orb2 for Eiffel
User’s Guide

orb2 Release 3.2.9
orb2 for Eiffel
User’s Guide

Subject
Instructions for developing applications with the orb2 for Eiffel developer’s kit.

Software Supported
orb2 Release 3.2.9

Revision History
DAIS 3.2 support June 1997
DAIS 3.2 support PDF version January 1998
DAIS 3.2.9 support December 1999
Rebranded September 2001

Document Number
ORB-EIFFEL-01
2AB, Inc. disclaims the implied warranties of merchantability and fitness for a particular purpose and makes no express warranties except as may be stated in its written agreement with and for its customer. In no event is 2AB, Inc. liable to anyone for any indirect, special or consequential damages.

The information and specifications in this document are subject to change without notice. Consult your 2AB, Inc. marketing representative for product or service availability.

U.S. Government Restricted Rights. The Software Program(s) and Documentation furnished under this Agreement were developed at private expense and are provided with Restricted Rights. Any use, duplication, or disclosure by and for any agency of the U.S. Government shall be subject to the Restricted Rights applicable to commercial computer software under FAR Clause 52.227-19 or DFAR Clause 252.277-7013 or any successor thereof.

Copyright © 1996-2001 by 2AB, Inc. All Rights Reserved.

2AB, Inc. hereby grants licensees of Eiffel the right to print this document for internal use.

TRADEMARKS
2AB, Inc., the 2AB, Inc. logo, iLock, orbLock and orb2 are trademarks of 2AB, Inc.

The orb2 Eiffel IDL compiler has been produced using the ISE Eiffel development system

Eiffel is a trademark of NICE (the Non Profit making International Consortium for Eiffel)

Microsoft, MS-DOS, Windows, Visual Basic, Visual C++ and COM are registered trademarks of the Microsoft Corporation.

IIOP, OMG Interface Definition Language (IDL), Unified Modeling Language and UML are trademarks, and CORBA and ORB are registered trademarks of the Object Management Group.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

IBM is a registered trademark of International Business Machines Corporation.

Sun, Sun Microsystems, Solaris and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Apache, Apache Server, Apache Group and Apache HTTP Server Project are trademarks of the Apache Software Foundation.

SAP is a registered trademark of SAP AG.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.
Chapter 3 Application Development in Eiffel: CORBA

3.1 Introduction ........................................... 3-1
3.2 Capsules, Stubs and Skeletons .................................. 3-1
  3.2.1 Stubs and Skeletons .................................. 3-1
  3.2.2 Server Side Class Hierarchy .................................. 3-2
  3.2.3 Client Side Class Hierarchy .................................. 3-4
  3.2.4 Stubs and Skeletons, Servers and Clients: Summary ................. 3-7
3.3 Inheritance ........................................... 3-7
3.4 Capsule Construction .................................... 3-7
  3.4.1 Starting a Capsule .................................... 3-8
  3.4.2 Stopping Capsules .................................... 3-9
  3.4.3 Stopping Windows Capsules ................................ 3-9
3.5 Locating Objects .................................... 3-9
  3.5.1 Binding ........................................... 3-9
  3.5.2 Getting Object References ................................ 3-10
    3.5.2.1 Third Party Service Approach ......................... 3-10
    3.5.2.2 IDL Operation Return Value, out or inout Parameter Approach ............ 3-11
    3.5.2.3 Externalization Approach ............................. 3-11
3.6 Invocation Call Types ................................ 3-12
3.7 Exceptions ........................................... 3-13
3.8 Memory Management .................................. 3-13

Chapter 4 Trader Object Support Service

4.1 Introduction ........................................... 4-1
4.2 Trader Organization .................................. 4-3
  4.2.1 Interface Type ........................................... 4-3
  4.2.2 Context Space ........................................... 4-4
  4.2.3 Properties ........................................... 4-6
    4.2.3.1 Standard Properties ................................ 4-7
  4.2.4 Property Constraint Language .............................. 4-7
4.3 Trader Interfaces .................................. 4-8
  4.3.1 Trading ........................................... 4-8
  4.3.2 Context ........................................... 4-9
  4.3.3 Federation ........................................... 4-10
    4.3.3.1 Context Federation ................................ 4-10
    4.3.3.2 Proxy Federation ................................ 4-12
  4.4 Trader’s Use of the Relocation Architecture .................. 4-14
  4.5 Programmatic Interface to Trading .......................... 4-14
    4.5.1 Secondary Trader ................................ 4-14
    4.5.2 Trader Member Functions .............................. 4-14

Chapter 5 Application Development Using Eiffel: orb2

5.1 Introduction ........................................... 5-1
5.2 Concurrency and Multi threading .......................... 5-1
5.3 Extended Objects and Managed Objects .................. 5-2
Chapter 6  Capsules on Demand

6.1 Introduction .................................................. 6-1
6.2 Capsule Factory .............................................. 6-1

Appendix A  IDL Syntax and Semantics

A.1 Lexical Conventions ............................................. A-1
A.1.1 Tokens ....................................................... A-2
A.1.2 Comments ................................................... A-2
A.1.3 Identifiers ................................................... A-2
A.1.4 Keywords .................................................... A-2
A.1.5 Literals ....................................................... A-3
  A.1.5.1 Integer Literals ........................................ A-3
  A.1.5.2 Character Literals ...................................... A-3
  A.1.5.3 Floating-point Literals ................................ A-4
  A.1.5.4 String Literals ......................................... A-4
A.2 Preprocessing .................................................. A-4
A.3 IDL Grammar ................................................... A-5
A.4 IDL Specification ............................................... A-8
  A.4.1 Module Declaration ....................................... A-9
  A.4.2 Interface Declaration .................................... A-9
    A.4.2.1 Interface Header .................................... A-9
    A.4.2.2 Inheritance Specification ............................ A-9
    A.4.2.3 Interface Body ....................................... A-9
    A.4.2.4 Forward Declaration ................................ A-10
A.5 Inheritance .................................................... A-10
A.6 Constant Declaration .......................................... A-11
A.6.1 Syntax ...................................................... A-11
A.6.2 Semantics .................................................. A-12
A.7 Type Declaration ............................................... A-13
  A.7.1 Basic Types ............................................... A-14
    A.7.1.1 Integer Types ......................................... A-14
    A.7.1.2 Floating-point Types ................................ A-14
    A.7.1.3 Char Type ............................................. A-14
    A.7.1.4 Boolean Type ......................................... A-15
    A.7.1.5 Octet Type ........................................... A-15
    A.7.1.6 Any Type ............................................. A-15
    A.7.1.7 Constructed Types ................................... A-15
    A.7.1.8 Structures ............................................ A-15
    A.7.1.9 Discriminated Unions ................................. A-16
    A.7.1.10 Enumerations ....................................... A-17
  A.7.2 Template Types ........................................... A-17

5.4 Capsules and Managed Objects ................................ 5-2
5.5 Object Relocation ............................................. 5-3
  5.5.1 Overview .................................................. 5-3
  5.5.2 The Trader’s Use of the Relocator Interface ............ 5-4
  5.5.3 Relocation Service Example .............................. 5-4
  5.5.4 Resilience Improvements: Example ....................... 5-5

5.5.2 The Trader

5.5.1 Overview

5.5.3 Relocation Service Example

5.5.4 Resilience Improvements: Example
Appendix B  Property Constraint Language

Index
Figures

Figure 1.1   An orb2 Object ...............................................................1-2
Figure 1.2   Interaction through the ORB........................................1-3
Figure 3.1   A Capsule .................................................................3-2
Figure 3.2   Target/Client ..............................................................3-2
Figure 3.3   Narrowing .................................................................3-6
Figure 3.4   Proxy Access ..............................................................3-6
Figure 3.5   Trading ..................................................................3-11
Figure 3.6   Invocation Call Types ..............................................3-12
Figure 3.7   Trader and its Objects ..............................................4-2
Figure 3.8   Interface Conformance ..........................................4-3
Figure 3.9   Context or Name Spaces ......................................4-4
Figure 3.10  Offers in the Context Space ................................4-8
Figure 3.11  Default Local-Master Federation .........................4-11
Figure 3.12  Federation Example .............................................4-13
Figure 3.13  Object Reference ..................................................5-3
Figure 3.14  Architectural Model ..............................................5-5
Figure 3.15  Object References (1) ............................................5-6
Figure 3.16  Object References (2) ............................................5-6
Figure 3.17  Object References (3) ............................................5-6
Figure 3.18  Object References (4) ............................................5-7
Figure 3.19  Node, Trader, Factory and Domain Relationship ..........5-7
Figure 3.20  Figures
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>Basic IDL Data Types</td>
<td>2-7</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Trader Objects</td>
<td>4-1</td>
</tr>
<tr>
<td>Table A.1</td>
<td>Symbols Used in IDL Syntax Notation</td>
<td>A-1</td>
</tr>
<tr>
<td>Table A.2</td>
<td>Reserved Keywords</td>
<td>A-2</td>
</tr>
<tr>
<td>Table A.3</td>
<td>Escape Sequences for Non-Graphic Characters</td>
<td>A-3</td>
</tr>
<tr>
<td>Table A.4</td>
<td>Case Labels and Discriminator Types</td>
<td>A-16</td>
</tr>
</tbody>
</table>
Preface

This document and the orb2 for Eiffel Reference documentation facilitate the use of Eiffel in orb2 application environments. This manual describes the object-oriented environment, the Interface Definition Language (IDL), the techniques for application development and the various services and nodes that comprise the orb2 environment. Code examples are also provided to illustrate good practice in the use of the orb2 facilities.

Should you have any problems with the information provided here please do not hesitate to contact 2AB, Inc.

Related Publications

This document describes how to use the Eiffel language binding for orb2. Refer to the documentation that is distributed with the orb2 ORB or orb2 J² for information about using the core ORB services and developing orb2 applications.

Purpose of this Edition

This edition of the orb2 for Eiffel User’s Guide supports orb2 Release 3.2.9.

The guide provides information on orb2 concepts and facilities and contains instructions for developing orb2 applications in Eiffel. It is not a tutorial or guide to the Eiffel language itself.

Target Audience

This document is aimed at designers and programmers who are developing applications in the Eiffel programming language, using the Eiffel facilities of the orb2 product.
Chapter 1

Object-oriented development in distributed environments

1.1 Introduction

In a distributed application developed in a pure object-oriented language like Eiffel, most of the objects are not accessed remotely. However, some, typically those representing business model objects, will offer services to remote clients. If these services are offered in a heterogeneous environment, then CORBA is the appropriate choice for the interface specification. The object model of the CORBA distribution technology must be applied to these coarse grain object services.

1.2 The orb2 Object Model

The CORBA object model allows object-oriented applications to be deployed in a heterogeneous environment, with clients and servers in different languages and on different platform architectures.

The CORBA object model shares most of the attributes common to all object systems, such as inheritance and encapsulation. However, there are subtle differences in detail from the models of the different languages that might be used to deploy the applications.

This chapter describes the object model of CORBA and orb2, which is similar in concept to the Eiffel object model. The information in this chapter applies to the coarse grain objects which offer and use remote services.

1.2.1 Object Semantics

- Clients
  An object system provides services to clients. A client of a service is any entity capable of requesting the service.

- Objects
  An orb2 object can be represented pictorially as shown in Figure 1.1.
In general terms, an object
- Provides a service based on the set of operations defined by its interface
- Hides all details of its internal construction from other parts of the system with which it interacts
- Interacts with the rest of the world (typically other objects) by means of the invocation of operations on interfaces

The benefits of this are
- Decomposition of a system into simpler components (objects) that are only dependent on each others’ interfaces, not implementation details
- Creation of libraries of potentially useful code (resulting in the re-use of code)
- Simplification of the programming environment due to stylized object interaction, so that much of the support code common to all objects in a system is included automatically
- Independent, modular development, conforming to predefined interfaces

1.2.1.1 Requests
Clients obtain services by issuing requests. A request is an event, that is, something that occurs at a particular time. The information associated with a request consists of an operation, a target object, zero or more parameters, and an optional request context. orb2 does not support the propagation of contexts.

A value is anything that can be a legitimate parameter in a request. A value is an instance of an OMG IDL datatype and may identify an object, in which case it is called an object reference.

An object reference is a special data type that reliably denotes a specific object. An object reference identifies the same object each time the reference is used in a request.

A request causes a service operation to be performed on behalf of the client. One outcome of performing a service operation is returning to the client the results, if any, defined for the request.

If an abnormal condition occurs during the performance of a request, an exception is returned. The exception may contain return parameters specific to that exception.

The request parameters are identified by position. A parameter may be an input to the object, an output from the object or both. A request may also return a single result value, as well as any output parameters.
1.2.1.2 Interfaces

An interface is a description of a set of possible operations that a client may request of an object. An object satisfies an interface if it can be specified as the target object in each potential request described by the interface.

Interfaces are specified in the CORBA Interface Definition Language (IDL). Interface inheritance provides the composition mechanism for permitting an object to support multiple interfaces.

1.2.1.3 Operations

An operation is

- An identifiable entity that denotes a service that can be requested.
- Identified by an operation identifier. An operation is not a value.

An operation has a signature that describes the legitimate values of request parameters and returned results. A signature consists of either

- A specification of
  - The parameters required in requests for that operation
  - The result of the operation
  - The exceptions that may be raised by a request for the operation and the types of the parameters accompanying them
  - Additional contextual information that may affect the request
- An indication of the execution semantics the client should expect from a request for the operation

1.3 Object Interaction

Within individual interactions, objects take on the role of service providers to clients which are the service requesters. See Figure 1.2.

orb2 objects may themselves make requests on other objects as part of the implementation to satisfy a particular operation.
An interface is "a well-defined encapsulating boundary which isolates the implementation from the outside world." Interfaces are defined in terms of the operations they support. Clients, having access to an interface, may make requests of the target object by invoking an operation. Each operation may take arguments that are passed to the object as part of the invocation, the target then processes the request and, possibly, returns results.

A model of this type will work only if both parties have a mutual understanding of the interface, in terms of what the operations are and the data types of their arguments and results. This information constitutes the interface type and is described by an interface definition expressed in IDL.

IDL supports inheritance construction, allowing new interface types to be derived from existing interface types which inherit all their operations and definitions. Objects supporting a derived interface can be substituted for objects supporting the base types from which the inheritance was constructed.

Several objects may offer interfaces of the same type within a distributed environment but client invocations are always directed to one specific object instance. This is achieved by means of an object reference that contains sufficient information for the object to be correctly identified. Several clients may hold the same object reference and may even invoke operations on the same interface concurrently. Object references are treated like other data types; in particular, they may be declared as parameters within operations.

Distributed systems are highly dynamic and it is often impossible to know the exact location of objects within a system until run time. orb2 solves this problem by providing a number of object location strategies, the primary mechanism being trading. See Chapter 4 for information about the Trader.

Once an object is located, operations can be invoked on it. Operation invocation results in execution of the object code (object methods within the implementation) which may manipulate encapsulated data and private state variables within the object.

1.4 orb2

orb2 is a powerful system for developing and deploying distributed, object-oriented applications that conform to the CORBA object model. This is achieved through a set of software tools, runtime libraries and object support services. orb2 support for the Eiffel development language conforms to the mapping widely accepted by the Eiffel suppliers.

orb2 operates across heterogeneous distributed networks composed of collections of computer nodes. Nodes can have different architectures and can be from different vendors connected together by some well-defined network standards, for example, TCP/IP or OSI. The client and server applications can be written in a variety of languages including C, C++, Eiffel, Java, Smalltalk, Ada, and can use a number of different ORBs interworking over the IIOP protocol.

For distributed object interaction to work across dissimilar nodes, both client and target need a common understanding of two things

- Data types. Data types may have different representation across dissimilar hosts. An integer, for example, may well have a different internal hardware representation from vendor to vendor. Some means is needed to express the data types in a common way across all participating nodes.
- The nature of the interaction. The interaction needs to be expressed in terms of the data types involved. For example, an operation might be expressed in terms of "Send 2 Float Numbers and a String and return an Object Reference". This together with the "name" of the operation provides the operation signature.

The data types and the collection of related operations that can be performed with them, are termed an interface definition. The language used to unambiguously describe an interface is IDL. Each interface definition is given a name which is called the interface type or just the type. The text file used to store the interface definition written in IDL can now be shared by
both client and target. The sharing is achieved through intermediate Eiffel code generated as a result of compiling the IDL file. Such intermediate code representations are known as \textit{stubs} by the client and as \textit{skeletons} by the target and provide the basis for \textit{invocation transparency}.

The model so far described has a further characteristic: the model has described a service that an arbitrary object can offer on a particular node; multiple clients across multiple locations must be able to contact such objects over the network. Some information about where the object resides could be built into the IDL. However, this would prevent the object ever being moved to other nodes. What is needed is some way of encapsulating the object type and location into a data structure that can be created whenever an object instance of the specific type is active on a particular node. This data structure is known as an object reference, providing the basis for \textit{location transparency}.

The object reference is not directly visible to the Eiffel programmer. The programmer sees a deferred object which represents the interface and which has the object reference embedded in it. These interface objects can be freely passed around the network, and the object reference itself is transparently taken with it. Any user of the interface object is routed to the server currently supporting that interface instance.

Invocation transparency combined with location transparency provide the fundamental techniques that allow for distributed, object based, application development. Client access to objects remains syntactically and semantically identical, regardless of the location of the components.
Chapter 2

Interface specification in IDL

Chapter 1 explained the reason for creating an interface definition. This chapter describes IDL, the OMG-defined language in which object interfaces are specified. Appendix A contains the full IDL syntax.

IDL is programming language independent. It is lexically similar to C++ and is used to describe interfaces supported by objects. It is worth noting that the data types which can be expressed in IDL are not as flexible as those available in most object-oriented languages, Eiffel included. Consequently, the design of the Eiffel objects that may become interface objects is subject to some restrictions. An IDL specification can contain a single interface definition or multiple interface definitions by grouping the definitions into modules. Each individual interface definition defines the interface in terms of the operations that can be invoked by clients, the parameters to those operations and any exceptions that might arise. There is no mechanism in IDL for expressing the other aspects of the programming by contract conventions of Eiffel: there are no pre, post, or invariant assertions.

Example IDL

```idl
module Example
{
  // Global type definition common to all interfaces
  // in this module
  typedef long Counter;
  // Declare an interface of type 'X' from which
  // 'X' objects can be constructed
  interface X
  {
    // declare an operation 'op1' passing one argument
    // to the target
    void op1( in Counter arg1 );
    // declare an operation 'op2' passing one argument
    // to the target and receiving a short back
    // together with an object reference to an instance
    // of an 'X' object
    X    op2( in Counter arg1, out short arg2 );
  };
  // Declare a second interface of type 'Y'
  interface Y
  {
    typedef short Index;
    typedef Y     ObjGen;
    long op1( out Index res1 );
    // 'Y' objects support an 'op2' operation being
    // passed an object reference to an 'X' object
    // and getting back a reference to a 'Y' object
    ObjGen    op2( in X obj );
  };
}
```

Interface definitions are created as normal text files with an .idl suffix; the usual practice is to name the file after the interface or module name that it defines. However, there may be circumstances when an abbreviated name must be used, for example, DOS files can only have
eight characters. Current convention on case sensitive architectures is that the first character of the file name is uppercase; *MyDef.idl* is a valid and conventional type of name for an IDL definition file.

By default, IDL is case sensitive, it obeys the same lexical rules as C++, with additional keywords to support distributed applications. In this document, all optional syntax use square brackets [ ] to distinguish optional syntax from mandatory. Where a character is part of the syntax, it is shown in double quotes, for example "" or "".

## 2.1 IDL Specification

An IDL specification consists of declarations comprising one or more of the following

- module: A module is an IDL name scope. It can contain declarations of other modules, interfaces, data types, constants and exceptions. The definition is delimited by {} and terminated with ;.
- interface: An interface names a particular CORBA object type. It is also a name scope, and can contain declarations of operations, attributes, data types, exceptions and constants. The definition is delimited by {} and terminated with ;.
- operation: Within an interface declaration, an operation declaration defines its name, its parameters and returns, and the names of any previously declared exceptions which it might raise.
- datatype: There is a small number of primitive types. Named type declarations define further types built from the primitive types or previously defined named types.
- constant: Constant declarations define named values derived from expressions of the primitive types.
- exception: An exception declaration defines its name and any data which may be included when the exception is raised.

## 2.1.1 Names and Scoping

This section describes naming and name scoping conventions. For the full IDL name scoping rules refer to Appendix A.

### 2.1.1.1 Naming Conventions

orb2 uses these naming conventions to avoid clashes in the construction of the scoped names detailed in Section 2.1.1.2.

- **IDL file names**: IDL must be contained in files with the extension *idl*. For some languages the IDL compiler generates additional files based on the IDL file name with an additional two characters. Beware of any operating system restrictions on file name length. The orb2 compiler invokes the C preprocessor so directives such as *#include* are supported. orb2 also supports an *#import* directive for referring to previous declarations without regenerating the source for them.

  - **Module and interface names**: These are typically constructed as a concatenation of words, where each word has an initial capital letter followed by lowercase letters or digits.

    Interface OutputMedia

  - **Operation names**: These are constructed as a sequence of words, where each word is a sequence of lowercase letters or digits, and words are separated by single underscore characters.

    ```
    void debit_transaction ();
    ```
• **User defined data types and exceptions**: User declarations for types and exceptions are constructed in the same way as interface and module names, where each word has an initial capital letter followed by lowercase letters or digits.

```cpp
typedef string OperatorCmd;
```

• **Parameter names and members**: Parameter names defined in operations, members in structures, unions and enumerations are constructed in the same way as operation names, where each word is a sequence of lowercase letters or digits and each word is separated by a single underscore.

```cpp
void barcode_input(in Barcode item);
```

### 2.1.1.2 Name Scoping

An IDL specification defines a name space; every IDL definition in the specification has a *scoped name*. The purpose of this scoping is to allow application code to reference types, constants, exceptions and operations unambiguously, using the global name.

This name scoping approach is quite different from the Eiffel conventions, where all types (class names) have a global scope. The rules for the Eiffel mapping are described in the mapping section of the *orb2 for Eiffel Reference*.

These differences between IDL and Eiffel name scoping conventions can result in name clashes if IDL declares identifiers with the same name in different scopes.

To overcome these clashes orb2 provides a name collision avoidance mechanism. This allows any Eiffel generated from the IDL to be given alternative names from that defined in the IDL. This is described in Section 2.2.

### 2.1.2 Module Definition

By convention, related type definitions are grouped together using the module construct. The module definition takes this form

```cpp
module ModuleName "{" ModuleDefinition "}; ";
```

where:

- **module** The IDL keyword for a module declaration
- **ModuleName** The identifier naming this module. This name scopes all identifiers declared within the module definition
- **ModuleDefinition** Consists of the interface, type, constant and exception declarations which comprise this module

The module construct is used to scope identifiers declared within the IDL specification. Scoped names provide the means by which types are referenced by clients and object implementations. Names and scoping are described in Section 2.1.1 and Appendix A.

### 2.1.3 Interface Definition

An interface definition takes the form

```cpp
interface InterfaceName ["InheritedInterfaces] "{" InterfaceDetails ");;
```

where:

- **interface** The IDL keyword for an interface declaration
- **InterfaceName** The identifier for this interface
- **InheritedInterfaces** Optional; if specified it is the identifier of one or more previously defined interfaces. These interfaces can be in the same IDL file as **InterfaceName** or another IDL file. If they are in another file that file should be referenced using the #import directive. This parameter allows **InterfaceName** to use (inherit) all the declarations (for example operations, data types) in **InheritedInterfaces**. If more than one interface is specified they are comma-separated. See the example below, and Appendix A for details of inheritance
- **InterfaceDetails** This can contain any of these declarations to define the interface operation declarations type declarations constant declarations exception declarations attribute declarations

**Note** **InterfaceDetails** can be empty; it does not have to contain any declarations.

Example IDL

```idl
module Transaction
{
  interface Cash
  {
    // declarations to define any required data types
    // and two operations named deposit and withdraw
  };
  interface Check : Cash
  {
    // declaration to define an operation named cleared
  };
};
```

This IDL specification defines a module named **Transaction** that contains two interface definitions **Cash** and **Check**. The interface **Cash** declares the data types that it requires, as well as two operations, deposit and withdraw. The interface **Check** specifies **Cash** as an inherited interface and an operation named cleared. This means that interface **Check** now provides operations deposit, withdraw and cleared and can also reference any data types declared in interface **Cash**.

### 2.1.3.1 Forward Declaration

A forward declaration declares the name of an interface without defining it. This allows the definition of interfaces that refer to each other. The form of a forward declaration is the keyword **interface** followed by an identifier giving the interface name.

For example

```idl
interface Calculator;
```

The actual interface definition must follow later in the same IDL specification.
2.1.4 Interface Operations

The operations supported by the interface are specified by making operation declarations. The full form of an operation declaration is:

\[\text{oneway} \text{ OpType OperationName "([parameters])" Opexceptions Opcontext} ;\]

The options \texttt{oneway}, \texttt{Opexceptions} and \texttt{Opcontext}, are described in Section 2.1.4.1, Section 2.1.4.2 and Section 2.1.4.3.

\texttt{OpType} The type of the operation’s return result. If the operation does not return a result, \texttt{OpType} must be specified as void. See Section 2.1.5 for IDL data types

\texttt{OperationName} The identifier naming this operation in the scope of the interface in which it is defined

\texttt{parameters} Empty, or a list of one or more parameter declarations for the operation. Parameter declarations are a comma separated list:

\texttt{param1, param2, ..., paramN}

and each parameter declaration takes one of the following forms

\begin{align*}
\text{in} & \quad \text{Datatype} \quad \text{Identifier} \\
\text{out} & \quad \text{Datatype} \quad \text{Identifier} \\
\text{inout} & \quad \text{Datatype} \quad \text{Identifier}
\end{align*}

where

\begin{itemize}
  \item \text{Datatype} is an IDL type specifier and \text{Identifier} is the parameter name.
  \item \text{in}, \text{out} and \text{inout} are direction attributes
  \item with the following meanings:
  \begin{itemize}
    \item \text{in} Data is passed from client to target
    \item \text{out} Data is passed from target to client
    \item \text{inout} Data is passed in both directions
  \end{itemize}
\end{itemize}

This example shows the operation declarations that could be used with the example in Section 2.1.3.

\begin{verbatim}
module Transaction
{
   interface Cash
   {
      float deposit ( in float credit_amount, 
                      out float balance );
      float withdraw ( in float debit_amount, 
                      out float balance );
   };
   interface Check : Cash
   {
      boolean cleared( in long ref_num, 
                       in short num_of_days );
   };
}
\end{verbatim}

where highlighted codes are IDL reserved words.

2.1.4.1 Operation Qualification

In the operation declaration

\[\text{oneway} \text{ OpType OperationName "([parameters])" Opexceptions Opcontext} ;\]

the optional qualifier is the keyword \texttt{oneway}.
A one-way operation passes parameters (if there are any) to the object implementing the operation but does not wait for completion of the operation and does not return results. An operation with a one-way attribute cannot contain any out or inout parameters and must specify a void return type. One-way operations are not guaranteed delivery.

For example

```plaintext
interface Stats
{
    oneway void send(in StatsRecord sr);
};
```

is an example of a one-way operation declaration.

2.1.4.2 Raising Operation-Specific Exceptions

In the operation declaration

```plaintext
[oneway] OpType OperationName "([parameters])" [Opexceptions] [Opcontext];"
```

the optional Opexceptions is used to specify operation-specific exceptions that may be raised as a result of invoking the operation. The form of the expression is

```plaintext
raises(ExceptionNames)
```

where:

- **raises** The IDL keyword for raising user defined exceptions
- **ExceptionNames** One or more previously declared exceptions in the form:
  ```plaintext
  Exception1, Exception2, ..., ExceptionN
  ```

For details of exception declarations, see Section 2.1.6.

User exceptions cannot be raised for a one-way operation.

In addition to the user defined exceptions specified in the raises expression, there are system exceptions corresponding to runtime errors that may occur during execution of an operation. The system exceptions are listed in the orb2 for Eiffel Reference documentation. System exceptions cannot be listed in a raises expression.

This IDL fragment defines an exception if a Divide operation attempts to divide by zero

```plaintext
interface Calculator
{
    //exception declaration
    exception DivideByZero ();

    long divide(in long op1, in long op2, out long remainder)
    raises (DivideByZero);
    ...
};
```

This IDL declares an exception named DivideByZero; the divide operation has selected this as a possible exception that might be raised during the execution of the operation.

2.1.4.3 Operation Contexts

In the operation declaration

```plaintext
[oneway] OpType OperationName "([parameters])" [Opexceptions] [Opcontext];"
```

the optional Opcontext is used to pass context information to the object implementing the operation. Although the grammar is supported by the compiler, orb2 does not support the passing of context information.
2.1.5 Data Types

The IDL data types are grouped into
- Basic data types
- Constructed data types
- Template data types
- Complex data types

IDL uses declarations similar to the C and C++ languages for naming data types. The keyword `typedef` associates a name with a data type, as do the constructed data types `struct`, `union` and `enum`. The following sections describe the IDL data types. How these data types map onto Eiffel data types is described in the `orb2 for Eiffel Reference` documentation.

2.1.5.1 Basic Data Types

Table 2.1 describes the basic IDL data types.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description and Example</th>
</tr>
</thead>
</table>
| `float`     | A single-precision floating point number encoded as a 32 bit, IEEE single precision value, most significant byte first  
              |  
              | `typedef float balance;`                                                                |
| `double`    | A double-precision floating point number encoded as a 64 bit, IEEE double precision value, most significant byte first  
              |  
              | `typedef double balance;`                                                               |
| `short`     | A signed integer in the range $-2^{15}$ to $2^{15} - 1$  
              |  
              | `typedef short size;`                                                                    |
| `long`      | A signed integer in the range $-2^{31}$ to $2^{31} - 1$  
              |  
              | `typedef long parm;`                                                                     |
| `unsigned short` | An unsigned integer in the range 0 to $2^{16} - 1$  
              |  
              | `typedef unsigned short result;`                                                         |
| `unsigned long` | An unsigned integer in the range 0 to $2^{31} - 1$  
              |  
              | `typedef unsigned long product;`                                                         |
| `char`      | A single character data type  
              |  
              | `typedef char indicator;`                                                                |
| `boolean`   | A data type which can take one of two values; TRUE or FALSE. Note that **TRUE** and **FALSE** are IDL keywords  
              |  
              | `typedef boolean onflag;`                                                                |
| `octet`     | An 8-bit quantity  
              |  
              | `typedef octet stateflags;`                                                              |
| `any`       | A data type that permits the specification of values that can express any IDL data type  
              |  
              | `typedef any value;`                                                                     |
### 2.1.5.2 Constructed Data Types

IDL has three constructed data types; structures, discriminated unions and enumerations.

**structures**

The IDL structure data type is most similar to a C structure except that a structure tag name is mandatory, and has the general form:

```
struct StructName
{
    type membername1;
    type membername2;
    ...
    type membernameN;
}
```

where

- **struct**: IDL keyword for structure declarations
- **StructName**: Structure name
- **type**: Any legal IDL type
- **membername1 to membernameN**: Unique identifiers within the enclosing scope of the structure
Alternatively

typedef struct StructTagName
{
    type membername1;
    type membername2;
    ...
    type membernameN;
} StructTypedefName;

where
struct          IDL keyword for structure declarations
StructTagName   Structure tag name
type            Any legal IDL type
membername1 to   Unique identifiers
membernameN
StructTypedefName Structure type definition name

For example

   // IDL
   struct Pigment
   {
       string   shade;
       float    balance;
   };
   typedef struct ColorTag
   {
       Pigment   pigment;
       string    color;
       short     cost;
   } Color;
   typedef Color Mosaic;
   Mosaic paint( in Pigment choice );

discriminated unions  IDL unions are a combination of C and C++ union and switch statements. They are discriminated by a mandatory typed name in the union header that determines which union member to use at run time. The general form of the union declaration is

   union UnionName
   switch (Switchtype)
   {
       case constant1 : type MemberName1;
       case constant2 : type MemberName2;
       ...
       case constantN : type MemberNameN;
       [default : type MemberDefault;]
   };
Alternatively

typedef union UnionTagName
  switch (Switchtype)
  {
    case constant1 : type MemberName1;
    case constant2 : type MemberName2;
    ...
    case constantN : type MemberNameN;
    [default : type MemberDefault;]
  } UnionTypedefName ;

where union, switch, case and default are IDL keywords. The default case is optional and need not be present.

UnionName
  Union name
Switchtype
  Either long, short, unsigned long,
  unsigned short, char, boolean, enum
  or a previously declared identifier of one of
  these seven types
constant1 to constantN
  Constants or constant expressions consistent
  with Switchtype. See Appendix A.
type
  Legal IDL type
MemberName1 to Unique identifiers, within
MemberNameN
  the enclosing scope of the union

For example

// IDL

union Details
  switch (short)
  {
    case 1 : short arg1;
    case 2 : long arg2;
    case 3 : string arg3;
    default : boolean arg4;
  };

interface Records
{
  typedef Details Choice;
  boolean insert( in Choice c1 );
};

enumerations

The IDL enumerated type is similar to the C and C++ enumerated type except that the enumeration name is mandatory and has the general form

enum EnumName "{"id1, id2, ..., idn"};"

where
enum
  IDL keyword for an enumeration declaration
EnumName
  Enumeration name
id1 to idn
  Legal values that a type of EnumName may have

For example

enum color {red, blue, green};
2.1.5.3 Template Data Types

IDL has two template data types: sequences and strings.

**sequences**

The IDL sequence data type is a one-dimensional array of a specified type. The general form of a sequence declaration is

```
typedef sequence "<"type"," size">" identifier;```

where

- `sequence` is an IDL keyword
- `type` is a legal IDL data type
- `size` is an optional positive constant which indicates the maximum sequence size

For example

```c
struct AccountSummary
{
    short account_num;
    float balance;
};

typedef sequence <AccountSummary> Accounts;
```

**strings**

The IDL string data type has the following declaration

```
string ["<"size">"] identifier;
```

where

- `string` is an IDL keyword
- `size` is an optional positive constant specifying the maximum string size
- `identifier` is any unique name

For example

```c
string <10> propname;     // bounded
typedef string anyname;   // unbounded
```

2.1.5.4 Complex Data Type

The IDL complex data type defines multidimensional, fixed size arrays.

**arrays**

The IDL array data type includes explicit sizes for each array dimension. The general form of the declaration is

```
type ArrayName ["size1"]...["sizeN"]
```

where

- `type` is a legal IDL data type
- `ArrayName` is an array name
- `size1` to `sizeN` are positive integer constants specifying the size of each dimension or slice

For example

```c
#define MAXMEMBERS 1000

typedef string MemberNames[MAXMEMBERS];

long Matrix[4][50];
```
2.1.6 Exception Declarations

There are two types of exception that can be returned during the execution of an operation

- System exception
- User defined exception, specified in IDL by exception declarations. See also the raises expression in Section 2.1.4.2.

The form of an exception declaration is similar to a struct declaration. Users should note that exceptions, unlike structs, cannot be type defined (typedef):

```idl
exception ExceptionName
{
    type membername1;
    type membername2;
    ...
    type membernameN;
}
```

where

- `exception` The IDL keyword for an exception declaration
- `ExceptionName` An IDL identifier naming this exception type
- `type` A valid IDL data type
- `membername1` to `membernameN` Unique IDL identifiers, within the scope of the exception block

The `membername` fields are optional, for example the exception declaration

```idl
exception BadType {};
```

is valid.

The members of the exception declaration are used to provide supplementary information. The value of the exception identifier is available to the client code to determine which particular exception has occurred. See Chapter 3 for details of exception handling from client code.

This IDL fragment declares an exception with a member used to pass details of the exception.

```idl
interface Computer
{
    typedef enum ProblemTag
    {
        NoPowerLead,
        NoSocket,
        Fused
    } Problem_t;

    exception Diagnostics
    {
        Problem_t p;
    }
    exception InvalidUnit{};
    void switch_on() raises (Diagnostics, InvalidUnit);
}
```

**Note** The raises expression is used to specify which exceptions can be raised by an operation and is described in Section 2.1.4.2.
IDL Preprocessing

IDL preprocessing is based on ANSI C and C++ preprocessing, and provides

- Macro substitution
- Conditional compilation
- Source file inclusion (#include)
- orb2 specific type declaration inclusion (#import)

Directives are also provided to control line numbering in diagnostics, for symbolic debugging, to generate a diagnostic message with a given token sequence and to perform implementation-dependent actions (the #pragma directive).

Lines beginning with # are called directives and communicate with the pre-processor. White spaces appearing before the # character are ignored. These lines have syntax independent of the rest of IDL, they may appear anywhere and have effects that last, independent of the IDL scoping rules described in the following section, until the end of the IDL specification.

You can continue a preprocessing directive (or any line) on the next line in the IDL specification by placing a back slash character (\) at the end of the line to be continued. A back slash character cannot be the last character in an IDL source definition.

The IDL pre-processor directives are the following ANSI standard directives

```
#ifdef
#define
#else
#else if
#include
#include
#else
#define
#else
#error
#endif
#pragma
```

which have their normal usage. The orb2 IDL compiler also supports an exclusive #import directive. This directive is in addition to the CORBA IDL standard and takes the form:

```
#import "Filename.idl"
```

One of the primary uses for the preprocessing facilities is to include definitions from other IDL specifications. Text in files included with a #include directive is treated as if it appeared in the including file. Text in files specified with a #import directive is examined to locate the declarations for any unresolved data types in the importing file.

There is an important distinction between #include and #import. No code is generated for types included in the imported files. The content is processed by the compiler and used for cross reference purposes, but the assumption is that the file will have been pre-compiled to generate any supporting code. This technique is frequently used to pick up the definition of inherited interfaces, without regenerating the code for them.

#include, on the other hand, is recognized by the preprocessor, which inserts the entire text of the #included file into the parent file before the compiler parses the text. Files that are #included must contain only valid IDL syntax. The resulting generated output reflects all types and routines, just as if the user had written the #included file in line.

**Note** #included files must not contain #import directives.
### 2.1.8 Example IDL

This is an example of an IDL specification as part of a machine tool application.

```idl
File: Wmaker.idl
#include "Common.idl"  // Common global definitions
#import "Tool.idl"     // Tool Operations and definitions
#import "Widget.idl"   // For definition of a Widget
interface WidgetMaker : Tool
    // Derive tool type 'WidgetMaker'
{
    // Local definitions
    enum Qualifier{ blunt, bent, snapped };
    exception ToolDefective( Qualifier q; );
    exception OutOfMaterial();

    enum Shape { probe, chisel, dart };  // Create a Widget tool of the required shape
    Widget cast( in Shape s )
        raises( ToolDefective, OutOfMaterial );
    // Polish a Widget
    void polish( in Widget w );
};
```

Most of the principles of IDL are demonstrated by the inclusion of common definitions. WidgetMaker becomes a specialization of a Tool, also providing extra operations for `cast` and `polish`, where `cast` can raise one of two exceptions, the first of which is further qualified with a field value. The WidgetMaker can create Widgets through the `cast` operation and definitions for Widget are got from the imported Widget definition.

In this particular example both the interface type Widget and the imported widget definition file name `Widget.idl` are, coincidentally, the same. However, users should be careful to import the exact case sensitive IDL file name and use any interface names declared there explicitly. An assumption that the IDL file name and the interface name contained within are the same can never be made.

It is important to note that all operations declared within an IDL specification must be implemented within each object type supporting this interface. orb2 supports the persistent server policy, in which each server is capable of supporting multiple active objects of the same type or different types, where only one complete implementation, per type, is required.

### 2.2 Name Collision Avoidance

As mentioned earlier, it is possible to have name clashes arising in the generated Eiffel application code if the same identifier is declared in different IDL name scopes. It is also possible for names in IDL to clash with Eiffel keywords, or with names used in your applications. The name collision avoidance mechanism provides the means of overcoming these clashes.

Whenever an IDL file is compiled, the Eiffel stub generator looks for a file with the same name as any input IDL file but with the extension `enl` (Eiffel Name List). The contents of this `enl` file describe how the compiler should map IDL identifiers to Eiffel identifiers (class and feature names). For the topmost IDL file, at the end of the compilation, the stub generator outputs an `enl` file containing all the identifiers found in that IDL file, as modified by any input `enl` file. You can then edit this to modify the stub generator's behavior on subsequent compilations of that IDL file.
The stub generator also looks for a default \textit{enl} file containing mappings for use in all compilations. This latter default \textit{enl} file is supplied originally with name mappings which avoid clashes with Eiffel keywords and well-known Eiffel kernel features. This can also be edited to add additional site-wide name mappings.

The \textit{enl} files are searched for in the working directory, and in the directories added with the \texttt{-I} option (they are expected to reside in the same directories as the IDL files. The default \textit{enl} file is also looked for in $DAIS_ROOT/idl even if this is not included in an \texttt{-I} option (this is where it is originally delivered).

The \textit{enl} file which is output by the compiler is put into the topmost output directory (either that identified by \texttt{-c} option, or one named after the topmost IDL file). If you want to use this \textit{enl} file (possibly edited) on subsequent input, you must move it back into the \textit{idl} directory.

It is important to note that although the generated Eiffel class and feature names are changed by this mechanism, the names which appear "on the wire" still reflect the original IDL names. This ensures that the Eiffel application can interwork with CORBA applications derived from the same IDL but in different languages, or on different ORBs.

The format of the file is very simple.

\begin{verbatim}
( [comment]  idl-name eiffel-name (pragma-name pragma-value)* )*
\end{verbatim}

where

| ( | Optional |
| )* | Zero or more repetitions |

\begin{verbatim}
idl-name ::= <simple IDL identifier> | <fully scoped IDL identifier>
eiffel-name ::= <an Eiffel identifier>
pragma-name ::= '#'<identifier>
pragma-value ::= <string literal>
\end{verbatim}

Everything is separated by white space.

Simple IDL identifiers are names that will be remapped whatever the context. This is used for reserved words, features of \texttt{GENERAL} and so on.

Fully scoped IDL names are mapped only in a specific context.

Each entry has both IDL name and Eiffel name even if no translation occurs, so the \textit{enl} file represents a complete list of names in a specification. On output, the file will be sorted in IDL name order.

Comments have the same syntax as in Eiffel and are associated with the following item; this is important when the compiler inserts new entries. By implication, comments only appear before, not inside, items.

The \texttt{pragmas} are used to change the Repository Ids and may have the values \texttt{id}, \texttt{version} and \texttt{prefix}. They have the same semantics as the equivalent IDL \texttt{#pragma} directive. orb2 does not currently support the use of these pragmas directly in IDL.

\textbf{Example}

\begin{verbatim}
-- Global name replacements
copy           idl_copy
export         idl_export
::Calculator    CALCULATOR
::Calculator::Add add
::Calculator::Add::op_1 a_op_1
::CORBA         CORBA #prefix "omg.org"
::CORBA::Object CORBA_OBJECT
\end{verbatim}
Notes

1. The compiler never deletes entries, only adds them.
2. The -a option (which prefixes argument names) only applies to new names. If you change your mind about the prefixing of arguments, you have to delete or edit the *enl* file.
3. If you have previously specified a mapping for a particular name, then deleted that name from the IDL and reintroduced it, the mapping is preserved. This is why the compiler never deletes entries. When the name is reintroduced it might be a different sort of object, for example an operation *fred* might be replaced by a struct *fred*. As far as the compiler is concerned, this is still the same name, and so still gets mapped as before.

2.3 Compiling the IDL

IDL specifications are compiled to generate stubs and skeletons, these are Eiffel source code files that are compiled with your application code. The orb2 IDL compiler is called *stubgen*; it is invoked as

```
```

These [flags] are used to modify the default behavior of the compiler

-a  Prefix argument names with a_.
Because the Eiffel language does not allow argument names to be the same as feature names, clashes can arise. This option causes a_ to be prepended to all IDL argument names to minimize the risk of clashes.

-b  The compiler is to build the classes required to compile a complete application from this IDL specification. These will be placed in a cluster IDL_Root within the target code directory. By default, this cluster is not generated.

-u<idlfile>  Indicates that the application uses classes from the specified IDL file. By default, the list starts off with *reloc.idl* and *trader.idl* since these are used by the orb2 runtime system. Leaving out the IDL file name causes the list to be emptied.

-e  This option is not used in the Eiffel binding.

-leiffel  Eiffel stub/skeleton generation language.
Specifies Eiffel as the language for generating the stubs and skeleton.

-m  Check mixing of float and integer in expressions.
When this flag is set the CORBA compiler will report an error if an attempt is made to mix integers and floats in a constant expression such as

```
const short Value = 4*72/3+2.3;
```
otherwise the compiler will truncate as necessary

-n  Switch on case sensitivity for identifier names.
The default behavior of the compiler is such that it will not differentiate between identifier names on case alone. Thus *NaMe* and *name* are considered equivalent. This behavior may be modified by this flag to be CORBA conformant.

-o  Output suppression - semantic check only.
Produces no generated code output; however all the semantic checking is still performed.
2.3 Compiling the IDL Interface specification in IDL

orb2 for Eiffel User’s Guide

orb2 Release 3.2.9

The remaining command line parameters are

-oc Output suppression - generate client code only.
This option may be used if only the client side of an application is
to be written in Eiffel. This reduces the number of classes
generated. It is not vital to use this option, as the superfluous
classes would be ignored by the Eiffel compiler.

-p Suppress Type Codes.
Causes the compiler to generate code which does not include
support for CORBA::Any. This simplifies the generated classes
considerably, but of course cannot be used if the application does
need to use CORBA::Any.

-r Suppress relocation code in client stubs.
Causes a failed call from a client to not attempt to find a relocated
replacement object reference.

-s Strict CORBA conformance (equivalent to -n -m).
Combines the behavior of the -n and -m flags such that both are
enforced.

-t Produce prototype server templates.
Causes the compiler to generate a sample _IMPL class for each
interface into a file with the extension .esk.

-c <dir> Output generated source files in the specified directory.
The default effect is that all source files and directories are
generated in the current working directory. This flag permits an
alternative directory to be specified.

-h <dir> This option is not used in the Eiffel mapping.

-x This option is not used in the Eiffel stubgen compiler.

The remaining command line parameters are

-I <dir> Allows users to specify additional paths that the compiler will search to
locate other .idl files needed for #include or #import. This may be used
as often as required to enable multiple directories to be specified.
For example

    stubgen -I -I/usr/idl -I<DAIS_ROOT>/idl test.idl

-D <define> Allows users to present optional constant definitions to the CORBA
compiler in the form

    -Dname=value

This constant definition is processed by the preprocessor and can be used
to conditionally change the IDL syntax presented to the compiler. The
command line value overrides any equivalent value encountered within
the file.
For example

    // File.idl
    #ifdef _do_it
    // some IDL syntax
    #endif

will only be processed if -D_do_it appears on the command line

IDLfile This must be a text file containing valid IDL syntax whose extension
suffix must be .idl.
This is an example UNIX command line (assuming $DAIS_ROOT is set in your environment)

    stubgen -leiffel -ISDAIS_ROOT/idl -I/usr/dev/interfaces Intrst.idl

This passes the file Intrst.idl into the CORBA compiler and sets up the paths $DAIS_ROOT/idl and /usr/dev/interfaces for #include or #import files. The generated output is for the Eiffel mapping.

### 2.4 Clusters to Include in an Eiffel orb2 Application

The orb2 system as delivered includes two main runtime clusters:

- $DAIS_ROOT/eiffel/corba contains the core CORBA functionality.
- $DAIS_ROOT/eiffel/dais contains the core orb2 functionality which is used by the CORBA classes. It also contains additional functionality which may be directly used by the application.

All orb2 Eiffel applications must include these two clusters.

There are also a number of clusters which are generated by compiling the standard orb2 IDL files. In fact there are two variants of these clusters. One compiled with support for CORBA::Any, and one compiled without. These clusters are under the above orb2 cluster in two directories called idl and idl_tc. The latter supplying the CORBA::Any support. Every orb2 Eiffel application needs to include the subset of these clusters which are used by the orb2 runtime system. Examine the example applications to see which of the clusters are required.

When you compile any IDL file you will find that you will have generated a number of directories, one for each namespace in the IDLT. These are named from the IDL files, modules and interfaces defined. All of these clusters must be included in any application which uses them. The clusters contain classes based on the types etc. defined in those name scopes, as well as a number of other helper classes which you will not directly use. These latter classes are normally prefixed with G_.

If you compile the IDL with the -b option, an additional directory called IDL_Root is produced. This contains a few classes which are referred to from the CORBA cluster, and must therefore have names which do not vary from one specification to another.

For any Eiffel orb2 application you should pick one IDL file as the topmost IDL file for the application. This will #import or #include, possibly indirectly, every other IDL file which it depends on. You should compile this file with the -b option, and include the IDL_Root cluster from this compilation in the application, and no other IDL_Root cluster.

The application may also use IDL objects in its implementation which the IDL specification does not itself depend on. All constants and typecodes which the application might use must also be reachable from the classes in the IDL_Root cluster. While it is possible to artificially add #import directives into the topmost IDL file a better way is to add them using the -u option on the compilation of the topmost file.
Chapter 3

Application Development in Eiffel: CORBA

3.1 Introduction

This chapter discusses the basic services provided by orb2 that are entirely conformant with CORBA, enabling users to develop CORBA conformant, distributed Eiffel applications on the orb2 platform.

The techniques presented here are the principal tools for orb2 application development. The concepts are outlined first and you should understand these before proceeding. In addition, all the Class Library Interfaces documented in this chapter are applicable throughout orb2. For exact descriptions and formal definitions of these classes, refer to the orb2 for Eiffel Reference documentation.

3.2 Capsules, Stubs and Skeletons

Objects cannot exist in isolation; they require an executable framework in which to operate. This framework is called a capsule. A capsule is a self-contained, complete executable process and its construction is operating system specific. Capsules containing orb2 objects are also further classified as servers. A capsule is a single process and is the generic term used to refer to orb2 executables across all operating environments.

In Eiffel, the code to support such a capsule is generated by compiling a root class. orb2 requires certain behavior at the level of the capsule, and the deferred class DAIS_APPLICATION provides the basis of this behavior. The compiled application must include a single instance of DAIS_APPLICATION, typically by inheritance from the application’s root class.

3.2.1 Stubs and Skeletons

Conceptually, a client invocation on an object can be viewed as shown in Figure 3.1.

1. The invocation originates in the client code and the request is passed into the stub code. The request is normalized for network transmission and then passed into the orb2 Infrastructure that handles all the inter- and intra-capsule messaging.

2. The request arrives at the target and is passed to the appropriate skeleton, taking the normalized request and converting it to local representation.

3. Arguments and results parameters are initialized before the desired method is invoked.

4. Both return values and return results are passed from the object method into the skeleton that normalizes them for transport back to the caller.

5. The results are shipped back through the infrastructure to the caller’s stubs where the data is converted to local format and passed to the client.
This conceptual model is correct for any invocation, whether the client and target are
- Co-located within the same capsule
- Separated across capsules, within the same node
- Separated across the network

The nature of the operating system, and the underlying communications mechanisms, are of no importance to the interaction. See Figure 3.2.

The stubs effectively insulate the client from the complexities of both intra- and inter-capsule communications and data representation. Operation invocations remain syntactically and semantically identical, regardless of the location of the client with respect to the target object.

orb2 is a multithreading implementation of CORBA, and the precise description of way the stubs and skeletons are executed in this multi threaded environment is described in Section 5.2.
3.2.2 Server Side Class Hierarchy

Code generated from IDL contains a representation of the target interface, for example, this IDL extract

File: vdb.idl
// Vehicle Data Base (VDB)
interface VehicleDB
{
  void add();
};

This constitutes a simple service definition that can be implemented as an object in Eiffel and contacted through an appropriate client. The vehicle data base interface supports the single operation `add`, which returns a `void` and accepts no parameters. The semantics of this operation are of no consequence within this description. However, this interface will continue to evolve throughout this chapter.

The IDL compiler generates a number of Eiffel source files among which is found the class `VEHICLEDB`. This is an abstract class defining the common semantics of the `VehicleDB` interface which both clients and servers must conform to

```eiffel
default class VEHICLEDB
  inherit
    CORBA_OBJECT
  feature
    add is deferred
end
```

where the base class `CORBA_OBJECT` is deemed to be the interface root within CORBA.

Now you have a clean Eiffel representation of the interface from which concrete classes can be derived. However, the server side requires an upcall mechanism and marshaling code (the skeleton code) to the developer-supplied implementation. The skeleton code is provided by some additional generated classes. These classes can be, to a large extent, disregarded by the programmer, but their effect is to make calls on the class `VEHICLEDB_IMPL` which forms the basis of the implementation. This class inherits from `VEHICLEDB`. As generated it is effective with null implementations of each operation.

To supply the server operations the developer either edits the null implementations, or makes the class deferred, and effects them in a descendent. The deferred technique minimizes the impact of recompiling on the programmer-supplied source text, and allows a programmer to have a single Eiffel server class supplying more than one IDL interface without delegation. However, it complicates the inheritance of IDL operations effected in inherited interfaces and is really only appropriate where the Eiffel application design does not match the IDL interface structure.

As generated the `__IMPL` class has a sample creation which just makes it visible as a CORBA object. The programmer would typically edit this to perform any necessary initialization. It is not mandatory for the Eiffel object to be visible as a CORBA object all its life. This policy allows `IMPL` classes to become CORBA objects whenever the application chooses, rather than for the lifetime of their existence. Two procedures are supplied in the `IMPL` class to enable this

- `corba_create` makes an `IMPL` object available as a CORBA object. The effect of this procedure is to cause the orb2 ORB to adopt the Eiffel object. This will prevent it being garbage collected until the `corba_dispose` procedure is called. This adoption is justified on the basis that the ORB is acting as a proxy client of the `IMPL` object, and the Eiffel GC is not aware of this remote reference.
- `corba_dispose` is used to inform the ORB that no clients are expected to need to invoke this object, and it can be removed from the ORB’s domain.
Two additional procedures `u_export`, and `u_withdraw` are supplied to make the CORBA object visible in the Trader name spaces and withdraw it respectively. See Section 3.5.2 and Chapter 4.

Example Server Application

```eiffel
class VEHICLEDB_APPLICATION
  inherit DAIS_APPLICATION
  redefine terminate end
  feature
    initial_object : VEHICLEDB_IMPL
    initialize_application is
      do
        initial_object.make
      end
    terminate is
      do
        if initial_object /= Void then
          initial_object.terminate
          initial_object := Void
        end
      end
  end
end

class VEHICLEDB_IMPL
  ... creation
  make
  feature
    make is
      do
        corba_create
        u_export ("/",""
      end
    add is
      ...
    End
  terminate is
    do
      u_withdraw
      corba_dispose
    end
end
```

3.2.3 Client Side Class Hierarchy

An application level client interested in invoking operations on remote objects must obtain a legitimate object reference to the desired target. Co-located IDL interfaces, within the same capsule, are considered to be remote and are accessed in the same manner, that is, location transparency is not broken, regardless of any optimizations taking place within the ORB implementation.

In the Eiffel mapping the client of the above VehicleDB interface makes calls on entities declared as VEHICLEDB.

```eiffel
class CLIENT
  feature
    db : VEHICLEDB
    call_db is
      do
```
3.2.3  Client Side Class Hierarchy

Application Development in Eiffel: CORBA

```eiffel
db.add
end
end

VEHICLEDB is deferred, so cannot be created. It remains to show how the entity db points to an effective object which references the remote VEHICLE_DB_IMPL.

An additional class is generated from the IDL: VEHICLEDB_REF, which inherits from VEHICLEDB. This class is effective, embeds the CORBA object reference, and contains the stub marshaling code. An instance of this is actually used to make the call, it is not normally sufficient for a client to just create one of these, it must acquire a valid embedded object reference from somewhere. Acquiring the object reference is referred to as binding.

The act of binding can be achieved in many ways, most of which are catered for by creation procedures on VEHICLEDB_REF. For example, someone may give you an object reference externalized as a string on a floppy disk which you read into your program. You would then create the VEHICLEDB_REF by

```eiffel
db : VEHICLEDB
objc_ref_as_string : STRING
...
!VEHICLEDB_REF!db.from_string(objc_ref_as_string)
```

The routine from_string is one such creation procedure. This example may seem odd but it is, nevertheless, valid, though not the norm.

Typically, orb2 clients procure their initial object references from the Trader. The Trader is an orb2 object support service and is treated like any other remote object, with the exception that its own object reference is a well known piece of information within orb2. It has an IDL interface described in Chapter 4. The Trader operates much like using Directory Assistance, providing information about how to contact some service that exists. You, the user, have sufficient information to allow Directory Assistance to sort through its database to provide either a range of contact points, an explicit contact point, or none at all.

**Example**

```eiffel
class CLIENT
  creation
    make
  feature
    db : VEHICLEDB
    make is
      do
        !VEHICLEDB_REF!db.u_import("/",""")
      end
    call_db is
      do
        db.add
      end
  end
end
```

The u_import routine applied to db is a creation procedure which invokes the Trader behind the scenes, and embeds the imported object reference inside the created VEHICLEDB_REF object.

In fact, this import routine hides some complexity which ensures the VEHICLEDB_REF object is of the right type. Without this simplification, the code for importing would read

```eiffel
!VEHICLEDB_REF!db.narrow(  
  dais.trader.import("VehicleDB","/",""))
```
The return from the trader import routine is of type CORBA_OBJECT_REF which is not suitable for use as a VEHICLEDB_REF. The narrow procedure takes the embedded object references and narrows it to a VEHICLEDB_REF. This is somewhat similar to the assignment attempt operator in Eiffel but operates in the CORBA distributed type space.

**Figure 3.3 Narrowing**

Figure 3.4 depicts the various components involved in an invocation between the client and server implementation.

**Figure 3.4 Proxy Access**
The object references embedded in _REF and _IMPL classes consume resources inside orb2. The resource will be released automatically when the object is garbage collected. However, in the case of the server _IMPL class, the garbage collector only scavenges the object after the call to corba_dispose.

### 3.2.4 Stubs and Skeletons, Servers and Clients: Summary

Key points to bear in mind from this section are

- Server implementations must be effective in the _IMPL class or a descendent thereof
- Servers may optionally advertise their object references through the Trader
- Clients may choose to obtain object references from the Trader

### 3.3 Inheritance

CORBA offers multiple inheritance for interfaces with similar semantics to that offered by Eiffel. The Eiffel generated interface classes manifest the same inheritance tree as that in the IDL.

The IDL syntax for inherits is very similar to C++ and takes the form

```
interface VehicleDB : DB,Vehicle {};
```

This results in the class texts

```
defered class VEHICLEDB
inherit
  DB
  VEHICLE
...

class VEHICLEDB_REF
inherit
  DB_REF
  VEHICLE_REF
...
class VEHICLEDB_IMPL
inherit
  DB_IMPL
  VEHICLE_IMPL
...
```

CORBA IDL does not offer any scheme for renaming or redefinition of operations or attributes. It is worth noting that while the Eiffel server classes must be based on the _IMPL generated classes, the programmer can make full use of the Eiffel rename and redefinition facilities in providing the more restricted CORBA interface definitions.

If more than one _IMPL class is inherited by an Eiffel object supplying more than one IDL interface, then that class must rename the feature names which each _IMPL class have in common.
3.4 Capsule Construction

Eiffel capsules are constructed by compiling an Eiffel root class, the creation procedure of which represents the entry point for the running process image. There are certain ORB-related functions that must be called in order to provide a viable orb2 execution domain, and to this end a DAIS_APPLICATION class is provided. The make_dais_application routine of this class must be called early in the application startup. The feature has three main phases:

1. ORB (DAIS) initialization (orb_init)

This is supplied by the DAIS_APPLICATION class and is unlikely to need any changes by the application.

2. Application start (initialize_application)

This stage is deferred and must be supplied by the application. This is where a server capsule creates the initial CORBA objects, and where a client capsule performs its work. This is also the place where a GUI application which creates windows in its main thread puts its main thread message loop.

3. Synchronization (capsule_is_ready)

The final phase is supplied by the DAIS_APPLICATION class and is unlikely to need redefinition. The effect is for client only capsules to terminate orb2. For server capsules, it awaits calls on the server objects in this thread.

More details about the DAIS_APPLICATION class can be found from the class text.

The orb2 ORB and its support for Eiffel is provided in a number of components:

- A core library (and DLL) common to all language variants ($DAIS_ROOT/lib/libdais.a on UNIX and %DAIS_ROOT%\lib\dais32.lib on Windows NT)
- A library of Eiffel-specific support ($DAIS_ROOT/lib/dais_eiffel.a on UNIX and %DAIS_ROOT%\lib\dais_eiffel.lib on Windows NT)
- CORBA and orb2 clusters of classes wrapping the above libraries

The clusters become compiled into the application capsule as required. The libraries must be linked with the final application using the facilities of the Eiffel compiler.

3.4.1 Starting a Capsule

orb2 capsules are constructed as operating system specific, executable images and require no special attention to start them. orb2 employs the Persistent Server policy, which means that a capsule can be started in the same way as any other process. It is possible to use orb2 Support Services to launch capsules (and their top level objects) on demand transparently, as part of the Trading cycle. See Chapter 6. For example:

On UNIX systems

```
// start capsule in foreground
prompt> mycapsule

// start capsule in background
prompt> mycapsule&

// start multiple capsules in background
prompt> for i in 1 2 3
> do
> mycapsule &
> done

// script file
#!/bin/sh

echo "Starting DAIS capsule"
nohup $(DAIS_BIN)/mycapsule >
$(DAIS_LOGDIR)/mycapsule.log 2>&1&
```
echo "Running"

On Windows systems

// starting from the command line
c:\dais\bin\mycapsul.exe
// starting from File Manager or Explorer
// double click on executable icon for 'mycapsul.exe'

3.4.2 Stopping Capsules

As indicated above, the make_dais_application routine never returns, but waits for any
incoming calls to objects that have been created. In the case of clients, typically, there are no
active objects; then the ORB will terminate the capsule. There is one exception to this rule:
when the client capsule has no active orb2 objects but can receive requests from other sources
such as sockets and pipes. Such capsules require special construction and functionality to
perform non-blocked user input/output.

For capsules supporting active orb2 objects, the ORB continues to wait for incoming calls until
told otherwise. Termination in this environment can be achieved in a number of ways

• Through the use of the LifeCycleObject::remove operation on the capsule object. This
  results in the should_terminate function of the DAIS_APPLICATION being invoked. If
  this returns TRUE (the default behavior), the capsule will begin to terminate.

• Through external stimulus, for example operating system signal.

• By releasing all the active server objects, and exiting any GUI message loop. In a server
capsule not started by the NodeManager/Factory pair, if all server CORBA objects are
  disposed, then the capsule will terminate.

• Calling the abort routine on the application. This is a mechanism for the application itself
to force termination in extreme circumstances.

Each of the above will invoke the DAIS_APPLICATION terminate routine which should be
redefined to cleanly close down all server objects, otherwise they will be summarily
terminated. Generally, the call to abort is reserved for extreme cases. CORBA IDL exceptions
occurring in server invocations that are left unrescued do not cause the capsule to terminate.
They are merely passed back to the client.

3.4.3 Stopping Windows Capsules

Unlike most other operating system environments, Windows GUI applications do not support
signals. orb2 Windows capsules are typically destroyed as a result of user direction, such as
File exit, or through a CAPSULE.remove request. In the former case, a call to abort should be
made on receipt of the termination messages of the form WM_CLOSE and WM_DESTROY.

3.5 Locating Objects

Locating CORBA objects consists of getting an appropriate instance of a _REF class created
with an embedded object reference. The client application code is then at liberty to invoke
operations on that object. Unlike single address space applications, there is no guarantee of
service from the remote object.

3.5.1 Binding

Binding is the process of using an object reference for the first time. An orb2 object reference is
a complex data structure containing, among other things, network addressing information for
the Network Service Access Point (Node), the Transport Service Access Point (Capsule) and a
choice of protocol stacks that may be used (for example IIOP, TCP/IP, UDP/IP, OSI). Instance
data is also encoded so that the correct instance of an object type is addressed within the server capsule. It is not until the client ORB has issued a request and received an acknowledgment, that the client capsule binds the session and caches the addressing information locally.

Once a session is bound, it is reasonable to expect service from the target object. However, it is possible for such sessions to break down; breakdowns can occur because

- The target node has gone down
- The target operating system has summarily removed the capsule from its working environment
- The interface is congested due to overload and the request not being serviced within the time out period
- A LAN routing problem arises

It is for these reasons that exception handling over distributed object communications is so important. It is only by relying on the underlying infrastructure that callers are notified of failure, otherwise requests could lock up resources indefinitely pending responses.

### 3.5.2 Getting Object References

As discussed earlier, there is no single way to obtain an object reference. Three principal approaches for obtaining an object reference are

- Use a third party service
- As an IDL operation return value, `out` parameter or `inout` parameter
- Through externalization

A third party service is some object that acquired object references on behalf of clients. This could easily be constructed from application code. However, orb2 already delivers a tried and tested service of this kind: it is called the Trader.

#### 3.5.2.1 Third Party Service Approach

The Trader capsule supports a number of service objects among which is the Trader object. See the interface listings in the orb2 for Eiffel Reference documentation. Access to the Trading interface is through an attribute called `trader`. This attribute can be acquired by inheriting from the class `CORBA_ORB_ACCESS`. Note that this is already the case within the class `DAIS_APPLICATION`. The Trading interface supports these IDL operations

- `import`
- `export`
- `withdraw`
- `unexport`

These are hidden behind a similarly named pair of features (with `u_` prefix), the first being a creation procedure on the `_REF` class and the rest on the `_IMPL` class.

Examples of these operations appear in the code sections in this chapter. The first operation, `import`, is for client usage, the remaining operations are for server usage.

The normal cycle of events is to have a Trader running, providing a distributed Trading domain. See Figure 3.5 and the orb2 Administrator’s Guide. In a static configuration the servers are started before the clients. The servers are launched first so that their objects, and hence the object references, are created ahead of any request for service by clients.
For each instance of a CORBA object (static, automatic or new) an instance of an object reference is created; if the server application wishes to advertise this object reference to other users then it may do so through the \texttt{u\_export} routine of the \_IMPL class, or the \texttt{idl\_export} routine of the \texttt{trader} attribute. The former merely ensures the IDL type naming convention is conformed to. See Chapter 4.

The newly created object that has now had its object reference registered with the Trader can expect to receive requests on its interface from clients who have requested access to a VehicleDB from the Trader.

If the object needs to expunge knowledge of its existence from the Trader, for example, during termination, then one of two operations may be invoked, \texttt{withdraw} or \texttt{unexport}.

Users should be aware that \texttt{unexport} is a \texttt{oneway} operation and is therefore appropriate for those situations in which it is not desirable to issue a synchronous request, where scheduling, awaiting a response, could compromise the capsule.

Once the server has used the \texttt{idl\_export} routine to make an object available, any client can acquire access to it through the \texttt{import} routine. Should the Trader be holding a number of object references to objects of the same type, then it will select one at random to return to import requests.

### 3.5.2.2 IDL Operation Return Value, out or inout Parameter Approach

Object references (or more strictly, the enclosing \_REF or \_IMPL objects) are like any other IDL data type and can be passed as \texttt{in}, \texttt{out}, \texttt{inout}, or returned values, during operation invocations.

### 3.5.2.3 Externalization Approach

It is possible to normalize object references to string format in such a way that they can be written to a simple text file in human readable format. It is possible to pass the normalized version as a simple IDL String type across an interface and reconstitute the object reference at the receiver.
This is not the norm. Object references do not need normalizing in this way for transport across interfaces. It is more likely that this technique is used where there are no alternatives for object location. For example, the bootstrap object reference for the Trader is represented in string format within the operating system environment of each orb2 capsule.

### 3.6 Invocation Call Types

The examples so far encountered, with the exception of Trading::unexport, have related to a simple synchronous call/response situation, in which the caller makes an invocation on an object and blocks, pending the result. The response can sometimes be an error. However, the call semantics remain constant, that is, synchronous.

It is possible to influence the nature of the call so that the caller either continues operating immediately after invoking the request and never expects a response, or the caller continues operating immediately after invoking the request and expects to collect the results at some later time (Deferred Synchronous).

The three invocation call types are shown in Figure 3.6.

In the synchronous approach, the client stubs effectively block the caller until results have been returned from the object, or an exception has been raised. This form of invocation is the default, requiring no special call syntax and requiring no special IDL directive.

In the one way approach, the client stubs issue, effectively, a datagram or oneway call and the caller application code proceeds without blocking; no response pending event is posted within the ORB and the caller logic does not expect or attempt to retrieve results. This type of call can only be made on operations that have been designated oneway within the IDL.

![Figure 3.6 Invocation Call Types](image-url)
For example

```eiffel
interface Statistics
{
  // We don’t care if the stats don’t get through
  // on the assumption that they will eventually
  // in some subsequent call
  oneway void update( in Metrics m );
};
```

Operations designated in this way must be treated with care because there is no remote failure detection.

**Note** Operations marked `oneway` must be of type `void`; they cannot contain any `out` or `inout` parameters or user exceptions within the signature. Failure to observe these rules results in an IDL compilation error.

In the deferred synchronous approach, the stubs effectively hand control back to the application as they proceed to deliver the request; it is the caller’s responsibility to collect the results at a later date. Should the results be returned before the caller attempts to collect them, they are buffered within the ORB, pending collection.

No qualification is required to IDL for deferred synchronous as this is an ORB implementation issue. Each `_REF` class provides additional routines for controlling deferred synchronous invocations

```eiffel
i_<operation_name> ( <in parameters> ) : DAIS_VOUCHER
<operation_name> ( <in parameters> ) : RESULT_TYPE
and the VOUCHER class supplies a routine
poll : BOOLEAN
```

Where the `i_` form initiates the request, the `c_` form collects the results and the `poll` routine indicates if results are ready to collect.

### 3.7 Exceptions

The `orb2` runtime system may at times raise `CORBA_SYSTEM_EXCEPTIONs` as a result of unexpected failure conditions. A CORBA object operation can also raise exceptions defined in the IDL. In a sense, these exceptions are part of the operation contract, and differ from the normal Eiffel situation where exceptions represent a deviation from the contract. It is this difference in semantics which justifies the case for such unrescued exceptions being passed back to the client caller without impacting the server capsule.

To raise an IDL defined exception it is merely necessary to create an instance of the corresponding exception class (populating its IDL members), and invoke its `raise` procedure.

In its rescue clause a client can gain access to the exception object which the server has raised through the `last_exception` feature.

**Note** If the exception that caused entry to the rescue clause was not a `CORBA_EXCEPTION` then `last_exception` will return `void`. This is an improvement on the standard defined in the mapping.

When a client uses the deferred synchronous calling technique, the exception is received when the client collects the results (it calls `c_<operation_name>`).
3.8 Memory Management

Memory management within a distributed environment is primarily concerned with getting information to the target object for manipulation and returning results to the caller. There should be no excessive copying, no excessive maintenance of relationships to track references and no memory leakage.

The Eiffel language mapping follows the usual Eiffel convention whereby memory management is, on the whole, too important to be left to the programmer. All objects created during marshaling and unmarshaling are subject to the usual Eiffel garbage collection philosophy. Where any such objects are holding on to intimate orb2 resources, explicit release and dispose routines are supplied, but implicit release on garbage collection is the normal behavior expected by the programmer.
Chapter 4  Trader Object Support Service

4.1 Introduction

The Trader is an independent orb2 application providing an object support service. The Trader application contains a single instance of six objects with the interface type names.

<table>
<thead>
<tr>
<th>Object Name</th>
<th>Interface Type Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trader</td>
<td>Primary interface for object location finding</td>
</tr>
<tr>
<td>TrCtx</td>
<td>Interface for manipulating contexts</td>
</tr>
<tr>
<td>TrType</td>
<td>Interface for manipulating types</td>
</tr>
<tr>
<td>TrFed</td>
<td>Interface for controlling federation</td>
</tr>
<tr>
<td>Reloc</td>
<td>An instance of a relocator</td>
</tr>
<tr>
<td>RelocDB</td>
<td>The relocation data base</td>
</tr>
</tbody>
</table>

These objects, see also Figure 4.1, are closely related through common data upon which they act. The Trader application can be modeled as a composition of objects, or more simply, a composite.
Applications normally only contact the trader object during object location; the other interfaces are primarily used by system administration specific clients like trman. The Trader acts as an intermediary between clients and servers, offering a registration scheme for servers and a lookup scheme for clients.

A client normally interacts with the Trader via a pseudo IDL object of interface type Trading. This is a pseudo object in that much of its behavior resides inside the client and is, strictly speaking, part of the orb2 runtime system. The DAIS_ACCESS class provides an orb2 attribute which has three features v of which points to one of these trading pseudo objects:

- Trader
- Master_trader
- Secondary_trader

Applications that expose their objects to client access do so by contacting a Trader to register the respective object references (the export operation). Once an object reference is registered, it constitutes a service offer and is advertised by the Trader.

Clients requiring a specific service may contact the Trader with a request for a service of this kind. The Trader searches its offer space for an appropriate offer. If one is found, the client is given the location of the object in the form of an object reference, so that the client can establish a connection; further dialogue is conducted directly between the client and the designated object. This matching of service offers to service requests is known as trading. The process, transparent to both client and server, of establishing a connection using an object reference, is known as binding.

The dating agency analogy aptly describes how the Trader works. One goes to a dating agency with a list of requirements for one’s ideal partner. The agency searches its records for suitable matches and comes up with one or more people who meet these requirements. You are then
provided with a telephone number that you can use to establish contact with the selected person; this is what happens in trading. The Trader provides a third party dynamic binding mechanism for clients accessing objects through their object references. It is dynamic because offers can be added to, or removed from the Trader at any time. The resulting binding takes place at run time.

4.2 Trader Organization

The Trader organizes its offers according to three criteria

- Interface type
- Context: the position of a type within the Trader’s administrative structure
- The properties of an object

To provide a suitable offer, the Trader searches through its contexts, starting from the point supplied by the client, to find any offers of the appropriate interface type. If there is more than one conformant offer, the Trader uses any client-supplied property constraints to determine the best possible match. This is a passive process, implemented through the client lookup (import) operation.

Both interface types and contexts must be established before any attempt is made by applications to use them, this is usually the user’s responsibility. Should the Trader be unaware of either the type or the context that an application is attempting to use, an operation exception will be raised.

4.2.1 Interface Type

The interface type system works on the principle of conformance. An interface is said to conform to another interface when it provides all the operations of that interface. For example, as shown in Figure 4.2, interface IF2 conforms to interface IF1 if it provides all the operations that IF1 provides and possibly more.

Conformance is achieved through inheritance. The IDL interface declaration statement supports an inheritance construct allowing one or more interface types to be listed against the interface that is being declared. The resulting interface definition is said to be derived from its base definitions. A derived interface may conform to more than one other interface, thus providing support for multiple inheritance.

The Trader holds the interface types for all the offers registered; offers are internally represented as a directed acyclic graph. The Trader refers to this graph when asked to provide a service by a client.
The importance of conformance is clear if you look at the implications of the inheritance tree in Figure 4.2, which shows that IF2, IF3 and IF4 conform to IF1. IF5 conforms to IF2, IF6 conforms to IF3, and IF7 conforms to IF3 and IF4. Inheritance is transitive, therefore IF5, IF6, and IF7 conform to IF1.

![Figure 4.2 Interface Conformance](image)

If a service of type IF1 is requested by a client, and IF1 is not available, the conformance structure means that all the interfaces in the above diagram would be contenders to take its place. If, however, an instance of IF3 was requested, but was unavailable, only IF6 and IF7 could be supplied in its place.

Simply declaring an inheritance relationship in IDL does not configure the Trader to establish the same type conformance. The Trader type conformance must be explicitly stated when adding type names to the Trader type space.

For example

```
prompt> typecl add IF1 object
prompt> typecl add IF3 IF1
prompt> typecl add IF4 IF1
prompt> typecl add IF7 IF3 IF4
```

The Trader is set up initially so that all types are compatible with a fundamental type Object, the root base definition. This fundamental type can be considered equivalent to ::CORBA::Object; all types are considered to be derived from this.

### 4.2.2 Context Space

The context space is the administrative structure of the Trader, it forms a hierarchy of names called contexts or name spaces. The structure is similar to the UNIX directory structure. See Figure 4.3.

The hierarchy forms a tree. The root of the tree is known as /.
A particular context name is defined by concatenating the contexts crossed leading to it and using / as a separator. The context names described in Figure 4.3 are:

```
/  
/Name1
/Name2
/Name3
/Name4
/Name3/Name5
/Name3/Name6
/Name3/Name5/Name7
```

Any application wishing to register an offer with a Trader must specify the context in which the offer is to be placed. The form for the `export` operation in the `_Impl` class is:

```
func u_export( context : STRING  
               properties : STRING )
```

The form for the `export` operation in the Trading object is:

```
IDL_export  

object : CORBA_OBJECT ;  --Object to be advertised  
          --typically Current

interface_type : STRING ;
context : STRING ;
properties : STRING )
```

For example:

```eiffel
class VEHICLEDB_IMPL  
inherit CORBA_OBJECT_IMPL  
feature  
make_and_export is  
do  
corra_create
```
Any client requesting a service from a trader must define the starting point for the set of contexts in which the Trader is to search. All contexts from that point down the subtree are searched, the general form of these for the _REF creation procedure is

```eiffel
import( context : STRING;
        constraints : STRING )
```

The form for the Trading import function is

```eiffel
import( interface_type : STRING;
        context : STRING;
        constraints : STRING ) : CORBA_OBJECT
```

For example

```eiffel
class NAME_DATA_BASE_CLIENT
feature
db : VEHICLEDB
make_and_import is
do
  !VEHICLEDB_REF!db.import(
    "/Name3",
    ""
  )
end
end
```

In this example the starting point of `/Name3` searches the contexts `/Name3`, `/Name3/Name5`, `/Name3/Name5/Name7` and `/Name3/Name6`. A starting point of `/Name3/Name6` searches only the context `/Name3/Name6`, alternatively a starting point of `/` searches the entire tree.

### 4.2.3 Properties

The search for a specific interface type, in a particular context, may yield more than one offer that could fulfill the request. To further refine the selection, each offer may have an associated set of properties against which the client can apply constraints.

Properties are associated with an offer when it is registered with the Trader. They take the form

```
Propertyname value
```

A Propertyname consists of alphanumeric characters, of which the first character must be a letter; value can be a number, a string or a set of strings. For example, one property associated with an offer might specify it as a particular bank

```
Bankname National
```

If the property has a set of strings associated with it, they must be enclosed within braces

```
AccountTypes {current shares deposit}
```

If more than one property is to be specified, the successive pairs are separated by a space

```
Bankname National AccountTypes (current shares deposit)
```

A client requesting service provision might anticipate a number of matching offers and refine the selection by providing the property values that the object must minimally support. This is achieved by constraining the selection using the property constraint language, supplied as the third parameter to the import request.
4.2.4 Property Constraint Language

A client requesting access to a Bank service might constrain the request as

Bankname==’National’

The property list in the example above would minimally satisfy this constraint. If selection of property values is acceptable, the logical and, or and not qualifiers can be used

Bankname==’National’ or Bankname==’First_Credit’

The constraint language is a sophisticated mechanism for enabling clients to fine tune their service requirements.

- Single quotes are used to surround string values in the import statement, see Section 4.3.1.
- Quotes are not always needed in the export statement.

The Trader assumes that a property value it receives is a string unless it starts with a number, in which case it is taken as numeric. If you want to force an export property value to be interpreted as a string, surround it with single quotes.

For example

idl_export( "/", "Identifier ’123’" )
idl_export( "/", "Identifier 123" )

where the Trader will treat the first property list as the name Identifier having a string value "123" and the second property list as the name Identifier, having a numerical value 123.

Clients requesting such services could employ the following selection criteria

import( "/", "Identifier==’123’" )
import( "/", "Identifier==123" )
import( "/", "Identifier==’123’ or Identifier==123")

This results in the first import matching only the first export and the second import matching the second export, while the third import will match both exports and the Trader will randomly return one of the offers.

4.2.3.1 Standard Properties

There are a number of standard property name-value pairs that orb2 supplies which can optionally be applied to offers. These are the host name of the system on which the server is running, the identity of the process containing the server and, on UNIX systems only, the userid of the process containing the server. A list of these properties, suitable for export, can be prefixed to the export property list. If the property list begins with "+" (plus followed by a space) then the Trader export method replaces the plus with the standard properties. Note that if only the standard properties are required, the property list must still contain plus followed by a space.

For example

idl_export( "/", "+ Identifier ’123’" )
idl_export( "/", "+ " )

4.2.4 Property Constraint Language

The property constraint language is a sophisticated method of constraining the number of offers selected by the Trader to satisfy a client import. A full definition can be found in Appendix B. However, the constraint language’s main purpose is to allow clients to tighten the specification so that exactly the right sort of offer is returned.
The property constraint language takes the form of arguments on the import statement and is therefore the client’s responsibility. It is made up of expressions acting upon the property names and values of the prospective offers. The expressions that can be used are

- Superlative functions: \texttt{min, max}
- Comparative functions: \\texttt{==, \!\!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!, \!...
creation procedure with the same name on the _REF class. The Trading routines export, withdraw and unexport on the attribute trader are also simplified by the same named routines on the _IMPL class.

### 4.3.1 Trading

To illustrate how constraints influence the results of Trading::import() consider the context space populated by the offers shown in Figure 4.4.

![Figure 4.4 Offers in the Context Space](image)

Several queries are shown below with the possible responses from the Trader::import() invocation:

- **Type = "StringOps", Context = "/", Constraint = ""**
  The Trader searches through the entire tree anchored at / for instances compatible with StringOps; no additional constraints have been supplied; the Trader will return one of the three matching offers by selecting one at random. It is important to note that only two of these offers are of type StringOps, the third is of type MoreStringOps, which is acceptable due to the type conformance principles set out in Section 4.2.1.

- **Type = "StringOps", Context = "/", Constraint = "Foo > 2"**
  The Trader searches through the entire tree anchored at / for instances compatible with StringOps and with a value of the Foo property greater than 2. The Trader will return the offer found in /production.

- **Type = "StringOps", Context = "/", Constraint = "Type == ‘StringOps’"**
  The first parameter to an export operation is the TypeName and is automatically converted to a name-value property on the offer of the form Type Typename. The Trader searches through the entire tree anchored at / for instances of StringOps; the Constraint expression eliminates the use of any compatible types. Of the two possible offers found, one will be returned at random.
4.3.2 Context

The Trader context space and type space are managed through two additional objects, instances of which are created as part of every Trader capsule. These are instances of the interface types TrCtx and TrTy, for which the user can find IDL definitions in the orb2 for Eiffel Reference documentation.

Applications should not normally need to invoke any of these operations directly as all of their functionality has already been incorporated into ready built clients such as trman, typecl and ctxtcl. However applications which automatically manage the context and type name spaces will need to access them dynamically.

For example

```eiffel
class TRADER_MANAGEMENT
feature
  cxman : TRCTX
  tpman : TRTYPE
make is
  do
    !TRTX_REF!cxman.import("TrCtx","\="/\",""")
    !TRTYPE_REF!tpman.import("TrType","\="/\",""")
  end
add_context ( cx_name : STRING ) is
  do
    cxman.add_name ( cx_name )
  end
add_type ( type_name : STRING ) is
  do
    tpman.add_type ( type_name )
  end
end
```

- Type = "StringOps", Context = "/", Constraint = "Type = 'StringOps' and Foo < 2"
  The Trader searches through the entire tree anchored at / for instances of exactly type StringOps and with a value of Foo less than 2. The Trader will return the offer found at /dais/services.

- Type = "StringOps", Context = "/production", Constraint = ""
  The Trader searches through the entire tree anchored at /production for instances compatible with type StringOps. As no further constraints have been supplied the offer at that location is returned.

- Type = "StringOps", Context = "/master", Constraint = ""
  The Trader searches through the entire tree anchored at /master for instances compatible with type StringOps. No offers are found and a user exception having the identity Trading::NoMatchOffers will be returned.

- Type = "StringOps", Context = "/", Constraint = "Foo > 3"
  The Trader searches through the entire tree anchored at / for instances compatible with type StringOps and with a value of the Foo property greater than 3. No offers are found and a user exception having the identity Trading::NoMatchOffers will be returned.

- Type = "StringOps", Context = "/", Constraint = "->min[Foo]"
  The Trader searches through the entire tree anchored at / for instances compatible with type StringOps. After locating all the offers the Trader will return the one having the minimum value of Foo, in this case, the offer at /dais/services.

- Type = "MoreStringOps", Context = "/", Constraint = ""
  The Trader searches through the entire tree anchored at / for instances compatible with type MoreStringOps. There are no further constraints and the only available offer at /dais/test is returned.
4.3.3 Federation

orb2 supports two distinct federation concepts

- Context Federation
- Proxy Federation

4.3.3.1 Context Federation

It is usual for several Traders to exist supporting different user communities. A Trader may wish to make its context space, and the offers it contains, accessible to users of another Trader. This limited accessibility is achieved by binding one Trader’s context space into a context name in another Trader’s context space.

For users familiar with NFS, a useful analogy is to think of mounting the remote Trader’s context space at a local context name. If the context name bound to the remote Trader is contained within an import context then the local Trader may perform a nested invocation on the remote Trader to satisfy the request. Traders adopt a shallow search policy; only if there are no local requests satisfying the query will a nested invocation be made to any bound context Traders.

This process is illustrated in Figure 4.5.
Figure 4.5 Default Local-Master Federation
Now reassess some of the failed import invocations from the single Trader example in Section 4.3.1, in particular

- **Type = "StringOps", Context = "/master", Constraint = ""**
  
  In this import, a nested invocation to the Trader bound at /master, will be made for instances of type StringOps. One of the offers contained in /master/production, /master/dais/services or /master/dais/test will be returned.

- **Type = "StringOps", Context = "/", Constraint = "Foo > 3"**
  
  In this import, no matching offers of type StringOps with Foo greater than 3 are found in the local Trader. Since the subtree specified in the query "/" contains the bound context /master, the query is forwarded to the federated Trader. The remote Trader will return the offer in /production.

orb2 supports a default context federation that is enabled through the configuration utility. If, during the configuration phase, a remote Trader is nominated as the Master Trader then the Local Trader automatically places a binding to the Master Trader into its /master context during its initialization.

As the Master Trader can perform the role of a Local Trader within its own domain, it is clearly possible to define multiple federation links across configuration domains.

For example, orb2 is running on a network of three hosts, X, Y and Z.

1. **X::configuration** specifies X as the local Trader and Y as the Master Trader
2. **Y::configuration** specifies Y as the local Trader and Z as the Master Trader
3. **Z::configuration** specifies Z as the local Trader and itself (or nothing) as the Master Trader

As a result of this federation

1. Clients within the X domain will have access to services at X, Y and Z
2. Clients within the Y domain will have access to services at Y and Z
3. Clients within the Z domain will only have access to local services

### 4.3.3.2 Proxy Federation

Another form of federation may be achieved through the use of proxy offers. A proxy offer is an object reference that is associated in the Trader with a particular TypeName (for example, Calculator) but is, in fact, an object reference to another remote Trader, an object supporting the lookup operation. This constitutes a delegation, where the interface supporting the proxy must conform to a Trader interface, but behind which a service specific implementation is provided. Whenever the actual Trader encounters a proxy offer, it effectively delegates responsibility for providing a service offer to the service identified by the proxy. The proxy service itself presents a Trader-conformant method for the import signature but may choose any form of object location or manufacture. This provides a powerful hook for third-party user mechanisms.

When an import operation resolves to a proxy offer that lookup is federated to the object designated by the proxy. In this way, a client request for a service can be intercepted and some special processing can be performed before returning the true object reference required by the client. This is the principal mechanism used by the Node Manager for dynamically instantiating capsules.

Both federation techniques are managed through a single Federation interface TrFed.
Using the example shown in Figure 4.6 it is possible to construct a resource scheduler that masquerades as one or more peripherals through proxy offers, in this way, load balancing of a set of peripherals could be achieved seamlessly.

For example

- Resource Scheduler (RS) exports its B object to the Trader. Any peripherals coming online would import the RS and then perform local registration of their existence through object B. This is a dynamic system in which peripherals could come and go provided that RS was informed.
- On detection of the first peripheral registration, RS posts a proxy offer to match that peripheral type through \texttt{trfed}. (Only one peripheral offer of a type is required for all instances of peripherals of that type.) The A object constitutes the masquerading offer and provides a many-faceted implementation of the operation \texttt{import}.
- Any clients wanting access to specific peripherals trades for them through an \texttt{import} call. If the trader lookup resolved to the RS proxy, then it would be federated to A.
- A can now inspect its internal database (maintained by B) and pass back the least used peripheral matching the type requested. Applications containing more than one object, wanting access to common state, should use the ECS detailed in Chapter 6 for controlled access.
- The client receives the peripheral’s object reference and proceeds to use it directly.
4.4  **Trader’s Use of the Relocation Architecture**

The Trader capsule in orb2 supports an instance of a relocation object. See Section 5.5 for details on relocator construction. This is not a real relocation service, in the sense that it can be used to map old object references to new; its purpose is to enable the Trader to intercept client request failures and by doing so, purge potentially stale offers.

The Trader implements this scheme as follows:

1. Every object reference registered through `Trader::export()` will have the Trader’s relocation object reference prefixing its list of relocators. In this way the first relocator that gets called on client failure is the Trader’s relocator.

2. The implementation of this Trader’s relocator simply causes the Trader to `ping()` the target object to confirm its demise and, if that is confirmed, the Trader removes the associated offer from its database.

3. The Trader’s relocator asynchronously returns an exception to the client which then proceeds to consult any other relocators.

4.5  **Programmatic Interface to Trading**

A client normally interacts with the trading facilities through the packaged `u_import` and `u_export` routines on the `_REF` and `_IMPL` classes. These in turn access the local trading facilities. There may be situations (particularly in management applications) where the client needs to access the master trader, or the secondary trader directly, or may need to supply the full parameter set to the local Trading operations. The `DAIS_ACCESS` class makes available three attributes which all offer the Trading interface: `trader`, `master_trader`, and `secondary_trader`, for these more unusual situations.

4.5.1  **Secondary Trader**

Secondary Traders are very much the preserve of orb2 Systems Management and configuration. However, they are mentioned here briefly because of their potential impact on the programmatic interface.

Although it is possible to manipulate the secondary trader directly, it is standard practice to allow the local Trader do this. The local trader always examines to see if a Secondary Trader has been nominated. If a Secondary Trader is detected, all access by applications to Trader object operations are intercepted:

- Client side access, through `import`, is first tried on the local Trader. If unsuccessful an attempt is made to locate the same service using the Secondary Trader. If both accesses fail, the application is notified with an exception.

- Server side access through `export`, `withdraw` and `unexport` is mirrored, that is, both local and Secondary Traders receive the same invocation so their internal databases are kept in step.

An interesting feature of Secondary Traders is that they themselves could be local Traders in their own right. They may contain contexts, types and federations not reflected in the local Trader to which they are secondary. However, they must minimally contain the set of contexts and types represented within the local Trader.
4.5.2 Trader Member Functions

The Trading interface of the trader attribute defines four high level operations

- import
- idl_export
- withdraw
- unexport -- oneway

Trading inherits from Trader which is a proper remote object (rather than a pseudo-object). You can use the lower level operations of the Trader interface on the trader attribute in spite of this strange inheritance relationship. What actually happens is that the trading pseudo-object delegates the inherited operations to the real remote Trader. These inherited operations do not, however, get the benefit of the transparent access to the Secondary Trader. These lower level operations are

- register
- lookup
- delete
- try_delete -- oneway

The register operation is directly analogous to idl_export.

The operations delete and try_delete are analogous to the high level withdraw and unexport operations respectively, with the added capability of refining deletion of the offers within the Trader by supplying constraints. The constraint language employed is identical to that employed by clients during the import operation.

The lookup operation is analogous to the high level import operation with the added capability to provide a lookup policy which may take any of these values

- lookup_random
- lookup_all
- lookup_proxy_all

lookup_random, the policy adopted by the high level import language, will cause the Trader to return exactly one offer from a set of matching offers.

lookup_all causes the Trader to return all matching offers.
Chapter 5

Application Development Using Eiffel: orb2

5.1 Introduction

In addition to providing the functionality required by the CORBA 2.0 Architecture and Specification, orb2 offers added functionality, such as Object Support Services like the Trader. CORBA also defines the Basic Object Adapter, the principal interface through which object implementations access ORB functions. This concept has been extended to the orb2 Extended Object Adapter.

This chapter describes the features offered by the orb2 Extended Object Adapter, together with the orb2 class library and extensions to the ORB. To get the most out of orb2, users, at some stage, need to use some or all of the facilities detailed here.

5.2 Concurrency and Multi threading

orb2 is a multi threading ORB. That is, capsules that are operating as clients only, or servers only or both, can invoke orb2 or be invoked by orb2 in concurrent threads.

orb2 supports threads in two ways.

- On those operating system platforms that provide primitive preemptive threading facilities, orb2 uses those facilities, and the application can also use those facilities independently of orb2.

- On platforms that do not provide such features, orb2 offers its own thread facilities

The Eiffel/orb2 binding is currently offered only on platforms that provide native preemptive facilities. The applications use the thread and mutex facilities provided by the Eiffel language thread library. The orb2 thread facilities are not explicitly made visible to the application, although for the server invocations, the orb2 threading facilities are employed implicitly.

The threading model supplied by the ISE Eiffel development system is such that all objects are "owned" by the thread that created them. They cannot exist beyond the lifetime of that thread. Objects can be accessed from other threads, but such invocations cannot manifest factory-style behavior (They cannot create new objects whose existence lasts beyond the invocation.). Because of these constraints, orb2 ensures that all CORBA invocations on Eiffel CORBA objects take place in the thread that owns that CORBA object. This model differs from the behavior of orb2 in most of its other language bindings. In those, each invocation is taken on a separate thread which is created (typically re-used) just for that invocation.

There are some consequences of this Eiffel invocation style which are worth exploring briefly. On the face of it, this style appears to be safer than the normal orb2 behavior, requiring no mutexing in the called object, since it appears it can only be called in its owning thread once at a time. If a server invocation in turn makes a synchronous CORBA call, then one possible implementation would have been to block the thread until the response has been received, and the original call has exited. Such a style would be prone to deadlock if the called object made a call back into the same or another object in the same thread as the original. It is felt that such deadlocks could not be easily avoided by the application programmer. Consequently, while a thread is awaiting a response, it is available to receive other calls. This allows the
legitimate callback described above. However, it also leaves the thread open to calls from completely unrelated execution requests. It is the application’s responsibility to protect itself from the consequences of this.

It is also worth noting that, if the server object which is awaiting a response is in a Windows GUI thread, then while it is waiting, there is a nested Windows message loop in operation, which will continue to dispatch messages to windows in the same thread.

5.3 Extended Objects and Managed Objects

An extended object is a derived type of a standard CORBA object, that is, developers can do all the things to an extended object that can be done to a CORBA object plus added capability. The extended object functionality is used in orb2-supplied interfaces (such as Node Manager and Trader) but is of no consequence to Eiffel objects. Where an Eiffel object needs to be a client of any of the orb2 interfaces, the extended object IDL is included in the generated code.

Note the containment hierarchy of managed objects and application objects supplied in the C and C++ mappings is not offered in the Eiffel mapping. It is considered a somewhat naive attempt to impose a discipline of composition which is not appropriate to the Eiffel language with its more sophisticated client relationships and garbage collection mechanisms for maintaining referential integrity.

5.4 Capsules and Managed Objects

Those familiar with the C and C++ orb2 offerings will be aware of a containment hierarchy of capsule, managed objects and application objects. The first two of these, the Capsule and the ManagedObject, are IDL objects implemented in the orb2 runtime system, and cannot be inherited from in the same way as other IDL-based objects. The use made of these in the Eiffel mapping is limited to enabling the automatic creation and termination of capsules through the facilities of the orb2 NodeManager and Factory services.

When any CORBA object is created, it can be registered with the Trader for clients to find. There is a technique in the Trader whereby an agent can offer an interface, and when the corresponding import request is made, the agent creates and returns the object supporting the interface. The agent is often referred to as a factory object, and the offer in the Trader is a proxy offer. When an Eiffel application is running it is quite likely to offer factory-style behavior, and can use the TrFed proxy mechanism as one means to achieve this.

orb2 also offers a standard service for starting capsules as a result of proxy offers being taken up. This is called Capsule Factory and is described in Chapter 6.

When an importer causes such a capsule to be automatically started, a two-stage operation is initiated

1. The DAIS::Factory instantiates the capsule and a hidden managed object. The latter merely prevents a console style application from prematurely terminating when it gets to the call of the dais.scheduler.schedule.
2. Once the capsule is up and running, the Factory asks it for a top level managed object and at least one application object.

The Capsule and ManagedObject are part of the orb2 runtime system and are of limited concern to the Eiffel application. The application object, however, is created under the control of the application. An Eiffel application that must be started in this way must call make_dais_application with the managed argument set to True, and must redefine the get_traded_object feature to return a genuine application-created CORBA object. See Chapter 6 for more information.

```eiffel
Class VEHICLEDB_APPLICATION
inherit
    DAIS_APPLICATION
```
5.5 Object Relocation

For systems requiring high dependability the possibility of failure must always be a consideration.

5.5.1 Overview

An operation invocation on a remote object may fail for one reason or another. Such failures may arise because of network, hardware or software failures. In these circumstances, services that appear to be available (for example, as registered in the Trader) may no longer be able to be contacted. Another reason might be that a service that does not rely on the Trader, has been deliberately shut down and started elsewhere. Under these conditions, clients that have made successful invocations on objects may find that subsequent invocations fail.

It could be beneficial at such times for an object reference to carry within itself knowledge of an alternative service, so that a connection could be made to that object. Such a connection should be transparent to the client, which should be capable of invoking operations on the object without the knowledge that the original object has relocated. This ability to relocate services transparently should be as general as possible, permitting a wide range of potential relocation strategies to be designed.

orb2 provides support for such a strategy as all orb2 systems have the capability to support a Relocator service, which is an implementation of a Relocator interface. A Relocator object supports operations that manage and maintain a look-up table comprising the original object reference and its replacement.

All object references have the capability to support one or more Relocator object references, these references are embedded in the original object reference in such a way that they can be accessed by a local operation on the primary object reference.
The ability to transparently redirect references to services is provided automatically in the client stubs for synchronous calls. If the application is using deferred synchronous calls, the programmer can still request the relocation using features on CORBA_SYSTEM_EXCEPTION, CORBA_OBJECT and CORBA_OBJECT_REF.

In Figure 5.1 the first embedded object reference would be a Relocator object reference. Such Relocator references are embedded Last In First Out (LIFO) basis, that is, the latest is at the top of the list.

The features on CORBA_SYSTEM_EXCEPTION, CORBA_OBJECT and CORBA_OBJECT_REF that enable object reference manipulation are:

- `u_add_relocator(CORBA_OBJECT)` through which a relocator object reference can be embedded within another object reference.
- `u_extract_relocator(CORBA_OBJECT)` through which a relocator object reference can be obtained from within an existing object reference.
- `u_clean_of_relocators(CORBA_OBJECT_REF)` returns an object reference with only the primary reference remaining.
- `last_exception(CORBA_GENERAL)` through which the last exception is made available (as long as it was a CORBA exception).
- `u_worth_attempting_relocation(CORBA_SYSTEM_EXCEPTION)` indicates if the reason for failure was one of the CORBA_SYSTEM_EXCEPTIONs which might be due to loss of server availability.
- `u_attempt_relocation(CORBA_OBJECT_REF)` through which orb2 invokes the embedded relocators in turn, and tries to find a replacement server to replace the original. If successful, the reference is now pointing at the new server, and the feature returns TRUE. Otherwise, it returns FALSE.

### 5.5.2 The Trader’s Use of the Relocator Interface

The Trader implements a special instance of a Relocator that is used to implement a stale offer purge whenever a client invocation fails on an advertised service.

Whenever an offer is exported to the Trader, the Trader performs an implicit `u_add_relocator` embedding a Relocator object representing its own implementation for the Relocator interface. This is always placed at the top of the list of embedded Relocator objects.

When a client invocation fails the stub code:

- Invokes each embedded Relocator in turn; the Trader’s implementation is always the first. The Trader consequently receives a Relocator::lookup request and deduces from this that the service is no longer available.
- It tries to ping the capsule in question and, if this confirms the failure, it then purges the offer from its database so that later clients do not fall into the same trap.
5.5.3 Relocation Service Example

Central to the realization of all of the above is the availability of one or more capsules that embody the actual Relocation Service implementation, this is the service that provides the mapping of object reference to their relocated replacement.

Unlike other orb2 services, such as Trader or Factory, the Relocation service is not provided as standard service. Instead, an example (relocation in the examples directory) is supplied which users can modify to suit their own requirements.

The example contains default implementations for all the operations in the Relocator interface (see Reloc.idl in orb2 for Eiffel Reference documentation). You may provide your own implementations if you wish to override the defaults. In particular, the example does not include any persistent information store.

- register This operation associates a backup server object reference with a primary server object reference.
- de_register This operation removes all knowledge of a particular object reference from the Relocation Service.
- lookup This operation obtains the backup (replacement) object reference for the primary object reference.

5.5.4 Resilience Improvements: Example

There are many ways to use this architecture to improve resilience. Figure 5.2 illustrates one example.

The purpose of the model is to demonstrate the features of the relocator architecture. It is not a solution for all resilience problems.

Figure 5.2 shows a simple example of relocation in which one server is the primary. A second server registers itself with a Relocator service as the backup server. The client makes a call on ServiceA (primary), as obtained from the Trader. Should this, or a subsequent call fail, the call is relocated transparently to ServiceA (backup) and processed without the client being aware.
The steps involved in relocation for this example are
1. Initialize the Relocator process. The Relocator Service exports itself to the Trader.
2. Initialize the Primary Server capsule. The Primary Server imports the Relocator object reference and extracts the object reference (extract_relocator). This is then embedded within its own object reference (add_relocator) and exported to the Trader; see Figure 5.3.

![Figure 5.3 Object References (1)]

3. Initialize the backup server process. The Backup Server registers its object reference with the Relocator Service (Relocator::register) as a possible replacement for the Primary Server.
4. The client imports the server from the Trader, and is given the Primary Server object reference; see Figure 5.4.

![Figure 5.4 Object References (2)]

5. The client makes invocations on the Primary Server. During an invocation, a failure is detected by the client.
6. The client, by means of the client stub code generated by the CORBA compiler, transparently invokes relocate_if_possible. This in turn obtains the Relocator object reference (extract_relocator). The client now invokes a lookup operation on the Relocator. If successful, the operation returns the replacement server object reference, the Backup Server, and overwrites the Primary Server object reference; see Figure 5.5.

![Figure 5.5 Object References (3)]

7. The client transparently retries the same invocation on the Backup Server and, if successful, continues as if nothing had happened.

More than one Relocator may be embedded within the object reference. Should one Relocator be unsuccessful in finding a replacement, the next Relocator is tried.
The object reference obtained from the Trader will already have the Trader Relocator service embedded within it; see Figure 5.6.

![Figure 5.6 Object References (4) Primary object reference Trader Relocator object reference](image)

The Trader uses its own private instance of Relocation service to manage the purging of stale offers. These are offers that remain registered with the Trader when the object they represent is no longer available.

In this situation the client stub first contacts the Trader’s instance of a Relocator service. The Trader can recognize that such an offer is stale and take the necessary purging action on its database of offers.

Because of the special nature and purpose of the Trader’s instance of a Relocator, users cannot use this to implement their own Relocation service.
6.1 Introduction

orb2 provides real time support for the distributed applications in which CORBA objects exist through concepts such as Capsules, Trading, Object References and Factories. Users can combine all these concepts together to provide strategies for dynamic creation of CORBA objects, or, more simply, Objects on Demand. Applications typically contain Factories to manufacture fine grain objects, but another important element is the ability to dynamically create application capsules.

This chapter describes in detail at how capsules are launched on demand.

6.2 Capsule Factory

The term Factory denotes an entity capable of producing or creating something. There are many types of factory, each creating entities of a specific type, and orb2 uses this concept for creating software entities in the form of executables and objects.

You will have used and developed Factories (perhaps unwittingly). Applications creating objects are object factories. This chapter applies this concept to the explicit creation of the applications themselves.

Capsule Factories are orb2 capsules that provide highly specific services for starting, stopping and terminating capsules. Capsule Factories do not themselves utilize any distribution techniques, nor do they attempt the remote initialization of capsules using remote execution commands, such as the UNIX rsh command, as these are far too operating-system specific. Capsule Factories are the most operating-system familiar components of orb2 and must be co-resident, on the same node, with the capsules that they control.

Capsule Factories present an object interface to the outside world and are therefore part of the orb2 Object Support Services. Factories operate within Trader domains. See Figure 6.1.
Factories may span a number of nodes. Only one factory per node, per domain, is allowed, and the factory units protect against infraction of this rule.

The factory interfaces are rarely used directly within an application. Rather, the orb2 Trader, NodeManager and Factory services launch application capsules automatically as a result of clients importing proxy offers from the Trader. The interfaces between these three services are, however, freely available via IDL and so can be used directly for specialized purposes. A description of the process follows.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The standard orb2 Trader is started.</td>
<td>Every orb2 system requires a Trader, as this is the initial object reference available in all capsules.</td>
</tr>
<tr>
<td>2</td>
<td>A pair of orb2 Services (Factory and NodeManager) are started.</td>
<td>A node Manager/Factory pair are required on every hardware node which will dynamically create capsules.</td>
</tr>
<tr>
<td>3</td>
<td>The Node Manager lodges a proxy offer with the Trader for a nominated Interface together with some other offer properties.</td>
<td>This is as a result of either using the nmclient install and postproxy commands or direct use of the equivalent Node Manager IDL operations. This proxy offer remains in the Node Manager’s persistent data until removed. The arguments to the install command include a. The alias string: the name by which the Node Manager knows the offer. b. The interface name, context and properties are used to register the offer with the Trader.</td>
</tr>
<tr>
<td>4</td>
<td>A client imports that Interface from the Trader.</td>
<td>The Interface imported must be the same as that registered by the Node Manager.</td>
</tr>
<tr>
<td>5</td>
<td>The Trader federates the offer to the Node Manager.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>The Node Manager asks the Factory to start up the executable (using Factory::Instantiate).</td>
<td>The pathname of the executable is another of the arguments to the install command. Before returning to the Node Manager, the Factory attempts a rendezvous with the started capsule to ensure subsequent actions can be carried out.</td>
</tr>
<tr>
<td>7</td>
<td>The Factory returns the capsule object reference of the started executable to the Node Manager.</td>
<td></td>
</tr>
</tbody>
</table>
The Node Manager asks the capsule for the object reference that is to be returned to the original importer (using `Capsule::Instantiate`).

The result of this call is that the feature `get_traded_object` is called. This gets passed two strings from the original `install` command (the `argv` and `argc` arguments). These can be used to influence which object is actually returned.

There are two conditions that must be true before the `get_traded_object` feature is called:

a. The application must have declared itself as a "managed" application on the call to `orb_init`.

b. The Template Name (sometimes known as Object Name) supplied to the `nmclient install` command must match the template name which the application supplies to `orb_init`.

Otherwise the `get_traded_object` feature is not called and the client will not get the required Object Reference.

That Object Reference is returned via the trader to the client.

If the server object exports itself with the same interface, context and properties as the original proxy offer, there will now be two offers in the Trader, and subsequent imports will randomly select the proxied and the real offer.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
</table>
| 8    | The Node Manager asks the capsule for the object reference that is to be returned to the original importer (using `Capsule::Instantiate`). | The result of this call is that the feature `get_traded_object` is called. This gets passed two strings from the original `install` command (the `argv` and `argc` arguments). These can be used to influence which object is actually returned. There are two conditions that must be true before the `get_traded_object` feature is called:
   a. The application must have declared itself as a "managed" application on the call to `orb_init`.
   b. The Template Name (sometimes known as Object Name) supplied to the `nmclient install` command must match the template name which the application supplies to `orb_init`. Otherwise the `get_traded_object` feature is not called and the client will not get the required Object Reference. |
| 9    | That Object Reference is returned via the trader to the client. | If the server object exports itself with the same interface, context and properties as the original proxy offer, there will now be two offers in the Trader, and subsequent imports will randomly select the proxied and the real offer. |
Appendix A

IDL Syntax and Semantics

This appendix defines the Interface Definition Language (IDL) used to specify the interfaces in the orb2 implementation. Applications are not written in IDL but in a programming language, such as Eiffel, and operations on defined interfaces are invoked by function calls in the application code. See Chapter 2, and Chapter 3 for descriptions of the use of the IDL.

The current IDL conforms to the CORBA specification as detailed in the OMG document number 93.12.43, Revision 1.2. Not all preprocessing syntax, as defined in that document, will apply. Developers should only expect preprocessing to conform to descriptions in this manual.

IDL obeys the same lexical rules as C++ with additional keywords to support distribution concepts. It also provides full support for standard C++ preprocessing features.

The IDL grammar is a subset of ANSI C++ with additional constructs to support the mechanism for operation invocation. IDL is a declarative language supporting C++ syntax for constant, type, and operation declarations, but it does not include any algorithmic structures or variables.

A source file containing interface specifications written in IDL must have the filename extension .idl.

The description of IDL grammar uses a syntax notation that is similar to Extended Backus-Naur Form (EBNF). The symbols used in this format and their meaning are shown in Table A.1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
<td>Is defined to be</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;text&gt;</td>
<td>Non-terminal</td>
</tr>
<tr>
<td>&quot;text&quot;</td>
<td>Literal</td>
</tr>
<tr>
<td>*</td>
<td>The preceding syntactic unit can be repeated zero or more times.</td>
</tr>
<tr>
<td>+</td>
<td>The preceding syntactic unit can be repeated one or more times.</td>
</tr>
<tr>
<td>{}</td>
<td>The enclosed syntactic units are grouped as a single syntactic unit.</td>
</tr>
<tr>
<td>[]</td>
<td>The enclosed syntactic unit is optional; it may occur zero or one time.</td>
</tr>
</tbody>
</table>
A.1 Lexical Conventions

This section presents the lexical conventions of IDL. It defines tokens in an IDL specification and describes comments, identifiers, keywords, and literals, integer, character, and floating point constants and string literals.

An IDL specification logically consists of one or more files. A file is conceptually translated in several phases. The first phase is preprocessing, which performs file inclusion and macro substitution. Preprocessing is controlled by directives introduced by lines having # as the first character other than white space. The result of preprocessing is a sequence of tokens. Such a sequence of tokens is called a translation unit.

A.1.1 Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators and other separators. Blanks, horizontal tabs and comments (collective, white space), as described below, are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords and constants.

A.1.2 Comments

The characters // start a comment, which terminates at the end of the line on which they occur. C-style comments are also supported.

A.1.3 Identifiers

An identifier is an arbitrarily long sequence of alphabetic, digit and underscore characters. The first character must be an alphabetic character. All characters are significant.

Identifiers that differ only in case collide and yield a compilation error. An identifier for a definition must be spelled consistently (with respect to case) throughout a specification.

When comparing two identifiers to see if they collide

- Uppercase and lowercase letters are treated as the same letter.
- The comparison does not take into account equivalences between digraphs and pairs of letters (for example "ae" and "æ" are not considered equivalent) or equivalences between accented and non-accented letters (for example "Å" and "Å" are not considered equivalent).
- All characters are significant.

There is only one name space for IDL identifiers. Using the same identifier for a constant and an interface, for example, produces a compilation error.

A.1.4 Keywords

The identifiers in Table A.2 are reserved for use as keywords, and must not be used otherwise.

<table>
<thead>
<tr>
<th>Table A.2 Reserved Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
</tr>
<tr>
<td>default</td>
</tr>
<tr>
<td>interface</td>
</tr>
<tr>
<td>readonly</td>
</tr>
<tr>
<td>unsigned</td>
</tr>
<tr>
<td>attribute</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>long</td>
</tr>
<tr>
<td>sequence</td>
</tr>
<tr>
<td>union</td>
</tr>
<tr>
<td>boolean</td>
</tr>
<tr>
<td>enum</td>
</tr>
<tr>
<td>module</td>
</tr>
<tr>
<td>short</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>case</td>
</tr>
<tr>
<td>exception</td>
</tr>
<tr>
<td>octet</td>
</tr>
<tr>
<td>string</td>
</tr>
<tr>
<td>FALSE</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>oneway</td>
</tr>
<tr>
<td>struct</td>
</tr>
<tr>
<td>Object</td>
</tr>
<tr>
<td>const</td>
</tr>
<tr>
<td>in</td>
</tr>
<tr>
<td>out</td>
</tr>
<tr>
<td>switch</td>
</tr>
<tr>
<td>TRUE</td>
</tr>
<tr>
<td>context</td>
</tr>
<tr>
<td>inout</td>
</tr>
<tr>
<td>raises</td>
</tr>
<tr>
<td>typedef</td>
</tr>
</tbody>
</table>
Keywords obey the rules for identifiers and must be written exactly as shown in the above list. For example, `boolean` is correct; `Boolean` produces a compilation error. IDL specifications use the following characters as punctuation.

```
; { } : , = + - ( ) < > [ ]
\" \ | \^ \* / \%
```

**Note** The punctuation characters `< > { } [ ]` are also used in the EBNF syntax notation. In the following pages, these characters are shown as literals in double quotes, that is "<" ">" "{" "} "[" and "]" where they are used to represent IDL syntax.

In addition, these tokens are used by the preprocessor

```
# ## ! || &&
```

### A.1.5 Literals

In the lexical conventions, several different forms of literal representation are used, integer, character, floating-point and string. The definition of these literal types are contained in the following sections.

#### A.1.5.1 Integer Literals

An integer literal consisting of a sequence of digits is taken to be decimal (base ten) unless it begins with 0 (digit zero).

A sequence of digits starting with 0 is taken to be an octal integer (base eight). The digits 8 and 9 are not octal digits.

A sequence of digits preceded by `0x` or `0X` is taken to be a hexadecimal integer (base sixteen). The hexadecimal digits include `a` or `A` through `f` or `F`, with decimal values ten through fifteen, respectively.

For example, the number twelve can be written `12`, `014`, or `0XC`.

#### A.1.5.2 Character Literals

A character literal is one or more characters enclosed in single quotes, as in `'x'`. Character literals have type `char`.

A character is an 8-bit quantity with a numerical value between 0 and 255 (decimal). The value of a space, alphabetic, digit or graphic character literal is the numerical value of the character as defined in the ISO Latin-1 (8859.1) character set standard. The value of a null is 0. The value of a formatting character literal is the numerical value of the character as defined in the ISO 646 standard. The meaning of all other characters is implementation-dependent.

Non-graphic characters must be represented using the escape sequences defined in Table A.3. Escape sequences must be used to represent single quote and backslash characters in character literals.

<table>
<thead>
<tr>
<th>Description</th>
<th>Escape Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>newline</td>
<td>\n</td>
</tr>
<tr>
<td>horizontal tab</td>
<td>\t</td>
</tr>
<tr>
<td>vertical tab</td>
<td>\v</td>
</tr>
<tr>
<td>backspace</td>
<td>\b</td>
</tr>
</tbody>
</table>

---

Table A.3 Escape Sequences for Non-Graphic Characters
A.1.5.3  Floating-point Literals

A floating-point literal consists of an integer part, a decimal point, a fraction part, an e or E, an optionally signed integer exponent, and an optional type suffix. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (but not both) may be missing; either the decimal point or the letter e (or E) and the exponent (but not both) may be missing.

A.1.5.4  String Literals

A string literal is a sequence of characters (as defined in Section A.1.5.2) surrounded by double quotes, as in "...". Adjacent string literals are concatenated. Characters in concatenated strings are kept distinct. For example, "\\xA" "B" contains the two characters \xA and B after concatenation (and not the single hexadecimal character \xAB).

The size of a string literal is the number of character literals enclosed by the quotes, after concatenation. The size of the literal is associated with the literal. Within a string, the double quote character " must be preceded by a \.

A string literal may not contain the character \0.
A.2 Preprocessing

IDL preprocessing, based on ANSI C++ preprocessing, provides
- Macro substitution
- Conditional compilation
- Source file inclusion (#include)
- Type declaration inclusion (#import)

Lines beginning with # (also called directives) communicate with the preprocessor. White space may appear before the #.

A preprocessing token is an IDL token, a file name as in an #include directive, or any single character, other than white space, that does not match another preprocessing token.

The IDL preprocessor directives are the following ANSI standard directives
- #if
- #ifdef
- #ifndef
- #else
- #elif
- #include
- #define
- #undef
- #line
- #error
- #pragma

plus an additional IDL directive #import.

The primary use of the preprocessing facilities is to include definitions from other IDL specifications. Text in files included with a #include directive is treated as if it appeared in the including file. Text in files specified with a #import directive is examined to locate the declarations for any unresolved data types in the importing file.

When developing IDL on OpenVME systems the # (hash) character in the above directives, must be replaced by £ or $, depending on how the keyboard has been set up. (The character is hexadecimal 7B.)

A.3 IDL Grammar

This section lists the IDL grammar.

(1) <specification> ::= <definition>+

(2) <definition> ::= <type_dcl> ";"
  | <const_dcl> ";"
  | <except_dcl> ";"
  | <interface> ";"
  | <module> ";"

(3) <module> ::= "module" <identifier> "{" <definition>+ "}"

(4) <interface> ::= <interface_dcl>
  | <forward_dcl>

(5) <interface_dcl> ::=<interface_header> "{" <interface_body> "}"

(6) <forward_dcl> ::= "interface" <identifier>

(7) <interface_header> ::= "interface" <identifier> [ <inheritance_spec> ]

(8) <interface_body> ::= <export>
(9) `<export>` ::= `<type_dcl> ";"
| `<const_dcl> ";"
| `<except_dcl> ";"
| `<attr_dcl> ";"
| `<op_dcl> ";"

(10) `<inheritance_spec>` ::= ":" `<scoped_name> ";" `<scoped_name> {"," `<scoped_name>` }*

(11) `<scoped_name>` ::= `<identifier>
| "::" `<identifier>
| `<scoped_name> "::" `<identifier>

(12) `<const_dcl>` ::= "const" `<const_type> `<identifier>` "=" `<const_exp`

(13) `<const_type>` ::= `<integer_type`
| `<char_type`
| `<boolean_type`
| `<floating_pt_type`
| `<string_type`
| `<scoped_name`

(14) `<const_exp>` ::= `<or_expr`

(15) `<or_expr>` ::= `<xor_expr`
| `<or_expr> "|" `<xor_expr`

(16) `<xor_expr>` ::= `<and_expr`
| `<xor_expr> "^" `<and_expr`

(17) `<and_expr>` ::= `<shift_expr`
| `<and_expr> "&" `<shift_expr`

(18) `<shift_expr>` ::= `<add_expr`
| `shift_expr">>>" `<add_expr`
| `shift_expr"<<<" `<add_expr`

(19) `<add_expr>` ::= `<mult_expr`
| `<add_expr> "+" `<mult_expr`
| `<add_expr> "-" `<mult_expr`

(20) `<mult_expr>` ::= `<unary_expr`
| `<mult_expr> "*" `<unary_expr`
| `<mult_expr> "/" `<unary_expr`
| `<mult_expr> "%" `<unary_expr`

(21) `<unary_expr>` ::= `<unary_operator><primary_expr`
| `<primary_expr`

(22) `<unary_operator>` ::= "-"
| "+"
| "~"

(23) `<primary_expr>` ::= `<scoped_name`
| `<literal`
| "(" `<const_exp> ")"

(24) `<literal>` ::= `<integer_literal`
| `<string_literal`
| `<character_literal`
| `<floating_pt_literal`
| `<boolean_literal`

(25) `<boolean_literal>` ::= "TRUE"
| "FALSE"

(26) `<positive_int_const>` ::= `<const_exp`


(27) <type_dcl> ::= "typedef" <type_declarator>
| <struct_type>
| <union_type>
| <enum_type>

(28) <type_declarator> ::= <type_spec> <declarators>

(29) <type_spec> ::= <simple_type_spec>
| <constr_type_spec>

(30) <simple_type_spec> ::= <base_type_spec>
| <template_type_spec>
| <scoped_name>

(31) <base_type_spec> ::= <floating_pt_type>
| <integer_type>
| <char_type>
| <boolean_type>
| <octet_type>
| <any_type>

(32) <template_type_spec> ::= <sequence_type>
| <string_type>

(33) <constr_type_spec> ::= <struct_type>
| <union_type>
| <enum_type>

(34) <declarators> ::= <declarator> { "," <declarator> }*

(35) <declarator> ::= <simple_declarator>
| <complex_declarator>

(36) <simple_declarator> ::= <identifier>

(37) <complex_declarator> ::= <array_declarator>

(38) <floating_pt_type> ::= "float"
| "double"

(39) <integer_type> ::= <signed_int>
| <unsigned_int>

(40) <signed_int> ::= <signed_long_int>
| <signed_short_int>

(41) <signed_long_int> ::= "long"

(42) <signed_short_int> ::= "short"

(43) <unsigned_int> ::= <unsigned_long_int>
| <unsigned_short_int>

(44) <unsigned_long_int> ::= "unsigned" "long"

(45) <unsigned_short_int> ::= "unsigned" "short"

(46) <char_type> ::= "char"

(47) <boolean_type> ::= "boolean"

(48) <octet_type> ::= "octet"

(49) <any_type> ::= "any"

(50) <struct_type> ::= "struct" <identifier> "{" <member_list> "}"

(51) <member_list> ::= <member>+

(52) <member> ::= <type_spec> <declarators>";

(53) <union_type> ::= "union" <identifier> "switch" "{"<switch_type_spec> "}" "{" <switch_body> "}"
A.4 IDL Specification

An IDL specification consists of one or more type definitions, constant definitions, exception definitions, or module definitions.
The syntax is
<specification> ::=<definition>+
<definition> ::=<type_dcl> ";"
|<const_dcl> ";"
<except_dcl> ";"
@interface> ";"
|<module> ";"

A.4.1 Module Declaration
A module definition satisfies the following syntax:
,module <identifier> "(" <definition>+ ")"
The module construct is used to scope IDL identifiers. See Section A.11 for details of scoping.

A.4.2 Interface Declaration
An interface definition satisfies the following syntax:
@interface> ::=<interface_dcl>
|<forward_dcl>
@interface_dcl> ::=<interface_header> "{"
@interface_body> "}"
<forward_dcl> ::= "interface" <identifier>
@interface_header> ::= "interface" <identifier> [<inheritance_spec> ]
@interface_body> ::=<export>*
<export> ::=<type_dcl> ";"
|<const_dcl> ";"
<except_dcl> ";"
|<attr_dcl> ";"
|<op_dcl> ";"

A.4.2.1 Interface Header
The interface header consists of two elements:
- The interface name. The name must be preceded by the keyword interface, and consist of an identifier that names the interface.
- An optional inheritance specification.

The <identifier> that names an interface defines a legal type name. Such a type name may be used anywhere an <identifier> is legal in the grammar, subject to semantic constraints as described in the following sections. Since one can only hold references to an object the meaning of a parameter or structure member which is an interface type is as a reference to an object supporting that interface.
A.4.2.2 Inheritance Specification

The syntax for inheritance is:

\[
\text{<inheritance_spec} := ":" \text{<scoped_name> \{} "\},
\]

\[
\text{<scoped_name> := <identifier>}
| "::" \text{<identifier>}
| \text{<scoped_name> "::" <identifier>}
\]

Each <scoped_name> in an <inheritance_spec> must denote a previously defined interface. See Section A.5 for details of inheritance.

A.4.2.3 Interface Body

The interface body contains these kinds of declarations:

- Constant declarations specify the constants that the interface exports. See Section A.6.
- Type declarations specify the type definitions that the interface exports. See Section A.7.
- Exception declarations specify the exception structures that the interface exports. See Section A.8.
- Attribute declarations specify the associated attributes exported by the interface. See Section A.10.
- Operation declarations specify the operations that the interface exports and the format of each, including operation name, the type of data returned, the types of all parameters of an operation, legal exceptions which may be returned as a result of an invocation, and contextual information which may affect method dispatch, see Section A.9.

Empty interfaces containing no declarations are permitted.

A.4.2.4 Forward Declaration

A forward declaration declares the name of an interface without defining it. This permits the definition of interfaces that refer to each other. The syntax consists simply of the keyword interface followed by an <identifier> that names the interface. The actual definition must follow later in the specification.

Multiple forward declaration of the same interface name are legal.

A.5 Inheritance

An interface can be derived from another interface which is then called a base interface of the derived interface.

A derived interface, like all interfaces, may declare new elements (constants, types, attributes, exceptions, and operations). In addition, unless redefined in the derived interface, the elements of a base interface can be referred to as if they were elements of the derived interface. The name resolution operator (::) may be used to refer to a base element explicitly. This permits reference to a name that has been redefined in the derived interface.

A derived interface may redefine any of the type, constant, and exception names which have been inherited the scope rules for such names are described in Section A.11.

An interface is called a direct base if it is mentioned in the <inheritance_spec> and an indirect base if it is not a direct base but is a base interface of one of the interfaces mentioned in the <inheritance_spec>.

An interface may be derived from any number of base interfaces. Such use of more than one direct base interface is often called multiple inheritance. The order of derivation is not significant.
An interface may not be specified as a direct base interface of a derived interface more than once. It may be an indirect base interface more than once. Consider the following example:

```eiffel
interface A {...}
interface B: A {...}
interface C: A {...}
interface D: B, C {...}
```

The declaration for interface D is legal, even though the inherited interfaces B and C both inherit themselves inherit interface A. But a declaration of the form `interface D: A, A {...}` is illegal.

Reference to base interface elements must be unambiguous. Reference to a base interface element is ambiguous if the expression used refers to a constant, type, or exception in more than one base interface. (It is currently illegal to inherit from two interfaces with the same operation or attribute name, or to redefine an operation or attribute name in the derived interface.) Ambiguities can be resolved by qualifying a name with its interface name (using a `<scoped_name>`).

References to constants, types, and exceptions are bound to an interface when it is defined, that is replaced with the equivalent global `<scoped name>`s. This guarantees that the syntax and semantics of an interface are not changed when the interface is a base class for a derived class. For example

```eiffel
const long L = 3;
interface A
{
    void f (in float s[L]);  //s has 3 floats
};
interface B
{
    const long L = 4;
};
interface C: B, A {}        //what is f()’s signature?
```

The early binding of constants, types, and exceptions at interface definition guarantees that the signature of operation `f` in interface `C` is

```eiffel
void f(in float s[3]);
```

which is identical to that in interface `A`. This rule also prevents the redefinition of a constant, type, or exception in the derived interface from affecting, the operations and attributes inherited from a base interface.

Interface inheritance causes all identifiers in the closure of the inheritance tree to be imported into the current naming scope. A type name, constant name, enumeration value name, or exception name from an enclosing scope, can be redefined in the current scope. An attempt to use an ambiguous name without qualification is a compilation error.

Operation names are used at runtime by both the stub and dynamic interfaces. As a result, all operations that might apply to a particular object must have unique names. This requirement prohibits the redefinition of an operation name in a derived class, as well as inheriting two operations with the same name.
A.6 Constant Declaration

This section describes the syntax for constant declarations.

A.6.1 Syntax

The syntax for a constant declaration is

<const_dcl> ::= "const" <const_type> <identifier> "=" <const_exp>
<const_type> ::= <integer_type>
| <char_type>
| <boolean_type>
| <floating_pt_type>
| <string_type>
| <scoped_name>
<const_exp> ::= <or_expr>
<or_expr> ::= <xor_expr>
| <or_expr> "|" <xor_expr>
<xor_expr> ::= <and_expr>
| <xor_expr> "^" <and_expr>
<and_expr> ::= <shift_expr>
| <and_expr> "&" <shift_expr>
<shift_expr> ::= <add_expr>
| <shift_expr> ">>" <add_expr>
| <shift_expr> "<<" <add_expr>
<add_expr> ::= <mult_expr>
| <add_expr> "+" <mult_expr>
| <add_expr> "-" <mult_expr>
<mult_expr> ::= <unary_expr>
| <mult_expr> "*" <unary_expr>
| <mult_expr> "/" <unary_expr>
| <mult_expr> "%" <unary_expr>
<unary_expr> ::= <unary_operator> <primary_expr>
| <primary_expr>
<unary_operator> ::= "+"
| "-"
| "~"
<primary_expr> ::= <scoped_name>
| <literal>
| "(" <const_exp> ")"
<literal> ::= <integer_literal>
| <string_literal>
| <character_literal>
| <floating_pt_literal>
| <boolean_literal>
<boolean_literal> ::= "TRUE"
| "FALSE"
<positive_int_const> ::= <const_exp>
A.6.2 Semantics

The `<scoped_name>` in the `<const_type>` production must be a previously defined name of an `<integer_type>`, `<char_type>`, `<boolean_type>`, `<floating_pt_type>` or `<string_type>` constant.

Mixed type expressions (for example integers mixed with floats) are illegal.

An integer constant expression is evaluated as unsigned long, unless it contains a negated integer literal or the name of an integer constant with a negative value. In the latter case, the constant expression is evaluated as signed long. The computed value is coerced back to the target type in constant initializers.

It is an error if the computed value exceeds the precision of the target type. It is an error if any intermediate value exceeds the range of the evaluated-as type (`long` or `unsigned long`).

All floating-point literals are double, all floating-point constants are coerced to double, and all floating-point expressions are computed as doubles. The computed double value is coerced back to the target type in constant initializers.

It is an error if this coercion fails or if any intermediate values (when evaluating the expression) exceed the range of double.

Unary (+ -) and binary (* / + -) operators are applicable in floating-point expressions.

Unary (+ - ~) and binary (* / % + - << >>> & | ^) operators are applicable in integer expressions.

The ~ unary operator indicates that the bit-complement of the expression to which it is applied should be generated. For the purposes of such expressions, the values are two's complement numbers.

As such, the complement can be generated as

```
long     -(value+1)
unsigned long    (2^{32}-1) -value
```

The `%` binary operator yields the remainder from the division of the first expression by the second. If the second operand of `%` is 0, the result is undefined; otherwise: \((a/b)\times b + a \% b\) is equal to ‘a’. If both operands are non-negative, then the remainder is non-negative; if not, the sign of the remainder is implementation dependent. The `<<` binary operator indicates that the value of the left operand should be shifted left the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand must be in the range 0 <= right operand < 32.

The `>>>` binary operator indicates that the value of the left operand should be shifted right the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand must be in the range 0 <= right operand < 32.

The `&` binary operator indicates that the logical, bitwise AND of the left and right operands should be generated.

The `|` binary operator indicates that the logical, bitwise OR of the left and right operands should be generated.

The `^` binary operator indicates that the logical, bitwise EXCLUSIVE-OR of the left and right operands should be generated.

`<positive_int_const>` must evaluate to a positive integer constant.
A.7 Type Declaration

IDL provides constructs for naming data types providing C language-like declarations that associate an identifier with a type. IDL uses the `typedef` keyword to associate a name with a data type. A name is also associated with a data type through the `struct`, `union`, and `enum` declarations. The syntax is

```plaintext
<type_dcl> ::= "typedef"<type_declarator>
| <struct_type>
| <union_type>
| <enum_type>

<type_declarator> ::= <type_spec> <declarators>
```

For type declarations, IDL defines a set of type specifiers to represent typed values. The syntax is

```plaintext
<type_spec> ::= <simple_type_spec>
| <constr_type_spec>

<simple_type_spec> ::= <base_type_spec>
| <template_type_spec>
| <scoped_name>

<base_type_spec> ::= <floating_pt_type>
| <integer_type>
| <char_type>
| <boolean_type>
| <octet_type>
| <any_type>

<template_type_spec> ::= <sequence_type>
| <string_type>

<constr_type_spec> ::= <struct_type>
| <union_type>
| <enum_type>

<declarators> ::= <declarator> { "," <declarator> }*

<declarator> ::= <simple_declarator>
| <complex_declarator>

<simple_declarator> ::= <identifier>

<complex_declarator> ::= <array_declarator>
```

The `<scoped_name>` in `<simple_type_spec>` must be a previously defined type.

As seen above, IDL type specifiers consist of scalar data types and type constructors. IDL type specifiers can be used in operation declarations to assign data types to operation parameters. The next sections describe basic and constructed type specifiers.
A.7.1 Basic Types

The syntax for the supported basic types is

\[
\begin{align*}
<\text{floating-pt-type}> & ::= "\text{float}" \\
& | "\text{double}" \\
<\text{integer-type}> & ::= <\text{signed-int}> \\
& | <\text{unsigned-int}> \\
<\text{signed-int}> & ::= <\text{signed-long-int}> \\
& | <\text{signed-short-int}> \\
<\text{signed-long-int}> & ::= "\text{long}" \\
<\text{signed-short-int}> & ::= "\text{short}" \\
<\text{unsigned-int}> & ::= <\text{unsigned-long-int}> \\
& | <\text{unsigned-short-int}> \\
<\text{unsigned-long-int}> & ::= "\text{unsigned long}" \\
<\text{unsigned-short-int}> & ::= "\text{unsigned short}" \\
<\text{char-type}> & ::= "\text{char}"
\end{align*}
\]

Each IDL data type is mapped to a native data type through the appropriate language mapping. Conversion errors between IDL data types and the native types to which they are mapped can occur during the performance of an operation invocation. The invocation mechanism (client stub, dynamic invocation engine, and skeletons) may signal an exception condition to the client if an attempt is made to convert an illegal value. The standard exceptions that are to be signaled in such situations, are defined elsewhere in this document.

A.7.1.1 Integer Types

IDL supports `long` and `short` signed and unsigned integer data types. `long` represents the range `-2^{31}..2^{31}-1` while `unsigned long` represents the range `0..2^{32}-1`. `short` represents the range `-2^{15}..2^{15}-1`, while `unsigned short` represents the range `0..2^{16}-1`.

A.7.1.2 Floating-point Types

IDL floating-point types are `float` and `double`. The `float` type represents IEEE single-precision floating-point numbers; the double type represents IEEE double-precision floating-point numbers. The IEEE floating point standard specification, *IEEE Standard for Binary Floating-Point Arithmetic: ANSI/IEEE Std 754-1985*, should be consulted for more information on the precision afforded by these types.

A.7.1.3 Char Type

IDL defines a `char` data type consisting of 8-bit quantities.

The ISO Latin-1 (8859.1) character set standard defines the meaning and representation of all possible graphic characters. The meaning and representation of the null and formatting characters is the numerical value of the character as defined in the *ASCII (ISO 646) standard*. The meaning of all other characters is implementation-dependent.

A.7.1.4 Boolean Type

The `boolean` data type is used to denote a data item that can only take one of the values `TRUE` and `FALSE`. 
A.7.1.5 Octet Type

The octet type is an 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system.

A.7.1.6 Any Type

The any type permits the specification of values that can express any IDL type.

A.7.1.7 Constructed Types

The constructed types are

\[
\text{<constr_type_spec>} ::= \text{<struct_type>} \\
\text{<union_type>} \\
\text{<enum_type>}
\]

Although it is syntactically possible to generate recursive type specifications in IDL, such recursion is semantically constrained. The only permissible form of recursive type specification is through the use of the sequence template type.

For example, this is legal

```idl
struct foo
{
    long value;
    sequence<foo>chain;
}
```

See Section A.7.2 for details of the sequence template type.

A.7.1.8 Structures

The structure syntax is

\[
\text{<struct_type>} ::= \text{"struct"} \text{<identifier>} \{ \\
\text{<member_list>} \}
\]

\[
\text{<member_list>} ::= \text{<member>}+
\]

\[
\text{<member>} ::= \text{<type_spec>} \text{<declarators>} ;
\]

The <identifier> in <struct_type> defines a new legal type. Structure types may also be named using a typedef declaration.

Name scoping rules require that the member declarators in a particular structure be unique. The value of a struct is the value of all of its members.

A.7.1.9 Discriminated Unions

The discriminated union syntax is

\[
\text{<union_type>} ::= \text{"union"} \text{<identifier>} \text{"switch"} \{ \\
\text{<switch_type_spec>} \}
\]

\[
\text{<switch_type_spec>} ::= \text{<integer_type>} \\
\text{<char_type>} \\
\text{<boolean_type>} \\
\text{<enum_type>} \\
\text{<scoped_name>}
\]
IDL unions are a cross between the C union and switch statements. IDL unions must be discriminated, that is, the union header must specify a typed tag field that determines which union member to use for the current instance of a call.

The <identifier> following the union keyword defines a new legal type. Union types may also be named using a typedef declaration.

The <const_exp> in a <case_label> must be consistent with the <switch_type_spec>. A default case can appear at most once.

The <scoped_name> in the <switch_type_spec> production must be a previously defined integer, char, boolean or enum type.

Case labels must match or be automatically castable to the defined type of the discriminator. The complete set of matching rules are shown in Table A.4.

Table A.4 Case Labels and Discriminator Types

<table>
<thead>
<tr>
<th>Discriminator Type</th>
<th>Matched by</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>Any integer value in the value range of long.</td>
</tr>
<tr>
<td>short</td>
<td>Any integer value in the value range of short.</td>
</tr>
<tr>
<td>unsigned long</td>
<td>Any integer value in the value range of unsigned long.</td>
</tr>
<tr>
<td>unsigned short</td>
<td>Any integer value in the value range of unsigned short.</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>boolean</td>
<td>TRUE or FALSE</td>
</tr>
<tr>
<td>enum</td>
<td>Any enumerator for the discriminator enum type.</td>
</tr>
</tbody>
</table>

Name scoping rules require that the element declarators in a particular union be unique. If the <switch_type_spec> is an <enum_type> the identifier for the enumeration is in the scope of the union, as a result it must be distinct from the element declarators.

All possible values of the union discriminator do not have to be listed in the <switch body>. The value of a union is the value of the discriminator together with one of the following

- If the discriminator value was explicitly listed in a case statement, the value of the element associated with that case statement
- If a default case label was specified, the value of the element associated with the default case label
- No additional value

Access to the discriminator and the related element is language-mapping dependent.
A.7.1.10 Enumerations

Enumerated types consist of ordered lists of identifiers. The syntax is

\[
<\text{enum_type}> ::= "enum"<\text{identifier}>
\{"<\text{enumerator}>
\","<\text{enumerator}>\}*
\]

\[
<\text{enumerator}> ::= <\text{identifier}>
\]

A maximum of \(2^{32}\) identifiers may be specified in an enumeration. The \(<\text{identifier}>\) following the \textit{enum} keyword defines a new legal type. Enumerated types may also be named using a typedef declaration.

A.7.2 Template Types

The template types are

\[
<\text{template_type_spec}>::= <\text{sequence_type}>
\mid <\text{string_type}>
\]

A.7.2.1 Sequences

IDL defines the sequence type \textit{sequence}. A sequence is a one-dimensional array with two characteristics: a maximum size (that is fixed at compile time) and a length (which is determined at run time).

The syntax is

\[
<\text{sequence_type}> ::= "sequence" "<" <\text{simple_type_spec}> ","<\text{positive_int_const}> ">"
\mid "sequence" "<" <\text{simple_type_spec}> ">
\]

The second parameter in a sequence declaration indicates the maximum size of the sequence. If a positive integer constant is specified for the maximum size the sequence is termed a \textit{bounded sequence}. Prior to passing a bounded sequence as a function argument (or as a field in a structure or union) the length of the sequence must be set using normal C compiler techniques. After receiving a sequence result from an operation invocation the length of the returned sequence will have been set, this value may be obtained using normal C compiler techniques.

If no maximum size is specified size of the sequence is unspecified (unbounded). Prior to passing a sequence such as a function argument (or as a field in a structure or union) the length of the sequence, the maximum size of the sequence and the address of a buffer to hold the sequence must be set using normal C techniques. After receiving such a sequence result from an operation invocation, the length of the returned sequence will have been set, this value may be obtained using normal C techniques.

A.7.2.2 Strings

IDL defines the \textit{string} type as a string consisting of all possible 8-bit quantities except null. A string is similar to a sequence of \textit{char}. As with sequences of any type, prior to passing a string as a function argument (or as a field in a structure or union), the length of the string must be set using normal C techniques. The syntax is

\[
<\text{string_type}> ::= "string" "<" <\text{positive_int_const}> ">
\mid "string"
\]

The argument to the string declaration is the maximum size of the string. If a positive integer maximum size is specified, the string is termed a bounded string; if no maximum size is specified, the string is termed an unbounded string.
A.7.3 Complex Declarator - Arrays

IDL defines multidimensional, fixed-size arrays. An array includes explicit sizes for each dimension.

The syntax for arrays is

\[\langle array\_declarator\rangle ::= \langle identifier\rangle \langle fixed\_array\_size\rangle^*\]
\[\langle fixed\_array\_size\rangle ::= \"[\" \langle positive\_int\_const\rangle \"]\]

The array size (in each dimension) is fixed at compile time. When an array is passed as a parameter in an operation invocation, all elements of the array are transmitted.

A.8 Exception Declaration

Exception declarations permit the declaration of struct-like data structures which may be returned to indicate that an exceptional condition has occurred during the performance of a request. The syntax is:

\[\langle except\_dcl\rangle ::= \"exception\" \langle identifier\rangle \{"\langle \langle member\rangle\rangle^* \"\}\"

Each exception is characterized by its IDL identifier, an exception type identifier and the type of the associated return value (as specified by the \langle member\rangle s in its declaration). If an exception is returned as the outcome to a request then the value of the exception identifier is accessible to the programmer for determining which particular exception was raised. If an exception is declared with members, users will be able to access the values of those members when an exception is raised. If no members are specified, no additional information is accessible when an exception is raised.

A set of system exceptions is defined corresponding to standard runtime errors which may occur during the execution of a request.

A.9 Operation Declaration

Operation declarations in IDL are similar to C function declarations. The syntax is

\[\langle op\_dcl\rangle ::= \[\langle op\_attribute\rangle \] \langle op\_type\_spec\rangle \langle identifier\rangle\langle parameter\_dcls\rangle \[ \langle raises\_expr\rangle \] \[ \langle context\_expr\rangle \]
\[\langle op\_type\_spec\rangle ::= \langle param\_type\_spec\rangle \| \"void\"

An operation declaration consists of

- An optional operation attribute that specifies which invocation semantics the communication system should provide when the operation is invoked. Operation attributes are described in Section A.9.1.
- The type of the operation’s return result; the type may be any type which can be defined in IDL. Operations that do not return a result must specify the void type.
- An identifier that names the operation in the scope of the interface in which it is defined
- A parameter list that specifies zero or more parameter declarations for the operation. Parameter declaration is described in Section A.9.2.
- An optional raises expression which indicates which exceptions may be raised as a result of an invocation of this operation. Raises expressions are described in Section A.9.3.
- An optional context expression which indicates which elements of the request context may be consulted by the method that implements the operation. Context expressions are described in Section A.9.4.
A.9.1 Operation Attribute

The operation attribute specifies which invocation semantics the communication service must provide for invocations of a particular operation. An operation attribute is optional. The syntax for its specification is

<op_attribute> ::= "oneway"

When a client invokes an operation with the oneway attribute, the invocation semantics are best-effort, which does not guarantee delivery of the call; best-effort implies that the operation will be invoked at most once. An operation with the oneway attribute must not contain any output parameters and must specify a void return type. An operation defined with the oneway attribute may not include a raises expression; invocation of such an operation, however, may raise a standard exception.

If an <op_attribute> is not specified, the invocation semantics is at-most-once if an exception is raised; the semantics are exactly-once if the operation invocation returns successfully.

A.9.2 Parameter Declarations

Parameter declarations in IDL operation declarations have the following syntax

<parameter_dcls> ::= "(" <param_dcl> { "," <param_dcl> }* ")"
<param_dcl> ::= <param_attribute> <param_type_spec>
<param_attribute> ::= "in"
| "out"
| "inout"
<param_type_spec> ::= <base_type_spec>
| <string_type>
| <scoped_name>

A parameter declaration must have a directional attribute that informs the communication service in both the client and the server of the direction in which the parameter is to be passed. The directional attributes are:

- in, the parameter is passed from client to server
- out, the parameter is passed from server to client
- inout, the parameter is passed in both directions

If an exception is raised as a result of an invocation, the values of the return result and any out and inout parameters are undefined.

When an unbounded string or sequence is passed is an inout parameter, the returned value cannot be longer than the input value.

A.9.3 Raises Expressions

A raises expression specifies which exceptions may be raised as a result of an invocation of the operation. The syntax for its specification is

<pre><code>&lt;raises_expr&gt; ::= "raises" "] (" &lt;scoped_name&gt; 
{"," &lt;scoped_name&gt; }* ")"
</code></pre>

The &lt;scoped_name&gt;s in the raises expression must be previously defined exceptions.
In addition to any operation-specific exceptions specified in the raises expression, there are a set of system exceptions that may be signaled. However, system exceptions may not be listed in a raises expression.

The absence of a raises expression on an operation implies that there are no operation specific exceptions. Invocations of such an operation are still liable to receive one of the standard exceptions.

### A.9.4 Context Expressions

A context expression specifies which elements of the client’s context may affect the performance of a request by the object. The syntax for its specification is

\[
<context_expr> ::= "context" "(" <string_literal> { "," <string_literal> }* ")"
\]

The absence of a context expression indicates that there is no request context associated with requests for this operation. Each string literal is an arbitrarily long sequence of alphabetic, digit, period ("."), underscore ("_") and asterisk ("*") characters. The first character of the string must be an alphabetic character. An asterisk may only be used as the last character of the string. Some implementations may use the period character to partition the name space.

### A.10 Attribute Declaration

An interface can have attributes as well as operations. As such, attributes are defined as part of an interface. An attribute definition is logically equivalent to declaring a pair of accessor functions: one to retrieve the value of the attribute and one to set the value of the attribute.

The syntax for attribute declaration is

\[
<attr_dcl> ::= [ "readonly" ] "attribute" <param_type_spec> <simple_declarator> {"," <simple_declarator> }*
\]

The optional readonly keyword indicates that there is only a single accessor function, the retrieve value function. For example

```plaintext
interface foo
{
    enum material_t {rubber, glass};
    struct position_t
    {
        float x, y;
    };
    attribute float radius;
    attribute material_t material;
    readonly attribute position_t position;
    ...
}
```

The attribute declarations are equivalent to the following pseudo-specification fragment

```plaintext
float _get_radius ();
void _set_radius (in float r);
material_t _get_material ();
void _set_material (in material_t m);
position_t _get_position ();
```
Only the attribute name is subject to IDL name scoping rules, the accessor function names are guaranteed not to collide with any legal operation names specifiable in IDL. Refer to Chapter 2 for more details.

Attribute operations return errors by means of standard exceptions.

Attributes are inherited. An attribute name cannot be redefined to be a different type. See Section A.11 for more information on redefinition constraints and the handling of ambiguity.

A.11 Names and Scoping

To prevent names defined within the CORBA specification from clashing with names in programming languages and other software systems each name defined by CORBA is treated as if it were defined within a module named CORBA.

A.11.1 CORBA Module

Within an IDL specification however, IDL keywords representing entities within the CORBA module, such as Object, must not be preceded by a CORBA:: prefix. Other interface names, such as TypeCode, are not IDL keywords and so must be referred to by their fully scoped names (for example, CORBA::Type Code).

A.11.2 Naming Scopes

An entire IDL file forms a naming scope. In addition, the following kinds of definitions form nested scopes

- module
- interface
- structure
- union
- operation
- exception

Identifiers for the following kinds of definitions are scoped

- types
- constants
- enumeration values
- exceptions
- interfaces
- attributes
- operations

An identifier can only be defined once in a scope, identifiers can also be redefined in nested scopes.

IDL identifiers are case insensitive, so two identifiers that differ only in the case of their characters are considered redefinition’s of one another. However, all references to a definition must use the same case as the defining, occurrence.

Type names defined in a scope are available for immediate use within that scope. A name can be used in an unqualified form within a particular scope; it will be resolved by successively searching farther out in enclosing scopes. Once an unqualified name is used in a scope, it cannot be redefined. So if you have used a name defined in an enclosing scope in the current scope, you cannot then redefine a version of the name in the current scope. Such redefinitions yield a compilation error.
A qualified name (one of the form `<scoped_name>::<identifier>`) is resolved by first resolving the qualifier `<scoped_name>` to a scope S, and then locating the definition of `<identifier>` within S. The identifier must be directly defined in S or (if S is an interface) inherited into S. The `<identifier>` is not searched for in enclosing scopes.

When a qualified name begins with `::` the resolution process starts with the smallest enclosing module and locates subsequent identifiers in the qualified name by the rule described in the previous paragraph.

Every IDL definition in a file has a global name within that file. The global name for a definition is constructed as follows:

- Prior to starting to scan a file containing an IDL specification the name of the current root is initially empty (""") and the name of the current scope is initially empty ("")
- Whenever a `module` keyword is encountered, the string, ":=" and the associated identifier are appended to the name of the current root; upon detection of the termination of the module, the trailing ":=" and identifier are deleted from the name of the current root
- Whenever an `interface`, `struct`, `union`, or `exception` keyword is encountered, the string ":=" and the associated identifier are appended to the name of the current scope; upon detection of the termination of the interface, struct, union, or exception, the trailing ":=" and identifier are deleted from the name of the current scope
- A new, unnamed, scope is entered when the parameters of an operation declaration are processed; this allows the parameter names to duplicate other identifiers; when parameter processing has completed, the unnamed scope is exited

The global name of an IDL definition is the concatenation of the current root, the current scope, a ":=" and the <> which is the local name; for that definition.

Inheritance produces shadow copies of the inherited identifiers, it introduces names into the derived interface. These names are considered to be semantically the same as the original definition. Two shadow copies of the same original introduce a single name into the derived interface and do not conflict with each other.

Inheritance introduces multiple global IDL names for the inherited identifiers. For example

```idl
interface A
{
    exception E { long L; };
    void f() raises(E);
};
interface B: A
{
    void g() raises(E);
};
```

In this example, the exception is known by the global names ::A::E and ::B::E.

Ambiguity can arise in specifications due to the nested naming scopes. For example

```idl
interface A
{
    typedef string<128> string_t;
};
interface B
{
    typedef string<256> string_t;
};
interface C: A, B
{
    attribute string_t Title;   /* AMBIGUOUS!!! */
};
```
The attribute declaration in interface C is ambiguous, since the compiler does not know which string_t is desired. Ambiguous declarations yield compilation errors.

A.12 Differences from C++

The IDL grammar, while attempting to conform to the C++ syntax, is somewhat more restrictive. The current restrictions are

- A function return type is mandatory
- A name must be supplied with each formal parameter to an operation declaration
- A parameter list consisting of the single token void is not permitted as a synonym for an empty parameter list
- Tags are required for structures, discriminated unions and enumerations
- Integer types cannot be defined as simply int or unsigned; they must be declared explicitly as short or long
- Char cannot be qualified by signed or unsigned keywords
Appendix B

Property Constraint Language

The BNF for the property constraint language as supported by the Trader is

<program> ::= <empty>
  | <expr>
  | <expr> -> <superlative>
<empty> ::=<expr> ::= <expr> or <expr>
  | <expr> and <expr>
  | not <expr>
  | ( <expr> )
  | <nexpr> in <nexpr>
  | <nexpr> == <nexpr>
  | <nexpr> != <nexpr>
  | <nexpr> <nexpr>
  | <nexpr> <= <nexpr>
  | <nexpr> == <nexpr>
  | <nexpr> > <nexpr>
  | <nexpr> >= <nexpr>
<superlative> ::= min [ <nexpr> ]
  | max [ <nexpr> ]
<nexpr> ::= <term>
  | <nexpr> + <term>
  | <nexpr> - <term>
<term> ::= <factor>
  | <term> * <factor>
  | <term> / <factor>
<factor> ::= <identifier>
  | <constant>
  | ( <nexpr> )
  | -<factor>
<identifier> ::= <letter><characters>
<letter> ::= a | b | c | d | e | f | g | h | I | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<characters> ::= <empty>
  | <character>
  | <characters> <character>
<character> ::= <letter>
  | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Property Constraint Language

<constant> ::= <floatnumber>  
| <string>  

<floatnumber> ::= <mantissa><exponent>  

<string> ::= ’<chars>’  

<mantissa> ::= <digits>  
| <digits>.<digits>  
| .<digits>  

<exponent> ::= <empty>  
| <sign>e <digits>  
| <sign>E <digits>  

<sign> ::= <empty>  
| -  
| +  

<digits> ::= <digit>  
| <digits> <digit>  

<chars> ::= <empty>  
| <chars><char>  

<char> ::= <letter>  
| <digit>  
| <other>  

<other> ::= ’*’ | ~ | ! | @ | # | $ | % | ^ | &  
| * | ( | ) | - | = | + | [ | ]  
| | ; | : | “ | | | , | <  
| . | > | / | ?
Symbols

#import directive 2-13

A
Addressing information cache 3-9
ANSI C++ pre-processing 2-12
Any type A-15
Application Level Factory 6-1
Application level object composites 6-1
ASCII (ISO 646) standard A-14

B
Basic types A-14
Binding 3-9
  how to achieve 3-5
Body
  interface A-9
Boolean A-2
  type A-15

C
C++
  lexical rules A-1
Call type
  one way 3-12
  Synchronous 3-12
Call types
  deferred synchronous 3-12
  invocation 3-12
Capsule construction 3-7
Capsule Factories 6-1
  object interface 6-1
Capsule objects 6-1
Capsules 3-1
  Eiffel 3-7
    launching 6-1
    launching on demand 6-1
    stopping them 3-9
    tidy shutdown 3-9
Capsules in Windows
  stopping them 3-9
Char type A-14
Character literals A-3
Clash
  naming 2-14
Class Library Interfaces 3-1
Client code

invocation 3-1
Client requesting a service
  Trader 4-5
Clients 1-1
Collision avoidance
  naming 2-14
Composite 4-1
Concurrency 5-1
Conditional compilation
  IDL pre-processing 2-13
Constraint language
  used for fine tuning 4-6
Constraints 4-6
Constructed types A-15

D
DAIS_APPLICATION 3-9
Data types 1-4
  mapping onto C++ data types 2-7
  object references 1-4
  representation 1-4
Declarations
  forms of A-9
Default implementation
  de_register operation 5-5
  lookup operation 5-5
  register operation 5-5
Deferred synchronous calls 3-12
Definition
  interface A-9
  module A-9
Diagnostic message
  token sequence 2-13
Directives 2-13
Discriminated unions A-16
Distributed environments A-16
DOS file naming 2-2

E
EBNF
  Extended Backus-Naur Form A-1
Eiffel Name List
  .enl file 2-14
enl file 2-14
Enumerations A-17
Exception declaration
  members 2-12
Exceptions 3-9
Index-2 Release 3.2.9 orb2 for Eiffel User Guide

DivideByZero declaration 2-6
during execution 2-12
system 2-6, 2-12
user 2-6
user defined 2-12
Extended Backus-Naur Form
EBNF A-1
Extended object
standard orb2 object 5-2
Extended object adapter
features 5-1
Extended Objects 5-2

F
Factory 6-1
Application Level 6-1
Trader domains 6-1
Federation
default context 4-12
delegation 4-12
proxy 4-12
Federation concepts 4-10
File naming 2-2
Floating-point literals A-4
Floating-point types A-14
Forward declaration 2-4

H
Header
interface A-9

I
IDL
case sensitivity 2-2
complex fixed data type 2-11
constructed data types 2-8
CORBA conformance A-1
declarative language A-1
description 2-1
grammar A-5
inheritance construction 1-4
Interface Definition Language 1-4, A-1
lexical conventions A-1
operation declarations 2-4
pre-processing 2-12
preprocessing A-4
Stub code generation 3-2
IDL compiler
C++ source file generation 3-3
orb2 2-16
IDL data types 2-7
basic data types 2-6
complex data types 2-7
constructed data types 2-7
native data type mapping A-14
template data types 2-7
IDL files
compiling 1-4
IDL pre-processor directives
standard directives 2-13
IDL specification A-8
e Example 2-13
IDL specifications
generating stubs and skeletons 2-16
IEEE Standard for Binary Floating-Point
Arithmetic A-14
Inheritance A-9
code re-use 3-7
Inheritance construction
IDL 1-4
Integer literal A-3
Integer types A-14
Interface 1-4
body A-9
definition 1-3, 2-3, A-9
header A-9
type 4-3
Interface Definition Language
definition A-1
IDL 1-4, A-1
Invocation call types 3-12
Invocation transparency 1-4
ISO Latin-1 A-3, A-14

L
Lexical Conventions
literals A-3
libraries
Eiffel
dais_eiffel.a 3-8
dais_eiffel.lib 3-8
orb2 core
dais32.lib 3-8
libdais.a 3-8
Literals
Lexical Conventions A-3
Location transparency
object reference 1-5

M
Macro substitution
IDL pre-processing 2-13
Managed Objects 5-2
Memory management
in distributed environments 3-13
Module
definition A-9
Module construct
scoping identifiers 2-3
Multiple clients 1-4
Multiple inheritance 4-3
Multi-threading 5-1
N
Name collision avoidance 2-14
Name scoping conventions 2-2
Names
  portability to Windows 2-2
Naming conventions 2-2
  exception names 2-3
  IDL file names 2-2
  interface names 2-2
  operation names 2-2
  parameter names 2-3
  user defined type names 2-3
Narrow 3-5
Network Service Access Point 3-9

O
Object 1-1, 3-9
  interaction 1-3
  location 3-9
Object references 1-4, 3-9, 3-11
  binding 3-9
  definition 3-4
  normalising to string format 3-11
Object Relocation 5-3
Octet type A-15
One way call 3-12
Oneway operation
  declaration 2-6
  exceptions 2-6
Operation declaration
  Opcontext 2-4
  Opexceptions 2-4
Operation invocation 1-4
ORB entry point
  schedule 3-8
orb2
  basic services 3-1
  bootstrap object references 3-5
  core libraries
    dais32.lib 3-8
    libdais.a 3-8
  Eiffel libraries
    dais_eiffel.a 3-8
    dais_eiffel.lib 3-8
  Extended Object Adapter 5-1
    functionality 5-1
  IDL compiler 2-16
orb2 IDL compiler 2-16
orb2 ORB
  instances 3-11
  linking client and server applications 3-8
OSI 3-9

P
Pre-processing
  primary uses 2-13
Preprocessing
  IDL A-4
  Properties 4-6
    specifying more than one 4-6
    standard 4-7
  Property constraint language 4-6, 4-7
    expressions 4-7
  Property list 4-6
  Propertyname
    constituents 4-6
  Protocol stacks 3-9
  Proxy 3-5
  Proxy Federation 4-12

R
Relocate of services 5-3
Relocation service
  issued state 5-4
  Relocator_lookup() 5-4

S
Sequence
  template type A-15
Sequences A-17
  servers
  capsules 3-1
Session Breakdowns
  reasons 3-9
Signature 1-4
Skeleton 1-4, 3-1
  code 3-3
Source file inclusion (#include)
  IDL pre-processing 2-13
Standard properties 4-7
Stopping Windows Capsules 3-9
String literals A-4
Stub
  client 1-4
  code 3-1
  intra and inter-capsule communications 3-2
Synchronous call type 3-12
System exceptions 2-6

T
TCP/IP 3-9
Template types A-17
Trader 3-5, 3-10, 4-12
  access 4-8
  application contact 4-2
  assumptions concerning property values 4-6
  binding mechanism 4-2
  client requesting a service 4-5
  clients requiring specific service 4-2
  context space 4-4, 4-9
description 4-1
object reference 3-5
organisation 4-3
relocation architecture 4-14
Relocator interface 5-4
Secondary Traders 4-14
Trader interfaces 4-8
Trader object
  service objects 3-10
Trading
  delete operation 4-15
  export 4-15
  lookup operation 4-15
  object location strategy 1-4
  register operation 4-15
  shallow search policy 4-10
  try_delete operation 4-15
Transport Service Access Point 3-9
Type
  any A-15
  basic A-14
  Boolean A-15
  char A-14
  constructed A-15
  declaration A-13
  floating-point A-14
  integer A-14
  octet A-15
  Template A-17

U
UDP/IP 3-9
UNIX command line
  example 2-17

W
WM_CLOSE 3-9
WM_DESTROY 3-9