Interface Definition Language

A. David McKinnon

Washington State University

An Interface Definition Language (IDL) is a language that is used to define the interface between a client and server process in a distributed system. Each interface definition language also has a set of associated IDL compilers, one per supported target language. An IDL compiler compiles the interface specifications, listed in an IDL input file, into source code (e.g., C/C++, Java) that implements the low-level communication details required to support the defined interfaces. IDL can also be used to populate an implementation repository, which other programs can use to look up information on an interface at runtime. This is necessary when a program, such as a debugger or interface browser, does not have access to an application's IDL file.

One advantage of an interface definition language is that it does not contain any mechanism for specifying computational details. The stubbed out routines, generated by the IDL compiler, must be filled in with implementation specific details provided by the application developer. Thus, an IDL clearly enforces the separation of a distributed application's interface from its implementation.

Another advantage of an IDL is the productivity enhancement provided by the IDL compiler. Without the IDL compiler, the developer would have to custom craft the network protocol for each distributed application developed, which would be both time consuming and error prone. The IDL compiler frees the developer from these low-level details, thus providing more time for the developer to focus on the application's core functionality.

These IDL benefits, as well as others described below, have enabled many of the successes achieved by middleware (*see* Middleware) and distributed applications (*see* Distributed Applications).

IDL Example

The example is a simple banking application. The bank will allow one to open a checking account, write checks, and make deposits. The example IDL specification is shown in Figure 1. It is written in CORBA IDL [3] (*see* CORBA), a standard that is maintained by the Object Management Group (OMG) (*see* Object Management Group). CORBA IDL is an easy to read and object oriented IDL. Its syntax is similar to C++'s syntax, but a few differences do exist. For example, IDL does not use several C++ keywords (e.g., `public`, `private`), nor does it allow the specification of data members.

The IDL file, shown in Figure 1, has defined all of its definitions within the scope of the `BankExample` module. A `typedef` and `struct` have been used to provide the `MoneyType` and `NameType` abstract data types that will be used in the example. The first interface to be defined is the `BankAccount` interface, which in turn has been defined to include three operations. They are `balance`, which returns the current account balance; `deposit`, which returns an updated account balance after a deposit has been made; and `withdraw`. The second interface, `CheckingAccount`, is derived from the `BankAccount` interface. It extends the `BankAccount` interface definition by defining a new exception, `BadCheck`, and a new operation, `writeCheck` that can raise the `BadCheck` exception. The final interface, `BankManager`, uses the factory pattern to create new `CheckingAccount` objects. When the `openAccount` operation is invoked on a `BankManager` object, a new `CheckingAccount` object is created and its object reference is returned to the client process.

Modules and Interfaces

`module` is a scoping keyword that allows a developer to group related definitions into a common namespace. Each IDL module may consist of one or more of the following elements: data structure definitions, exception definitions and interfaces. Each IDL interface is comprised of one or more operations (e.g., `deposit`, `openAccount`) and interface specific exception definitions (e.g., `BadCheck`), as required. When an operation is invoked on an interface, a message is sent to the object implementing the given interface. Exceptions may be raised when an error occurs (e.g., `BadCheck` is raised when a `writeCheck` operation is invoked on a `CheckingAccount` object that contains insufficient funds).

```
module BankExample {
  typedef float MoneyType;
  struct NameType {
    string first;
    string last;
  };
  interface BankAccount {
    MoneyType balance();
    MoneyType deposit(
      in MoneyType amount);
    MoneyType withdraw(
      in MoneyType amount);
  };
  interface CheckingAccount :
                        BankAccount {
    exception BadCheck {
      MoneyType fee;
    };
    MoneyType writeCheck(
      in MoneyType amount)
      raises (BadCheck);
  };
  interface BankManager {
    CheckingAccount openAccount(
      in NameType name,
      in MoneyType deposit,
      out MoneyType balance);
  };
};
```

Figure 1: Bank IDL Example

Data Structures

IDL provides a basic set of atomic data types (e.g., `long`, `double`, `string`) and a mechanism, `struct`, for combining these atomic types into more complex structures. The `typedef` command can be used to create a new name for a data type.

Parameter Passing

A client and server are typically on different machines, so message passing must be used to pass parameters, including any return value, between them. Thus, the client sends a request message to the server object, which sends a reply message back (if required). However, it is unnecessary for both messages to contain all parameters, so CORBA IDL introduces constructs to deal with this. The `in` keyword denotes the parameters that the client sends to the server object. The `out` keyword denotes parameters which the server will pass back to the client; if a routine includes a return value it is always included in the reply message. The `inout` keyword indicates that the parameter will be passed in both messages. If there are no `out` or `inout` parameters or a return value for a method, then no reply message is sent.

Inheritance

Interfaces can inherit functionality from other interfaces. The example's `CheckingAccount` interface has a `deposit` operation because `CheckingAccount` inherits this operation from the `BankAccount` interface.

IDL Compilation

An IDL compiler takes as input an IDL file, with its associated interface definitions, and produces a set of output files for both the client and server application. The names and number of generated files varies from one development environment to another. The client side code consists of a set of routines that transparently access the server. On the server side, the IDL compiler generates a skeleton framework that must be fleshed out with application specific implementation details.

Heterogeneity

By definition, interface definition languages are both platform and implementation language neutral. The IDL compiler achieves this neutrality. During compilation the compiler generates any platform conversion code that is necessary (e.g., big endian to little endian). Language neutrality is achieved by executing the appropriate compiler (e.g., `idl2ada`, `idl2c`, `idl2java`). Thus, a java client (using `idl2java` client code) on a little endian machine can communicate with a Cobal based server (using idl2cobal server code) on a big endian machine.

A more in-depth introduction to CORBA IDL is given in [2].

Benefits of IDL

IDL offers benefits beyond those described above.

Good Software Engineering Practice

The success of a distributed system may be strongly influenced by the design of its interfaces. The use of an IDL reinforces the idea of good interface design by forcing the developer to consider the interfaces to the system before the implementation details are coded.

Interoperability

IDL enables cross platform and cross language applications development.

Enhanced Productivity

The IDL compiler automates the generation of the low-level communications details (e.g., data marshaling, wire protocols) that must be addressed by a working application.

Multi-language Support

IDL mappings exist for a wide variety of languages (e.g., Ada, C/C++, Cobal, Java, Smalltalk) and a number of different middleware systems (e.g., CORBA, DCE, DCOM).

Object Oriented Design

Focusing on the interfaces to the servers emphasis their object oriented nature within a distributed system.

Interface vs. Implementation

Most IDLs can only specify the interfaces within a system—the implementation details are completely unspecified. In these cases, the server designer is free to deploy any of a number of suitable implementations. For example, an initial bank implementation could give anyone that opens a new account a 5% matching bonus (i.e., when `openAccount` returns `balance = 1.05*deposit`). Later on, a bank manager optimizing revenue at the expense of good will could decide to charge all new accounts a $10 processing fee to open an account (i.e., `balance = deposit-10`). Both of these implementations would be correct and the change from one to another could be made without changing any of the client code.

Scalability/Performance

Since IDL does not constrain the server-side implementation, various techniques could be used to enhance the performance and/or scalability of the server. Possible examples include: multi-threading, persistent state, redundancy, transactional processing, caching and so forth.

History of IDL

Remote Procedure Calls (RPC) were implemented in the early to mid-1980s. Both Sun and Apollo had early RPC implementations. Sun RPC, defined in RFC 1057 [4], is now referred to as Open Network Computing (ONC) RPC. `rpcgen`, an IDL compiler-like tool, was developed to automate the ONC RPC process. The Apollo RPC research lead to OSF's Distributed Computing Environment (DCE) RPC and its associated IDL and IDL compiler. ONC RPC and DCE IDL are both service oriented rather than object oriented.

The concept of an interface definition language was created in the mid-1980s with the development of the Cronus Type Definition Language (*see* Cronus) and the Mach Interface Generator (*see* Mach). The Cronus Type Definition Language is an object oriented language used to specify the data types within a distributed Cronus application. MIG, the Mach Interface Generator, supported a subset of the Mach Matchmaker language and was used to generate the remote procedure call interfaces that are used by Mach interprocess communication.

The 1990s saw the introduction of CORBA IDL, Microsoft's IDL and Java's Remote Method Invocation (RMI). CORBA IDL is an object oriented IDL that supports return values and user exceptions. Microsoft's IDL is often generated by development tools, rather than by hand. Microsoft's IDL can trace some of its roots back to DCE IDL. Java RMI is single-language solution to developing distributed applications. It is object oriented, but it is limited to Java-only development environments.

Future of IDL

The future of interface definition languages is inseparably intertwined with the future of distributed systems. Yesterday's stand-alone applications are becoming increasingly networked. As that happens, designers will be forced to define the interfaces with which these applications will communicate on the network. But, even though they will not disappear, tomorrow's IDL will likely be geared to tomorrow's challenges.

Faster Networks

Traditionally, the network has been the performance bottleneck of a distributed application. As networks become faster, the host-based communications overhead will become an increasingly larger portion of the overall communications bottleneck. To that end, some researchers are adapting standard compiler techniques to the IDL compilers. One such effort, Flick, the Flexible IDL Compiler Kit, optimizes code for marshalling and unmarshalling data. These optimization have resulted in Flick-generated code that performs from 2 to 17 times as fast as code generated by other IDL compilers [1].

Interoperability

IDL based solutions such as CORBA have been quite successful at integrating legacy systems into the overall network based computing environment. Writing a small IDL based wrapper for the legacy application is one way of doing this, as was explained earlier. Another approach that is starting to be practiced reverses the standard IDL to implementation language mapping. For example, rather than starting with an IDL file, these tools will start with an existing piece of legacy code. After analyzing this code, these tools produce a reverse-engineered IDL file that corresponds to the existing implementation. The generated IDL is generally not very readable. However, once a generated IDL definition is available, it can be used with the developer's existing IDL based tools. Thus, these tools enable the quick development of new applications that can interoperate with the original (and unchanged) application. One such current effort is OMG's Java to IDL mapping standard.

Quality of Service

Perhaps the biggest stimuli to IDL development in the future will be quality of service (QoS) (*see* Quality of Service) in all of its various forms. Two schools of thought have emerged. One seeks the inclusion of QoS parameters into the existing IDL specification languages. They argue that this is a natural progression that will enable the development of future QoS aware applications. The other school of thought argues that including QoS information within an IDL specification will break the separation of interface vs. implementation, which is so fundamental to IDL's success, or it causes combinatorial problems in the kinds of QoS properties supported in an interface for a service. They argue that a clean IDL definition will allow back-end tools to provide the QoS requirements and modules needed by the application. A current example of this is the Quality Object (QuO) architecture [5].

Object-by-Value

IDLs have largely been limited to the representation of static data structures. The object-by-value standard for CORBA augments IDL so that both the data within an object as well as its state may be transmitted between processes. This is similar to Java's object serialization, but still usable in other implementation languages. Object-by-value can allow a copy of an object to be passed instead of just its reference, which can in some cases greatly speed up execution. It can in some cases also be used to provide caching and replication support.

References

[1] E. Eide, J. Lepreau and J. Simister, *Flexible and optimized IDL compilation for distributed applications*. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, D. O'Hallaron (ed.), *Lecture Notes in Computer Science 1511*, Springer-Verlag, 1998.

[2] M. Hennig and S. Vinnoski, *Advanced CORBA Programming in C++*. Addison Wesley Longman, Inc., 1999.

[3] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 2.3*. Object Management Group, Framingham, MA, 1998.

[4] Sun Microsystems, Inc., *RPC: Remote Procedure Call Protocol Specification, Version 2,* RFC 1057, June 1988.

[5] J. Zinky, D. Bakken, and R. Schantz, *Architectural Support for Quality of Service for CORBA Objects*, Theory and Practice of Object Systems, 3:1, April 1997.

Cross Reference:

IDL *see* Interface Definition Language.

CORBA *see* Interface Definition Language.

CORBA IDL *see* Interface Definition Language.

COM IDL *see* Interface Definition Language.

IDL Compiler *see* Interface Definition Language.

Language Mapping *see* Interface Definition Language. {???}

Middleware *see* Interface Definition Language.

Dictionary Terms:

Object Management Group (OMG)

The OMG is a world-wide, open membership consortium that produces, among other things, the CORBA specification (see CORBA).

Marshal (Unmarshal)

The act of preparing data for transmission (reception). It includes gathering the data and packaging it in a manner suitable for transport.

Wire Protocol

A protocol that describes how data is transmitted over a network (i.e., wire).