Interface Definition Language

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

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

Section 1. Introduction

Who should take this tutorial?

This tutorial gives a hands-on introduction to the basic building blocks for applications that communicate two ways using Web protocols. If you are working on dynamic Web applications or distributed programming, this tutorial will get you started.

CORBA Interface Definition Language (IDL) is the prevalent language used for defining how components connect together. Beyond its use in CORBA systems, IDL has proven a popular way to describe platform and language-neutral connection interfaces, including the Document Object Model (DOM) -- the core API for XML. Even variations on IDL, such as the one used by Component Object Model (COM), tend to be similar to IDL. Understanding IDL brings about key insights to many of the techniques of component programming.

If you are not very familiar with component development but you plan to do some work in this area, this tutorial is recommended. It is also recommended for those who will be writing XML programs using the DOM.

Navigation

Navigating through the tutorial is easy:

* Use the Next and Previous buttons to move forward and backward through the tutorial.
* When you're finished with a section, select Next section for the next section. Within a section, use the Section menu button to see the contents of that section. You can return to the main menu at any time by clicking the Main menu button.
* If you'd like to tell us what you think, or if you have a question for the author about the content of the tutorial, use the Feedback button.

Prerequisites

At its core, IDL is quite simple. If you are familiar with any object-oriented language such as C++, Java, or Python you probably have a decent idea what it specifies. No software is required for this tutorial.

Section 2. Planning and preparation

What is an interface?

The interface is arguably the most important aspect of component development. The interface determines the strict form of all remote requests and responses. It can be very expensive to modify or even extend an interface after it has been deployed widely. By contrast, changes in implementation can often be made to one code base without affecting any other code base, as long as the interface is maintained. Such changes tend to be far less expensive.

How do I make an interface?

It is very important to get the interface right when developing components, and this applies in the case of XML messaging as well. Traditionally, interfaces have been defined in many ways, including:

* Very specific prose description
* CORBA Interface Definition Language (IDL)
* Unified Modeling Language (UML)
* Various XML schema and data-typing languages

IDL is probably the most common and well known, and it is certainly the most broadly implemented.

An example

As our example, we'll design an interface for employee information. First of all, since we said that designing the interface is very important, we'll spend some time planning what we need from this component.

The sales commission authorization application

Let's say we want to build a simple employee application for a sales department. It will be used to authorize commission payments for sales. The software developers and a few folks from human resources and accounting sit around the table and brainstorm what objects the application needs to model. The first obvious object is the salesperson. From our brainstorming we get the following list:

* Every salesperson has a unique ID
* The salesperson's name is important
* The salesperson's social security number is important
* The salesperson needs to be paid commission based on their sales
* Each salesperson is in a particular department

Sticking to the essentials

These are all points that are relevant to how we want salesperson objects to behave in our applications. It may be important for a departmental listing to have employee home telephone numbers on file in the system. But this is not an issue for our current application, so it is left out of the above list. Note that one could have different aspects of employee objects represented in different interfaces, possibly to different applications. If we think of this while we're designing our interface, we'll want to have a unique identifier that can bridge all the various representations of the employee or salesperson object.

In fact, for this reason, let's imagine our salesperson interface as simply one form of employee interface. Other information relevant to the employee, such as home phone number and address, could be represented in a parallel interface that we can easily bind in our applications if necessary.

Try it yourself

Of course, no two groups of people considering even such a simple example would come up with the same interface. As an exercise, list the various objects and aspects of such objects you might need to model in an interface to an application for authorizing sales commissions.

Almost certainly your list will have differences. This is why developing interfaces is such a hard task. You have to exclude some things that you might think are natural, and think hard so as not to forget others that don't immediately come to mind.

Object relationships

Once the list of the core objects comes to mind, it's time to look for relationships. Some of the objects you mentioned in your list of relevant employee characteristics themselves need to be recognized as objects by the application.

In our example we mentioned that each employee is in a particular department. If we think about our application, we realize that the department is something we probably want to treat as a separate object in our application. We'd like to organize reports by department. We'd like to be able to locate employees by department, etc.

So "Department" is established as an object related to employee. What other object relationships do you find in your own list of employee characteristics?

Section 3. IDL basics

Constructing IDL: The module

This is the outline of how the IDL will look based on our employee component design.

```
module EmployeeInfoServer {

  //More code goes here

}
```

The `module` specification groups closely related interface elements. It usually corresponds to modules in the relevant programming language, such as Java or Python. You would group related classes, exceptions, data types and other such matter in a single module.

We'll be adding more to our IDL, in the spot we've marked with a comment to this effect. A comment in IDL either consists of any text on a line following `//` or any text at all between `/*` and `*/`.

Forward declarations

First we'll add two *forward declarations* to the IDL:

```
  interface Employee;
  interface Department;
```

A forward declaration basically foreshadows a more complete definition to follow, and can be used by the IDL parser if we need to refer to an object that has not yet been defined. It is a good idea to have a forward declaration for each distinct business object that will be represented in the IDL. Each one will have a full *interfaces* definition farther down.

In our case we had the `Employee` and `Department` objects, which are represented in appropriate forward declarations.

Exceptions

Exceptions are used to signal unexpected conditions during processing. An unexpected condition could be an error (say the server ran out of memory), or it could simply be an event that has to be handled in a very different way. For instance, it might be an exception if the amount of an employee's commission causes the department to exceed its budget.

Adding an exception

```
exception EmployeeInfoException {
  string message;
};
```

We define an exception of name and type `EmployeeInfoException`. Exceptions we define can be raised by a method call. This means that the client code that makes a call to a server might not get a normal function return. Rather, an exception may be signaled. This is the normal method of defining error handling using IDL.

Section 4. Modeling objects

IDL interfaces

Interfaces are the meat of IDL. Each one gives a detailed map for accessing a given type of object. First comes the header, with the interface name:

```
interface Employee {
  //interface body
};
```

Methods, part 1

The first two lines in the Employee interface body are methods requesting data from the represented object. The first requests the employee's ID, a simple integer.

```
  unsigned long getId();
```

Methods, part 2

The second method the interface body requests the employee's department.

```
  Department getDepartment();
```

Note that the return value is the Department interface, which we've already mentioned in a forward declaration, and which we'll fully define soon.

Object stubs and strong typing

When an interface is specified as a return value from a method, as with Department in the last panel, it means that some representation of an object with that interface will be returned, which you can further manipulate. This representation is usually called a *stub*.

As you can see, everything we specify in the IDL has a type associated with it. This is called *strong typing*. The interface is a contract that mandates how one interacts with an object regardless of its location or how it is implemented.

The IDL so far

We now have a framework for the basic employee object, and we are a little over halfway through the sample IDL.

```
module EmployeeInfoServer {
  interface Employee;
  interface Department;

  exception EmployeeInfoException {
    string message;
  };

  interface Employee {
    unsigned long getId();
    Department getDepartment();
    //More to come in this interface
  };

  //And more to come over all

};
```

Section 5. Digging more deeply

A more complex method

So far our methods have been quite simple, merely returning a requested value. The next couple of lines in the Employee interface are a bit more interesting.

```
float authorizeCommission(in float saleVolume)
  raises (EmployeeInfoException);
```

Parameters and remote operation

`authorizeCommission` is a method that probably requires more processing on the back end. The code that calls this method must provide a parameter (unlike with the preceding two methods), which is a floating-point number representing the amount of a sale on which the caller authorizes commission.

The remote object will cause the commission amount to be computed based on all the complex factors that usually govern sales commissions. Perhaps it will also trigger an electronic fund transfer from the company's bank to that of the employee represented by the remote object.

Return values and exceptions

If all goes well, the return value of `authorizeCommission` is the amount of actual commission granted.

This method can raise an `EmployeeInfoException` exception. Again, this could be a programming error, say the method was called with a negative `saleVolume` parameter. It could be simply an unusual condition, perhaps if the employee has already reached his or her cap on commissions for the quarter.

Since we defined a `message` string as one of the attributes of the exception, `EmployeeInfoException`, the remote side can set this string to a description of the reason for the exception (for our information). What exactly this message is depends on what is done with it. If it is an error message to be displayed on a screen it will probably be something like "negative value given for saleVolume". If it's to be used for automated processing it might be something more unfriendly such as "data error 34533".

IDL attributes

We round our interface body with a couple of *attributes*.

```
  attribute string name;
  readonly attribute string ssn;
```

Each attribute represents atomic values of the remote object. This means that they behave as if they were simply stored in a data field to which the client is given access. This can be read/write access as in the `name` attribute above, or read-only as the `ssn` attribute. Both attributes are of string type.

Defining our own types

```
typedef sequence<Employee> EmployeeList;
```

In the next panel, we'll see a method that returns a list of employees for a particular department. We accomplish this in IDL by creating a type definition for a *sequence* of `Employee` object stubs, called `EmployeeList`.

One more interface

```
interface Department {
  unsigned long getId();
  attribute string name;
  EmployeeList employees();
};
```

There are few surprises here, except perhaps the `employees` method whose return value is of the `EmployeeList` type we just defined. This way our application can obtain the list of employees in that department by calling the `employees` method.

The complete IDL

That's all there is to it. Here then is our complete code so far:

```
module EmployeeInfoServer {
  interface Employee;
```

```
   interface Department;

   exception EmployeeInfoException {
     string message;
   };

   interface Employee {
     unsigned long getId();
     Department getDepartment();
     float authorizeCommission(in float saleVolume)
       raises (EmployeeInfoException);
     attribute string name;
     attribute string ssn;
   };

   typedef sequence<Employee> EmployeeList;

   interface Department {
     unsigned long getId();
     attribute string name;
     EmployeeList employees();
   };
 };
```

Your turn

As an exercise, write up an IDL file that models the objects and relationships you came up with in your own analysis of the application for commission authorization.

Section 6. Resources and feedback

Summary

As you can see, basic IDL is rather simple. There are many other features of IDL we have not covered, some quite simple and some extremely complex. We have focused on providing enough IDL background to cover the interface definitions introduced in this series. If you follow the path outlined in this tutorial in developing your own technology, you will find that IDL is a very effective way to establish the rules for communication.

IDL definitions are mostly designed for access of objects separated by a network, but IDL itself doesn't deal with any of the details of how this network is traversed. That is the realm of network protocols, of which there are a multitude. HTTP is the network protocol we shall mostly use in this series on XML messaging; it is covered in the next tutorial in this series.

Resources

* Visit the official page for CORBA information.
* See this *quick and dirty start to understanding CORBA* .
* Check out information and downloads for *JavaIDL* , a CORBA client and IDL compiler that comes with Java 2.
* See , by George Lebl, which covers the CORBA-based Bonobo component library, and includes a description of IDL.
* *idldoc* is a tool that generates HTML documentation from CORBA IDL.
* , an article by Dave Bartlett, offers a more advanced look at IDL and is a good place to go from here.
* , by Mike Olson, introduces CORBA, including an explanation of IDL.
* *omniORB* is an open-source CORBA ORB with a flexible IDL compiler, omniidl. It supports C++ and Python.
* *ORBit* is an open-source CORBA implementation that is used in the core of the GNOME project. It supports C, C++, Python Perl and Eiffel.
* *The W3C Document Object Model (DOM)* , an XML API expressed in IDL, is available on the W3C site.
* *The Mozilla XPIDL compiler* generates component specifications from a format much like CORBA IDL.

Giving feedback and finding out more

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

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

Colophon

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