HEWLETT PACKARD

Distributed Object Computing With CORBA

Steve Vinoski

Hewlett-Packard Company
Distributed Computing Program
300 Apollo Drive
Chelmsford, MA 01824

vinoski@ch.hp.com

This paper was originally published in the July/August 1993 issue of C++ Report magazine.

1.0 Introduction

The Object Management Group (OMG) was formed in 1989 with the purpose of creating standards allowing for the interoperability and portability of distributed object-oriented (OO) applications. Unlike the Open Software Foundation (OSF), the OMG does not actually produce software, only specifications. These specifications are created using ideas and technology from OMG members who respond to Requests For Information (RFI) and Requests For Proposals (RFP) issued by the OMG. A strength of this approach is that most of the major players in the commercial distributed OO computing arena are among the several hundred companies that belong to the OMG.

The OMG *Object Management Architecture* (OMA) attempts to define, at a high level of abstraction, the various facilities necessary for distributed OO computing. The core of the OMA is the *Object Request Broker* (ORB), a mechanism that provides transparency of object location, activation, and communication. In 1991 the OMG published revision 1.1 of the *Common Object Request Broker Architecture* (CORBA) specification, a concrete description of the interfaces and services that must be provided by compliant ORBs [1]. Since then, many OMG member companies have either started shipping or have announced plans to ship products based on the CORBA specification.

The CORBA is composed of five major components:

- ORB Core
- Interface Definition Language
- Dynamic Invocation Interface
- Interface Repository
- Object Adapters

The next few sections describe each of these components and show how they provide the flexibility required to support a variety of distributed OO systems, emphasizing those developed in C++.

2.0 C++ and Distributed OO Computing

Almost every C++ programmer who has written a distributed application has run into the same problem: How does one ship a C++ object across a network from one process to another? Systems like the Distributed Computing Environment (DCE) from the OSF provide remote procedure call (RPC) mechanisms that can ship complicated data structures across a network, but they do not provide explicit support for C++ objects. The fact that C++ objects can contain hidden pointers to virtual function tables eventually becomes a stumbling block for even the most determined programmer, especially when networks of heterogeneous computers are involved. What if the receiving process has never heard of that particular C++ class? Can the necessary member functions be dynamically loaded? How do the virtual tables get set up properly when an object is received from a remote process? Solving these kinds of problems in a portable fashion is difficult.

Unfortunately, shipping C++ objects across a network typically requires a violation of object encapsulation. Private data used for object implementation must somehow be accessed so that the object can be taken apart, transmitted, and then reconstructed on the receiving side. Knowing how objects are laid out in memory can make such a task possible, but such knowledge is compiler-specific and is therefore not portable. A cleaner approach is to make every class support member functions for marshaling and unmarshaling its own data, but such programming can be tedious and error-prone.

Neither of the solutions to the encapsulation problem mentioned here solves the biggest problem with C++ object migration, which is keeping each separate executable component of a distributed system synchronized with respect to different versions of the C++ class types involved. Making sure that each component has been compiled with the same declaration for each class type is difficult, even with access to the source code. The bugs that result from transmitting an object to a remote process that has different knowledge of the object's type are often difficult to isolate and repair. Third-party applications and applications written in other languages further compound the problems.

The fact that C++ object migration is such a tough problem leads to an examination of why an application would want to do it in the first place. It is often the case that existing client-server applications, initially written in C, are over time upgraded to use C++ and OO programming techniques. As part of the conversion, C structs are often changed into C++ classes. If these structs were transmitted over the wire by the original application, it becomes necessary to be able to send them as full-fledged objects once the application is converted to C++.

Applications that evolve in this manner fail to distinguish between *object state* and *object behavior*. They equate what an object is made of with what the object does. They are tuned to know that some objects work by being transmitted across a network while other objects exist as servers and work by receiving RPCs. Applications that know this much about their objects already know too many implementation details. In simple terms, they do not abide by the principles of information hiding and encapsulation.

To avoid breaking encapsulation, distributed OO applications must deal only with object interfaces and should not care whether the object implementations are in the same process or on another machine halfway around the world. This ideal requires an object model that allows applications to transparently utilize both local and remote objects without sacrificing efficiency. Such an object model must address issues faced by all developers of distributed applications, providing a standard object programming interface that is not only system-independent but is also language-independent. In today's market where customers demand open systems and standard interfaces that let them continue to utilize their existing software bases, distributed systems that require the use of one particular programming language are unlikely to be commercially successful.

3.0 ORB Core

In the OMA object model, objects provide services, and clients issue requests for those services to be performed on their behalf. The purpose of the ORB is to deliver requests to objects and return any output values back to clients. The ORB services necessary to accomplish this are completely transparent to the client. Clients do not need to know where on the network objects reside, how they communicate, how they are implemented, how they are stored, nor how they execute.

Before a client can issue a request to an object, it must hold an *object reference* for that object. An ORB uses object references to identify and locate objects so that it can direct requests to them. As long as the referenced object exists, the ORB allows the holder of an object reference to request services from it. Object references can be made persistent by first asking the ORB to convert them to strings. Clients can store these string object references in their own private data files and later retrieve them, ask the ORB to change them back into object references, and use them to make requests. This capability can be used to maintain persistent links between objects and the applications that use them, such as hypertext links in compound documents.

The CORBA specifies two different ways in which clients can issue requests to objects:

- static invocations via interface-specific stubs
- dynamic invocations via the ORB Dynamic Invocation Interface (DII)

Regardless of which of these methods the client uses to makes a request, the ORB locates the desired object, activates it if it is not already executing, and delivers the request to it. The object has no knowledge of whether the request was made through a static stub or through the DII, nor does it know where it came from (all security implications aside). It performs the requested service and returns any output data back to the ORB which then returns it to the client.

4.0 Interface Definition Language (IDL)

Even though an object reference identifies a particular object, it does not necessarily describe anything about the object's interface. Before an application can make use of an object, it must know what services the object provides. In CORBA, object interfaces are described in IDL, a declarative language with a syntax resembling that of C++. IDL provides basic data types (such as short, long, float, double, and boolean), constructed types (such as struct and discriminated union), and template types (such as sequence and string). These are used in operation declarations to define arguments types and return types. In turn, operations are used in interface declarations to define the services provided by objects. IDL also provides a module construct that can hold interfaces, type definitions, and other modules for name scoping purposes.

Of all the types provided in IDL, interfaces are the most important. In addition to describing CORBA objects, they are also used as object reference types. Operations can be declared to return object references and to take object references as arguments simply by using interface names as follows:

```
interface MailMsg;
interface Mailbox
{
   MailMsg next_msg();
};
```

In this example, a client of a Mailbox object can use the return value of the next_msg() operation to invoke operations on a MailMsg object, since the return value is an object reference.

IDL provides interface inheritance in which derived interfaces inherit the operations and types defined in their base interfaces. In C++ terms, IDL interface inheritance exhibits the following characteristics:

- all base interfaces are effectively public virtual
- all operations are effectively virtual
- operations cannot be redeclared in derived interfaces
- there is no notion of implementation inheritance

The fact that CORBA IDL is a declarative language heightens the separation of interface and implementation that is emphasized in OO systems development. For example, in C++ the concepts of interface inheritance and implementation inheritance are mixed together. An object of a derived C++ class always contains all the data members of its bases classes, and for the purposes of polymorphism, a derived class can only redefine those base class member functions explicitly declared virtual. Since IDL is not an implementation language, it does not confuse these two types of inheritance.

Base interfaces are only inherited once because interfaces represent object behavior, not object state. In C++, unless virtual inheritance is used, an object of a derived class is composed of multiple objects of a multiply-inherited base class type. This is because a C++ class represents both what an object is made of and what an object can do. There is no concept of state in IDL, only behavior. By inheriting a base interface, an object of a derived interface promises to support that interface, but it makes no promise about how its implementation will do so.

C++ programmers may initially be bothered by the inability to redeclare operations inherited from base interfaces in derived interfaces. It might seem that this restriction prevents CORBA objects from being used polymorphically. Again, due to the split between interface and implementation, this is not the case. Object implementations are free to utilize any inheritance features of their implementation languages, independent of IDL inheritance. The degree of polymorphism seen by client applications depends only on how object references are mapped to the client implementation language.

In IDL, all interfaces implicitly derive from a root interface called Object. (This is an unfortunate choice of names since many existing software libraries also use the name ''Object'' for other purposes.) The Object interface provides services common to all ORB objects, such as duplicate and release for object references, and is_nil for checking the validity of object references. Most C++ programmers frown on single-rooted inheritance hierarchies because they result in C++ classes with bloated or "fat" interfaces. However, with IDL interface inheritance, the use of a common base interface makes sense since all objects in the system are by definition CORBA objects and so must provide these basic services.

IDL compilers translate IDL language constructs into specific programming language modules according to CORBA language bindings. For example, CORBA revision 1.1 specifies a language binding for C in which object references are mapped to void* data pointers. Basic data types, such as short, long, string, struct, and array, are mapped to C in the obvious manner. Operations are mapped to C functions that take an object reference as a distinguished first parameter.

The next IDL language binding to be specified by CORBA will be C++. In December 1992 the OMG issued a RFP for C++ language bindings for IDL, and they expect to issue a final decision on such a binding in December 1993 [2]. The RFP states that responses are free to use any C++ features as defined in Stroustrup's The C++ Programming Language, Second Edition (presumably referring to the reference manual portion of the book), including templates and exceptions [3].

What might a C++ language mapping for CORBA IDL look like? Hewlett-Packard Company, IONA Technologies Ltd., and SunSoft, Inc. (a division of Sun Microsystems, Inc.) have jointly developed a CORBA IDL C++ mapping that attempts to make CORBA objects look as much like normal C++ objects as possible. To achieve that goal, IDL interfaces are mapped to C++ classes called *surrogates*, and IDL operations are mapped to member functions of those surrogates. Public virtual derivation for surrogate base classes is necessary to mimic IDL interface inheritance semantics. Object references are implemented as pointers to surrogate classes, allowing C++ to implicitly convert pointers to derived interfaces to pointers to base interfaces.

With this C++ language mapping, a client can invoke an operation on an object reference for the Mailbox interface shown earlier in the following manner:

```
// assuming "mbox_objref" is a string
// representing an object reference
Mailbox *mbox = ORB::string_to_objref(mbox_objref);
if (!Mailbox::is_nil(mbox)) {
   MailMsg *msg = mbox->next_msg();
   if (!MailMsg::is_nil(msg)) {
      // call operation on msg
   }
}
```

Not only do object surrogate classes provide a natural mapping for IDL to C++, they also maintain the compile-time type checking that C++ users rely on. In the example above, the C++ compiler will ensure that only Mailbox operations are invoked on the Mailbox object reference and not on the MailMsg object reference, and it will check that each member function call has the correct number and types of arguments.

Our experiences with such an IDL-to-C++ mapping have shown the need to be able to typecast object references for base interfaces to those for derived interfaces (others have described similar findings [4]). This is ordinarily frowned upon for typical C++ programming, but for distributed programming it is sometimes a necessity because the real type of the object may be unknown in some parts of the system.

Downcasting or ''narrowing'' object references can be done in several ways. An approach familiar to C++ users is to provide typesafe casting by augmenting objects with both a type identification field and member functions that use it, as is done in Keith Gorlen's NIH Class Library [5]. An alternative to providing a downcasting mechanism for surrogates might appear to be the run-time type identification (RTTI) mechanism recently voted into the C++ language by the ANSI X3J16 and ISO WG21 standardization committees [6]. The RTTI mechanism allows downcasting to be performed via the dynamic_cast<T*> operator. For example, the ORB::string_to_objref function used in the Mailbox example actually returns a pointer to a root IDL Object which must be cast down to the desired type:

```
CORBA::Object *obj = ORB::string_to_objref(mbox_objref);
Mailbox *mbox = dynamic_cast<Mailbox*>(obj);
```

Unfortunately, this downcasting method does not work as expected. The CORBA::Object pointer obtained by converting the string to an object reference does not really point to a Mailbox object surrogate. It instead points to a CORBA::Object surrogate, and so dynamic_cast will fail and return 0. The dynamic_cast operator only knows about the C++ type of the object surrogate, not the IDL type supported by the remote object implementation.

Accurate run time type checking of an object reference could be performed by an ORB via an Interface Repository (explained below), but such checking may require multiple remote procedure calls to figure out if the object really is of the desired type. It is almost always more efficient for the ORB to just assume that the object is of the more derived type and provide a narrowed surrogate for it. If an operation invoked on the narrowed surrogate is not supported by the object implementation it refers to, a BAD_OPERATION exception is returned to the caller.

C++ programmers may question why CORBA specifies an entirely new language for describing object interfaces rather than using a declarative subset of C++. There are several good reasons for this choice:

- the CORBA specification is intended to be language-independent; using a subset of C++ for IDL lessens the chances of CORBA being widely accepted in the distributed OO programming community
- C++ is known to be difficult to parse [7]; using a subset of C++ might ultimately limit the number of IDL compiler implementations available
- some features of C++, notably pointers, make marshaling and unmarshaling of data difficult
- any declarative subset of C++ chosen would most likely be different enough from normal C++ so as to be confusing to experienced C++ users

As it is, CORBA IDL strongly resembles C++, and in our experience most C++ programmers find it easy to understand and use effectively.

5.0 Interface Repository (IR)

Another service supported by the Object interface and hence all object references is the get_interface() operation. This operation returns an object reference to an InterfaceDef that describes the object's interface. The InterfaceDef is stored in an Interface Repository (IR) which provides persistent storage for IDL interface declarations. The services offered by an IR allow navigation of an object's inheritance hierarchy and provide descriptions of all operations that an object supports. Some of these services return references to other IR objects, such as OperationDef objects that describe operations and ExceptionDef objects that describe user-defined exception types.

Interface Repositories can be used for several purposes. Interface browsers can traverse IR information to help application developers locate potentially reusable software components. ORBs could use them to check operation parameter types at run time (but such overhead can normally be avoided by using IDL-generated stubs mapped to a statically typed language like C++). The primary function of the IR, however, is to provide the type information necessary to issue requests using the Dynamic Invocation Interface.

6.0 Dynamic Invocation Interface (DII)

The compilation of IDL declarations into C++ or C stubs allows clients to invoke operations on known objects, but some applications must be able to make calls on objects without having compile-time knowledge of their interfaces. For example, a graphical user interface (GUI) builder might, given a list of object references for drawable components, allow users to browse Interface Repositories, learn about the operations supported by each object, and then invoke operations on them to see how they present themselves on a display. Such a GUI builder would only have to know how to traverse IR information and prompt the user for any data necessary to fulfill operation parameter requirements. It could then invoke any operation the user desires via the Dynamic Invocation Interface (DII).

In essence, the DII is a generic client-side stub capable of forwarding any request to any object. It does this by run time interpretation of request parameters and operation identifiers. Clients can create requests via the Object::create_request operation. This operation returns an object reference to a Request object which is actually implemented as part of the ORB. Request objects support services such as add_arg for filling in request parameters and invoke for calling the operation represented by the Request. They also allow deferred synchronous requests via their send and get_response operations.

The flexibility provided by the DII can be costly, however. A remote request made through a compiler-generated stub/skeleton pair can be achieved in a single RPC, but the same call made through the DII requires calls to:

- Object::get_interface to obtain an InterfaceDef object
- InterfaceDef::describe_interface to obtain information about the operations supported by the object
- Object::create_request to create a Request object for the desired operation
- Request::add_arg for each request argument
- Request::invoke to actually make the request

For an operation with no arguments and a void return type, the DII requires a minimum of two function calls, at least one of which may result in a RPC. There is also the overhead of the DII having to interpret the request, not to mention the bulky application code required to implement this series of steps. For most applications, especially those written in a compiled language like C++, it is far more efficient to make requests through static IDL stubs than through the DII.

7.0 Object Adapters (OA)

CORBA allows object implementations to vary widely. In some cases multiple IDL interfaces may be implemented by a single server program, while in other cases an IDL interface may be implemented by a series of shell scripts, one for each operation. Some may be ‘‘legacy applications’’ developed well before CORBA came about, while others may be OO systems developed specifically to work with an ORB. The ORB provides this flexibility to permit the straightforward integration of legacy applications without locking new objects into a limited set of implementation criteria.

An *object adapter* (OA) provides the means by which various types of object implementations utilize ORB services, such as:

- object reference generation
- object method invocation
- security
- activation and deactivation of objects and implementations

Depending on the underlying ORB, an OA may choose to provide these services by delegating to the ORB or by performing the work itself. In either case, object implementations are not aware of the difference because they interface only to the OA.

Each ORB is expected to provide a general OA, called the *Basic Object Adapter* (BOA), which is intended to support objects implemented as separate programs. It is expected that most object implementations can work with the BOA because its services are flexible enough to accomodate several different kinds of object implementations:

- *persistent* implementations which are activated by something other than the BOA
- *shared* implementations in which multiple objects coexist in the same program
- *unshared* implementations in which only one object implemention exists per program
- *server-per-method* implementations in which each operation supported by an object is a separate program

Except for *persistent* implementations, the BOA will automatically activate an object implementation if a request comes in for it and it is not yet executing.

Object Adapters other than the BOA may exist. The CORBA specification mentions a *Library Object Adapter* (LOA) for use with lightweight object implementations that are co-resident with client applications, and an *object-oriented database adapter* that provides an interface from object-oriented databases to the ORB. All in all, it is expected that only a few different kinds of OAs will ever be needed to satisfy nearly every kind of object implementation.

8.0 Fitting It All Together

Figure 1 shows how the various CORBA components work together to facilitate distributed OO computing. Assuming the client has a valid object reference to the object implementation, it can invoke an operation supported by that object. The request passes into the IDL surrogate which directs the request to the ORB. Alternatively, the client uses the Interface Repository to dynamically create a request that is dispatched through the DII. In either case, the request passes into the IDL stub which directs the request to the ORB. The ORB uses the object reference to locate the object implementation and then delivers the request into the object adapter managing that object. The OA feeds the request into the IDL-generated skeleton where it is then passed to the object implementation. Any return values are passed back through the skeleton and OA to the ORB Core. Then, depending on the origin of the call, the ORB Core returns the values either through the IDL surrogate or the DII to the client application.

9.0 Future of CORBA

Some member companies working with the CORBA 1.1 specification have realized that some parts of it are incomplete. Subsequently, the OMG issued a RFI for ORB 2.0 extensions [8]. In particular, the OMG is looking for information on how to fill in portions of CORBA 1.1 that were intentionally left incomplete, such as parts of the specification of the Interface Repository. They are also asking for ideas for new IDL language bindings, new object adapters, and extensions to ORB functionality such as support for transactions. It is hoped that the additions and extensions will enhance both the portability of CORBA-compliant software and the level of interoperability between independent ORBs.

10.0 Conclusion

With the weight of over three hundred OMG member companies behind it, the CORBA Specification is well on its way to becoming the standard base for distributed OO applications. Together, its flexible components allow legacy systems to work seamlessly with new applications, independent of the systems they run on and the languages used to implement them. C++ users can rely on CORBA-compliant ORBs to help them develop portable distributed OO applications using C++ in a natural fashion.

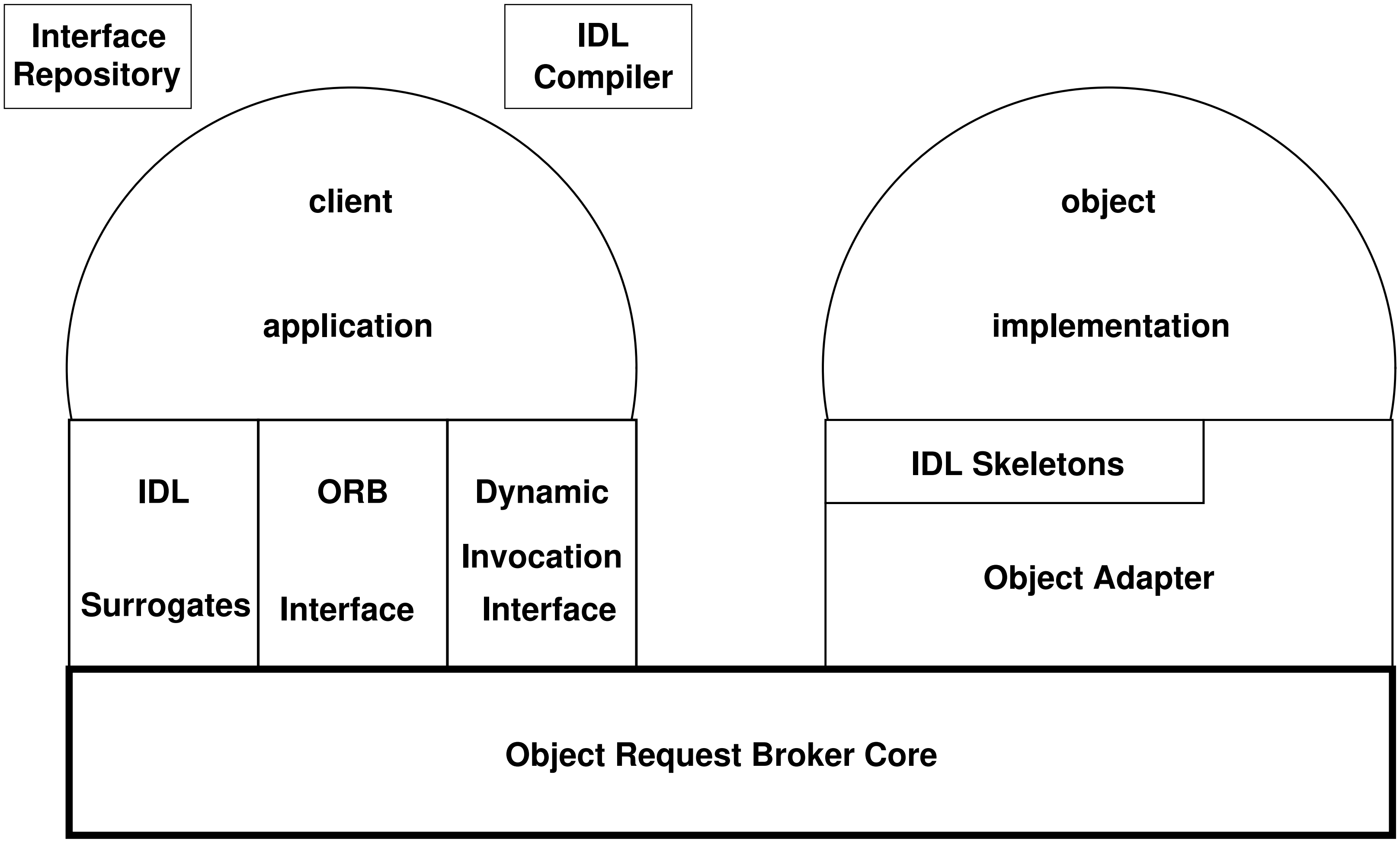

Figure 1: Complete Distributed Application Using the ORB

Acknowledgments

I would like to thank Tom Lyons, Mark Morwood, Joe Sventek, and Teri Witham for their helpful reviews of early drafts of this article.

References

[1] The Common Object Request Broker: Architecture and Specification, OMG TC Document Number 91.12.1, Revision 1.1, December 6, 1991.

[2] C++ Language Mapping Request For Proposals, OMG TC Document Number 92.12.11, 1992.

[3] Stroustrup, Bjarne. The C++ Programming Language, Second Edition, Addison-Wesley, Reading, MA, 1991.

[4] Don Vines and Zen Kishimoto, "Smalltalk's Runtime Type Support For C++", C++ Report 5(1):44-52, January 1993.

[5] Gorlen, K., S. Orlow, and P. Plexico. Data Abstraction and Object-Oriented Programming in C++, John Wiley and Sons, New York, 1991.

[6] Bjarne Stroustrup and Dmitry Lenkov, "Run-Time Type Identification for C++ (Revised yet again)", document X3J16/92-0121, American National Standards Institute Accredited Standards Committee X3, Washington, D.C., 1992.

[7] Ball, Michael, "Inside Templates: Implementing C++ Strategies", C++ Report 4(7):36-40, September 1992.

[8] Object Request Broker 2.0 Extensions Request For Information, OMG TC Document Number 92.12.10, 1992.