Bridging XPCOM/Bonobo -- techniques

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 presents an example of how a developer can build an adapter and static bridge from XPCOM (particularly within Mozilla) and the GNOME Bonobo component system. 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, component bridging allows you 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 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 19section should suffice.

Section 2. Introduction to bridging

The problem

There are many component systems in use today: XPCOM, COM, CORBA Component Model (CCM), Bonobo, OpenParts, JavaBeans, etc. The component technologies typically define only a method for determining and working with the interface of some particular code -- leaving the implementation itself to the individual component developer.

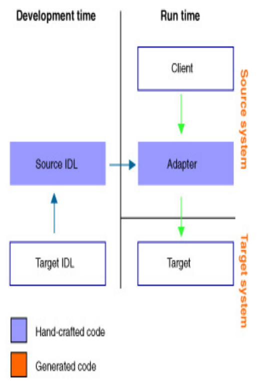
The magic of components is that when you need a bit of functionality, you can just look for a component that someone else has implemented. That is, as long as the target is designed for a component system that is available to you.

Unfortunately, there are so many component systems that it is just as likely that your perfect component is only supported on another system.

The solutions

If you are skillful with the component system that you are using, and the one which the target component supports, you have several options available. They tend to fall into three broad approaches:

* Adapters
* Static bridges
* Dynamic bridges



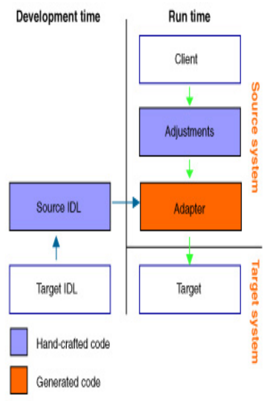
Custom bridges or adapters

Adapters, also known as wrappers, are a design pattern for making an object that's designed to work with one interface, work with another.

An adapter is basically a custom bridge that you design for your particular needs. You write code that accepts the precise interface you designed in component system 1, then translates the calls to the interface designed for the target object using component system 2. The diagram to the left illustrates the adapter pattern.

Adapters are an age-old technique, but they are very well formalized in the popular "Gang of Four" design patterns book (See Resources on page 19).

Note that the adapter pattern can also be used to connect objects with varying interfaces within the same component system.

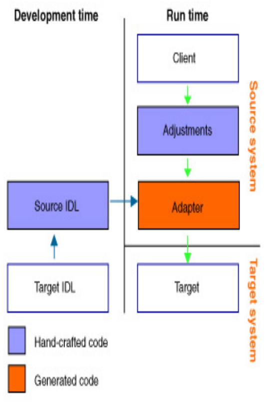


Static bridges

A static bridge is a piece of software designed to automate the development of adapters between one component system and another. It allows the developer to use any number of components based on the target system without having to manually develop the code that receives requests from one system and forwards them to the other.

A static bridge requires all aspects of the originating component system and the target component to be known ahead of when the target will be used. It involves a preparation step as part of setting up the component system. The diagram to the left illustrates a static component bridge.

The static bridge requires a special set up of the component systems before use.



Dynamic bridges

Using a static component bridge can be cumbersome, especially when there are multiple target components, or when the developer wants to merely put together a quick prototype to be sure the bridging will work.

The main problem is the need to set up the bridge for each specific target object. Most component systems provide mechanisms for dynamically finding and invoking components at runtime, without the need to set up all the interfaces statically. Some component bridges take advantage of this capability to provide the ability to dynamically invoke target objects on demand, and are thus dynamic component bridges. The diagram to the left illustrates a dynamic component bridge.

Dynamic component bridges use the interface discovery features of the originating and target systems.

The realities

Bridging can be a very complicated and expensive business. Commercial bridges can cost upwards of six figures (in US Dollars).

Most bridges are constructed for very specialized purposes and thus support a subset of either component system's features.

Dynamic bridging is especially difficult. The dynamic features of most component systems can be quite tricky. This situation is improving as such dynamic features become more normalized for use in scripting languages such as Python or Javascript.

For example, XPCOM's `xptcall` facility is a stable building block for dynamic systems, and Windows COM has very strong support for Python and VBScript. Unfortunately, many EJB and CORBA-based systems are less accessible for dynamic bridging.

Section 3. Two open-source favorites

Bonobo

Bonobo is a component system based on CORBA. It is different from the Object Management Group's official CORBA Component Model (CCM), primarily in that it is much simpler.

Bonobo is used as the core component technology for the GNOME desktop environment. It combines interface discovery, object identification and object lifecycle management ideas from Microsoft's COM with the speed and versatility of the ORBit CORBA package, also used in GNOME.

Many system tools and applets in GNOME are available as Bonobo components.

XPCOM

Cross-Platform COM (XPCOM) is a component system that originated as the glue logic for the Mozilla Web browser project. It is very similar to Microsoft's COM.

COM provides special hooks for scripting languages, such as Javascript and Python, which makes it a very good originating system for dynamic bridges, as we'll see in Bridging XMCOM/Bonobo --   implementation, the second tutorial in this series.

Almost all aspects of Mozilla can be accessed using XPCOM, and XPCOM is also an effective means for developing add ons and new behavior for Mozilla.

Bridging XPCOM and Bonobo

We'll look at techniques for building an adapter and a static bridge between XPCOM and Bonobo. By learning how to develop a custom bridge from an XPCOM client to a simple Bonobo implementation, we'll examine how to make the two environments work together.

Such a bridge would, for instance, allow the user to control or query system functions wrapped in Bonobo from a control embedded in Mozilla. Or it could allow a Windows user to invoke Bonobo controls on a Linux box.

More realities

Since XPCOM and Bonobo both use a mechanism similar to that of MS COM for interface discovery and object lifecycle management, one might consider a dynamic approach to bridging.

Unfortunately, this approach may not be realistic for some time. The main problem is that Bonobo has very poor support for scripting. ORBit, the CORBA product on which Bonobo is based, does have some support for Python scripting. But a good deal of Bonobo is rooted in specialized GTK processing that would need a great deal of core GNOME work to make readily available for scripting.

And even using adapters and static bridging, the path is precarious. Bonobo has only just been incorporated in to the stable GNOME tree as of version 1.4, and there is still some turbulence in the CORBA infrastructure of the GNOME project. Luckily, the XPCOM side of things is very solid.

Bonobo and XPCOM differences

These are some of the differences between XPCOM and Bonobo that can affect the bridge developer:

* XPCOM is geared towards C++, and Bonobo towards C. Note that the C bindings for CORBA are still being settled, and are quite convoluted, if necessarily so.
* The language difference means that in XPCOM, you use a C++ class for custom implementation of components. In Bonobo you create some implementation functions which you manage with a top-level vector, as is usual for CORBA in C, but you also take many significant detours into GTK.
* In XPCOM the natural approach to object management is Bjarne Stroustroup's famous dictum: "resource acquisition is initialization". In Bonobo you use helper functions provided by GTK architecture.
* In XPCOM you register objects with `regxpcom` using an implicit factory. In Bonobo you might use object activation framework (OAF), which is a method for registering and finding CORBA servers locate objects with CORBA object references or the naming service (or even the deprecated Gnorba library).
* In XPCOM you use method return values to signal exceptions (one wonders why not the C++ throw and catch). In Bonobo, you use the CORBA environment structure.

Section 4. An example component

Introducing "Give and Take"

In order to keep this tutorial to manageable size and complexity, the sample Bonobo component will be very simple. With only a little experience in XPCOM and Bonobo, it should become quite apparent how to extend the techniques to more complex target component.

The example problem we'll set out to solve is that we have a Bonobo component to which we can pass a string, and at the same time receive the last string that was passed to it. This is like a simple trading post: we trade one item for another. We'll call this component "Give and Take."

Our task is to make this Bonobo component available for use by XPCOM programs, including Javascript or Python scripts.

Download the code

All the code in this tutorial is available in this *code package* . Because of the amount of code involved, there will only be explanations of code that is specifically related to the component in question within the tutorial itself.

If there are other aspects of the code you'd like explained, please see the articles in the Resources on page 19section.

The GiveTake CORBA IDL, Part 1

From *GiveTake.idl*:

We wrap our component's interface definition in all the usual Bonobo trappings.

```
#include <Bonobo.idl>

module Bonobo {
  //Custom component module here
};
```

The GiveTake CORBA IDL, part 2

We'll call our module "Adapter" and our component's interface "GiveTake." The interface derives from the Bonobo `Unknown` interface for interface discovery and object lifecycle management.

The single `give` method executes the trade of strings.

```
module Adapter {
  interface GiveTake : Bonobo::Unknown {
    string give(in string gift);
  };
};
```

The Bonobo object header

givetake_bo.h provides definitions for convenient use of the `GiveTake` Bonobo object. A structure that particularly interests us is below.

This Bonobo object structure, `GiveTake`, stores the last string that was passed to it as private instance data: the member `last_gift`.

```
typedef struct {
        BonoboObject parent;
        char *last_gift;
} GiveTake;
```

The Bonobo object implementation: cleaning up

Bonobo object implementations carry some behaviors specific to GTK and some specific to CORBA. *givetake_bo.c* represents these behaviors. There is the handling of object destruction, which is shown below.

We use the GLIB `gfree` function to release the memory used by the `last_gift` string, cleaning up the only private data this object maintains.

```
static void
givetake_object_destroy(GtkObject *object)
{
        GiveTake *givetake = GIVETAKE(object);

        g_free(givetake->last_gift);
        GTK_OBJECT_CLASS (givetake_parent_class)->destroy(object);
}
```

The "give" method implementation: inputs

The string passed in by the client is the `gift` parameter.

First we extract `gt`, the Bonobo object structure, from the CORBA servant we received from the object adapter.

```
static CORBA_char *
impl_givetake_give(PortableServer_Servant servant,
                   const CORBA_char *gift,
                   CORBA_Environment *ev)
{
  GiveTake *gt = GIVETAKE(bonobo_object_from_servant(servant));
```

The "give" method implementation: return values

We prepare the return value by copying the value of the `last_gift` data member. The first time this method is invoked, there will be no value, so we use the empty string.

```
CORBA_char *retval;

if (gt->last_gift != NULL)
  retval = CORBA_string_dup(gt->last_gift);
else
  retval = CORBA_string_dup("");
```

That about wraps it up for the "give" method

Now we set the `last_gift` value for the next caller. Release the memory from the old value, then copy the string value passed in.

```
g_free(gt->last_gift);
if (gift != NULL)
  gt->last_gift = (char *)g_strdup(gift);
else
  gt->last_gift = (char *)g_strdup("");
```

Then we send back the results.

```
return retval;
```

The Bonobo object factory

Bonobo takes care of a lot of the grunt work of building an object factory for us, but we still have to set up the factory and provide a way for clients to find it.

Traditionally in CORBA, one would either distribute the factory's interoperable object reference (IOR) or use the CORBA naming service. Starting with the new GNOME 1.4, another option is the OAF.

main.c sets up the Bonobo object factory for `GiveTake` instances, and handles registration with OAF. *givetake.oaf* is the OAF configuration file for the object factory.

A direct Bonobo client

givetake-client.c is a test client, a command-line program which finds a factory, creates a `GiveTake` instance, and then invokes the `give` method a couple of times.

Section 5. The approach from XPCOM

XPCOM to Bonobo

XPCOM has some possibly useful similarities with Bonobo. They both use concepts from Microsoft COM. The nsISupports and Bonobo::Unknown root interfaces allow for a reasonable degree of transparency for the core component management.

One of the differences between Bonobo and XPCOM is that XPCOM implementations are predominantly in C++ and Bonobo implementations in C. In general, one should be careful when linking the two code bases together by using the `extern "C"` specification in headers used by C++ code.

```
#ifdef __cplusplus
extern "C"
{
#endif

//Function declarations here

#ifdef __cplusplus
}
#endif
```

We won't run into any trouble with the C/C++ interface in this tutorial.

The XPCOM IDL module level

An adapter usually involves translation of the IDL from the target system. Luckily, XPCOM IDL is similar to CORBA IDL -- with the exception of the `scriptable` stereotype.

The UUID is generated using the `uuidgen`.

```
#include "nsISupports.idl"

[scriptable, uuid(49aba66f-15b1-42e6-8245-12578b6d1f7b)]
```

The GiveTake interface in XPCOM IDL

This is the rest of *GiveTakeXP.idl*.

```
interface GiveTake : nsISupports
{
  string give(in string gift);
};
```

Identifying the adapter

The code for implementing the interface, and for registration with the XPCOM runtime, is combined in *GiveTakeXP.cpp*. First come the contract and interface IDs:

```
#define NS_GIVETAKE_CONTRACTID "@uche.ogbuji.net/GiveTake;1.0"
#define NS_GIVETAKE_CLASSNAME "Give a gift, take a gift"
#define NS_GIVETAKE_CID {0x49aba66f, 0x15b1, 0x42e6, \
  { 0x82, 0x45, 0x12, 0x57, 0x8b, 0x6d, 0x1f, 0x7b }}
```

The GiveTake class definition

We need to keep reference to the Bonobo object in our XPCOM implementation.

As you can see in this sample, `gt_server` is the CORBA server, which we keep track of in order to clean up at destruction time. The complexity of dealing with multiple component systems makes especially the "resource acquisition is initialization" rule. `gt` is the Bonobo object to which the method calls will be delegated.

```
class GiveTake_i : public GiveTake
{
private:
  Bonobo_Adapter_GiveTake gt;
  BonoboObjectClient *gt_server;
public:
  //The component interfaces
};
```

The adapter constructor: retrieving the object server

The constructor grabs an instance from the `GiveTake` object factory.

First `InitBonobo()` handles all the details of starting up the Bonobo runtime to make the object factory ready for clients.

The OAF is used to activate and retrieve a `GiveTake` CORBA instance. The ID used is the same one registered in *givetake.oaf*.

```
GiveTake_i::GiveTake_i()
{
  NS_INIT_ISUPPORTS();
  CORBA_Environment ev;
  char *obj_id = "OAFIID:Bonobo_Adapter_GiveTake";

  InitBonobo();
  this->gt_server = bonobo_object_activate(obj_id, 0);
```

The adapter constructor: narrowing the resulting object

After verifying that we have a valid CORBA server, we convert it into a Bonobo object.

```
  if (!this->gt_server) {
    printf(_("Unable to create an instance of the %s component"), obj_id);
    return;
  }

  this->gt = BONOBO_OBJREF(this->gt_server);
}
```

The adapter destructor

Once the XPCOM adapter is destroyed, we no longer need the Bonobo object and can unreference it, which leads to its reclamation.

```
GiveTake_i::~GiveTake_i()
{
  bonobo_object_unref(BONOBO_OBJECT(this->gt_server));
}
```

Initializing Bonobo through XPCOM

The key thing to note in the Bonobo initialization function lies in the first part:

```
static void InitBonobo()
{
  CORBA_ORB orb;
  char *argv[] = {"regxpcom"};

  gnome_init_with_popt_table("xpcom-givetake", "1.0", 1, argv,
                             oaf_popt_options, 0, NULL);

  orb = oaf_init(1, argv);
```

Usually, when writing a command-line program for initialization, you pass the command-line option information `argc` and `argv` to `gnome_init_with_popt_table` and `oaf_init`.

However, since this component is called from XPCOM, we don't have access to the command-line. So we simply fake a command-line with no arguments. This corresponds to an `argc` of 1 and an `argv` which is an array of a single string with the name of the command executed. We simply assume regxpcom, which is usually used to register components.

The method implementation: setting up

We have to prepare the CORBA environment for the delegated call to the CORBA object.

```
NS_IMETHODIMP GiveTake_i::Give(const char *gift, char **_retval)
{
  CORBA_Environment ev;
  char *msg, *msg_copy;

  CORBA_exception_init (&ev);
```

The method implementation: making the call and handling errors

So we simply take the argument string we got in the call to the adapter, and pass it on by calling the Bonobo object's matching method. We get the method return value as the return value from the C function in our call to CORBA.

If the CORBA call results in an exception, which can be checked through the environment structure, this is signalled back to the XPCOM caller by returning the failure flag `NS_ERROR_FAILURE`.

```
  msg = (char *)Bonobo_Adapter_GiveTake_give(this->gt, gift, &ev);
  if (ev._major != CORBA_NO_EXCEPTION){
    CORBA_exception_free(&ev);
    return NS_ERROR_FAILURE;
  }
}
```

The method implementation: handling success

In the XPCOM binding the return value is passed back by passing assigning to the `_retval` parameter.

The string returned from the CORBA call is duplicated for the return. Finally, success is signalled with the `NS_OK` flag.

```
  else {
    CORBA_exception_free(&ev);
    msg_copy = g_strdup(msg);
    *_retval = msg_copy;
    return NS_OK;
  }
}
```

XPCOM registration

In order to provide access to client scripts, we must register the XPCOM adapter we have created. We can add any actions we need in the component registration by adding to the `nsGiveTakeRegistrationProc` function.

```
static NS_METHOD nsGiveTakeRegistrationProc(
    nsIComponentManager *aCompMgr, nsIFile *aPath,
    const char *registryLocation, const char *componentType)
{
  // Registration specific activity goes here
  return NS_OK;
}
```

Section 6. Resources and feedback

Resources

Continue to the second part of this series Bridging XPCOM/Bonobo - implementation. If you've had enough for now, you can always come back and access the second part of this series at a later date. If you decide to come back later, you'll just be asked for your username and password -- you won't need to register again.

GNOME and Bonobo resources
* *The Bonobo home page*
* *Miguel de Icaza's Introduction to Bonobo*
* *GNOME & CORBA: a tutorial by Ruiz, Lacage and Binnema*
* *GNOMEnclature: Intro to Bonobo by George Lebl*
* *The GNOME components list home page* , including archives, where Bonobo is discussed
* *On Writing A Bonobo Control, a tutorial by Dirk-Jan C. Binnema*

XPCOM resources
* Check out Rick Parrish's *Introduction to XPCOM*
* *The Mozilla home page*
* *XUL tutorial that covers XPCOM*
* *Tutorial on XPCOM and extending Mozilla*
* *Information for XPConnect* , for XPCOM interface to Javascript, including XPIDL

Bridging resources
* *Announcement of XParts* , by Matthias Ettrich, a system for embedding XPCOM and potentially other components into KDE's KParts
* *The XParts home page*
* W. Eliot Kimber presents his *bridge from Microsoft COM to CORBA* using Python and omniORB
* *Arpad Kiss's home page* includes links to a Dynamic COM/CORBA bridge, again using Python and omniORB
* *Project Blackwood* includes BlackConnect: A Java Bridge to XPCOM

Other resources
* Pattern Languages of Program Design by James O. Coplien and Douglas C. Schmidt is a culmination of an intensive effort to capture and refine a broad range of software development expertise in a systematic and highly accessible manner.

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.