

# SWIGing for fun and profit

**Marc Hadley**

Chrystal Software, Slough, United Kingdom  
marc\_hadley@chrystal.co.uk  
<http://www.chrystal.com>

## **Abstract:**

*SWIG (simplified wrapper and interface generator) is a freely available tool for generating multiple language bindings for existing C/C++ code. This paper discusses the use of SWIG to generate language bindings for Python, Perl and Java and shows how to embed code written in these languages into existing C/C++ applications. Examples include using a Java XSLT processor from C++, using Python to process SAX (simple API for XML) events generated by C++ and enabling Perl scripting of a large C++ class library.*

## **Introduction**

In many applications there is a need for a high level programmatic interface to allow automation of repetitive tasks and some level of end user customisation or integration with other systems. Whilst it is often tempting to craft a custom scripting language for the job at hand, a better solution is almost always achieved by creating an interface to an existing feature-rich scripting language. At first sight this can appear daunting, but fortunately there are a number of tools available to ease the pain and one in particular, SWIG, makes it relatively straightforward.

This paper discusses the use of SWIG to generate language bindings to enable Python, Perl and Java to call native C/C++ code. It also describes how such bindings can be used to simplify the task of embedding one of these languages in an existing C/C++ application. Whilst not strictly a scripting language, Java is included here because of its popularity and ease of use compared to C/C++.

## **Using SWIG**

This section introduces SWIG and shows how to develop wrappers for a simple C++ class. This is not intended to be an exhaustive description of SWIG but rather to show some of its features and the general approach used to generate different language bindings. For a complete description of refer to the on-line documentation at <http://www.swig.org/>.

## **Interface files**

The input to SWIG is an interface file that describes the C/C++ functions and classes for which interfaces should be generated. Interface files are similar in syntax to C/C++ include files with some additional functionality provided using SWIG specific directives. A sample interface file is shown below.

```
%module test;

%{
#include "test.h";
%}

class Test
{
```

```

public:
    Test();
    ~Test();
    void PutNum(int num);
    int GetNum();
};

```

The first line declares the name of this module. This is used as the package name in the target scripting language, e.g. to use the module in Python one would type `import test` and in Perl one would type `use test;`. The lines between the `%{` and `%}` markers are copied verbatim into the generated wrapper code. Normally one inserts the headers required to use the classes/functions being wrapped in an external C/C++ module. Following this are the declarations of the classes and functions for which wrappers should be generated - in this case the single C++ class named `Test`.

## Running SWIG

SWIG is a command line tool that operates much like an IDL compiler. As input it takes an interface file and as output it generates some C/C++ glue code, some scripting language specific glue code if required and optionally some scripting language classes to emulate the functionality of the C++ classes being wrapped. Below is an example of generating Python wrappers for the interface file above (`test.i`).

```
swig -python -c++ -shadow -o test_wrap.cxx test.i
```

By default, SWIG treats all interfaces as C, the `-c++` switch turns on C++ support. The `-shadow` switch causes SWIG to emit Python classes to emulate the C++ classes and the `-o` switch causes SWIG to write the C/C++ glue code to `test_wrap.cxx`. To generate wrappers for Perl, the corresponding command is shown below.

```
swig -perl5 -c++ -shadow -o test_wrap.cxx test.i
```

Note that the only difference is the switch passed to SWIG. This allows the same interface file to be used to generate interfaces for any of the supported languages.

## Using the generated code

The first step is to build a shared library (or DLL on Windows platforms) from the generated C/C++ glue code and any implementation code not present in other dynamically loadable libraries. The exact command required to build a shared library varies between platforms but is usually straightforward. Naming of the shared library is important, the following table shows the recommended naming conventions for Unix and Windows platforms:

Python	<code>testc.pyd</code>	<code>testc.pyd</code>
Perl	<code>test.dll</code>	<code>test.so</code>
Java	<code>test.dll</code>	<code>libtest.so</code>

Table 1

The above conventions work on WinNT4.0 and Solaris 7 with Python 1.5.2, Perl 5.004 and Java 2, depending on the platform you choose, your mileage may vary.

Using the wrapped class from the target language is very simple, see the following examples.

## Python

```
from test import *
t=Test()
t.PutNum(5)
print t.GetNum()
```

## Perl

```
use test;
$t=Test::new();
$t->PutNum(5);
printf "%d", $t->GetNum()
```

## Java

```
import test.*;
public class TryIt
{
    static
    {
        try
        {
            System.loadLibrary("test");
        }
        catch (UnsatisfiedLinkError e)
        {
            System.err.println(e.toString());
            System.exit(-1);
        }
    }
    public static void main(String args[])
    {
        Test t = new Test();
        t.PutNum(5);
        System.out.println(t.GetNum());
    }
}
```

The `static` block can be placed in one of the generated Java files so that clients do not need to implicitly load the JNI library, but it is shown here for completeness. Perl and Python automatically load the dynamic library and hence do not require specific code to achieve this.

## Tips

### Modifying function signatures

Wrappers need not be generated for all members of the class, this is accomplished by omitting any desired members from the interface file. SWIG also allows generation of new methods not actually present in the C++ class, but appearing to be so from the scripting language interface. The following modified interface file shows an example of this.

```
%module test;

%{
#include "test.h";
```

```

%}

class Test
{
public:
    Test();
    ~Test();
    void PutNum(int num);
    int GetNum();
};

%addmethods Test
{
    int SwapNum(int newNum)
    {
        int oldNum = self->GetNum();
        self->PutNum(newNum);
        return oldNum;
    }
}

```

In the above case a new method `SwapNum` is added to the `Test` class that stores the number provided and returns the previously stored number. The above approach can be used to modify method signatures by omitting the original method from the interface file and providing an alternative in an `%addmethods` block.

### Exception handling

If your native C++ code throws exceptions to indicate errors then these will need to be converted into some kind of error in the scripting language. SWIG provides a very convenient method for doing this in the interface file. The following examples show how to catch C++ exceptions of type `XD_Exception` and process them in a manner suitable for Python, Java, and Perl.

```

%except(python)
{
    try
    {
        $function
    }
    catch (XD_Exception &err)
    {
        PyObject *retVal = Py_BuildValue("l", err.m_xdStatus);
        if (err.m_xdStatus==XD_ERR_INVALID_INDEX)
            PyErr_SetObject(PyExc_IndexError, retVal);
        else
            PyErr_SetObject(PyExc_RuntimeError, retVal);
        return NULL;
    }
}

%except(java)
{
    try
    {
        $function
    }
    catch (XD_Exception &err)

```

```

        {
            ThrowJavaException(err);
        }
    }

%except(perl5)
{
    try
    {
        $function
    }
    catch (XD_Exception &err)
    {
        croak(err.m_errMsg);
    }
}

```

Note that all three may be present in the interface file, SWIG will use the one corresponding to the current language when it is generating the wrapper code. Creating and throwing a Java exception from C/C++ is covered in the JNI documentation.

## Memory management

One of the more complex issues surrounding integration between C++ and a scripting language is that of memory management. Perl, Python and Java all support garbage collection whilst C++ requires the programmer to delete objects once they are no longer required.

It would be most convenient if C++ objects created by users of the scripting language wrapper could be automatically deleted as part of the garbage collection mechanism of that scripting language. This turns out to be default mechanism if the C++ objects are managed by script language shadow classes and C++ functions that create and return objects return such objects by value rather than as pointers. e.g. the following C++ function would generate wrapping code that will automatically delete the returned C++ object when the corresponding scripting language object is garbage collected.

```
Object GetObject();
```

whilst the following C++ function

```
Object *GetObject();
```

would by default generate wrapping code that would not delete the C++ object when the corresponding scripting language object is destroyed. In some cases, this behaviour is exactly what is required and it is certainly more efficient since it avoids object copying. However, for an easy life at the cost of raw efficiency it is recommended that the former method be adopted.

## Embedding

Embedding a scripting language in a native C/C++ application can have many benefits, particularly in the area of user automation and customisation. In addition it can allow native C/C++ applications access to libraries of useful code written in a scripting language. In the XML application space many of the emerging standards are first implemented in Java with a C/C++ port sometimes, but often not, following later. To access such code from a native C/C++ application requires either a port of the code to C++ or embedding a

JVM (Java Virtual Machine) in the C/C++ application and calling the Java code directly. Whilst porting is often straightforward, it can prove time consuming and tedious, particularly when the application makes use of a number of other libraries unavailable in the native language and for this reason it is this latter approach that is discussed here.

## Approach

The approach to embedding described here is only one of several alternatives, but it is relatively straightforward and should work in most cases. There are two interfaces to consider:

1. C/C++ to Scripting Language This interface is specific to the language being embedded. The simplest solution is to implement a scripting language shim for the library of interest with a standardised entry point that is as simple to call as possible from C/C++. The scripting language shim then gathers any required information before calling the library of interest in the normal way. SWIG cannot help to generate the code to call a scripting language routine from C/C++, but it can help by generating an interface through which the scripting language shim can gather it's required information by calling back into the C/C++ native code.
2. Scripting Language to C/C++ This interface provides a means for output from the scripting language code to be sent back to the native C/C++ application. In addition it allow the scripting language shim to query the C/C++ application for arguments and other information prior to calling the scripting language library.

The typical sequence of events is shown below:

1. C/C++ application initialises scripting language environment.
2. C/C++ application locates scripting language shim entry point.
3. C/C++ application initialises callback information object for shim.
4. C/C++ application calls shim entry point.
5. Shim code calls back to C/C++ code via passed-in information object to gather arguments.
6. Shim code arranges for output to be sent to C/C++ application via information object. This step may require some creativity. For the simplest case where the scripting language just returns a simple result then the shim code can write this result to the information object directly using the interface provided. For more complex requirements, the shim code may need to implement the expected output interface fro the scripting language code and route the output back to the C/C++ application using the interface provided
7. Shim code calls target script language function.
8. Script language function does it's work and sends results to C/C++ application via the information object.
9. Shim code returns.
10. C/C++ application processes results.

## A simple example

To illustrate, consider the following very simple example <sup>1</sup>: we have a routine coded in a scripting language that does something clever with two integers. The routine takes two integers as its input and returns one integer as its output.

The first task is to define a SWIG interface file that will provide all of the information needed to call the scripting language routine and process its results. The following interface file should fit the bill:

```
%module info;

%{
#include "Info.h";
%}

class Info
{
public:
    Info();
    ~Info();
    int GetArg1();
    int GetArg2();
    void PutResult(int num);
};
```

Using this interface, SWIG can generate all the code necessary to call the methods of this interface from any of the supporting scripting languages. The next task is to implement the Info class in C++, as it's so simple we can do this using all inline functions like so:

```
#ifndef __Info_H
#define __Info_H

class Info
{
public:
    Info();
    ~Info();
    void PutArg1(int num) {m_arg1=num;};
    int GetArg1() {return m_arg1;};
    void PutArg2(int num) {m_arg2=num;};
    int GetArg2() {return m_arg2;};
    void PutResult(int num) {m_result=num;};
    int GetResult() {return m_result;};
private:
    int m_result;
    int m_arg1;
    int m_arg2;
};

#endif
```

Notice the additional methods and member data not exposed by the interface passed to SWIG. SWIG should now be run on the interface file and a shared library made as discussed earlier.

---

<sup>1</sup> *It's so simple that you would never consider implementing it in this way, but it serves to illustrate the points without distraction.*

The remaining steps are specific to the scripting language being used, the following subsections show these for Python and Java.

## Python specifics

### Coding the shim

The following code shows the shim implementation.

```
from info import *
import cleverModule

def CallCleverMethod(infoThis):
    infoPtr = InfoPtr(infoThis)
    arg1 = InfoPtr.GetArg1()
    arg2 = InfoPtr.GetArg2()
    result = cleverModule.cleverMethod(arg1, arg2)
    infoPtr.PutResult(result)
```

The InfoPtr class is created by SWIG and is a Python shadow class that will not delete the corresponding C++ class when it is destroyed. This is the desired outcome other wise we wouldn't be able to get the result from it once we are back in C++.

### Calling the shim from C++

The following code fragment shows how to call the shim from C++.

```
// initialise our information object
Info info;
info.SetArg1(10);
info.SetArg2(20);

// initialise the Python interpreter
if (!Py_IsInitialized())
    Py_Initialize();

// create a SWIG pointer string
char infoBuf[128];
SWIG_MakePtr(infoBuf, &info, "_Info_p");

// find the shim module and entry point function
PyObject *pyModule = PyImport_ImportModule("CleverModuleShim");
PyObject *pyModuleDict = PyModule_GetDict(pyModule);
PyObject *pyFunc = PyDict_GetItemString(pyModuleDict, "CallCleverMethod");

// call the shim code
Py_INCREF(pyFunc);
PyObject *pyRetVal = PyObject_CallFunction(pyFunc, "(s)", infoBuf);
Py_DECREF(pyFunc);

// output the result
printf("%d", info.GetResult());
```

Error handling has been omitted for brevity.

## Java specifics

### Coding the shim

```
import info.*;
import cleverModule;

public class CleverMethodShim
{
    public static void CallCleverMethod(long infoThis)
    {
        Info infoPtr = (Info)Info.newInstance(infoThis);
        int arg1 = InfoPtr.GetArg1();
        int arg2 = InfoPtr.GetArg2();
        int result = cleverModule.cleverMethod(arg1, arg2)
        infoPtr.PutResult(result)
    }
}
```

### Calling the shim from C++

```
// initialise our information object
Info info;
info.SetArg1(10);
info.SetArg2(20);

// create Java VM
JavaVMInitArgs vm_args;
JavaVM *pVM;
JNIEnv *pEnv;
vm_args.version = JNI_VERSION_1_2;
vm_args.options = NULL;
vm_args.nOptions = 0;
vm_args.ignoreUnrecognized = TRUE;
jint res = JNI_CreateJavaVM(&pVM, (void **)&pEnv, &vm_args);

// find the shim class and entry point function
jclass pClass = pEnv->FindClass("CleverMethodShim");
jmethodID methodID = pEnv->GetStaticMethodID(pClass, "CallCleverMethod",
    "(J)V");

// call the shim code
pEnv->CallStaticVoidMethod(pClass, methodID, (jlong)&info);
pEnv->ExceptionClear();

// output the result
printf("%d", info.GetResult());
```

The call to `ExceptionClear` is required to clear any exceptions arising during execution of the Java VM, failure to call this method can leave the JVM in an unusable state for the next invocation.

## Real world examples

The preceding sections have shown the use of SWIG through two simple examples. This section builds on that information to show how SWIG has been used in real-world applications to enable simple cross-language development. Whilst not described as in-depth as the preceding section, each of the real world examples was implemented using the exact same methods as described above.

## Wrapping a large C++ class library

Chrystal Software's Astoria product supports automation and integration via a large C++ class library. The library consists of more than 80 different classes, some of which represent database objects and some of which are utility classes to aid in various processing task. To provide a wider variety of developers access to the Astoria API, it was decided to produce language bindings for Python, Perl and Java.

The C++ API was not thought suitable for direct wrapping since it makes heavy use of callback functions for returning data and pointers to allow multiple return values from one function. In addition, many of the function fingerprints were not thought "script friendly" and needed to be replaced. SWIG was chosen as the tool to automate the wrapper generation as far as possible and to allow the work in wrapping for one language to be easily reused for the others.

### Modularisation

Because of the size of the class library to be wrapped, it was decided to break up the interface files using one interface file per class and then combining them all in a single interface file using the SWIG `%include` directive. The top level `asp.i` file looks something like:

```
%title Astoria Script API
%module asp;

%section "XD_Database"
%include xddatbse.i

%section "Core API"
%include xdoobject.i

%subsection "XD_CapabilityGroup"
%include xdcapgrp.i

%subsection "XD_LinkItem"
%include xdlnkitm.i

%subsection "Named Objects"
%include xdnamobj.i

...
```

This allows easy navigation of the code and enables parallel development since individual class wrappers can be edited separately.

### Callback functions

Callback functions are used in the C++ API for any function that may return a list of results. The user supplied callback function is called once for each result. Whilst possible, it would have been very labour intensive to hand-code each of these callbacks to support calling callback functions implemented in the scripting language. Instead, the methods requiring callback functions were replaced in the script API by functions that return result list objects. The result list objects were coded in C++ and then wrapped using SWIG just like the main API classes. The wrapper code would then build the list internally using the callback interface and then return the complete list to the calling script function.

## Error handling

Almost every function in the C++ API returns an XDSTATUS return value. Java and Python both support the more modern exception based error handling so it was decided to removed the XDSTATUS return value from the script function and replace it with native scripting runtime errors. This was implemented in the wrapper code by checking the return value from the C++ function and then if an error had occurred throwing a C++ exception. SWIG error handling was then used as described above to convert these C++ exceptions into script runtime exceptions.

## Function signature changes

A typical function in the C++ API would have the following fingerprint:

```
XDSTATUS GetObject(char *name, XD_Object *pObj, BOOL *foundIt);
```

To make such functions easier to use from a scripting language, these were modified in the wrapping code to look like:

```
XD_Object GetObject(char *name);
```

In the this example, this would be achieved using the following in the interface file:

```
%addmethods XD_SomeClass
{
    XD_Object GetObject(char *name)
    {
        BOOL foundIt = FALSE;
        XD_Object object;
        XDSTATUS status;
        status = self->GetObject(name, &object, &foundIt);
        CheckStatus(status);
        CheckFound(foundIt);
        return object;
    }
}
```

where CheckStatus and CheckFound are utility function that throw a C++ exception if an error has occurred.

## Using Python to process SAX events generated in C++

Python is an excellent language for working with XML and the standard XML library contains several useful utilities. Many of these are built on SAX foundations and will happily work with any SAX compliant parser. Chrystal Software's XML repository, Astoria, stores XML internally as an object hierarchy, similar to a DOM tree. It does not provide a native SAX API, but building one on top of the existing object oriented API is straightforward. To further improve the scripting API described in the previous section, a SAX parser interface was added to allow simple interfacing between XML utilities written in a scripting language and XML data stored in the Astoria repository.

A SAX client implements at a minimum the SAX DocumentHandler interface. This interface consists of a set of functions that are called by the SAX parser to indicate the occurrence of various events during parsing of an XML document. These events include: start element, end element and character data amongst others and the parameters passed to each of the DocumentHandler functions provide details about each event.

The integration requires the following interfaces:

1. Python to C++ This allows the Python client code to register its handler functions and initiate parsing of a document stored in the repository.

2. C++ to Python This allows the C++ SAX parser to call the Python handler functions as the SAX events occur.

This two way interface is expressed in the following SWIG interface file.

```
%typemap(python,in) PyObject *pHandler
{
    $target=$source;
}

class XD_SAXDriver
{
public:
    XD_SAXDriver();
    ~XD_SAXDriver();

    // Python callback function registration
    void SetStartElementHandler(PyObject *pHandler);
    void SetEndElementHandler(PyObject *pHandler);
    void SetCharacterHandler(PyObject *pHandler);
    void SetPIHandler(PyObject *pHandler);

    // Entry point to start parsing from a given repository object
    void ParseObject(XD_Database *database, XD_Object *object);

    // Astoria specific options
    void SetAddAstID(long addIt);
    void SetAddCustAttrs(long addEm);
    void SetResolveEntAttrs(long resolveEm);
};
```

The %typemap lines is a little bit of SWIG magic used to prevent SWIG from attempting to wrap the native PyObjecttype.

A Python shim module was then written to hide the details of the SWIG interface from Python clients and make the Astoria parser appear like any other Python based SAX parser. This is shown below.

```
from xml.sax import saxlib,saxutils
from asp import *

class asp_sax(saxlib.Parser):
    "SAX parser for Astoria documents and sub documents"

    def __init__(self):
        saxlib.Parser.__init__(self)
        self.xdDriver = XD_SAXDriver()
        self.xdDriver.SetStartElementHandler(self.startElement)
        self.xdDriver.SetEndElementHandler(self.endElement)
        self.xdDriver.SetCharacterHandler(self.characters)
        self.xdDriver.SetPIHandler(self.processingInstruction)

    def startElement(self,name,attrs):
        attrList = XD_StringMapPtr(attrs)
        at={}
        for key in attrList:
            at[key]=attrList.GetValueByKey(key)
        self.doc_handler.startElement(name,saxutils.AttributeMap(at))
```

```

def endElement(self, name):
    self.doc_handler.endElement(name)

def characters(self, data):
    self.doc_handler.characters(data, 0, len(data))

def processingInstruction(self, target, data):
    self.doc_handler.processingInstruction(target, data)

def parseObject(self, database, obj):
    "Parse an Astoria structured document"
    self.doc_handler.startDocument()
    self.xdDriver.ParseObject(database, obj)
    self.doc_handler.endDocument()

def create_parser():
    return asp_sax()

```

Python clients can now use the Astoria SAX parser just like any other:

```

from xml.sax import saxlib
import asp_sax

class XMLPrinter(saxlib.HandlerBase):
    def __init__(self):
        saxlib.HandlerBase.__init__(self)

    def startDocument(self):
        print "<?xml version=\"1.0\">\n"

    def endDocument(self):
        pass

    def startElement(self, name, attrs):
        print "<%s" % (name)
        for i in attrs.keys():
            print " %s=\"%s\"" % (i, attrs[i])
        print ">"

    def endElement(self, name):
        print "</%s>\n" % (name)

    def characters(self, ch, start, length):
        print ch

parser = asp_sax.create_parser()
dh = XMLPrinter()
parser.setDocumentHandler(dh)
parser.parseObject(db, doc)

```

## Using a Java XSLT processor from C++

The previous example described handling SAX events from a C++ based parser using Python callback functions. In that case, the Python interpreter was the "top level" client program, calling into a C++ library that then called back into the Python world to deliver its output. This arrangement can be further extended by the use of embedding to allow a C++ program to make use of the script based code.

This section describes the use of a Java based XSLT processor, James Clark's XT, from a C++ client. The XSLT processor will receive it's input, an XML document and stylesheet, from a C++ based parser and send it's output to a C++ based interface.

The interface to the parser is similar to the previous example but modified to support Java instead of Python. For completeness it is shown below:

```
%typemap(java,jni) jobject *IN {jobject}
%typemap(java,jtype) jobject *IN {Object}
%typemap(java,in) jobject *IN {
    $target = &($source);
}
%typemap(java,out) jobject IN {
    *$target = (jobject)$source;
}
%apply jobject IN {jobject cbHandler};
%apply jobject IN {jobject callerData};

class XD_JSAXDriver
{
public:
    XD_JSAXDriver();
    ~XD_JSAXDriver();

    void ParseObject(XD_Database *database, XD_Object *object,
        jobject cbHandler, jobject callerData);
    void SetAddAstID(long addIt);
    void SetAddCustAttrs(long addEm);
    void SetResolveEntAttrs(long resolveEm);
};
```

Again, the %typemap declarations at the start of the interface file prevent SWIG from trying to wrap the jobject type. A Java shim class, XD\_SAXDriver, was written to adapt the C++ interface to the org.xml.sax.Parser interface to enable clients to treat the C++ parser in the same way as any Java based SAX parser.

The interface used to output the results of the XSLT transformation is shown below:

```
%module wsp
%{
#include "HttpUtil.h"
%}

class XW_HttpUtil
{
public:
    XW_HttpUtil(XD_Database *pDB=NULL,XWIO *pXWIO=NULL);
    ~XW_HttpUtil();

    XD_Database *GetDB();
    char *GetParameterValue(char *parameterName);
    void WriteHtml(char *string);
};
```

This is a very simple interface that allows the XSLT client program to obtain the document for transformation from the repository and to write the XSLT output back to the C++ program as HTML text. The Java XSLT client code is shown below.

```

import com.chrysal.wsp.*;
import com.chrysal.asp.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import com.jclark.xsl.sax.*;

class XSLSupport extends HandlerBase
{
    public static void DisplayObject(long objPtr)
    {
        try
        {
            XW_HttpUtil xwio = (XW_HttpUtil)XW_HttpUtil.newInstance(objPtr);
            XD_Database db = (XD_Database)XD_Database.newInstance(xwio.GetDB());

            // get the currently selected object
            String selStr = xwio.GetParameterValue("xwObject");

            // get the associated document
            XD_SGMLDocument docObj =
                (XD_SGMLDocument)asp_factory.Construct(db,selStr);
            String docStr = asp.ObjectToString(db._self(),docObj._self());

            // get the associated style object
            XD_Style styleObj = docObj.GetStyle(db);
            String styleStr = asp.ObjectToString(db._self(),styleObj._self());

            // apply the stylesheet
            XSLSupport xslProc = new XSLSupport(db,xwio);
            xslProc.ApplyStylesheet(docStr,styleStr);
        }
        catch (Exception ex)
        {
        }
    }

    XSLSupport(XD_Database db, XW_HttpUtil xwio)
    {
        m_db = db;
        m_xwio = xwio;
    }

    protected void ApplyStylesheet(String docObj,String styleObj)
        throws SAXException, java.io.IOException
    {
        // initialise document source
        InputSource docSource = new InputSource(docObj);
        XD_SAXDriver docParser = new XD_SAXDriver();
        docParser.setDatabase(m_db);

        // initialise style source
        InputSource styleSource = new InputSource(styleObj);
        XD_SAXDriver styleParser = new XD_SAXDriver();
    }
}

```

```

        styleParser.setDatabase(m_db);

        // initialise XSL processor
        XSLProcessorImpl xslProcessor = new XSLProcessorImpl();
        xslProcessor.setParser(docParser,styleParser);
        DocumentHandler outputHandler = (DocumentHandler)this;
        xslProcessor.setDocumentHandler(outputHandler);

        // run the transformation
        xslProcessor.loadStylesheet(styleSource);
        xslProcessor.parse(docSource);
    }

    // SAX DocumentHandler overrides
    public void characters (char ch[], int start, int length)
    {
        String str = new String(ch,start,length);
        m_xwio.WriteHtml(str);
    }

    public void startElement (String name, AttributeList atts)
    {
        m_xwio.WriteHtml("<" + name);
        for (int i = 0; i < atts.getLength(); i++)
        {
            m_xwio.WriteHtml(" "+atts.getName(i)+"="+atts.getValue(i));
        }
        m_xwio.WriteHtml(">");
    }

    public void endElement (String name)
    {
        m_xwio.WriteHtml("</" + name + ">" + "\n");
    }

    XD_Database m_db;
    XW_HttpUtil m_xwio;
}

```

Where `DisplayObject` is the entry point function.

## Conclusion

This paper shows how SWIG may be used to generate interfaces between C/C++ native code and Python, Perl and Java. In many cases, the process is straightforward and does not require the developer to have an in-depth knowledge of the target scripting language. When embedding a scripting language in a C/C++ application, the developer does require some knowledge of the target scripting language internals, but SWIG still simplifies the process considerably.

## Acknowledgments

Thanks to David Beazley at the University of Chicago for writing SWIG and releasing it for free use.

## **Author**

### **Marc Hadley**

Solutions Architect

Chrystal Software

Postal Address:

Key West

56-61 Windsor Road

SL1 2EE Slough

Berkshire

United Kingdom

E-mail: [marc\\_hadley@chrystal.co.uk](mailto:marc_hadley@chrystal.co.uk)

Web: [www.chrystal.com](http://www.chrystal.com)

**Marc Hadley** - Marc Hadley is a Solutions Architect for Chrystal Software in Europe. He has 9 years of experience in technical software development, mainly in the areas of communications and distributed systems. Marc has spent the last 2 years working on the design and implementation of SGML and XML component management systems.

Marc is a recent convert to the Python scripting language and is also looking forward to making extensive use of Java now that he has completed the Java language bindings for the Astoria API.