Interface Definition Language

by David G. Messerschmitt

Supplementary section for Understanding Networked Applications: A First Course, Morgan Kaufmann, 1999.

An interface definition language (IDL) supports the specification and description of object interfaces. Systems programming languages like Java can also describe an interface (see "Java: A Systems Programming Language"). However, the major goal of an IDL is to *separate* the definition from any specific systems programming language, while the Java interface is particular to Java. For purposes of Chapters 10 and 11, an IDL is useful to define interfaces without consideration of implementation issues.

Distributed object management (DOM) also requires an IDL (see Chapter 16). One goal of DOM is to allow objects to interact even if they were implemented using different systems programming languages. IDL allows object interfaces to be published and advertised in a system programming language-independent way.

The first step in implementing an object is to describe its interface in IDL. There are then tools for translating that IDL description into a template for the object in any of several system programming languages, like C++ and Java. A programmer can then proceed to implement the internal processing and data structures by adding program code to that template.

This appendix describes one such IDL, that defined as part of the CORBA standard (see "One DOM Standard: CORBA" in Chapter 16). This is not a complete description, but includes sufficient detail to illustrate application architectures at the level of detail in Chapter 10.

Data types

IDL defines a number of basic data types because this is necessary to specify the structure of data passing through the interface. The following is a representative subset:

- **`float`**, **`double`**: single and double precision floating point, approximate representation of real numbers
- **`short`**, **`long`**: 16-bit and 32-bit representations of integers
- **`boolean`**: assumes one of two values, `TRUE` or `FALSE`
- **`string`**: an arbitrary length string of characters

A special bold font is used for keywords like **`float`**, which are reserved words used by the language. This bold fond is *not* part of the IDL specification, but will make clearer below what words are reserved and what words are created by the designer. The statement

```
typedef string Date;
```

simply states that `Date` is another name for **`string`**. The ability to rename types can make code more readable.

Often it is appropriate to create user-defined data types that assemble basic types and other user-defined types. The keyword for this is **`struct`**, which is followed by a list of the assembled types separated by “;” and delimited by “{}”.

Example… A phone number has an internal structure consisting of country and city codes and local number. It is useful to capture this structure—as opposed to storing the phone number as one character string—so that an application can construct phone numbers for different circumstances. For example, the application may want to easily construct only the local portion of the number. This can be accomplished by defining a **`struct`** representing the phone number, as in:

```
struct Phone_number {
      short prefix;
      short country_code;
      short city_code;
      short local_number;
      }
```

Subsequently, parameters and return values can be `Phone_number`’s, as a shorthand.

IDL interfaces

The keyword **`interface`** is used to define the interface to an object, and is followed by a list of attributes and methods separated by “;” and delimited by “{}”. In IDL, any line beginning with ”//” is a comment that helps understand the interface but is otherwise ignored.

Example… Let `Person` be an object interface that represents relevant information about an individual, like a citizen or customer. In a simple case, `Person` may have only three attributes and no methods:

```
interface Person {
//Interface to object representing information about a person
      attribute Name name;
      attribute Address address;
      attribute Phone_number phone_number;
      }
```

Note that each attribute has a type (like `Address`) and a name (like `address`). The types must either be basic types or must have been previously defined by a **`struct`** statement.

Methods at the interface are described as the name of the method followed by a list of parameters and return values (collectively called *arguments*) separated by “,” and delimited by “()”. A parameter is indicated by the keyword **`in`** and a return value by **`out`**, each followed by a data type and name. It is also possible to specify an **`inout`**, which means a value that is passed as a parameter, and also returned, its value possibly having been changed by the method. Each method also has a single unnamed return value. If there is in fact no return value, it can be declared as type **`void`**. Thus, there are three ways to get data back from a method: either as a return value, or as an **`out`** or **`inout`** argument.

Example… Suppose a method `add` does a simple addition of two **`long`** integers, returning the result. In the notation of Chapter 6, this method might be written

```
add: A,B → C, status
```

but in IDL it would be specified as

```
//Method adds to integers and returns their sum
//Method returns FALSE if the result cannot be trusted
```

```
boolean add (in long addend1, in long addend2, out long sum)
```

The unnamed return value with type **`boolean`**, would typically be used to indicate some status condition, equivalent to `status` in the earlier notation. In the case of `add()`, for example, the return would be `TRUE` if everything went well, and `FALSE` is something aberrant happened and the results cannot be trusted, like for example there was arithmetic overflow (`addend1` and `addend2` were together too large to be represented by a **`long`**).

Methods are simply listed in the interface specification along with attributes.

Example... A drivers license might be represented by an object with interface `Driver's_license`, which includes methods to check the current validity of the license either today or at some other date:

```
interface Driver's_license {
//Interface to object that maintains pertinent information
//on a driver
     attribute Person license_holder;
     attribute short drivers_license_number;
     attribute Date expiration_date;

     //Method returns TRUE if the license is currently valid
     boolean license_valid_now ();

     //Method returns TRUE if license was valid on given date
     boolean was_license_valid (in Date as_of_date)
     }
```

Note that an attribute can be either data (as in `license_number`) or an object reference (as in `license_holder`). The method `license_valid_now ()` may, for example, compare the current date to the `expiration_date`, and return `TRUE` or `FALSE` depending on the outcome of that comparison. The return value of the method is thus **`boolean`**.

Interface inheritance

IDL can specify interface inheritance as shown in the following example in which Person is extended to include some additional attributes:

```
interface Person_with_age : Person {
     attribute short age;
     Date age_as_of_date (in Date date);
     }
```

Objects with *derived interface* `Person_with_age` have available all the attributes and methods of the *base interface* `Person`, and in addition the attribute `age` and method `age_as_of_date()`. It is also implicitly allowed (but not visible at the interface) for the implementation of objects with the derived interface to redefine any methods of the base interface, so a given method of either `Person` or `Person_with_age` may behave differently.

Polymorphism is allowed in the sense that any method that declares an **`in`** argument of interface `Person` can actually be passed the reference to an object with interface `Person_with_age`. Multiple inheritance—a derived interface inheriting more than one base interface—is also allowed (although not discussed in the book).

Address book example

To illustrate the use of IDL, consider an address book that might be used in an email application. The address book has a list of entries, each entry including the name, nickname, and email address of the person.

Example… Say the nickname of Joe Bartender is "Joe", and his email address is "joeb@sims.berkeley.edu". In the email address book user interface, he would be listed under "Joe", but the user will also want to keep around his full name for reference.

An architecture for this application was discussed in "Address Book Example". That architecture can be described using IDL. Objects with interfaces `Name`, `Address`, and `Phone_number` are created as follows:

```
interface Name {
// Interface of object that stores a person's name
     // We keep parts of the name separate so
     // that we can easily implement
     // operations like sort on last name only
     attribute string first_name;
     attribute string middle_name_or_initial;
     attribute string last name;

// The following illustrates how methods can assist
// even a simple interface like this

     // This method will tell us if this Entry
     // matches a certain lastname, as a convenience for
     // searches
     boolean does_lastname_match? (in string trial_name);

     }

interface Address {
// This illustrates that data structures rather than
// objects can be used

     // By convention, any of these data items that don't
     // apply are zero-length strings
     attribute string street_addr_1;
     attribute string street_addr_2;
     attribute string city;
     attribute string state;
     attribute string state_appreviation;
     attribute string country;
     attribute string mail_code;

     // No methods, although there could be
     }

interface Phone_number {
```

```
          // By convention, any of these data items that don’t
          // apply are “-1” (since “0” may be legitimate)
          attribute short local_number;
          attribute short city_code;
          attribute short country_code;
          attribute short dial_prefix;

          // No methods, although there could be
          }
```

It is also convenient to make Email_addr an object, in part because it can have some useful methods:

```
     interface Email_addr {
          // Convention is that total address is
          // “login_name@host_name”
          attribute string host_name;
          attribute string login_name;

          // Method asks the object to verify that the
          // hostname is valid (however it wants to do that)
          boolean is_hostname_valid ();
          boolean is_login_name_valid ();
          }
```

Now, the Entry interface can be constructed by multiple interface inheritance:

```
     interface Entry : Name, Address, Phone_number {
     // This interface implicitly includes attributes and
     // methods of Name, Address, and Phone_number through
     // multiple interface inheritance

          attribute string nickname;

          // This attribute illustrates an important point:
          // attributes can be references to objects
          attribute Email_addr email_address;

          // No more methods, although there could be
          }
```

Next, a `List` object stores a list of generic `Object`’s:

```
     interface List {
     // This interface stores a list of Object’s, and provides
     // a set of generic useful operations on this list

     // We assume a concept of the “current Object”, which
     // may be any one of the contained Object’s

          // This attribute tells us how many Object’s
          // are on the list
```

```
        // The keyword readonly specifies that the attribute
        // cannot be modified at the interface; changing it
        // is encapsulated in the Object
        readonly attribute short number_of_objects;

        // This method adds an new Object to the end of the list
        // It also illustrates that parameters can be objects
        void add (in Object object_to_add);

        // We need a way to iterate through the Object's in the
        // list, changing the "current" Object; by convention,
        // this method cycles through all Object's as it is called
        // multiple times
        void iterate_current();

    // The following methods operate on the current Object

        // This method does nothing if there are no Object's
        // currently stored on the List
        void delete();

        // This method returns the "current" Object
        Object current();
        }
```

A `List` of **`Object`**'s is useful because IDL also supports polymorphism. Since literally any object interface is implicitly derived from **`Object`**, any interface can be substituted for **`Object`** as it is stored in the `List`. Polymorphism has allowed `List` to maintain a list of literally anything.

Now, it is simple to build an `Address_book` by inheriting the generic `List`:

```
    interface Address_book : List [
    // Note that Address_book now has all the attributes and
    // methods of List. Interface Entry can be readily
    // substituted anywhere for Object

    // We can now add things particular to Address_book
    // not supported by interface List

        // Support queries on nickname or lastname
        // These methods return TRUE if an entry is found,
        //         FALSE if no entry is found
        // After these methods, the "current" Entry
        // is the one found
        // (so, for example, we can conveniently delete it)
        boolean entry_with_lastname(
                    in string lastname,
                    out Entry entry
                    );
        boolean entry_with_nickname(
                    in string nickname,
```

```
                out Entry entry
                );
        }
```