Certain Enhancements on CORBA Standard
CHAPTER 4
CERTAIN ENHANCEMENTS ON CORBA STANDARD

4.1 Introduction

In CORBA, objects are created as an outcome of issuing requests from clients. The list of active objects is maintained in Active Object Map (AOM). The outcome of object creation is revealed to the client in the form of an object reference that denotes the new object as shown in figure 4.1. The object is destroyed when the server is shut down.

![Figure 4.1: CORBA Object Life Cycle](image)

CORBA supports multiple interface inheritance, where references to base interface must be unambiguous. Ambiguities can be resolved by qualifying a name with its interface name for constant, type, or exception. In CORBA, it is illegal to inherit from two interfaces with the same operation or attribute name, or to redefine an operation or attribute name in the derived interface. Hence selective interface inheritance is not supported in CORBA.

This chapter proposes the following changes to the CORBA standard:

1) A stub-based method for creation and destruction of CORBA objects.
2) A method to implement selective interface inheritance in CORBA statically.
4.2 Proposed Stub-based framework for Creation and Destruction of CORBA objects

Garbage collection is crucial in a distributed environment, to avoid invalid object reference and to minimize wastage of space in the implementation repository (IMR). CORBA uses IMR to bind client requests to objects which are uniquely identified by Interoperable Object Reference (IOR). Garbage collection and referential integrity are areas where CORBA offers only partial solutions. Garbage collection issues apply to both CORBA objects and implementation repositories. To create a new CORBA object, a client invokes an operation on the factory object. To delete an object, the client invokes an operation on the object, instructing it to destroy. But if a client crashes, the server has no means of detecting it and the object simply hangs forever [146]. Garbage collection also addresses the issue of referential integrity. It can be used to prevent deletion of objects of interest and to guarantee deletion of objects that are no longer wanted. The object life time manager pattern [147] can be used to govern the entire life time of object from creation to destruction of objects. It deals with dynamic object preallocation and deallocation that occurs automatically during application initiation and destruction. It destroys the objects on program termination. In DCOM [148] garbage collection is in built. The DCOM wire protocol uses a pinging mechanism to garbage collect remote server object references. But CORBA specification does not give a scheme for general purpose distributed garbage collection.

Creation of the CORBA objects is done using object factories [149]. Deactivation of the CORBA object is left to the ORB, which deactivates the object instances only when the server terminates. Thus an ORB may choose to never deactivate object instances until the server exits. This is not practical for long-lived servers, as eventually system resources are consumed leaving the server unable to continue executing. This leads to under-utilization of resources. This section proposes a framework for creation, destruction and garbage collection of CORBA objects by defining an interface. This interface serves as the base for the user defined interfaces and contains functions to
manage the object life cycle and to deactivate the CORBA objects when not used by the client. Since garbage collection is currently a platform specific issue in CORBA, an interface called as IRelease has been developed, which serves as the base for the user defined interfaces. It has operations for reference counting, instantiation and destruction of CORBA objects. The following issues have to be considered in the design of the framework for creation and deletion of the CORBA objects:

a) Object Registration: When a new class defining the operations in an interface is implemented in the CORBA server, it is registered in the IMR using the RegisterObject method.

b) Implementation Repository (IMR): An implementation repository is a repository, which maintains the mapping between the abstract object name and its physical address. The implementation repository has the following fields:
- IP address and adapter name
- Names of registered classes from the CORBA server
- Object path
- Reference Count

The Object Reference Table or AOM maintains a mapping of the object names to the object reference (OR). It has object references associated with the currently active objects. The object reference is a generic pointer, which is allocated from a pool of object references. It has the following fields:
- Object Name
- Object Reference

CORBA Object Model uses IOR to uniquely identify an object. When the clients invoke an operation on an object, the IOR binds the request to the servant. The servant represents a programming language instance of a CORBA object and the process of associating a request with its servant is called as binding. Object references are opaque to the clients. The ORB hides the details of dispatching the client's request to the servant.
This opacity of the object reference provides programming language, transport and protocol transparencies. It thus brings about interoperability in a heterogeneous environment.

An IOR has the following information:

- **Repository Id**: It serves as an index into the interface repository (IFR) for runtime type checking.

- **Protocol and Address Information**: CORBA uses Internet Inter Operable Protocol (IIOP) as the medium of transmission. It is an implementation of General Inter Operable Protocol (GIOP) over TCP/IP. The address information has the host name and TCP port number.

- **Object Key**: It identifies the object pointed to by the IOR. An object key comprises of the object adapter name and the object name within the adapter.

CORBA provides two types of IORs. They are the transient IOR and the persistent IOR. A transient IOR continues to work as long as the server process remains available. A persistent IOR denotes the same CORBA object even when the server is shutdown and restarted. It makes use of implementation repository. An implementation repository maintains a registry of servers, host addresses and port numbers. It can also activate the servers on demand. The standard CORBA specifications do not give the design of the implementation repository because it is related to the underlying platform and is highly implementation dependent.

c) **IRelease Interface**: The IRelease Interface has three methods. They are AddRef, DeleteRef and DestroyObject. If the client happens to be the first to request a particular object then the AddRef method creates a new object and the count field in the IMR is set to one. If the object is already created, then the reference count is incremented by one. When a client leaves, the DeleteRef method is called to decrement the count parameter in the implementation repository. When the count becomes zero, the reference in the table is invalidated and the DestroyObject method is
called to delete the servant object. This method minimizes the wastage of memory resources, by deactivating the CORBA objects when not used by the clients.

The steps involved in creation and deletion of CORBA objects using a IRelease Interface as shown in figure 4.2 are listed below:
1) The CORBA server registers its classes in the implementation registry. This is done by the RegisterObject method.
2) The user-defined interface file is given as input to the IDL compiler.
3) The compiler maps the interface specifications to a programming language, namely C++ and generates stubs and skeletons.

---

**Figure 4.2: Illustration of Proposed method of CORBA object creation and destruction by Reference Counting**
4) When the client specifies the object name, class name and function name, it is checked for validity using the C++ header file generated by the IDL compiler. If valid these details are passed to the name server, which manages the implementation repository. 4a) The name server uses the IRelease methods to create the objects, destroy the objects and to change the reference count.

5) From the Object name, the object reference of the CORBA server object is returned to the client.

6) The client reissues the operation request to the new location of the CORBA server, using OR.

7) The CORBA server returns the results of the operation to the client.

The framework has been implemented on Windows 2000 using VC++ and has been tested.

4.3 Proposed Method of Implementing Selective Inheritance in CORBA

Selective inheritance based on implementation inheritance can be used in specifying rules describing semantic information of the subclasses that can be inferred from their superclasses for subtype, parameterized and selective inheritance [150]. A complex object model [76] based on specialization constraint (SC) that allows a subclass to disinherit some properties from a superclass. This is selective implementation inheritance. Conditional Interfaces [151] can be used to implement selective inheritance. Omnibuilder [76] is a tool that manages the full lifecycle of an application. It generates all application components like the server objects, client modules and middleware protocols. Here object inheritance is used to simplify the business model. Subclasses can have selective inheritance from the parent and can have additional properties and behaviors beyond those of the superclass based on implementation inheritance. COM and CORBA specifications support interface inheritance but not implementation inheritance. Component Object Model (COM) allows only single interface inheritance. Instead of supporting multiple inheritance, COM uses multiple interfaces to an object.
There is no selective inheritance in COM [152]. CORBA specifications do not support selective inheritance as function overloading is not allowed in the derived interface. This section proposes a method to implement selective multiple inheritance in CORBA using conditional interfaces.

In interface inheritance, a subclass can see only the parent's interface. Interfaces inherit from other interfaces and it deals with programming to an interface and not implementation. The implementation of the classes that provide the functionality is hidden from the user. The implementation can be changed without affecting the external application. It thus provides an easy approach to encapsulation and abstraction in object-oriented systems. But it is costly in terms of code size and execution speed. The abstract interface is not instantiated. It is inherited from and an implementation is provided for this subtype (derived interface) as shown in figure 4.3. The application thus defines subclasses of the abstract interface.

![Interface Inheritance Diagram](image)

Figure 4.3: Interface Inheritance.

Interface inheritance provides a composition mechanism for promoting an object to support multiple interfaces. The principal interface is the most specific interface that the object supports and consists of all operations in the transitive closure of the interface inheritance graph. It is also called as subtyping.

The following are the characteristics of interface inheritance:

1) Any method that a subclass defines will override methods in the super class.
2) Super class method is left unchanged for objects from the subclass.
3) Subclass inherits the object interface but changes some of the implementation it gets from the super class.
4) It is useful in cases when data types need to be changed but functionality 
of the code need to be preserved.
5) It is an easy approach to encapsulation and abstraction.

4.3.1 Interface Inheritance in CORBA

OMG IDL defines the types of objects by specifying their interfaces. It is a language used to describe the interfaces that client objects call and the object implementations provide. An interface consists of a set of named operations and the parameters of those operations. It provides a conceptual framework for describing the objects. IDL is a means by which a particular object implementation tells its clients what operations are available and how they should be invoked. Since IDL is purely a declarative language, a mapping is provided from it to a particular programming language like C++ or Java.

Consider a CORBA IDL file as shown in figure 4.4a, the base interface serves as a base to the derived interface, which inherits all its functions. The operations map to pure virtual functions in C++, which is implemented by the server program. The base class implementation is shown in figure 4.4b.

```plaintext
module example
 {
   interface base
   {
     void op1();
   };
   interface derived: base
   {
     void op2();
   };
 };

Figure 4.4a: An Example CORBA Interface

class baselmpl : virtual public base_skel
{
 public:
   baselmpl();
   void op1()
   {......};
};

Figure 4.4b: Base Class Implementation
There are two ways in which derived implementation inherits the features of the base implementation. They are listed below:

1) When the derived implementation inherits from the base implementation, the operations in the base implementation need not be implemented in the derived implementation. This is the type of derived implementation used in the proposed model of our system. It is shown in figure 4.5.

```cpp
class dervlmpl : virtual public base_impl, derv_skel
{
    public:
    dervlmpl();
    void op2()
    {......};
};
```

**Figure 4.5: Derived Class Implementation - 1**

2) When the derived implementation inherits from the base skeleton, the operations in the base implementation have to be reimplemented in the derived implementation as shown in figure 4.6.

```cpp
class dervlmpl : virtual public base_skel, derv_skel
{
    public:
    dervlmpl();
    void op2()
    {......};
    void op1()
    {......};
};
```

**Figure 4.6: Derived Class Implementation - 2**

Figure 4.7 explains the inheritance of derived implementation. In this figure, Base is the base interface and Derived is the derived interface. The stub and skeleton generated by the base interface are Base_stub and Base_skel. The derived stub is generated by the derived interface and inherits from the Base_stub. The Derived_skel is generated by the Derived interface.

The Base_impl is an implementation of the base interface. Based on the two methods in which the derived interface is implemented, there are two
types of derived implementations. They are Derived_impl1 and Derived_impl2. As Derived_impl1 inherits from the Base_impl, the operations in the Base_impl need not be redefined in the Derived_impl1. Derived_impl2 inherits from the Base_skel and all the methods in the Base_impl have to be redefined in the Derived_impl2.

![Diagram of inheritance in derived implementations of CORBA]

Figure 4.7: Inheritance in derived implementations of CORBA

Figure 4.8 explains multiple interface inheritance in CORBA. The derived interface inherits from Base1 and Base2 interfaces. The derived implementations can be of the following types:

1. In the first method, the Derived_impl1 inherits from the Base_impl1 and Base_impl2.
2. In the second method, the Derived_impl2 inherits from Base_skel1 and Base_skel2. Hence the methods in the Base_impl1 and Base_impl2 have to be rewritten.
4.3.2 Implementing Selective Interface Inheritance in CORBA

The following conditions have to be followed, to support multiple interface inheritance in CORBA:
1) An interface can inherit from two interfaces that include a const, type or exception definition of the same name.
2) An interface cannot inherit from two interfaces which include operations or attributes that have the same name. So in CORBA overloading a function in a derived interface is not allowed.

Due to these conditions, CORBA does not address the issue of selective inheritance. But in general, interface inheritance allows any method that a subclass defines to override methods in the super class, if present.

It is useful to change the interface of the container based on the type of the object contained in it. So conditional interfaces [151] are important. An example CORBA IDL file is shown in figure 4.9 It supports an interface named add_mult which has two operations, namely add() and mult() to perform the addition and multiplication operations.
module example
{
    interface add_mult
    {
        short add(in short x, in short y);
        short mult(in short x, in short y);
    }
};

Figure 4.9: An Example CORBA IDL

If the example::add() method is not used most of the time, then the following methods can be used:

1) An interface simple without any methods can be used as a base interface from which simple_add and simple_mult interfaces inherit. These interfaces declare add() and mult() methods respectively. Then an interface add_mult can be derived from simple_add and simple_mult interfaces. It inherits the operations of simple_add and simple_mult interfaces. Additionally, it declares its own operations as shown in figure 4.10. But the number of interfaces grows exponentially with the number of switchable features [151].

module example
{
    interface simple {}
    interface simple_add: simple
    {
        short add(in short x, in short y);
    }
    interface simple_mult: simple
    {
        short mult(in short x, in short y);
    }
    interface add_mult: simple_add, simple_mult
    {.............};
};

Figure 4.10: An interface is designed for each possible combination

2) Due to the above disadvantage, conditional interfaces are used. We define a container class which derives from the base interface based on the required conditions. The representation of selective inheritance using conditional interfaces for an example CORBA IDL is shown in figure 4.11. This translates to conditional if-then-else statements in the presentation (interface) file generated in C++. 
module example
{
    interface simple_add: simple
    {
        short add(in short x, in short y);
        short mult(in short a, in short b);
    };
    interface simple_mult: simple
    {
        short add(in short x, in short y);
        short mult(in short x, in short y);
    };
    interface add_mult<condn>: simple_add, simple_mult
    {...............};
};

Figure 4.11: Selective Inheritance using Conditional Interfaces

4.3.3 Design of the IDL Compiler

An interface definition language (IDL) compiler generates stub code from an interface definition. The stub code marshals the parameters on the client side, communicates through IPC/RPC kernel primitives with the server,
unmarshals the parameters on the server side and invokes the corresponding server procedure. The result returned from the server procedure has to be marshalled back to the client. As a result, the program can specify and use the remote interfaces as easily as local interfaces.

The design of the IDL compiler as shown in figure 4.12, has two phases. They are the front end and the back end. The details of these two phases are given below.

1) **The front end:** It reads the IDL source file and produces an interface definition in a particular target programming language like C++. This file is called as the presentation or interface file. This phase of the compiler uses Lex tool for lexical analysis and YACC tool for syntax analysis. The mapping from CORBA IDL keywords to C++ keywords is shown in table 4.1. Table 4.2 shows the mapping for different types of parameters, namely in, out, inout and return.

2) **The back end:** This phase is responsible to produce the stub code from the presentation file. It takes care of marshalling and unmarshalling the parameters through the Internet Interoperable Protocol (IIOP) in Common Data Representation (CDR). The C++ declaration file is included in the stub and skeleton for providing the declarations. IIOP is layered over TCP/IP. Hence the marshaled data are transmitted over TCP/IP sockets in CDR format.

<table>
<thead>
<tr>
<th>CORBA keyword</th>
<th>C++ keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td>header file</td>
</tr>
<tr>
<td>interface</td>
<td>abstract class</td>
</tr>
<tr>
<td>*condn interface</td>
<td>abstract class and condn if-then-else</td>
</tr>
<tr>
<td>short</td>
<td>int</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>arrays</td>
<td>arrays</td>
</tr>
<tr>
<td>const</td>
<td>const</td>
</tr>
<tr>
<td>struct</td>
<td>struct</td>
</tr>
<tr>
<td>enum</td>
<td>enum</td>
</tr>
</tbody>
</table>
### Table 4.2. Mapping of in, out, inout and ret types in CORBA to C++

<table>
<thead>
<tr>
<th>CORBA keywords</th>
<th>In parameter</th>
<th>Out parameter</th>
<th>Inout parameter</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>CORBA_Long</td>
<td>CORBA_Long&amp;</td>
<td>CORBA_Long&amp;</td>
<td>CORBA_Long</td>
</tr>
<tr>
<td>unsigned long</td>
<td>CORBA_Ulong</td>
<td>CORBA_Ulong&amp;</td>
<td>CORBA_Ulong&amp;</td>
<td>CORBA_Ulong</td>
</tr>
<tr>
<td>short</td>
<td>CORBA_Short</td>
<td>CORBA_Short&amp;</td>
<td>CORBA_Short&amp;</td>
<td>CORBA_Short</td>
</tr>
<tr>
<td>unsigned short</td>
<td>CORBA_UShort</td>
<td>CORBA_UShort&amp;</td>
<td>CORBA_UShort&amp;</td>
<td>CORBA_UShort</td>
</tr>
<tr>
<td>float</td>
<td>CORBA_Float</td>
<td>CORBA_Float&amp;</td>
<td>CORBA_Float&amp;</td>
<td>CORBA_Float</td>
</tr>
<tr>
<td>double</td>
<td>CORBA_Double</td>
<td>CORBA_Double</td>
<td>CORBA_Double</td>
<td>CORBA_Double</td>
</tr>
<tr>
<td>char</td>
<td>CORBA_Char</td>
<td>CORBA_Char&amp;</td>
<td>CORBA_Char&amp;</td>
<td>CORBA_Char</td>
</tr>
<tr>
<td>boolean</td>
<td>CORBA_Bool</td>
<td>CORBA_Boolean</td>
<td>CORBA_Boolean</td>
<td>CORBA_Bool</td>
</tr>
<tr>
<td>octet</td>
<td>CORBA_Octet</td>
<td>CORBA_Octet&amp;</td>
<td>CORBA_Octet&amp;</td>
<td>CORBA_Octet</td>
</tr>
<tr>
<td>string</td>
<td>const char *</td>
<td>char *&amp;</td>
<td>char *&amp;</td>
<td>char</td>
</tr>
<tr>
<td>enum(E)</td>
<td>E</td>
<td>E&amp;</td>
<td>E&amp;</td>
<td>E</td>
</tr>
<tr>
<td>struct(S1)</td>
<td>const S1&amp;</td>
<td>S1&amp;</td>
<td>S1&amp;</td>
<td>S1</td>
</tr>
<tr>
<td>array(A)</td>
<td>const A</td>
<td>A</td>
<td>A</td>
<td>A-slice*</td>
</tr>
<tr>
<td>sequence(Q)</td>
<td>Const Q&amp;</td>
<td>Q*&amp;</td>
<td>Q&amp;</td>
<td>Q*</td>
</tr>
<tr>
<td>interface(l)</td>
<td>l_ptr</td>
<td>l_ptr&amp;</td>
<td>l_ptr&amp;</td>
<td>l_ptr</td>
</tr>
</tbody>
</table>

* keyword for the proposed method

The results of implementation of selective inheritance in CORBA are given in section 4.4.

### 4.4 Experimental Results

This section presents the experimental results for selective inheritance using conditional interfaces. An example CORBA IDL file is shown in figure 4.13.
module example
{
    interface addf
    {
        short add(in short a, in short b);
        short mul(in short a, in short b);
    }
    interface mulf
    {
        short add(in short a, in short b);
        short mul(in short a, in short b);
    }
    interface<2> cond: addf, mulf
    {
        short sub(in short a, in short b);
        short div(in short a, in short b);
    }
};

Figure 4.13 : Input CORBA IDL – example.idl

class addf
{
    public:
        virtual CORBA_Short add(CORBA_Short &a,
                                CORBA_Short &b)=0;
        virtual CORBA_Short mul(CORBA_Short &a,
                                CORBA_Short &b)=0;
};
class mulf {
#define X 2
#if X==1
    class cond: public addf
#else
    class cond: public mulf
#endif
{
    public:
        virtual CORBA_Short sub(CORBA_Short &a,
                                CORBA_Short &b)=0;
        virtual CORBA_Short div(CORBA_Short &a,
                                CORBA_Short &b)=0;
};

Figure 4.14 : Presentation File – example.cpp

The output of the front end of the IDL compiler is shown in figure 4.14. It is the interface file in the target programming language C++. The mapping from IDL to C++ is taken from tables 4.1 and 4.2. Based on this interface, stubs and skeletons are generated which take care of the communication mechanism.
4.5 Summary

The conventional creation and destruction of CORBA objects, leads to under-utilization of resources. Hence a base interface with operations for reference counting, instantiation and destruction of CORBA objects is implemented. All the user defined interfaces inherit from this base interface. It includes operations to initialize increment and decrement reference count, and to destroy the CORBA object. This method minimizes the wastage of memory resources, by deactivating the CORBA objects when not used by the clients.

CORBA specifications do not address the issue of selective inheritance and its implementation. An IDL compiler is designed to incorporate selective inheritance of the interfaces using conditional interfaces. This method is simple but effective.