Bridging XPCOM/Bonobo -- implementation

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial introduction

Who should take this tutorial?

This tutorial illustrates the implementation of component bridging according to techniques introduced in the first tutorial, Bridging XPCOM/Bonobo -- techniques. It walks the reader through the building and testing of a static bridge from the XPCOM component system to Bonobo. This tutorial also introduces Buxom, an open-source software package to build the foundation of a static bridge automatically from the CORBA IDL of the target component. If you need to use components across environments, the techniques presented in this tutorial will be useful.

Component bridging can be used to embed or access controls from one system to another. For instance, you can use component bridging to be able to embed a file manipulation control from Bonobo into Mozilla. It can also be used to take advantage of complex behavior that has been implemented in another system.

Prerequisites

You should be familiar with either CORBA or a COM-like architecture like XPCOM or MS COM, preferably both. The code is written in C and C++. Familiarity with Bonobo is important, but the basic introductions listed in the Resources on page 21section should suffice.

Section 2. Executing the adapter

Setting up

Basically, GNOME 1.4 and Mozilla 0.8 are required. This means Bonobo 0.37 or higher and OAF 0.6.5 or higher.

Download the *code package* and unpack it to a suitable location.

Create a relative `build` directory in order to easily tell the special code from the generated code. Copy the code files to the build directory.

The environment

Set the `MOZILLA_FIVE_HOME` environment variable to the location of your Mozilla binaries. For instance `/usr/local/bin/mozilla-dist/bin`.

Set `OAF_INFO_PATH` to where you'll want to keep the OAF information files for the `GiveTake` factory. This may not be necessary if you can edit the OAF (object activation framework) master configuration, as described below, but it doesn't hurt. For example: `/usr/local/share/oaf`.

You'll notice a file called *COMMON_LIBS*. Rather than fuss with a Makefile, I'll present all the build commands directly. This can make the compilation command lines very long, so I packed a file with the portions of command lines that specify libraries used by all the modules. This way I can just use `` `cat COMMON_LIBS` `` (note the back-ticks), as we'll soon see.

Preparing OAF

If you have access to modify the OAF configuration file, usually at */etc/oaf/oaf-config.xml* , add in the paths where you plan to place the OAF configuration files for your GiveTake and other components. Here's an example:

```
<?xml version="1.0"?>
<oafconfig>
  <searchpath>
    <item>/usr/share/oaf</item>
    <item>/usr/local/share/oaf</item>
  </searchpath>
  <searchpath>
    <item>/home/uogbuji/tmp/oaf</item>
  </searchpath>
</oafconfig>
```

OAF scans these paths, and the paths in the `OAF_INFO_PATH` environment variable on startup.

Copy the *givetake.oaf* file to an appropriate location:

```
$ cp givetake.oaf /usr/local/share/oaf
```

The Bonobo build: compiling

Compile the IDL:

```
$ orbit-idl -I/usr/share/idl GiveTake.idl
```

Compile the Bonobo sources:

```
$ gcc -g -c `cat HEADERS` givetake_bo.c
$ gcc -g -c `cat HEADERS` main.c
$ gcc -g -c `cat HEADERS` GiveTake-common.c
$ gcc -g -c `cat HEADERS` GiveTake-skels.c
$ gcc -g -c `cat HEADERS` GiveTake-stubs.c
```

The Bonobo build: linking

Link the Bonobo server:

```
$ gcc -g `cat COMMON_LIBS` -o bonobo-givetake \
      GiveTake-skels.o GiveTake-common.o givetake_bo.o main.o
```

Link the test client:

```
$ gcc -g `cat COMMON_LIBS` -o givetake-client \
      GiveTake-stubs.o GiveTake-common.o givetake-client.c
```

Running the test client

Open up a couple of consoles and change to the directory in which you executed your build. On one console run the server:

```
$ ./bonobo-givetake
```

And in another, the client:

```
$ ./givetake-client

Gift recieved:
Gift recieved: Trip to Hawaii
```

You can see that on the first call to `gift` (in which we send "Trip to Hawaii"), we get an empty string back. And the second time, we get back the "Trip to Hawaii" string.

Compiling the XPCOM IDL

Generate the type library for the scripting support:

```
$ xpidl -I $MOZILLA_FIVE_HOME/../idl -m typelib GiveTakeXP.idl
```

And copy it to the Mozilla components directory:

```
$ cp GiveTakeXP.xpt $MOZILLA_FIVE_HOME/components/
```

Compiling the implementation

Build the shared object file for the XPCOM adapter:

```
$ gcc -g `cat COMMON_LIBS` -I$MOZILLA_FIVE_HOME/../include \
      -L$MOZILLA_FIVE_HOME/../lib -lxpcom \
      -shared -o libGiveTakeXP.so\
      GiveTake-stubs.o GiveTake-common.o GiveTakeXP.cpp
```

And copy it to the Mozilla components directory:

```
$ cp libGiveTakeXP.so $MOZILLA_FIVE_HOME/components/
```

Register the component

And we're almost there. The final step is to register the component with the XPCOM runtime.

```
$ regxpcom
```

That's it! We are now free to use the Bonobo component within XPCOM through Python, Javascript, or directly from C++.

Adapter troubleshooting

* If you run into problems running the Bonobo object server, you may not have OAF properly configured or running (check for the *oafd* process. This type of trouble is usually manifested by error messages of the form "Unable to create an instance of the OAFIID:Bonobo_Adapter_GiveTake component." Also check that there is only one instance of *oafd* running.
* Make sure your `MOZILLA_FIVE_HOME` environment refers to Mozilla's binary directory, not the directory above it. Some of Mozilla's own documentation is wrong about this.
* If you make changes to the GiveTakeXP IDL or source code and they do not seem to be taking effect, don't forget to rerun `regxpcom`. Make sure that the changed files are dated more recently than your last run of `regxpcom`.

Section 3. Buxom: a static bridge from XPCOM to Bonobo

Why Buxom?

Hand crafting an adapter can be hard work, which comes about in two distinct phases. First of all comes the setup for the static interface delegation, which can be tedious and error prone. Then comes the intellectually challenging work of tailoring the adapter to deal with the environmental issues that are particular to the task at hand.

Unfortunately, there is rarely decent help available for the latter phase except perhaps in expensive commercial bridges. A static bridge at least minimizes the overall effort by assisting with the former phase.

What does Buxom do?

Buxom generates boilerplate XPCOM code for accessing Bonobo components. The only input required is the Bonobo IDL file. From this, Buxom generates an XPCOM IDL file and an all-but-complete C++ implementation that delegates to the Bonobo object. The generated XPCOM code is quite similar to that in the hand-built adapter I presented earlier.

Buxom works on a subset of Bonobo controls and tries to accommodate the various Bonobo pitfalls I've already covered. It requires the target Bonobo component to provide an OAF factories, which shouldn't be a problem since as of GNOME 1.4, OAF is the preferred mechanism.

The workings of Buxom

Buxom is a series of Python libraries (and helpers for Bonobo's occasional CORBA noncompliance). Two of the modules, `GenXpidl` and `GenXpImpl`, work as back ends to omniidl, the superlative IDL compiler in omniORB. This Buxom requires omniORB or just omniidl. This is a much simpler approach than dealing with ORBit's libIDL.

The openness of omniidl's architecture means that Buxom should be quite straightforward to extend for more tailored static bridging tasks. But as it is, it generates the code necessary for direct delegation to the Bonobo component.

The configuration module

Buxom's configuration file is *BuxomConfig.py*, which follows a straightforward Python format. All of the Buxom files can be modified for extension and specialization, but *BuxomConfig.py* is especially meant to store parameters for ad-hoc tweaking.

As shipped, the configuration allows the user to tweak the form of generated contract IDs for XPCOM.

Bonobo IDL problems

The full IDL for Bonobo is quite extensive, and not quite CORBA compliant. The main offense is the misuse of the reserved `object` keyword.

Luckily, many bridging tasks will need no more than the heart of the Bonobo component system: the `Unknown` interface. Therefore Buxom ships with an IDL for `Bonobo:Unknown`, avoiding any problems with the very strict omniidl compiler.

Buxom subset of Bonobo.idl

This simple interface is the heart of Bonobo, and similar to the core `nsISupports` interface in XPCOM.

```
module Bonobo {
  interface Unknown {
    void ref();
    void unref();
    Unknown queryInterface(in string repoid);
  };
};
```

IDL translation

GenXpidl basically translates CORBA IDL to XPCOM IDL. The two are similar enough to make the task feasible, but different enough to make the task tricky.

The heart of the translation is the `BuxomXpidlVisitor` class, which uses omniidl's visitor pattern approach of walking the abstract syntax tree (AST) that results from IDL parsing.

BuxomXpidlVisitor

The class derives from omniidl's AST visitor, and provides specialized actions to be executed on the AST.

```
class BuxomXpidlVisitor(idlvisitor.AstVisitor):
    def __init__(self, cfg):
        #...

    def visitAST(self, node):
        #...

    def visitModule(self, node):
        #...

    def visitInterface(self, node):
        #...
```

BuxomXpidlVisitor.visitInterface: the interface name and ancestors

The node is the current position in the parse tree of the IDL, in this case, an interface definition. Each node has the relevant parsed information available through methods such as `identifier()` which given the interface name. `inherits()` returns a list of ancestor `Interface` objects, and here they are converted to a list of ancestor interface names using `map`.

```
    def visitInterface(self, node):
        name = node.identifier()
        inherits_names = map(lambda x: x.identifier(), node.inherits())
```

Buxom doesn't do anything special for `Bonobo::Unknown`, or for interfaces which to not derive from `Bonobo::Unknown`. It also doesn't support interface inheritance except directly from `Bonobo::Unknown`.

```
        if name == "Unknown" or not inherits_names:
            return
        if len(inherits_names) > 1 or inherits_names[0] != "Unknown":
            raise Exception("Only inheritance from Bonobo:Unknown supported")
```

BuxomXpidlVisitor.visitInterface: CIDs, IIDs, etc.

A universally-unique identifier (UUID) is a 128-bit number often used for global identification with minimal fear of collisions. While writing an adapter by hand, I suggested using *uuidgen* to generate these, but the `Uuid` module provided with Buxom provides UUID generation with a Pythonic interface. Furthermore, this module can return UUIDs in number form (using the `GenerateUuid()`) method or using the string representation (with the `UuidAsString()` conversion).

```
uuid = UuidAsString(GenerateUuid())
```

The description is set up simply as the interface name, which can be modified in the generated code. The contract ID (CID) is constructed using the user preferences copied in from the configuration module.

```
desc = name
cid = "%s/%s;%s"%(self._cfg.cidBase, name, self._cfg.idlVersion)
```

BuxomXpidlVisitor.visitInterface: operations

At this point, the output IDL is beginning to take shape. All the information we've been gathering is formatted to XPCOM IDL and printed to standardized output. The code to do so is not dissected in this tutorial (because of its simplicity).

Here we iterate over the operations defined in the CORBA IDL in order to render their equivalent in XPCOM IDL. The `GenerateMethodForOperation()` function generates the output for each operation.

```
callables = node.callables()
for callable in callables:
    if callable.__class__ == idlast.Operation:
        GenerateMethodForOperation(callable)
```

Interpreting parameters and return values in GenerateMethodForOperation

The operation object has a `parameters()` method which returns a list of the arguments defined in the IDL.

```
def GenerateMethodForOperation(operation):
    params = operation.parameters()
```

The return type of an operation is modeled as a `idltype.type` object. The `kind()` method on this object is an enumerated code of IDL types. It's possible that these types are not directly expressed in the IDL, but are perhaps expressed using typedefs. The `unalias()` method gets at the actual type in question.

```
    return_type = operation.returnType()
    return_kind = return_type.kind()
    if return_kind == idltype.tk_alias:
        return_type = operation.returnType().unalias()
```

Here the parameter list for XPCOM IDL is built up using item by item. At the end, the redundant comma and space are removed.

```
    param_list_str = ""
    if len(params) > 0:
        for param in params:
            #Handle each param list entry
        param_list_str = param_list_str[:-2]
```

For each parameter, the XPIDL is generated based on whether it's an `in` or `out` parameter, the parameter name (the `identifier()`) value, and `GetTypeRepr()` -- a function that maps CORBA IDL types to XPCOM IDL types.

```
            param_list_str = param_list_str \
                           + (param.is_in() and "in " or "out ") \
                           + GetTypeRepr(param.paramType()) \
                           + " " + param.identifier() \
                                + ", "
```

The main routine

omniidl's main front end calls the `run()` function on the `GenXpidl` class.

```
def run(tree, args):
    from BuxomConfig import CFG
    visitor = BuxomXpidlVisitor(CFG)
```

`BuxomConfig.CFG` is the structure with the user preferences. A visitor object is created in order to operate over the parse tree built by omniidl. This invokes the visitor object on the parse tree.

```
    tree.accept(visitor)
    return
```

Section 4. Generating the implementation boilerplate

BuxomXpImplVisitor

The basic outline of the implementation generator is the same as for the IDL generator. The `visitInterface` method is one portion that is significantly different.

```
    def visitInterface(self, node):
        name = node.identifier()
        if name == "Unknown":
            return
        corba_stub_name = string.join(node.scopedName(), "_")
```

You may have noticed from the adapter code that the CORBA implementation base names are based on the fully scoped interface name, with the scoping indicated with underscores. Therefore the fully scoped name `Bonobo::MyModule::MyClass` leads to implementation function names beginning with `Bonobo_MyModule_MyClass`. The string operation above generates this form.

BuxomXpImpl.GenerateMethodForOperation

Most of `BuxomXpImplVisitor` takes the same information extracted from the IDL abstract syntax tree and generates the C++ and XPCOM macro declarations and the implementation definitions for a direct delegation of interface. The `visitInterface` generates material for each interface and the `GenerateMethodForOperation` function generates the considerable material for each operation.

```
def GenerateMethodForOperation(operation, ifname, corbaStubName):
    #put together parameter list and return type as before

    #create parameter list for C++ methods
    #by translating from IDL parameters
```

Specializing the code for handling return values

The generated C++ code actually treats string return values and scalar type return values with entirely different logic. This is because of the special memory management needs of C/C++ strings. There is a special block of code in `GenerateMethodForOperation` that allows for this specialization.

```
#0 = void, 1 = scalar, 2 = char *
return_value_spec = 1
if return_kind == idltype.tk_string:
    return_value_spec = 2
    print "  char *rv, *rv_copy;"
elif return_kind == idltype.tk_short:
    print "  short rv;"
#Similar elif statements for other scalar types:
#tk_long, tk_ushort, tk_ulong, tk_float, tk_double
#tk_boolean and tk_void
```

The `return_value_spec` variable keeps track so we can specialize later code generation.

Generating return value code for implementations

If the return value is void, we don't set up the `rv` variable to handle the return from the Bonobo component call.

```
if return_value_spec:
    print "  rv = (char *)%s_%s(this->delegate, %s&ev);"%(corbaStubName, operation.identifier(), param_list
else:
    print "  (char *)%s_%s(this->delegate, %s&ev);"%(corbaStubName, operation.identifier(), param_list_str)
```

Dealing with return value storage management

The code for handling successful results from Bonobo methods of return value other than void is as follows:

```
if return_value_spec == 1:
    print """\
*_retval = rv;
    """
```

In this case, the result is simply copied into the storage provided by the `_retval` pointer.

```
elif return_value_spec == 2:
    print """\
rv_copy = g_strdup(rv);
*_retval = rv_copy;
    """
```

Here, the string returned from the Bonobo method call is duplicated and this new buffer is set in the `_retval` pointer. Be careful, because in cases where the caller is to manage the return value memory, this can cause leaks.

The main routine

The `run()` function for `BuxomXpImpl` also generates the global code for the C++ implementation. The section of most interest is in the beginning:

```
def run(tree, args):
    ifname = os.path.splitext(os.path.split(tree.file())[1])[0]
    lifname = string.lower(ifname)
```

`tree.file()` returns the file name of the IDL source specified on the omniidl command line. We use this to specify the filenames for the generated `#include` commands in the C++ implementation code. The filename sans extension is extracted into the variable `ifname` for this purpose.

Section 5. Working Buxom

Setting up

Download *Buxom 0.1* and decompress the archive.

In addition to the requirements mentioned for running the adapter example, Buxom needs Python 1.5.2 or higher and either omniidl 3.0.2 standalone or the full-blown omniORB 3.0.2. See the Resources on page 21section for links to these packages.

And that's pretty much it. Buxom is ready to use out of the box.

Using Buxom to Generate XPCOM IDL

The `-I` path points to the location of the *Bonobo.idl* file that came with Buxom. You can add other -I paths as necessary. The `-p` option is not necessary if you installed Buxom to your `PYTHONPATH`.

```
$ omniidl -I<path-to-bonobo-idl> [ -p<path-to-GenXpidl.py> ] \
          -bGenXpidl <path-to-CORBA-idl>
```

The resulting XPCOM IDL is printed to standard output. An example invocation is:

```
$ omniidl -I. -p/usr/local/Buxom -bGenXpidl GiveTake.idl > GiveTakeXP.idl
```

Where the output IDL is captured in the *GiveTakeXP.idl* file.

Using Buxom to generate C++ boilerplate

Where the parameters are similar to the `GenXpidl` form, and output also goes to the console.

```
$ omniidl -I<path-to-bonobo-idl> [ -p<path-to-GenXpImpl.py> ] \
          -bGenXpImpl <path-to-CORBA-idl>
```

An example invocation is:

```
$ omniidl -I. -p/usr/local/Buxom -bGenXpImpl GiveTake.idl > GiveTakeXP.cpp
```

Where the output C++ is captured in the *GiveTakeXP.cpp* file.

Rounding up

At this point, you should just be able to follow the instructions given for the manual adapter code earlier in this tutorial to build the generated files. The Buxom output should compile "out of the box," ready for direct delegation. If not, contact the author as mentioned in the Feedback on page 21panel.

If you need any specialized additions to the straightforward bridging, you can tweak the Buxom output as long as you're careful not to overwrite your work with future Buxom invocations.

Troubleshooting Buxom

* Be sure that omniORB is properly set up and in your PYTHONPATH.
* Be careful that you are not including the "official" Bonobo IDL files into the omniidl command line because these files have some CORBA compliance problems.

Section 6. Final thoughts

Where you might take Buxom

The primary intention of Buxom is as a learning and prototyping tool in your own bridge projects. As I'll soon discuss, bridging is a very complex business, and it helps to have the more mundane tasks automated for you.

One example of where you could use Buxom's bridging is to create a control panel using Mozilla to manipulate the emerging set of GNOME functions available in componentized form.

With a good deal of work, Mozilla's XUL user interface system could even be used to wrap the original GTK interfaces of Bonobo components.

Heavy-duty static bridges . . .

Most open-source bridges such as Buxom are pretty straightforward interface adapter generators. See the Resources on page 21section of the previous tutorial in this series for links to some examples.

Again, more sophisticated bridges tend to be commercial and very expensive (and implemented for more commercial tasks such as bridging Microsoft COM and commercial CORBA).

Such bridges would typically find sophisticated translations between different interface models, provide full proxy wrappers around all returned object stubs, and automatically translate exceptions.

. . . and dynamic bridges?

Dynamic bridges are an even more complex proposition.

A dynamic bridge could, for instance, translate query-interface requests on the fly from XPCOM to Bonobo, and do the same for reference management calls and factory requests. More sophisticated dynamic bridges could even translate Microsoft COM interface queries to CORBA interface repository queries.

At the very high end, a dynamic bridge might even proxy the user interface from one system to another, although this would be a very difficult task to do generically for Mozilla's UI because of its heavy cross-platform layering.

The Buxom project

Buxom is hosted as a project in the developerWorks Open source zone, where you can make contributions, as can I and anyone else. It is fully open-source and can be redistributed freely.

Some things such as attribute support, more type-conversion support, and more flexibility of inheritance are probably within reach for a manageable single-developer project. Turning it into a commercial-quality bridge would probably be a major team effort, although quite within the reach of the culture that developed Samba and Wine.

Word to the intrepid

And if you do end up using Buxom, extending Buxom, or rolling your own bridge from scratch, the main thing to keep in mind is that XPCOM and Bonobo are both under development and moving targets to some extent -- Bonobo especially. Expect the unexpected and make good use of Mozilla's and GNOME's bug tracker. Use dynamic language such as Javascript or Python for prototyping and testing, since you'll need maximum flexibility.

Section 7. Resources

Resources

* *Bridging XPCOM/Bonobo -- techniques* , the first part of this tutorial series (which includes a more extensive list of resources)
* Download Buxom from the developerWorks Open source zone
* *omniidl home page*
* *The omniORB home page*
* *The Python home page*
* See Rick Parrish's *Introduction to XPCOM*

Feedback

For technical questions about the content of this tutorial, contact the author, *Uche Ogbuji* .

Uche Ogbuji is a computer engineer, co-founder and principal consultant at *Fourthought, Inc* . He has worked with XML for several years, co-developing *4Suite* , a library of open-source tools for XML development in *Python* and *4Suite Server* , an open-source, cross-platform XML data server providing standards-based XML solutions. He writes articles on XML for IBM developerWorks, LinuxWorld, SunWorld and XML.com. Mr. Ogbuji is a Nigerian immigrant living in Boulder, CO.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.