Objects By Value

Joint Revised Submission - w/ Errata

BEA Sytems, Inc.
International Business Machines Corporation
Iona Technologies, Plc.
Netscape Communications Corporation
Novell, Inc.
Visigenic Software, Inc.

Supported by:
Hewlett-Packard Company

OMG TC Document orbos/98-01-18

The submitting companies listsed above have all contributed to this “merged” submission. These companies recognize that this draft joint submission is the joint intellectual property of all the submitters, and may be used by any of them in the future, regardless of whether they ultimately participate in a final joint submission.

Table of Contents

Preface 1

1.1 Cosubmitting Companies

The following companies are pleased to jointly submit this specification in response to the OMG Objects By Value RFP (Document ORBOS/96-06-14):

- BEA Systems, Inc.
- International Business Machines Corporation
- Iona Technologies, Plc.
- Netscape Communications Corporation
- Novell, Inc.
- Visigenic Software, Inc.

1.2 Status of this document

This document is the final revised submission with several minor errors (listed below) fixed and indicated by change bars in the document.

It is assumed that when this submission is adopted it will become part of the CORBA 2.3 core. The nature of this submission is that its changes are not isolated to small parts of the core.

Chapter 10 “merely” outlines the exact changes to CORBA 2.2. It does not contain the complete “editing instructions” required to update CORBA 2.2 because CORBA 2.2 is still under development and it was not possible to provide the exact text required. The motion to adopt this submission will include a motion to form an RTF made up of the submitters (at least) whose charter will be to produce the detailed changes for CORBA 2.3. The changes will then be voted on following normal OMG RTF procedures.

1.2.1 Changes to 98-01-01

- In “Partial Type Information and Versioning” on page 5-54, specifying the list of safe bases, in addition to the actual type, was omitted. The corresponding change to the GIOP format in “The Format” on page 5-57 was also made.
- A dependency on new functionality being added to the JDK was identified. In order to provide an alternative, a note was added to the **org.omg.CORBA.portable.OutputStream** in “Java ORB Portability Interfaces” on page 6-70. A value’s mapped Java class must also implement **org.omg.portable.Streamable** if it is to be used with a version of the JDK that does not provide the new feature (see Section 6.3.1, “Basics for Stateful Values,” on page 6-61).
- In the C++ mapping (see Section 7.3.1, “Value Data Members,” on page 7-72 and the examples following) accessors and modifers are now virtual in order to allow application developers to reimplement the generated default behavior.
- Specification of RepositoryID format was added. See Section 5.6.1, “CORBA Repository Ids,” on page 5-49.
- A typo in “Value Operations” on page 7-74 was fixed.
- Fixed the formatting of this section.
- Missing “in” keyword in syntax rule for <init_param_decl> in Section 5.4.1, “Syntax,” on page 5-43.
- Missing “in” specifiers in operation declarations in some of the examples.
- Missing <identifier> in syntax productions for <value_abs_dcl> and <header> Section 5.4.1, “Syntax,” on page 5-43.
- Change the syntax to not overload the colon to indicate both the inheritance from abstract interfaces and values, and the support of interfaces. Although there are many changes throughout the document, they are mostly to change the use of inherits/derives to supports and fix the grammar. The primary change is to the syntax production in Section 5.4.1, “Syntax,” on page 5-43.

1.3 Guide to the Submission

This revised submission proposes an approach to handling objects by value in an OMA context. Elements include:

- Value Interfaces
- IDL extensions
- Parameter Passing Semantics
- GIOP/IIOP Extensions
- Abstract Interfaces

Chapter 5 describes the approach and contains the bulk of the proposal including IDL extensions and GIOP/IIOP extensions. Chapter 6 describes the changes to Java mapping. Chapter 7 describes the required changes to the C++ mapping. Chapter 8 contains the proposal to add abstract interfaces to CORBA.

1.4 Missing Items

1.5 Conventions

IDL appears using this font.

Concrete programming language (Java, C++, etc.) code appears using this font.

Please note that any change bars have no semantic meaning. They are present for the convenience of readers and submitters (and the editor who wants to be able to tell what changed between various drafts).

1.6 Submission Contact Points

The primary editor, and contact point for this submission is:

Jeff Mischkinsky
Visigenic Software, Inc.
951 Mariner's Island Blvd. Suite 120
San Mateo, CA 94404
USA
phone: +1 650 312 5158
email: jeffm@visigenic.com

The contact information for the other co-submitting companies is:

Dan Frantz
BEA Systems, Inc..
436 Amherst Street
Nashua, NH 03063
USA
*phone*e: +1 603 579-2519
*email*l: dan.frantz@beasys.com

Randy Fox
IBM Corp.
11400 Burnet Road
Austin, TX 78758
*phone*e: +1 512 838 2310
*email*l: randyfox@austin.ibm.com

Martin Chapman
IONA Technologies
The IONA Building
8-10 Lower Pembroke Street
Dublin 2, Ireland

phone: +353 1 662 5255
fax: +353 1 662 5244
email: mchapman@iona.com

David Stryker
Netscape Communications Corporation
501 E. Middlefield Road
Mountain View, CA 94043
USA
phone: +1 650 937-3454
email: stryker@netscape.com

Bill Cox
Novell, Inc.
2 Oak Way
Berkeley Heights, NJ 07922
phone: +1 908 790-5123
email: bill@novell.com

Proof of Concept 2

The core ideas presented in this submission are based on Visigenic's experience in implementing our Caffeine extensions to CORBA for Visibroker/Java.

Many of the ideas for custom marshaling are based upon experience with Iona's opaque extensions.

Issues dealing with codebase and code downloading are based on RMI work in partnership with JavaSoft on mapping RMI to IIOP.

The abstract interface extensions are based upon IBM's work on their San Francisco project.

Response to RFP Requirements 3

The following is a list of the requirements from the Objects By Value RFP (OMG orbos/96-06-14), specifying how this submission is responsive to the RFP.

Introduction

In the following requirements discussion, the terms “sending context” and “receiving context” are relative roles played by client and server contexts (programs, processes, whatever). If an object is being passed by value as an in parameter, then the caller is the sending context and the callee is the receiving context. If the object is an out parameter, then the caller is the receiving context and the callee is the sending context. If the parameter is an inout parameter, then the roles are meaningful only with respect to a single parameter directional flow. If the parameter is a return parameter, the caller is the receiving context and the callee is the sending context.

3.1 Pass By Value Semantics

Submissions shall define precisely the semantics of passing objects by value. Specifically, the following issues shall be addressed:

- ***what is the relationship between the identity of the object in the sending context and the object in the receiving context (including any security implications)?***

None, copy semantics are used.

- ***what is the relationship between the implementation in the sending context and the implementation in the receiving context? Must they be identical? If not, how is equivalency for the purposes of passing by value established?***

Structural compatibility. We guarantee that code will not break, but as with “normal” CORBA there are no semantic guarantees.

- ***what happens when there is no appropriate implementation available in the receiving context?***

A policy for searching for compatible implementations is defined.

- ***what are the relationships between the primary (or most-derived) interface of an object being passed in the sending context, the interface type of the parameter declaration, and the primary (or most-derived) interface of the object in the receiving context? Can any object supporting the declared parameter interface type be passed by value? If an object being passed in the sending context supports interfaces that are more derived than the parameter interface, will the resulting object in the receiving context also support those more-derived interfaces?***

The submission supports “regular” CORBA subtyping semantics. Stateful values may only singly inherit from other stateful values. Parameters are passed by value or by reference depending upon their formal type. Parameters with a formal value type are passed by value. Parameters with a formal ordinary interface type are passed by reference. If the formal type is an abstract interface, then the determination of how an actual parameter is passed is made at runtime.

3.2 Interoperability

- ***Submissions shall describe how interoperability may be ensured between different ORB implementations when passing objects by value. All information required for ORB interoperability shall be precisely described and exposed in some standard format. For example, if a submission proposes to ensure interoperability of passing by value based on information in the object's interface (IDL or IR) with some additional annotations, the submission shall describe a standard format for the annotation and medium (media) though which it will be expressed.***

The submission does this. It extends GIOP and IIOP.

3.3 Memory Management

- ***Submissions shall specify the memory management rules for by-value parameters.***

They are provided.

3.4 IDL Changes

- ***Submissions shall provide justification and rationale for any modifications or extensions to IDL, and to any new prescribed interfaces or modifications of existing CORBA interfaces.***

They are provided.

3.5 Language Mapping Changes

- ***Any changes to adopted IDL language mappings necessary to allow use of your objects-by-value mechanism from those languages shall be specified.***

They are provided.

3.6 GIOP and Wire Protocol Changes

- ***Any changes to GIOP and the CORBA 2 interoperability architecture, including marshalling and on-the-wire formats, shall be specified.***

They are specified.

3.7 Discuss Upwardly Incompatible Changes

- ***Because of the consequences of changing existing specifications, particularly upwardly incompatible changes, submitters should carefully consider the implications of changes, and fully document their implications in their submissions. A migration plan shall be included for upwardly incompatible changes.***

There are none.

3.8 Design Rationale

- ***The language mapping should be prefaced by an explanation of the design rationale.***

A separate chapter outlining some of the design rationale is provided. Where appropriate there is discussion in the main body of the submission.

Overall Design Rationale

4

This chapter discusses some of the rationale behind the choices that were made for this mapping.

General issues are discussed in the Introduction.

4.1 Protected Fields in State Definition

We chose not to introduce a notion of protected data member following the lead of the Enhanced Portability specification which states that a derived implementation can only rely on the public interface of its base. If it becomes clear that this notion is needed for value types it will be possible to revise the specification in an upwardly compatible way.

4.2 Single Inheritance Issues

Because we are stepping over the line that separates pure interface specification from implementation specification, concrete programming language issues must be considered. There are well-known difficulties in implementing multiple implementation inheritance. As a result many OO languages do not support it, e.g. Java and Smalltalk. We therefore choose to adopt a model which supports multiple inheritance for interfaces or value with no state, but only single inheritance as soon as some state is declared.

This restriction allows a clean and efficient mapping to the major OO programming languages, Java, C++, and Smalltalk.

4.3 Base Value Type

By default value types are not CORBA Objects, i.e. they do not inherit from CORBA::Object. The rationale for that decision is that mandating CORBA::Object semantics for all values is overkill: A important expected use of value types is to

support lightweight “data objects” which by they very nature are always passed by value (like a date, or a matrix), if value types had existed at the time CORBA1.0 was defined pseudo objects (e.g. TypeCodes) could have been expressed cleanly as values too. Forcing these kinds of data objects to support all the apparatus of a CORBA::Object (e.g., get_interface, IOR support, etc.) is an unnecessary burden.

4.4 Narrowing from interface to value

Narrowing from interface to value is not automatically allowed. To do so requires the creation of a local copy of the value type instance in the receiving context. In order to successfully perform this operation, the receiving context would have to “go back” to the object reference’s implementation (server) and download the value. There is no guarantee that the receiving context even “knows” about such an implementation. It was deemed much safer and less confusing to force the designer to define a specific application level operation if this feature is desired.

4.5 Passing A Value Instance for an Interface Type

This section discusses in more detail the rationale behind the decision to require a reference to be sent when an instance of a value type whose type supports an interface is passed as an actual parameter for a formal parameter whose type is the interface.

Again consider the example of a value type that supports an interface and an operation which has a parameter whose formal type is the interface. The question is what is actually passed in an invocation when the actual instance is the value type. Serious consideration was given to allowing the actual value to be passed, under the covers so to speak. However there were several issues with this approach.

From an application/client perspective there is no problem. It only knows about the base interface and hence only uses the instance as if it were the base type. However there may be a change in the semantics in that the application is now manipulating a local copy. Any changes it makes stay local and are not propagated back to the sending context.

The receiving context also now has an implementation “living” within itself when all it thought was that it was receiving a reference. The situation is even stranger if one considers the case of an out parameter. In that case the client, which is initiating the invocation, could find itself all of sudden receiving an implementation.

There would have to be a way to let the sending side control whether a reference or a value was to be sent, thus necessitating an additional api of some sort.

A major change to GIOP would have to occur to allow an encoding of a reference either as an IOR or as a value. This change would either cause older servers to break (if they were not careful about checking version numbers) or limit the usefulness of new clients.

These issues and problems arise because the receiving context is not aware that it can now be sent a value instead of a reference. Hence the decision was made to force designers to make an explicit decision to support this capability by adding a new kind of interface to IDL which has the semantics that it may be implemented either by a reference (IOR) or by a value.

4.6 Boxed Values

Sometimes it is necessary to define a value type with a single data member inside its state section and no inheritance or methods. For example, when transmitting a string or sequence as an actual parameter on an interface operation or as a data member of a value type that is an actual parameter, it may be important to preserve any sharing of the string or sequence within the object graph being transmitted. Because current IDL data types do not preserve referential integrity in this way, this requirement is conveniently handled by using a value type. Value types also support the transmission of nulls (as a distinguished value), whereas IDL data types such as string and sequence (which are mapped to empty strings and sequences) do not. The Java to IDL mapping requires both preservation of referential integrity and transmission of nulls. Because it would be cumbersome to require the full IDL syntax for a value type for this specific usage, a shorthand IDL notation is introduced to cover this use of value types for simple containment of a single data member.

4.7 value keyword

There is a fair amount of language required to specify the obvious uses of the value keyword. Basically **value** is analogous to **interface**, and is used in the grammar to introduce a value declaration. **ValueBase** is analogous to **Object** and is used to denote the base type. The rest of the language is to describe how the parsing occurs.

4.8 Use of Helper classes

Helper and holder classes are used in keeping with the decisions that were made in the adopted Java language mapping specification.

Value Types 5

5.1 Introduction

Pure CORBA (2.1) objects cannot be passed by value, only object reference semantics are supported. Objects, more specifically, interface types that objects support, are defined by an IDL interface, allowing arbitrary implementations. There is great value, which is described in great detail elsewhere, in having a distributed object system that places almost no constraints on implementations.

However there are many occasions in which it is desirable to be able to pass an object by value, rather than by reference. This may be particularly useful when an object's primary "purpose" is to encapsulate data, or an application explicitly wishes to make a "copy" of an object. We assume that the purpose of this feature is NOT to implement replication and/or caching[1].

In this submission, we define the semantics of pass by value as being similar to that of standard programming languages. The receiving side of a parameter passed by value receives a "new" instance of the object, with a separate identity from that of the sending side. Once the parameter passing operation is complete, no relationship is assumed to exist between the two instances.

Because it is necessary for the receiving side to instantiate an instance, it must necessarily know something about the object's state. This submission extends CORBA (and IDL) to include the notion of a **value type**.

It has also proved desirable to allow the receiving side the flexibility to receive either an Object type or a Value type. This submission proposes extending CORBA to support the notion of an **abstract interface type** which allows a designer to specify that an operation can explicitly support receiving either a value type or an interface type at runtime. See Chapter 8, "Abstract Interfaces" for more details.

1.But we believe that more complex caching and replication functionality can be built on the top of the feature set proposed here.

Value types provide semantics that bridge between CORBA structs and CORBA interfaces:

- They support description of complex state (i.e arbitrary graphs, with recursion and cycles)
- Their instances are always local to the context in which they are used (because they are always copied when passed as a parameter to a remote call)
- They support both public and private (to the implementation) data members.
- They can be used to specify the state of an object implementation i.e they can support an interface.
- They support single inheritance (of **value**) and can support multiple **interfaces**.
- They may be also be **abstract**.

The submitters were also faced with the problem of adding several new keywords to IDL. The submission proposes some minor extensions to the way in which IDL is parsed in order to make it possible to add keywords to IDL without breaking existing programs which use those keywords as identifiers.

The submitters are also working closely with submitters on several other ongoing RFPs. These include the reverse Java to IDL language mapping, Persistent State Service, and CORBA Components RFP. We expect that facilities added by this proposal to be used in those submissions. The final adopted specifications should be closely coordinated.

It is our belief that the concepts provided by this submission would enable the OMG to eliminate almost all the PIDL contained in the CORBA specification. It is our recommendation that an RFP to do so be issued.

5.2 Goals

The goals of this proposal are to:

- provide a simple and very robust model that builds on existing CORBA semantics.
- respect the current CORBA model that distinguishes, in its type system, interface types from constructed data types.
- minimize changes to IDL.
- maintain flexibility in ORB implementations, while guaranteeing interoperability and portability.
- expose complex implementation states in a language independent manner.
- guarantee consistent semantics and use across languages.
- support passing of objects by value between clients and servers implemented in different languages.
- provide natural and convenient support in Java and C++

A key design center for this submission is to provide natural and convenient support for Java users of CORBA. Our feeling is that a key factor in the continued adoption and deployment of CORBA will be the ease of interoperability between Java (clients and servers) and other language platforms, particularly C++. Hence this proposal is designed so that it may be efficiently and easily implemented in Java. Where potential features (e.g. multiple inheritance) increased complexity, we chose to simplify.

5.3 Description

5.3.1 Value Types

5.3.1.1 Architecture

This submission introduces the notion of a **value** type to the OMA. This has profound implications on the type system and has forced careful consideration of issues surrounding the nature of interfaces, the separation of interface definition and implementation, and the guarantees that CORBA makes with respect to issues of state consistency and coherency.

The basic notion is relatively simple. A **value** type is, in some sense, half way between a “regular” IDL interface type and a struct. The use of a value type is a signal from the designer that some additional properties (state) and implementation details be specified beyond that of an interface type. Specification of this information puts some additional constraints on the implementation choices beyond that of interface types. This is reflected in both the semantics specified herein, and in the language mappings.

An essential property of value types is that their implementations are always local. That is, the explicit use of value type in a concrete programming language is always guaranteed to use a local implementation, and will not require a remote call. They have no identity (their value is their identity) and they are not “registered” with the ORB.

There are two kinds of value types, concrete (or stateful) value types, and abstract (stateless) ones. As explained below the essential characteristics of both are the same. The differences between them result from the differences in the way they are mapped in the language mappings. In this specification the semantics of value types apply to both kinds, unless specifically stated otherwise.

Concrete (stateful) values add to the expressive power of (IDL) structs by supporting:

- single derivation (from other value types)
- support of multiple interfaces
- arbitrary recursive value type definitions, with sharing semantics providing the ability to define lists, trees, lattices and more generally arbitrary graphs using value types.
- null value semantic

When an instance of such a type is passed as a parameter, the sending context marshals the state (data) and passes it to the receiving context. The receiving context instantiates a new instance using the information in the GIOP request and unmarshals the state. It is assumed that the receiving context has available to it an implementation that is consistent with the sender's (i.e. only needs the state information), or that it can somehow download a usable implementation. Provision is made in the on-the-wire format to support the carrying of an optional call back object (**CodeBase**) to the sending context which enables such downloading when it is appropriate.

It should be noted that it is possible to define a concrete value type with an empty state as a degenerate case.

Abstract Values

Value types may also be abstract. They are called abstract because an abstract value type may not be instantiated. Only concrete types derived from them may be actually instantiated and implemented. Their implementation, of course, is still local. However, because no state information may be specified (only local operations are allowed), abstract value types are not subject to the single inheritnace restrictions placed upon concrete value types. Essentially they are a bundle of operation signatures with a purely local implementation. This distinction is made clear in the language mappings for abstract values.

Note that a concrete value type with an empty state is not an abstract value type. They are considered to be stateful, may be instantiated, marshaled and passed as actual parameters. Consider them to be a degenerate case of stateful values.

5.3.1.2 State Definition

Data members that define the state of a value type may be private or public. The default is private and the **public** modifier can be used. The annotation directs the language mapping to hide or expose the different parts of the state to the clients of the value type. The private part of the state is only accessible to the implementation code and the marshaling routines.

Note that certain programming languages may not have the built in facilities needed to distinguish between public and private members. In those cases, the language mapping will specify the rules that programmers have to follow.

5.3.1.3 Operations

Operations defined on a value type specify signatures whose implementation can only be local. Because these operations are local, they must be directly implemented by a body of code in the language mapping (no proxy or indirection is involved).

The language mappings of such operations require that instances of value types passed into such local methods are passed by reference (programming language reference semantics, not CORBA object reference semantics) and that a copy is not made. Note, such a (local) invocation is not a CORBA invocation. Hence it is not mediated by the ORB, although the API to be used is specified in the language mapping.

The (copy) semantics for instances of value type are only guaranteed when instances of these value types are passed as a parameter to an operation defined on a CORBA interface, and hence mediated by the ORB. If an instance of a value type is passed as a parameter to a method of another value type in an invocation, then this call is a “normal” programming language call. In this case both of the instances are local programming language constructs. No CORBA style copy semantics are used and the normal semantics for the programming language in question apply.

Operations on the value type are supported in order to guarantee the portability of the client code for these value types. They have no representation on the wire and hence no impact on interoperability.

5.3.1.4 Initializers

In order to ensure portability of value implementations, designers may also define the signatures of initializers (or constructors) for non abstract value types. Syntactically these look like local operations except that they use the keyword identifier **init** for the “name” of the operation, have no return type, and must use only **in** parameters. There may be any number of init() declarations, as long as the signatures of all the initializers declared within the same scope are unique. Using the same signature as one found in a less-derived type is allowed.

The mapping of initializers is language specific and may not always result in a one to one correspondence between initializer signatures and the programming language constructs into which they map. This is because the mapping from IDL types into programming language types is not isomorphic; several different IDL types may map to the same programming language type. Hence defining initializers with the same number of parameters with types that are “similar” (e.g. **char** and **wchar**, signed and unsigned integers, etc.) should be done with care.

5.3.1.5 Example

```
interface A {};

interface B {};
```

```
value Example supports A, B {
    // state definition
        short a;
        public long b; // public field, the default is private
        string c;
        float d;
        Example eg;    // passed by value + recursive definition
        A anA;         // passed by reference
    // initializers
        init(in short a);
        init(in short a, in long b);
    // operations
        short f (in A x);
        long g(in Example x);
};
```

A more realistic value type might be:

```
interface AnInterface {};

typedef sequence<unsigned long> WeightSeq;

value WeightedBinaryTree {
    // state definition
        unsigned long weight;
        WeightedBinaryTree left;
        WeightedBinaryTree right;
    // initializer
        init(in unsigned long w);
    // local operations
        WeightSeq pre_order();
        WeightSeq post_order();
};

value WTree: WeightedBinaryTree supports AnInterface {
    };
```

5.3.2 *Typing and Substitutability Issues*

5.3.2.1 *Inheritance Relationships*

Values may be derived from other values and can support interfaces.

Once implementation (state) is specified at a particular point in the inheritance hierarchy, all derived value types (which must of course implement the state) may only derive from a single (concrete) value type. They can however derive from other additional abstract values and support additional interfaces.

The single immediate base concrete value type, if present, must be the first element specified in the inheritance list of the value declaration's IDL. It may be followed by other abstract values from which it inherits. The interfaces and abstract interfaces that it supports are listed in any order following the **supports** keyword.

A stateful value that derives from another stateful value may specify that it is safe to "truncate" (seeSection , "Value instance -> Value type," on page 5-29) an instance to be an instance of its immediate parent (stateful) value type.

These rules are summarized in the following table:

Table 5-1 Allowable Inheritance Relationships

may inherit from:	Interface	Abstract Value	Stateful Value
Interface	multiple	no	no
Abstract Value	supports	multiple	n/a
Stateful Value	supports	multiple	single (may be safe)

5.3.2.2 Value Base Type

All value types have a conventional base type called **CORBA::ValueBase.** This is a type which fulfills a role that is similar to that played by CORBA::Object. Conceptually it supports the common operations available on all value types. See Section 5.4.3, "ValueBase Operations," on page 5-45 for a description of those operations. In each language mapping **CORBA::ValueBase** will be mapped to an appropriate base type that supports the marshaling/unmarshaling protocol as well as the model for custom marshaling.

The mapping for other operations which all value types must support, such as getting meta information about the type, may be found in the specifics for each language mapping.

5.3.2.3 Value Type vs. Interfaces

By default value types are not CORBA Objects. In particular instances of value types do not inherit from CORBA::Object and do not support normal object reference semantics. However it is always possible to explicitly declare that a given value type supports an interface type. In this case instances of the type may support CORBA object reference semantics (if they are registered with the ORB using an object adapter.).

5.3.2.4 Parameter Passing

This section describes semantics when a value instance is passed as parameter in a CORBA invocation. It does not deal with the case of calling another non-CORBA (i.e. local) programming method which happens to have a parameter of the same type.

Value vs. Reference Semantics

Determination of whether a parameter is to be passed by value or reference is made by examining the parameter's formal type (i.e the signature of the operation it is being passed to). If it is a value type then it is passed by value. If it is an ordinary interface then it is passed by reference (the case today for all CORBA objects). This rule is simple and consistent with the handling of the same situation in recursive state definitions or in structs.

In the case of abstract interfaces, the determination is made at runtime. See Section 8.3, "Semantics of Abstract Interfaces" for a description of the rules.

Sharing Semantics

In order to be expressive enough to describe arbitrary graphs, lattice, trees etc., value types support sharing and null semantics. Instances of a value type can be shared by others across or within other instances. They can also be null. This is unlike other IDL data types such as structs, unions, and sequences which can never be shared. The sharing of values within and between the parameters to an operation, is preserved across an invocation, i.e the graph which is reconstructed in the receiving context is structurally isomorphic to the sending context's.

Identity Semantics

When an instance of the value type is passed as a parameter, an independent copy of the instance is instantiated in the receiving context. That copy is a separate independent entity and there is no explicit or implicit sharing of state.

Any parameter type

When an instance of a value type is passed to an **any**, as with all cases of passing instances to an **any**, it is the responsibility of the implementer to insert and extract and the value according to the language mapping specification.

5.3.2.5 *Substitutability Issues*

The substitutability requirements for CORBA require the definition of what happens when an instance of a derived value type is passed as a parameter that is declared to be a base value type or an instance of a value type that supports an interface is passed as a parameter that is declared as the interface type.

There are two cases to consider in this submission: the parameter type is an interface, and the parameter type is a value type.

Value instance -> Interface type

Assume that we have an instance of a value type that supports an interface type. We have an IDL operation whose signature contains a parameter whose formal type is the interface. The following rule applies to this situation:

- If the value type instance (in the sending context) has not been registered with ORB, then a **OBJECT_NOT_EXIST** exception with an identified minor code (see Section 5.10, “Minor Exception Codes”) is raised. Otherwise the instance’s object reference is used and it is passed as normal.

Value instance -> Value type

In this case the receiving context is expecting to receive a value type. If the receiving context currently has the appropriate implementation class then there is no problem.

If the receiving context does not currently hold an implementation with which to reconstruct the original type then the following algorithm is used to find such an implementation:

1. *Load*
   - Attempt to load (locally in C/C++, possibly remotely in Java and other “portable” languages) the real type of the object (with its methods). If this succeeds, OK
2. *Truncate*
   - Truncate the type of the object to the base type (if specified as **safe** in the IDL). Truncation can never lead to faulty programs because, from a structural point view base types structurally subsume a derived type and an object created in the receiving context bears no relationship with the original one. However, it might be semantically puzzling, as the derived type may completely re-interpret the meaning of the state of the base. For that reason a derived value needs to indicate if it is is safe to truncate to its immediate non-abstract parent.
3. *Raise Exception*
   - If none of these work or are possible, then raise the **NO_IMPLEMENT** exception.

Safeness is a transitive property.

Example

```
value EmployeeRecord {        // note this is not a CORBA::Object
    // state definition
        string name;
        string email;
        string SSN;
    // initializer
        init(in string name, in string SSN);
};

value ManagerRecord: safe EmployeeRecord {
    // state definition
        sequence<EmployeeRecord> direct_reports;
};
```

5.3.2.6 Widening/Narrowing

As has been described above, value type instances may be widened/narrowed to other value types. Each language mapping is responsible for specifying how these operations are made available to the programmer.

Narrowing from an interface type instance to a value type instance is not allowed. If the interface designer wants to allow the receiving context to create a local implementation of the value type, i.e. a value representing the interface, an operation which returns the appropriate value type may defined.

5.3.3 Value Boxes

It is often convenient to define a value type with no inheritance or methods and with a single data member. A shorthand IDL notation is used to simplify the use of value types for this kind of simple containment, sometimes referred to as a “value box”.

This particularly useful for strings and sequences. Basically one does not have to create what is in effect an additional namespace that will contain only one name.

For example, the IDL definition

```
module Example {
    interface Foo {
        ...  /* anything */
    };
    value FooSeq sequence<Foo>;
    interface Bar {
        void doIt (in FooSeq seq1, in FooSeq seq2);
    };
};
```

could be used to ensure a single copy of the Foo sequence is transmitted and received when the operation **doIt()** is invoked with the same sequence data item passed as the seq1 and seq2 parameters.

This IDL provides similar functionality to writing the following IDL. However the type identities (repository ID’s) would be different in this case.

```
module Example {
    interface Foo {
        ... /* anything */
    };
    value FooSeq {
        public sequence<Foo> data;
    };
    interface Bar {
        void doIt (in FooSeq seq1, in FooSeq seq2);
    };
};
```

The former is easier to manipulate after it is mapped to a concrete programming language.

5.3.3.1 Standard Value Box Definitions

For some CORBA-defined types for which preservation of sharing and transmission of nulls are likely to be important, the following value box type definitions are added to the CORBA module:

```
module CORBA {
    value StringValue string;
    value WStringValue wstring;
};
```

5.3.4 LifeCycle issues

Value type instances are always local to their creating context, i.e in a given language mapping an instance of a value type is always created as a local “language” object with no POA semantics attached to it initially.

When passed using a CORBA invocation, a copy of the value is made in the receiving context and that copy starts its life as a local programming language entity with no POA semantics attached to it.

If a value type supports an ordinary interface type, its instances may also be passed by reference when the formal parameter type is an interface type (see Section 5.3.2.4, “Parameter Passing,” on page 5-27). In this case they behave like ordinary object implementations and must be associated with a POA policy (CORBA 2.2) or a BOA (CORBA 2.1) and also be registered with the ORB (e.g. POA::activate_object() (CORBA 2.2), BOA::obj_is_ready() (CORBA 2.1), etc.) before they can be passed by reference. Not registering the value as a CORBA object and/or not associating an appropriate policy with it results in an exception when trying to use it as a remote object, the “normal” behavior. The exception raised shall be **OBJECT_NOT_EXIST** with an identified minor code (see Section 5.10, “Minor Exception Codes”).

5.3.4.1 Creation and Factories

When an instance of a value type is received by the ORB, it must be demarshaled and an appropriate factory for its actual type found in order for the new instance to be created. The type is encoded by the RepositoryID which is passed over the wire as part of an invocation. The mapping between the type (as specified by the RepositoryID) and the factory is language specific. In certain languages it may be possible to specify default policies that are used to find the factory, without requiring that specific routines be called. In others the runtime and/or generated code may have to explicitly specify the mapping on a per type basis. In others a combination may be used. In any event the ORB implementation is responsible for maintaining this mapping See Section 5.3.6.3, “Language Specific Value Factory Requirements” for more details on the requirements for each language mapping.

5.3.5 Security Considerations

The addition of value types has few impacts on the CORBA security model. In essence, the security implications in defining and using value types are similar to those involved with the use of IDL structs. Instances of value types are mapped to local, concrete programming language constructs. Except for providing the marshaling mechanisms, the ORB is not directly involved with accessing value type implementations. This specification is mostly about 2 things: how value types manifest themselves as concrete programming language constructs and how they are transmitted.

To see this consider how value types are actually used. The IDL definition of a value type in conjunction with a programming language mapping is used to generate the concrete programming language definitions for that type.

Let us consider its lifecycle. In order to use it, the programmer uses the mechanisms in the programming language to instantiate an instance. This is instance is a local programming language construct. It is not “registered” with the ORB, object adapter, etc. The programmer may manipulate this programming construct just like any other programming language construct. So far there are no security implications. As long as no ORB-mediated invocations are made, the programmer may manipulate the construct. Note, this includes making “local”, non ORB-mediated calls to any locally implemented operations. Any assignments to the construct are the responsibility of the programmer and have no special security implications.

Things get interesting when the program attempts to pass one of these constructs through an orb-mediated invocation (i.e. calls a stub which uses it as a parameter type, or uses the DII). There are two cases to consider:

Value as Value

The formal type of the parameter is a value. This case is no different from using any other kind of a value (long, string, struct, etc.) in a CORBA invocation, with respect to security. The value (data) is marshaled and delivered to the receiving context. On the receiving context, the knowledge of the type is used (at least implicitly) to find the factory to create the correct local programming language construct. The data is then

unmarshaled to fill in the newly created construct. This is similar to using other values (longs, strings, structs, etc.) except that the knowledge of the factory is not “built-in” to the ORB’s skeleton/DSI engine.

Value as Object Reference

The formal type of the parameter is an interface type which is supported by a value. The program must have “registered” the value with an object adapter and is really using the returned object reference (see for the specific rules.) Thus this case “reduces” to a regular CORBA invocation, using a regular object reference. An IOR is passed to the receiving context. All the “normal” security considerations apply. From the point of view of the receiving context, the IOR is a “normal” object reference. No “special” rules, with respect to security or otherwise, apply to it. The fact that it is ultimately a reference to an implementation that was created from instantiating and registering an value type implementation is not relevant.

In both of these cases, security considerations are involved with the decision to allow the ORB-mediated invocation to proceed. The fact that a value type is involved is not material.

5.3.6 Language Mappings

5.3.6.1 General Requirements

A concrete value will map to a concrete usable “class” construct in each programming language, plus possibly some helper classes where appropriate. In Java, C++, and Smalltalk this will be a real concrete class. In C it will be a struct. This mapping will be captured in the Interface Repository by storing the extra information.

An abstract value is mapped to some sort of an abstract construct--an interface in Java, and an abstract class with pure virtual function members in C++.

Tools that implement the language mapping are free to “extend” the implementation classes with “extra” data members and methods. When an instance of such a class is used as a parameter, only the portions that correspond directly to the IDL declaration, are marshaled and delivered to the receiving context. This allows freedom of implementations while preserving the notion of contract and type safety in IDL.

5.3.6.2 Language Specific Marshaling

Each language mapping specifies an appropriate marshaling/unmarshaling protocol and the entry point for custom marshaling/unmarshaling.

5.3.6.3 *Language Specific Value Factory Requirements*

Each language mapping shall specify the algorithm and means by which RepositoryIDs are used to find the appropriate factory for an instance of a value type so that it may be created as it is unmarshaled “off the wire”.

It is desirable, where it makes sense, to specify a “default” policy for automatically using RepositoryIDs that are in common formats to find the appropriate factory. Such a policy can be thought of as an implicit registration. Additionally, IDL is defined which provides a portable way to register the association between an arbitrary RepositoryID and value factory with the ORB runtime.

Each language mapping shall specify how and when the registration occurs, both explicit and implicit. The registration must occur before an attempt is made to unmarshal an instance of a value type. If the ORB is unable to locate and use the appropriate factory, then a **MARSHAL** exception with an identified minor code (see Section 5.10, “Minor Exception Codes”) is raised.

Because the type of the factory is programming language specific and each programming language platform has different policies, the factory type must be specified as **native**. It is the responsibility of each language mapping to specify the actual programming language type of the factory.

```
// IDL
native ValueFactory;

interface ORB {
    ValueFactory register_value_factory(
                            in RepositoryId id,
                            in ValueFactory factory
                        );
    void unregister_value_factory(in RepositoryId id);
    ValueFactory lookup_value_factory(in RepositoryId id);
    ...
};
```

The **register_value_factory()** operation registers the **ValueFactory** passed to it as the factory for the type identified by the **RepositoryId** string argument. If a factory was already registered for that type, the old factory is returned, otherwise a language mapping specified specific value (usually null if the language mapping supports it) for the native ValueFactory. If the registration fails then a **BAD_PARAM** exception with an identified minor code (see Section 5.10, “Minor Exception Codes”) is raised.

The **lookup_value_factory()** operation returns the ValueFactory registered for the specified RepositoryId string, either explicitly (because the registration routine was called) or implicitly, for the specified RepositoryId string. If it is unable to locate a factory then a **BAD_PARAM** exception with an identified minor code (see Section 5.10, “Minor Exception Codes”) is raised.

The **unregister_value_factory()** operation unregisters the factory already associated with the specified **RepositoryId** string argument. If it is unable to locate the factory then a **BAD_PARAM** exception with an identified minor code (see Section 5.10, “Minor Exception Codes”) is raised.

Although technically these definitions are PIDL (because the operations are defined on the ORB pseudo-object), the language mappings for these types and operations shall treat them as if they are regular IDL and the standard language mapping rules shall be followed.

5.3.6.4 Value Method Implementation

The mapped class must support method bodies (i.e. code) that implement the required IDL operations. The means by which this association is accomplished is a language mapping “detail” in much the same way that an IDL compiler is.

5.3.7 Custom Marshaling

Value types can override the default marshaling/unmarshaling model and provide their own way to encode/decode their state. Custom marshaling is intended to be used to facilitate integration of existing “class libraries” and other legacy systems. It is explicitly not intended to be a standard practice, nor used in other OMG specifications to avoid “standard ORB” marshaling.

The fact that a value type has some custom marshaling code is declared explicitly in the IDL. This explicit declaration has two goals:

- type safety: stub and skeleton can know statically that a given type is custom marshalled and can then do sanity check on what is coming over the wire.
- efficiency: for value types that are not custom marshaled no run time test is necessary in the marshaling code.

A custom marshaled value type is indicated syntactically by use of the custom modifier. It may also have an optional state definition. The state definition is treated the same as that of a non custom value type for mapping purposes, i.e. the fields show up in the same fashion in the concrete programming language. It is provided to help with application portability.

A custom marshalled value type is always a stateful value type.

```
// Example IDL

custom value T {
    // optional state definition
    ...
};
```

Custom value types can never be safely truncated to base i.e they always require an exact match for their RepositoryId in the receiving context.

Once a value type has been marked as custom, a marshaling policy object needs to be registered for it. The marshaling policy encapsulates the application code that can marshal and unmarshal instances of the value type over a stream using the CDR encoding. It is the responsibility of the implementation to marshal the state of all of its base types.

Non-custom value types may not (transitively) inherit from custom value types.

5.3.7.1 Streaming A Custom Value

The following IDL defines the interfaces that are used to support custom marshaling.

```
module CORBA {
    abstract value StreamingPolicy {
        void marshal (in CDROutputStream os, in ValueBase value);
        ValueBase unmarshal (in CDRInputStream is);
    };
};
```

The StreamingPolicy is a abstract value type that is meant to be used by the ORB, not the user. The implementer of a custom value type must provide an implementation of a StreamingPolicy object. Each custom marshaled value type will have its own implementation. The interface is exposed in the CORBA module so that the implementer can use the skeletons generated by the IDL compiler as the basis for the implementation. Hence there is no need for the application to acquire a reference to a Stream.

The implementation requirements of the streaming mechanism require that the implementations must be local since local memory addresses (i.e. the marshal buffers) have to be manipulated.

A StreamingPolicy can be shared by several value types or each type can have its own policy. The ORB run time simply maintains an association between CORBA::RepositoryId and StreamingPolicies.

```
module CORBA {
    interface ORB {
    ...
        StreamingPolicy register_streaming_policy(
            in CORBA::RepositoryId id,
             in StreamingPolicy policy);

        StreamingPolicy lookup_streaming_policy(
            in CORBA::RepositoryId id);

        void unregister_streaming_policy(
            in CORBA::RepositoryId id);
    };
};
```

The register operation replaces an existing registration if a policy has already been registered for the specified type. If a policy was already registered for that type, the old policy is returned, otherwise a null value type (as defined in the language mapping) is returned.

This API is guaranteed to be supported by the ORB but each language mapping is free to add extra facilities to support more automatic ways for a given value type to define and register its marshaling policy.

CDR Streams are defined by the following interfaces:

```
module CORBA {

    typedef sequence<any> AnySeq;
    typedef sequence<boolean> BooleanSeq;
    typedef sequence<char> CharSeq;
    typedef sequence<wchar> WCharSeq;
    typedef sequence<octet> OctetSeq;
    typedef sequence<short> ShortSeq;
    typedef sequence<unsigned short> UShortSeq;
    typedef sequence<long> LongSeq;
    typedef sequence<unsigned long> ULongSeq;
    typedef sequence<long long> LongLongSeq;
    typedef sequence<unsigned long long> ULongLongSeq;
    typedef sequence<float> FloatSeq;
    typedef sequence<double> DoubleSeq;
    typedef sequence<string> StringSeq;
    typedef sequence<wstring> WStringSeq;

    abstract value CDROutputStream {
        void write_any                  (in any value);
        void write_boolean              (in boolean value);
        void write_char                 (in char value);
        void write_wchar                (in wchar value);
        void write_octet                (in octet value);
        void write_short                (in short value);
        void write_ushort               (in unsigned short value);
        void write_long                 (in long value);
        void write_ulong                (in unsigned long value);
        void write_longlong             (in long long value);
        void write_ulonglong            (in unsigned long long value);
        void write_float                (in float value);
        void write_double               (in double value);
        void write_longdouble           (in long double value);
        void write_string               (in string value);
        void write_wstring              (in wstring value);
        void write_objref               (in Object value);
        void write_value                (in ValueBase value);
        void write_TypeCode             (in TypeCode value);
```

```
void write_any_array(       in AnySeq seq,
                            in unsigned long offset,
                            in unsigned long length);
void write_boolean_array( in BooleanSeq seq,
                            in unsigned long offset,
                            in unsigned long length);
void write_char_array(      in CharSeq seq,
                            in unsigned long offset,
                            in unsigned long length);
void write_wchar_array(     in WcharSeq seq,
                            in unsigned long offset,
                            in unsigned long length);
void write_octet_array(     in OctetSeq seq,
                            in unsigned long offset,
                            in unsigned long length);
void write_long_array(      in LongSeq seq,
                            in unsigned long offset,
                            in unsigned long length);
void write_ulong_array(     in ULongSeq seq,
                            in unsigned long offset,
                            in unsigned long length);
void write_ulonglong_array(in ULongLongSeq seq,
                            in unsigned long offset,
                            in unsigned long length);
void write_longlong_array(in LongLongSeq seq,
                            in unsigned long offset,
                            in unsigned long length);
void write_float_array(     in FloatSeq seq,
                            in unsigned long offset,
                            in unsigned long length);
void write_double_array(    in DoubleSeq seq,
                            in unsigned long offset,
                            in unsigned long length);
void write_string_array(    in StringSeq seq,
                            in unsigned long offset,
                            in unsigned long length);
void write_wstring_array(  in WStringSeq seq,
                            in unsigned long offset,
                            in unsigned long length);
};
```

```
abstract value CDRInputStream {
    any read_any();
    boolean  read_boolean();
    char read_char();
    wchar read_wchar();
    octet read_octet();
    short read_short();
    unsigned short read_ushort();
    long read_long();
    unsigned long read_ulong();
    long long long read_long();
    unsigned long long read_ulonglong();
    float read_float();
    double read_double();
    long double read_longdouble();
    string read_string  ();
    wstring read_wstring();
    Object read_objref();
    ValueBase read_value();
    TypeCode read_TypeCode();
```

```
                void read_any_array(        inout AnySeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
                void read_boolean_array(    inout BooleanSeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
                void read_char_array(       inout CharSeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
                void read_wchar_array(      inout WcharSeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
                void read_octet_array(      inout OctetSeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
                void read_long_array(       inout LongSeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
                void read_ulong_array(      inout ULongSeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
                void read_ulonglong_array(inout ULongLongSeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
                void read_longlong_array(inout LongLongSeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
                void read_float_array(      inout FloatSeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
                void read_double_array(     inout DoubleSeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
                void read_string_array(     inout StringSeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
                void read_wstring_array(    inout WStringSeq seq,
                                            in unsigned long offset,
                                            in unsigned long length);
            }
        };
    };
```

Note that the CDR streams are abstract value types. This ensures that their implementation will be local, which is required in order for them to properly flatten and encode nested value types.

The ORB (i.e. the CDR encoding engine) is responsible for actually constructing the value's encoding. The application marshaling code merely calls the above operations. The details of writing the value tag, header information, end tag(s), etc. are specifically not exposed to the application code. In particular the size of the custom data is not written by the application. This guarantees that the custom marshaling (and unmarshaling code) cannot corrupt the other parameters of the call.

If an inconsistency is detected, including not having registered a streaming policy, then the standard system exception **MARSHAL** is raised.

A possible implementation might have the engine determine that a custom marshal parameter is "next". It would then write the value tag and other header information and then return control back to the application defined marshaling policy which would do the marshaling by calling the CDROutputStream operations to write the data as appropriate. (Note the stream takes care off breaking the data into chunks, if necessary.) When control was returned back to the engine, it performs any other cleanup activities to complete the value type, and then proceeds onto the next parameter. How this is actually accomplished is an implementation detail of the ORB.

The CDR Streams shall test for possible shared or null values and place appropriate indirections or null encodings (even when used from the custom streaming policy).

There are no explicit operations for creating the streams. It is assumed that the ORB implicitly acts as a factory. In a sense they are always available.

5.3.8 Access to the Sending Context Run Time

There are two cases where a receiving context might want to access the run time environment of the sending context:

- To attempt the downloading of some missing implementation for the value
- To access some meta information about the version of the value just received

In order the provide that kind of service a call back object interface is defined. It may optionally be supported by the sending context (it can be seen as a service). If such a callback object is supported its IOR may be added to an optional service context in the GIOP header passed from the sending context to the receiving context.

A new constant is added to the IOP Module to define the new service context:

```
module IOP {

    ...
    const ServiceID      SendingContextRunTime = 5;
};
```

A service context tagged with the ServiceID 5 contains an encapsulation of the IOR for a **SendingContext::RunTime** object. Because ORBs are always free to skip a service context they don't understand, this addition does not impact IIOP interoperability.

```
module SendingContext {

    interface RunTime {}; // so that we can provide more
                          // sending context run time
                          // services in the future

    interface CodeBase: RunTime {
        typedef string URL;
        typedef sequence<URL> URLSeq;
        typedef sequence <CORBA::FullValueDescription> ValueDescSeq;

        // Operation to obtain the IR from the sending context
        CORBA::InterfaceRepository get_ir();

        // Operations to obtain a URL to the implementation code
        URL Implementation(in CORBA::RepositoryId x);
        URLSeq implementations(in CORBA::RepositoryIdSeq x);

        // Operations to obtain complete meta information about a Value
        // This is just a performance optimization the IR can provide
        // the same information
        CORBA::FullValueDescription meta(in CORBA::RepositoryId x);
        ValueDescSeq metas(in CORBA::RepositoryIdSeq x);

        // To obtain a type graph for a value type
        // same comment as before the IR can provide similar
        // information
        CORBA::RepositoryIdSeq bases(in CORBA::RepositoryId x);
    };
};
```

Supporting the CodeBase interface for a given ORB run time is an issue of quality of service. The point here is that if the sending context does not support a CodeBase then the receiving context will simply raise an exception with which the sending context had to be prepared to deal. There will always be cases where a receiving context will get a value type and won't be able to interpret it because:

- It can't get a legal implementation for it (even if it knows where it is, possibly due to security and/or resource access issues)
- Its local version is so radically different that it cannot make sense out of the piece of state being provided

These two failure modes will be represented by the CORBA system exception **NO_IMPLEMENT** with identified minor codes, for a missing local value implementation and for incompatible versions (see Section 5.10, "Minor Exception Codes").

Under certain conditions it is possible that when several values of the same CORBA type (same repository id) are sent in either a request or reply, that the reality is that they have distinct implementations. In this case, in addition to the codebase URL that is sent in the service context, each value which has a different codebase may have a codebase URL associated with it. This is encoded by using a different tag to encode the value on the wire.

5.4 IDL Extensions

5.4.1 Syntax

The following new syntax productions are added to IDL:

<value_token> ::= "value"

<value_type_spec_token> ::= "ValueBase"

<safe_token> ::= "safe"

<custom_token> ::= "custom"

<public_token> ::= "public"

<init_token> ::= "init"

<abstract_token> ::= "abstract"

<supports_token> ::= "supports"

<value> ::= (<value_dcl> | <value_abs_dcl> | <value_box_dcl> |
<value_forward_dcl>) ";"

<value_forward_dcl> ::= <value_token> <identifier>

<value_box_dcl> ::= <value_token> <identifier> <type_spec>

<value_abs_dcl> ::= <abstract_token> <value_token> <identifier>
[<value_inheritance_spec>] “{“ <export>* “}”

<value_dcl> ::= <value_header> “{“ < value_body > “}”

<value_header> ::= [<custom_token>] <value_token> <identifier>
[<value_inheritance_spec>]

<value_inheritance_spec> ::= “:” [<safe_token>] <scoped_name>
{ “,” <scoped_name>}*
[<supports_token> <scoped_name> { “,” <scoped_name> }*]

<value_body> ::= <value_element>*

<value_element> ::= <export> | < state_member> | <init_dcl>

<state_member> ::= <public_token> <type_spec> <declarators> “;”
| <type_spec> <declarators> “;”

<init_dcl> ::= <init_token> “(“ [<init_param_decls>] “)” “;”

<init_param_decls> ::= <init_param_decl> { “,” <init_param_decl> }

<init_param_decl> ::= “in” <param_type_spec> <simple_declarator>

The <definition> rule is modified to allow values:

<definition> ::= <type_dcl> “;” | <const_dcl> “;” | <except_dcl> “;”
| <interface> “;” | <value> “;” | <module> “;”

The type specification rules are modified to allow use of the base value as a type specification.

<base_type_spec> ::=... | <value_type_spec>

<value_type_spec> ::= <value_type_spec_token>

5.4.2 New lexical type - Keyword Identifier

In order to allow for the addition of new keywords to IDL in a way that will not invalidate existing IDL, a new lexical class is added to IDL--keyword identifier.

Keyword identifiers obey the rules for identifiers and must be written exactly as shown in the list below. However an identifier that matches a word on the keyword identifier list is treated as a keyword and not as an identifier if it occurs in a context in which it would be legal to interpret it as the reserved word according to the syntax.

Keyword identifiers are used in the grammar by adding a new non-terminal production, conventionally called <yyy_token>, where yyy stands for the name of the token. A <yyy_token> ::= “terminal_string” is also added. Since the production defining <identifier> is not explicitly shown, an alternative containing the new token is added to the definition of <identifier>. The token is then used at higher levels in the grammar.

Note: It is recommended that new keyword identifiers only be added such that the resulting grammar is still easily parsable, e.g. is LALR(1).

This specification adds the following keyword identifiers:

value ValueBase safe custom public init abstract supports

The keyword identifier **value** is a keyword (in the appropriate context) that is used to introduce a value declaration.

The keyword identifier **ValueBase** is the name of the base type for value types and may be used as a type specifier.

The keyword identifier **safe** is a keyword (in the appropriate context) that is used to indicate that it is safe to truncate a derived value instance to a less derived value instance.

The keyword identifier **custom** is a keyword (in the appropriate context) that is used to indicate that a value type should be custom marshaled whenever it is used.

The keyword identifier **public** is a keyword (in the appropriate context) that is used to indicate that a value data member should be public.

The keyword identifier **init** is a keyword (in the appropriate context) that is used to define an initializer for a value type.

The keyword identifier **abstract** is a keyword (in the appropriate context) that is used to introduce an abstract declaration, either value or interface.

The keyword identifier **supports** is a keyword (in the appropriate context) that is used to indicate that a value type supports an interface.

5.4.3 ValueBase Operations

There are some operations that can be done on any value. These operations are analogous to the operations that are defined on Object. Conceptually they are defined on all values, but in reality their actual mapping depends upon on the language mapping.

```
module CORBA {

    value ValueBase {};
};
```

5.5 Interface Repository

Mirroring the syntax a new meta object type **ValueDef** is added to the Interface Repository definition as well as two structs **ValueDescription** and **FullValueDescription.** The interface repository needs to be also modified to support the creation of such entities.

New DefinitionKinds, called **dk_Value** and **dk_ValueBox** are added:

```
enum DefinitionKind {
    dk_none, dk_all,
    dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
    dk_Module, dk_Operation, dk_Typedef,
    dk_Alias, dk_Struct, dk_Union, dk_Enum,
    dk_Primitive, dk_String, dk_Sequence, dk_Array,
    dk_Repository,
    dk_Wstring, dk_Fixed,
    dk_Value, dk_ValueBox
};
```

A new creation operation is added called **create_value**

```
module CORBA{
    ...
    interface Container: IRObject {
    ...
        ValueDef create_value(
            in RepositoryId     id,
            in Identifier       name,
            in VersionSpec      version,
            in boolean          is_custom,
            in boolean          is_abstract,
            in octet            flags,          // must be 0
            in ValueDef         base_value,
            in boolean          has_safe_base,
            in ValueDefSeq      abstract_base_values,
            in InterfaceDefSeq supported_interfaces
        );

        ValueBoxDef create_value_box(
            in IDLType original_type_def
        );

    };
};
```

The interface ValueDef is also added:

```
module CORBA {
...
```

```
typedef short Visibility;
const Visibility PRIVATE = 0;
const Visibility PUBLIC = 1;

struct ValueMember {
    Identifier name;
    TypeCode type;
    IDLType type_def;
    Visibility access;
};


interface ValueMemberDef : Contained {
    readonly    attribute    TypeCode  type;
                attribute    IDLType   type_def;
                attribute    Identifier name;
                attribute    Visibility access;
};

typedef sequence <ValueMember> ValueMemberSeq;


struct Initializer {
    StructMemberSeq members;
};

typedef sequence<Initializer> InitializerSeq;


interface InitializerDef : Contained {
    attribute StructMemberSeq members;
};


interface ValueDef : Container, Contained, IDLType  {
    // read/write interface


    attribute InterfaceDefSeq supported_interfaces;
    attribute ValueDef        base_value;
    attribute ValueDefSeq     abstract_base_values;
    attribute boolean         is_abstract;
    attribute boolean         is_custom;
    attribute octet           flags; // always 0
    attribute boolean         has_safe_base;

    // read interface

    boolean is_a(in RepositoryId value_id);
```

```
struct FullValueDescription {
    Identifier          name;
    RepositoryId        id;
    boolean             is_abstract;
    boolean             is_custom;
    octet               flags;  // always 0
    RepositoryId        defined_in;
    VersionSpec         version;
    OpDescriptionSeq    operations;
    AttrDescriptionSeq  attributes;
    ValueMemberSeq      members;
    InitializerSeq      initializers;
    RepositoryIdSeq     supported_interfaces;
    RepositoryIdSeq     abstract_base_values;
    boolean             has_safe_base;
    RepositoryId        base_value;
    TypeCode            type;
};

FullValueDescription describe_value();

// write interface

ValueMemberDef create_value_member(
    in Identifier name,
    in IDLType type_def,
    in Visibility access
);

InitializerDef create_initializer(
    in StructMemberSeq members
);

AttributeDef create_attribute(
    in RepositoryId     id,
    in Identifier       name,
    in VersionSpec      version,
    in IDLType          type,
    in AttributeMode    mode
);

OperationDef create_operation (
    in RepositoryId         id,
    in Identifier           name,
    in VersionSpec          version,
    in IDLType              result,
    in OperationMode        mode,
    in ParDescriptionSeq    params,
    in ExceptionDefSeq      exceptions,
    in ContextIdSeq         contexts
);
```

```
};

struct ValueDescription {
    Identifier          name;
    RepositoryId        id;
    boolean             is_abstract;
    boolean             is_custom;
    octet               flags;  // always 0
    RepositoryId        defined_in;
    VersionSpec         version;
    RepositoryIdSeq     supported_interfaces;
    RepositoryIdSeq     abstract_base_values;
    boolean             has_safe_base;
    RepositoryId        base_value;
};
```

The interface ValueBoxDef is also added:

```
module CORBA {
...

    interface ValueBoxDef : IDLType {
        attribute IDLType original_type_def;
    };
}
```

5.6 Repository Id and Value Types

5.6.1 CORBA Repository Ids

CORBA RepositoryIds are defined as opaque semantic markers. In the core specification they are just arbitrary strings and their association with an IDL type is purely *conventional* i.e it must be maintained explicitly somewhere (in existing ORB implementation this is done either in an IR or in memory tables).

This core model and has real value in allowing great flexibility in federation and interoperability by:

- remapping ids while keeping IDL source or stub libraries unchanged
- changing IDL source or stub libraries while maintaining ids the same.This core model is very useful as a “lowest possible denominator”.

The CORBA however recognized that “styles” of ID can be defined that may help the management, versioning, interoperability across domains/organization that choose to agree on these “styles”.

If one uses such a style, the association between the id and the IDL type it denotes may not be arbitrary. The ids have some intrinsic information about the type they denote. For instance one of the formats defined is

IDL:/<name>/<name>/<name>:<major version number>.<minor version number>

where it is assumed that the names are the fully scoped names of the IDL type definition.

Systems or organizations that use this style can detect versioning problems simply by comparing ids locally, without having to lookup schema information in an interface repository.

5.6.2 RepositoryId for Value Type

RepositoryIds are used by the ORB to identify value types as they are being unmarshaled. If the Ids are truly arbitrary, then the ORB must be able to lookup the association in a registry somewhere in its environment or use the IR.

This method can be used to locate unmarshaling code, perform version checking, deal with schema evolution etc. ...) with truly opaque ids.

For portability and interoperability reasons, having a defined, "standard" style for RepositoryIds associated with value types would be very useful.

This submission defines such a new "standard" format as follows:

"H:" <scoped_name> ":" <64 bits hash code>

The separator between the components of the scoped name shall be a "**::**".

5.6.2.1 Versioning Issues

We don't recommand the classic id format **"IDL:" <scoped name> ":" <major> "." <minor>** because it is not "foolproof" enough. (It is of course allowable to use this format, since the CORE specification does not mandate any particular form.) The problem with the existing scheme as it is used by most vendors today, is that it is an *optimistic* scheme. sIDL compiler keep on spitting out the default version 1.0 unless somebody place a explicit #pragma that bumps the version up. Because people are sloppy that method generates a lot of interfaces that are not really in synch but uses the same id.

This is not a too severe problem for interfaces. If stubs and skeleton are not actually in synch, even though the RepositoryIds report they are, the worse that can happen is that the result of an invocation is a **`BAD_OPERATION`** exception. The issue with value types it is more problematic because the inconsistency between the stub and skeleton marshaling/unmarshaling code can confuse the marshaling engine and may corrupt memory and/or dump core.

A *pessimistic* scheme whereby any *structural* change in the IDL source results in the versions being incompatible. With such a scheme the ORB can raise a meaningful exception instead of corrupting memory or dumping core.

A hash code is to be computed from the IDL definition. Everytime the IDL definition changes the hash function will (statistically) produce a hash code which is different from the previous one. When an ORB run time receives a value with a different hash

than what is expected, it is free to raise a BAD_PARAM exception. It may also try to resolve the incompatibility by some means. If it is not successful, then it shall raise the BAD_PARAM exception.

A user may still use the #pragma id to force an id to be a specific hash value, although this is not recommended.

Note: The Versioning scheme based on a major and minor version could be made foolproof but it would require more sophisticated IDL compilers than are in widespread use today. The compiler would have to be stateful (i.e operate out of a persistent IR), be integrated with a version management system, and automatically compare newer versions of IDL definition with older version. If it determined that a newer version was structurally compatible it would bump the minor number, otherwise it would bump the major version number.

5.6.3 Hashing Algorithm

The hash code is computed using the signature of a sequence of longs (four octets) written in network byte order (big endian) that reflects the value type definition. The National Institute of Standards and Technology (NIST) Secure Hash Algorithm (SHA-1) is used to compute a signature for the stream. The first two 32-bit quantities are used to form a 64-bit hash.

The sequence of long is equivalent to a fully expanded typecode for the value type that would ignore all field names, it is constructed as follows:

Each IDL type constructor is identified with its tk constant i.e

- interface: tk_objref
- struct: tk_struct
- value: tk_value
- union: tk_union
- sequence: tk_sequence
- ....

Each base type is identified with its tk constant i.e

- short: tk_short
- long:tk_long
- ....

The value type definition is traversed depth first, in the order of the fields declaration starting from its highest base value type the following the derivation chain downward (only single inheritance is involved since only concrete base value type are involved).

For each field:

- If it is a base type, its tk constant is appended to the sequence of long
- If it is an interface type, tk_objref is appended to the sequence of long

- If it is a constructed type or a value type, the tk constant of the constructed type is appended to the sequence. The algorithm is applied recursively to its remaining components
- If the field uses a type which has already been processed, 0xffffffff is appended to the sequence followed by an indirection to the position in the sequence where the type encoding is located. (similar to what is done for recursive typecode)
- If it is a bounded sequence, or an array, the dimensions are appended to the octet sequence after tk_sequence or tk_array
- Typdefs are ignored (i.e resolved to the type being aliased)

The SHA-1 algorithm is executed on the sequence of long in network byte order and produces five 32-bit values sha[0..4].

- The hash value is assembled from the first and second 32-bit values.

  long hash = sha[1] << 32 + sha[0].

5.7 Dynamic Any

The following operations are added to support value types in dynamic anys:

```
module CORBA {
    ...

interface DynAny {
...
void insert_value (in ValueBase value) raises (InvalidValue);
ValueBase get_value() raises (TypeMismatch);

...

interface DynValue : DynAny {
    FieldName current_member_name();
    TCKind current_member_kind();
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value) raises (InvalidSeq);
};
```

Any attempt to set or get a member which has been declared private in the IDL definition of the value contained in the dynamic any raises the exception **NO_PERMISSION.**

The **next()** and **rewind()** operations skip private members. The **seek()** operation will raise the **NO_PERMISSION** exception if an attempt is made to seek to a private member.

Unlike other constructed types, we don't provide a way to construct new dynamic values from scratch in order to avoid the creation of values with an initial state that would violate the value type invariant.

5.8 TypeCodes

Two new TypeCodes are introduced.

5.8.1 New TCKinds

Two new TCKinds are introduced:

```
module CORBA {

    enum TCKind {
        ....
        tk_value,
        tk_value_box,
    };
...
```

5.8.2 New ORB operations

The following new operations are added to the ORB in order to create the two new TypeCodes needed for value types:

```
interface ORB {
...
TypeCode create_value_tc (
    in RepositoryId     id;
    in Identifier    name;
    in boolean is_custom;
    in RepositoryId      base_id; // immediate concrete parent type, "" if none
    in ValueMembersSeq members;    // may be null if the typecode is a
                                   // placeholder  in a recursive typecode definition
);
TypeCode fill_in_recursive_value_tc (
    in TypeCode tc;
    in Repository id;
    in ValueMembersSeq placeholder_replacement;
);
TypeCode create_box_value_tc (
    in RepositoryId      id,
    in Identifier        name,
    in TypeCode orignal_type
    );
```

```
...
}
```

These operations have to handle the potentially recursive nature of value TypeCodes.

It is legal to create a typecode with only an id, name, custom flag with a null member. That TypeCode can be used as a “placeholder” for other typecodes that refer to it. Then it is possible to “go back” to the placeholder typecode and fill in the correct members field using the **fill_in_recursive_value_tc()** operation.

5.9 GIOP/IIOP Extensions and Mapping

GIOP messages and the on-the-wire format are extended to support passing complex object state by value.

A new minor revision number for GIOP and IIOP is required by this submission.

The general philosophy is to add support for transmission of

- the data (state)
- type information (encoded as RepositoryIDs)

The loading (and possible transmission) of code is outside of the scope of the IIOP definition but enough information must be carried to support it (codebase).

The format also makes provision for custom marshaling i.e the fact that a value object is encoded using application-defined code.

The encoding supports all of the features of value types as well as supporting the fragmentation or “chunking” of value types. It does so in a compact way.

At a high level the format can be described as the linearization of a graph. The graph is the depth-first exploration of the transitive closure that starts at the top level value object and follows its “reference to value objects” fields (an ordinary remote reference its just written as an IOR). It is a recursive encoding similar to the one used for TypeCodes. An indirection is used to point to a value that has already been encoded.

The data members are written beginning with the highest possible base type to the most derived type in the order of their declaration.

5.9.1 Partial Type Information and Versioning

The format provides support for partial type information and versioning issues in the receiving context.

The type information is specified by providing a list of repositoryIDs, preceded by a long specifying the number of repositoryIDs.

The first repositoryID, which is always present is the id for the real type of the value. If the value's real type is a derived type, the sending context is responsible for listing the repositoryIDs for all the base types to which it is safe to truncate the real type, going up (the derivation hierarchy) to, and including if appropriate, the formal type.

If the receiving context needs more typing information it can go back to the sending context do a lookup based on that repositoryID to retrieve more typing information (e.g. the type graph).

CORBA RepositoryIDs also contain standard version identification (major and minor version numbers). It is up to the ORB run time to check whether the version of the value being transmitted is compatible with the version expected, and to apply whatever truncation/conversion rules might be appropriate (with the help of a local interface repository or the SendingContext::RunTime service). The RMI model of truncation/conversion across versions can be supported here.

5.9.2 Scope of the Indirections

The special value 0xffffffff introduces an indirection, i.e it directs the decoder to go somewhere else in the marshaling buffer to find what it is looking for. This can be either a URL which has already been encoded, or another value object which is shared in a graph. 0xffffffff is always followed by a long indicating where to go in the buffer.

The encoding used for indirection is the same as that used for recursive TypeCodes with the following exception:

> Indirections are assumed to work across parameters i.e the same value object can be shared across multiple parameters of an IDL call.

5.9.3 Other Encoding Information

A "new" value is coded as a value header followed by a list of value chunks. The header contains a tag, a codebase URL if appropriate, followed by the repositoryID and an octet flag of bits. Because the same repositoryID (and codebase URL) could be repeated many times in a single request when sending a complex graph, they are encoded as a regular string the first time they appear, and use an indirection for later occurrences.

The octet flag contains information which makes operating on value types easier. The flag is reserved for later use and shall be 0.

5.9.4 Fragmentation

It is anticipated that value types may be rather large, particularly when a graph is being transmitted. Hence the encoding supports the breaking up of the serialization into an arbitrary number of "chunks" in order to facilitate incremental processing.

The data may be split into multiple chunks at arbitrary points Any given CDR type representation may be split across multiple chunks. It is the responsibility of the CDR stream to hide the chunking from the marshaling code.

The use of chunking is signaled by the appearance of the appropriate tag at the beginning of the value.

Each chunk is preceded by a positive long which specifies the length of the chunk.

A value is terminated by an end tag which is a non-positive long so it can be differentiated from the start of another chunk. In the case of values which contain other values (e.g. a linked list) the “recursive” value is started without there being an end tag. The absolute value of an end tag (when if finally appears) indicates the number of levels of “recursion” to pop, i.e. how many nested values are actually being terminated. The detailed rules are as follows:

- End tags and value size tags are encoded using non-overlapping ranges so that the unmarshaling code can tell after reading each chunk whether:
  - another chunk follows (positive tag)
  - one or multiple value types are ending at a given point in the stream (negative or null tag)
- The end tag is a non-positive long indicating the number of value types (recursion level) ending at this point in the CDR stream. A recursion depth of zero indicates that more than 2^31 recursion levels are ending, and at least one more end tag follows. The following tag represents the number of recursion levels to be added to the previous end tag. All value types using a chunked encoding will always be terminated by at least one end tag with a value of -1.

  Because data members are encoded in their declaration order, declaring a data member containing value type last is likely to result in more compact encoding on the wire because it maximizes the number of value ending at the same place, the canonical example for that is a linked list.

Truncating a value type in the receiving context may require keeping track of unused buffer chunks (only during unmarshaling) in case further indirection tags point back to values that appear in the unused chunks, which means that they must then be unmarshalled.

Value types that are custom marshaled are encoded as chunks in order to let the ORB run time know exactly where they end in the stream without relying on user defined code.

5.9.5 Notation

The on the wire format is described by a BNF grammar with similar conventions as the one used by the CORBA2.2 specification to define IDL syntax. ***The terminals of the grammar are to be interpreted differently though***: We are describing a protocol format and the terminal although they bear the same name as IDL token represents either:

- constant tags (TCKind)
- the GIOP CDR encoding of the corresponding IDL construct

i.e **short** is a shorthand for the GIOP encoding of the IDL short data type (with all the GIOP alignment rules). Similarly **struct** is a shorthand for the GIOP CDR encoding of a struct, etc.

5.9.6 The Format

```
<value> ::=
    |   <value_tag>  <flag_tag> <rep_ids> <value_chunk>+ <end_tag>+
    |   <value_codebase_tag>  <flag_tag> <codebase_URL>  <rep_ids>
                <value_chunk>+ <end_tag>+
    |   <indirection_tag>  <indirection>
    |   <null_tag>

<rep_ids> ::= long <repository_id>+

<repository_id> ::= ( string | <indirection_tag> <indirection> )

<flag_tag> := (octet) 0

<value_chunk> ::= <chunk_size_tag> <octets>

<null_tag> ::= (long) 0

<value_tag> ::= (long) 1

<value_codebase_tag> ::= (long) 2

<indirection_tag> ::= 0xffffffff

<codebase_URL> ::= ( string | <indirection_tag> <indirection> )

<chunk_size_tag> ::= long //  0 < chunk_size_tag < 2^31-1

<end_tag> := long             // -(2^31-1) < end_tag <= 0

<indirection> ::=  long
```

<octets> := octet | octet <octets>

<state members> ::=
<member>
| <member> <state members>

<state_member> ::=// All legal IDL types should be here
value
| octet
| boolean
| char
| short
| unsigned short
| long
| unsigned long
| float
| wchar
| wstring
| string
| struct
| union
| sequence
| array
| CORBA::Object
| CORBA::ValueBase
| any
| <CDR encapsulation >

<CDR encapsulation> ::= <size> <octets>

5.9.7 New TypeCodes Encoding

The following rows are added to Table 12-2 “TypeCode enum value, parameter list types, and parameters” in Section 12.3.4 to describe the encoding of the two new TypeCodes added by this specification.

TCKind	Integer Value	Type	Parameters
`tk_value`	29	complex	string (repository ID), string (name), ulong (count) {string (member name), TypeCode (member type)}
tk_value_box	30	complex	string (repository ID), string (name), TypeCode

5.10 Minor Exception Codes

This submission specifies several minor exception codes in order to make it considerably easier for clients to diagnose, in a portable fashion, some of the more important new failure modes introduced by value objects.

In CORBA 2.1, the OMG divided the minor exception code space so that ranges of exception codes could be allocated to vendors, using a vendor exception minor code id.

The Interoperability 1.2 RTF is using this methodology to define a space for standard OMG minor exception codes.

The minor exception codes specified in this submission shall be assigned to this space. The low order bits for codes being standardized by this submission are specified in the following table. The high order bits shall be administratively assigned by the OMG once the Interoperability 1.2 report has been accepted.

Table 5-2 Minor exception codes

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
OBJECT_NOT_EXIST	1	Attempt to pass an unactivated (unregistered) value as an object reference
NO_IMPLEMENT	1	Missing local value implementation
	2	Incompatible value implementation version
MARSHAL	1	Unable to locate value factory
BAD_PARAM	1	Unable to register value factory

Java Language Mapping 6

6.1 Introduction

The mapping for a **value** type is similar to the mapping for an IDL **struct**. However, unlike the struct, the mapped value type must implement the standard Java interface **java.io.Serializable**.

6.2 Names

The mapping follows the conventions established for the IDL Java Language mapping. No additional reserved names are required.

6.3 Mapping for Value

6.3.1 Basics for Stateful Values

An IDL **value** is mapped to a public Java class with the same name, and an additional "helper" Java class with the suffix **Helper** appended to the interface name.

The class contains instance variables that correspond to the fields in the state definition in the IDL declaration. The order and name of the Java instance variables are the same as the IDL state fields. Fields which are identified as **public** in the IDL are mapped to **public** instance variables.The rest are **private** instance variables in the mapped Java.

The mapped Java class contains method definitions which correspond to the operations defined on the value type in IDL. These definitions are defined by the implementer of the class in Java. As noted in the Section 5.3.6.4, "Value Method Implementation," on page 5-35, the actual code for the methods must be provided before the (mapped) value type can be used. The way in which the association is made is a an issue left to tool vendors.

The mapped Java class contains a Java class constructor for each init() declaration. The parameters follow the standard mapping rules.

In the absence of JDK support for GIOP serialization, the class must also implement **org.omg.CORBA.Streamable**. The choice of whether to generate direct support for Streamable or to depend upon the JDK is an implementation choice of the code generator. Note that the ORB runtime does not have to use the **Streamable** methods if JDK support is available. See Section 6.7, “Java ORB Portability Interfaces,” on page 6-70 for more information.

The inheritance scheme and specifics of the mapped class depends upon the inheritance and implementation characteristics of the value type and is described in the following subsections.

6.3.1.1 Inheritance from Value

A value type that does not inherit from any other value type or interface implements **java.io.Serializable**.

A value type that inherits from another “pure” value type, i.e. one that does not inherit from an interface (CORBA::Object), extends the Java class to which that value type is mapped.

6.3.1.2 Support of Interface

A value type which supports an IDL interface uses the tie mechanism for its implementation.

The details of the tie mechanism are awaiting approval as part of the IDL/Java Language mapping. All ORB products which support Java, currently provide a (non-portable) tie mechanism.

6.3.1.3 Basics for Abstract Values

Abstract value types follow the same rules as stateful ones, except for as described below.

Abstract value types are mapped to a Java interface. The mapped Java interface also contains abstract methods which correspond to the operations defined on the value type in IDL. It must also extend j**ava.io.Serializable**.

The implementer must, of course, provide a class which implements the Value type.

6.3.1.4 CORBA::ValueBase

CORBA::ValueBase is mapped to j**ava.io.Serializable**.

6.3.2 Helper Class

Value types, like all other user defined IDL types have an additional “helper” Java class.

In addition to the normal methods, the helper class for a value type also contains operations that conceptually belong on **CORBA::ValueBase**. This is to make it possible to use and pass Java classes, which did not originate as IDL definitions such as the Java builtins, as CORBA values without first having to wrap them. Forcing users to define such a wrapping for Java builtins would be awkward to say the least.

The following is the standard helper class generated for a value type named **<*typename*>**:

```
// generated Java helper

public class <typename>Helper {
    public static void
        insert(org.omg.CORBA.Any a, <typename> t) {...}
    public static <typename> extract(Any a) {...}
    public static org.omg.CORBA.TypeCode type() {...}
    public static String id() {...}
    public static <typename> read(
                    org.omg.CORBA.portable.InputStream istream)
        {...}
    public static void write(
                    org.omg.CORBA.portable.OutputStream ostream,
                    <typename> value)
        {...}

    // only for value helpers
    public static
        org.omg.CORBA.ValueDef get_value_def();
}
```

6.3.3 Holder Class

The holder class for the value type is also generated. Its name is the values’s mapped Java classname with **Holder** appended to it as follows:

```
final public class <value_class>Holder
        implements org.omg.CORBA.portable.Streamable {
    public <value_class> value;
    public <value_class>Holder() {}
    public <value_class>Holder(
                        <value_class> initial) {
        value = initial;
    public void _read(org.omg.CORBA.portable InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

6.3.4 Example A

```
// IDL

typedef sequence<unsigned long> WeightSeq;

module Example {
    value WeightedBinaryTree {
       unsigned long weight;
       WeightedBinaryTree left;
       WeightedBinaryTree right;
         init(in long w);
       WeightSeq preOrder();
       WeightSeq postOrder();
    };
};

// generated Java

package Example;

public class WeightedBinaryTree implements java.io.Serializable {

   // instance variables
   private int weight;
   private WeightedBinaryTree left;
   private WeightedBinaryTree right;

   // methods implemented by the interface developer in the file
   // WeightedBinaryTree.impl
   // included (glued) here by the IDL compiler
   public WeightedBinaryTree(long w) {...}
     public int[] preOrder() {...}
   public int[] postOrder() {...}
```

```
}

final public class WeightedBinaryTreeHelper {
// ... other standard helper methods
    public static org.omg.CORBA.ValueDef get_value_def()
        {...}
}

// Holder class
final public class WeightedBinaryTreeHolder
        implements org.omg.CORBA.portable.Streamable {
    public WeightedBinaryTree value;
    public WeightedBinaryTreeHolder() {}
    public WeightedBinaryTreeHolder(WeightedBinaryTreeHolder initial) {...}
    public void _read(org.omg.CORBA.portable.InputStream i) {
    // read state information using the wire format and construct
    // value
    ...
    }
    public void _write(org.omg.CORBA.portable.OutputStream o) {
        // write state information using the wire format
        ...
    }
    public org.omg.CORBA.TypeCode _type() {...}
}
```

6.3.5 *Example B*

```
// IDL

module Example {
    interface Printer {
        typedef sequence<unsigned long> ULongSeq;
        void print(in ULongSeq data);
        };
    value WeightedBinaryTree supports Printer {
        unsigned long weight;
        WeightedBinaryTree left;
        WeightedBinaryTree right;
        ULongSeq preOrder();
        ULongSeq postOrder();
        };
    };
```

```
// generated Java

package Example;

public class WeightedBinaryTree extends Example._PrinterImplBase {
    // instance variables
    private int weight;
    private WeightedBinaryTree left;
    private WeightedBinaryTree right;
    // methods implemented by the interface developer in the file
    // WeightedBinaryTree.impl
    // included here by the IDL compiler
    public int[] preOrder() {...}
    public int[] postOrder() {...}
}

final public class WeightedBinaryTreeHelper {
    // ... other standard helper methods
    public static org.omg.CORBA.ValueDef get_value_def()
        {...}
}
```

6.3.6 Parameter Passing Modes

If the formal parameter in the signature of an operation is **value**, then the actual parameter is passed by value. If the formal parameter type of an operation is an interface, then the actual parameter is passed by reference, i.e. it is widened to the mapped Java interface before being passed.

IDL **value in** parameters are passed as the mapped Java class as defined above.

IDL **value in** and **inout** parameters are passed using the Holder classes defined above.

6.3.6.1 Example

```
// IDL

module Example {

    interface Target {
        WeightedBinaryTree operation(in WeightedBinaryTree inArg,
                                     out WeightedBinaryTree outArg,
                                     inout WeightedBinaryTree inoutArg);
        };

    };
```

```
// generated Java code

package Example;

public interface Target {
    WeightedBinaryTree operation(WeightedBinaryTree inArg,
                                 WeightedBinarTreeHolder outArg,
                                 WeightedBinaryTreeHolder inoutArg);
}
```

6.4 Value Factory and Marshaling

Marshaling Java value instances is straightforward, but unmarshaling value instances is somewhat problematic. In Java there is no a priori relationship between the RepositoryID encoded in the stream and the class name of the actual Java class that implements the value. However, in practice we would expect that there will be a one-to-one relationship between the RepositoryID and the fully scoped name of the value type. However the RepositoryID may have an arbitrary prefix prepended to it, or be completely arbitrary.

The following algorithm will be followed by the ORB:

- If the RepositoryId is a standard IDL repository id (i.e. it starts with “IDL:” then attempt to interpret it as a fully scoped class name by stripping off the “IDL:” header and “:<major>.<minor>” version information trailer, and replacing the “/”s which separate the module names with “.”s.
- If this is not successful, then look up the class name in the RepositoryID to class name map.
- If this is not successful, then raise the **MARSHAL** exception.

The IDL native type **ValueFactory** is mapped in Java to **java.lang.Class**.

A null is returned when **register_value_factory()** is called and no previous RepositoryId was registered.

As usual, it is a tools issue, as to how RepositoryIDs are registered with classes. It is our assumption that in the vast majority of times, the above default implicit registration policies will be adequate. A tool is free to arrange to have the ORB’s **register_value_factory()** explicitly called if it wishes to explicitly register a particular Value Factory with some RepositoryID. For example, this could be done by an “installer” in a server, by pre-loading the ORB runtime, etc..

6.5 Value Box Types

The rules for mapping value box types are specified below.

In addition, helper and holder classes are generated for the value box type in the same way as for other value types.

There are two general cases to consider: value boxes that are mapped to Java primitive types, and those that are mapped to Java classes.

6.5.1 Primitive Types

If the value box IDL type maps to a Java primitive (e.g. **float**, **long**, **char**, **wchar**, **boolean**, **octet**, etc.), then the value box type is mapped to a Java class whose name is the same as the IDL value type. The class has a public data member named **value**, and has the appropriate Java type. The holder and helper class are also generated.

```
// IDL
value <box_name> <primitive_type>;

// generated Java

 public class <box_name> {
      public <mapped_primitive_Java_type> value;
      public <box_name>(<mapped_primitive_Java_type> initial)
           { value = initial; }
}

final public class <box_name>Holder
                implements org.omg.CORBA.portable.Streamable {
      public <mapped_primitive_Java_type> value;
      ...
}

public class <box_name>Helper {
      ...
}
```

6.5.1.1 Primitive Type Example

```
// IDL

value MyLong long;

interface foo {
      void bar_in(in MyLong number);
      void bar_inout(inout MyLong number);
};
```

```
// Generated Java

public class MyLong {
    public int value;
    public MyLong(int initial) {value = initial;}
}

final public class MyLongHolder
            implements org.omg.CORBA.portable.Streamable {
    public MyLong value;
    ...
}

public class MyLongHelper {...}

public interface foo extends org.omg.CORBA.Object {
    void bar_in(MyLong number);
    void bar_inout(MyLongHolder number);
}
```

6.5.2 Complex Types

If the value box IDL type is more complex and maps to a Java class (e.g. **string**, **wstring**, **enum**, **struct**, **sequence**, **array**, **any**, **interface**, etc.), then the value box type is mapped to the Java class that is appropriate for the IDL type. The appropriate holder and helper class are also generated. The details for the mapped class can be found in the Java Language mapping specification and are not repeated here.

6.5.2.1 Complex Type Example

```
// IDL

value MySequence sequence<long>;

interface foo {
    void bar_in(in MySequence seq);
    void bar_inout(inout MySequence seq);
};
```

```
// Generated Java

final public class MySequenceHolder
                implements org.omg.CORBA.portable.Streamable {
    public int[] value;
    ...
}

public class MySequenceHelper {...}

public interface foo extends org.omg.CORBA.Object {
    void bar_in(int[] seq);
    void bar_inout(MySequenceHolder seq);

}
```

6.6 Any

The following methods are added to the Any class:

```
abstract public java.io.Serializable        extract_Value()
                                    throws org.omg.CORBA.BAD_OPERATION;
abstract public void                    insert_Value(java.io.Serializable v);
abstract public void                    insert_Value(
                                                java.io.Serializable v,
                                                org.omg.CORBA.TypeCode t)
                                    throws org.omg.CORBA.MARSHAL;
```

6.7 Java ORB Portability Interfaces

In order to support marshaling of value types additions are made to the input and output stream APIs which are found in the **org.omg.CORBA.portable** package.

Add the following method to **org.omg.CORBA.portable.InputStream**:

```
public abstract java.io.Serializable read_Value();
```

Add the following method to **org.omg.CORBA.portable.OutputStream**:

```
public abstract void write_Value(java.io.Serializable obj);
```

Note: The most efficient implementation of **write_Value()** is dependent upon an enhancement to the JDK that supports the writing of Java serializable objects in GIOP format. It is however possible to implement the marshaling of values without JDK support provided that the mapped Java class implements **org.omg.CORBA.Streamable** (see Section 6.3.1, “Basics for Stateful Values,” on page 6-61). In that case the implementation of **write_Value()** will have to cast its **obj** parameter to be a **Streamable** and then use its write() method for marshaling. The decision of which algorithm to be used is the implementation choice of the ORB runtime.

C++ Language Mapping

7.1 Introduction

The **value** type has features that make its C++ mapping unlike that of any other IDL type. Specifically, all other IDL types comprise either pure state or pure interface, but the **value** type can include both. Because of this, the C++ mapping for the **value** type is necessarily more restrictive in terms of implementation than other parts of the C++ mapping.

7.2 Names

The **value** mapping follows the naming conventions established for the OMG IDL C++ Language Mapping. Each IDL **value** type maps to a C++ class with the same name, a corresponding **_var** type, and for all value types with initializers, an associated **_init** factory type.

7.3 Mapping for Value

An IDL **value** type is mapped to a C++ class with the same name as the IDL **value**. This class is a partially-concrete base class, with virtual accessor and modifier functions corresponding to the state members of the **value** type, and pure virtual functions corresponding to the operations of the **value** type. In order to provide implementations for the pure virtual functions, they must be overridden in a concrete class derived from the base class by the application developer.

Applications are responsible for the creation of **value** instances, and after creation, they deal with **value** instances only through C++ pointers. Unlike object references, which map to C++ **_ptr** types that may be implemented either as actual C++ pointers or as C++ pointer-like objects, "handles" to C++ **value** instances are actual C++ pointers. This helps to distinguish them from object references.

Because **value** types support the sharing of instances within other constructed types (such as graphs), the lifetimes of C++ **value** instances are managed via reference counting. Unlike the semantics of object reference counting, where neither **duplicate** nor **release** actually affect the object implementation, reference counting operations for C++ **value** instances are directly implemented by those instances. Reference counting mix-in classes are provided by ORB implementations for use by **value** implementors.

As for most other types in the C++ mapping, **value** types also have associated C++ **_var** types that automate their reference counting.

All **init** initializers declared for a **value** type are mapped to pure virtual functions on a separate abstract C++ factory class. The class is named by appending "_init" to the name of the **value** type, *e.g.*, type **A** has a factory class named **A_init**.

7.3.1 Value Data Members

The C++ mapping for **value** data members follows the same rules as the C++ mapping for unions except that the accessors and modifers are virtual. Public state members are mapped to public virtual accessor and modifier functions of the C++ **value** base class, and private **value** state members are mapped to protected C++ virtual accessor and modifier functions (so that derived concrete classes may access them). Portable applications, including derived **value** classes, shall not access the actual data members used to store the **value** state, and ORB implementations are free to make such members private. The only restriction on the actual data members is that they be self-managing so that derived classes can correctly implement copying without needing direct access to them.

Like C++ unions, the accessor and modifier functions for **value** state members do not follow the regular C++ parameter passing rules. This is because they allow local program access to the state stored inside the **value** instance. Modifier functions perform the equivalent of a deep-copy of their parameters, and accessors that return a reference or pointer to a state member can be used for read-write access. For example:

```
// IDL
typedef octet Bytes[64];
struct S { ... };
interface A { ... };

value Val {
        public Val t;
        long v;
        public Bytes w;
        public string x;
        S y;
        A z;
};
```

```
// C++
typedef Octet Bytes[64];
typedef Octet Bytes_slice;
...
struct S { ... };

typedef ... A_ptr;

class Val : public virtual ValueBase {
 public:
    ...
    virtual Val* t() const;          // add_ref not called on return value
    virtual void t(Val*);            // remove_ref old Val, add_ref argument

    virtual const Bytes_slice* w() const;
    virtual Bytes_slice* w();
    virtual void w(const Bytes);

    virtual const char* x() const;
    virtual void x(char*);           // free old storage, adopt argument
    virtual void x(const char*);     // free old storage, copy argument
    virtual void x(const String_var&);// free old storage, copy argument

 protected:
    virtual Long v() const;
    virtual void v(Long);

    virtual const S& y() const;      // read-only access to y member
    virtual S& y();                  // read-write access to y member
    virtual void y(const S&);        // deep copy

    virtual A_ptr z() const;         // return value not duplicated
    virtual void z(A_ptr);           // release old objref, duplicate argument
    ...
};
```

State members of anonymous array types require the same supporting C++ typedefs as required for union members of anonymous array types; see the OMG C++ union mapping for more details.

7.3.2 Constructors, Assignment Operators, and Destructors

A C++ **value** class defines a protected default constructor, a protected constructor that takes an initializer for each **value** data member, and a protected virtual destructor. The constructors are protected to allow only derived class instances to invoke them, while the destructor is protected to prevent applications from invoking **delete** on pointers to **value** instances instead of using reference counting operations. The destructor is virtual to provide for proper destruction of derived **value** class instances when their reference counts drop to zero.

The parameters of the constructor that takes an initializer for each member appear in the same order as the data members appear, top to bottom, in the IDL **value** definition, regardless of whether they are public or private. All parameters for the member initializer constructor follow the C++ mapping parameter passing rules for **in** arguments of their respective types.

Portable applications shall not invoke a **value** class copy constructor or default assignment operator. Due to the required **value** reference counting, the default assignment operator for a **value** class shall be private and preferably unimplemented to completely disallow assignment of **value** instances.

7.3.3 Value Operations

Operations declared on a **value** type are mapped to public pure virtual member functions in the corresponding **value** C++ class. None of these pure virtual member functions shall be declared **const** because unlike C++, IDL provides no way to distinguish between operations that change the state of an object and those that merely access that state. This choice, similar to the choice made for the C++ mapping for operations declared in IDL **interface** types, has an impact on parameter passing rules, as described below. The alternative, declaring all pure virtual member functions as **const**, is less desirable because it would not allow member functions inherited from **interface** classes to be invoked on **const** value instances, since all such member functions are already mapped as non-**const**.

The C++ signatures and memory management rules for **value** operations are identical to those described in the OMG IDL C++ mapping for client-side **interface** operations.

A static **_narrow** function is provided by each **value** class to provide a portable way for applications to cast down the C++ inheritance hierarchy. This is especially required after an invocation of the **_copy_value** function (see **"ValueBase and Reference Counting" on page 7-76**). If a null pointer is passed to **_narrow**, it returns a null pointer. Otherwise, if the **value** instance pointed to by the argument is an instance of the **value** class being narrowed to, its reference count is incremented and a pointer to the narrowed-to class type is returned. If however the **value** instance pointed to by the argument is *not* an instance of the **value** class being narrowed to, a null pointer is returned.

7.3.4 Example

For example, consider the following IDL **value** type:

```
// IDL
value Example {
    short op1();
    long op2(in Example x);
        short val1;
        public long val2;
        string val3;
        float val4;
        Example val5;
};
```

The C++ mapping for this **value** type is:

```
// C++
class Example : public virtual CORBA::ValueBase {
  public:
    virtual CORBA::Short op1() = 0;
    virtual CORBA::Long op2(Example*) = 0;

    virtual CORBA::Long val2();
    virtual void val2(CORBA::Long);

    Example* _narrow(CORBA::ValueBase*);

  protected:
    Example();
    Example(CORBA::Short init1, CORBA::Long init2,
            const char* init3, CORBA::Float init4, Example* init5);
    virtual ~Example();

    virtual CORBA::Short val1();
    virtual void val1(CORBA::Short);

    virtual const char* val3();
    virtual void val3(char*);
    virtual void val3(const char*);
    virtual void val3(const String_var&);

    virtual CORBA::Float val4();
    virtual void val4(CORBA::Float);

    virtual Example* val5();
    virtual void val5(Example*);

  private:
    void operator=(const Example&);
};
```

7.3.5 ValueBase and Reference Counting

The C++ mapping for the **ValueBase** IDL type serves as an abstract base class for all C++ **value** classes. **ValueBase** provides several pure virtual reference counting functions inherited by all **value** classes:

```
// C++
namespace CORBA {
    class ValueBase {
     public:
        virtual _add_ref() = 0;
        virtual _remove_ref() = 0;
        virtual ValueBase* _copy_value() = 0;
        virtual CORBA::ULong _refcount_value() = 0;

        static ValueBase* _narrow(ValueBase*);

     protected:
        ValueBase();
        ValueBase(const ValueBase&);
        virtual ~ValueBase();

     private:
        void operator=(const ValueBase&);
    };
}
```

- **_add_ref**, used to increment the reference count of a **value** instance.
- **_remove_ref**, used to decrement the reference count of a **value** instance and **delete** the instance when the reference count drops to zero. Note that the use of **delete** to destroy instances requires that all **value** instances be allocated using **new.**
- **_copy_value**, used to make a deep copy of the **value** instance. The copy has no connections with the original instance and has a lifetime independent of that of the original. Since C++ supports covariant return types, derived classes can override the **_copy_value** function to return a pointer to the derived class rather than **ValueBase***, but since covariant return types are still not commonly supported by commercial C++ compilers, the return value of **_copy_value** can also be **ValueBase***, even for derived classes. A compliant ORB implementation may use either approach. For now, portable applications will not rely on covariant return types and will instead use narrowing[1] to regain the most derived type of a copied **value**.
- **_refcount_value**, which returns the value of the reference count for the **value** instance on which it is invoked.

The names of these operations begin with underscore to keep them from clashing with user-defined operations in derived **value** classes.

1. The C++ dynamic_cast<> operator may also be used to cast down the value hierarchy, but it too is still not available in all C++ compilers and thus its use is still not portable at this time.

ValueBase also provides a protected default constructor, a protected copy constructor, and a protected virtual destructor. The copy constructor is protected to disallow copy construction of derived **value** instances except from within derived class functions, and the destructor is protected to prevent direct deletion of instances of classes derived from **ValueBase**.

With respect to reference counting, **ValueBase** is intended to introduce only interface. Depending upon the inheritance hierarchy of a **value** class, its instances may require different reference counting mechanisms. For example, the reference counting mechanisms needed for a **value** class that supports an **interface** are likely to be different from those needed for a regular concrete **value** class, since the former has object adapter issues to consider. Therefore, **ValueBase** normally serves as a virtual base class multiply inherited into a **value** class. One inheritance path is through the IDL inheritance hierarchy for the **value** type, since all **value** types inherit from **ValueBase**, which provides the reference counting interface, and the other inheritance path is through the reference counting implementation mix-in base class (see Section 7.3.6, “Reference Counting Mix-in Classes,” on page 7-78), which itself also inherits from **ValueBase**.

7.3.5.1 CORBA Module Additions

The C++ mapping also adds two additional reference counting functions to the **CORBA** namespace, as shown below:

```
// C++
namespace CORBA {
    void add_ref(ValueBase* vb)
    {
        if (vb != 0) vb->_add_ref();
    }

    void remove_ref(ValueBase* vb)
    {
        if (vb != 0) vb->_remove_ref();
    }
}
```

These functions are provided for consistency with object reference reference counting functions. They are similar in that unlike the **_add_ref** and **_remove_ref** member functions, they can be called with null **value** pointers. The **CORBA::add_ref** function increments the reference count of the **value** instance pointed to by the function argument if non-null, or does nothing if the argument is a null pointer. The **CORBA::remove_ref** function behaves the same except it decrements the reference count. (The implementations shown above are intended to specify the required semantics of the functions, not to imply that conforming implementations must inline the functions.)

7.3.6 Reference Counting Mix-in Classes

The C++ mapping provides two standard reference counting implementation mix-in base classes:

- **CORBA::DefaultValueRefCountBase**, which can serve as a base class for any application-provided concrete **value** class whose corresponding IDL value type *does not* derive from any IDL **interface**s. For these types of **value** classes, applications are also free to use their own reference-counting implementation mix-ins as long as they fulfill the **ValueBase** reference counting interface.
- **PortableServer::ValueRefCountBase**, which *must* serve as a base class for any application-provided concrete **value** class whose corresponding IDL value type *does* derive from one or more IDL **interface**s, and whose instances will be registered with the POA as servants. If IDL interface inheritance is present, but instances of the application-provided concrete **value** class will not be registered with the POA, the **CORBA::DefaultValueRefCountBase** or an application-specific reference counting implementation mix-in may be used as a base class instead.

Each of these classes shall be fully concrete and shall completely fulfill the **ValueBase** reference counting interface, except that since they provide implementation, not interface, they shall not provide support for narrowing. In addition, each of these classes shall provide a protected default constructor that sets the reference count of the instance to one, a protected virtual destructor, and a protected copy constructor that sets the reference count of the newly-constructed instance to one. Just as with the **ValueBase** base class, the default assignment operator should be private and preferably unimplemented to completely disallow assignment.

Note that it is the application-supplied concrete **value** classes that must derive from these mix-in classes, not the partially-abstract **value** classes generated by the IDL compiler. This is to avoid the need to inherit these mix-ins as virtual bases, or to avoid inheriting multiple copies of the mix-ins(and thus multiple reference counts) if virtual bases are not employed. Also, only the final implementor of a **value** knows whether it will ever be used as a POA servant or not, and thus the implementor must specify the desired reference counting mix-in.

7.3.7 Value Boxes

A value box class essentially provides a reference-counted version of its underlying type. Unlike normal **value** classes, C++ classes for *value boxes* can be concrete since value boxes do not support methods, inheritance, or interfaces. Value box classes differ depending upon their underlying types.

To fulfill the **ValueBase** interface, all value box classes are derived from **CORBA::DefaultValueRefCountBase**.

7.3.7.1 Parameter Passing for Underlying Boxed Type

All value box classes provide **boxed_in**, **boxed_inout**, and **boxed_out** member functions that allow the underlying boxed value to be passed to functions taking parameters of the underlying boxed type. The signatures of these functions depend on the parameter passing modes of the underlying boxed type. The return values of the **boxed_inout** and **boxed_out** functions shall be such that the boxed value is referenced directly, allowing it to be replaced or set to a new value. For example, invoking **boxed_out** on a boxed string allows the actual string owned by the value box to be replaced:

```
// IDL
value StringValue string;
interface X {
    void op(out string s);
};

// C++
StringValue* sval = new StringValue("string val");
X_var x = ...
x->op(sval->boxed_out());     // boxed string is replaced by op() invocation
```

Assume the implementation of **op** is as follows:

```
// C++
void MyXImpl::op(CORBA::String_out s)
{
    s = CORBA::string_dup("new string val");
}
```

The return value of the **boxed_out** function shall be such that the string value boxed in the instance pointed to by **sval** is set to **"new string val"** after **op** returns, with the instance pointed to by **sval** maintaining ownership of the string.

7.3.7.2 Basic Types, Enums, and Object References

For all the integer types, **boolean**, **octet**, **char**, **wchar**, and enumerated types, and for typedefs of all of these, value box classes provide:

- A public default constructor. Note that except for the object reference case, the value of the underlying boxed value will be indeterminate after this constructor runs, *i.e.*, the default constructor does *not* initialize the boxed value to a given value. (This is because the built-in constructors for each of the basic types and enumerations do not initialize instances of their types to particular values, either.) For boxed object references, this constructor sets the underlying boxed object reference to nil.
- A public constructor that takes one argument of the underlying type. This argument is used to initialize the value of the underlying boxed type.
- A public assignment operator that takes one argument of the underlying type. This argument is used to replace the value of the underlying boxed type.

- Public accessor and modifier functions for the boxed value. The accessor and modifier functions are always named **value**. For boxed object references, the return value of the accessor is not a duplicate.
- Explicit conversion functions that allow the boxed value to be passed where its underlying type is called for. These functions are named **boxed_in**, **boxed_inout**, and **boxed_out**, and their return types match the **in**, **inout**, and **out** parameter passing modes, respectively, of the underlying boxed type. Implicit conversions to the underlying type are not provided because values are normally handled by pointer.
- A public copy constructor.
- A public static **_narrow** function.
- A protected destructor.
- A private and preferably unimplemented default assignment operator.

An example value box class for an enumerated type is shown below:

```
// IDL
enum Color { red, green, blue };
value ColorValue Color;

// C++
class ColorValue : public CORBA::DefaultValueRefCountBase {
 public:
    ColorValue();
    ColorValue(Color val);
    ColorValue(const ColorValue& val);

    ColorValue& operator=(Color val);

    Color value() const;          // accessor
    void value(Color val);        // modifier

    // explicit conversion functions for underlying boxed type
    //
    Color boxed_in() const;
    Color& boxed_inout();
    Color& boxed_out();

    static ColorValue* _narrow(CORBA::ValueBase* base);

 protected:
    ~ColorValue();

 private:
    void operator=(const ColorValue& val);
};
```

7.3.7.3 Struct Types

Value box classes for struct types map to classes that provide accessor and modifier functions for each struct member. Specifically, the classes provide:

- A public default constructor. The underlying boxed struct type is initialized as it would be by its own default constructor.
- A public constructor that takes a single argument of type **const T&**, where **T** is the underlying boxed struct type.
- A public assignment operator that takes a single argument of type **const T&**, where **T** is the underlying boxed struct type.
- Public accessor and modifier functions, all named **value**, for the underlying boxed struct type. Two accessors are provided: one a const member function returning **const T&**, and the other a non-const member function returning a **T&**. The modifier function takes a single argument of type **const T&**.
- The **boxed_in**, **boxed_inout**, and **boxed_out** functions that allow access to the boxed value to pass it in signatures expecting the underlying boxed struct type. The return values of these functions correspond to the **in**, **inout**, and **out** parameter passing modes for the underlying boxed struct type, respectively.
- For each struct member, a set of accessor and modifier functions. These functions have the same signatures as accessor and modifier functions for union members.
- A public copy constructor.
- A public static **_narrow** function.
- A protected destructor.
- A private and preferably unimplemented default assignment operator.

As with other value box classes, no implicit conversions to the underlying boxed type are provided since values are normally handled by pointer.

For example:

```
// IDL
struct S {
    string str;
    long len;
};
value BoxedS S;
```

```
// C++
class BoxedS : public CORBA::DefaultValueRefCountBase {
  public:
      BoxedS();
      BoxedS(const S& val);
      BoxedS(const BoxedS& val);

      BoxedS& operator=(const S& val);

      const BoxedS& value() const;
      BoxedS& value();
      void value(const BoxedS& val);

      const BoxedS& boxed_in() const;
      BoxedS& boxed_inout();
      BoxedS*& boxed_out();

      static BoxedS* _narrow(CORBA::ValueBase* base);

      const char* str() const;
      void str(char* val);                          // adopt
      void str(const char* val);                    // copy
      void str(const CORBA::String_var& val);// copy

      CORBA::Long len() const;
      void len(CORBA::Long val);

  protected:
      ~BoxedS();

  private:
      void operator=(const BoxedS& val);
};
```

7.3.7.4 String and WString Types

In order to allow boxed strings to be treated as normal strings where appropriate, value box classes for strings provide largely the same interface as the **String_var** class. The only differences from the interface of the **String_var** class are:

- The value box class interface does not provide the **in**, **inout**, **out**, and **_retn** functions that **String_var** provides. Rather; the value box class provides replacements for these functions called **boxed_in**, **boxed_inout**, and **boxed_out**. They have mostly the same semantics and signatures as their **String_var** counterparts, but their names have been changed to make it clear that they provide access to the underlying string, not to the value box itself.
- There are no overloaded operators for implicit conversion to the underlying string type because values are normally handled by pointer.

In addition to most of the **String_var** interface, value box classes for strings provide:

- Public accessor and modifier functions for the boxed string value. These functions are all named **value**. The single accessor function takes no arguments and returns a **const char***. There are three modifier functions, each taking a single argument. One takes a **char*** argument which is adopted by the value box class, one takes a **const char*** argument which is copied, and one takes a **const String_var&** from which the underlying string value is copied.
- A public default constructor that initializes the underlying string to an empty string.
- Three public constructors that take string arguments. One takes a **char*** argument which is adopted, one takes a **const char*** which is copied, and one takes a **const String_var&** from which the underlying string value is copied. If the **String_var** holds no string, the boxed string value is initialized to the empty string.
- Three public assignment operators: one that takes a parameter of type **char*** which is adopted, one that takes a parameter of type **const char*** which is copied, and one that takes a parameter of type **const String_var&** from which the underlying string value is copied. Each returns a reference to the object being assigned to. If the **String_var** holds no string, the boxed string value is set equal to the empty string.
- A public copy constructor.
- A public static **_narrow** function.
- A protected destructor.
- A private and preferably unimplemented default assignment operator.

An example of a value box class for a string is shown below:

```
// IDL
value StringValue string;
```

```
// C++
class StringValue : public CORBA::DefaultValueRefCountBase {
  public:
      // constructors
      //
      StringValue();
      StringValue(const StringValue& val);
      StringValue(char* str);                          // adopt
      StringValue(const char* str);                    // copy
      StringValue(const String_var& var);              // copy

      // assignment operators
      //
      StringValue& operator=(char* str);               // adopt
      StringValue& operator=(const char* str);         // copy
      StringValue& operator=(const String_var& var);   // copy

      // accessor
      //
      const char* value() const;

      // modifiers
      //
      void value(char* str);                           // adopt
      void value(const char* str);                     // copy
      void value(const String_var& var);               // copy

      // explicit argument passing conversions for
      // the underlying string
      //
      const char* boxed_in() const;
      char*& boxed_inout();
      char*& boxed_out();

      // ...other String_var functions such as overloaded subscript operators, etc....

      static StringValue* _narrow(CORBA::ValueBase* base);

  protected:
      ~StringValue();

  private:
      void operator=(const StringValue& val);
};
```

Note that even though value box classes for strings provide overloaded subscript operators, the fact that values are normally handled by pointer means that they must be dereferenced before the subscript operators can be used.

7.3.7.5 *Union, Sequence, Fixed, and Any Types*

Value boxes for these types map to classes that have exactly the same public interfaces as the underlying boxed types, except that each has:

- In addition to the constructors provided by the class for the underlying boxed type, a public constructor that takes a single argument of type **const T&**, where **T** is the underlying boxed type.
- An assignment operator that takes a single argument of type **const T&**, where **T** is the underlying boxed type.
- Accessor and modifier functions for the underlying boxed value. All such functions are named **value**. There are two accessor functions, one a const member function returning a **const T&**, and the other a non-const member function returning **T&**. The modifier function takes a single argument of type **const T&**.
- The **boxed_in**, **boxed_inout**, and **boxed_out** functions that allow access to the boxed value to pass it in signatures expecting the underlying boxed value type. The return values of these functions correspond to the **in**, **inout**, and **out** parameter passing modes for the underlying boxed type, respectively.
- A protected destructor.
- A private and preferably unimplemented default assignment operator.

As with other value box classes, no implicit conversions to the underlying boxed type are provided since values are normally handled by pointer.

Note that the value box class for sequence types provides overloaded subscript operators (**operator[]**) just as a sequence class does. However, since values are normally handled by pointer, the value instance must be dereferenced before the overloaded subscript operator can be applied to it.

Value box instances for the **any** type can be passed to the overloaded operators for insertion and extraction by invoking the appropriate explicit conversion function:

```
// C++
AnyValueBox* val = ...
val->boxed_inout() <<= something;
if (val->boxed_in() >>= something_else) ...
```

Below is an example value box along with its corresponding C++ class:

```
// IDL
typedef sequence<long> LongSeq;
value LongSeqValue LongSeq;
```

```
// C++
class LongSeqValue : public CORBA::DefaultValueRefCountBase {
 public:
    LongSeqValue();
    LongSeqValue(CORBA::ULong max);
    LongSeqValue(CORBA::ULong max,
                 CORBA::ULong length,
                 CORBA::Long* buf, CORBA::Boolean release = 0);
    LongSeqValue(const LongSeq& init);
    LongSeqValue(const LongSeqValue& val);

    LongSeqValue& operator=(const LongSeq& val);

    const LongSeq& value() const;
    LongSeq& value();
    void value(const LongSeq&);

    const LongSeq& boxed_in() const;
    LongSeq& boxed_inout();
    LongSeq*& boxed_out();

    static LongSeqValue* _narrow(CORBA::ValueBase*);

    CORBA::ULong maximum() const;
    CORBA::ULong length() const;
    void length(CORBA::ULong len);

    CORBA::Long& operator[](CORBA::ULong index);
    CORBA::Long operator[](CORBA::ULong index) const;

 protected:
    ~LongSeqValue();

 private:
    void operator=(const LongSeqValue&);
};
```

7.3.7.6 Array Types

In order to allow boxed arrays to be treated as normal arrays where appropriate, value box classes for arrays provide largely the same interface as the corresponding array **_var** class. The only differences from the interface of the **_var** class are:

- The value box class interface does not provide the **in**, **inout**, **out**, and **_retn** functions that **_var** provides. Rather; the value box class provides replacements for these functions called **boxed_in**, **boxed_inout**, and **boxed_out**. They have mostly the same semantics and signatures as their **_var** counterparts, but their names have been changed to make it clear that they provide access to the underlying array, not to the value box itself.

- There are no overloaded operators for implicit conversion to the underlying array type because values are normally handled by pointer.

In addition to most of the **_var** interface, value box classes for arrays provide:

- Public accessor and modifier functions for the boxed array value. These functions are named **value**. The single accessor function takes no arguments and returns a pointer to array slice. The modifier function takes a single argument of type const array.
- A public default constructor.
- A public constructor that takes a const array argument.
- A public assignment operator that takes a const array argument.
- A public copy constructor.
- A public static **_narrow** function.
- A protected destructor.
- A private and preferably unimplemented default assignment operator.

An example of a value box class for an array is shown below:

```
// IDL
typedef long LongArray[3][4];
value ArrayValue LongArray;
```

```
// C++
typedef CORBA::Long LongArray[3][4];
typedef CORBA::Long LongArray_slice[4];
class ArrayValue : public CORBA::DefaultValueRefCountBase {
 public:
    ArrayValue();
    ArrayValue(const ArrayValue& val);
    ArrayValue(const LongArray val);

    ArrayValue& operator=(const LongArray val);

    const LongArray_slice* value() const;
    LongArray_slice* value();

    void value(const LongArray val);

    // explicit argument passing conversions for
    // the underlying array
    //
    const LongArray_slice* boxed_in() const;
    LongArray_slice* boxed_inout();
    LongArray_slice* boxed_out();

    // ...overloaded subscript operators...

    static ArrayValue* _narrow(CORBA::ValueBase* base);

 protected:
    ~ArrayValue();

 private:
    void operator=(const ArrayValue& val);
};
```

Note that even though value box classes for arrays provide overloaded subscript operators, the fact that values are normally handled by pointer means that they must be dereferenced before the subscript operators can be used.

7.3.8 Abstract Values

Abstract IDL **value** types follow the same C++ mapping rules as concrete IDL **value** types, except that since they have no data members, they do not have member initializer constructors.

7.3.9 Value Inheritance

For an IDL **value** type derived from other **value** types or that supports **interface** types, several C++ inheritance scenarios are possible:

- *Concrete value base classes* are inherited as public non-virtual bases. Concrete **value** types may only be singly inherited in IDL, so they are not multiply inherited in C++ either.
- *Abstract value base classes* are inherited as public virtual base classes, since they may be multiply inherited in IDL.
- *Interface classes* supported by the IDL value type are **not** inherited. Instead, their corresponding POA skeleton classes are derived from.

The reason that **interface** classes are not inherited is that **value** instances, like POA servants, are themselves *not* object references. Providing this inheritance would allow for error-prone code that implicitly widened pointers to **value** instances to C++ object references for the supported interfaces, but without first obtaining valid object references for those **value** instances from the POA. When such an application attempted to use an invalid object reference obtained in this manner, it would encounter errors that could be difficult to track back to the implicit widening of the pointer to **value** to object reference. The C++ language provides no hooks into the implicit pointer-to-class widening mechanism by which an application might guard against this type of error.

By instead deriving **value** classes from the POA skeleton classes for those supported interfaces, **value** class instances can yield object references for the desired supported interfaces by normal POA operations, *e.g.*, via invocation of **_this** or by explicit registration of the value instance as a POA servant.

Avoiding the derivation of **value** classes from **interface** classes also separates the lifetimes of **value** instances from the lifetimes of object reference instances. It would be surprising to an application if a valid object reference that had not yet been released unexpectedly became invalid because another part of the program had decremented the **value** part of the object reference instance to zero. This scenario could be solved by the provision of an appropriate reference counting mix-in class. However, given that such an approach breaks local/remote transparency by having object reference release operations affect the servant, and given the associated problems described in the preceding paragraphs, deriving **value** classes from **interface** classes is best avoided.

7.3.10 Value Factories

Because concrete **value** classes are provided by the application developer, the creation of values is problematic under certain circumstances. These circumstances include:

- *Unmarshaling*. The ORB cannot know *a priori* about all potential concrete value classes supplied by the application, and so the ORB unmarshaling mechanisms do not possess the capability to directly create instances of those classes.
- *Component Libraries*. Portions of an application, such as parts of a framework, may be limited to only manipulating **value** instances while leaving creation of those instances to other parts of the application.

Just as they provide concrete C++ **value** classes, applications must also provide factories for those concrete classes. The base of all value factory classes is the C++ **CORBA::ValueFactoryBase** class:

```
// C++
namespace CORBA {
      class ValueFactoryBase;
      typedef ValueFactoryBase* ValueFactory;
      class ValueFactoryBase {
       public:
           virtual ~ValueFactoryBase();

           static ValueFactory _narrow(ValueFactory vf);

       protected:
           ValueFactoryBase();

       private:
           virtual ValueBase* create_for_unmarshal() = 0;
      };
}
```

The C++ mapping for the IDL **CORBA::ValueFactory** native type is a pointer to a **ValueFactoryBase** class, as shown above. Applications derive concrete factory classes from **ValueFactoryBase**, and register instances of those factory classes with the ORB via the **ORB::register_value_factory** function. If a factory is registered for a given value type and no previous factory was registered for that type, the **register_value_factory** function returns a null pointer.

When unmarshaling value instances, the ORB needs to be able to call up to the application to ask it to create those instances. Value instances are normally created via their type-specific value factories (see Section 7.3.10.1, “Type-Specific Value Factories,” on page 7-91) so as to preserve any invariants they might have for their state. However, creation for unmarshaling is different because the ORB has no knowledge of application-specific factories, and in fact in most cases may not even have the necessary arguments to provide to the type-specific factories.

To allow the ORB to create value instances required during unmarshaling, the **ValueFactoryBase** class provides the **create_for_unmarshal** pure virtual function. The function is private so that only the ORB, through implementation-specific means (*e.g.*, via a friend class), can invoke it. Applications are not expected to invoke the **create_for_unmarshal** function. Derived classes shall override the **create_for_unmarshal** function and shall implement it such that it creates a new value instance and returns a pointer to it. The caller assumes ownership of the returned instance and shall ensure that **_remove_ref** is eventually invoked on it. Since the **create_for_unmarshal** function returns a pointer to **ValueBase**, the caller may use the narrowing functions supplied by value types to downcast the pointer back to a pointer to a derived value type.

Once the ORB has created a value instance via the **create_for_unmarshal** function, it can use the value data member modifier functions to set the state of the new value instance from the unmarshaled data. How the ORB accesses the protected value data member modifiers of the value is implementation-specific and does not affect application portability.

The **_narrow** function on the factory allows the return type of the **ORB::lookup_value_factory** function to be narrowed to a pointer to a type-specific factory (see Section 7.3.10.1). It is important to note that the return value of the factory **_narrow** does *not* become the memory management responsibility of the caller, and thus **delete** should never be called on it. (In this regard it is exactly like the narrowing supplied by the C++ mapping for the IDL exception hierarchy.)

7.3.10.1 Type-Specific Value Factories

All **value** types that have **init** initializer operations declared for them also have type-specific C++ value factory classes generated for them. For a **value** type **A**, the name of the factory class, which is generated at the same scope as the value class, shall be **A_init** Each **init** initializer operation maps to a pure virtual function in the factory class, and each of these initializer functions is named **create**. The initializer parameters are mapped using normal C++ parameter passing rules for **in** parameters. The return type of each **create** function is a pointer to the created **value** type.

For example, consider the following **value** type:

```
// IDL
value V {
    init(boolean b);
    init(char c);
    init(octet o);
    init(short s, string p);
    ...
};
```

First, note that this type presents a minor problem due to the fact that the C++ mapping does not require the **boolean**, **char**, **octet**, and **wchar** types to map to different C++ types. Similar to the support provided by the **Any** type for allowing overloading of insertion and extraction functions, disambiguating types that allow overloaded factory functions for these three types are provided in the **ValueFactoryBase** class:

```
// C++
class ValueFactoryBase {
  public:
      ...
      struct BooleanValue {
            BooleanValue(Boolean b) : value(b) {}
            Boolean value;
      };
      struct CharValue {
            CharValue(Char c) : value(c) {}
            Char value;
      };
      struct OctetValue {
            OctetValue(Octet o) : value(o) {}
            Octet value;
      };
      struct WCharValue {
            WCharValue(WChar wc) : value(wc) {}
            WChar value;
      };
      ...
};
```

These types allow the factory class for the example given above to be generated as follows:

```
// C++
class V_init : public CORBA::ValueFactoryBase {
  public:
      virtual ~V_init();

      virtual V* create(ValueFactoryBase::BooleanValue val) = 0;
      virtual V* create(ValueFactoryBase::CharValue val) = 0;
      virtual V* create(ValueFactoryBase::OctetValue val) = 0;
      virtual V* create(Short s, const char* p) = 0;

      static V_init* _narrow(ValueFactory vf);

  protected:
      V_init();
};
```

Each generated factory class shall have a public virtual destructor, a protected default constructor, and a public **_narrow** function allowing narrowing from the base **ValueFactoryBase** class. Each also supplies a public pure virtual **create** function corresponding to each **init** initializer. Applications derive concrete factory classes from these classes and register them with the ORB. Note that since each generated value factory derives from the base **ValueFactoryBase**, all derived concrete factory classes shall also override the private pure virtual **create_for_unmarshal** function inherited from **ValueFactoryBase**.

Note that the **BooleanValue**, **CharValue**, **OctetValue**, and **WCharValue** types shall be provided by a conforming ORB implementation exactly as shown above (though the constructors need not be inlined), since portable derived factory implementations require access to the values stored in their **value** data members.

For **value** types that have no **init** initializers, there are no type-specific abstract factory classes, but applications must still supply concrete factory classes. These classes are derived directly from **ValueFactoryBase**, need not supply **_narrow** functions[2], and only need to override the **create_for_unmarshal** function.

7.3.10.2 Unmarshaling Issues

When the ORB unmarshals a **value** for a request handled via C++ static stubs or skeletons, it tries to find a factory for the **value** type via the **ORB::lookup_value_factory** operation. If the factory lookup fails, the client application receives a **CORBA::MARSHAL** exception. Thus, applications utilizing static stubs or skeletons must ensure that a value factory is registered for every **value** type it expects to receive via static invocation mechanisms.

Because of their dynamic nature, applications using the DII or DSI are not expected to have compile-time information for all the **value** types they might receive. For these applications, **value** instances are represented as **CORBA::Any**, and so value factories are not required to be registered with the ORB to allow such values to be unmarshaled. However, value factories must be registered with the ORB and available for lookup if the application attempts extraction of the values via the statically-typed **Any** extraction functions. See Section 7.3.16, “Value Interaction With Any,” on page 7-96 for more details.

7.3.11 Custom Marshaling

The C++ mappings for the IDL **CORBA::StreamingPolicy**, **CORBA::CDROutputStream**, and **CORBA::CDRInputStream** types follow normal C++ **value** mapping rules.

7.3.12 Parameter Passing Modes

Value instances are never passed by value or by reference; instead, they are always accessed through C++ pointers. Null **value** instances are indicated by null C++ pointers.

The parameter passing rules described below use an example value type named **ValType**:

- An **in value** is passed as **ValType***. It is explicitly not passed as pointer to const because the callee would be unable to invoke any operations on the received **value**, since all **value** operations are non-const member functions. This is because IDL has

2. Since the factory class hierarchy has virtual functions in it, a C++ dynamic_cast can always be used to traverse the factory inheritance hierarchy, but it is not portable since all C++ compilers do not yet support it.

no similar or corresponding notion of const operations. Since **value** instances are copied when passed as arguments, the callee gets its own copy, so despite the lack of const the callee is unable to affect the caller's copy anyway. The callee shall not invoke **_remove_ref** on the received **value** without first invoking a matching **_add_ref**.

- An **inout value** is passed as **ValType*&**. If the callee wishes to return a different value as an **out** back to the caller, it shall invoke **_remove_ref** on the incoming **value** and then assign a **value** pointer to the **ValType*&** argument. The caller shall eventually invoke **_remove_ref** on the returned **value** regardless of whether the callee returned a different **value** or not.
- As with other IDL types, an **out value** is passed as **ValType_out**. The **ValType_out** type must ensure that a **ValType_var** passed by the caller has its **value** instance reference count decremented before the operation is invoked. For the callee, the **ValType_out** type has the same semantics as a **ValType*&**, and should set the **ValType_out** to point to a **value** instance to be returned to the caller. The caller becomes responsible for eventually invoking **_remove_ref** on the returned pointer to **value** instance.
- A **value** is returned as **ValType***. The callee should return a pointer to **value** instance, and the caller becomes responsible for eventually invoking **_remove_ref** on the returned pointer to **value** instance.

These parameter passing rules follow the C++ mapping rules for other pointer-type or pointer-like IDL types, such as strings and object references.

7.3.13 Memory Management Considerations

Regardless of their mode (**in**, **inout**, **out**, or return), **value** instances that use the standard reference counting mix-ins described in "Reference Counting Mix-in Classes" on page 7-78 shall always be allocated using **new**. The standard **_remove_ref** mechanisms supplied by the standard mix-in classes call **delete this** on a **value** instance when its reference count drops to zero. Applications that use their own reference counting mix-in classes have no restrictions on where they may allocate their **value** instances.

Note that care must be taken by application developers when dealing with cycles in the reference counting of **value** instances, otherwise memory management problems may occur.

7.3.14 Another Example

```
// IDL
value node {
      public long data;
      public node next;
      void print();
      node change(in node inval,
                  inout node ioval,
                  out node outval)
};

// generated C++ node.hh
class node_var {
...
};

class node : public virtual CORBA::ValueBase {
  public:
      virtual CORBA::Long data() const;
      virtual void data(CORBA::Long);

      virtual node* next() const;
      virtual void node(next*);

      node* _narrow(CORBA::ValueBase*);

  protected:
      node();
      node(CORBA::Long data_init, node* next_init);
      virtual ~node();

      virtual void print() = 0;
      virtual node* change(node* _inval, node*& _ioval, node_out _outval) = 0;
};
```

7.3.15 Value Members of Structs

The C++ mapping requires struct members to be self-managing. This results in the need for manager types for both strings and object references. Since **value** types are handled by pointer,similar to the way strings and object references are handled, they too require manager types to represent them when they are used as struct members.

The **value** instance manager types have semantics similar to that of the manager types for object references:

- Any assignment to a managed **value** member causes that member to decrement the reference count of the **value** it is managing, if any.
- A **value** pointer assigned to a managed **value** member is adopted by the member.

- A **value _var** assigned to a managed **value** member results in the reference count of the instance being incremented. The **_var** types and **value** member manager types follow the same rules for widening assignment that those for object references do.
- If the constructed type holding the managed **value** member is assigned to another constructed type (for example, an instance of a struct with a **value** member is assigned to another instance of the same struct), the reference count of the managed **value** instance in the struct on the right-hand side of the assignment is incremented, while the reference count of the managed instance on the left-hand side is decremented. As usual in C++, assignment to self must be guarded against to avoid any mishandling of the reference count.
- When it is destroyed, the managed **value** member decrements the reference count of the managed **value** instance.

The semantics of **value** managers described here provide for sharing of **value** instances across constructed types by default. Each C++ **value** type also provides an explicit copy function that can be used to avoid sharing when desired.

7.3.16 Value Interaction With Any

As for other IDL types, type-safe insertion and extraction of **value** types is supported by **CORBA::Any**.

7.3.16.1 Any Insertion

A **value** instance is inserted into a **CORBA::Any** by the following function at global scope:

```
// C++
void operator<<=(CORBA::Any& any, T* val);
```

Here, **T** represents the **value** type. This function increments the reference count of the instance pointed to by **val**, assuming **val** is not a null pointer. After insertion, when the **CORBA::Any** is destroyed, or when a different instance of a **value** type or any other IDL type is inserted into the **CORBA::Any**, the reference count of the instance pointed to by **val** is decremented.

Adopting insertion is also supported:

```
// C++
void operator<<=(CORBA::Any& any, T** val_p);
```

Assuming that **val_p** is not a null pointer, the underlying **T** instance is adopted by the **CORBA::Any**. The reference count of the **T** instance is not incremented in this case. When the **CORBA::Any** is later destroyed, or when a different instance of a **value** type or any other IDL type is inserted into the **CORBA::Any**, the reference count of the instance referred to by **val_p** is decremented (if ***val_p** is not a null pointer, of course).

7.3.16.2 *Any Extraction*

Instances of **value** types are extracted from **CORBA::Any** instances using the following function available at global scope:

```
// C++
CORBA::Boolean operator>>=(const CORBA::Any& any, T*& val_ref);
```

If an instance of the **value** type represented by type **T** is actually present in the **CORBA::Any** instance as determined by **TypeCode** equality, this function will extract the typed value. Proper extraction of a value instance may require a factory for the **value** type. The extraction function can use the **ORB::lookup_value_factory** function to locate an appropriate factory. If a factory is needed for extraction and the factory lookup fails, the extraction function returns FALSE. Assuming an appropriate factory is found (if needed), the extraction function sets the **val_ref** argument to point to the **T** instance in the **CORBA::Any**, and returns TRUE. The **TypeCode** for **val_ref** is implied by the C++ type of **val_ref**. After extraction, the **value** type instance pointed to by **val_ref** is still owned by the **CORBA::Any** and shall not have its **_remove_ref** function invoked by anything other than the owning **CORBA::Any**. Note that since **value** types may be null, a successful extraction sets the **val_ref** argument to a null pointer if the **CORBA::Any** contains a null **value** pointer.

Otherwise, if the type instance stored in the **CORBA::Any** has a different type than that of **val_ref** as determined by **TypeCode** equality, the value of **val_ref** is unchanged and the extraction function returns FALSE.

Abstract Interfaces 8

8.1 Introduction

In this submission, the decision whether an object is sent by reference or by value is determined by the type specification of the formal parameter in the operation signature. Consider the example

```
interface Example {
  void foo(in MyType mydata);
};
```

The following cases apply:

a) **MyType** is an interface type (not an abstract interface). An object ref (IOR) is always passed. If the implementation object is not registered with the ORB/OA as exportable, the invocation fails.

b) **MyType** is a value. A value (marshalled object state) is always passed, even if the value object inherits from an interface and is registered with the ORB/OA.

Both of these assume that on every invocation of the **foo** operation, either object references are always passed as the **mydata** parameter, or values are always passed. There is no way to sometimes invoke the **foo** operation with a reference actual parameter, and sometimes invoke it with a value actual parameter.

In many cases, this restriction causes no problems. However, there are occasions when more flexibility is needed. See Section 8.10, “Usage Scenarios” for some examples. This submission provides this extra flexibility through a new IDL type called an **abstract interface**. This adds a third case for the above example:

c) **MyType** is an abstract interface. Either an object ref (IOR) or a value is passed, depending on some rules about the runtime type and state of the actual object passed. See Section 8.3, “Semantics of Abstract Interfaces” for details of these rules.

8.2 Syntax for Abstract Interfaces

An optional keyword **abstract** is added to th IDL interface definition syntax. In the above example, to define **MyType** as an abstract interface, we would write an interface definition such as

```
abstract interface MyType {
  void bar(in long avalue);
};
```

This specifies that whenever a formal parameter of type **MyType** appears in an IDL operation definition, either a value or an object reference can be passed as the actual parameter. In both cases, the object that is passed must support the **bar** operation as declared in the abstract interface.

8.3 Semantics of Abstract Interfaces

Abstract interfaces differ from regular IDL interfaces in the following ways:

1. When used in an operation signature, they do not determine whether actual parameters are passed as an object reference or by value. Instead, the type of the actual parameter (regular interface or value) is used to make this determination using the following rules:

   - The actual parameter is passed as an object reference if it is a regular interface type (or a subtype of a regular interface type), and that regular interface type is a subtype of the signature abstract interface type, and the object is already registered with the ORB/OA.
   - The actual parameter is passed as a value if it cannot be passed as an object reference but can be passed as a value. Otherwise, a **BAD_PARAM** exception is raised.

2. The GIOP encoding of an abstract interface type is a boolean (TRUE if it is an IOR, FALSE if it is a value) followed by either the IOR or the value. This allows the demarshaling code to determine whether an object reference or a value was passed.

3. Abstract interfaces do not implicitly inherit from **CORBA::Object.** This is because they can represent either value types or CORBA object references, and value types do not necessarily support the object reference operations defined in section 5.2 of the CORBA 2.1 specification (see Section 5.3.2.3, “Value Type vs. Interfaces”). If an IDL abstract interface type can be successfully narrowed to an object reference type (a regular IDL interface), then the **CORBA::Object** operations can be invoked on the narrowed object reference.

4. Abstract interfaces do not imply copy semantics for value types passed as arguments to their operations. This is because their operations may be either CORBA invocations (for abstract interfaces that represent CORBA object

references) or local programming language calls (for abstract interfaces that represent CORBA value types). See Section 5.3.1.3, “Operations” and Section 5.3.2.4, “Parameter Passing” for details of these differences.

5. Abstract interfaces may only inherit from other abstract interfaces.

6. In other respects, abstract interfaces are identical to regular IDL interfaces.

For example, consider the following operation **m1()** in abstract interface **foo**:

```
abstract interface foo {
    void m1(in AnInterfaceType x, in AnAbstractInterfaceType y,
            in AValueType z);
};
```

x’s are always passed by reference,

z’s are:

- passed as copied values if **foo** refers to an ordinary interface.
- passed as non-copied values if **foo** refers to a value type

y’s are:

- passed as reference if their concrete type is an ordinary interface subtype of **AnAbstractInterfaceType** (registered with the ORB), no matter what **foo**’s concrete type is.
- passed as copied values if their concrete type is value and **foo**’s concrete type is ordinary interface.
- passed as non-copied values if their concrete type is value and **foo**’s concrete type is value.

8.4 Usage Guidelines

Abstract interfaces are intended for situations where it cannot be known at compile time whether an object reference or a value will be passed. In other cases, a regular interface or value type should be used. Abstract interfaces are not intended to replace regular CORBA interfaces in situations where there is no clear need to provide runtime flexibility to pass either an object reference or a value. If reference semantics are intended, regular interfaces should be used.

8.5 IDL Extensions

The <**abstract_token**> is added to the production defining interface as optionally preceding the keyword **interface**.

8.6 Interface Repository Extensions

1. The **FullInterfaceDescription** and **InterfaceDescription** structs in the CORBA module are extended to add a new member:

boolean is_abstract;

2. The **InterfaceDef** interface in the CORBA module is extended to add a new attribute:

attribute boolean is_abstract;

3. The **create_interface** operation in the **Container** interface is extended to add a new formal parameter:

in boolean is_abstract

8.7 Java Language Mapping for Abstract Interfaces

Abstract interfaces are mapped to Java interfaces in the same way as regular IDL interfaces, with the exception that the mapped interfaces do not extend **org.omg.CORBA.Object**.

Helper and holder classes are generated in the usual way.

8.7.1 Java ORB Portability Interfaces

In order to support marshaling of parameters whose formal type is abstract interface, additions are made to the input and output stream APIs which are found in the **org.omg.CORBA.portable** package.

Add the following method to **org.omg.CORBA.portable.InputStream**:

public abstract java.lang.Object read_Abstract();

Add the following method to **org.omg.CORBA.portable.OutputStream**:

public abstract void write_Abstract(java.lang.Object obj);

The **read_Abstract()** and **write_Abstract()** methods are used to marshal and unmarshal abstract interface types. They use **java.lang.Object**, in order to be able to read and write both value types and regular interface types. The **read_Abstract()** method returns either a value type or an **org.omg.CORBA.Object**, depending on the data in the input stream. The **write_Abstract()** method marshals either a value or an IOR to the output stream, depending on its argument's runtime type and whether it is registered with the ORB/OA. See Section 8.3, “Semantics of Abstract Interfaces” for more details.

8.8 C++ Language Mapping for Abstract Interfaces

The C++ mapping for abstract interfaces is almost identical to the mapping for regular interfaces. Rather than defining a complete C++ mapping for abstract interfaces, which would only duplicate much of the specification of the mapping for regular interfaces, only the ways in which the abstract interface mapping differs from the regular interface mapping are described here.

8.8.1 Abstract Interface Base

C++ classes for abstract interfaces are not derived from the **CORBA::Object** C++ class. In IDL, abstract interfaces have no common base. However, to facilitate narrowing from an abstract interface base class down to derived abstract interfaces, derived interfaces, and derived **value** types, all abstract interface base classes that have no other base abstract interfaces derive directly from **CORBA::AbstractBase**. This base class provides the following:

- a protected default constructor
- a protected copy constructor
- a protected pure virtual destructor
- a public static **_duplicate** function
- a public static **_narrow** function
- a public static **_nil** function

The **AbstractBase** class is shown below:

```
// C++
class AbstractBase;
typedef ... AbstractBase_ptr;       // actually either pointer or class

class AbstractBase {
  public:
      static AbstractBase_ptr _duplicate(AbstractBase_ptr);
      static AbstractBase_ptr _narrow(AbstractBase_ptr);
      static AbstractBase_ptr _nil();

  protected:
      AbstractBase();
      AbstractBase(const AbstractBase& val);
      virtual ~AbstractBase() = 0;
};
```

The **_duplicate** function operates polymorphically over both object references and **value** types. If an **AbstractBase_ptr** that actually refers to an object reference is passed to the **_duplicate** function, the object reference is duplicated and returned. Otherwise, the argument refers to a **value** instance, so the **_add_ref** function is called on the **value** and the argument is returned. If the argument is nil, the return value is nil.

The implementation of **AbstractBase::_narrow** merely passes its argument to **_duplicate** and uses the value it returns as its own return value. Strictly speaking, the **_narrow** function is not needed in the **AbstractBase** interface since it is rather pointless to narrow an **AbstractBase** to its own type, but it is required by all conforming implementations in order to make writing C++ templates that deal with abstract interfaces easier (since **AbstractBase** does not present a special case).

As with regular object references, the **_nil** function returns a typed **AbstractBase** nil reference.

Both the **is_nil** and **release** functions in the **CORBA** namespace are overloaded to handle abstract interface references:

```
// C++
namespace CORBA {
    Boolean is_nil(AbstractBase_ptr);
    void release(AbstractBase_ptr);
}
```

These behave the same as their object reference counterparts. Note that **release** is expected to operate polymorphically over both **value** types and object reference types. If its argument is nil, it does nothing. If its argument refers to a **value** instance, it invokes **_remove_ref** on that instance. Otherwise, its argument refers to an object reference, on which it invokes **CORBA::release**.

8.8.2 Client Side Mapping

The client side mapping for abstract interfaces is almost identical to the mapping for object references, except:

- C++ classes for abstract interfaces derive from **CORBA::AbstractBase**, not **CORBA::Object**. Accordingly, their static **_duplicate** and **_narrow** member functions have arguments and return values of type **CORBA::AbstractBase_ptr**, not **CORBA::Object_ptr**.
- Because abstract interface classes can serve as base classes for application-supplied concrete **value** classes, they shall provide a protected default constructor, a protected copy constructor, and a protected destructor (which is virtual by virtue of inheritance from **AbstractBase**).
- The mapping for object reference classes does not specify the type of inheritance used for base object reference classes. However, since abstract interfaces can serve as base classes for application-supplied concrete **value** classes, which themselves can be derived from regular interface classes, abstract interface classes shall always be inherited as public virtual base classes.
- Inserting an abstract interface reference into a **CORBA::Any** operates polymorphically; either the object reference or **value** to which the abstract interface reference refers is what actually gets inserted into the **Any**. This is because there is no **TypeCode** for abstract interfaces. Since abstract interfaces cannot actually be inserted into an **Any**, there is no need for abstract interface extraction operators, either. However, the **CORBA::Any::to_abstract_base** type allows the contents of an **Any** to be extracted as an **AbstractBase** if the entity stored in the **Any** is an object reference type or a **value** type directly or indirectly derived from the **AbstractBase** base class. The **to_abstract_base** type is shown below:

```
// C++
class Any {
 public:
     ...
     struct to_abstract_base {
          to_abstract_base(AbstractBase_ptr& base) : ref(base) {}
          AbstractBase_ptr& ref;
     };
     ...
};
```

Boolean operator>>=(const Any& any, Any::to_abstract_base val);

Other than that, the mapping for abstract interfaces is identical to that for regular interfaces, including the provision of **_var** types, **_out** types, manager types for struct, sequence, and array members, identical memory management, and identical C++ signatures for operations.

Both interfaces that are derived from one or more abstract interfaces and **value** types that support one or more abstract interfaces shall support implicit widening to the **_ptr** type for each abstract interface base class. Specifically, the **T*** for **value** type **T** and the **T_ptr** type for interface type **T** shall support implicit widening to the **Base_ptr** type for abstract interface type **Base**. The only exception to this rule is for **value** types that directly or indirectly support one or more regular interface types; the C++ classes for these **value** types are derived from the POA skeletons for the base interface types, not from the C++ classes for the interface types themselves (as described in Section 7.3.9, “Value Inheritance,” on page 7-88). In these cases, it is the object reference for the **value**, not the pointer to the **value**, that supports widening to the abstract interface base.

8.8.3 Server Side Mapping

The only circumstances under which an IDL compiler should generate C++ code for abstract interfaces for the server side are when either an interface is derived from an abstract interface, or when a **value** type supports an abstract interface indirectly through one or more intermediate regular interface types. Abstract interfaces by themselves cannot be directly implemented or instantiated by portable applications. Because of this, standard C++ skeleton classes for abstract interfaces are not necessary.

The requirements for the C++ server-side mapping for abstract interfaces are therefore quite simple:

- The IDL compiler shall ensure that POA skeletons for interfaces derived from abstract interfaces somehow include pure virtual functions for the IDL operations[1] defined in the base abstract interface(s). These functions can either be generated directly into the POA skeleton class, or can be generated into an implementation-specific base class inherited by the POA skeleton. If the latter approach is used, it should be done in a way that does not require special constructor invocations by

1. This refers only to the operations defined in IDL, not to the C++-specific _duplicate, _narrow, and _nil functions supplied by all abstract interface C++ classes.

application-supplied servant classes (for example, if it were a virtual base class without a default constructor, it would require the most derived servant class to explicitly initialize it in its own constructor member initialization lists).

8.9 Security Considerations

Security considerations for abstract interfaces are similar to those for regular interfaces and values (see Section 5.3.5, “Security Considerations”). This is because an abstract interface formal parameter type allows either a regular interface (IOR) or a value to be passed. Likewise, an operation defined in an abstract interface can be implemented by either a regular interface (with “normal” security considerations) or by a value type (in which case it is a local call, not mediated by the ORB). The security implication of making the choice between these alternatives a runtime determination is that the programmer must ensure that for both alternatives, no security violations can occur. For example, a technique similar to that described in Section 8.10.2, “Passing Values to Trusted Domains” could be used to avoid inadvertently passing values outside a domain of trust.

8.10 Usage Scenarios

8.10.1 Base Types and Mixin Types

The introduction of value types into CORBA will enable the creation of business object frameworks that contain both interface types and value types. In order for these frameworks to support polymorphism with static type checking, it is necessary to be able to specify operations with arguments whose type abstractions can be satisfied by objects implemented as either CORBA interface types (passed by object reference) or value types (passed by value).

For example, in a business application it is extremely common to need to display a list of objects of a given type, with some identifying attribute like account number and a translated text description such as “Savings Account.” A business object framework might define an interface such as ***Describable*** whose methods provide this information, and implement this interface on a wide range of business object types. This allows the method that displays items to take an argument of type ***Describable*** and query it for the necessary information. The ***Describable*** objects passed in to the ***display*** method may be either CORBA interface types (passed in as object references) or CORBA value types (passed in by value).

In this example, ***Describable*** is used as a polymorphic abstract type. No objects of implementation type ***Describable*** exist, but many different implementation types support the ***Describable*** type abstraction. In C++, ***Describable*** would be an abstract base class; in Java, an interface. In statically typed languages, the compiler can check that the actual parameter type passed by callers of ***display*** is a valid subtype of ***Describable*** and therefore supports the methods defined by ***Describable***. The ***display*** method can simply invoke the methods of ***Describable*** on the objects that it receives, without concern for any details of their implementation.

Unfortunately it is not possible to define ***Describable*** as a regular IDL interface. This is because arguments of declared interface type are always passed as object references (see Section 5.3.2.4, “Parameter Passing”) and we also want the ***display*** method to be able to accept value type objects that can only be passed by value. Similarly we cannot define ***Describable*** as an IDL value type because then the ***display*** method would not be able to accept actual parameter objects that only support passing as an object reference. Abstract interfaces are needed to cover such cases.

This usage of abstract interfaces includes both base types and mixin types. A base type is a type that appears at the top of a hierarchy that includes both regular interface and value types. For example, it could be the root type of a business object framework that includes both regular interface types and value types. A mixin type represents a property of some (but not all) types in a hierarchy. For example, it could be inherited by some regular interface types and supported by value types in a framework, but not by other types in the framework.

8.10.1.1 Example

The ***Describable*** abstract interface could be defined and used by the following IDL:

```
abstract interface Describable {
    string get_description();

};

interface Example {
    void display (in Describable anObject);
};

interface Account supports Describable {  // passed by reference
  // add Account methods here
};

value Currency: Describable {  // passed by value
  // add Currency methods here
};
```

If **Describable** were defined as a regular interface instead of an abstract interface, then it would not be possible to pass a **Currency** value to the display method, even though the **Currency** IDL type supports the **Describable** interface. See Section 4.5, “Passing A Value Instance for an Interface Type”, for the rationale.

8.10.2 Passing Values to Trusted Domains

When a server passes an object reference, it can be sure that access control policies will apply to any attempt to access anything through that object reference. When the underlying object is passed as a value, the granularity and level/semantics of access control are different. In the “by value” case, all the data for the object is passed, and method invocations on the passed object are local calls that are not mediated by the

ORB. Whether the server wants to use the (potentially more permissive) pass by value access control or not could depend on the security domain which is receiving the said object or object reference.

Consider the case where the server S has an object O that it is willing to pass only in the form of an object reference Or' to a domain Du that it does not trust, but is willing to pass the object by value Ov to another domain Ot that it trusts.

This flexibility is not possible without abstract interfaces. Signatures would have to be written to either always pass references or always pass values, irrespective of the level of trust of the invocation target domain. However, abstract interfaces provide the necessary flexibility. The formal parameter type **MyType** can be declared as an abstract interface and the method invocation can be coded along the lines of

```
myExample->foo(security_check(myExample,mydata));
```

where the **security_check** function determines the level of trust of **myExample**'s domain and returns an regular interface subtype of **MyType** for untrusted domains and a value subtype of **MyType** for trusted domains. The rules for abstract interfaces will then pass the correct thing in both these cases.

Conformance Issues 9

9.1 Introduction

This chapter specifies the compliance points for this specification

9.2 Compliance

This submission adds no additional compliance points to the CORBA specification. It defines new semantics and IDL extensions that must be implemented in order to claim conformance to CORBA CORE.

Changes to CORBA 2.2

10

This submission proposes extensions to CORBA 2.2 to support passing objects by value. It

- adds the concepts of value types and abstract interfaces to CORBA
- extends IDL
- extends the language mappings to support these IDL extensions
- extends GIOP to further support these extensions in an interoperable fashion

This chapter outlines in detail the probable changes to the CORBA 2.2 specification. See Section 1.4, “Missing Items,” on page 1-9 for further discussion.

10.1 Changes to CORBA 2.2

The following is an extracted set of notes (from a .pdf file) relative to the beta draft of CORBA 2.2 that was made available by the OMG for review purposes. Each note contains the (absolute) page number of the draft, as well as the section number to which it applies.

Changes needed to apply OBV to CORBA 2.2 review

Page 42

Note 1; Label: jeffm; Date: 1/18/98 5:10:06 PM
Section 1.2.4 Object Model Types
Add value type description

Page 43

Note 1; Label: jeffm; Date: 1/18/98 5:10:20 PM
Section 1.2.4 Object Model Types
Add value type at the same level as Object Reference

Note 2; Label: jeffm; Date: 1/18/98 5:10:38 PM
Section 1.2.4 Object Model Types
Replace use of Value in this section with "entity" as appropriate,
so that Value type can be used for the new entity being defined

Page 44

Note 1; Label: jeffm; Date: 1/18/98 5:11:22 PM
Section 1.2.5 Object Model Interfaces
Add new section 1.2.5a
Value types
describe their general characteristics

Page 73

Note 1; Label: jeffm; Date: 1/18/98 5:11:39 PM
Section 3.2.4 Keywords
Add new section 3.2.4a : Keyword Identifiers
(from submission 5.4.2)

Page 76

Note 1; Label: jeffm; Date: 1/18/98 5:12:04 PM
Section 3.4 OMG IDL Grammar
update complete grammar with new productions
(from submission 5.4.1)

Page 80

Note 1; Label: jeffm; Date: 1/18/98 3:06:10 PM
Section 3.5 OMG IDL Specification
add <value> to <definition>

Page 82

Note 1; Label: jeffm; Date: 1/18/98 5:17:17 PM
Section 3.5 OMG IDL Specification
Add new Section 3.5.a on Value Types
Add new Section 3.5.b on Value Boxes
Add new Section 3.5.c on Abstract Value
Add new Section 3.5.d on Abstract Interfaces
These sections include most of the material from submission 5.3, 5.4 and 8.2-8.5

Note 2; Label: jeffm; Date: 1/18/98 5:17:25 PM
Section 3.6 Inheritance

generalize to apply scoping rules to value defs as well as interface defs

Page 88

Note 1; Label: jeffm; Date: 1/18/98 3:09:38 PM
Section 3.8
add <value_type_spec> to <base_type_spec>

Page 101

Note 1; Label: jeffm; Date: 1/18/98 3:09:57 PM
Section 3.13 Names and Scoping
Add value types

Page 105

Note 1; Label: jeffm; Date: 1/18/98 3:10:35 PM
Section 3.15 Exceptions
Add table with new standard minor exception codes from submission 5.9

Page 127

Note 1; Label: jeffm; Date: 1/18/98 5:18:43 PM
Section 4.10
Add new sections 4.10a... to add all the addional ORB and CORBA module functions
(from submission 5.3.6, 5.3.7)

Page 155

Note 1; Label: jeffm; Date: 1/18/98 4:12:40 PM
Section 7.2 Dynamic Any
Add insert_value()

Page 156

Note 1; Label: jeffm; Date: 1/18/98 4:13:03 PM
Section 7.2 Dynamic Any
add get_value() operation

Page 157

Note 1; Label: jeffm; Date: 1/18/98 4:13:12 PM
Section 7.2 Dynamic Any
Add DynValue to IDL

Page 166

Note 1; Label: jeffm; Date: 1/18/98 5:19:18 PM
Section 7.2.9
Add Section 7.2.10 The DynValue Interface
(from submission 5.6)

Page 175

Note 1; Label: jeffm; Date: 1/18/98 4:16:00 PM
Section 8.4.2 IR
Add ValueDef to list

Page 176

Note 1; Label: jeffm; Date: 1/18/98 4:16:41 PM

Section 8.2 IR
add ValueDef to Figure 8-2

Page 177

Note 1; Label: jeffm; Date: 1/18/98 4:17:42 PM
Section 8.5.1 IR
add new DefinitionKinds
(from submission 5.5)

Page 183

Note 1; Label: jeffm; Date: 1/18/98 4:19:07 PM
Section 8.5.3 IR
Add create_value

Page 200

Note 1; Label: jeffm; Date: 1/18/98 5:19:58 PM
Section 8.23 IR
Add new section 8.24 ValueDef
contains bulk of new IDL
(from submission 5.5 and 8.6)

Page 206

Note 1; Label: jeffm; Date: 1/18/98 5:20:09 PM
Section 8.7.1 TypeCodes
Add new TCKind
(from submission 5.7.1)

Page 209

Note 1; Label: jeffm; Date: 1/18/98 5:20:14 PM
Section 8.7.1 Typecodes
Add new info to Table 8-1
(from submission 5.8.7)

Page 212

Note 1; Label: jeffm; Date: 1/18/98 5:20:20 PM
Section 8.7.3 Creeating Typecodes
Add new operations
(from submission 5.7.2)

Page 213

Note 1; Label: jeffm; Date: 1/18/98 4:28:23 PM
Section 8.8 OMG IDL for IR
Update complete IDL with new IDL (from submssion 5.5)

Page 321

Note 1; Label: jeffm; Date: 1/18/98 5:20:44 PM
Section 11.6.7 Interop
Add new service context: SendingContextRunTime
(from submission 5.3.8)

Page 340

Note 1; Label: jeffm; Date: 1/18/98 5:20:55 PM
Section 11.10
Add a new Section 11.11 SendingContextRuntime includes IDL and semantics of new interface RunTime
(from submission 5.3.8)

Page 366

Note 1; Label: jeffm; Date: 1/18/98 4:42:41 PM
Section 13.3 CDR Transfer Syntax
Add new Section 13.3.6 Value Types
specifies encoding rules for values (from submission 5.8-5.8.6)

Page 672

Note 1; Label: jeffm; Date: 1/18/98 5:05:08 PM
Section 19.15 C++ Mapping for Any

Add new Section 19.15a Mapping for Value
(from submission 7.1, 7.2, 7.3)

Page 892

Note 1; Label: jeffm; Date: 1/18/98 4:46:57 PM
Section 23.4 Java Helper
Add helper class for Value also has get_value_def() method

Page 906

Note 1; Label: jeffm; Date: 1/18/98 4:50:53 PM
Section 3.11
Add new Section 3.11a Mapping for Value Types
(from submssion 6.3))
Add new Section 3.11b Mappging for Boxed Values
(from submission6.5).
Add new Section 3.11c Mapping for Abstract Interfaces
(from submission8.7)

Page 910

Note 1; Label: jeffm; Date: 1/18/98 5:21:32 PM
Section 23.13 Java Mapping for Any
Add value support to any
(from submission 6.6)

Page 914

Note 1; Label: jeffm; Date: 1/18/98 4:53:53 PM
Section 23.15
Add new section 23.15a Value Factory and Marshaling
(from submission 6.4)

Page 937

Note 1; Label: jeffm; Date: 1/18/98 5:21:47 PM
Section 23.18.4 Java Streamable APIs
Add support for new types
(from submission 6.7 and 8.7)