The GNOME & CORBA

Mathieu Lacage

Dirk-Jan C. Binnema

The GNOME & CORBA
by Mathieu Lacage and Dirk-Jan C. Binnema

Table of Contents

List of Tables

List of Figures

Introduction

This is a book on using CORBA in the context of the GNOME free software desktop enviroment. It is divided into two parts:

- The first part discusses GNOME and 'conventional' CORBA;
- The second part discusses the Bonobo technology, which is built on CORBA.

The aim of the first part of this book is to give the reader a good introduction into the overall deployment of CORBA within the GNOME environment. (the GNU Object Model Environment (http://www.gnome.org)). GNOME is a GNU project, aiming at bringing the benefits of free software to the end-user. It does so by providing a framework for the development of end-user applications. This framework is made up of some libraries built on top of GTK (http://www.gtk.org) (The *Gimp Toolkit*: a C-language toolkit for building GUIs) which provide applications with a common look and feel. GNOME is also provides the standard things for a graphical desktop, such as program launchers, taskbars, clocks etc. Interaction between various GUI components is done through *CORBA*.

Because C is the language of most core GNOME applications, this book will cover programming CORBA using C. In principle, CORBA doesn't care about the programming language. However, so far there's only a C-binding provided by the GNOME's standard ORB, *ORBit*. Note that bindings Perl and Python are in beta, a C++ binding is being worked on, and a Guile-binding is being discussed.

The first part assumes that the reader has at least some knowledge of OOP (Object Oriented Programming), C, C++ (for the first 4 chapters) and GNOME-programming (for the 5th chapter)

The second part of this book discusses *Bonobo*. Bonobo is a system which will ultimately provide the ability to a lot of things that users of GUI-systems such as MacOS and Windows have gotten used to, such as embedding spreadsheets in word-processing documents, reusable userinterface controls, scriptable components etc. We try to give the reader an understanding of both the Bonobo inner-workings and its API. We'll also provide the reader with a lot of sample code.

The second part would be best read if the reader has already some knowledge of COM/DCOM and OLE. However, it shouldn't be required.

Chapter 1. The IDL and its C mapping

Why the IDL ?

CORBA allows programmers to access the same object from any langage: this can only be achieved through the use of a common "contract". All programs accessing this object need to know the methods' signatures, their names, etc... To achieve this, CORBA defines the IDL (Interface Definition Langage) which, as its name suggests it, is used to write the common "contract". The IDL is a langage which roots itself in C++.

The IDL is also used by the idl-compiler to generate sample stubs and skeletons to ease the programmers' life :)

IDL basics

IDL modules and interfaces

The IDL must allow programmers to define thousands of different objects. The CORBA standard itself defines hundreds of them and any project with a medium size will need hundreds of them. To achieve this, one must take care of the name scope: different interfaces' names must not clash. To adress this problem, IDL has "modules": "interfaces" -that is object declarations- are valid only within a "module".

```
module FruitsBasket {
   interface Apple {};
   interface Orange {};
};
```

Here, we define two objects: Apple and Orange within the FruitsBasket module. If we wish to reference an object from another module, we have to give the full object's reference. For example, from the VegetablesBasket module, references to Apple are made as follows: `FruitsBasket::Apple`.

It is also possible to reopen the same module to add interface definitions: the folowing example is strictly equivalent to the precedent one.

```
module FruitsBasket {
   interface Apple {};
};

module FruitsBasket {
   interface Orange {};
};
```

The IDL, as a child of C++ has also precompiler directives: #*include* and #*pragma* are supported (these directives are expanded by the idl-compiler). #include has the same use as in C++: it will allow you to split different interfaces from the same module in different files -clarity-.

```
/* this is a comment like in C */
// this is a comment like in C++ : both are valid
// this code is the file apple.idl
```

```
#include "orange.idl"
module FruitsBasket
   interface Apple {};
};

// this code is the file orange.idl
module FruitsBasket
   interface Orange {};
};
```

IDL types

Because the IDL main purpose is to achieve portability between numerous langages, and on different network systems, it is very strongly typed. Here are the most basic (standard) CORBA types:

Table 1-1. Basic CORBA types

Type	Signification
short	16 bit signed integer
unsigned short	16 bit unsigned integer
long	32 bit signed integer
unsigned long	32 bit unsigned integer
long long	64 bit signed integer
unsigned long long	64 bit unsigned integer
float	32 bit IEEE float
double	64 bit IEEE float
long double	128 bit float
boolean	boolean value: TRUE or FALSE
octet	8 bit byte
char	8 bit character (ISO latin-1)
wchar	international character format. FixMe: who knows what format?
string	character string based on char
wstring	character string based on wchar

Their use is fairly straightforward: here is some IDL interface declaration using the string type and the float type:

```
module FruitsBasket {
   interface Apple {
      attribute string color;
   };
   interface Orange {
      attribute float size;
   };
};
```

Each object gets an attribute of type float and type string...

The IDL will alow you to define constants thanks to the *const* modifier (like in C++). Here is some sample IDL code:

```
module FruitsBasket {
   interface Apple {
      attribute string color;
      const float weight = 2.3;
      const string type = "sample type";
   };
   interface Orange {
      attribute float size;
   };
};
```

It is also possible to define your own types using the *typedef* keyword (again, like in C and C++). Again, sample code which will define an Apple Object with the type attribute which is a string.

```
module FruitsBasket {

   typedef string fruit_type;

   interface Apple {
       attribute fruit_type type;
   };
   interface Orange {
   };
};
```

We have also standard C/C++ structures, enumerations and arrays. Structures, enumerations and fixed size arrays are straight C code like "typedef":

```
module calendar {

    enum a_month {
        january, february, march, april, may, june,
        july, august, september, october, december
    };

    enum a_day {
        monday, tuesday, wednesday, thursday, friday, saturday, sunday
    };

    typedef long a_year;

    struct a_date {
        a_day    the_day;
        a_month  the_month;
        a_year   the_year;
    };

    interface calendar {
        attribute a_date the_date;
        // 1 dimensional array
        typedef a_date a_date_array[20];
        attribute a_date_array the_date_array;
        // 2 dimensional array
        typedef a_date_array a_date_array_multi[20];
        attribute a_date_array_multi the_date_array_multi;
    };
};
```

Variable size arrays are a little more complex: they are named sequences. Sequences are just one dimensional arrays.

```
module calendar {

    interface calendar {
        /* This defines a one dimensional array
         * of longs with a maximum size of 20.
         */
        typedef sequence <long,20> array_1;
        // an unbounded one dimensional array of strings
        typedef sequence <string> array_2;
        // more tricky: a sequence of sequences: used
        // to simulate multi dimensional variable size arrays
        typedef sequence <sequence <long,20> > array_3;
    };
};
```

IDL methods

All these different standardized types allow programmers to define object attributes and methods. Objects methods are declared with:

```
module FruitsBasket {
    interface Apple {

         // this is the declaration of the eat_me method
         // which takes as parameters a boolean variable.
         void eat_me (in boolean eat_yes_or_not );

         // returns whether the Apple was eaten or not.
         boolean eaten ();

         // returns whether the apple was eaten or not and the surname
         // and name of the man who ate the Apple.
         boolean who_ate ( out string who_surname, out string who_name );
    };
};
```

These declarations are straight C++ code except for the *in* keyword: parameters may have 3 types: *in*, *out*, *inout*. Their use should be rather easy: "in" parameters are data send from the client to the object, "out" parameters are data send from the object to the client and "inout" parameters are first send from the client to the object then from the object to the client.

All these method declarations are synchronous operations. ie: using one of these methods means that your program will wait for the object to answer to continue its execution. It is possible to define asynchronous methods so that the method caller continues to execute and does not wait blocked for the object answer: the *oneway* keyword is used.

```
module FruitsBasket {
    interface Apple {

         // this is the declaration of the eat_me method
         // which takes as parameters a boolean variable.
         void eat_me (in boolean eat_yes_or_not );

         // returns whether the Apple was eaten or not.
```

```
          // asynchronous method.
          oneway boolean eaten ();

          // returns whether the apple was eaten or not and the surname
          // and name of the man who ate the Apple.
          boolean who_ate ( out string who_surname, out string who_name );
     };
};
```

Methods may also throw exceptions. Exceptions are defined that way:

```
module FruitBasket {

    exception no_more_fruits {
        string reason;
    };

    interface Apple {
        // whatever methods we wish
    };
};
```

Last, a method declares that it may throw exceptions that way:

```
module FruitsBasket {

    exception no_more_fruits {
        string reason;
    };

    interface Apple {
        void eat_me (in boolean eat_yes_or_not )
             raises ( no_more_fruits );
        // whatever methods we wish
    };
};
```

Like in C++, objects may inherit other objects behaviour and multiple inheritance is allowed. Syntax is standard C++ but redefining generic parent methods in children is forbidden as the IDL does not allow name conflicts.

```
module FruitsBasket {

    exception no_more_fruits {
        string reason;
    };

    interface generic_fruit {
         void eat_me (in boolean eat_yes_or_not )
              raises (no_more_fruits);

         oneway boolean eaten ();
    };

    interface generic_fruit_one {
         boolean who_ate ( out string who_surname, out string who_name );
    };

    interface Apple : generic_fruit, generic_fruit_one {
         // Apple specific methods go there.
    };
};
```

We have had a small trip around the IDL existing types, ways to create new ones and objects' attributes/methods declaration . However, the reader should read the IDL definition from the CORBA standard to discover a few new types like: *any*, *TypeCode* and *union*. These types are not that important for a first reading.

IDL and its C mapping

The IDL compiler: orbit-idl

We have seen in the above section how powerfull the IDL is to describe objects which can be accessed from any langage: the IDL encapsulates all the necessary information on objects so that any langage may build its own representation of these objects and access them using the langage object tools. The IDL is nothing but a description langage which is then used to build the langage's effective objects. This building process is specific to each langage: it is called the langage mapping.

Langage mappings are part of the CORBA standard. This means that the standard describes for many langages how objects should be accessed. Among the existing mappings i know of are : C, C++ and COBOL. The C++ mapping is generally explained in most books about CORBA but the C mapping is not widely used except in The GNOME (which is why you're reading this, don't you ?).

ORBit has an idl compiler which automatically generates the C headers declaring the functions which correspond to the objects' IDL definitions (it follows the standardized mapping). Its use is: "orbit-idl file.idl". If you had the apple.idl file, running the idl compiler will generate apple.h, a C header (only C now but C++ later perhaps) containing all the function and/or type declarations corresponding to the C mapping. It will also generate apple-common.c, apple-stubs.c and apple-skels.c. These files should be compiled and linked against the client and server to provide the default stubs and skeletons.

IDL mapping basics: methods and attributes

The C mapping is "standard" C object. That is: it follows the rules found in gtk, gnome and motif when dealing with objects. Here are the simplest rules.

Objects' methods must be fully qualified. ie: the C name of the eat_me method of the Apple interface of the FruitsBasket module is `FruitsBasket_Apple_eat_me`. (that is why the standard discourages the use of method names with undescores and encourages the use of mixed case. The standard would prefer EatMe in place of eat_me so that we have `FruitsBasket_Apple_EatMe`. That way, module/interface/method are easily identified). However, i noticed the standard does not follow this rule very strictly.

For example, our Apple interface would translate into the folowing C declaration:

```
module FruitsBasket {
    interface Apple {
         void eat_me (in boolean eat_yes_or_not );
    };
};
```

is mapped to :

```
typedef CORBA_Object FruitsBasket_Apple;
void FruitsBasket_Apple_eat_me ( Fruits_Apple object,
                                 CORBA_boolean eat_yes_or_not,
                                 CORBA_Environment *ev);
```

All method calls need an object reference as first parameter and a CORBA_Environment object as last parameter. Object References are of the CORBA_Object type which is an opaque type (ie: you cannot access any of its fields or anything directly). In fact the CORBA_Object type is a pointer to a struct which is dependant on the CORBA implementation you are using.

Operation specific arguments are passed between the CORBA_Object and the CORBA_Environment parameters. Operations specific arguments should be passed as pointers if used as out or inout parameters. If struct, string or array are used as in parameters, pointers must be passed (a pointer to the first element of the array and to the first element of the string). Return arguments are used exactly in the same way: they will be pointers for structures, unions, strings, sequence and arrays. More details on this lies on pages 19-21 ... 19-24 of the CORBA specifications.

Attributes mapping is special: folowing good OOP practice, objects' attributes may not be accessed directly: two functions are defined: a _get and a _set function. For exemple, the folowing two definitions are strictly equivalent:

```
module FruitsBasket {
    interface Apple {
         attribute boolean is_eaten;
    };
};
// The folowing is theorically equivalent to the first
// definition but is illegal as IDL forbids the use of
// names preceded by an underscore.
module FruitsBasket {
    interface Apple {
         boolean _get_is_eaten (void);
         void _set_is_eaten ( in boolean b );
    };
};
```

consequently, the folowing C code will allow you to access this interface is_eaten attribute.

```
CORBA_Object app;
CORBA_Environment *ev;
CORBA_boolean bool;

  /* here is some code to link app to the real
   * running server object
   */

/* please, note the double underscore. One is due to the
 * full scoped name and the other one is due to the method
 * name
 */
FruitsBasket_Apple__set_is_eaten (app, TRUE, ev);
bool = FruitsBasket_Apple__get_is_eaten (app, ev);
```

IDL types mapping

Constants defined with the *const* keyword are #defined:

```
const long myconstant = 43;
```

is mapped to :

```
#define my_constant 43
```

Basic CORBA types are simply mapped using the folowing table:

Table 1-2. Basic CORBA types

IDL Type	C type mapping
short	CORBA_short
unsigned short	CORBA_unsigned_short
long	CORBA_long
unsigned long	CORBA_unsigned_long
long long	CORBA_long_long
unsigned long long	CORBA_unsigned_long_long
float	CORBA_float
double	CORBA_double
long double	CORBA_long_double
boolean	CORBA_boolean
char	CORBA_char
wchar	CORBA_wchar

CORBA_boolean is mapped to C unsigned char and enums are #defined .

All other types (struct, sequence, string, wstring) are variable lenght types and are specially handled. If a function returns one of these variable type, one must free the allocated memory with `CORBA_free()` which will take care of getting rid of this memory.

Structs are straight C structs...

The folowing sequence would be mapped as follows:

```
sequence <char,10> my_var;
```

is mapped to:

```
typedef struct {
    CORBA_unsigned_long _maximum;
    CORBA_unsigned_long _lenght;
    CORBA_char *_buffer;
} CORBA_sequence_char;
typedef CORBA_sequence_char my_var
```

This C structure is self explanatory: _maximum is used to store the *maximum* number of elements in the sequence. *_lenght* maintains the effective number of elements and *_buffer* is a pointer to the C array containing the data.

Sequence names are as follows: *CORBA_sequence_anytype* where anytype is the sequence specified type.

When passing such a structure as an in parameter, you should take care of setting the proper values for all fields and make sure proper memory allocation is made. If used as an out parameter, you will need to take care only of the deallocation by using the `CORBA_free` function. Lastly, if using an inout parameter, you must make sure that *_maximum*, *_lenght* and *_buffer* are correctly set (ie: like for the in parameter). Upon return, *_lenght* will hold the number of returned elements and *_buffer* memory may be freed using `CORBA_free`. It is recommended that *_buffer* memory be allocated with `CORBA_sequence_*_allocbuf`. This is so because if the parameter is inout, it may be modified by the callee and its size and thus its allocated memory may change. Sample code:

```
// here is the interface we wish to access:
module my {
    interface seq {
        void SetThing ( in sequence <long long, 20> );
        void GetThing ( out sequence <long long, 20> );
        void SendReceiveThing ( inout sequence <long long, 20> );
    };
};

/* here is the C code to call the different methods */

CORBA_Object obj;
CORBA_Environment ev;
CORBA_sequence_long_long *seq_one;
CORBA_sequence_long_long seq;

/* code to link to the real object server */
my_seq_GetThing (obj, seq_one, &ev);
/* here, seq_one is correctly assigned */
/* get rid of the memory allocated to store
 * temporarily the object sequence */
CORBA_free (seq_one);


seq._maximum = 20;
seq._lenght = 2;
seq._buffer = CORBA_sequence_long_long_allocbuf (2);
seq._buffer = { 1, 1 };
my_seq_SetThing (obj, &seq, &ev);

my_seq_SendReceiveThing (obj, &seq, &ev);
/* here, _buffer may be anything: it may be
 * memory not allocated by us because
 * the callee will CORBA_free our _buffer and
 * reallocate a new buffer is its size
 * is not big enough
 */
CORBA_free (seq._buffer);
```

To get a better understanding of how these sequences are working, you should read the CORBA standard from page 19-13 to page 19-16. Interesting stuff can be found there (but nothing really vital lies there). This should be done upon a second reading of this document.

Strings are simple C CORBA_char arrays which should be allocated using `CORBA_string_allocate`. Their use is similar to that of sequences. Here is sample code:

```
// here is the interface we wish to access:
module my {
    interface str {
        void SetThing ( in string my_str );
        void GetThing ( out string my_str );
```

```
        void SendReceiveThing ( inout string my_str );
    };
};

/* here is the C code to call the different methods */

CORBA_Object obj;
CORBA_Environment ev;
CORBA_char *str;

/* code to link to the real object server */

my_str_GetThing (obj, &str, &ev);
/* here, str is correctly assigned */
/* get rid of the memory allocated to
 * store temporarily the object sequence */
CORBA_free (str);


str = CORBA_string_alloc (5);

my_str_SetThing (obj, str, &ev);

my_str_SendReceiveThing (obj, &str, &ev);
/* here, str may be anything: it may be
 * memory not allocated by us because
 * the callee will CORBA_free our str and
 * reallocate a new buffer if its size
 * is not big enough
 */
CORBA_free (str);
```

Again, to better understand their use and be more efficient in the allocation/deallocation process read the CORBA standard from page 19-16 to page 19-18. Interesting stuff lies there which could be of some use for you. If strange things happen while passing or getting strings to/from methods, you should definitely read this stuff. Note that i kept thing simple and what i explain is definitely not what should be done or what could be done but is what is simple to do.

Exception handling

Exception handling in C will be explained later: it is a little complex because C has no built in mechanism to handle exceptions.

Chapter 2. The CORBA module

Introduction

An important part of the CORBA standard is the *Interface Definition Language* (IDL) and its various mappings. IDL is used by the standard to describe all standardized objects. The ORB is a CORBA object which has IDL interfaces: programs access the ORB through IDL interface methods. This allows programmers to be completely independant of different ORB implementation at the code level.

The CORBA module is the most important one. Here is a simple summary of what can found there:

```
module CORBA {
 // IDL definitions of the CORBA bus interfaces
 // IDL definitions of the POA and the BOA
 // IDL definitions of the Interface repository
 // IDL definitions of Dynamic mechanisms

  module IOR {
    // The IOR definitions.
  };

  module IIOP {
    // The IIOP protocol definitions.
  };

  module DCE_CIOP {
    // The DCE_CIOP protocol definitions.
    // This protocol was defined for systems integration
  };

  module GIOP {
    // The GIOP protocol definitions
  };
};
```

We will study the CORBA bus definitions in this chapter: the Object and the ORB interfaces. This will allow us to write our first simple client.

The different protocols will not be studied: the standard was written to be read and unless you are willing to write an ORB implementation, you don't need to know how these work.

The `CORBA::Object` interface

Here is an abstract of the Object interface:

```
module CORBA {

  interface Object {
    InterfaceDef get_interface ();

    boolean is_nil ();
    Object duplicate ();
    void release ();
```

```
    boolean is_a (in string logical_type_id);
    boolean non_existant ();
    boolean is_equivalent (in Object other_object);
    unsigned long hash (in unsigned long maximum);
  };
};
```

This interface is very important, because it is implicitly inherited by all interfaces defined by either the CORBA standard or by you. This means that all the operations defined in this interface are available on all other objects.

Among these standard operations are those which deal with the object identity: copy, test, destruction...

The `is_nil` method allows you to test whether an object exists or not. If true, the object does not exist but otherwise, the object is not completly guaranteed to exist (see exception handling). This method is guaranteed to return true when tested against the *CORBA_OBJECT_NIL* macro.

The `is_equivalent` method will test whether two objects are "equal" (ie: one is a copy of the other). To do so it will use the `hash` method. Last, we have the `duplicate` and `release` methods which are used to do copies of objects references and destroy these copies. Please, note that these last 2 methods work on object references, not on the object itself.

When dealing with objects in CORBA, you allways use a *CORBA_Object* opaque type variable which holds the object information and which is passed as first argument to every method. The duplicate method allows you to make a copy of this object reference and the release method to destroy this copy. These operations are necessary because the *CORBA_Object* type is opaque: you don't know its stucture and it is thus impossible to copy its internal representation by yourself.

Here is some sample code to allow you see these things work:

```
CORBA_Object obj;
CORBA_Environment ev;
CORBA_Object obj_copy = CORBA_OBJECT_NIL;

/* here lies some code to link to the target object */
/* now, obj references a real object */

if (CORBA_Object_is_nil (obj, &ev))
  printf (stderr, "Error: could not get object reference !!\n");

obj_copy = CORBA_Object_duplicate (obj, ev);

if (CORBA_Object_is_equivalent(obj, obj_copy, &ev))
  printf (stderr, "Cool: the copy succeeded\n");

CORBA_Object_release (obj_copy, &ev);
```

I think that this code does not need any comments.

There is one last method we did not go into: `get_interface`. This method will be discussed in a later chapter on the interface repository. However, ORBit does not implement yet, so it is of little use :)

The `CORBA::ORB` interface

In all the code examples we have seen so far, there are 2 or 3 lines at the beginning of the program in the like:

```
/* here lies some code to link to the target object */
/* now, obj references a real object */
```

Now it's about time to try to replace these with some real code. This is where real magic lies :) The ORB is the secret repository of all this magic.

```
module CORBA {

  typedef string ORBid;
  typedef sequence <string> arg_list;
  ORB ORB_init (inout arg_list argv, in  ORBid orb_identifier);

  interface ORB {
    string object_to_string (in Object obj);
    Object string_to_object (in string str);
  };
};
```

Initialisation

One of the first thing to do is to initialize the magic: the `CORBA_ORB_init` function must be called prior to anything else. The ORBit parameter must uniquely identify the ORB in environments where more than one ORB is running. We will use: "orbit-local-orb". This string is specific to each ORB implementation. For example, here is how we could use this function:

```
#include <orb/orbit.h>

int main (int argc, char **argv)
{
   CORBA_Object orb;
   CORBA_Environment ev;

   orb = CORBA_ORB_init (&argc, argv, "orbit-local-orb", &ev);

};
```

However, it is obvious this will not work because the ev variable is not initialized. It should be initialized with a call to `CORBA_exception_init (CORBA_Environment *ev)`. This call, specific to the C mapping, will be explained in this chapter's last part with exception handling. On the whole, this set of calls are complete magic: upon startup, the program has no Object reference and cannot call any interface' method: that is why the `CORBA_ORB_init` function exists which does not need any Object parameter. It is a "bootstrap function of the core ORB".

Object reference serialization: ior strings

Once the ORB is initialized, we would like to get our object reference to call its methods. To do so, it is possible to use CORBA naming service which associates to each registered object a human readable string but this is kept for chapter 3. Hopefully CORBA has another simpler (i am not convinced ior strings are simpler but... They are more basic. Less elegant for sure.) mechanism: IOR strings.

When you write the server object, you have to register this object in the ORB so that the ORB may generate object references to be used in clients. These object references contain all the necessary information to access the object: its location, etc... The ORB provides a mean to "serialize" these object references. That way, when you launch your server, it will register in the ORB, getting in reponse its object reference which it will serialize in a file. This file may then be opened by the client and be used to get the object reference back.

This is achieved through the use of two ORB methods: `object_to_string` and `string_to_object`. The server will call `object_to_string` and the client `string_to_object`. These strings are IOR strings and their structure depends on the protocol used, the kind of object... They are usually rather long (a few hundred characters).

Here is simple server code to generate the serialized file:

```
#include <orb/orbit.h>

int main (int argc, char **argv)
{
  CORBA_ORB orb;
  CORBA_Environement ev;
  CORBA_Object obj;
  FILE *file;

  CORBA_exception_init(&ev);
  orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", &ev);

  /* code to register the different methods */
  /* this step results in getting a CORBA_Object,
   * obj, which contains the object reference */

  if (!file = fopen ("object.ior", "w")) {
    fprintf (stderr, "error: could not open serializing file\n);
    exit (1);
  }

  fprintf (file, "%s", CORBA_ORB_object_to_string (orb, obj, &ev));

  fclose (file);

  return 0;
}
```

Now, client code to invert the process

```
int main (int argc, char **argv)
{
  CORBA_ORB orb;
  CORBA_Environement ev;
  CORBA_Object obj;
  FILE *file;
  /* rather long string: should be enough for all our needs :) */
  char buffer[10000];
  char *ior;

  CORBA_exception_init(&ev);
  orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", &ev);

  if (!file = fopen ("object.ior", "r")) {
    fprintf (stderr, "error: could not open serializing file\n);
    exit (1);
  }
```

```
  fgets (buffer, 10000, file);
  ior = g_strdup (buffer); /* gets the raw string */

  /* now, we have it finally !! */
  obj = CORBA_ORB_string_to_object (orb, ior, &ev);
  /* now, it is possible to use this object reference
   * to call some of its methods
   */
  fclose (file);

  return 0;
}
```

This code obviously assumes that client and server are started in the same directory on the same machine. We'll see later how to deal with the naming service so that we don't have to meddle with IOR strings.

A complete Hello World client

Exception handling: more C mapping

Until now, in all the code fragments I showed, there were some CORBA_Environment variables lying aroun with some `CORBA_exception_init` calls... Let's see what these variables are about.

This is no news: there is no support in C for exception handling. One has to create complete systems to support signals and the like (look at gtk signal system). CORBA has such a system. Its use is really simple. Each method has a *CORBA_Environment* * parameter which is used to store exceptions data. If, during a method execution an exception happens, the fact the exception occured and the exception data is stored in this structure. Here it is:

```
typedef struct CORBA_Environment {

  CORBA_exception_type _major;

  /* .... */

} CORBA_Environment;
```

The *_major* field indicates whether an exception occured or not during method invocation. It may have 3 values: *CORBA_NO_EXCEPTION*, *CORBA_USER_EXCEPTION* or *CORBA_SYSTEM_EXCEPTION*. Their signification is rather clear. Typically, good code should test after each method invocation if an exception occured with:

```
CORBA_Environment ev;

if (ev->_major != CORBA_NO_EXCEPTION)
  /* goooood :) */
else
  /* baaaaad :( */
```

We have also a nice set of functions to operate on these exceptions. We can throw exceptions with `CORBA_exception_set`. `CORBA_exception_id` returns a string to clearly identify the exception. `CORBA_exception_value` returns a pointer to the exception struct and `CORBA_exception_free` is used to reinitialize the ev struct after an exception. Here are their signatures:

```
extern CORBA_char *CORBA_exception_id (CORBA_Environment *ev);

extern void *CORBA_exception_value (CORBA_Environment *ev);

extern void CORBA_exception_free (CORBA_Environment *ev);

extern void CORBA_exception_set (CORBA_Environment *ev,
                                 CORBA_exception_type major,
                                 CORBA_char *exception_id,
                                 void *parameters);
```

And here is some sample code which uses the 3 first:

```
// here is the interface we wish to access
module FruitsBasket {
  interface Apple {
    exception eaten {
      string<40> why;
    };
    void EatMe () raises (eaten);
  };
};

#include <orb/orbit.h>
/* here, code to access the interface */
int main (int argc, char *argv)
{
  CORBA_Object obj;
  CORBA_Environment ev;
  CORBA_ORB orb;

  CORBA_exception_init (&ev);
  orb = CORBA_ORB_init (&ev);

  if (ev->_major != CORBA_NO_EXCEPTION) {
    fprintf (stderr,
             "Error: unable to initialise the ORB: %s\n",
             CORBA_exception_id (&ev));
    CORBA_exception_free (&ev);
    exit (1);
  }

  /* code to link to an Apple object */

  /* note that a switch /case statement would
   * have been cleaner but i am sick of all this code ...
   */
  FruitsBasket_Apple_EatMe (orb, &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
    if (ev->_major == CORBA_USER_EXCEPTION) {
      CORBA_char *buffer;
      buffer = CORBA_exception_id (&ev);
      if (strcmp (buffer, "FruitsBasket_Apple_eaten")) {
        fprintf (stderr,
                 "user exception: Apple allready eaten: %s\n",
                 (CORBA_exception_value (&ev))->why);
        exit (1);
      } else
```

```
        fprintf (stderr, "unknown exception raised !! Error !!\n");
    } else
      fprintf (stderr,
               "System exception: %s\n",
               CORBA_exception_id (&ev));
  exit (1);
  }

  return 0;
}
```

Finaly, to throw exceptions from your code, you must pass to `CORBA_exception_set` the correct parameters: the *CORBA_exception_type* must be either *CORBA_USER_EXCEPTION* or *CORBA_SYSTEM_EXCEPTION*. The CORBA_char must be the exception name #defined in the headers and the void * a pointer to an exception structure where fields are filled and where memory is allocated with *__alloc()*. Here is a small example using the same Apple object:

```
/* here, we are in the server and we wish to throw
 * the "eaten" exception with the correct "why"
 * reason.
 * note that our Apple Object exception will map
 * to the folowing C structure:
   typedef struct FruitsBasket_Apple_eaten {
     CORBA_char why[40];

     plus misc implementation specific members

   } FruitsBasket_Apple_eaten;
   #define ex_FruitsBasket_Apple_eaten <identifier>
   FruitsBasket_Apple_eaten *FruitsBasket_Apple_eatme__alloc ()
 */
  FruitsBasket_Apple_eaten *exception;

  exception = FruitsBasket_Apple_eaten__alloc ();
  strcpy (exception->why, "mmmm...");
  CORBA_exception_set (&ev, CORBA_USER_EXCEPTION,
                       ex_FruitsBasket_Apple_eaten,
                       (void *) exception);
```

Commented code

Now that exception handling has no secret for you anymore, you may look at this hello world code easily. We will use the folowing interface definition:

```
interface Echo {
  void echo (in string<40> str);
};
```

And here is the code:

```
main ()
{
  CORBA_ORB orb;
  CORBA_Environement ev;
  CORBA_Object obj;
  FILE *file;
  /* rather long string: should be enough for all our needs :) */
  char buffer[10000];
```

```
  char *ior;

  CORBA_exception_init(&ev);
  orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
    fprintf (stderr,
             "Error: exception during ORB init: %s\n",
             CORBA_exception_id(&ev));
    exit (1);
  }

  if (!file = fopen ("object.ior", "r")) {
    fprintf (stderr,
             "error: could not open serializing file\n");
    exit (2);
  }

  fgets (buffer, 10000, file);
  ior = g_strdup (buffer); /* gets the raw string */

  /* now, we have it finally !! */
  obj = CORBA_ORB_string_to_object (orb, ior, &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
    fprintf (stderr,
             "Error: could not get object reference: %s\n",
             CORBA_exception_id(&ev));
    exit (3);
  }
  fclose (file);

  Echo_echo (obj, ,&ev);
  if (ev->_major != CORBA_NO_EXCEPTION)
    fprintf (stderr,
             "Error: could not get method: %s\n",
             CORBA_exception_id(&ev));

  CORBA_exception_free(&ev);

   /* now, we could call many other objects... */

}
```

In fact, this code is nothing but code we already seen, with exception handling code added.

Chapter 3. Some usefull standardized interfaces

Introduction

In this chapter, we'll look at some of the many standardized CORBA service: the name services, the security services and the interface repository which is actually not really a service but which is nonetheless interesting.

The CosNaming module

Travelling in a naming graph

The CosNaming module is the module which implements the CORBA name service. It is used to associate to each object a human readable string which is then used to find the object reference upon need. This is exactly similar to the DNS which allows you to associate simple hierarchised names like www.mathieu.rezel.enst.fr to less usefull IP adresses: 137.194.8.56 .

Here, names like "GNOME.servers.Panel" are used to register object references. Typically, the server registers the name and clients ask the naming service whether the name was registered or not. If it was registered, clients get the server object reference from the naming service and send requests to the server object.

Here is some vocabulary specific to the CORBA naming service: names are bound to objects' references in a given context. Resolving a name is finding the object reference bound to the name within a given context. Contexts are CORBA objects: they may be bound like other objects and are used to create naming graphs.Each naming graph node is a naming context object and leafs are your own objects bindings. An Object name is the compound name made of the node names and the leaf name necessary to describe the path in the naming graph from the root to the object binding. For example, Gnome's naming graph is rather simple. We have: root -> GNOME -> servers

The Gnome Panel compound name is : root -> GNOME -> servers -> Panel

Here is the CosNaming module IDL:

```
module CosNaming {
  typedef string Istring;
  struct NameComponent {
    Istring id;
    Istring kind;
  };
  typedef sequence<NameComponent> Name;
  enum BindingType {
    nobject,
    ncontext
  };
  struct Binding {
    Name binding_name;
    BindingType binding_type;
  };
```

```
  interface NamingContext {
    void bind(in Name n, in Object obj)
             raises(NotFound,
                    CannotProceed,
                    InvalidName,
                    AlreadyBound);
    void rebind(in Name n, in Object obj)
             raises(NotFound,
                    CannotProceed,
                    InvalidName);

    void bind_context(in Name n, in NamingContext nc)
             raises(NotFound,
                    CannotProceed,
                    InvalidName,
                    AlreadyBound);
    void rebind_context(in Name n, in NamingContext nc)
             raises(NotFound,
                    CannotProceed,
                    InvalidName);

    Object resolve(in Name n)
             raises(NotFound,
                    CannotProceed,
                    InvalidName);
    void unbind(in Name n)
             raises(NotFound,
                    CannotProceed,
                    InvalidName);

    NamingContext new_context();
    NamingContext bind_new_context(in Name n)
             raises(NotFound,
                    AlreadyBound,
                    CannotProceed,
                    InvalidName);

    void destroy()
             raises(NotEmpty);
  };
};
```

I purposefully removed some definitions from this huge interface to make things clearer.

We have the binding operations which allow for the creation of the naming graph: `bind`, `rebind`, and `unbind` allow you to create name bindings within a given context (they allow you to create a naming graph leaf), `bind_context`, `rebind_context`, `new_context` and `bind_new_context` allow you to create naming contexts (they allow you to create the naming graph nodes). Lastly, resolve will help you to get the object reference from the naming graph with the compound name.

Here is some server code to bind an object to a name.

```
/* This is some code the Panel could
 * execute upon startup to register
 * against the name service
 */
main ()
{
  CORBA_Object orb;
  CORBA_Environment ev;
```

```
  CosNaming_NamingContext root;
  CosNaming_NameComponent name_component[3] = { {"GNOME", "subcontext"},
                                                {"Servers", "subcontext"}
                                                {"gnome_panel", "server"} };
  CosNaming_Name name = {3, 3, name_component, CORBA_FALSE};
  CORBA_Object panel;

  CORBA_exception_init (&ev);
  CORBA_ORB_init (&ev);

  root = CORBA_ORB_resolve_initial_service (orb,
                                            "NameService",
                                            &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
    fprintf (stderr,
             "Error: could not get name service: %s\n",
             CORBA_exception_id(&ev));
    exit (1);
  }

  /* here, we have some code to register
   * the server against the ORB and create
   * a valid object reference stored in panel.
   * this is for chapter 4: the POA
   */

  CosNaming_NamingContext_bind (root, &name, panel, &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
      fprintf (stderr,
               "Error: could register object: %s\n",
               CORBA_exception_id(&ev));
      exit (1);
  }
  CORBA_exception_free (&ev);
  return 0;
}
```

Here is some client code to resolve the server name

```
#include <corba.h>

int main (int argc, char *argv)
{
  CORBA_Object orb;
  CORBA_Environment ev;
  CosNaming_NamingContext *root;
  CosNaming_NameComponent name_component[3] = { {"GNOME", "subcontext"},
                                                {"Servers", "subcontext"}
                                                {"gnome_panel", "server"} };
  CosNaming_Name name = {3, 3, name_component, CORBA_FALSE};
  CORBA_Object panel;

  CORBA_exception_init (&ev);
  CORBA_ORB_init (&ev);

  root = CORBA_ORB_resolve_initial_service (orb,
                                            "NameService",
                                            &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
    fprintf (stderr,
             "Error: could not get name service: %s\n",
```

```
            CORBA_exception_id(&ev));
    exit (1);
  }

  panel = CosNaming_NamingContext_resolve (root,
                                          &name,
                                          &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
      fprintf (stderr,
               "Error: could resolve object: %s\n",
               CORBA_exception_id(&ev));
      exit (1);
  }

  /* there, the panel variable holds a CORBA_Object to the gnome panel
   * it is now possible to call some of the panel methods defined in
   * gnome-core/idl/gnome-panel.idl
   */

  CORBA_free (root);
  CORBA_exception_free (&ev);

  return 0;
}
```

The *CosNaming_NameComponent* and *CosNaming_Name* structures are straight C mappings of the corresponding IDL definitions in the CosNaming module.

Please, note that if the client does not know exactly the server name, he may use the `list` method to get a list of a context bindings.

Initialisation

One problem still hasn't been resolved: in these code examples, one implicitely admits that we have had before begening to use the NameService a NamingContext object. The problem is how to get one ? The only object we know we have access to is the ORB because there is a special initialisation function `CORBA_ORB_init`.

In fact, it happens that the ORB interface has a special method to get an initial refrence to a naming context and use its methods to resolve or bind names.

Here is the IDL

```
module CORBA {
  interface ORB {
    typedef string ObjectId;
    typedef sequence <ObjectId> ObjectIdList;
    ObjectIdList list_initial_services ();

    exception InvalidName {};
    Object resolve_initial_reference (in ObjectId identifier)
           raises (InvalidName);
  };
};
```

The ORB has two interesting methods: `list_initial_services` and `resolve_initial_reference`. The first one will return a list of the available services this ORB may resolve. As of CORBA 2.2, the folowing are standardized: "RootPoa", "POACurrent", "InterfaceRepository", "NameService", "TradingService", "SecurityCurrent" and "TransactionCurrent". This means that a call to a 2.2 ORB should return a list of these strings. The folowing code displays on screen the list of ORBit available services

```
#include <corba.h>

int main (int argc, char **argv)
{
  CORBA_ORB orb;
  CORBA_Environment ev;
  CORBA_sequence_ObjectId *seq;
  CORBA_unsigned_long i;

  CORBA_exception_init (&ev);
  orb = CORBA_ORB_init (&ev);

  seq = CORBA_ORB_list_initial_services (orb, &ev);

  for (i=0; i<seq->_lenght ;i++) {
    fprintf (stdout, "%s\n" ,(seq->_buffer)[i]);
  }

  return 0;
}
```

`The resolve_initial_service` method will return an object reference for the string passed as a parameter. This is how you get the object reference of the naming service. Before calling `CORBA_ORB_resolve_initial_service (orb, "NameService", &ev)`, you had no access to any object (except if you used ior string but this required some hacking as you saw earlier). Now, you have an object reference to a naming context and may register youself if you are a server or get a server object reference if you are a client using the object name

Note: The only problem with this code is that it does not work. ORBit has a NameService which may be enabled by launching the corresponding "orbit-name-server" but this name server cannot be resolved with this `CORBA_ORB_init` call. One needs to use the name-server IOR string which is displayed upon the name server launch. The recommended method to use it is to launch it with: "orbit-name-server >/tmp/name-service-ior &". This will store the IOR string in /tmp/name-service-ior which can then be read with code similar to the one we studied a little sooner.

The GNOME solves this problem quite elegantly with the `gnome_get_name_service` api function but this will be explained in this part's last chapter.

The Interface Repository interface

This interface has not been implemented by ORBit yet. Perhaps i could do it someday.

The Security module

This service has not been implemented by ORBit yet. I heard work was under the way.

Chapter 4. The POA interface

Introduction

The POA (Portable Object Adapter) is an object which is visible from the server object only. While clients hold object references on which they invoke methods, the ORB, the POA and the server skeleton all cooperate to decide to which function/code this request must be passed to.

Figure 4-1. The server structure

When the server is launched, it will first create a POA. Then, it will tell the POA about its servants (step 1) (servants are the IDL definitions' implementations we coded. In fact, the interface implementations). Then it will ask the POA for an object reference (step 2) which is then advertized to the outside world either with an ior string (step 3)(we allready saw this in chapter 2) or by binding this object reference to a name with the Naming Service.

Figure 4-2. The server registration

When the client is launched, it first asks the Naming Service about the object reference by using the resolve method (step 1). Then, it invocates the object's method with this object reference. This is possible because the object reference holds enough information to locate the server. This invocation is passed to the ORB (step 2) which seeks the correct server. Once the ORB has found the correct server (thanks to the object reference), it handles the request to the server.

Figure 4-3. A client request : 1

The server will then locate the POA which created the Object reference (thanks once more to the information contained in the object reference) and pass the request (step 2) to the POA. The request will finally be passed to the correct servant by the POA (step 3).

Figure 4-4. A client request : 2

The client does not really care about all this stuff which is handled by the ORB. The client only sees an object reference. The server only sees the POA: he does not care about anything else. We have allready seen how the client handles object references. Now is time to get hands dirty about the server stuff.

The POA mapping

The POA interface

The POA interface is rather big but simple:

```
module PortableServer {
  // lots of useless stuff here
  interface POA {

    // Object activation functions
    ObjectId activate_object (in Servant p_servant)
              raises ServantAlreadyActive, WrongPolicy);
    void activate_object_with_id (in ObjectId,
                                  in Servant p_servant)
              raises (ServantAlreadyActive, ObjectAlreadyActive,
                      WrongPolicy);
    void deactivate_object (in ObjectId id)
              raises (ObjectNotActive, WrongPolicy);

    // Reference creation functions
    Object create_reference (in CORBA::RepositoryId repid)
              raises (WrongPolicy);
    Object create_reference_with_id (in ObjectId id,
                                     in CORBA::RepositoryId repid)
              raises (WrongPolicy);

    // Identity Mapping functions
    ObjectId servant_to_id (in Servant p_servant)
              raises (ServantNotActive, WrongPolicy);
    Object servant_to_reference (in Servant p_servant)
              raises (ServantNotActive, WrongPolicy);
    Servant reference_to_servant (in Object reference)
              raises (ObjectNotActive, WrongAdapter, WrongPolicy);
    ObjectId reference_to_id (in Object reference)
              raises (WrongAdapter, WrongPolicy);
    Servant id_to_servant (in ObjectId oid)
              raises (ObjectNotActive, WrongPolicy);
    Object id_to_reference (in ObjectId oid)
              raises (ObjectNotActive, WrongPolicy);


  };
};
```

What you see here is an excerpt of the real POA interface: i removed all the exception definitions and many functions definitions which are not relevant here. The functions i made disappear will be discussed in the last part of this chapter.

Now, let's look at these functions and their use:

Table 4-1. The POA interface

Function Name	Function Use
activate_object	This function will register the object implementation represented by the Servant object against the POA and will return the ObjectId which can then be used with the `id_to_reference function to get the object reference associated with this interface implementation.`
activate_object_with_id	This function will register the object implementation represented by the Servant object against the POA and will associate it the id parameter as an *ObjectId. The ObjectId could be generated with a call to* `create_reference, then reference_to_id.`
deactivate_object	This function will unregister an object specifed by its *ObjectId (obtained through one of* `reference_to_id or servant_to_id function).`
create_reference	This function will create an Object reference from the object implementation (The Servant object).
cre-ate_reference_with_id	This function should be used only for well known objects whose ObjectId is well known...
servant_to_id	This function's signature makes things rather obvious. However, it is interesting to note that if the Servant was not activated, it will return a ServantNotActive exception which means that this function cannot be called before activating the object with one of `activate_object or activate_object_with_id.`
servant_to_reference	This function's signature makes things rather obvious. However, it is interesting to note that if the Servant was not activated, it will return a ServantNotActive exception which means that this function cannot be called before activating the object with one of `activate_object or activate_object_with_id.`
id_to_servant	This function's signature makes things rather obvious. However, it is interesting to note that if the object associated to the ObjectId was not activated, it will return an ObjectNotActivated exception which means that this function cannot be called before activating the object with one of `activate_object or activate_object_with_id.`
id_to_reference	This function's signature makes things rather obvious. However, it is interesting to note that if the object associated to the ObjectId was not activated, it will return an ObjectNotActivated exception which means that this function cannot be called before activating the object with one of `activate_object or activate_object_with_id.`

Function Name	Function Use
reference_to_servant	This function's signature makes things rather obvious. However, it is interesting to note that if the object associated to the object reference was not activated, it will return an ObjectNotActivated exception which means that this function cannot be called before activating the object with one of `activate_object or activate_object_with_id.`
reference_to_id	This function's signature makes things rather obvious. This function is different from all the other precedent functions: it will not throw any NotActivated exception which means that it may be used after a call to `create_reference to generate the correct ObjectId to use for the activate_object_with_id functions.`

Here is some sample code using these functions:

```
/* sample code using activate_object */
  ObjectId obj_id;
  CORBA_Object obj;
  obj_id = PortableServer_POA_activate_object (poa,
                                               servant,
                                               &ev);
  /* now, we can register this CORBA_Object
   * in the Naming Service or with an ior string
   */
  obj = PortableServer_POA_id_to_reference (poa,
                                            obj_id,
                                            &ev);

/* sample code using activate_object_with_id */
  CORBA_Object obj;
  /* note that PortableServer_string_to_id's use
   * is detailed a later. suffice it to say that
   * it returns an ObjectId
   */
  PortableServer_POA_activate_object_with_id (poa,
                                              PortableServer_string_to_id (
                                                      "my_small_servant",
                                                      &ev),
                                              servant,
                                              &ev);

  obj = Portable_Server_POA_servant_to_reference (poa,
                                                  servant,
                                                  &ev);
  /* now, we can register this CORBA_Object
   * in the Naming Service or with an ior string
   */
```

Servants

All these functions show how a servant is registered against the POA. Servants are language dependant. Their structure and their use is completely mapping dependant. Here is the C version of Servants.

The idea is to pass some initialisation function a data structure which holds the list of function pointers for the implementation. Such a data structure is called an EPV (entry point vector). The Servant data structure holds such an EPV.

Here is for example the data structure which would be generated for the folowing interface definition:

```
// idl interface
module FruitsBasket {
  interface Apple {
    void EatMe ();
    void ThrowMe ();
  };
};

typedef struct {
    void *_private; /* ORB specific data */
    void (*EatMe) (PortableServer_Servant servant,
                   CORBA_Environment * ev);
    void (*ThrowMe) (PortableServer_Servant servant,
                     CORBA_Environment * ev);
} POA_FruitsBasket_Apple__epv;

typedef struct {
    PortableServer_ServantBase__epv *_base_epv;
    POA_FruitsBasket_Apple__epv *FruitsBasket_Apple_epv;
} POA_FruitsBasket_Apple__vepv;

typedef struct {
    void *_private; /* ORB specific data */
    POA_FruitsBasket_Apple__vepv *vepv;
} POA_FruitsBasket_Apple;

extern void POA_FruitsBasket_Apple__init(POA_FruitsBasket_Apple servant,
                                         CORBA_Environment * ev);
extern void POA_FruitsBasket_Apple__fini(POA_FruitsBasket_Apple servant,
 CORBA_Environment * ev);
```

A servant is thus nothing more than a structure which holds an EPV specific to the object implementation and a generic EPV called the _base_epv. The specific EPV is rather self explanatory: function pointers are listed there, waiting to be filled. The base EPV was not shown there: it is used to hold some information specific to the servant itself (in fact, a servant may be responsible for more than one object implementation).

Now, here is the *PortableServer_ServantBase__epv* structure:

```
typedef struct PortableServer_ServantBase__epv {
  void *private;  /* ORB specific data */
  /* finalization routine called
   * to destroy any Servant data
   */
  void (*finalize) (PortableServer_Servant,
                    CORBA_Environment *);
  /* The POA used by this Servant */
  PortableServer_POA (*default_POA) (PortableServer_Servant,
                                     CORBA_Environment *);
} PortableServant_ServantBase__epv;
```

If the *finalize* and *default_poa* members are NULL when the Servant is passed to the `POA_FruitsBasket_Apple__init`, the ORB will initialize them with standard values. *default_POA* is defaulted to the "RootPOA", the POA returned by `CORBA_ORB_resolve_initial_reference` `(orb, "RootPOA", &ev)`. Otherwise, the ORB will assume the application initialized these fields correctly.

Typically, such a set of data structures would be initialized as follows:

```
void MyEatMe (PortableServer_Servant servant,
              CORBA_Environment * ev)
{
  /* here is your own code which
   * corresponds to the object implementation
   */
}
void MyThrowMe (PortableServer_Servant servant,
                CORBA_Environment * ev)
{
  /* here is your own code which
   * corresponds to the object implementation
   */
}

PortableServer_ServantBase__epv base_epv {
  NULL, /* ORB specific data */
  NULL, /* finalize method */
  NULL  /* default POA */
}

POA_FruitsBasket_Apple__epv Apple_epv {
  NULL,    /* ORB specific data */
  MyEatMe,
  MyThrowMe
};

POA_FruitsBasket_Apple__vepv Apple_vepv {
  &base_epv,
  &Apple_epv
};

POA_FruitsBasket_Apple Apple_servant = {
  NULL,
  &Apple_vpev
};
```

Then, the only thing left to do is to call `POA_FruitsBasket_Apple__init ( Apple_servant, &ev)`. Now, the Servant is registered against the POA but we still need to activate this object implementation, get a valid object reference back and register it against the Name Service. Here is the complete sample code:

```
#include <orb/orbit.h>
#include <ORBitservices/CosNaming.h>

void MyEatMe (PortableServer_Servant servant,
              CORBA_Environment * ev)
{
  /* here is your own code which
   * corresponds to the object
   * implementation
   */
}
void MyThrowMe (PortableServer_Servant servant,
                CORBA_Environment * ev)
{
  /* here is your own code which
```

```
   * corresponds to the object
   * implementation
   */
}

PortableServer_ServantBase__epv base_epv {
  NULL, /* ORB specific data */
  NULL, /* finalize method */
  NULL  /* default POA */
}

POA_FruitsBasket_Apple__epv Apple_epv {
  NULL,    /* ORB specific data */
  MyEatMe,
  MyThrowMe
};

POA_FruitsBasket_Apple__vepv Apple_vepv {
  &base_epv,
  &Apple_epv
};

POA_FruitsBasket_Apple Apple_servant = {
  NULL,
  &Apple_vpev
};

int main (int argc, char **argv)
{
  CORBA_Environment ev;
  CORBA_ORB orb;
  PortableServer_POA root_poa;
  ObjectId *obj_id;
  CORBA_Object obj;
  CosNaming_NamingContext *root_name;
  CosNaming_NameComponent name_component[1] = { {"MyApple", "server"} };
  CosNaming_Name name = {1, 1, name_component, CORBA_FALSE};


  CORBA_exception_init (&ev);
  orb = CORBA_ORB_init (&argc, argv,
                        "orbit-local-orb",
                        &ev);

  POA_FruitsBasket_Apple__init (orb,
                                Apple_servant,
                                &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
    fprintf (stderr,
             "Error: exception during Servant initialisation: %s\n",
             CORBA_exception_id(&ev));
    exit 1;
  }

  root_poa = CORBA_ORB_resolve_initial_references (orb,
                                                   "RootPOA",
                                                   &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
    fprintf (stderr,
             "Error: exception during RootPOA initialisation: %s\n",
             CORBA_exception_id(&ev));
    CORBA_exception_free (&ev);
    exit 1;
```

```
  }

  /* activate object */
  obj_id = PortableServer_POA_activate_object (root_poa,
                                               &Apple_servant,
                                               &ev);
  /* get an object reference */
  obj = PortableServer_POA_id_to_reference (poa,
                                            obj_id,
                                            &ev);

  /* register the object against the naming service */
  root_name = CORBA_ORB_resolve_initial_service (orb,
                                                 "NameService",
                                                 &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
    fprintf (stderr,
             "Error: could not get name service: %s\n",
             CORBA_exception_id(&ev));
    exit 1;
  }
  CosNaming_NamingContext_bind (root_amp,
                                &name,
                                obj,
                                &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
      fprintf (stderr,
               "Error: could not register object: %s\n",
               CORBA_exception_id(&ev));
      exit 1;
  }
  CORBA_free (root_poa);
  CORBA_exception_free (&ev);
  exit 0;
}
```

pfew !! This was tough !! ... However, you are just beginning to get into CORBA hard stuff. This code has simplifications over real things. Here, we assumed that the poa was already ready to run, and we used the RootPOA for simplification which is not a good idea if you have hundreds of objects...

More functions

Before getting into the real hard stuff, here is a small but useful API which is specific to the C mapping: it is useful when dealing with *ObjectId* structures. Their signatures are enough to understand their use so...

```
PortableServer_ObjectId   * PortableServer_string_to_ObjectId (CORBA_Char *string,
                                                               CORBA_Environment *ev);

CORBA_Char                * PortableServer_string_to_ObjectId (Porta-
bleServer_ObjectId *obj,
                                                               CORBA_Environment *ev);
```

More POA architecture

POAManagers

POAs are used to dispatch the requests to their servants. However, there must be one way to handle the load on POAs and servants because we may have huge amounts of objects making requests to one object at the same time. This is achieved through the use of POAManagers. Each POA is associated to a *POAManager*.

POAManagers receive requests and send them to their POAs depending on their running state. A POAManager may be in one of 4 states: *active*, *inactive*, *holding* or *discarding*. If in the active state, requests are send to POAs, in the holding state, requests are queued till the POAManager gets back to the active state and in the discarding state, request are simply thrown out.

Generally, there is no need to write your own implementation of a POAManager: you can use the POAManager of the RootPOA thanks to the folowing interface:

```
module PortableServer {
  interface POA {

    POA create_POA (in string adapter_name,
                    in POAManager a_POAManager,
                    in CORBA::PolicyList policies)
              raises (AdapterAllreadyExists,
                      InvalidPolicy);
    void destroy (in boolean etherealize_objects,
                  in boolean wait_for_completion);
    POA find_poa (in string adapter_name,
                  in POAManager a_POAManager
                  in CORBA::PolicyList ListPolicies);

    readonly attribute POA the_parent;
    readonly attribute string the_name;
    readonly attribute POAManager the_POAManager;
  };
};
```

with:

```
  PortableServer_POA root_poa;
  PortableServer_POAManager root_poa_manager;
  root_poa = CORBA_ORB_resolve_initial_references (orb,
                                                   "RootPOA",
                                                   &ev);
  root_poa_manager = PortableServer_POA_get_the_POAManager (root_poa,
                                                            &ev);

  PortableServer_POA_create_poa (root_poa, "my_own_poa",
                                 root_poa_manager,
                                 ..., &ev);
```

a POA is allways created as the child of another POA: the RootPOA is the root of the POA tree. It is possible to find a particuliar children provided you know its name with the `find_poa` method. The *the_parent* attribute holds the parent POA, and the *the_name* attribute holds the POA name assigned with the `create_poa` method. `destroy_poa` will destroy the poa passed as parameter but also all its chidren.

To hold your servants, it is thus a good idea to create a child POA of the RootPOA and use it as the *default_poa* member of the *_base_epv* of your servants. You must also make sure that the POAManager used for your POA (as i allready said, generally the RootPOA's own POAManager) is in the active state as POAManagers are initialized in the holding state. Sample code for this is presented in the chapter's last section.

POA Policies

To each POA is also associated a set of Policies. If a NULL parameter is passed, the default policies are used. Here is the list of the policy types and their possible values. Default values are *like this*.

Table 4-2. POA Policies

Policy type	Possible values	Policy use
ThreadPolicy	*ORB_CTRL_MODEL*, SINGLE_THREAD_MODEL	In SINGLE_THREAD, only one poa thread may exist.
LifespanPolicy	*TRANSIENT*, PERSISTANT	In TRANSIENT mode, object implementations will disapear with their POA. Otherwise, a new POA is created when necessary.
ObjectIdUniqueness	*UNIQUE_ID*, MULTIPLE_ID	Each object wihtin a given POA has a unique ObjectId.
IdAssignmentPolicy	USER_ID, *SYSTEM_ID*	ObjectId are either given by the user or the system. ie: you may use only one of the 2 functions for a given POA: activate_object or activate_object_with_id.
ServantRetentionPolicy	*RETAIN*, NON_RETAIN	In NON RETAIN mode, a request to a servant allready processing another request will throw an ObjectNotExist exception.
RequestProcessingPolicy	*USE_ACTIVE_OBJECT_MAP_ONLY*, USE_DEFAULT_SERVANT, USE_SERVANT_MANAGER	Any request to a servant not found in the POA map will result in either a call to a default servant, an ObjectNotExist exception or a request to the Servant Manager to create the requested servant.
ImplicitActivationPolicy	IMPLICIT_ACTIVATION, *NO_IMPLICIT_ACTIVATION*	If NO_IMPLICIT_ACTIVATION is used, the programmer will need to activate all the created servants.

These policy objects should be created with the folowing calls which are part of the POA interface.

```
module PortableServer {
  interface POA {
```

```
    ThreadPolicy
       create_thread_policy
         (in ThreadPolicyValue value);
    LifespanPolicy
       create_lifespan_policy
         (in LifespanPolicyValue value);
    IdUniquenessPolicy
       create_id_uniqueness_policy
         (in IdUniquenessPolicyValue value);
    IdAssignementPolicy
       create_id_assignment_policy
         (in IdAssignmentPolicyValue value);
    ImplicitActivationPolicy
       create_implicit_activation_policy
         (in ImplicitActivationPolicyValue value);
    ServantRetentionPolicy
       create_servant_retention_policy
         (in ServantRetentionPolicyValue value);
    ProcessingPolicy
       create_processing_policy
         (in ProcessingPolicyValue value);
  };
};
```

Then these policies should be put together in a `CORBA::PolicyList` and passed to the `create_poa` function to create the poa with the correct policies. This set of policies gives the programmer great power on server programming as they very precisely describe the server behaviour and allow the programmer to control the server load. Here is some sample code to create a new POA using these policies. Note that when a policy type is not present in the PolicyList, the policy type is defaulted to the default value stated in the precedent table.

```
// idl Policy Object
module CORBA {
  typedef unsigned long PolicyType;
  interface Policy
  {
    readonly attribute PolicyType policy_type;
    Policy copy();
    void destroy();
  };

  typedef sequence <Policy> PolicyList;
};

/* C code building a PolicyList object */

  CORBA_Policy policies[2];
  CORBA_PolicyList list = {2, 2, policies};

  policies[1] = PortableServer_POA_create_Id_Uniqueness_Policy (MULTIPLE_ID,
                                                                  &ev);
  policies[2] = PortableServer_POA_create_Id_Assignment_Policy (USER_ID,
                                                                  &ev);
  PortableServer_POA_create_poa (poa,
                                 "my_poa",
                                 poa_manager,
                                 list,
                                 &ev);
  CORBA_Policy_destroy (policies[1]);
  CORBA_Policy_destroy (policies[2]);
```

A nice trick/hack

It is often necessary to pass to your implementation functions some common parameters. The idea is that all your functions receive as first parameter a pointer to a *PortableServer_Servant* structure. Look at the folowing code:

```
typedef struct {
  PortableServer_Servant servant;
  long my_field;
  long my_secon_field;
  void *pointer_to_my_structure;
} my_POA_FruitsBasket_Apple

my_POA_FruitsBasket_Apple Apple;
```

It happens that *&Apple* is a pointer to the Apple structure but also to the first field of the structure as C assures that *(&Apple == &Apple.servant)* is true. This means that if you register your servant with a pointer to Apple, this pointer will be passed to all the implementation functions. This is possible because a *my_POA_FruitsBasket_Apple* pointer is also a pointer to a *PortableServer_Servant* structure.

A real server : finally !!

Everything by hand

Once more, we'll code the `FruitsBasket::Apple` server. Without preliminaries, here is the code: every piece was allready studied separately.

```
#include <orb/orbit.h>
#include <ORBitservices/CosNaming.h>

void MyEatMe (PortableServer_Servant servant,
              CORBA_Environment * ev)
{
  /* here is your own code which
   * corresponds to the object
   * implementation
   */
}
void MyThrowMe (PortableServer_Servant servant,
                CORBA_Environment * ev)
{
  /* here is your own code which
   * corresponds to the object
   * implementation
   */
}

PortableServer_ServantBase__epv base_epv {
  NULL, /* ORB specific data */
  NULL, /* finalize method */
  NULL  /* default POA */
}

POA_FruitsBasket_Apple__epv Apple_epv {
  NULL,    /* ORB specific data */
```

```
  MyEatMe,
  MyThrowMe
};

POA_FruitsBasket_Apple__vepv Apple_vepv {
  &base_epv,
  &Apple_epv
};

POA_FruitsBasket_Apple Apple_servant = {
  NULL,
  &Apple_vpev
};

int main (int argc, char **argv)
{
  CORBA_Environment ev;
  CORBA_ORB orb;
  PortableServer_POA poa;
  /* we use the default policies */
  CORBA_PolicyList policy_list = {0, 0, NULL};
  PortableServer_POA root_poa;
  PortableServer_POAManager root_poa_manager;
  ObjectId *obj_id;
  CORBA_Object obj;
  CosNaming_NamingContext *root_name;
  CosNaming_NameComponent name_component[1] = { {"MyApple", "server"} };
  CosNaming_Name name = {1, 1, name_component, CORBA_FALSE};

  /* initialize the orb and the servant
  CORBA_exception_init (&ev);
  orb = CORBA_ORB_init (&argc,
                        argv,
                        "orbit-local-orb",
                        &ev);

  POA_FruitsBasket_Apple__init (orb,
                                Apple_servant,
                                &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
    fprintf (stderr,
             "Error: exception during Servant initialisation: %s\n",
             CORBA_exception_id(&ev));
    exit (1);
  }

  /* initialize the poa */
  root_poa = CORBA_ORB_resolve_initial_references (orb,
                                                   "RootPOA",
                                                   &ev);
  root_poa_manager = PortableServer_POA_get_the_POAManager (root_poa,
                                                            &ev);
  PortableServer_POAManager_activate (root_poa_manager,
                                      &ev);
  PortableServer_POA_create_poa (root_poa,
                                 "my_own_poa",
                                 root_poa_manager,
                                 policy_list,
                                 &ev);
  poa = PortabelServer_POA_find_poa (root_poa,
                                     "my_own_poa",
                                     &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
```

```
    fprintf (stderr,
             "Error: exception during POA initialisation: %s\n",
             CORBA_exception_id(&ev));
    CORBA_exception_free (&ev);
    exit (1);
  }

  /* activate the object */
  obj_id = PortableServer_POA_activate_object (poa,
                                               &Apple_servant,
                                               &ev);
  /* get an object reference */
  obj = PortableServer_POA_id_to_reference (poa,
                                            obj_id,
                                            &ev);

  /* register the object against the naming service */
  root_name = CORBA_ORB_resolve_initial_service (orb,
                                                 "NameService",
                                                 &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
    fprintf (stderr,
             "Error: could not get name service: %s\n",
             CORBA_exception_id(&ev));
    exit (1);
  }
  CosNaming_NamingContext_bind (root_amp,
                                &name,
                                obj,
                                &ev);
  if (ev->_major != CORBA_NO_EXCEPTION) {
      fprintf (stderr,
               "Error: could register object: %s\n",
               CORBA_exception_id(&ev));
      exit (1);
  }
  CORBA_free (root_poa);
  CORBA_exception_free (&ev);
  /* pfew !! here we are !! */

  return 0;
}
```

Orbit’s skeletons

The code we just seen is rather generic: obviously, almost nothing is specific to the `FruitsBasket::Apple` interface except for the function implementation (this is however quite normal). Most of this code may be used in any other server with any other interface.

That is why ORBit provides an easy way to automaticaly generate the code I just spent a chapter explaining you. The –skeleton-impl option will generate a C file which contains sample server code. Here is for example the code generated for this `FruitsBasket::Apple` interface.

```
#include "apple.h"

/*** App-specific servant structures ***/
typedef struct {
```

```
   POA_FruitsBasket_Apple servant;
   PortableServer_POA poa;

} impl_POA_FruitsBasket_Apple;

/*** Implementation stub prototypes ***/
static void impl_FruitsBasket_Apple__destroy(impl_POA_FruitsBasket_Apple * servant,
     CORBA_Environment * ev);
static void
 impl_FruitsBasket_Apple_EatMe(impl_POA_FruitsBasket_Apple * servant,
       CORBA_Environment * ev);

static void
 impl_FruitsBasket_Apple_ThrowMe(impl_POA_FruitsBasket_Apple * servant,
 CORBA_Environment * ev);

/*** epv structures ***/
static PortableServer_ServantBase__epv impl_FruitsBasket_Apple_base_epv =
{
   NULL, /* _private data */
   NULL, /* finalize routine */
   NULL, /* default_POA routine */
};
static POA_FruitsBasket_Apple__epv impl_FruitsBasket_Apple_epv =
{
   NULL, /* _private */
   (gpointer) & impl_FruitsBasket_Apple_EatMe,

   (gpointer) & impl_FruitsBasket_Apple_ThrowMe,

};

/*** vepv structures ***/
static POA_FruitsBasket_Apple__vepv impl_FruitsBasket_Apple_vepv =
{
   &impl_FruitsBasket_Apple_base_epv,
   &impl_FruitsBasket_Apple_epv,
};

/*** Stub implementations ***/
static FruitsBasket_Apple
impl_FruitsBasket_Apple__create(PortableServer_POA poa,
                                CORBA_Environment * ev)
{
   FruitsBasket_Apple retval;
   impl_POA_FruitsBasket_Apple *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_FruitsBasket_Apple, 1);
   newservant->servant.vepv = &impl_FruitsBasket_Apple_vepv;
   newservant->poa = poa;
   POA_FruitsBasket_Apple__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

   return retval;
}

static void
impl_FruitsBasket_Apple__destroy(impl_POA_FruitsBasket_Apple * servant,
 CORBA_Environment * ev)
{
```

```
   PortableServer_ObjectId *objid;

   objid = PortableServer_POA_servant_to_id(servant->poa, servant, ev);
   PortableServer_POA_deactivate_object(servant->poa, objid, ev);
   CORBA_free(objid);

   POA_FruitsBasket_Apple__fini((PortableServer_Servant) servant, ev);
   g_free(servant);
}

static void
impl_FruitsBasket_Apple_EatMe(impl_POA_FruitsBasket_Apple * servant,
      CORBA_Environment * ev)
{
}

static void
impl_FruitsBasket_Apple_ThrowMe(impl_POA_FruitsBasket_Apple * servant,
CORBA_Environment * ev)
{
}
```

The only thing left to do is to implement a `main` function that calls the `create` function and initialize a few other things...

Note that this code uses the trick presented in the precedent section to associate *PortableServer_POA poa* to the servant by the mean of the folowing structure

```
typedef struct {
   POA_FruitsBasket_Apple servant;
   PortableServer_POA poa;

} impl_POA_FruitsBasket_Apple;
```

Chapter 5. CORBA in The GNOME

Introduction

CORBA is present everywhere in the GNOME: the File Manager, the Panel, the Window Manager and Bonobo all use CORBA to achieve clean interfaces between applications. This chapter aims at giving you an idea of where you will find CORBA stuff in the GNOME, how it solves some problems of ORBit and gives a few hints about some nice hacks used everywhere in Bonobo.

The Gnorba library

Introduction

gnorba is mainly a wrapper around two corba functions (`CORBA_ORB_init` and `CORBA_ORB_resolve_initial_services`) and implements an Activation Directory. The Activation Directory is a filesystem which holds a list of the available CORBA servers on a machine which are not necessarily running.

Initialisation

Gnome applications usually have a GTK graphic interface. GTK uses a special event loop to handle all the X server events and all its own events. ORBit uses also its own event loop. This means that graphical applications need a way to have these two loops run together. This is basically why some special initialisation needs to be done.

The `gnome_CORBA_init` function is used and is simply a wrapper around the `CORBA_ORB_init` and `gnome_init` functions. It will integrate ORBit event loop into GTK's own event loop. The folowing code will show its use:

```
#include <orb/orbit.h>
#include <libgnorba/gnorba.h>
#include <gnome.h>

int main (int argc, char **argv)
{
  CORBA_ORB orb;
  CORBA_Environment ev;

  CORBA_exception_init (&ev);
  orb = gnome_CORBA_init ("a simple gnorba test prgm",
                          "v 1.0",
                          &argc,
                          argv,
                          &ev);

  return 0;
}
```

Naming Services

We allready seen why the Name Service of ORBit has a few problems: 1) one needs to launch the orbit-name-server by hand and 2) the Name Service does not work with `CORBA_ORB_resolve_initial_reference`.

The GNOME implements the `gnome_name_service_get` function which will return a `CORBA::Object` representing the Root NamingContext as it should be returned by `CORBA_ORB_resolve_initial_refrence`. Its use is staightforward:

```
#include <orb/orbit.h>
#include <libgnorba/gnorba.h>
#include <gnome.h>

int main (int argc, char **argv)
{
  CORBA_Object name_server;
  CORBA_Environment ev;

  CORBA_exception_init (&ev);
  orb = gnome_CORBA_init ("a simple gnorba test prgm",
                          "v 1.0",
                          &argc,
                          argv,
                          &ev);


  name_server = gnome_name_service_get ();
  return 0;
}
```

This function will first check if a name server was launched then it will lauch it if not allready and it will return the Object's Object reference. To do so, it stores information on the Root Window of the X window system which is why this function was not added to ORBit code: it will not work outside of a graphical X system.

The GOAD

Introduction

The GOAD allows you to list the CORBA servers ready to run and to eventually lauch them. For example, the Help system of the gnome is listed in the GOAD repository and may be launched by activating it.

This is necessary because CORBA provides a mean to use active servers but, to be active, they need to be fed in memory which may lead to memory shortage. There are ways in CORBA to get around this problem by using servant managers but on the whole, the GOAD is a simpler solution.

It is possible to look at this repository without having to code anything: the "goad-browser" program is distributed with libgnorba and will list the servers and allow you to activate them.

Using the GOAD

To use the GOAD, one has access to a simple yet powerfull api:

Table 5-1. The GOAD api

Function name	Function use
goad_server_list_get	This function will return a *GoadServerList * which lists all the servers registered against the goad.*
goad_server_list_free	This will free the GoadServerList allocated by *goad_server_list_get.*
goad_server_activate	This function will return the *CORBA_Object corresponding to the GoadServer passed as a parameter.*
goad_server_activate_with_id	This function will ultimately call *goad_server_activate but it needs other parameters.*
goad_server_activate_with_repo_id	This function will ultimately call *goad_server_activate but it needs other parameters.*

These functions play with some well known structures: *GoadServer* and *GoadServerList*.

```
typedef struct {
          GoadServerType type;
          /* only GOAD_ACTIVATE_NEW_ONLY
             currently parsed in */
          GoadActivationFlags flags;
          char     **repo_id;
          char     *server_id;
          char     *description;

          /*
           * Executable/shlib path, relayer IOR,
           * whatever. This field may disappear
           * at any time. You have been warned ;-)
           */
          char     *location_info;
  } GoadServer;

typedef struct {
    GoadServer *list;
    GHashTable *by_goad_id;
  } GoadServerList;

typedef enum {
          GOAD_SERVER_SHLIB = 1,
          GOAD_SERVER_EXE = 2,
          GOAD_SERVER_RELAY = 3,
          GOAD_SERVER_FACTORY = 4
  } GoadServerType;

typedef enum {
          /* these two are mutually exclusive */
          /* prefer shlib activation */
          GOAD_ACTIVATE_SHLIB = 1 « 0,
          /* prefer remote activation */
          GOAD_ACTIVATE_REMOTE = 1 « 1,

          /* these two are mutually exclusive */
          /* Only do lookup in name
           * service for currently running
           * version. */
          GOAD_ACTIVATE_EXISTING_ONLY = 1 « 2,
          /* No lookup in name service. */
```

```
        GOAD_ACTIVATE_NEW_ONLY = 1 « 3,
        /* Just make sure that the object is running */
        GOAD_ACTIVATE_ASYNC = 1 « 4
} GoadActivationFlags;
```

Here is sample code using these...

```
/* this function will return a clist widget
 * which holds a list of the available servers
 */
GtkWidget *get_list (void)
{
  GoadServer *tmp;
  char *titles[2] = {"Description", "Activated ?"};
  GtkWidget *clist;
  GtkWidget *popup;

  list = goad_server_list_get ();
  tmp = list->list;

  clist = gtk_clist_new_with_titles (2, titles);
  gtk_signal_connect (GTK_OBJECT(clist),
                      "select_row",
                      select_row_cb,
                      NULL);
  gtk_clist_set_column_width (GTK_CLIST(clist), 0, 250);
  gtk_clist_set_column_width (GTK_CLIST(clist), 1, 30);
  gtk_clist_set_selection_mode (GTK_CLIST(clist),
                                GTK_SELECTION_EXTENDED);
  gtk_widget_show (clist);

  for (tmp=tmp; tmp->repo_id; ++tmp) {
    gchar *table[2] = {tmp->description, is_active(tmp)?"yes":"no"};
    gtk_clist_append(GTK_CLIST(clist), table);
  }

  return (clist);
}
```

The Right Thing to do now is to browse the goad-browser code: gnome-libs/gnorba/goad-browser.c .

Adding servers to the GOAD

To add servers to the GOAD, one should use .gnorba files. Upon the GOAD startup during the GNOME session initialization, the GOAD reads a special directory (/etc/CORBA or /usr/etc/CORBA or...). Each file in this directory may contain one or more server definition. Here is one such file:

```
[gnome_panel]
type=exe
repo_id=IDL:GNOME/Panel:1.0
description=GNOME Panel
location_info=panel
```

The mapping of these informations to the GoadServer structure fields is straightforward. The first *[gnome_panel]* is the name the server will be registered against in the name server. The second *type=exe* is used internally by the GOAD to determine how to launch the server.The last *location_info=panel* is used to find the corresponding implementation of the server. Here, the *gnome_panel* was implemented in the *panel* executable.

The GNOME Desktop

Introduction

The GNOME Desktop uses a number of CORBA interfaces. These interface definitions may be found in gnome-libs/idl and gnome-core/idl. They are more thouroughfully listed in Appendix C.

Table 5-2. Gnome CORBA interfaces

Interface filename	Interface use
desktop-viewer.idl	A set of interfaces which define a text viewer. Any text viewer implementing these interfaces may be used to display text by the file manager for example.
desktop-editor.idl	A set of interfaces which define a text editor. This interface inherits the desktop-viewer interface. Any text editor implementing these interfaces may be used to display and edit text by the file manager for example.
Table.idl	A simple set of interfaces to define arrays. I never found any use to this.
gnome-unknown.idl	This interface defines the COM Unknown interface. It is used as a basis of the bonobo objects. This will be discussed later in the part on the bonobo.
GnomeObject.idl	It is used as a basis of the bonobo objects. This will be discussed later in the part on the bonobo.
gnome-factory.idl	A simple interface to a factory object.
gnome-panel.idl	The interface to the Gnome Panel. This interface is really complex and big. We'll see this later.
help-browser.idl	A simple interface to an html browser to display the help documentation. This interface is implemented by the help-browser.

All of these interfaces are REALLY simple except for the Panel interfaces. The Panel's complexity is interesting because it solves some problems common to Bonobo. The 2 main problems are: 1) how to display in an application a widget build in another application (running in another adress space) and 2) how to create object instances.

The GNOME provides a C api to wrap the CORBA calls into more developer-friendly C calls.

The GNOME::Factory interface

The Factory interface is used to provide a set of construct mechanisms. In OOP, Objects are often created by a special function. This function is called a constructor. For example, it would be good to have a constructor for our Panel Applets. This is generally what is beeing done and the Factory interface provides an abstraction layer on top of the construct mechanism.

```
// The factory interface.

module GNOME {
typedef sequence<string> stringlist;
  interface GenericFactory {
    exception CannotActivate { };
    boolean supports(in string obj_goad_id);

    Object create_object(in string goad_id, in stringlist params)
           raises(CannotActivate);
  };
};
```

An object should wrap its constructor function behind am implementation of `GNOME::GenericFactory::create_object()`. Then, it is possible to ask the factory to generate a `CORBA::Object` reference for the newly created object. Here is some sample code which sets up a factory for a `FruitsBasket::Apple` object. This code was created with the –skeleton-impl parameter of orbit-idl. It allows you to very easily implement a factory as the `*__create` functions are allready constructors.

```
// idl interface
#include <gnome-factory.idl>
module FruitsBasket {

  interface AppleFactory : GNOME::GenericFactory {};

  interface Apple {
    void EatMe ();
  };
};

/********** C Factory skeleton implementation ***********/
#include "apple.h"

/*** App-specific servant structures ***/

typedef struct {
   POA_FruitsBasket_AppleFactory servant;
   PortableServer_POA poa;
} impl_POA_FruitsBasket_AppleFactory;

typedef struct {
   POA_FruitsBasket_Apple servant;
   PortableServer_POA poa;

} impl_POA_FruitsBasket_Apple;

/*** Implementation stub prototypes ***/

static void
impl_FruitsBasket_AppleFactory__destroy(impl_POA_FruitsBasket_AppleFactory
                                        * servant,
CORBA_Environment * ev);
static CORBA_boolean
impl_FruitsBasket_AppleFactory_supports(impl_POA_FruitsBasket_AppleFactory
                                        * servant,
                                        CORBA_char * obj_goad_id,
                                        CORBA_Environment * ev);
```

```
static CORBA_Object
impl_FruitsBasket_AppleFactory_create_object(impl_POA_FruitsBasket_AppleFactory
                                             * servant,
                                             CORBA_char * goad_id,
                                             GNOME_stringlist * params,
                                             CORBA_Environment * ev);

static void impl_FruitsBasket_Apple__destroy(impl_POA_FruitsBasket_Apple * servant,
                                             CORBA_Environment * ev);
static void impl_FruitsBasket_Apple_EatMe(impl_POA_FruitsBasket_Apple * servant,
                                          CORBA_Environment * ev);

/*** epv structures ***/

static PortableServer_ServantBase__epv
impl_FruitsBasket_AppleFactory_base_epv =
{
   NULL, /* _private data */
   NULL, /* finalize routine */
   NULL, /* default_POA routine */
};
static POA_FruitsBasket_AppleFactory__epv
       impl_FruitsBasket_AppleFactory_epv =
{
   NULL, /* _private */
};
static POA_GNOME_GenericFactory__epv
       impl_FruitsBasket_AppleFactory_GNOME_GenericFactory_epv =
{
   NULL, /* _private */
   (gpointer) & impl_FruitsBasket_AppleFactory_supports,
   (gpointer) & impl_FruitsBasket_AppleFactory_create_object,
};
static PortableServer_ServantBase__epv
       impl_FruitsBasket_Apple_base_epv =
{
   NULL, /* _private data */
   NULL, /* finalize routine */
   NULL, /* default_POA routine */
};
static POA_FruitsBasket_Apple__epv
       impl_FruitsBasket_Apple_epv =
{
   NULL, /* _private */
   (gpointer) & impl_FruitsBasket_Apple_EatMe,

};

/*** vepv structures ***/

static POA_FruitsBasket_AppleFactory__vepv
impl_FruitsBasket_AppleFactory_vepv =
{
   &impl_FruitsBasket_AppleFactory_base_epv,
   &impl_FruitsBasket_AppleFactory_GNOME_GenericFactory_epv,
   &impl_FruitsBasket_AppleFactory_epv,
};
static POA_FruitsBasket_Apple__vepv
       impl_FruitsBasket_Apple_vepv =
{
   &impl_FruitsBasket_Apple_base_epv,
   &impl_FruitsBasket_Apple_epv,
};
```

```
/*** Stub implementations ***/


/* This is the Factory constructor.
 * It must be called from the main function so that
 * factory  can create new Apple objects
 */
static FruitsBasket_AppleFactory
       impl_FruitsBasket_AppleFactory__create(PortableServer_POA poa,
                                              CORBA_Environment * ev)
{
   FruitsBasket_AppleFactory retval;
   impl_POA_FruitsBasket_AppleFactory *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_FruitsBasket_AppleFactory, 1);
   newservant->servant.vepv = &impl_FruitsBasket_AppleFactory_vepv;
   newservant->poa = poa;
   POA_FruitsBasket_AppleFactory__init((PortableServer_Servant) newser-
vant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

   return retval;
}

static void
impl_FruitsBasket_AppleFactory__destroy(impl_POA_FruitsBasket_AppleFactory *
                                        servant, CORBA_Environment * ev)
{
   PortableServer_ObjectId *objid;

   objid = PortableServer_POA_servant_to_id(servant->poa, servant, ev);
   PortableServer_POA_deactivate_object(servant->poa, objid, ev);
   CORBA_free(objid);

   POA_FruitsBasket_AppleFactory__fini((PortableServer_Servant) servant, ev);
   g_free(servant);
}

static CORBA_boolean
impl_FruitsBasket_AppleFactory_supports(impl_POA_FruitsBasket_AppleFactory * servant,
                                        CORBA_char * obj_goad_id,
                                        CORBA_Environment * ev)
{
   CORBA_boolean retval;

   return retval;
}

static CORBA_Object
impl_FruitsBasket_AppleFactory_create_object(impl_POA_FruitsBasket_AppleFactory * servant,
                                             CORBA_char * goad_id,
                                             GNOME_stringlist * params,
                                             CORBA_Environment * ev)
{
   CORBA_Object retval;
     /* here, the only things to do are to create or reuse
      * a poa and a poamanager and register with them
      * the Apple thanks to the
      * impl_FruitsBasket_Apple__create
      */
```

```
   retval = impl_FruitsBasket_Apple__create (servant->poa, ev);

   return retval;
}

/* This is the Apple constructor which just needs to be
 * called by the create_object function of the factory
 * instead of beeing directly called by the main function
 */

static FruitsBasket_Apple
impl_FruitsBasket_Apple__create(PortableServer_POA poa,
                                CORBA_Environment * ev)
{
   FruitsBasket_Apple retval;
   impl_POA_FruitsBasket_Apple *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_FruitsBasket_Apple, 1);
   newservant->servant.vepv = &impl_FruitsBasket_Apple_vepv;
   newservant->poa = poa;
   POA_FruitsBasket_Apple__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

   return retval;
}

static void
impl_FruitsBasket_Apple__destroy(impl_POA_FruitsBasket_Apple * servant,
                                 CORBA_Environment * ev)
{
   PortableServer_ObjectId *objid;

   objid = PortableServer_POA_servant_to_id(servant->poa, servant, ev);
   PortableServer_POA_deactivate_object(servant->poa, objid, ev);
   CORBA_free(objid);

   POA_FruitsBasket_Apple__fini((PortableServer_Servant) servant, ev);
   g_free(servant);
}

static void
impl_FruitsBasket_Apple_EatMe(impl_POA_FruitsBasket_Apple * servant,
                              CORBA_Environment * ev)
{
}

void main (int argc, char **argv)
{
  CORBA_Environment ev;
  CORBA_ORB orb;
  CORBA_Object root_poa;
  PortableServer_POAManager root_poa_manager;
  CosNaming_NamingContext name_server;
  CORBA_Object obj;

  CORBA_exception_init (&ev);
  orb = gnome_CORBA_init ("An Apple",
                          "1.0",
                          &argc,
                          argv,
```

```
                        GNORBA_INIT_SERVER_FUNC,
                        &ev);

  root_poa = CORBA_ORB_resolve_initial_references (orb,
                                                   "RootPOA",
                                                   &ev);

  root_poa_manager = PortableServer_POA__get_the_POAManager (
                     (PortableServer_POA) root_poa, &ev);
  PortableServer_POAManager_activate (root_poa_manager, &ev);

  /* this function will create all the other
   * needed interfaces
   */
  obj = impl_FruitsBasket_AppleFactory__create ((PortableServer_POA) root_poa,
                                                &ev);
  /* register against the GOAD */
  name_server = gnome_name_service_get ();
  /* This is really important: make sure you use these
     2 functions when registering */
  goad_server_register (name_server, obj,
                        goad_server_activation_id(),
                        "server",
                        &ev);
  /* finished !! */
  CORBA_exception_free (&ev);
  gtk_main ();
  return 0;
}
```

Once you have a `CORBA::Object` allready implemented, it is generally a matter of minutes to add factory support to it: all the things which were originally done in the main function are now done in the implementation of `GNOME::GenericFactory::create_object` while a `GNOME::GenericFactory` object is created in the main function instead of the original object.

The only difficult thing is to use `goad_server_register` to register in the name service and `goad_server_activation_id` to get the proper registration name.

If you had the folowing apple.gnorba file before adding Factory support, you should now use a newer apple.gnorba file.

```
# old .gnorba file
[Apple]
type=exe
repo_id=IDL:GNOME/Apple:1.0
description=A nice Apple
location_info=apple

#new .gnorba file
[Apple_factory]
type=exe
repo_id=IDL:GNOME/GenericFactory:1.0
description=internal
location_info=apple

[Apple]
type=factory
repo_id=IDL:GNOME/Apple:1.0
description=A nice Apple
location_info=Apple_factory
```

To understand the subtle differences between the first version and the second one, one must look at the GOAD code.

In the first case, if we called `goad_server_activate` on the Apple GoadServer, the GOAD would call `goad_server_activate_exe`, an internal function of the GOAD which does a fork/exec to launch the program as stated by the location_info. Here, the apple program would instantiate our object and a servant for it and would register it in the name service. It would be possible for the GOAD to sit in a loop to wait for the new CORBA::Object to register but instead, it uses a pipe to read the IOR string outputed by our client on stdout (this is more efficient. dixit Eliot Lee, the code author).

In the second case, if we called `goad_server_activate` on the Apple GoadServer, the GOAD would call `goad_server_activate_factory`, another internal function of the GOAD. This function will first make sure the factory is started by calling `goad_server_activate_with_id` on the Apple location_info field. This will end by calling `goad_server_activate_exe` on Apple_factory if the factory is not allready running. Then, the `goad_server_activate_factory` function calls the create_object method of the factory object returned by `goad_server_activate_with_id`.

I strongly suggest you to look at gnome-libs/libgnorba/goad.c to get convinced of this.

On the whole, using factories is really nice and not much work: it is sometimes necessary and offers many advantages. The most difficult thing is to set up the .gnorba file to make sure the scenario detailed just a few lines ago will work. It should be noted that for example, applets have some C api calls to ease the use of factories: `applet_factory_new` is one of these.

The GNOME::Panel interface

The GNOME Panel has three interfaces: `GNOME::Panel`, `GNOME::PanelSpot` and `GNOME::Applet`. `GNOME::Panel` and `PanelSpot` must be implemented by the Panel itself while `GNOME::Applet` should be implemented by the Applets themselves.

The Panel interface is defined as follows:

```
module GNOME {
  struct Color {
    unsigned short red, green, blue;
  };
  typedef short AppletId;
  typedef short PanelId;

  interface Panel {
    enum OrientType {
      ORIENT_UP,
      ORIENT_DOWN,
      ORIENT_LEFT,
      ORIENT_RIGHT
    };
    enum BackType {
      BACK_NONE,
      BACK_COLOR,
      BACK_PIXMAP
    };
    union BackInfoType switch(BackType) {
      case BACK_COLOR: Color c;
      case BACK_PIXMAP: string pmap;
    };
```

```
    enum SizeType {
      SIZE_TINY,
      SIZE_STANDARD,
      SIZE_LARGE,
      SIZE_HUGE
    };

    PanelSpot add_applet(in Applet panel_applet,
                         in string goad_id,
                         out string cfgpath,
                         out string globcfgpath,
                         out unsigned long winid);

    oneway void quit ();

    readonly attribute boolean in_drag;
  };
};
```

This interface will init a `GNOME::PanelSpot` which is what the Applet talks to. The idea is that the applet builds a widget to be displayed and the panel gives the applet a window to draw this widget onto. The applet also tells the panel what menu items should apear when right clicking and the panel tells the applet what menu item was cliked so that the correct callback is called. All this is made through the PanelSpot and the Applet interface.

If I am an applet, I must first make sure that my implementation of the Applet interface is working and is registered against the ORB. Then, I call the `GNOME::Panel::add_applet` function to tell the Panel that I want to be on the Panel.

Upon return, it will return a `GNOME::PanelSpot` interface and a *unsigned long winid*.

The GTK Plug and Socket widgets are used to convert this winid in something useful. The Socket widget will generate a window on which things can be drawn. The Socket widget is converted to a winid with the *GTK_WINDOW_XWINDOW(widget)* macro (from gtk/gtkx.h) and the Plug widget will convert back this winid to a GtkWidget. The panel creates a socket and passes the winid back to the applet asking for a window on the panel.Then, the applet will add its widgets to the corresponding gtk_plug widget.

Because i believe that the gtk_plug/gtk_socket/winid system is very important in understanding how things work, i wrote some test code. (see appendix D)

Hopefully, all this complexity is hidden behind C api functions designed for the Applet interface.

To handle the right click context menu, one needs some work. All events on the Socket window are sent to the socket widget which has no knowledge of the plug widget drawing things on it. This means that the socket cannot pass the events to the plug widget and to the application which handles the inside of the socket window. Events must thus be handled with another system. The right click event is handled through the folowing interfaces:

```
module GNOME {
  interface PanelSpot {
    // gosh !! i removed many things from here !!
    oneway void add_callback (in string callback_name,
    in string stock_item,
    in string menuitem_text);
    oneway void remove_callback (in string callback_name);
  };
```

```
  interface Applet {
    // gosh !! i removed many things from here !!
    oneway void do_callback (in string callback_name);
  };
};
```

One should note that most function interface declarations are oneway to save network bandwidth and system load.

The Applet can build the menu to be displayed upon right click with the `GNOME::PanelSpot::add_callback`, `GNOME::PanelSpot::remove_callback`. Then, the Panel will call `GNOME::Applet::do_callback` when the user has clicked on a menu item.

The PanelSpot and Applet interfaces are much bigger than what i showed of them: there are many other things to take care of: the orientation changes, the size changes, tooltips showing, background color/pixmap... I leave you the rest to look at and spend some time on. It is hopefully much simpler :)

Hopefully, once more, all the complexity of CORBA is hidden behind C functions which are more or less wrappers around the CORBA code. This is what the libapplet library does. (gnome-core/panel/applet-widget.c and applet-widget.h)

Chapter 6. The GNOME::Unknown interface

Introduction

All Bonobo interfaces inherit from the `GNOME::Unknown` interface. `GNOME::Unknown` is a CORBA translation of the COM `IUnknown` interface. Readers familiar with COM/DCOM may thus skip the next section presenting this interface and its use.

Definition of GNOME::Unknown

The `GNOME::Unknown` interface is defined as follows:

```
module GNOME {
interface Unknown {
void ref ();

void unref ();

Object query_interface (in string repoid);
};
};
```

`GNOME::Unknown` has two main uses. its first two functions are used to maintain a reference count of the references to the interface. The `query_interface` method is used to ask an object what CORBA interfaces it supports, as well as to switch between the various interfaces an object supports.

Reference counting allows objects to manage their lifetimes. CORBA has no build-in mechanism for this. In CORBA, if an object dies, the clients will be notified of its destruction by a *CORBA_SYSTEM_EXCEPTION* exception when sending requests to the now unexisting object.

The idea of using reference counting is an idea which is at the basis of COM Distributed Object System: every time you get a reference to an interface, you must increase the reference count by calling `ref()`, and every time you have finished using an object, you should `unref()` it. The object simply maintains an internal counter which is increased by each call to ref and decreased by each call to unref. This allows the object to know when it is not useful anymore to stay alive. When the count reaches zero, the object may decide to destroy itself. Note: interfaces acquired through the `query_interface` have their reference count already increased. The must be `unref`'ed, however.

When writing code, you should be careful to balance `ref`'s and `unref`'s. If there are too few `unref`, effectively you have a memory leak, especially if the component is implemented in a shared library. Especially, make sure components are released (`unref`'ed) before going out of scope. On the other hand, if there are to many `unref`'s, you may kill the component prematurely, while your program still needs it.

The `query_interface` allows you to browse the interfaces supported by some object. Let's imagine I have a word processor capable of embedding Bonobo components, but which can also be embedded itself, in other containers. This program is thus both a container and a component. This means that it must support both the `GNOME::embeddable` and the `GNOME::container` interfaces.

The idea is that if you ask for a reference to the `CORBA::Object` which represents this application, you will get a pointer to its `GNOME::Unknown` interface. From here on, you may ask the object for either its `GNOME::Container` interface or the `GNOME::Embeddable` interface with `query_interface` to get an object reference to the implementation of the requested interface. For example, to get the `GNOME::Embeddable` interface:

```
CORBA_Object obj_unknown;
CORBA_Object obj_embeddable;

obj_unknown = CosNaming_NameService_resolve (orb, "Word97", &ev);

obj_embeddable = GNOME_Unknown_query_interface (obj_unknown, "IDL:GNOME/embeddable:1.0", &ev);
```

This system also allows you to work around versioning problems. Let's imagine I have defined a new embeddable interface. this interface's name will be `IDL:GNOME/embeddable:2.0` which means that if you also preserve the old `IDL:GNOME/embeddable:1.0` interface on your object, any container willing to embed your component will do so without any problem, provided it supports either 1.0 or 2.0 embeddable objects. This allows you to make your applications evolve in time, without breaking backward compatibility.

A sample extendable implementation of the `GNOME::Unknown` interface is detailed in the next section.

The GnomeObject object

As usual, the GNOME hackers have done everything they could to ease your development. Bonobo provides a nice default impplementation for all its CORBA interfaces so that the programmer is not obliged to learn CORBA. This implementation is C code, as everywhere in The GNOME. The implementation is based on the GTK Object system.

The GTK Object system allows you to do semi object-oriented programming in C. The reader is strongly encouraged to read the GTK code (gtk/gtk/gtkobject.c, gtk/gtk/gtkobject.h, gtk/gtk/gtkwidget.c and gtk/gtk/gtkwidget.h) and books detailling the GTK architecture.

The `GnomeObject`, which inherits from `GtkObject` also implements the `GNOME::Unknown` interface (bonobo/bonobo/gnome-objet.c and bonobo/bonobo/gnome-object.h). Here are its basic structures:

```
typedef struct {
POA_GNOME_Unknown servant_placeholder;
gpointer gnome_object;
} GnomeObjectServant;

typedef struct _GnomeObjectPrivate GnomeObjectPrivate;

typedef struct {
GtkObject base;

CORBA_Object          corba_objref;
gpointer              servant;
GnomeObjectPrivate   *priv;
} GnomeObject;

typedef struct {
GtkObjectClass parent_class;
```

```
/*
 * signals.
 */
void  (*query_interface) (GnomeObject *object,
                                  const char *repo_id,
                                  CORBA_Object *retval);
void  (*system_exception)(GnomeObject *object,
                                  CORBA_Object cobject,
                                  CORBA_Environment *ev);
void  (*object_gone)     (GnomeObject *object,
                                  CORBA_Object cobject,
                                  CORBA_Environment *ev);
} GnomeObjectClass;

typedef struct {
int   ref_count;
GList *objs;
} GnomeAggregateObject;

struct _GnomeObjectPrivate {
GnomeAggregateObject *ao;
int destroy_id;
};
```

As you may have seen it, to each `GnomeObject` is associated a servant through the "servant" field:

```
GnomeObject obj;
PortableServer_Servant servant = obj->servant;
```

Also, to each servant is associated a `GnomeObject` through the "gnome_object" field:

```
PortableServer_Servant servant;
GnomeObject obj = servant->gnome_object;
```

The `GnomeObject` is thus a wrapper around the CORBA servant which effectively implements the `GNOME::Unknown` interface. We have wrapper functions: `gnome_object_get_servant` and `gnome_object_from_servant` to switch from the servant to the `GnomeObject`. We have also the `gnome_object_bind_to_servant` which is used to establish this one to one mapping.

The `gnome_object_ref()` and `gnome_object_unref()` functions implement the `ref()` and `unref()` interface functions. These functions are called internally by the `impl_GNOME_Unknown_ref()` and `impl_GNOME_Unknown_unref()` functions.

```
void
gnome_object_ref (GnomeObject *object)
{
g_return_if_fail (object != NULL);
g_return_if_fail (GNOME_IS_OBJECT (object));
g_return_if_fail (object->priv->ao->ref_count != 0);

object->priv->ao->ref_count++;
}

void
gnome_object_unref (GnomeObject *object)
{
g_return_if_fail (object != NULL);
g_return_if_fail (GNOME_IS_OBJECT (object));
g_return_if_fail (object->priv->ao->ref_count > 0);
```

```
object->priv->ao->ref_count-;

if (object->priv->ao->ref_count == 0){
GnomeAggregateObject *ao = object->priv->ao;
GList *l;

for (l = ao->objs; l; l = l->next){
GnomeObject *o = l->data;

gtk_signal_disconnect (GTK_OBJECT (o), o->priv->destroy_id);
gtk_object_destroy (l->data);
}

g_list_free (ao->objs);
g_free (ao);
}
}

static void
impl_GNOME_Unknown_ref (PortableServer_Servant servant, CORBA_Environment *ev)
{
GnomeObject *object;

object = gnome_object_from_servant (servant);
        /* baaad !! we MUST call gnome_object_ref() */
object->priv->ao->ref_count++;
}

static void
impl_GNOME_Unknown_unref (PortableServer_Servant servant, CORBA_Environment *ev)
{
GnomeObject *object;

object = gnome_object_from_servant (servant);

gnome_object_unref (object);
}
```

The implementation of the `query_interface` function is a little more complex: the `GnomeObject` structure has a pointer to the GnomeObjectPrivate structure which itself has a pointer to a GnomeAggregateObject which itself has a pointer to a GList which holds the list of the GnomeObjects linked to this GnomeObject.

```
GnomeObject object;
GList list = object->priv->ao->objs;
```

This means that your GnomeObjects are aggregated independently. Let's look at the container/component exemple. First, i implement the component stuff. I have lots of code which builds a GnomeObject which is used to browse the interfaces of a component.

Now, i can implement the container side, independently of the component side. When this is done, all i have to do is to aggregate my GnomeObject object implementing the component to the GnomeObject implementing the container:

```
GnomeObject container;
GnomeObject component;
/* this aggegates only the first object to my GnomeObject object */
/* one should loop through the GList */
container->priv->ao->objs = g_list_prepend (container->priv->ao->objs,
                                            component->priv->ao->objs-
>data)
```

This is exactly what is being done in the gnome_object_add_interface function.

On the whole, the GnomeObject object is used as the basis of all the bonobo objects. To implement a CORBA interface which inherits from the `GNOME::Unknown` interface, the Bonobo hackers implement it by defining an object which inherits from the `GnomeObject` object.

Then, they can aggregate these objects with the `gnome_object_add_interface`.

Chapter 7. A hand made container

Introduction

Containers must implement the `GNOME::container` interface. This interface is used as a two way communication facility with objects implementing the `GNOME::embeddable` interface.

Containers must also implement the `GNOME::ClientSite` and the `GNOME::ViewFrame` interfaces. Containers which support in place activation must also implement the `GNOME::UIHandler` interface. We'll focus here on simple non-in-place non-linking and non-saving containers. Containers which embedd components and lauch components in a new window when editing is necessary.

Basically, the idea is that if you ask a container to embed a component (by clicking on "insert from" and choosing a special application, the container will add this component to its list of embedded components and will create a ClientSite for this component. Then, it should create a ViewFrame for the component to display its view. A `ViewFrame` should be created for each of the component view. There should be one and only one ClientSite for each embedded component.

Each interface implements some "sinks". ie: functions which are designed to notify an object of something. That way, we have a two way communication: some of the component sinks are called by the container and some of the container sinks are called by the component. Here, the `GNOME::ClientSite` interface of the component is a sink for the `GNOME::embeddable` interface of the component and the `GNOME::ViewFrame` interface of the container is a sink for the `GNOME::View` interface of the component.

The folowing figure should nicely (:) sum up these relationships:

Figure 7-1. Container/component relationships

The interfaces

`GNOME::container`

The `GNOME::container` interface is rather simple:

```
module GNOME {

  interface ParseDisplayName : Unknown {
    exception SyntaxError {};
  };

  interface Container : ParseDisplayName {
    typedef sequence<Unknown> ObjectList;
```

```
    ObjectList enum_objects ();

    exception NotFound {};
    Unknown get_object (in string item_name, in boolean only_if_exists)
            raises (SyntaxError, NotFound);
  };
};
```

This interface is mainly used to activate the embedded components when linking is used. Nothing really interesting.

The other interfaces, mainly `GNOME::ClientSite` and `GNOME::ViewFrame` are much more interesting. They are responsible for the real embedding of the components.

GNOME::ClientSite

Here is the ClientSite idl (bonobo/idl/gnome-client-site.idl)

```
module GNOME {
  interface ClientSite : GNOME::Unknown {

    Container get_container ();

    void show_window (in boolean shown);

    Persist::Status save_object ();
  };
};
```

- The `get_container` function is used for linking. we'll see this later.
- The `show_window` method is rather obvious: when the embedded component has been notified by the container that it must activate itself (ie: enter in edit mode) through the `GNOME::View::activate`, it will notify the container that it has shown its own editing window through this method if it does not support inplace activation. This will allow the container to show in the composed document that the component is beeing edited in another window by greying the component views. `GNOME::ClientSite::show_window` is similar to `IOleClientSite::OnShowWindow`.
- The `save_object` method is used by the embedded component to ask the container to save itself with all its requested views. This will return a `Persist::Status` value which will tell the component whether the save suceeded or not. `GNOME::ClientSite::save_object` is similar to `IOleClientSite::SaveObject`.

GNOME::ViewFrame

Here is the `GNOME::ViewFrame` idl definition (bonobo/idl/gnome-embeddable.idl):

```
module GNOME {
  interface ViewFrame : GNOME::Unknown {
    typedef sequence<octet> pixbuf;

    ClientSite get_client_site ();

    UIHandler get_ui_handler ();
```

```
    void view_activated (in boolean state);

    void deactivate_and_undo ();

    void request_resize (in short new_width, in short new_height);

    void activate_uri (in string uri, in boolean relative);
  };
};
```

- `request_resize` is a sink for the component: the component will call `request_resize` to ask the container for more or less space if its size changed. In turn, the container will call the `GNOME::View::size_allocate` function of the View connected to the `GNOME::ViewFrame` the request was sent to. The component's View will thus know what his size can be and will update the corresponding view accordingly.
- `view_activated` is also a sink for the component: it must be called by the component to ensure the container that it accepts that its view is to be activated. Typically, this function should be called by the component implementation of `GNOME::View::activate`. The `view_activated` function is mainly used to grey the Views to show that they are beeing edited in a separate window.
- `get_ui_handler` must return a reference to an object implementing the `GNOME::UIHandler` interface. This new interface is used for the merging of the component menus and toolbar with the container own menus and toolbars for in-place editing. We'll discuss this later.
- `deactivate_and_undo` should be called by the View beeing edited if undo is called just after activation. This means that the user requested to exit the editing mode.
- `get_client_site` is rather simple: must return an object reference to the `GNOME::ViewFrame`'s `GNOME::ClientSite` interface
- `activate_uri` ?? gloups ??

A summary

The complexity of the interfaces presented should not hide from you the fact that what happens is *REALLY* simple.

Let's imagine i ask the container to insert a component by the "insert from" menu. The container will first try to locate the desired component with the GOAD. The component's factory will be started if not allready and an instance of the component will be build. The container will add to its internal list of components this component.

The container then creates a ClientSite for the new component and the designed ClientSite creates a ViewFrame. The ClientSite is passed to the component as well as the ViewFrame so that it may build the corresponding `GNOME::embeddable` and `GNOME::View` interfaces.

Once all the interfaces are build, the container intercepts all events on the non-activated component view. If a double-click occurs, the container notifies the component of this to allow the component to activate itself (ie: create a new window for editing or merge its toolbars/menus for in-place editing). The container waits for the component to tell him it is activated and then it greys the edited view to show the user it is beeing edited.

That's all ! The folowing section will show you how to implement such a set of interfaces and will show you how the different interfaces interact.

A sample container

To create the code which is detailed here, i first wrote the bonobo.idl file which is more or less the equivalent of the bonobo.idl file delivered with Bonobo. Just a few things were removed (ie: linking/saving and in-place activation stuff). Then, i ran orbit-idl on it with the –skeleton-impl option. This gave me a bonobo-skel-impl.c file. I then removed everything from this file which was a reference to the Embeddable interfaces and saved the result in the container.c file. I then created the component.c file by removing all references to container side stuff from bonobo-skel-impl.c. The resulting code with the Makefile is in Appendix D.

Then, all you need to do is to implement the different interfaces left in both files. The Makefile will build both files and the skeletons and stubs.

To build this code, you will need to latest CVS versions of Bonobo and ORBit. Once compiled, just launch the "container" executable and you will be able to add components and components' views. There is only One kind of component: a hand-made button... Here is a small screen shot from my PC with 2 different embeddables each having multiple views with different data.

Figure 7-2. Container/Button Containee

While this chapter concentrates on the inner workings of the container, the folowing chapter will answer all your questions on the button componnent.

Creating the Container interface.

The first step is to create the container interface. The folowing code is called when the "container" executable is launched. It will simply create a Container interface...

```
int main (int argc, char **argv)
{
  CORBA_Environment ev;
  CORBA_ORB orb;
  CORBA_Object root_poa;
  PortableServer_POAManager root_poa_manager;

  CORBA_Object obj;

  CORBA_exception_init (&ev);
  orb = gnome_CORBA_init ("a \"simple\" container",
  "1.0",
  &argc,
  argv,
  GNORBA_INIT_SERVER_FUNC,
  &ev);

  root_poa = CORBA_ORB_resolve_initial_references (orb, "RootPOA", &ev);

  root_poa_manager = PortableServer_POA__get_the_POAManager (
                    (PortableServer_POA) root_poa, &ev);
```

```
  PortableServer_POAManager_activate (root_poa_manager, &ev);

  /* this function will create all the other
   * needed interfaces
   */
  obj = impl_GNOME_Container__create ((PortableServer_POA)
      root_poa,
      &ev);


  CORBA_exception_free (&ev);
  gtk_main();
  return 0;
}
```

The `impl_GNOME_Container__create` was slightly modified from the automatically denerated form to build the GUI to allow to user to interact with the program.

```
static GNOME_Container
impl_GNOME_Container__create(PortableServer_POA poa,
     CORBA_Environment * ev)
{
   GNOME_Container retval;
   impl_POA_GNOME_Container *newservant =
     g_new0(impl_POA_GNOME_Container, 1);
   PortableServer_ObjectId *objid;
   GtkWidget *app;
   GnomeUIInfo menu_file[] = {
     GNOMEUIINFO_MENU_NEW_ITEM("New Component",
       "Add a new component",
       add_component,
       newservant),
     GNOMEUIINFO_MENU_NEW_ITEM("New Component View",
       "Add a new View to a component",
       add_component_view,
       newservant),

     GNOMEUIINFO_SEPARATOR,
     GNOMEUIINFO_MENU_EXIT_ITEM(app_quit, NULL),
     GNOMEUIINFO_END
   };
   GnomeUIInfo menu_help[] = {
     GNOMEUIINFO_END
   };
   GnomeUIInfo main_menu[] = {
     GNOMEUIINFO_SUBTREE("Files", menu_file),
     GNOMEUIINFO_SUBTREE("Help", menu_help),
     GNOMEUIINFO_END
   };


   newservant->servant.vepv = &impl_GNOME_Container_vepv;
   newservant->poa = poa;
   newservant->client_site_list = NULL;

   app = gnome_app_new ("container 1.0", "Container");
   gnome_app_create_menus (GNOME_APP(app), main_menu);
   gtk_window_set_default_size (GTK_WINDOW(app), 200, 200);
   newservant->box = gtk_vbox_new (TRUE, TRUE);
   gnome_app_set_contents (GNOME_APP(app), newservant->box);
   gtk_widget_show_all (app);
```

```
   POA_GNOME_Container__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

   return retval;
}
```

This function simply creates a GnomeApp which contains a small menu. The "File" menu allows you to create a new component and embedd it, create a new view for an allready created component and exit. The exit callback is not fully implemented: it should destroy all allocated data and make sure embedded components are disconnected before quiting (i am lazy).

```
void app_quit (void)
{
  gtk_main_quit ();
}
```

The 2 other callbacks are much more complex and are discussed below.

Adding a Component

Creating a component is done with the `add_component` function which is passed as parameter the container servant. The first thing done is to ask the GOAD to create a component. Here, we ask for our hand-made component called "Component".

```
  list = goad_server_list_get ();
  embeddable = goad_server_activate_with_id (list,
                                             "Component",
                                             GOAD_ACTIVATE_NEW_ONLY,
                                             NULL);
  goad_server_list_free (list);
```

Then, we create the ClientSite corresponding to this Embeddable.

```
  client_site_servant = impl_GNOME_ClientSite__create (poa,
                                                       embeddable,
                                                       container_servant,
                                                       &ev);
  client_site = PortableServer_POA_servant_to_reference (poa,
                                                         client_site_servant,
                                                         &ev);
  /* keep the client site ref in some safe place
     for the container... */
  container_servant->client_site_list =
    g_slist_append (container_servant->client_site_list,
                    client_site_servant);
```

The `impl_GNOME_ClientSite__create` function was also modified from its automatically generated form: it now has 2 more parameters: "embeddable" and "container_servant".

```
static impl_POA_GNOME_ClientSite *
impl_GNOME_ClientSite__create(PortableServer_POA poa,
                              GNOME_Embeddable embeddable,
                              impl_POA_GNOME_Container *container_servant,
                              CORBA_Environment * ev)
{
```

```
   impl_POA_GNOME_ClientSite *retval;
   impl_POA_GNOME_ClientSite *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_GNOME_ClientSite, 1);
   newservant->servant.vepv = &impl_GNOME_ClientSite_vepv;
   newservant->poa = poa;
   newservant->embeddable = embeddable;
   newservant->container_servant = container_servant;

   POA_GNOME_ClientSite__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = newservant;

   return retval;
}
```

The ClientSite stores a pointer to its embeddable and to its container servant in its embeddable and container_servant fields. (the corresponding structure is shown below.)

```
struct _impl_POA_GNOME_ClientSite
{
  POA_GNOME_ClientSite servant;
  PortableServer_POA poa;
  impl_POA_GNOME_Container *container_servant;
  impl_POA_GNOME_ViewFrame *active;
  GSList *view_frame_list;

  GNOME_Embeddable embeddable;
  CORBA_boolean show;
};
```

Once the ClientSite and its Embeddable are created, we must tell the Embeddable what ClientSite it will talk to:

```
  /* now, just give the newly created component
     a ref to its ClientSite */
  GNOME_Embeddable_set_client_site (embeddable,
                                    client_site,
                                    &ev);
  /* and set its host name. ie: the name which
     will be displayed on the handle of its
     editing window */
  GNOME_Embeddable_set_host_name (embeddable,
                                  "my_component",
                                  "my_component",
                                  &ev);
```

Last, we must create a ViewFrame/View so that the user has visual feedback of the fact the component was activated.

First, a ViewFrame is created. We give the newly created ViewFrame a pointer to its ClientSite with `client_site_servant`. Then, the ViewFrame is added to the ClientSite list of ViewFrames and then we add the ViewFrame GtkSocket to the container main window.

Once the ViewFrame is correctly intialized, we wan create the corresponding View with `GNOME::Embeddable::new_view` which is given its ViewFrame CORBA::Object reference as parameter and returns a GNOME::View object reference. The View object reference is stored in the ViewFrame structure.

Finally, we must give the GNOME::View its GtkSocket with `GNOME::View::set_window`. Now, the View should be displayed. and everything is fine :).

```
  view_frame_servant = (impl_POA_GNOME_ViewFrame *)
    impl_GNOME_ViewFrame__create (poa, client_site_servant, &ev);
  view_frame = PortableServer_POA_servant_to_reference (poa,
view_frame_servant,
&ev);
  client_site_servant->view_frame_list =
    g_slist_append (client_site_servant->view_frame_list,
    view_frame_servant);
  gtk_box_pack_start (GTK_BOX(container_servant->box),
      view_frame_servant->wrapper,
      FALSE, 0, 0);
  /* realize the socket ... so that it has a GdkWindow
     otherwise, the GdkWindow not existing will make the
     prgm crash  when calling set_window...
     A hard to find bug...
  */
  gtk_widget_show_all (container_servant->box);

  /* now, create the View for this ViewFrame */
  view = GNOME_Embeddable_new_view (embeddable, view_frame, &ev);
  view_frame_servant->view = CORBA_Object_duplicate (view, &ev);


  /* Now, give the View its window */
  GNOME_View_set_window (view,
 GDK_WINDOW_XWINDOW(view_frame_servant->socket->window),
 &ev);

  /* pfew !! now, the view should be displayed ...*/
```

For your pleasure, here is the ViewFrame structure:

```
struct _impl_POA_GNOME_ViewFrame
{
   POA_GNOME_ViewFrame servant;
   PortableServer_POA poa;
   impl_POA_GNOME_ClientSite *client_site_servant;
   GNOME_View view;
   GtkWidget *socket;
   GtkWidget *wrapper;
   gint activate;
   GtkRequisition request;
   GtkAllocation allocation;

};
```

Adding a Component View

The function responsible for this task is obvioulsy add_component_view. It receives as parameter a pointer to the container servant.

I decided to add a View to the last ClientSite of container->client_site_list. The first step is thus to find this ClientSite.

```
  last = g_slist_last (container_servant->client_site_list);
  client_site_servant = (impl_POA_GNOME_ClientSite *) last->data;
  client_site = PortableServer_POA_servant_to_reference (poa,
```

```
                                                       client_site_servant,
                                                       &ev);
```

Then, we create a new ViewFrame for our future View and update the ViewFrame list of the ClientSite. We also pack the new ViewFrame into the container main window and make sure the GtkSocket is realized (ie: it is given an X window by GTK).

```
  view_frame_servant = (impl_POA_GNOME_ViewFrame *)
    impl_GNOME_ViewFrame__create (poa, client_site_servant, &ev);
  view_frame = PortableServer_POA_servant_to_reference (poa,
                                                        view_frame_servant,
                                                        &ev);
  client_site_servant->view_frame_list =
    g_slist_append (client_site_servant->view_frame_list,
                    view_frame_servant);
  gtk_box_pack_start (GTK_BOX(container_servant->box),
                      view_frame_servant->wrapper,
                      FALSE, 0, 0);
  gtk_widget_show_all (container_servant->box);
```

Last, we create the Embeddable new view and we give the newly created View its X window.

```
  view = GNOME_Embeddable_new_view (client_site_servant->embeddable,
                                    view_frame,
                                    &ev);
  view_frame_servant->view = CORBA_Object_duplicate (view, &ev);
  GNOME_View_set_window (view,
                         GDK_WINDOW_XWINDOW(view_frame_servant->socket-
>window),
                         &ev);
```

Now, we have a brand new window displayed in our container window.

Activating a View

Once all views are created, we must allow the user to activate a View to edit it. Obviously, this is done with the `view_frame_cb` callback which is called everytime the user double clicks on one of the inactive view.

There are 2 possibilities: either we have allready an activated view or we have no other activated view. In the first case, we just deactivate the active view and we activate the requested view. In the second simpler case, we just activate the requested view.

```
void view_frame_cb (GtkWidget *widget,
            GdkEvent *event,
                    gpointer data)
{
  impl_POA_GNOME_ViewFrame *servant =
    (impl_POA_GNOME_ViewFrame*) data;
  impl_POA_GNOME_ClientSite *active_client_site =
    servant->client_site_servant->container_servant->active;

  if (event->type == GDK_2BUTTON_PRESS
      && active_client_site == NULL)
    activate_view (servant);
  /* here, no other view in the container was activated */
  else if (event->type == GDK_2BUTTON_PRESS
   && active_client_site != NULL) {
```

```
    /* here, another view is active. */
    deactivate_view (active_client_site->active);
    activate_view (servant);
  }
}
```

The activation is done with the `activate_view` function. We simply update the fields of our Container and ClientSite to be able to track active views and we call `GNOME::View::activate` to tell the newly activated view that it is now active.

```
void
activate_view (impl_POA_GNOME_ViewFrame *servant)
{
  CORBA_Environment ev;

  CORBA_exception_init (&ev);
  GNOME_View_activate (servant->view,
                       1,
                       &ev);
  servant->client_site_servant->active = servant;
  servant->client_site_servant->container_servant->active =
    servant->client_site_servant;
  CORBA_exception_free (&ev);
}
```

deactivation is done with `deactivate_view`.

```
void
deactivate_view (impl_POA_GNOME_ViewFrame *servant)
{
  CORBA_Environment ev;

  CORBA_exception_init (&ev);
  GNOME_View_activate (servant->view,
                       0,
                       &ev);
  CORBA_exception_free (&ev);
}
```

Finally, the ViewFrame cover must be removed and painted to show the user that the editing took place. This is done by the GNOME::View which must call its `GNOME::ViewFrame::view_activated`.

```
static void
impl_GNOME_ViewFrame_view_activated(impl_POA_GNOME_ViewFrame * servant,
                                    CORBA_boolean state,
                                    CORBA_Environment * ev)
{
  servant->activate = state;
  if (state == 1)
    gnome_wrapper_set_covered (GNOME_WRAPPER(servant->wrapper),
                               0);
  else if (state == 0    )
    gnome_wrapper_set_covered (GNOME_WRAPPER(servant->wrapper),
                               1);

}
```

The user should now be able to edit the button text.

Chapter 8. A hand made containee

Introduction

In the previous chapter, we saw the container interfaces responsible for component embedding. There were three. You have probably noticed how deep their relationship with the component interfaces was.

The component has only 2 necessary interfaces. `GNOME::embeddable` and `GNOME::View` which are closely related to respectively `GNOME::ClientSite` and `GNOME::ViewFrame`. A component should of course implement the `GNOME::PersisFile` interface to support saving but we'll discuss this later.

The interfaces

GNOME::EmbeddableFactory

The main interface for components is `GNOME::Embeddable`. Objects implementing it should be created with a `GNOME::EmbeddableFactory`, which is a factory for Embeddable objects. Such a factory should be declared with the standard .gnorba files we studied in chapter 5.

```
module GNOME {

  typedef sequence<string> stringlist;

  interface GenericFactory {
    exception CannotActivate { };
    boolean supports(in string obj_goad_id);

    Object create_object(in string goad_id, in stringlist params)
           raises(CannotActivate);
  };

  interface EmbeddableFactory : GNOME::GenericFactory {
    Embeddable create_path (in string path);
  };
};
```

- The `create_path` function is used to handle linking support.
- The `create_object` function is used to create the object itself, as we allready discussed it in chapter 5.

GNOME::Embeddable

Here is the `GNOME::embeddable` interface:

```
module GNOME {
  interface Embeddable : GNOME::Unknown {
```

```
    void set_client_site (in ClientSite client_site);

    ClientSite get_client_site ();

    void set_host_name (in string name, in string appname);

    void set_uri (in string uri);

    exception UserCancelledSave {};
    enum CloseMode {
      SAVE_IF_DIRTY,
      NO_SAVE,
      PROMPT_SAVE
    };
    void close (in CloseMode mode)
          raises (UserCancelledSave);

    struct GnomeVerb {
      string name;
      string label;
      string hint;
    };
    typedef sequence<GnomeVerb> verb_list;
    verb_list get_verb_list ();

    void advise (in AdviseSink advise);

    void unadvise ();

    long get_misc_status (in long type);

    exception MultiViewNotSupported {};
    View new_view (in ViewFrame frame)
         raises (MultiViewNotSupported);
  };
};
```

This interface is a big, but fortunately easily understandable.

- The `set_client_site` function is used by the container to tell the component who it is talking to. This function is generally called once the container has created the corresponding ClientSite (yes... it's useful) to allow the embeddable created with its factory to communicate with the container (to handle the size negotiation for example).
- The `set_host_name` function is rather simple: it is used to set the window's name - displayed in the window handle by the window manager - of the window used for editing if the object does not support in-place activation. This name should also be used for the application's window if in-place editing is used.
- The `close` function is the embeddable destructor: it will be called by the container when the composed document is closed.
- The `advise` and `unadvise` functions will be discussed later. No real implementation is needed now: the `GNOME::AdviseSink` interface is not yet defined.
- `new_view` is THE main function of this interface. It is used by the container to request new views of the component. The component is not obliged to be able to give away more than one view of the same object (which is why the MultiViewNotSupported signal exists) but is recommended to. This function is the `GNOME::View`

factory. One should note that new views are given their `GNOME::ClientSite` upon creation with the "frame" parameter.

- `get_misc_status` is used for miscellaneous things. Don't really known what things...
- `get_verb_list` allows the container to ask an Embeddable what special verbs it supports. A verb is a simple non-parametrized action which the Embeddable can execute through `GNOME::View::do_verb`. The classic example of a verb is "next_page" for a Postscript Viewer.

GNOME::View

Here is this famous `GNOME::View` interface.

```
module GNOME {

  interface View : GNOME::Unknown {

    void size_allocate (in short width, in short height);

    void size_query (out short desired_width, out short desired_height);

    typedef unsigned long windowid;
    void set_window (in windowid id);

    void activate (in boolean activated);

    void reactivate_and_undo ();

    void do_verb (in string verb_name);

    void set_zoom_factor (in double zoom);
  };
};
```

- `size_query` and `size_allocate` are used to handle the size negotiation mechanism of the container: when the container needs to be resized, it asks all of its children for their preferred size. Then, the container allocates corresponding areas for these children, given its size, and notifies them with `size_allocate`.

  `size_allocate` is also used by the container when the child view requests a resize with `GNOME::ViewFrame::resize_request`.
- `set_window` is used by the container to give the containee a window to draw its view on. The container calls this function and gives the View an X window. This is achieved through the use of the GtkSocket/GtkPlug mechanism - allready studied in chapter 5. see appendix D for sample code.-.
- `activate` is used to tell the view to activate itself. When called, the view will either create a separate edit window or will try in-place activation, depending on the type of component. Please, note that the Embeddable must tell the container about its type - in-place or non-in-place - with a call to `GNOME::ClientSite::show`.
- `reactivate_and_undo` is called by the container when the user did an 'undo' just after having de-activated a component.
- `do_verb` is mainly used to ask the component to execute simple non-parametrized actions like "next_page" for a Postscript Viewer.

- Finally, the `set_zoom_factor` function's use is obvious: it sets the View zoom factor.

A sample containee

I have already detailed how this code was written from automatically generated code from ORBit. So, here is the raw code.

Creating a `GNOME::Embeddable`

The first thing to do is to implement the `GNOME::EmbeddableFactory`. This is what the folowing code does. The factory is created, and it is registered against the GOAD. Typically, this code will be called by the `goad_activate_factory` function which itself is called by `goad_activate_with_id (list, "Component", ...)`.

```
int main (int argc, char **argv)
{
  CORBA_Environment ev;
  CORBA_ORB orb;
  CORBA_Object root_poa;
  PortableServer_POAManager root_poa_manager;
  CosNaming_NamingContext name_server;
  CORBA_Object obj;

  CORBA_exception_init (&ev);
  orb = gnome_CORBA_init ("a \"simple\" container",
  "1.0",
  &argc,
  argv,
  GNORBA_INIT_SERVER_FUNC,
  &ev);

  root_poa = CORBA_ORB_resolve_initial_references (orb,
   "RootPOA",
   &ev);

  root_poa_manager = PortableServer_POA__get_the_POAManager (
     (PortableServer_POA) root_poa, &ev);
  PortableServer_POAManager_activate (root_poa_manager, &ev);

  /* this function will create all the other
   * needed interfaces
   */
  obj = impl_GNOME_EmbeddableFactory__create ((PortableServer_POA)
      root_poa,
      &ev);
  /* register against the name service */
  name_server = gnome_name_service_get ();
  goad_server_register (name_server, obj,
goad_server_activation_id(),
"server",
&ev);
  /* finished !! */
  CORBA_exception_free (&ev);
  gtk_main ();
  return 0;
}
```

Once the EmbeddableFactory is created and registered, the GOAD is likely to call the `GNOME::EmbeddableFactory::create_object` function to create our Embeddable object.

```
static CORBA_Object
impl_GNOME_EmbeddableFactory_create_object(impl_POA_GNOME_EmbeddableFactory *
   servant, CORBA_char * goad_id,
   GNOME_stringlist * params,
   CORBA_Environment * ev)
{
   CORBA_Object retval = CORBA_OBJECT_NIL;

   retval = impl_GNOME_Embeddable__create (servant->poa, ev);
   goad_server_register (CORBA_OBJECT_NIL, retval, goad_id, "server", ev);

   return retval;
}
```

The `GNOME::Embeddable` creation function, `impl_GNOME_Embeddable__create` is rather simple. It will simply do standard stuff and initialize the Embeddable with the "Ok ?!" data. (ie: the data which will be displayed on the button).

```
static GNOME_Embeddable
impl_GNOME_Embeddable__create(PortableServer_POA poa,
      CORBA_Environment * ev)
{
   GNOME_Embeddable retval;
   impl_POA_GNOME_Embeddable *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_GNOME_Embeddable, 1);
   newservant->servant.vepv = &impl_GNOME_Embeddable_vepv;
   newservant->poa = poa;
   strcpy (newservant->data, "Ok ?!");;

   POA_GNOME_Embeddable__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

   return retval;
}
```

The next call which will be done to the Embeddable is to the `Embeddable::set_client_site` function. This function will call the `ClientSite::show_window` function to make sure the ClientSite knows that we will be doing some non-in-place activation.

```
static void
impl_GNOME_Embeddable_set_client_site(impl_POA_GNOME_Embeddable * servant,
      GNOME_ClientSite client_site,
      CORBA_Environment * ev)
{
  servant->client_site = CORBA_Object_duplicate (client_site, ev);
  /* tell the container that the embeddable will
     be activated in an external window */
  GNOME_ClientSite_show_window (client_site, 1, ev);
}
```

Creating a new `GNOME::View`

The next step in this process is creating a new view which will be done with `GNOME::Embeddable::new_view`. We'll just create the view itself and store a pointer to this view in the embeddable servant. The view is also given a pointer to the corresponding ViewFrame and Embeddable.

```
static GNOME_View
impl_GNOME_Embeddable_new_view(impl_POA_GNOME_Embeddable * servant,
       GNOME_ViewFrame frame,
       CORBA_Environment * ev)
{
   GNOME_View view;
   impl_POA_GNOME_View *view_servant;
   view_servant = impl_GNOME_View__create (servant->poa,
   frame,
   servant,
   ev);
   servant->view_servant_list = g_slist_append (servant->view_servant_list,
view_servant);

   view = PortableServer_POA_servant_to_reference(servant->poa, view_servant, ev);

   return view;
}
```

The `impl_GNOME_View__create` function just initializes some stuff.

```
static impl_POA_GNOME_View *
impl_GNOME_View__create(PortableServer_POA poa,
GNOME_ViewFrame frame,
impl_POA_GNOME_Embeddable *embeddable_servant,
CORBA_Environment * ev)
{

   impl_POA_GNOME_View *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_GNOME_View, 1);
   newservant->servant.vepv = &impl_GNOME_View_vepv;
   newservant->poa = poa;
   newservant->frame = CORBA_Object_duplicate (frame, ev);
   newservant->embeddable_servant = embeddable_servant;
   /* this is what will be displayed in the component view ! */
   newservant->widget = gtk_button_new_with_label (embeddable_servant-
>data);

   POA_GNOME_View__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);

   return newservant;
}
```

Once the view is created, it will wait to be given an X window. This will be done by the Container which will call `GNOME::View::set_window`

```
static void
impl_GNOME_View_set_window(impl_POA_GNOME_View * servant,
   GNOME_View_windowid id,
   CORBA_Environment * ev)
{
```

```
  servant->plug = gtk_plug_new (id);
  gtk_container_add (GTK_CONTAINER(servant->plug),
     servant->widget);
  gtk_widget_show_all (servant->plug);
}
```

Activating a View

Views are activated through the `GNOME::View::activate` function which must be called by the container. First, it will check if the View is to be activated or deactivated and build the editing GUI or not. Last, it will call `GNOME::ViewFrame::view_activated` to tell the container to set on or off the cover over the View.

```
static void
impl_GNOME_View_activate(impl_POA_GNOME_View * servant,
 CORBA_boolean activated,
 CORBA_Environment * ev)
{
  static GtkWidget *window = NULL;
  GtkWidget *vbox;
  GtkWidget *hbox;
  GtkWidget *button;
  GtkWidget *entry;

  if (activated == 1) {
    /* activate view */
    /* sample code to create the external editing
       window of the view */
    /* ...... here, code for the editing GUI ...... */
  } else if (activated == 0 && window != NULL) {
     /* deactivate view */
    gtk_widget_destroy (window);
  }
  /* tell the container the view was activated or not*/
  GNOME_ViewFrame_view_activated (servant->frame,
  activated,
  ev);
}
```

If the view is to be activated, the GUI will be build:

```
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_object_set_data (GTK_OBJECT (window),
 "view_servant",
 servant);
    gtk_signal_connect (GTK_OBJECT (window),
"destroy",
GTK_SIGNAL_FUNC (cancel_cb),
window);

    gtk_signal_connect (GTK_OBJECT (window),
"delete_event",
GTK_SIGNAL_FUNC (cancel_cb),
window);

    vbox = gtk_vbox_new (TRUE, TRUE);
    hbox = gtk_hbox_new (TRUE, TRUE);

    entry = gtk_entry_new_with_max_length (10);
```

```
    gtk_entry_set_text (GTK_ENTRY (entry),
servant->embeddable_servant->data);
    gtk_box_pack_start (GTK_BOX (vbox),
entry,
TRUE,
0,
0);

    gtk_box_pack_start (GTK_BOX (vbox),
gtk_hseparator_new(),
TRUE,
0,
0);

    gtk_box_pack_end (GTK_BOX(vbox), hbox,
      TRUE,
      0,
      0);
    button = gtk_button_new_with_label ("Cancel");
    gtk_box_pack_end (GTK_BOX(hbox),
      button,
      TRUE,
      0,
      0);
    gtk_signal_connect (GTK_OBJECT(button),
"clicked",
GTK_SIGNAL_FUNC(cancel_cb),
window);
    button = gtk_button_new_with_label ("Apply");
    gtk_object_set_data (GTK_OBJECT (button),
 "window",
 window);
    gtk_box_pack_end (GTK_BOX(hbox),
      button,
      TRUE,
      0,
      0);
    gtk_signal_connect (GTK_OBJECT(button),
"clicked",
GTK_SIGNAL_FUNC(apply_cb),
entry);
    button = gtk_button_new_with_label ("Ok");
    gtk_object_set_data (GTK_OBJECT (button),
 "window",
 window);
    gtk_box_pack_end (GTK_BOX(hbox),
      button,
      TRUE,
      0,
      0);
    gtk_signal_connect (GTK_OBJECT(button),
"clicked",
GTK_SIGNAL_FUNC(ok_cb),
entry);

    gtk_container_add (GTK_CONTAINER(window),
       vbox);
    gtk_widget_show_all (GTK_WIDGET(window));
```

The important idea is that the buttons of the GUI are given enough parameters to either close the GUI or apply the changes to all the views of the activated Embeddable.

Here is for example the `apply_cb` function which loops through all the embeddable views to change their buttons' labels.

```
void apply_cb (GtkWidget *button,
       gpointer data)
{
  GSList *list;
  impl_POA_GNOME_View * servant;
  GtkWidget *label;
  GtkWidget *entry = (GtkWidget *) data;
  GtkWidget *window = gtk_object_get_data (GTK_OBJECT(button),
   "window");
  impl_POA_GNOME_View *view_servant = (impl_POA_GNOME_View *)
    gtk_object_get_data (GTK_OBJECT (window),
 "view_servant");
  strcpy (view_servant->embeddable_servant->data,
  gtk_entry_get_text (GTK_ENTRY (entry)));

  list = view_servant->embeddable_servant->view_servant_list;
  while (list != g_slist_last (view_servant->embeddable_servant->view_servant_list)) {
    servant = (impl_POA_GNOME_View *)list->data;
    label = GTK_BIN(servant->widget)->child;

    gtk_label_set_text (GTK_LABEL(label),
view_servant->embeddable_servant->data);
    list = list->next;
  }
  servant = (impl_POA_GNOME_View *)list->data;
  label = GTK_BIN(servant->widget)->child;
  gtk_label_set_text (GTK_LABEL(label),
      view_servant->embeddable_servant->data);
}
```

Chapter 9. The GNOME Storage Interfaces

Introduction

Often it is needed to save the state of an object in some persistant way. For example, think of a word processor component that must store your hard work in some place. The well-known, simple solution is of course to store the object state (the document) in a file.

Enter documents in the component age. Documents may contain *GIMP*-components showing a picture, *Gnumeric*-components showing some spreadsheet etc., possibly recursively. How would your word processor know how to save the state for these components's state? And how could this all be stored in your average linear file?

Note that their are two questions here:

- How can we store compound documents, ie. documents consisting of various (heterogenous) components?
- How can we do file operations on a document consisting of components we do not know about at compile time?

This chapter addresses the first question, with the GNOME::Stream and GNOME::Store interfaces. The second question is dealt with in the next chapter.

Streams and stores

There a various ways to store the state of a group of related objects. A working solution would be to use a conventional file system directory in which each of the objects is a file, or, if it contains objects itself, a subdirectory. Unfortunately, this is an inconvenient solution, for example when copying the document. Experience has shown that dividing one logical file into multiple physical files leads to problems.

The solution used by the OLE2-developers was to use *compound files*. Simply put, a compound file is *filesystem in a file*. Just like a file system consists of directories and files, a compound file consists of *streams* and *storages*.

Bonobo follows this OLE2-approach, and defines interfaces for stores and streams:

- `GNOME:Stream`: the Bonobo interface for streams. This is the counterpart of OLE2's `IStream` (surprisingly).
- `GNOME:Storage`: the Bonobo interface for streams. This is the counterpart of OLE2's `IStorage`.

Next to these interfaces, Bonobo also provides an implementation for compound files. This implementation is called *libefs*. In the next sections `GNOME::Stream`, `GNOME::Storage` and libefs will be discussed.

Stores

The `GNOME::Storage` interface specifies the 'directory-part' of the filesystem-in-a-file. The following methods are exported (bonobo-storage.idl):

```
interface Storage : Unknown {

typedef long OpenMode;
const OpenMode READ  = 1;
const OpenMode WRITE = 2;
const OpenMode DENY_READ = 4;
const OpenMode DENY_WRITE = 8;

exception NameExists {};
exception NotFound {};
exception NoPermission {};

typedef sequence<string> directory_list;

Stream create_stream (in string path) raises (NameExists);
Stream open_stream   (in string path, in OpenMode mode) raises (NotFound, NoPermission);
Storage create_storage (in string path) raises (NameExists);
Storage open_storage   (in string path, in OpenMode mode) raises (Not-
Found, NoPermission);
void copy_to (in Storage target);
void rename (in string path_name, in string new_path_name) raises (Name-
Exists, NotFound);
void commit ();
directory_list list_contents (in string path) raises (NotFound);
void erase (in string path) raises (NotFound);
};
```

- `create_stream`: Creates a new stream object located in the given path. The path needs not to be a file system path, as stream object are not necessarily files. `create_stream` raises a NameExist when there's already a stream located a the given path. If all goes well, the function return a brand new stream object.
- `open_stream`: Opens an existing stream object in the given path and with the given permissions. `open_stream` raises a NotFound exception, when the file is, well, not found, and a NoPermission when the client doesn't have sufficient permissions. If all goes well, it returns an open stream object.
- `create_storage`: Creates a new storage at the given path. This is equivalent to the **mkdir** shell command. This function raises a NameExist when there's already a storage at the given path. The path needs not to be a filesystem path. If all goes well, the function returns a new storage object.
- `open_storage`: Open the storage at the given path with the given permissions. `open_storage` raises NotFound or NoPermission, as expected.
- `copy_to`: Copies the storage to the given target storage. This is comparable to the shell command **cp -R**. No exceptions are defined for `copy_to`.
- `rename`: Renames the given substorage or stream. Exceptions are raised when the source doesn't exist (NotFound), or the target already exists (NameExists).
- `commit` `commit` reflects changes (for a transacted storage) to the parent level, and flushes any buffers. Note that, at this time, the accompanying `revert`- function does not exist.
- `list_contents` The `list-contents` lists the contents of the storage at the given path, like **ls** does in the shell. On succes, the `list_contents` returns a list with the of streams and storages at the given path.
- `erase` The `erase` erases the stream or storage existing at the given path. Raises a NotFound-exception when the stream or storage can't be found.

Stream

```
module GNOME {
interface Stream : Unknown {
typedef sequence<octet> iobuf;

enum SeekType {
SEEK_SET,
SEEK_CUR,
SEEK_END
};

long read     (in long count, out iobuf buffer);
long write    (in iobuf buffer);
long seek     (in long offset, in SeekType whence);
void truncate (in long length);
void copy_to  (in string dest, in long bytes,
inout long read, inout long written);
void commit ();
void close ();
boolean eos ();
long length ();
};
};
```

- `read`: Reads 'count' numbers from the stream, and places it in the i/o buffer 'buffer'. `read` returns with the number of bytes read. `read` returns -1 on failure in case of a file-based stream.
- `write`: Writes data from 'buffer' to the stream. `write` returns the number of bytes written.
- `seek`: Moves the stream pointer to the given offset, from the based denoted by the 'SeekType'. The seek type can be eiter the beginning (SEEK_SET), the end (SEEK_END), or the current position (SEEK_CUR), like the `lseek` library function.
- `truncate`: Changes the size of a stream to the given length.
- `copy_to`: Copies 'bytes' bytes from the stream to some destination stream 'dest'. After returning, 'read' contains the number of bytes actually read, while 'written' contains the number of bytes actually written.
- `commit`: Ensures that changes made to the streams are reflected in the parent storage.
- `close`: Closes the stream.
- `eos`: Returns a boolean denoting whether or on the stream pointer is a the end of the stream. Similar to the `feof` library call.
- `length` Returns the length of the stream (in bytes).

libefs

The `GNOME::Stream` and `GNOME::Storage` interfaces specify how we can deal with compound documents. Of course, without an implementation these interfaces aren't terribly useful. Fortunately, Bonobo comes with such an implementation: *libefs*. Note that libefs addresses the need for a file-based stream/storage implementation. However, Bonobo also provides a memory-based implementation.

The libefs library was written by Dietmar Maurer and implements the filesystem-in-a-file principle. However, libefs is quite flexible and allows the use of various backend drivers - i.e. the physical structure of a compound file could be either one single file, or a directory tree where all objects in the compound files are separate file.

libefs has a very familiar UNIX-like API, thus making working with compound files relatively easy. The main difference is probably speed - libefs is quite a bit slower than the 'regular' UNIX API. This issue might be resolved by writing a kernel driver for compound files.

The libefs library is very much a work-in-progress. For example, the compound file implementation does not free any deleted streams or storages. This means a document can collect a lot of garbage. At this moment the only work-around is to do a 'Save-as', which will create a tidy new file.

We won't discuss the details of libefs here - please refer to the libefs documentation. For our (Bonobo) purposes it suffices to say that libefs is the file-based implementation of streams and stores.

Bonobo storage example

So far, we have only discussed the IDL interfaces of `GNOME::Store` and `GNOME::Stream`. Now, we will show an example of the use these interfaces. In the example we use the GTK+-based implementation of the Bonobo interfaces.

```
        /* TODO:
 * here comes some nice example of GNOME::Stream and GNOME::Store
 */

        #include <stdio.h>
        int main ()
        {
            printf("Hello, Bonobo\n");
            return 0;
        }
```

Chapter 10. The GNOME Persistance Interfaces

Introduction

Nowadays, a document (such as an *AbiWord* word-processing document) may contain things like *GIMP*-pictures, *Gnumeric*-spreadsheets etcetera. In the previous chapter, we saw how compound files (stream/stores) can be used to store heterogenous content. However, this still leaves a problem: how do we know how to save all these objects? We cannot 'simply' add code to support file operations for every possible embeddable component; this would bloat our code tremendously, and be outdated as soon as somebody writes a new component.

There must be a better solution. And indeed, there is. Instead of dealing with the saving and loading (persistance) of containees (components) ourselves, this task is delegated to the components: we make them responsible for their own persistence.

Every component that wants to support storing its state to some persistent medium (such as a disk file), must support one or more of the following Bonobo interfaces:

- `GNOME::Persist`;
- `GNOME::PersistFile`;
- `GNOME::PersistStorage`;
- `GNOME::PersistStream`.

These interfaces are defined in the 'gnome-persist.idl' IDL file. By supporting these interfaces, a component says it knows how to deal with persistence: it knows how to store its state as well as how to retrieve it. Depending on the type of component and its specific needs, it can support on or more of these interfaces. Note that this supports persistence for a complete hierarchy of components (like a Guppi-component in a Gnumeric-spreadsheet in a Abiword-document) quite nicely: every container simply delegates persistence to its containees.

In this chapter we will describe these interfaces, and show how to use them.

The persistence interfaces

Persist

The `GNOME::Persist`-interface inherits directly from `GNOME::Unknown` and serves as a base class for the other Persist*-classes. It supports the following method:

```
interface Persist : Unknown {
enum Status {
SAVE_OK,
SAVE_CANCEL,
SAVE_FAILED
};
string get_goad_id ();
```

```
exception WrongDataType {};
};
```

- `get_goad_id` Gets the GOAD id for the datatype.

PersistFile

The `PersistFile`-interface specifically addresses the need to write and save to files (rather than stream and storages). As such, the main application of this interface is to for (embedded) components store/retrieve their state in/for separate files.

```
interface PersistFile : Persist {

exception NoCurrentName {
string extension;
};

void load (in string path) raises (WrongDataType);
void save (in string path);
boolean is_dirty ();
string get_current_file () raises (NoCurrentName);
};
```

- `load`: Loads a component's persistent state from a file on the given path. `load` raises a 'WrongDataType' when the persistent state is not valid for this type of component.
- `save`: Saves a component's state to a file on the given path.
- `is_dirty` Returns whether the internal component's state has changed from its persistent (file) state.
- `get_current_file` Returns the name of the file we're using for the persistence of this component. Raises a NoCurrentName if there's not such a file yet.

PersistStorage

Embedded components supporting the `PersistStorage`-interface enable their container to pass them a storage object (as described in the previous chapter). The `PersistStorage`-interface addresses embedded components that are containers themselves, and therefore need a storage (rather than a stream) to store their state.

The main application of this interface is to support persistance for contained objects that are containers themselves.

```
interface PersistStorage : Persist {
boolean is_dirty ();
void load (in Storage storage) raises (WrongDataType);
void save (in Storage storage, in boolean same_as_loaded);
void init_new (in Storage storage);
};
```

- `load`: Loads a component's persistent state from a storage. `load` raises a 'WrongDataType' when the persistent state is not valid for this type of component.
- `save`: Saves a component's state to the given storage. Function also needs a same_as_loaded boolean parameter, denoting we're doing a 'save' or a 'save-as' operation.
- `is_dirty`: Returns true if the internal component's state has changed, from its persistent (file) state, and false otherwise.
- `init_new`: Initializes the given storage.

PersistStream

Embedded components supporting the `PersistStream`-interface enable their container to pass them a stream object (as described in the previous chapter). The `PersistStream`-interface addresses embedded components that are non-aggregates and can thus be stored in a stream.

```
interface PersistStream : Persist {
boolean is_dirty ();
void load (in Stream stream) raises (WrongDataType);
void save (in Stream stream);
long get_size_max ();
};
```

- `is_dirty`: Returns true if the internal component's state has changed, from its persistent state, and false otherwise.
- `load`: Loads a component's persistent state from a stream. `load` raises a 'WrongDataType' when the persistent state is not valid for this type of component.
- `save`: Saves a component's state to the given stream.
- `get_size_max`: Returns the size (in bytes) of the stream needed to save the component's state.

Bonobo persistence example

So far, we have only discussed the IDL interfaces of `GNOME::Store` and `GNOME::Stream`. Now, we will show an example of the use these interfaces. In the example we use the GTK+-based implementation of the Bonobo interfaces.

```
        /* TODO:
 * here comes some nice example of the persistence interfaces.
 */

        #include <stdio.h>
        int main ()
        {
            printf("Hello, I'm Persistent\n");
            return 0;
        }
```

Appendix A. A Note about the Authors

Thanks

We would like first to thank our many readers who sent in comments and corrections. More precisely:

- David C. Manson
- Dietmar Maurer
- Michaël O'Reilly
- ...

And many others who were lost in one of Mathieu's numerous Mail folder destruction. If you feel you should be there, just drop one of the authors a mail.

Dirk-Jan C. Binnema

After finishing university, I'm working as a C++ programmer for WinNT. I do a lot of COM-programming, and I'm glad I can apply my knowledge to the benefit of the GNOME project. By the way: Mathieu did probably over 90% of this tutorial, so 90% of all praise should go to him...

Mathieu Lacage

I am currently a student at the ENST (http://www.enst.fr) (Ecole Nationale Supérieure des Télécomunications). I am interested in Computer science and Microelectronics. I live in Paris, France and i use to seek a girlfriend. (i wonder: why do they leave me so quickly ?)

Dirk Jan is just too humble: he did quite some job too (more than 10 % :). I wish to thank him for all this. I owe also most of my knowledge of CORBA and Bonobo to Miguel de Icaza, Nat Friedman and Eliot Lee, Three of the top GNOME hackers.

More seriously, i currently seek a job in the US from the 1st July 2000 to the 31st December 2000 in the field of Software Engineering. My resume can be found here (http://www.stud.enst.fr/~lacage/cv/cv/html).

To reach me, <lacage@email.enst.fr>

Appendix B. Online ressources

There are not that many resources online on CORBA and components... However, here is a list of sites which may be of some use to you.

- GNOME developement (http://developer.gnome.org) : this is the official GNOME developer page. It contains many documentations (including this one), tutorials and the latest news on the front of GNOME technologies.
- ORBit (http://www.labs.redhat.com/orbit/) : ORBit is the CORBA 2.2 ORB used in The GNOME. Its homepage contains an interesting tutorial on CORBA and some links to various other CORBA resources.
- The OMG (http://www.omg.org) : the Object Managment Group is the organism which is responsible for the evolution of the CORBA standard. All the CORBA reference papers are available there. A must !
- Microsoft (http://msdn.microsoft.com/default.asp) : they invented OLE. THE reference book on OLE 2 (Inside OLE) is available online for free on their developer site.

Appendix C. Idl files in The GNOME

The GNOME is made of many source packages: one should download the gnome-libs (core GNOME libraries), gnome-core (core GNOME desktop) and bonobo (obviously, bonobo stuff) packages. Here is a list of the idl files which are included in there.

- gnome-unknown.idl: this file can be found in gnome-libs/idl. It details the GNOME:Unknown interface.
- gnome-factory.idl: this file can be found in gnome-libs/idl. It details the GNOME::GenericFactory interface.
- name-service.idl: this file can be found in gnome-libs/idl. It details the various interfaces implemented and used by the GOAD
- Table.idl: this file can be found in gnome-libs/idl. It details a GNOME::Table interface... I wonder what it could be used for ...
- desktop-editor.idl: this file can be found in gnome-libs/idl. It details various interfaces which must be implemented by a text-editor to integrate well in a GNOME environment.
- desktop-textviewer.idl: this file can be found in gnome-libs/idl. It details various interfaces which must be implemented by a text-viewer to integrate well in a GNOME environment.
- gnome-panel.idl: this file can be found in gnome-core/idl. It simply details the various interfaces implemented by a good GNOME Panel and its applets.
- help-browser.idl: this file can be found in gnome-core/idl. It simply details the various interfaces implemented by a good GNOME help browser.
- control-center.idl: this file can be found in control-center/idl. It simply details the various interfaces implemented by a good GNOME Control Center and its applets.
- gnome-container.idl: this file can be found in bonobo/idl. Of course, we have there the GNOME::Container interface.
- gnome-embeddable.idl: this file can be found in bonobo/idl. Very surprisingly, the GNOME::Embeddable interface is detailed there.
- gnome-client-site.idl: this file can be found in bonobo/idl. The Container GNOME::ClientSite interface definition.
- gnome-ui-handler.idl: this file can be found in bonobo/idl. You want in place activations ? Just implement this set of interfaces.
- gnome-storage.idl, gnome-persist.idl and gnome-progressive.idl: these file can be found in bonobo/idl. They are used to implement saving support on components.
- gnome-advise.idl: this file can be found in bonobo/idl. A copy of the IAdviseSink interface from COM/DCOM. Not used yet. Is it deprecated ?
- gnome-moniker.idl: this file can be found in bonobo/idl. Implement linking support on your components !
- gnome-canvas.idl: this file can be found in bonobo/idl. Experimental: Non squre components !!
- gnome-control.idl: this file can be found in bonobo/idl. Experimental: Controls like with activeX.

Appendix D. Code listing

This apendix presents the miscellaneous programs' code presented in this book.

GtkSocket/GtkPlug

Makefile

```
CFLAGS=-g -Wall
CC=gcc

all: client server

plug.h: plug.idl
orbit-idl ./plug.idl

# build the client. ie: the thing which gives a window to
# display the true GUI
client: socket.o plug-stubs.o plug-common.o plug.h
${CC} ${CFLAGS} `gnome-config -libs gnorba gnomeui` \
socket.o plug-stubs.o plug-common.o -o client
socket.o: socket.c plug.h
${CC} ${CFLAGS} `gnome-config -cflags gnorba` \
-c socket.c -o socket.o

# build the server. ie: the thing which builds the true GUI
server: plug.o plug-skels.o plug-common.o plug.h
${CC} ${CFLAGS} `gnome-config -libs gnorba gnomeui` \
plug.o plug-skels.o plug-common.o -o server
plug.o: plug.c plug.h
${CC} ${CFLAGS} `gnome-config -cflags gnorba` \
-c plug.c -o plug.o

# build stubs/skeletons
plug-skels.o: plug-skels.c plug.h
${CC} ${CFLAGS} `orbit-config -cflags server` \
`gnome-config -cflags gnorba` \
-c plug-skels.c -o plug-skels.o
plug-stubs.o: plug-stubs.c plug.h
${CC} ${CFLAGS} `orbit-config -cflags client` \
`gnome-config -cflags gnorba` \
-c plug-stubs.c -o plug-stubs.o
plug-common.o: plug-common.c plug.h
${CC} ${CFLAGS} `orbit-config -cflags` \
`gnome-config -cflags gnorba` \
-c plug-common.c -o plug-common.o


clean:
rm *.o client server *~ plug-skelimpl.c \
plug-stubs.c plug-common.c plug-skels.c \
plug.h 2>/dev/null
```

plug.idl

```
module view {
  interface plug {
    void set_window_id (in unsigned long winid);
  };
};
```

plug.c

```
#include "plug.h"
#include <gnome.h>
#include <libgnorba/gnorba.h>
#include <ORBitservices/CosNaming.h>

 /*** App-specific servant structures ***/
 typedef struct {
   POA_view_plug servant;
   PortableServer_POA poa;
   GtkWidget *widget;
 } impl_POA_view_plug;

 /*** Implementation stub prototypes ***/
 static void impl_view_plug__destroy(impl_POA_view_plug * servant,
                                     CORBA_Environment * ev);
 static void
  impl_view_plug_set_window_id(impl_POA_view_plug * servant,
 CORBA_unsigned_long winid,
 CORBA_Environment * ev);

 /*** epv structures ***/
 static PortableServer_ServantBase__epv impl_view_plug_base_epv =
 {
    NULL,                        /* _private data */
    NULL,                        /* finalize routine */
    NULL,                        /* default_POA routine */
 };
 static POA_view_plug__epv impl_view_plug_epv =
 {
    NULL,                        /* _private */
    (gpointer) & impl_view_plug_set_window_id,

 };

 /*** vepv structures ***/
 static POA_view_plug__vepv impl_view_plug_vepv =
 {
    &impl_view_plug_base_epv,
    &impl_view_plug_epv,
 };

 /*** Stub implementations ***/
 static view_plug
 impl_view_plug__create(PortableServer_POA poa,
CORBA_Environment * ev)
 {
    view_plug retval;
    impl_POA_view_plug *newservant;
    PortableServer_ObjectId *objid;
```

```
   newservant = g_new0(impl_POA_view_plug, 1);
   newservant->servant.vepv = &impl_view_plug_vepv;
   newservant->poa = poa;
   newservant->widget = gtk_button_new_with_label ("A small button :)");

   POA_view_plug__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);


   return retval;
}

static void
impl_view_plug__destroy(impl_POA_view_plug * servant,
CORBA_Environment * ev)
{
   PortableServer_ObjectId *objid;

   objid = PortableServer_POA_servant_to_id(servant->poa, servant, ev);
   PortableServer_POA_deactivate_object(servant->poa, objid, ev);
   CORBA_free(objid);

   POA_view_plug__fini((PortableServer_Servant) servant, ev);
   g_free(servant);
}

static void
impl_view_plug_set_window_id(impl_POA_view_plug * servant,
                             CORBA_unsigned_long winid,
                             CORBA_Environment * ev)
{
  GtkWidget *window;
  window = gtk_plug_new (winid);
  gtk_container_add (GTK_CONTAINER(window),
     servant->widget);
  gtk_widget_show_all (window);
}

int main (int argc, char **argv)
{
  CORBA_Environment ev;
  CORBA_ORB orb;
  PortableServer_POA root_poa;
  PortableServer_POAManager root_poa_manager;
  CosNaming_NamingContext name_server;
  CosNaming_NameComponent name_component[1] =
  {{"view_plug","server"}};
  CosNaming_Name name =
  {1, 1, name_component, CORBA_FALSE};

  CORBA_Object obj;

  CORBA_exception_init (&ev);

  orb = gnome_CORBA_init ("a simple socket client",
                          "0.1",
                          &argc,
                          argv,
                          GNORBA_INIT_SERVER_FUNC,
                          &ev);
  root_poa = (PortableServer_POA)
```

```
  CORBA_ORB_resolve_initial_references (orb,
 "RootPOA",
 &ev);
 root_poa_manager = PortableServer_POA__get_the_POAManager (root_poa,
    &ev);
 PortableServer_POAManager_activate (root_poa_manager, &ev);

 obj = impl_view_plug__create (root_poa, &ev);

 name_server = gnome_name_service_get ();
 CosNaming_NamingContext_bind (name_server, &name, obj, &ev);

 CORBA_exception_free (&ev);
 gtk_main();
 return 0;

}
```

socket.c

```
#include <orb/orbit.h>
#include <ORBitservices/CosNaming.h>
#include <gnome.h>
#include <gdk/gdkx.h>
#include <libgnorba/gnorba.h>
#include "./plug.h"

 int main (int argc, char **argv)
 {
   CORBA_Environment ev;
   CORBA_ORB orb;
   CosNaming_NamingContext name_server;
   CosNaming_NameComponent name_component[1] =
   {{"view_plug","server"}};
   CosNaming_Name name =
   {1, 1, name_component, CORBA_FALSE};
   CORBA_Object obj;
   GtkWidget *win;
   GtkWidget *socket;

   CORBA_exception_init (&ev);

   orb = gnome_CORBA_init ("a simple socket client",
                         "0.1",
                         &argc,
                         argv,
                         GNORBA_INIT_SERVER_FUNC,
                         &ev);

   name_server = gnome_name_service_get ();

   obj = CosNaming_NamingContext_resolve (name_server,
  &name,
  &ev);

   win = gtk_window_new (GTK_WINDOW_TOPLEVEL);
   gtk_window_set_default_size (GTK_WINDOW (win),
300,
300);
   socket = gtk_socket_new ();
```

```
  gtk_container_add (GTK_CONTAINER(win),
     socket);
  gtk_widget_show_all (win);

  view_plug_set_window_id (obj, GDK_WINDOW_XWINDOW(socket->window), &ev);
  CORBA_exception_free (&ev);

  gtk_main ();
  return 0;
}
```

Container/Embeddable

Makefile

```
CC = gcc
CFLAGS = -Wall -g
GNOMEPATH = /opt/gnome
ORBIT_INCLUDE_DIR = -I${GNOMEPATH}/share/idl


all: container component

idl: bonobo.h

bonobo.h:
orbit-idl ${ORBIT_INCLUDE_DIR} bonobo.idl

container: container.o bonobo-skels.o \
bonobo-common.o bonobo-stubs.o
${CC} ${CFLAGS} `gnome-config -libs gnorba gnomeui bonobo` \
container.o bonobo-skels.o bonobo-common.o bonobo-stubs.o \
-o container
container.o: container.c bonobo.h
${CC} ${CFLAGS} `gnome-config -cflags gnorba bonobo` \
-c container.c -o container.o

component: component.o bonobo-skels.o \
bonobo-common.o bonobo-stubs.o
${CC} ${CFLAGS} `gnome-config -libs gnorba gnomeui` \
component.o bonobo-skels.o bonobo-common.o bonobo-stubs.o \
-o component
component.o: component.c bonobo.h
${CC} ${CFLAGS} `gnome-config -cflags gnorba` \
-c component.c -o component.o

bonobo-skels.o: bonobo-skels.c bonobo.h
${CC} ${CFLAGS} `orbit-config -cflags server` \
`gnome-config -cflags gnorba` \
-c bonobo-skels.c -o bonobo-skels.o
bonobo-common.o: bonobo.h bonobo-common.c
${CC} ${CFLAGS} `orbit-config -cflags server` \
`gnome-config -cflags gnorba` \
-c bonobo-common.c -o bonobo-common.o
bonobo-stubs.o: bonobo.h bonobo-stubs.c
${CC} ${CFLAGS} `orbit-config -cflags client` \
`gnome-config -cflags gnorba` \
```

```
-c bonobo-stubs.c -o bonobo-stubs.o


clean:
rm bonobo-stubs.c bonobo.h bonobo-skels.c \
bonobo-common.c *~ *.o container component \
test 2>/dev/null
```

bonobo.idl

```
#include <gnome-unknown.idl>
#include <gnome-factory.idl>

module GNOME {
  interface Persist : Unknown {
    enum Status {
      SAVE_OK,
      SAVE_CANCEL,
      SAVE_FAILED
    };
  };
};


module GNOME {
  interface ParseDisplayName : Unknown {
    exception SyntaxError {};
  };

  interface Container : ParseDisplayName {
    typedef sequence<Unknown> ObjectList;
    ObjectList enum_objects ();
    exception NotFound {};
    Unknown get_object (in string item_name,
                        in boolean only_if_exists)
            raises (SyntaxError, NotFound);
  };

  interface ClientSite : GNOME::Unknown {
    Container get_container ();
    void show_window (in boolean shown);
  };

  interface ViewFrame : GNOME::Unknown {
    ClientSite get_client_site ();
    void view_activated (in boolean state);
    void deactivate_and_undo ();
    void request_resize (in short new_width,
                         in short new_height);
    void activate_uri (in string uri,
                       in boolean relative);
  };
};

module GNOME {
  interface View : GNOME::Unknown {
    typedef unsigned long windowid;
    void size_allocate (in short width,
                        in short height);
```

```
    void size_query (out short desired_width,
                     out short desired_height);
    void set_window (in windowid id);
    void activate (in boolean activated);
    void reactivate_and_undo ();
    void do_verb (in string verb_name);
    void set_zoom_factor (in double zoom);
  };

  interface Embeddable : GNOME::Unknown {
    void set_client_site (in ClientSite client_site);
    ClientSite get_client_site ();

    void set_host_name (in string name,
                        in string appname);
    void set_uri (in string uri);

    exception UserCancelledSave {};
    enum CloseMode {
      SAVE_IF_DIRTY,
      NO_SAVE,
      PROMPT_SAVE
    };
    void close (in CloseMode mode)
         raises (UserCancelledSave);

    struct GnomeVerb {
      string name;
      string label;
      string hint;
    };
    typedef sequence<GnomeVerb> verb_list;
    verb_list get_verb_list ();

    exception MultiViewNotSupported {};
    View new_view (in ViewFrame frame)
         raises (MultiViewNotSupported);
  };

  interface EmbeddableFactory : GNOME::GenericFactory {
  };
};
```

container.c

```
#include <gnome.h>
#include <libgnorba/gnorba.h>
#include <ORBitservices/CosNaming.h>
#include <gdk/gdkx.h>
#include <bonobo/gnome-wrapper.h>

#include "bonobo.h"

/*** App-specific servant structures ***/

typedef struct
{
   POA_GNOME_ParseDisplayName servant;
   PortableServer_POA poa;
```

```
}
impl_POA_GNOME_ParseDisplayName;

typedef struct _impl_POA_GNOME_Container impl_POA_GNOME_Container;
typedef struct _impl_POA_GNOME_ClientSite impl_POA_GNOME_ClientSite;
typedef struct _impl_POA_GNOME_ViewFrame impl_POA_GNOME_ViewFrame;

struct _impl_POA_GNOME_Container
{
  POA_GNOME_Container servant;
  PortableServer_POA poa;
  GSList *client_site_list;
  impl_POA_GNOME_ClientSite *active;
  GtkWidget *box;
  gint refcount;
};


struct _impl_POA_GNOME_ClientSite
{
  POA_GNOME_ClientSite servant;
  PortableServer_POA poa;
  impl_POA_GNOME_Container *container_servant;
  impl_POA_GNOME_ViewFrame *active;
  GSList *view_frame_list;

  GNOME_Embeddable embeddable;
  CORBA_boolean show;
};


struct _impl_POA_GNOME_ViewFrame
{
   POA_GNOME_ViewFrame servant;
   PortableServer_POA poa;
   impl_POA_GNOME_ClientSite *client_site_servant;
   GNOME_View view;
   GtkWidget *socket;
   GtkWidget *wrapper;
   gint activate;
   GtkRequisition request;
   GtkAllocation allocation;

};


/*** Implementation stub prototypes ***/

void add_component (GtkWidget *menu_item,
    impl_POA_GNOME_Container *container);

void app_quit (void);

static void
impl_GNOME_ParseDisplayName__destroy(impl_POA_GNOME_ParseDisplayName *
     servant, CORBA_Environment * ev);

static void
impl_GNOME_ParseDisplayName_ref(impl_POA_GNOME_ParseDisplayName * servant,
CORBA_Environment * ev);
static void
impl_GNOME_ParseDisplayName_unref(impl_POA_GNOME_ParseDisplayName * servant,
```

```
  CORBA_Environment * ev);
static CORBA_Object
impl_GNOME_ParseDisplayName_query_interface(impl_POA_GNOME_ParseDisplayName *
    servant, CORBA_char * repoid,
    CORBA_Environment * ev);

static void impl_GNOME_Container__destroy(impl_POA_GNOME_Container * servant,
  CORBA_Environment * ev);

static GNOME_Container_ObjectList
   *impl_GNOME_Container_enum_objects(impl_POA_GNOME_Container * servant,
      CORBA_Environment * ev);

static GNOME_Unknown
impl_GNOME_Container_get_object(impl_POA_GNOME_Container * servant,
CORBA_char * item_name,
CORBA_boolean only_if_exists,
CORBA_Environment * ev);

static void
impl_GNOME_Container_ref(impl_POA_GNOME_Container * servant,
 CORBA_Environment * ev);
static void
impl_GNOME_Container_unref(impl_POA_GNOME_Container * servant,
   CORBA_Environment * ev);
static CORBA_Object
impl_GNOME_Container_query_interface(impl_POA_GNOME_Container * servant,
     CORBA_char * repoid,

     CORBA_Environment * ev);

/* this signature was modified because
   the reference_to_servant function is
   not yet implemented by ORBit */
static impl_POA_GNOME_ClientSite *
impl_GNOME_ClientSite__create(PortableServer_POA poa,
      GNOME_Embeddable embeddable,
      impl_POA_GNOME_Container *container_servant,
      CORBA_Environment * ev);



static void impl_GNOME_ClientSite__destroy(impl_POA_GNOME_ClientSite *
   servant, CORBA_Environment * ev);
static GNOME_Container
impl_GNOME_ClientSite_get_container(impl_POA_GNOME_ClientSite * servant,
    CORBA_Environment * ev);

static void
impl_GNOME_ClientSite_show_window(impl_POA_GNOME_ClientSite * servant,
  CORBA_boolean shown,

  CORBA_Environment * ev);

static void
impl_GNOME_ClientSite_ref(impl_POA_GNOME_ClientSite * servant,
  CORBA_Environment * ev);
static void
impl_GNOME_ClientSite_unref(impl_POA_GNOME_ClientSite * servant,
    CORBA_Environment * ev);
static CORBA_Object
impl_GNOME_ClientSite_query_interface(impl_POA_GNOME_ClientSite * servant,
      CORBA_char * repoid,
```

```
      CORBA_Environment * ev);
/* this signature was modified because
   the reference_to_servant function is
   not yet implemented by ORBit */
static impl_POA_GNOME_ViewFrame *
impl_GNOME_ViewFrame__create(PortableServer_POA poa,
     impl_POA_GNOME_ClientSite *client_site_servant,
     CORBA_Environment * ev);

static void impl_GNOME_ViewFrame__destroy(impl_POA_GNOME_ViewFrame * servant,
  CORBA_Environment * ev);
static GNOME_ClientSite
impl_GNOME_ViewFrame_get_client_site(impl_POA_GNOME_ViewFrame * servant,
     CORBA_Environment * ev);

static void
impl_GNOME_ViewFrame_view_activated(impl_POA_GNOME_ViewFrame * servant,
    CORBA_boolean state,

    CORBA_Environment * ev);

static void
impl_GNOME_ViewFrame_deactivate_and_undo(impl_POA_GNOME_ViewFrame * servant,
 CORBA_Environment * ev);

static void
impl_GNOME_ViewFrame_request_resize(impl_POA_GNOME_ViewFrame * servant,
    CORBA_short new_width,
    CORBA_short new_height,
    CORBA_Environment * ev);

static void
impl_GNOME_ViewFrame_activate_uri(impl_POA_GNOME_ViewFrame * servant,
  CORBA_char * uri,
  CORBA_boolean relative,

  CORBA_Environment * ev);

static void
impl_GNOME_ViewFrame_ref(impl_POA_GNOME_ViewFrame * servant,
 CORBA_Environment * ev);
static void
impl_GNOME_ViewFrame_unref(impl_POA_GNOME_ViewFrame * servant,
   CORBA_Environment * ev);
static CORBA_Object
impl_GNOME_ViewFrame_query_interface(impl_POA_GNOME_ViewFrame * servant,
     CORBA_char * repoid,

     CORBA_Environment * ev);

/*** epv structures ***/

static PortableServer_ServantBase__epv impl_GNOME_ParseDisplayName_base_epv = {
   NULL, /* _private data */
   NULL, /* finalize routine */
   NULL, /* default_POA routine */
};
static POA_GNOME_ParseDisplayName__epv impl_GNOME_ParseDisplayName_epv = {
   NULL, /* _private */

};
static POA_GNOME_Unknown__epv impl_GNOME_ParseDisplayName_GNOME_Unknown_epv = {
   NULL, /* _private */
```

```
   (gpointer) & impl_GNOME_ParseDisplayName_ref,
   (gpointer) & impl_GNOME_ParseDisplayName_unref,
   (gpointer) & impl_GNOME_ParseDisplayName_query_interface,
};
static PortableServer_ServantBase__epv impl_GNOME_Container_base_epv = {
   NULL, /* _private data */
   NULL, /* finalize routine */
   NULL, /* default_POA routine */
};
static POA_GNOME_Container__epv impl_GNOME_Container_epv = {
   NULL, /* _private */

   (gpointer) & impl_GNOME_Container_enum_objects,

   (gpointer) & impl_GNOME_Container_get_object,

};
static POA_GNOME_Unknown__epv impl_GNOME_Container_GNOME_Unknown_epv = {
   NULL, /* _private */
   (gpointer) & impl_GNOME_Container_ref,
   (gpointer) & impl_GNOME_Container_unref,
   (gpointer) & impl_GNOME_Container_query_interface,
};
static POA_GNOME_ParseDisplayName__epv
   impl_GNOME_Container_GNOME_ParseDisplayName_epv = {
   NULL, /* _private */
};
static PortableServer_ServantBase__epv impl_GNOME_ClientSite_base_epv = {
   NULL, /* _private data */
   NULL, /* finalize routine */
   NULL, /* default_POA routine */
};
static POA_GNOME_ClientSite__epv impl_GNOME_ClientSite_epv = {
   NULL, /* _private */
   (gpointer) & impl_GNOME_ClientSite_get_container,

   (gpointer) & impl_GNOME_ClientSite_show_window,

};
static POA_GNOME_Unknown__epv impl_GNOME_ClientSite_GNOME_Unknown_epv = {
   NULL, /* _private */
   (gpointer) & impl_GNOME_ClientSite_ref,
   (gpointer) & impl_GNOME_ClientSite_unref,
   (gpointer) & impl_GNOME_ClientSite_query_interface,
};
static PortableServer_ServantBase__epv impl_GNOME_ViewFrame_base_epv = {
   NULL, /* _private data */
   NULL, /* finalize routine */
   NULL, /* default_POA routine */
};
static POA_GNOME_ViewFrame__epv impl_GNOME_ViewFrame_epv = {
   NULL, /* _private */
   (gpointer) & impl_GNOME_ViewFrame_get_client_site,

   (gpointer) & impl_GNOME_ViewFrame_view_activated,

   (gpointer) & impl_GNOME_ViewFrame_deactivate_and_undo,

   (gpointer) & impl_GNOME_ViewFrame_request_resize,

   (gpointer) & impl_GNOME_ViewFrame_activate_uri,

};
```

```
static POA_GNOME_Unknown__epv impl_GNOME_ViewFrame_GNOME_Unknown_epv = {
   NULL, /* _private */
   (gpointer) & impl_GNOME_ViewFrame_ref,
   (gpointer) & impl_GNOME_ViewFrame_unref,
   (gpointer) & impl_GNOME_ViewFrame_query_interface,
};

/*** vepv structures ***/

static POA_GNOME_ParseDisplayName__vepv impl_GNOME_ParseDisplayName_vepv = {
   &impl_GNOME_ParseDisplayName_base_epv,
   &impl_GNOME_ParseDisplayName_GNOME_Unknown_epv,
   &impl_GNOME_ParseDisplayName_epv,
};
static POA_GNOME_Container__vepv impl_GNOME_Container_vepv = {
   &impl_GNOME_Container_base_epv,
   &impl_GNOME_Container_GNOME_Unknown_epv,
   &impl_GNOME_Container_GNOME_ParseDisplayName_epv,
   &impl_GNOME_Container_epv,
};
static POA_GNOME_ClientSite__vepv impl_GNOME_ClientSite_vepv = {
   &impl_GNOME_ClientSite_base_epv,
   &impl_GNOME_ClientSite_GNOME_Unknown_epv,
   &impl_GNOME_ClientSite_epv,
};
static POA_GNOME_ViewFrame__vepv impl_GNOME_ViewFrame_vepv = {
   &impl_GNOME_ViewFrame_base_epv,
   &impl_GNOME_ViewFrame_GNOME_Unknown_epv,
   &impl_GNOME_ViewFrame_epv,
};

/*** Stub implementations ***/
void add_component_view (GtkWidget *menu_item,
 impl_POA_GNOME_Container *container_servant)
{
  CORBA_Environment ev;
  PortableServer_Servant poa = container_servant->poa;

  CORBA_Object view;

  GSList *last;

  CORBA_Object client_site;
  impl_POA_GNOME_ClientSite *client_site_servant;
  CORBA_Object view_frame;
  impl_POA_GNOME_ViewFrame *view_frame_servant;


  last = g_slist_last (container_servant->client_site_list);
  client_site_servant =   (impl_POA_GNOME_ClientSite *) last->data;
  client_site = PortableServer_POA_servant_to_reference (poa,
 client_site_servant,
 &ev);
  view_frame_servant = (impl_POA_GNOME_ViewFrame *)
    impl_GNOME_ViewFrame__create (poa, client_site_servant, &ev);
  view_frame = PortableServer_POA_servant_to_reference (poa,
view_frame_servant,
&ev);
  client_site_servant->view_frame_list =
    g_slist_append (client_site_servant->view_frame_list,
    view_frame_servant);
  gtk_box_pack_start (GTK_BOX(container_servant->box),
      view_frame_servant->wrapper,
```

```
      FALSE, 0, 0);
  gtk_widget_show_all (container_servant->box);

  view = GNOME_Embeddable_new_view (client_site_servant->embeddable,
view_frame,
&ev);
  view_frame_servant->view = CORBA_Object_duplicate (view, &ev);
  GNOME_View_set_window (view,
 GDK_WINDOW_XWINDOW(view_frame_servant->socket->window),
 &ev);
}

void add_component (GtkWidget *menu_item,
    impl_POA_GNOME_Container *container_servant)
{
  CORBA_Environment ev;
  GoadServerList *list;
  PortableServer_Servant poa = container_servant->poa;

  CORBA_Object embeddable;
  CORBA_Object view;

  CORBA_Object client_site;
  impl_POA_GNOME_ClientSite *client_site_servant;
  CORBA_Object view_frame;
  impl_POA_GNOME_ViewFrame *view_frame_servant;


  /* Now, create the Component: call its factory */
  list = goad_server_list_get ();
  embeddable = goad_server_activate_with_id (list,
    "Component",
    GOAD_ACTIVATE_NEW_ONLY,
    NULL);
  goad_server_list_free (list);

  /* create the ClientSite for our new component and
     give it the embeddable ref */
  client_site_servant = impl_GNOME_ClientSite__create (poa,
      embeddable,
      container_servant,
      &ev);
  client_site = PortableServer_POA_servant_to_reference (poa,
 client_site_servant,
 &ev);
  /* keep the client site ref in some safe place
     for the container... */
  container_servant->client_site_list =
    g_slist_append (container_servant->client_site_list,
    client_site_servant);


  /* now, just give the newly created component
     a ref to its ClientSite */
  GNOME_Embeddable_set_client_site (embeddable,
    client_site,
    &ev);
  GNOME_Embeddable_set_host_name (embeddable,
  "my_component",
  "my_component",
  &ev);

  /* now, create a new ViewFrame for the
```

```
     future View of the component*/
  view_frame_servant = (impl_POA_GNOME_ViewFrame *)
    impl_GNOME_ViewFrame__create (poa, client_site_servant, &ev);
  view_frame = PortableServer_POA_servant_to_reference (poa,
view_frame_servant,
&ev);
  client_site_servant->view_frame_list =
    g_slist_append (client_site_servant->view_frame_list,
    view_frame_servant);
  gtk_box_pack_start (GTK_BOX(container_servant->box),
      view_frame_servant->wrapper,
      FALSE, 0, 0);
  /* realize the socket ... so that it has a GdkWindow
     otherwise, the GdkWindow not existing will make the
     prgm crash  when calling set_window...
     A hard to find bug...
  */
  gtk_widget_show_all (container_servant->box);

  /* now, create the View for this ViewFrame */
  view = GNOME_Embeddable_new_view (embeddable, view_frame, &ev);
  view_frame_servant->view = CORBA_Object_duplicate (view, &ev);


  /* Now, give the View its window */
  GNOME_View_set_window (view,
 GDK_WINDOW_XWINDOW(view_frame_servant->socket->window),
 &ev);

  /* pfew !! now, the view should be displayed ...*/
};

void
deactivate_view (impl_POA_GNOME_ViewFrame *servant)
{
  CORBA_Environment ev;

  CORBA_exception_init (&ev);
  GNOME_View_activate (servant->view,
       0,
       &ev);
  CORBA_exception_free (&ev);
}
void
activate_view (impl_POA_GNOME_ViewFrame *servant)
{
  CORBA_Environment ev;

  CORBA_exception_init (&ev);
  GNOME_View_activate (servant->view,
       1,
       &ev);
  servant->client_site_servant->active = servant;
  servant->client_site_servant->container_servant->active =
    servant->client_site_servant;
  CORBA_exception_free (&ev);
}

void
view_frame_cb (GtkWidget *widget,
       GdkEvent *event,
       gpointer data)
{
```

```
  impl_POA_GNOME_ViewFrame *servant =
    (impl_POA_GNOME_ViewFrame*) data;
  impl_POA_GNOME_ClientSite *active_client_site =
    servant->client_site_servant->container_servant->active;

  if (event->type == GDK_2BUTTON_PRESS
      && active_client_site == NULL)
    activate_view (servant);
  /* here, no other view in the container was activated */
  else if (event->type == GDK_2BUTTON_PRESS
   && active_client_site != NULL) {
    /* here, another view is active. */
    deactivate_view (active_client_site->active);
    activate_view (servant);
  }
}


void
size_request_cb (GtkWidget *widget, gpointer data)
{
}

void
size_allocation_cb (GtkWidget *widget, gpointer data)
{
}



void app_quit (void)
{
  gtk_main_quit ();
}


static GNOME_ParseDisplayName
impl_GNOME_ParseDisplayName__create(PortableServer_POA poa,
    CORBA_Environment * ev)
{
   GNOME_ParseDisplayName retval;
   impl_POA_GNOME_ParseDisplayName *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_GNOME_ParseDisplayName, 1);
   newservant->servant.vepv = &impl_GNOME_ParseDisplayName_vepv;
   newservant->poa = poa;
   POA_GNOME_ParseDisplayName__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

   return retval;
}

static void
impl_GNOME_ParseDisplayName__destroy(impl_POA_GNOME_ParseDisplayName *
     servant, CORBA_Environment * ev)
{
   PortableServer_ObjectId *objid;

   objid = PortableServer_POA_servant_to_id(servant->poa, servant, ev);
   PortableServer_POA_deactivate_object(servant->poa, objid, ev);
```

```
   CORBA_free(objid);

   POA_GNOME_ParseDisplayName__fini((PortableServer_Servant) servant, ev);
   g_free(servant);
}

static void
impl_GNOME_ParseDisplayName_ref(impl_POA_GNOME_ParseDisplayName * servant,
CORBA_Environment * ev)
{
}

static void
impl_GNOME_ParseDisplayName_unref(impl_POA_GNOME_ParseDisplayName * servant,
  CORBA_Environment * ev)
{
}

static CORBA_Object
impl_GNOME_ParseDisplayName_query_interface(impl_POA_GNOME_ParseDisplayName *
    servant, CORBA_char * repoid,
    CORBA_Environment * ev)
{
   CORBA_Object retval;

   return retval;
}


static GNOME_Container
impl_GNOME_Container__create(PortableServer_POA poa,
     CORBA_Environment * ev)
{
   GNOME_Container retval;
   impl_POA_GNOME_Container *newservant =
     g_new0(impl_POA_GNOME_Container, 1);
   PortableServer_ObjectId *objid;
   GtkWidget *app;
   GnomeUIInfo menu_file[] = {
     GNOMEUIINFO_MENU_NEW_ITEM("New Component",
       "Add a new component",
       add_component,
       newservant),
     GNOMEUIINFO_MENU_NEW_ITEM("New Component View",
       "Add a new View to a component",
       add_component_view,
       newservant),

     GNOMEUIINFO_SEPARATOR,
     GNOMEUIINFO_MENU_EXIT_ITEM(app_quit, NULL),
     GNOMEUIINFO_END
   };
   GnomeUIInfo menu_help[] = {
     GNOMEUIINFO_END
   };
   GnomeUIInfo main_menu[] = {
     GNOMEUIINFO_SUBTREE("Files", menu_file),
     GNOMEUIINFO_SUBTREE("Help", menu_help),
     GNOMEUIINFO_END
   };


   newservant->servant.vepv = &impl_GNOME_Container_vepv;
```

```
   newservant->poa = poa;
   newservant->client_site_list = NULL;

   app = gnome_app_new ("container 1.0", "Container");
   gnome_app_create_menus (GNOME_APP(app), main_menu);
   gtk_window_set_default_size (GTK_WINDOW(app), 200, 200);
   newservant->box = gtk_vbox_new (TRUE, TRUE);
   gnome_app_set_contents (GNOME_APP(app), newservant->box);
   gtk_widget_show_all (app);

   POA_GNOME_Container__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

   return retval;
}

static void
impl_GNOME_Container__destroy(impl_POA_GNOME_Container * servant,
      CORBA_Environment * ev)
{
   PortableServer_ObjectId *objid;

   objid = PortableServer_POA_servant_to_id(servant->poa, servant, ev);
   PortableServer_POA_deactivate_object(servant->poa, objid, ev);
   CORBA_free(objid);

   POA_GNOME_Container__fini((PortableServer_Servant) servant, ev);
   g_free(servant);
}

static GNOME_Container_ObjectList *
impl_GNOME_Container_enum_objects(impl_POA_GNOME_Container * servant,
  CORBA_Environment * ev)
{
   GNOME_Container_ObjectList *retval;

   return retval;
}

static GNOME_Unknown
impl_GNOME_Container_get_object(impl_POA_GNOME_Container * servant,
CORBA_char * item_name,
CORBA_boolean only_if_exists,
CORBA_Environment * ev)
{
   GNOME_Unknown retval;

   return retval;
}

static void
impl_GNOME_Container_ref(impl_POA_GNOME_Container * servant,
 CORBA_Environment * ev)
{
  servant->refcount++;

}

static void
impl_GNOME_Container_unref(impl_POA_GNOME_Container * servant,
   CORBA_Environment * ev)
```

```
{
  if ((-servant->refcount) == 0)
    impl_GNOME_Container__destroy (servant, ev);

}

static CORBA_Object
impl_GNOME_Container_query_interface(impl_POA_GNOME_Container * servant,
     CORBA_char * repoid,
     CORBA_Environment * ev)
{
   CORBA_Object retval = CORBA_OBJECT_NIL;

   if (strcmp (repoid, "IDL:GNOME/Container:1.0"))
     retval = PortableServer_POA_servant_to_reference (servant->poa,
       servant,
       ev);

   return retval;
}



static impl_POA_GNOME_ClientSite *
impl_GNOME_ClientSite__create(PortableServer_POA poa,
      GNOME_Embeddable embeddable,
      impl_POA_GNOME_Container *container_servant,
      CORBA_Environment * ev)
{
   impl_POA_GNOME_ClientSite *retval;
   impl_POA_GNOME_ClientSite *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_GNOME_ClientSite, 1);
   newservant->servant.vepv = &impl_GNOME_ClientSite_vepv;
   newservant->poa = poa;
   newservant->embeddable = embeddable;
   newservant->container_servant = container_servant;

   POA_GNOME_ClientSite__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = newservant;

   return retval;
}

static void
impl_GNOME_ClientSite__destroy(impl_POA_GNOME_ClientSite * servant,
       CORBA_Environment * ev)
{
   PortableServer_ObjectId *objid;

   objid = PortableServer_POA_servant_to_id(servant->poa, servant, ev);
   PortableServer_POA_deactivate_object(servant->poa, objid, ev);
   CORBA_free(objid);

   POA_GNOME_ClientSite__fini((PortableServer_Servant) servant, ev);
   g_free(servant);
}

static GNOME_Container
impl_GNOME_ClientSite_get_container(impl_POA_GNOME_ClientSite * servant,
```

```
    CORBA_Environment * ev)
{
   GNOME_Container retval;

   return retval;
}

static void
impl_GNOME_ClientSite_show_window(impl_POA_GNOME_ClientSite * servant,
  CORBA_boolean shown,
  CORBA_Environment * ev)
{
  /* store whether the component will be
     edited in-place or out-place */
  servant->show = shown;

}

static void
impl_GNOME_ClientSite_ref(impl_POA_GNOME_ClientSite * servant,
  CORBA_Environment * ev)
{
}

static void
impl_GNOME_ClientSite_unref(impl_POA_GNOME_ClientSite * servant,
    CORBA_Environment * ev)
{
}

static CORBA_Object
impl_GNOME_ClientSite_query_interface(impl_POA_GNOME_ClientSite * servant,
      CORBA_char * repoid,
      CORBA_Environment * ev)
{
   CORBA_Object retval;

   return retval;
}

static impl_POA_GNOME_ViewFrame *
impl_GNOME_ViewFrame__create(PortableServer_POA poa,
     impl_POA_GNOME_ClientSite *client_site_servant,
     CORBA_Environment * ev)
{
   impl_POA_GNOME_ViewFrame *retval ;
   impl_POA_GNOME_ViewFrame *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_GNOME_ViewFrame, 1);
   newservant->servant.vepv = &impl_GNOME_ViewFrame_vepv;
   newservant->poa = poa;
   newservant->client_site_servant = client_site_servant;

   /* create the GtkSocket for the embeddable view */
   newservant->socket = gtk_socket_new ();
   newservant->wrapper = gnome_wrapper_new ();
   gnome_wrapper_set_covered (GNOME_WRAPPER(newservant->wrapper),
      1);
   gtk_container_add (GTK_CONTAINER(newservant->wrapper),
      newservant->socket);
   gtk_signal_connect (GTK_OBJECT (newservant->wrapper),
       "button_press_event",
```

```
       GTK_SIGNAL_FUNC (view_frame_cb),
       newservant);


   POA_GNOME_ViewFrame__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = newservant;

   return retval;
}

static void
impl_GNOME_ViewFrame__destroy(impl_POA_GNOME_ViewFrame * servant,
      CORBA_Environment * ev)
{
   PortableServer_ObjectId *objid;

   objid = PortableServer_POA_servant_to_id(servant->poa, servant, ev);
   PortableServer_POA_deactivate_object(servant->poa, objid, ev);
   CORBA_free(objid);

   POA_GNOME_ViewFrame__fini((PortableServer_Servant) servant, ev);
   g_free(servant);
}

static GNOME_ClientSite
impl_GNOME_ViewFrame_get_client_site(impl_POA_GNOME_ViewFrame * servant,
     CORBA_Environment * ev)
{
   GNOME_ClientSite retval;

   return retval;
}

static void
impl_GNOME_ViewFrame_view_activated(impl_POA_GNOME_ViewFrame * servant,
    CORBA_boolean state,
    CORBA_Environment * ev)
{
  servant->activate = state;
  if (state == 1)
  gnome_wrapper_set_covered (GNOME_WRAPPER(servant->wrapper),
     0);
  else if (state == 0    )
    gnome_wrapper_set_covered (GNOME_WRAPPER(servant->wrapper),
       1);

}

static void
impl_GNOME_ViewFrame_deactivate_and_undo(impl_POA_GNOME_ViewFrame * servant,
 CORBA_Environment * ev)
{
}

static void
impl_GNOME_ViewFrame_request_resize(impl_POA_GNOME_ViewFrame * servant,
    CORBA_short new_width,
    CORBA_short new_height,
    CORBA_Environment * ev)
{
  servant->request.width = new_width;
```

```
  servant->request.height = new_height;
}

static void
impl_GNOME_ViewFrame_activate_uri(impl_POA_GNOME_ViewFrame * servant,
  CORBA_char * uri,
  CORBA_boolean relative,
  CORBA_Environment * ev)
{
}

static void
impl_GNOME_ViewFrame_ref(impl_POA_GNOME_ViewFrame * servant,
 CORBA_Environment * ev)
{
}

static void
impl_GNOME_ViewFrame_unref(impl_POA_GNOME_ViewFrame * servant,
   CORBA_Environment * ev)
{
}

static CORBA_Object
impl_GNOME_ViewFrame_query_interface(impl_POA_GNOME_ViewFrame * servant,
     CORBA_char * repoid,
     CORBA_Environment * ev)
{
   CORBA_Object retval;

   return retval;
}

int main (int argc, char **argv)
{
  CORBA_Environment ev;
  CORBA_ORB orb;
  CORBA_Object root_poa;
  PortableServer_POAManager root_poa_manager;

  CORBA_Object obj;

  CORBA_exception_init (&ev);
  orb = gnome_CORBA_init ("a \"simple\" container",
  "1.0",
  &argc,
  argv,
  GNORBA_INIT_SERVER_FUNC,
  &ev);

  root_poa = CORBA_ORB_resolve_initial_references (orb, "RootPOA", &ev);

  root_poa_manager = PortableServer_POA__get_the_POAManager (
                    (PortableServer_POA) root_poa, &ev);
  PortableServer_POAManager_activate (root_poa_manager, &ev);

  /* this function will create all the other
   * needed interfaces
   */
  obj = impl_GNOME_Container__create ((PortableServer_POA)
      root_poa,
      &ev);
```

```
  CORBA_exception_free (&ev);
  gtk_main();
  return 0;
}
```

component.c

```
#include <gnome.h>
#include <libgnorba/gnorba.h>
#include <ORBitservices/CosNaming.h>


#include "bonobo.h"

/*** App-specific servant structures ***/

typedef struct
{
   POA_GNOME_Persist servant;
   PortableServer_POA poa;

}
impl_POA_GNOME_Persist;

typedef struct
{
  POA_GNOME_Embeddable servant;
  PortableServer_POA poa;

  GNOME_ClientSite client_site;
  gchar data[11];
  GSList *view_servant_list;
  gchar *host_name;
  gchar *uri;
}
impl_POA_GNOME_Embeddable;

typedef struct
{
  POA_GNOME_View servant;
  PortableServer_POA poa;

  impl_POA_GNOME_Embeddable *embeddable_servant;
  GNOME_ViewFrame frame;
  GtkWidget *widget;
  GtkWidget *plug;

}
impl_POA_GNOME_View;



typedef struct
{
   POA_GNOME_EmbeddableFactory servant;
   PortableServer_POA poa;
}
impl_POA_GNOME_EmbeddableFactory;
```

```
/*** Implementation stub prototypes ***/


static void impl_GNOME_View__destroy(impl_POA_GNOME_View * servant,
     CORBA_Environment * ev);

static void
impl_GNOME_View_size_allocate(impl_POA_GNOME_View * servant,
      CORBA_short width,
      CORBA_short height, CORBA_Environment * ev);

static void
impl_GNOME_View_size_query(impl_POA_GNOME_View * servant,
   CORBA_short * desired_width,
   CORBA_short * desired_height,

   CORBA_Environment * ev);

static void
impl_GNOME_View_set_window(impl_POA_GNOME_View * servant,
   GNOME_View_windowid id,
   CORBA_Environment * ev);

static void
impl_GNOME_View_activate(impl_POA_GNOME_View * servant,
 CORBA_boolean activated,
 CORBA_Environment * ev);

static void
impl_GNOME_View_reactivate_and_undo(impl_POA_GNOME_View * servant,
    CORBA_Environment * ev);

static void
impl_GNOME_View_do_verb(impl_POA_GNOME_View * servant,
CORBA_char * verb_name,
CORBA_Environment * ev);

static void
impl_GNOME_View_set_zoom_factor(impl_POA_GNOME_View * servant,
CORBA_double zoom,
CORBA_Environment * ev);

static void
impl_GNOME_View_ref(impl_POA_GNOME_View * servant,
    CORBA_Environment * ev);
static void
impl_GNOME_View_unref(impl_POA_GNOME_View * servant,
      CORBA_Environment * ev);
static CORBA_Object
impl_GNOME_View_query_interface(impl_POA_GNOME_View * servant,
CORBA_char * repoid,
CORBA_Environment * ev);

static void impl_GNOME_Embeddable__destroy(impl_POA_GNOME_Embeddable *
   servant,
   CORBA_Environment * ev);
static void impl_GNOME_Embeddable_set_client_site(impl_POA_GNOME_Embeddable *
  servant,
  GNOME_ClientSite
  client_site,
  CORBA_Environment * ev);
```

```
static GNOME_ClientSite
impl_GNOME_Embeddable_get_client_site(impl_POA_GNOME_Embeddable * servant,
      CORBA_Environment * ev);

static void
impl_GNOME_Embeddable_set_host_name(impl_POA_GNOME_Embeddable * servant,
    CORBA_char * name,
    CORBA_char * appname,
    CORBA_Environment * ev);

static void
impl_GNOME_Embeddable_set_uri(impl_POA_GNOME_Embeddable * servant,
      CORBA_char * uri,
      CORBA_Environment * ev);

static void
impl_GNOME_Embeddable_close(impl_POA_GNOME_Embeddable * servant,
    GNOME_Embeddable_CloseMode mode,
    CORBA_Environment * ev);

static GNOME_Embeddable_verb_list
   *impl_GNOME_Embeddable_get_verb_list(impl_POA_GNOME_Embeddable * servant,
CORBA_Environment * ev);

static GNOME_View
impl_GNOME_Embeddable_new_view(impl_POA_GNOME_Embeddable * servant,
       GNOME_ViewFrame frame,
       CORBA_Environment * ev);

static void
impl_GNOME_Embeddable_ref(impl_POA_GNOME_Embeddable * servant,
  CORBA_Environment * ev);
static void
impl_GNOME_Embeddable_unref(impl_POA_GNOME_Embeddable * servant,
    CORBA_Environment * ev);
static CORBA_Object
impl_GNOME_Embeddable_query_interface(impl_POA_GNOME_Embeddable * servant,
      CORBA_char * repoid,
      CORBA_Environment * ev);

static void
impl_GNOME_EmbeddableFactory__destroy(impl_POA_GNOME_EmbeddableFactory *
      servant,
      CORBA_Environment * ev);
static CORBA_boolean
impl_GNOME_EmbeddableFactory_supports(impl_POA_GNOME_EmbeddableFactory *
      servant,
      CORBA_char * obj_goad_id,
      CORBA_Environment * ev);
static CORBA_Object
impl_GNOME_EmbeddableFactory_create_object(impl_POA_GNOME_EmbeddableFactory *
   servant,
   CORBA_char * goad_id,
   GNOME_stringlist * params,
   CORBA_Environment * ev);

/*** epv structures ***/

static PortableServer_ServantBase__epv impl_GNOME_View_base_epv = {
   NULL, /* _private data */
   NULL, /* finalize routine */
   NULL, /* default_POA routine */
};
```

```
static POA_GNOME_View__epv impl_GNOME_View_epv = {
   NULL, /* _private */

   (gpointer) & impl_GNOME_View_size_allocate,

   (gpointer) & impl_GNOME_View_size_query,

   (gpointer) & impl_GNOME_View_set_window,

   (gpointer) & impl_GNOME_View_activate,

   (gpointer) & impl_GNOME_View_reactivate_and_undo,

   (gpointer) & impl_GNOME_View_do_verb,

   (gpointer) & impl_GNOME_View_set_zoom_factor,

};
static POA_GNOME_Unknown__epv impl_GNOME_View_GNOME_Unknown_epv = {
   NULL, /* _private */
   (gpointer) & impl_GNOME_View_ref,
   (gpointer) & impl_GNOME_View_unref,
   (gpointer) & impl_GNOME_View_query_interface,
};
static PortableServer_ServantBase__epv impl_GNOME_Embeddable_base_epv = {
   NULL, /* _private data */
   NULL, /* finalize routine */
   NULL, /* default_POA routine */
};
static POA_GNOME_Embeddable__epv impl_GNOME_Embeddable_epv = {
   NULL, /* _private */
   (gpointer) & impl_GNOME_Embeddable_set_client_site,

   (gpointer) & impl_GNOME_Embeddable_get_client_site,

   (gpointer) & impl_GNOME_Embeddable_set_host_name,

   (gpointer) & impl_GNOME_Embeddable_set_uri,

   (gpointer) & impl_GNOME_Embeddable_close,

   (gpointer) & impl_GNOME_Embeddable_get_verb_list,

   (gpointer) & impl_GNOME_Embeddable_new_view,

};
static POA_GNOME_Unknown__epv impl_GNOME_Embeddable_GNOME_Unknown_epv = {
   NULL, /* _private */
   (gpointer) & impl_GNOME_Embeddable_ref,
   (gpointer) & impl_GNOME_Embeddable_unref,
   (gpointer) & impl_GNOME_Embeddable_query_interface,
};
static PortableServer_ServantBase__epv impl_GNOME_EmbeddableFactory_base_epv = {
   NULL, /* _private data */
   NULL, /* finalize routine */
   NULL, /* default_POA routine */
};
static POA_GNOME_EmbeddableFactory__epv impl_GNOME_EmbeddableFactory_epv = {
   NULL, /* _private */
};
static POA_GNOME_GenericFactory__epv
   impl_GNOME_EmbeddableFactory_GNOME_GenericFactory_epv = {
   NULL, /* _private */
```

```
   (gpointer) & impl_GNOME_EmbeddableFactory_supports,
   (gpointer) & impl_GNOME_EmbeddableFactory_create_object,
};

/*** vepv structures ***/

static POA_GNOME_View__vepv impl_GNOME_View_vepv = {
   &impl_GNOME_View_base_epv,
   &impl_GNOME_View_GNOME_Unknown_epv,
   &impl_GNOME_View_epv,
};
static POA_GNOME_Embeddable__vepv impl_GNOME_Embeddable_vepv = {
   &impl_GNOME_Embeddable_base_epv,
   &impl_GNOME_Embeddable_GNOME_Unknown_epv,
   &impl_GNOME_Embeddable_epv,
};
static POA_GNOME_EmbeddableFactory__vepv impl_GNOME_EmbeddableFactory_vepv = {
   &impl_GNOME_EmbeddableFactory_base_epv,
   &impl_GNOME_EmbeddableFactory_GNOME_GenericFactory_epv,
   &impl_GNOME_EmbeddableFactory_epv,
};

/*** Stub implementations ***/



static impl_POA_GNOME_View *
impl_GNOME_View__create(PortableServer_POA poa,
GNOME_ViewFrame frame,
impl_POA_GNOME_Embeddable *embeddable_servant,
CORBA_Environment * ev)
{

   impl_POA_GNOME_View *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_GNOME_View, 1);
   newservant->servant.vepv = &impl_GNOME_View_vepv;
   newservant->poa = poa;
   newservant->frame = CORBA_Object_duplicate (frame, ev);
   /* this is what will be displayed in the component view ! */
   newservant->embeddable_servant = embeddable_servant;
   newservant->widget = gtk_button_new_with_label (embeddable_servant-
>data);

   POA_GNOME_View__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);


   return newservant;
}

static void
impl_GNOME_View__destroy(impl_POA_GNOME_View * servant,
 CORBA_Environment * ev)
{
   PortableServer_ObjectId *objid;

   objid = PortableServer_POA_servant_to_id(servant->poa, servant, ev);
   PortableServer_POA_deactivate_object(servant->poa, objid, ev);
   CORBA_free(objid);
```

```
   POA_GNOME_View__fini((PortableServer_Servant) servant, ev);
   g_free(servant);
}

static void
impl_GNOME_View_size_allocate(impl_POA_GNOME_View * servant,
      CORBA_short width,
      CORBA_short height,
      CORBA_Environment * ev)
{
}

static void
impl_GNOME_View_size_query(impl_POA_GNOME_View * servant,
   CORBA_short * desired_width,
   CORBA_short * desired_height,
   CORBA_Environment * ev)
{
}

static void
impl_GNOME_View_set_window(impl_POA_GNOME_View * servant,
   GNOME_View_windowid id,
   CORBA_Environment * ev)
{
  servant->plug = gtk_plug_new (id);
  gtk_container_add (GTK_CONTAINER(servant->plug),
     servant->widget);
  gtk_widget_show_all (servant->plug);
}


void cancel_cb (GtkWidget *widget,
gpointer data)
{
  CORBA_Environment ev;
  GtkWidget *window = data;
  impl_POA_GNOME_View *view_servant = (impl_POA_GNOME_View *)
    gtk_object_get_data (GTK_OBJECT (window),
 "view_servant");

  CORBA_exception_init (&ev);
  gtk_widget_destroy (window);

  GNOME_ViewFrame_view_activated (view_servant->frame,
  0,
  &ev);
  CORBA_exception_free (&ev);
}

void apply_cb (GtkWidget *button,
       gpointer data)
{
  GSList *list;
  impl_POA_GNOME_View * servant;
  GtkWidget *label;
  GtkWidget *entry = (GtkWidget *) data;
  GtkWidget *window = gtk_object_get_data (GTK_OBJECT(button),
   "window");
  impl_POA_GNOME_View *view_servant = (impl_POA_GNOME_View *)
    gtk_object_get_data (GTK_OBJECT (window),
 "view_servant");
  strcpy (view_servant->embeddable_servant->data,
```

```
  gtk_entry_get_text (GTK_ENTRY (entry)));

  list = view_servant->embeddable_servant->view_servant_list;
  while (list != g_slist_last (view_servant->embeddable_servant->view_servant_list)) {
    servant = (impl_POA_GNOME_View *)list->data;
    label = GTK_BIN(servant->widget)->child;

    gtk_label_set_text (GTK_LABEL(label),
view_servant->embeddable_servant->data);
    list = list->next;
  }
  servant = (impl_POA_GNOME_View *)list->data;
  label = GTK_BIN(servant->widget)->child;
  gtk_label_set_text (GTK_LABEL(label),
      view_servant->embeddable_servant->data);


}

void ok_cb (GtkWidget *button,
    gpointer data)
{
  GtkWidget *entry = (GtkWidget *) data;
  apply_cb (button, entry);
  cancel_cb (button,
     gtk_object_get_data (GTK_OBJECT (button),
  "window"));
}


static void
impl_GNOME_View_activate(impl_POA_GNOME_View * servant,
 CORBA_boolean activated,
 CORBA_Environment * ev)
{
  static GtkWidget *window = NULL;
  GtkWidget *vbox;
  GtkWidget *hbox;
  GtkWidget *button;
  GtkWidget *entry;

  if (activated == 1) {

    /* activate view */
    /* sample code to create the external editing
       window of the view */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_object_set_data (GTK_OBJECT (window),
 "view_servant",
 servant);
    gtk_signal_connect (GTK_OBJECT (window),
"destroy",
GTK_SIGNAL_FUNC (cancel_cb),
window);

    gtk_signal_connect (GTK_OBJECT (window),
"delete_event",
GTK_SIGNAL_FUNC (cancel_cb),
window);

    vbox = gtk_vbox_new (TRUE, TRUE);
    hbox = gtk_hbox_new (TRUE, TRUE);
```

```
    entry = gtk_entry_new_with_max_length (10);
    gtk_entry_set_text (GTK_ENTRY (entry),
servant->embeddable_servant->data);
    gtk_box_pack_start (GTK_BOX (vbox),
entry,
TRUE,
0,
0);

    gtk_box_pack_start (GTK_BOX (vbox),
gtk_hseparator_new(),
TRUE,
0,
0);

    gtk_box_pack_end (GTK_BOX(vbox), hbox,
      TRUE,
      0,
      0);
    button = gtk_button_new_with_label ("Cancel");
    gtk_box_pack_end (GTK_BOX(hbox),
      button,
      TRUE,
      0,
      0);
    gtk_signal_connect (GTK_OBJECT(button),
"clicked",
GTK_SIGNAL_FUNC(cancel_cb),
window);
    button = gtk_button_new_with_label ("Apply");
    gtk_object_set_data (GTK_OBJECT (button),
 "window",
 window);
    gtk_box_pack_end (GTK_BOX(hbox),
      button,
      TRUE,
      0,
      0);
    gtk_signal_connect (GTK_OBJECT(button),
"clicked",
GTK_SIGNAL_FUNC(apply_cb),
entry);
    button = gtk_button_new_with_label ("Ok");
    gtk_object_set_data (GTK_OBJECT (button),
 "window",
 window);
    gtk_box_pack_end (GTK_BOX(hbox),
      button,
      TRUE,
      0,
      0);
    gtk_signal_connect (GTK_OBJECT(button),
"clicked",
GTK_SIGNAL_FUNC(ok_cb),
entry);

    gtk_container_add (GTK_CONTAINER(window),
       vbox);
    gtk_widget_show_all (GTK_WIDGET(window));

  } else if (activated == 0 && window != NULL) {
    gtk_widget_destroy (window);
  }
```

```
  /* tell the container the view was activated or not*/
  GNOME_ViewFrame_view_activated (servant->frame,
  activated,
  ev);
}

static void
impl_GNOME_View_reactivate_and_undo(impl_POA_GNOME_View * servant,
    CORBA_Environment * ev)
{
  /* i am lazy: i don't want to implement this... */
}

static void
impl_GNOME_View_do_verb(impl_POA_GNOME_View * servant,
CORBA_char * verb_name,
CORBA_Environment * ev)
{
  /* i am lazy: i don't want to implement this... */
}

static void
impl_GNOME_View_set_zoom_factor(impl_POA_GNOME_View * servant,
CORBA_double zoom,
CORBA_Environment * ev)
{
  /* i am lazy: i don't want to implement this... */
}

static void
impl_GNOME_View_ref(impl_POA_GNOME_View * servant,
    CORBA_Environment * ev)
{
}

static void
impl_GNOME_View_unref(impl_POA_GNOME_View * servant,
      CORBA_Environment * ev)
{
}

static CORBA_Object
impl_GNOME_View_query_interface(impl_POA_GNOME_View * servant,
CORBA_char * repoid,
CORBA_Environment * ev)
{
   CORBA_Object retval;

   return retval;
}

static GNOME_Embeddable
impl_GNOME_Embeddable__create(PortableServer_POA poa,
      CORBA_Environment * ev)
{
   GNOME_Embeddable retval;
   impl_POA_GNOME_Embeddable *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_GNOME_Embeddable, 1);
   newservant->servant.vepv = &impl_GNOME_Embeddable_vepv;
   newservant->poa = poa;
   strcpy (newservant->data, "Ok ?!");;
```

```
   POA_GNOME_Embeddable__init((PortableServer_Servant) newservant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

   return retval;
}

static void
impl_GNOME_Embeddable__destroy(impl_POA_GNOME_Embeddable * servant,
       CORBA_Environment * ev)
{
   PortableServer_ObjectId *objid;

   objid = PortableServer_POA_servant_to_id(servant->poa, servant, ev);
   PortableServer_POA_deactivate_object(servant->poa, objid, ev);
   CORBA_free(objid);

   POA_GNOME_Embeddable__fini((PortableServer_Servant) servant, ev);
   g_free(servant);
}

static void
impl_GNOME_Embeddable_set_client_site(impl_POA_GNOME_Embeddable * servant,
      GNOME_ClientSite client_site,
      CORBA_Environment * ev)
{
  servant->client_site = CORBA_Object_duplicate (client_site, ev);
  /* tell the container that the embeddable will
     be activated in an external window */
  GNOME_ClientSite_show_window (client_site, 1, ev);
}

static GNOME_ClientSite
impl_GNOME_Embeddable_get_client_site(impl_POA_GNOME_Embeddable * servant,
      CORBA_Environment * ev)
{
   GNOME_ClientSite retval;

   return retval;
}

static void
impl_GNOME_Embeddable_set_host_name(impl_POA_GNOME_Embeddable * servant,
    CORBA_char * name,
    CORBA_char * appname,
    CORBA_Environment * ev)
{
  servant->host_name = g_strdup (name);

}

static void
impl_GNOME_Embeddable_set_uri(impl_POA_GNOME_Embeddable * servant,
      CORBA_char * uri,
      CORBA_Environment * ev)
{
  servant->uri = g_strdup (uri);

}

static void
```

```
impl_GNOME_Embeddable_close(impl_POA_GNOME_Embeddable * servant,
    GNOME_Embeddable_CloseMode mode,
    CORBA_Environment * ev)
{
  /* there, i must close the external editing application... */
}

static GNOME_Embeddable_verb_list *
impl_GNOME_Embeddable_get_verb_list(impl_POA_GNOME_Embeddable * servant,
    CORBA_Environment * ev)
{
   GNOME_Embeddable_verb_list *retval;
   /* our embeddable will not support
      any non-standard action
      ie: it won't export any verb
      thus, return an empty verb_list
   */

   return retval;
}

static GNOME_View
impl_GNOME_Embeddable_new_view(impl_POA_GNOME_Embeddable * servant,
       GNOME_ViewFrame frame,
       CORBA_Environment * ev)
{
   GNOME_View view;
   impl_POA_GNOME_View *view_servant;
   view_servant = impl_GNOME_View__create (servant->poa,
   frame,
   servant,
   ev);
   servant->view_servant_list = g_slist_append (servant->view_servant_list,
view_servant);

   view = PortableServer_POA_servant_to_reference(servant->poa, view_servant, ev);

   return view;
}

static void
impl_GNOME_Embeddable_ref(impl_POA_GNOME_Embeddable * servant,
  CORBA_Environment * ev)
{
}

static void
impl_GNOME_Embeddable_unref(impl_POA_GNOME_Embeddable * servant,
    CORBA_Environment * ev)
{
}

static CORBA_Object
impl_GNOME_Embeddable_query_interface(impl_POA_GNOME_Embeddable * servant,
      CORBA_char * repoid,
      CORBA_Environment * ev)
{
   CORBA_Object retval;

   return retval;
}

static GNOME_EmbeddableFactory
```

```
impl_GNOME_EmbeddableFactory__create(PortableServer_POA poa,
     CORBA_Environment * ev)
{
   GNOME_EmbeddableFactory retval;
   impl_POA_GNOME_EmbeddableFactory *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_GNOME_EmbeddableFactory, 1);
   newservant->servant.vepv = &impl_GNOME_EmbeddableFactory_vepv;
   newservant->poa = poa;
   POA_GNOME_EmbeddableFactory__init((PortableServer_Servant) newser-
vant, ev);
   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

   return retval;
}

static void
impl_GNOME_EmbeddableFactory__destroy(impl_POA_GNOME_EmbeddableFactory *
      servant,
      CORBA_Environment * ev)
{
   PortableServer_ObjectId *objid;

   objid = PortableServer_POA_servant_to_id(servant->poa, servant, ev);
   PortableServer_POA_deactivate_object(servant->poa, objid, ev);
   CORBA_free(objid);

   POA_GNOME_EmbeddableFactory__fini((PortableServer_Servant) servant, ev);
   g_free(servant);
}

static CORBA_boolean
impl_GNOME_EmbeddableFactory_supports(impl_POA_GNOME_EmbeddableFactory *
      servant, CORBA_char * obj_goad_id,
      CORBA_Environment * ev)
{
   CORBA_boolean retval;

   return retval;
}

static CORBA_Object
impl_GNOME_EmbeddableFactory_create_object(impl_POA_GNOME_EmbeddableFactory *
   servant, CORBA_char * goad_id,
   GNOME_stringlist * params,
   CORBA_Environment * ev)
{
   CORBA_Object retval = CORBA_OBJECT_NIL;

   retval = impl_GNOME_Embeddable__create (servant->poa, ev);
   /* Use GNOME's default name server */
   goad_server_register (CORBA_OBJECT_NIL, retval, goad_id, "server", ev);

   return retval;
}


int main (int argc, char **argv)
{
  CORBA_Environment ev;
```

```
  CORBA_ORB orb;
  CORBA_Object root_poa;
  PortableServer_POAManager root_poa_manager;
  CosNaming_NamingContext name_server;
  CORBA_Object obj;

  CORBA_exception_init (&ev);
  orb = gnome_CORBA_init ("a \"simple\" container",
  "1.0",
  &argc,
  argv,
  GNORBA_INIT_SERVER_FUNC,
  &ev);

  root_poa = CORBA_ORB_resolve_initial_references (orb,
   "RootPOA",
   &ev);

  root_poa_manager = PortableServer_POA__get_the_POAManager (
     (PortableServer_POA) root_poa, &ev);
  PortableServer_POAManager_activate (root_poa_manager, &ev);

  /* this function will create all the other
   * needed interfaces
   */
  obj = impl_GNOME_EmbeddableFactory__create ((PortableServer_POA)
      root_poa,
      &ev);
  /* register against the name service */
  name_server = gnome_name_service_get ();
  goad_server_register (name_server, obj,
goad_server_activation_id(),
"server",
&ev);
  /* finished !! */
  CORBA_exception_free (&ev);
  gtk_main ();
  return 0;
}
```

Appendix E. Copyright Notice

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

```
Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable
    source code, which must be distributed under the terms of Sections
    1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three
    years, to give any third party, for a charge no more than your
    cost of physically performing source distribution, a complete
    machine-readable copy of the corresponding source code, to be
    distributed under the terms of Sections 1 and 2 above on a medium
    customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer
    to distribute corresponding source code.  (This alternative is
    allowed only for noncommercial distribution and only if you
    received the program in object code or executable form with such
    an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it.  For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to
control compilation and installation of the executable.  However, as a
special exception, the source code distributed need not include
anything that is normally distributed (in either source or binary
form) with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is made by offering
access to copy from a designated place, then offering equivalent
access to copy the source code from the same place counts as
distribution of the source code, even though third parties are not
compelled to copy the source along with the object code.

  4. You may not copy, modify, sublicense, or distribute the Program
except as expressly provided under this License.  Any attempt
otherwise to copy, modify, sublicense or distribute the Program is
void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under
this License will not have their licenses terminated so long as such
parties remain in full compliance.

  5. You are not required to accept this License, since you have not
signed it.  However, nothing else grants you permission to modify or
distribute the Program or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Program (or any work based on the
Program), you indicate your acceptance of this License to do so, and
```

```
all its terms and conditions for copying, distributing or modifying
the Program or works based on it.

  6. Each time you redistribute the Program (or any work based on the
Program), the recipient automatically receives a license from the
original licensor to copy, distribute or modify the Program subject to
these terms and conditions.  You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties to
this License.

  7. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Program at all.  For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices.  Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Program under this License
may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded.  In such case, this License incorporates
the limitation as if written in the body of this License.

  9. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time.  Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number.  If the Program
specifies a version number of this License which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation.  If the Program does not specify a version number of
this License, you may choose any version ever published by the Free Software
Foundation.
```

```
If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) year  name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate
parts of the General Public License.  Of course, the commands you use may
be called something other than ‘show w’ and ‘show c’; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

  (signature of Ty Coon), 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Library General
Public License instead of this License.
```

Bibliography

Index

IDL Basics