

# **Web Services Integration Toolkit for OpenVMS**

---

## **Developer's Guide**

September 2005

This document contains information that will help you use the development tools in this release of WSIT for OpenVMS.

### **Software Version**

Web Services Integration Toolkit Version 1.0

Hewlett-Packard Company  
Palo Alto, Calif.

---

**© 2005 Hewlett-Packard Development Company, L.P.**

Intel, Intel Inside, and Itanium are trademarks of Intel Corporation in the U.S. and/or other countries.

Microsoft, Windows, Windows XP, Visual Basic, Visual C++, and Win32 are trademarks of Microsoft Corporation in the U.S. and/or other countries.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All other product names mentioned herein may be trademarks of their respective companies.

Confidential computer software. Valid license from HP and/or its subsidiaries required for possession, use, or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Neither HP nor any of its subsidiaries shall be liable for technical or editorial errors or omissions contained herein. The information in this document is provided "as is" without warranty of any kind and is subject to change without notice. The warranties for HP products are set forth in the express limited warranty statements accompanying such products. Nothing herein should be construed as constituting an additional warranty.

# **C O N T E N T S**

---

## **INTRODUCTORY INFORMATION – FOR ALL USERS**

1. USING THE WEB SERVICES INTEGRATION TOOLKIT FOR OPENVMS
    - 1.1 Overview
    - 1.2 Preparing the Original (Legacy) Application
    - 1.3 Exposing an OpenVMS 3GL Application: Typical Development Steps
    - 1.4 Wrapping a 3GL Application: C Sample
    - 1.5 Wrapping an ACMS Application: ACMS Sample
  2. DEPLOYMENT CONSIDERATIONS
    - 2.1 Types of OpenVMS Applications
    - 2.2 In-Process Deployment
    - 2.3 Out-of-Process Deployment
      - 2.3.1 Sessions
      - 2.3.2 Application Reusability
      - 2.3.3 Using Multiple Processes to Scale Applications
      - 2.3.4 Specifying Out-of-Process Deployment Options
- 

## **ADVANCED INFORMATION – FOR EXPERIENCED USERS**

3. ADVANCED OUT-OF-PROCESS CONFIGURATION
  - 3.1 Case A: Not Reusable
  - 3.2 Case B: Sequentially Reusable
  - 3.3 Case C: Concurrently Reusable
  - 3.4 Case D: Concurrently Reusable with Multiple Threads
4. USING TEMPLATES TO GENERATE CODE
  - 4.1 Modifying Velocity Templates
  - 4.2 Generating Code with idl2code.jar
  - 4.3 Example 1: Writing a New Template
  - 4.4 Example 2: Modifying an Existing Template

## **APPENDIX**

- A. Program Listing - STOCK.C
  - B. Program Listing - STOCK.XML
  - C. Program Listing - StockCaller.Java
-

## **About Web Services Integration Toolkit for OpenVMS Documentation**

This *Developer's Guide* contains information about how to use the tools in the Web Services Integration Toolkit for OpenVMS, and things to consider as you prepare your legacy application.

The *Installation Guide and Release Notes* includes system requirements and installation instructions for OpenVMS, as well as release notes for the current release of the Web Services Integration Toolkit for OpenVMS.

For the latest release information, refer to the Web Services Toolkit for OpenVMS web site at <http://www.hp.com/products/openvms/webservices/>.

## USING WEB SERVICES INTEGRATION TOOLKIT

### 1.1 Overview

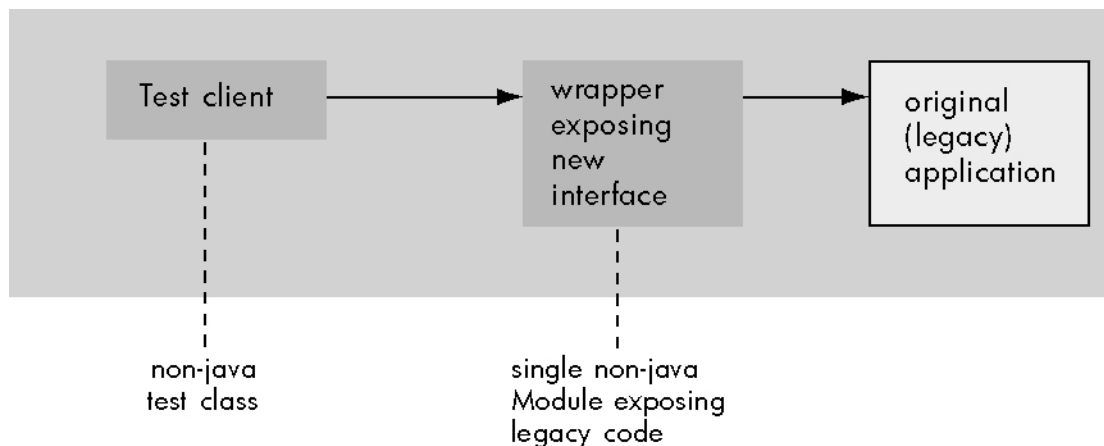
The Web Service Integration Toolkit for OpenVMS (WSIT) contains a collection of integration tools. These tools are easy to use, highly extensible, based on standards and built on open source technology. The toolkit can be used to call OpenVMS applications written in 3GL languages, such as C, BASIC, COBOL, FORTRAN, and ACMS from newer technologies and languages such as Java, Microsoft .NET, Java -RMI, JMS, and web services.

The Web Service Integration Toolkit is focused on integrating at the API level. It generates a JavaBean wrapper for a supplied OpenVMS application interface (API). At runtime, a deployment descriptor specifies if the application will be run in the process of the caller (in-process) or in separate processes (out-of-process) managed by the WSIT runtime.

### 1.2 Preparing the Original (Legacy) Application

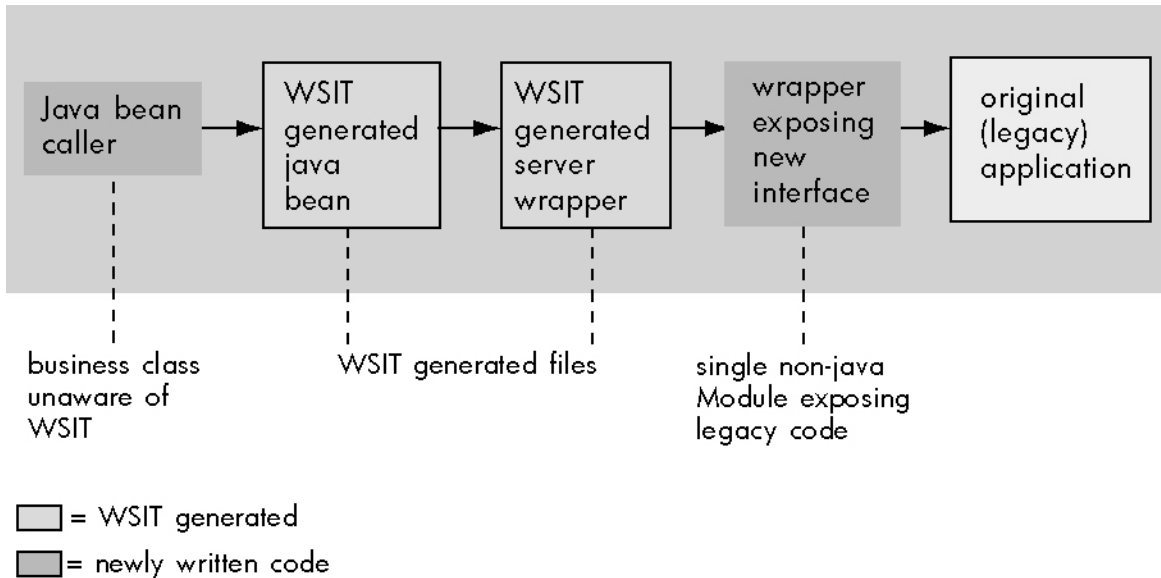
Using the Web Services Integration Toolkit for OpenVMS, as with all programmatic integration, requires some upfront development work before you can begin performing the integration. Your existing application is likely to have been written long ago and will benefit from having a *wrapper* expose a new and clean interface. The new interface will expose the legacy implementation. Separating the interface from the implementation provides encapsulation and the ability to easily extend and reuse the implementation.

Before you use the Web Services Integration Toolkit or any other integration technology, you must evaluate the original application and *design one or more interface classes* to expose different features of the business logic. These new interfaces should be tested with a simple client before you use the Web Services Integration Toolkit. When you know that the interface classes are working properly, you can use WSIT to extend the use of the new interface to the Java environment.



■ = newly written code

After you have prepared the application, WSIT can extend the features of the new interface to Java as shown in the following diagram.



### 1.3 Exposing an OpenVMS 3GL Application: Typical Development Steps

Following are the development steps required to use the Web Services Integration Toolkit to expose an OpenVMS 3GL or ACMS application. Note that these steps are only required for the development phase. It is expected that the application has been prepared as discussed in the previous section.

**Note:** Beginning in the T1.4 field test kit, these tools have been renamed. See Chapter 3 in the *Installation Guide and Release Notes* for a table containing the old and new file names.

#### 1. Create XML IDL file (on I64)

Create an XML interface definition file (IDL) that describes the interface to be exposed. You create an XML IDL file using the tool named **OBJ2IDL.JAR** (for 3GL languages) or **STD2IDL.JAR** (for ACMS). **Note:** OBJ2IDL.JAR runs on OpenVMS I64 only. If you are using WSIT on OpenVMS Alpha, you can create the XML IDL file manually in any editor using the sample file `WSI$ROOT:[SAMPLES.C]STOCK.XML` as a guide. If you have both OpenVMS I64 and Alpha systems, run **OBJ2IDL.JAR** on I64 and copy the resulting XML IDL file to your Alpha system.

#### 2. Validate XML IDL file

Verify that the XML IDL file correctly describes the interface being exposed. If it does not, manually update the XML IDL file until the interface definition is correct. The **VALIDATE.JAR** allows you to verify that an XML IDL file conforms to the `openvms-integration.xsd` schema.

#### 3. Generate components

For the interface being exposed, generate one WSIT server interface wrapper and one WSIT Java Bean using **IDL2CODE.JAR**. The generated source code must be built on the OpenVMS system that hosts the application.

#### 4. Use the generated code

Call the generated WSIT JavaBean from the technology of your choice, including BEA WLS, Apache Axis, JMS, Java RMI, J2EE or another JavaBean.

## 1.4 Wrapping a 3GL Application: C Sample

The following steps demonstrate how to wrap a 3GL application using the *stock* sample program found in `WSI$ROOT:[SAMPLES.C]`. Other 3GL sample programs can be found in `WSI$ROOT:[SAMPLES.COBOLE]` and `WSI$ROOT:[SAMPLES.FORTRAN]`. (See Section 1.5 for information about a sample program that wraps an ACMS application.)

The information in this section is also included in `WSI$ROOT:[SAMPLES.C]STOCK-SAMPLE.README`.

### **Step 1: Generate an Interface Definition with OBJ2IDL**

The tool `obj2idl.jar` is used to generate an XML interface definition.

#### **Establish a foreign command:**

```
$ obj2idl = "$WSI$ROOT:[tools]obj2idl.exe"
```

#### **Compile the wrapper that exposes the new interface:**

```
$ set def WSI$ROOT:[samples.c]
$ cc/debug/noopt stock.c
```

#### **Use `obj2idl` to generate an xml file with the interface definition:**

```
$ obj2idl -f WSI$ROOT:[samples.c]stock.obj
```

The tool `obj2idl` creates the file `stock.xml`. See the Appendix for a full listing of `stock.xml`.

You should become familiar with the XML description of OpenVMS applications. Review the `stock.xml` file and notice the overall structure of the file. Following are the level 1 tags used to define an interface. These tags contain lower level tags and more information.

- <Primitives> define the fundamental types referenced in the interface.
- <TypeDefs> define a mapping between types, if used in the interface.
- <Structures> define the user-defined structure, if used in the interface.
- <Routines> define the callable routines of the interface.

**Hint:** To view the XML file with coloring and a collapsible outline, use Internet Explorer.

### **Step 2: Validate the Generated XML File**

The `obj2idl` tool is sometimes unable to extract a complete interface definition from the supplied object file. When the tool is missing data or has made assumptions, a comment is placed in the XML file below the line of concern.

The file `stock.log` is also generated from `obj2idl`. Use this file to conveniently see an overview of the comments within the XML file. (ACMS does not create a .log file.)

```
$ ty stock.log
Generated IDL file: WSI$ROOT:[samples.c]stock.xml
Tue Apr 5 11:22:37 2005
```

In this case the tool did not report any issues. However, even in cases where the log file has not generated any error or warning, you should always review the XML file to ensure that the interface definition is exactly correct. It is very important that the XML IDL describe the interface accurately to generate correct code in Step 3.

The **validate.jar** tool is provided to allow you to verify that an XML IDL file conforms to the `openvms-integration.xsd` schema. Use this tool to validate all XML IDL files before they are passed to the `idl2code` tool. The `idl2code` tool does not validate the XML IDL file.

The `validate` tool is an executable JAR file. To run the tool, you must supply two parameters: an XML IDL file and the `openvms-integration.xsd` schema. For example:

```
$java -jar wsi$root:[tools]validate.jar -x wsi$root:[samples.c]stock.xml -s
wsi$root:[tools]openvms-integration.xsd
```

### **Step 3: Generate WSIT Components with IDL2CODE**

Use the tool `idl2code.jar` (also called the Generator) to create a server wrapper for the application and a JavaBean client. This tool requires certain JAR files to be in the Java classpath. A command procedure is supplied to add these files to the `java$classpath` logical. (The `java$classpath` logical lets you define a class path using OpenVMS file specification syntax. Defining this logical overrides the `classpath` logical, if set.)

```
$ @WSI$ROOT:[tools]wsi-setenv - wsi$dev
The New JAVA$CLASSPATH is:
  "JAVA$CLASSPATH" = "WSI$ROOT:[LIB]WSIRTL.JAR" (LNM$JOB_86F82E00)
    = "[]"
    = "WSI$ROOT:[LIB]VELOCITY-DEP-1_4.JAR"
    = "WSI$ROOT:[TOOLS]IDL2CODE.JAR"
$
```

To generate files for the stock demo, use the following command. In this case we pass the tool the `stock.xml` file and we call the application `stock`. We also place all generated file for the application in a subdirectory named `generated`.

```
$ create/dir [.generated]
$ java "com.hp.wsi.Generator" -i stock.xml -a stock -o [.generated]
Master file listing successfully generated.
File: ./generated/stock_server/build-stock-server.com generated.
File: ./generated/stock_server/methIds.h generated.
File: ./generated/stock_server/structkeys.h generated.
File: ./generated/stock_server/stock.wsi generated.
File: ./generated/stock_server/stock.opt generated.
File: ./generated/stock_server/stock-server.h generated.
File: ./generated/stock_server/stock-server.c generated.
File: ./generated/stock_server/build-stock-jb.com generated.
File: ./generated/stock/Istock.java generated.
File: ./generated/stock/_buyerData.java generated.
File: ./generated/stock/_sellerData.java generated.
File: ./generated/stock/_tickerData.java generated.
*** Application stock generated! ***
$
```



## Build the server

```
$ set def WSI$ROOT:[samples.c.generated.stockserver]
$ @BUILD-STOCK-SERVER
Begin server build procedure.
  ..configuring switches and compiler options
  ..compiling native server code
  ..linking shareable image
  ..installing server image
End server build procedure.
$
```

## Build the client

The JavaBean build procedure creates a JAR file that contains the WSI Java classes used to call the server created earlier.

```
$ SET DEF WSI$ROOT:[samples.c.generated.stock]
$ @BUILD-STOCK-JB
Begin java bean build procedure.
The New JAVA$CLASSPATH is:
  "JAVA$CLASSPATH" = "WSI$ROOT:[LIB]WSIRTL.JAR" (LNM$JOB_86D5AE00)
    = "[]"
    = "WSI$ROOT:[LIB]VELOCITY-DEP-1_4.JAR"
    = "WSI$ROOT:[TOOLS]IDL2CODE.JAR"
  ..Compiling structure classes
  ..Compiling stock Interface classes
  ..Creating stock.JAR file from classes
End of JavaBean build procedure.
```

### **Step 4a: Run the Stock Sample “In-Process”**

You can run the stock sample program in-process or out-of-process. In-process means that the application will be run in the process of the caller. (Follow the instructions in step 4b instead of 4a if you want to run the sample out-of-process.)

Add the stock.jar file to the java\$classpath.

```
$ @WSI$ROOT:[tools]wsi-setenv - WSI$ROOT:[samples.c.GENERATED.stock]stock.jar
The New JAVA$CLASSPATH is:
  "JAVA$CLASSPATH" = "WSI$ROOT:[LIB]WSIRTL.JAR" (LNM$JOB_86F82E00)
    = "[]"
    = "WSI$ROOT:[LIB]VELOCITY-DEP-1_4.JAR"
    = "WSI$ROOT:[TOOLS]IDL2CODE.JAR"
    = "WSI$ROOT:[SAMPLES.C.GENERATED]STOCK.JAR"
$
```

In the normal course of development, you would now need to write a JavaBean to call the stock JavaBean that was generated above. However, for the purpose of this demonstration, a JavaBean file named stockcaller.java is provided in the directory WSI\$ROOT:[samples.c]. See the Appendix for a source listing of this file.

### **Look at the provided Java file**

Compile the supplied JavaBean:

```
$ set def WSI$ROOT:[samples.c]
$ javac stockcaller.java
```

Run the supplied JavaBean:

```
$ java stockcaller
The sellers number_shares_available: 1000
The sellers number_shares_available: 5
$
```

The output from the Java program illustrates that the seller's number of shares available to sell is reduced by the number of shares bought by the buyer. The object `seller` contains a data class `sellerData`. `SellerData` maps directly to the `sellerData` structure in the C wrapper that is called. In this case the field `number_shares_available` is modified by the C application on return from the `buy` Java method.

#### **Step 4b: Run the Stock Sample “Out-of-Process”**

You can run the stock sample program either in-process or out-of-process. **Out-of-process** means that the sample will be run in a separate process managed by the WSIT runtime. (Follow the instructions in step 4a instead of 4b if you want to run the sample in-process.)

Add the `stock.jar` file to the `java$classpath`.

```
$ @WSI$ROOT:[tools]wsi-setenv - WSI$ROOT:[samples.c.GENERATED]stock.jar
The New JAVA$CLASSPATH is:
  "JAVA$CLASSPATH" = "WSI$ROOT:[LIB]WSIRTL.JAR" (LNM$JOB_86F82E00)
    = "[]"
    = "WSI$ROOT:[LIB]VELOCITY-DEP-1_4.JAR"
    = "WSI$ROOT:[TOOLS]IDL2CODE.JAR"
    = "WSI$ROOT:[SAMPLES.C.GENERATED]STOCK.JAR"
$
```

In the normal course of development, you would now need to write a JavaBean to call the stock JavaBean that was generated above. However, for the purpose of this demonstration, a JavaBean file named `stockcaller.java` is provided in the directory `WSI$ROOT:[samples.c]`.

See the Appendix for a source listing of this file.

#### **Modify the provided Java file**

To run the stock sample out-of-process, you need to make the following changes to the `stockcaller.java` file:

- Call a different constructor
- Call release when done with server

These changes are highlighted in **red** in the following listing of `stockcaller.java`.

```
$ type stockcaller.java
import stock.*;
import java.io.*;
import com.hp.wsi.WsiIpcContext;
import javax.xml.rpc.holders.StringHolder;
import javax.xml.rpc.holders.StructureHolder;
```

```

public class stockcalleroutproc {
    /** Creates a new instance of Main */
    public stockcalleroutproc() {
    }

    public static void main(String[] args) {

        try {

            stockImpl stock = new stockImpl(new WsiIpcContext());

            // create a seller object and place in holder
            _sellerData sellerData = new _sellerData("Mr Seller", 12345, 1000000, 1000);
            StructureHolder seller = new StructureHolder(sellerData);

            // create a buyer object and place in holder
            _buyerData buyerData = new _buyerData("Mr Buyer", 67890, 5000, 995);
            StructureHolder buyer = new StructureHolder(buyerData);

            // create a ticker object and place in holder
            _tickerData tickerData = new _tickerData("HPQ", "Hewlett Packard");
            StructureHolder ticker = new StructureHolder(tickerData);

            System.out.println("The sellers number_shares_available: " +
            sellerData.getNumber_shares_available());
            stock.buy(27, ticker, seller, buyer);
            System.out.println("The sellers number_shares_available: " +
            sellerData.getNumber_shares_available());

            stock.remove();

        } catch (Exception e) {
            System.out.println("Exception thrown");
        }
    }
}

```

**Important:** Review `WSI$ROOT:[DEPLOY]STOCK.WSI`. By default, the deployment configuration file is the most restrictive. It assumes the application is not reusable, therefore it needs a new server process for every client. After evaluating your application, you can modify `stock.wsi` to scale the deployment configuration for the application. See Chapter 2, Deployment Considerations, for more information.

#### Compile the supplied JavaBean:

```

$ set def WSI$ROOT:[samples.c]
$ javac stockcaller.java

```

#### Run the supplied JavaBean:

```

$ java stockcaller
The sellers number_shares_available: 1000
The sellers number_shares_available: 5
$

```

The output from the Java program illustrates that the seller's number of shares available to sell is reduced by the number of shares bought by the buyer. The object `seller` contains a data class `sellerData`. `SellerData` maps directly to the `sellerData` structure in the C wrapper that is called. In this case the field `number_shares_available` is modified by the C application on return from the `buy` Java method.

## 1.5 Wrapping an ACMS Application: ACMS Sample

The following steps demonstrate how to wrap an ACMS application using the sample program found in `WSI$ROOT:[SAMPLES.ACMS]`. Sample programs written in 3GL languages can be found in `WSI$ROOT:[SAMPLES.C]`, `WSI$ROOT:[SAMPLES.COBOLE]`, and `WSI$ROOT:[SAMPLES.FORTRAN]`. (See Section 1.4 for information about a sample program that wraps a C application.)

The information in this section is also included in `WSI$ROOT:[SAMPLES.ACMS]ACMS-SAMPLE.README`.

**IMPORTANT:** Before you run this sample program, make sure ACMS is properly configured and running on your system.

This ACMS application exists in a nondistributed environment and illustrates some common functions of an administrative system using an Rdb database. For example, in this system, a user adds a new employee record to a master file or updates an existing employee record.

The following files (a modified version of the *Getting Started* tutorial included with ACMS for OpenVMS) are installed by the Web Services Integration Toolkit for OpenVMS installation in the `WSI$ROOT:[SAMPLES.ACMS]` directory:

```
ACMSCALLER.JAVA;1          ACMSEXAMPLE_SETUP.COM;1  ACMS-SAMPLE.README;1
EMPLOYEE_INFO_APPL_WSI.ADF;1  WSI_ADD_EMPL_INFO.TDF;1
WSI_EMP_INFO_TASK_GROUP.GDF;1  WSI_GET_EMPL_INFO.TDF;1
WSI_PUT_EMPL_INFO.TDF;1
```

To run the Web Services Integration Toolkit ACMS sample program, perform the following steps.

### **Step 1: Execute the WSIT-supplied command file to set up the ACMS application**

On the OpenVMS system on which you installed WSIT, log in using an account with SYSTEM privileges.

Create a directory to set up the application. For example:

```
$ create /dir [.acmsgenerated]
```

Set default to the newly created directory:

```
$ set def [.acmsgenerated]
```

Execute the following command:

```
$ @WSI$ROOT:[samples.acms]acmsexample_setup.com
```

This assumes that the `ACMS$EXAMPLES` logical is present and correct on your system.

This DCL script does the following:

- Creates a local data dictionary for this application
- Defines a CDD (common dictionary data) record (using the supplied .CDO files)
- Defines a CDD entry task
- Builds the application, generating a STDLE file (used to import ACMS task and structure definitions)
- Starts the ACMS application.

When prompted for a CDD directory, you can press Enter to accept the default (which will be under the directory you just created and set default to), or you may choose another name or location.

For example:

```
CDD Directory? DKA100:[USER.ACMSGGENERATED.DICTIONARY] :
```

The sample application is set up and started when you see the following:

```
%ACMSINS-S-ADBINS, Application
DISK: [USER.ACMSGGENERATED]EMPLOYEE_INFO_APPL_BWX.ADB;
has been installed to ACMS$DIRECTORY
```

### **Step 2: Generate an interface definition with STD2IDL.JAR**

Use the `std2idl.jar` tool to generate an XML interface definition (IDL file) from the STD file generated in Step 1.

Run the `std2idl` importer:

```
$ java -classpath WSI$ROOT:[TOOLS]std2idl.jar "com.hp.wsi.Import"
-a AcmsApp -f EMPLOYEE_INFO_APPL_WSI.STDL
Import File was successfully processed.
File: ./acmsapp2_idl.xml generated.
*** Files for Application acmsapp2 successfully generated! ***
```

### **Step 3: Review and validate the generated XML file**

Because STD files completely describe the ACMS application, the `std2idl` tool is able to use the STD file to create a complete WSIT interface definition representation of that ACMS application. However, even if the `std2idl` tool specifies that the IDL generation was successful, you should review and validate the generated XML file to ensure complete accuracy. The XML IDL must accurately describe the interface to generate correct code in Step 4.

For this reason, WSIT includes the `validate.jar` tool to allow you to verify that an XML IDL file conforms to the `openvms-integration.xsd` schema before it is passed to the `idl2code.jar` tool. (The `idl2code.jar` tool does not validate the XML IDL file.) To run the `validate.jar` tool, supply two parameters: an XML IDL file and the `openvms-integration` schema. For example:

```
$java -jar wsi$root:[tools]validate.jar
-x wsi$root:[samples.acms]AcmsApp_idl.xml
-s wsi$root:[tools]openvms-integration.xsd
```

### **Step 4: Generate WSIT components with idl2code.jar**

Use the `idl2code.jar` tool to create a server wrapper for the application and a JavaBean client. This tool requires certain Jar files to be in the Java classpath. A command procedure is supplied to add these files to the `java$classpath` logical.

```
$ @WSI$ROOT:[tools]wsi-setenv - wsi$dev
The New JAVA$CLASSPATH is:
"JAVA$CLASSPATH" = "WSI$ROOT:[LIB]WSIRTL.JAR" (LNM$JOB_86F82E00)
= "[]"
= "WSI$ROOT:[LIB]VELOCITY-DEP-1_4.JAR"
```

```
= "WSI$ROOT:[TOOLS]IDL2CODE.JAR"
$
```

To generate files for the ACMS sample, use the following command. In this case we pass the tool the `AcmsApp_idl.xml` file (generated above) and we call the application `AcmsApp`. We also place all generated files for the application in a subdirectory named `generated`.

```
$ create/dir [.generated]
$ java "com.hp.wsi.Generator" -i AcmsApp_idl.xml -a AcmsApp -o [.generated]
File: ./generated/acmsapp_server/build-acmsapp-server.com generated.
File: ./generated/acmsapp_server/methIds.h generated.
File: ./generated/acmsapp_server/structkeys.h generated.
File: ./generated/acmsapp_server/acmsapp.wsi generated.
File: ./generated/acmsapp_server/acmsapp.opt generated.
File: ./generated/acmsapp_server/acmsapp-server.h generated.
File: ./generated/acmsapp_server/acmsapp-server.c generated.
File: ./generated/acmsapp_server/build-acmsapp-jb.com generated.
File: ./generated/acmsapp/Iacmsapp.java generated.
File: ./generated/acmsapp/acmsappImpl.java generated.
File: ./generated/acmsapp/CONTROL_WORKSPACE.java generated.
File: ./generated/acmsapp/EMPLOYEE_INFO_WKSP.java generated.
*** Application acmsapp generated! ***
$
```

### Build the server

The server build procedure links the generated server files with the user's application, which creates a dynamically loadable sharable image.

```
$ set def [.generated]

$ @BUILD-ACMSAPP-SERVER
Begin server build procedure.
  ..configuring switches and compiler options
  ..compiling native server code
  ..linking shareable image
  ..installing server image
End server build procedure.
$
```

### Build the client

The JavaBean build procedure creates a JAR file that contains the WSIT Java classes used to call the server created earlier.

```
$ @BUILD-ACMSAPP-JB
Begin Java bean build procedure.
The New JAVA$CLASSPATH is:
  "JAVA$CLASSPATH" = "WSI$ROOT:[LIB]WSIRTL.JAR" (LNM$JOB_86F82E00)
    = "[]"
    = "WSI$ROOT:[LIB]VELOCITY-DEP-1_4.JAR"
    = "WSI$ROOT:[TOOLS]IDL2CODE.JAR"
  ..Compiling structure classes
  ..Compiling acmsapp Interface classes
  ..Creating acmsapp.JAR file from classes
End of JavaBean build procedure.
```

## **Step 5: Run the ACMS Sample**

Add the `AcmsApp.jar` file to the `java$classpath`.

```
$ @WSI$ROOT:[tools]wsi-setenv - acmsapp.jar
```

The new `JAVA$CLASSPATH` is:

```
"JAVA$CLASSPATH" = "WSI$ROOT:[LIB]WSIRTL.JAR" (LNM$JOB_86F82E00)
= "[]"
= "WSI$ROOT:[LIB]VELOCITY-DEP-1_4.JAR"
= "WSI$ROOT:[TOOLS]IDL2CODE.JAR"
= "ACMSAPP.JAR"
```

```
$
```

In the normal course of development, you would now need to write a client to call the `AcmsApp` JavaBean that was generated above. However, for the purpose of this sample, a client file named `acmscaller.java` is provided in the directory `WSI$ROOT:[samples.acms]`.

### **Compile the supplied client**

```
$ set def WSI$ROOT:[samples.acms]
$ javac acmscaller.java
```

### **Run the supplied client**

```
$ java acmscaller
***** Creating JavaBean & Server *****
***** Calling AcmsSignIn *****
***** Calling add and get empl_info tasks *****
***** Calling AcmsSignOut *****
123456
John Adams
1 Beacon Hill
Boston
MA
01776
***** Removing the JavaBean & Server *****
***** End ACMS Client *****
$
```

The output from the Java program shows the code that the client is executing, as well as the calls that it is making into ACMS. The data displayed was first entered into an Rdb database via ACMS, then retrieved using ACMS for display purposes.

## DEPLOYMENT CONSIDERATIONS

---

### 2.1 Types of OpenVMS Applications

Applications running on OpenVMS systems can be roughly divided into two groups, as follows:

- Applications designed for a single client environment
- Applications that can be called by multiple clients

The first group, **applications designed for a single client environment**, are often older OpenVMS applications that assume a timesharing runtime environment. The user logs into the OpenVMS system, which in turn creates a process. The applications are typically executed entirely in the user's process. In this design, there is a single user (the client). There is an assumed one-to-one relationship between the client and the application.

The second group, **applications that can be called by multiple clients**, are often newer OpenVMS applications. These applications are designed to *serially* process multiple clients (one at a time), or to *concurrently* process multiple clients (all at the same time).

When the Web Services Integration Toolkit exposes an OpenVMS application as a JavaBean, the application becomes callable from the second (newer) design model in which multiple clients can call the application from multiple processes or threads. You should understand in which group your wrapped application belongs (the specific design model) and manage client access to the application accordingly. WSIT provides a number of features to help in managing this interaction, which are discussed in the following sections.

In the following sections, the term **application** is used to represent the original application being exposed. The term **client** is used to represent the JavaBean caller which makes calls to the WSIT-generated JavaBean.

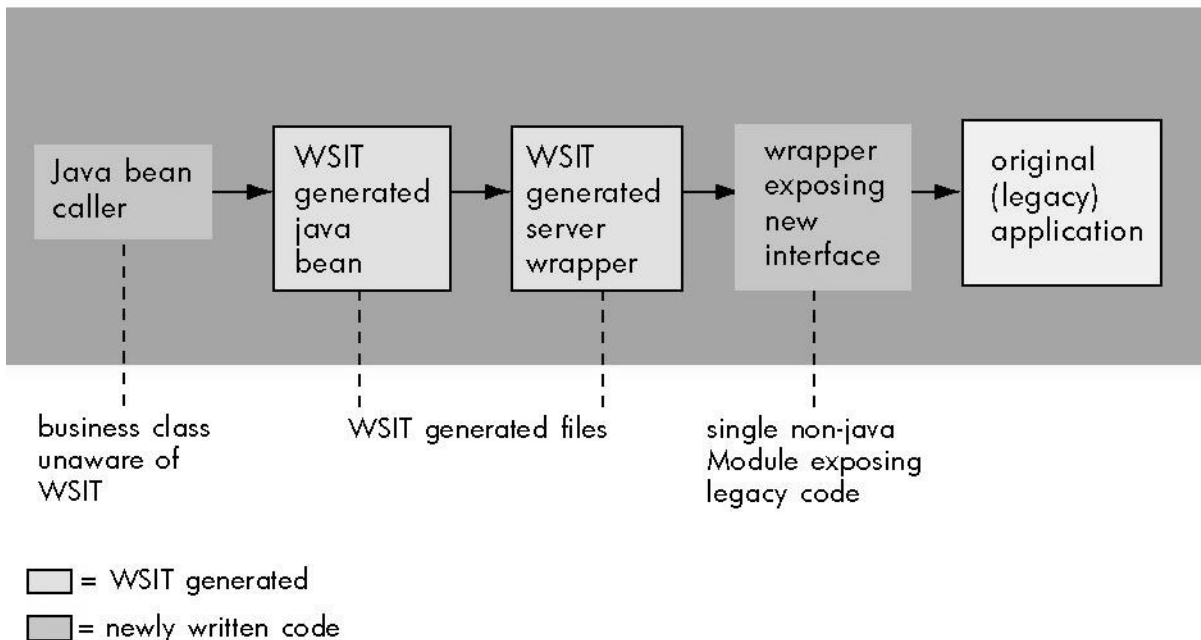
### 2.2 In-Process Deployment

There are two ways in which you can deploy your application using WSIT: **in-process** deployment and **out-of-process** deployment.

**In-process deployment** occurs when the **application and the client are called from the same process**, as illustrated in the following diagram.



Process A



There are advantages and disadvantages to using in-process deployment.

Pros: Fastest return time for client calls to application. No overhead added by the WSIT runtime.

Cons: A crash will bring down all components in the process (client and application).

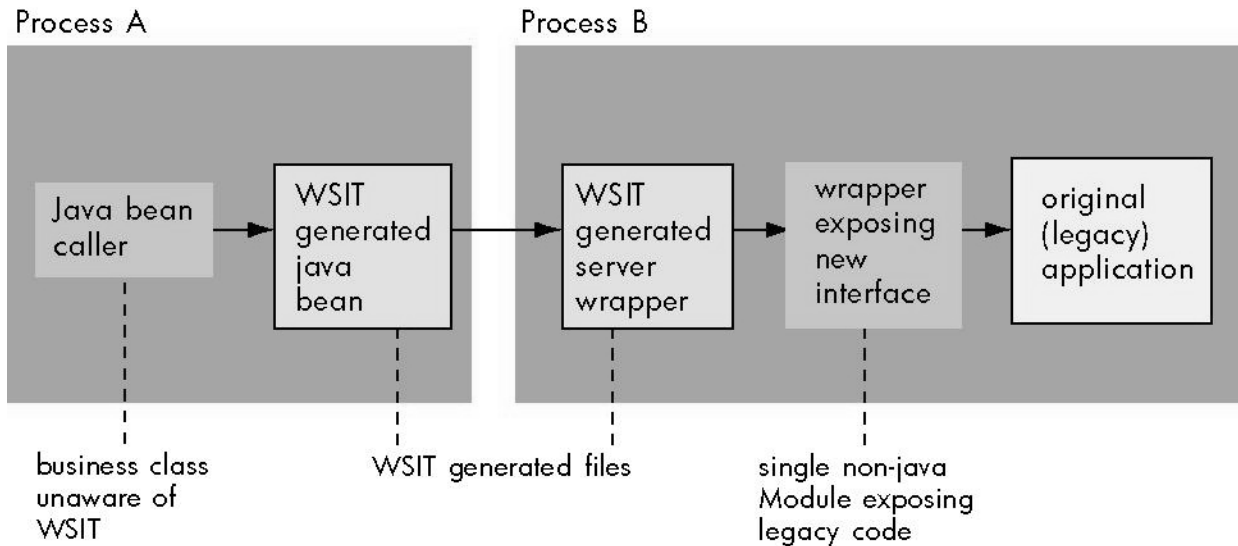
There are no WSIT deployment settings for in-process applications -- the interaction between the client and the application is not managed by the WSIT runtime. In-process deployment provides the fastest execution time, but it requires that the developer ensure that the client does not establish an environment in which the application will fail.

For example, some J2EE application servers may use multiple threads to call the client. This requires that the developer determine if the application can successfully operate in this environment. If the developer determines that the application can only support one client at a time, then the client must use a mechanism to *order the calls* before they are sent to the application (via the WSIT-generated objects).

If you do not specify out-of-process deployment settings (described in the following sections), **your application will run in-process by default.**

## 2.3 Out-of-Process Deployment

**Out-of-process deployment** occurs when the **client and application are run in different processes**, as illustrated in the following diagram. The WSIT runtime environment manages the interaction between the two processes. You can customize this environment by modifying a deployment descriptor file.



□ = WSIT generated  
 □ = newly written code

There are advantages and disadvantages to using out-of-process deployment.

**Pros:** Typically scales better than in-process deployments. Allows the use of the WSIT runtime deployment properties.

**Cons:** Adds complexity and overhead to every call.

**Most older applications benefit from using an out-of-process deployment** to avoid complex issues that result from mixing older and newer environments. The WSIT deployment properties, described in the following sections, allow out-of-process applications to choose from a wide variety of configurations.

### 2.3.1 Sessions

A session is the period of time in which a client uses an application. A session can last for:

- The duration of a single call
- The lifetime of the client

The type of session you use should mimic the original design of the application. For example, in older timesharing applications, a session is often the entire time that the client uses the application. In newer applications, the client may use a session to perform a specific task and then declare that it is finished with the session.

WSIT allows the developer to specify when a session with an application begins and when it ends. The WSIT-generated JavaBean has a constructor named `<application-name>Impl`. For example, the stock sample has a constructor named `stockImpl`. To establish an out-of-process deployment, call the constructor with an instance of the class `WsiIpcContext`. The `WsiIpcContext` constructor can be called with one of three different session types.

1. **LIFETIME\_SESSION:** This is the default session type. The session begins when the applications Impl object is created and the session ends when the remove method is called.

2. **NO\_SESSION:** The session begins when a method call is made on the application and the session ends when that call returns. The lifetime of the session is a single method call.
3. **TX\_SESSION:** The session begins when the client logs into the application by calling the methods `AcmsSignIn` or `OpenVMSLogin` of the application `Impl` object. The session ends when the client calls the methods `AcmsSignOut` or `OpenVMSLogOut`.

### 2.3.2 Application Reusability

The default configuration for all WSIT out-of-process applications is **not reusable**.

An application is **not reusable** when it can only be used for one client session. When the session is finished, the application has created state that prevents it from being called again. The next client session requires a new instance of the application.

An application can also be sequentially reusable, concurrently reusable, or concurrently reusable with multiple threads. See Chapter 3 for more information about these types of applications.

### 2.3.3 Using Multiple Processes to Scale Applications

When deploying an application out-of-process, WSIT allows the creation of a **process pool**, which is a collection of processes for the application that WSIT manages in the background to improve response time. Each process is running the application. The XML tag `<ProcessPooling>` is used to configure the properties of the pool.

- Use the tag `<MaximumProcesses>` to specify an upper limit for the largest number of processes that WSIT can create for the application.
- Use the tag `<MinimumProcesses>` to specify a lower limit for the fewest number of processes that WSIT should maintain for the application. The number specified will be the number of processes WSIT starts initially.
- Use the tag `<MinimumIdleProcesses>` to specify the number of non-busy processes to keep on an ongoing basis. WSIT creates more processes as needed to maintain these free processes. WSIT does not create more than `<MaximumProcesses>` of processes.
- Use the tag `<MaxInactivitySeconds>` to specify when a process should be removed from the pool and run down. Specify the maximum number of seconds that an application can be idle before it is automatically stopped.

### 2.3.4 Specifying Out-of-Process Deployment Options

Running your application out-of-process allows you to specify configuration options. These options are contained in the XML file `wsis$root:[deploy]application-name.wsi`.

The **out-of-process configuration options** are as follows:

Server Application Options	Description
Account (Username)	Name of the account you want the server to run in, which determines the access rights and quotas that the server will have. Requires NETMBX and TMPMBX privileges.
WorkingDirectory	Working directory for the server component. This is important if the server component opens files with names relative to some assumed working directory of the application.
SetupCommandFile	File specification of a DCL command file you want to run before the server component starts up.
ServerPath	Complete file specification where the server component image is located.
StackSize	Stack size to use for each thread within the server component. The default of 0 means to use the default DECthreads (POSIX Threads) stack size.
Reusable	Default is not reusable. Uncomment this option if the server application is reusable, which means the server can be called by more than one client sequentially. (See Advanced Out-of-Process Configuration chapter.)
MaximumClients	Maximum number of clients handled per server. If the server is not reusable, the default is 1. Modify <code>MaximumClients</code> if, in addition to being reusable, the server application process can handle multiple clients concurrently. If this property is greater than 1, the order of client calls coming into the server process is indeterminate. (See Advanced Out-of-Process Configuration chapter.)
MaximumThreads	Maximum number of threads allowed to run concurrently. This option is never greater than <code>MaximumClients</code> . If the server is not reusable, the default is 1. Modify <code>MaximumThreads</code> if, in addition to being able to handle multiple clients, the server application is also thread safe. (See Advanced Out-of-Process Configuration chapter.)

Server Process Options	Description
MaximumProcesses	Defines the maximum number of server processes that are allowed to run concurrently to handle client requests. The total capacity of the application is <code>MaximumProcesses</code> multiplied by <code>MaximumClients</code> . The default is 5.
MinimumProcesses	Minimum number of server processes that are automatically started to service requests from clients. This value is never greater than <code>MaximumProcesses</code> . The default is 0.
MinimumIdleProcesses	Sets the number of server processes that will be maintained as idle to serve requests from clients. As servers become busy, new server processes are started to act as idle servers. The number of idle server processes can reach (but never be greater than) the number of <code>MaximumProcesses</code> .
MaxInactivitySeconds	Maximum number of seconds that a server can be idle before it is automatically stopped. The default is 1000 seconds.
MaxStartupSeconds	Maximum number of seconds to wait for a process to startup.

	Default is 45 seconds.
ClientsWaitForServer	Specifies whether a client request should wait for a server process to become available. If set to 0 (the default), the client request fails with an error if a server is unavailable. If set to 1, the client waits for an available server. Waiting may appear to be a hung client if no server processes become available.

## ADVANCED OUT-OF-PROCESS CONFIGURATION

This chapter is intended for experienced Web Services Integration Toolkit users.

Before you configure the out-of-process deployment file, **identify your application's level of reusability**. If your application is reusable, you can significantly reduce the number of processes needed to service the clients.

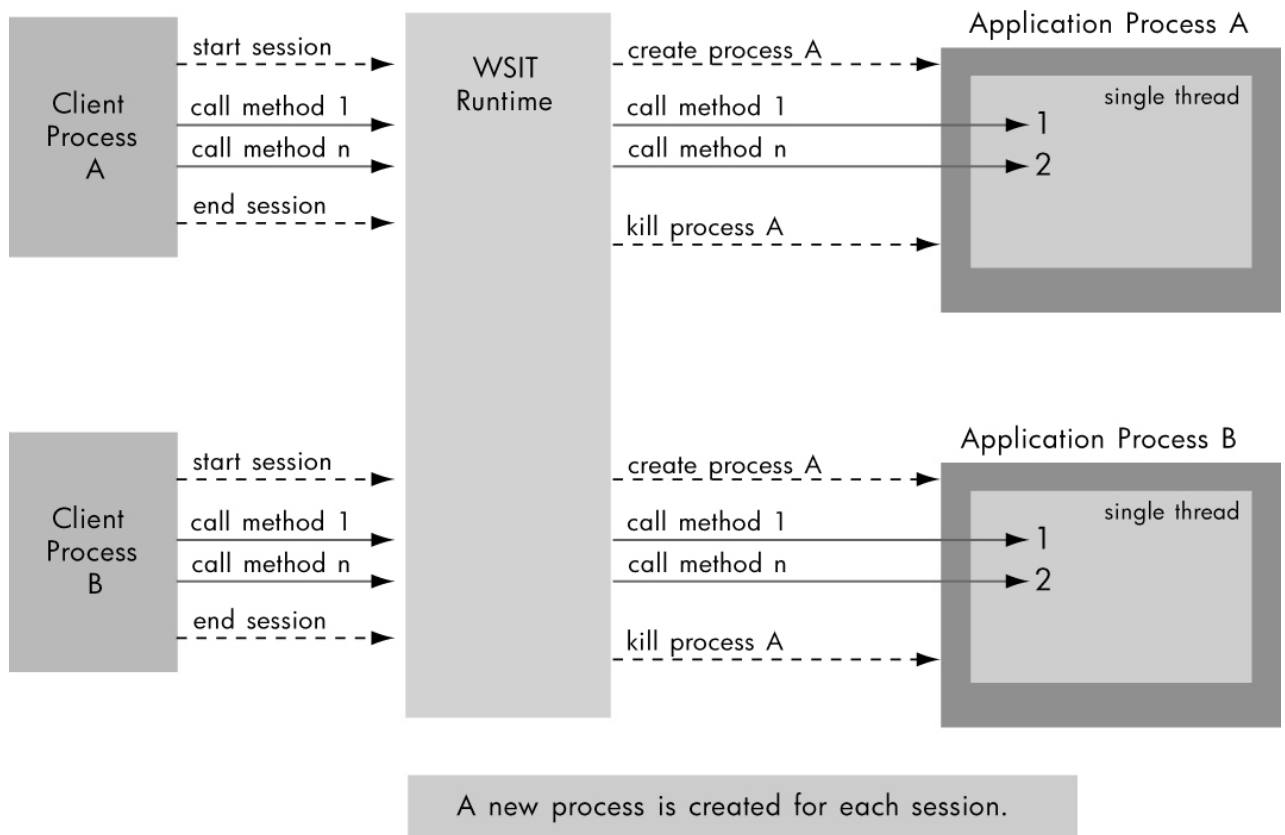
Next, to determine if a single instance of an application can handle multiple clients, **consider how you maintain state and manage I/O** within your application. Applications are frequently designed to accumulate state from call to call. For example, the first call opens a file, the second call reads from the file, and the third call updates and writes back the modified record. In cases like this, multiple clients within the same application may “step on” each other trying to access the same files using the same channels.

To help you determine your application's level of reusability, the following sections describe four applications, each with a different level of reusability or combination of reusability and multiple threads.

### 3.1 CASE A: NOT REUSABLE

This is the **default configuration** for all WSIT out-of-process applications.

An application is **not reusable** when it can only be used for **one client session**. When the session is finished, the application has created state that prevents it from being called again. The next client session requires a new instance of the application. This situation can exist in older applications that assume a single long-lived client is their only client. To identify this situation look for global (or static) variables that are used to identify stored client-specific data. The following figure shows an application that is **not reusable**:



For example, an application may not be reusable if it has a global variable to hold a client account number and the account number cannot be modified or reset with a subsequent call or other mechanism. If a second session requires a different account number to be set, then the application is not reusable.

Deploying an application as not reusable is the most restrictive case and has the highest runtime overhead, but is also the safest configuration. When an application is not reusable, WSIT ensures that the client always receive a new instance of the application.

The order of events for a non-reusable application is as follows:

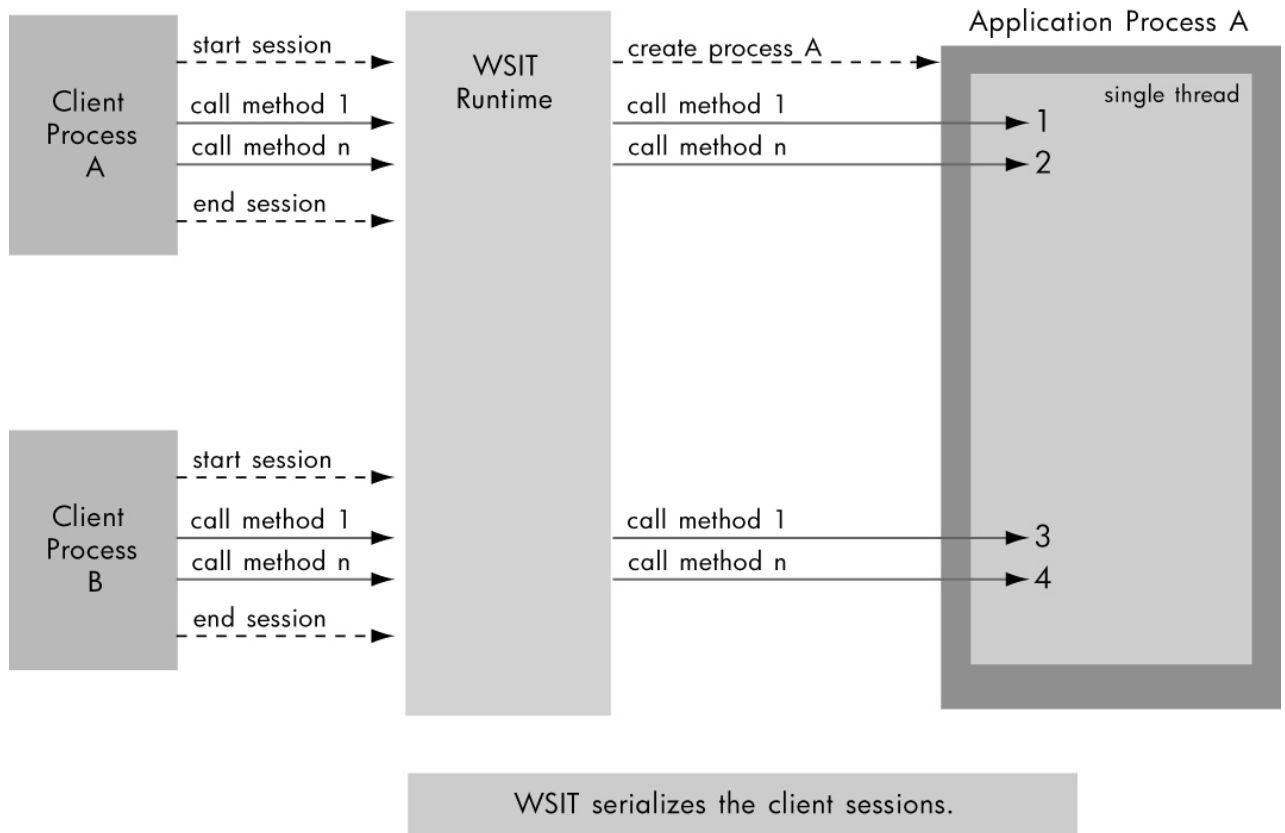
1. The client instantiates the WSIT-generated JavaBean. This starts a session with a free application.
2. WSIT assigns the client the exclusive use of a process that is running the application. The process may be newly created or may have been previously created but never used.
3. The client uses the application as it desires. One or more calls are made as part of the session.
4. The client tells the WSIT runtime that it is finished with the session by calling the remove method of the WSIT-generated JavaBean. This assumes that a session type of `LIFETIME_SESSION` is being used. A non-reusable application should not use a `NO_SESSION` session because of the extremely high overhead which occurs from creating and deleting a process for every method call.
5. WSIT deletes the process.

### 3.2 **CASE B: SEQUENTIALLY REUSABLE**

An application is **sequentially reusable** when it is able to process more than one client session, but requires that **exactly one session be active at a time**. The application must initialize its state before processing the next client session.

When an application is sequentially reusable, WSIT will not delete the application process when the client is finished using it. WSIT ensures that only one client can have a session outstanding with the application.

The following figure shows an application that is sequentially reusable.



For example, an application is serially reusable if it has a global variable to hold a client account number and also a method to initialize (or modify) the account number. In this way, each client can call the initialize method to erase the state of a previous client's session.

The order of events for a sequentially reusable application is as follows:

1. The client instantiates the WSIT generated JavaBean. This starts a session with a free application.
2. WSIT assigns the client the exclusive use of a process that is running the application. The process may be newly created or may have been previously created.
3. The client uses the application as it desires. One or more calls are made as part of the session.
4. The client tells the WSIT runtime it is finished with the session based on the type of session used.
5. WSIT places the application in a pool so that it is available for another client to use.

To deploy an application as sequentially reusable, make the following change to the file `wsit$root:[deploy]<application-name>.wsi`:

- Uncomment the XML tags `<Reusable>` `</Reusable>`

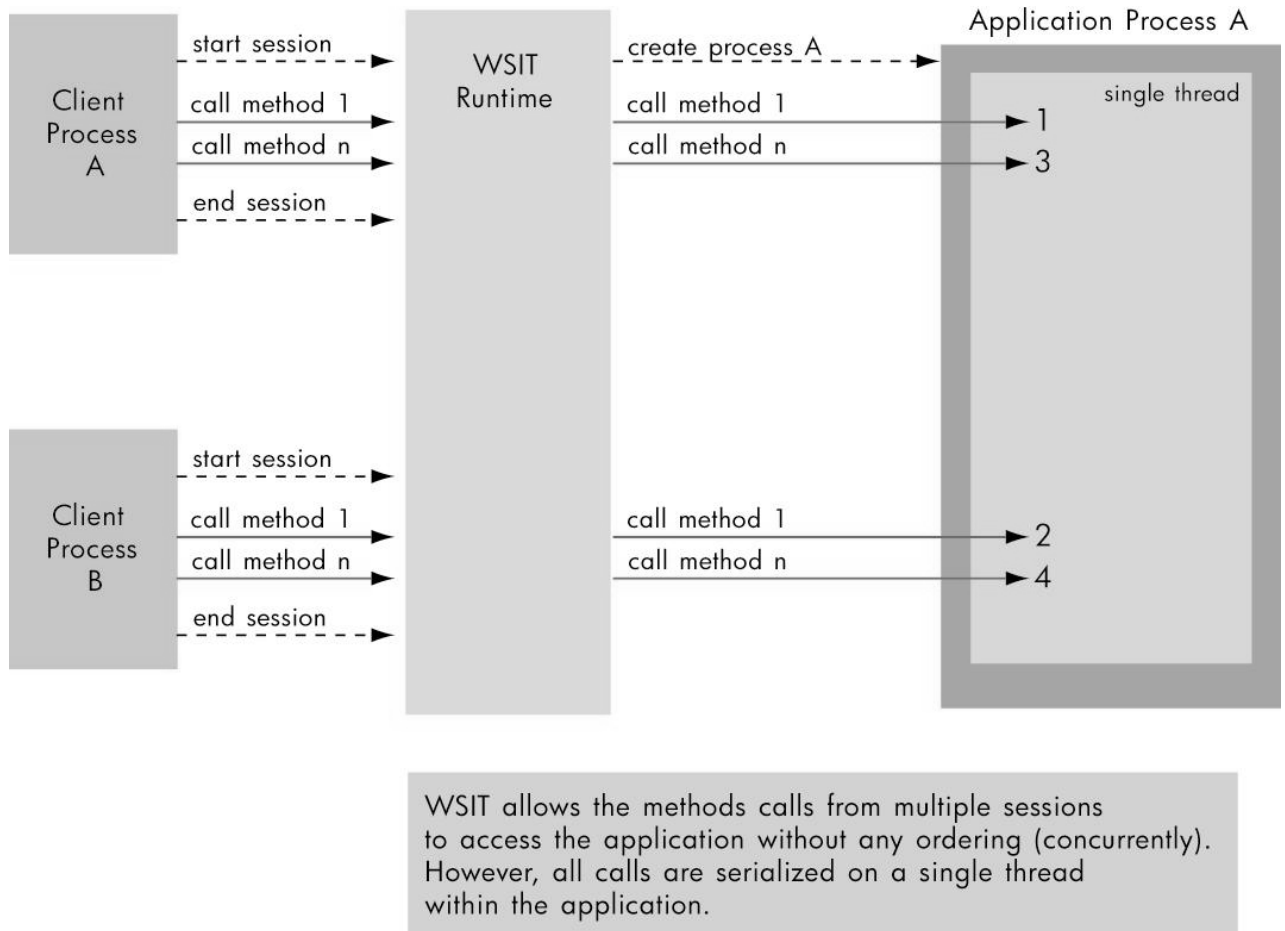
### 3.3 **CASE C: CONCURRENTLY REUSABLE**

An application is **concurrently reusable** when it can be called from **multiple clients without regard to the order of the clients' sessions**. This type of application has a mechanism for keeping the state of each of the clients separated. WSIT uses a single thread when forwarding the clients' calls to the application. From



the application's perspective, the clients' *sessions* may be nested. This ensures that the application processes one client call at a time.

The following figure shows an application that is concurrently reusable.



For example, an application is concurrently reusable if its interface uses a context block to hold client data. In this way, the logic in the application is generic in regard to the clients.

The order of events for a **concurrently reusable** application is as follows:

1. The client instantiates the WSIT-generated JavaBean. This starts a session with a free application.
2. WSIT assigns the client the use of a process that is running the application. The process may be newly created or may have been previously created. Other instances of the client may also be using the same application, but WSIT ensures that all method calls are made one at a time on the same thread.
3. The client uses the application as it desires. One or more calls are made as part of the session.
4. The client tells the WSIT runtime it is finished with the session based on the type of session used.
5. WSIT places the application in a pool so that it is available for another client to use.

To deploy an application as sequentially reusable make the following changes to the file `wsi$root:[deploy]<application-name>.wsi`:

1. Uncomment the XML tags `<Reusable>` `</Reusable>`
2. Uncomment the XML tag `<MaximumClients>` and specify a number **greater than one** for the number of client sessions that the application can process

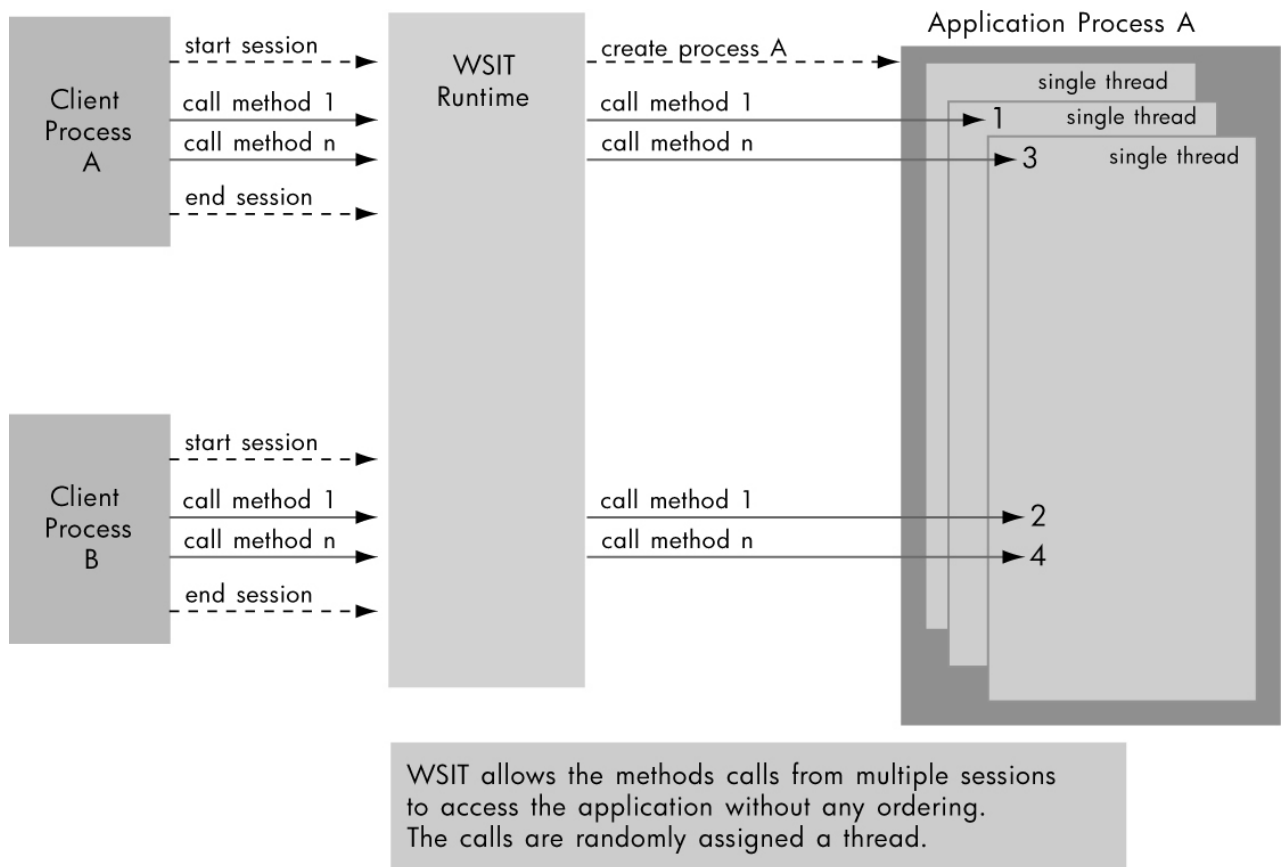
### 3.4 **CASE D: CONCURRENTLY REUSABLE WITH MULTIPLE THREADS**

An application is **concurrently reusable and thread-safe** when it can be **called from multiple clients, at the same time, on multiple threads**. WSIT allows multiple client sessions to call the application without any attempt to serialize them. The application was designed to lock shared data when called by multiple threads. It is also written in a language that is capable of generating thread-safe code. (For example, COBOL and BASIC do not generate thread-safe code.)

To determine if the application can handle calls from different clients concurrently, consider the following:

- Are all of the application resources that are shared among clients protected in a thread-safe way? For example, using global symbols can cause problems unless they are protected by a mutex or equivalent concept.
- Is the language that was used to write the application thread-safe?

The following figure shows an application that is concurrently reusable with multiple threads.



For example, an application is concurrently reusable and thread safe if its interface uses a context block to hold client data, and all access to global data, such as a global queue of client context blocks, is protected by a thread-safe locking mechanism such as a mutex.

The order of events for a **concurrently reusable with multiple threads** application is as follows:

1. The client instantiates the WSIT-generated JavaBean. This starts a session with a free application.
2. WSIT assigns the client the use of a process that is running the application. The process may be newly created or may have been previously created. Other instances of the client may also be using the same application. WSIT uses multiple threads to call the application.
3. The client uses the application as it desires. One or more calls are made as part of the session.
4. The client tells the WSIT runtime it is finished with the session based on the type of session used.
5. WSIT places the application in a pool so that it is available for another client to use.

To deploy an application as sequentially reusable make the following changes to the file

`wsiroot:[deploy]<application-name>.wsi:`

- Uncomment the XML tags `<Reusable>` `</Reusable>`
- Uncomment the XML tags `<MaximumClients>` `</MaximumClients>` and specify a number **greater than one** for the number of client sessions that the application can process.
- Uncomment the XML tag `<MaximumThreads>` `</MaximumThreads>` and specify a number **greater than one** for the number of threads that WSIT can use to call the application.

The number of threads allowed to run concurrently should be a percentage of the number of clients specified. A good rule of thumb is to look at the average amount of time that each application call is expected to take.

- If the calls are small and quick, then the number of threads allowed to run concurrently could be 25% of the number of clients.
- If you expect calls to take longer, then you should use a larger value, such as 50 to 60%.

For example, if you specify ten clients per application, and each application call will take some time, then allow six threads to run. Note that once a system is in place, this number should be monitored and adjusted as needed.

## USING TEMPLATES TO GENERATE CODE

---

**This chapter is intended for experienced Web Services Integration Toolkit users.**

The Web Services Integration Toolkit uses **Velocity templates**, which encapsulate language syntax to specify the code that will be generated based on the IDL. Velocity is an open source Java-based template engine provided by the **Apache Jakarta** project. The Velocity Template Language (VTL) is the scripting language used in the Velocity engine.

The Web Services Integration Toolkit provides an optional extensibility feature – the ability to modify or replace the Velocity templates that WSIT uses to generate code. (This feature is described in the following sections.) You can change the template to generate different source code. You can also change the template, for example, to improve performance or to add security to your specific application.

For more information about Velocity, see <http://jakarta.apache.org/velocity/>.

For more information about VTL, see <http://jakarta.apache.org/velocity/user-guide.html> and <http://jakarta.apache.org/velocity/vtl-reference-guide.html>.

### 4.1 Modifying Velocity Templates

The Web Services Integration Toolkit (WSIT) provides a number of different tools that complement each other in the process of wrapping existing OpenVMS applications. One of the tools in the toolkit is **idl2code.jar**, also known as the WSIT **generator**.

The WSIT generator uses a mechanism based on Velocity templates. A set of template files are read, and placeholders within those templates are replaced by application-specific values. This becomes the generated code.

More specifically, the generator reads a WSIT-specific IDL description of the application to be wrapped, then generates code based on this description. The generated wrapping code contains the following:

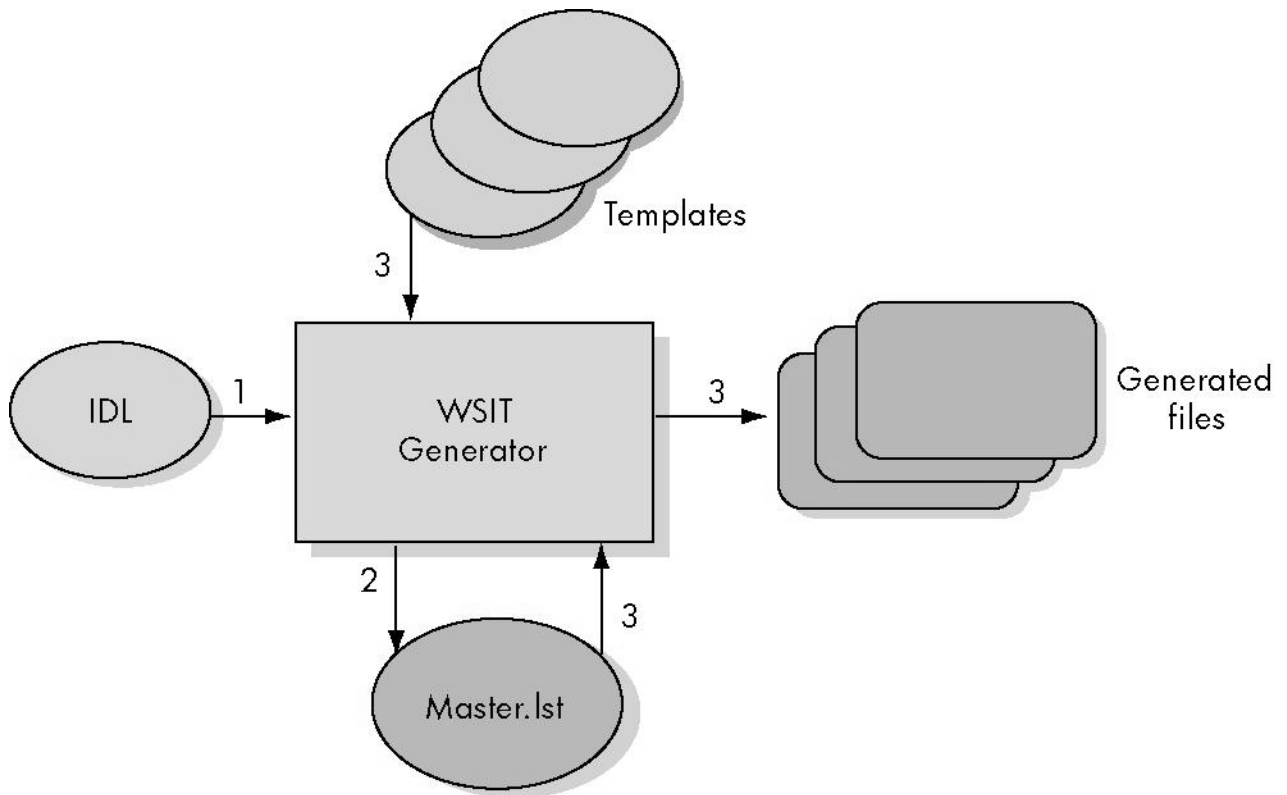
- Server wrapper component that builds against the existing application
- JavaBean component that provides the new interface into this application

**The *standard* code that is generated *out of the box* has been tested and is robust enough to handle most cases.** However, you may want to tailor what is generated by `idl2code.jar`. The following sections describe the process that `idl2code.jar` uses to generate code, and then describes how you can modify what is generated.

**Note:** The Velocity templates are contained in the subdirectory `WSI$ROOT:[TOOLS.TEMPLATES]`. You can modify the current set of Velocity template files, or you can add new template files.

### 4.2 Generating Code with `idl2code.jar`

The generation process occurs in four distinct phases. The first three phases offer you an opportunity to modify what ultimately is generated. The phases are described in the following sections.



### **Phase 1: Parse IDL File**

In Phase 1, the generator reads the WSIT IDL file describing the application, parses it, and then populates an object model representation of its contents. This object model is directly accessed by the templates and template engine in later phases. The IDL should accurately reflect the interface to the application being wrapped. This phase is your first opportunity to affect what is generated.

See the Exposing an OpenVMS 3GL Application section for more information.

### **Phase 2: Generate File List**

In Phase 2, the generator generates a master list called `Master.lst` of files to generate based on the template file called `Master.vm`. (In the next phase, the generator steps through this list of files to generate the actual files.) Modifying `Master.vm` allows you to change the list of files to generate, as well as to change which templates to use in generating these files.

Each line within `Master.vm` and `Master.lst` has the following format:

```
<Object> <Name> <Output Filename> <Template Filename>
```

where:

**Object** describes the component to which this generated file will belong. **Object** can have one of four values:

```
SW      File belongs to the server wrapper component
        (The generated file is placed into the subdirectory [.appNameServer])
```

- JB     File belongs to the **JavaBean** component  
(The generated file is placed into the subdirectory [ . appname ])
- I     File represents an **interface** within the JavaBean  
(The generated file is placed into the subdirectory [ . appname ])
- S     File represents a **structure** definition within the Javabeen  
(The generated file is placed into the subdirectory [ . appname ])

**Name** identifies the name of the object within the object model that this file represents.

**Output filename** is the filename of the file to be generated.

**Template filename** is the filename of the template to use in creating this file.

### **Phase 3: Generate Files**

In Phase 3, the generator reads the previously generated `Master.lst` file, then generates the listed files.

Each line within `Master.lst` contains an output filename and an associated template file to use in its generation. It uses these mappings, along with this previously populated object model, to create the output files.

You can modify the template files to change the contents of the individually generated files. The object model can be accessed using Velocity identifiers.

### **Phase 4: Generate Javadocs (Optional)**

In Phase 4, the generator optionally runs the Javadoc utility against the generated JavaBean files, creating a set of .html files that document the generated interface.

## **4.3 Example 1: Writing a New Template**

The following example shows you how to write a new template and add it to the master list so that it will be used within the generation process.

This example assumes the following simple WSIT IDL:

```

<?xml version="1.0" encoding="UTF-8"?>
<OpenVMSInterface
  xmlns="hp/openvms/integration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="hp/openvms/integration openvms-
integration.xsd"
  ModuleName="simple.OBJ"
  Language="C89">
  <Primitives>
    <Primitive Name = "signed int"
      Size = "4"
      VMSDataType = "DSC$K_DTYPE_L"/>
  </Primitives>
  <Routines>
    <Routine Name = "add"
      ReturnType = "signed int">
      <Parameter Name = "p1"
        Type = "signed int"
        PassingMechanism = "Value"
        Usage = "IN"/>
      <Parameter Name = "p2"
        Type = "signed int"
        PassingMechanism = "Value"
        Usage = "IN"/>
      </Routine>
    </Routines>
  </OpenVMSInterface>

```

### Step 1: Write Template using Velocity

For this example, we will write a template file using Velocity that generates a log file containing the list of the routines being exposed, along with their parameters. (Save the file as `app-logfile.vm`.) The completed file will look similar to the following:

```

#set( $server = $application.Server )

          Logfile for ${application.Name}
          -----

Routines being exposed                      Parameters
-----
#foreach( $routine in $server.Routines)
  ${routine.Name}
    Description: ${routine.Description}
    Return Type: #if( $routine.Returnparam )
    $routine.Returnparam.SWFormalDefinition#else void#end
#foreach( $param in $routine.Parameters)
                                ${param.Name}
                                ($param.SWFormalDefinition)
#end
#end

```

Note the Velocity syntax that is used to pull information from the provided object model. Within WSIT:

- All templates are passed `$application`, which is a reference to the top level `genConnection` object.
- All structure type templates are passed `$structure`, which is a reference to the `genStructure` object.
- All interface templates are passed `$interface`, which is a reference to the `genInterface` object.

The template files located in `WSI$ROOT:[TOOLS.TEMPLATES]` show many different examples of using these tags.

When you finish writing the template, place it where Velocity can find it. The path(s) that the Velocity engine uses to find template files is specified in the file `velocity.properties`. You can modify this file to add your directory to the search path, or you can copy your newly written template into a directory that is already in the path.

### **Step 2: Modify Master.vm**

Next, modify `Master.vm` to add the new file to the list of files that are generated. The new file that you created in Step 1 is considered a **server wrapper** template file because it lists the interface as exposed by the application.

Add the following line to `Master.vm` in the appropriate place. Type the line exactly as shown below. (Like any other template file, the `${application.Name}` placeholder is automatically replaced with the name of the application, causing the generated file to contain the application name in its filename.)

```
SW ${application.Name} ${application.Name}.log app-logfile.vm
```

### **Step 3: Run idl2code.jar**

Run the generator and review the newly generated files. If the run is successful, a log file that looks similar to the following can be found in the `[.ServerSimple]` subdirectory.

```

                                Logfile for Simple
                                -----

Routines being exposed          Parameters
-----
add
  Description: This is the description for the add routine
  Return Type: signed int
                                P1 (signed int)
                                P2 (signed int)
```

## **4.4 Example 2: Modifying an Existing Template**

Example 1 showed you how to write and add your own new template. However, there may be cases in which you want to directly **modify the behavior of an existing WSIT template**.



For instance, you may want to remove the restriction that every client needs to have its own instantiation of the JavaBean interface class. The way to remove this restriction is to make sure that the generated JavaBean class has appropriate synchronization code to serialize all calls through it. (This is not generated by the default WSIT templates.) This example steps you through this simple process.

### **Step 1: Set Default Directory**

Assuming the default location for the templates, set your default directory to  
`WSI$ROOT:[tools.templates.javabean].`

This is where you will find the templates used to generate the javabean files.

### **Step 2: Edit Files to Add Synchronize Keyword**

Assuming that the templates have their default names, edit the files  
`INTERFACE-JAVA.VM` and `INTERFACEIMPL-JAVA.VM`.

These files define the interface, and the implementation of the interface, for the newly generated application.

For every public method definition within the two files, you must add the **synchronized** keyword. In particular, add the **synchronized** keyword to the: `AcmsSignIn`, `AcmsSignOut`, `OpenVmsLogin`, `OpenVmsLogout`, and remove methods in each file.

Next, add the `synchronized` keyword to each method definition within `$interface.Methods` and `$interface.AcmsMethods`.

For example:

#### **Before:**

```
public#if($routine.Returnparam)
$dtutility.getJBtype($routine.Returnparam.Datatype)#else void#end
${routine.WebServiceName} ($paramformal)
    throws WsiException;
```

#### **After:**

```
Public synchronizedif($routine.Returnparam)
$dtutility.getJBtype($routine.Returnparam.Datatype)#else void#end
${routine.WebServiceName} ($paramformal)
    throws WsiException;
```

### **Step 3: Run idl2code.jar**

Run `idl2code.jar` to generate an application wrapper. The generated interface methods are synchronized. All method calls to each JavaBean is serialized.

## APPENDIX

---

This appendix contains program listings for the C sample program. Other sample programs can be found in WSI\$ROOT:[SAMPLES.ACMS], WSI\$ROOT:[SAMPLES.COBOLE], and WSI\$ROOT:[SAMPLES.FORTRAN].

### A Program Listing - STOCK.C

```
$ ty stock.c
//
// This is a sample file intended to be used to demonstrate the WSIT product.
// It defines 2 routines: (buy and sell). Each routine accepts 3 structures:
// buyerData, sellerData, tickerData.
//
// This code is intended only to illustrate the WSIT product and is not intended provide
// a usefull stock trading application.
//
//
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

const MAX_STRING = 20;

typedef struct _buyerData {
    char buyer_name[MAX_STRING];
    unsigned int member_number;
    unsigned int balance_dollars;
    unsigned int number_shares_desired;
} buyerData;

typedef struct _sellerData {
    char owner_name[MAX_STRING];
    unsigned int member_number;
    unsigned int balance_dollars;
    unsigned int number_shares_available;
} sellerData;

typedef struct _tickerData {
    char symbol[MAX_STRING];
    char company_name[MAX_STRING];
} tickerData;

unsigned int buy (unsigned int max_price, tickerData *symbol, sellerData *pSeller,
buyerData *pBuyer) {

    //
    // Sell as many shares as possible
    //
    unsigned int shares_purchased = 0;
    if ( pSeller->number_shares_available >= pBuyer->number_shares_desired) {
        shares_purchased = pBuyer->number_shares_desired;
    } else {
        shares_purchased = pSeller->number_shares_available;
    }

    pSeller->number_shares_available = pSeller->number_shares_available -
shares_purchased;

    return shares_purchased;
}
```

```
}  
  
unsigned int sell (unsigned int min_price, tickerData *symbol, sellerData *pSeller,  
buyerData *pBuyer) {  
  
    //  
    // Sell as many shares as possible  
    //  
    unsigned int shares_sold = 0;  
    if ( pSeller->number_shares_available >= pBuyer->number_shares_desired) {  
        shares_sold = pBuyer->number_shares_desired;  
    } else {  
        shares_sold = pSeller->number_shares_available;  
    }  
  
    pSeller->number_shares_available = pSeller->number_shares_available -  
shares_sold;  
  
    return shares_sold;  
}  
$
```

## B Program Listing - STOCK.XML

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenVMSInterface
  xmlns="hp/openvms/integration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="hp/openvms/integration openvms-integration.xsd"
  ModuleName="wsi$root:[samples.c]stock.obj"
  Language="C89">
  <Primitives>
    <Primitive Name = "unsigned int"
      Size = "4"
      VMSDataType = "DSC$K_DTYPE_LU"/>
    <Primitive Name = "AutoGen_FixedString19"
      Size = "19"
      VMSDataType = "DSC$K_DTYPE_T"
      NullTerminatedFlag = "0"
      FixedFlag = "1"/>
  </Primitives>
  <Typedefs>
    <Typedef Name = "tickerData"
      TargetName = "_tickerData"/>
    <Typedef Name = "sellerData"
      TargetName = "_sellerData"/>
    <Typedef Name = "buyerData"
      TargetName = "_buyerData"/>
  </Typedefs>
  <Structures>
    <Structure Name = "_tickerData"
      TotalPaddedSize = "40">
      <Field Name = "symbol"
        Type = "AutoGen_FixedString19"
        Offset = "0"/>
      <Field Name = "company_name"
        Type = "AutoGen_FixedString19"
        Offset = "20"/>
    </Structure>
    <Structure Name = "_sellerData"
      TotalPaddedSize = "32">
      <Field Name = "owner_name"
        Type = "AutoGen_FixedString19"
        Offset = "0"/>
      <Field Name = "member_number"
        Type = "unsigned int"
        Offset = "20"/>
      <Field Name = "balance_dollars"
        Type = "unsigned int"
        Offset = "24"/>
      <Field Name = "number_shares_available"
        Type = "unsigned int"
        Offset = "28"/>
    </Structure>
    <Structure Name = "_buyerData"
      TotalPaddedSize = "32">
      <Field Name = "buyer_name"
        Type = "AutoGen_FixedString19"
        Offset = "0"/>
      <Field Name = "member_number"
        Type = "unsigned int"
        Offset = "28"/>
    </Structure>
  </Structures>
</OpenVMSInterface>
```

```

        Offset = "20"/>
    <Field Name = "balance_dollars"
        Type = "unsigned int"
        Offset = "24"/>
    <Field Name = "number_shares_desired"
        Type = "unsigned int"
        Offset = "28"/>
</Structure>
</Structures>
<Routines>
    <Routine Name = "buy"
        ReturnType = "unsigned int">
        <Parameter Name = "max_price"
            Type = "unsigned int"
            PassingMechanism = "Value"
            Usage = "IN"/>
        <Parameter Name = "symbol"
            Type = "tickerData"
            PassingMechanism = "Reference"
            Usage = "IN/OUT"/>
        <Parameter Name = "pSeller"
            Type = "sellerData"
            PassingMechanism = "Reference"
            Usage = "IN/OUT"/>
        <Parameter Name = "pBuyer"
            Type = "buyerData"
            PassingMechanism = "Reference"
            Usage = "IN/OUT"/>
    </Routine>
    <Routine Name = "sell"
        ReturnType = "unsigned int">
        <Parameter Name = "min_price"
            Type = "unsigned int"
            PassingMechanism = "Value"
            Usage = "IN"/>
        <Parameter Name = "symbol"
            Type = "tickerData"
            PassingMechanism = "Reference"
            Usage = "IN/OUT"/>
        <Parameter Name = "pSeller"
            Type = "sellerData"
            PassingMechanism = "Reference"
            Usage = "IN/OUT"/>
        <Parameter Name = "pBuyer"
            Type = "buyerData"
            PassingMechanism = "Reference"
            Usage = "IN/OUT"/>
    </Routine>
</Routines>
</OpenVMSInterface>
$

```

---

## C Program Listing - StockCaller.Java

```
$ type stockcaller.java
import stock.*;
import java.io.*;
import javax.xml.rpc.holders.StringHolder;
import javax.xml.rpc.holders.StructureHolder;

public class stockcaller {

    /** Creates a new instance of Main */
    public stockcaller() {
    }

    public static void main(String[] args) {

        try {

            stockImpl stock = new stockImpl();

            // create a seller object and place in holder
            _sellerData sellerData = new _sellerData("Mr Seller", 12345, 1000000,
1000);

            StructureHolder seller = new StructureHolder(sellerData);

            // create a buyer object and place in holder
            _buyerData buyerData = new _buyerData("Mr Buyer", 67890, 5000, 995);
            StructureHolder buyer = new StructureHolder(buyerData);

            // create a ticker object and place in holder
            _tickerData tickerData = new _tickerData("HPQ", "Hewitt Packard");
            StructureHolder ticker = new StructureHolder(tickerData);

            System.out.println("The sellers number_shares_available: " +
sellerData.getNumber_shares_available());
            stock.buy(27, ticker, seller, buyer);
            System.out.println("The sellers number_shares_available: " +
sellerData.getNumber_shares_available());

        } catch (Exception e) {
            System.out.println("Exception thrown");
        }
    }
}
$
```