Integrating IP telephony with the 3Com SDK

Understanding and using the 3Com IP telephony Web-services solution developer kit

Jon Rush
ISV Business Strategy and Enablement
September 2007

Table of Contents

Abstract

Integrating existing IBM System i applications with the 3Com IP Telephony for IBM System i *product offering is a natural progression from the first step of using the new IP telephony and voice over Internet protocol (VoIP) capabilities on System i to running in parallel with those business applications. This white paper lays out what is needed to make this second phase a reality.*

Introduction

The 3Com IP Telephony for IBM System i product offers an integrated, highly secure and reliable communications and voice over Internet protocol (VoIP) solution that runs on the IBM® System i™ platform, allowing your business applications to run along with new telephony solutions that leverage existing IT infrastructures. This white paper provides guidance on integrating the business processes related to your applications with the telephony capabilities delivered with the 3Com IP Telephony for IBM System i product.

This white paper discusses the following topics:

- What the 3Com Solution Developer Kit (SDK) is
- What development tools are needed to use this product
- How to import and use the SDK in a development tool
- How to understand the coding examples provided in this white paper

Prerequisites

To fully benefit from the information in this white paper, you need to have the following prerequisites:

- A basic understanding of Java™ programming
- A good knowledge of the System i platform and its Java environment
- Familiarity and knowledge of Eclipse-based development-tool environments
- To test the coding samples, a working 3Com IP Telephony solution needs to be installed and available on a System i model with IP telephony phones and hardware
- An integrated development environment for examining and modifying the samples that support IBM JDK 1.5. IBM WebSphere® Application Server Toolkit Version 6.1 that comes with the WebSphere offering was used for the testing done for this white paper. IBM Rational® Application Developer V7 also is a viable choice.
- Apache Axis 2 for Java v1.1.1, which includes the Web Services Description Language (WSDL) to Java tooling (downloadable at http://ws.apache.org/axis2/download/1_1_1/download.cgi)
- A System i model with the IBM i5/OS® V5R4 operating system and the following features:
  - 5722JV1 *BASE IBM Developer Kit for Java
  - 5722JV1 5 Java Developer Kit 1.3
  - 5722JV1 6 Java Developer Kit 1.4
  - 5722JV1 7 Java Developer Kit 5.0
  - 5722JV1 8 Java 2 Platform, Standard Edition (J2SE) 5.0 32 bit
- To run the samples outlined in the section entitled “Migrating and running the sample on System i,” you need to install the System i Tools for Developers PRPQ (5799PTL) (which is downloadable from www14.software.ibm.com/webapp/download/preconfig.jsp?id=2004-08-18+12%3A25%3A25.057448R&S_TACT=104CBW71&S_CMP=&s=).

Overview

The 3Com Web services SDK is a Java toolkit that contains the following components:

- The WSDL files that allow applications to consume and use the services defined in the Web-service definition-language (WSDL) file
- A Web-service client-side JAR file that contains classes for use on the client to invoke the 3Com Web service
- A graphical sample application to test the Web service

The 3Com software is installed and runs in a dedicated System i Linux partition, allowing remote applications to use the following service points, programmatically:

- Call control
  - Initiate a phone call
  - Transfer a call
  - Conduct a conference call
  - Hold a call
  - End a call
- Phone configuration
  - Enable hands-free operation
  - Mute the phone
  - Enable or disable Do Not Disturb (DND)
  - Enable or disable forwarding of voice mail
- Phone status
  - Get the phone state
  - Get the DND state
  - Get the voice-mail state

The SDK itself

The first step is to register with the *3Com Open Networks Partner program* (www.open.3com.com/tcom/). This registration permits you to download the SDK toolkit. After downloading, unzip the file to a directory that is accessible to your workstation. Remember this directory because it is used in subsequent sections relating to the development tools and sample code.

Documentation

The SDK for 3Com IP Telephony contains documentation, including an *SDK User Guide* (in PDF file format) and HTML files that describe each Web-service endpoint with the required input and output parameters. To access the documentation, perform the following steps:

1. Navigate to the **docs** directory that was unzipped onto your workstation, and open the index.html file in the Web browser of your choice. This opens the document tree for all the Web-service capabilities included in the SDK as shown in Figure 1.

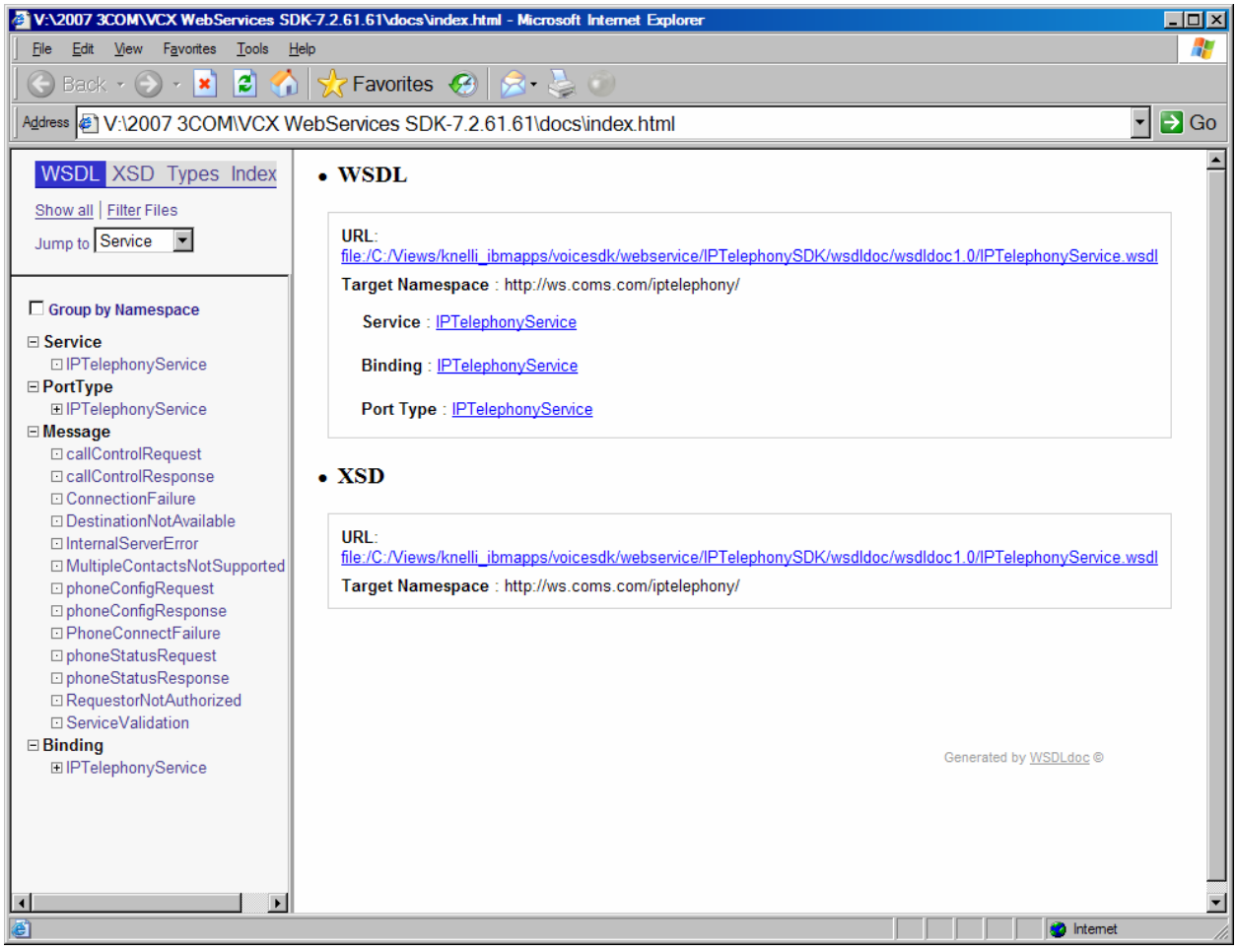

Figure 1. HTML Web-services documentation

2. For example, to understand the Web-service parameters to run a call-control request, in the browser window with the index page of the telephony documentation, click **callControlRequest** in the left-hand navigation frame (labeled number 1 in Figure 2) and then click **callControlRequest** in the main frame (labeled **2** in Figure 2).

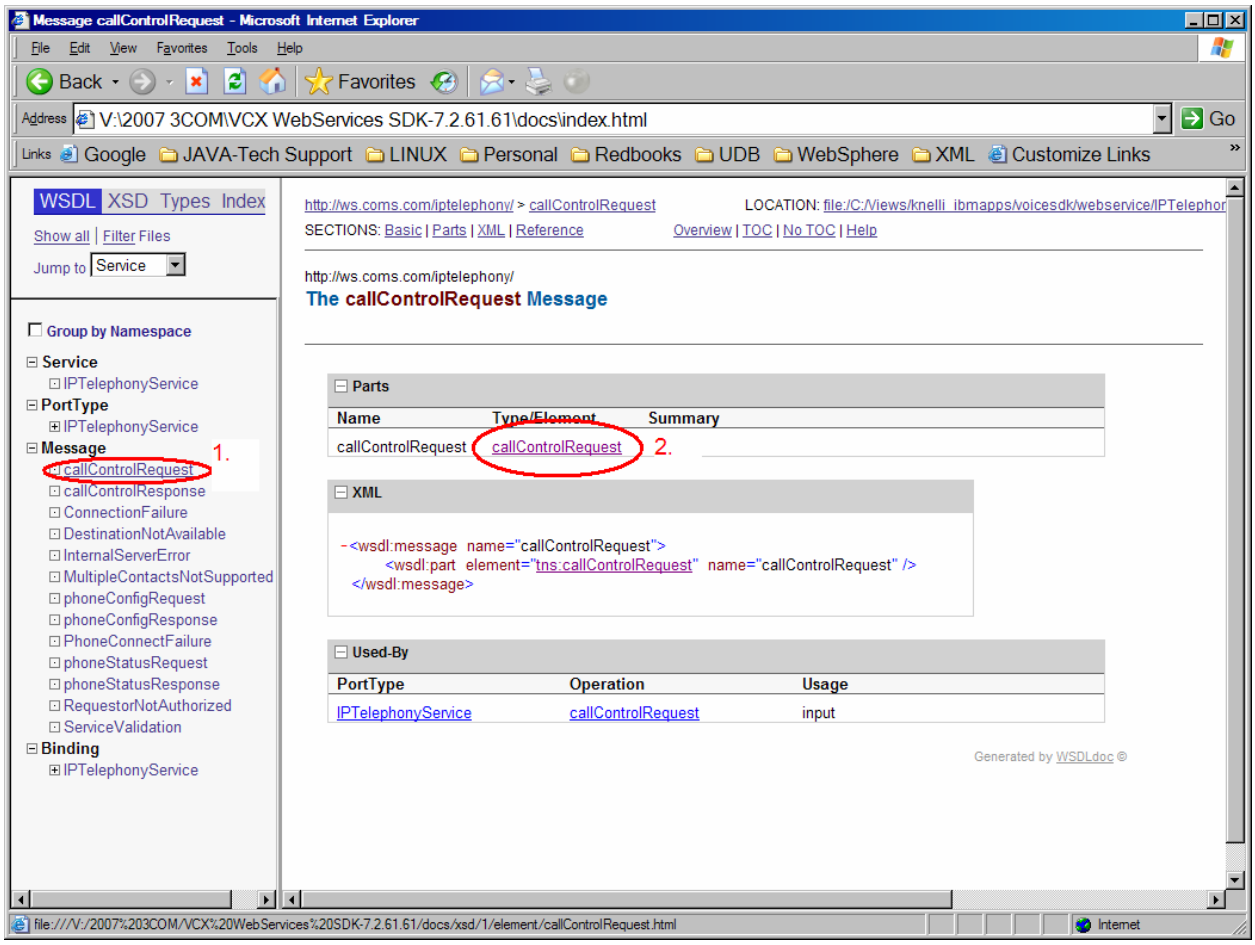

Figure 2. Finding parameters for a call-control request

The documentation now shows you the parameters required to invoke a call-control request of the 3Com Web service. You can see that the first parameter is the actionType, then the credentials, followed by destinationNumber and serviceValidator.

3. To further determine what action types are available, click **CallControlActionType** in the main frame, as shown in Figure 3.

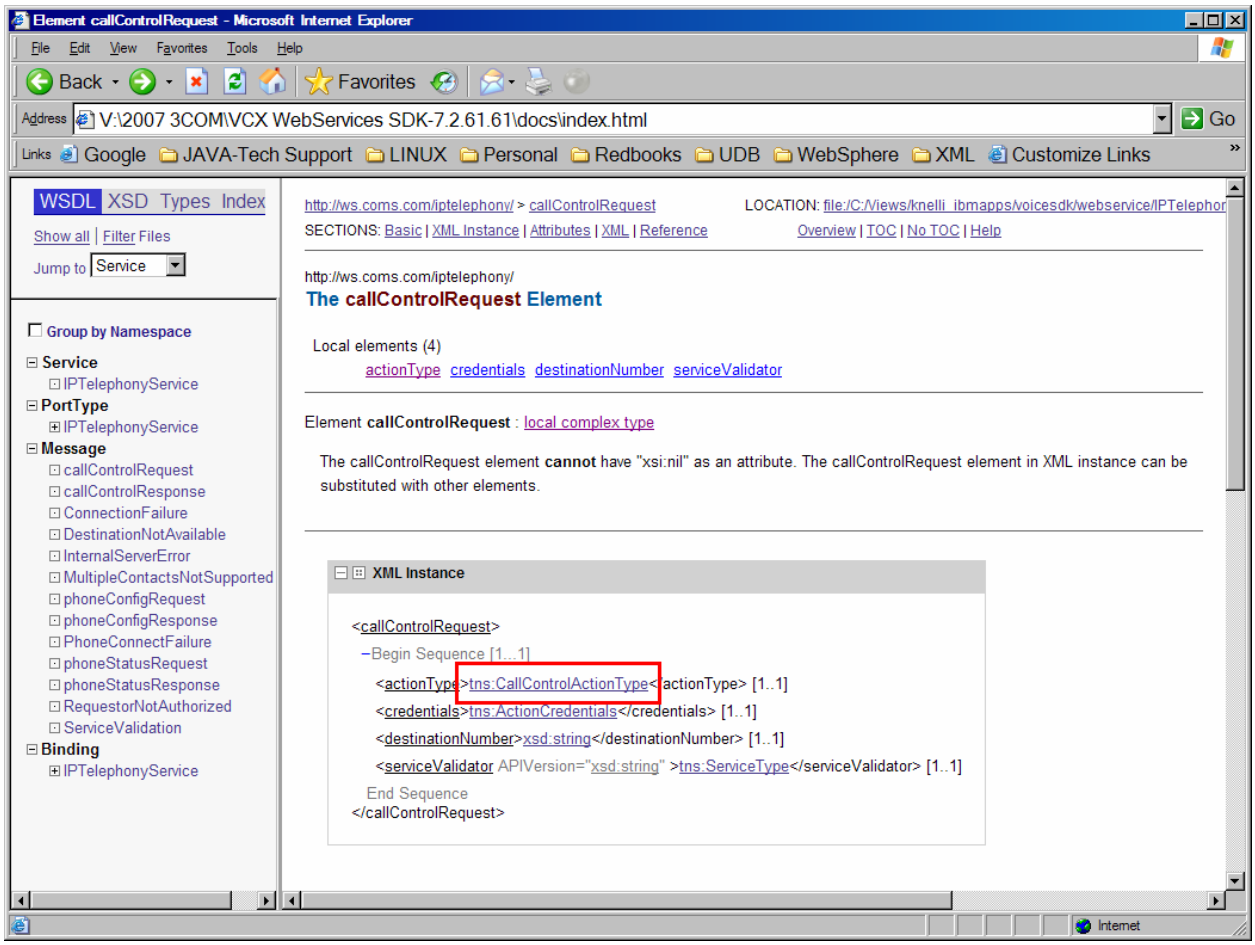

Figure 3. Selecting CallControlActionType

As you can see, the allowable action types include makeCall and transferCall. Figure 4 shows all the allowable action types.

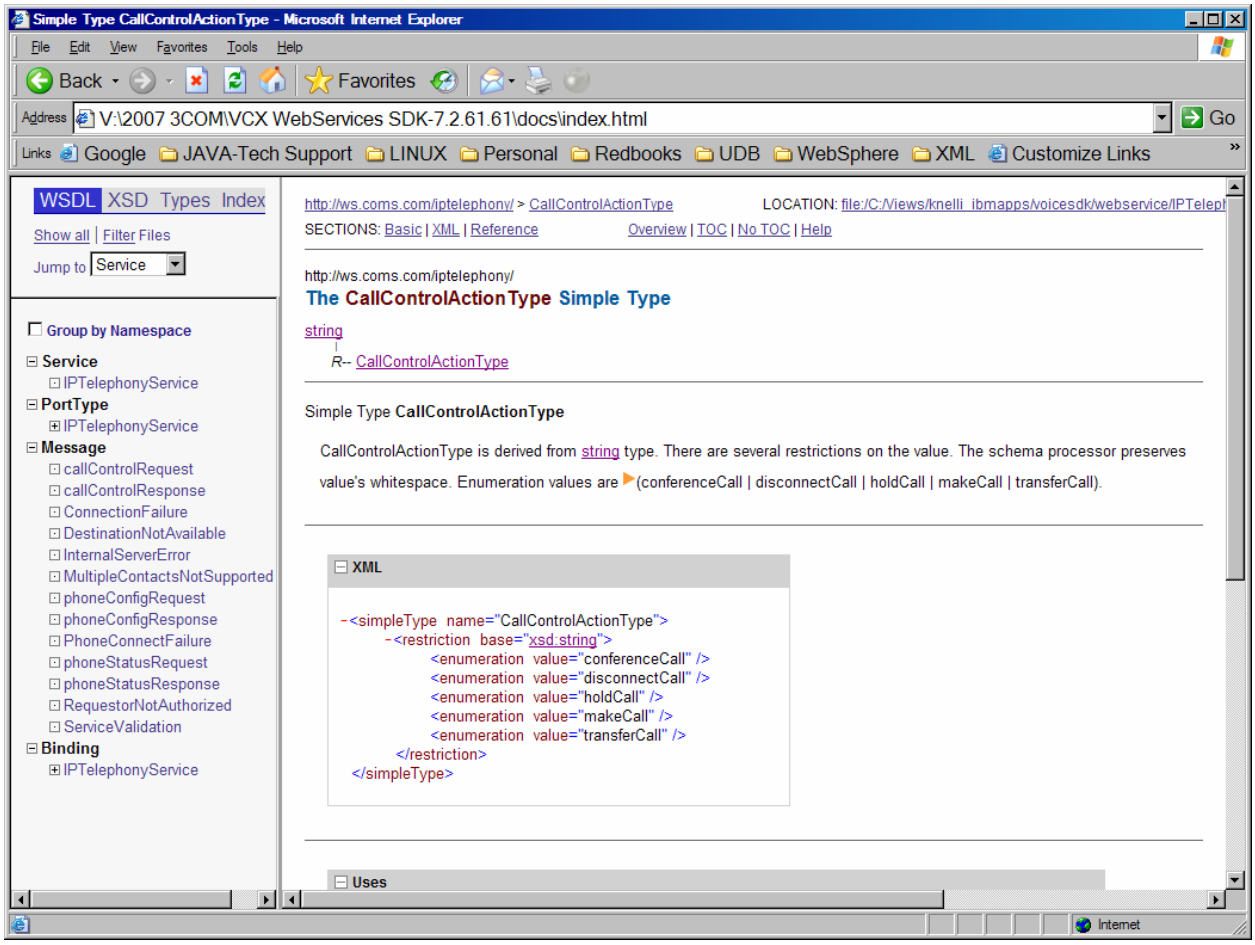

Figure 4. Allowed action types

This documentation allows you to determine what parameters are required for any of the Web-service endpoints provided by the 3Com IP Telephony Web service.

The IP telephony WSDL

As discussed in this section, the 3Com SDK lets you access the specific capabilities exposed to control calls and configure the 3Com IP telephones, as well as to get the state of those phones. These capabilities are exposed externally through the IPTelephony.wsdl XML file. The following section describes the steps needed to get this toolkit into a development tool that you can use to programmatically integrate and build applications that exploit the IP telephony services.

Setting up the Java IDE

You can import the SDK into the integrated development environment (IDE) of your choice, but for this white paper, the WebSphere Application Server Toolkit IDE is used for building the samples. Note that the IDE must support the IBM JDK 1.5 runtime environment. Rational Application Developer version 6 or IBM WebSphere Development Studio Client V5 do not support JDK 1.5 and do not allow the 3Com SDK samples to run. IBM Rational Application Developer (RAD) V7 (and, therefore, WebSphere Development Studio Client V7, which is based on RAD V7 and which has System i extensions) does support the IBM JDK 1.5 runtime environment. It should therefore work, although it was not used for this white paper and has not been tested with the 3Com SDK.

Importing the SDK

Importing the 3Com SDK into one of the IDEs involves the following processes:

- Creating a new Java project
- Importing the SDK from the file system where you unzipped the downloaded toolkit
- Updating the project properties to use the JAR files included in the project's Java build path
- Making a package name correction

To validate the success of the importing effort, you can run the graphical sample that is provided with the SDK (as you will see later in this white paper):

1. Create a new Java project in the IDE, as shown in Figure 5 through Figure 8. For illustrative purposes in this document, WebSphere Application Server Toolkit V6.1 is used.

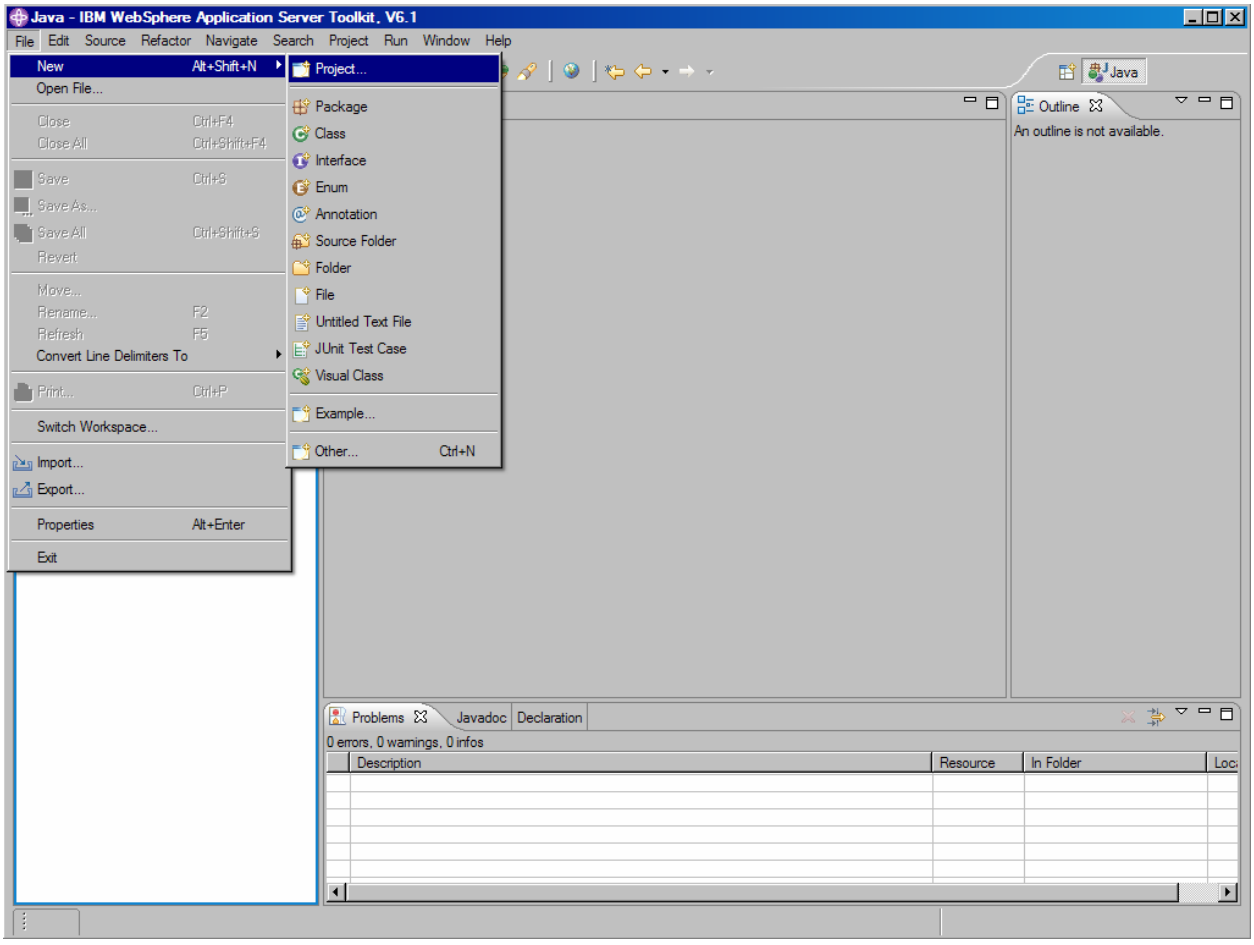

Figure 5. Creating a new project

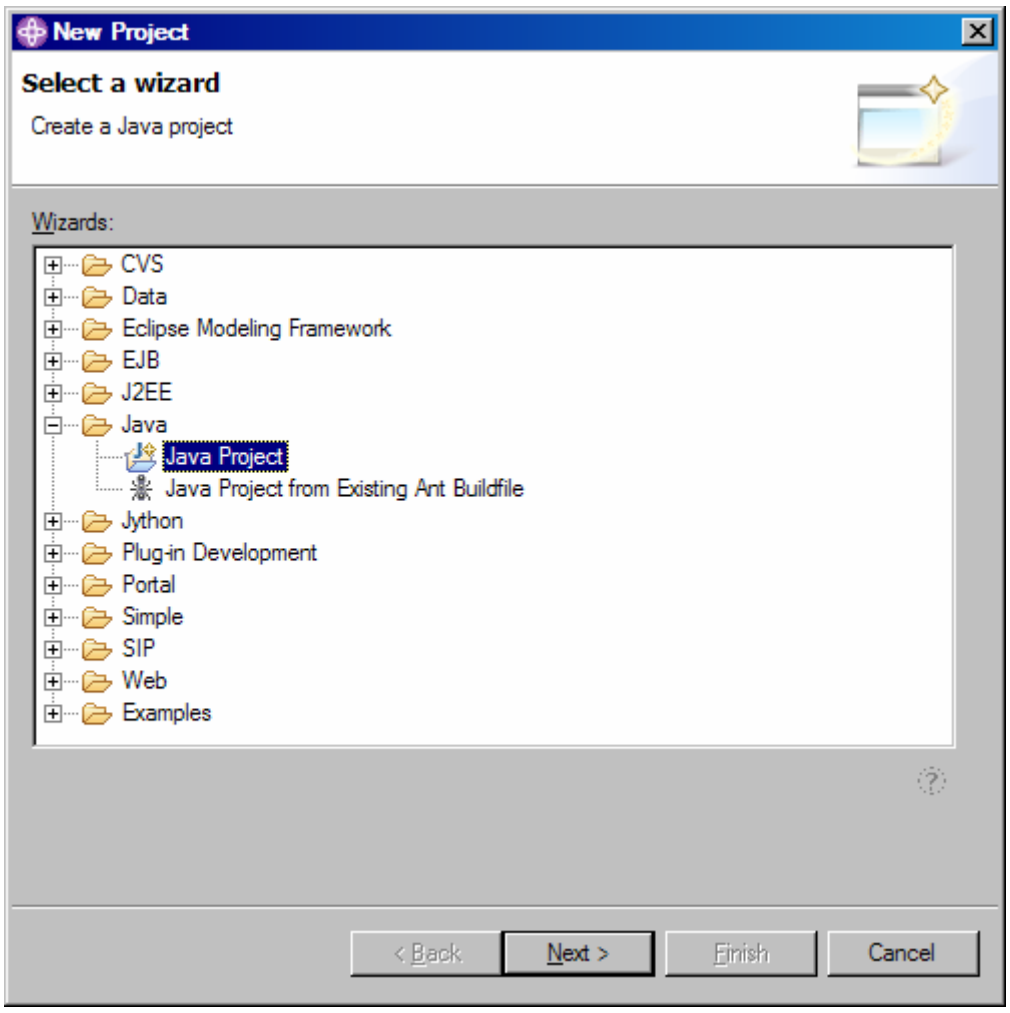

Figure 6. Selecting the Java project

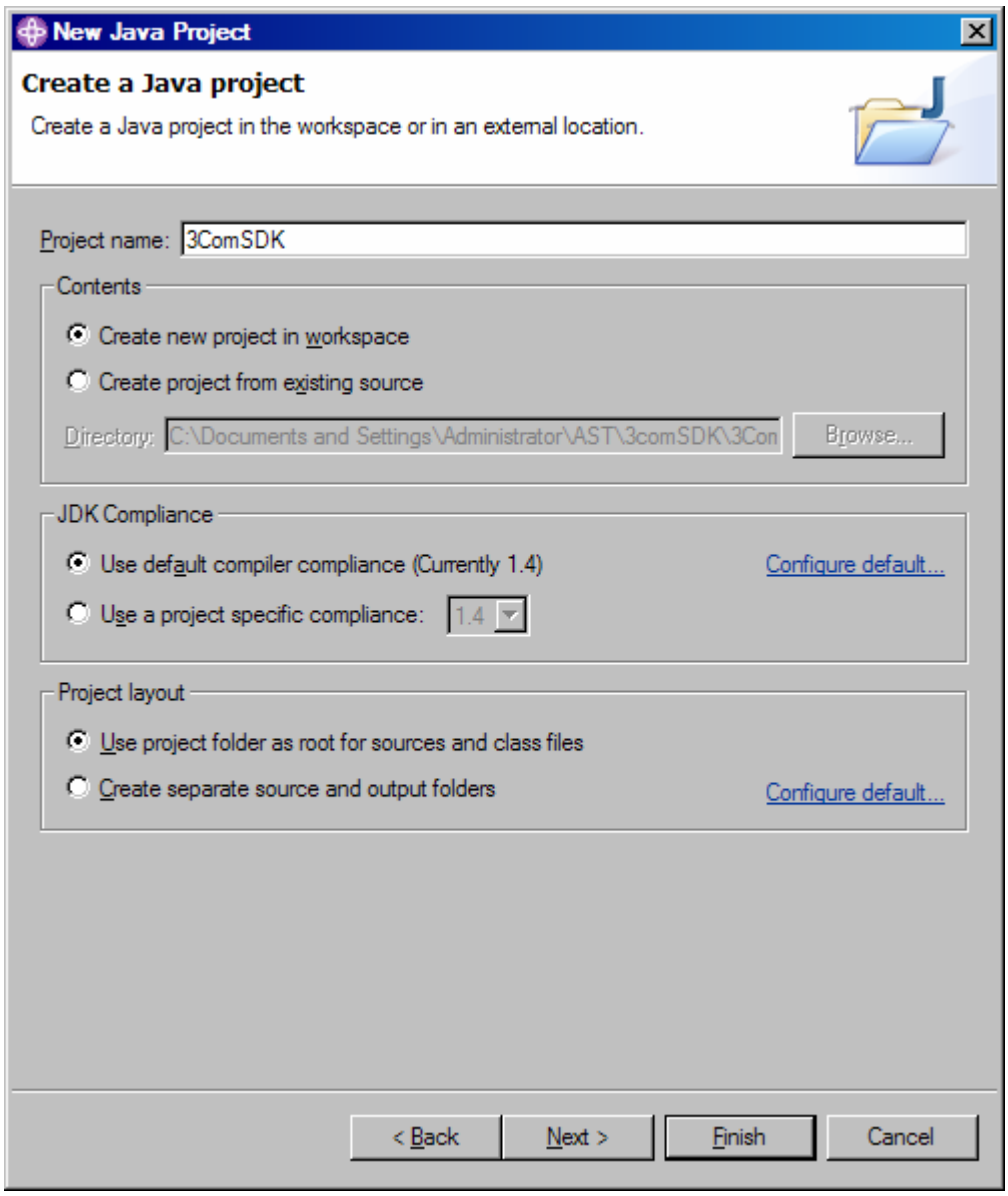

Figure 7. Selecting the Java project (continued)

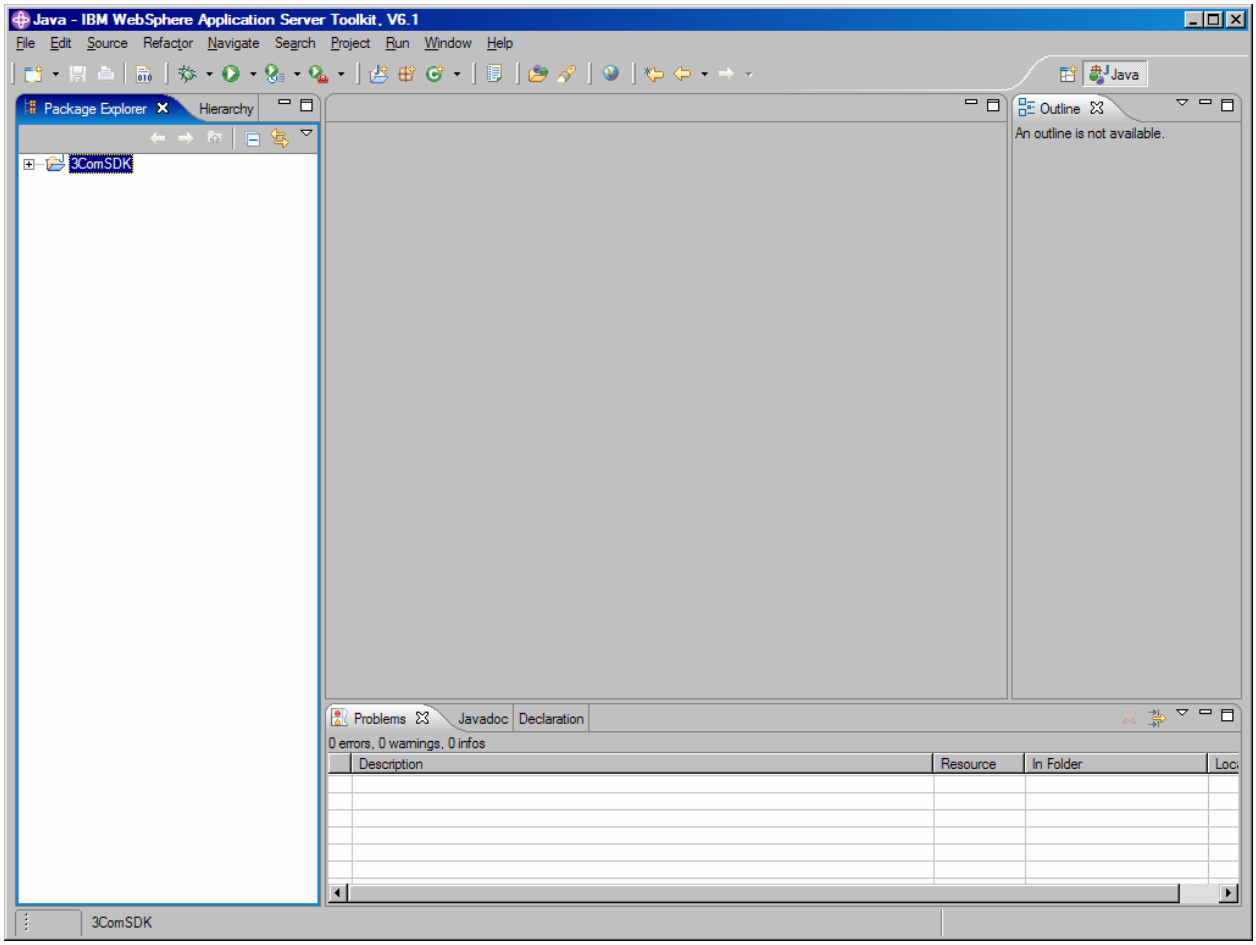

Figure 8. Completing the new Java project

2. After the new, empty project is completed, import the SDK, as shown in Figure 9 and Figure 10.

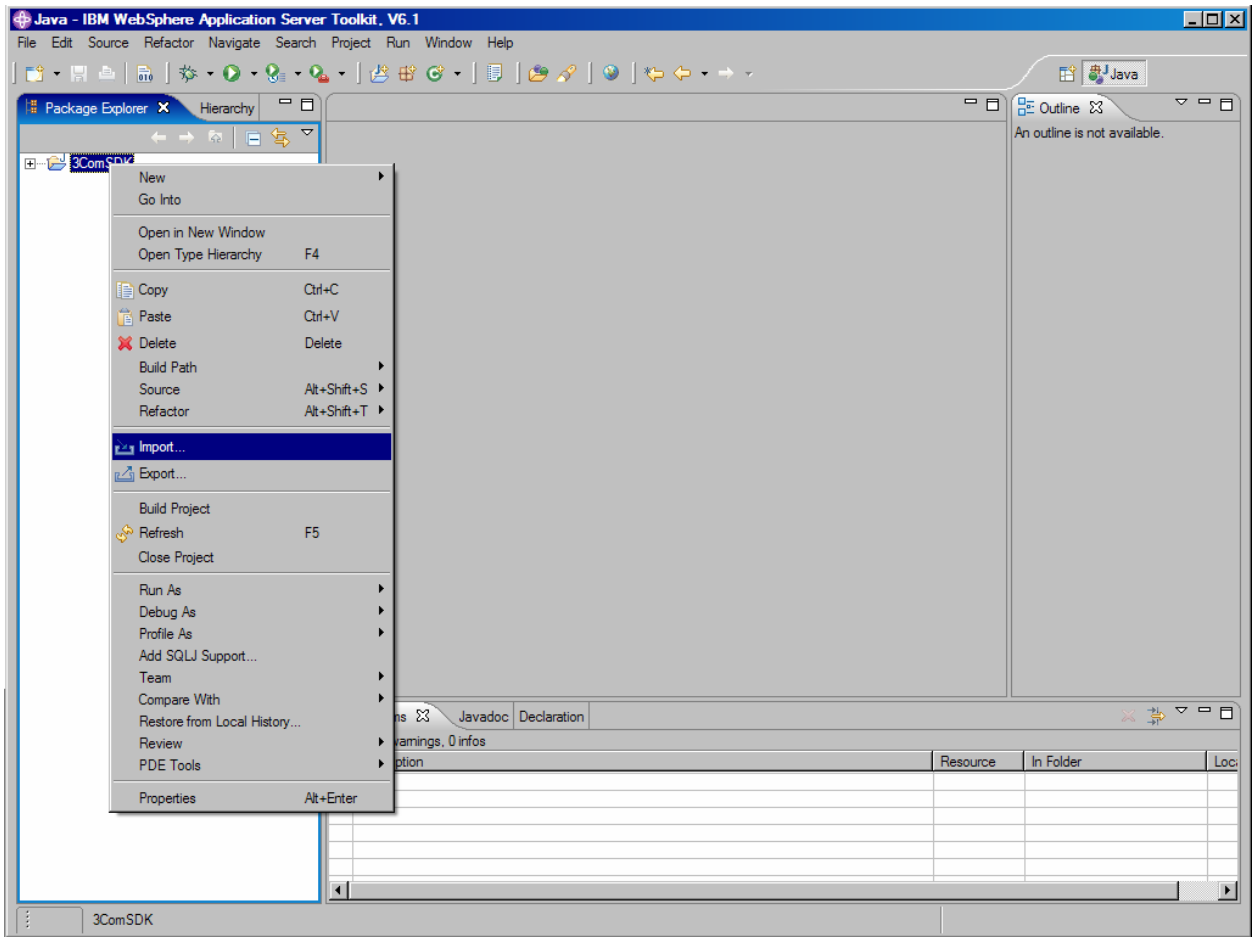

Figure 9. Selecting to import the SDK

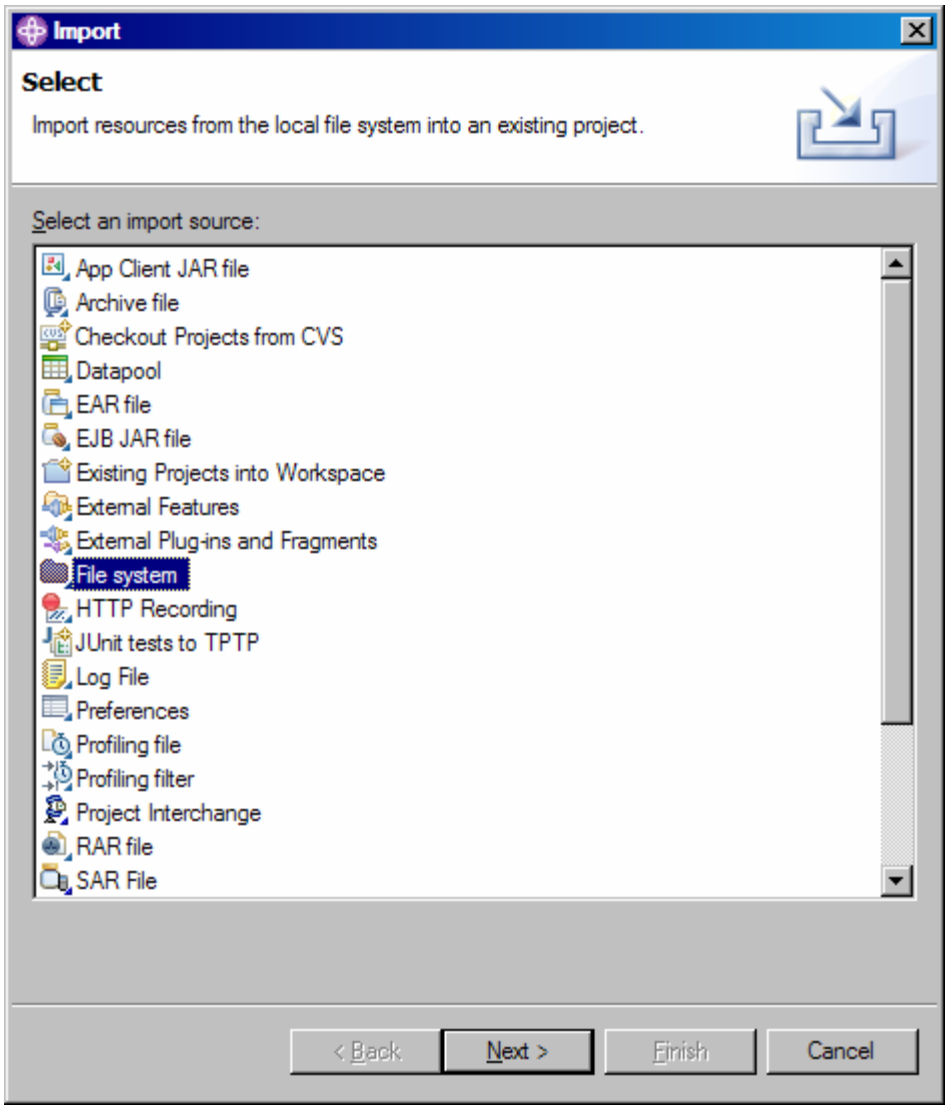

Figure 10. Selecting an import source from the file system

3. Recalling where the SDK zip file was placed in your directory tree, click **Browse** (see Figure 11).

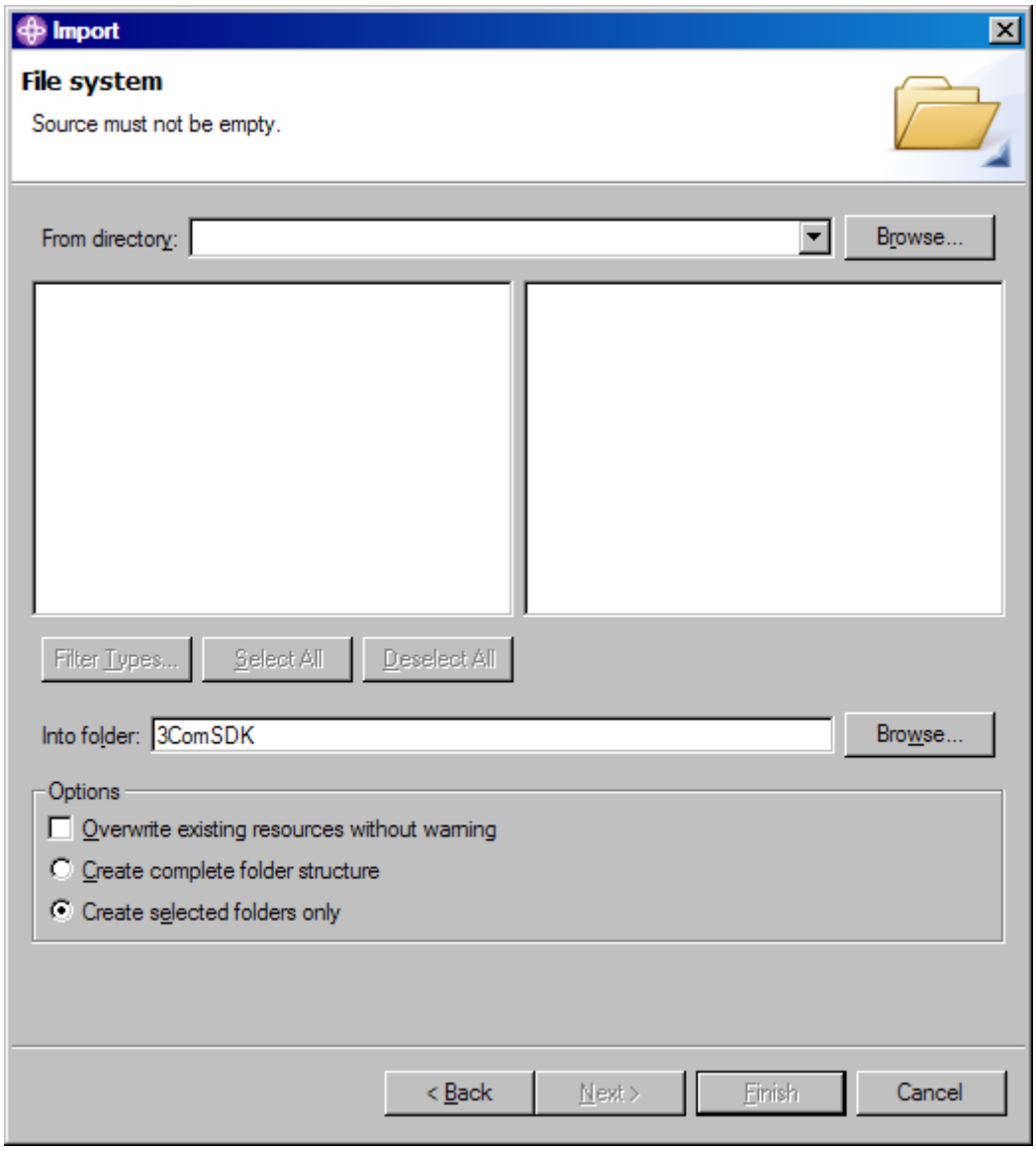

Figure 11. Importing from the file system

4. Select the directory where you placed the 3Com Web service SDK in the workstation (as shown in Figure 12). In this example, **VCX Webservices SDK-7.2.61.61** is selected) and then click **OK**.

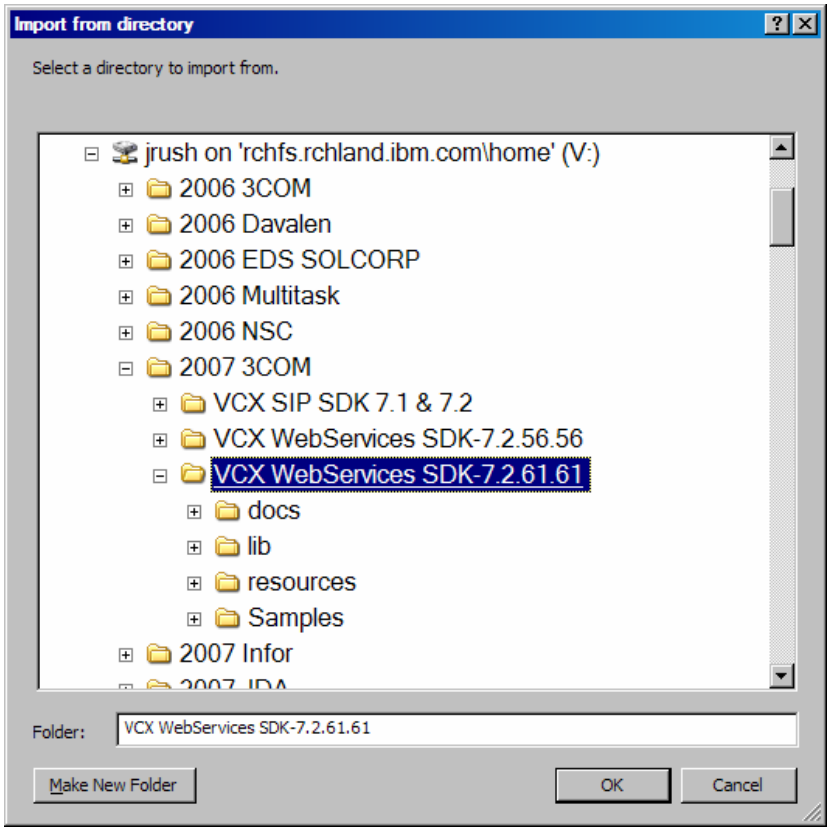

Figure 12. Directory where SDK was unzipped

5. In the window that is shown in Figure 13, expand the directory and click **lib** to select all JAR files from the SDK.
6. Click **resources** to select the WSDL files from the SDK (see Figure 13).
7. Expand the **Samples** directory and select the **java** subdirectory as well as the **build**, **pathrefs**, and **properties** XML files (see Figure 13).
8. Ensure that the **Create selected folders only** option is selected. Then click **Finish** (see Figure 13).

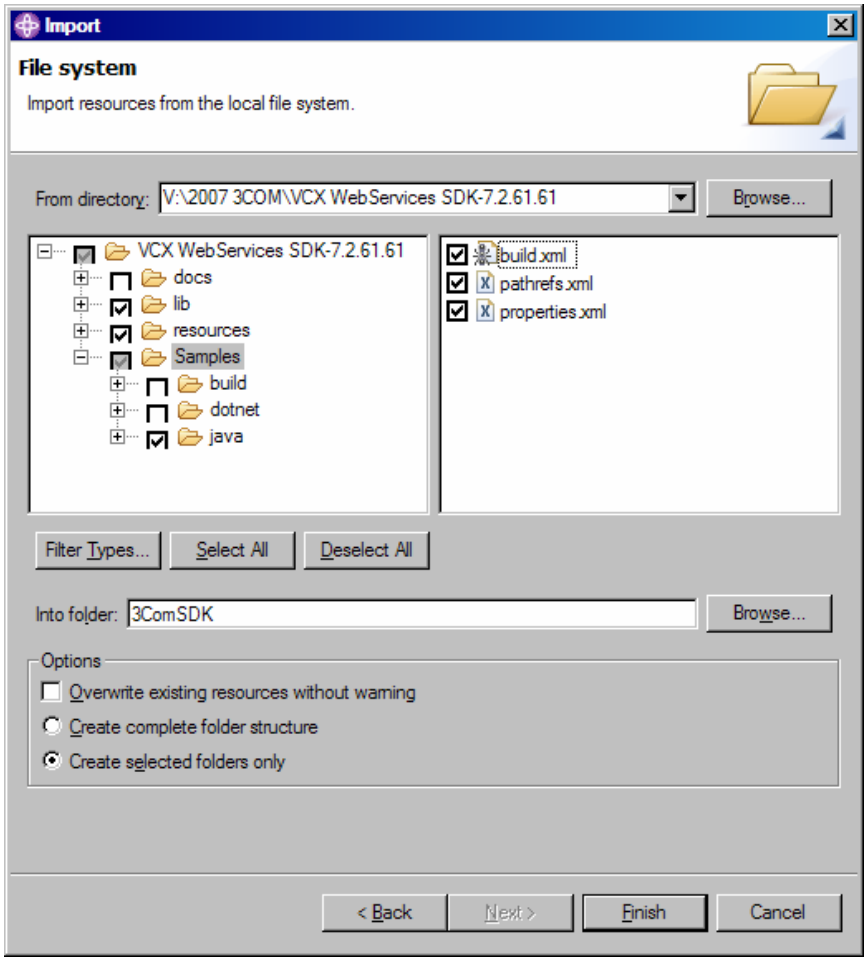

Figure 13. Selecting an artifact for the import

9. After the import has been completed, expand the project and you will see a workspace similar to that in Figure 14.

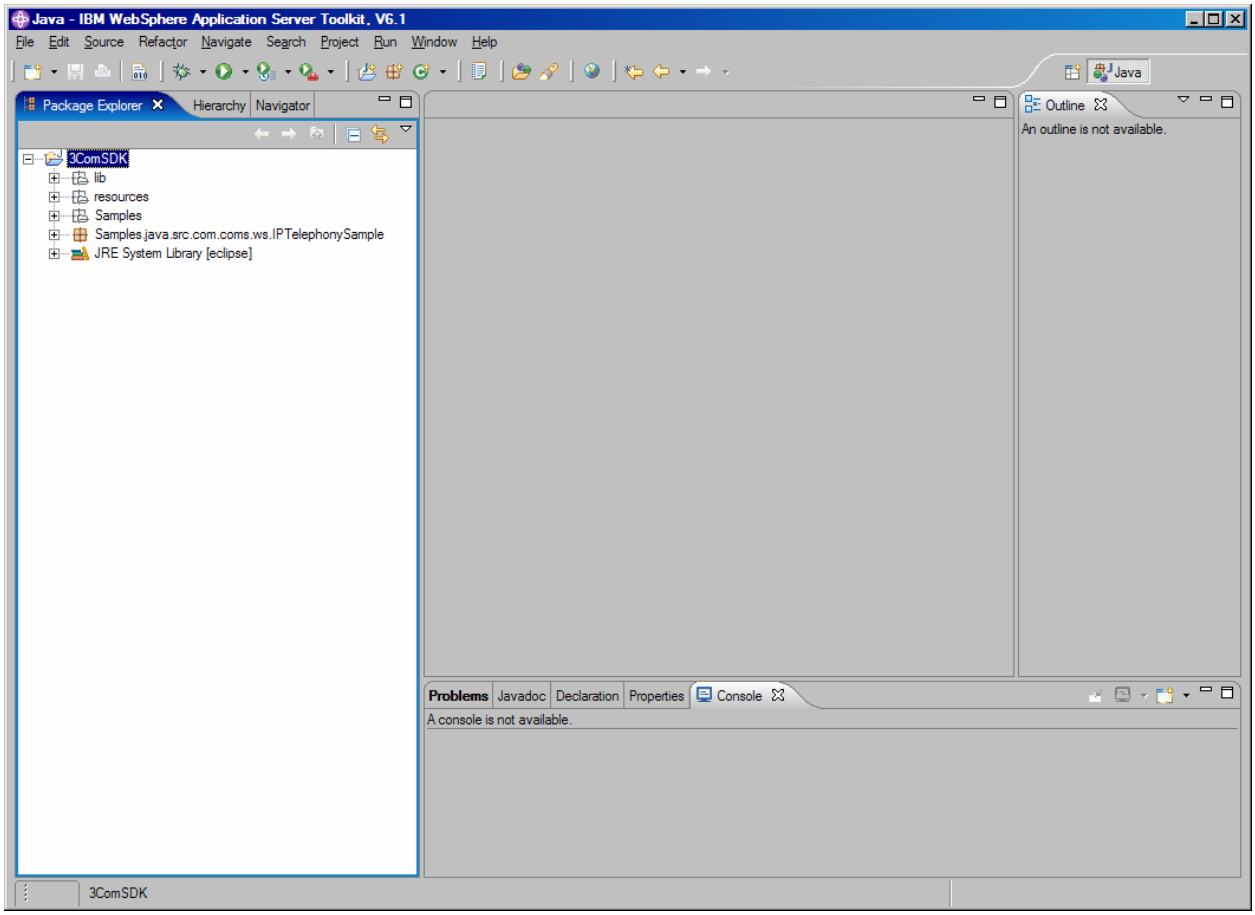

Figure 14. Workspace after import

10. Next, move the XML files under the root of the project, as shown starting in Figure 15:

    a. Expand the **Samples** project and select all the XML files.
    b. Right-click and select **Refactor** and then select **Move**.

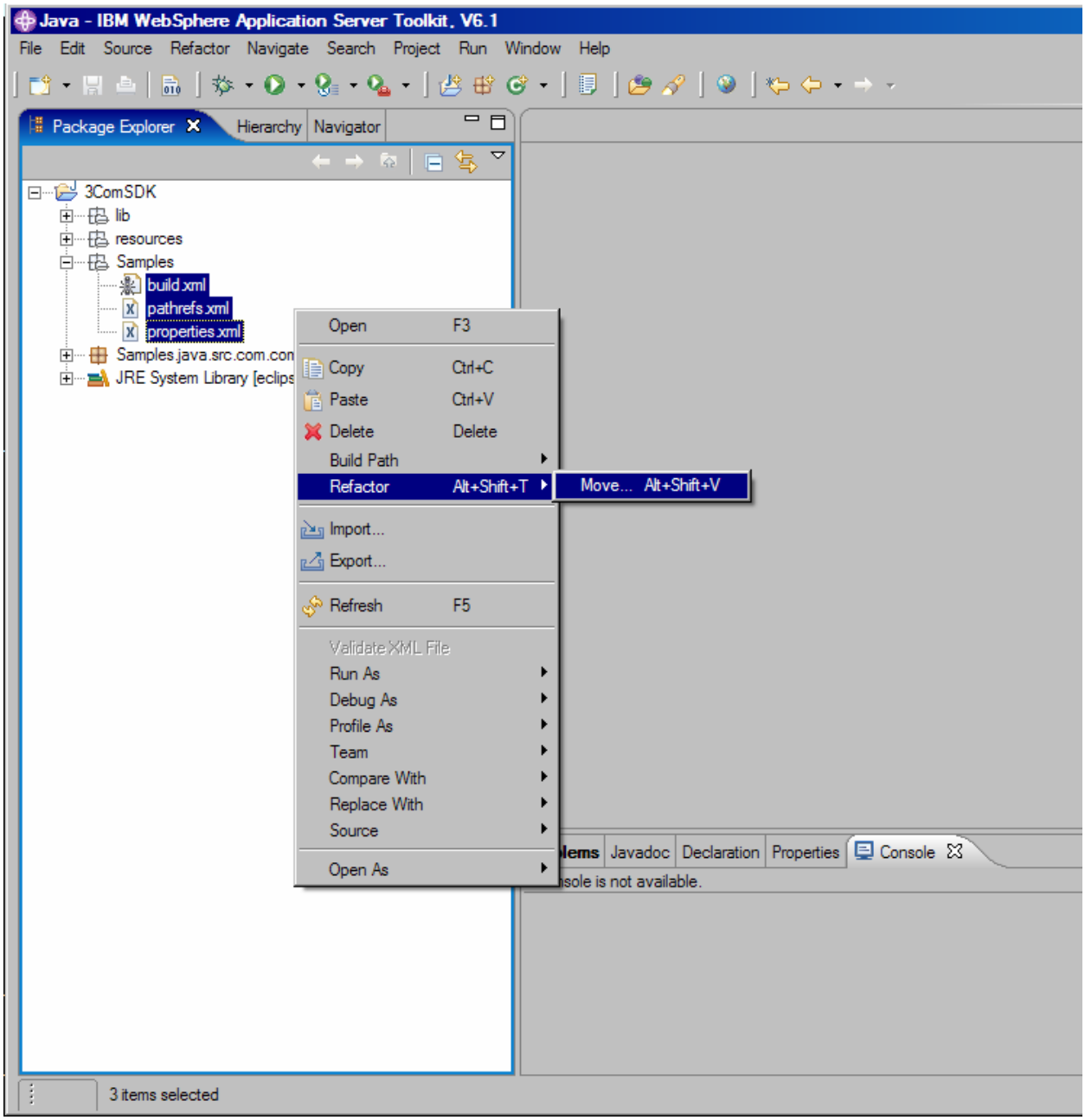

Figure 15. Moving XML files

c. Select the destination for the three XML files that you just selected to move. (In this case, the destination is the 3Com IP Telephony sample directory.) Then, click **OK** (see Figure 16).

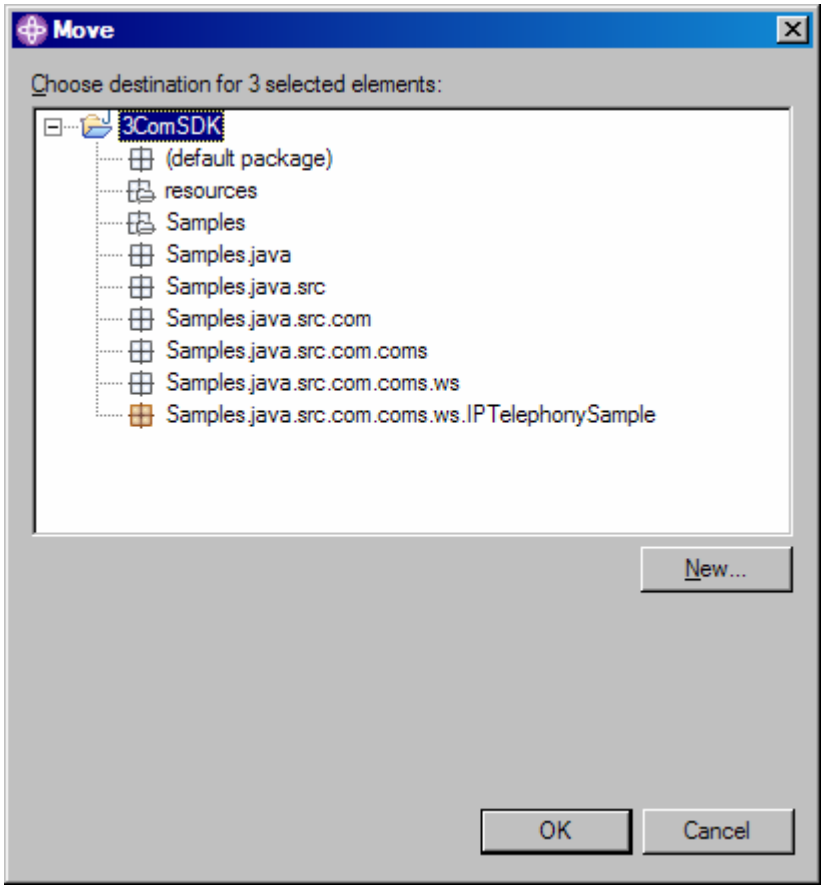

Figure 16. Selecting the destination for the project

Figure 17 shows how the project looks after moving the XML files into the 3Com IP Telephony sample directory.

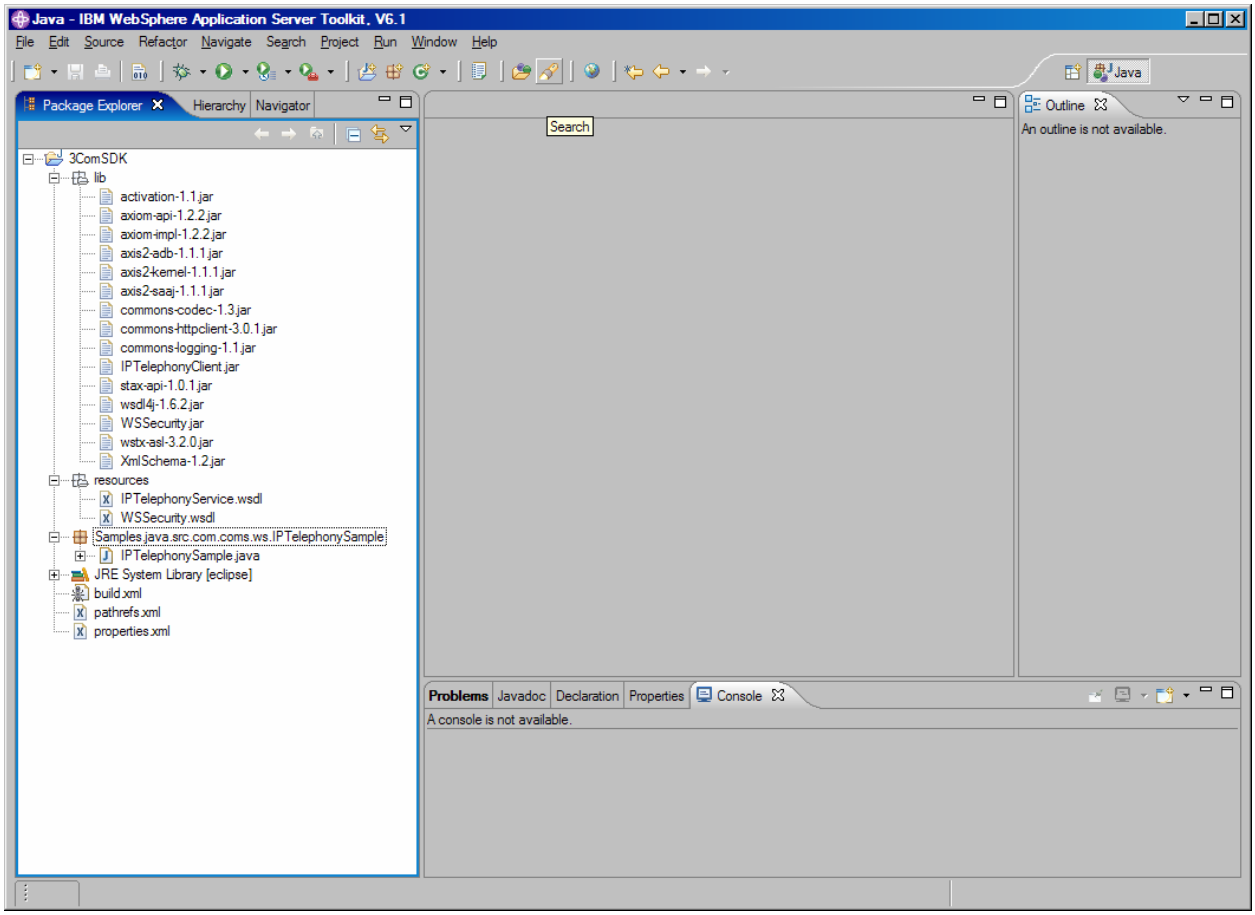

Figure 17. Project after moving XML files

You must put the JAR files that were imported to the lib directory in the project into the project's Java build path so that the sample program can find the required classes that allow the samples to run. Steps to do this start with Figure 18.

11. Right-click the **3ComSDK** directory and select **Properties** (see Figure 18).

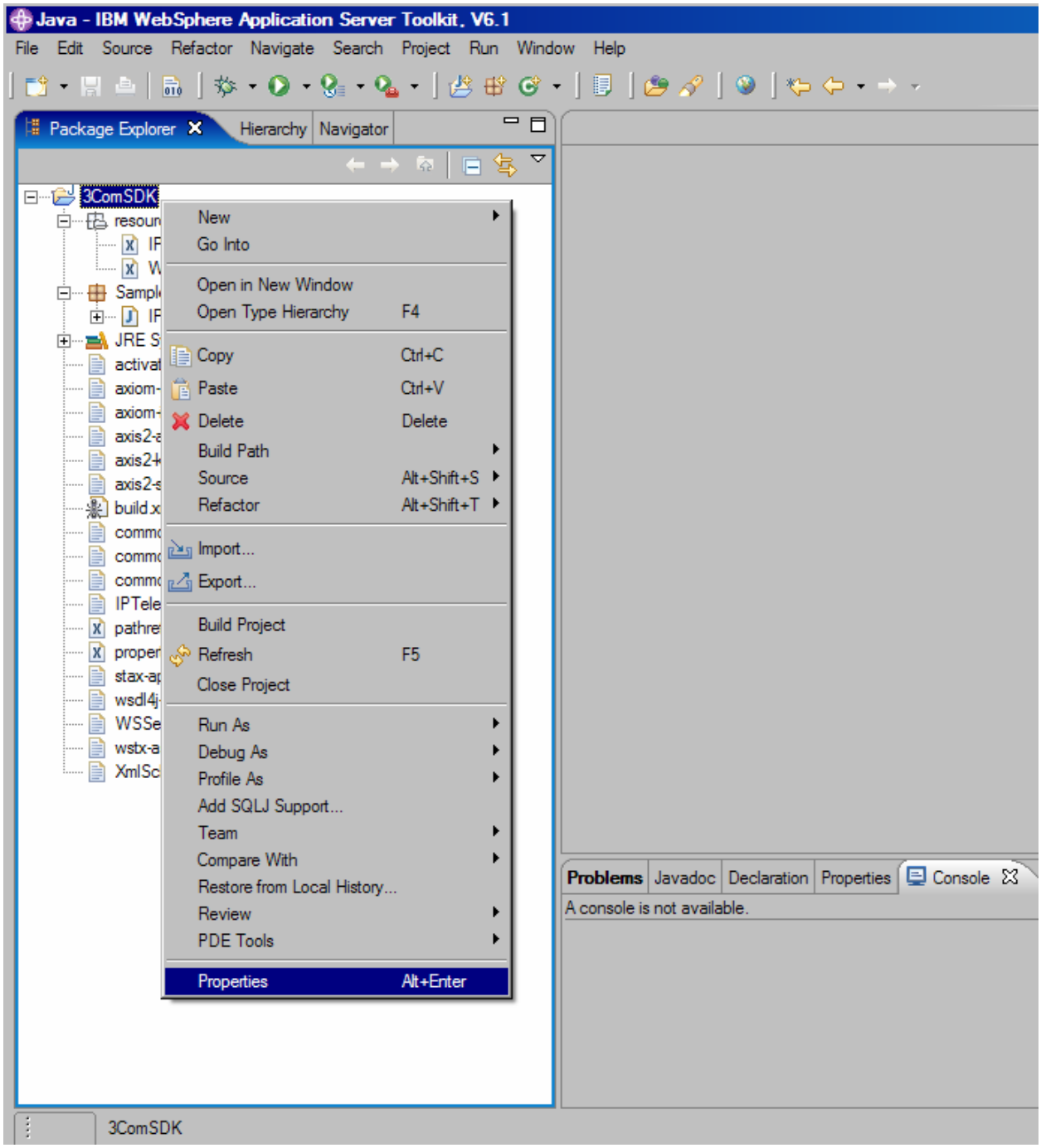

Figure 18. Changing project properties

12. From the *Properties for 3ComSDK* screen (see Figure 19), click **Java Build Path** and then select **Add JARs**.

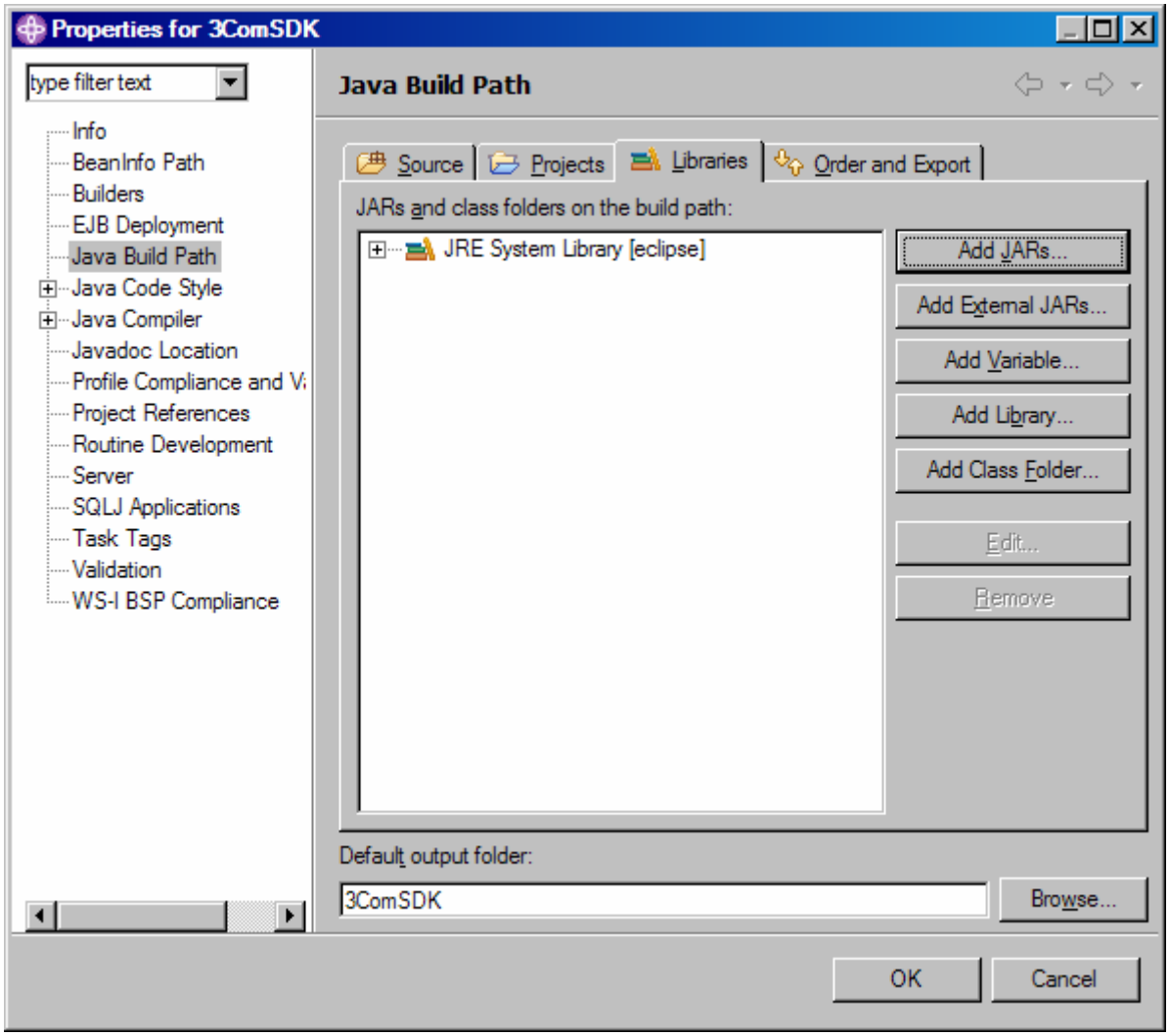

Figure 19. Adding JARs

13. Select the JAR archives that you want to add to the build path. In this example, all the JAR archives are selected (see Figure 20). Then click **OK**.

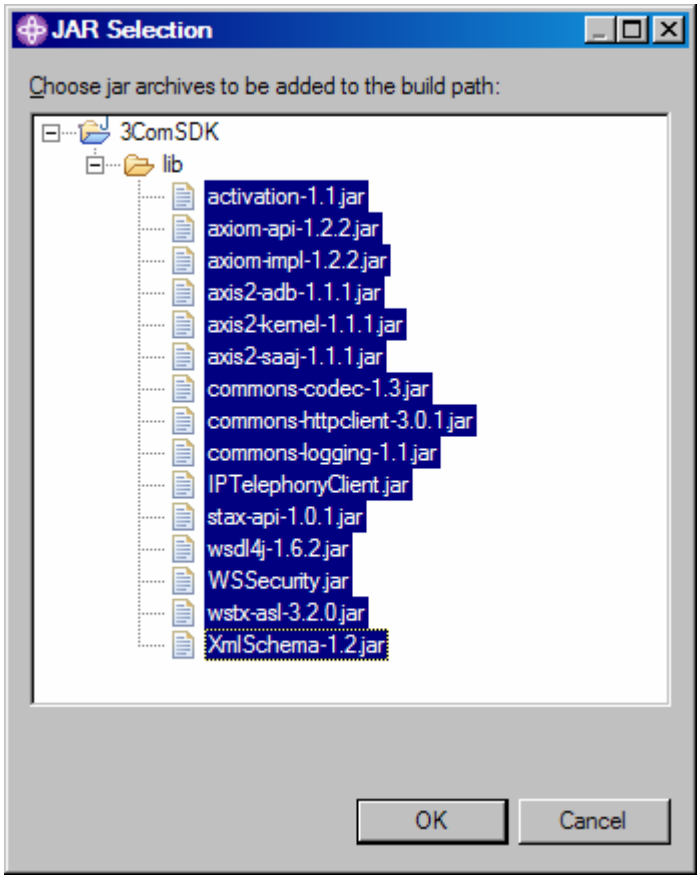

Figure 20. Selecting all JAR archives to add them to the build path

14. Review the newly displayed Java build path to see that all the JARs and class folders are included (see Figure 21). Then click **OK**.

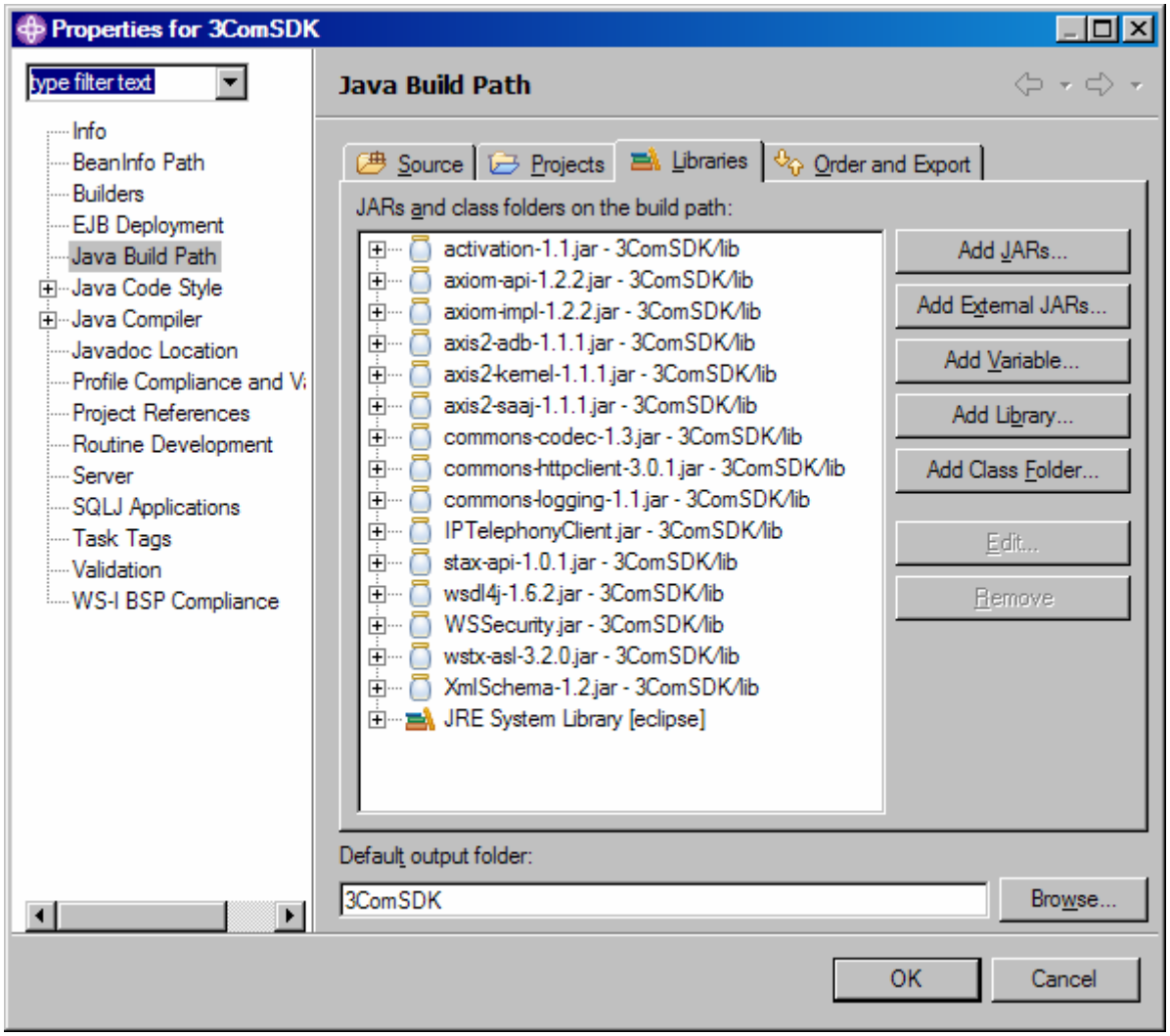

Figure 21. Reviewing the new Java build path

Figure 22 shows what the project looks like after you have successfully added the 3Com JARs.

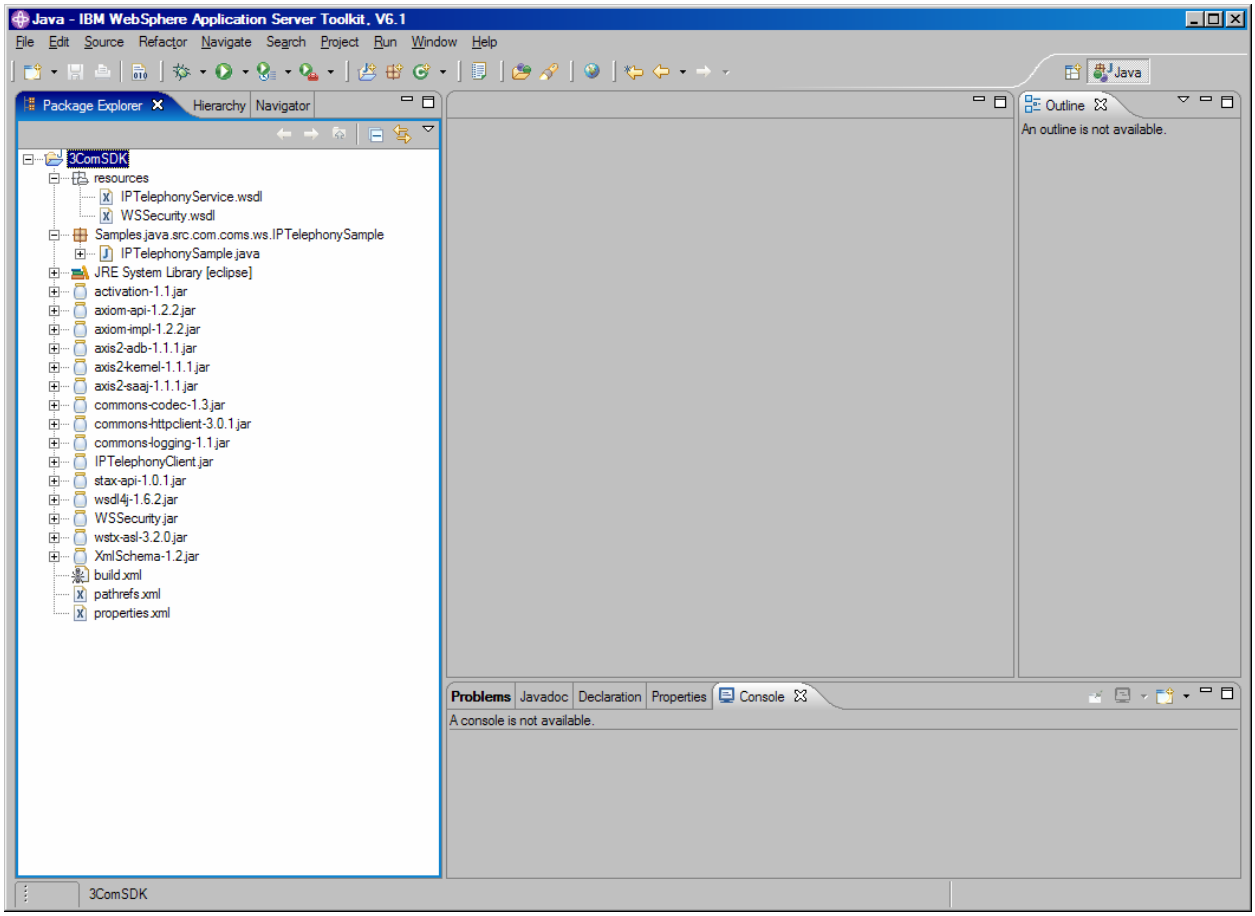

Figure 22. Project after adding 3Com JARs

15. In the left-hand navigator, expand the **Samples.java.src.com.coms.ws.IPTelephony** package.
16. Double-click the **IPTelephonySample.java** file to open it in the IDE editor in the main panel. As shown in Figure 23, there is a red X in the left column of the main-panel editor beside the line, * IP Telephony Web Service Sample.
17. Expand the code section by clicking the plus (**+**) sign beside the line and pointing the cursor to the red X (the cursor is indicated by a light bulb symbol) and right-click.

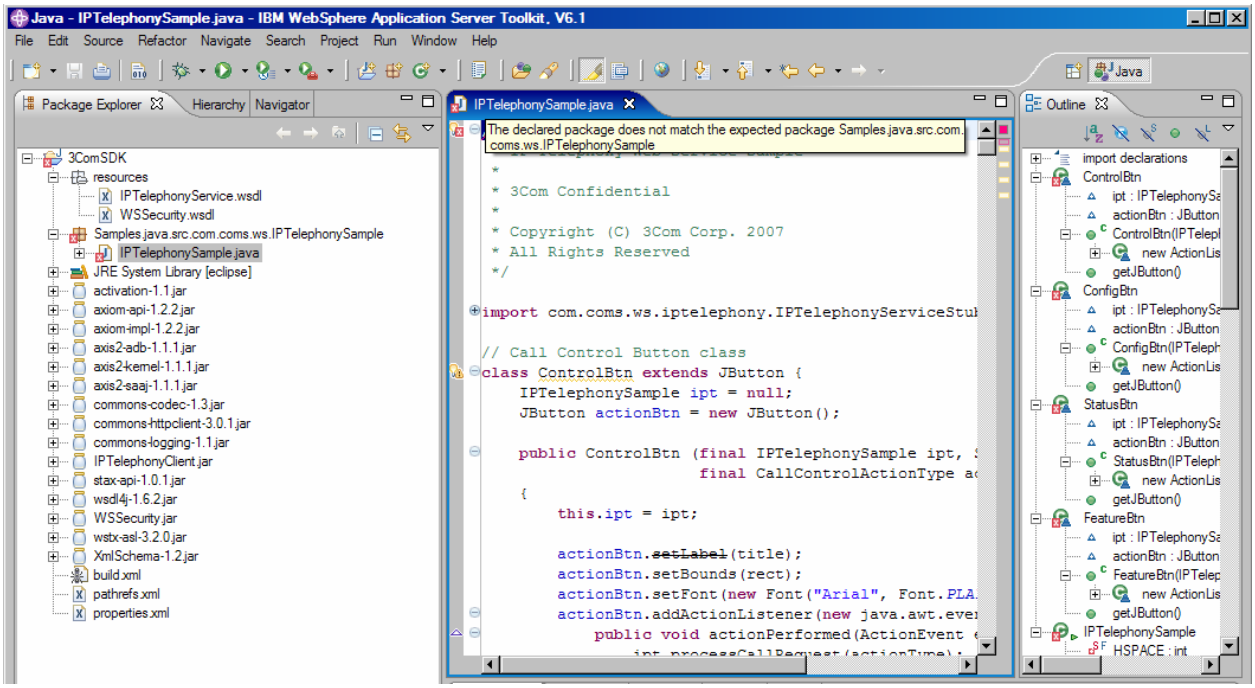

Figure 23. Selecting the IPTelephonySample file

18. Select **Quick Fix** from the options pop-up menu, as shown in Figure 24.

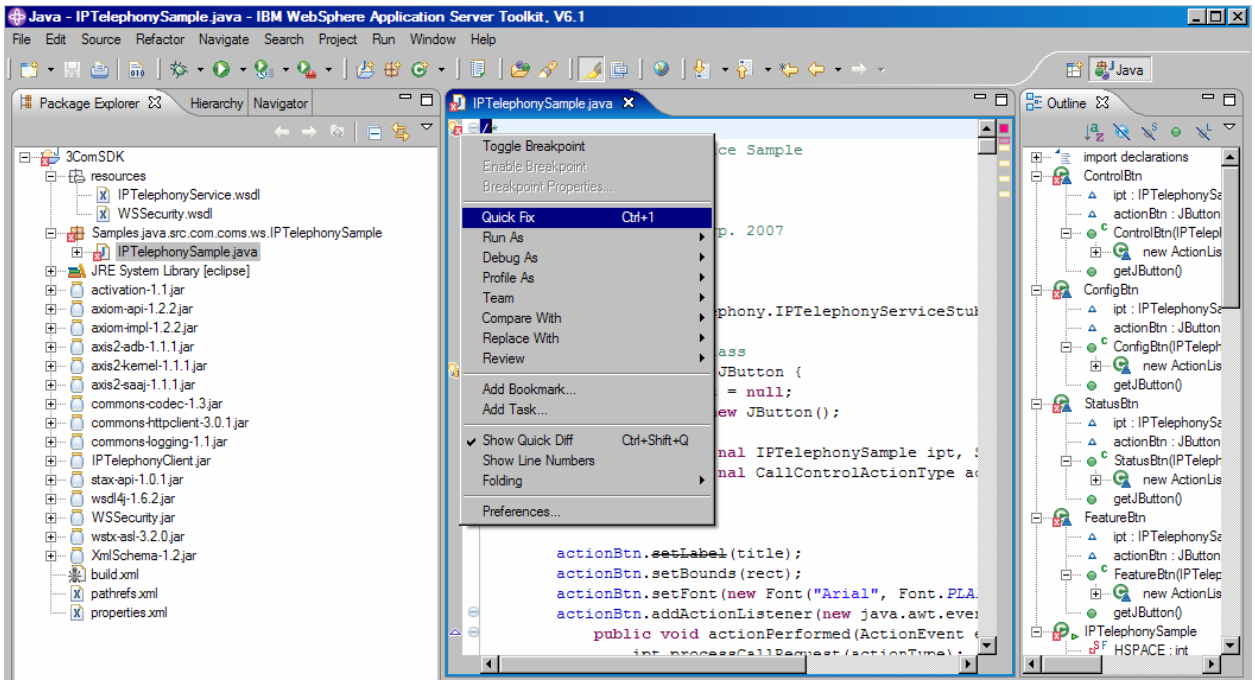

Figure 24. Correcting package name

19. The pop-up list shown in Figure 25 gives you two choices to resolve the problem. Select **Move IPTelephonySample.java to the default package**.

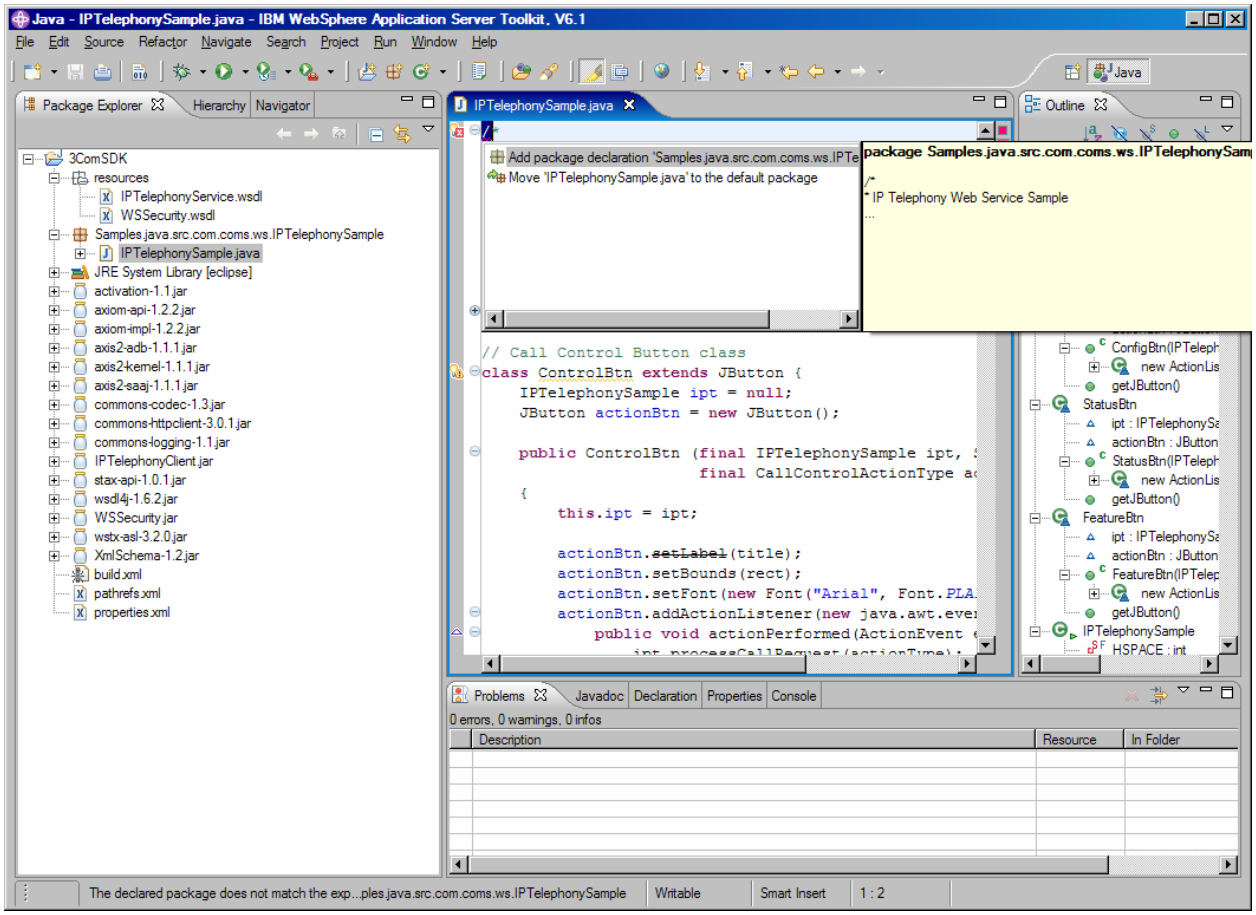

Figure 25. Correcting package name (continued)

That should remove the red X and resolve the package-name issue.

The workspace will look similar to Figure 26.

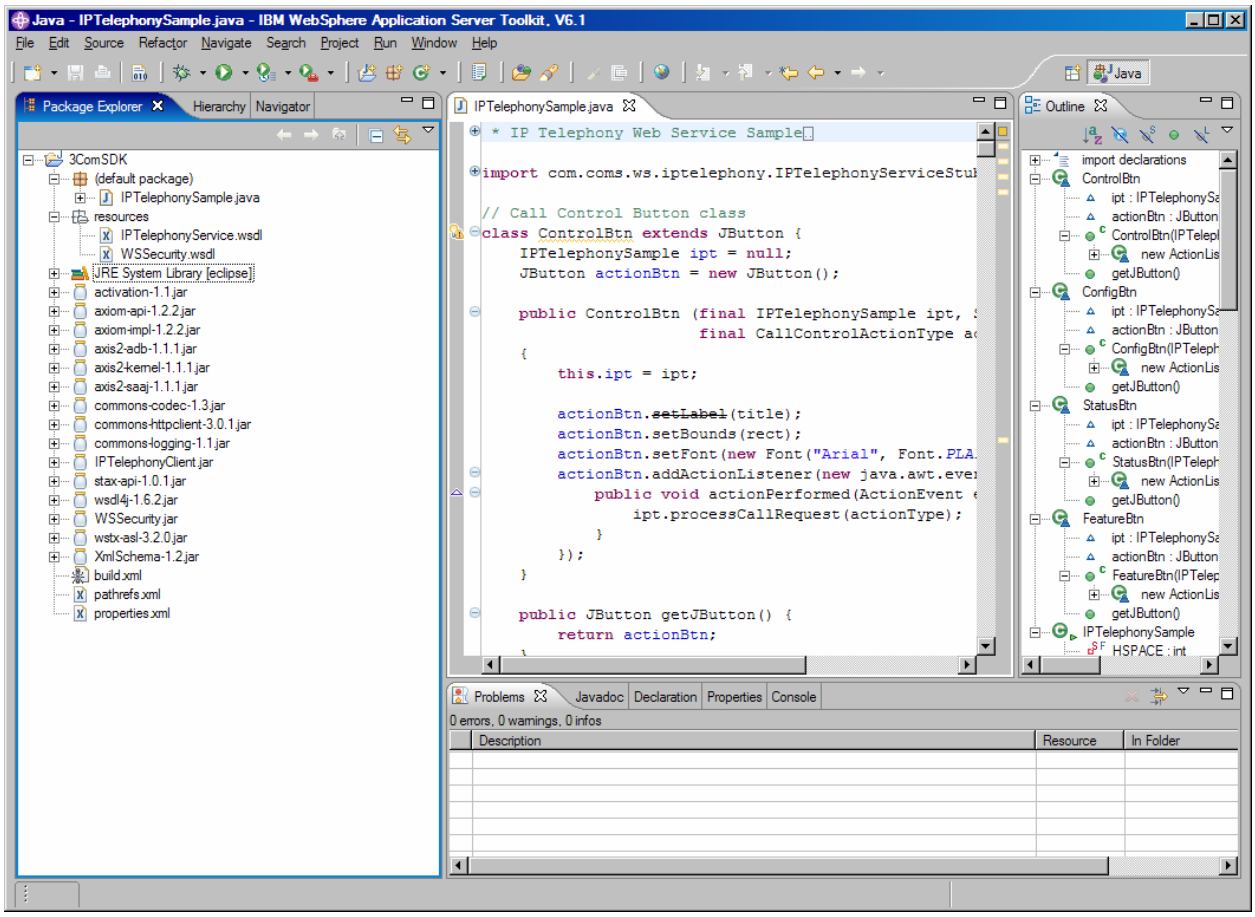

Figure 26. Project workspace after fixing package name

There might still be some warnings, but there is no concern with those at this point.

Running the sample program

With the IDE properly configured, it is possible to run the sample program in one of two ways:

- One method involves using Apache Ant (which stands casually for *another neat tool*). Ant is a build tool that is based on Java and that is also similar to the *make* and *makefile* commands for C programs. The 3Com SDK was built using Ant and ships with build.xml files for preparing, compiling and running the sample programs.
- The other method uses the IBM IDE Java application-run capabilities.

Using Ant

Before continuing, it is important to make some changes to the properties.xml file so that the Ant builds work properly. (See Figure 27; changes are highlighted in italicized, bold, red letters.)

```
<!-- This is an xml entity included in IPTelephonyService build files -->

<property name="sample.name"      value="IPTelephonySample"/>
<property name="wsdl.name"        value="IPTelephonyService"/>

<property environment="env"/>

<!-- set project directories -->
<property name="root.dir"       value="."/>
<property name="lib.dir"        value="${root.dir}/lib"/>

<property name="build.dir"      value="${root.dir}/build"/>
<property name="build.lib"      value="${build.dir}/lib"/>
<property name="build.class"    value="${build.dir}/classes"/>
<property name="java.dir"       value="${root.dir}"/>
<property name="res.dir"        value="${root.dir}/resources"/>

<!-- Give user a chance to override without editing this file -->
<property file="${root.dir}/build.properties"/>
<property file="${user.home}/build.properties"/>

<!-- debug flag for ant javac, values are "on" and "off" -->
<property name="debug"          value="on" />
<property name="nowarn"         value="off" />
<property name="optimize"       value="on" />

<!-- what gets pulled in to the binaries: everything -->
<property name="debuglevel"     value="lines,vars,source" />
<property name="deprecation"    value="true" />
<property name="source"         value="1.0" />
<property name="target"         value="1.0" />

<property name="build.file"     value="build.xml" />

<property name="exclude.log4j.configuration" value="true"/>
```

Figure 27. Changes required to the properties.xml file

20. To run the sample using Ant, right-click the **BUILD.XML**file.
21. Select **Run As → Ant Build…**
22. When the Ant dialog box appears, select the **run** check box and clear any others that might be selected.
23. Click **Apply** and, then **Run**. The GUI window (as shown in Figure 28) is presented.
    a. On this user interface, change the **Web Service URL** to be the URL of your 3Com VCX. That is, replace http://localhost/axis2/services/IPTelephonyService with the host name of your VCX. For example, http://<your VCX host name>/axis2/services/IPTelephonyService. (**Note:** The URI portion remains the same as axis2/services/IPTelephoneService.)
    b. For the Security Header fields of **Username** and **Password**, the shipped defaults are *wsuser* and *wspwd*, respectively. The **Origination Number**, **Phone Password** and **Destination Number** fields are specific to your 3Com VCX hardware configuration.

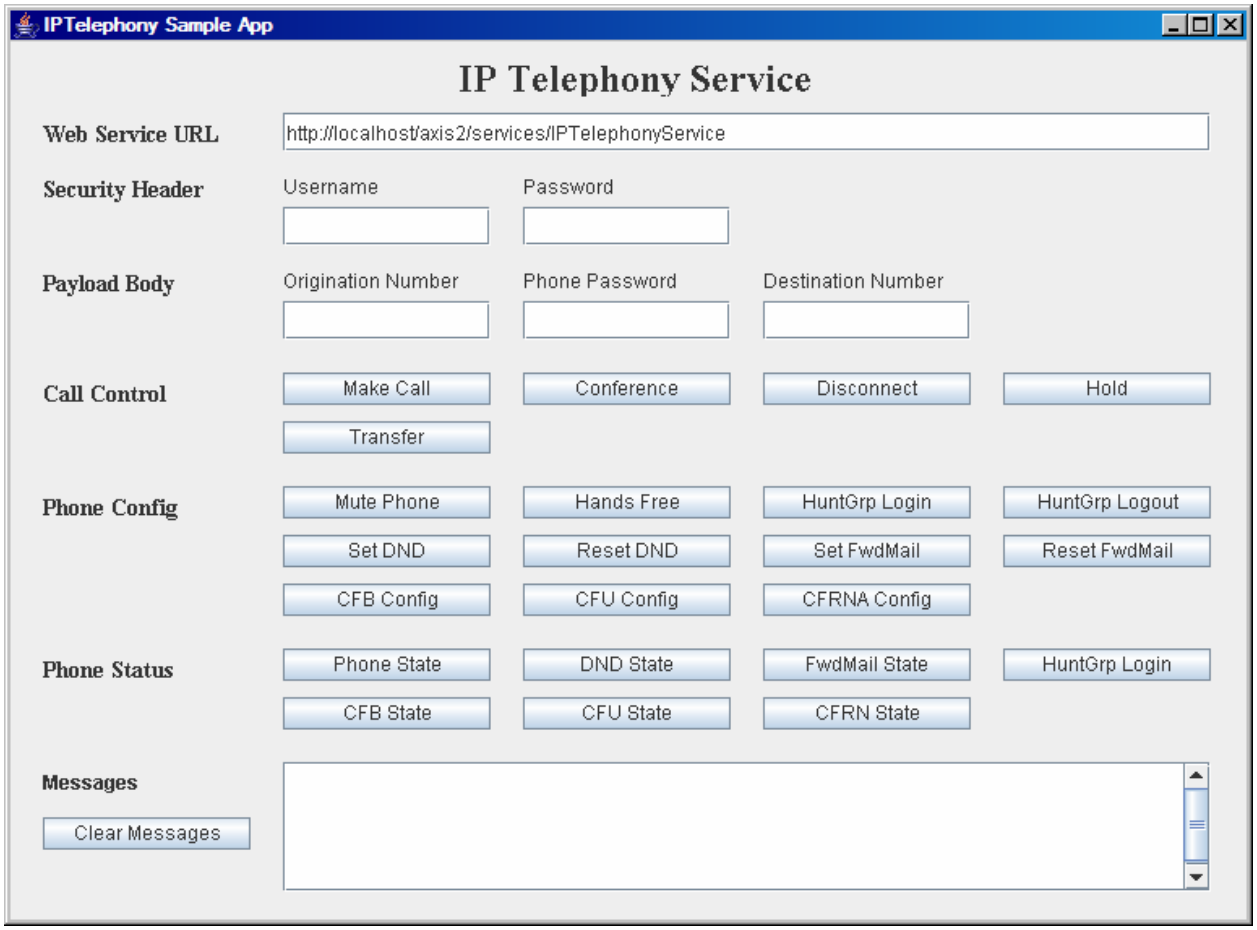

Figure 28. Sample program

Using the IDE run

Figure 29 shows how to start the GUI sample application using the IDE Java run capability.

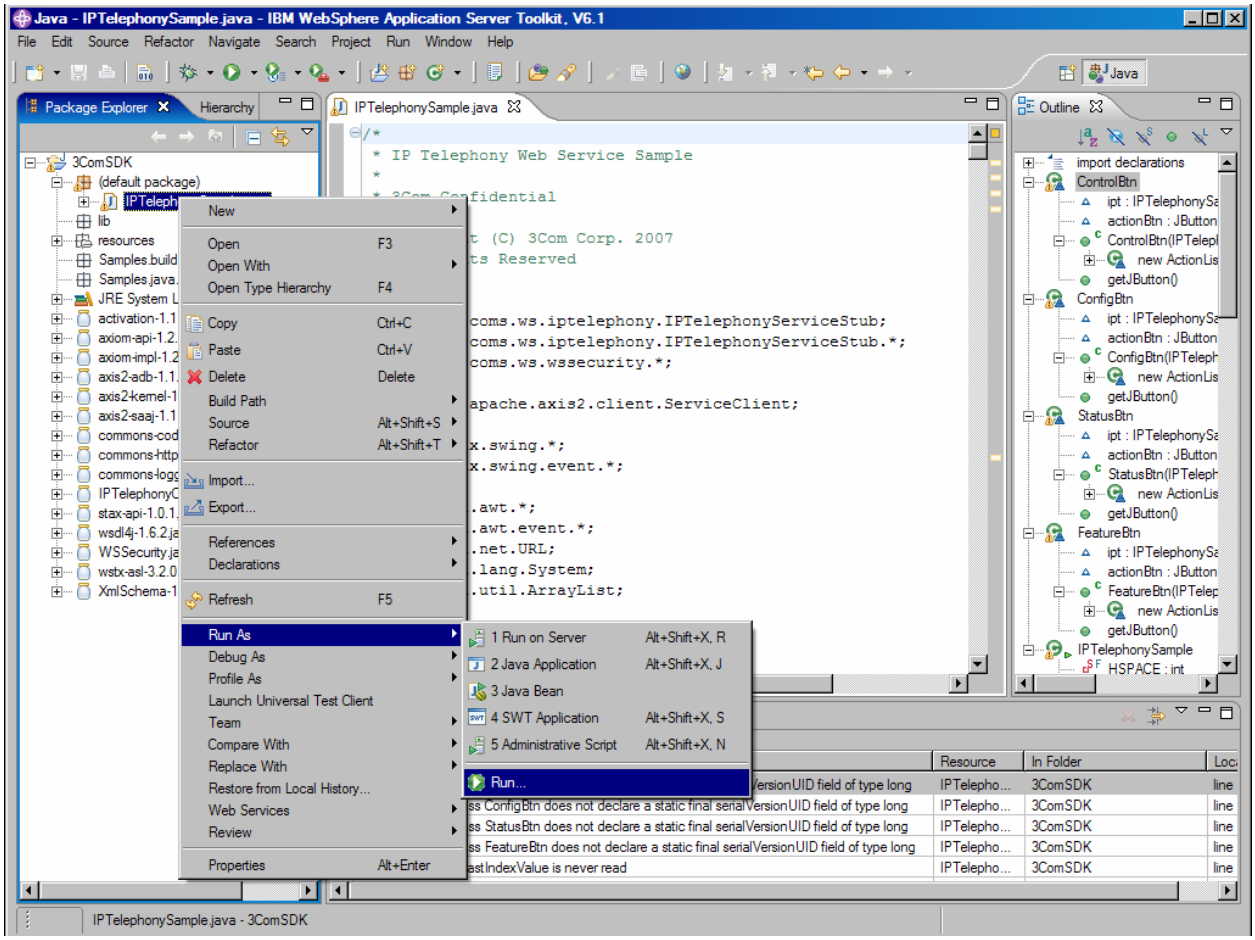

Figure 29. Running the sample

Note: The first time you run this program, the panel shown in Figure 30 might look somewhat different than it does here. If the panel highlights *Eclipse Application*, select **Java Application** and then click **New,** Which opens the panel shown in Figure 30.

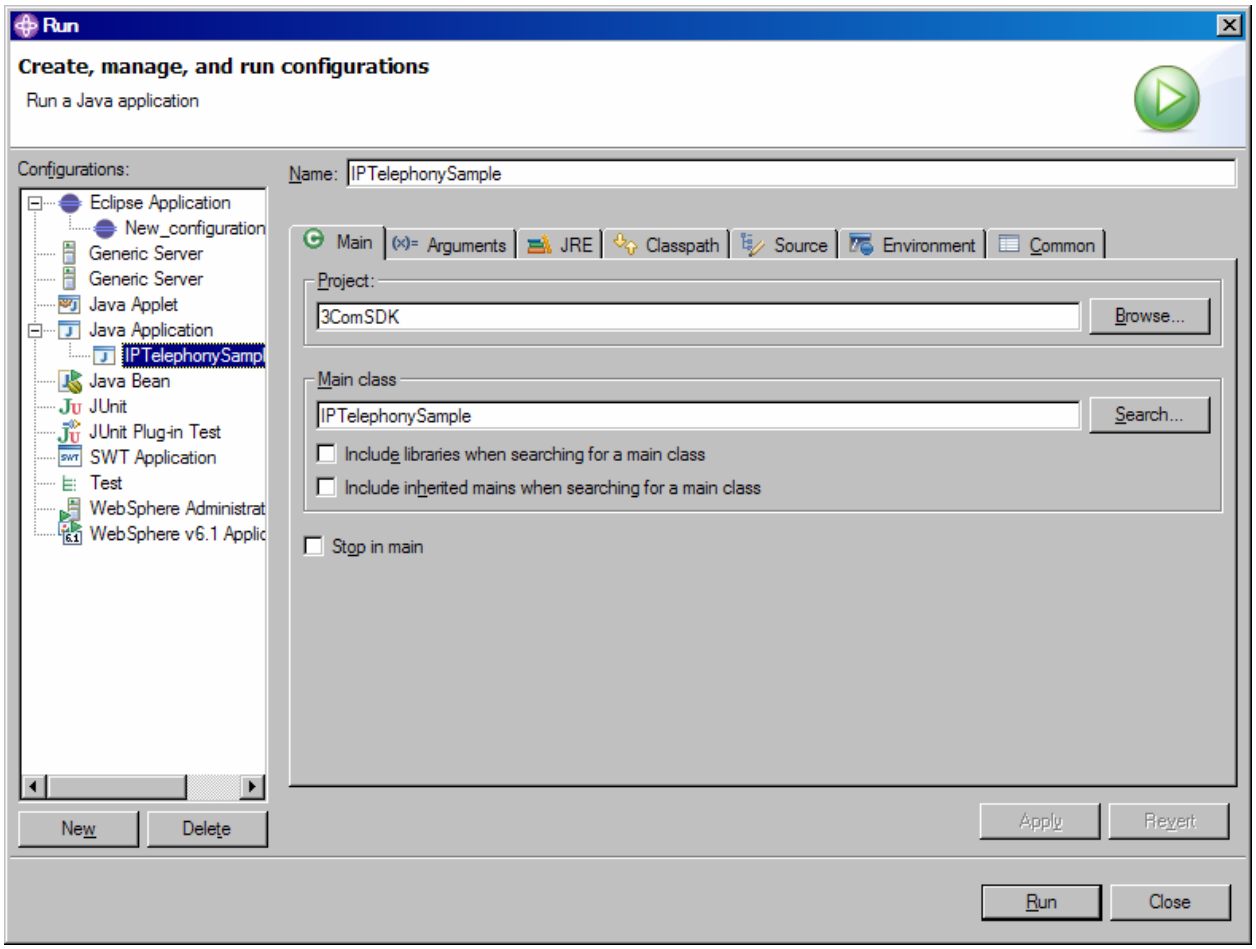

Figure 30. Running the sample (continued)

24. Click **Run** and the GUI shown previously in Figure 28 now appears:

    a. On this user interface, you must change the Web Service URL to be the URL of your 3Com product. That is, replace *http://localhost/axis2/services/IPTelephonyService* with the host name of your 3Com software. For example, *http://<your VCX host name>/axis2/services/IPTelephonyService*.
    Note: The URI portion remains the same as axis2/services/IPTelephoneService.

    b. For the Security Header fields of **Username** and **Password**, the shipped defaults are *wsuser* and *wspwd*, respectively. The **Origination Number**, **Phone Password** and **Destination Number** fields are specific to your 3Com hardware configuration.

Migrating and running the sample on System i

Running the sample GUI program on the System i itself is optional, but the section on “Setting up the Java IDE” that starts on page 7 is a required step for running the non-GUI sample code in the “Sample” section on page 54.

Exporting the code

To run the sample on the System i platform, you must export the project from the IDE to the integrated file system (IFS) on your System i model.

1. Select the Java project and proceed to export it to the file system, as shown in Figure 31.
2. The directory specified in the **To directory** field must on the System i model that has TCP/IP network access to the System i model with the 3Com telephony partition. In Figure 31, the **K:** drive is a mapped drive to the System i IFS.

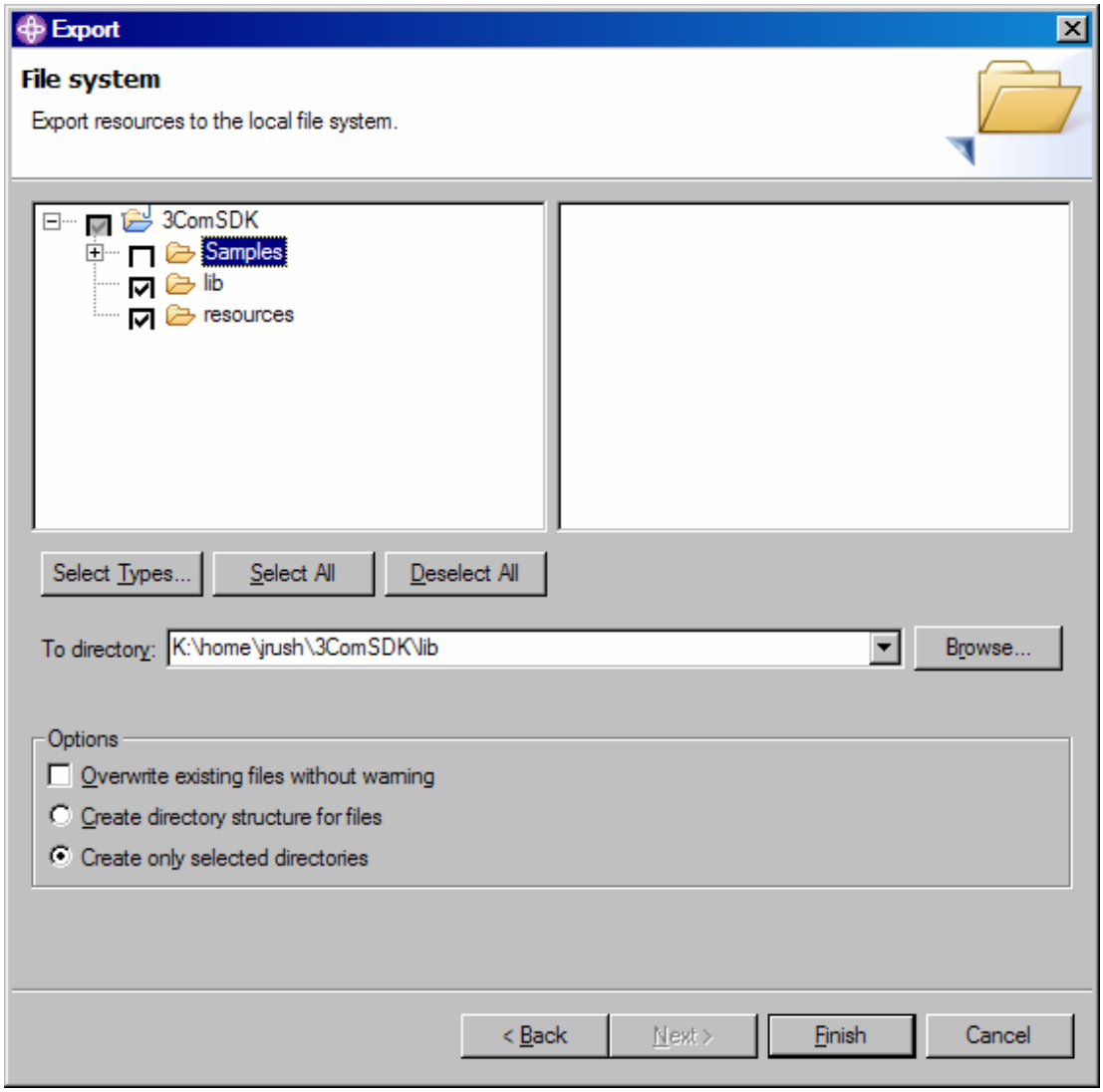

Figure 31. Exporting the project to the System i IFS

Setting up the Java environment

1. Assuming you have mapped a drive letter to the System i IFS, in Microsoft® Windows® Explorer, move the *lib* directory contents (the lib directory from the “Exporting the code” step on page 34) to the /QIBM/UserData/Java400/ext directory. These JAR files are then added to the class path and their classes are loaded by the extensions class loader.
2. Log on to the System i model that was the target of the export.
3. Create or modify the SystemDefault.properties file in your home directory to look like the following code. The java.class.path file must minimally contain a period (.). You can add your specific JARs and directories, also.

```
os400.awt.native=true
java.class.path=.
```

Setting up System i remote-graphics capabilities

Before the sample program can work properly, the System i host must have Native Abstract Windowing Toolkit (NAWT) installed and running. This white paper uses the Virtual Network Computing (VNC) server that is included in the System i Tools for Developers PRPQ (5799PTL). This section details how to set up the VNC server so that the IPTelephonySample program remotely streams the GUI through the VNC server to the VNC client that runs on the workstation desktop, either in the VNCviewer client or a Web browser. (Refer to http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp, search on **VNCviewer**).

Running the sample

1. Start the VNC client in a Web browser, as shown in Figure 32.

   Note: The port number (5899 in Figure 32) is 5800 plus the VNC display number that was configured in the VNC-setup step discussed in the previous section.

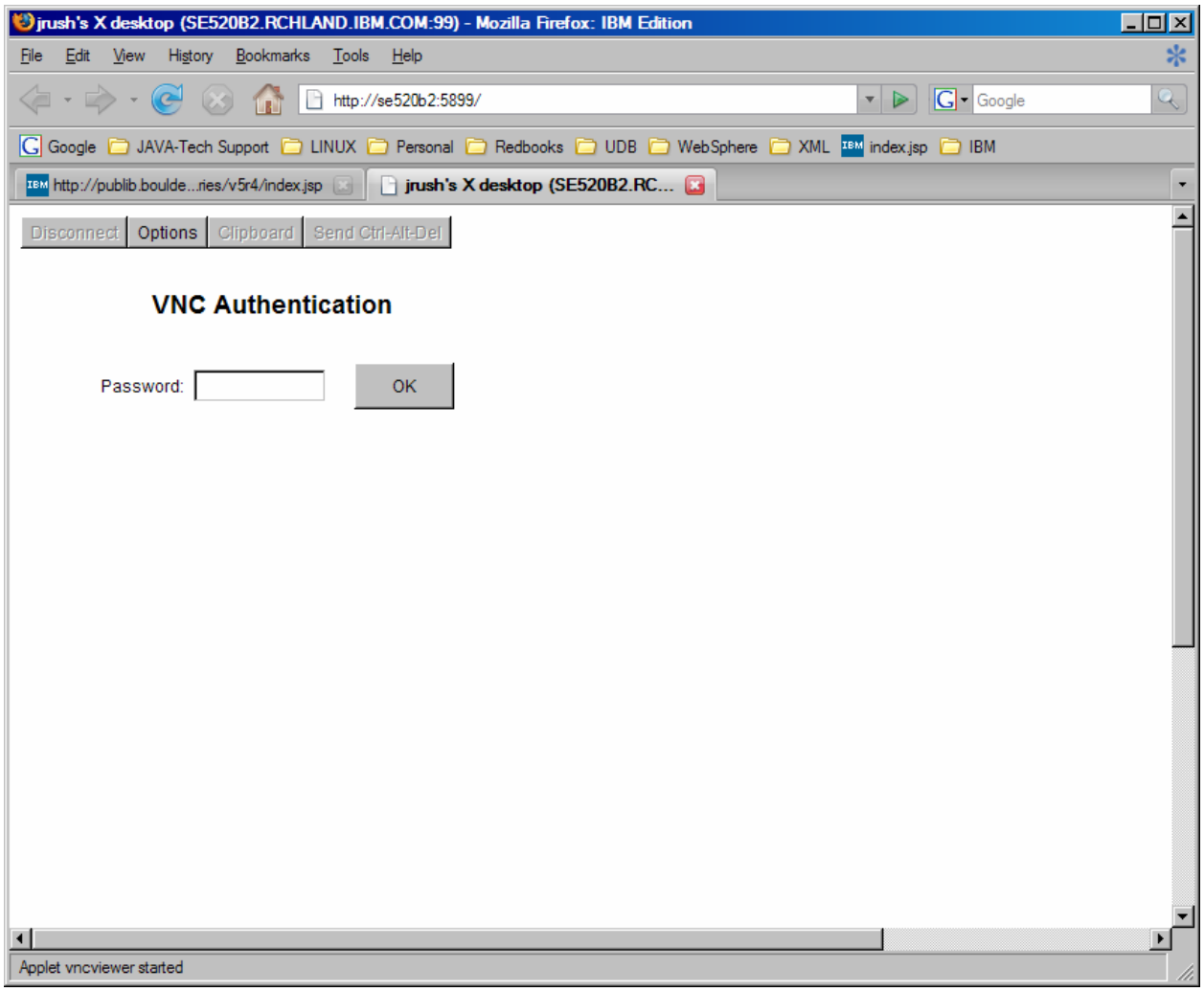

Figure 32. Starting the VNC client in a Web browser

2. Add an environment variable for JAVA_HOME and set it to be **/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit** as illustrated in Figure 33 and Figure 34.

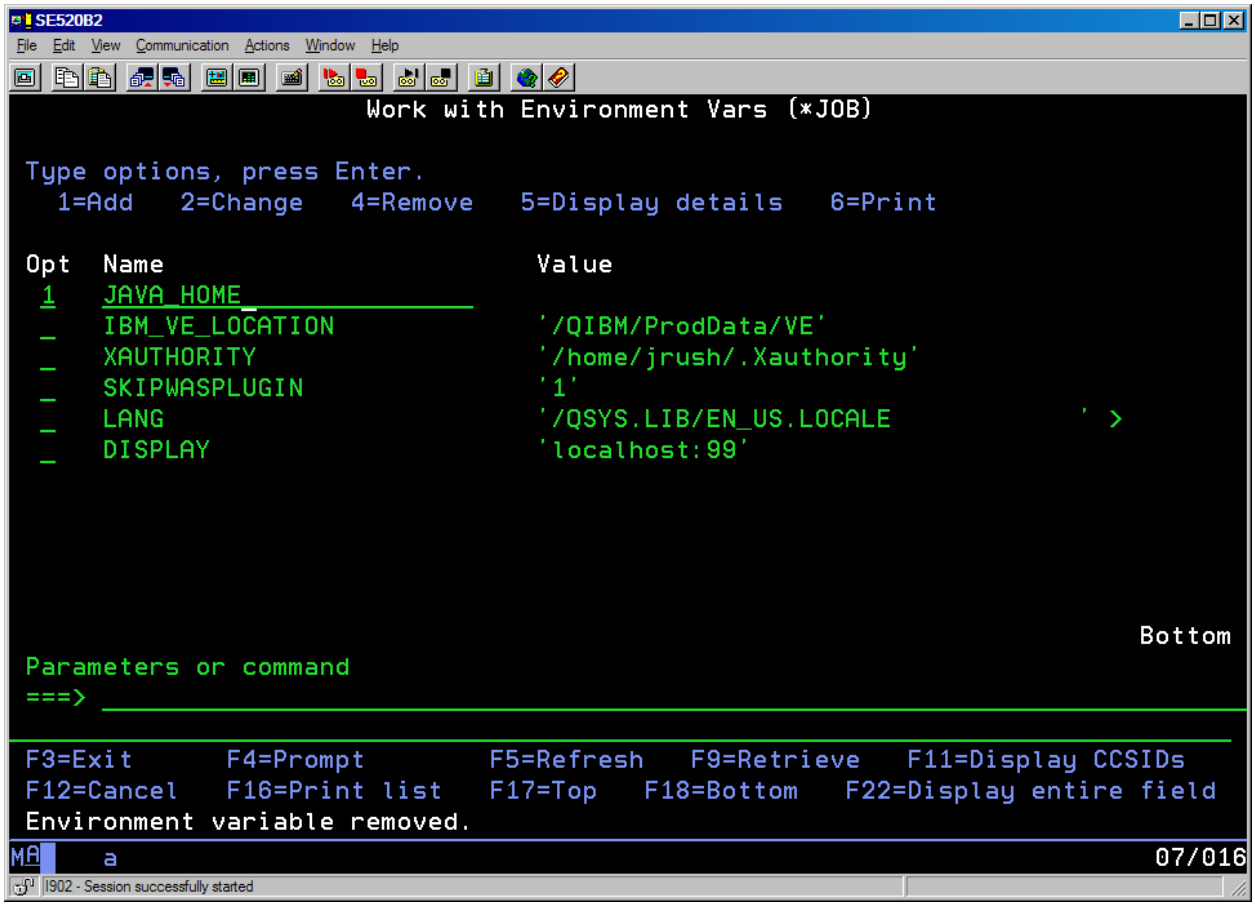

Figure 33. JAVA_HOME environment variable

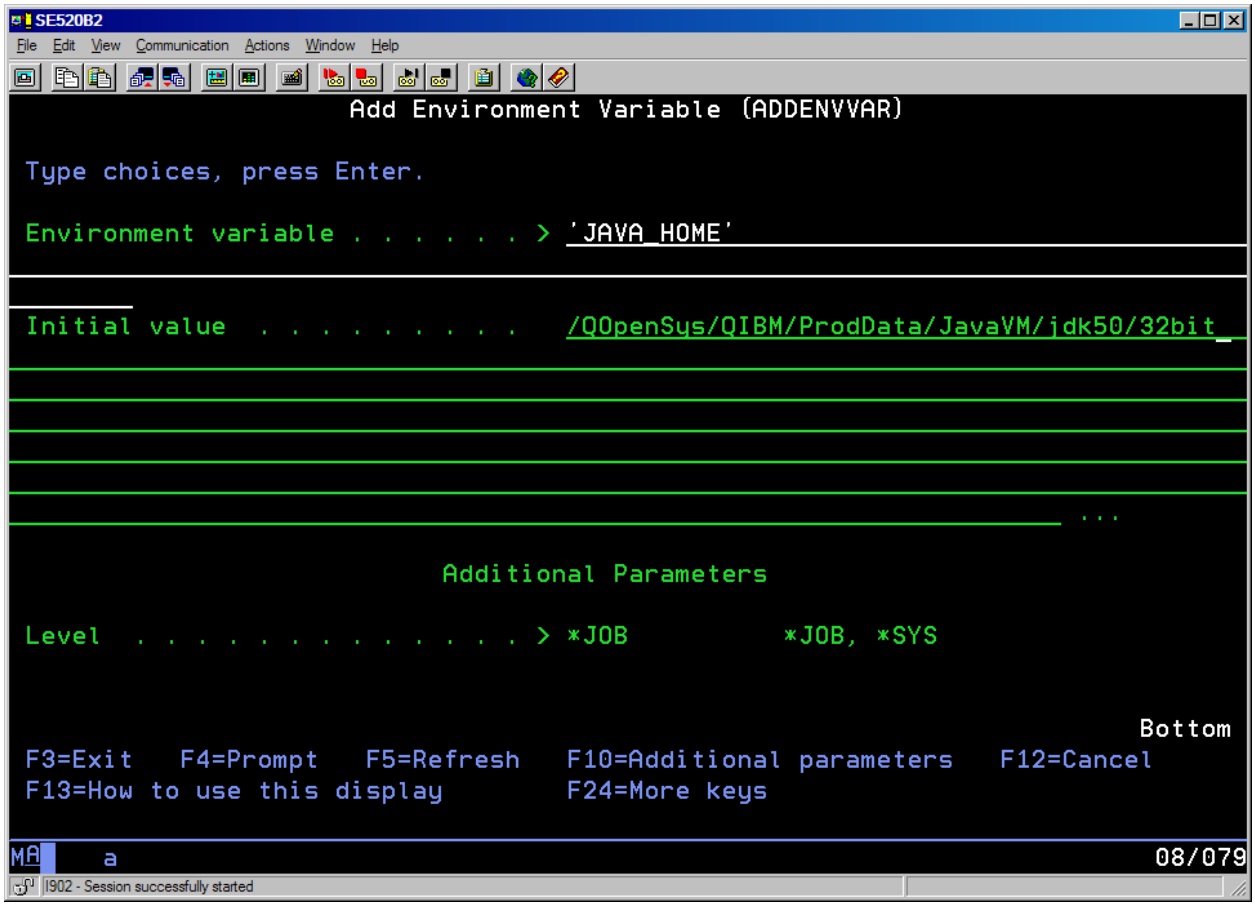

Figure 34. JAVA_HOME environment variable (continued)

3. Invoke the **STRQSH** command and then enter `java -version` to see the Java virtual machine (JVM) version, as shown in Figure 35.

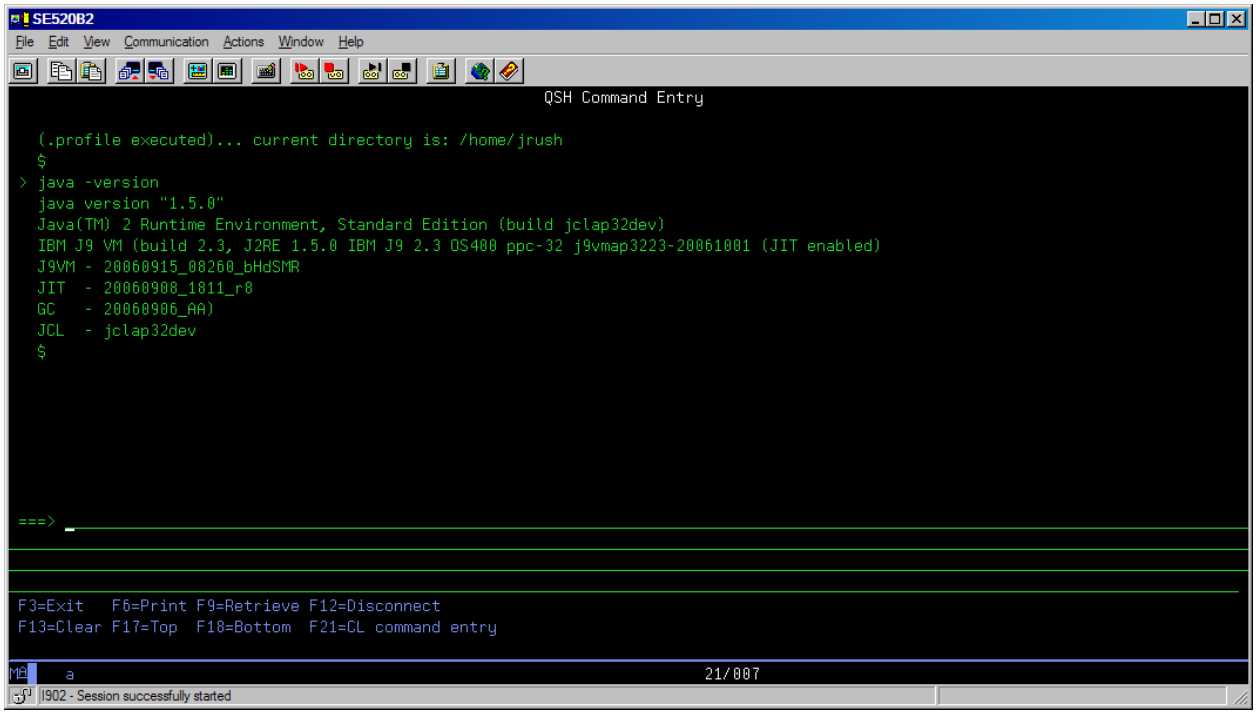

Figure 35. QSH

Change to the directory where the IPTelephonySample.class is.

4. On the command line, enter `java IPTelephonySample`. The GUI shows up in the VNC client as seen in Figure 36.

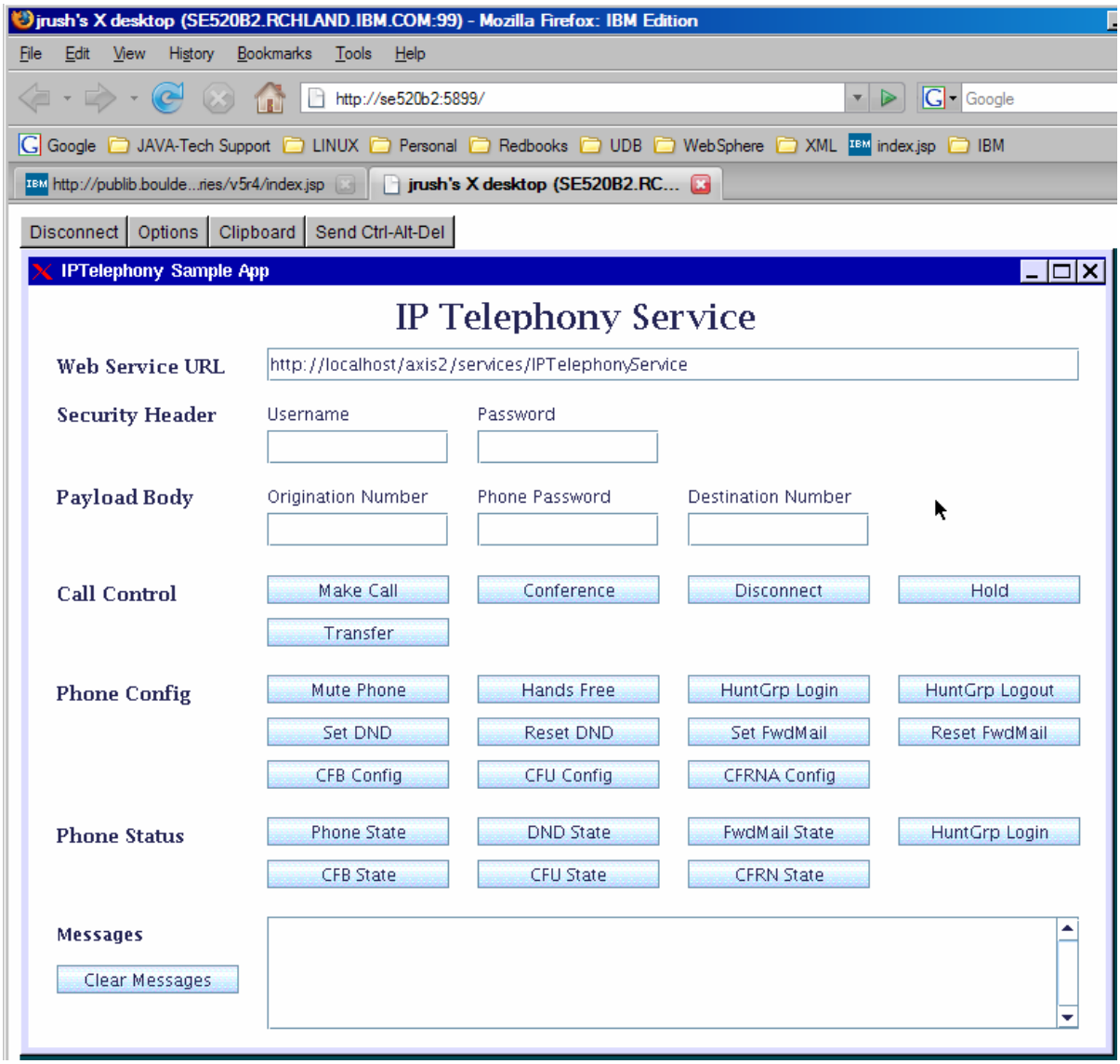

Figure 36. Sample GUI in Web browser

Using the IDE WSDL editor

The WebSphere Application Server Toolkit IDE includes a graphical viewer and editor for WSDL files. Within this editor, there is the ability to expand the WSDL to view input and output parameters that are defined and required by the telephony service points. As shown in Figure 37, the WSDL definition file for IP Telephony Web-service points has three categories: *call control*, *phone configuration* and *phone state*.

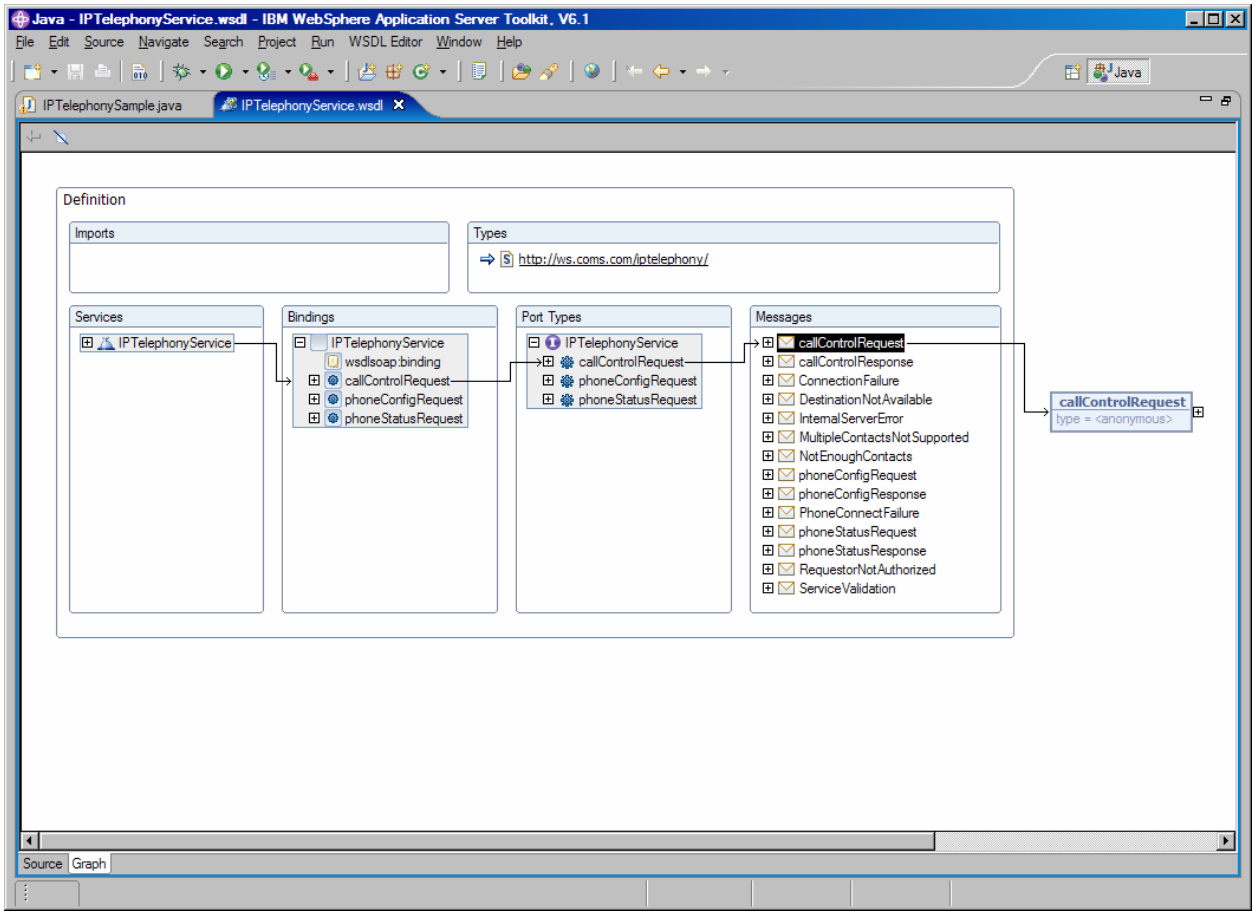

Figure 37. IP Telephony WSDL in the WebSphere Application Server Toolkit IDE

1. For example, to see the parameters needed for a call-control request, you can expand **callControlRequest** in the Port Types box and expand the **callControlRequest** blue box to see the parameters, as shown in Figure 38.

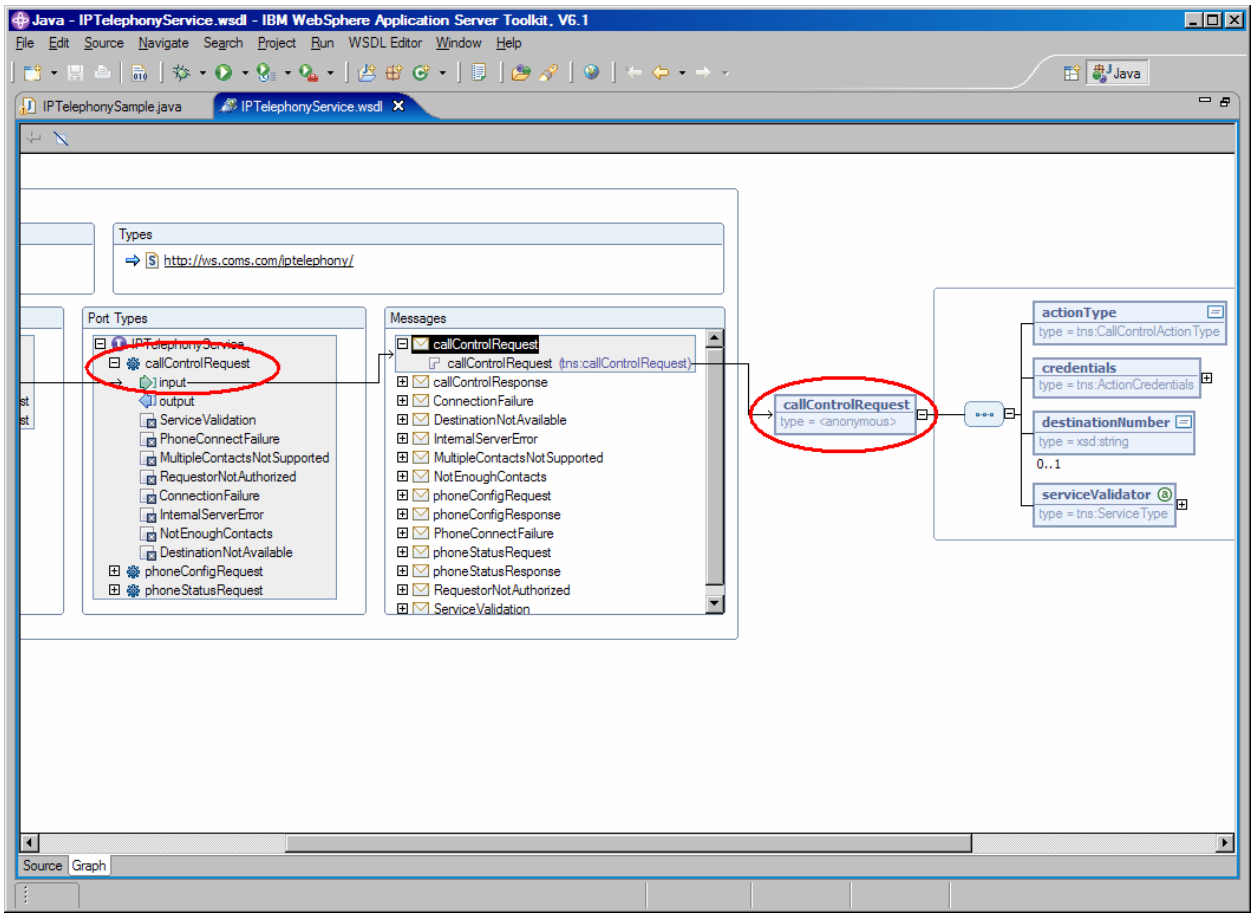

Figure 38. Parameters for a call-control request

In this example, the parameters are as follows:

- actionType
- credentials
- destinationNumber
- serviceValidator

There is an in-depth discussion of these parameters in the “Sample” section.

Customizing the SDK for your environment (optional)

Now that you have imported the SDK successfully into the IDE, you can make some customizations that are specific to the local 3Com environment.

Modifying the IPTelephony WSDL file

The SDK comes with the service endpoint set to http://localhost/axis2/services/IPTelephonyService/. You can change this to be the URL of your 3Com, but the client stub shipped in the SDK allows this to be programmatically set to your 3Com host upon instantiation of the Web-service client object. To change the endpoint, perform the following steps:

1. Double-click the IPTelephony.wsdl file in the IDE in the resources directory in the project you created in the toolkit. The WSDL file opens in the WSDL editor, as shown in Figure 3940.
2. Click the **Source** tab, as illustrated in Figure 3940 (labeled number **2**).
3. Move to the bottom of the file and find the section starting with
   `<wsdl:service name="IPTelephonyService">`
   and click the line that begins with
   `<soap:address…`
   as shown in Figure 3940 (labeled number **3**).
4. In the properties tab, change the value of the location property to contain your 3Com address, as illustrated in Figure 3940 (labeled **4**).
5. Save the WSDL file changes.

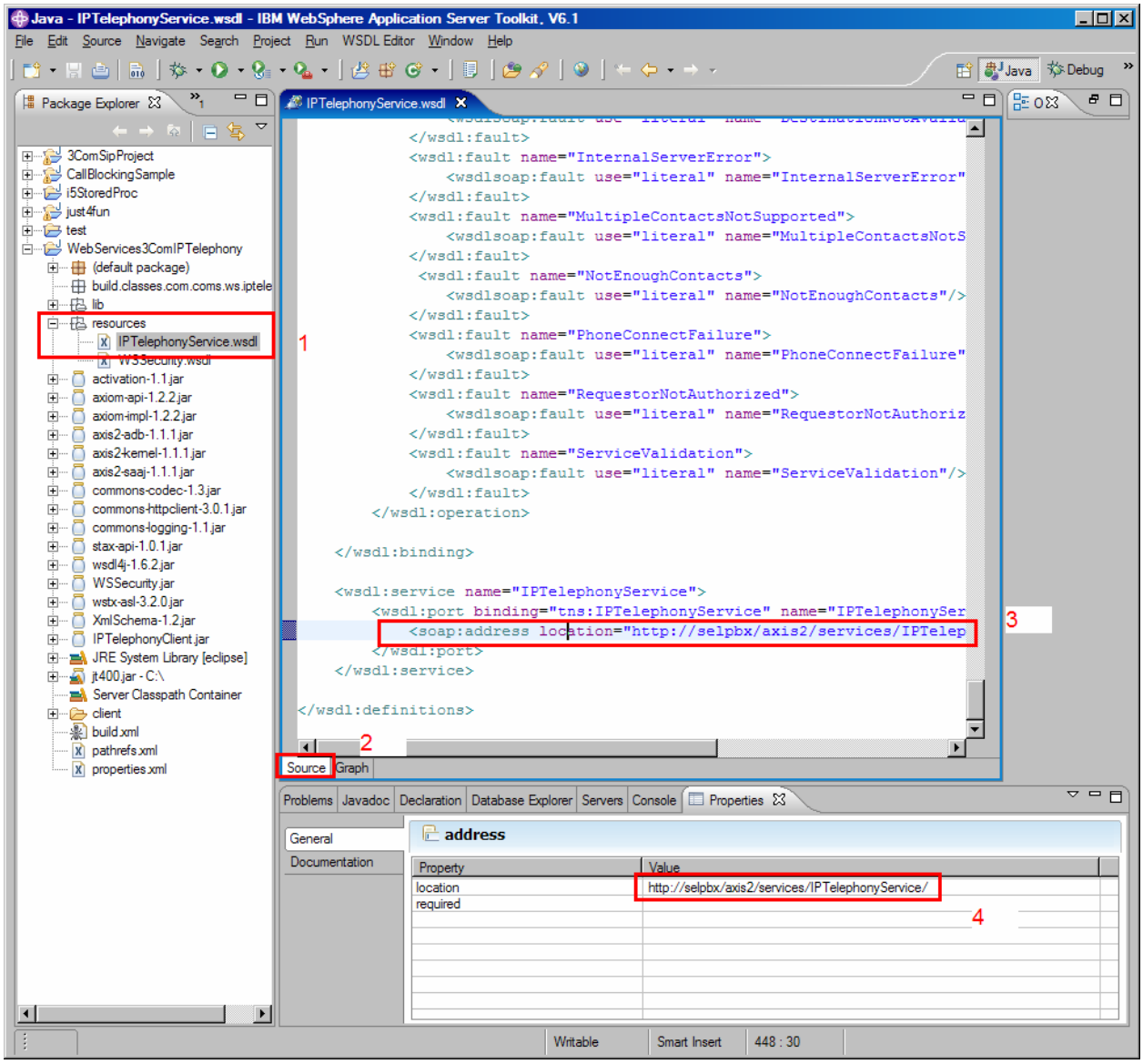

Figure 39. Modifying the IPTelephony WSDL file

Generating the client-service stubs

This step is optional and is not required to integrate and build applications that use the 3Com Web services in your environment. If you want to analyze the Web-services client code used by the 3Com SDK (or if you modify the IPTelephony WSDL as described in section “Modifying the IPTelephony WSDL file” to change the service endpoint from http://localhost/axis2/services/IPTelephonyService/ to your 3Com VCX host name), then you need to regenerate the client stubs and rebuild the IPTelephonyClient JAR.

The SDK includes the IPTelephonyClient.jar file, which contains the Java classes that were generated by 3Com to be used to invoke and call the Web-service endpoints. After modifying the IPTelephony.wsdl file to point to your 3Com host, you must regenerate the IPTelephony client-side classes by using the Ant build.xml code that is included with the SDK.

The first step is to make the Axis 2 JAR files available to the development environment. These JAR files include utilities that are used to create Java source files from the IPTelephony.wsdl file.

1. Right-click your project and import the Axis 2 JARs that you downloaded as part of the prerequisites, as illustrated beginning in Figure 40 and Figure 41.

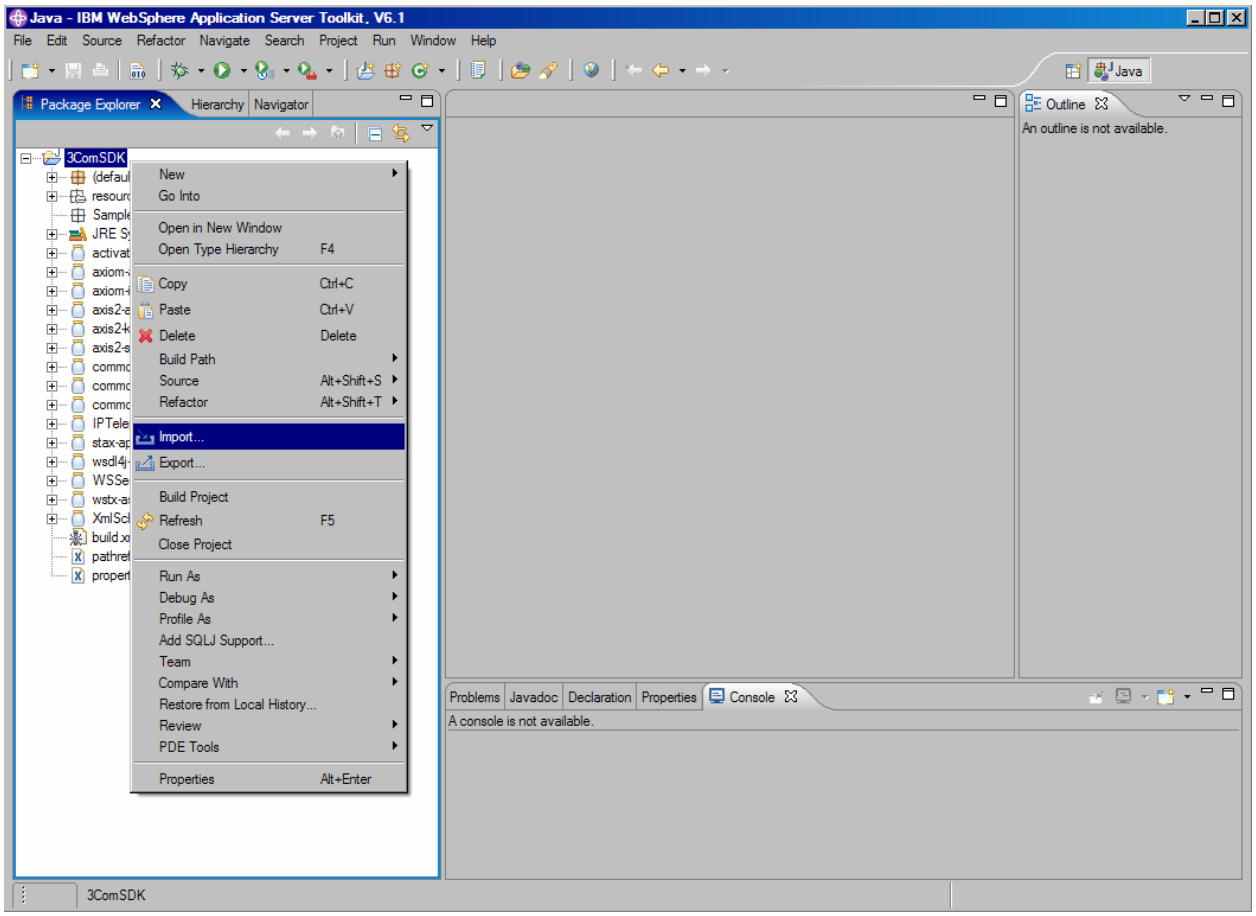

Figure 40. Importing Axis 2 JARs

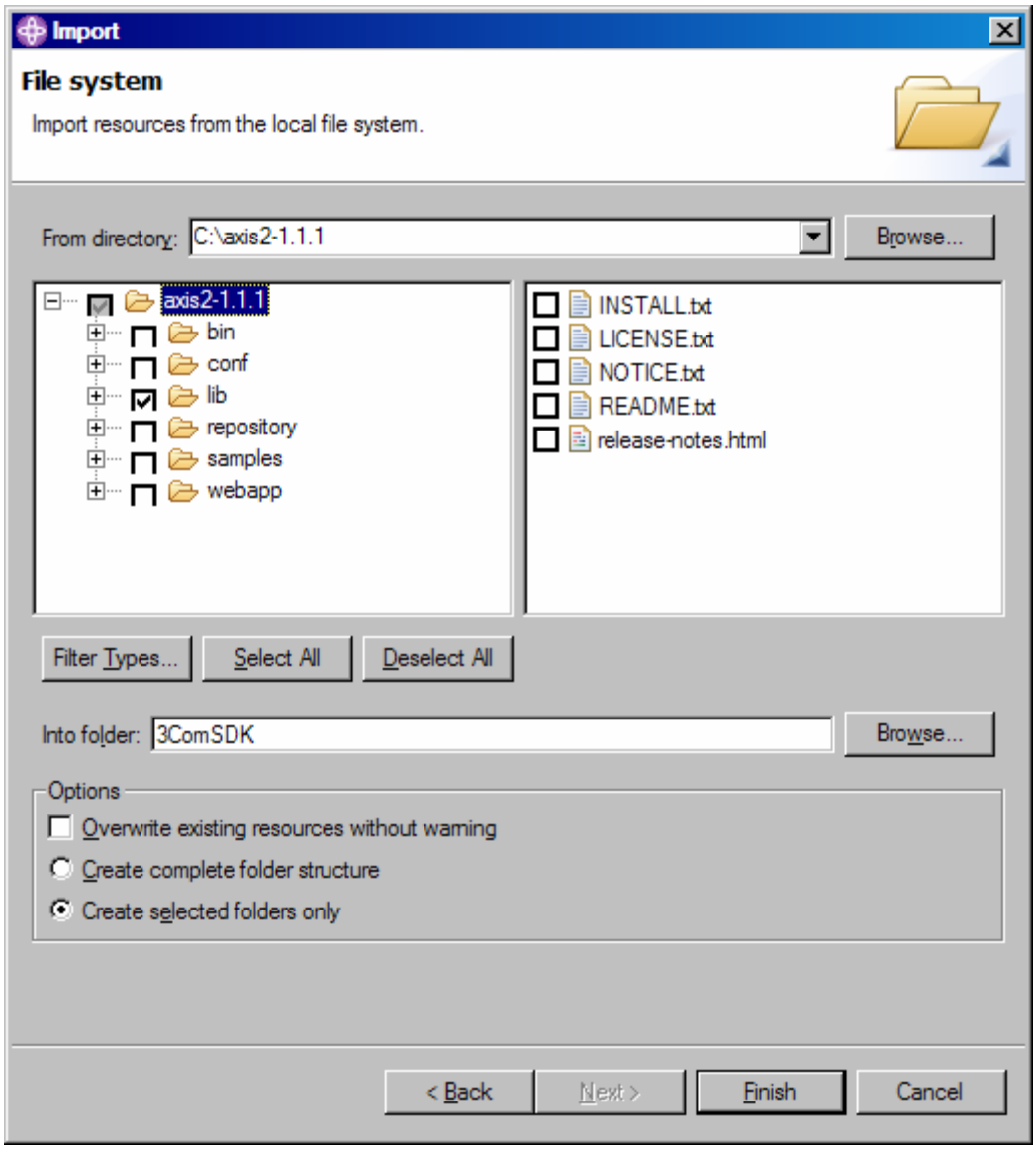

Figure 41. Importing Axis 2 JARs (continued)

Note: As you proceed through this process, select **No to all** when prompted to replace or overwrite the JAR files that already exist. The **lib** directory will then contain the Axis 2 JAR file that was downloaded from the Apache Axis Web site, as shown in Figure 44.

2. Add the AXIS2_HOME environment variable to your system by right-clicking the **My Computer** icon on your desktop and selecting **Properties**.
3. Click the **Advanced** tab and then click **Environment Variables** at the bottom of the panel to open the window as shown in Figure 42.

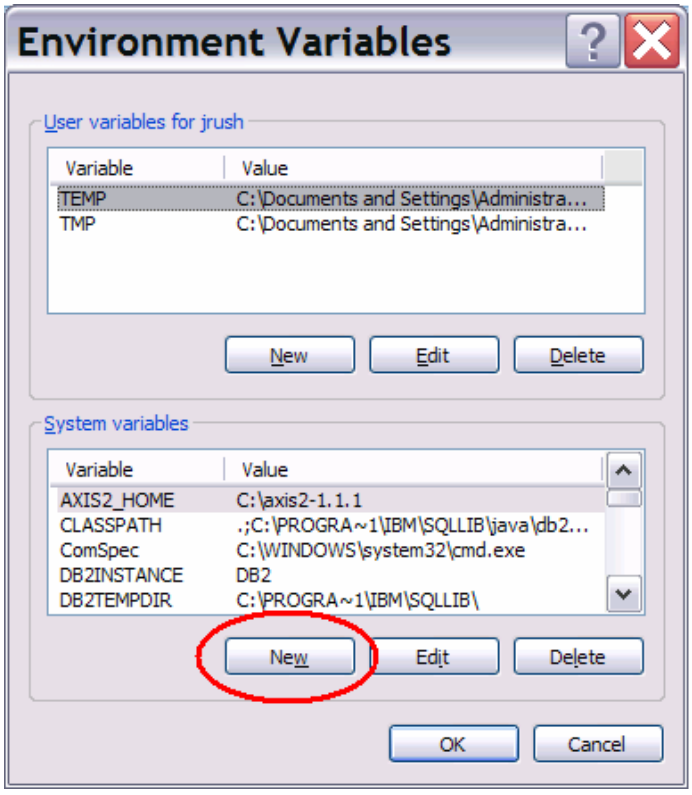

Figure 42. New system environment variable

4. Click **New** and add the AXIS2_HOME, as shown in Figure 43 — the Variable value is where you put the downloaded Axis 2 files.
5. You must then exit the IDE and restart it so that it picks up the AXIS2_HOME environment variable.

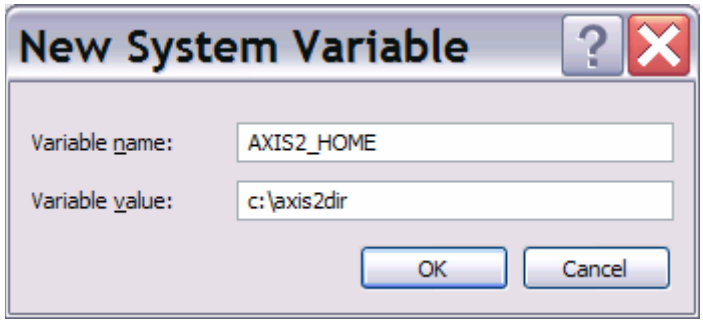

Figure 43. New AXIS2_HOME variable

Figure 46 shows the GUI after hitting the **Run** key.

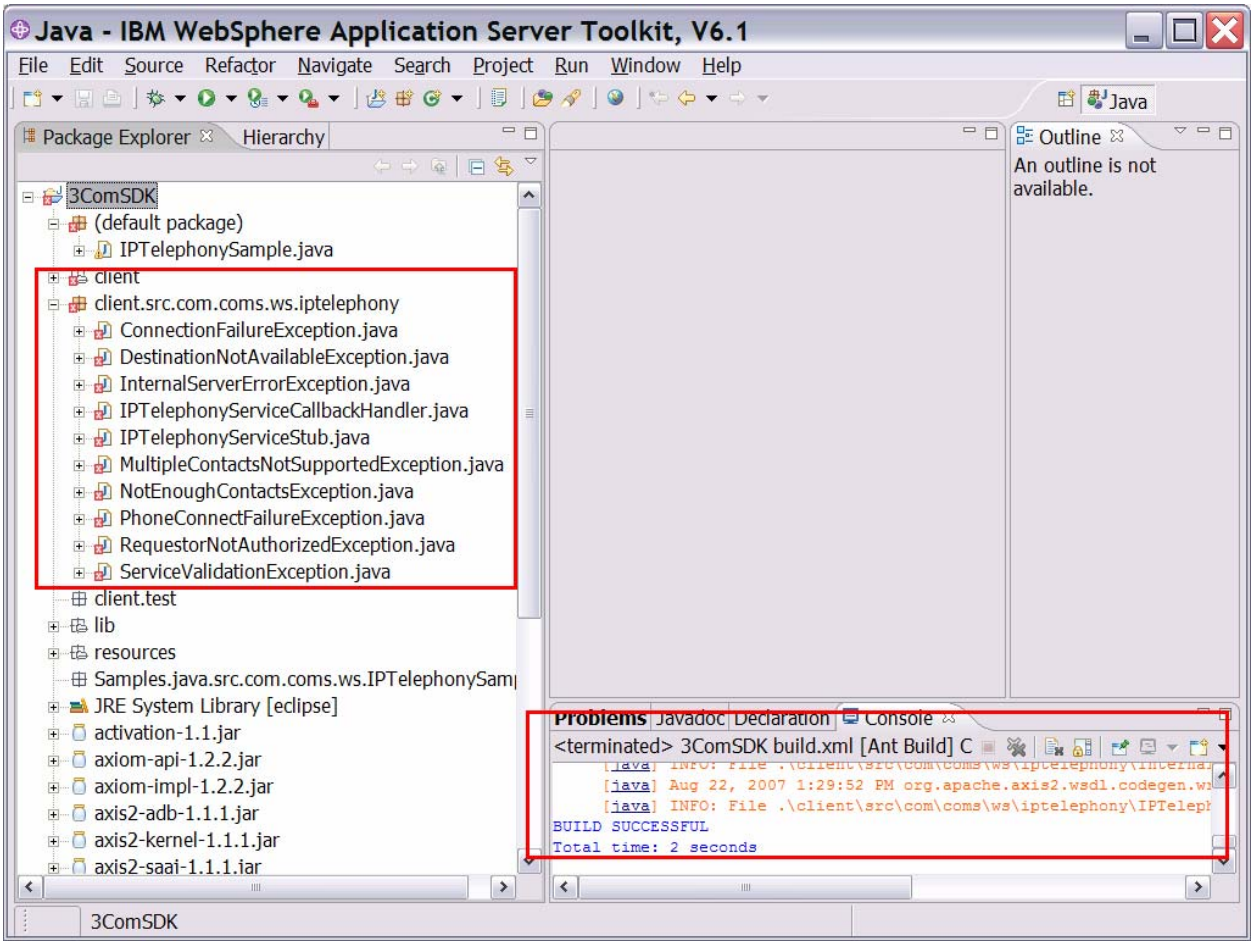

Figure 46. After w2j

Note: You can ignore the red Xs here that signal errors because the generated Ant build.xml is used under the client package/folder to create the IPTelephonyClient JAR file.

Building a new IPTelephony client JAR file

The next step is to build a test JAR file from the Java source created by the w2j Ant build step (the WSDL2JAVA utility that ran to create the client Java stubs).

1. Select the **build.xml** file under the client directory, as shown in Figure 47.
2. Right-click **Run As -> Ant Build…**
   Note: Select the **Ant Build** with the ELIPSES (**…**).

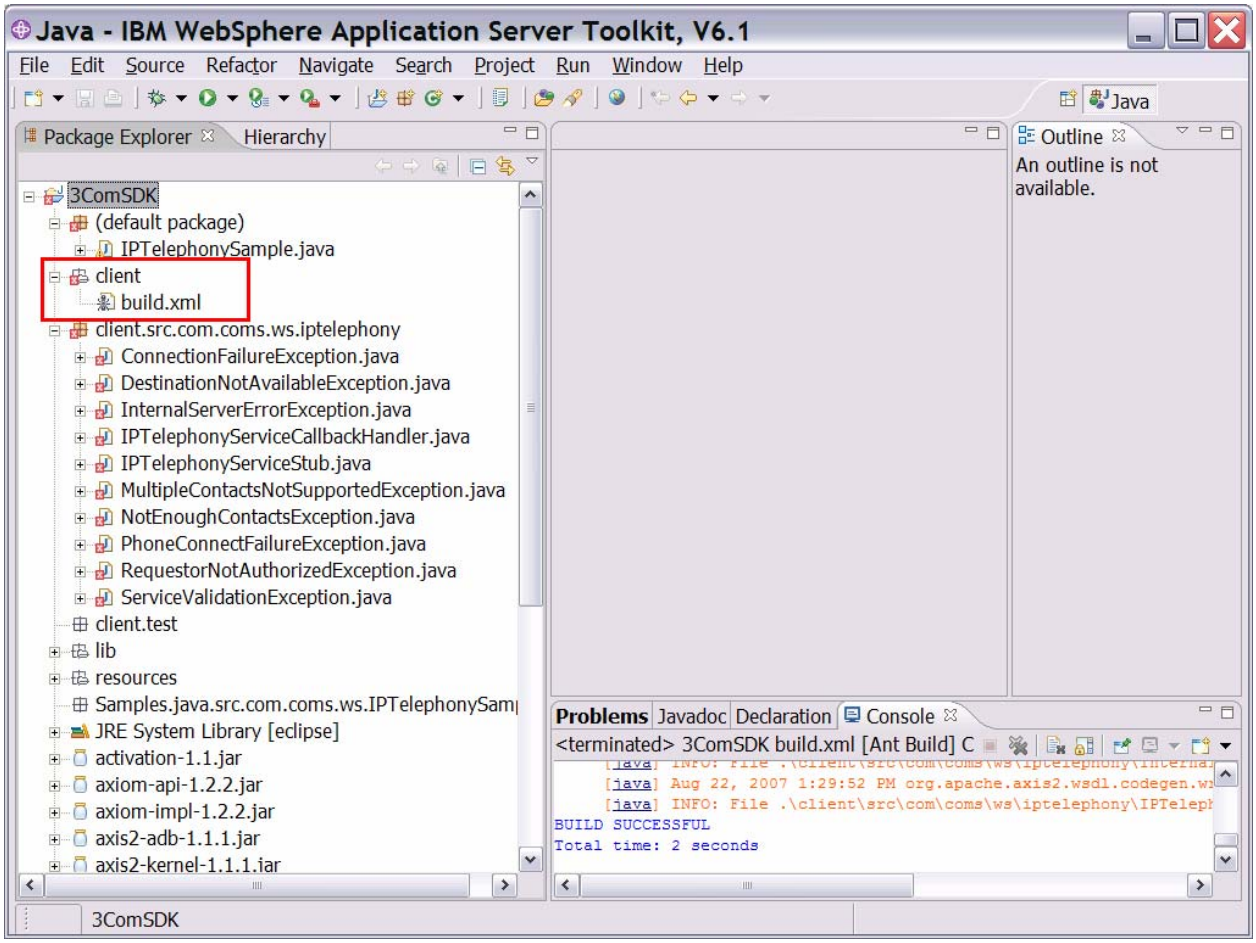

Figure 47. Client stub BUILD.XML

3. Click only **JAR client (default)** to generate the test IPTelephonyClient JAR and select **Run** (see Figure 48).

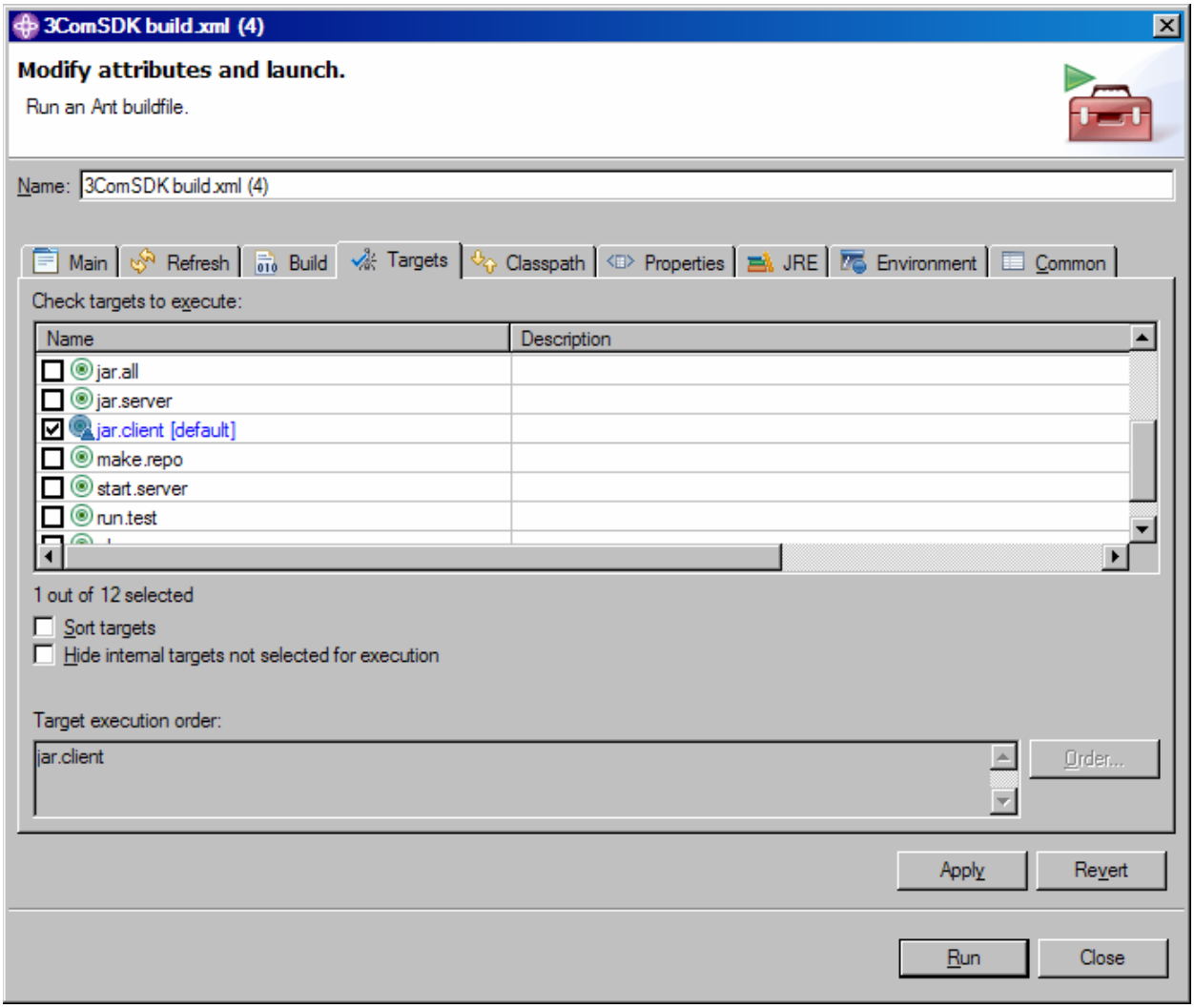

Figure 48. Ant options for client build

Upon successful completion, an IPTelephonyService-test-client.jar file is built (see Figure 49). You can export this file to the System i platform, and modify the SystemDefault.properties file to put this new JAR file into the classpath. Then, you can test the success of this process by running the sample again.

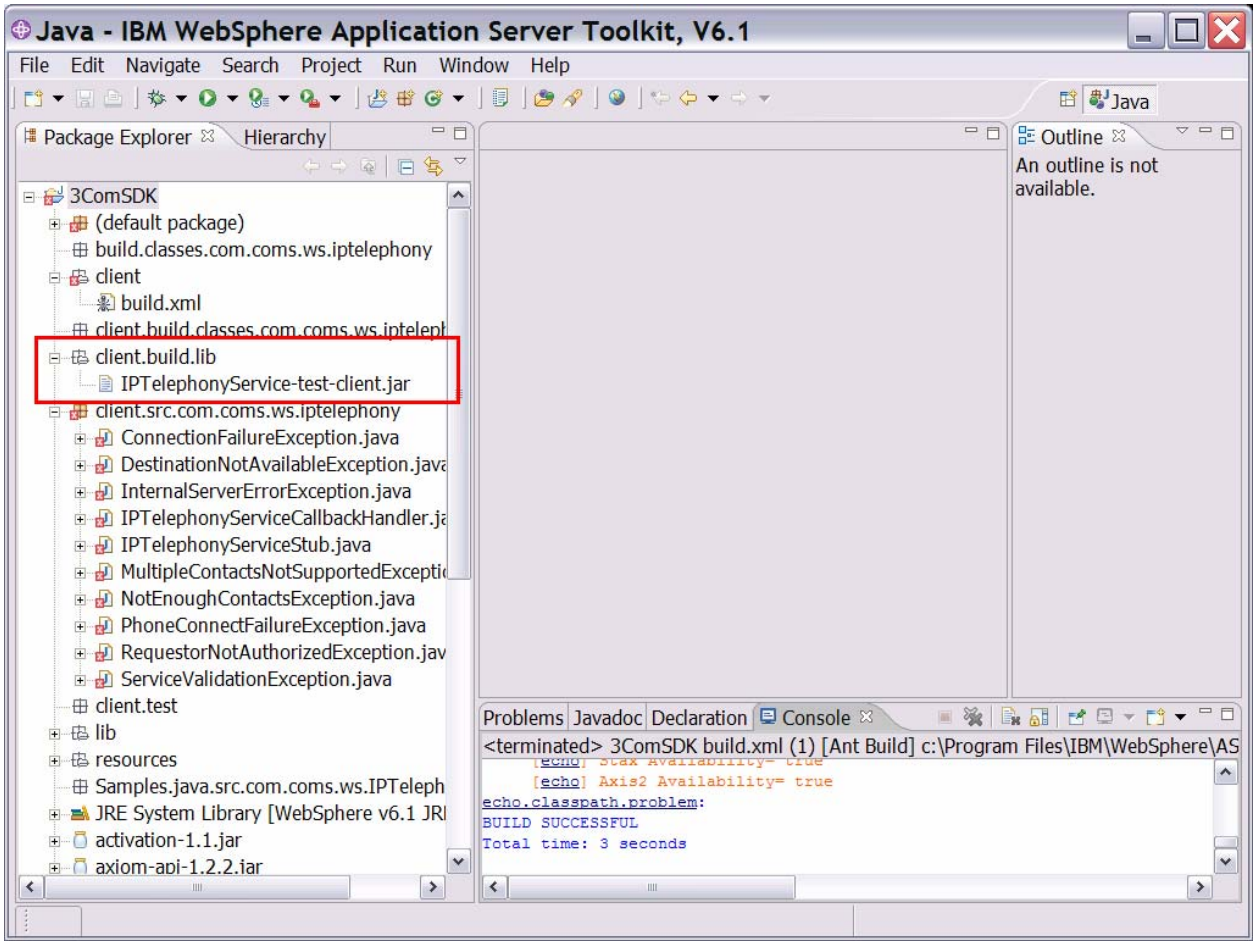

Figure 49. After building the client JAR file

Sample

The sample code included here is architected to use a messaging layer on the System i platform that is implemented by using keyed-data queues. These data queues accept messages from external programs to run 3Com Web-service requests and receive responses back from the 3Com platform. These external programs can be any System i program object that can interact with data queues. Alternatively, they can be Java servlets running in a WebSphere container. Although Java servlets can directly call the 3Com Web services, using this proposed methodology can seem to be more than is necessary. However, this architecture was chosen for this example because any System i application program can use it in a general way to integrate VoIP functions. By using data queues, with which all System i program objects can interact, you can use this example in many application scenarios, RPG, COBOL, C and CL, as well as Java and WebSphere Java applications.

A Java server program monitors the data queues for transactions. It is this Java program that instantiates the IP telephony client stubs that were generated from the IPTelephony.wsdl file in the IDE. Depending on the message on the request data queue, the Java program invokes the appropriate Web service through the client-stub object. A transaction response is posted back to the response data queue from 3Com. (See Figure 50.)

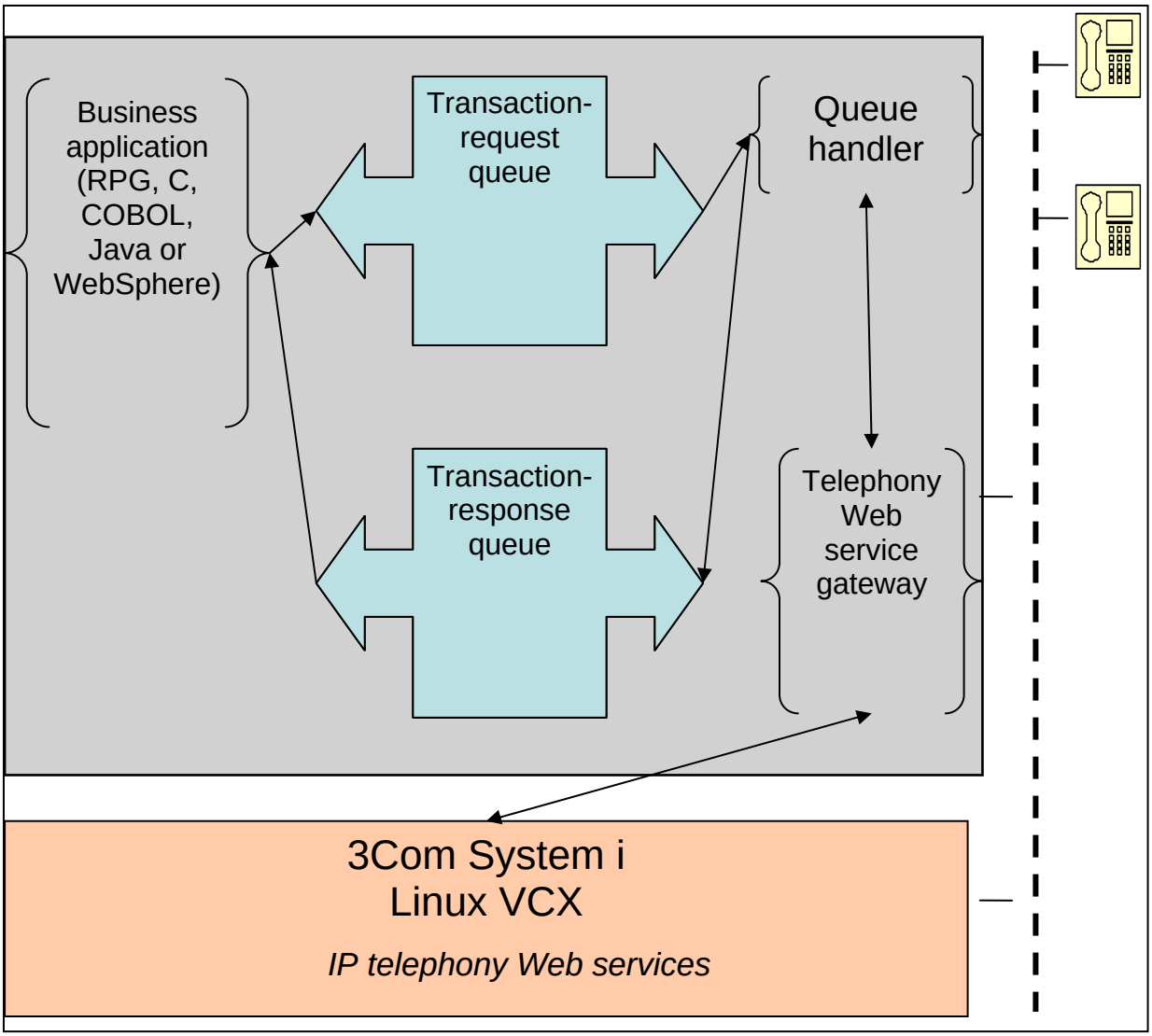

Figure 50. Sample logic flow

Setting up System i keyed-data queues

The samples require that the request and response keyed-data queues are already created on the System i model. You use the i5/OS CRTDTAQ command (through a 5250 session) to create these queues, as Figure 51 and Figure 52 illustrate.

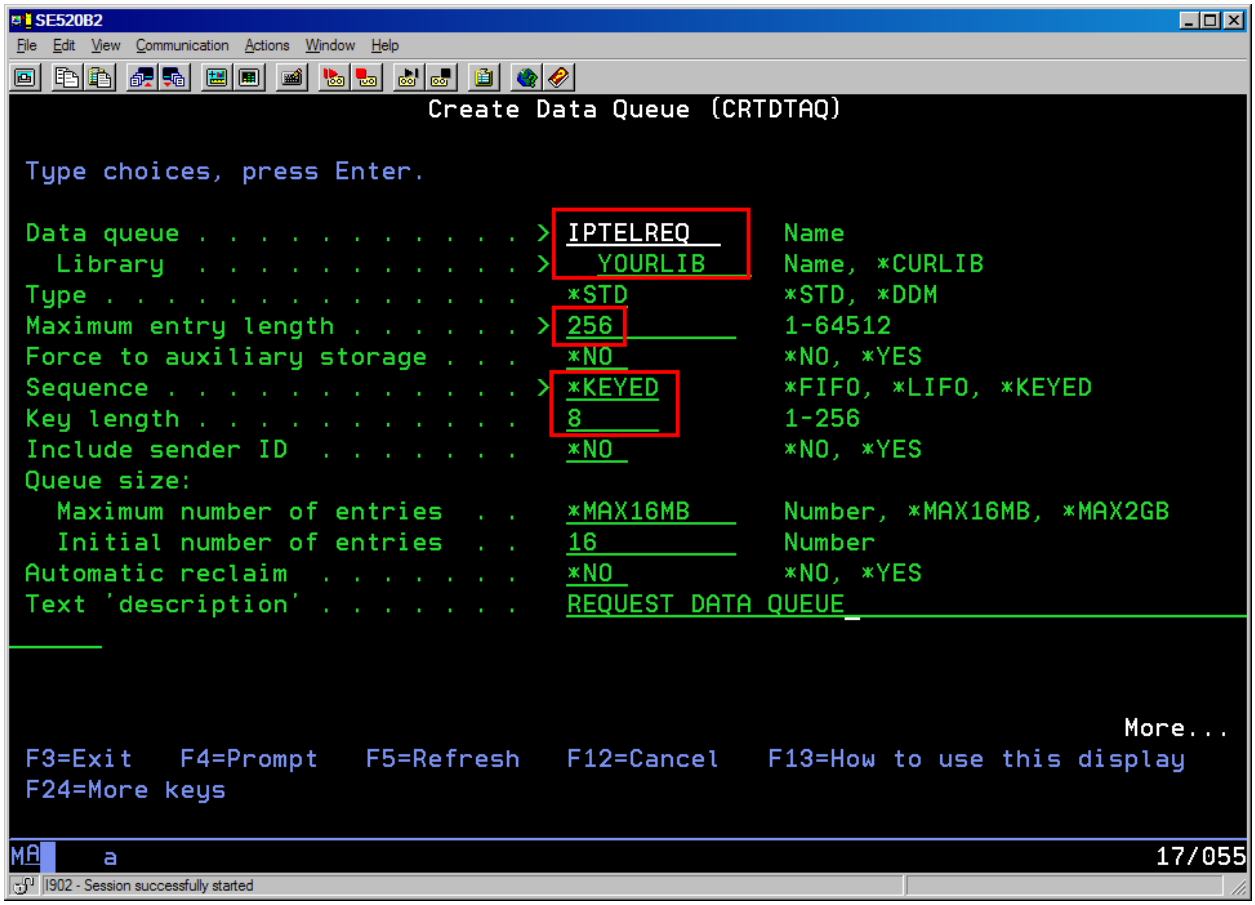

Figure 51. Creating a request keyed-data queue

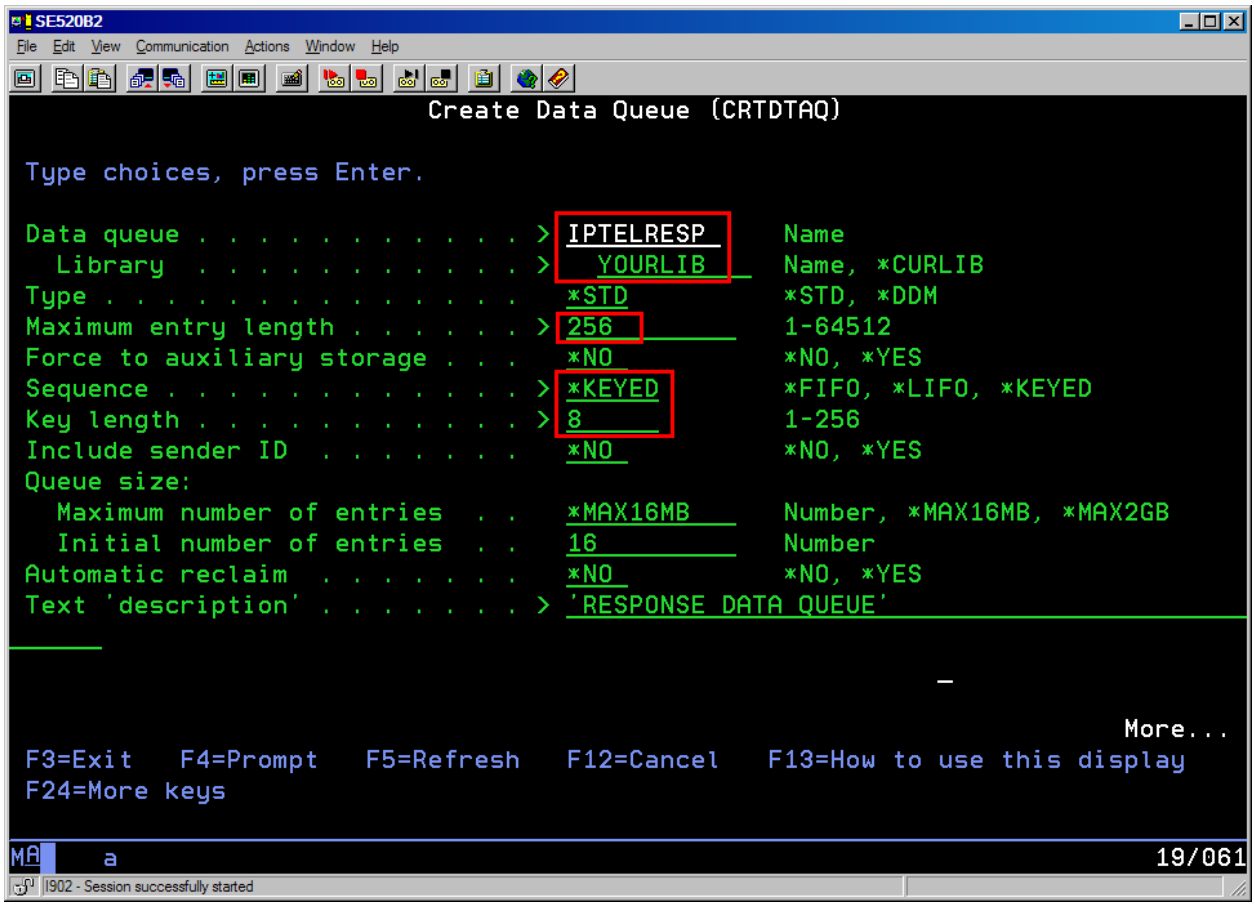

Figure 52. Creating a response keyed-data queue (continued)

The code itself

This section reviews the various segments of the sample code.

The test driver program (*IPTelTransactionDataQueueDriver*)

This module is used to place the transaction on the request keyed-data queue. The key of the message on the queue is the originating telephone number, and the message is formatted as follows:

1. STRING: `makeCall, tranCall, getState, shutdown`
   Note: The first parameter must be eight characters long.
2. STRING: `origination phone number`
3. STRING: `origination phone password`
   Note: This string is configured as part of the 3Com setup.
4. STRING: `destination number`

Figure 53 shows the code that places these transactions on the request keyed-data queue.

```
public static void main(String[] args) {
System.out.println("**Putting " + args[0] + ", " + args[1] + ", " + args[2] + ", "
+ args[3] + " ON DATA QUEUE**");
 try {
  AS400 sys = new AS400("se520b2.rchland.ibm.com","<userid>",”<password>");
  sys.setGuiAvailable(true);
  KeyedDataQueue requestdq = new KeyedDataQueue(sys,
"/QSYS.LIB/JRUSH.LIB/IPTELREQ.DTAQ");
  KeyedDataQueue responsedq = new KeyedDataQueue(sys,
"/QSYS.LIB/JRUSH.LIB/IPTELRESP.DTAQ");
  /*Write the transaction to the queue with the second parameter, the origination
phone number, as the key.*/
  requestdq.write(args[1], args[0]+" " +args[1]+" " +args[2]+" " +args[3]);
  /*WAIT for response on the response keyed data queue with a key that matches the
origination number we used as the key on the write to the request keyed data queue
above.*/
  DataQueueEntry dqData = responsedq.read(args[1], -1, "EQ");
  System.out.println(dqData.getString());
  System.out.println("TRANSACTION COMPLETE");
 } catch (Exception e) {
    System.out.println(e);
 }
}
```

*Figure 53. The test-driver program (*IPTelTransactionDataQueueDriver*)*

The messaging layer (*IPTelephonyDQHandler)*

This code interacts with the keyed request-data queue and accepts incoming requests from external programs, such as the IPTelTransactionDataQueueDriver program in the previous section. This layer instantiates a new server-gateway object.

1. Instantiate the IPTelephonyServerGateway (see Figure 54).

```
public class IPTelephonyDQHandler {
  static IPTelephonyServerGateway ipt = new
IPTelephonyServerGateway(null,"wsuser","wspwd");
```

Figure 54. Instantiating the IPTelephonyServerGateway

2. Create a new AS400 system object for connectivity to the System i model that hosts the keyed-data queues. Create the request and response keyed-data queue objects (see Figure 55).

   Note: The code does not create the data queues on System i, as they were already created. (This was discussed in the “Setting up System i keyed-data queues” section.) The KeyedDataQueue objects that have been created are the objects that communicate with the queues on System i.

```
AS400 sys = new AS400("se520b2.rchland.ibm.com", "<userid>", "<password>");
KeyedDataQueue requestdq = new KeyedDataQueue(sys,
"/QSYS.LIB/JRUSH.LIB/IPTELREQ.DTAQ");
KeyedDataQueue responsedq = new KeyedDataQueue(sys,
"/QSYS.LIB/JRUSH.LIB/IPTELRESP.DTAQ");
```

Figure 55. Creating the KeyedDataQueue objects

3. Loop through the code while there are transactions on the request queue and shut down when a *kill* transaction is requested. Read the request queue when the key is greater than, or equal to, zero. If there is no request, wait until there is a transaction (see Figure 56).

```
while (listen == true) {
   System.out.println("**WAITING FOR IP TELEPHONY TRANSACTION ON DATA QUEUE**");
   /*Wait on any entry on the data queue with key > or = "0". In this    sample we
are using the originating phone number/extension as the key*/
   KeyedDataQueueEntry dqData = requestdq.read("0", -1, "GE");
```

Figure 56. Using loop while *code*

4. Process the entry that was read from the keyed-data queue. In this sample, the phone numbers are all four characters in length (parameters 2 and 4 in the data-queue message). Your environment might be different, so you might have to change the code shown in bold or underlined font (in Figure 57). Based on the type of transaction requested (the first parameter in the data-queue message), the appropriate subroutine is called. For example, the makeCall(a) method is called for a make-call request that passes the other parameters to the subroutine in a string array. The returned response is placed on the response keyed-data queue with the same key that was initially used to place the transaction on the request queue. This is done so that the requesting program (which made the request) can differentiate its response from other request responses that might be placed on the same response queue. This is illustrated with the code: responsedq.write(keyString, respString). (See Figure 57.)

```
String dqEntry = dqData.getString();
/*Save the key so when we put response on the response keyed data queue it
matches the key we used/put on the request keyed data queue.*/
keyString = dqData.getKeyString();
String a[] = { dqEntry.substring(9, 13),
               dqEntry.substring(14, 19),
               dqEntry.substring(20, 24) };
IPTelDirective = (dqData.getString().charAt(0));
switch (IPTelDirective) {
// make call
case 'm':
    makeCall(a);
    //PUT RESPONSE ON RESPONSE DATA QUEUE
    responsedq.write(keyString, respString);
    break;
// disconnect call
case 'd':
    disconnectCall(a);
    responsedq.write(keyString, respString);
    break;
// get phone state
case 'g':
    getPhoneStatus(a);
    responsedq.write(keyString, respString);
    break;
// transfer call
case 't':
    transferCall(a);
    responsedq.write(keyString, respString);
    break;
// shutdown
case 's':
    listen = false;
    responsedq.write(keyString, "Shutting down");
    break;
default:
    break;
}
}
} catch (Exception e) {
      System.out.println(e);
}
}
```

Figure 57. Processing the entry that was read from the keyed-data queue

For example, if the transaction on the queue is *makecall 3001 12345 3002*, the switch statement (in Figure 57) passes control to the makeCall method. This method creates a 3Com Telephony action-type object. The IPTelephonyService WSDL from the SDK defines three action types:

- CallControlActionType (makeCall, transferCall, disconnectCall and others)
- PhoneConfigActionType (mutePhone, handsFree, fwdMailSet and others)
- PhoneStatusActionType (getPhoneState, getFwdMailState, getDndState and others)

The WSDL-to-Java generator created objects for each of these action types that are used to invoke the Web service from the IPTelephonyServerGateway sample code in Figure 58. It then invokes the processCallRequest of the IPTeleponyServerGateway object and gets a response string back from the method invocation. Figure 58 shows the makeCall method.

```
public static void makeCall(String[] args) {
try {
    actionType = CallControlActionType.makeCall;
    respString = ipt.processCallRequest(actionType,args[0],args[1], args[2]);
    } catch (Exception e) {
         e.printStackTrace();
    }
}
```

Figure 58. Using the makeCall method

The methodology for handling the other action types is similar to the makeCall action.

The server gateway (*IPTelephonyServerGateway)*

This is the object code that handles requests coming from the messaging layer. This object instantiates a Web-service client stub (IPTelephonyServiceStub), which was supplied with the SDK or generated in the “Generating the client-service stubs” section. The IPTelephonyServerGateway class is derived from the IPTelephonySample class that was supplied with the 3Com SDK. This class contains the methods that validate the parameters being passed to the Web services. it also contains the methods to create the required objects and Web-service parameters that invoke the SDK client-service stub to invoke the Web services.

1. For example, the code sections in Figure 59 show the code that makes a call. It then passes the response from the Web service back to the IPTelephonyDQHandler.

```
/* Process a Call Control request */
public String processCallRequest(CallControlActionType request,
       String orig, String telepasswd, String dest) {
       origination = orig;
       phonePasswd = telepasswd;
       destination = dest;
       String callResponse = null;
       callParamsValid = validateCallParams();
       if (callParamsValid) {
          try {
            callResponse = sendCallControlRequest(request);
            // PASS RESPONSE BACK
            return callResponse;
           // System.out.println("\n" + callResponse);
         } catch (Exception ee) {
            // System.out.println("\n" + ee.getMessage());
            return ee.getMessage();
          }
      } else {
          return "Parameters Invalid";
      }
}
```

Figure 59. The server-gateway code that makes a call and passes the response to IPTelephonyDQHandler

2. The next section of code creates the objects necessary to build the SOAP Web-service request and then creates a service object to perform and route the request to the 3Com Web-service server (see Figure 60).

```
/* Send the Call Control request */
public String sendCallControlRequest(CallControlActionType actionType)
    throws Exception {
// build request body
    ActionCredentials cred = new ActionCredentials();
    cred.setOriginNumber(origination);
    cred.setPassword(phonePasswd);
    ServiceType ver = new ServiceType();
    ver.setString("Version");
    ver.setAPIVersion("V1");
    CallControlRequest req = new CallControlRequest();
    req.setActionType(actionType);
    req.setCredentials(cred);
    req.setDestinationNumber(destination);
    req.setServiceValidator(ver);
// init the client
    IPTelephonyServiceStub stub = new IPTelephonyServiceStub(ipAddress);
    ServiceClient serviceClient = stub._getServiceClient();
// build security header
    SecurityType sec = new SecurityType(username, appPasswd);
    serviceClient.addHeader(sec.toElement());
// execute the request
    CallControlResponse response = stub.callControlRequest(req);
    ActionResultType resultType = response.getCallControlResponse();
    return actionType.getValue() + " : " + resultType.getValue();
}
```

Figure 60. The server-gateway code that creates the objects to build SOAP Web-service requests

Web-service client stub (*IPTelephonyServiceStub)*

If you generated the stubs in the “Generating the client-service stubs” section, these are the classes contained in the IPTelephonyClient.jar or the IPTelephonyService-test-client.jar files that are the generated interfaces to the 3Com Web services.

1. The SDK comes with a pregenerated stub file that was generated from the IPTelphonyService.wsdl file. It contains the classes required to create a request that is handled by the Web service. The classes include the three request types:

   - *CallControlRequest* for making or transferring calls
   - *PhoneConfigRequest* for setting phone features
   - *PhoneStatusRequest* for querying the state of a 3Com attached phone

   The IPTelephonyServerGateway class instantiates a new service stub for each request it reads from the data queue.

Running the sample

To run the sample, it is necessary to export the IPTelphonyDQHandler and IPTelephonyServerGateway classes to the System i model. The IPTelephonyServiceStub class might already be on the System i model if you went through running the sample GUI program provided by 3Com in the “Migrating and running the sample on System i” section.

1. If not, follow the steps in the “Exporting the code” step in the “Migrating and running the sample on System i” section. Make sure to create an environment variable on the System i JAVA_HOME call and set it to /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit. This causes the IBM 1.5 JVM to be used when the sample runs.

2. Then, export the IPTelephonyDQHandler and IPTelephonyServerGatewway class files to a directory on the System i model where they can be invoked with the Java IPTelephonyDQHandler command in a Qshell environment on the System i model (as shown in Figure 61). The handler starts and waits for an entry to be placed on the request keyed-data queue.

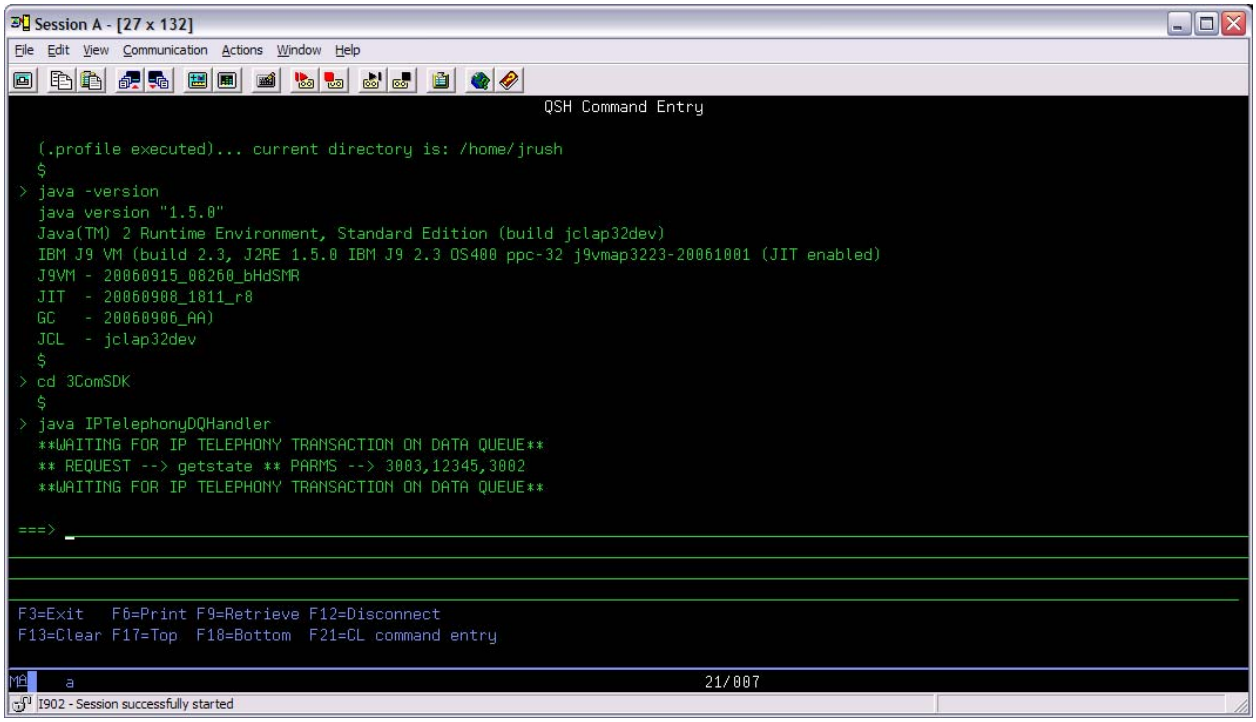

Figure 61. Running the DQ handler and gateway on the System i model

3. In the IDE, the IPTelTransactionDataQueueDriver class runs with arguments to place a transaction on the request queue. In the left-hand navigator, right-click **IPTelTransactionDataQueueDriver**; select **Run As → Run**. You see the panel as shown in Figure 62.

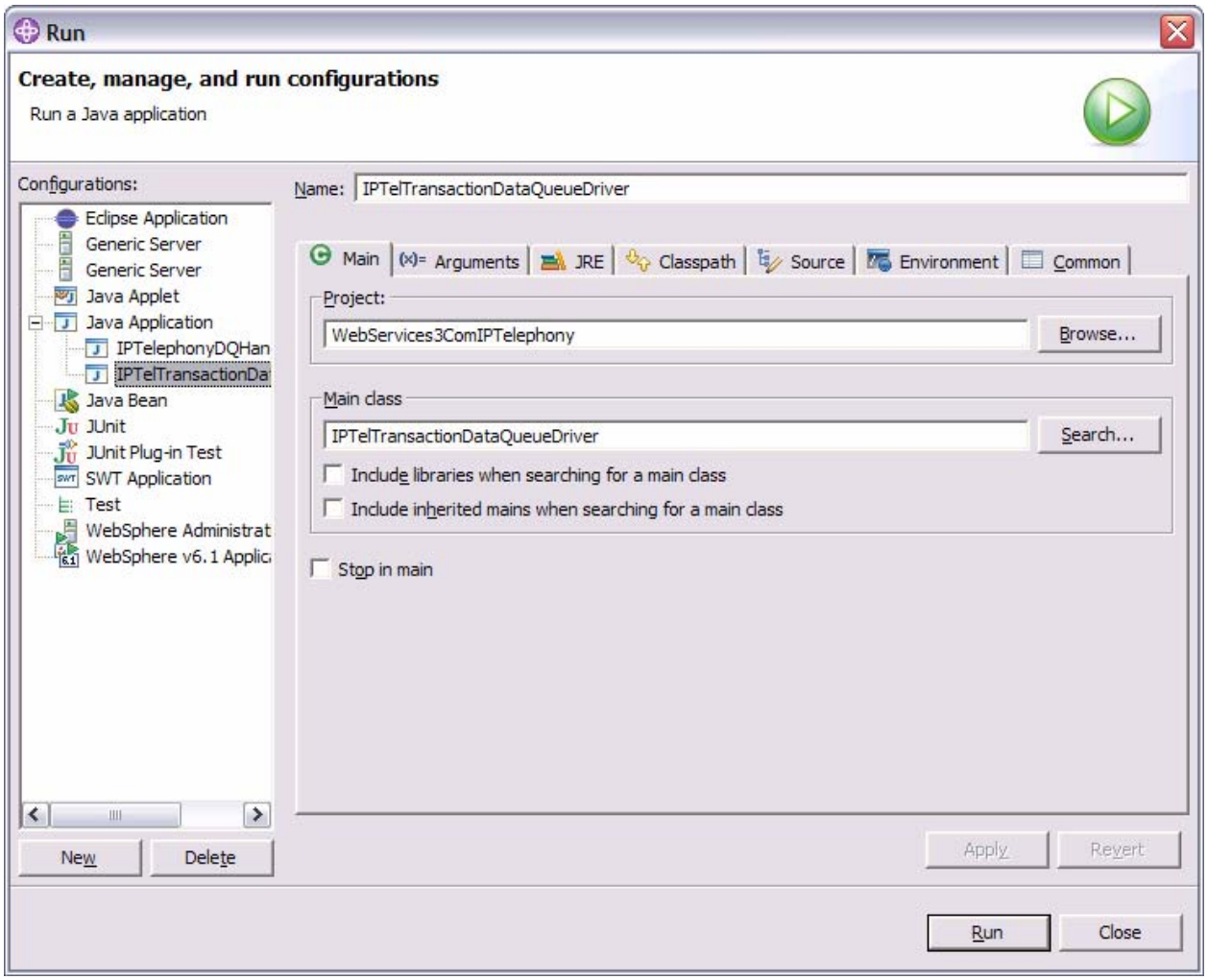

Figure 62. Running the driver program in the IDE

4. On the Arguments tab, put in the parameters, as shown in Figure 63.

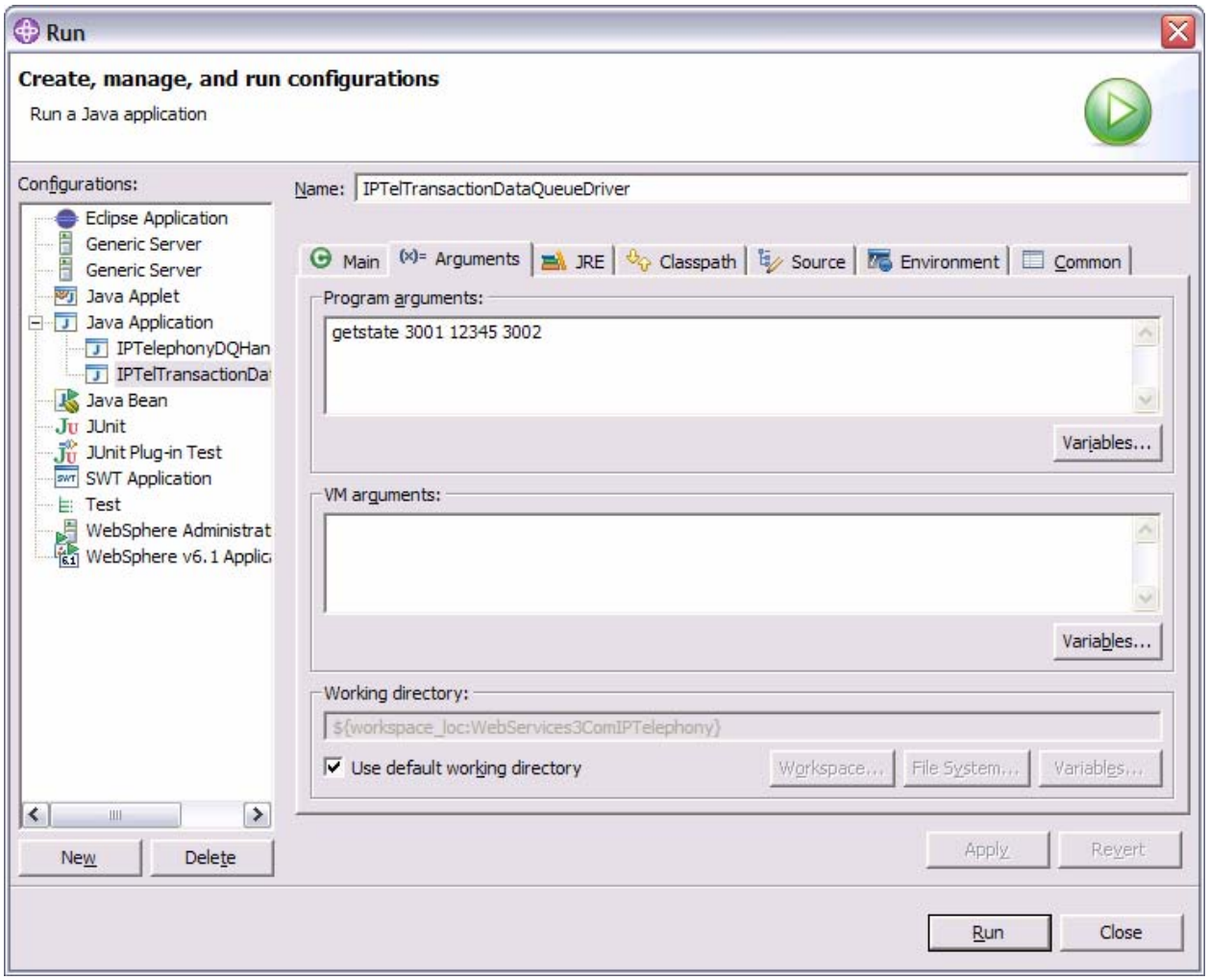

Figure 63. Arguments for the driver program

5. Click **Apply**, then **Run**. The transaction is placed on the data queue. Figure 64 and Figure 65 are displayed. On the System i server-gateway side:

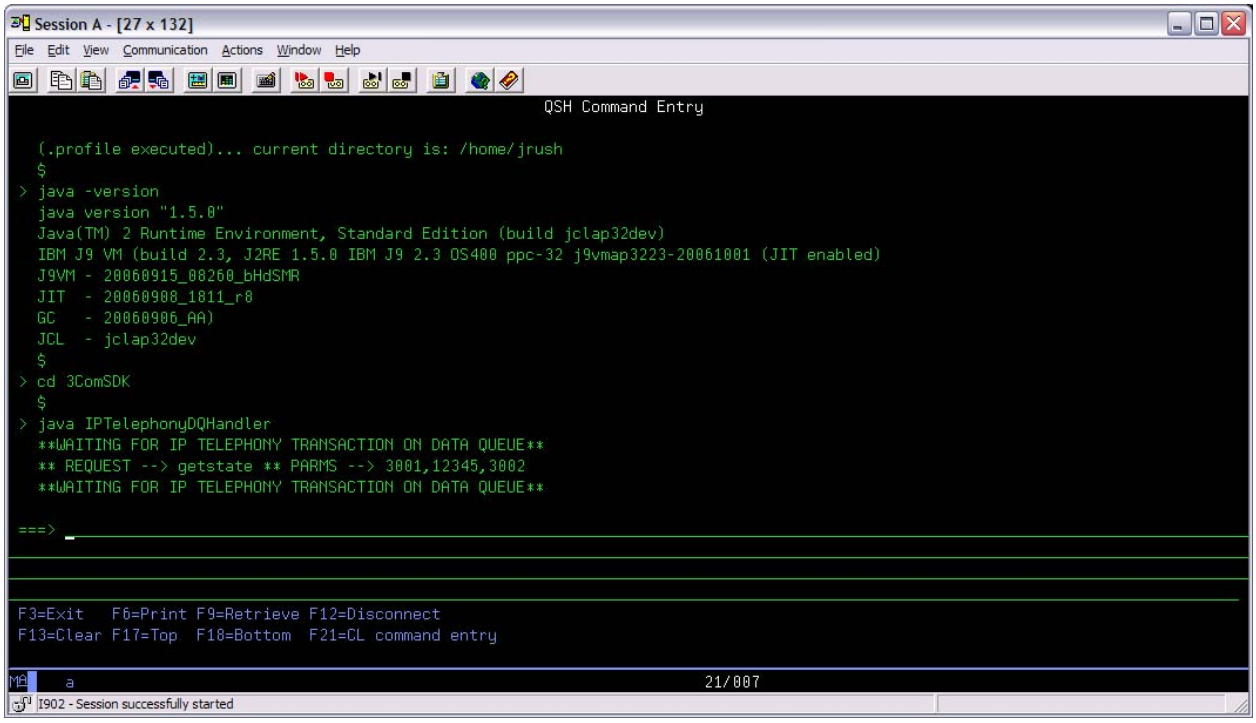

Figure 64. Server-transaction results

On the driver program side:

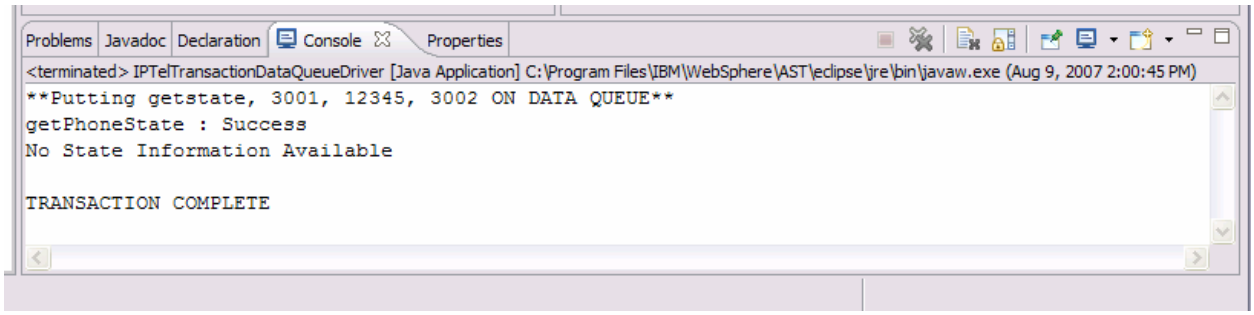

Figure 65. Client-transaction results

6. To shut down the server gateway, run the driver program again with the arguments as shown in Figure 66. Click **Apply**, then **Run**.

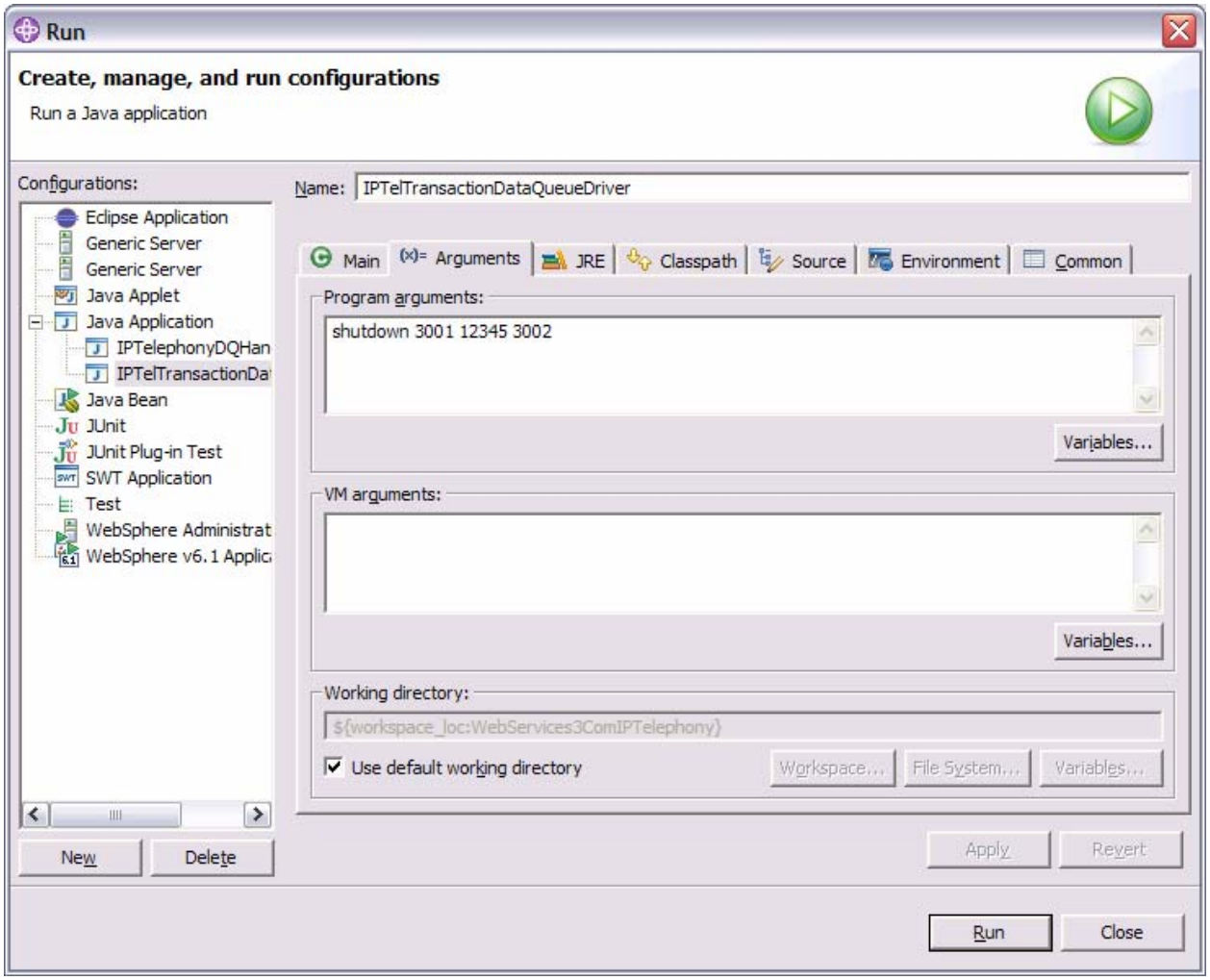

Figure 66. Shutting down the server

Summary

The goal of this white paper was to guide you through the installation and use of the 3Com IP Telephony SDK, allowing integration between business applications that run on the System i platform with the new features of the 3Com voice over Internet Protocol (VoIP) solution.

Resources

These Web sites provide useful references to supplement the information contained in this document:

- IBM eServer i5 Information Center
  http://publib.boulder.ibm.com/iseries/

- IBM Publications Center
  www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi?CTY=US

- IBM System i on IBM PartnerWorld®
  ibm.com/partnerworld/systems/i

- IBM Redbooks®
  ibm.com/redbooks

- 3Com Open Network
  www.open.3com.com/tcom/

- Web Services Description Language (WSDL) to Java tooling
  http://ws.apache.org/axis2/download/1_1_1/download.cgi

- System i Tools for Developers PRPQ (5799PTL)
  www14.software.ibm.com/webapp/download/preconfig.jsp?id=2004-08-18+12%3A25%3A25.057448R&S_TACT=104CBW71&S_CMP=&s=

- Registration with the 3Com Open Networks Partner program
  www.open.3com.com/tcom/

- Using a VNCviewer client or a Web browser
  http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp (search on **VNCviewer**)

About the author

Jon Rush is a technical consultant in ISV Business and Solution Enablement. He is a senior software engineer specializing in WebSphere, IBM Hypertext Preprocessor (PHP) and IP Telephony on the System i platform. Jon has helped hundreds of System i solution providers enhance their applications to use IBM e-business technologies such as IBM Net.Data®, WebSphere and PHP.

Trademarks and special notices

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

i5/OS, IBM, the IBM logo, Net.Data, PartnerWorld, Rational, Redbooks, System i and WebSphere are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.