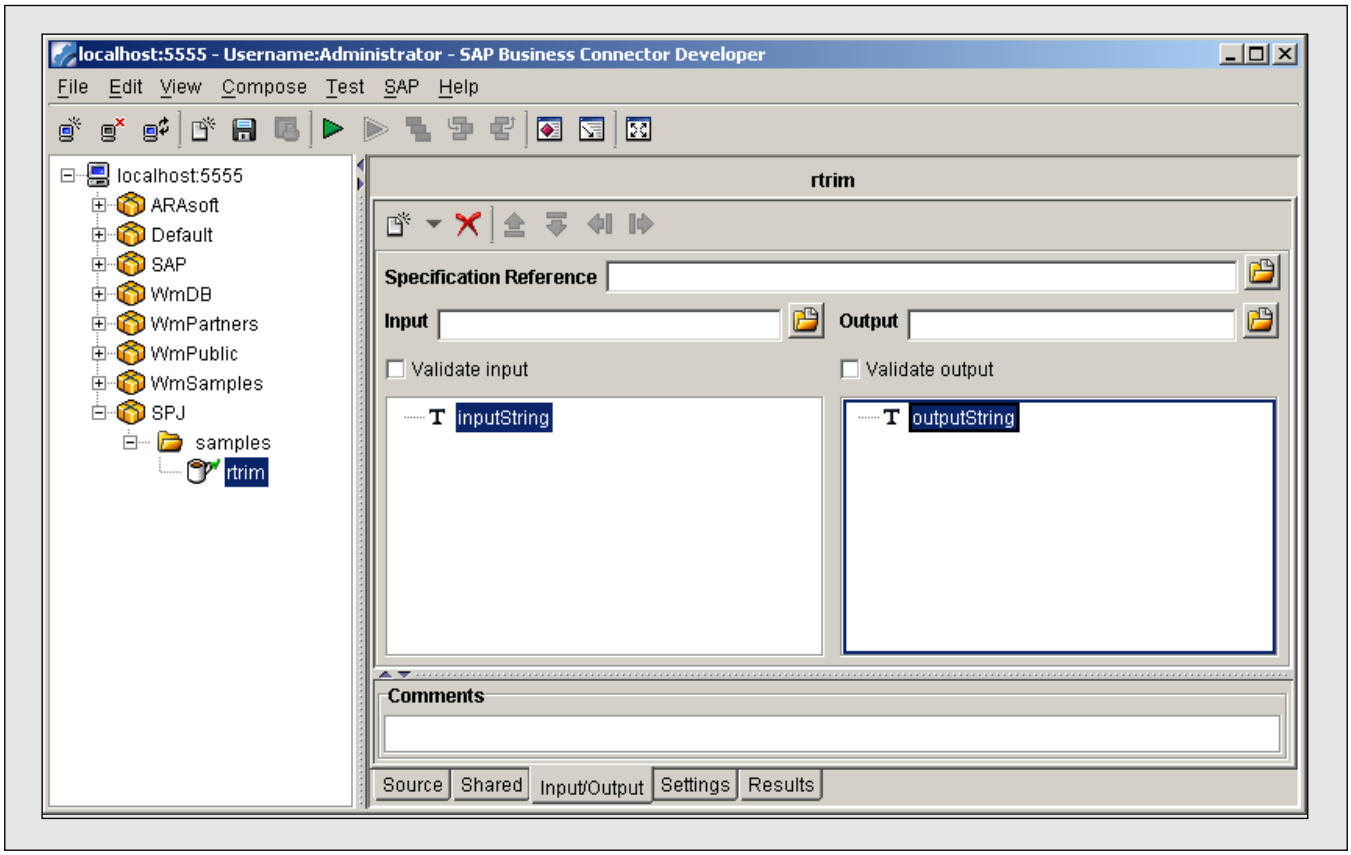


Figure 8 Generating Code, Step 1

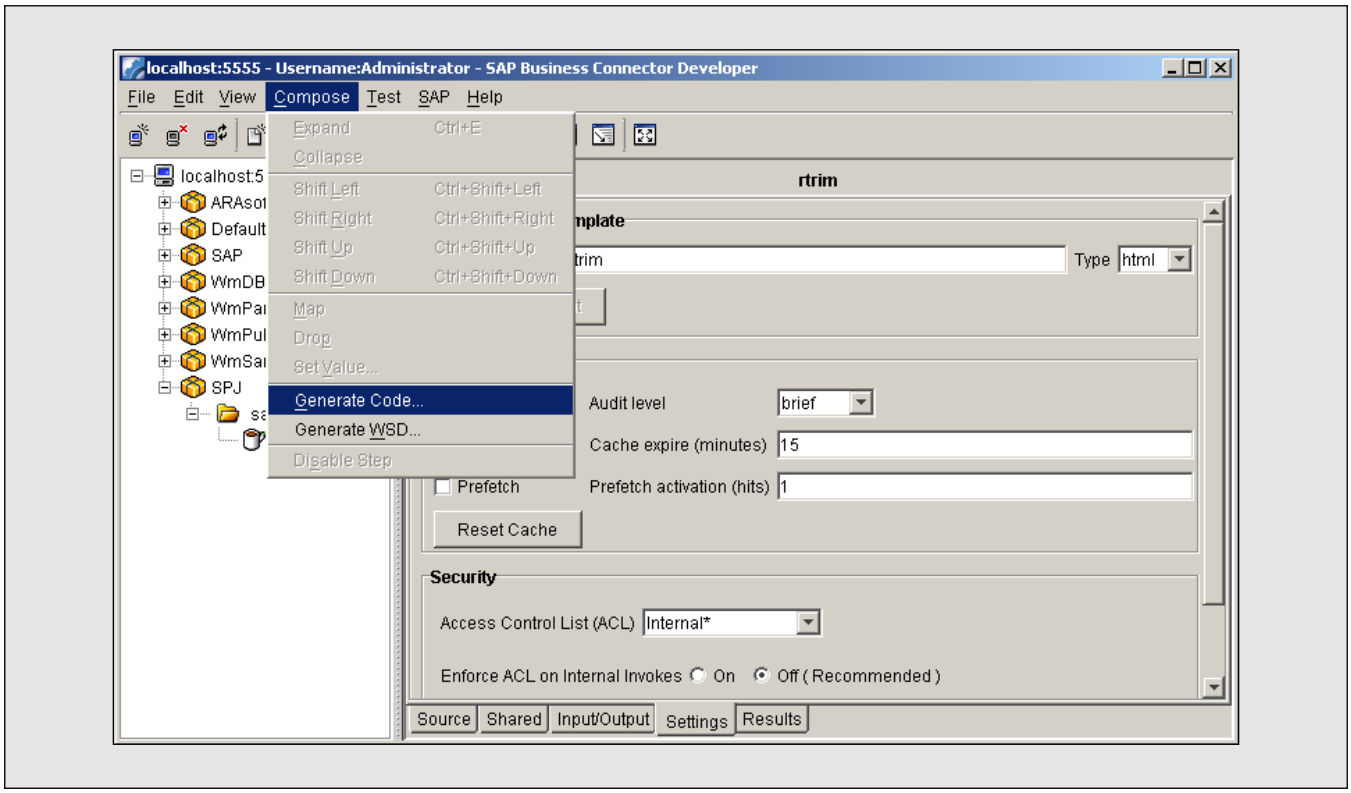
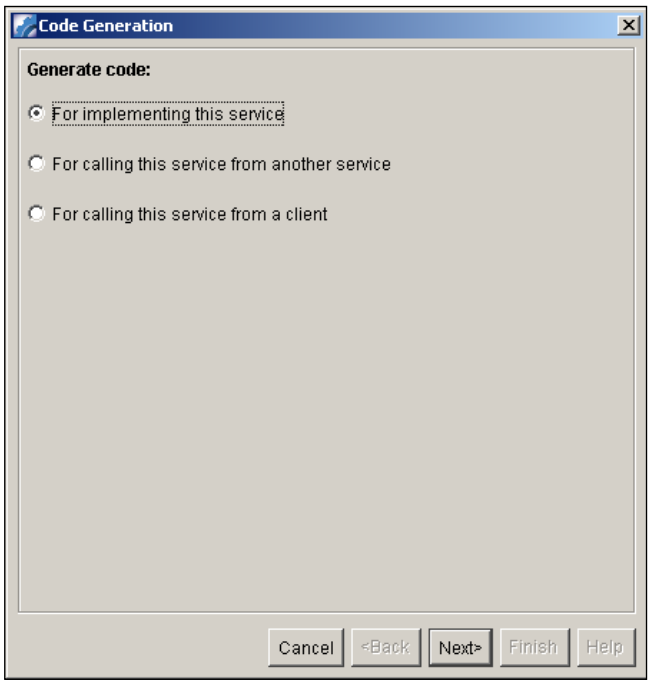


Figure 9 Generating Code, Step 2



"Compose → Generate Code…" from the menu (**Figure 8**).

As you can see in **Figure 9**, different types of code can be generated. Since you are creating a new service, select the first option ("For implementing this service") and click the "Next" button.

In **Figure 10**, you could now limit the generation to only input or only output parameters, but we want it all, so just click the "Finish" button.

As you can see in **Figure 11**, the code is now ready to be pasted into the source code editor. **Figure 12** contains the freshly pasted source code.

To understand the generated code (and to write your own), you will need to understand the API provided by SBC for Java development. The documentation of this API can be accessed at `<sapbc_root_directory>\Developer\doc\api\Java\index.html`. You should also check

Figure 10 Generating Code, Step 3

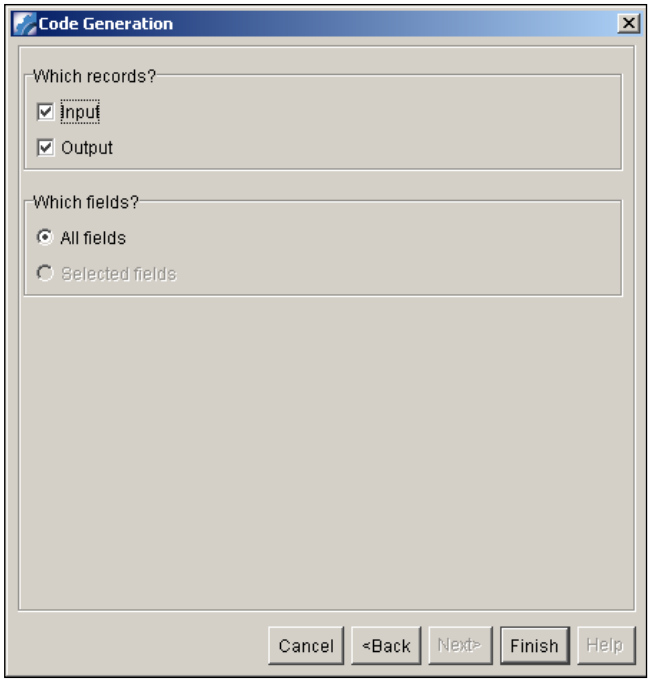
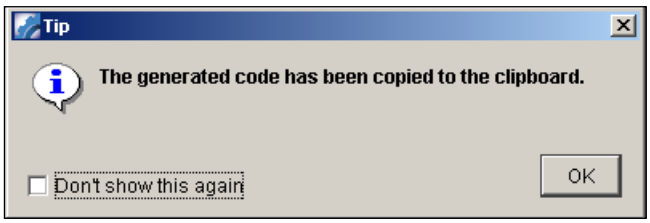


Figure 11 Generating Code, Step 4



out the SBC Developer Guide at `<sapbc_root_directory>\Developer\doc\SAPBCDeveloperGuide.pdf` and the `SAPBCDevTutorial.pdf` and `SAPBCBuiltInServices.pdf` files in the same directory. The documentation of the SAP-specific capabilities of SBC is a little more difficult to find (which has led some customers to believe that there is none). `SAPBCSapAdapterGuide.pdf` is located in the `<sapbc_root_directory>\Server\`

Figure 12 Generating Code, Step 5

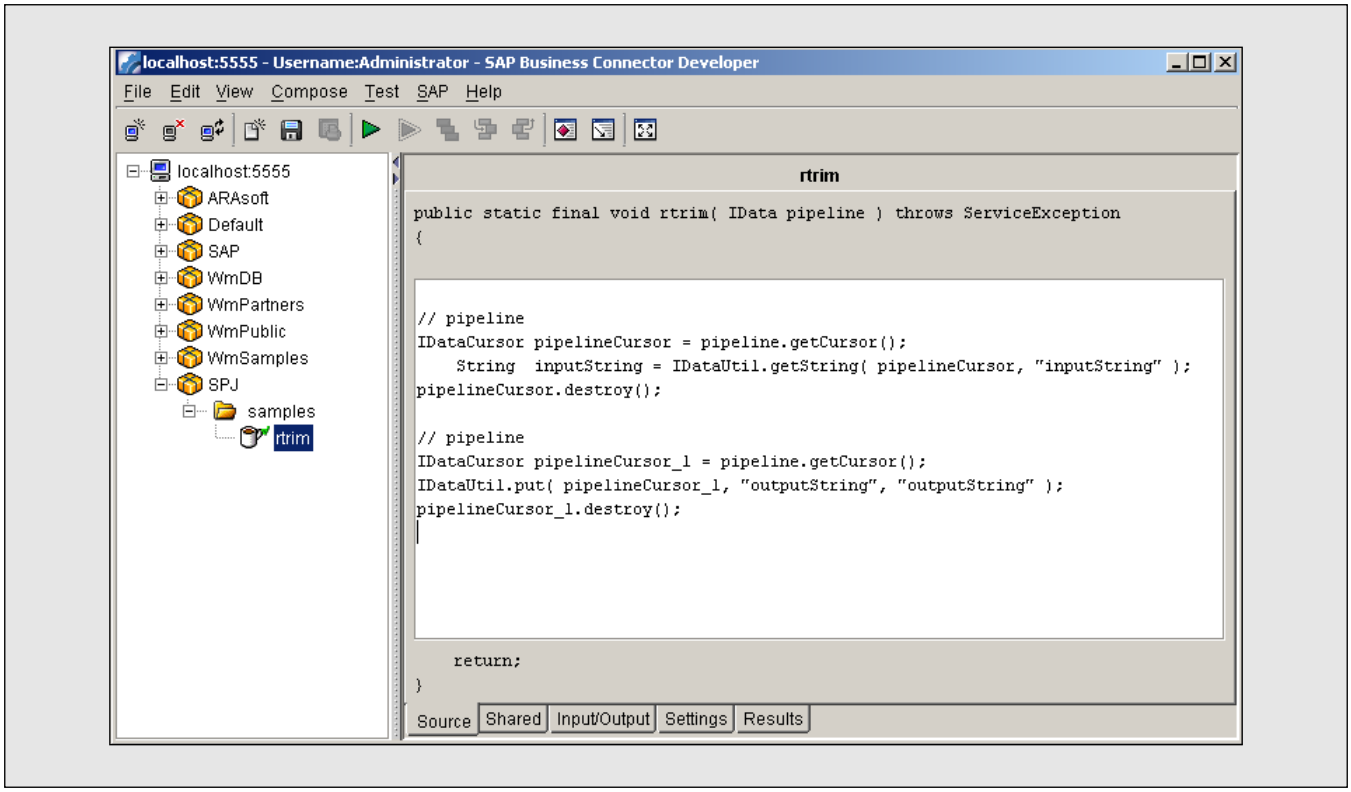
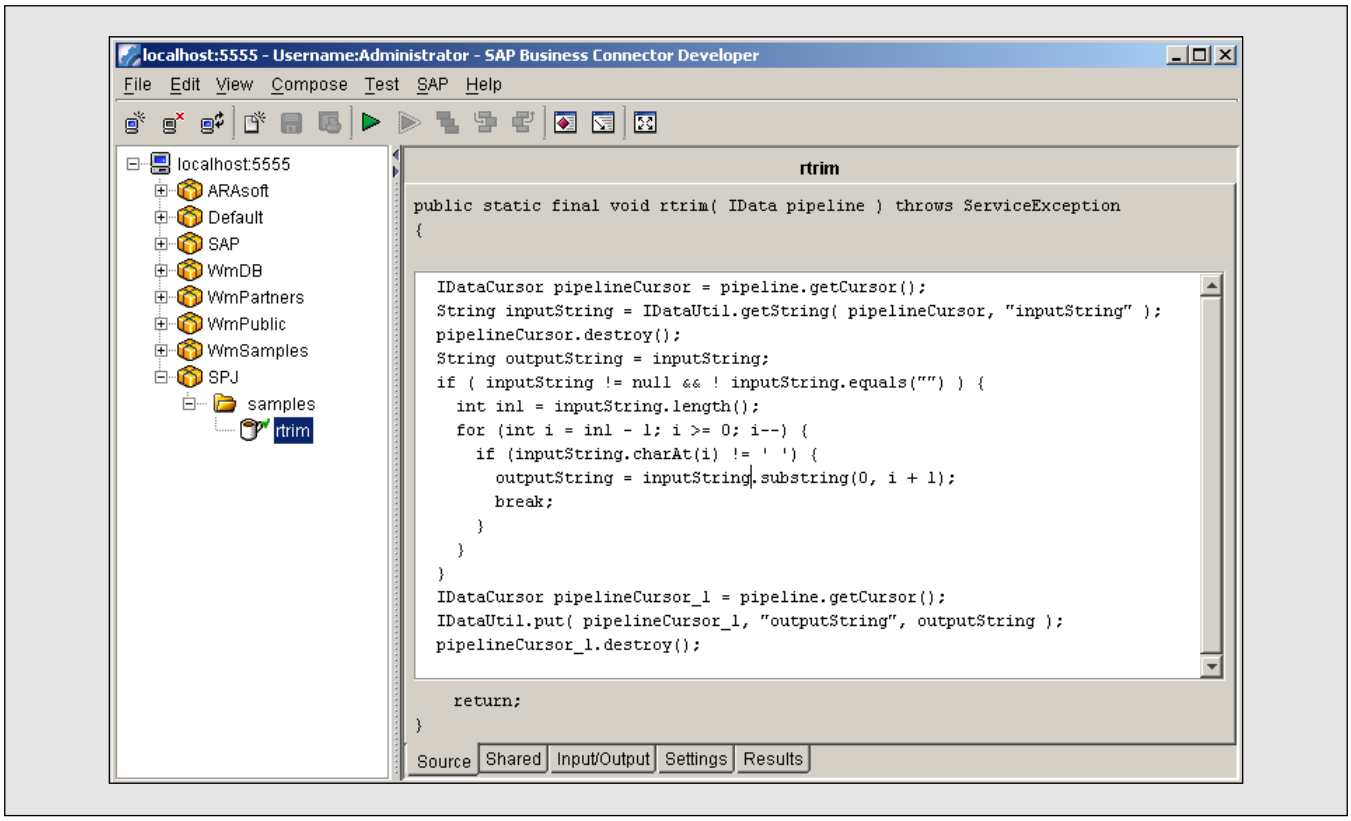


Figure 13 The Service Is Complete



`packages\SAP\doc` directory. Note that it is in the `Server` branch of SBC, not in the `Developer` branch like most other SBC documentation. To complete the enumeration of the documentation, two more documents can be found at `<sapbc_root_directory>\Server\doc`.

Back to the Java API. You should start your study by looking up the *IData* and *IDataCursor* interfaces, and the *IDataUtil* class.

As you can see in the method signature for our service (cf. Figure 12), the pipeline passed to the service is an object that implements the *IData* interface. The actual object type is *com.wm.util.Values*, but you should normally access the pipeline (and any other *Values* object) through the *IData* interface. This interface has only one non-deprecated method, *getCursor()*, which returns an object of type *IDataCursor*. It is through this cursor that we access the objects in the pipeline.

The generated code pasted in Figure 12 contains two blocks of code: one to retrieve the value of the input parameter and one to set the value of the output parameter. Let us begin with the first block.

The first statement in this block creates a cursor for the pipeline and assigns it to variable `pipelineCursor`. The cursor is then used in the second statement to retrieve the value for the input parameter `inputString` by calling the *getString()* method of class *IDataUtil*. This class provides a large number of static utility methods that facilitate access to *IData* objects. The third statement destroys the cursor.

The first and third statements of the second block are identical to the first block, creating a cursor and destroying it. The second statement sets the value of the output variable `outputString`, using the *put()* method of class *IDataUtil*. This method checks whether the pipeline already contains a variable with

Figure 14 *The rtrim Service*

```
IDataCursor pipelineCursor = pipeline.getCursor();
String inputString = IDataUtil.getString
                              ( pipelineCursor, "inputString" );
  String outputString = inputString;
  if ( inputString != null && ! inputString.equals("") ) {
    int inl = inputString.length();
    for (int i = inl - 1; i >= 0; i--) {
      if (inputString.charAt(i) != ' ') {
        outputString = inputString.substring(0, i + 1);
        break;
      }
    }
  }
IDataUtil.put( pipelineCursor, "outputString", outputString );
pipelineCursor.destroy();
```

the passed name. If so, its value is overwritten by the new value passed. Otherwise, a new variable is created and its value set.

All we have to do now is add our own processing logic between the two blocks, which, in our example, will remove trailing blanks. **Figure 13** is a screenshot of the completed service. Note that I have removed the comments that SBC generated and fixed the indentation.

Do we really need two different cursors to access the pipeline, as the generated code seems to imply? Absolutely not, so to optimize the performance of our service, we can change the generated code. The new version of the *rtrim* service is listed in **Figure 14**.

Once you have completed the source code for your service, you must compile it. This is accomplished via the save icon (), which first saves the source code and then invokes the Java compiler. If you get an error message at this point saying that the Java compiler could not be found, the Windows `path` does not include the `\bin` directory of the JDK. Modify the `path` and restart the SBC Server.

If the Java compiler is found and your source code contains no syntax errors, the hourglass cursor will simply change back to the normal cursor. Otherwise a pop-up window will appear showing you the syntax errors (see **Figure 15** for an example).

Unfortunately, SBC does not allow you to save

Figure 15 *Syntax Error Display*

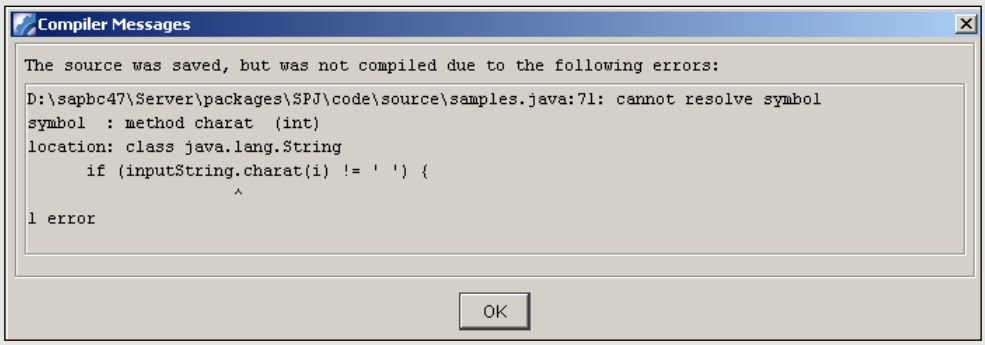


Figure 16 Testing the rtrim Service

the source code without compiling it. Once all syntax errors have been fixed, the new service is ready to be tested. This can be done by calling it from another Java service or a flow service, or by simply running it by itself. To do the latter, select the service in the hierarchy pane and click the "Run" icon (▶). A pop-up window will appear (**Figure 16**) where you can enter the input variables for the service. **Figure 17** shows the output of the service. It is a little difficult to see, but rest assured that the trailing blanks have been removed.[3]

[3] Hegel and Schopenhauer are the names of my cats, by the way. They fight a lot, just like the two philosophers did. Someone once said that some people only have cats to be able to give them strange names…

So far, so good. Our first Java service is ready. Let us create another one. You would probably rather not use the SBC Java source code editor this time, though. Its capabilities make Notepad look like a world-class editor. There is support for cut-and-paste and literally nothing else. Don't worry, I will now show you how to use an external development environment for editing the source code.

Using a Java IDE for SBC Java Service Development

You can use any Java IDE you like for the development of SBC Java services. Since it is becoming more and more fashionable all the time, I have selected Eclipse for this article. The first step required to build an additional service for the same package is to find the source code of our existing service and copy it. The directory that contains the source code is `<sapbc_root_directory>\Server\packages\SPJ\code\source`. The name of the source code file is `samples.java`. All Java services within one folder are bundled into one class,

Figure 17 The Output of the rtrim Service

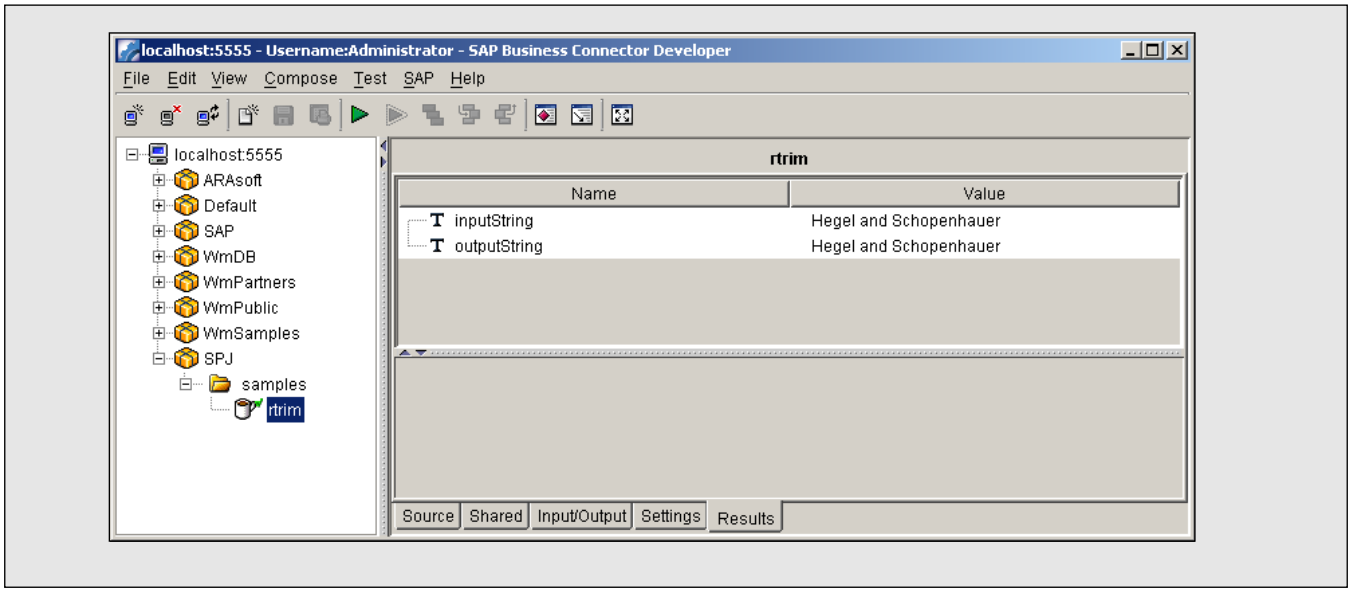
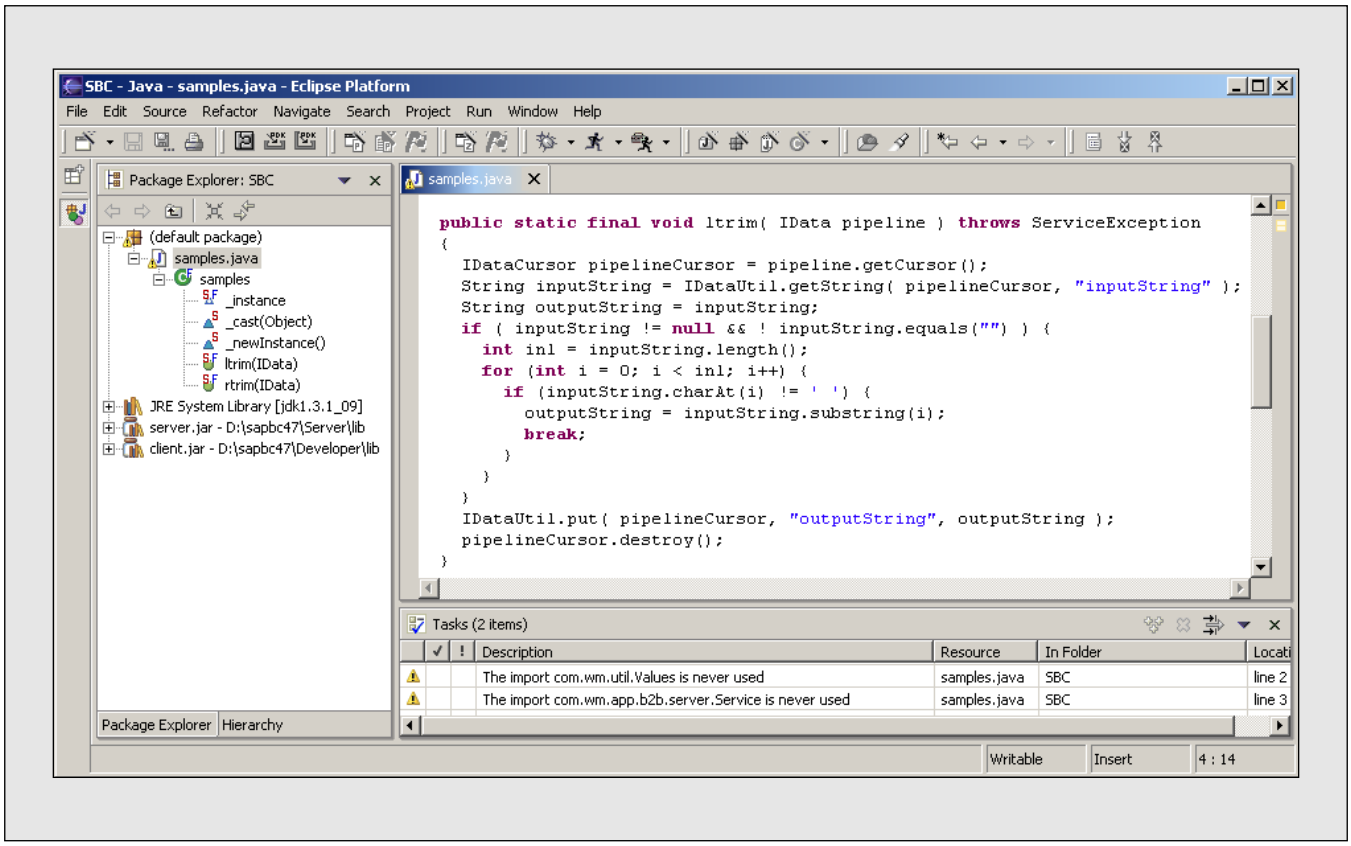


Figure 18 The Source Code for the Services in the samples Folder

```
// -----( B2B Java Code Template v1.2
// -----( CREATED: Thu Sep 04 20:31:10 CEST 2003
// -----( ON-HOST: HostName

import com.wm.data.*;
import com.wm.util.Values;
import com.wm.app.b2b.server.Service;
import com.wm.app.b2b.server.ServiceException;
// --- <<B2B-START-IMPORTS>> ---
// --- <<B2B-END-IMPORTS>> ---

public final class samples {
  // ---( internal utility methods )---
  final static samples _instance = new samples();
  static samples _newInstance() { return new samples(); }
  static samples _cast(Object o) { return (samples)o; }

  // ---( server methods )---
  public static final void rtrim (IData pipeline)
        throws ServiceException {
    // --- <<B2B-START(rtrim)>> ---
    // @sigtype java 3.5
    // [i] field:0:required inputString
    // [o] field:0:required outputString
    IDataCursor pipelineCursor = pipeline.getCursor();
    String inputString = IDataUtil.getString
                         ( pipelineCursor, "inputString" );
    String outputString = inputString;
    if ( inputString != null && ! inputString.equals("") ) {
      int inl = inputString.length();
      for (int i = inl - 1; i >= 0; i--) {
        if (inputString.charAt(i) != ' ') {
          outputString = inputString.substring(0, i + 1);
          break;
        }
      }
    }
    IDataUtil.put( pipelineCursor, "outputString", outputString );
    pipelineCursor.destroy();
    // --- <<B2B-END>> ---
  }
}
```

the name of which is identical to the name of the folder. Copy the file to some other destination, start your IDE, create a new project, and add the source file to the project. Before you can compile the file you need to make sure that the classpath for your project contains all required libraries. According to the SBC Developer Guide, you only need to include `<sapbc_root_directory>\Developer\lib\client.jar`. This is only partially true. Look at **Figure 18**, which contains the source code in

Figure 19 *The ltrim Service in Eclipse*



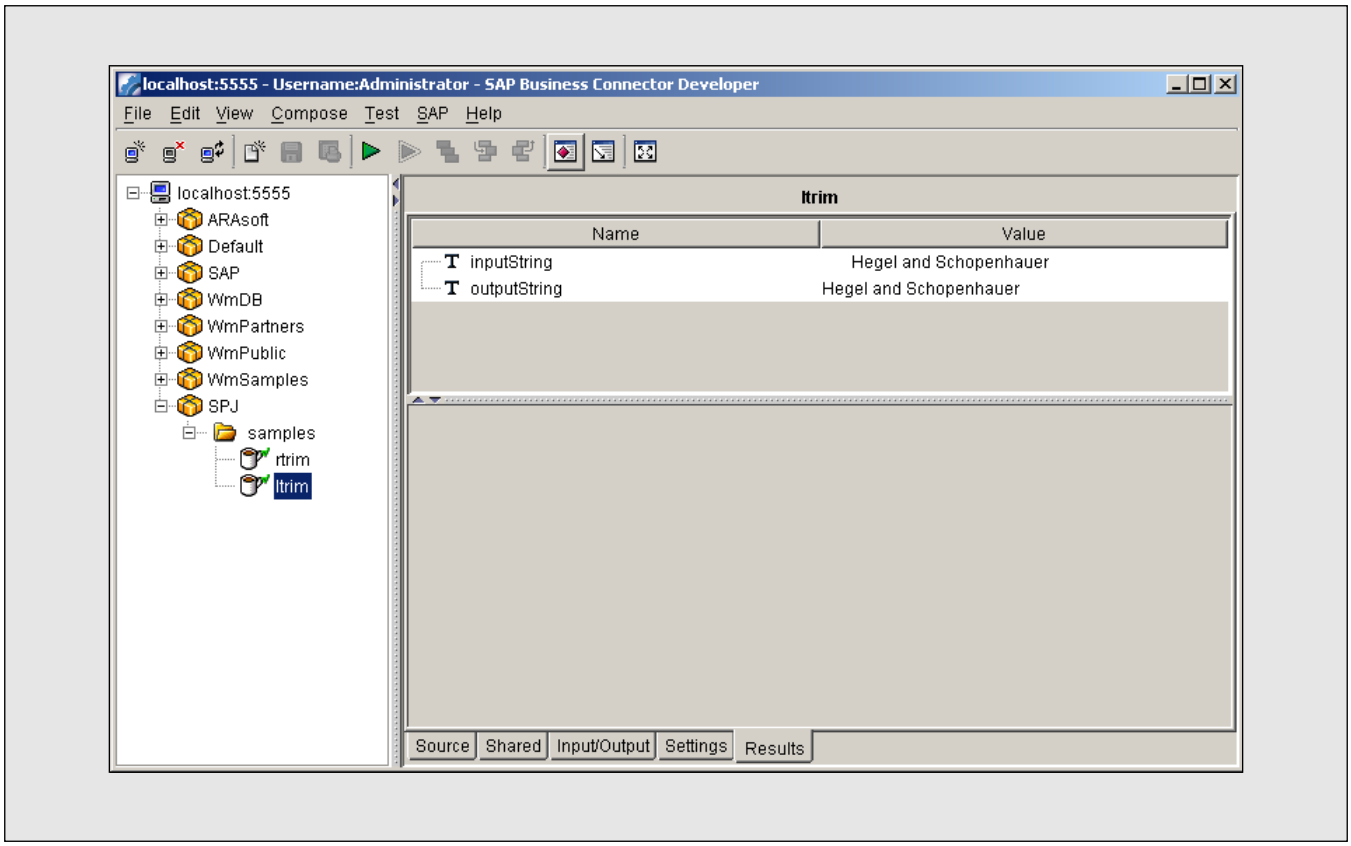
`samples.java` (slightly beautified for presentation purposes). SBC has generated four import statements for the class. Three of them reside in `client.jar`, but *com.wm.app.b2b.server.Service* will be reported as missing. You can simply comment out its import statement, or you can include `<sapbc_root_directory>\Server\lib\server.jar` in the classpath. As you can see in **Figure 19**, I have opted for the latter, just in case I ever actually need class *com.wm.app.b2b.server.Service*.

Let us now add a new service, *ltrim*, to the *samples* class. To yield the result in Figure 19, I have copied the code for the *rtrim* method and modified it so that the new method removes leading instead of trailing blanks. Obviously, using an IDE like Eclipse has made my life a lot easier — although the task at hand is comparatively simple. A decent IDE will help you by instantly notifying you of syntax errors and by offering code completion capabilities so that you do not have to program from memory and type everything yourself.

To add the new service to SBC, just add a new service and define its input and output parameters as we did before for *rtrim*. Copy and paste the appropriate source code from Eclipse and save the code in SBC. You should get no syntax errors since you have verified the correctness of the syntax in Eclipse. **Figure 20** contains the result of testing the new service in SBC.

This approach, in which you develop new services (or modify existing ones) in Eclipse (or any other Java IDE), define new services with their input and output parameters in SBC, and then copy and paste the source code from Eclipse, is my favorite way of dealing with the suboptimal Java editor in SBC, but there is an alternative that avoids the copy-and-paste

Figure 20 Testing the ltrim Service



process as well as the duplicate source issue (remember that I have copied the source file for use in Eclipse). SBC offers a utility, *jcode*, that supports the use of an external IDE. The main disadvantage of this utility is that it requires you to insert special tags so that it can interpret your source file appropriately (cf. Figure 18). The SBC Developer Guide contains a description of the utility and the required tags. Study the pertinent information and then decide for yourself which approach you prefer.

Publishing a Package

You have completed a set of related services in your development environment and now want to create a backup of your package or move it to the production (or QA) SBC Server. SBC makes this very easy. You can use SBC's replication mechanism, which is based on the publish/subscribe paradigm. Subscribers to a package can be defined on the SBC Server where the package was created or on the SBC Server that wants to receive the package. There can be multiple subscribers to one package.[4]

If you do not want to define a publish/subscribe relationship (or you just want to make an intermediate backup of a package), you can simply create an archive. The only real difference between an archive and a published package is that the latter will be sent to all subscribers automatically upon publication, while an archive is not visible to subscribers. You can manually copy your archive to a target SBC Server, though.

To create an archive, start the SBC Administrator and select "Packages → Management" from the

[4] More information about managing packages can be found in the SBC Administration Guide.

Figure 21 Creating a Package Archive

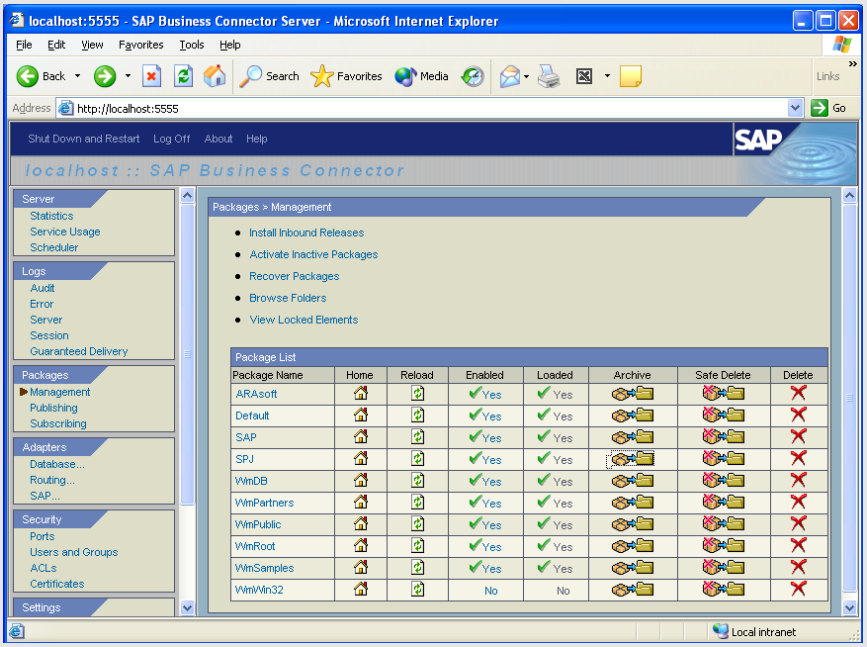
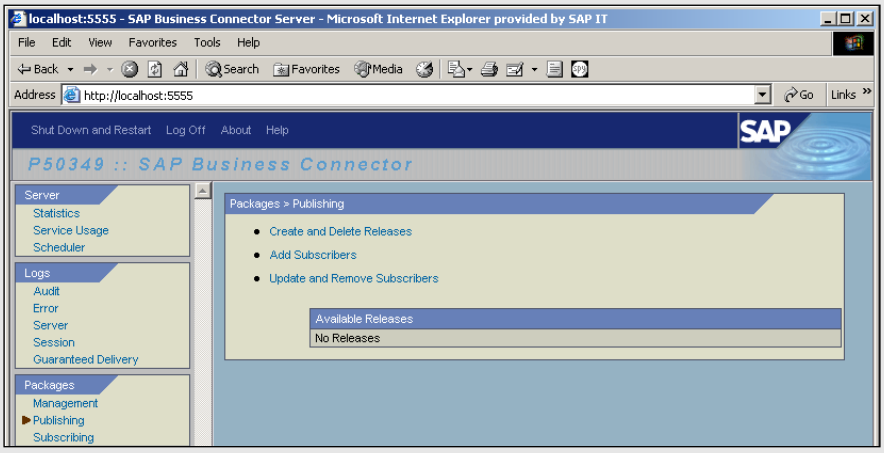


Figure 22 Publishing a Package, Step 1



navigation area on the left side (**Figure 21**). All packages available for this SBC Server are listed. Click the icon in the "Archive" column ( ) for the package you want to archive.

Since the next steps are very similar for both creating an archive and publishing a package, we will leave the archive creation scenario and create a release of our package instead.

To publish a release of a package, select "Packages → Publishing" in the SBC Administrator (**Figure 22**). Click on "Create and Delete Releases", which takes you to **Figure 23**.

Click on “Create Release” in the right-hand column for your package. In **Figure 24** you can select which files are to be included. There are two main reasons for excluding certain files:

- You are creating a patch release that only contains the objects that were changed since the last release.
- You do not want to include the Java source code for your package, because, for instance, you will ship the package to customers who should not know your trade secrets. Or you do not want to run the risk of somebody modifying the source code in your production SBC Server.

Unless you have a gigantic screen, you will now have to scroll down in Figure 24 (the result of which is **Figure 25**).

You can select the release type (the default is a full release) and specify the release name (the default is the package name). The generated file will have the extension “.zip”. Assign a version to your release and click the “Create Release” button. As you can see in **Figure 26**, a release has been created and is now ready for the package’s subscribers.

Figure 23 Publishing a Package, Step 2

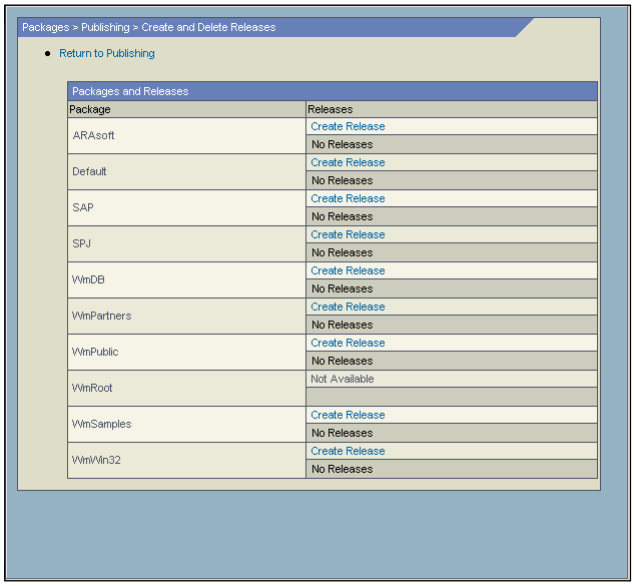


Figure 24 Publishing a Package, Step 3

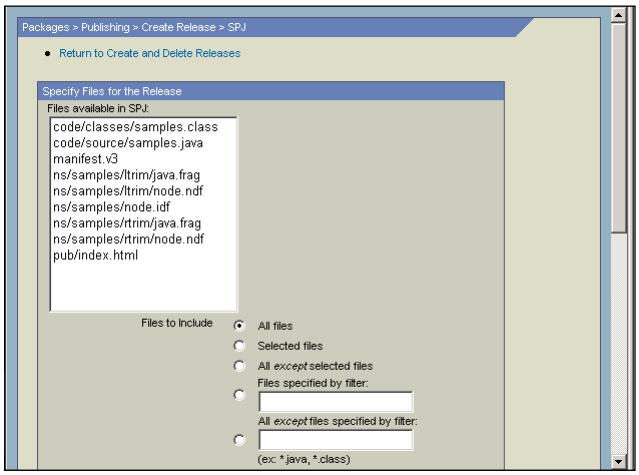


Figure 25 Publishing a Package, Step 4

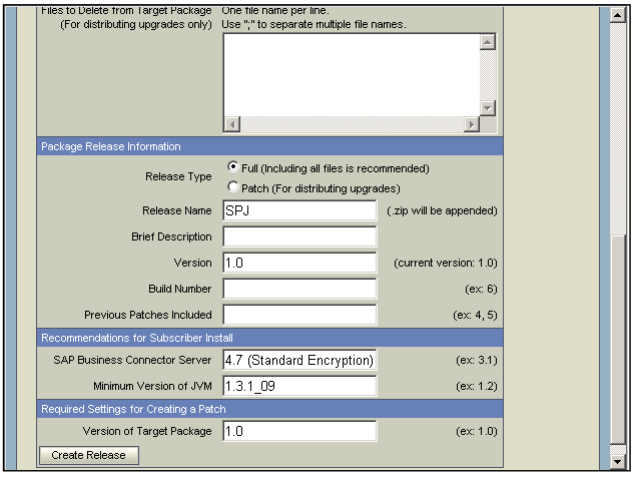


Figure 26 Publishing a Package, Step 5

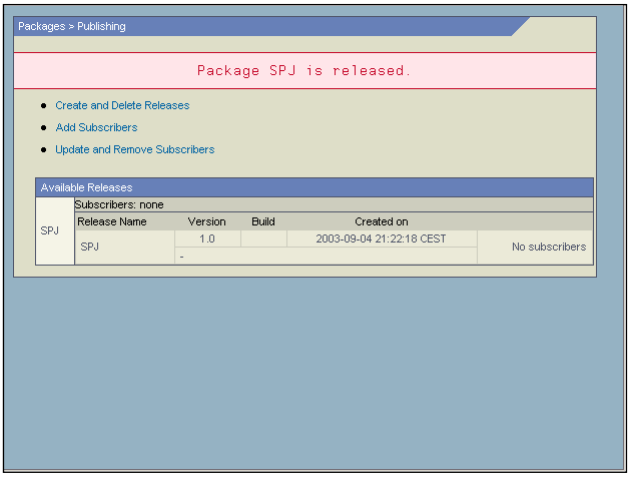
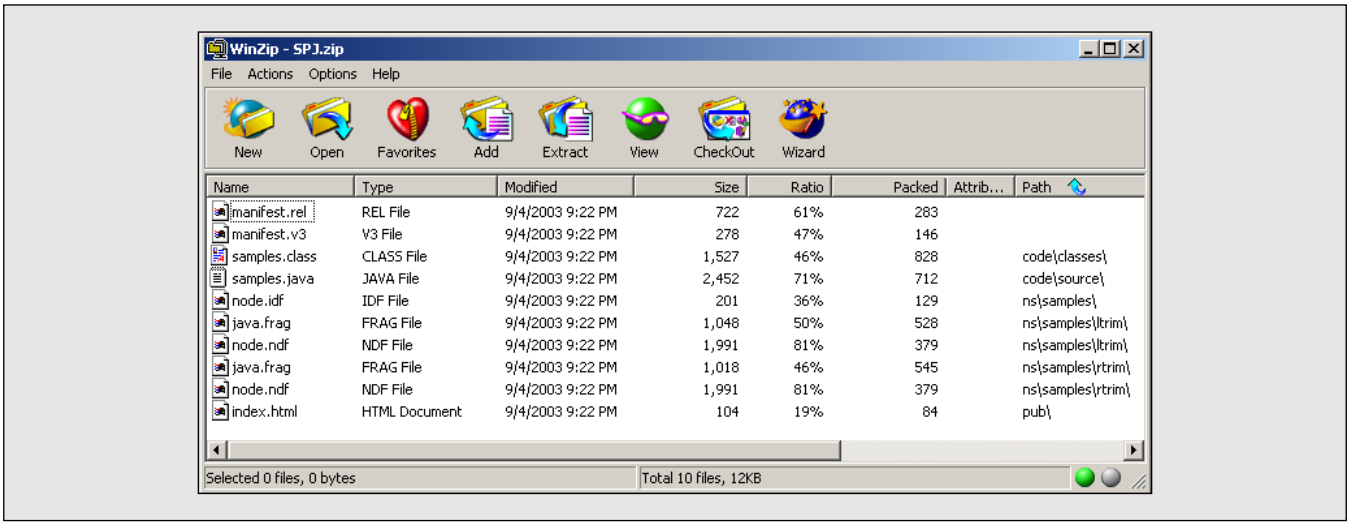


Figure 27 The Contents of the ZIP File



Regardless of whether you have created an archive or a release, the generated file resides in the `<sapbc_root_directory>\Server\replicate\outbound` directory. **Figure 27** is a screenshot of the contents of the generated file.

Using Additional Java Libraries in SBC

So far, we have written services from scratch, using nothing more than the standard Java and SBC libraries. Of course, it is also possible to take advantage of other existing Java libraries. To access these libraries, you need to make sure that they are accessible to SBC. There are four ways to accomplish this:

- Edit the `server.bat` file in the `<sapbc_root_directory>\Server\bin` directory. You can choose to add a library right after the `server.jar` file (PREPENDCLASSES) or at the end of the classpath (APPENDCLASSES). **Figure 28** shows the relevant section in `server.bat`. Since the library is not associated with a package, it will not be included in the ZIP file generated by SBC when you archive or release a package. When a package using the library is installed in another SBC Server, you need to make sure that the library is available to this server and that its `server.bat` file is updated accordingly.

  You will normally only use this approach if the specific position of your library in the classpath is important.

- Copy your library to the `<sapbc_root_directory>\Server\lib\jars` directory. Since the library is not associated with a package, it will not be included in the ZIP file generated by SBC when you archive or release a package. When a package using the library is installed in another SBC

Figure 28 Changing the Classpath in server.bat

```
set PREPENDCLASSES=
set APPENDCLASSES=D:\ARAsoft\JCoExtLib\lib\ARAsoftJCo.jar
```


Server, you need to make sure that the library is copied to its `<sapbc_root_directory>\Server\lib\jars` directory.

- Copy your library to the `<sapbc_root_directory>\Server\packages\<package_name>\code\jars\static` directory of the package with which this library is (mainly) associated. The last part of this directory structure ("static") is not automatically created by SBC, so you will have to add the subdirectory yourself. This approach ensures that the library is accessible by all packages, not just the one identified by "package_name". In addition, the library will be included in the ZIP file generated by SBC when you archive or release the package. If other packages also use the library, the package in which the library is contained must be available to any SBC Server on which the other packages are deployed. SBC allows you to define package dependencies to ensure that all required packages are available.
- Copy your library to the `<sapbc_root_directory>\Server\packages\<package_name>\code\jars` directory of the package with which this library is associated. This approach is similar to the previous one in that the library will also be included in the ZIP file created for archiving or releasing the package, but there are two important differences: the library can only be used in this package — it is unavailable to any other package — and you can no longer compile the package from within the SBC Developer. In other words: this is not a very popular approach.

I will now show you examples of how I made functionality offered by an existing library available as SBC services. This will allow me to discuss several advanced features of Java service development in SBC.

In the first example, I want to retrieve the list of countries defined in R/3 and expose the result as a string table. The complete source code for the *getCountries()* service can be found in **Figure 29**.

Figure 29 Service getCountries

```
public static final void getCountries (IData pipeline)
                         throws ServiceException {

  IDataCursor pipelineCursor = pipeline.getCursor();

  String serverName = IDataUtil.getString
                                (pipelineCursor, "serverName");
  if ( serverName == null || serverName.equals("") ) {
    throw new ServiceException("serverName must be specified.");
  }
  serverName = serverName.toUpperCase();

  ObjectFactory bof = getObjectFactory(serverName);
  try {
    Countries countries = bof.getSapData().getCountries();
    String[][] table = new String[countries.getSize()][2];
    for (int i = 0; i < countries.getSize(); i++) {
      String[] item = new String[2];
      Country country = countries.getCountry(i);
```

(continued on next page)

Figure 29 (continued)

```
        item[0] = country.getInternalCode();
        item[1] = country.getDescription();
        table[i] = item;
      }
      IDataUtil.put(pipelineCursor, "countries", table);
    }
    catch (Exception ex) {
      throw new ServiceException(ex.toString());
    }
    finally {
      pipelineCursor.destroy();
    }
  }
```

Figure 30 Throwing an Exception

The first interesting feature in the source code is that the service throws an exception if input variable `serverName` does not exist or is an empty string. Throwing an exception with a meaningful error message (see **Figure 30**) is preferable to a strange error happening later in the service (like the popular *NullPointerException*). Remember that Java services are only allowed to throw exceptions of type *ServiceException* or its subclasses.

Figure 31 The Interface for the getCountries Service

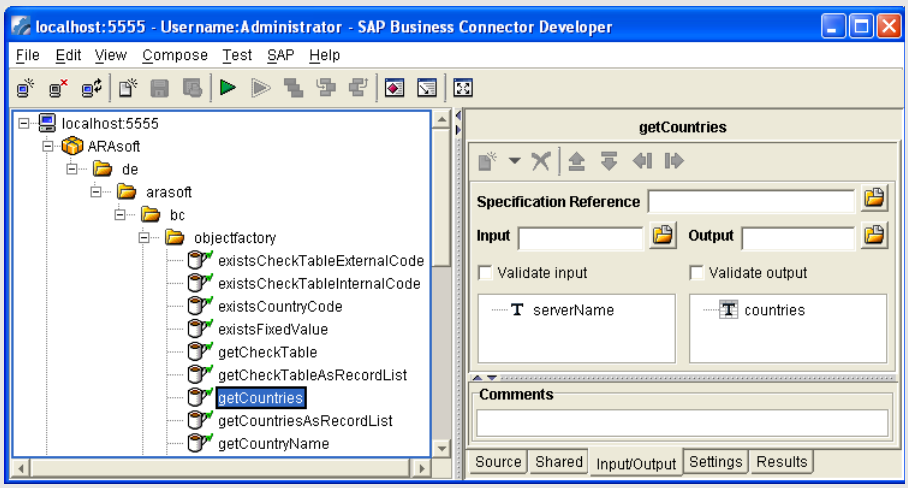
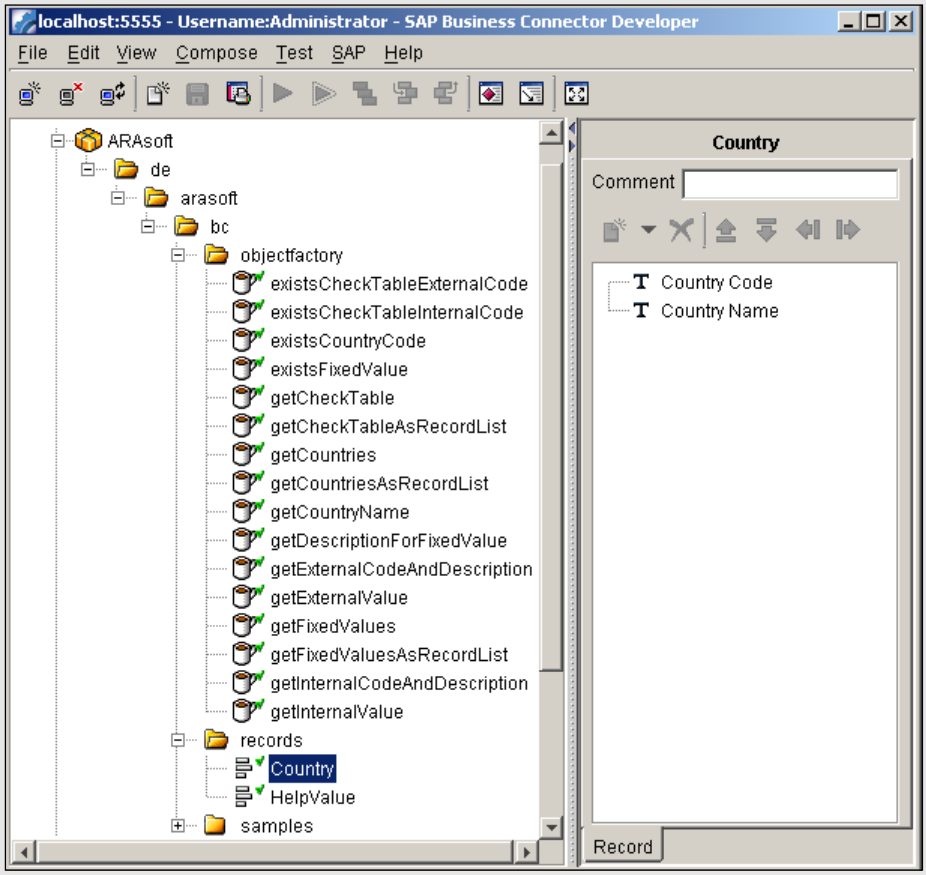


Figure 32 The Country Record Definition



After we have retrieved the list of countries from R/3 (using functionality of the existing library), we create a two-dimensional string array (the string table) and populate it with the countries' codes and names. Finally, the string table is put into the output variable `countries`.

While *getCountries()* is a useful service, its definition of the output it produces is a little vague (see **Figure 31**). The only thing a programmer trying to use the service can be sure of is that a string table is returned under the name `countries`. How many columns will this string table have? Which information will be available in which column? Of course, we could write documentation describing this, but it is always better to have a formal definition of a method's interface.

In SBC, you can define data structures in so-called *records*. A record can be a flat structure with a few fields, but it can also be a complex data description with a tree-like structure. For our current purposes, all we need is a flat structure with a field for the country code and another one for the country name. **Figure 32** shows the record that contains these fields.

Utilizing this record, we can build a new service, called *getCountriesAsRecordList*, the interface of

Figure 33 The Interface for the getCountriesAsRecordList Service

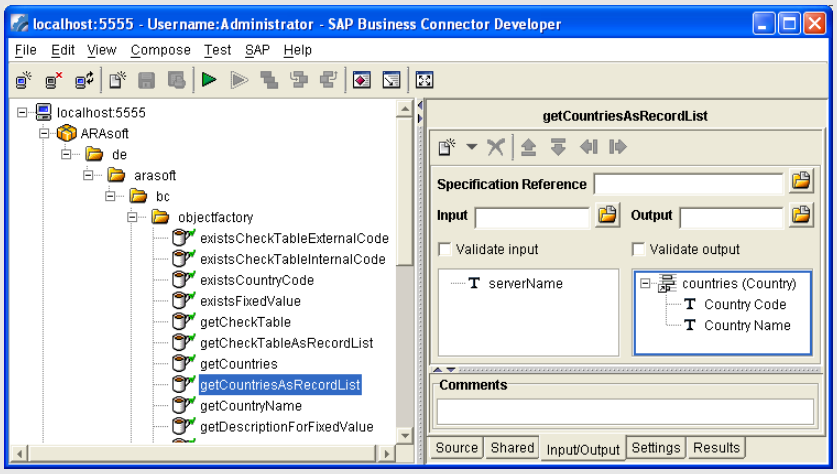


Figure 34 Service getCountriesAsRecordList

```
public static final void getCountriesAsRecordList (IData pipeline)
                         throws ServiceException {
  IDataCursor pipelineCursor = pipeline.getCursor();

  String serverName = IDataUtil.getString
                                (pipelineCursor, "serverName");
  if ( serverName == null || serverName.equals("") ) {
    throw new ServiceException("serverName must be specified.");
  }
  serverName = serverName.toUpperCase();

  ObjectFactory bof = getObjectFactory(serverName);
  try {
    Countries countries = bof.getSapData().getCountries();
    String[] columnNames =
      getColumnNamesForRecord("ARAsoft",
                              "de.arasoft.bc.records:Country");
    Table table = new Table("countries", columnNames);
    for (int i = 0; i < countries.getSize(); i++) {
      Country country = countries.getCountry(i);
      table.addRow(new Object[]
        { country.getInternalCode(), country.getDescription() });
    }
   IDataUtil.put(pipelineCursor, "countries", table);
  }
  catch (Exception ex) {
    throw new ServiceException(ex.toString());
  }
  finally {
    pipelineCursor.destroy();
  }
}
```


which is depicted in **Figure 33**. As you can see, a developer using the service now has a clear idea of what to expect: a list of records with specific fields that can be accessed by their names.

Let us look at the source code of the new service (**Figure 34**). In order to create the record list, we use SBC's *com.wm.util.Table* utility class. We call its constructor, passing a table name and a string array containing the column names. These names could be hard-coded, but we retrieve them dynamically from the record definition, using the *getColumnNamesForRecord* method, which will be discussed below. To add a new row to the table, we employ the *Table* class's *addRow()* method.

Why did I decide not to hard-code the column names? Firstly, because hard-coding can easily lead to trouble in the future. If I ever decided to change the field names of the *Country* record, I would have to remember to change the hard-coded names in all services using this record. Secondly, providing a special method for dynamically looking up the field names allows me to introduce the concept of shared code for Java services.

Remember that all Java services inside a folder are actually methods of the same class. SBC allows you to define shared code for all these services. Take a look at **Figure 35**. There are four sections for the different types of shared code:

- **Extends:** Here you can define a superclass for your class. If you leave this area empty, your class is derived directly from *java.lang.Object*.
- **Implements:** If your class needs to implement any interfaces, they can be enumerated here.
- **Imports:** Here you list all the packages, classes, and interfaces that you want SBC to generate Java import statements for. Note that in Figure 35, I

Figure 35 *Shared Code*

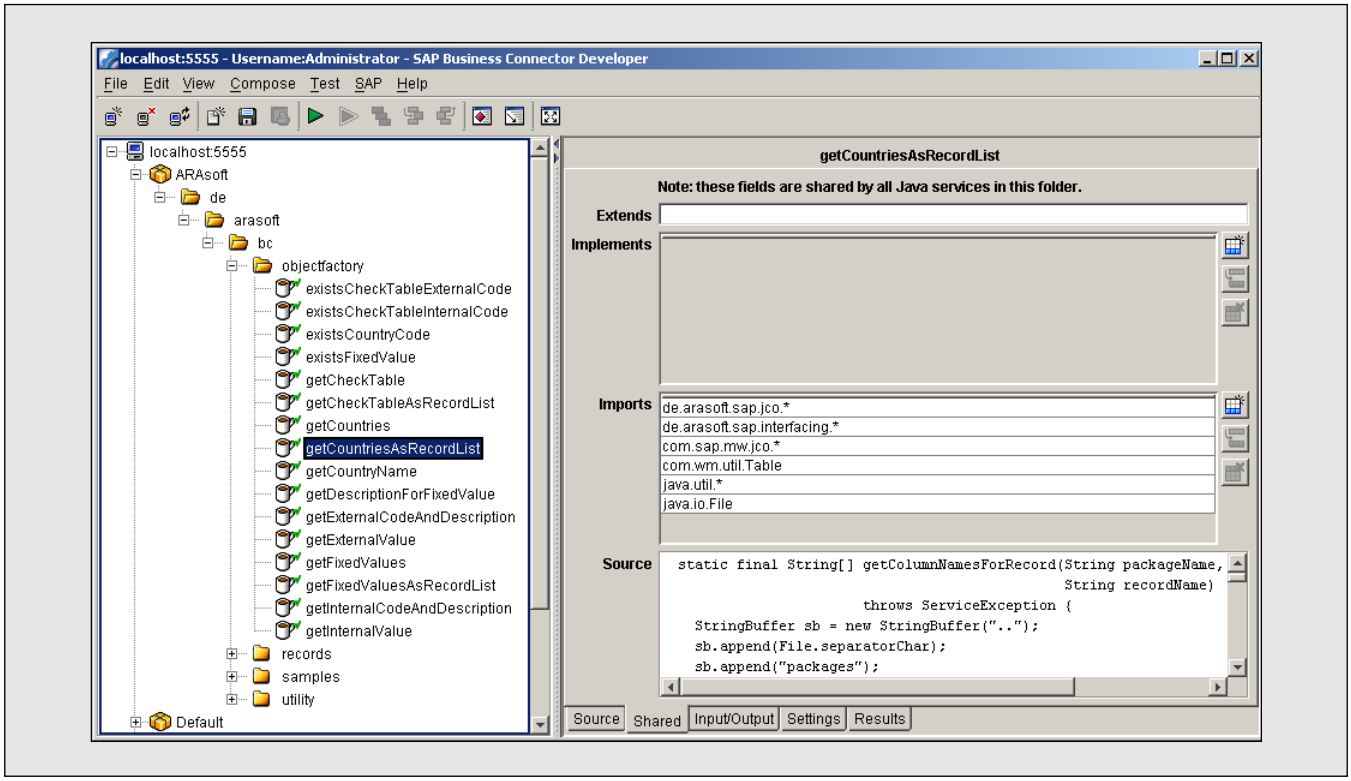


Figure 36 Static Method getColumnNamesForRecord

```
static final String[] getColumnNamesForRecord(String packageName,
                                               String recordName)
                      throws ServiceException {
  StringBuffer sb = new StringBuffer("..");
  sb.append(File.separatorChar);
  sb.append("packages");
  sb.append(File.separatorChar);
  sb.append(packageName);
  sb.append(File.separatorChar);
  sb.append("ns");
  sb.append(File.separatorChar);
  int offset = sb.length() - 1;
  sb.append(recordName);

  for (int i = offset; i < sb.length(); i++) {
    char c = sb.charAt(i);
    if (c == '.' || c == ':')
      sb.replace(i, i + 1, File.separator);
  }
  sb.append(File.separatorChar);
  sb.append("node.ndf");
  String path = sb.toString();
```

have included *com.wm.util.Table*, since this class is utilized in the *getCountriesAsRecordList* service.

- **Source:** Here you place any methods that should be available to the services in the folder, but not available as services themselves. These methods must be static.

Since SBC does not have a built-in service to retrieve a record definition, we do it ourselves in the shared *getColumnNamesForRecord* method (listed in **Figure 36**).

This method invokes the standard SBC service *pub.flow:restorePipelineFromFile* in order to load the file that contains the record definition into the pipeline. Well, not exactly *the* pipeline, but a new pipeline created just for the invocation of this service. Remember that the pipeline passed to a service is an *IData* object. If we had used the pipeline passed to the service that invokes the *getColumnNamesForRecord()* method (viz., the *getCountriesAsRecordList* service), we would have had to add any input and output variables required by the *pub.flow:restorePipelineFromFile* service (`fileName`) to the pipeline, thus polluting it with variables that are totally irrelevant for our clients.

It is much easier to create a new *Values* object (the *Values* class implements the *IData* interface) and pass it as the pipeline to *pub.flow:restorePipelineFromFile*.

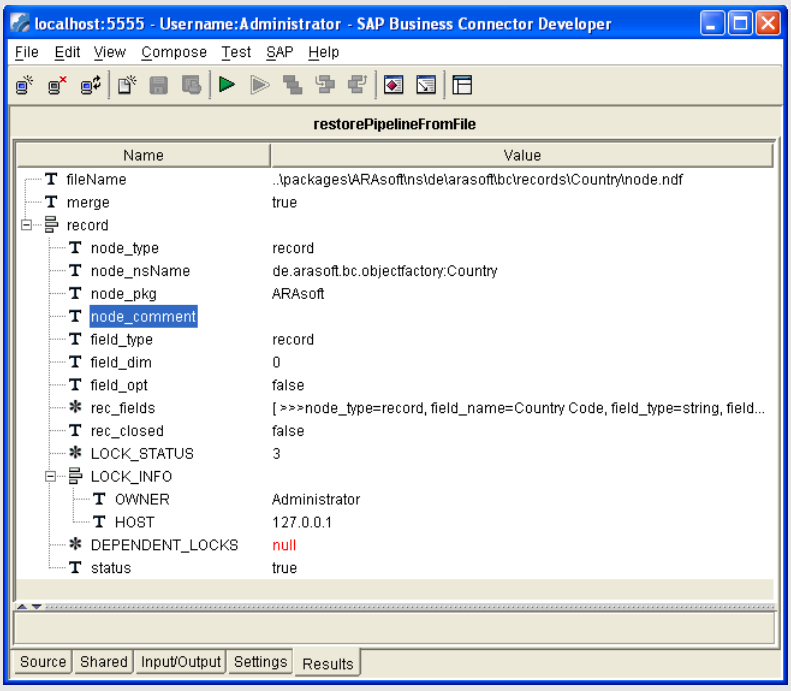
The *getColumnNamesForRecord()* method expects two parameters: the first one is the name of the package in which the record is defined and the second one is the (fully qualified) name of the record itself. See Figure 34 for the correct values in our case.

Figure 37 shows the pipeline (that we specifically created for this purpose) after the call to *pub.flow:restorePipelineFromFile*. Figuring out how

Figure 36 (continued)

```
    Values data = new Values();
    data.put("fileName", path);
    try {
      Service.doInvoke("pub.flow", "restorePipelineFromFile", data);
    }
    catch (Exception ex) {
      throw new ServiceException(ex.getMessage());
    }
    Values v = data.getValues("record");
    Object object = v.get("rec_fields");
    if ( !(object instanceof Vector) )
      throw new ServiceException(path + " is not a flat record.");
    Vector columns = (Vector) object;
    Enumeration e = columns.elements();
    String[] names = new String[columns.size()];
    int i = 0;
    while (e.hasMoreElements()) {
      Values column = (Values) e.nextElement();
      names[i++] = column.getString("field_name");
    }
    return names;
  }
```

Figure 37 The Pipeline After Calling pub.flow:restorePipelineFromFile



to get at the field names (defined in `rec_fields`) was not exactly easy (and I got help from one of the developers of SBC), but the result (Figure 36) works just fine.

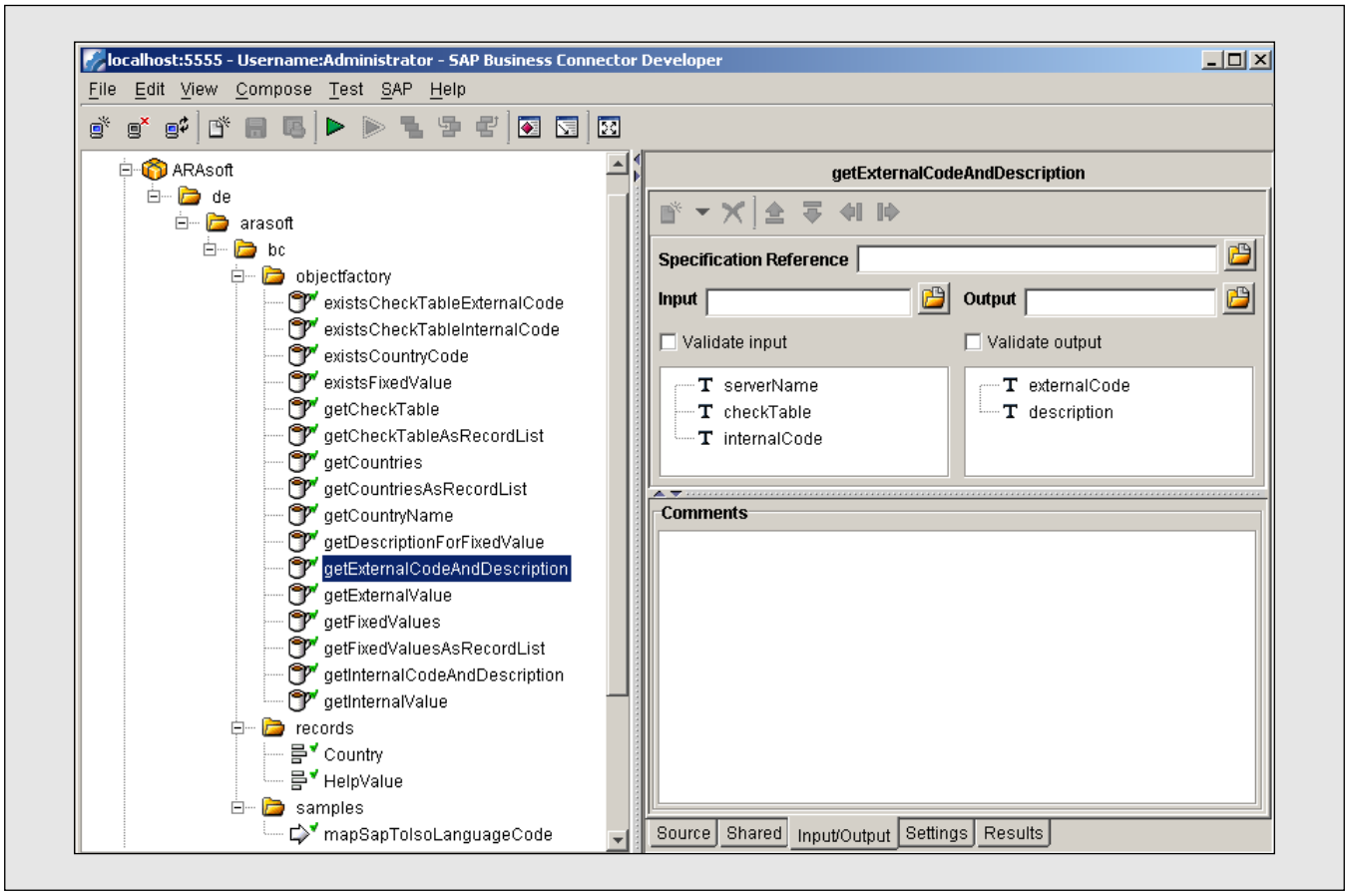
If you believe that the functionality offered by the *getColumnNamesForRecord()* method should not be restricted to the services in the folder that contains the shared method, please see the appendix to this article. It contains the source code for a service that expects the package name in which the record is defined in input variable `packageName` and the fully qualified record name in input variable `recordName`. The service creates a string list with the column names in output variable `columnNames`. When you want to use this service in SBC, make sure that you define `java.util.*` and `java.io.File` in the imports section of its shared code tab.

Using Java Services in Flow Services

Flow services can invoke Java services and vice versa. Let us wind down with an example of a flow service that invokes two Java services in order to change a language code returned by a BAPI from the internal to the external format.

BAPIs mostly use the internal format of those fields that have an internal and an external representation. Examples are language-dependent codes (e.g., document type of a sales order) and master data keys (e.g., customer numbers). SAPGUI automatically invokes the conversion exits defined for those fields, but most BAPIs do not. If you link an external system to SAP using BAPIs, you will have to do the conversion yourself.

Figure 38 The Interface of Service getExternalCodeAndDescription



Our scenario for this example is that you have called a BAPI that, amongst other data, returns a language code. This code will be in the internal, 1-byte SAP format. The external partner system in all likelihood does not understand this format, so you need to translate the 1-byte code into its ISO standard, 2-byte equivalent. Service *getExternalCodeAndDescription* (**Figure 38** shows its interface and **Figure 39** lists the source code) can be used to translate the internal code for a check table (`T002` for language codes) into its associated external code, as well as retrieve the description text for the code. Our flow service will also utilize the *existsCheckTableInternalCode* service to verify that

Figure 39 Service getExternalCodeAndDescription

```
public static final void getExternalCodeAndDescription (IData pipeline)
                         throws ServiceException {
  IDataCursor pipelineCursor = pipeline.getCursor();

  String serverName = IDataUtil.getString
                                (pipelineCursor, "serverName");
  if ( serverName == null || serverName.equals("") ) {
    throw new ServiceException("serverName must be specified.");
  }
  serverName = serverName.toUpperCase();

  String code = IDataUtil.getString(pipelineCursor, "internalCode");
  if ( code == null || code.equals("") ) {
    throw new ServiceException("internalCode must be specified.");
  }

  String checkTable = IDataUtil.getString
                                (pipelineCursor, "checkTable");
  if ( checkTable == null || checkTable.equals("") ) {
    throw new ServiceException("checkTable must be specified.");
  }
  checkTable = checkTable.toUpperCase();

  ObjectFactory bof = getObjectFactory(serverName);
  try {
    GenericItem item =
      bof.getSapData()
         .getGenericCollectionForCheckTable(checkTable)
         .getInternalItem(code);
    IDataUtil.put(pipelineCursor, "externalCode",
                  item.getExternalCode());
    IDataUtil.put(pipelineCursor, "description",
                  item.getDescription());
  }
  catch (Exception ex) {
    throw new ServiceException(ex.toString());
  }
  finally {
    pipelineCursor.destroy();
  }
}
```

the specified internal code exists (see **Figure 40** for the source code).

Our flow service, called *mapSapToIsoLanguageCode* (shown in **Figure 41**) has defined two input variables (`serverName` and `sapLanguageCode`) and one output variable (`isoLanguageCode`). In the first step, Java service *existsCheckTableInternalCode* is invoked. Its `internalCode` input variable is mapped to `sapLanguageCode`, while the `checkTable` input variable is set to `T002`, the check table for language codes (to verify this, you would double-click the arrow just left of the variable name). Since the `serverName` input variable has the same name in both *mapSapToIsoLanguageCode* and *existsCheckTableInternalCode*, an implicit link is maintained by SBC. Service

Figure 40 Service existsCheckTableInternalCode

```
public static final void existsCheckTableInternalCode (IData pipeline)
                         throws ServiceException {
  IDataCursor pipelineCursor = pipeline.getCursor();

  String serverName = IDataUtil.getString
                                (pipelineCursor, "serverName");
  if ( serverName == null || serverName.equals("") ) {
    throw new ServiceException("serverName must be specified.");
  }
  serverName = serverName.toUpperCase();

  String code = IDataUtil.getString(pipelineCursor, "internalCode");
  if ( code == null || code.equals("") ) {
    throw new ServiceException("internalCode must be specified.");
  }

  String checkTable = IDataUtil.getString
                                (pipelineCursor, "checkTable");
  if ( checkTable == null || checkTable.equals("") ) {
    throw new ServiceException("checkTable must be specified.");
  }
  checkTable = checkTable.toUpperCase();

  ObjectFactory bof = getObjectFactory(serverName);
  try {
    GenericCollection collection =
      bof.getSapData().
          getGenericCollectionForCheckTable(checkTable);
    IDataUtil.put(pipelineCursor, "exists",
                  collection.existsInternalCode(code) ?
                  "true" : "false");
  }
  catch (Exception ex) {
    throw new ServiceException(ex.toString());
  }
  finally {
    pipelineCursor.destroy();
  }
}
```


existsCheckTableInternalCode returns a variable called `exists`, which is checked in the subsequent `BRANCH` step of the flow. If `true` is returned, service *getExternalCodeAndDescription* is invoked to do the actual conversion (**Figure 42**). There is no separate code for `false`, so the final `MAP` step

Figure 41 Flow Service mapSapToIsoLanguageCode, Part 1

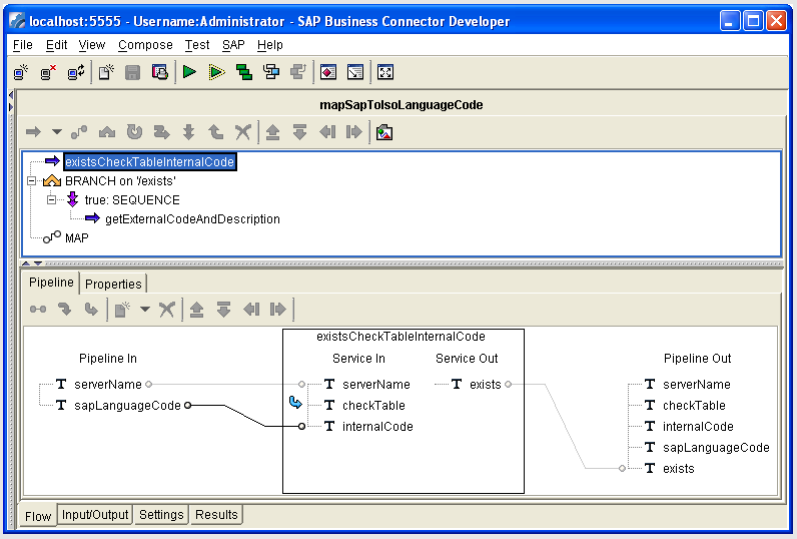


Figure 42 Flow Service mapSapToIsoLanguageCode, Part 2

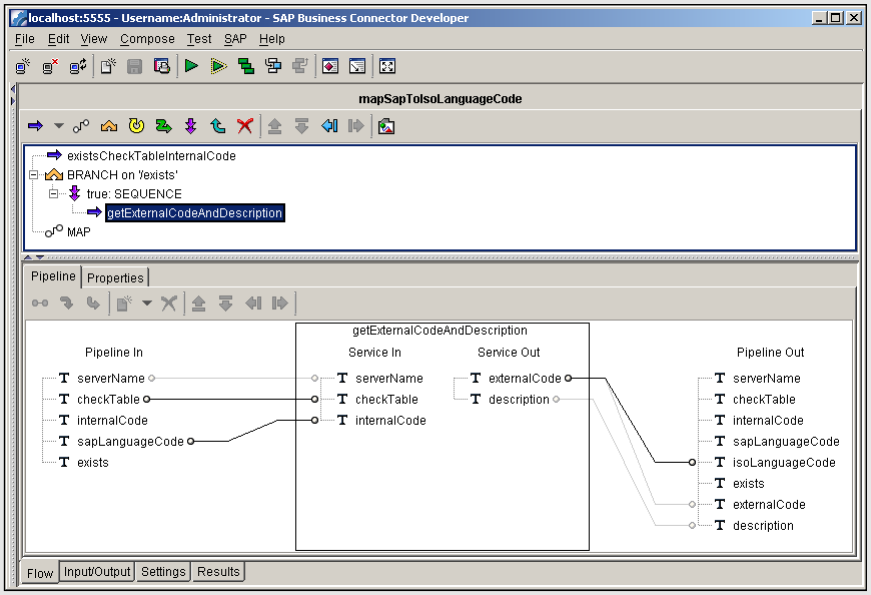
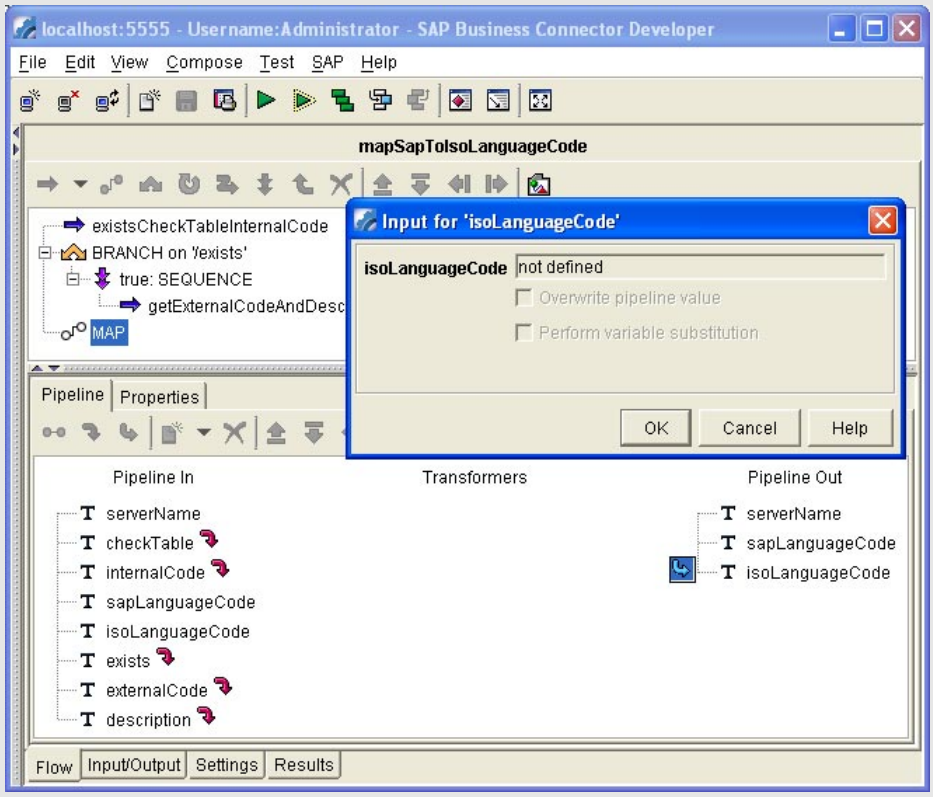


Figure 43 Flow Service mapSapToIsoLanguageCode, Part 3



will be executed next regardless of the outcome of the *existsCheckTableInternalCode* call.

In Figure 42, the `externalCode` output variable of service *getExternalCodeAndDescription* is mapped to the `isoLanguageCode` variable defined as the output variable of the flow service.

In the final `MAP` step (**Figure 43**) several variables in the pipeline are dropped (indicated by the drooping arrows just right of the variable names) in order to clean up the pipeline. The output variable `isoLanguageCode` (on the right side of the lower pane) is assigned the value "not defined". Since the "Overwrite pipeline value" flag (see the pop-up in Figure 43) is not checked for this assignment, it will only be executed if `isoLanguageCode` does not contain a value yet: in other words, if service *getExternalCodeAndDescription* was never called because *existsCheckTableInternalCode* returned `false`. This is all, folks, our flow service is ready to go. **Figure 44** shows test values entered for the

Figure 44 Running Flow Service mapSapToIsoLanguageCode

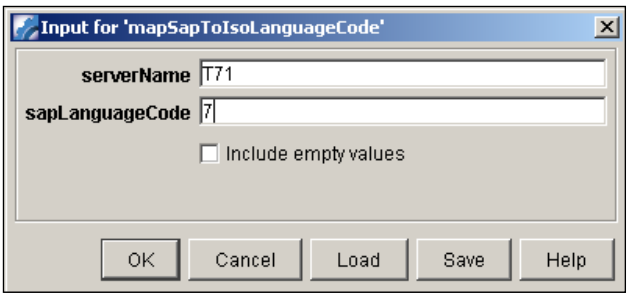
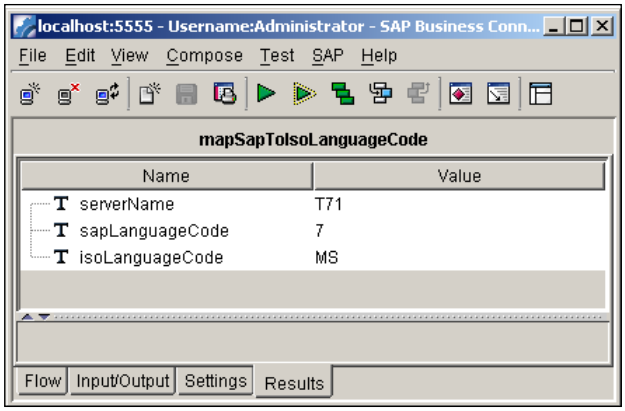


Figure 45 The Result of Running Flow Service mapSapToIsoLanguageCode



service, and **Figure 45** the result of the service. SAP language code "7" is correctly translated into the "MS" ISO code.

Conclusion

The SAP Business Connector is a mature and stable product that is perfect for the XML-enabling of SAP systems like R/3. Many useful services are built in, and you can easily develop your own either using the graphical programming environment (to create flow services) or writing Java code (to create Java services). The choice is yours, just go do it![5]

[5] If you would like a copy of the packages built for this article, please send me an email.

Thomas G. Schuessler is the founder of ARAsoft (www.arasoft.de), a company offering products, consulting, custom development, and training to a worldwide base of customers. The company specializes in integration between SAP and non-SAP components and applications. ARAsoft offers various products for BAPI-enabled programs on the Windows and Java platforms. These products facilitate the development of desktop and Internet applications that communicate with R/3. Thomas is the author of SAP's BIT525 "Developing BAPI-enabled Web Applications with Visual Basic" and BIT526 "Developing BAPI-enabled Web Applications with Java" classes, which he teaches in Germany and in English-speaking countries. Thomas is a regularly featured speaker at SAP TechEd and SAPPHIRE conferences. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years. Thomas can be contacted at thomas.schuessler@sap.com or at tgs@arasoft.de.

Appendix: Service getColumnNamesForFlatRecord

```
public static final void getColumnNamesForFlatRecord (IData pipeline)
                        throws ServiceException {
  IDataCursor pipelineCursor = pipeline.getCursor();

  String packageName = IDataUtil.getString(pipelineCursor,
                                           "packageName");
  if ( packageName == null || packageName.equals("") ) {
    throw new ServiceException("packageName must be specified.");
  }
  String recordName = IDataUtil.getString(pipelineCursor,
                                          "recordName");
  if ( recordName == null || recordName.equals("") ) {
    throw new ServiceException("recordName must be specified.");
  }

  StringBuffer sb = new StringBuffer("..");
  sb.append(File.separatorChar);
  sb.append("packages");
  sb.append(File.separatorChar);
  sb.append(packageName);
  sb.append(File.separatorChar);
  sb.append("ns");
  sb.append(File.separatorChar);
  int offset = sb.length() - 1;
  sb.append(recordName);
  for (int i = offset; i < sb.length(); i++) {
    char c = sb.charAt(i);
    if (c == '.' || c == ':')
      sb.replace(i, i + 1, File.separator);
  }
  sb.append(File.separatorChar);
  sb.append("node.ndf");
  String path = sb.toString();

  Values data = new Values();
  data.put("fileName", path);
  try {
    Service.doInvoke("pub.flow", "restorePipelineFromFile", data);
  }
  catch (Exception ex) {
    throw new ServiceException(ex.getMessage());
  }
  Values v = data.getValues("record");
  Object object = v.get("rec_fields");
  if ( !(object instanceof Vector) )
    throw new ServiceException(path + " is not a flat record.");
  Vector columns = (Vector) object;
  Enumeration e = columns.elements();
  String[] names = new String[columns.size()];
  int i = 0;
  while (e.hasMoreElements()) {
    Values column = (Values) e.nextElement();
    names[i++] = column.getString("field_name");
  }

  IDataUtil.put(pipelineCursor, "columnNames", names);
  pipelineCursor.destroy();
}
```

