VisualWorks®

Distributed Smalltalk Application Developer's Guide

P46-0114-03
## Contents

### About This Book

- Audience ................................................................................................................... 15
- Conventions .............................................................................................................. 16
  - Typographic Conventions ............................................................................. 16
  - Special Symbols ........................................................................................... 16
  - Mouse Buttons and Menus .......................................................................... 17
- Getting Help .............................................................................................................. 17
  - Commercial Licensees ................................................................................. 18
    - Before Contacting Technical Support ...................................................... 18
    - Contacting Technical Support ................................................................. 18
  - Non-Commercial Licensees ......................................................................... 19
- Additional Sources of Information ............................................................................. 19

### Chapter 1 Introducing Distributed Smalltalk

- **Distributed Objects** ................................................................. 23
  - What is CORBA? .............................................................. 23
  - Why Distributed Smalltalk? ..................................................... 24
- **CORBA Components** ........................................................................ 24
  - Object Request Broker (ORB) .................................................. 26
  - Interface Definition Language (IDL) Compiler .............................................. 27
  - Internet Inter-Orb Protocol (IIOP) .................................................. 27
  - Interface Repository (IR) ............................................................ 27
- **CORBA Services** .................................................................................. 27
  - Naming ......................................................................................... 28
  - Basic Lifecycle ............................................................................ 28
  - Event Notification ........................................................................ 28
Chapter 2 Installing and Configuring DST

Installing Distributed Smalltalk ................................................................. 30
DST Directory Structure ............................................................................. 30
Loading DST into the Image ..................................................................... 30
Configuring Naming Services ..................................................................... 31
Configuring the Naming Service Server Image ....................................... 31
Configuring a Naming Service Client Image .......................................... 32
Advanced Configurations ......................................................................... 34
Types of Settings .................................................................................... 34
Filename .............................................................................................. 34
Port Number ....................................................................................... 34
Sharing an Interface Repository ............................................................. 35
About the Master Interface Repository .................................................. 35
Systems That Share a Master Interface Repository .............................. 35
Establishing a Shared Repository ........................................................... 36
Interface Version Control ....................................................................... 36
Advanced Configuration Example ........................................................... 36

Chapter 3 A Distributed Smalltalk Example

Preparing the Example ............................................................................ 38
Load the Example Parcel ......................................................................... 38
Configuring Images to Support the Example ........................................ 38
Running DSTSampleComputeService .................................................... 39
Exploring DSTSampleComputeService ............................................... 40
Class Methods: naming ......................................................................... 40
createDefaultService .......................................................................... 40
destroyDefaultService ....................................................................... 41
Instance Methods: initialize-release ..................................................... 42
Instance Methods: computing ............................................................... 42
slowComputationWith:and: ................................................................ 42
Adding IDL Interfaces ............................................................................ 42
Editing the Sample IDL ......................................................................... 43
Generating IDL Code ........................................................................... 44
Running DSTSampleComputeService with IDL ................................... 45
Contents

Instance Methods: repository ................................................................. 46
abstractClassID .................................................................................. 46
CORBAName .................................................................................... 46
IDL Definition: DSTRepository>>DSTSampleComputeService .............. 47

Chapter 4 System Architecture 48

Summary of Services ............................................................................ 48
Object Services and Policies ................................................................. 49
Naming Service .................................................................................. 49
Event Notification Service ................................................................. 49
Basic Lifecycle .................................................................................. 49
Concurrency Control Service ............................................................... 50
Transaction Service ........................................................................... 50
Persistence ......................................................................................... 50
Implementation of the CORBA Specification .......................................... 50
Components of the ORB ................................................................. 50
 Interface Definition Language (IDL) Compiler ..................................... 50
 Interface Repository (IR) ................................................................... 51
 Static Invocation Interface (SII) .......................................................... 51
 Dynamic Invocation Interface (DII) .................................................... 51
 CORBA Terminology .......................................................................... 51
 Distributed Smalltalk’s ORB Implementation ....................................... 53
 Interface Definition Language (IDL) Compiler ..................................... 54
 Interface Repository (IR) ................................................................... 54
 Invocation Interface ........................................................................... 54
 Sending Messages via Surrogates and Object References .................... 54
 Implementing Surrogate Objects — DSTObjRef .................................... 55
 Object Identification .......................................................................... 56
 Object Creation Using Factories and Factory Finders ......................... 56
 Marshalling and Unmarshalling .......................................................... 57

Chapter 5 The DST Development Process 58

Choosing a Paradigm ............................................................................ 58
Issues in Distributed Computing .......................................................... 59
Optimizing Distributed Resources ....................................................... 59
Remote System Autonomy ................................................................. 59
Shared Objects .................................................................................. 60
Interoperability Through Standards Compliance .................................... 60
Creating Applications with Distributed Smalltalk .................................. 60
## Chapter 6  Designing and Implementing 62

General Design ................................................................. 62  
Sharing Objects ................................................................. 62  
Creating and Destroying Objects ......................................... 62  
Referencing Remote Objects .............................................. 63  
Performance Considerations .............................................. 63  
Avoiding Traps .................................................................. 64  

## Chapter 7  Implicit Invocation Interface (I3) 65

How I3 Works ................................................................. 65  
Required Methods .......................................................... 66  
Instance Methods ............................................................ 67  
  passInstVars ................................................................... 68  
Return Values ................................................................. 68  
Passing Classes ............................................................... 68  
I3 Instances and Garbage Collection .................................... 69  
I3 Traps ........................................................................... 69  

## Chapter 8  Defining IDL Interfaces 70

Overview ........................................................................... 70  
Comparing Smalltalk & IDL .............................................. 71  
Basic IDL Syntax ............................................................. 71  
  IDL Specification .......................................................... 72  
Declaring Modules ........................................................... 72  
  Example .......................................................................... 72  
  Getting Information About a Module ............................... 73  
Declaring Interfaces ........................................................ 73  
  Example .......................................................................... 74  
  Inheritance ...................................................................... 74  
    Inheritance Syntax ....................................................... 74  
  Forward Declaration ....................................................... 75  
  Pass by Reference .......................................................... 76  
Declaring Constants ......................................................... 76  
Declaring Data Types ....................................................... 80  
  Using Declarators to Give a Name to a Type .................... 80  
    typedef ......................................................................... 81  
      Example ...................................................................... 81  
    Simple Types ................................................................ 82  
      Base Data Types ......................................................... 82  
    Template Types .......................................................... 82  
    Strings ......................................................................... 83
Contents

Sequence .......................................................... 83
Example ...................................................... 84
Constructed Types .............................................. 85
Structures .......................................................... 85
Enumerations ..................................................... 86
Discriminated Unions ........................................ 88
Declaring Operations ........................................ 91
Example ........................................................ 92
Raises Expressions ........................................... 93
Context Expressions ........................................ 94
Declaring Exceptions ......................................... 95
User-Defined Exceptions .................................. 95
Example ....................................................... 95
Standard Exceptions ......................................... 96
Declaring Attributes ......................................... 99
Example ....................................................... 99
Inheritance ........................................................ 100
IDL Preprocessing ........................................... 104
#include .......................................................... 105
CORBAModule .................................................. 105
Names and Scopes ............................................ 105
IDL Traps .......................................................... 108
  Magnitude Mismatches .................................. 109
  Mismatched IDL Interfaces and Smalltalk Selectors 109
  Inheritance and Overriding Operations ............... 109
  Passing Values and References: Interfaces and Structures 109
SmalltalkTypes ............................................... 110
IDL void and Smalltalk nil ............................... 110

Chapter 9  Mapping of IDL to Smalltalk 112

Overview ....................................................... 112
Constraints on Smalltalk Mappings ....................... 112
Default Mapping for IDL to Smalltalk .................... 113
Handling Return Values .................................... 114
  Memory Usage .............................................. 114
  Limitations .................................................. 115
Mapping of IDL Elements to Smalltalk .................. 115
SmalltalkTypes .............................................. 116
Mapping for Interface ....................................... 117
  CORBAName Method ................................... 117
  Getting Information About an Interface ............... 117

Distributed Smalltalk Application Developer’s Guide
Contents

Selector Pragma ................................................................. 141
Access Pragma ................................................................. 141
About IDL and DSTRepository ........................................... 141
Editing the Interface Repository ....................................... 142
IDL Mapping to Smalltalk .................................................. 142

Chapter 10  Working with Object Interfaces  144
Making a Class a Factory .................................................... 144
Adding an Interface to the Interface Repository .................. 145
Creating an IDL Module Using the IDL Generator ............... 148
Refining the Module ......................................................... 150
Interface Repository Browser ............................................ 150
Opening the Browser ........................................................ 150
IR Browser Icons ............................................................ 151
IR Browser menus ............................................................ 152
Action menu ................................................................. 152
Edit menu ................................................................. 152
View menu ................................................................. 152
Importing IDL files .......................................................... 153
Setup for Preprocessing .................................................... 153
Annotate the IDL with Pragmas Where Necessary ............... 154
Avoiding Interface Problems ............................................. 154
Keeping Interface Repositories Updated ........................... 154
Edit Lock ................................................................. 154

Chapter 11  Initialization Service  155
Programmatically Initializing, Starting, and Stopping the ORB ................................................ 155
Getting Remote ORB References .................................... 157
Initial Object References ..................................................... 157
Distributed Smalltalk Implementation .................................. 158
ORB Utility Methods ......................................................... 158

Chapter 12  Naming Service  160
What Constitutes a Name? .................................................. 161
Name Components ......................................................... 161
Name Contexts and Naming Graphs ..................................... 161
Naming Service Operations .............................................. 162
Creating Names ............................................................. 162
Binding and Unbinding ..................................................... 163
Resolving and Listing Contexts ......................................... 163
Chapter 13  Event Notification 166

Overview ................................................................................................................. 166
   Need for Event Notification in a Distributed System ........................................ 166
   Terminology .......................................................................................................... 167
Event Channel ............................................................................................................ 168
   Multiple Event Channels ..................................................................................... 168
   Event Channel Administration ............................................................................ 168
Push and Pull Models .................................................................................................. 168
   Push ...................................................................................................................... 168
   Pull ....................................................................................................................... 169
   Disconnect to Terminate Communications ....................................................... 169
Consumers and Suppliers ......................................................................................... 170
   Proxies ................................................................................................................. 170
Event Data ................................................................................................................... 171
Using Events .............................................................................................................. 171
Example Code for Events .......................................................................................... 172
   Example: Connecting a Push Consumer to a Channel .................................... 172
   Example: Connecting a Pull Consumer to an Event Channel ....................... 172
   Example: Connecting a Push Supplier to a Channel ........................................ 173
   Example: Testing the Event Example ................................................................ 173
Using Typed Events .................................................................................................... 173
   Example: Connecting to a Channel .................................................................. 174
   Example: Implementing a Typed Push Connection ........................................... 174
      Typed Push Supplier and Interface ................................................................. 174
      Corresponding Typed Push Consumer and Interface .............................. 175
   Example: Implementing a Typed Pull Connection ............................................ 176
      Typed Pull Supplier and Interface ................................................................. 177
      Corresponding Typed Pull Consumer and Interface .............................. 177
   Example: Determining Quality of Service ....................................................... 178
Interfaces .................................................................................................................... 179
Implementation ......................................................................................................... 181

Chapter 14  Basic Lifecycle 182

Overview ................................................................................................................... 182
   Terminology .......................................................................................................... 183
Lifecycle Operations .................................................................................................. 183
Create ......................................................................................................................... 183
Contents

Copy and Deep Copy ................................................................. 183
Move ....................................................................................... 184
Destroy .................................................................................... 184
Throw Away .......................................................................... 184
Externalize and Internalize ......................................................... 184

Creating Objects ...................................................................... 185
COS on Factories and Factory Finders ...................................... 185
Distributed Smalltalk’s Implementation ...................................... 185
Examples: With and Without the Factory Representative ......... 186
  Example 1: Stringified Object Reference ........................ 186
  Example 2: Naming Service as Registry ....................... 187
  Using FactoryFinder Directly ........................................... 188
  Using the Factory Representative—Option #1 ................ 188
  Using the Factory Representative—Option #2 ............... 189
Example: Copying an Object ...................................................... 189
Commentary .......................................................................... 190

Chapter 15 Concurrency Control Service 191

Overview ................................................................................... 191
Terminology .............................................................................. 192
Lock Modes .............................................................................. 193
Lock Mode Compatibility ....................................................... 194
Multiple Lock Semantics ....................................................... 194
Locks and LockSets ................................................................. 195
Interfaces ............................................................................... 196
Implementation ........................................................................ 196
Using Distributed Smalltalk Concurrency Service ................. 196
  Using the Class DSTResourceManager ........................ 197
    Creating Locks .............................................................. 197
    Acquiring Locks .......................................................... 197
    Releasing Locks ........................................................... 198
    Destroying Locks .......................................................... 198
  Using Transactional Locksets .............................................. 199

Chapter 16 Transaction Service 200

Overview .................................................................................. 200
Distributed Smalltalk’s Implementation of Transactions .......... 201
Terminology ............................................................................ 204
Transactional Applications ..................................................... 206
Transactional Client ................................................................. 208
Contents

Possible Causes ................................................................. 229
Solutions .............................................................................. 229
Coding Style Hints ............................................................... 229
  Method Size ........................................................................ 229
  Multiple Inheritance in DSTRepository ......................... 229
  Blocks ................................................................................ 229

Chapter 18 Creating a Deployment Image ..................... 230
  Overview ............................................................................. 230
  Design and Preparation ..................................................... 230
    Possible Runtime Configurations .................................. 230
    Providing a Desktop Icon .............................................. 231
  Creating a Deployment ORB Image ................................. 232
    Candidate Classes for Removal ................................... 232
    Steps for Creating a Deployment Image ...................... 234
  Optimizing Runtime Applications .................................. 235
    Exception Handling ....................................................... 235
    Minimizing Footprint ..................................................... 236

Chapter 19 Troubleshooting ............................................. 237
  Overview ............................................................................. 237
  Marshalling and Unmarshalling Errors ......................... 237
    Symptoms ........................................................................ 237
    Possible Causes ........................................................... 238
    Solutions ......................................................................... 238
  Object Availability Exceptions ...................................... 238
    Symptoms ........................................................................ 238
    Possible Causes ........................................................... 239
    Solutions ......................................................................... 239
  Synchronization Problems .............................................. 239
    Symptoms ........................................................................ 239
    Possible Causes ........................................................... 239
    Solutions ......................................................................... 240
  Dangling References ....................................................... 240
    Symptoms ........................................................................ 240
    Possible Causes ........................................................... 240
    Solutions ......................................................................... 240
  Remote Access to Overridden Methods ......................... 240
    Symptoms ........................................................................ 240
    Possible Causes ........................................................... 240
    Solutions ......................................................................... 241
About This Book

This manual gives an overview of the Distributed Smalltalk development process, and describes programming resources for building distributed applications.

Audience

Distributed Smalltalk is a CORBA 2.1-compliant framework for developing distributed applications, and supports several of the primary CORBA Object Services (COS).

This book is written for experienced Smalltalk developers who are writing their first Distributed Smalltalk application. Readers should have a good understanding of VisualWorks®, review the VisualWorks manuals for more information. For additional help, a large number of books and tutorials are available from commercial book sellers and on the worldwide web. In addition, Cincom and some of its partners provide VisualWorks training classes. This book does not assume any prior knowledge of Distributed Smalltalk or of CORBA.

Be sure to read Chapters 1-4 to understand basic Distributed Smalltalk concepts. After reading Chapter 4, decide whether to use the Implicit Invocation Interface (I3) or IDL interfaces in your application and then read the corresponding chapter.
Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>template</em></td>
<td>Indicates new terms where they are defined, emphasized words, book titles, and words as words.</td>
</tr>
<tr>
<td><em>cover.doc</em></td>
<td>Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).</td>
</tr>
<tr>
<td><em>filename.xwd</em></td>
<td>Indicates a variable element for which you must substitute a value.</td>
</tr>
<tr>
<td><em>windowSpec</em></td>
<td>Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.</td>
</tr>
<tr>
<td><em>Edit menu</em></td>
<td>Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.</td>
</tr>
</tbody>
</table>

Special Symbols

This book uses the following symbols to designate certain items or relationships:

<table>
<thead>
<tr>
<th>Examples</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File ➔ New</td>
<td>Indicates the name of an item (New) on a menu (File).</td>
</tr>
<tr>
<td>&lt;Return&gt; key</td>
<td>Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.</td>
</tr>
<tr>
<td>&lt;Select&gt; button</td>
<td></td>
</tr>
<tr>
<td>&lt;Operate&gt; menu</td>
<td></td>
</tr>
<tr>
<td>&lt;Control&gt;-&lt;g&gt;</td>
<td>Indicates two keys that must be pressed simultaneously.</td>
</tr>
<tr>
<td>&lt;Escape&gt; &lt;c&gt;</td>
<td>Indicates two keys that must be pressed sequentially.</td>
</tr>
<tr>
<td>Integer&gt;&gt;asCharacter</td>
<td>Indicates an instance method defined in a class.</td>
</tr>
<tr>
<td>Float class&gt;&gt;pi</td>
<td>Indicates a class method defined in a class.</td>
</tr>
</tbody>
</table>
Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Select&gt; button</td>
<td>Select (or choose) a window location or a menu item, position the text cursor, or highlight text.</td>
</tr>
<tr>
<td>&lt;Operate&gt; button</td>
<td>Bring up a menu of operations that are appropriate for the current view or selection. The menu that is displayed is referred to as the &lt;Operate&gt; menu.</td>
</tr>
<tr>
<td>&lt;Window&gt; button</td>
<td>Bring up the menu of actions that can be performed on any VisualWorks window (except dialogs), such as move and close. The menu that is displayed is referred to as the &lt;Window&gt; menu.</td>
</tr>
</tbody>
</table>

These buttons correspond to the following mouse buttons or combinations:

<table>
<thead>
<tr>
<th>3-Button</th>
<th>2-Button</th>
<th>1-Button</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Select&gt;</td>
<td>Left button</td>
<td>Left button</td>
</tr>
<tr>
<td>&lt;Operate&gt;</td>
<td>Right button</td>
<td>Right button</td>
</tr>
<tr>
<td>&lt;Window&gt;</td>
<td>Middle button</td>
<td>&lt;Ctrl&gt;+&lt;Select&gt;</td>
</tr>
</tbody>
</table>

**Note:** This is a different arrangement from how VisualWorks used the middle and right buttons prior to 5i.2.
If you want the old arrangement, toggle the Swap Middle and Right Button checkbox on the UI Feel page of the Settings Tool.

---

**Getting Help**

There are many sources of technical help available to users of [supportweb@cincom.com](mailto:supportweb@cincom.com). Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.
Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available. For more information, send email to supportweb@cincom.com.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The version id, which indicates the version of the product you are using. Choose Help → About VisualWorks in the VisualWorks main window. The version number can be found in the resulting dialog under Version Id:

- Any modifications (patch files) distributed by Cincom that you have imported into the standard image. Choose Help → About VisualWorks in the VisualWorks main window. All installed patches can be found in the resulting dialog under Patches:

- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select copy stack in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail
To get technical assistance on VisualWorks products, send email to supportweb@cincom.com.

Web
In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone
Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.
Non-Commercial Licensees

VisualWorks Non-Commercial is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are not alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:
  
  vwnc-request@cs.uiuc.edu

  with the SUBJECT of "subscribe" or "unsubscribe".

- An excellent Smalltalk archive is maintained by faculty and students at UIUC, who are long-time Smalltalk users and leading lights in the Smalltalk community, at:
  
  http://st-www.cs.uiuc.edu/

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:
  
  http://wiki.cs.uiuc.edu/VisualWorks

- A variety of tutorials and other materials specifically on VisualWorks at:
  

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincom.com/smalltalk/documentation

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.
Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available. For more information, send email to supportweb@cincom.com.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The version id, which indicates the version of the product you are using. Choose Help → About VisualWorks in the VisualWorks main window. The version number can be found in the resulting dialog under Version Id:.

- Any modifications (patch files) distributed by Cincom that you have imported into the standard image. Choose Help → About VisualWorks in the VisualWorks main window. All installed patches can be found in the resulting dialog under Patches:.

- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select copy stack in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to supportweb@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com
Telephone
Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees
VisualWorks Non-Commercial is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are not alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:
  
  vwnc-request@cs.uiuc.edu

  with the SUBJECT of “subscribe” or “unsubscribe”.

- An excellent Smalltalk archive is maintained by faculty and students at UIUC, who are long-time Smalltalk users and leading lights in the Smalltalk community, at:

  http://st-www.cs.uiuc.edu/

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

  http://wiki.cs.uiuc.edu/VisualWorks

- A variety of tutorials and other materials specifically on VisualWorks at:


The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.
Chapter - About This Book

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincom.com/smalltalk/documentation

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.
1

Introducing Distributed Smalltalk

Distributed Smalltalk is an integrated set of frameworks providing an advanced object-oriented environment for rapid development and deployment of multi-user, enterprise-wide distributed applications. Distributed Smalltalk provides a superior environment for rapid prototyping, application development, and deployment of CORBA-compliant applications.

Distributed Objects

Modern corporations have moved from the old mainframe paradigm, in which all programs ran on one central computer, to decentralized computing, in which many different computers cooperate to run programs. In this distributed environment, it is no longer practical to design each program as a stand-alone system; programs must be designed to share and exchange data and services. Distributed object systems can be both more robust and more powerful than either conventional object-oriented systems or conventional client-server systems.

Designs based on distributed objects offer all the power of object-oriented design, and in addition allow widely separated applications to collaborate. Distributed applications also facilitate load distribution among clients and servers and provide increased reliability by facilitating mirroring and replication.

What is CORBA?

In 1989, the Object Management Group (OMG) began to specify the Common Object Request Broker Architecture (CORBA). This standard defines common methods of communication between distributed objects on disparate platforms. A revised CORBA standard, version 2.0, was
agreed to in late 1994. By 1996, there were many different CORBA implementations on the market; such major software vendors as Oracle and Netscape have announced support for CORBA interoperability. The CORBA standard has continued to undergo revision and update since then.

An application that is based on the CORBA infrastructure is well-positioned to be integrated into today’s diverse computing environment.

**Why Distributed Smalltalk?**

Distributed Smalltalk allows you to develop applications that are compliant with the CORBA 2.1 standard, while offering the superior application-development facilities of VisualWorks. Distributed Smalltalk hides some of the complexity of CORBA, while making its full power available to developers.

Distributed Smalltalk is a complete implementation of CORBA 2.0, with additions to comply to the 2.1 standard. Distributed Smalltalk’s CORBA compliance provides the basis for object- and application-interoperability. Distributed Smalltalk also offers the Implicit Invocation Interface (I3), an extension to the CORBA facilities that provides a more natural Smalltalk paradigm for developing distributed object systems.

**How Distributed Smalltalk Works**

An application written in Distributed Smalltalk is able to respond to service requests from remote systems and to request services in its turn. Remote entities that request services of an application need not be written in Distributed Smalltalk, as long as they use an OMG-standard Object Request Broker to make the requests.

The component objects that make up a Distributed Smalltalk application are often themselves distributed across several systems. These distributed objects can interact transparently, without regard to object location or platform.

To a running application, remote message sends appear to be taking place in the local image. What actually happens to implement a remote send is more interesting.

1. Arthur obtains an object reference for Bonita.
2. Arthur sends a message to the object reference.
3. The Object Request Broker on machine A (ORB A) translates the message and its arguments into a platform-neutral format ("marshalls the message") and transmits the message over the network to the ORB on machine B (ORB B).
4. ORB B translates the platform-neutral message back into Smalltalk ("unmarshalls the message") and sends the message to Bonita for processing.
5. Bonita processes the message and returns a value to ORB B.
6. ORB B marshalls the return value and transmits it over the network to ORB A.
7. ORB A unmarshals the return value and returns it to Arthur.

**CORBA Components**

CORBA 2.1 specifies core functions that are required of an Object Request Broker in order to support interoperable distributed computing. The CORBA specification requires the following core components, all of which Distributed Smalltalk supplies:
Chapter 1 - Introducing Distributed Smalltalk

- **Object Request Broker (ORB)**
  
The ORB is the brain of a CORBA implementation. It facilitates the transmission and interpretation of messages across diverse software and hardware platforms.

- **Interface Definition Language (IDL) Compiler**
  
The Interface Definition Language is used to define objects' public interfaces in a language- and platform-independent fashion, so that object services may be requested from any supported environment.

- **Internet Inter-Orb Protocol (IIOP)**
  
The Internet Inter-Orb Protocol is used to communicate between ORBs. An object running on one ORB can make requests of objects served by any other connected ORB, whether those objects were written in Smalltalk, C, C++, Ada, or COBOL. Developers can build distributed systems using multiple languages where appropriate. Thus, a Smalltalk object may request services from or provide services to a C++ object, an Ada object, or any object that supports an IDL interface.

- **Interface Repository (IR)**
  
The Interface Repository is the registry of distributable object interfaces for a given system. Any remotely accessible object has an interface in the Interface Repository. Interface Repository interfaces are described in IDL.

The CORBA standard also specifies optional object services (CORBA services) which may be provided by CORBA implementations. DST implements support for some of these, as described below.

**Object Request Broker (ORB)**

The Object Request Broker (ORB) is the key to distribution support. By providing an ORB on each system, Distributed Smalltalk makes the location of any object transparent to clients requesting services from the object.

When a message is sent to a local object, the activity is handled normally. When a message is sent to a remote object, the remote object's local object reference (a proxy created automatically by the ORB) intercepts the message, then uses the ORB to locate the remote object and communicate with it. Results returned to the calling object appear exactly the same, whether the message went to a local or remote object.

An ORB performs all of the following tasks:
CORBA services

- **Marshalling and unmarshalling messages** (translating objects to and from byte streams for network transmission)
- Locating objects in other images or systems
- Routing messages between surrogates and the objects they represent

While a request is active, both client and server ORBs exchange packet information to track the course of the request and resolve any network or transmission errors that might occur.

**Interface Definition Language (IDL) Compiler**

When distributed objects collaborate in an application, they interact by sending messages to one another’s interfaces. These interfaces are described in the Interface Definition Language; this makes the interface descriptions language-independent. Because external clients have access to an object’s services only through the object’s interface, the implementation of the object is private. This privacy provides a variety of benefits, including security, language independence, encapsulation, and the freedom to modify the implementation of how a service is performed without external repercussions. The IDL compiler translates interface definitions into the objects used by the ORB and Interface Repository.

**Internet Inter-Orb Protocol (IIOP)**

The Internet Inter-Orb Protocol (IIOP) is the protocol used for communication between ORBs. The IIOP is based on TCP/IP, and adds some additional message exchanges to provide a backbone protocol.

**Interface Repository (IR)**

The Interface Repository stores all of the IDL interfaces for objects available through an ORB.

CORBA services

Object services extend the core ORB capabilities to support more advanced object interaction. Distributed Smalltalk implements several of OMG’s **CORBA services**. These services extend CORBA to provide protocols for common operations like creating, exporting, and destroying objects (Lifecycle), locating objects (Naming), managing transactions, and providing asynchronous event notification.

Distributed Smalltalk provides the following CORBA services:
Naming
Assigns an object a unique user-visible name. Names are used to identify and locate both local and remote objects.

Basic Lifecycle
Provides standard mechanisms for creating and initializing, deleting, externalizing (preparing for transmission to remote systems), and internalizing.

Event Notification
Allows objects to notify each other of an interesting occurrence using an agreed protocol and set of objects.

Concurrency Control
Enables multiple clients to coordinate access to shared resources. This service supports two modes of operation: transactional and non-transactional. Concurrent use of a resource is regulated with locks. Each lock is associated with a single resource and a single client. There are several lock modes: read/write/update and intention-mode.

Transaction
Provides the infrastructure supporting the ACID (atomicity, consistency, isolation, and durability) transaction properties for operations among multiple objects. There is support for multiple transaction models. This service also provides transaction wrappers for existing applications and support for transaction monitors.

Implicit Interface Invocation (I3)
DST includes an implicit interface invocation mechanism that provides Smalltalk-to-Smalltalk communication without the need for IDL. This is useful especially for rapid prototype development between a DST server and client. The interface is then easily translatable into IDL for more general deployment.
Development Tools

Distributed Smalltalk extends VisualWorks with tools that support the development and testing of distributed applications. These include a distributed debugger, a remote browser, an automated IDL generator, and other tools.
Installing and Configuring DST

This chapter describes how to install Distributed Smalltalk into a VisualWorks image, and how to configure DST images as Naming Service servers and clients. This is the recommended configuration.

Installing Distributed Smalltalk

Distributed Smalltalk is provided as a collection of parcels with the standard VisualWorks distribution. Installation of the DST parcels is optional. Refer to the VisualWorks Installation Guide for instructions on installing the Distributed Smalltalk components.

DST Directory Structure

The DST parcels are installed with VisualWorks in a separate subdirectory, dst/.

Under this directory are two further subdirectories, icons/unselect. This directory contains icon files required by DST, and so must be present.

Loading DST into the Image

To use the Distributed Smalltalk components, you load them into the VisualWorks image, typically using the Parcel Manager. The components you load depends on which features of DST you intend to use. These four commonly loaded components are listed in the Parcel Manager Distributed Computing folder:

- **DST_COS_Services**
  - DST with CORBA COS services.

- **DST_I3**
  - DST with Implicit Invocation Interface (I3) support, only.
Configuring Naming Services

When dealing with multiple images on a single system or between systems, it is necessary to configure the following settings. You only need to do this configuration if you communicate with or connect to other images (ORBs).

IIOP Transport
Internet Inter-ORB Protocol that enables objects and applications to interoperate over a network with other OMG CORBA applications.

Naming Service
This service supports naming and locating local and remote objects.

Repository
A service used to share the interface repository on a remote system.

Note: NCS Transport and Security features of previous releases are obsolete, but are available for a limited time in the obsolete/dst directory.

Note: Windows 98 typically cannot run two DST orbs on the same machine, so you will need to run the server on one and the client on another.

Configuring the Naming Service Server Image

1. Create a new image:
   a. Start a Distributed Smalltalk image. This image should have DST_Tools_Development loaded, since it needs the DST Tools.
   b. Save a copy of that image, to be configured as the Naming Service server image.
Chapter 2 - Installing and Configuring DST

In the main window, select File → Save As. In the resulting dialog, supply the name of the new image, such as “naming”.

2 If the DST main window is not open, execute the expression
   DSTTool open.

3 Open the DST Settings tool (File → Settings).

4 Set the IIOP Transport settings.
   a From the Distributed Smalltalk main window, in the Transports menu, make sure that IIOP Transport is turned on (checked).
   b In the Settings notebook, select the IIOP Transport page.
   c In the IIOP Port Number box, select Configured To. Leave the port number unchanged.
   d If you changed a setting, click Accept.

5 Set the Naming Service to Local.
   a In the Settings notebook, select the Naming Service page.
   b Click Local.
   c If you changed the setting, click Accept.

6 Set the Repository to Local.
   a In the Settings notebook, select the Repository page.
   b Click Local.
   c If you changed the setting, click Accept.

7 Save the configured image.

8 Click Start in the Distributed Smalltalk main window to start the request broker.

**Configuring a Naming Service Client Image**

1 Create a new image:
   a Start a Distributed Smalltalk image. This image should have DST_Tools_Development loaded, since it needs the DST Tools.
   b Save a copy of that image, to be configured as the Naming Service client image.

   In the main window, select File → Save As. In the resulting dialog, supply the name of the new image, such as “client01”.
2 If the DST main window is not open, execute the expression
DSTTool open.
3 Open the DST Settings tool (File → Settings).
4 Set the IIOP Transport settings:
   a From the Distributed Smalltalk main window, in the Transports
      menu, make sure that IIOP Transport is turned on.
   b In the Settings notebook, select the IIOP Transport page.
   c In the IIOP Port Number box, select Dynamically Allocated.
   d If you changed the setting, click Accept.
5 Set the Naming Service settings:
   a In the Settings notebook, select the Naming Service page.
   b Click Hostname.
   c Enter the hostname of the Naming Service server image
      (localhost, the machine name, and the IP address, are all
      acceptable entries).
   d Check that the port shown is the port that the Naming Service
      server image is configured to use for IIOP transport.
   e Click Accept.
6 Set the Repository settings:
   a In the Settings notebook, select the Repository page.
   b Click Local.
   c If you changed the setting, click Accept.
7 Save the configured image.
8 Click Start in the Distributed Smalltalk main window to start the
   request broker.
Advanced Configurations

This section shows how to configure a set of Distributed Smalltalk images so they can share the responsibility of providing locating, naming, repository and security services. The standard configuration, in which all such responsibilities are centralized in a single image, is described above, in “Configuring Naming Services”. This section deals with more selective arrangements.

The DST Settings are used to obtain one or more of the services from a foreign image. First, this chapter explains the kinds of settings that are involved, and limitations that apply to them. Next, the benefits of sharing a master image’s interface repository are explored. Finally, this chapter presents an example configuration involving three images that obtain services from one another.

Types of Settings

Each ORB image in a network of ORBs needs to know which ORB provides the Naming and Repository services and, if in use, the optional Security service. The DST Settings are used to identify the provider of each service, using either the provider’s hostname or the name of a file containing an object reference to the provider. This section discusses each of these ways of identifying a service provider.

In addition, communications between ORBs require the use of a port number, which is also described in this section.

Filename

Some configuration pages in the DST Settings notebook provide a Filename field. This field is expected to contain the name of a file. The file is expected to contain an object reference to the ORB that supplies a particular service.

To export an object reference to a file, make sure your ORB is running, then execute the following:

```smalltalk
    ORBObject referenceToFile: 'filename' object: anObject
```

To get an object reference from a file, make sure your ORB is running, then execute the following:

```smalltalk
    ORBObject referenceFromFile: 'filename'
```

Port Number

In the DST Settings, port numbers can be dynamically allocated by Distributed Smalltalk, or you can specify them explicitly.
If you choose to let the system allocate a port number, it will change each time the ORB is restarted.

If you choose to specify a port number, it must be an integer between 1024 and 65536. Port numbers between 1 and 1023 are reserved. The port number can be reused in the Naming Service configuration.

**Sharing an Interface Repository**

In Distributed Smalltalk, each image contains an interface repository, which provides an object interface for each distributed service that is requested. Successful interaction among distributed objects requires that the interface repositories in all participating systems be identical. When they are not identical, errors will be reported and communication will be interrupted.

The shared interface repository service helps you maintain identical interface repositories on different systems. When you set up your systems to use a shared repository, one of the connected systems holds the *master interface repository*; other systems hold a subset of the master interface repository, which they build on an as-needed basis.

**About the Master Interface Repository**

Generally, the master interface repository should be on a system that is:

- Available at all times
- Easily accessed on the local- or wide-area network (to optimize communications overhead)
- Not stripped of the compiler and repository classes (classes in categories CORBA-Compilers and CORBA-Repository)

**Systems That Share a Master Interface Repository**

Other systems, which share the master interface repository, each start with a minimum set of interfaces in their local interface repositories (the minimum required for basic communications). When another interface is requested, it is loaded along with its superclass interfaces. These newly loaded interfaces are cached locally and retained for the remainder of the session. At the end of a session (when you stop the ORB), the interface cache is cleared.

When you are preparing to deploy an application that will run on a system with a shared interface repository, you can remove all classes in categories CORBA-Repository and CORBA-Compilers.
Establishing a Shared Repository

By default, each image has its own (nonshared) interface repository. You can change an image to use a shared interface repository by following these steps:

1. If a session is running, stop it. In the Request Broker panel, click on Stop.
2. Open a DST Settings window by selecting File ➔ Settings in the Distributed Smalltalk main window.
3. In the Settings notebook, select the Repository page.
4. Supply either a Filename or a Hostname to identify the master interface repository.

Interface Version Control

Distributed objects can only communicate correctly through compatible interfaces. Each interface is identified by a RepositoryId. One of the fields in the RepositoryId is the version number and this version number can be modified by using the VERSION pragma. The default is version 1.0.

For more information on using version control, see “VersionPragma” in Chapter 9, “Mapping of IDL to Smalltalk.”

Advanced Configuration Example

As an example of an advanced configuration, suppose you want to connect three separate images as follows:

- Image 1 provides the user security database for all images.
- Image 2 provides the naming service for all images.
- Image 3 provides the repository for image 2.

In this case, you should configure the images as follows:

1. In Image 1:
   a. Configure the IIOP ports to known ports.
   b. Set the Naming Service, Repository and Security settings to Local. (The Naming Service will be reset later.)
   c. Start the ORB.

2. In Image 2:
   a. Let the IIOP ports be dynamically allocated.
b  Set the Naming Service to Local.
c  For now, set the Repository to Local.
d  Set Security to Image 1’s Hostname.
e  Start the ORB.

3  In Image 1:
   a  Set the Naming Service to Image 2’s Hostname.

4  In Image 3:
   a  Let the IIOP ports be dynamically allocated.
   b  Set the Naming Service to Image 2’s Hostname.
   c  Set the Repository to Local.
   d  Set Security to Image 1’s Hostname.
   e  Start the ORB.

5  In Image 2:
   a  Set the Repository to Image 3’s Hostname.
This chapter examines a simple DST application, DSTSampleComputeService. DSTSampleComputeService is designed to perform some expensive computation on a dedicated compute server and return the results to the caller. The server provides a named service that performs the computation, which a client applications can then request.

The initial implementation uses the DST I3 mechanism, for simple communication between two Smalltalk images. Later in the chapter, the interface is converted to IDL, so the service can be accessed by non-Smalltalk programs using CORBA services.

Preparing the Example

Load the Example Parcel

DSTSampleComputeService is installed separately from the development tools. To load the example, load both the DST_Sample and the DST_Tools_Development parcels using the Parcel Manager.

Configuring Images to Support the Example

Before you test the example, you must configure two Distributed Smalltalk images: a server image that runs the Naming service, and a client image requests the service from the server. This configuration is described in Chapter 2, “Installing and Configuring DST” under “Configuring Naming Services”.

After both images are configured, go to the Distributed Smalltalk window in each image, select Transports:I3 to turn the Implicit Invocation Interface on, and click Start.
**Running DSTSampleComputeService**

The DSTSampleComputeService example contains both server and client functionality, for simplicity. In most applications, this functionality would be factored into separate classes.

The server functionality responds to a request for its named service, also called ‘DSTSampleComputeService’, performs a computation, and returns the result to the request client.

To run the example, do the following:

1. Enable the I3 transport (check Transports → I3) and click Start in the DST main windows of both images.

2. In a workspace in the server image, evaluate (Dolit) the message:

   ```smalltalk
   DSTSampleComputeService createDefaultService.
   ```

   This message creates an instance of the compute service and registers it with the ORB and Naming service.

3. In a workspace in the client image, evaluate the following messages:

   ```smalltalk
   | namingService aName objRef |
   namingService := ORBObj ect namingService.
   aName := DSTName on:
     (DSTName onString: 'DSTSampleComputeService'),
   objRef := namingService contextResolve: aName.
   Transcript cr; show:
     (objRef slowComputationWith: 5100000 and: 5200000) asString.
   ```

   Note that this is a very slow computation, and may take several minutes to return. The Transcript should show 5148001.

4. To disable the service, evaluate this message in a workspace in the server image:

   ```smalltalk
   DSTSampleComputeService destroyDefaultService
   ```
Exploring DSTSampleComputeService

DSTSampleComputeService is designed to run some expensive computation on a dedicated compute server. It provides a service method, slowComputationWith:and:, which invokes another method to perform the calculation, and returns the result. All other DSTSampleComputeService methods are support methods necessary to providing a distributed service or do the actual calculation.

Class Methods: naming

A crucial difference between developing applications in Distributed Smalltalk and in local Smalltalk is in retrieving well-known instances. In non-distributed Smalltalk, applications retrieve well-known instances from some globally accessible location in the local image. In order to share objects with other images, distributed applications must also retrieve shared objects from other images. The most common way to do this is through the CORBA Naming service.

DSTSampleComputeService manages its interactions with the Naming service through the class messages defaultService and destroyDefaultService in the naming protocol. DSTSampleComputeService uses a class variable, DefaultService, to hold the unique instance of the class. The class method createDefaultService returns the unique instance, creating it if necessary; destroyDefaultService destroys the instance.

createDefaultService

This method returns the unique instance of the compute service, creating it if necessary.

createDefaultService
   "Return the service. If no instance exists, create one and, register it with the naming service."

   "DSTSampleComputeService createDefaultService"

   | DefaultService |
   | isNil         |
   | ifTrue: [     |
   |   DefaultService := DSTSampleComputeService new. |
   |   ORObject namingService contextBind: self serviceName to: DefaultService |
   | ]             |

   | ^DefaultService |

If the instance already exists, createDefaultService returns it; otherwise, it creates the instance.
Before requesting services from the Naming service, `createDefaultService` must get an object reference to the Naming service itself. The method gets that reference from the class `ORBObject`. This class always holds valid references to the Interface Repository, Naming service, factory finder, and user security database.

After the instance exists, `createDefaultService` binds the instance to the name `DSTSampleComputeService`. Binding an instance to a name associates the name with the instance for the life of the image containing the Naming service.

After the instance has been created and bound to a name in the Name service server image (by sending the message `DSTSampleComputeService createDefaultService`), other images can get an object reference for `DSTSampleComputeService` by sending the `contextResolve:` message to the Naming service.

```smalltalk
| namingService aName objRef |
 namingService := ORBObject namingService.
 aName := DSTName on: (DSTName onString: 'DSTSampleComputeService').
 objRef := namingService contextResolve: aName.
```

**destroyDefaultService**

This message destroys the unique instance of the class.

```smalltalk
destroyDefaultService

"Remove the singleton instance of the service from the class variable, the naming service, and the lifecycle service."

"DSTSampleComputeService destroyDefaultService"

( DefaultService isNil )
 ifFalse:
   [ ORBObject namingService contextUnBind: self serviceName.
     DefaultService release.
     DefaultService := nil
   ]
```

The instance is destroyed in three steps. First the method unbinds the instance from the Naming service (destroys the association between the name and the instance), then it releases the instance, and finally it resets the instance variable to `nil`. An instance must be unbound before it is released.
Instance Methods: initialize-release

These are the standard instance creation and destruction methods.

initialize
   ‘Register instance with lifecycle service’
   DSTObjRef registerObject: self

release
   ‘Remove reference to object from lifecycle service.’
   DSTObjRef unRegisterObject: self

Because instances of this class must interact with remote objects, each instance should be registered with the ORB when it is created, and unregistered when it is no longer useful.

Instance Methods: computing

There are several computation methods, providing alternate calculations. At this point, we are only concerned with slowComputationWith:and:.

slowComputationWith:and:

The slowComputationWith:and: method invokes another method, CarmichaelNumber:, which actually performs the computation. Neither method contains DST-specific code.

\[
\text{aPositiveInteger1 to: aPositiveInteger2 do:}
\begin{cases}
   \text{:n | (self carmichaelNumber: n) ifTrue: [^n]} & \text{ifTrue: } \text{[^n]}
\end{cases}
\]

\[\text{^0}\]

Browse other calculation methods for implementations. These calculation methods can be invoked directly, by replacing the Transcript display and calculation line in the workspace with, for example:

\[
5100000 to: 5200000 do: [\text{:n | (objRef carmichaelNumber: n)}
\begin{cases}
   \text{ifTrue: [Transcript cr; show: n]} & \text{ifFalse: [Transcript cr; show: 0]}
\end{cases}
\]

Adding IDL Interfaces

This section demonstrates how to convert an I3 application into an IDL-interface application, by creating IDL interface definitions. Two approaches are offered.

If you already understand IDL, you may prefer to write your own. In this case, there is a sample IDL module provided, as well as required methods that you can rename and use for the rest of the example.
If you are new to CORBA and IDL, or prefer the convenience of a generator, you can use the IDL Generator tool. We will generate only a fragment of the full IDL provided in the sample module, but enough to see how it works and complete the example. You may browse the sample IDL for comparison with the generated IDL.

In this example, we will configure both the server and client images with their own local IDL repositories. Although it is possible for VisualWorks images to share a single repository, we will not use that feature here.

**Editing the Sample IDL**

A complete IDL module for the computational methods in DSTSampleComputeService is provided, together with the CORBAName and abstractID methods required to support service lookup. They are, however, named differently than is required. Rather than use the IDL Generator to produce these, you can simply rename the sample definitions and proceed.

To use these definitions:

1. In the server image, browse the DSTRepository class, DSTSampleComputeService0 method in the DEMO method category. Past the comment lines, find the line:
   ```
   module DSTSampleComputeService0 {
   ```
   and remove the 0, so it is now:
   ```
   module DSTSampleComputeService {
   ```
   Accept the change. You now have a DSTSampleComputeService method.

2. Also in the server image, browse the DSTSampleComputeService class in the browser, and select the repository method category. Edit the method selectors, changing:
   ```
   CORBAName0 to CORBAName, and
   abstractID0 to abstractID.
   ```
   Accept the changes. You now have new methods matching the required names.

3. In each client image, make the same change as in step 1, renaming the module to DSTSampleComputeService.

The CORBAName and abstractID methods are not needed in the client images, so you don’t need to do anything with them. You are now set to proceed with “Running DSTSampleComputeService with IDL”.
Generating IDL Code

To add IDL interfaces, use the Generate IDL tool. Do this in both the server and the client images.

1. In the server image, select Tools → Generate IDL in the DST main window.

2. In the Server Classes pane, find and select your server class, DST.DSTSampleComputeService. Enter a pattern, such as DST.DSTSample*, in the entry field to simplify the search.

Selecting a client class is not useful in this example. Selecting a client class and clicking Filter Methods with Client Classes would mark (check) methods in the list that are sent by a client class. This is useful when you know the application well, so know precisely which methods will be invoked remotely, in which case it may simplify the process. In our example, we only want to generate IDL for the slowComputationWith:and: message. (Exercise for later: Select the same class in the client classes list and see what is checked; then compare with the sample IDL in DSTSampleComputeService0.)

3. In the Target Methods pane, select slowComputationWith:and:.
Adding IDL Interfaces

4 In the Module name: field, enter a name for the module (e.g., DSTSampleComputeService).

5 Click the Generate IDL button.

When the IDL is generated, a class browser on DSTRepository opens, with the newly generated IDL displayed.

6 In the browser, scroll down to:

```smalltalk
SmalltalkObject slowComputationWithAnd (  
in SmalltalkObject a,  
in SmalltalkObject b );
```

Replace the first instance of “SmalltalkObject” with “unsigned long long” and the other two instances with “unsigned long”:

```smalltalk
unsigned long long slowComputationWithAnd (  
in unsigned long a,  
in unsigned long b );
```

Accept your changes.

7 (Server image only) In the IDL Generator, select Generate Glue.

This creates CORBAName and abstractID methods in DSTSampleComputeService that specify the mapping between the IDL interface and Smalltalk. See below, and Chapter 9, “Mapping of IDL to Smalltalk” for more information.

This is necessary in the server image only, because these methods are used by the ORB to locate the class and methods needed to service the request from the client.

8 Close the IDL Generator.

Browse DSTSampleComputeService and find the two new methods, CORBAName and abstractClassID, which are needed for IDL-interface execution. These are instance methods rather than class methods because it is sometimes useful for a class to have an abstractClassID different from those of its instances.

Running DSTSampleComputeService with IDL

You test the example with IDL interfaces exactly as before:

1 Turn off I3 in both server and client images, by selecting Transports → I3.

2 In a workspace in the server image, send the message

```
DSTSampleComputeService defaultService
```
This message creates an instance of the compute service and registers it with the ORB and Naming service.

3 In a workspace in the local image, send the following messages:

```
| namingService aName objRef |
```

```
namingService := ORBObject namingService.
aName := DSTName on:
  (DSTName onString: 'DSTSampleComputeService').
objRef := namingService contextResolve: aName.
Transcript cr; show:
  (objRef slowComputationWith: 5100000 and: 5200000) asString.
```

This is the same workspace as we used for the I3 example. The Transcript should show 5148001, though the processing may be slow since this is a slow computation.

4 In a workspace in the server image, remove the service from the ORB and naming services by sending the message

```
DSTSampleComputeService destroyDefaultService.
```

**Instance Methods: repository**

When browsing these methods, realize that they are generated, and so are not yet assigned to either a parcel or a package.Browse by Category or select Unparceled.

**abstractClassID**

The abstractClassID method returns a unique identifier for the class, for example (the identifier in your image will be different from the one shown here):

```
abstractClassId
  'return the abstract class Id of the receiver'
  ^'7883b3d3-3fe7-0000-02c7-bec093000000' asUUID
```

This number is generated uniquely for each class. Never copy this identifier from one class to another. Whenever you create a new class, create a new abstractClassID by sending the ORBObject newld message and copying the returned value into the new class's abstractClassID method.

**CORBAName**

The CORBAName method provides the link between a Smalltalk class and its IDL definition. When invoked, the method returns the symbol corresponding to the IDL definition's entry in the Interface Repository.
CORBAName
'return the name of my CORBA interface in the repository'
'"::DSTSampleComputeService::DSTSampleComputeServiceInterface'

**IDL Definition: DSTRepository>>DSTSampleComputeService**

To examine the IDL definition itself, browse the DSTRepository method DSTSampleComputeService. This method, which is written in IDL rather than Smalltalk, looks like this (with the lengthy comment omitted):

```idl
// DSTSampleComputeService
// This module defines the types and interfaces which form the
// DSTSampleComputeService
// protocol or service.
//
module DSTSampleComputeService {

    #pragma selector slowComputationWithAnd
    slowComputationWith:and:
    // This computation is not really slow enough to justify a remote
    // message send.
    unsigned long long slowComputationWithAnd (
        in unsigned long a,
        in unsigned long b);

};
```

**Note:** If you generated this method as described above, it should look like the above (plus comment). If instead you renamed the provided IDL method DSTSampleComputeService0, the code is somewhat longer, providing more IDL interfaces.

IDL syntax is discussed in detail in “Defining IDL Interfaces”. For now, examine the line that actually describes the method:

```idl
unsigned long long slowComputationWithAnd (
    in unsigned long a,
    in unsigned long b);
```

In particular, examine the datatypes that are assigned IDL datatypes to the return value and the arguments of the slowComputationWith:and: message. Getting the datatypes incorrect is a major issue in distributed computer, so take care to select appropriate values, as described in “Mapping for Basic Data Types”.
Distributed Smalltalk lets you develop and deliver *distributed* applications, that is, applications built with objects that may be running at different locations and on different systems. To do this, Distributed Smalltalk provides a range of services, from example code and developer’s tools, to a full implementation of industry standards for distributed object systems.

<table>
<thead>
<tr>
<th>Service layer</th>
<th>Service provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer Services</td>
<td>Administrative interface, Conversation monitoring, IDL interface generation,</td>
</tr>
<tr>
<td></td>
<td>Interface Repository browsing</td>
</tr>
<tr>
<td></td>
<td>Remote class browsing</td>
</tr>
<tr>
<td></td>
<td>Remote debugging</td>
</tr>
<tr>
<td></td>
<td>Simulated RPC testing</td>
</tr>
<tr>
<td>Object Services and Policies</td>
<td>Naming, Event notification</td>
</tr>
<tr>
<td></td>
<td>Lifecycle (basic and compound)</td>
</tr>
<tr>
<td></td>
<td>Concurrency &amp; Transactions</td>
</tr>
<tr>
<td></td>
<td>Relationships (links, containment)</td>
</tr>
<tr>
<td></td>
<td>Properties and property sets</td>
</tr>
<tr>
<td></td>
<td>Distributed Smalltalk specific:</td>
</tr>
<tr>
<td></td>
<td>Application objects and assistants</td>
</tr>
<tr>
<td></td>
<td>Presentation/ Semantic split</td>
</tr>
<tr>
<td>Core Services (CORBA)</td>
<td>Object Request Broker, with:</td>
</tr>
<tr>
<td></td>
<td>IDL compiler</td>
</tr>
<tr>
<td></td>
<td>Interface Repository</td>
</tr>
<tr>
<td></td>
<td>Static &amp; Dynamic invocation interfaces</td>
</tr>
</tbody>
</table>
Object Services and Policies

Distributed Smalltalk implements OMG’s *CORBA Services Volumes 1 and 2* specification that extends CORBA to provide protocols for common operations such as creating and destroying objects (lifecycle), locating objects (naming), and asynchronous event notification.

**Naming Service**

A standard for assigning each object a unique user-visible name. Naming policies set a standard for identifying and locating objects in both local and remote images, and in non-Distributed Smalltalk systems. Distributed Smalltalk provides a complete naming service based on the containment model of object organization. Application developers can use this service as implemented, or create their own naming services from Distributed Smalltalk’s basic naming policy and interface support. (The naming service is specified in OMG’s *CORBA Services*).

**Event Notification Service**

This service allows objects to notify each other of interesting occurrences using agreed protocols. The Distributed Smalltalk implementation of event notification provides decoupled, asynchronous communication between objects. It allows graceful object interactions even when objects are temporarily unavailable because the network or a remote system is “down.” Developers can extend the event notification service to support specific types or levels of service. (The event notification service is specified in OMG’s *CORBA Services*).

**Basic Lifecycle**

Standard ways for objects to implement activities such as create and initialize, delete, copy and move both simple and compound objects, externalize, and internalize.

<table>
<thead>
<tr>
<th>Service layer</th>
<th>Service provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication Support</td>
<td>RPC (NCS 1.5.1) conversations and packet transfer (Distributed Smalltalk to Distributed Smalltalk communication only)</td>
</tr>
<tr>
<td></td>
<td>Internet Inter-ORB Protocol (IIOP) — CORBA 2.0 specifies this protocol that enables objects and applications to interoperate over a network with other OMG CORBA applications</td>
</tr>
</tbody>
</table>
Concurrency Control Service

This enables multiple distributed objects to coordinate access to shared resources. There is support for two modes of operation: transactional and non-transactional. Concurrent use of a resource is regulated with locks. Each lock is associated with a single resource and a single client. There are several lock modes: read/write/update and intention-mode. (The concurrency control service is specified in CORBA services)

Transaction Service

The Transaction Service defines interfaces that allow multiple, distributed objects to cooperatively to provide transaction atomicity, consistency, isolation, and durability (ACID properties). There is support for multiple transaction models, including the flat and nested models. Also, transaction wrappers are provided for existing applications and support for transaction monitors. (The transaction service is specified in CORBA services).

Persistence

Distributed Smalltalk implements persistence within the Smalltalk image and does not conform to the CORBA services standard.

Implementation of the CORBA Specification

The Object Request Broker and the services it provides are at the core of Distributed Smalltalk. The ORB and its components provide the services that allow object systems, objects, and applications to interoperate.

Components of the ORB

CORBA 2.0 or the Object Management Group’s Common Object Request Broker Architecture, is the industry-standard specification for an ORB architecture. The CORBA specification describes the following core services as part of an ORB:

Interface Definition Language (IDL) Compiler

OMG has defined the IDL language to be independent of other programming languages. IDL is used for public interfaces, so that both service providers and service requestors can be written in Smalltalk, C, C++, or another language. See Chapter 5, “Defining IDL Interfaces”, and Chapter 9, “Mapping of IDL to Smalltalk” for instructions on using IDL and Smalltalk-to-IDL mappings.
Interface Repository (IR)
The registry of distributable object interfaces for a given system. Any object that remote objects can access has an interface in the Interface Repository. IR interfaces are written in the OMG-defined IDL, thus allowing language independence and interoperability for service providers (servers) and requestors (clients).

Static Invocation Interface (SII)
The SII supports requests for specific operations on remote objects and surrogate object creation. One of two invocation interfaces specified by CORBA, the SII is a better choice for Smalltalk applications, since the invocation of the static interface is dynamic in Smalltalk.

Dynamic Invocation Interface (DII)
The DII supports dynamic object request building and sending, thus providing an alternative to the SII for languages that do not support dynamic binding. The DII is included in Distributed Smalltalk for completeness, but it is not recommended.

While Distributed Smalltalk supports DII as specified, it is not recommended for use. Smalltalk supports dynamic binding, and thus DII is redundant and inefficient. (For more information on DII, see the implementation of the two classes that support dynamic invocation: ORBNVList and ORBRequest.)

CORBA Terminology
The CORBA specification defines the following terms and concepts:

ORB
CORBA specifies an Object Request Broker (ORB) to serve as the interface that isolates external service requestors (clients) from internal service providers (objects).

Clients send service requests to an ORB. When the ORB receives a request, the ORB translates it into the local implementation language (for example, Smalltalk), then locates the object that will provide the service, and forwards the request to the object. When the request is complete, control and output values are returned to the client.
object
An encapsulated entity that provides one or more services that can be requested by a client. Sometimes referred to as a server object or a CORBA object.

Smalltalk objects are more numerous than CORBA objects. While some of the Smalltalk objects are also CORBA objects, most provide support services within an image and are not accessible to external service requests from clients.

Objects in Distributed Smalltalk can function both as CORBA clients and CORBA objects, that is, both as requesters and providers of services.

client
An entity capable of requesting a service. A client need not be implemented in an object-oriented language. A client requesting services need not know where the object is located, nor how it is implemented.

Within and between Distributed Smalltalk images, objects interact as peers, without the implied hierarchy of clients and servers. Each object requests and/or provides services as necessary.

request
The communication between a client and an object. A request specifies: (1) an operation to be performed, (2) an object reference identifying the object that will perform the service, and (3, optionally) parameters that the object needs to perform the request.

interface
Defines which tasks (operations) a CORBA object can do and what information it needs to do those tasks. An object's interface is distinct from its implementation. Interfaces are stored in the Interface Repository, within the ORB.

IDL
*Interface Definition Language*. An implementation-neutral language specified by OMG as part of the CORBA core services. In all CORBA-compliant systems, interface definitions are written in IDL. By sending messages written in IDL between systems, objects implemented in different languages can communicate.
object reference

Identifies the server object, acting as an intermediary between client and object. An object reference identifies the same object each time the reference is used in a request. A single object may be denoted by multiple, distinct object references.

Distributed Smalltalk’s ORB Implementation

Distributed Smalltalk includes a complete implementation of CORBA 2.0 that includes Internet Inter-ORB Protocol (IIOP). This protocol enables objects and applications to interoperate over a network with other OMG CORBA applications. This section provides an overview of the OMG CORBA implementation.

Distributed Smalltalk’s ORB extends VisualWorks to support communication with objects that may be in the current local image, or in another image running either locally or remotely. Access to remote...
objects is generally transparent to the Smalltalk programmer; however, when defining an object that will be accessible remotely, the programmer must define an IDL interface (including operations) for it.

**Interface Definition Language (IDL) Compiler**

In Distributed Smalltalk, class IDLCompiler implements the IDL parser/compiler. IDL is a compiled language. (When you add or make changes to interface definitions in the Interface Repository, it will recompile.)

Note that the IDL compiler does not support ValueTypes.

**Interface Repository (IR)**

Class DSTRepository is the repository for CORBA object interfaces that are available for access by remote clients in Distributed Smalltalk. Interface definitions in DSTRepository specify the messages, or operations, that can be sent between objects in different images, as well as attributes, types, constants and exceptions. Before you can communicate with objects outside the current local image, you must define interfaces for these objects in the Interface Repository.

Of course, not all Smalltalk objects are CORBA objects, and only CORBA objects need an interface.

DSTRepository modules contain IDL (the Interface Definition Language); this is the only Distributed Smalltalk class where an application developer needs to write in IDL. For information on working with interfaces and IDL Syntax, see Chapter 10, “Working with Object Interfaces”.

**Invocation Interface**

As specified in CORBA, the invocation interface handles message sending and object invocation. While both the Dynamic and Static Invocation Interfaces are available in Distributed Smalltalk, it is more efficient to use the Static Invocation Interface, since dynamic binding is already provided by the Smalltalk language itself.

**Sending Messages via Surrogates and Object References**

In making a service request, a client need not know where a server object is located in order to send it messages. If the server is on the same system as the client (that is, local), the request is a normal Smalltalk request. However, if the server is on a remote system, the ORB gets involved to intercept and forward the message appropriately.
When a client requests a service, Distributed Smalltalk either sends the message directly to the object, if it is local, or to the object reference, if the object is remote. (An object reference identifies the server object, acting as an intermediary between client and object.) Object references act as publicly available surrogates for remote objects.

Specifically, since the surrogate cannot implement the operation (method) requested, it sends itself the message doesNotUnderstand:. Distributed Smalltalk traps the doesNotUnderstand: message and uses its own mechanisms (including the ORB and RPC) to locate and communicate with the remote object.

Results returned to the client appear exactly the same, whether the message went to a local or remote object. (The only difference a programmer or end-user sees is that the performance of a remote object request is, naturally, somewhat slower than that of a local request.)

Note: During remote execution, the local process thread is blocked until the result values have been received and decoded into internal Smalltalk representation. At that point, the local thread is resumed and local execution continues.

Implementing Surrogate Objects — DSTObjRef

DSTObjRef and its subclasses are instantiated to create the surrogates for remote objects. A direct subclass of Object, DSTObjRef provides the basic mechanisms for transparent distribution using the ORB RPC mechanism.

- In a local object message invocation (“local” with respect to the client system), message invocation is unaffected by these distribution mechanisms.

If the server object is a remote object, then the client holds an object reference that is an instance of DSTObjRef or one of its subclasses. Since this instance has none of the methods which the client object is expecting of the server, the local message send results in a #doesNotUnderstand: call instead. The method #doesNotUnderstand: is
overridden so that it actually starts the remote RPC operation (see also \#perform:on:, implemented in DSTObjRef, DSTObjRefWidened, and DSTObjRefLocal). Smalltalk objects may be referenced directly, or also via instances of DSTObjRef subclasses:

- Inactive — Local objects which normally exist within this image but which are currently residing as passive data on a mass storage device, such as an ODBMS, may be referenced by a suitable DSTObjRefInactive instance.
- Local — Local objects may be referenced by a DSTObjRefLocal so that messages sent to them will be processed by the ORB instead of the normal method invocation.
- Remote — Objects which exist on remote systems are accessed locally via an instance of a DSTObjRefRemote.
- Widened — Local objects may choose to allow a subset of their most derived interface operations to be made available to clients by returning a suitable DSTObjRefWidened instance as a result value rather than self.

**Object Identification**

In Distributed Smalltalk, the CORBAName is the tie between an interface and its corresponding implementation. That is, any object that has an interface (that is, a CORBA object), must implement the CORBAName method, which specifies the interface name. When the ORB receives an incoming request, it locates the interface in the Interface Repository, and the Smalltalk class by this CORBAName.

For information on writing and using CORBAName methods, see Chapter 10, “Working with Object Interfaces.”

**Object Creation Using Factories and Factory Finders**

As in standard Smalltalk, classes function as what CORBA refers to as factories, templates for creating object instances. The ORB uses its factory finder to locate a class. That is, when a client requests a service of an uninstantiated (non-existent) object, the ORB is able to instantiate the object if its class has been registered with the factory finder.

To register with the factory finder, a class needs the method abstractClassId, which returns a UUID (universally unique identifier) for the class.

For information on writing and using abstractClassId methods, see Chapter 10, “Working with Object Interfaces.”
Marshalling and Unmarshalling

The ORB is also responsible for converting Smalltalk objects into a byte stream for transmission to a remote server, a process called *marshalling*. (Unmarshalling creates a Smalltalk object from a marshalled byte stream.)
Choosing a Paradigm

DST offers two development paradigms: Implicit Invocation Interface (I3) and IDL-interface.

- IDL-interface supports all the interfaces specified in the CORBA 2.1 Smalltalk mapping. In this paradigm, developers explicitly describe the interfaces of distributed classes using the IDL language.
- I3 is an enhancement to IDL-interface that provides a more natural Smalltalk development paradigm. I3 allows developers to create distributed Smalltalk applications without having to define IDL interfaces.

Each of the two development paradigms has its advantages. The following table lists factors that might encourage you to choose one over the other.

*Comparison of I3 and IDL-Interface Development Paradigms*

<table>
<thead>
<tr>
<th>Design Goal</th>
<th>I3</th>
<th>IDL-interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype quickly and easily</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Use natural Smalltalk development paradigm</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Interoperate with non-Smalltalk applications or with other ORBs</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Achieve maximum performance</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Define external interfaces explicitly</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
It is straightforward to take an application developed with I3 and create explicit interfaces for all or part of it. This means that an application can be developed under any of three paradigms: I3; IDL-interface; or prototype in I3, then add IDL interfaces later.

### Issues in Distributed Computing

The architecture and implementation of Distributed Smalltalk facilitates, but does not dictate, distributed application design. To make your applications robust and efficient, you should understand both basic distribution concepts and how to use Distributed Smalltalk to your best advantage. This section introduces some of the issues you should consider while you are designing distributed applications.

#### Optimizing Distributed Resources

A distributed environment can provide a rich environment and a variety of resources. However, network traffic can be a performance bottleneck, unless your system optimizes communications, network traffic, and processing resource sharing. While the most powerful workstations and servers on a network should perform the most difficult processing, transferring information across the network should also be kept to a reasonable minimum.

Wise use of Distributed Smalltalk’s presentation/semantic split can help you optimize both processor and network resources. For suggestions on using the presentation/semantic split, see Chapter 6, “Designing and Implementing.”

#### Remote System Autonomy

In a distributed computing environment, the local system cannot control remote systems or the networks that connect them. This issue is critical in a distributed object environment, where collaborating objects and clients requesting object services can “live” in more than one image, and on both local and remote systems.

Distributed Smalltalk provides graceful ways to handle situations when a network or remote system becomes unexpectedly unavailable, including:

- Link policies allow various levels of existence guarantees between objects that make up an application.
- Lifecycle policies allow correct creation, destruction, copying, and moving of applications that contain objects existing in different images.
Event notification policies allow delayed delivery of messages in case of temporary network unavailability.

Shared Objects
A strength of distributed systems (and a challenge to software developers and administrators) is that more than one user may have access to any given object. Shared information is one of the most important advantages of distributed computing.

Distributed Smalltalk’s fundamental architecture supports multiple representations of shared objects, using the presentation/semantic split. This way, different users can see different presentations, while the underlying object and its information remain consistent for all users.

Interoperability Through Standards Compliance
Communication between objects on different systems, and written in languages other than Smalltalk, is only possible if they support standard interfaces and use the same network protocol.

Distributed Smalltalk implements OMG’s CORBA core and CORBA services, Internet Inter-ORB Protocol (IIOP) and other standards including NCS RPC.

IIOP is the CORBA 2.0 protocol that specifies objects and applications to interoperate over a network with other OMG CORBA applications.

In addition, Distributed Smalltalk’s Interface Definition Language (IDL) implementation provides a language-neutral interface between objects created in both Smalltalk and other programming languages.

Creating Applications with Distributed Smalltalk
As with any object-oriented software project, the process of designing and implementing a Distributed Smalltalk application is iterative. You are likely to revise and refine the initial design as the application needs become clearer or change.

In general, the development processes for the I3 interface is simpler than for the IDL-interface, since you do not have to create and register the IDL.
During each iteration, you follow these basic steps:

1. Design. Refer to Chapter 6, “Designing and Implementing.”

2. Implement objects. Use standard object-oriented design and analysis tools to create the basic design. Then, use the structures that Distributed Smalltalk provides to refine the design for making effective distributed applications. Be sure the application runs successfully in a local environment before distributing it.

3. Write IDL. Refer to Chapter 8, “Defining IDL Interfaces.” Write IDL that will reside in the Interface Repository to support communication between distributed objects.

4. Register interfaces. Refer to Chapter 10, “Working with Object Interfaces.” When you create new distributable classes, you define identifier methods that CORBA objects need to be registered with the factory finder and the Interface Repository. Classes that can be instantiated in response to remote client requests must be registered with the local Factory Finder. The interfaces for all new and changed objects that can respond to remote requests must be registered in the Interface Repository.

5. Test, tune, and distribute. Refer to Chapter 17, “Debugging and Tuning.” Use Distributed Smalltalk’s simulated remote testing tools to verify the interfaces specified in the Interface Repository, track messages, and tune performance. You can also use Distributed Smalltalk’s remote debugger to identify and fix problems on remote systems.

Once the application is tested and tuned, you can distribute it to other systems that are running Distributed Smalltalk. Be sure to update the Interface Repository for all images.

6. Create a runtime package (optional). Refer to Chapter 18, “Creating a Deployment Image.” If you wish to distribute the application as a runtime system, you can remove unneeded classes. This helps you create applications that are protected and require minimal system resources.

There are, of course, variations on this general procedure. In some development models, the first step might be defining the IDL interfaces and then doing the Smalltalk implementation.
The design of distributed systems is a complex topic, on which many books have been written. This chapter highlights only some of the more common issues that you should consider when designing Distributed Smalltalk applications. See the Bibliography for some suggestions for further reading.

General Design

Sharing Objects

Distributed objects, by their nature, are shared. Because distributed objects can be accessed by multiple images running on different processors, they are effectively executing in a multi-threaded environment.

The effect of this is that you cannot send two messages in succession to a remote object and assume that the object's state when receiving the second message is identical to the object's state immediately after processing the first message; another image may have sent intervening messages to the object that changed its state. Use the Transaction and Concurrency services to preserve object consistency.

Creating and Destroying Objects

Use the Lifecycle service, not the new: message, to create remote objects.

If you create remote objects by sending instance-creation messages to their classes, the objects will get garbage collected, because no objects on the remote machines will hold references to them. Create and destroy remote objects with the Lifecycle Service.
Referencing Remote Objects

Remote object references can become invalid without warning. If a machine goes down or there are network problems, a valid object reference can become invalid between one message and the next. It is the designer's responsibility to trap unprocessed remote messages and recover gracefully.

Object references are short-lived; they cannot outlive the process that contains their ORB. You cannot assume that an object reference is valid across image invocations. While the client image was not executing, the remote object may have been deleted, moved, or modified. Applications must retrieve their remote object references anew whenever they begin executing.

Performance Considerations

Performance issues are very important in a distributed application, since too much delay in a transaction can effectively cripple an application. The following suggestions may help improve your application's performance.

- Local information should be local; shared information should be distributed.

  This is a platitude, but it's an important design consideration nonetheless. Many of the objects in a Distributed Smalltalk application will not have CORBA interfaces. There are performance costs to making objects distributed, and designers should not incur those costs unless they are necessary.

- Remote messages are costly.

  Remote message sends are approximately a hundred times slower than local message sends. The latency for remote messages is measured in milliseconds; the latency for local messages is measured in microseconds. Since the overhead for a remote message is so high, you should design to make each remote message give as much value as possible.

- The expense of remote messages increases with the size of the objects being marshalled.

  It is slower to transmit a 1000-element Dictionary than to transmit an Integer. Where possible, design so that large objects are transmitted rarely.
There is a conflict between the last two suggestions. You must trade the two off against one another depending on your application's goals and constraints.

**Avoiding Traps**

There are some common traps in designing a distributed application. To avoid these, follow these suggestions:

- Keep class definitions synchronized in local and remote images.
  
  Any object that is passed by value (in attribute parameter) must have identical instance variables, declared in the same order, in both images. In general, be sure to propagate instance variable changes to all images.

- Avoid passing blocks in remote messages. Blocks are not usually meaningful outside their local context; in particular, they are not meaningful to non-Distributed Smalltalk applications.
Implicit Invocation Interface (I3)

The Implicit Invocation Interface (I3) provides a Smalltalk-to-Smalltalk mechanism for developing Distributed Smalltalk applications. Instead of explicitly specifying their distributed classes’ interfaces in IDL, developers can turn on the I3 message transmission mechanism and allow I3 to handle object marshalling and unmarshalling. I3 is useful for rapid prototype development of a distributed application, as well as for developing a purely Smalltalk distributed application.

This chapter discusses how to develop applications that take advantage of I3.

How I3 Works

I3 is a mechanism for message transmission. When I3 is turned on, remote messages that have IDL operation definitions are sent by the normal mechanism; remote messages that have no IDL operation definitions are sent by the I3 mechanism. When I3 is turned off, remote messages that have no IDL operation definition fail.

**Note:** Remote messages sent by the I3 mechanism are approximately 20% slower than remote messages sent by the IIOP mechanism.

Whether a particular message is transported by I3 or through an IDL interface is determined per-message, not per-class. Two ORBS that both have I3 message transmission turned on can send and receive messages that do not have IDL operation definitions, no matter what class sends or receives the message. For convenience, this chapter
refers to classes that have no IDL interface definition of their own as *interfaceless classes*, and to messages that have no corresponding IDL operation definition as *interfaceless messages*.

One of the functions of the operation definitions in an IDL interface is to give the ORB information on how to marshall and unmarshall messages. Since not all methods in an image with I3 enabled have operation definitions, it is up to the class developer to provide marshalling hints. These hints can be provided on a per-class or a per-instance basis, as appropriate.

### Required Methods

All classes whose instances may be passed or returned in remote message sends without IDL operation definitions must understand the instance method `isPassedByValue`.

This method determines how instances of the class are treated when they are passed as arguments to a remote message. If an object’s `isPassedByValue` method returns true, then instances of the class are passed by value: the receiver gets local copies of the instances. (If the receiver modifies the copies, the changes are not propagated to the sender.) If an object’s `isPassedByValue` method returns false, then instances of the class are passed by reference: the receiver gets remote object references to the instance, and the receiver can both read and modify the objects referred to by those references.

By default, objects are passed by reference. However, some base classes are passed by value. The following table shows which base Smalltalk classes are passed by value.
Instance Methods

Smalltalk Classes I3 Passes by Value

<table>
<thead>
<tr>
<th>BOSSCompiledCodeHolder</th>
<th>BOSSContents</th>
<th>BOSSRegisteredObject</th>
<th>Class (only class name is passed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection (except SystemDictionary)</td>
<td>CompiledBlock</td>
<td>DSTI3ObjRef</td>
<td>Exception</td>
</tr>
<tr>
<td>Filename</td>
<td>Geometric</td>
<td>Magnitude</td>
<td>Menu</td>
</tr>
<tr>
<td>Message</td>
<td>Metaclass</td>
<td>ReadStream</td>
<td>RemoteString</td>
</tr>
<tr>
<td>Signal</td>
<td>UndefinedObject</td>
<td>UninterpretedBytes</td>
<td></td>
</tr>
</tbody>
</table>

If you are creating an isPassedByValue method for a class, you need to decide whether instances of the class are allowed to override the class’s pass-by-value setting.

If instances are allowed to override the class’s setting, the isPassedByValue method should look like the following:

```smalltalk
isPassedByValue
  ^self isPassedByValueDefault: aBoolean
```

where you replace `aBoolean` by either `true` or `false`.

**Note:** No class should override the isPassedByValueDefault: method.

If instances are not allowed to override the class’s setting, the isPassedByValue method should look like:

```smalltalk
isPassedByValue
  ^aBoolean
```

where you replace `aBoolean` by either `true` or `false`.

Instance Methods

To change the way in which an instance is passed, the developer sends the passByValue or passByReference messages, as appropriate. If the class’s isPassedByValue method does not invoke isPassedByValueDefault:, the passByValue and passByReference messages are ignored.
passInstVars

Developers can specify how each of an object's instance variable should be passed by overriding passInstVars. passInstVars returns an array containing one entry for each of the object's instance variables; the array entries specify how the instance variables are passed. Each entry in the array must be #true, #false, #ref, or #value.

Meaning of Entries in passInstVars Argument Array

<table>
<thead>
<tr>
<th>Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>#true</td>
<td>Pass this instance variable as specified by its class's isPassedByValue method.</td>
</tr>
<tr>
<td>#false</td>
<td>Pass nil instead of this instance variable.</td>
</tr>
<tr>
<td>#ref</td>
<td>Pass this instance variable by reference.</td>
</tr>
<tr>
<td>#value</td>
<td>Pass this instance variable by value.</td>
</tr>
</tbody>
</table>

If a class overrides the default passInstVars method, the passInstVars settings takes precedence over the passing strategy for any individual instance, which in turn takes precedence over the passing strategy for the class.

Return Values

Marshalling of return values is controlled by the class definitions in the image that is returning the value. Objects are returned by value if their classes are passed by value, or if the returned instance has received the passByValue message. All other objects are returned by reference. The sender can determine whether a returned object is local (passed by value) or remote (passed by reference) by sending the object the isLocal or isRemote messages.

Passing Classes

By default, all classes are passed by value; the value passed is the class's name. The receiving ORB looks for a class with a matching name and uses it.
Caution: It is the developer's responsibility to keep classes on all ORBs identical. DST will not detect incompatibilities between classes on remote and local ORBs. If classes with identical names have different instance variables or instance variables in different orders, the application will not behave as expected.

I3 Instances and Garbage Collection

In general, remote objects are garbage collected unless some object in the same image holds a reference to them. There are two methods of explicitly preventing remote objects from being garbage collected: through the lifecycle service, and implicitly through I3.

- Since the Lifecycle service depends on the corbaName and abstractClassID methods, you cannot conveniently use it with interfaceless objects.
- When I3 is active, all objects passed by reference (or returned by reference) through I3 messages are protected from garbage collection for the life of the ORB on the image that passed or returned the objects. If you stop an ORB, any objects that had no local references will be subject to garbage collection, and object references to those objects in remote images can become invalid.

I3 Traps

- Because I3 is a Distributed Smalltalk extension to CORBA, messages that have no IDL operation definition cannot be sent from or received by non-DST ORBs.
- If an application attempts to modify an object that was passed or returned by value, the local copy of that object is modified; the change does not propagate to the remote object. Developers be consistent when handling objects passed by value.
- Developers should be careful not to destroy remote instances until they have verified that no other image has object references to those instances.
8

Defining IDL Interfaces

Overview

The language-neutral Interface Definition Language (IDL) is a declarative language used to describe interfaces that client objects call and object implementations provide. All ORBs speak IDL as their common language, and you use IDL to define the interfaces for an application’s remotely-accessible objects. In these interfaces, you define the externally visible functionality of each object (but you implement this functionality elsewhere in VisualWorks Smalltalk). A critical part of an ORB’s activities is the translation service (via a language binding) between the local language (such as Smalltalk) and IDL.

The interface definition specifies the operations the object is prepared to perform, the input and output parameters required, and any exceptions that might be generated. The interface constitutes a contract with clients of the object, who use the same interface definition to build and dispatch invocations as the object implementation uses to receive and respond to these requests.

Interfaces form the backbone of the IDL framework. A “good” set of interfaces embodies a coherent structure that clearly defines how a service, or set of services, can be used.

This chapter describes the basic IDL syntax. It provides an introduction to coding in IDL, and describes how to declare:

- Modules
- Interfaces
- Constants
- Data types
Comparing Smalltalk & IDL

Smalltalk is a general-purpose programming language; IDL is an interface definition language, which specifies interfaces but does not (and cannot) specify implementation.

Smalltalk is a dynamically typed language, one in which the type of an object is defined by the set of messages it can process. IDL is a statically typed language, which specifies the type of each argument to a message at IDL compilation time. As a result, IDL definitions can seem unnecessarily restrictive to Smalltalk programmers.

Basic IDL Syntax

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

An IDL specification consists of one or more interfaces declared in one or more IDL files. Interfaces are usually organized into modules, which represent services such as “mail services” or “display services.”

By convention, a source file containing interface specifications written in IDL has an .idl extension. For example, the file orb.idl contains IDL-type definitions.

See “IDL Lexical Conventions” on page 245 for a description of IDL lexical conventions.
IDL Specification

An IDL specification consists of one or more type definitions, constant definitions, exception definitions, or module definitions. The syntax is:

```plaintext
<specification> ::= <definition>+
<definition> ::= <type_dcl>";" |
              <const_dcl>";" |
              <except_dcl>";" |
              <interface> ";" |
              <module> ";"
```

Refer to the following sections for further expansion of these definitions.

Declaring Modules

The module construct is used to scope IDL identifiers (see “Names and Scopes” on page 105). You should limit each IDL file to either a single module or a set of small modules that are related to each other.

A module declaration takes the form:

```plaintext
module identifier {
    module_definition;
    […]
}
```

Example

```plaintext
module CentralOffice {
    interface Depot {
        #pragma selector find_item_info
        findItemInfo:barCode:quantity:itemInfo:
        void find_item_info ( 
            in AStore::AStoreId store_id,
            in POS::Barcode item,
            in long quantity,
            out AStore::ItemInfo item_info)
        raises (AStore::BarcodeNotFound);
    }
}
```
Declaring Interfaces

Note: A module frequently includes two or more related interfaces. For example, when a single logical object is split between presentation and semantic objects, the module would include interfaces for both.

The identifier is a scoped name of a module.

The module_definition can include:

- Type declarations (see “Declaring Data Types” on page 80)
- Constant declarations (see “Declaring Constants” on page 76)
- Exception declarations (see “Declaring Exceptions” on page 95)
- Interface declarations (see “Declaring Interfaces” on page 73)
- Another module

Getting Information About a Module

To search a module registered in the ORB's Interface Repository for the module contents, use either of the following methods to access it:

```plaintext
ORBObject class >> lookupName:levels:limit:excludeInherited:
ORBObject class >> lookup:
```

You can also use the IR Browser.

Declaring Interfaces

An interface declaration, which provides the information needed to develop clients that use the interface’s operations, takes the form:

```plaintext
interface identifier
{
    [interface_definition;
    [...]]
};
```

The identifier, which defines a new legal type name representing a reference to an object, can be used anywhere a type name is legal. For example, it can be used as a parameter or return value of an operation, or as a member of a struct or union. Since an identifier represents a reference to an object, the meaning of a parameter or member that uses that name is the same as a reference to the object supporting the interface. Empty interfaces are legal in IDL.
Example

interface Store {
    readonly attribute AStoreId store_id;
    readonly attribute float store_total;
    readonly attribute float store_tax_total;
    #pragma selector login login:
    StoreAccess login (in POS::POSId id);
    #pragma selector get_POS_totals getPOSTotals:
    void get_POS_totals (out POSList POS_list);
    #pragma selector update_store_totals updateStoreTotals:
    void update_store_totals (in POS::POSId id, in float price,
                               in float taxes);
};

The interface definition can include:

- Type declarations (see “Declaring Data Types” on page 80)
- Constant declarations (see “Declaring Constants” on page 76)
- Exception declarations (see “Declaring Exceptions” on page 95)
- Operation declarations (see “Declaring Operations” on page 91)
- Attribute declarations (see “Declaring Attributes” on page 99)

Inheritance

An interface can be derived from one or more previously defined (base) interfaces (see “Inheritance” on page 100). Both single and multiple inheritance are allowed. An inherited interface declaration can be identical to its parent, or it can extend or override inherited definitions. (Overriding is allowed for all types, constants, and exceptions but not for operations and attributes.)

Note: Name overloading is not allowed in IDL. Thus, multiple inheritance should never lead to a situation where an interface inherits from two interfaces (or interface components) of the same name but differing definitions.

Inheritance Syntax

An interface is separated from its parent interfaces with a single colon. Commas separate multiple parent interfaces. When an interface inherits from an interface that is declared in another module, its declaration should include both module and interface, separated by two colons.

Thus, for example:
• The AudioMedia interface does not inherit from other interfaces:
  interface AudioMedia {};
• The UserPres interface inherits from the ContainerPres and Message interfaces, all of which are defined in the same module.
  interface UserPres : ContainerPres, Message {};
• The TypedPushConsumer interface inherits from the PushConsumer interface, which is defined in the CosEventComm module.
  interface TypedPushConsumer :
  CosEventComm::PushConsumer
  Object
  get_typed_consumer();

Forward Declaration

In addition, an interface declaration can declare the name of another interface without defining it. This is called a “forward declaration” of an interface. A forward declaration makes it possible for the definition of interfaces to refer to each other.

The syntax for specifying a forward declaration is:
  interface identifier;

Forward declarations are useful when two interfaces make reference to each other, as in the following example:
  interface WindowObject;
  interface Notifier
  {
    void addWindow(in WindowObject win);
  };

  interface WindowObject
  {
    void addNotifier(in Notifier notifier);
  };

In this example, Notifier contains an operation, addWindow(), that takes a WindowObject as its argument, but the IDL compiler won’t know what a WindowObject is unless it has been declared. In addition, WindowObject itself contains an operation, addNotifier(), that refers to Notifier. Regardless of whether WindowObject or Notifier is declared first, the IDL compiler needs to know about the existence of the interface that has not yet been defined.
Note: You can specify multiple forward declarations of the same name, but you cannot derive an interface from a forward-declared interface (see “Inheritance” on page 100). Furthermore, to reduce the possibility of creating circular dependencies on modules, the IDL syntax does not support forward declaration of an interface from another module.

Pass by Reference

It is very important to realize that interfaces are passed by reference; otherwise it is passed by value. It is by carefully using base types, constructed types, and typedefs, or structures of these, rather than interface names, that you determine what will and will not be passed as a copy or passed as an object reference.

Declaring Constants

Constants are identifiers that represent values of a given type. They can be declared anywhere in an IDL file using const. A constant declaration takes the form:

```idl
const constant_type identifier = constant_expression;
```
### Declaring Constants

The identifier is a name representing a constant.

The constant_expression is a sequence of operators and operands that specifies a computation.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>Integer from $-2^{31}$ to $2^{31} - 1$.</td>
</tr>
<tr>
<td>short</td>
<td>Integer from $-2^{15}$ to $2^{15} - 1$.</td>
</tr>
<tr>
<td>unsigned long</td>
<td>Integer from 0 to $2^{32} - 1$.</td>
</tr>
<tr>
<td>unsigned short</td>
<td>Integer from 0 to $2^{16} - 1$.</td>
</tr>
<tr>
<td>char</td>
<td>8-bit character. See “IDL Grammar” on page 255 for a complete list of the space, alphabetic, digit, and graphic characters, as well as the meaning and value of the null and formatting characters. The meaning of all other characters is implementation-specific.</td>
</tr>
<tr>
<td>boolean</td>
<td>Value of TRUE or FALSE.</td>
</tr>
<tr>
<td>float</td>
<td>Floating point constants are coerced to double-precision floating point numbers.</td>
</tr>
<tr>
<td>string</td>
<td>Bounded or unbounded sequence of 8-bit quantities, except null.</td>
</tr>
<tr>
<td>scoped_name</td>
<td>Previously defined name of an integer type, character type, boolean type, floating point type, or string type.</td>
</tr>
</tbody>
</table>

### Valid constant_types
### Operators that can be used in a constant_expression

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unary Operators</strong>&lt;br&gt;(+, -, ~)</td>
<td>Unary + and - operators are valid for either floating point or integer expressions.&lt;br&gt;Unary ~ generates the bit-complement of the expression to which it is applied. For the purposes of such expressions, the values are 2’s complement numbers. Thus, the complement of a long integer is generated as -(value+1), and the complement of an unsigned long integer is generated as ((2^{32} - 1))-value. It is valid for integer expressions.</td>
</tr>
<tr>
<td>*</td>
<td>Binary “multiplication” operator generates the product of the operands. It is valid for either floating point or integer expressions.</td>
</tr>
<tr>
<td>/</td>
<td>Binary “division” operator generates the quotient of the operands. It is valid for either floating point or integer expressions.</td>
</tr>
<tr>
<td>%</td>
<td>Binary “remainder” operator generates the remainder from the division of the first expression by the second. If the second operand is 0, the result is undefined; otherwise ((a/b)*b+a%b) is equal to (a). If both operands are non-negative, the remainder is non-negative; otherwise the sign of the remainder is implementation-dependent. It is valid for integer expressions.</td>
</tr>
<tr>
<td>+</td>
<td>Binary “addition” operator generates the sum of the operands. It is valid for either floating point or integer expressions.</td>
</tr>
<tr>
<td>-</td>
<td>Binary “subtraction” operator generates the difference between the operands. It is valid for either floating point or integer expressions.</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Binary “left shift” operator shifts the value of the left operand left the number of bits specified in the right operand, with 0 fill for the vacated bits; the right operand must be in the range 0 - 31, inclusive. It is valid for integer expressions.</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Binary “right shift” operator shifts the value of the left operand right the number of bits specified in the right operand, with 0 fill for the vacated bits; the right operand must be in the range 0 - 31, inclusive. It is valid for integer expressions.</td>
</tr>
<tr>
<td>&amp;</td>
<td>Binary “and” operator generates the logical, bitwise AND of the left and right operands. It is valid for integer expressions.</td>
</tr>
</tbody>
</table>
Declaring Constants

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; if it contains the name of an integer constant with a negative value, the constant_expression is evaluated as signed long. The computed value is coerced back to the specified constant_expression in constant initializers. An error condition occurs if the computed value exceeds the precision of the specified constant_expression, or if any intermediate value exceeds the range of the evaluated-as type (long or unsigned long).

Floating point literals are double, all floating point constants are coerced to double, and floating point expressions are computed as doubles; the computed double value is coerced back to the specified constant_expression in constant initializers. An error condition occurs if the coercion back to the specified constant_expression fails, or if any intermediate values (when evaluating the expression) exceed the range of double.

A constant_expression can contain a unary expression, which takes the form:

unary_operator primary_expression;

The table above describes the unary operators that can be used. A primary_expression is defined as a:

- Scoped name (see “Names and Scopes” on page 105)
- Literal (integer, string, character, floating point, or boolean)
- Constant_expression

---

Operators that can be used in a constant_expression (Continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Binary “exclusive or” operator generates the logical, bitwise EXCLUSIVE-OR of the left and right operands. It is valid for integer expressions.</td>
</tr>
<tr>
<td></td>
<td>Binary “or” operator generates the logical, bitwise OR of the left and right operands. It is valid for integer expressions.</td>
</tr>
</tbody>
</table>

Note: It is not legal to mix type expressions. For example, you cannot mix integer types with floating point types in a constant declaration.
If the primary_expression contains a constant_expression, the constant_expression must be contained in a "( )" combination.

**Note:** The operands in a constant_expression are literals or scoped names that have been previously defined in constant declarations.

In the example:

```c
const short MODIFY_PERMISSION = 0x01;
```

the constant declares that whenever the MODIFY_PERMISSION identifier is specified, the short integer 0x01 is to be used.

One constant can be the base for a family of other constants. This allows you to change the values of the entire family at once. In the example:

```c
const long SIZE = 8;
const long wordsize = 4 * SIZE;
const long segments = 1024 / SIZE;
```

SIZE represents the size of the basic storage unit of a computer (byte), and wordsize and segments change when the declaration of the constant SIZE changes.

### Declaring Data Types

IDL provides constructs for naming data types. Specifically, IDL uses the `typedef` keyword, as well as constructed type declarations to IDL defines a set of type specifiers to represent typed values. The type specifiers are:

- Simple type specifiers
- Constructed type specifiers

#### Using Declarators to Give a Name to a Type

The declarator can be an:

- Identifier
- Array

An identifier is a type name.
Declaring Data Types

An array can specify a fixed-size array (of one or more dimensions) whose size is fixed at compile time. An array declarator contains an identifier (as described above) and the size of each dimension; the syntax is:

```
identifier[constant_expression][[constant_expression]]
```

**Note:** Each `constant_expression` must result in a positive integer constant.

Use a “[]” combination to specify the size of each dimension. For example:

```
table[6][7]
```

defines table as a two-dimensional array whose dimensions are 6 elements by 7 elements.

When an array is passed as a parameter in an operation invocation, all elements of the array are transmitted. However, because the implementation of array indexes is language mapping-specific, passing an array index as a parameter may yield incorrect results.

**typedef**

The typedef keyword can be used anywhere in an IDL file to declare new data type names. The syntax for a using typedef to associate a name with a data type is:

```
typedef type_specifier declarator[,declarator]…;
```

**Example**

```
typedef long AStoreId;
```

The type_specifier can be a:

- Base data type (see “Base Data Types” on page 82)
- Template type (see “Template Types” on page 82)
- Constructed type (see “Constructed Types” on page 85)
- Scoped name (see “Names and Scopes” on page 105), which must be the name of a type

**Note:** Scoped names must be defined before they can appear in a typedef.
Simple Types

Simple type specifiers can be:

- Base data types
- Template types (see “Template Types” on page 82)
- Scoped names (see “Names and Scopes” on page 105)

Base Data Types

Base data Types Supported by IDL

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>IEEE single-precision floating point number.</td>
</tr>
<tr>
<td>double</td>
<td>IEEE double-precision floating point number.</td>
</tr>
<tr>
<td>long</td>
<td>Any integer from $-2^{31}$ to $2^{31} - 1$.</td>
</tr>
<tr>
<td>short</td>
<td>Any integer from $-2^{15}$ to $2^{15} - 1$.</td>
</tr>
<tr>
<td>unsigned long</td>
<td>Any integer from 0 to $2^{32} - 1$.</td>
</tr>
<tr>
<td>unsigned short</td>
<td>Any integer from 0 to $2^{16} - 1$.</td>
</tr>
<tr>
<td>boolean</td>
<td>Value of TRUE or FALSE.</td>
</tr>
<tr>
<td>octet</td>
<td>8-bit quantity (no conversion).</td>
</tr>
<tr>
<td>char</td>
<td>8-bit character. See “IDL Grammar” on page 255 for a complete list of the space, alphabetic, digit, and graphic characters, as well as the meaning and value of the null and formatting characters. The meaning of all other characters is implementation-specific.</td>
</tr>
<tr>
<td>any</td>
<td>The any type permits the specification of values that can express any IDL type.</td>
</tr>
</tbody>
</table>

Each data type is mapped to a base data type via the appropriate language mapping (see “Mapping of IDL to Smalltalk” on page 112 for information about the Smalltalk language mappings). For example, to declare a long integer x, use the following IDL statement:

```
long x;
```

Template Types

IDL supports the following template types:

- String
- Sequence
Strings

A string can be any 8-bit quantity except null. It is similar to a sequence of chars. Prior to passing a string as a function of an argument (or an member in a struct or union), the length of the string must be set in a language-mapping dependent manner.

A string identifier takes the form:

```
string [<constant_expression>]
```

The constant_expression, if specified, is a sequence of operators and operands that specifies a computation. The table above lists the operators that can be used in a constant_expression.

**Note:** The constant_expression must result in a positive integer constant.

A string can be unbounded (that is, have no specified maximum size) or can be bounded (that is, include the max_size, enclosed in a "< >" combination, as the first parameter). For example:

```
string address;
string <16> name;
```

The first string, address, is unbounded (that is, no upper limit of characters is specified). The second string, name, has a maximum length of 16 characters.

**Note:** Strings are defined as a separate data type because many languages have special built-in functions or standard library functions for string manipulation. This allows substantial optimization in the handling of strings compared to what can be done with sequences of general types.

Sequence

A sequence is a one-dimensional array with a maximum size (which is fixed at compile time), and a length (which is determined at run time). It can be unbounded (that is, have no specified maximum size) or bounded (that is, include the max_size parameter).

A sequence type identifier takes the form:

```
sequence < simple_type_specifier[,constant_expression] >
```
Example

typedef sequence<POSInfo> POSList;

The simple_type_specifier can be a:

- Base data type (see “Base Data Types” on page 82)
- Template type
- Scoped name (see “Names and Scopes” on page 105), which must
  the name of a type

The optional constant_expression is a sequence of operators and
operands that specifies a computation. The table above lists the
operators that can be used in a constant_expression.

**Note:** The constant_expression must result in a positive integer
constant.

You must enclose the type name in a "< >" combination to specify the
type of data that belongs in the sequence and, optionally, the upper limit
of the number of elements. For example:

```c
sequence < long > list;
sequence < float,20 > datapoints;
```

In this example, list is an unbounded sequence of long data types, and
datapoints is a sequence of up to 20 floating point data types.

Sequences of strings are useful for describing multi-line text fields. For
example:

```c
sequence < string<80> > body;
```
defines body, which is made up of an indefinite (unbounded) number of
lines, each of which cannot exceed 80 characters.

Before an unbounded sequence can be passed as a function argument
(or as a field in a struct or union), the length and maximum size of the
sequence, as well as the address of a buffer to hold the sequence, must
be set in a language-mapping dependent manner. After receiving such a
sequence result from an operation invocation, the length of the returned
sequence, which can be obtained in a language-mapping dependent
manner, will have been set.

Before a bounded sequence can be passed as a function argument (or
as a field in a struct or union), the length of the sequence must be set in a
language-mapping dependent manner. After receiving such a sequence
result from an operation invocation, the length of the returned sequence, which can be obtained in a language-mapping dependent manner, will have been set.

**Constructed Types**

IDL supports the following constructed types:

- Structures
- Enumerations
- Discriminated unions

**Structures**

Declaring a structure with `struct` defines a new legal data type. The syntax of a structure data type is:

```plaintext
struct struct_identifier
{
    member +; //one or more members
};
```

The value of a struct is the value of all of its members.

For example:

```plaintext
struct POSInfo {
    POS::POSId id;
    Object store_access_reference;
    float total_sales;
    float total_taxes;
};
```

A `struct_identifier` is the type name of the structure.

**Note:** Structure types can also be named using a `typedef` declaration (see “typedef” on page 81).

The syntax for a member is:

```plaintext
type_specifier declarator[[,declarator]…]
```

**Note:** Each member in the structure must have a unique name.
The type specifier can be a:

- Base data type (see “Base Data Types” on page 82)
- Template type (see “Template Types” on page 82)
- Constructed type
- Scoped name (see “Names and Scopes” on page 105)

In the example:

```pascal
struct mystructure
{
    long x,y;
    double z;
};
```

mystructure is declared to be a structure data type containing three members: x, y, and z.

By default mapping, a struct is mapped to a Smalltalk Dictionary, which is answered by any operation with a structure as a return value type. When a structure is specified as a parameter type, then under the default mapping, an appropriate Dictionary must be explicitly constructed on the Smalltalk side to serve as the parameter. For example, here is an invocation of a `has_middle_name` operation that takes a single `personalName` structure as an input parameter:

```pascal
relevantObject hasMiddleName:
((Dictionary new)
    at: #firstNameput: 'Max';
    at: #middleNameput: 'Karl';
    at: #surnameput: 'Scheler';
    yourself).
```

A structure may be mapped to the instance of a named class by using the class pragma.

**Enumerations**

An enumeration (enum) is an ordered set of identifiers, called enumerators, that specify all of the legal values that a variable may have. As such, an enumeration defines a new data type that takes one of that set of specified values. The syntax of an enumeration data type is:

```pascal
enum enum_identifier
{
    enumerator[,[enumerator]…]
}
```

An `enum_identifier` is the type name of the enumerator.
For example:

```lisp
enum colors_allowed
{
    red, blue, white, green
};
```

declares that variables of the type `colors_allowed` can be assigned the values red, blue, white, or green.

**Note:** Enumeration types can also be named using a `typedef` declaration (see “typedef” on page 81).

Each enumerator in the enumeration must specify a unique name. The maximum number of enumerators is $2^{32}$; therefore, the enumerated names must be mapped to a native data type capable of representing a maximally-sized enumeration.

An identifier in an enumerator that is declared as a constant.

The order in which the enumerators are listed defines their relative order. Enumerators can be compared, and a successor/predecessor function defined on the type will reflect the ordering.

From within Smalltalk code, both enumerations and enumerators are accessed via the `CORBAConstants` dictionary, an entity specified in the Smalltalk Binding. For example, given the declaration

```lisp
module Music {
    enum WIND_INSTRUMENTS { oboe, recorder, clarinet, flute };
    enum BRASS { trombone, french_horn, trumpet };
};
```

the windInstruments enumeration is accessed by

```lisp
(CORBAConstants at: #'::music::WIND_INSTRUMENTS').
```

This answers an instance of class `Array` that contains symbols. An enumerator is accessed similarly. For example, the oboe enumerator is accessed by

```lisp
(CORBAConstants at: #'::Music::WIND_INSTRUMENTS::oboé').
```

This does not answer an instance of `Symbol`, but rather an instance of class `DSTEnumerator`. 
Discriminated Unions

IDL unions are a cross between C union and switch statements. IDL unions must be discriminated; that is, the header must specify a typed tag field that determines which union member is valid.

Discriminated unions are the preferred way of returning a value of one of a limited number of data types. If an interface’s operation, for example, is to return one of three kinds of values, use a union rather than the data type any.

The syntax of a discriminated union data type is:

union union_identifier
(switch switch_type_specifier)
{
    case constant_expression : element_specifier;
    default : element_specifier;
    …;
}

For example:

union cell_value
switch(enum cell_content {numeric, string, formula})
{
    case numeric:numeric_valuenumber;
    case string:stringvalue;
    case formula:formula_valuethe_formula;
};
interface cell
{
    attribute cell_value value;
    …
};
declares a spreadsheet cell value that can be either a numeric, a string constant, or a formula. The reasons for defining a spreadsheet_cell with union (rather than any) in this example are that it:

• Denotes the interface most accurately because it specifies all allowed types.

• Makes the resulting code easier to work with because the potential types of data that the caller must deal with are specified.

Discriminated union types can also be named using a typedef declaration (see “typedef” on page 81). The default is optional.

A union_identifier is the type name of the union.
A switch_typeSpecifier specifies the type with which the cases constant_expression must be consistent.

*Valid Switch_Type_Specifiers*

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>Any integer from -2(^{31}) to 2(^{31}) - 1.</td>
</tr>
<tr>
<td>short</td>
<td>Any integer from -2(^{15}) to 2(^{15}) - 1.</td>
</tr>
<tr>
<td>unsigned long</td>
<td>Any integer from 0 to 2(^{32}) - 1.</td>
</tr>
<tr>
<td>unsigned short</td>
<td>Integer from 0 to 2(^{16}) - 1.</td>
</tr>
<tr>
<td>char</td>
<td>8-bit character. See “IDL Grammar” on page 255 for a complete list of the space, alphabetic, digit, and graphic characters, as well as the meaning and value of the null and formatting characters. The meaning of all other characters is implementation-specific.</td>
</tr>
<tr>
<td>boolean</td>
<td>Value of TRUE or FALSE.</td>
</tr>
<tr>
<td>enum</td>
<td>Any enumerator for the discriminator enum type. The identifier for the enumeration is in the scope of the union; it must be distinct from the member declarators.</td>
</tr>
<tr>
<td>scoped_name</td>
<td>Previously defined name of an integer type, character type, boolean type, or enum type.</td>
</tr>
</tbody>
</table>

The constant_expression is a sequence of operators and operands that specifies a computation. The table above lists the operators that can be used in a constant_expression.

The constant_expression must be consistent with the switch_typeSpecifier. The syntax for a elementSpecifier is:

```
   typeSpecifier declarator[,declarator]…
```

Class DSTUnion is the Smalltalk implementation of the CORBA union protocol. DSTUnion has two instance variables, discriminator and value. Instances of DSTUnion are usually created using the asCORBAUnion: method, implemented in class Object. This may be sent to any object and must have an appropriate discriminator value as its argument. Sample IDL for using an explicit mapping is shown below:

```
#pragma class AccountNumber DSTUnion
union AccountNumber switch(boolean) {
   case true: long l;
   case false: string s;
};
```
This code explicitly maps a declared union to the DSTUnion implementation class. The mention of DSTUnion in the class pragma could be replaced by the name of any class that also supported the CORBA union protocol: value, value:, discriminator, discriminator:, a class side instance creation method named discriminator:value:, and the semantics that goes with these messages.

Each element in the discriminated union must specify a unique name.

The type_specifier can be a:

- Base data type (see “Base Data Types” on page 82)
- Template type (see “Template Types” on page 82)
- Constructed type
- Scoped name (see “Names and Scopes” on page 105)

An identifier is a name of an element.

An array can specify a fixed-size array (of one or more dimensions) whose size is fixed at compile time.

case declarations must match or be automatically castable to the defined type of the discriminator.

The following table shows matching rules for consistency check between case constant_expressions and switch_type_specifiers.

Matching rules

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>Integer value in the value range of long.</td>
</tr>
<tr>
<td>short</td>
<td>Integer value in the value range of short.</td>
</tr>
<tr>
<td>unsigned long</td>
<td>Integer value in the value range of unsigned long.</td>
</tr>
<tr>
<td>unsigned short</td>
<td>Integer value in the value range of unsigned short.</td>
</tr>
<tr>
<td>char</td>
<td>char.</td>
</tr>
<tr>
<td>boolean</td>
<td>Value of TRUE or FALSE.</td>
</tr>
<tr>
<td>enum</td>
<td>Enumerator for the discriminator enum type.</td>
</tr>
</tbody>
</table>

It is not necessary to list all possible values of the unions. The value of a union is the value of the discriminator together with one of the following:

- If the discriminator value is listed explicitly in the case statement, the value of the element associated with the case statement
Declaring Operations

- If the default case label is specified, the value of the element associated with the default case label

  **Note:** default can appear no more than once.

- No additional value

Declaring Operations

An interface can have operations. Operation declarations define the set of operations that a client can invoke on an object supporting the interface. You declare operations within the interface definition.

**Note:** A derived interface (see “Inheritance” on page 100) automatically supports any operations in the interface(s) it inherits from, and can add its own operations. However, a derived interface cannot re-declare any of the operations it inherits.

The syntax of an operation declaration is:

```
[oneway] operation_type_specifier identifier
  ([parameter_declaration[,parameter_declaration]…])

  [raises (scoped_name[,scoped_name]…)]

  [context (string_literal[,string_literal]…)];
```

The `oneway` operation attribute is optional. When it is not specified and an exception is raised, the operation is invoked no more than once. If, however, an exception is not raised, the operation is invoked exactly once. When it is specified, the operation is invoked no more than once, which does not guarantee that the call will be delivered successfully.

**Note:** If the optional `oneway` operation attribute is specified, the operation cannot contain any output parameters, and must specify a `void` return type. In addition, a one-way operation cannot include a `raises` expression; however, invocation of the operation may raise a standard exception.
Example

```c
void find_price (  
    in POS::Barcode item,  
    in long quantity,  
    in long store_id,  
    out float item_price,  
    out float item_tax_price,  
    out ItemInfo item_info)  
raises (BarcodeNotFound);
};
```

The operation_typeSpecifier can be a:

- Base data type (see “Base Data Types” on page 82)
- Template type (see “Template Types” on page 82)
- Scoped name (see “Names and Scopes” on page 105)
- void return type

The identifier is the type name of the operation. Identifiers are used at runtime by both the static and dynamic interfaces. As a result, all operations that might apply to a particular object must have unique names. This requirement prohibits redefining an operation name in a derived class, as well as inheriting two operations with the same name.

The syntax for a parameter_declaration is:

```
parameter_attribute simple_typeSpecifier declarator[,[declarator]…]
```

The parameter_attribute must specify:

- In (passed from client to server)
- Out (passed from server to client)
- Inout (passed in both directions)

**Note:** Implementations should not attempt to modify an in parameter; the ability to do so is language-mapping specific, and the effect is undefined. Also, you should avoid using the inout parameter because some languages have difficulty managing the memory associated with processing inout.

The simple_typeSpecifier can be a:

- Base data type (see “Base Data Types” on page 82)
- Template type (see “Template Types” on page 82)
Declaring Operations

- Scoped name (see “Names and Scopes” on page 105)

  **Note:** When an unbounded string or sequence (see “Template Types” on page 82) is passed as an inout parameter, the returned value cannot be longer than the input value.

An identifier is the type name of a parameter of the operation.

The following example defines an interface to a stack object that supports pop() and push() operations for long integer elements and an is_empty() operation:

```smalltalk
interface Stack
{
  long pop();
  void push(in long n);
  boolean is_empty();
};
```

In this example, pop() takes no argument but returns a long integer, and push() makes use of an `in` parameter, indicating that the parameter is passed from client to server.

**Raises Expressions**

You can declare an operation to raise user-defined exceptions (see “User-Defined Exceptions” on page 95) as a result of an invocation of the operation.

  **Note:** Standard (CORBA predefined) exceptions cannot be listed in raises expressions.

The syntax of an optional raises expressions is:

```smalltalk
raises(scoped_name[, scoped_name]…);
```

  **Note:** If the optional oneway operation attribute is specified, the operation cannot include a raises expression; however, invocation of the operation may raise a standard exception.

Each scoped_name specified in an optional raises expression must be a previously defined exception.
Note: An invocation can raise a standard exception even though standard exceptions cannot be listed in raises expressions; that is, the absence of raises expressions does not prevent an operation from raising standard exceptions.

Context Expressions

You can also declare which elements of the client's context can affect the performance of a request by the object. The syntax of an optional context expression is:

```plaintext
context(string_literal[,string_literal]…);
```

Each string Literal is an arbitrarily long sequence of alphabetic, digit, period (.), underscore (_), and asterisk (*) characters.

Note: The first character of a string_literal must be an alphabetic character. Furthermore, an asterisk can be used only as the last character in the string.

The value associated with each string_literal in the client's context is provided to the object implementation when the request is delivered. The object can use the information in this request context during request resolution and performance.

Note: The absence of a context expression indicates that there is no request context associated with requests for this operation.

In the following example, the values associated with create_request_op_id are available for requests resolution and to the object implementation when the request is delivered:

```plaintext
interface Stack
{ …
  long pop()
  context(create_request_op_id);
  …
};
```
Declaring Exceptions

Exceptions are alternate results that an operation can return when it encounters an exceptional condition (usually an error).

**Note:** If an exception is raised, the out and normal return values are not valid; instead, only the values of the raised exception’s members are valid.

In addition to the standard exceptions specified by CORBA (see “Standard Exceptions” on page 96), user-defined exceptions can be declared anywhere in an IDL file.

User-Defined Exceptions

The syntax for declaring user-defined exceptions is:

```idl
exception exception_identifier {
    [member];
    […]
};
```

An exception_identifier is the type name of an exception. When an exception is returned as the outcome of a request, the value of the exception identifier can be accessed to determine which exception was raised.

If an exception is declared with members, you can access the values of those members when an exception is raised. If no members are specified, however, no additional information can be accessed.

**Example**

```idl
// The barcodeNotFound exception indicates that the
// input barcode does not match to any known item.
exception BarcodeNotFound {POS::Barcode item; ;
```

The syntax for a member is:

```idl
type_specifier declarator[, declarator]…
```

**Note:** Each member in the exception structure must specify a unique name.
The type_specifier can be a:

- Base data type (see “Base Data Types” on page 82)
- Template type (see “Template Types” on page 82)
- Constructed type
- Scoped name (see “Names and Scopes” on page 105)

An identifier is the type name of a member.

The following Stack interface example includes two exception conditions: underflow and overflow. The overflow exception is defined such that the client code can examine the value of size_limit, which, in this implementation, is defined to be the maximum stack size.

```idl
interface Stack
{
    exception underflow {};
    exception overflow
    {
        long size_limit;
    };
    readonly attribute long size;

    long pop()
    raises(underflow);
    void push(in long n)
    raises(overflow);
    boolean is_empty();
};
```

**Note:** This example contains operations, raises operations, and attributes (see “Declaring Operations” on page 91 and “Declaring Attributes” on page 99).

**Standard Exceptions**

Standard exceptions can be returned as a result of any operation invocation, regardless of the interface specification.

**Note:** Standard exceptions cannot be listed in raises expressions.

The CORBA specification keeps the set of standard exceptions to a manageable size in order to reduce the complexity of handling them. This constraint requires the definition of equivalence classes of exceptions rather than enumerating many similar exceptions. For example, an
operation invocation can fail at many different points due to the inability to allocate dynamic memory. Rather than specify several different exceptions corresponding to the different ways that memory allocation failure causes the exception (during marshalling, unmarshalling, in the client, in the object implementation, allocating network packets, and so on), a single exception corresponding to dynamic memory allocation failure is defined.

Each standard exception also includes a completion_status code which takes one of the values YES, NO, or MAYBE. “Completion,” from the client’s point of view, means exception-free execution of the requested operation by the implementation. The values have the following meanings:

- **YES**
  The state of the object, including objects it acts on, are in the same state as they would be if the operation completed normally. An error was detected after the operation’s termination. Implementations are discouraged from returning this value. Requests that return this value should not be retried.

- **NO**
  The state of the object, including objects it acts on, are in a state that is identical to its state prior to execution of the operation. This may mean that the operation was never started, that the operation was started but did not change the object state, or that the implementation restored the object to the state it had before the operation began. Requests that return this value can be retried.

- **MAYBE**
  The state of the object, including objects it acts on, is not known. Implementations are discouraged from returning this value.

The standard exceptions are defined below:
#define ex_body {unsigned long minor;
    completion_status completed;}
enum completion_status {COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE};
enum exception_type {NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION};
exception UNKNOWN ex_body; // the unknown exception
exception BAD_PARAM ex_body; // an invalid parameter was passed
exception NO_MEMORY ex_body; // dynamic memory allocation failure
exception IMP_LIMIT ex_body; // violated implementation limit
exception COMM_FAILURE ex_body; // communication failure
exception INV_OBJREF ex_body; // invalid object reference
exception NO_PERMISSION ex_body; // no permission for attempted op.
exception INTERNAL ex_body; // ORB internal error
exception MARSHAL ex_body; // error marshalling param/result
exception INITIALIZE ex_body; // ORB initialization failure
exception NO_IMPLEMENT ex_body; // operation implementation unavailable
exception BAD_TYPECODE ex_body; // bad typecode
exception BAD_OPERATION ex_body; // invalid operation
exception NO_RESOURCES ex_body; // insufficient resources for req.
exception NO_RESPONSE ex_body; // response to req. not yet available
exception PERSIST_STORE ex_body; // persistent storage failure
exception BAD_INV_ORDER ex_body; // routine invocations out of order
exception TRANSIENT ex_body; // transient failure - reissue request
exception FREE_MEM ex_body; // cannot free memory
exception INV_IDENT ex_body; // invalid identifier syntax
exception INV_FLAG ex_body; // invalid flag was specified
exception INTF_REPOS ex_body; // error accessing interface repository
exception BAD_CONTEXT ex_body; // error processing context object
exception OBJ_ADAPTER ex_body; // failure detected by object adapter
exception DATA_CONVERSION ex_body; // data conversion error
Declaring Attributes

An interface can have attributes that are within the interface definition.

**Note:** A derived interface (see “Inheritance” on page 100) automatically supports any attributes in the interface(s) it inherits from, and can add its own attributes. However, a derived interface cannot re-declare an attribute as a different type, but it can re-declare it as readonly.

Declaring an attribute 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.

**Note:** The actual accessor function names are language-mapping specific. Only the attribute name is subject to IDL’s name scoping rules. The accessor function names are guaranteed not to collide with any legal operation names that can be specified in IDL.

The syntax of an attribute declaration is:

```
[readonly] attribute simple_typeSpecifier declarator[,declarator]…;
```

The optional readonly keyword indicates that only the implementation can set its value; that is, that there is only a single accessor function—the retrieve value function. You can, for example, declare an operation in IDL and the implementation of that operation would set the value of a readonly attribute.

**Example**

```plaintext
readonly attribute AStoreId store_id;
readonly attribute float store_total;
readonly attribute float store_tax_total;
```

The `simple_typeSpecifier` can be a:

- Base data type (see “Base Data Types” on page 82)
- Template type (see “Template Types” on page 82)
- Scoped name (see “Names and Scopes” on page 105)

An `identifier` is the type name of the attribute.
Declaring an attribute in IDL is similar to defining a pair of public methods to get and set the value of a private data member. You can also define a readonly attribute, which prevents the client from setting the attribute value:

```idl
interface Stack
{
    readonly attribute long size;
    long pop();
    void push(in long n);
    boolean is_empty();
};
```

In this example, Stack includes a readonly attribute, size. Although the client can determine the size of the stack from the attribute, it cannot set it.

Declaring an attribute is different from declaring “get” or “set” operations because attributes—unlike operations (see “Declaring Operations” on page 91)—cannot “raise” user-defined exceptions (see “Declaring Exceptions” on page 95).

Attribute operations return errors by means of standard exceptions (see “Standard Exceptions” on page 96).

**Note:** Although attributes may appear to be a data item or member (the value of which does not change), the implementation can be written to recompute the value on each access. This can be the case even if the readonly attribute is specified.

## Inheritance

An interface can inherit from (that is, be derived from) one or more interfaces. An inheritance specification, which declares that an interface is derived from one or more interfaces, takes the following form:

```idl
interface identifier : scoped_name[,scoped_name[...]]
{
    [interface_definition;
    [...]]
};
```

The identifier (name) of an interface can be used as a type name (once the interface has been declared). For example, it can be used as a parameter or member that uses that name is the same as a reference to
the object supporting the interface. Any object whose type is an interface identifier is expected to support the operations in that interface. For example, if a parameter of type interface Stack, it would be expected to support the Stack operations, pop, push, and is_empty (see page 100 for the stack interface definition).

**Note:** An interface that inherits elements from other interfaces is called a derived interface; the interface from which elements are inherited is called a base interface. Furthermore, an interface is called a direct base interface if it is specified directly in the inheritance specification; it is called an indirect base interface if it is, in turn, a base interface of another direct base interfaces specified in the inheritance specification.

Elements of a base interface can be referred to as if they were elements of the derived interface. A derived interface can redefine inherited types, constants, and exceptions, as well as declare operations and attributes.

Multiple inheritance means that an interface is derived from more than one direct base interface. In these situations, the order of derivation (that is, the order in which the base interfaces are specified) is not significant. However, since multiple inheritance may cause ambiguity, you should use the resolution operator (that is, ::) to explicitly identify the desired element (see “Names and Scopes” on page 105).

An interface definition, which can declare new elements, as well as redefine elements in base interfaces, can include:

- Type declarations (see “Declaring Data Types” on page 80)
- Constant declarations (see “Declaring Constants” on page 76)
- Exception declarations (see “Declaring Exceptions” on page 95)
- Operation declarations (see “Declaring Operations” on page 91)
- Attribute declarations (see “Declaring Attributes” on page 99)

The rules for scoping these names are described in “Names and Scopes” on page 105.

A base interface cannot be specified as a direct base interface of a derived interface more than once; however, it can be an indirect base interface any number of times. For example, the inheritance illustrated in can be declared as follows:
interface A {...};
interface B:A {...};
interface C:A {...};
interface D:B,C {...}

Illustration of Interfaces Inheritance

References to types, constants and exceptions are bound to an interface when it is defined (that is, when it is replaced with the equivalent global scoped name). This guarantees that the syntax and semantics of an interface are not changed when the interface is a base class for a derived class. Consider this example:

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?

In the example above, the early binding of types, constants, and exceptions at interface definition guarantees that the signature of operation f in interface C is:

void f(in float s[3]);

which is identical to that in interface A. This rule also prevents the redefinition of a type, constant, or exception in a 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 will cause an IDL compilation error.

In addition to using the elements defined in the base interface(s), a derived interface can declare new elements (for example, constants) and/or redefine the elements defined in the base interface. A derived interface that redefines any of the inherited type, constant, or exception names must satisfy the scope rules described in “Names and Scopes” on page 105.

**Note:** A derived interface cannot inherit from multiple interfaces (that is, from either direct base or indirect base interfaces) that provide the same operation or attribute name (see “Declaring Operations” on page 91 and “Declaring Attributes” on page 99). An implementation of an interface, however, can redefine the implementation of those operations and attributes.

**Note:** The implementation of the operation can be redefined, not the interface.

A derived interface has an “is a” relationship with its base interfaces (that is, it is the same kind of whatever the base interface describes). For example, a DroppableObject interface is a specific kind of Document interface; therefore, the DroppableObject could inherit from the Document:

```idl
interface DroppableObject : Document
{
    IDL statements describing DroppableObject
};
```

DroppableObject, in this example, inherits every element of the Document interface. That is, if DroppableObject is declared as shown above, but includes no IDL statements of its own, its interface is identical to that of Document.

The body of the DroppableObject interface contains any needed definitions that are not already defined in the Document interface. It also can include redefinitions of type, constant, or exception names already defined in Document.
Multiple inheritance allows you to derive an interface from more than one base interface. If the derived interface inherits from multiple interfaces, the order in which the interfaces are inherited is not significant.

For example, assume there are three interfaces: Queue, Directory, and Mailbox. The Queue interface stores a sequence of arbitrary type; a Directory interface provides access to objects by name; and a Mailbox interface behaves like a Queue and a Directory, as well as a Mailbox. Thus, if a Mailbox inherits from both Queue and Directory, a browsing tool that works on directories can work on mailboxes, and a mail filter can use the Queue interface allowing various filters to be composed in a Mailbox:

```
interface Mailbox : Queue, Directory
{
   IDL statements describing Mailbox
};
```

You can specify the base interfaces (in this example, Queue and Directory) in any order since the order is not significant.

### IDL Preprocessing

IDL preprocessing is based on ANSI C++ preprocessing and provides macro substitution, conditional compilation, and source file inclusion. The preprocessing facilities are used to include definitions from other IDL specifications.

For a comprehensive discussion of C++ preprocessing, refer to the “Preprocessing” chapter in *The Annotated C++ Reference Manual*.

Directives (that is, lines beginning with `#`) communicate with the preprocessor. These lines, which may include white space before the `#`, have a syntax that is independent of IDL. Except for the IDL-specific pragmas, which are semantically constrained, they may appear anywhere, and remain in effect until the end of the translation unit. Furthermore, a preprocessing directive can continue on another line; to continue a line, place a backslash (`\`) immediately before the new line at the end of the line to be continued.

**Note:** A backslash (`\`) character cannot be the last character in a source file.
#include

You can include definitions from other IDL specifications with #include. If the #include appears in a global scope, text from any included file is treated as if it appeared in the including file (that is, the types, constants, modules, and interfaces are declared and available), although no additional code is generated by the IDL compiler as a result of the inclusion. It simply allows you to resolve external IDL references within the IDL file.

Note: #include should be used only in global scopes because including scopes modify the names of all included constructs.

CORBA::Module

In order to prevent names defined within the CORBA specification from clashing with names in programming languages and other software systems, all names defined by CORBA are treated as if they were defined within a module named CORBA. Within an IDL specification, however, IDL keywords such as Object must not be preceded by a CORBA:: prefix. Other interface names are not IDL keywords and so must be referred to by their fully scoped names within an IDL specification.

Names and Scopes

An entire IDL file forms a naming scope. Each time you use one of the following kinds of declarations, you form a nested scope:

- Module
- Interface
- Structure
- Union
- Operation
- Exception

The names of declarations reflect the naming scope in which they reside. The names you use when declaring any of the following are scoped:

- Types
- Constants
Enumeration values
Exceptions
Interfaces
Attributes
Operations

The syntax of a scoped_name is:

\[ \text{[scoped_name][::]identifier} \]

The optional scoped_name is the name of a scope. The identifier is an element or member declared in the scope. It can be referenced explicitly by using a scope resolution operator (that is, ::), as illustrated in the following examples of valid references to scoped names:

A::except1
::except1
except1

Note: You cannot define an identifier more than once in a scope, although you can redefine identifiers in nested scopes.

Since the IDL compiler is not case sensitive, respelling an identifier (in terms of case) is considered to be reuse of the name in its scope. For example, Test in the following sample, is a re-declaration of test and, therefore, is not allowed:

```idl
interface check_segment
{
    exception test {};
    void Test();
    // ILLEGAL: this is a re-declaration of test!
};
```

Type names defined in a scope are available for immediate use within that scope. Specifically, 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, the following is a valid form of recursive type specification

```idl
struct foo
{
    long value;
    sequence<foo> chain;
};
```
An unqualified name can be used in a particular scope. However, once it is used in a scope, it cannot be redefined in that scope; such redefinitions cause compilation errors.

**Note:** An unqualified name used in a particular scope is resolved by successively searching farther out in enclosing scopes. Thus, if you use a name defined in an enclosing scope in the current scope, you cannot redefine a version of that name in the current scope.

When a qualified name begins with the scope resolution operator (for example, `::identifier`), the resolution process starts at file scope. A qualified name is resolved by first resolving the qualifier (scoped_name), then locating the definition of the identifier within that scope.

**Note:** The identifier must be directly defined in the scope (or, in the case of an interface, inherited into the scope). The identifier is not searched for in enclosing scopes.

It is sometimes useful for interfaces to refer to exceptions, types, and constants that are declared in other interfaces or modules. You make this reference by using the scope resolution operator (that is, `::`) as illustrated in the following example. In this example, an operation in interface B raises an exception declared in interface A:

```idl
interface A
{
    exception general_error ();
    ...
};

interface B
{
    void op1()
        raises(A::general_error);
};
```

Every IDL definition in a file has a global name within that file. The global name is the concatenation of the current root, the current scope, the scope resolution operator (that is, `::`), and the local name for the definition.

Inheritance produces shadow copies of inherited identifiers, but these are semantically the same as the original definition. Therefore, as shown in the following example, you can refer to the inherited `except1` exception as either `A::except1` or `B::except1:`
interface A
{
    exception except1 {};
};

interface B : A {};

interface C
{
    void op1()
        raises(A::except1);
    void op2()
        raises(B::except1);
        //op1() and op2() raise the same
        //exception.
};

Ambiguity can occur in specifications due to the nested naming scopes. For example, in:

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 because the IDL compiler cannot determine which string_t is desired.

**IDL Traps**

These are some of the more common problems developers encounter when defining IDL interfaces for Smalltalk classes. For a list of common design issues in all distributed systems, see “Hints for Distributed Design” on page 43.
Magnitude Mismatches

The standard IDL numeric quantities do not match the sizes of the standard Smalltalk Magnitude subclasses. An IDL long (-2^{31} to 2^{31}-1) may be represented as a Smalltalk SmallInteger (-2^{29} to 2^{29}-1), LargeNegativeInteger, or LargePositiveInteger, depending on its size and sign. Conversely, some valid LargePositiveIntegers cannot be represented as IDL longs.

The marshalling engines, which translate objects from their Smalltalk representation to transmission format and back again, do not check the magnitudes of numbers when marshalling. If you define an IDL interface to return a short value, then return the SmallInteger 2 raisedTo: 28, the number that arrives at the receiver will not be interpreted as 2^{28}. It is the developer's responsibility to check magnitudes before transmitting them in remote messages.

Mismatched IDL Interfaces and Smalltalk Selectors

When you change a message selector in Smalltalk, you must remember to change the corresponding IDL definitions. If you change the type of an argument, this will not cause Smalltalk errors, but may make the IDL invalid, causing marshalling exception.

Inheritance and Overriding Operations

Unlike Smalltalk, IDL forbids interfaces to override operations defined in their superclass. If the superclass definition specifies an operation, the subclass must use that operation definition.

Passing Values and References: Interfaces and Structures

You use interfaces and structures in your operation definitions to control whether an object is passed by value or by reference. Use the name of an interface as a type in your operation declaration when you want to pass an instance by reference. Use the name of a struct in your operation declaration when you want to pass an instance by value. This works for either the type of the return value or the type of one or more of the parameters.

For example, building on the Person example above, you may have these definitions:
module PersonModule {

#pragma class PersonStruct Person
struct PersonStruct {
  string name;
  short age;
};
interface PersonInterface {

  string name();
  #pragma selector set_name name:
  void set_name (in string s);

  short age();
  #pragma selector set_age age:
  void set_age (in short s);

  PersonStruct personCopyWithName (in string n);
  PersonInterface personReferenceWithName (in string n);
};

In this case, the method personCopyWithName: will answer a local copy of a Person (pass by value), but the method personReferenceWithName: will answer a reference to a remote instance of Person (pass by reference).

SmalltalkTypes

It is important to look at what is in SmalltalkTypes in DSTRepository. There you will find a series of typedefs that were useful to the developers of DST. These typedefs arrange it so that instances of the following classes, for example, are passed by value:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ByteArray</th>
<th>ByteString</th>
<th>OrderedCollection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>Bag</td>
<td>Association</td>
<td>Dictionary</td>
</tr>
<tr>
<td>Point</td>
<td>Rectangle</td>
<td>Date</td>
<td>Time</td>
</tr>
</tbody>
</table>

IDL void and Smalltalk nil

There is no Smalltalk equivalent of void, just as there is no IDL equivalent of nil. Thus, it is important to be aware of the following:

- A remote object, when sent an operation whose signature specifies a void return value, will answer a local nil. The same object, sent the same message locally, could answer self. (For consistency, you can make the methods answer nil locally as well.)
• A remote object, sent an operation whose signature specifies a base or defined type as a return value, will generate a marshalling error if the remote object answers nil. (This is common if you fail to initialize instance variables with an object of the appropriate kind.)

• A remote object, sent an operation whose signature specifies an interface as a return value, may answer an object reference to a remote nil.
Overview

This chapter describes the mapping of OMG IDL constructs to Smalltalk constructs. You must use IDL to define the interfaces for an application’s remotely-accessible objects. In these interfaces, you define the externally-visible functionality of each object (but you implement this functionality elsewhere in VisualWorks Smalltalk).

A critical part of the ORB’s activities is the translation service (via a language binding) between the local language (such as Smalltalk) and IDL, the language-neutral Interface Definition Language that all ORBs speak as their common language. This chapter describes the IDL semantics, gives the syntax for IDL grammatical constructs and the Smalltalk-to-IDL language binding implemented in Distributed Smalltalk, and explains how to use it.

Constraints on Smalltalk Mappings

- Whenever possible, IDL types are mapped directly to existing, portable Smalltalk classes.
- The Smalltalk mapping only describes the public (client) interface to Smalltalk classes and objects supporting IDL. Individual IDL compilers or CORBA implementations might define additional private interfaces.
- The implementation of IDL interfaces is left unspecified. Implementations may choose to:
  - Map each IDL interface to a separate Smalltalk class
Default Mapping for IDL to Smalltalk

- Provide one Smalltalk class to map all IDL interfaces
- Allow arbitrary Smalltalk classes to map IDL interfaces
- Because of the dynamic nature of Smalltalk, the mapping of the \textit{any} and \textit{union} types is such that an explicit mapping is unnecessary. Instead, the value of the \textit{any} and \textit{union} types can be passed directly. In the case of the \textit{any} type, the Smalltalk mapping will derive a TypeCode which can be used to represent the value. In the case of the \textit{union} type, the Smalltalk mapping will derive a discriminator which can be used to represent the value.
- The explicit passing of environment and context values on operations is not required.
- Except in the case of object references, no memory management is required for data parameters and return results from operations. All such Smalltalk objects reside within Smalltalk memory, and so garbage collection will reclaim their storage when they are no longer used.

Default Mapping for IDL to Smalltalk

The use of underscore characters in IDL identifiers is not allowed in all Smalltalk language implementations. Thus, a conversion algorithm is required to convert names used in IDL to valid Smalltalk identifiers.

To convert an IDL identifier to a Smalltalk identifier, remove each underscore and capitalize the following letter (if it exists). For example:

- `add_to_copy_map` becomes `addToCopyMap`
- `describe_contents` becomes `describeContents`

Smalltalk implementations generally require that class names and global variables have an uppercase first letter, while other names have a lowercase first letter.

One aspect of the language mapping can cause an IDL compiler to map incorrectly to Smalltalk code resulting in name space collisions. Because Smalltalk implementations generally only support a global name space, and disallow underscore characters in identifiers, the mapping of identifiers used in IDL to Smalltalk identifiers can result in a name collision. As an example of name collision, consider the following IDL declaration:
interface Example {
    void sample_op () ;
    void sampleOp () ;
};

Both of these operations map to the Smalltalk selector sampleOp. In order to prevent name collision problems, each implementation of the IDL language binding must support an explicit naming mechanism, which can be used to map an IDL identifier into an arbitrary Smalltalk identifier. Distributed Smalltalk uses #pragma as the mechanism.

Handling Return Values

IDL and Smalltalk message syntaxes both allow zero or more input parameters to be supplied in a request. For return values, Smalltalk methods yield a single result object, whereas IDL allows an optional result and zero or more out or inout parameters to be returned from an invocation. In this binding, the non-void result of an operation is returned as the result of the corresponding Smalltalk method, whereas out and inout parameters are to be communicated back to the call via instances of a class conforming to the CORBAParameter protocol, passed as explicit parameters.

To create an object that supports the CORBAParameter protocol, the message asCORBAParameter can be sent to any Smalltalk object. This will return a Smalltalk object conforming to the CORBAParameter protocol, whose value will be the object it was created from. The value of that CORBAParameter object can be subsequently changed with the value: message. asCORBAParameter is implemented in Object and returns a ValueHolder, which latter is used to represent inout and out parameters that are present in addition to the return value.

Memory Usage

One of the design goals is to make every Smalltalk object used in the mapping a pure Smalltalk object: namely datatypes used in mappings do not point to operating system defined memory. This design goal permits the mapping and users of the mapping to ignore memory-management issues, since Smalltalk handles this itself (via garbage collection). Smalltalk objects which are used as object references may contain pointers to operating system memory, and so must be freed in an explicit manner.
Limitations

The proposed language mapping places limitations on the use of certain types defined in IDL.

For the any and union types, specific integral and floating point types may not be able to be specified as values. The implementation will map such values into an appropriate type, but if the value can be represented by multiple types, the one actually used cannot be determined. For example, consider the union definition below:

```smalltalk
union Foo switch (long) {
  case 1: long x;
  case 2: short y;
}
```

When a Smalltalk object corresponding to this union type has a value that fits in both a long and a short, the Smalltalk mapping can derive discriminator 1 or 2, and map the integral value into either a long or short value (corresponding to the value of the discriminator determined).

This limitation can be overcome in some cases by a careful ordering of the union types, and in all cases by use of DSTDUnion, which allow explicit specification of the value and the discriminator for parameters.

Mapping of IDL Elements to Smalltalk

The following overview provides a brief description of the mapping of IDL elements to the Smalltalk language.

<table>
<thead>
<tr>
<th>IDL Element</th>
<th>Smalltalk Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>object references</td>
<td>Smalltalk objects which represent CORBA objects. The Smalltalk objects must respond to all messages defined by the CORBA objects interface.</td>
</tr>
<tr>
<td>interfaces</td>
<td>A set of messages that Smalltalk objects which represent object references must respond to. The set of messages corresponds to the attributes and operations defined in the interface and inherited interfaces.</td>
</tr>
<tr>
<td>operations</td>
<td>Smalltalk messages.</td>
</tr>
<tr>
<td>attributes</td>
<td>Smalltalk messages.</td>
</tr>
<tr>
<td>constants</td>
<td>Smalltalk objects available in CORBAConstants dictionary.</td>
</tr>
</tbody>
</table>
A large number of special mappings used in DST are defined in the SmalltalkTypes module in DSTRepository. It is helpful to become familiar with these default mappings between IDL and Smalltalk types in DST.

Of particular interest are the *OrNull types, which handle many cases of translating between IDL `void` and Smalltalk nil. Note that if you are not going to use *OrNull unions in specifying return values, it is very important that you carefully and fully initialize classes that will be accessed remotely. Refer to “IDL `void` and Smalltalk nil” for details.

<table>
<thead>
<tr>
<th>IDL Element</th>
<th>Smalltalk Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>integral types</td>
<td>Smalltalk objects which conform to the <code>Integer</code> class.</td>
</tr>
<tr>
<td>floating point type</td>
<td>Smalltalk objects which conform to the <code>Float</code> class.</td>
</tr>
<tr>
<td>boolean type</td>
<td>Smalltalk true or false objects.</td>
</tr>
<tr>
<td>enumeration types</td>
<td>Smalltalk objects which conform to the <code>CORBAEnum</code> protocol.</td>
</tr>
<tr>
<td>any type</td>
<td>Smalltalk objects that can be mapped into an IDL type.</td>
</tr>
<tr>
<td>structure types</td>
<td>Smalltalk objects which conform to the <code>Dictionary</code> class.</td>
</tr>
<tr>
<td>union types</td>
<td>Smalltalk objects which map to the possible value types of the IDL <code>union</code>, or which conform to the <code>CORBAUnion</code> protocol.</td>
</tr>
<tr>
<td>sequence type</td>
<td>Smalltalk objects which conform to the <code>OrderedCollection</code> class.</td>
</tr>
<tr>
<td>string type</td>
<td>Smalltalk objects which conform to the <code>String</code> class.</td>
</tr>
<tr>
<td>array type</td>
<td>Smalltalk objects which conform to the <code>Array</code> class.</td>
</tr>
<tr>
<td>exception type</td>
<td>Smalltalk objects which conform to the <code>Dictionary</code> class.</td>
</tr>
</tbody>
</table>
Mapping for Interface

Each IDL interface defines the operations that object references with that interface must support. In Smalltalk, each IDL interface defines the methods that object references with that interface must respond to.

Implementations are free to map each IDL interface to a separate Smalltalk class, map all IDL interfaces to a single Smalltalk class, or map arbitrary Smalltalk classes to IDL interfaces.

**CORBAName Method**

In Distributed Smalltalk, the CORBAName is the tie between an interface and its corresponding implementation. That is, any object that has an interface (that is, a CORBA object), must implement the CORBAName, which specifies the interface name. When the ORB receives an incoming request, it locates the interface in the Interface Repository, and the Smalltalk class by this CORBAName.

Thus, in the Smalltalk class Depot, repository>>CORBAName would map the class Depot to the IDL interface Depot as follows:

```
CORBAName
^#::CentralOffice::Depot'
```

The CORBAName method is usually implemented on the instance side of a class definition and provides the link to the interface for instances of the class, but it may also be implemented on the class side, in which case the CORBAName method provides the link to the interface for the class.

**Getting Information About an Interface**

Object references to both local and remote objects supporting IDL interfaces are via the normal Smalltalk object reference mechanism. To obtain the interface associated with the object reference, invoke the getInterface method; this returns the actual interface meta object which models its type information.

In addition, and when the programmer knows she is dealing with a surrogate object reference, the interface will return the local repository’s meta object for that type. Access to local meta objects is considerably faster, of course.
Mapping for Objects

A CORBA object is represented in Smalltalk as a Smalltalk object called an object reference. The object reference must respond to all messages defined by that CORBA object’s interface.

An object reference can have a value which indicates that it does not represent a CORBA object. This value is the standard Smalltalk value nil.

Invocation of Operations

IDL and Smalltalk message syntaxes both allow zero or more input parameters to be supplied in a request. For return values, Smalltalk methods yield a single result object, whereas IDL allows an optional result and zero or more out or inout parameters to be returned from an invocation. In this binding, the non-void result of an operation is returned as the result of the corresponding Smalltalk method, whereas out and inout parameters are to be communicated back to the caller via instances of a class conforming to the CORBAParameter protocol, passed as explicit parameters.

For example, the following operations:

boolean definesProperty(in string key);
void defines_property(
  in string key,
  out boolean is_defined);

are used as follows:

self
definesProperty: aString
isDefined: (aBool := nil asCORBAParameter).

As another example, the operations:

boolean has_property_protection(
  in string key,
  out Protection pval);

ORBStatus create_request (in Context ctx,
  in Identifier operation,
  in NVList arg_list,
  inout DynamicInvocation::NamedValue result,
  out Request request,
  in Flags reg_flags);
would be invoked as:

```smalltalk
aBool := self
    hasPropertyProtection: aString
    pval: (protection := nil asCORBAParameter).

aStatus := ORBObj
    createRequest: aContext
    argList: anNVList
    result: (result := aNamedValue asCORBAParameter)
    request: (request := nil asCORBAParameter)
    reqFlags: aFlags.
```

The return value of IDL operations that are specified with a `void` return type is undefined.

**Mapping for Attributes**

IDL attribute declarations are a shorthand mechanism to define pairs of simple accessing operations: one to get the value of the attribute and one to set it. Such accessing methods are common in Smalltalk programs as well, so attribute declarations are mapped to standard methods to get and set the named attribute value, respectively.

For example:

```
attribute string title;
readonly attribute string my_name;
```

means that Smalltalk programmers can expect to use `title` and `title:` methods to get and set the `title` attribute of the CORBA object, and the `myName` method to retrieve the `my_name` attribute.

Although attributes provide a shorthand for setters and getters, the syntax for attributes does not allow you to specify the exceptions that might be returned, as does an ordinary operation declaration.

**Readonly Attributes for Security**

By default, attributes are read-write. However, you can declare read-only attributes to keep clients from changing what they should not. For example, the `CompoundLifecycle::LinkRead` module declares a variety of attributes including these read-only attributes:

```
readonly attribute LinkSet head;
readonly attribute boolean is_existence_ensuring;
```

An IDL operation can set the value of a read-only attribute in its implementation.
Mapping for Constants

IDL allows constant expressions to be declared globally as well as in interface and module definitions. IDL constant values are stored in a dictionary named CORBAConstants under the fully qualified name of the constant, not subject to the name conversion algorithm. The constants are accessed by sending the at: message to the dictionary with an instance of a String whose value is the fully qualified name.

For example, given the following IDL specification:

```idl
module ApplicationBasics {
    const CopyDepth shallow_cpy = 4;
};
```

the `ApplicationBasics::shallow_cpy` constant can be accessed with the following Smalltalk code:

```smalltalk
value := CORBAConstants at: '::ApplicationBasics::shallow_cpy'.
```

After this call, the `value` variable will contain the integral value `4`.

Here is another example where constant declarations map Smalltalk names to IDL:

```smalltalk
const LinkType containment_link = 1;
```

Tells the compiler to substitute link type 1 whenever the identifier `containment_link` is used.

```smalltalk
const LinkType reference_link = 3;
```

Substitute link type 3 when `reference_link` is used.

```smalltalk
const LinkType designation_link = 5;
```

Substitute link type 5 when `designation_link` is used.

```smalltalk
const LinkType weak_link = 7;
```

Substitute link type 7 when `weak_link` is used.

Getting More Information About a Constant

IDL constant values are stored in the global dictionary CORBAConstants under the fully qualified name of the constant.
Mapping for Basic Data Types

Each of the parameters of an IDL operation definition has an associated data type which must be declared in advance, since IDL is a statically-typed definition language. This means that some operations that can be implemented in Smalltalk cannot be declared in IDL at all. It is also complicated by the fact that, all Smalltalk values are instances of a Smalltalk class. In order to be able to construct valid calls on IDL operations, however, a mapping must be devised. Fortunately, the following type-to-class mapping works well enough and useful distributed systems can be constructed.

**Base Type Mapping**

Since Smalltalk is not a typed language, various classes in the Magnitude categories are used to map Smalltalk objects to IDL data types.

- Smalltalk Magnitude classes map directly onto the required IDL basic datatypes, and the subclasses of this abstract class are concerned with their representation in all situations.
- Boolean values TRUE and FALSE are used by the Smalltalk programmer to represent IDL boolean types.
- Character values are used by the Smalltalk programmer to represent IDL char types.
- Float and Double values are used by the Smalltalk programmer to represent IDL float and double types.
- Integer values are used by the Smalltalk programmer to represent IDL long and short integer types.
- Character and SmallInteger values may be used by the Smalltalk programmer to represent IDL octet types.

The following basic datatypes are mapped into existing Smalltalk classes. In the case of short, unsigned short, long, unsigned long, float, double, and octet, the actual class used is left up to the implementation, for the following reasons:

- There is no standard for Smalltalk that specifics integral and floating point classes and the valid ranges of their instances.
- The classes themselves are rarely used in Smalltalk. Instances of the classes are made available as constants included in code, or as the result of computation.
The basic datatypes are mapped as follows:

**short**
An IDL short integer falls in the range \([-2^{15}, 2^{15} - 1]\). In Smalltalk, a short is represented as an instance of an appropriate integral class.

**long**
An IDL long integer falls in the range \([-2^{31}, 2^{31} - 1]\). In Smalltalk, a long is represented as an instance of an appropriate integral class.

**long long**
An IDL long long integer falls in the range \([-2^{63}, 2^{63} - 1]\). In Smalltalk, a long long is represented as an instance of an appropriate integral class.

**unsigned short**
An IDL unsigned short integer falls in the range \([0, 2^{16} - 1]\). In Smalltalk, an unsigned short is represented as an instance of an appropriate integral class.

**unsigned long**
An IDL unsigned long integer falls in the range \([0, 2^{32} - 1]\). In Smalltalk, an unsigned long is represented as an instance of an appropriate integral class.

**unsigned long long**
An IDL unsigned long long integer falls in the range \([0, 2^{64} - 1]\). In Smalltalk, an unsigned long long is represented as an instance of an appropriate integral class.

**float**
An IDL float type represents IEEE single-precision (32-bit) floating point numbers. In Smalltalk, a float is represented as an instance of an appropriate floating point class.

**fixed**
An IDL fixed is represented as an instance of an appropriate fractional class with a fixed denominator (see “Mapping for Fixed Type”).

**double**
An IDL double type represents IEEE single-precision 64-bit) floating point numbers. In Smalltalk, a double is represented as an instance of an appropriate floating point class.

**long double**
An IDL long double conforms to the IEEE double extended (a mantissa of at least 64 bits, a sign bit, and an exponent of at least 15 bits) floating point standard (ANSI/IEEE Std 754-1985). In Smalltalk, a long double is represented as an instance of an appropriate floating point class.
char
An IDL char holds an 8-bit quantity mapping to the ISO Latin-1 8859.1 character set. In Smalltalk, a char is represented as an instance of Character.

wchar
An IDL wchar defines a wide character from any character set. A wide character is represented as an instance of the Character class.

boolean
An IDL boolean may hold one of two values: TRUE or FALSE. In Smalltalk, a boolean is represented by the values true or false, respectively.

octet
An IDL octet is an 8-bit quantity that undergoes no conversion during transmission. In Smalltalk, an octet is represented as an instance of an appropriate integral class with a value in the range [1,255].

Mapping for Fixed Type
An IDL fixed is represented as an instance of an appropriate fractional class with a fixed denominator.

Smalltalk class FixedPoint is the only Smalltalk class with an explicit, default mapping to the IDL fixed type.

Mapping for the Any Type
Due to the dynamic nature of Smalltalk, where the class of objects can be determined at runtime, an explicit mapping of the any type to a particular Smalltalk class is not required. Instead, wherever an any is required, the user may pass any Smalltalk object which can be mapped into an IDL type. For instance, if an IDL structure type is defined in an interface, a Dictionary for that structure type will be mapped. Instances of this class can be used wherever an any is expected, since that Smalltalk object can be mapped to the IDL structure.

Likewise, when an any is returned as the result of an operation, the actual Smalltalk object which represents the value of the any data structure will be returned.

Any Smalltalk class may be mapped to an instance of an IDL type any in an operation invocation parameter list. By default, type any output parameters and results are returned as the value of the object.
However, type any should not be used indiscriminately, because it has additional overhead. The ORB has to marshal information about exactly which type of object is coming across as an any in addition to the value itself.

**CORBAType Method**

The CORBAType method specifies how an object is to be marshaled when it is passed under the umbrella of type any. By default, we pass by reference. If you want to pass an object by value when it is passed as an any, you must override CORBAType.

The default implementation of CORBAType is in class Object. It returns a meta object for the object’s interface, which marshals it as an object reference, not an IDL data type. This object reference corresponds to the CORBAName method associated with this object (a fully qualified interface name). Browse implementors of CORBAType to see other implementations.

As a rule, you should implement the CORBAType method in any class that uses the CLASS pragma in its IDL interface definition. For example, see the CosNaming module in DSTRpository, where pragmas are defined for classes DSTNameComponent and DSTName.

**Mapping for Enum**

IDL enumerators are stored in a dictionary named CORBAConstants under the fully qualified name of the enumerator, not subject to the name conversion algorithm. The enumerators are accessed by sending the at: message to the dictionary with an instance of a String whose value is the fully qualified name.

These enumerator Smalltalk objects must support the aCORBAEnum protocol, to allow enumerators of the same type to be compared. The order in which the enumerators are named in the specification of an enumeration defines the relative order of the enumerators. The protocol must support the following instance methods:

- `< aCORBAEnum
  Answers true if the receiver is less than aCORBAEnum, otherwise answers false.

- `<= aCORBAEnum
  "Less than or equal to" comparison.
Mapping for Struct Types

An IDL struct is mapped to an instance of the Dictionary class. The key for each IDL struct member is an instance of Symbol whose value is the name of the element converted according to the algorithm given earlier.

For example, given the following IDL declaration:

```idl
struct Binding {
  Name binding_name;
  BindingType binding_type;
};
```

the binding_name element can be accessed as follows:

```smalltalk
aBindingStruct at: #bindingName
```

and set as follows:

```smalltalk
aBindingStruct at: #bindingName put: aName
```
Mapping for Union Types

For IDL union types, two binding mechanisms are provided: an implicit binding and an explicit binding. Although not required, implementations may choose to provide both implicit and explicit mappings for other IDL types, such as structs and sequences. In the explicit mapping, the IDL type is mapped to a user-specified Smalltalk class. The implicit binding takes maximum advantage of the dynamic nature of Smalltalk and is the least intrusive binding for the Smalltalk programmer. The explicit binding retains the value of the discriminator and provides greater control for the programmer.

Although the particular mechanism for choosing implicit vs. explicit binding semantics is implementation specific, all implementations must provide both mechanisms. Binding semantics is expected to be specifiable on a per-union declaration basis, for example using #pragmas.

Implicit Binding

Wherever a union is required, the user may pass any Smalltalk object that can be mapped to an IDL type, and whose type matches one of the types of the values in the union. Consider the following example:

```smalltalk
structure S { long x; long y; };
union U switch (short) {
    case 1: S s;
    case 2: long 1;
    default: char c;
};
```

In the example above, a Dictionary for structure S will be mapped. Instances of Dictionary with runtime elements as defined in structure S, integral numbers, or characters can be used wherever a union of type U is expected. In this example, instances of these classes can be mapped into one of the S, long, or char types, and an appropriate discriminator value can be determined at runtime.

Likewise, when a union is returned as the result of an operation, the actual Smalltalk object which represents the value of the union will be returned.
Explicit Binding

Use of the explicit binding will result in specific Smalltalk classes being accepted and returned by the ORB. Each union object must conform to the CORBAUnlon protocol. This protocol must support the following instance methods:

- discriminator
  Answers the discriminator associated with the instance

- discriminator: anObject
  Sets the discriminator associated with the instance

- value
  Answers the value associated with the instance

- value: anObject
  Sets the value associated with the instance

To create an object that supports the CORBAUnlon protocol, the instance method asCORBAUnlon: aDiscriminator can be invoked by any Smalltalk object. This method will return a Smalltalk object conforming to the CORBAUnlon protocol, whose discriminator will be set to aDiscriminator and whose value will be set to the receiver of the message.

Mapping for Sequence Types

Instances of the OrderedCollection class are used to represent IDL elements with the sequence type.

A sequence is a one-dimensional array with two characteristics: a subtype, and an optional maximum size. Use a “< >” combination to specify the type of data that belongs in a sequence, and optionally, the upper limit of elements. For example:
Mapping for String Types

Instances of the Smalltalk String class are used to represent IDL elements with the string type.

**Strings**

Smalltalk strings and their subclasses may be passed and will be returned by IDL operations involving string arguments. A string can be unbounded or can have a maximum size (specified via the “< >” combination). For example:

<table>
<thead>
<tr>
<th>Example</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string username is unbounded.</td>
</tr>
<tr>
<td>string&lt;25&gt;</td>
<td>The string ChartLabel may be no longer than 25 characters.</td>
</tr>
</tbody>
</table>

Mapping for Wide String Types

An IDL wide string is represented as an instance of an appropriate Smalltalk String class.

Mapping for Array Types

Instances of the Smalltalk array class are used to represent IDL elements with the array type.
Mapping for Exception Types

Each defined exception type is mapped to an instance of Dictionary.

Exception handling is implemented using the VisualWorks Signal exception handling mechanisms. Thus to raise an exception, the program can simply invoke #error:.

Since IDL exceptions are allowed to have arbitrary structured values returned with the exception, the programmer needs a way to specify this information as well. Fortunately, Smalltalk is up to the task. Consider the example Smalltalk fragment, which raises the BAD_INV_ORDER exception (one of the standard exceptions defined in interface Object):

```
^ErrorSignal
  raiseWith: (Array
    with: #'BAD_INV_ORDER'
    with: (Array
      with: minor
      with: #NO))
  errorString: 'routine invocations out of order'
```

In order to allow the ORB to correctly return the error result structure to the sender of the method, an array must be returned as the parameter of the error. Here, the symbolic name of the event is provided in an array along with the type-structure representation of the required error result values. These values will be marshalled by the ORB to ensure that the same exception can be raised in the context of the client of the remote operation.

As with normal Signal exceptions, a handle:do: recovery block may be used to catch and recover from these exceptions. The main difference is that the ORB call context will have already unwound to the site of the remote call before the exception is raised. This greatly limits the extent to which recovery can be accomplished.

For example, the NamingContext interface in the CosNaming module declares these exceptions:
interface NamingContext {
  …
  enum NotFoundReason {missing_node, not_context, not_object};

  exception NotFound (NotFoundReason why; Name rest_of_name; );
  exception CannotProceed {NamingContext ctx; Name rest_of_name; }
  exception InvalidName ();
  exception AlreadyBound ();
  exception NotEmpty ();
  …
};

For exceptions declared with empty braces, no additional information is available to the client code when the exception is raised.

Exceptions can be declared anywhere within an IDL module. Exceptions declared at the beginning of the module apply to the module as a whole; exceptions declared within an interface apply to that interface only.

Note, however, that [1] you cannot specify what exceptions an operation declared as an attribute might return, and [2] if you do not specify what exceptions might be returned by an operation, you will, when an exception is returned, get the UNKNOWN exception.

**Getting More Information on Exceptions**

The Object interface (in DSTRepository>>core IFs>>CORBA) declares all the standard exceptions.

Each DSTexception meta object is also an instance of the ExceptionDef interface in the Repository, and may be accessed accordingly.

IDL attribute names containing an underscore (_) character are automatically converted to conventional Smalltalk using the capitalization rule.

**Mapping for Operations**

IDL operations having zero parameters map directly to Smalltalk unary messages, while IDL operations having one or more parameters correspond to Smalltalk keyword messages. To determine the default selector for such an operation, begin with the IDL operation identifier and concatenate the parameter name of each parameter followed by a colon, ignoring the first parameter. The selector name is subject to the identifier conversion algorithm.
For example, the following IDL operations:

```csharp
void add_to_copy_map(
    in CORBA::ORBId id,
    in LinkSet link_set);

void connect_push_supplier(
    in EventComm::PushSupplier push_supplier);

void add_to_delete_map(
    in CORBA::ORBId id,
    in LinkSet link_set);
```

become selectors:

```csharp
addToCopyMap:linkSet:
connectPushSupplier:
addToDeleteMap:linkSet:
```

### Implicit Arguments to Operations

Unlike the C mapping, where an object reference, environment, and optional context must be passed as parameters to each operation, the Smalltalk mapping does not require these parameters to be passed to each operation.

The object reference is provided in the client code as the receiver of a message. So although it is not a parameter on the operation, it is a required part of the operation invocation.

This mapping defines the CORBAExceptionEvent protocol to convey exception information in place of the environment used in the C mapping. This protocol can either be mapped into native Smalltalk exceptions or used in cases where native Smalltalk exception handling is unavailable.

A context expression can be associated with the current Smalltalk process by sending the message `corbaContext:` to the current process, along with a valid context parameter. The current context can be retrieved by sending the `corbaContext` message to the current process.

The current process may be obtained by sending the message `activeProcess` to the Smalltalk global variable named `Processor`. 
Argument-Passing Considerations

All parameters passed into and returned from the Smalltalk methods used to invoke operations are allocated in memory maintained by the Smalltalk virtual machine. Thus, explicit free( )ing of the memory is not required. The memory will be garbage-collected when it is no longer referenced.

The only exception is object references. Since object references may contain pointers to memory allocated by the operating system, it is necessary for the user to explicitly free them when no longer needed. This is accomplished by using the operation release of the CORBA::Object interface.

Unmapped Interfaces

It is sometimes convenient or necessary to define an interface without providing an implementation. For example, DST defines an interface,

```
interfaceNamingContextExt : NamingContext
```

in CosNaming, to support ORBs that are ahead of DST in CORBA compliance. Browse this interface definition for an example.

Handling Exceptions

IDL allows each operation definition to include information about the kinds of run-time errors which may be encountered. These are specified in an exception definition which declares an optional error structure which will be returned by the operation should an error be detected. Since Smalltalk exception handling classes are not yet standardized between existing implementations, a generalized mapping is provided.

In this binding, the IDL compiler creates exception objects and populates the CORBAConstants dictionary. These exception objects are accessed from the CORBAConstants dictionary by sending the at: message with an instance of a String whose value is the fully qualified name. Each exception object must conform to the CORBAExceptionEvent protocol. This protocol must support the following instance methods:

```
corbaHandle: aHandlerBlock do: aBlock
```

Exceptions may be handled by sending an exception object the message corbaHandh:d: with appropriate handler and scoping blocks as parameters. The aBlock parameter is the Smalltalk block to evaluate. It is
Handling Exceptions

passed no parameters. The aHandlerBlock parameter is a block to evaluate when an exception occurs. It has one parameter: a Smalltalk object which conforms to the CORBAExceptionValue protocol.

corraRaise

Exceptions may be raised by sending an exception object the message corraRaise.

corraRaiseWith: aDictionary

Exceptions may be raised by sending an exception object the message corraRaiseWith:. The parameter is expected to be an instance of the Smalltalk Dictionary class, as described below.

For example, given the following IDL specification:

```smalltalk
interface NamingContext {
    ... 
    exception NotEmpty ();
    void destroy ()
        raises (NotEmpty);
    ... 
};
```

the NamingContext::NotEmpty exception can be raised as follows:

```smalltalk
(CORBAConstants at: '::NamingContext::NotEmpty')
corraRaise.
```

The exception can be handled as follows:

```smalltalk
(CORBAConstants at: '::NamingContext::NotEmpty')
corraHandle: [:ev I "error handling logic here"]
do: [aNamingContext destroy].
```

Exception Values

CORBA IDL allows values to be returned as part of the exception. Exception values are constructed using instances of the Smalltalk Dictionary class. The keys of the dictionary are the names of the elements of the exception, the names of which are converted using the name conversion algorithm. The following example, which illustrates how exception values are used:
interface NamingContext {
    ...
    exception CannotProceed {
        NamingContext ctx;
        Name rest_of_name;
    };
    Object resolve (in Name n)
        raises (CannotProceed);
    ...
};

would be raised as follows:

(CORBAConstants at: '::NamingContext::CannotProceed')
corbaRaiseWith: (Dictionary
    with: (Association key: #cxt value: aNamingContext)
    with: (Association key: #restOfName value: aName)).

The CORBAExceptionValue Protocol

When an exception is raised, the exception block is evaluated, passing it one argument which conforms to the CORBAExceptionValue protocol. This protocol must support the following instance message:

corbaExceptionValue

It answers the Dictionary with which the exception was raised.

Given the NamingContext interface defined above, the following code illustrates how exceptions are handled:

(CORBAConstants at: '::NamingContext::NotEmpty')
corbaHandle: [:ev |
    cxt := ev corbaExceptionValue at: #cxt.
    restOfName := ev corbaExceptionValue at: #restOfName
    do: [aNamingContext destroy].

In this example, the cxt and restOfName variables will be set to the respective values from the exception structure, if the exception is raised.

Pragmas

Pragmas are implementation-dependent messages to the IDL compiler that can be ignored by another compiler without harm. There are two categories of pragmas:

- RepositoryId pragmas: ID. version. prefix
- Distributed Smalltalk specific pragmas: selector, class, and access.
IDL-specific pragmas may appear anywhere in a specification.

**Mapping Pragmas to IDL Types**

<table>
<thead>
<tr>
<th>Pragma</th>
<th>IDL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>module, interface, attribute, operation, typedef, constant, exception</td>
</tr>
<tr>
<td>prefix</td>
<td>anywhere (no restrictions)</td>
</tr>
<tr>
<td>version</td>
<td>module, interface, attribute, operation, typedef, constant, exception</td>
</tr>
<tr>
<td>selector</td>
<td>operation</td>
</tr>
<tr>
<td>class</td>
<td>struct, enum, union, typedef</td>
</tr>
<tr>
<td>access</td>
<td>operation</td>
</tr>
</tbody>
</table>

**RepositoryIds**

RepositoryIds are globally unique values that can be used to establish the identity of information in the repository. A RepositoryId is represented as a string, allowing programs to store, copy, and compare them without regard to the structure of the value. It does not matter what format is used for any particular RepositoryId. However, conventions are used to manage the name space created by these IDs. All repository objects will have RepositoryIds (module, interface, attribute, operation, typedef, constant, exception).

RepositoryIds may be associated with IDL definitions in a variety of ways: Installation tools might generate them, they might be defined with pragma's in IDL source, or they might be supplied with the package to be installed.

The format of the ID is a short format name followed by a colon (:) followed by characters according to the format. The three formats are:

- **IDL format derived from IDL names**
  
  If no RepositoryId is specified, the system will generate an IDL format RepositoryId.

- **DCE format**

- **Local format that is intended for short-term use, e.g., in a development environment**.
Note: IDL format is the recommended type. In general, it is preferred to let the system allocate the RepositoryId.

IDL Format

The IDL format for RepositoryIds primarily utilizes IDL-scoped names to distinguish between definitions. It also includes an optional unique prefix, and major and minor version numbers.

IDL format RepositoryIDs consist of three components, separated by colons (:).

The first component is the format name:

IDL

The second component is a list of identifiers, separated by slashes (/). These identifiers are arbitrarily long sequences of alphabetic, digit, underscore (_), hyphen (-), and period (.) characters. Typically, the first identifier is a unique prefix, and the rest are the IDL Identifiers that make up the scoped name of the definition.

The third component is made up of major and minor version numbers, in decimal format, separated by a period (.). When two interfaces have RepositoryIds differing only in minor version number, it can be assumed that the definition with the higher version number is upwardly compatible with (i.e. can be treated as derived from) the one with the lower minor version number.

For example, the RepositoryId for the initial version of interface Printer defined on module Office by an organization known as “ABCCo” might be:

IDL:ABCCo/Office/Printer:1.0

This format makes it convenient to generate and manage a set of IDs for a collection of IDL definitions. The person creating the definitions sets a prefix (“ABCCo”), and the IDL compiler or other tool can synthesize all the needed IDs.

Because RepositoryIds may be used in many different computing environments and ORBs, as well as over a long period of time, care must be taken in choosing them. Prefixes that are known to be distinct for other reasons (e.g., trademarked names, domain names, UUIDs, etc.) are preferable to generic ones (e.g., Document).
Repositoryld Pragmas

DCE Format

DCE format Repositorylds start with the characters DCE: and are followed by the printable form of the UUID, a colon, and a single digit decimal minor version number, for example:

DCE:700dc518-0110-11ce-ac8f-0800090bSd3e:l

Local Format

Local format Repositorylds start with the characters LOCAL: and are followed by an arbitrary string. Local format IDs are not intended for use outside a particular repository, and thus do not need to conform to any particular convention. Local IDs that are just consecutive integers might be used within a development environment to have a very cheap way to manufacture the IDs while avoiding conflicts with well-known interfaces.

Note: DCE and Local formats are not recommended.

Repositoryld Pragmas

A mechanism is provided to include Repositorylds with published IDL specifications. A convention is specified for using #pragma directives to annotate IDL specifications with these IDs. Whether an IDL compiler uses these annotations directly, or some other tool is involved, is implementation defined.

Three IDL pragmas are specified in order to support arbitrary Repositoryld formats while supporting the IDL Repositoryld format with minimal annotation. An IDL compiler must either interpret these annotations as specified, or ignore them completely.

ID Pragma

An IDL pragma of the format:

```
#pragma ID <name> "<id>
```

associates an arbitrary Repositoryld string with a specific IDL name. The <name> can be a fully or partially scoped name or a simple identifier, interpreted according to the usual IDL name lookup rules relative to the scope within which the pragma is contained.

The use of an ID pragma is discouraged because the system will generate it for you.
Prefix Pragma

An IDL pragma of the format:

```
#pragma prefix "<string>"
```

sets the current prefix used in generating IDL format RepositoryIds. The specified prefix applies to RepositoryIds generated after the pragma until the end of the current scope is reached or another prefix pragma is encountered.

Version Pragma

A VERSION pragma is optional for interface, module, attribute, constant, exception, operation, and typedef declarations. An IDL pragma of the format:

```
#pragma version <name> <major>.<minor>
```

provides the version specification used in generating an IDL format RepositoryId for a specific IDL name. The <name> can be a fully or partially scoped name or a simple identifier, interpreted according to the usual IDL name lookup rules relative to the scope within which the pragma is contained. The <major> and <minor> components are decimal unsigned shorts. If no version pragma is supplied for a definition, version 1.0 is assumed.

Interfaces and Version Control

When a remote object reference is received by a client and the client wishes to send a message to that object, the client side ORB checks to determine if the interface is contained in the local repository. If the local repository contains the correct interface (identified by its RepositoryId) compatibility is assumed and the operation continues normally. If the local repository holds the correct interface, but the version fields of the RepositoryIds mismatch, then an exception is raised.

If you are using a shared repository, such an exceptions suggests that a new version of the interface should be brought into the repository.

A higher version of an interface must support all the operations of previous versions. Only one version of an interface is accessible (and stored) in the DSTRepository.

When you create a new version of an existing interface and assign (or reassign) it a version number:

- Do not change existing operation signatures.
Only add new operations at the lexical end of the interface definition. (That is, do not insert a new operation between existing operation declarations.)

If appropriate, add additional types for the new operations after the lexical end of the previously existing interface.

Generating Repository IDs

If no ID pragma is specified, a definition is globally identified by an IDL format RepositoryId.

The ID string is generated by starting with the string IDL: . Then, if any prefix pragma applies, it is appended, followed by a slash (/). Next, the components of the scoped name of the definition, relative to the scope in which any prefix that applies was encountered, are appended, separated by slashes. Finally, a colon (:) and the version specification are appended.

For example, the following IDL:

```
module M1 1
    typedef long T1;
    typedef long T2;
    #pragma ID T2 "DCE:d62207a2-011e-11ce-88b4-0800090b5d3e:3"
};
#pragma prefix "P1"
module M2 {
    module M3 {
        #pragma prefix "P2"
        typedef long T3;
    }
    typedef long T4;
    #pragma version T4 2.4
};
```

specifies types with the following scoped names and RepositoryIds:
For this scheme to provide reliable global identity, the prefixes used must be unique. Two non-colliding options are suggested: Internet domain names and DCE UUIDs.

Furthermore, in a distributed world, where different entities independently evolve types, a convention must be followed to avoid the same RepositoryId being used for two different types. Only the entity that created the prefix has authority to create new IDs by simply incrementing the version number. Other entities must use a new prefix, even if they are only making a minor change to an existing type.

Prefix pragmas can be used to preserve the existing IDs when a module or other container is renamed or moved.

```idl
module M4 {
    #pragma prefix P1/M2
    module M3 {
        #pragma prefix P2
        typedef long T3;
    };
    typedef long T4;
    #pragma version T4 2.4
};
```

This IDL declares types with the same global identities as those declared in module M2 above.

### Distributed Smalltalk Pragmas

In a Distributed Smalltalk Interface Repository, the most commonly used pragmas are class and selector.

#### Class Pragma

A class pragma is recommended for new data type declarations. It maps the declared data type to a Smalltalk class.

```
#pragma class <idl type> <Smalltalk class>
```
For example:

```smalltalk
#pragma class NameComponent DSTNameComponent
    struct NameComponent (Istring id; Istring kind; );
```

**Selector Pragma**

A selector pragma is recommended, but not required due to the default mapping rules for each operation declaration. It maps the declared operation to its Smalltalk implementation method.

```smalltalk
#pragma selector <idl operation> <Smalltalk method>
```

For example:

```smalltalk
#pragma selector rebind contextReBind:to:
    void rebind (in Name n, in Object obj);
```

**Access Pragma**

An access pragma is optional for operation declarations; it is used to check specified level(s) of access for authorized users to this operation. Classes that inherit from ORBObject and DSTPresenter can specify access control for any operation in their interfaces. To set access control for an operation in an interface, include the following line at the beginning of an operation definition (in the Interface Repository):

```smalltalk
#pragma access <idl operation> nameOfAccessValue
```

For example:

```smalltalk
#pragma access set admin
    void set (in AccessList access,
        in SymbolOrORBId user);
```

where the `nameOfAccessValue` is a string, such as read or admin (as specified in AccessSymbols). To see examples of how the access pragma is used, see class DSTRepository, module Security.

---

**About IDL and DSTRepository**

IDL is represented in the system in two ways:

- the text of the IDL is represented in the several methods of DSTRepository, and
- the marshaling machinery produced from the text is present as a privileged instance of class DSTModuleRepository.
With one exception, all the code you write in Distributed Smalltalk is standard Smalltalk code. The exception is in the Interface Repository (IR), which appears as class DSTRepository. Methods in this class are written in IDL. IDL is used here because this is the public access registry of objects available to all CORBA-compliant applications, regardless of language.

The class DSTRepository is used as a container of IDL code. An instance of DSTModuleRepository is the marshalling engine that converts messages sent to objRefs, and the return values from such calls, to and from the on-the-wire encodings defined in the CORBA specification. All messages that can be sent between images are registered here.

**Editing the Interface Repository**

You can edit the Interface Repository, DSTRepository, using any system browser. Note that there is no attempt to prevent simultaneous editing by multiple users.

To open a browser from the DST Tool, select **Tools → Browse Repository**, and then select **Edit → Definition**. To get read-only access, select **View → as Text** (or, **as Picture**).

**IDL Mapping to Smalltalk**

The DSTMetaObject class, in conjunction with its subclasses and the immediate subclasses of ORBObject, implements the IDL mapping to the Smalltalk programming language. These classes provide the Smalltalk programmer with mechanisms for expressing the following IDL concepts:

<table>
<thead>
<tr>
<th>Classes</th>
<th>Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSTTypeBase,</td>
<td>All IDL basic datatypes</td>
</tr>
<tr>
<td>subclasses</td>
<td></td>
</tr>
<tr>
<td>DSTTypeConstr,</td>
<td>All IDL constructed datatypes</td>
</tr>
<tr>
<td>DSTTypeTemplate</td>
<td></td>
</tr>
<tr>
<td>DSTconstant</td>
<td>References to constants defined in IDL</td>
</tr>
<tr>
<td>DSTObjRef</td>
<td>References to objects defined in IDL</td>
</tr>
<tr>
<td>DSTOperation,</td>
<td>Invocations of operations, including passing</td>
</tr>
<tr>
<td>DSTparameter,</td>
<td>parameters and receiving results</td>
</tr>
<tr>
<td>DSTsignature</td>
<td></td>
</tr>
<tr>
<td>DSTException</td>
<td>Exceptions, including what happens when an</td>
</tr>
<tr>
<td></td>
<td>operation raises an exception and how the</td>
</tr>
<tr>
<td></td>
<td>exception parameters are accessed</td>
</tr>
</tbody>
</table>
In addition to defining the language mapping from IDL to Smalltalk, these meta objects are themselves remotely accessible and provide the Interface Repository behavior which is defined for all CORBA implementations.

<table>
<thead>
<tr>
<th>Classes</th>
<th>Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSTattribute</td>
<td>Access to attributes</td>
</tr>
<tr>
<td>ORBObject</td>
<td>Signatures for the operations defined by the ORB, such as the dynamic invocation interface, and the object adapters</td>
</tr>
</tbody>
</table>
Before an application can run in a distributed environment, its object interfaces must be available to remote calls via the ORB. To do this, you must add hooks in both the Smalltalk class definitions and in the IDL interface definitions, so when an interface is invoked it can be found.

Specifically, you need to:

- Identify classes as “factories,” which create instances of objects.
- Define interfaces, which specify what services an object can provide, and register them in the Interface Repository.

This chapter explains how to make a class a factory and register it with the interface repository. It also describes ways for working and maintaining interfaces in the repository.

Making a Class a Factory

According to the CORBA specification, an object that can be instantiated to create another object is a factory. Thus, all non-abstract Smalltalk classes are potentially factories.

In order for a class to be identified as a factory, it must be registered with the ORB as a factory. To do so, the class must define this instance method:

abstractClassId

This method returns an abstract class identifier (a UUID) for the class, which the ORB uses to locate the class, and in turn to instantiate the desired object.

The method typically looks like this:
abstractClassId

`c815c088-4901-0000-02d8-2421ae000000` asUUID

The trick is that the UUID **must be unique for this class** in the image. To generate a unique identifier for a class, evaluate the following in a workspace, using `PrintIt`:

```smalltalk
ORBObject newId
```

Then copy the resulting string into the method body, in place of the string in the above example.

**Note:** *DO NOT* copy the UUID from the body of any existing `abstractClassId` method. This number must be unique for this class in the image.

With this method defined, the ORB recognizes and registers the class as a factory when it initializes. An ORB initializes its factories registry when it starts. You can also force initialization in these ways:

- In the DST Tool, choose `Initialize ➔ Initialize Factories`.
- Evaluate the expression:

  ```smalltalk
  ORBObject initializeFactories
  ```

---

**Adding an Interface to the Interface Repository**

The class `DSTRepository` is used as a container of IDL code. An instance of `DSTmoduleRepository` is the marshalling engine that converts messages sent to `objRefs`, and the return values from such calls, to and from the on-the-wire encodings defined in the CORBA specification. All messages that can be sent between images are registered here.

When a remote object (client) requests a service from a local object, the local ORB’s Interface Repository is used to identify the local object, which services it can perform, and which messages are sent to provide these services. The association between an interface and its supporting class is made with the `CORBAName` message, which supplies the interface name.
To add an interface to the repository, do the following:

1. **Create the CORBAName method.**
   
   Every class that implements an interface registered in the Interface Repository must implement the CORBAName instance method. This identifies a specific module and interface in the Interface Repository. Code for this method (usually in the message category repository) looks something like this:
   
   ```smalltalk
   CORBAName
   "Answer the name of the receiver’s CORBA interface in the IDL repository."
   
   ^#'::DSTSampleComputeService::DSTSampleComputeServiceInterface'
   
   The two words in the CORBAName method code correspond to the name of the module and the name of the interface as specified in DSTRepository.

2. **Generate an IDL interface definition.**

   Send the message `asIDLDefinition` to your interface class, to create a first draft of the interfaces for a Smalltalk class:
   
   ```smalltalk
   yourClassName asIDLDefinition
   
   The interface definition created will include an operation for every method in the class, many of which will be inappropriate for distributed access and can thus be removed. `asIDLDefinition` also makes best guesses at return types and parameters, but you will probably need to edit these as well.

   Once generated, the definition is displayed in a text window.

   This interface can also be generated using the IDL Generator tool. Refer to “Creating an IDL Module Using the IDL Generator” below for instructions.
3 Edit the definition, as required.

   You need to:
   
   • Verify that the interface name is correct, and corresponds to the name you supplied in the CORBAName method for the class.
   • Add parent interfaces, as needed.
   • Delete operation definitions that correspond to messages that should not be available to remote clients (including but not limited to messages in private protocols).
   • For the remaining operation definitions, edit the result types, parameters, and operation names.
   • Edit definitions for types, constants, attributes, exceptions (for the module as a whole or for specific interfaces).
   • Delete unnecessary pragmas. For example selector pragmas are not needed for any unary messages, as the mapping between the Smalltalk message and IDL operation is clear.
   • Add access pragmas for access controlled operations.

4 Browse DSTRepository to add these interfaces to a new module.

   • Copy the text from the workspace and paste it into the module you are creating.
   • Add a final " };") at the end of the module.
   • Verify that each interface ends properly with a brace and a semicolon: "};".

For specifics on IDL syntax and Smalltalk language bindings, see Chapter 8, “Defining IDL Interfaces.”.
Creating an IDL Module Using the IDL Generator

The IDL Generator is used when you have a Smalltalk class and want to generate the IDL interface for it. This is an alternative to using the asIDLDefinition method described above. Note the difference between server and client classes distinguishes between the requester of the service (client) and the provider of the service (server).

To launch the IDL Generator you can choose the Distributed Smalltalk main window’s menu option **DST → IDL Generator** or you can execute the following:

```
DSTIDLTool open
```

Use the IDL Generator to specify the interface and its characteristics.

1. Choose the class(es) of the server object(s).

   Click in the **Servers** list to choose one or more classes whose interface(s) will appear in this module. (Related interfaces, such as a presentation and semantic class, are usually included in the same module.)

   A check indicates that the class is marked for inclusion.

   To limit the list of classes, in the box above the class list, specify a pattern that will be used as a filter. For example, to see a list of semantic objects, you would specify “*SO. You can also specify that metaclasses should be included in the list (by default they are not).
Creating an IDL Module Using the IDL Generator

**Note:** Metaclasses refers to class side methods.

2 Choose the corresponding client(s).

Click in the Clients list to choose the class(es) of client objects that will be able to make requests of the specified server's interface. The client classes you choose here help filter the list of methods included in Step 3.

Again, you can limit the list of classes by specifying a filter pattern. You can also specify that subclasses of the specified classes should appear in the list.

3 Choose method(s) that will respond to client requests.

In the Methods list area, click the Filter Methods button to show check marks for those methods that correspond to messages the selected client(s) can send.

Click on any additional method names you wish to mark for inclusion (indicated with a check mark). Or, click again to unmark a method name.

The most recently clicked (checked) method name will be the selected method. For the selected method, the corresponding IDL operation will appear in the arguments list (step 4) below.

4 Specify argument types for the selected method.

In the Arguments list area, the small box at the top of the argument list shows the return type for this operation (method). The larger box lists any arguments for this operation, and their types. By default, all types are shown as SmalltalkObject, which is a legal type but generally not specific enough to be useful.

Use the <operate> mouse button to get a pop-up list of IDL types (boolean, character, long, float, octet, any, string, sequence), or choose other to provide a Smalltalk class name or other type.

5 Specify a module name.

Note that if you use an existing module name, when you compile the module (in step 6, below), the new module will overwrite the existing module. However, you can change the module name you specify here before compiling, and thus avoid difficulties.
6 Click **Generate IDL** to generate the interface.

The IDL module will appear in the IR Browser. Review and edit the module, then choose **accept**.

7 Click **Generate Glue** to generate the corresponding Smalltalk repository methods.

This generates a repository protocol that includes the appropriate abstractClassId and CORBAName methods for each of the server classes in the module.

8 Click **Verify IDL** to perform a consistency check.

This checks consistency of types, operations, and interfaces between Smalltalk and IDL. Any errors detected are printed in the System Transcript (in the main Distributed Smalltalk main window).

**Refining the Module**

You can only use the IDL Generator to create modules that include interfaces and their operations. If you wish to define other IDL elements (types, attributes, constants, and exceptions), you must do so directly in the module itself.

Edit the interface using a system browser on the image containing the interface definitions. Browse DSTRepository and find the definition to edit.

**Interface Repository Browser**

The Repository Browser gives you a graphical and a textual view of the interface repository. Editing the repository is done using the normal system browsers.

**Opening the Browser**

Open the Repository Browser by selecting **Tools → Browse Repository** in the DST Tool.

From the Repository Browser, you can open further browsers on either semantic or presentation interfaces. In the Repository Browser, select **Tester → Open Interface**, and then select either **Semantic** or **Presentation**. The semantic is either local or remote, depending on whether the repository is local or remote, and the presentation is local.
IR Browser Icons

IR Browser icons, based on a metaphor of blueprints and construction equipment, indicate the component displayed. (An interface is represented as a blueprint, a module as a group of interfaces; parameter types show if a nail can be hammered in, pulled out, or both; an operation as a cement mixer; a result as a filled wheelbarrow.)
# IR Browser menus

## Action menu

<table>
<thead>
<tr>
<th>Menu Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print</td>
<td>Not yet implemented</td>
</tr>
<tr>
<td>Open Definer</td>
<td>Opens an IR Browser on the meta object in which this item is defined.</td>
</tr>
<tr>
<td>Open Repository</td>
<td>Opens an IR Browser on the root (DSTRepository).</td>
</tr>
<tr>
<td>Open Referent</td>
<td>Provides a list of meta objects that reference this item by name; double click in the list or select an item and click <strong>OK</strong> to open a new IR Browser window.</td>
</tr>
<tr>
<td>Close</td>
<td>Closes this window.</td>
</tr>
</tbody>
</table>

## Edit menu

<table>
<thead>
<tr>
<th>Menu Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>Opens a class browser where you can edit the definition.</td>
</tr>
</tbody>
</table>

## View menu

<table>
<thead>
<tr>
<th>Menu Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>as Text</td>
<td>Show this window of the IR Browser as read-only IDL text.</td>
</tr>
<tr>
<td>as Picture</td>
<td>Show this window of the IR Browser graphically (default).</td>
</tr>
</tbody>
</table>
Importing IDL files

Distributed Smalltalk provides a mechanism to import IDL files generated outside of Distributed Smalltalk. These external IDL files can have preprocessing directives. The important steps are listed below.

**Setup for Preprocessing**

Distributed Smalltalk *does not include* a preprocessor. You can use one of the following approaches:

- **Use an ANSI C++ preprocessor to preprocess the IDL file.**
  
  a. Preprocess the IDL using the ANSI C++ preprocessor that is available on your system.
  
  b. Import the IDL file into Distributed Smalltalk using the IDLCompiler importIDLFile:category: class method.
     
     The first argument is the name of the preprocessed IDL file. The second argument is the name under which the contents of the IDL file will appear in the IR browser.

- **Use the VisualWorks DLL and C Connect preprocessor (CPreprocessor)**
  
  You can use the C preprocessor that comes with DLL and C Connect, included with VisualWorks. Even though this preprocessor is not ANSI C++ compliant, it will work in most cases. Load the DLLCC parcel, then do the following:
  
  a. Modify the IDLCompiler preprocess: class method to work with CPreprocessor. As described in the method comment, change the method to

     ```
     ^CPreprocessor preprocess: aStream
     ```

     Other preprocess messages are defined in CPreprocessor that you may need to use instead. Put whatever statement meets your needs as the method body.
  
  b. Send an importIDLFile:category: to the IDLCompiler class.
     
     This message sends the (modified) preprocess: message to the input stream. The arguments are a FileName specifying the IDL file, and a symbol specifying the DSTRepository method category.
Annotate the IDL with Pragmas Where Necessary

Use a System Browser to add any necessary pragmas to the imported IDL. See Chapter 8, “Defining IDL Interfaces” for details on pragmas.

Avoiding Interface Problems

Keeping Interface Repositories Updated

Each Distributed Smalltalk image contains its own interface repository, unless you are using a shared repository. Changes you make to interfaces in one image do not propagate automatically to others. When interface repositories are not in sync with each other, you can get communication failures.

One way to facilitate interface repository maintenance is with a shared repository. To share a repository, open the DST Settings tool (File ➤ Settings) in the DST Tool. On the Repository page, select the Host radio button and enter the host IP address or name. Do this in all images that will be sharing this repository, except the one hosting the repository, which will be configured to Local.

Edit Lock

It is possible to get the IR Browser in a state where there is an edit lock, but no user is actually editing the repository. This can happen when an exception is raised while editing the repository and instead of proceeding from the error, terminate is chosen from the Error Handling dialog box; thus by passing the cleaning up of the lock.

The following method is provided to clean-up this situation:

DSTRepository class>>dropEditLock.

It should be used with extreme caution because if the repository is locked for legitimate reasons and another user is currently editing, the lock will be removed and concurrent changes may be made to the same module within the repository.
ORB initialization defines the way in which an application can initialize itself in a CORBA environment. There are three aspects to ORB initialization as specified by the OMG:

- **ORB initialization**
  
  Initialize an application into the ORB and Object Adapter environments. Return ORB and OA pseudo object references to the application for use in future ORB and OA operations.

- **Object Adaptor (OA) initialization**
  
  Obtain a reference to an object adaptor pseudo-object so that object implementations have access to the ORB functionality.

- **Obtaining initial object references**
  
  Obtain initial object references for an application.

Distributed Smalltalk does not implement the ORB and OA initialization interfaces. These interfaces are not necessary because of the combination of Smalltalk’s dynamic nature and Distributed Smalltalk’s architecture. These interfaces are intended for support of objects generated with static languages like C/C++.

### Programatically Initializing, Starting, and Stopping the ORB

The ORB can be initialized, started, and stopped using the DST Tool panel. In many cases, however, it is desirable to do so programmatically, either with a custom tool or without any tool at all.

To initialize an ORB, send an `initializeORBAtHost:nodeId:` message to the ORBObject class:
ORObject initializeORBAtHost: \textit{aHostName} nodeId: \textit{aHostAddress}

where \textit{aHostName} is the host name and \textit{aHostAddress} is the host IP address, as defined for IPSocketAddress. Given a host name only, you can get the address and initialize the ORB like this:

ORObject
initialzeORBAtHost: \textit{aHostName}
nodeId: (IPSocketAddress hostAddressByName: \textit{aHostName}).

To initialize the ORB on the local machine, simply evaluate:

hosts := SocketAccessor getHostname.
ORObject initializeORBAtHost: host
nodeId: (IPSocketAddress
    hostAddressByName: host).
ORBDaemon startUpCoordinator startRequestBroker

Once initialized, you can start the ORB.

Starting and stopping the ORB is managed by an instance of ORBStartUpCoordinator that is held by ORBDaemon. To start or stop the ORB, send one of these messages to the coordinator:

\texttt{startRequestBroker}
Starts the request broker.

\texttt{shutDown: minutes}
Shuts down the ORB in \textit{minutes}, an integer value. To shutdown immediately, specify 0.

You get the coordinator from the ORBDaemon by sending the startUpCoordinator message, so you can start or stop the ORB as follows:

ORBDaemon startUpCoordinator startRequestBroker
or

ORBDaemon startUpCoordinator shutDown: 3
Getting Remote ORB References

Message sends to class OrbResolver are used to generate references to remote ORBs. Several methods are provided.

Given the ORB hostname and port number, use:

```
OrbResolver
    generateOrbProxy: hostname
    transport: ORBDaemon configurationManager
    configurationOf: #IIOP
    port: portNumber
```

If you already have an ObjRef to an object on the remote ORB, you can get a reference to the ORB using:

```
OrbResolver generateOrbProxyOnReference: aCoLocatedReference
```

Browse OrbResolver for other options, and search the system for senders for examples.

Initial Object References

The initial object reference service logically is a simplified local version of the naming service which an application can use to obtain a small defined set of object references which are essential to the application’s operation.

Because only a small well defined set of objects are expected to be available via the initial object reference mechanism the naming context is flattened to a single level namespace. Only two operations are defined for the initial object reference mechanism. The pseudo IDL for these operations is:

```
module CORBA {

    interface ORB {
        typedef string ObjectId;
        typedef sequence <ObjectId> ObjectIdList;
        exception InvalidName {};
        ObjectIdList list_initial_services ();
        Object resolve_initial_references (in ObjectId identifier) raises (InvalidName) ;
    };

};
```
In order to allow an application to determine which objects have references available via the initial references mechanism the list_initial_services operation is provided. It returns a sequence of strings. Each string represents an object which is available through this mechanism.

The resolve_initial_references operation returns the object reference associated with its argument. Arguments can be any of the strings returned by the list_initial_services operation.

Distributed Smalltalk Implementation

The list_initial_services operation is implemented by:

```
ORBObject class>>listInitialServices
```

When invoked it will return an IdentitySet of symbols representing objects which have references available via the initial references mechanism.

```
ORBObject class>>resolveInitialReferences:
```

The argument to this method is one of the symbols returned by listInitialServices. This method returns the object reference associated with the argument.

Currently there are four objects whose references are available via the initial references mechanism:

- InterfaceRepository
- NameService
- FactoryFinder
- UserSecurityDatabase

ORB Utility Methods

There are a variety of utility methods available in ORBObject for retrieving information about the ORB or various

hostName

You often need to get the local host name, for example, in the process of initializing an ORB. This is the easiest way to do it.
**namingService**
This is the quickest way to get the naming service. It usually will be either a DSTNameContext or a DSTObjRefRemote, depending on whether the DST image has been configured to use a local or a remote naming service.

**factoryFinder**
This is the quickest way to get the factory finder, which is a directory used by the Lifecycle Service.

**repository**
This is the quickest way to get the interface repository. You will get either a privileged instance of DSTmoduleRepository or a reference to a one, depending on whether the image is configured to use a local or a remote repository.

**referenceToFile: aString object: anObject**
This message returns what is known as a “stringified object reference.” A reference, in the form of a hexadecimal string, to anObject is written to the file name aString. Such files are a usual way of providing clients, at startup, with initial references to an remote object.

**referenceFromFile: aString**
This method produces an object reference from a stringified object reference contained in the file aString.

**explainIOR:**
Provides a detailed description of an IOR. This is useful in debugging when it is found that the object reference obtained from a stringified object reference cannot be resolved.

Browse the class side of ORBObj for additional utility methods.
The Naming Service is part of the Common Object Services specification published by OMG. It specifies what CORBA object names should consist of and how names can be set and accessed, so that local and remote objects in any CORBA implementation can be correctly identified. Clients can use the naming service to locate and identify objects in both local and remote systems.

Distributed Smalltalk implements this service by providing a simple mapping between the specified name structure, and a ation of this includes support both for standard naming policies. You use the naming service directly to obtain initial access to an object programmatically (see “Initial Object References”).

Note that the ORB does not use the naming service, but identifies objects by the unique identifiers or the objects themselves (if local) or of their object references (if remote). The Naming Service is a service provided for developers, so that they can make certain privileged objects publicly available.
What Constitutes a Name?

A name is an ordered sequence of components that is bound to a specific object. A name binding is a name-to-object association.

**Name Components**

*Name components* are the parts that make up a name. A name is comprised of one or more components. A component is the name of either the bound object or a naming context. There can be more than one naming context in a compound name.

A *simple name* is a name with a single component (an object). Simple names are guaranteed unique only within a context.

A *compound name* is made up of an ordered sequence of contexts and the name of the bound object. In a compound name, each component except the last is used to name a context; the last component names an object. Compound names are guaranteed to be unique system-wide.

**Name Contexts and Naming Graphs**

A *naming context* is an object that contains a set of name bindings in which each name is unique. A name binding is always defined relative to a naming context. Different names can be bound to an object in the same or different contexts at the same time. Since a name context is itself an object, it may be bound to a name in another name context.

A *naming graph* is a directed graph with contexts (nodes) and labeled edges. A naming graph is created by binding contexts in other contexts. A naming graph is similar to a file system's directory structure, with contexts that are similar to directories, and objects similar to file names.
Naming Service Operations

The classes that implement the naming service are in the class category COS-Naming. You can browse those classes and the following examples to learn more about naming policies and services.

Creating Names

To create an instance of a name, class DSTName implements these class methods:

- To create a new name given the name components, use on:
  For example:

  \[
  \text{DSTName on: (Array with: (DSTNameComponent id: 'simple' kind: 'text') ).}
  \]

  or:

  \[
  \text{DSTName on: (Array with: (DSTNameComponent id: 'example' kind: 'dir') with: (DSTNameComponent id: 'myImage' kind: 'im') ).}
  \]

- To return a new name on a given name string, use onString: or asDSTName. For example:

  \[
  \text{DSTName onString: 'simple' or 'simple' asDSTName}
  \]
Naming Service Operations

- To return a new compound name, use onStrings: or the shortcut method asDSTName: For example:
  
  DSTName onStrings: #(‘component1’ ‘component2’ ‘component3’)
  
  or

  #(‘component1’ ‘component2’ ‘component3’) asDSTName.

Binding and Unbinding

The naming policy establishes how to bind each object to a unique name within a given context. A name binding is a name-to-object association. Only one object can be bound to a particular name in a context.

Class DSTNameContext implements methods for the following binding operations:

- You can bind both objects and other contexts to contexts (bindNewContext:, contextBind:to:, contextBindContext:to:).
- If a context does not exist, you can create it, or create it and bind an object to it (newContext, bindNewContext:).
- If an object or context is already bound, it can be rebound (contextReBind:to:, contextReBindContext:to:).
- If rebinding is not strong enough, you can unbind or destroy a context (contextUnBind, destroyContext).

For example:

```
| cxt |
cxt := DSTNameContext new.
cxt contextBind: (‘foo’ asDSTName) to: 7.
cxt contextReBind: (‘foo’ asDSTName) to: 8.
cxt contextBindContext:(DSTName onString:‘aContext’) to: DSTNameContext new.
cxt contextBind: (DSTName onStrings: #(‘aContext’ ‘fee’)) to: #fum.
cxt contextUnBind: (DSTName onStrings: #(‘aContext’ ‘fee’)).
```

Resolving and Listing Contexts

A name can be resolved to determine which object it represents. A name resolution uses a name to identify an object. Because names can have multiple components, name resolution can traverse multiple contexts.

Class DSTNameContext implements these methods resolving names and listing contexts:
To resolve a name, use `contextResolve:`.

To return a given number of bindings contained in the specified naming context, you can use `listContext:`. (Use with `DSTBindingIterator` to iterate through the list.)

For example:

```smalltalk
| cxt |
cxt := DSTNameContext new.
cxt contextBind: (DSTName onString: 'foo') to: 7.
cxt contextResolve: (DSTName onString: 'foo').
cxt bindNewContext: (DSTName on: (Array with: (DSTNameComponent id: 'foo' kind: 'cxt'))).
cxt newContext.
cxt destroyContext.
```

**Syntax-Independent Kinds and Identifiers**

To avoid issues of differing name syntax, the naming service always deals with names in their structural form, which consists of two attributes: the `identifier` attribute and the `kind` attribute. Both the identifier attribute and the kind attribute are represented as IDL strings.

The kind attribute adds descriptive information to names independent of their syntax. For example, suffixes such as (for C language in Unix) ".c" or ".o" would be replaced with "c_source" or "object_code". Applications like the C compiler depend on these syntactic conventions to make name transformations such as from `foo.c` to `foo.o`. Such syntactic convention is not explicit; software that does not depend on the syntactic conventions for names does not have to be changed to adapt to new conventions.

An empty string indicates no kind. The naming service does not interpret, assign or manage these values in any way. Higher levels of software may make policies about the use and management of these values.

**Exceptions**

Class `DSTNameContext` implements these exceptions for the naming service:

`notFoundError:restOfName:`

> The name does not identify a binding.

`cannotProceedError:nameComponent:`

> The implementation has given up for some reason. (For example, when there is a network problem during a resolve operation that involves several systems.) The client, however, may be able to continue operation at the returned naming context.
invalidNameError
   The name is invalid.

alreadyBoundError
   A name binding using this name already exists.

notEmptyError
   The context cannot be destroyed because it is not empty.

Interfaces

The CosNaming module in DSTRepository defines the NamingContext and BindingIterator interfaces for object naming. Browse these interfaces for details.

Implementation

Four classes interact to provide the primary support for the naming service. Browse these classes for variables and methods. Their positions in the class hierarchy are:

Object
   DSTNameComponent
      Collection
         SequenceableCollection
            OrderedCollection
         DSTName
      Model
         ORBObject
            DSTPersistentObject
               DSTNameContext
                  DSTfactoryFinder
                     SessionContext
                        DSTDesktopSessionContext
       Stream
          PeekableStream
             PositionableStream
                InternalStream
                   ReadStream
                      DSTBindingIterator
Event Notification

The event notification service enables objects to notify one another of interesting occurrences using an agreed protocol and set of objects. As designed, it provides optimal communication between objects in a distributed computing environment.

Overview

The event notification service supports decoupled, asynchronous communication between objects. Objects that generate events (event suppliers) place the event information in an event channel. From the event channel, event information is either pushed to event consumers (objects that wish to receive the event information), or pulled by the event consumers from the event channel at the consumer’s convenience.

There can be more than one event channel. Each event channel can have one or more event suppliers and one or more event consumers.

Need for Event Notification in a Distributed System

In a distributed object system, objects that interact may “live” in various images and machines, both local and remote. By decoupling communication, the event channel provides support for object interaction when objects are unavailable temporarily because the network or a remote system is “down.”

The CORBA2.0 architecture specifies a synchronous notification mechanism (RPC) between a single client and single server, both of which must be available when the service request is made. The need for asynchronous communication prompted OMG to include the event service in the Common Object Services Specification.
Event Channel

Terminology

<table>
<thead>
<tr>
<th>term</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>An occurrence within an object that may be of interest to other objects. For example, when a model object changes, it generates an event to inform its view-controller pairs in both local and distributed images.</td>
</tr>
<tr>
<td>event consumer</td>
<td>An object that receives and processes event data.</td>
</tr>
<tr>
<td>event supplier</td>
<td>An object that produces event data for distribution to consumers.</td>
</tr>
<tr>
<td>event channel</td>
<td>An object that holds event information until consumers are ready to receive it.</td>
</tr>
</tbody>
</table>

Event Channel

An event channel stores event information, thus allowing objects to communicate asynchronously. Although consumers and suppliers communicate with the event channel using standard CORBA requests, the event channel need not supply the event data to its consumer at the same time it consumes the data from its supplier.

- An event channel can have any number of consumers and suppliers.
- Consumers and suppliers must register with the event channel in order to participate in it.
- The event channel is a consumer of event information from its suppliers, and a supplier of event information to its consumers.
Multiple Event Channels

Usually, each group of related objects and activities should have its own event channel. Channels can be used to handle multiple sources and sinks irrespective of their function. Typically you would organize it around specifically related objects and activities.

However, the same channel can be allowed to handle both data changed and data deleted event information.

Event Channel Administration

An event channel is built up incrementally. When an event channel is created, there are no suppliers and no consumers associated with it. Upon creation of the channel, the factory returns an object reference supporting the EventChannel interface. The ConsumerAdmin and SupplierAdmin interfaces allow consumers and suppliers to be added to the event channel. (That is, the ConsumerAdmin and SupplierAdmin interfaces provide client access to the services implemented in class DSTEventChannel’s message categories Consumer Admin and Supplier Admin.)

Push and Pull Models

The event notification service supports two models of communication: push (the default) and pull. When adding a consumer or supplier to an event channel, you specify it as either pull or push type.

Push

The push model is similar to an interrupt: when an event occurs, the supplier initiates the notification of the event to the consumer immediately.

For example, the push model would be the best choice for a disk (an event supplier) that needs to notify a system administration tool (an event consumer) immediately if the disk runs out of space. When the disk is full, it pushes event information to announce the space problem; any connected consumer, in this case the system administration tool, is notified immediately of the event.
Push and Pull Models

Pull

The pull model is similar to polling: a consumer initiates a request for event data when it is convenient for the consumer to do so, regardless of when (or if) the event occurs.

For example, the pull model would be a good choice for an event notification relationship between a document (event consumer) that needs to know about changes to an embedded table (event supplier). If the document is closed when the table changes, notification of the change can wait until the document is next opened or accessed. If a user is editing the document, the editing process should probably not be interrupted to update the table; this can also wait until there is a pause in editing. That is, if you set up the document as a pull consumer, it can request update information at its convenience (for example, at start up, or when open but idle).

Disconnect to Terminate Communications

The disconnect operations allow either a consumer, supplier, or channel to terminate communications by severing its ties with the supplier or consumer. These operations are useful when an object should not be interrupted with event information. The disconnect operations are:

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSTPullConsumer</td>
<td>disconnectPullConsumer</td>
</tr>
<tr>
<td>DSTPullSupplier</td>
<td>disconnectPullSupplier</td>
</tr>
<tr>
<td>DSTPushConsumer</td>
<td>disconnectPushConsumer</td>
</tr>
<tr>
<td>DSTPushSupplier</td>
<td>disconnectPushSupplier</td>
</tr>
</tbody>
</table>
Consumers and Suppliers

Any of the event channel’s relationships with either suppliers or consumers can use either the push or pull model.

Any supplier or consumer can be disconnected from the event channel (in order to avoid inconvenient event notification).

Proxies

Proxy consumers and suppliers (classes DSTProxyConsumer and DSTProxySupplier) are system-level classes that are used to establish communication with an event channel. Application programmers should never work directly with proxy objects. As an example of push and pull, consider the following diagram:
Event Data

The push and pull operations of the consumer and supplier interfaces communicate event data as type any. This allows generic services, such as an event channel, to consume, store, and supply event data without understanding the type of the event data.

Using type any for event data does not mean that the data is untyped. Suppliers and consumers of event data need to agree on the type of the event data. Consumers of the event data need to interpret the data according to the agreed upon type.

However, using type any has a higher overhead and does not provide type checking; you can use explicitly typed events if you want to lower messaging overhead and check types (see “Using Typed Events”).

Using Events

If you wish to consume events from an EventChannel, you need to implement an class which will handle messages from the consumer (DSTPushConsumer or DSTPullConsumer). An instance of this class is held by the consumer in an attribute named host. Upon receiving a disconnect message, for instance, the consumer notify the host by sending the message:

    host removeConsumer: self.

A host to a DSTPushConsumer must also implement a method which will get called when an event is received by the consumer. This method is designated upon setup as the aspect. It should be noted that the Symbol that is passed to the consumer is converted to a setter method.

You must also implement two messages for supplier hosts that wish to participate as supplier to event channels. Upon receiving a disconnect message, the supplier will notify the host by sending the message:

    host removeSupplier: self.

A host to a DSTPullConsumer will additionally need to implement the method supplierNeedsEvent. This message gets sent when the pullConsumer has requested an event and no events are left in the queue for the supplier.

This gives the host a chance to process any outstanding events before the pull supplier responds to the pull consumer.
Example Code for Events

The examples below require that an EventChannel be created, for example:

```small
anEventChannel := DSTEventChannel new asRemotable
```

and that the appropriate hosts are constructed as described in “Using Events” above.

**Example: Connecting a Push Consumer to a Channel**

1. Create a new push consumer. For example:
   ```small
c consumer := DSTPushConsumer new.
   ```

2. Get a proxy supplier. For example:
   ```small
   proxySupplier := anEventChannel forConsumers obtainPushSupplier.
   ```

3. Connect the consumer to the supplier. For example:
   ```small
   proxySupplier connectPushConsumer: consumer.
   ```

4. Initialize the consumer. For example:
   ```small
   consumer host: aHost supplier: proxySupplier aspect: #displayEvent
   ```

**Note:** The host is expected to have implemented the `displayEvent:` method.

**Example: Connecting a Pull Consumer to an Event Channel**

1. Create a new pull consumer. For example:
   ```small
   consumer := DSTPullConsumer new.
   ```

2. Get a proxy supplier. For example:
   ```small
   proxySupplier := anEventChannel forConsumers obtainPullSupplier.
   ```

3. Connect the consumer to the supplier. For example:
   ```small
   proxySupplier connectPullConsumer: consumer.
   ```

4. Initialize the consumer. For example:
   ```small
   consumer host: aHost supplier: proxySupplier.
   ```
Using Typed Events

Example: Connecting a Push Supplier to a Channel

1. Create a new push supplier. For example:
   ```smalltalk
   supplier := DSTPushSupplier new.
   ```

2. Get a proxy consumer. For example:
   ```smalltalk
   proxyConsumer := anEventChannel forSuppliers
                  obtainPushConsumer.
   ```

3. Connect the supplier to the consumer. For example:
   ```smalltalk
   proxyConsumer connectPushSupplier: supplier.
   ```

4. Initialize the supplier. For example:
   ```smalltalk
   supplier host: s1_supplier consumer: proxyConsumer.
   ```

Example: Testing the Event Example

```smalltalk
supplier processEvent: 'hello'.
supplier processEvent: 'hello again'.
Transcript cr; show: 'Pull Consumer tryPull event ➔ ', consumer
                  pullEventData ;cr.
Transcript cr; show: 'Pull Consumer pull event ➔ ', consumer
                  pullEventData ;cr.
```

Using Typed Events

The typed event service extends basic event notification, allowing you to determine:

- Specific event types

  By default, event channels pass data of type. However, it can be more efficient if you type an event channel to pass only strings, or tables, or a specific type of structure that you have defined, as appropriate.

- Quality of service

  By default, a push supplier sends event notification to a consumer when the event channel receives the event data. If the consumer is unavailable for the original notification, the event data is held in the channel until another event occurs, at which time notification of both events is sent together. The event channel will continue to attempt to push the event data each time a new event occurs, until it is successful or the channel is destroyed.
Using the typed event service, you can specify the quality of service for each channel or consumer-supplier pair. That is, you can set the channel to retry at certain interval over a given time period, or you can set it to retry once only, or not to retry at all.

**Example: Connecting to a Channel**

To connect to a channel, at a minimum, you must:

1. Create a new consumer. For example:
   ```plaintext
   consumer := DSTPushConsumer new.
   ```
2. Get a supplier. For example:
   ```plaintext
   supplier := anEventChannel forConsumers obtainPushSupplier.
   ```
3. Connect the consumer to the supplier. For example:
   ```plaintext
   supplier connectPushConsumer: consumer.
   ```
4. Initialize the consumer. For example:
   ```plaintext
   consumer
   'host: self
   supplier: supplier
   aspect: #linkAdded
   ```

**Example: Implementing a Typed Push Connection**

If you wish to allow a supplier-to-consumer push connection that is typed, you should implement both a typed push supplier and a typed push consumer, as well as their corresponding interfaces.

**Typed Push Supplier and Interface**

If you want to create a push supplier that only generates event data of type String or type Point (a structured type with two elements: x and y), you can do this:

1. Subclass from DSTTypedPushSupplier.
2. Create the class repository methods abstractClassId and CORBAName.
3. Create the private method `pushToConsumer: anEvent`, which might be defined like this:

```smalltalk
pushToConsumer: anEvent
    'push the event to the consumer'
    (anEvent isKindOf: String)
        ifTrue: ["consumer pushString: anEvent].
    (anEvent isKindOf: Point)
        ifTrue: ["consumer pushPoint: anEvent].
    consumer pushEventData: anEvent
```

4. Create an IDL interface which might look something like this:

```idl
// This interfacedefines thebehavior of my new typed push supplier.
interfaceTypedPushSupplierExample.CosEventComm::PushSupplier{};
```

Notice that the interface can be simple (no operations) but it must be listed in the interface repository so that a typed event channel can create one using the CORBAName as the key. Also, it must inherit from an interface (such as `PushSupplier`) that defines or inherits the appropriate behavior.

5. Set up a typed supplier (subtly different from setting up a non-typed supplier):

```smalltalk
"create a new event channel"
  aTypedEventChannel := DSTTypedEventChannel new.

"create a new supplier"
  supplier := MyTypedPushSupplier new.

"hook the supplier to the event channel"
  supplierAdmin := aTypedEventChannel forSuppliers.
  proxyConsumer connectPushSupplier: supplier.

"initialize the supplier"
  supplier host: self
  consumer: proxyConsumer getTypedConsumer.
```

**Corresponding Typed Push Consumer and Interface**

To implement a consumer that interacts with the supplier you just created:

1. Subclass from `DSTTypedPushConsumer`.
2. Create the class repository methods `abstractClassId` and `CORBAName`. 
4. Create an IDL interface which might look something like this:

```idl
interface TypedPushConsumerExample :
  CosTypedEventComm::TypedPushConsumer {
    void pushString (in string str)
      raises (Disconnected);
    void pushPoint (in Point pt)
      raises (Disconnected);
  }
```

5. Set up a typed consumer:

```small
"create a new consumer"
consumer := MyTypedPushConsumer new.

"hook the consumer to the event channel"
consumerAdmin := aTypedEventChannel forConsumers.
proxySupplier := consumerAdmin obtainTypedPushSupplier:
  MyTypedPushSupplier new CORBAName.
proxySupplier connectPushConsumer: consumer.

"initialize the consumer"
consumer host: self
  supplier: proxySupplier
  aspect: #handleMyEvent.
```

### Example: Implementing a Typed Pull Connection

This example shows how you might implement typed pull supplier and consumer objects and their interfaces. Like the previous example, they will pass objects of types String and Point.
Typed Pull Supplier and Interface
1 Subclass from DSTTypedPullSupplier.
2 Give the class repository methods abstractClassId and CORBAName.
3 Give it the private methods pullString and tryPullString, which might be defined like this:
   pullString
   "return the next event that is a kind of String"
   ^self pullTypedEvent: String
   tryPullString
   "return the next event that is a kind of String, returning no data if necessary"
   ^self tryPullTypedEvent: String
4 Create an IDL interface, which might look something like this:
   // This interface defines the abstract behavior my typed pull supplier example
   //
   interface TypedPullSupplierExample :
   CosTypedEventComm::TypedPullSupplier {
   string pullString ()
   raises (Disconnected);
   string tryPullString (out boolean has_event)
   raises (Disconnected);
   Point pullPoint ()
   raises (Disconnected);
   Point tryPullPoint (out boolean has_event)
   raises (Disconnected);
};

Corresponding Typed Pull Consumer and Interface
1 Subclass from DSTPullConsumer.
2 Write the class repository methods abstractClassId and CORBAName.
3 Write the host messages methods pullTypedEvent: and tryPullTypedEvent:, which might be defined like this:
Chapter 13 - Event Notification

pullTypedEvent: aClass

"try to pull event data from the supplier"

connected
  ifTrue:
    [(aClass isKindOf: String)
      ifTrue: [^supplier pullString].
    (aClass isKindOf: Point)
      ifTrue: [^supplier pullPoint].
    ^supplier pull]
  ifFalse: [^self disconnectedError]

tryPullTypedEvent: aClass

"try to pull event data from the supplier"

connected
  ifTrue:
    [(aClass isKindOf: String)
      ifTrue: [^supplier tryPullString].
    (aClass isKindOf: Point)
      ifTrue: [^supplier tryPullPoint].
    ^supplier tryPull]
  ifFalse: [^self disconnectedError]

4 Create an IDL interface which might look something like this:

  interface TypedPullConsumerExample :
    CosEventComm::PullConsumer ()

Example: Determining Quality of Service

If you want to change the quality of service from the default (retry at every new event until successful), you can override the processEvent: method in a subclass of DSTPushSupplier.

For example, you might write:
processEvent: anAny

'process an event by sending it to the consumer. Discard the event if an error occurs'

| evt |
(self checkEvent: anAny)
  ifFalse: [^nil].
  events nextPut: anAny.
  connected ifTrue: [(events isEmpty]
    whileFalse:
      [evt := events next.
       self errorSignal handle: [:err | err signal == ORBObject
         invObjrefSignal
        ifTrue:
          [(host asLocal isKindOf: DSTEventChannel)
           ifTrue: [host removeSupplier: self].
           self connected: false]
        ifFalse: [Dialog onDebugNotify:
          'EventSupplier>>pushEvent
          error: ', err errorString]]
       do: [self pushToConsumer: evt]])

Interfaces

Four modules in DSTRepository define interfaces for the event notification service: CosEventComm, CosTypedEventComm, CosEventChannelAdmin and CosTypedEventChannelAdmin. Browse these interfaces for details.

The IDL hierarchy is as follows:
Chapter 13 - Event Notification

PushConsumer
  TypedPushConsumer
  ProxyPushConsumer
    TypedProxyPushConsumer : ProxyPushConsumer, TypedPushConsumer
    DSTProxyConsumer : TypedProxyPushConsumer, ProxyPullConsumer

PushSupplier
  TypedPushSupplier
  ProxyPushSupplier
    DSTProxySupplier : ProxyPushSupplier, TypedProxyPullSupplier

PullConsumer
  ProxyPullConsumer
    DSTProxyConsumer : ProxyPullConsumer, TypedProxyPushConsumer

PullSupplier
  TypedPullSupplier
  ProxyPullSupplier
    TypedProxyPullSupplier : ProxyPullSupplier, TypedPullSupplier

ConsumerAdmin
  TypedConsumerAdmin

SupplierAdmin
  TypedSupplierAdmin

EventChannel
  TypedEventChannel
    DSTEventChannel : TypedEventChannel, TypedSupplierAdmin, TypedConsumerAdmin
Implementation

The following classes interact to support the event notification service. Browse these classes for variables and methods. Their positions in the class hierarchy are:

Object
  Model
  ORBObject
    DSTProxyConsumer
    DSTProxySupplier
    DSTPersistentObject
    DSTEventChannel
      DSTTypedEventChannel
      DSTPullConsumer
        TypedPullConsumerExample
      DSTPullSupplier
        DSTTypedPullSupplier
        TypedPullSupplierExample
      DSTPushConsumer
        DSTTypedPushConsumer
        TypedPushConsumerExample
      DSTPushSupplier
        TypedPushSupplierExample

Note: Application developers should not use classes DSTProxyConsumer and DSTProxySupplier directly. No variables and messages appear here.
Basic Lifecycle

Overview

OMG’s Common Object Services specification defines basic lifecycle services for creating, deleting, copying and moving objects both locally and remotely. While standard Smalltalk handles most basic lifecycle implementation within a local image, Distributed Smalltalk adds lifecycle interfaces to support distribution and interoperability.

Lifecycle services are an extension to the core services provided in an ORB. Core services include activation, request delivery, principal authentication, and method dispatch, as well as the creation and destruction of object registration information. Any facilities required for object population control and migration are part of the lifecycle services.
Lifecycle Operations

Terminology

<table>
<thead>
<tr>
<th>term</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic lifecycle</td>
<td>Services that govern simple objects, such as create, delete, and move.</td>
</tr>
<tr>
<td>factory object</td>
<td>An object that creates objects in response to client service requests. In Distributed Smalltalk, a factory is any class that can be instantiated and has interfaces registered for creating objects in the Interface Repository. Any object that creates another object in response to some request is technically a factory; factory implementations are not special. The polymorphic createObject: method is available in ORBObject and DSTPresenter for Distributed Smalltalk. Factory objects are registered during the Initialize Factories phase of the ORB initialization. For a class to be registered as a factory, it must have an instance method abstractClassID (which returns the appropriate UUID value for the class as a symbol).</td>
</tr>
<tr>
<td>factory finder</td>
<td>An object at a specific location that helps clients obtain references to factories of a particular class. A factory registered with a factory finder represents an implementation at an abstract location; thus, factory finders permit clients to query a location for an implementation.</td>
</tr>
</tbody>
</table>

Lifecycle Operations

Create

An external client uses the local factory finder object to find the correct abstract class, which then creates the new object. For an example of using a factory finder to create an object, see “Examples: With and Without the Factory Representative”.

Resource allocation during object creation includes one or more Object Adapter object references, and persistent storage for the object’s persistent state.

Copy and Deep Copy

Copy makes a copy of the initial object, its children (containees and linked objects), their children and so forth. All non-containment links in the copy continue to refer to the same objects as the corresponding links in the original.

Deep copy is same as copy except that objects linked by non-containment links may be copied along with their children. A link’s deepCopyWith setting controls which referenced objects are copied, and
which continue to be shared between the original and the copied objects. You set linked objects to be shared or copied when you create the links. By default, they are shared.

**Move**

Moving an object means removing it from one container and putting it into another. In essence, a copy of the object is made at the destination, the original is removed, and the copy is renamed such that existing references to the original continue to refer to the copy.

**Note:** If the new container is at a different location, a move also relocates the object and any children.

**Destroy**

Destroy deletes an object and removes it from the system's registry. Nominally this also deletes the object's descendants, if any. However, if any of the components are referenced by reference links from outside the compound object, the delete does not occur, or the protected objects are moved to the orphanage so the delete can continue.

**Throw Away**

The throw away operation prepares an object for deletion by moving the object to a wastebasket (or similar service). Since this is potentially the last manual step before a subsequent automatic deletion, it prechecks for any constraints that would prevent the deletion.

**Externalize and Internalize**

Moving or copying objects between locations requires the objects to be externalized and subsequently internalized at the new location. In Distributed Smalltalk, the *marshall* operation converts a Smalltalk object into a byte stream for externalization (transmission to a remote server). To internalize, the *unmarshall* operation creates a Smalltalk object from a marshalled byte stream.
Creating Objects

COS on Factories and Factory Finders

COS specifies factories as the objects that create objects in response to a client request. It further specifies factory finders as the objects that help locate factories at a given location.

- **Factory object**—An object that creates an object(s) in response to a client service request. In Distributed Smalltalk, a factory is any class that can be instantiated and has interfaces registered for creating objects in the Interface Repository.

  When a client wishes to request a service of an object, the server object must first be created. Thus the client makes a request to a factory to create the server object.

- **Factory finder**—An object at a specific location that helps clients obtain references to factories of a particular class. A factory registered with a factory finder represents an implementation at an abstract location; thus, factory finders permit clients to query a location for an implementation.

  In order for a client to create a remote object, the client must have an object reference to the factory where the remote object will be created. Clients can use the naming service to get such object references.

Distributed Smalltalk’s Implementation

Any classes that inherit from ORBObject can be instantiated in response to a remote client’s request. Factory objects are registered during the Initialize Factories phase of the ORB initialization. For a class to be registered as a factory, it must have an instance method abstractClassId (which returns the appropriate UUID value for the class).

The interface FactoryRepresentative is an Distributed Smalltalk extension to the COS specification that improves performance in remote object creation. Most interfaces in the Interface Repository inherit from FactoryRepresentative, which implements the IDL create_object operation (mapped to the createObject: Smalltalk method). The create_object operation can be used as a short cut to create an instance of a particular class that is registered with the receiver. (If the create_object operation fails, you can use a more explicit interaction with the factory finder instead.) Examples of this follow next.
The interfaces that inherit from the FactoryRepresentative interface can be diagramed as follows:

![Diagram of FactoryRepresentative interfaces]

---

**Examples: With and Without the Factory Representative**

The following examples show how to create an instance of ShapeS0 on a remote machine called *worldly*.

The purpose of these examples is to show how to minimize the number of RPCs needed to create a remote object. However, in order to create a remote object, one must have an initial object reference to the remote image. Of the many ways to obtain this initial reference. Two examples are show below.

**Example 1: Stringified Object Reference**

This example uses the CORBA standard `objectToString` method to produce a “stringified” object reference which can be written to a file. This file is made available to client systems which uses the CORBA standard `stringToObject` method to read in the “stringified” object reference.

The following example code uses the DST convenience methods:

```plaintext
ORBOBJECT class>>referenceToFile:
This stringifies the object reference and stores it to a file.

ORBOBJECT class>>referenceFromFile:
This unstringifies the object reference and extracts it from a file.
```

On *worldly* execute:
ORBObject
  referenceToFile: 'myffinder'
  object: (ORBObject resolveInitialReferences: #FactoryFinder)

which creates a “stringified” reference to *worldly*'s factory finder and stores it in the file *myffinder*.

The client executes:

```smalltalk
| ff |
ff := ORBObject referenceFromFile: 'myffinder'
```

which returns an object reference to *worldly*'s factory finder.

**Example 2: Naming Service as Registry**
This approach uses the naming service to help locate an initial object reference to *worldly*'s factory finder.

On *worldly* execute:

```smalltalk
| ns |
ns := ORBObject resolveInitialReferences: #NameService.
ns contextBind: ('factoryFinder' asDSTName) to: (ORBObject
  resolveInitialReferences: #FactoryFinder)
```

This associates *worldly*'s factory finder with the name ‘factoryFinder’ and binds this association in the top level context of *worldly*'s naming service.

The client then configures its naming service to use *worldly*'s naming service. An object reference to *worldly*'s factoryFinder is then obtained by:

```smalltalk
| ns ff |
ns := ORBObject resolveInitialReferences: #NameService.
ff := ns contextResolve: (DSTName onString: 'factoryFinder')
```

Since the client is configured to use *worldly*'s naming service the first line returns a reference to *worldly*'s naming service. The second line extracts object reference from the top level context of the naming service.

**Note:** All of the examples in this chapter will use the approach of example 1 to obtain the factory finder object reference.
Using FactoryFinder Directly
By using the FactoryFinder directly, it requires 3 RPCs to get the object reference.

1. \textit{factory finder?} \hspace{1cm} \texttt{ffinder := ORBObject referenceFromFile: 'myffinder'}
   \textbf{ObjRef for factory finder}
   Locate the factory finder for the remote location where you will create the object.

2. \textit{factory?} \hspace{1cm} \texttt{shapeFactory := ffinder contextResolve: 'ShapeSO' asDSTName.}
   \textbf{ObjRef for factory}
   Query the factory finder for a reference to a factory to create the object.

3. \textit{newObject} \hspace{1cm} \texttt{shapeFactory createObjectKey: 'ShapeSO' asDSTName criteria: #().}
   \textbf{ObjRef for new object}
   Create the remote object that will return an object reference.

Using the Factory Representative—Option #1
By using the FactoryRepresentative interface, you can optimize communications (saving 1 of the original 3 RPCs).

1. \textit{factory finder?} \hspace{1cm} \texttt{ffinder := ORBObject referenceFromFile: 'myffinder'}
   \textbf{ObjRef for factory finder}
   Locate the factory finder for the remote location where you will create the object.
Creating Objects

Using the Factory Representative—Option #2
If you already have a reference to a factory representative, it takes one RPC.

<table>
<thead>
<tr>
<th></th>
<th>createObject</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ObjRef for new object</td>
</tr>
<tr>
<td></td>
<td>shapeObject := ffinder createObject: ShapeSO getInstanceACL.</td>
</tr>
</tbody>
</table>

Create the object through a request to the factory finder.

Note: Since most interfaces in Distributed Smalltalk inherit from FactoryRepresentative, this shortcut can be used with almost any object reference. In other words, most objects can be used to create other objects at a given location.

Example: Copying an Object
The following example shows how to locate a factory finder on a machine called worldly, create an instance of ShapeSO there, manipulate it, then copy it.

<table>
<thead>
<tr>
<th></th>
<th>createObject</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ObjRef for new object</td>
</tr>
<tr>
<td></td>
<td>shapeObject := aRemoteObjRef createObject: ShapeSO getInstanceACL.</td>
</tr>
<tr>
<td>This creates shape on the same host as that of aRemoteObjRef (a previously created remote object).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>createObject</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ObjRef for new object</td>
</tr>
</tbody>
</table>
| | shapeObject := worldlyShape.
| worldlyShape := ffinder createObject: ShapeSO getInstanceACL.
| worldlyShape setShape: #triangle by: nil.
| worldlyShape inspect.
| localShape := worldlyShape copyFactoryFinder: ORBObject factoryFinder criteria: #()
| localShape inspect.

<table>
<thead>
<tr>
<th></th>
<th>createObject</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ObjRef for new object</td>
</tr>
<tr>
<td></td>
<td>shapeObject := ffinder createObject: ShapeSO getInstanceACL.</td>
</tr>
</tbody>
</table>

Create the object through a request to the factory finder.
Commentary
1 Get worldly’s factory finder.
2 Tell the factory finder to create an object.
3 Change the shape of the remote shape object.
4 Inspect the remote shape object.
5 Create (using the local factory finder) and inspect a local copy of the remote object.

Interfaces

DSTRepository’s COSLifecycle module includes the basic factory and lifecycle object interfaces. Browse these interfaces for details. The IDL hierarchy is as follows:

- FactoryRepresentative
- FactoryFinder
- LifecycleObject
- GenericFactory

The SmalltalkTypes module includes interfaces that support externalization and internalization.

- StreamRead
- Stream

Implementation

The class that is primarily responsible for basic lifecycle is DSTFactoryFinder. Browse this class for variables and methods. Browse these classes for variables and methods. Its position in the class hierarchy is:

- Object
- Model
- ORBOBJECT
- DSTPersistentObject
- DSTNameContext
- DSTfactoryFinder
Concurrency Control Service

The concurrency control service defines how an object mediates simultaneous access by one or more clients such that the consistency of the object is not compromised when accessed by concurrently executing processes. It is an implementation of the Object Management Group's Common Object Services Specification.

Overview

The concurrency control service interface can be used in two ways:

1. Acquiring locks on behalf of the current thread (that must be executing outside the scope of a transaction).
2. Acquiring locks on behalf of a transaction, or

The principal difference between these transactional and non-transactional modes of operation is that when operating in a transactional mode, the transaction service drives the release of the locks as the transaction commits or aborts. In non-transactional mode, the responsibility for dropping locks at the appropriate time lies with the user of the concurrency control service.

The basic notion is that the concurrency control service provides a mechanism for a resource to be associated with a lock. In reality, because of the lock semantics, this turns out to be a collection of locks, or a lock set. The meaning of a resource is not defined by the concurrency control service but by some object implementation which uses the service. The concurrency control service coordinates concurrent use of a resource using locks.
**Terminology**

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td>A lock represents the ability of a specific client to access a specific resource in a particular way. Each lock is associated with a single resource and a single client. Coordination is achieved by preventing multiple clients from simultaneously possessing locks for the same resource if the activities of those clients might conflict. To achieve coordination, a client must obtain an appropriate lock before accessing a shared resource.</td>
</tr>
<tr>
<td>lock set</td>
<td>A collection of locks associated with a single resource.</td>
</tr>
<tr>
<td>lock modes</td>
<td>Lock modes correspond to different categories of access.</td>
</tr>
<tr>
<td>lock granularity</td>
<td>Typically, if an object is a resource, the object would internally create and retain a lock set. However, the mapping between objects and resources (and lock sets) is up to the object implementation; the mapping could be one to one, but it could also be one to many, many to many, or many to one.</td>
</tr>
<tr>
<td>conflict resolution</td>
<td>The service will grant a lock to a client only if no other client holds a lock on the resource that would conflict with the intended access to the resource. The decision to grant a lock depends upon the modes of the lock held or requested. For example, a read lock conflicts with a write lock. If a write lock is held on a resource by one client, a read lock will not be granted to another client.</td>
</tr>
<tr>
<td>lock duration</td>
<td>Typically, a transaction will retain all of its locks until the transaction is completed (either committed or aborted). This policy supports serializability of transactional operations. Using the two phase commit protocol, locks held by a transaction are dropped when the transaction completes.</td>
</tr>
<tr>
<td>transaction duration locking</td>
<td>This a special case of strict two-phase locking. In the first phase (the growing phase), a transaction obtains locks that are kept until the second phase (the shrinking phase), at which point they are released. A transaction must not release locks during the first phase, and must not obtain new locks during the second phase, otherwise concurrent computations may be able to view intermediate results of the transaction.</td>
</tr>
</tbody>
</table>
Lock Modes

The concurrency control service defines five types of lock modes which implement conventional multiple readers, one writer semantics.

- **read (R)**
  Read locks conflict with write locks.

- **write (W)**
  Write locks conflict with other write locks.

- **upgrade (U)**
  An upgrade mode lock is a read lock that conflicts with itself. It is useful for avoiding a common form of deadlock that occurs when two or more clients attempt to read and then update the same resource. If more than one client holds a read lock on the resource, a deadlock will occur as soon as one of the clients requests a write lock on the resource. If each client requests a single upgrade lock followed by a write lock, this deadlock will not occur.

- **intention read (IR)**

- **intention write (IW)**
  Both intention read and intention write support variable granularity locking and are used to exploit the natural hierarchical relationship between locks of different granularity. For example, consider the hierarchical relationship inherent in a database: a database consists of a collection of files, with each file holding multiple records. To access a record, a coarse grain lock may be set on the database, but at the cost of restricting other clients from accessing the database. Clearly, this level of locking is unsuitable. However, only setting a lock on the record is also inappropriate, because another client might set a lock on the file holding the record and delete or modify the file.

  Using variable granularity locking, a client first obtains intention locks on the ancestor(s) of the required resource. To read a record in the database, for example, the client obtains an intention read lock (IR) on the database and the file (in this order) before obtaining the read lock (R) on the record. Intention read locks (IR) conflict with write locks (W), and intention write locks (IW) conflict with read (R) and write (W) locks.

  The granularity of the resources locked by an application determines the concurrency within the application. Coarse granularity locks incur low overhead (since there are fewer locks to manage) but reduce
concurrency since conflicts are more likely to occur. Fine granularity locks improve concurrency but result in a higher locking overhead since more locks are requested. Selecting a suitable lock granularity is a balance between the lock overhead and the degree of concurrency required.

**Lock Mode Compatibility**

<table>
<thead>
<tr>
<th>Granted Mode</th>
<th>IR</th>
<th>R</th>
<th>U</th>
<th>IW</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intention Read (IR)</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read (R)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Upgrade (U)</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intention Write (IW)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write (W)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

This table defines the compatibility between the various locking modes (the symbol * is used to indicate when locks conflict). When a client requests a lock on a resource that cannot be granted because another client holds a lock on the resource in a conflicting mode, the client must wait until the holding client releases its lock. The service enforces a queueing policy such that all clients waiting for a new lock are serviced in a first in, first out order. Subsequent requests are blocked by the first request waiting to be granted the lock, unless the requesting client is a transaction that is a member of the same transaction family as an existing holder of the lock. For a description of transaction families, see “Transaction Service”.

**Multiple Lock Semantics**

In this model, a client can hold multiple locks on the same resource simultaneously and the locks can be of different modes. In addition, a client can hold multiple locks of the same mode on the same resource; effectively, a count is kept of the number of locks of a given mode that have been granted to the client. When a client holds locks on a resource in more than one mode, the other clients will not be granted a lock on the resource unless the requested lock mode is compatible with all of the modes of the existing locks. A user can hold a lock multiple times; it must be released as many times as it was acquired in order to free the resource.
Locks and LockSets

Locks are implemented in the class Lock. Since multiple possession semantics are supported, locks must maintain a count. Locks also hold on to the context of the lock’s owner for validation of requests. In the general case, is associated with a distributed thread of control (ORB context), for Transactional Locks, it is the Transaction Context.

Locksets are implemented in the Smalltalk Class LockSet. A Lockset is an object that manages the locks on some resource. In general, a lockset is associated with a single resource and may hold many locks. This object provides methods to acquire and release locks.

The various lock modes are described as enumerations (browse DSTTypeEnumerator) that have symbol values corresponding to the lock modes defined in the service. The symbols and their corresponding lock modes are, in descending order of precedence:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Lock Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>#write</td>
<td>Write</td>
</tr>
<tr>
<td>#intentionWrite</td>
<td>Intention Write</td>
</tr>
<tr>
<td>#upgrade</td>
<td>Upgrade</td>
</tr>
<tr>
<td>#read</td>
<td>Read</td>
</tr>
<tr>
<td>#intentionRead</td>
<td>Intention Read</td>
</tr>
</tbody>
</table>

When a lock is requested, the request is validated according to the compatibility rules defined in “Lock Mode Compatibility”. During validation a check is made to insure that the requested lock mode does not conflict with the strongest lock mode granted. If it does the lock request will be not be granted at that time. For example, if a write lock is held, any requests for a lock on that resource by a different owner will not be granted.

The COS specification defines a LockSet coordinator interface, in order to provide an administrative interface and avoid potential deadlocks. The coordinator interface is implemented in the separate class DSTLockSetCoordinator.
Chapter 15  - Concurrency Control Service

Interfaces

The CosConcurrencyControl module in DSTRepository defines interfaces for concurrency. Browse these interfaces for details. The IDL hierarchy is as follows:

- LockSet
  - DSTCoordinatingLockSet: LockSet, LockCoordinator
  - DSTLockSet: LockSet, LifeCycleObject, FactoryRepresentative
- TransactionalLockSet
  - DSTTransactionalLockSet: TransactionalLockSet, FactoryRepresentative, LifeCycleObject,
- LockSetFactory
- LockCoordinator

Implementation

In Distributed Smalltalk, the classes for concurrency and its subclasses take these positions in the class hierarchy. Browse these classes for variables and methods. Their positions in the class hierarchy are:

- Object
  - DSTLock
  - DSTLockCoordinator
- Model
  - ORBOBJECT
  - DSTPersistentObject
  - DSTLockSet
    - DSTTransactionalLockSet
- DSTServiceContext
- DSTConcurrencyContext

Using Distributed Smalltalk Concurrency Service

The concurrency service is a general service that may be used to manage access on any distributed resource. In Distributed Smalltalk, there are two examples of the use of the concurrency service. They are as follows:

- DSTRecoverableObject (part of transaction service)
- DSTResourceManager
Using the Class DSTResourceManager

Instances of this class manage resources that are shared by locks between several owners. Lock ownership is implicitly defined on the distributed thread of control. (This can be overwritten in DSTResourceManager>>setContext).

The resources are identified by name, which is a string or symbol. The resource owners have to agree on a name for each resource they want to share. The class DSTResourceManager is used by DSTSemantic objects. It is also used to synchronize access to the interface repository. This class can be used directly, or as an example for using the concurrency service. The following code example can be browsed under the category of CORBA-CORE, in the class DSTResourceManager.

Creating Locks

The following method returns the lockset for the resource key, and creates one if necessary. The factory creation method, create, must be used to instantiate a lockset.

The hint is any user defined string that will be associated with the lock.

```smalltalk
findOrCreateLockFor: key hint: aHint
  ^locksets at: key ifAbsent:
    [hints at: key put: aHint.
     locksets at: key put: DSTLockSet create]
```

Acquiring Locks

The following method acquires a lock on the named resource and gets the LockSet for the named resource first. Locksets manage ownership implicitly through the concurrency context which is propagated along the thread of control in the ORB context. Therefore, we need to set the context before attempting to acquire the lock.

In the following method, aLockModeEnum is an enumerator which identifies the strength of the desired lock. Browse senders of this method using an enumerator object. DSTLockSet has class side methods which return the enumerators which correspond to the defined lock modes. For example, DSTLockSet class > read will return the enumerator defined in the CORBAConstants dictionary as:

```
::CosConcurrencyControl::lock_mode::read.
```

If the lock is granted and the granted level is the highest for the lockset, then the hint is updated. This is because multiple holders are allowed for a lockset and multiple levels of locks are allowed for each owner. The
method returns a boolean that will be true if the lock was successful. The hint is a value holder whose value will contain information about the current owner of the highest, most recently granted lock.

This code example illustrates acquiring locks:

```plaintext
acquire: aResourceName mode: aLockModeEnum with: aHint

| key lock acquired |
DSTLockSet setContext.
key := aResourceName asSymbol.
lock := self findOrCreateLockFor: key hint: aHint value.
(acquired := lock tryLock: aLockModeEnum)
ifTrue: [lock granted == aLockModeEnum
    ifTrue: [hints at: key put: aHint value]
    ifFalse: [aHint value: (self hintOf: key)].
] acquired
```

**Releasing Locks**

The following method deals with releasing locks:

```plaintext
release: aResourceName mode: aLockModeEnum

| key lockset |
key := aResourceName asSymbol.
lockset := locksets at: key ifAbsent: [^false].
Object errorSignal
    handle: [:ex | ^false]
    do:
            ^true]
```

**Destroying Locks**

Since the class DSTLockSet is a subclass of ORBObject and created using factory methods, it must be explicitly destroyed. The following method destroys the receiver and any locksets that it may hold.

```plaintext
destroy
locksets values do: [ : lockset | lockset destroy ].
super destroy.
```
Using Transactional Locksets

A transactional lockset is a kind of lockset intended to be used within the scope of a transaction. It is used by clients of the transaction service, generally by DSTRecoverableObjects.

The only difference between locksets and transactional locksets is that with a transactional lockset, the transaction context is used for validation.
Overview

The transaction service supports transactions that have the following ACID characteristics:

- **Atomicity** — This ensures that the set of computations is either completely done or completely undone. A transaction whose work completes is said to commit. A transaction whose work is completely undone is said to rollback. A transaction may rollback due to system failures (for example, processor failure or a deadlock), or because a programmer chose to execute an rollback call.

- **Consistency** — The effects of a transaction preserves invariant properties. A transaction leaves the collection of objects in a consistent state. There may be integrity rules that must be checked before commit.

- **Isolation** — Transactions are allowed to execute concurrently, but the results will be the same as if the transactions executed serially. Isolation ensures that concurrently executing transactions cannot observe inconsistencies. Programmers are therefore free to cause
temporary inconsistencies during the execution of a transaction knowing that their partial modifications will never be visible.

- Durability — If a transaction completes successfully, the results of its operations will never be lost, except in the event of catastrophes. Systems can be designed to reduce the risk of catastrophes.

A transaction can be terminated in two ways: the transaction is either committed or rolled back. When a transaction is committed, all changes made by the associated requests are made permanent. When a transaction is rolled back, all changes made by the associated requests are undone.

The transaction service defines interfaces that allow multiple, distributed objects to cooperate to provide atomicity. These interfaces enable the objects to either commit all changes together or to rollback all changes together, even in the presence of (noncatastrophic) failure. No requirements are placed on the objects other than those defined by the transaction service interfaces.

Examples are OODBMS, and persistent objects. The value of a separate transaction service is that it allows:

- Transactions to include multiple, separately defined, ACID objects.
- The possibility of transactions which include objects and resources from the non-object world.

---

**Distributed Smalltalk’s Implementation of Transactions**

The Distributed Smalltalk implementation of the transaction service provides:

- A transactional infrastructure for coordination among transactional objects.
- Classes that implement transactional object and recoverable object behavior.
- A framework for the class implementor to develop specific kinds of transactional and recoverable objects based upon their persistence requirements.

The infrastructure is implemented as a set of services that allow the threads executing operations on objects within a transaction to work together harmoniously to provide the ACID properties. This infrastructure consists of:
Chapter 16 - Transaction Service

- Operations for begin, commit, and rollback of a transaction.
- Operations for associating transactions with threads and mechanisms for propagating transactions to other objects whose behavior is affected by a transaction.
- Operations for objects with recoverable state to participate in transaction completion.
- A protocol engine which implements the two-phase commit protocol with presumed abort optimization to ensure that all participants within a transaction commit or rollback together.
- An enhancement to CORBA to permit transaction context to be passed between cooperating implementations of the Transaction Service.

In addition to the transactional infrastructure, a class implementor needs tools in order to develop objects that have the requisite atomicity, durability, and isolation properties. The concurrency control service augmented with the transaction service provides this support. The transaction service support for the class implementor is a mechanism that coordinates the use of concurrency and persistence by an object as transactions are created, rolled back, and committed. For example, the concurrency control service supports transactional locking which ensures that locks acquired on objects during that transaction will be released at transaction termination.

In Distributed Smalltalk, support for this mechanism is built into the class DSTRecoverableObject. These objects hold transactional locksets that are used during the two phase commit process. Subclasses of DSTRecoverableObject need only implement the methods to save and recover their persistent state according to the underlying persistent store that is in place.

Two types of transactions are supported:

- **Flat transactions**—A flat transaction groups all operations within its scope into a single transactional entity.
- **Nested transactions**—A nested transaction is a transaction embedded within another transaction.

Nested transactions rollback independently from their parent transaction and can be used within concurrently executing threads to increase system performance while maintaining consistency (since the nested transactions serialize with respect to each other). The results of a nested transaction only become permanent when its top-
level transaction is committed. Nested transactions are especially valuable for encapsulating an object’s transactional behavior, and enable transactions to become a general programming mechanism for constructing reusable building blocks for reliable distributed applications.

A client can make requests to multiple objects that may be located on different nodes in the network within the scope of a transaction.

For compatibility with X/Open, implementations of the Object Transaction Service may track the “spread” of the transaction so that all participants in the transaction see the same outcome. Similarly, these OTS implementations track the spread when the implementation of an object in turns act as the client of other remotely-located objects.

Supporting these distributed transactions requires support from the ORB so that the “transactional-context” is passed transparently with each request.
## Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>The state of a transaction when processing is in progress and completion of the transaction has not yet commenced.</td>
</tr>
<tr>
<td>atomicity</td>
<td>A transaction property that ensures that if work is interrupted by failure, any partially completed results will be undone. A transaction whose work completes is said to commit. A transaction whose work is completely undone is said to rollback (abort).</td>
</tr>
<tr>
<td>begin</td>
<td>An operation on the Transaction Service which establishes the initial boundary of a transaction.</td>
</tr>
<tr>
<td>commit</td>
<td>Commit has two definitions as follows:</td>
</tr>
<tr>
<td></td>
<td>• An operation in the Current and Terminator interfaces that a program uses to request that the current transaction terminate normally and that the effects of that transaction be made permanent.</td>
</tr>
<tr>
<td></td>
<td>• An operation in the Resource interface which causes the effects of a transaction to be made permanent.</td>
</tr>
<tr>
<td>committed</td>
<td>The property of a transaction or a transactional object, when it has successfully performed the commit protocol.</td>
</tr>
<tr>
<td>completion</td>
<td>The processing required (either by commit or abort) to obtain the durable outcome of a transaction.</td>
</tr>
<tr>
<td>coordinator</td>
<td>A object involves Resource objects in a transaction when they are registered. A coordinator is responsible for driving the two-phase commit protocol.</td>
</tr>
<tr>
<td>concurrency control service</td>
<td>See “Concurrency Control Service” on page 191</td>
</tr>
<tr>
<td>direct context management</td>
<td>An application manipulates the Control object and the other objects associated with the transaction.</td>
</tr>
<tr>
<td>flat Transaction</td>
<td>A transaction that has no subtransactions, and that cannot have subtransactions.</td>
</tr>
<tr>
<td>indirect context management</td>
<td>An application uses the Current pseudo object, provided by the Transaction Service, to associate the transaction context with the application thread of control. See DSTTransactionalObject.</td>
</tr>
<tr>
<td>nested transaction</td>
<td>A transaction that either has a subtransaction or is a subtransaction on some other transaction.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>prepared</td>
<td>The state that a transaction is in when phase one of a two-phase commit has completed.</td>
</tr>
<tr>
<td>Propagation</td>
<td>A function of the transaction service that allows the Transaction context of a client to be associated with a transactional operation on a server object. The Transaction Service supports both implicit and explicit propagation of transaction context.</td>
</tr>
<tr>
<td>recoverable object</td>
<td>An object whose data is affected by committing or rolling back a transaction. See DSTRecoverableObject.</td>
</tr>
<tr>
<td>recoverable server</td>
<td>An object that registers a Resource (not necessarily itself) with a Transaction Coordinator to participate in transaction completion.</td>
</tr>
<tr>
<td>recovery service</td>
<td>An object service used by resource managers for restoring the state of objects to a prior state of consistency.</td>
</tr>
<tr>
<td>resource</td>
<td>An object in the transaction service that is registered for involvement in two-phase commit. An object that supports the Resource interface. See DSTRecoverableObject.</td>
</tr>
<tr>
<td>rollback</td>
<td>Rollback (also known as Abort) has two definition as follows:</td>
</tr>
<tr>
<td></td>
<td>• An operation in the Current and Terminator interfaces used to indicate that the current transaction has terminated abnormally and its effects should be discarded.</td>
</tr>
<tr>
<td></td>
<td>• An operation in the Resource interface which causes all state changes in the transaction to be undone.</td>
</tr>
<tr>
<td>thread</td>
<td>The entity that is currently in control of the processor.</td>
</tr>
<tr>
<td>TP (Transaction Process) monitor</td>
<td>A system component that accepts input work requests and associates resources with the programs that act upon these requests to provide a run-time environment for program execution.</td>
</tr>
<tr>
<td>transaction</td>
<td>A collection of operations on the physical and abstract application state.</td>
</tr>
<tr>
<td>transaction client</td>
<td>An arbitrary program that can invoke operations of many transactional objects in a single transaction. Not necessarily the Transaction originator.</td>
</tr>
<tr>
<td>transaction context</td>
<td>The transaction information associated with a specific thread. See Propagation.</td>
</tr>
<tr>
<td>transaction operation</td>
<td>An operation on an object that participates in the propagation of the current transaction.</td>
</tr>
</tbody>
</table>
The transaction service provides transaction synchronization across the elements of a distributed client/server application.

A transaction can involve multiple objects performing multiple requests. The scope of a transaction is defined by a transaction context that is shared by the participating objects. The transaction service places no constraints on the number of objects involved, the topology of the application or the way in which the application is distributed across a network. In this scenario, the transaction context is transmitted implicitly to the objects, without direct client intervention.

In a typical scenario, a client first begins a transaction (by issuing a request to an object defined by the transaction service), which establishes a transaction context associated with the client thread. The client then issues requests. These requests are implicitly associated with the client's transaction; they share the client's transaction context. Eventually, the client decides to end the transaction (by issuing another request). If there were no failures, the changes produced as a consequence of the client's requests would then be committed; otherwise, the changes would be rolled back.

The transaction service also supports scenarios where the client directly controls the propagation of the transaction context. For example, a client can pass the transaction context to an object as an explicit parameter in a request.

**transaction originator**
An arbitrary program—typically, a transactional client, but not necessarily an object—that begins a transaction.

**transaction object**
An object that offers at least one transactional operation, thus requiring the ORB and the transaction service to propagate a Transaction Context, usually used to refer to an object, none of whose operations are affected by being involved within the scope of a transaction. See DSTTransactionalObject.

**transaction server**
A collection of one or more objects whose behavior is affected by the transaction, but have no recoverable states of their own.

**two-phase commit**
A transaction manager protocol for ensuring that all changes to recoverable resources occur atomically and furthermore, the failure of any resource to complete will cause all other resource to undo changes.

### Transactional Applications

The transaction service provides transaction synchronization across the elements of a distributed client/server application.

A transaction can involve multiple objects performing multiple requests. The scope of a transaction is defined by a transaction context that is shared by the participating objects. The transaction service places no constraints on the number of objects involved, the topology of the application or the way in which the application is distributed across a network. In this scenario, the transaction context is transmitted implicitly to the objects, without direct client intervention.

In a typical scenario, a client first begins a transaction (by issuing a request to an object defined by the transaction service), which establishes a transaction context associated with the client thread. The client then issues requests. These requests are implicitly associated with the client's transaction; they share the client's transaction context. Eventually, the client decides to end the transaction (by issuing another request). If there were no failures, the changes produced as a consequence of the client's requests would then be committed; otherwise, the changes would be rolled back.

The transaction service also supports scenarios where the client directly controls the propagation of the transaction context. For example, a client can pass the transaction context to an object as an explicit parameter in a request.
The transaction service does not require that all requests be performed within the scope of a transaction. A request issued outside the scope of a transaction has no associated transaction context. It is up to each object to determine its behavior when invoked outside the scope of a transaction; an object that requires a transaction context can raise a standard exception.

This diagram shows a simple application that includes these basic elements: Transactional Client, Transactional Objects, Recoverable Objects, and Transactional Servers. The discussion of these follows.
Chapter 16 - Transaction Service

Transactional Client

A transactional client is an arbitrary program that can invoke operations of many transactional objects in a single transaction. The program that begins a transaction is called the transaction originator. The originator may be the same object as the client. In fact, often the transaction may be implicitly originated by invoking an operation on a recoverable object.

Transactional Object

We use the term transactional object to refer to an object whose behavior is affected by being involved within the scope of a transaction. A transactional object typically contains or indirectly refers to persistent data that can be modified by requests.

In Distributed Smalltalk this term refers to objects that implement the Current pseudo object interface. See DSTTransactionalObject.

The transaction service does not require that all requests have transactional behavior, even when issued within the scope of a transaction. An object can choose to not support transactional behavior, or to support transactional behavior for some requests but not others.

We use the term nontransactional object to refer to an object none of whose operations are affected by being involved within the scope of a transaction. If an object does not support transactional behavior for a request, then the changes produced by the request might not survive a failure and the changes will not be undone if the transaction associated with the request is rolled back.

An object can also choose to support transactional behavior for some requests but not others. This choice can be exercised by both the client and the server of the request.

The transaction service permits an interface to have both transactional and nontransactional implementations. No IDL extensions are introduced to specify whether or not an operation has transactional behavior. When objects use implicit context propagation transactional behavior can be a quality of service that differs in different implementations.

Transactional objects are used to implement two types of application servers:

1. Transactional Server
2. Recoverable Server
Recoverable Objects and Resource Objects

To implement transactional behavior, an object must participate in certain protocols defined by the transaction service. These protocols are used to ensure that all participants in the transaction agree on the outcome (commit or rollback), and to recover from failures.

To be more precise, an object is required to participate in these protocols only if it directly manages data whose state is subject to change within a transaction. An object whose data is affected by committing or rolling back a transaction is called a recoverable object.

A recoverable object is by definition a transactional object. However, an object can be transactional but not recoverable by implementing its state using some other (recoverable) object. A client is concerned only that an object is transactional; a client cannot tell whether a transactional object is or is not a recoverable object.

A recoverable object must participate in the transaction service protocols. It does so by registering an object that implements the `Resource` interface. The transaction service drives the commit protocol by issuing requests to the resources registered for a transaction.

A recoverable object typically involves itself in a transaction because it is required to retain in stable storage certain information at critical times in its processing. When a recoverable object restarts after a failure, it participates in a recovery protocol based on the contents (or lack of contents) of its stable storage.

A transaction can be used to coordinate non-durable activities which do not require permanent changes to storage.

Transactional Server

A transactional server is a collection of one or more objects whose behavior is affected by the transaction, but have no recoverable states of their own. Instead, it implements transactional changes using other recoverable objects. A transactional server does not participate in the completion of the transaction, but it can force the transaction to be rolled back.
Recoverable Server

A recoverable server is a collection of objects, at least one of which is recoverable.

A recoverable server participates in the protocols by registering one or more Resource objects with the transaction service. The transaction service drives the commit protocol by issuing requests to the resources registered for a transaction.

Transaction Service Functionality

The transaction service provides operations to:

- Control the scope and duration of a transaction
- Allow multiple objects to be involved in a single, atomic transaction
- Allow objects to associate changes in their internal state with a transaction
- Coordinate the completion of transactions

Transaction Models

Two distributed transaction models are supported: flat transactions and nested transactions.

Flat Transactions

The definition of the functionality provided, and the commitment protocols used, is modelled on the X/OpenDTP transaction model definition.

A flat transaction is considered to be a top-level transaction (see the next section) that cannot have a child transaction. In Distributed Smalltalk there is nothing to restrict a top-level transaction from having child transactions.

Nested Transactions

The transaction service also defines a nested transaction model. Nested transactions provide for a finer granularity of recovery than flat transactions. The effect of failures that require rollback can be limited so that unaffected parts of the transaction need not rollback.

Nested transactions allow an application to create a transaction that is embedded in an existing transaction. The existing transaction is called the parent of the subtransaction; the subtransaction is called a child of the parent transaction.
Multiple subtransactions can be embedded in the same parent transaction. The children of one parent are called _siblings_.

Subtransactions can be embedded in other subtransactions to any level of nesting. The ancestors of a transaction are the parent of the subtransaction and (recursively) the parents of its ancestors. The _descendants_ of a transaction are the children of the transaction and (recursively) the children of its descendants.

A top-level transaction is one with no parent. A top-level transaction and all of its descendants are called a _transaction family_.

A subtransaction is similar to a top-level transaction in that the changes made on behalf of a subtransaction are either committed in their entirety or rolled back. However, when a subtransaction is committed, the changes remain contingent upon commitment of all of the transaction’s ancestors.

Subtransactions are strictly nested. A transaction cannot commit unless all of its children have completed. When a transaction is rolled back, all of its children are rolled back.

Objects that participate in transactions must support isolation of transactions. The concept of isolation applies to subtransactions as well as to top level transactions. When a transaction has multiple children, the children appear to other transactions to execute serially, even if they are performed concurrently.

Subtransactions can be used to isolate failures. If an operation performed within a subtransaction fails, only the subtransaction is rolled back. The parent transaction has the opportunity to correct or compensate for the problem and complete its operation. Subtransactions can also be used to perform suboperations of a transaction in parallel, without the risk of inconsistent results.

**Transaction Termination**

A transaction is terminated by issuing a request to commit or rollback the transaction. Typically, a transaction is terminated by the client that originated the transaction — the transaction originator. Some implementations of the transaction service may allow transactions to be terminated by transaction service clients other than the one which created the transaction.
Any participant in a transaction can force the transaction to be rolled back (eventually). If a transaction is rolled back, all participants rollback their changes. Typically, a participant may request the rollback of the current transaction after encountering a failure.

Transaction Context

As part of the environment of each ORB-aware thread, the ORB maintains a transaction context. The transaction context associated with a thread is either nil (indicating that the thread has no associated transaction) or it refers to a specific transaction. It is permitted for multiple threads to be associated with the same transaction at the same time, in the same execution environment or in multiple execution environments.

The transaction context is implicitly transmitted to transactional objects as part of a transactional operation invocation. The transaction service also allows programmers to pass a transaction context as an explicit parameter of a request.

Service Architecture

The figure below illustrates the major components and interfaces defined by the transaction service. The transaction originator is an arbitrary program that begins a transaction.

![Diagram of Service Architecture](image-url)
The transaction originator creates a transaction using a Factory (DSTTransaction class>>create); a Control is returned that provides access to a Terminator and a Coordinator. The transaction originator uses the Terminator to commit or rollback the transaction. The Coordinator is made available to recoverable servers, either explicitly or implicitly (by implicitly propagating a transaction context with a request). Control, Coordinator and Terminator are IDL interfaces that are all implemented in the class DSTTransaction.

A recoverable server registers a Resource with the Coordinator. The Resource implements the two-phase commit protocol which is driven by the transaction service. A recoverable server can also register a specialized resource called a SubtransactionAwareResource to track the completion of subtransactions. DSTRecoverableObject supports both the Resource and SubtransactionAware interfaces.

To simplify coding, most applications use the Current pseudo object interface, which provides access to an implicit per-thread transaction context. DSTTransactionalObject implements the Current interface. DSTRecoverableObject is a subclass of DSTTransactionalObject and inherits from this implementation.

**Typical Usage**

A typical transaction originator uses the Current pseudo object to begin a transaction, which becomes associated with the transaction originator’s thread.

The transaction originator then issues requests. Some of these requests involve transactional objects. When a request is issued to a transactional object, the transaction context associated with the invoking thread is automatically propagated to the thread executing the method of the target object. No explicit operation parameter or context declaration is required to transmit the transaction context. Propagation of the transaction context can extend to multiple levels if a transactional object issues a request to a transactional object.

```smalltalk
| current |
current := DSTTransactionalObject new.
current begin
  ...
```

Using the pseudo object, the transactional object can unilaterally rollback the transaction and can inquire about the current state of the transaction. Using the pseudo object, the transactional object also can obtain a
Coordinator for the current transaction. Using the Coordinator, a transactional object can determine the relationship between two transactions, to implement isolation among multiple transactions.

By implementing the Current interface in DSTTransactionalObject, we allow all Transactional and Recoverable objects an interface that allows operations to be sent that effect the entire Transaction and not just the particular Resource object. (DSTRecoverableObject is a subclass of DSTTransactionalObject.)

Some transactional objects are also recoverable objects. A recoverable object has persistent data that must be managed as part of the transaction. A recoverable object uses the Coordinator to register a Resource object as a participant in the transaction. The resource represents the recoverable object’s participation in the transaction; each resource is implicitly associated with a single transaction. The Coordinator uses the resource to perform the two-phase commit protocol on the recoverable object’s data.

After the computations involved in the transaction have been completed, the transaction originator uses the pseudo object to request that the changes be committed. The transaction service commits the transaction using a two-phase commit protocol, wherein a series of requests are issued to the registered resources.

**Transaction Context**

A transaction context can be associated with each ORB-aware thread. The transaction context associated with a thread is either nil (indicating that the thread has no associated transaction), or it refers to a specific transaction. It is permitted for multiple threads to be associated with the same transaction at the same time.

When a thread in an object server is used by an object adapter to perform a request on a transactional object, the object adapter initializes the transaction context associated with that thread by effectively copying the transaction context of the thread that issued the request.

The transaction context is used to determine ownership of the resources involved in a transaction. DSTTransactionalObjects use DSTTransactionalLockSets to control access to resources, and the transactional locksets use information in the transaction context to determine ownership.
**Context Management**

The transaction service supports management and propagation of transaction context - using objects provided by service. Using this approach, the transaction originator issues a request to a Factory to begin a new top-level transaction. The factory returns a Control object specific to the new transaction that allows an application to terminate the transaction or to become a participant in the transaction (by registering a resource). An application can propagate a transaction context by passing the Control as an explicit request parameter.

The Control interface does not directly support management of the transaction. Instead, it supports operations that return two other objects, a Terminator and a Coordinator. The Terminator is used to commit or rollback the transaction. The Coordinator is used to enable transactional objects to participate in the transaction. These two objects can be propagated independently, allowing finer granularity control over propagation.

In Distributed Smalltalk, the Control, Terminator and Coordinator interfaces are all implemented in DSTTransaction. However, the implementation does not preclude the use of external or specialized Coordinators and Terminators. For this reason, it is strongly recommended that users of the service acquire and use the Control, Coordinator and Terminator interfaces as defined.

**Interfaces**

The CosTransactions module in DSTRepository’s defines interfaces for transactions. Browse these interfaces for details. The IDL hierarchy is as follows:

- **Factory**
  - **DSTTransactionFactory**: Factory, ClassObject
- **Control**
  - **DSTControl**
- **Terminator**
- **Coordinator**
  - **DSTCoordinator**
    - **DSTTransaction**: DSTCoordinator, DSTControl, Terminator
### Use of Transaction Service Functionality for Interfaces

<table>
<thead>
<tr>
<th>Function</th>
<th>Used by</th>
<th>Context Management</th>
<th>Indirecta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a transaction</td>
<td>Transaction originator</td>
<td>Factory::create</td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Control::get_terminator</td>
<td>set_timeout</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Control::get_coordinator</td>
<td></td>
</tr>
<tr>
<td>Terminate a transaction</td>
<td>Transaction originator - implicit</td>
<td>Terminator::commit</td>
<td>commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Terminator::rollback</td>
<td>rollback</td>
</tr>
<tr>
<td>Rollback a transaction</td>
<td>Server</td>
<td>Terminator::rollback_only</td>
<td>rollback_only</td>
</tr>
<tr>
<td>Control propagation of transaction to a server</td>
<td>Server</td>
<td>Declaration of method parameter TransactionalObject interface</td>
<td></td>
</tr>
<tr>
<td>Control by client of transaction propagation to a server</td>
<td>All</td>
<td>Request parameters get_control suspend resume</td>
<td></td>
</tr>
<tr>
<td>Become a participant in a transaction</td>
<td>Recoverable Server</td>
<td>Coordinator::register_resource</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>All</td>
<td>Coordinator::get_status Coordinator::get_transaction_name Coordinator::is_same_transaction Coordinator::hash_transaction</td>
<td>get_status get_transaction_name Not applicable Not applicable</td>
</tr>
</tbody>
</table>

a. All indirect context management operations are on the **Current** pseudo-object interface.
Implementation

In Distributed Smalltalk, the classes for transactions take these positions in the class hierarchy. Browse these classes for variables and methods. Their positions in the class hierarchy are:

Object
  Model
    ORBObj
      DSTPersistentObject
        DSTTransactionalObject
        DSTRecoverableObject
          DSTSampleRecoverableObject
          DSTRecoveryCoordinator
        DSTDTR
          DSTServiceContext
            DSTDTC
              DSTDTCI

Using the Distributed Smalltalk Transaction Service

The classes that implement the transaction service are in the class category COS-Transactions. You can browse those classes to learn more about transaction policies and services.

Implementing a Recoverable Object

The Recoverable Object framework in Distributed Smalltalk is designed to be extended for specific kinds of persistent objects, or objects which are stored in particular kinds of data stores (DBMS).

As an implementor of a Recoverable Object, you will subclass from DSTRecoverableObject and add behavior specific to that subclass. DSTTransactionalObject and DSTRecoverableObject are both abstract classes.

Example

An example of a concrete Recoverable Object can be seen in the class DSTSampleRecoverableObject. This object implements a simple persistence mechanism based upon the externalization and internalization mechanism in one of its parent classes, DSTPersistentObject. This example shows the minimum work a subclass of DSTRecoverableObject needs to do to participate as a Resource within a distributed transaction.
In this example, five methods are overwritten in the concrete class:

1. **commitData**
   - This message is sent by the recoverable object framework when an object's data should be committed.

2. **prepareState**
   - This message is sent by the recoverable object framework in the first phase of a two phase commit.

3. **rollbackState**
   - This message is sent by the recoverable object framework when an object's state should be rolled back within a transaction.

The first three messages are sent during the two phase commit process. In the first phase, the **prepareState** message will be sent. At this time an object must prepare itself to either rollback or commit its state. It must return either true or false.

4. **commitSubtransaction**:
   - This message is sent when a resource has been registered with a subtransaction and the subtransaction has been committed.

5. **rollbackSubtransaction**
   - This message is sent when a resource has been registered with a subtransaction and the subtransaction has been rolled back.

When the **commitData** or **rollbackState** message is sent, the object is responsible for performing the appropriate action.

For specific databases, this is the point where you will perform the actual database commit or rollback operations.

You will also need to determine when the actual starting of the database transaction should be performed.

One convenient method is to override the **beginTransaction** operation in **DSTTransactionalObject** to perform the operation at this time.

In practice, many persistent stores do not implement a full two-phase commit.

In this case, implementors of Recoverable Objects will need to override additional operations to insure consistency.
For example, you may be using a persistent store that only supports a single database transaction per image. In this case you may want to implement a scheme to maintain a single database transaction within a set of logical distributed transactions.

**Example**
You may create a notion of a root transaction within your Recoverable Object and when you begin a transaction, first check to see if you are operating within a transaction. If not, create one, if so, create a subtransaction within the current context:

```smalltalk
beginTransaction
    "Create a new transaction within the current thread. If I am in a transaction already, create this transaction as a subtransaction in order to operate within a single database transaction. Note: beginDatabaseTransaction will begin the actual transaction on the DBMS."

    self transaction isNil
    ifTrue: [
        self transaction: (DSTTransaction create: 0).
        rootTransaction := self transaction.
        self transaction getCoordinator registerResource: self.
        self beginDatabaseTransaction]
    ifFalse: [self beginSubTransaction]
```

**Creating a Transaction**

Transactions may be created using either:

- The Current pseudo object interface (implemented on DSTTransactionalObject)
- The explicit factory methods on DSTTransaction

If you use the Current interface, a transaction will be created on your behalf. The Current interface is intended to be used by transactional objects so that they do not need to directly manage the transaction.

A Recoverable Object may create a transaction by sending the `beginTransaction` message to itself. This will both create the transaction and register the object as a resource with the transaction. For example:

```smalltalk
| resource |
resource := DSTSampleRecoverableObject new initialize.
resource beginTransaction.
```

In the above code fragment the method `beginTransaction` is the Smalltalk implementation of the operation `begin` on the Current pseudo object.
This is equivalent to the following code fragment which explicitly creates a transaction and registers a resource with it:

```smalltalk
| resource control coord |
resource := DSTSampleRecoverableObject new initialize.
control := DSTTransaction create: 0.
coord := control getCoordinator.
coord registerResource: resource.
```

Creating a transaction will set the transaction context within the current (distributed) thread of control. This context information is implicitly passed along the thread of control until the transaction is terminated. Once started the transaction must be terminated by either rolling back or committing the transaction. At this point, the context information which is implicitly passed along the thread of control will be destroyed.

**Completing a Transaction**

While a transaction is in effect, messages may be sent to a Recoverable Object which effect the state of the object. During the first phase of the two phase commit, an object will be sent the `prepare` message by the transaction terminator. The Recoverable Object must reply with a vote which tells the terminator whether the object can commit or should be rolled back. The appropriate vote responses are defined as enumerations and can be accessed via accessors on `DSTTransaction` class.

If an object votes to rollback during the prepare (first) phase, all objects which are part of the transaction and have voted to commit will then be asked to rollback during the second phase. If no objects vote to roll back during the prepare phase, then all of the objects which voted to commit, will be asked to commit.

An object will generally maintain its state in such a way that when asked to prepare for a commit, its instance variable vote will be in an appropriate state. For example, an object may have started an operation which could leave it in an inconsistent state until completed. During that window, the object may choose to set its vote to rollback. Upon successful completion it may set its vote to commit.
Example
A subclass of DSTRecoverableObject which implements a distributed cache of inventory items may have an operation such as:

``` Smalltalk
cacheAll
  'Get all the inventory objects from the database and cache them into local objects in my image.'

lockset lock: self coordinator mode: lockset class read
  self vote: DSTTransaction voteRollback.
itemCache := InventoryItem new allObjects.
  self vote: DSTTransaction voteCommit.
lockset unlock: self coordinator mode: lockset class read.
```

In the above example, notice that the Recoverable Object uses an object named lockset to manage concurrent access to its resources. Instances of DSTRecoverableObject have their own Transactional Locksets which should be used by the Recoverable Object when appropriate. For more information on locksets, "Concurrency Control Service".

If the transaction were to be prepared during the period when the cache was being established, this object would vote to rollback and the transaction would be rolled back.

Transactions may be completed in either one of two ways:

- By explicitly using the transaction terminator
- By invoking methods which map to the Current pseudo object interface

A transaction may be rolled back by a Recoverable Object using the Current pseudo object as follows:

``` Smalltalk
| resource |
resource := DSTSampleRecoverableObject new initialize.
resource beginTransaction.
...
resource commitTransaction: false
```

This is equivalent to the following code fragment which explicitly creates a transaction and registers a resource with it:
| resource control coord terminator |
resource := DSTSampleRecoverableObject new initialize.
control := DSTTransaction create: 0.
coord := control getCoordinator.
coord registerResource: resource.

... 
terminator := control getTerminator.
terminator commit: false

Create a Transaction Example

This example can be browsed on the class side of
DSTSampleRecoverableObject and demonstrates the use a recoverable
object with indirect context.

Example: aFactory

"This example demonstrates the use of a recoverable object
with indirect context management (the Current pseudoObject)"
"Processor activeProcess orbContext transactionContext: nil."
"self example1: ORBObject factoryFinder"

| suspended resource |
resource := self create.
resource beginTransaction.
suspended := resource suspendTransaction.
DSTTransaction noTransactionSignal handle:
    [:x | "This should raise an exception"]
do:
    [ resource commitTransaction: false.
      self error: 'testCurrentInterfaceNear: suspend failed'; cr ].
resource resumeTransaction: suspended.
Object errorSignal handle:
    [:x | self error: 'example1: resume failed'; cr ]
do: [resource commitTransaction: false].
resource beginTransaction.
resource vote: DSTTransaction voteCommit.
resource commitTransaction: false.
resource getTransactionStatus == DSTTransaction statusCommitted
    ifFalse: [ self error: 'example1: Failure, transaction status'
      resource getTransactionStatus printString; cr ].
resource beginTransaction.
resource vote: DSTTransaction voteRollback.
resource rollbackTransaction.
^resource
17

Debugging and Tuning

Overview

Debugging and tuning is somewhat more complicated for distributed applications than for local applications. This chapter introduces the tools provided with DST for debugging and performance tuning. It also provides hints for optimizing performance.

The steps you should follow in troubleshooting and tuning are:

1. Get the application running in a local image.
2. Test in a simulated distributed environment (Local RPC).
3. Test in a distributed environment.
4. Optimize and tune for performance.

Debugging and Tuning Tools

The DST Tool is the primary user interface to Distributed Smalltalk, and provides access to further tools that are useful to administrators and developers for testing and maintenance. Specifically, it provides access to monitoring and debugging facilities in DST.

Debugging

Checking the Debugging checkbox enables the error handler for remote debugging. If an error is detected on an image with debugging enabled, a notifier will allow you to open a local Smalltalk Debugger on that machine. If debugging is enabled, and a remote error occurs, an exception handler displays, with the stack already unwound to the context of the client that sent the request.
**Message Logging**

Clicking the Monitor button opens a window that displays remote RPC activity.

The **Active Conversations** subview columns are:

- **TYPE** — RPC client or server.
- **STATE** — working, done, final, wait, or init.
- **OPERATION** — message sent.
- **TARGET** — target object’s abstractClassId.
- **ACTIVITY** — interface’s mostDerivedInterfaceID.

**Local RPC Testing**

Checking the **Local Testing** checkbox causes the local image to simulate a distributed object environment by using the full IDL marshaling and unmarshaling machinery.
Local RPC Testing

Local Testing simulates remote execution by wrapping a local object in a special proxy—an instance of DSTObjRefLocal—that invokes the lower-level marshalling and unmarshalling machinery typical of a remote call.

To enable local testing, you do two things:

1. Check Local Testing on the DST Tool, and
2. Sending asRemotable to the objects under test.

Local Testing is object-specific; you must explicitly wrap one or more objects implement the interface you intend to test in a proxy, by sending them the message asRemotable. For example, to test the interface for the operation quantity of an instance of Order, you would turn on Local Testing and evaluate something like:

   Order new asRemotable quantity.

   The effect of asRemotable can be undone by sending a wrapped object the message asLocal. If, during Local Testing, you want to ensure that an object is local, you can send it the message mustBeLocal, which will raise an exception if an object proves not to be local after an attempt to make it so.

   It is important to make all of the objects remoteable that need to be made remoteable, in order to completely test your interface(s).

   If you do not turn local testing OFF when testing a remote execution, you may have local, simulated behavior where you intended none. If you do not turn local RPC testing ON when testing a partially remote execution, you may have true, distributed behavior where you intended local simulation.

   Occasionally, you may notice an initially surprising number of DSTObjRefLocals. The principle cause is the creation of additional DSTObjRefLocals by the Basic Lifecycle service. If Local Testing is turned on, remote creation requests that utilize the Lifecycle Service will generate DSTObjRefLocals in order to simulate remote creation.
Remote Object Debugging

Once you have distributed your application among multiple machines, you will sometimes need to debug and inspect a remote image itself. The primary tools in Distributed Smalltalk to help you with this are:

- Remote Browser—a regular hierarchy browser that can look at or edit both local and remote classes.
- Remote Debugger—a full debugger, showing stack, method code and inspectors of receivers and their instance variables on all systems involved in the distributed execution context.

Using the Remote Object Debugger

The Remote Object Debugger is a complete debugger on a distributed process. You can use it to view the stack, step through message sends, edit methods, and examine and change the values of variables.

To enable remote debugging, check the Debugging checkbox in the DST tool.

**Note:** This is a powerful but dangerous tool. There is no locking mechanism to prevent concurrent editing. Use the Remote Object Debugger with caution, and only when you are sure that no other users are editing this image, or strange behavior may result.

With Debugging turned on, when an error occurs during request execution in that image, a notifier will appear, indicating that a remote error has occurred.
Note: If Debugging is not turned on and a remote error occurs, an exception handler will appear and the stack will already have been unwound to the context of the (client) sender of the request.

2 Click Debug to open Remote Object Debugger:

![Remote Object Debugger](image)

Each line in the stack is prefaced by either "local" or the name of the remote host, indicating the location of the error.

3 (Optional) Choose filter stack in the stack subview the pop-up menu, the system filters out RPC communications and other low-level support messages, leaving only application-relevant information.

Performance Tuning and Optimization

As with any Smalltalk development project, you can redesign and optimize applications you write in Distributed Smalltalk.

- Tune the application to run well on slower or less reliable networks.
- Make the user interface more efficient and easy to use.
- Optimize for a greater number of users sharing objects.
Network Performance

While the function and interaction with local and remote objects is the same, performance may vary.

Symptoms
- Acceptable performance during local and Local RPC activity but sluggish performance over the network.

Possible Causes
- Slow network.
- Non-optimized message-passing with remote objects.
- Non-optimized distribution of local and remote object responsibilities.

Solutions
- To help isolate whether the problem is with the network or the local application, make sure that Local RPC testing is off, since Local RPC testing will impact performance.
- Rethink the presentation/semantic split. In general, presentation objects run locally, while semantic objects may run anywhere on the network. Thus, behaviors and attributes that involve heavy user interaction should be assigned to presentation objects, while less volatile behaviors and attributes should be assigned to semantic objects.
- Group messages to reduce overhead. The network performance cost for any message is about the same regardless of its size. Thus, you can refine your application to eliminate needless message sends, and group small messages wherever possible.
- Improve hardware and network configurations or use faster transfer media.

User Interface Organization

Distributed applications have the potential of providing users a large quantity and variety of objects. One of the biggest challenges in creating usable distributed applications is to provide an information structure and interface that lets users find what they need easily and rapidly.

With Distributed Smalltalk, an interface to information can be structured hierarchically, or as a network, or as a hybrid of the two. By building on the range of options, you can tune accessing schemes to your users' needs.
Symptoms
- Users spend too much time looking for objects, or they get frustrated and give up.
- Usability testing shows that objects are duplicated unnecessarily because users cannot find what they really want.

Possible Causes
- Excessively deep burying of objects in many levels of containers.
- Excessively shallow organization of objects, all at the top level.
- Objects organized into containers by some criteria other than projects or related tasks, or by no criteria at all.

Solutions
- Use the appropriate mix of containers and links to simplify information access. If the number of objects is small, organize them into containers (such as folders and file cabinets), letting users browse to find what they need. If the volume of information is large, browsing becomes inefficient and time-consuming. Other policies, such as hierarchical search, may be used in conjunction with the containment model to let users locate information more quickly.

Coding Style Hints

As you develop applications using Distributed Smalltalk, you may wish to consider the following coding hints and guidelines.

Method Size
Since all remote calls require a certain minimum network overhead, you can reduce network traffic and optimize performance by grouping small methods into larger single calls.

Multiple Inheritance in DSTRepository
DSTRepository supports multiple inheritance for interfaces. Therefore, for corresponding Smalltalk classes, you may need to copy methods from more than one part of the class inheritance structure to achieve the effect of DSTRepository's multiple inheritance.

Blocks
The use of blocks in a distributed environment is different from their use in a local VisualWorks image. If you wish to pass blocks (by reference) across the network, you should define a corresponding interface in
Overview

In Distributed Smalltalk, runtime application creation is an extension of the VisualWorks runtime image-making process. This section assumes you are familiar with the VisualWorks process described in the VisualWorks Application Developer's Guide.

Design and Preparation

Before using the Image Maker to create a runtime application, there are several issues you may wish to resolve while you still have use of the full Distributed Smalltalk development environment.

Possible Runtime Configurations

Depending on the expected use of a runtime application, you can choose to create a single (ORB) runtime image, or multiple ORB runtime images. In runtime, as in development, an ORB image must be running on each system.

Each application is a separate ORB image. It is possible to configure a single ORB image to handle the naming service, the shared repository and security. This configured ORB runtime image can be reduced to an absolute minimum, so that it can run as a daemon process.
Runtime Request Broker Panel

Distributed Smalltalk provides a Request Broker panel that is used to start and stop the ORB in an image. You may include this default control panel in your application or runtime ORB image, or you may choose to modify the control panel, or even eliminate it if the ORB will be started programmatically.

If you want to start an ORB programmatically (without a control panel), do the following:

1. If the system is not initialized, initialize the system by executing:
   ```smalltalk
   ORBObject initializeORBAtHost: aHostname nodeId: aHostAddress.
   ```
   for information on the appropriate arguments for hostName and hostAddress, see the class IPSocketAddress.

2. Start the request broker by executing:
   ```smalltalk
   ORBDaemon startUpCoordinator startRequestBroker.
   ```

Note: A session is an object that keeps the transient state of your environment and lets you keep track of objects you are working on. It is re-initialized at system initialization.

Providing a Desktop Icon

If you wish to use an icon other than the default desktop icon, you must format, name and position it in ways that Distributed Smalltalk expects, as follows:

- **File format** must be XPM, Common Desktop Environment format, or Smalltalk store format (using Smalltalk’s internal screen capture fromUser:).

Classes XPixmapCompiler and XPixmapDescription translate XPM format to an internal Smalltalk image. See the class comments for more information.

- **File name** must be the same as the application's semantic object (for example ShapeSO).

If the class name is longer than allowed, the corresponding icon name must be truncated. (For example, some file systems have an 8-character limit.)

- **Path name** of the icon file must either be the default or you must make it explicit.
The icons provided with Distributed Smalltalk are in the subdirectories `/dst/icons/select` or `unselect`. These icons are loaded when an image is built, and are stored in DSTPresenter class variables IconSMap and IconUMap (selected and unselected versions).

To install an icon after the image is built, you must make its path explicit. To do this, you can use the DSTPresenter class method `readIcon:path:`.

**Modifying File and Directory Path Names**

If you develop on one platform and deliver on another, or if you deliver on the same platform but in a different directory tree, you should modify certain pathnames for your users. In addition to the standard VisualWorks files (`visual.sou` which is not needed for deployment, and the fonts directories), you may need to change the path names for the Distributed Smalltalk `icons` and `pixmaps` files. An easy way to make this change is to evaluate the following expression (substituting the correct path):

```
ORBObject installDir: 'pathname' asFilename
```

### Creating a Deployment ORB Image

#### Candidate Classes for Removal

<table>
<thead>
<tr>
<th>Category</th>
<th>Candidate for Removal</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA-Core</td>
<td>DSTPresenter</td>
<td>many dependencies in application presentations; remove with caution</td>
</tr>
<tr>
<td>CORBA-Contexts</td>
<td>(none)</td>
<td>removable if not using Request Broker panel interfaces</td>
</tr>
<tr>
<td>CORBA-MetaObjects</td>
<td>DSTMetaPO</td>
<td>many dependencies in application presentations; remove with caution</td>
</tr>
<tr>
<td>CORBA-Protocols-Core</td>
<td>(none)</td>
<td></td>
</tr>
<tr>
<td>CORBA-Protocols-NCS</td>
<td>(none)</td>
<td></td>
</tr>
<tr>
<td>CORBA-Protocol-IIOP</td>
<td>(none)</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Candidate for Removal</td>
<td>Comments</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CORBA-Compilers</td>
<td>all classes in category</td>
<td>remove IDLCompiler only if using shared repository</td>
</tr>
<tr>
<td>CORBA-Repository</td>
<td>DSTRepository (only class in category)</td>
<td>remove DSTRepository class only if using shared repository</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: the shared repository must be specified before the Image Maker is run.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unused interfaces can be removed if using local repository</td>
</tr>
<tr>
<td>CORBA-Tools</td>
<td>all classes in category except DSTClassFilter</td>
<td></td>
</tr>
<tr>
<td>CORBA-Debugging</td>
<td>all classes in category</td>
<td></td>
</tr>
<tr>
<td>COS-Events</td>
<td>all classes in category</td>
<td>many dependencies on these classes in the end-user environment; remove with caution</td>
</tr>
<tr>
<td>COS-Naming</td>
<td>(none)</td>
<td>do not remove!</td>
</tr>
<tr>
<td>COS-Lifecycle</td>
<td>(none)</td>
<td>do not remove!</td>
</tr>
<tr>
<td>COS-Concurrency</td>
<td>(none)</td>
<td></td>
</tr>
<tr>
<td>COS-Transactions</td>
<td>all classes in category</td>
<td>provides transaction service</td>
</tr>
<tr>
<td>DST-ObjectServices</td>
<td>all classes in category</td>
<td>many dependencies on these classes in the end-user environment; remove with caution</td>
</tr>
<tr>
<td>DST-AccessControl</td>
<td>all classes in category except DSTUserAcct, DSTUserDB, DSTPrincipal</td>
<td></td>
</tr>
<tr>
<td>DST-UserInterface</td>
<td>all classes in category</td>
<td></td>
</tr>
<tr>
<td>DST-Policy</td>
<td>all classes in category</td>
<td>many dependencies on these classes in the end-user environment; remove with caution</td>
</tr>
<tr>
<td>DST-UserServices</td>
<td>all classes in category</td>
<td></td>
</tr>
<tr>
<td>DST-Media</td>
<td>all classes in category</td>
<td>sample applications</td>
</tr>
<tr>
<td>DST-Collectors</td>
<td>all classes in category</td>
<td>sample applications</td>
</tr>
<tr>
<td>DST-Building</td>
<td>all classes in category</td>
<td></td>
</tr>
<tr>
<td>DST-Containers</td>
<td>all classes in category</td>
<td>sample applications</td>
</tr>
</tbody>
</table>
Steps for Creating a Deployment Image

1. Stop the ORB
2. Save your image.
   If you accidentally remove needed classes or have other problems during the deployment process, you will be able to start again from this saved image.
3. Open the Request Broker panel (or application).
   - If you want the runtime application to have a Request Broker panel, it should be open now (but not started).
   - If your application will not use a Request Broker panel, open the application.
4. Specify the shared interface repository, if appropriate.
   If the deployment image will use a shared interface repository, you must set this up before running the Image Maker. That is, once the ORB is set to use a shared interface repository, you can safely remove classes DSTRepository and IDLCompiler.

<table>
<thead>
<tr>
<th>Category</th>
<th>Candidate for Removal</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DST-Office</td>
<td>all classes in category</td>
<td></td>
</tr>
<tr>
<td>DST-Mapped Containers</td>
<td>all classes in category</td>
<td>sample applications</td>
</tr>
<tr>
<td>DST-AgentServices</td>
<td>all classes in category</td>
<td>required in default user interface; SessionContext may have other dependencies; many dependencies on these classes in the end-user environment — remove with caution</td>
</tr>
<tr>
<td>DST-Order Processing</td>
<td>all classes in category</td>
<td>sample applications</td>
</tr>
<tr>
<td>DST-Browser</td>
<td>all classes in category</td>
<td>provide services to standard applications</td>
</tr>
<tr>
<td>DST-Tools</td>
<td>all classes in category</td>
<td>development tools, not needed after deployment</td>
</tr>
<tr>
<td>DST-Testing</td>
<td>all classes in category</td>
<td></td>
</tr>
<tr>
<td>DST-Integration</td>
<td>all classes in category</td>
<td></td>
</tr>
<tr>
<td>DST-ORBLite-Examples</td>
<td>all classes in category</td>
<td></td>
</tr>
</tbody>
</table>
You could also leave a settings window up (and not remove class DSTSystemSettings) to allow shared repository configuration during runtime. However, any image with DSTRepository removed must be a shared repository.

5 Load and run the Image Maker, as described in the VisualWorks User's Guide. Keep in mind the following special situations:

- The agent (AgentPO and AgentSO) uses the Smalltalk compiler; do not remove compiler classes if you wish to use the agent.
- Your application will need a way to initialize and start the ORB, so either your application must do this, or you need to leave up an ORBControlPanel. If you leave up a window, you must not remove its associated class.
- Your application must already be configured before you run the Image Maker, or you must leave up a DSTSystemSettings window and not remove that class. Alternatively, your application must take responsibility for setting up the appropriate configuration.
- If DSTRepository is removed, the deployment image must use a shared repository. (See Step 4 on page 234).

Optimizing Runtime Applications

Exception Handling

Your application should catch, handle, and recover from errors that are associated with distributed systems, including: unreliable networks, shared objects, remote systems that become unavailable, and so on. For troubleshooting information, see Chapter 7, “Debugging and Tuning” starting on page 223.

That is, be sure to:

- Provide graceful handling of exceptions.
- Complete testing before creating the deployment image, and save the development image. (Debugging can be very difficult in a deployment image because development tools are no longer available.)
- Test the deployment image, to verify that no needed code was removed.
Chapter 18  - Creating a Deployment Image

- After deployment, you can use an Object Request Broker panel to monitor activity and start or stop the ORB on a remote image.

**Minimizing Footprint**

Frequently, a final runtime application needs to run on a platform with less memory and disk space, or a slower processor than your development platform. Some ideas for making a compact runtime application are:

- Remove all unnecessary classes during the image-making process.
- For the ORB image, remove all optional Distributed Smalltalk classes.
- Use the runtime Request Broker panel, or no Request Broker panel at all.
- Use a shared Interface Repository.
19
Troubleshooting

Overview
The most common issues that arise during development and operation are:

- Marshalling and unmarshalling errors
- Unavailable objects
- Synchronization errors
- Dangling references
- Interface access and editing errors
- Messages not understood
- Problems running multiple images
- Handling server-side transient errors

Marshalling and Unmarshalling Errors
Marshalling is the process of converting a Smalltalk message into a byte stream for transmission to a remote server. Unmarshalling creates a Smalltalk message from a marshalled byte stream. Marshalling and unmarshalling errors occur when the Interface Repository does not know how to deal with a given object.

Symptoms
- The application runs locally but not remotely.
- With Local RPC Testing turned on, or when running a distributed application, a marshallError or unmarshalError error notifier appears.
Possible Causes

- An object’s interface has not been registered with the Interface Repository.
- The object’s interface has changed since it was registered (repositories out of sync).
- A parameter or result value is inconsistent with the method’s operation declaration.
- A legal Smalltalk construct may be illegal in IDL (IDL typing is more constrained than Smalltalk allows).

Solutions

- Use the IR Browser to check the object’s interface.
- Edit the appropriate module in DSTRepository.
- Change the interface declaration in DSTRepository or change the value in the Smalltalk code.

Object Availability Exceptions

Since remote objects live in images over which the local image has no control, it is possible for remote objects to become unavailable. That is, when a local object sends a message to a remote object’s surrogate, the surrogate tries to pass the message to the remote object but gets no response.

Symptoms

- The following exceptions are raised when a message is sent to an unavailable object:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>commFailureError</td>
<td>Communication failure (the remote ORB is probably down)</td>
</tr>
<tr>
<td>invObjrefError:</td>
<td>Invalid object reference (most likely: the object no longer exists in the remote image)</td>
</tr>
<tr>
<td>noPermissionError:</td>
<td>No permission for attempted operation</td>
</tr>
<tr>
<td>noImplementError:</td>
<td>Operation implementation unavailable</td>
</tr>
<tr>
<td>noResponseError:</td>
<td>Response to request not yet available</td>
</tr>
<tr>
<td>transientError:</td>
<td>Transient failure—reissue request</td>
</tr>
</tbody>
</table>
Synchronization Problems

Possible Causes
- Sockets, network, or other communications services are busy or unavailable.
- Remote image is closed or the system it runs on is unavailable (it may be turned off, or off the network).
- Remote ORB is stopped.
- Remote garbage collect reclaimed the object.

Solutions
- Wait for other ORB to be started, or for its image to come back on line.
- Strengthen the link. For a discussion of the different kinds of links, see “Links” on page 79.
- If the problem is frequent, consider refining the application design to make the remote object local.
- When a communications failure occurs, in a Developer’s system, you can debug or proceed, however in a Runtime system there is no option to debug. If you wish to let users attempt to retry after a communication failure, you can set class DSTObjRef’s retrying flag to true (by default, it is set to false).
- For transient errors, consider using the transient error handler (see “Handling Server-side Transient Errors” below)

Synchronization Problems

Synchronization problems are most likely to occur when multiple users share objects.

Symptoms
- Unreliable behavior.
- Non-deterministic errors.

Possible Causes
- Users’ changes to a shared object conflict.
- Stale images access shared objects.
Solutions

- Use resource management to lock out multiple concurrent edits of shared resources. (Resource management is implemented in class DSTResourceManager and used by DSTSemantic.)
- Use semaphores to serialize critical parts of an application.
- Refine class DSTTraversal to provide tighter object locking and transaction control.
- Use an external database’s object locking facilities.

Dangling References

Dangling references can occur when a method refers to an obsolete interface in the Interface Repository.

Symptoms

- “Dangling references” notifier.

Possible Causes

- A thread of control or context holds a reference to an interface object that has been removed from the Interface Repository. For example, you have open instances of an application that rely on an interface that you changed.

Solutions

- Close the offending application instances.

Remote Access to Overridden Methods

Objects that override methods in class Object cannot make these methods remotely available since the surrogates inherit from Object.

Symptoms

- You expect a remote message response but get a local one.

Possible Causes

- Instances of DSTObjRef and its subclasses function as surrogates for remote objects. Messages sent to instances of DSTObjRef that are not implemented in DSTObjRef or its superclass Object are assumed to be implemented in the remote object. However, if a method is defined both in the local class Object and the remote object that the DSTObjRef
Interface Repository Accessing Errors

instance refers to, by default, Object's method will respond to the call; this is probably not the behavior you intended.

Solutions

- Define a method in DSTObjRef's message category override inheritance that overrides class Object's method of the same name. (See the other methods defined here for examples of the method definition.)

For example, in order to have the message broadcast: aDSTObjRef call the remote object's broadcast: method, you must define broadcast: in DSTObjRef's override inheritance message category. Define it as:

```
broadcast: aSymbol
  "pass this operation to the referenced object"
  ^self perform: #broadcast on: (Array with: aSymbol).
```

Interface Repository Accessing Errors

The Interface Repository Browser uses resource management to protect against concurrent editing of the same interfaces and of interfaces that are in use. Thus, the system can deny your attempts to edit interfaces.

(Since the System Browser does not protect against concurrent edits, it can be dangerous place to edit an interface repository.)

Symptoms

- When using the Interface Repository Browser, if you choose Edit menu: Definition, an error message appears and the view does not change (that is, it remains a read-only iconic or text view).

Possible Causes

You are trying to edit:

- Fundamental interfaces that the IR Browser uses to present information (for example, the PSSplit module),

- An interface repository that someone else is currently editing, or

- An interface repository that someone was browsing when their system crashed (this problem does not occur when the ORB stops normally).
Solutions

- Use the View menu as text option to get read-only access.
- If locks should be released but were not because the system from which the interface repository was being edited crashed, you must reinitialize the repository. To do this, stop the ORB, then in the Distributed Smalltalk main window, select DST → Initialize → Initialize Repository.

Interface Incompatibilities

Objects in different images communicate via their interfaces which are stored in each ORB’s interface repository. If the interfaces are not identical (or of compatible versions), you may see a badOperationError: or some other error.

Symptoms

- Errors (such as badOperationError:) occur during distributed execution that do not occur during Local RPC testing.

Possible Causes

- Interface definitions in the participating interface repositories are out of sync.
- Interface version numbers are out of sync. (The server interface’s version number must be greater than or equal to the client’s version.)
- If you are using a shared interface repository, the interfaces in the master repository may have changed since the local copy was cached.

Solutions

- If you are using a shared repository, stop the ORB, then in Distributed Smalltalk main window, select DST → Initialize → Initialize Repository.
- If you are using interface versioning, be sure that you use it correctly.
Other Exceptions

To support CORBA and object distribution, Distributed Smalltalk provides mechanisms for remote objects to send messages via surrogates and to access interfaces in the Interface Repository. As a result, there is an extra layer of complexity in pinpointing an error when a message to a distributed object is not understood.

Symptoms
- The following exceptions are raised when a message sent to a remote object is not understood:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>badInvOrderError</td>
<td>Routine invocations out of order</td>
</tr>
<tr>
<td>badOperationError</td>
<td>Invalid operation, or interfaces out of sync</td>
</tr>
<tr>
<td>badParamError</td>
<td>An invalid parameter was passed</td>
</tr>
<tr>
<td>badTypecodeError</td>
<td>Bad typecode</td>
</tr>
<tr>
<td>contextError</td>
<td>Error processing context object</td>
</tr>
<tr>
<td>dataConversionError</td>
<td>Data conversion error</td>
</tr>
<tr>
<td>intfReposError</td>
<td>Error accessing interface repository</td>
</tr>
<tr>
<td>invFlagError</td>
<td>Invalid flag was specified</td>
</tr>
<tr>
<td>invIdentError</td>
<td>Invalid identifier syntax</td>
</tr>
<tr>
<td>NotFound</td>
<td>Object not found (part of IDL interface)</td>
</tr>
<tr>
<td>SemanticError</td>
<td>(part of IDL interface)</td>
</tr>
<tr>
<td>UnknownID</td>
<td>Unknown object UUID (part of IDL interface)</td>
</tr>
</tbody>
</table>

Possible Causes
- Interface Repositories in the communicating images are out of sync. (The Interface Repository in each image is distinct; if you change one you must also change the others.)
- The message sent is not defined for the remote object.

Solutions
- Use the debugger to navigate to the remote image, where you can determine whether the message being sent is actually implemented.
Problems Running Multiple Images

**Cannot Start an ORB**
- If an ORB already running on a system, you cannot start another.
- If you cannot start the ORB because there is no Request Broker panel, execute the statement DSTControlPanel open to reopen an Request Broker panel.

**Cannot Determine Which Image You Are Using**
- Use the Office’s Action menu choice Open Building. Each office open on the current system will be shown here. Or, open the Request Broker panel’s Monitor to see active conversations; conversations within an image are local, conversations between ORBs are considered remote.

Handling Server-side Transient Errors

DST now has a customizable TransientErrorHandler for dealing with server side transient errors. To customize the handler, review the method comment of ORBDaemon class>>transientErrorHandler:.

Transients errors are usually raised when clients and servers are both overloaded, by the client spawning requests and the server receiving them. If the server is swamped with incoming requests, it spends all of its time spawning processes to respond to them, and those spawned processes never get a window in which to run. This can further entail that the requests will time out on the client side. When the server gets time to run its spawned, responding processes, and has return values to pass back, the client is too busy doing other things to listen, and this produces a transient error.

Transients are difficult to trap in user code, because they are generated on the server side, in a spawned process, created to respond to a remote request, initiated in another image. Because an unhandled transient will try to open a notifier, a user producing a headless server wants to trap them.

Before you decide to muffle notification of transients, using a handler block that throws them away, you really should find out exactly why and how the transients are occuring.
A

IDL Lexical Conventions

Overview

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

File Processing

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 (that is, a file after preprocessing), is called a translation unit.

Comparison With C++ Lexical Conventions

IDL obeys the same lexical rules as C++, although new keywords are introduced to support distribution concepts. It also provides full support for standard C++ preprocessing features. The IDL specification is expected to track relevant changes to C++ introduced by the ANSI standardization effort.

Character Set

IDL uses the ISO Latin-1 (8859.1) character set. This character set is divided into alphabetic characters (letters), digits, graphic characters, the space (blank) character and formatting characters. The table below shows the IDL alphabetic characters; upper- and lower-case equivalencies are paired.
### Alphabetic Characters (Letter)

<table>
<thead>
<tr>
<th>Char.</th>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aa</td>
<td>Upper/Lower-case A</td>
<td>Àà Upper/Lower-case A with grave accent</td>
</tr>
<tr>
<td>Bb</td>
<td>Upper/Lower-case B</td>
<td>Áá Upper/Lower-case A with acute accent</td>
</tr>
<tr>
<td>Cc</td>
<td>Upper/Lower-case C</td>
<td>Ââ Upper/Lower-case A with circumflex accent</td>
</tr>
<tr>
<td>Dd</td>
<td>Upper/Lower-case D</td>
<td>Ää Upper/Lower-case A with tildes</td>
</tr>
<tr>
<td>Ee</td>
<td>Upper/Lower-case E</td>
<td>Ææ Upper/Lower-case A with diaeresis</td>
</tr>
<tr>
<td>Ff</td>
<td>Upper/Lower-case F</td>
<td>Ââ Upper/Lower-case A with ring above</td>
</tr>
<tr>
<td>Gg</td>
<td>Upper/Lower-case G</td>
<td>Ææ Upper/Lower-case A with diphthong A with E</td>
</tr>
<tr>
<td>Hh</td>
<td>Upper/Lower-case H</td>
<td>Çç Upper/Lower-case A with cedilla</td>
</tr>
<tr>
<td>Ii</td>
<td>Upper/Lower-case I</td>
<td>Èè Upper/Lower-case A with grave accent</td>
</tr>
<tr>
<td>Jj</td>
<td>Upper/Lower-case J</td>
<td>Éé Upper/Lower-case A with acute accent</td>
</tr>
<tr>
<td>Kk</td>
<td>Upper/Lower-case K</td>
<td>Êê Upper/Lower-case A with circumflex accent</td>
</tr>
<tr>
<td>Ll</td>
<td>Upper/Lower-case L</td>
<td>Ëë Upper/Lower-case A with diaeresis</td>
</tr>
<tr>
<td>Mm</td>
<td>Upper/Lower-case M</td>
<td>Ìì Upper/Lower-case A with grave accent</td>
</tr>
<tr>
<td>Nn</td>
<td>Upper/Lower-case N</td>
<td>Íí Upper/Lower-case A with acute accent</td>
</tr>
<tr>
<td>Oo</td>
<td>Upper/Lower-case O</td>
<td>Ìì Upper/Lower-case A with circumflex accent</td>
</tr>
<tr>
<td>Pp</td>
<td>Upper/Lower-case P</td>
<td>Ìì Upper/Lower-case A with diaeresis</td>
</tr>
<tr>
<td>Qq</td>
<td>Upper/Lower-case Q</td>
<td>Ìì Upper/Lower-case A with Icelandic eth</td>
</tr>
<tr>
<td>Rr</td>
<td>Upper/Lower-case R</td>
<td>Ìì Upper/Lower-case A with tildes</td>
</tr>
<tr>
<td>Char.</td>
<td>Description</td>
<td>Char.</td>
</tr>
<tr>
<td>-------</td>
<td>------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Ss</td>
<td>Upper/Lower-case S</td>
<td>Œœ</td>
</tr>
<tr>
<td>Tt</td>
<td>Upper/Lower-case T</td>
<td>Œó</td>
</tr>
<tr>
<td>Uu</td>
<td>Upper/Lower-case U</td>
<td>Œò</td>
</tr>
<tr>
<td>Vv</td>
<td>Upper/Lower-case V</td>
<td>Œò</td>
</tr>
<tr>
<td>Ww</td>
<td>Upper/Lower-case W</td>
<td>Œò</td>
</tr>
<tr>
<td>Xx</td>
<td>Upper/Lower-case X</td>
<td>Øœ</td>
</tr>
<tr>
<td>Yy</td>
<td>Upper/Lower-case Y</td>
<td>Ūù</td>
</tr>
<tr>
<td>Zz</td>
<td>Upper/Lower-case Z</td>
<td>Ūû</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ūû</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ūû</td>
</tr>
<tr>
<td>††</td>
<td>Upper/Lower-case Y with acute accent</td>
<td></td>
</tr>
<tr>
<td>††</td>
<td>Upper/Lower-case Icelandic thorn</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>Lower-case German sharp S</td>
<td></td>
</tr>
<tr>
<td>ſy</td>
<td>Lower-case Y with diaeresis</td>
<td></td>
</tr>
</tbody>
</table>

†† denotes character unprintable in this document
# Chapter A - IDL Lexical Conventions

## Decimal Digit Characters

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

## Graphic Characters

<table>
<thead>
<tr>
<th>Char.</th>
<th>Description</th>
<th>Char.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>exclamation point</td>
<td>i</td>
<td>inverted exclamation mark</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote</td>
<td>£</td>
<td>pound sign</td>
</tr>
<tr>
<td>#</td>
<td>number sign</td>
<td>%</td>
<td>percent sign</td>
</tr>
<tr>
<td>$</td>
<td>dollar sign</td>
<td>&amp;</td>
<td>ampersand</td>
</tr>
<tr>
<td>%</td>
<td>percent sign</td>
<td>′</td>
<td>apostrophe</td>
</tr>
<tr>
<td>(</td>
<td>left parenthesis</td>
<td>)</td>
<td>right parenthesis</td>
</tr>
<tr>
<td>*</td>
<td>asterisk</td>
<td>†</td>
<td>broken bar</td>
</tr>
<tr>
<td>+</td>
<td>plus sign</td>
<td>¬</td>
<td>not sign</td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
<td>†</td>
<td>soft hyphen</td>
</tr>
<tr>
<td>:</td>
<td>colon</td>
<td>®</td>
<td>registered trade mark sign</td>
</tr>
<tr>
<td>;</td>
<td>semicolon</td>
<td>±</td>
<td>plus-minus sign</td>
</tr>
<tr>
<td>&lt;</td>
<td>less-than sign</td>
<td>2</td>
<td>superscript two</td>
</tr>
<tr>
<td>=</td>
<td>equals sign</td>
<td>3</td>
<td>superscript three</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater-than sign</td>
<td>′</td>
<td>acute</td>
</tr>
<tr>
<td>?</td>
<td>question mark</td>
<td>m</td>
<td>micro</td>
</tr>
<tr>
<td>@</td>
<td>commercial at</td>
<td>¶</td>
<td>pilcrow</td>
</tr>
<tr>
<td>[</td>
<td>left square bracket</td>
<td>•</td>
<td>middle dot</td>
</tr>
<tr>
<td>\</td>
<td>reverse solidus</td>
<td>,</td>
<td>cedilla</td>
</tr>
<tr>
<td>]</td>
<td>right square bracket</td>
<td>1</td>
<td>superscript one</td>
</tr>
</tbody>
</table>
Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, 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.

If the input stream has been parsed into tokens up to a given character, the next token is taken to be the longest string of characters that could possibly constitute a token.

Graphic Characters (Continued)

<table>
<thead>
<tr>
<th>Char.</th>
<th>Description</th>
<th>Char.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>circumflex</td>
<td>†</td>
<td>masculine ordinal indicator</td>
</tr>
<tr>
<td>_</td>
<td>low line, underscore</td>
<td>»</td>
<td>right angle quotation mark</td>
</tr>
<tr>
<td>`</td>
<td>grave</td>
<td>†</td>
<td>vulgar fraction 1/4</td>
</tr>
<tr>
<td>{</td>
<td>left curly bracket</td>
<td>†</td>
<td>vulgar fraction 1/2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vertical line</td>
<td>†</td>
</tr>
<tr>
<td>}</td>
<td>right curly bracket</td>
<td>ü</td>
<td>inverted question mark</td>
</tr>
<tr>
<td>~</td>
<td>tilde</td>
<td>x</td>
<td>multiplication sign</td>
</tr>
<tr>
<td>÷</td>
<td>division sign</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

† denotes character unprintable in this document.

The Formatting Characters

<table>
<thead>
<tr>
<th>Description</th>
<th>Abbreviation</th>
<th>ISO 646 Octal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>alert</td>
<td>BEL</td>
<td>007</td>
</tr>
<tr>
<td>backspace</td>
<td>BS</td>
<td>010</td>
</tr>
<tr>
<td>horizontal tab</td>
<td>HT</td>
<td>011</td>
</tr>
<tr>
<td>newline</td>
<td>NL, LF</td>
<td>012</td>
</tr>
<tr>
<td>vertical tab</td>
<td>VT</td>
<td>013</td>
</tr>
<tr>
<td>form feed</td>
<td>FF</td>
<td>014</td>
</tr>
<tr>
<td>carriage return</td>
<td>CR</td>
<td>015</td>
</tr>
</tbody>
</table>
Comments

The characters /* start a comment, which terminates with the characters */. These comments do not nest. The characters // start a comment, which terminates at the end of the line on which they occur. The comment characters //, /*, and */ have no special meaning within a // comment and are treated just like other characters. Similarly, the comment characters // and /* have no special meaning within a /* comment. Comments may contain alphabetic, digit, graphic, space, horizontal tab, vertical tab, form feed, and newline characters.

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:

- Upper- and lower-case letters are treated as the same letter. Table defines the equivalence mapping of upper- and lower-case letters.

- The comparison does not take into account equivalences between digraphs and pairs of letters (e.g., “æ” and “ae” are not considered equivalent) or equivalences between accented and non-accented letters (e.g., “Å” and “A” 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.
The identifiers listed in the following table are reserved for use as keywords, and may not be used otherwise.

Reserved Keywords

<table>
<thead>
<tr>
<th>any</th>
<th>default</th>
<th>interface</th>
<th>readonly</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>double</td>
<td>long</td>
<td>sequence</td>
<td>union</td>
</tr>
<tr>
<td>boolean</td>
<td>enum</td>
<td>module</td>
<td>short</td>
<td>void</td>
</tr>
<tr>
<td>case</td>
<td>exception</td>
<td>octet</td>
<td>string</td>
<td>FALSE</td>
</tr>
<tr>
<td>char</td>
<td>float</td>
<td>oneway</td>
<td>struct</td>
<td>Object</td>
</tr>
<tr>
<td>const</td>
<td>in</td>
<td>out</td>
<td>switch</td>
<td>TRUE</td>
</tr>
<tr>
<td>context</td>
<td>inout</td>
<td>raises</td>
<td>typedef</td>
<td></td>
</tr>
</tbody>
</table>

Keywords obey the rules for identifiers (see “Identifiers” on page 250) 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 characters shown in the following table as punctuation.

Punctuation Characters

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

In addition, the tokens listed in the following table are used by the preprocessor.

Preprocessor Tokens

| # | ## | ! | || | & |
Literals

This section describes literals—integer, character, and floating point constants and string literals.

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.

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 (see Table, “Alphabetic Characters (Letter),” on page 246, Table, “Decimal Digit Characters,” on page 248, and Table, “Graphic Characters,” on page 248). 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 (see Table, “IDL EBNF Format,” on page 255). The meaning of all other characters is implementation-dependent.

Non-graphic characters must be represented using escape sequences as defined below in the following table. Note that escape sequences must be used to represent single quote and backslash characters in character literals.
**Escape Sequences**

<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>
<tr>
<td>carriage return</td>
<td>\r</td>
</tr>
<tr>
<td>form feed</td>
<td>\f</td>
</tr>
<tr>
<td>alert</td>
<td>\a</td>
</tr>
<tr>
<td>backslash</td>
<td>\</td>
</tr>
<tr>
<td>question mark</td>
<td>?</td>
</tr>
<tr>
<td>single quote</td>
<td>'</td>
</tr>
<tr>
<td>double quote</td>
<td>&quot;</td>
</tr>
<tr>
<td>octal number</td>
<td>\ooo</td>
</tr>
<tr>
<td>hexadecimal number</td>
<td>\xhh</td>
</tr>
</tbody>
</table>

If the character following a backslash is not one of those specified, the behavior is undefined. An escape sequence specifies a single character.

The escape \ooo consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape \xhh consists of the backslash followed by x followed by one or two hexadecimal digits that are taken to specify the value of the desired character. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character constant is implementation dependent if it exceeds that of the largest char.

**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.
String Literals

A string literal is a sequence of characters (as defined in “Character Literals,” earlier in this chapter) 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".
The description of IDL grammar uses a syntax notation that is similar to Extended Backus-Naur format (EBNF).

The following table lists the symbols used in this format and their meanings.

### IDL EBNF Format

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
<td>Is defined as</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>
The following is 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>` }*
(11) \texttt{<scoped\_name>} ::= \texttt{<identifier>}
| \texttt{::} \texttt{<identifier>}
| \texttt{<scoped\_name> :: \texttt{<identifier>}}

(12) \texttt{<const\_dcl>} \texttt{``const'' \texttt{<const\_type> \texttt{<identifier>}}}
| \texttt{``='' \texttt{<const\_exp>}}

(13) \texttt{<const\_type>} ::= \texttt{<integer\_type>}
| \texttt{<char\_type>}
| \texttt{<boolean\_type>}
| \texttt{<floating\_pt\_type>}
| \texttt{<string\_type>}
| \texttt{<scoped\_name>}

(14) \texttt{<const\_exp>} ::= \texttt{<or\_expr>}

(15) \texttt{<or\_expr>} ::= \texttt{<xor\_expr>}
| \texttt{<or\_expr> | <xor\_expr>}

(16) \texttt{<xor\_expr>} ::= \texttt{<and\_expr>}
| \texttt{<xor\_expr> ^ <and\_expr>}

(17) \texttt{<and\_expr>} ::= \texttt{<shift\_expr>}
| \texttt{<and\_expr> & <shift\_expr>}

(18) \texttt{<shift\_expr>} ::= \texttt{<add\_expr>}
| \texttt{<shift\_expr> >> <add\_expr>}
| \texttt{<shift\_expr> << <add\_expr>}

\textit{IDL Grammar}
(19) <add_expr> ::= <mult_expr>
    | <add_expr> "+" <mult_expr>
    | <add_expr> "-" <mult_expr>

(20) <mult_expr> ::= <unary_expr,>
    | <mult_expr> "*" <unary_expr>
    | <mult_expr> "~" cunary_expr>
    | <mult_expr> "%" cunary_expr>

(21) <unary_expr> ::= <unary_operator> cprimary_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) \(<\text{positive\_int\_const}>\) ::= \(<\text{const\_exp}>\)

(27) \(<\text{type\_dcl}>\) ::= “typedef” \(<\text{type\_declarator}>\)
    | \(<\text{struct\_type}>\)
    | \(<\text{union\_type}>\)
    | \(<\text{enum\_type}>\)

(28) \(<\text{type\_declarator}>\) ::= \(<\text{type\_spec}>\) \(<\text{declarators}>\)

(29) \(<\text{type\_spec}>\) ::= \(<\text{simple\_type\_spec}>\)
    | \(<\text{constr\_type\_spec}>\)

(30) \(<\text{simple\_type\_spec}>\) ::= \(<\text{base\_type\_spec}>\)
    | \(<\text{template\_type\_spec}>\)
    | \(<\text{scoped\_name}>\)

(31) \(<\text{base\_type\_spec}>\) ::= \(<\text{floating\_pt\_type}>\)
    | \(<\text{integer\_type}>\)
    | \(<\text{char\_type}>\)
    | \(<\text{boolean\_type}>\)
    | \(<\text{octet\_type}>\)
    | \(<\text{any\_type}>\)

(32) \(<\text{template\_type\_spec}>\) ::= \(<\text{sequence\_type}>\)
    | \(<\text{string\_type}>\)
(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>`
IDL Grammar

(44) \(<\text{unsigned\_long\_int}>\) ::= \"unsigned\" \"long\"

(45) \(<\text{unsigned\_short\_int}>\) ::= \"unsigned\" \"short\"

(46) \(<\text{char\_type}>\) ::= \"char\"

(47) \(<\text{boolean\_type}>\) ::= \"boolean\"

(48) \(<\text{octet\_type}>\) ::= \"octet\"

(49) \(<\text{any\_type}>\) ::= \"any\"

(50) \(<\text{struct\_type}>\) ::= \"struct\" <\text{identifier}> { <\text{member\_list}> }\"

(51) \(<\text{member\_list}>\) ::= <\text{member}>+

(52) \(<\text{member}>\) ::= <\text{type\_spec}> <\text{declarators}> \";\"

(53) \(<\text{union\_type}>\) ::= \"union\" <\text{identifier}> \"switch\" ( ( <\text{switch\_type\_spec}> )

(54) \(<\text{switch\_type\_spec}>\) ::= <\text{integer\_type}>

| <\text{char\_type}>
| <\text{boolean\_type}>
| <\text{enum\_type}>
| <\text{scoped\_name}>

(55) \(<\text{switch\_body}>\) ::= <\text{case}>+
(56) \textit{<case>} ::= \textit{<case\_label>}+ \textit{<element\_spec>} ";"

(57) \textit{<case\_label>} ::= "case" \textit{<const\_exp>} ";"
| "default" ";"

(58) \textit{<element\_spec>} ::= \textit{<type\_spec>} \textit{<declarator>}

(59) \textit{<enum\_type>} ::= "enum" \textit{<identifier>} "{" \textit{<enumerator> '{','<enumerator>}*"}"

(60) \textit{<enumerator>} ::= \textit{<identifier>}

(61) \textit{<sequence\_type>} ::= "sequence" "<" \textit{<simple\_type\_spec>} "">
| "sequence" "<" \textit{<simple\_type\_spec>} ">

(62) \textit{<string\_type>} ::= "string" "<" \textit{<positive\_int\_const>} "">
| "string"

(63) \textit{<array\_declarator>} ::= \textit{<identifier>} \textit{<fixed\_array\_size> +}

(64) \textit{<fixed\_array\_size>} ::= "[" \textit{<positive\_int\_const>} "]"

(65) \textit{<attr\_dcl>} ::= [ "readonly" ] "attribute"
\textit{<param\_type\_spec>} \textit{<simple\_declarator>}
\textit{<declarators>}{"," \textit{<simple\_declarator>}*}

(66) \textit{<except\_dcl>} ::= "exception" \textit{<identifier> "{" \textit{<member>"}"}"}
(67) \( \text{op_dcl} \) ::=  
\[ \text{<op_attribute>} \text{I} \text{<op_type_spec>} \text{<identifier>} \text{<parameter_dcls>} \] 
\[ \text{[ <raises_expr> ] [ <context_expr> ]} \]

(68) \( \text{<op_attribute>} \) ::=  "oneway"

(69) \( \text{<op_type_spec>} \) ::=  \( \text{<simple_type_spec>} \)  
\|  "void"

(70) \( \text{<parameter_dcls>} \) ::=  "(" \text{<param_dcl>} \{ "," \text{<param_dcl>} \} *)"  
\|  "(" ")"

(71) \( \text{<param_dcl>} \) ::=  \( \text{<param_attribute>} \text{<simple_type_spec>} \text{<declarator>} \)

(72) \( \text{<param_attribute>} \) ::=  "in"  
\|  "out"  
\|  "inout"

(73) \( \text{<raises_expr>} \) ::=  "raises" "(" \text{<scoped_name>} \{ "," \text{<scoped_name>}\} *)"

(74) \( \text{<context_expr>} \) ::=  "context" "(" \text{<string_literal>} \{ "," \text{<string_literal>}\} ")"

(75) \( \text{<param_type_spec>} \) ::=  \( \text{<base_type_spec>} \)  
\|  \( \text{<string_type>} \)  
\|  \( \text{<scoped_name>} \)
Bibliography

**CORBA Resources**


An overview of CORBA, written in a chatty non-technical style.

Jon Siegel, *CORBA Fundamentals and Programming*


An excellent guide to developing CORBA applications. Gives a complete example in Smalltalk.

**Distributed Computing Resources**

Nancy A. Lynch, *Distributed Algorithms*


A survey of what is known about distributed algorithms, written for computer scientists who are familiar with the analysis of algorithms. Covers issues including consensus, communication, resource allocation, and synchronization.

Sape Mullender (editor), *Distributed Systems*


An overview of the design and implementation of distributed systems, aimed at graduate students in computer science.
M. Tamer Özsu, Umeshwar Dayal, and Patrick Valduriez, 
*Distributed Object Management*

A collection of papers from the International Workshop on Distributed Object Management. Addresses distributed database issues.
Index

Symbols
- 78
#include 105
#pragma 105
* 78
+ 78
: 100
:: 101, 106, 107
<< 78
<Operate> button 17
<Select> button 17
<Window> button 17
>> 78
^ 79
| 79
/ 78
‰ 78
″ 78
A
abstractClassId 144
abstractClassID method example 46
access control for an IDL operation 141
Access pragma 141
ACID
Atomicity, Consistency, Isolation, Durability 200
Action menu (IR Browser) 152
Open Repository 152
active 204
Alphabetic Character Set 245
ANSI C++ preprocessor 153
any 82
CORBAType 124
any type 116
any CLASS Pragma 124
application
creating 59
creation process 60
initialization 155
preparing for deployment 230
application, runs locally but not remotely 237
Array
Multi-Dimensional 81
One-Dimensional 81, 83
Smalltalk Mapping 128
array type 116
as Picture (IR Browser View menu) 152
as Text (IR Browser View menu) 152
asCORBAParameter 114
asIDLDefinition message 146
Atomicity
ACID 200
attribute 99
Attributes 99
Declaring 99
Smalltalk Mapping 119
attributes 115
readonly 119
B
Basic Data Types 82
Smalltalk Mapping 121
begin 204
bind 163
binding
using naming service 41
blocks (good usage) 229
Boolean 121
boolean 77, 82
boolean type 116
buttons mouse 17
C
channel (event) 167
administration 168
creation 168
multiple 168
char 77, 82
Character 121
Character Literals 252
Characters
Alphabetic 245
Punctuation 251
child transaction 210
CLASS pragma 124
Class pragma
Pragma class 140
client 52
client request 52
Comments 250
commit 204
committed 204
Common Object Request Broker
Architecture (CORBA) 50
implemented in Distributed Smalltalk 53
communication failures 239
communication resources 59
completion 204
completion_status 97
concurrency 50
enumerations 195
interfaces 196
locks 195
LockSets 195
Configuration menu
Shared Repository 36
Consistency
ACID 200
const 76
constant_expression 77
Constants
Declaring 76
Smalltalk Mapping 120
constants 115
Constructed Data Types 85, 86, 88
Constructed Types 85
corbaName method
event notification 166
core IFs 255
COS
lifecycle 182
naming 160
COS-Naming 162
COS-Transactions 217
core 183
create 183
createservice method 40
creating
an application 59, 60
creating names 162
Current 213
D
Data Types 82
Basic 82
Constructed 85
discriminated union 88
enum 86
struct 85
Declaring 80
Naming 81
Simple 82
Template 82
sequence 83
string 83
data types, IDL
base (char, octet, float, double, long
type integer, short integer, boolean, any)
121
boolean 121
corbaName 46
corbaName method
char (character) 121
double 121
float 121
integer 121
octet 121
sequence 127
string 128
data types, IDL
debugging 237
distributed 226
debugging (Request Broker panel option) 223
declarations 99
declaring Constants 76
declaring Data Types 80
declaring Exceptions 95
declaring Interfaces 73
Index

Declaring Modules 72
Declaring Operations 91
  arguments 92
  context 94
  in Parameter Attribute 92
  inout Parameter Attribute 92
  oneway Attribute 91, 93
  out Parameter Attribute 92
  raises 93
  void 91
deep copy 183
deepCopyWith 183
delete 182
  see destroy 184
deployment
  preparation 230
derived interface 101
descendants 211
designing distributed applications 62–64
destroy 184
development process
  distributed applications 60
  Distributed Smalltalk 58
Dictionary 129
  struct 125
DII, see Dynamic Invocation Interface
direct base interface 101
direct context management 204
disconnect (event notification) 169
Discriminated Unions 88
distributed applications
  designing 62–64
  development process 60
  performance 63
  performance hints 63
distributed computing concepts 59
distributed objects
  inherently shared 62
Distributed Smalltalk
  development process 58
doesNotUnderstand 55
Double 121
double 82
dropEditLock 154
DSTAttribute 143
DSTConstant 142
DSTException 130, 142
DSTFactoryFinder 190
DSTIDLTool open 148
DSTLockSetCoordinator 195
DSTModuleRepository class 142, 145
DSTObject 55, 142
DSTObjectLocal 56
DSTObjectWidened 56
DSTOperation 142
DSTParameter 142
DSTPullConsumer 171
DSTPushConsumer 171
DSTRecoverableObject 202, 213
DSTRepository 54
  IDL in 142
  interface 117
  multiple inheritance 229
DSTRepository class 142, 145
DSTSAMPLEComputeService example 38–39
DSTSAMPLERecoverableObject 222
DSTTransaction 215
DSTTransaction class>>create 213
DSTTransactionLockSet 214
DSTTransactionObject 208, 213, 214
DSTTypeBase 142
DSTTypeConstr 142
DSTTypeEnum 195
DSTTypeTemplate 142
Durability
  ACID 200
Dynamic Invocation Interface (DII) 51
E
edit lock 154
Edit menu
  (IR Browser) 152
electronic mail 18, 20
enum 86
enumeration types 116
Enumerations 86
enumerations 195
errors
  badInvOrderError: 243
  badOperationError: 242, 243
  badParamError: 243
  badTypecodeError: 243
  cannot start ORB 244
  commFailureError: 238
  contextError: 243
  dataConversionError: 243
  intfReposError: 243
  invFlagError: 243
  invIdentError: 243
  invObjrefError: 238
  marshalling and unmarshalling 237
  noImplementError: 238
  noPermissionError: 238
Index

noResponseError: 238
NotFoundException 243
objects unavailable 238
remote image or ORB is stopped 239
SemanticError 243
socket busy 239
synchronization 239
transientError: 238
unknown image 244
UnknownID 243

Escape Sequences 252
event channel 167
administration 168
creation 168
multiple 168

event consumer 167
event notification 28, 166
connecting a pull consumer to an event channel 172
connecting a push consumer to a channel 172
connecting a push supplier to a channel 173
connecting to a channel 174
disconnect 169
implementation 181
implementing a typed pull connection 176
implementing a typed push connection 174
push and pull 168
quality of service 173, 178
testing the event example 173
typed events 173

event supplier 167
examples
  abstractClassId method 46
corbaName method 46
DSTSampleComputeService 38–??

Exception
Smalltalk Mapping 129
exception 95
Exception handling 132
exception handling 223, 237
in runtime applications 235
exception type 116
Exceptions 95, 96
Declaring 95
Standard 96
User-defined 95
exceptions, IDL 129

explicit interface
  advantages 58
  defined 58
  externalize 184

F
Factory 213, 215
factory 56, 144
factory finder 56, 183, 185
factory object 183, 185
Filename
  shared repository setting 36
flat Transaction 204
flat transaction 210
flat transactions 202
Float 121
float 77, 82
floating point type 116
Floating-point Literals 253
fonts 16
Forward Declaration of Interfaces 75

G
Generate IDL tool
  example of use 44

I
I3 65–69
  advantages 58
  defined 58
icons, IR Browser 151
ID pragma 137
identifier 73
Identifiers 250

IDL
  Alphabetic Characters 245
  Escape Sequences 252
  exceptions 129
  interface 117
  mapped to Smalltalk 142
  Preprocessing 104
  Preprocessor 104
  Punctuation Characters 251
  traps 108

IDL (Interface Definition Language) 50
IDL identifiers
  underscore characters 113

IDL Preprocessing 104
#include 105
#include 105

IDL Syntax
  Basic Data Types 82
  Constructed Data Types 85, 86, 88
Index

- Constructed Types 85
- Declaring Attributes 99
- Declaring Constants 76
- Declaring Data Types 80
- Declaring Exceptions 95
- Declaring Interfaces 73
- Declaring Modules 72
- Declaring Operations 91
- Derived Interfaces
  - Inheritance 100
- Forward Declaration of Interfaces 75
- Multi-Dimensional Array 81
- One-Dimensional Array 81
- Simple Types 82
- Template Data Types 83
- Template Types 82
- typedef 81

- IDL to Smalltalk
  - mapping of identifiers 113

- IDLCompiler class>>importIDLFile
category
  - method 153
implementation
  - transactions 217
IIOP Transport 31
implementation
  - concurrency 196
  - event notification 181
implicit arguments
Operations 131
Implicit Invocation Interface (I3) 65–69
  - advantages 58
  - defined 58
importing IDL files 153
in 92
in Parameter Attribute 92
indirect base interface 101
indirect context management 204
Inheritance 100
inheritance (multiple), in DSTRepository 229
inheritance specification 100
initial object references 157
initialization
  - ORB, application 155
initializeFactories 145
inout 92
inout Parameter Attribute 92
Integer 121
Integer Literals 252
integral types 116
Interface
  - Declaring 73

- Forward Declaration 75
- Inherited Interface 100
- Smalltalk Mapping 117
interface 52, 73, 100
  - derived 101
  - direct base 101
  - indirect base 101
- Interface Repository 51, 142, 144
  - implemented in Distributed Smalltalk 54
  - interface 117
  - shared 35, 234
  - shared, how to enable 36
  - interface_definition 74
interfaces 115
  - concurrency 196
  - Lifecycle 190
  - version control 138
- Internet Inter-ORB Protocol 31
- interoperability 60
IR
  - see also DSTRepository
IR Browser
  - edit lock 154
  - icons 151
  - menus 152
IR Browser icons 151
IR, see Interface Repository
  - see DSTRepository
Isolation
  - ACID 200
K
Keywords 251
L
lifecycle 49, 182
  - copy 183
  - create 183
  - deep copy 183
  - destroy 184
  - externalize 184
  - move 184
  - relocating objects, see move 184
  - throw away 184
lifecycle service
  - using to create remote objects 62
list_initial_services operation 158
listInitialServices 158
Literals 252
  - Character 252
  - Floating Point 253
Index

Integer 252
String 254
Local RPC Testing 225
   option in Request Broker panel 224
Lock
   class 195
   locks 195
LockSet
   class 195
   LockSets 195
long 77, 82

M
Magnitude, mapped to IDL types 121
mail
   electronic 18, 20
   mapping of identifiers
      IDL to Smalltalk 113
Mapping Pragmas to IDL Types 135
   marshal 57
   marshall, see externalize 184
   marshallError 237
   marshalling and unmarshalling errors 237
member 95
menus
   Action (IR Browser) 152
   Edit (IR Browser) 152
   View (IR Browser) 152
Message Logging (option in Request Broker panel) 224
   method size, for performance 229
module 72
module_definition 73
mouse buttons 17
   <Operate> button 17
   <Select> button 17
   <Window> button 17
move 182, 184
Multi-Dimensional Arrays 81
   multiple event channels 168
Multiple inheritance 101
multiple lock semantics
   concurrency 194
   multiple lock semantics 194
   multiple subtransactions 211
N
name
   binding, unbinding, and rebinding 163
   component 161
   context 161
   identifier attribute 164
   kind attribute 164
name space collisions 113
naming 157
   implementation 165, 190
   initial object reference 157
   interfaces 165
   operations 162
   naming context 161
   naming graph 161
   Naming scopes 105
   naming service 49, 160
      binding instances to names 41
      DTSampleComputeService example 40
      unbinding instances 41
   nested transaction 204
   nested transactions 202
   network
      performance 228
      traffic 59
      unstable 59
   newObject 183
   notational conventions 16
O
Object Adapter Id
   shared repository setting 36
object reference 53, 54, 115
object references
   short lifespan 63
      vulnerability to network failures 63
Object Request Broker, see ORB
object services
   concurrency 50, 191
   event notification 166
   lifecycle 49, 182
   naming 49
      transactions 50, 200
objects
   CORBA 52
   objRef (object reference) 55
octet 82
One-Dimensional Array 81, 83
oneway 91, 93
Open Repository (Action menu) 152
Operation 91
   Parameter Attribute 92
   operation (IDL)
      access control 141
      defined in an interface 52
   Operation Arguments 92
Index

Operation Attribute 91, 93
Operations 130
   Declaring 91
   implicit arguments 131
   Smalltalk Mapping 130
operations 115
optimizing 223, 227
ORB (Object Request Broker) 51
cannot start 244
   components
ORBCControlPanel. 235
ORBDaemon startUpCoordinator 231
ORBNVList 51
ORBOBJECT 143
ORBOBJECT initializeORBAtHost 231
ORBRequest 51
OrderedCollection
   sequence 127
out 92
out Parameter Attribute 92
P
   paradigms
      choosing 58
   perform:on: 56
   performance
      distributed applications 63
      method size 229
      tuning 223, 227
   performance hints 63
   poor network performance 228
Pragma
   access 141
   ID 137
   prefix 138
   selector 141
   version 138
Pragma 134
Prefix pragma 138
prepared 205
Preprocessing 104
Preprocessing IDL 104
Preprocessor 104
Preprocessor Tokens 251
primary_expression 79
pull model (event notification) 168
Punctuation Characters 251
Q
   quality of service, events 173
   raises 93
   raises Expression 93
   readonly 99
   recoverable object 205
   recoverable objects 209
   recoverable server 205, 210, 213
   recovery service 205
   registering factory objects 183, 185
   relocating objects, see move 184
   Remote Object Debugger 226
   remote objects
      creating 62
   remote system control 59
   remove, see destroy 184
Repository
   Share Repository setting 36
   repository, shared 35
      how to enable 36
   RepositoryId 36
   RepositoryPragmas 137
   request 52
   Request Broker panel
      debugging option 223
      message logging option 224
      runtime 231
   Request Brokerl panel
      Local RPC testing option 224
Reserved Keywords 251
resolve 163
resolve_initial_references operation 158
resolveInitialReferences 158
resource 205
resource object 209
resource sharing 59
rollback 205
RPC (Remote Procedure Call) 60
RPC testing 225
runtime applications
   configuring images 230
   creating 230
   exception handling 235
   Request Broker panel 231
S
   scoped_name 77, 106
Scopes 105
selector 135
Selector pragma 141
Sequence
   Smalltalk Mapping 127
sequence 83, 127
OrderedCollection 127
sequence type 116
serialize, see externalize/internalize 184
server object 52
share repository setting
Filename 36
shared interface repository 35, 234
how to enable 36
Shared Repository (Configuration menu) 36
shared repository setting
Object Adaptor Id 36
sharing objects 60
sharing resources 59
short 77, 82
siblings 211
SII, see Static Invocation Interface
Simple Data Types 82
simulated distribution 225
SmallInteger 121
Smalltalk Mapping 130
Array 128
Attributes 119
Basic Data Types 121
Constants 120
Exception 129
Interface 117
Sequence 127
String 128
Smalltalk, mapped to IDL 142
special symbols 16
Standard Exception 96
Standard Exceptions 96
Static Invocation Interface (SII) 51
implemented in Distributed Smalltalk 54
String
Smalltalk Mapping 128
string 77, 83
String Literals 254
string type 116
Strings 83, 128
strings 83
struct 85
structure types 116
Structures 85
style hints 229
SubtransactionAwareResource 213
subtransactions 211
supplier, event 167
support, technical
  electronic mail 18, 20
  World Wide Web 18, 20
surrogate object 54
symbols used in documentation 16
T
  Table 1-2 83
technical support
electronic mail 18, 20
  World Wide Web 18, 20
Template Data Types 83
Template Types 82
Terminator 213, 215
thread 205
throw away 184
Tokens 249
  Identifiers 250
  Keywords 251
  Literals 252
  Preprocessor 251
tools
Generate IDL 44
TP (Transaction Process) monitor 205
transaction 205
  completing a transaction 220
  create a transaction 219, 222
  implementing a recoverable object 217
transaction client 205
transaction context 205, 206, 212, 214
transaction family 211
transaction object 206
transaction operation 205
transaction originator 206, 213
transaction server 206
transaction service 50, 200
  use of functionality for interfaces 216
transaction synchronization 206
transaction termination 211
transactional client 208
transactional object 208
transactional server 209
TransientErrorHandler 244
troubleshooting 223, 237
tuning 223, 227
two-phase commit 206
type any 171
typed events 173
typedef 81
typographic conventions 16
U
Unary Operators 78
unary_operator 79
unavailable object errors 238
Index

unbind 163
unbinding
  using naming service 41
underscore characters
    IDL identifiers 113
union 88
union types 116
Unions 88
unmarshal
  see also marshal
unmarshal, see marshal 237
unmarshallError 237
unsigned long 77, 82
unsigned short 77, 82
user interface
  optimizing usability 228
  User-defined Exceptions 95
users, multiple 60
UUID 56
V
ValueTypes 54
version control
  interfaces 138
Version pragma 138
View menu
  as Picture (IR Browser) 152
  IR Browser 152
View menu (IR Browser)
  as Text 152
void Return Type 91
W
wastebasket 184
well-known instances
  retrieving 40
World Wide Web 18, 20
X
X/Open 203
Reader Comment Sheet

Name: ________________________________

Job title/function: ________________________________

Company name: ________________________________

Address: _______________________________________

Telephone number: ( ) - _______ Date: _______ / _______ / _______

How often do you use this product? □ Daily □ Weekly □ Monthly □ Less

How long have you been using this product? □ Months □ Years

Can you find the information you need? □ Yes □ No

Please comment. ____________________________________________________________

Is the information easy to understand? □ Yes □ No

Please comment. ____________________________________________________________

Is the information adequate to perform your task? □ Yes □ No

Please comment. ____________________________________________________________

General comment: ___________________________________________________________

_________________________________________________________

WE STRIVE FOR QUALITY

To respond, please fax to Larry Fasse at (513) 612-2000.

CINCOM
The Smart Choice®