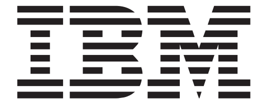
Component Broker

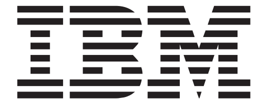


Application Development Tools Guide

Release 2.0

SC09-2705-03

Component Broker



Application Development Tools Guide

Release 2.0

SC09-2705-03

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xiii.

Fourth Edition (December 1998)

This edition applies to Release 2.0 of IBM Component Broker and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© Copyright International Business Machines Corporation 1997, 1998. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices xiii
Trademarks and Service Marks xiv

About This Book xvii
Who Should Read This Book xvii
How This Book is Organized xvii
Component Broker Information. xviii

Chapter 1. Object Builder Overview 1
Object Builder 1
What's New. 2
Design Principles for Component Broker Applications 3
Projects and Models 4
DBCS and Binary Data Support 5
Set up Object Builder 6
Open a Project 6
Set Object Builder Preferences 7
Migrate Old Projects 7
Requirements for Java Development 8
Work with Object Builder 9
Filters 9
Filter the Tasks and Objects Pane 9
Create a Filter for the Tasks and Objects Pane. 10
Search the Tasks and Objects Pane. 10
Run Object Builder in Batch Mode 11
Import C++ or Java Classes. 13

Chapter 2. Component Overview 15
Components 15
Component Assembly 16
Component Execution 16
Objects 17
Business Object 17
State Data 18
Data Object. 18
Persistent Object. 19
Schema Group 20
Schema 20
Key. 21
Copy Helper 21
Managed Object 22
Key Assistant 22
Methods and Attributes 23
User-Defined Methods. 23
Get and Set Methods 23
Framework Methods 24
Special Framework Methods 24
Push-Down Methods 25
Relationship Methods 25
Attributes. 26
Constructs 26
Business Object Behavior 27
Pattern for Handling State Data 27

© Copyright IBM Corp. 1997, 1998

Object Reference . . . 29
Data Object Interface . . . 29
Session Service . . . 30
Data Object Behavior . . . 30
Environment . . . 31
Form of Persistent Behavior and Implementation . . . 32
Data Access Pattern . . . 34
Handle for Storing Pointers . . . 35
Data Object Implementation Inheritance . . . 36

Chapter 3. Getting Started with Object Builder . . . 39
Getting Started with Object Builder . . . 39
Create a Component - Scenario . . . 39
Build DLLs or Shared Library Files - Scenario . . . 47
Package an Application - Scenario . . . 50
Install and Run an Application Using InstallShield - Scenario . . . 57
Install and Run an Application - Scenario . . . 61
Trace and Debug an Application - Scenario . . . 65
Uninstall an Application Using InstallShield - Scenario . . . 70
Uninstall an Application - Scenario . . . 71

Chapter 4. Working with Rose . . . 73
Using Rational Rose with Object Builder . . . 73
Rose . . . 74
Set up Rose 98 . . . 74
The Rose Bridge . . . 76
IDL Name Scoping in Rose . . . 77
Constructs You Can Export from Rose . . . 79
Class Properties You Can Export from Rose . . . 81
Class Relationships You Can Export from Rose . . . 84
Import Component Broker Frameworks . . . 86
Mapping Rules: Object Builder to Rose . . . 87
Component Broker Frameworks in Rose . . . 89
Export a Design from Rose . . . 89
Work with an Exported Design . . . 91
Import a Project into Rose . . . 92
Export from Rose - Scenario . . . 95
Import into Rose - Scenario . . . 98

Chapter 5. Creating Components in Object Builder . . . 101
Create a Component for Transient Data . . . 101
Create a Component for New DB Data . . . 101
Create a Component for New DB Data - Scenario . . . 102
Create a Component for Existing DB Data . . . 104
Mapping Helper . . . 105
Design Patterns and Iterators . . . 107
Customize Referential Integrity . . . 108
Data Encoding Schemes . . . 109
DB2 Data Type Mappings . . . 110
Oracle Data Type Mappings . . . 113
DDL . . . 114
Create a Component for PA Data . . . 115
Enterprise Access Builder (EAB) . . . 116
Transaction Object . . . 116
Transaction Record . . . 116
Procedural Adaptor Bean (PA Bean) . . . 117

Add endResource() to a Sessional Business Object . . . 117
Create a Component for PA Data - Scenario. . . . 118
Unit Test for Procedural Adaptors - Scenario. . . . 126

Chapter 6. Components Working Together . . . 129
Create a Relationship . . . 129
Dependencies within an IDL File . . . 129
Define a One-to-One Relationship . . . 130
Define a One-to-Many Relationship . . . 131
Define a Circular Relationship . . . 132
Foreign Key Patterns . . . 132
Define a Foreign Key Pattern . . . 133
Store an Object Reference . . . 135
Create a Child Component . . . 136
Inheritance . . . 137
Inheritance and Overriding in Helper Objects . . . 138
Inheritance and Overriding in Business Objects . . . 138
Inheritance and Overriding in Data Objects . . . 139
Abstract Base Class Inheritance . . . 140
Choosing an Inheritance Pattern for Persistence . . . 140
Inheritance with Attributes Duplication . . . 141
Define a Child with Attributes Duplication . . . 142
Inheritance with Attributes Duplication - Scenario . . . 144
Inheritance with Key Duplication . . . 147
Define a Child with Key Duplication . . . 149
Inheritance with Key Duplication - Scenario . . . 151
Inheritance with a Single Datastore . . . 155
Define a Child with a Single Datastore . . . 156
Inheritance with a Single Datastore - Scenario . . . 158
Inheritance with Views . . . 162
Define a Child with Views . . . 164
Inheritance with Views - Scenario. . . . 165
Create a Composite Component - Overview . . . 172
Composite Component . . . 173
Composition . . . 174
Composite Business Object . . . 175
Composite Key . . . 176
Composite Component Creation - Scenario . . . 177

Chapter 7. Multi-Platform Development . . . 187
Platform Differences . . . 188
Set Platform Constraints . . . 189
Develop a Multi-Platform Application - Scenario . . . 190

Chapter 8. Team Development . . . 201
Change Control . . . 202
Model Interchange with XML . . . 203
Set up a Team Environment. . . . 204
Export a Rose Design to a Team Environment . . . 204
Split up a Project for Team Development . . . 206
Add an Integration Project to a Team Environment . . . 208
Set up a Change Control Process . . . 209
Set up an Automated Build Process . . . 210
Set up a Team Development Environment . . . 211
Work in a Team Environment . . . 212
Import Projects from a Team Environment . . . 212

Create a Project in a Team Environment 215
Edit a Project in a Team Environment 216
Delete a Project in a Team Environment 217
Build DLLs in a Team Environment 217
Package an Application in a Team Environment 218
Team Development with Rose - Scenario 218
Maintain a Team Environment 223
Export XML 224
Import XML 225
Move a Project 227
Change Project Divisions 227
The Compare and Merge Tool for XML 228
Compare Files with the Compare and Merge Tool for XML 228
Merge Files with the Compare and Merge Tool for XML 229
Manage Cross-Project Dependencies 230

Chapter 9. XML Wizards 233
Create an XML Wizard 233
Start the SmartGuide Customizer for XML 234
Define XML Wizard Macros 235
Customize Value Lists in an XML Wizard 237
Derive Values in an XML Wizard 237
Propagate Values in an XML Wizard 239
Constrain Values in an XML Wizard 240
XML Wizard Constraints 241
Define the Layout of an XML Wizard 242
Test an XML Wizard 243
Run an XML Wizard 243
Edit an XML Wizard 244
Distribute an XML Wizard 245

Chapter 10. Object Development Tasks 247
Work with Attributes 247
Add an Attribute 247
Edit an Attribute 248
Delete an Attribute 249
Map a Data Object to a PA Persistent Object 249
Map a Data Object to a DB Persistent Object 251
Map a Data Object to the Parent’s Persistent Object 254
Map a Data Object to the Child’s Persistent Object 255
Map Data Object Attributes to Persistent Object Attributes 256
Map Attributes Using the Default Mapping Pattern 257
Map Attributes Using a Key 258
Map Attributes Using a Mapping Helper 260
Complex Attributes and Mapping Patterns 263
Map Complex Attributes Using the Primitive Pattern 264
Map Complex Attributes Using the Explode Pattern 265
Work with Methods 267
Add Code for User-Defined Methods 267
Add an Initializer Method 268
Edit a User-Defined Method 269
Edit Get and Set Methods 270
Edit Framework Methods 270
Edit Special Framework Methods 271
Import Changes to Methods 272
External Files for Method Bodies 273

Use Push-Down Methods with PA Persistent Objects 274
Customize Business Object OO-SQL Implementation Methods 275
Customize Persistent Object ESQL Framework Methods 276
Delete a Method 277
Work with Constructs 277
Define Constructs with File Scope 278
Define Constructs with Module Scope 279
Define Constructs With Interface Scope 279
Edit a Construct 280
Delete a Construct 280
Work with Business Objects. 281
Create a Business Object File 282
Add a Business Object Module 282
Add a Business Object Interface 283
Add a Business Object Implementation and Data Object Interface. 284
Add a Business Object from a Data Object 287
Map a Business Object to a Data Object 288
Create a Business Object Interface by Importing an IDL File 289
Edit a Business Object Interface 290
Edit a Business Object Implementation. 290
Delete a Business Object Interface 291
Delete a Business Object Implementation. 291
Work with Keys 292
Add a Key 292
Edit a Key 293
Delete a Key 293
Work with Copy Helpers 294
Add a Copy Helper 294
Edit a Copy Helper 295
Delete a Copy Helper 295
Work with Data Objects - Overview 296
Create a Data Object Interface. 297
Add a Data Object Implementation 299
Add a Data Object From a Business Object 302
Create a Data Object File 303
Add a Data Object Module 304
Add a Data Object from a DB Persistent Object 304
Add a Data Object from a PA Persistent Object 305
Create a Data Object Interface by Importing an IDL File 306
Edit a Data Object Interface. 309
Edit a Data Object Implementation 310
Delete a Data Object Interface 312
Delete a Data Object Implementation 313
Work with DB Persistent Objects 313
Add a Persistent Object and Schema 313
Add a Persistent Object from a DB Schema 316
Edit a DB Persistent Object 317
Delete a DB Persistent Object 317
Work with DB Schema Groups. 318
Create a DB Schema Group 318
Edit a DB Schema Group 319
Delete a DB Schema Group. 320
Work with DB Schemas 320
Create a DB Schema by Importing an SQL File 321
SQL View Editor 323
Create a View with the SQL View Editor 324

Edit a View with the SQL View Editor . . . 325
Use Complex Relationships in SQL Clauses . . . 326
Edit a View . . . 328
Edit a DB Schema . . . 329
Re-import an SQL File . . . 330
Edit a Generated SQL File . . . 331
Delete a DB Schema . . . 333
Work with PA Persistent Objects - Overview . . . 333
Add a Persistent Object from a PA Schema . . . 334
Map a Data Object to a PA Persistent Object . . . 334
Edit a PA Persistent Object . . . 336
Delete a PA Persistent Object . . . 336
Work with PA Schemas - Overview . . . 337
Create a PA Schema by Importing a PA Bean . . . 337
Edit a PA Schema . . . 339
Delete a PA Schema . . . 339
Work with Managed Objects - Overview . . . 339
Add a Managed Object . . . 340
Edit a Managed Object . . . 341
Delete a Managed Object . . . 341
Work with Customized Homes - Overview . . . 342
Home . . . 342
Create a Customized Home . . . 343
Edit a Customized Home . . . 344
Delete a Customized Home . . . 345
Work with Container Instances - Overview . . . 345
Container . . . 345
Create a Container Instance . . . 346
Edit a Container Instance . . . 348
Delete a Container Instance . . . 348
Work with Compositions - Overview . . . 348
Create a Composition File . . . 349
Add a Composition Module . . . 349
Add a Composition . . . 350
Edit a Composition . . . 352
Work with Composite Business Objects - Overview . . . 353
Add a Composite Business Object Interface . . . 354
Add a Composite Business Object Implementation and Data Object Interface 355
Edit a Composite Business Object Interface . . . 359
Edit a Composite Business Object Implementation . . . 360
Work with Composite Keys - Overview . . . 360
Add a Composite Key . . . 360
Edit a Composite Key . . . 362

Chapter 11. Configuration Tasks . . . 363
Build DLLs - Overview . . . 363
Generate Code . . . 363
Define a Client DLL . . . 364
Define a Server DLL . . . 366
Generate a Makefile . . . 367
Build the DLLs. . . . 368
Build Configuration Options . . . 370
Remote Build Configuration (OS/390) Remote Build . . . 372
Remote Build . . . 372
Pass Ticket . . . 372
Profile . . . 372

Launch a Remote OS/390 Build 373
Launch a Remote OS/390 Build - Scenario 373
Package an Application 375
Create an Application Family 375
Add a Client Application 376
Add a Server Application 377
Configure a Managed Object 377
Edit a Managed Object Configuration 379
Delete a Managed Object Configuration 379
Generate the Install Image 379
Application DDL Files 381
The DDL Editor 382
Creating and Editing DDL Files 384
Edit an Application DDL File. 387
The Structure of a DDL file 389

Chapter 12. Access a Component through FlowMark 395
FlowMark 395
FDL 395
Bag. 395
Add a Bag 396
Data Structure. 396
Add a Data Structure 397
Program 397
Activity 398
Add a Program 398
Map a Component to a Data Structure 400
Map Input Parameters to the Input Data Structure 400
Map Output Parameters to the Output Data Structure 401
Map Find Parameters to the Input Data Structure 402
Work with FlowMark Business Objects - Overview 403
Create a Component Instance through FlowMark 404
Call a Component Method from FlowMark 404
Delete a Component Instance through FlowMark 405
Work with Bags - Overview 406
Edit a Bag 406
Delete a Bag 407
Work with Data Structures - Overview 408
Edit a Data Structure 408
Delete a Data Structure 408
Work with Programs - Overview 409
Edit a Program 409
Delete a Program 410

Chapter 13. Troubleshooting 411
Check a Model for Consistency 412
Consistency Checker Errors. 412
Restrictions for R2.0 419
Composition Restrictions 425

Chapter 14. Debug Local Applications 427
Write Programs for Debugging. 427
Compile a Program for Debugging 427
Environment Variables. 429
Set Environment Variables for the Debugger. 429
Start or Stop Debugging a Program 433

Invoke the Debugger . . . 433
Start the Debugger . . . 433
Debugger Options . . . 434
Attach to a Process . . . 435
Specify Command-Line Parameters for Your Program . . . 436
Attach to a Running Java Virtual Machine . . . 437
Start Debugging a Java Applet . . . 437
Debug on Demand . . . 438
When You Start Debugging . . . 439
Search Order . . . 439
Debugger Windows . . . 440
Remote Debugging . . . 442
Start the Debugger and the Remote Program . . . 443
Start the Debugger and the Remote Java Program . . . 444
Breakpoints . . . 446
Set Breakpoints . . . 446
Set Breakpoints in the Breakpoints List Window . . . 447
Set and Delete Breakpoints from a Source Window . . . 447
Set Function or Method Breakpoints from the Session Control Window . . . 448
Set a Line Breakpoint . . . 448
Set a Deferred Breakpoint . . . 449
Set Multiple Breakpoints . . . 449
Delete Breakpoints . . . 450
Enable and Disable Breakpoints . . . 450
Modify Breakpoint Characteristics . . . 450
Debug a DLL . . . 451
Start Debugging a DLL from a Load Occurrence Breakpoint Dialog . . . 451
Start Debugging a DLL from a Source Window . . . 452
Start Debugging a DLL from the Breakpoints List Window . . . 452
Start Debugging a DLL from the Session Control Window . . . 452
Run, Step Through, or Stop a Program . . . 453
Run a Program . . . 453
Step Commands . . . 453
Skip over Sections of Code . . . 455
Halt Execution of a Debuggee Program . . . 455
Restart Your Program . . . 455
Terminate a Debug Session . . . 456
Debugger Monitors . . . 457
Differences between Program and Private Monitors . . . 458
Add Expressions and Variables to a Monitor . . . 458
Open a New Storage Monitor . . . 458
Open Other Debugger Windows from a Source Window . . . 459
View Variables, Memory, Registers, and the Stack . . . 460
View Variable Contents . . . 460
View a Location in Storage . . . 461
View the Contents of Registers . . . 461
View the Contents of the Call Stack . . . 462
Change the Representation of Storage . . . 462
Change the Contents of Storage, Variables, and Registers . . . 463
Debug Heap Use . . . 465
Heap Errors . . . 466
Debug Optimized Code . . . 467
Notes on Debugging Optimized Code . . . 468
Debugging Threads . . . 468
Critical Sections . . . 469
Deadlocks and Timing Problems . . . 470

Must Complete Sections 471
Race Conditions 471
Threads and C++ Class Members 472
Threads and Load Occurrence Breakpoints 472
Threads and Source Language Statements 473
Windowing System Lockups. 473
Troubleshooting and Limitations 474
C++ Expressions Supported. 474
Limitations when Debugging Visual C++ Programs 476
Interpreted Java Expressions Supported 477
Limitations When Debugging Interpreted Java 477
Debugger Is Using a Different Executable Version 478
Debugger Cannot Find Source Code 478
Values that Are Valid for the Current Representation. 479
Valid Addresses and Expressions. 479
Right Mouse Button Behavior 480
Change Right Mouse Button Behavior 480

Chapter 15. Trace and Debug Distributed Applications 481
Object Level Trace 481
What’s New. 483
Supported Platforms and Languages 484
Monitoring Modes 485
Prepare for Distributed Tracing and Debugging. 486
Compile Application Code with OLT Flags 486
Enable Remote Tracing and Debugging 488
Trace a Distributed Application. 489
Start the OLT Server and Viewer on Separate Machines 490
Use OLT with OS/390 492
Debug a Distributed Application 494
Set Breakpoints on the Trace 494
Debug Business Objects 495
Reading the Trace 499
Trace Symbols 500
Selected Event 501
Partial-order Display 502
Real-time Display 503
Performance Analysis 504
Navigate the Trace 504
Scroll the Trace 505
Reorder Trace Lines 506
Tag an Event 507
Save the Current Trace to a File 508
Open an Existing Trace File. 508
OLT Scenarios 509
Trace and Debug a Java Client and C++ BO - Scenario 512
Debug a Java Client from Startup - Scenario 515
Debug a C++ Client and C++ BO in Step by Step Mode - Scenario 518
Trace and Debug a C++ Client and C++ BO on AIX - Scenario. 522
OLT Reference 525
OLT Environment File 525
OLT Command-line Arguments. 527
OLT Troubleshooting 527

Chapter 16. IR Browser 533
Start the IR Browser 533

Configure Online Help 533
View Objects in the Repository. 533
View the Definition of an Object 533
View Relationships Between Objects 533
View the Operations of an Interface 534
Search the Repository 534
Find An Object 534
Search Using Wildcards 535
Find an Interface's Referencing Operations 535
Search by Object Type 535
Modify the Repository 536
Delete Objects from the Repository 536

Index 537

Notices

This publication was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter in this publication. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this document at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department LZKS
11400 Burnet Road
Austin, TX 78758
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This document may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

AIX
CICS
DB2
IBM
IMS
MVS/ESA
OS/2
PowerPC
VisualAge

AFS and DFS are trademarks of Transarc Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

Oracle and Oracle8 are registered trademarks of the Oracle Corporation in the United States and/or other countries.

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

Other company, product, and service names may be trademarks or service marks of others.

About This Book

The *Application Development Tools Guide* covers information about the Component Broker Toolkit (CBToolkit), which includes the following tools:

- Object Builder
- Local Debugger
- Object Level Trace
- Interface Repository (IR) Browser

The guide provides conceptual information, as well as a detailed explanation of how to generate and test multi-tier applications.

Who Should Read This Book

The *Application Development Tools Guide* is intended for administrators and application programmers who want to:

- understand the IBM Component Broker development environment
- create, execute and manage distributed applications across network computing environments
- connect multiple backend systems to dynamic, new applications
- capture information from database systems, transaction processing systems, and applications, as highly manageable components

How This Book is Organized

Chapter 1. Object Builder provides a list of newly-introduced features, an overview of Object Builder projects and models, and instructions on setting up and working with Object Builder.

Chapter 2. Component Overview provides an overview of the components, component objects, methods, attributes, and object properties from an Object Builder perspective.

Chapter 3. Getting Started with Object Builder provides a series of scenarios that walk you through the creation and deployment of a simple Component Broker application. These scenarios were formerly part of the *Quick Beginnings* book.

Chapter 4. Working with Rose provides instructions for using Rational Rose 98 with Object Builder, including set up, export, import, mapping rules, and some simple scenarios.

Chapter 5. Creating Components in Object Builder provides an overview of the steps involved in creating various types of component, based on the kind of data persistence they provide.

Chapter 6. Components Working Together covers object relationships (one-to-one, one-to-many, circular, and foreign key pattern), component inheritance (including overriding, inheritance patterns for persistence, and scenarios), and creating composite components.

Chapter 7. Multi-Platform Development describes how to create an application for deployment on multiple platforms.

© Copyright IBM Corp. 1997, 1998

Chapter 8. Team Development describes how to use Object Builder in a team environment, including set up, change control process, build processes, working with Rose, and maintenance of a team environment.

Chapter 9. XML Wizards describes how to create your own wizards for development in Object Builder, using Object Builder's exported XML format as a base.

Chapter 10. Object Development Tasks provides coverage of the various tasks you can perform (creating, editing, mapping, deleting) in the course of working with components.

Chapter 11. Configuration Tasks describes how to define and build component libraries (DLLs), and configure applications.

Chapter 12. Access a Component through FlowMark describes how to access components from FlowMark client processes.

Chapter 13. Troubleshooting describes how to validate your project model and find consistency errors, and covers some of the Object Builder restrictions for this release.

Chapter 14. Debug Local Applications explains how to prepare programs for local debugging, how to start and stop a debugging session, and how to interact with the debugger interface.

Chapter 15. Trace and Debug Distributed Applications describes how to use Object Level Trace to trace and debug multilingual, distributed applications. Several scenarios are provided to walk you through a typical OLT session.

Chapter 16. IR Browser provides instructions on how to view object definitions and relationships in the interface repository, how to perform various types of searches on the repository, and how to modify the contents of the repository.

Component Broker Information

The following information is available as part of Component Broker for Windows NT, AIX, and OS/390:

- Help information is available from Component Broker product panels, by pressing the F1 key.
- The Component Broker online library can be viewed using a frames-compatible Web browser:

  http://localhost:49213/cgi-bin/cbwebx.exe/en_US/cbdoc/Extract/0/index.htm
- *Component Broker for Windows NT and AIX Quick Beginnings*, G04L-2375 explains how to easily create and verify a starter Component Broker environment. These instructions walk the user through a typical server and client installation. Users can extend this configuration using the information in the Component Broker for Windows NT and AIX Planning, Performance, and Installation Guide.
-

Component Broker for Windows NT and AIX Planning, Performance, and Installation Guide, SC09-2798 provides a comprehensive overview of the Component Broker environment, then guides the user through planning considerations including capacity planning, performance tuning, prerequisites, and migration. It also leads the user through installation options for all Component Broker environments.

•

Component Broker for Windows NT and AIX CICS and IMS Application Adaptor Quick Beginnings, GC09-2703 provides a brief technical overview of the CICS and IMS application adaptor and guides the user through its installation and configuration. Step-by-step instructions guide the user through creating an initial CICS and IMS application using application development tools included in the CBToolkit package.

•

Component Broker for Windows NT and AIX Oracle Application Adaptor Quick Beginnings, GC09-2733 provides a brief technical overview of the Oracle application adaptor and guides the user through its installation and configuration. Step-by-step instructions guide the user through creating an initial Oracle application using application development tools included in the CBToolkit package.

•

Component Broker for Windows NT and AIX System Administration Guide, SC09-2704 provides information about configuring and operating one or more hosts managed by Component Broker. It also provides general information about using the System Manager User Interface.

•

Component Broker Programming Guide, G04L-2376 describes the programming model including business objects, data objects, and information about MOFW, IDL, and C++ CORBA programming.

•

Component Broker Advanced Programming Guide, SC09-2708 describes the Component Broker implementation for the CORBA Object Services and the Component Broker Object Request Broker (including remote method invocation and the Dynamic Invocation Interface (DII) procedures), Session Service, Cache Service, Notification Service, Interlanguage Object Model (IOM), and work-load management (WLM).

•

Component Broker Programming Reference, SC09-2810 contains information about the APIs available to Component Broker application developers.

•

Component Broker for Windows NT and AIX Problem Determination Guide, SC09-2799-00 explains how to identify and resolve problems within a Component Broker environment using the tools provided with Component Broker. The book includes information on installation problems, run time errors, debugging of applications, and analysis of log messages.

•

Component Broker Glossary, SC09-2710 contains terms and definitions relating to Component Broker.

•

OS/390 Component Broker Introduction, GA22-7324 describes the concepts and facilities of Component Broker and the value it has on the OS/390 platform. The audience is a knowledgeable decision maker or a system programmer.

- *OS/390 Component Broker Planning and Installation*, GA22-7331 describes the planning and installation considerations for Component Broker on OS/390.

- *OS/390 Component Broker System Administration*, GA22-7328 describes system administration tasks and operations tasks, as provided in the system administration user interface for OS/390.

- *OS/390 Component Broker Programming: Assembling Applications*, GA22-7326 provides information for assembling applications using Component Broker on OS/390.

- *OS/390 Component Broker Operations: Messages and Diagnosis*, GA22-7329 provides diagnosis information and describes the messages associated with Component Broker on OS/390.

Chapter 1. Object Builder Overview

Object Builder

The CBToolkit Object Builder is the development environment for the Component Broker product. You can use it to develop your application from start to finish, or start by designing in Rose and then import the design into Object Builder, where you add the final objects and program logic.

Object Builder supports the CORBA programming model using IDL, C++, and Java. You can generate complete working applications, including unit test versions and full client-server packages complete with server setup scripts.

You can use Object Builder to:

- develop new applications
- wrapper existing applications
- add new function to existing applications
- package an application

In order to build the application DLLs you define in Object Builder, you will need the Component Broker Server SDK installed, as well as any prerequisite application development software.

The model for your application is constructed out of components. Many of the development tasks in Object Builder revolve around defining components.

The Object Builder user interface is divided into panes, which provide access to different views of your application. Most interactions with Object Builder are through the panes and the pop-up menus for the objects in these panes. The Object Builder panes are:

Tasks and Objects pane
This pane contains multiple folders. These folders organize the component objects as they are created. The objects in these folders represent a rough task flow through your use of Object Builder.

- The *Framework Interfaces* folder shows the framework interfaces provided by Object Builder.
- The *User-Defined Business Objects* folder, the *User-Defined Data Objects* folder, the *DBA-Defined Schemas* folder, and the *User-Defined Compositions* folder are the folders you use to define component objects and show the component objects already defined.
- The *Non-IDL Type Objects* folder is the folder for C++ and Java objects defined outside of Object Builder.
- The *Build Configuration* folder is the folder where you configure component objects into the DLL files.
- The *Application Configuration* folder is the folder where you configure the DLL files into applications.
  For AIX only: The DLL files are called “shared library” files and are in the format lib*.so.
- The *Container Definition* folder and the *Default Homes* folder show the container and homes with which your applications can be configured.

© Copyright IBM Corp. 1997, 1998

Inheritance pane
This pane shows the inheritance structure for the selected component object. You can switch between the interface inheritance view and the implementation inheritance view. You can also turn off this pane, giving more room to the Methods pane, by either using **View - Minimize Pane**, or clicking the Minimize button on the upper left corner of the pane. You can also detach the pane from the Object Builder main window using **View - Detach Pane**.

Methods pane
This pane lists the methods and attributes for the object selected in the Tasks and Objects pane.

Source pane
This pane is used to edit the implementations for the selected method from the Methods pane. When a method is selected, its source code is displayed in this pane. It also displays the generated code for a particular object, when you select the **View Source** option from an object's pop-up menu.

AIX To start Object Builder from the command line, enter ob. Object Builder takes a few moments to start. The Specify Directory wizard is opened. Enter a directory name (for example, $HOME/MyProject) and click the **Finish** button. If this directory does not yet exist, a message is displayed asking if you want to create it. Click the **Yes** button, and provide a name for the new model.

WIN To start Object Builder from the Windows NT **Start** menu, select **Programs > IBM Component Broker for Windows NT > Object Builder**. Object Builder takes a few moments to start. The Specify Directory wizard is opened. Enter a directory name (for example, x:\CBroker\MyProject) and click the **Finish** button. If this directory does not yet exist, a message is displayed asking if you want to create it. Click the **Yes** button, and provide a name for the new model.

RELATED CONCEPTS
"Components" on page 15
"Projects and Models" on page 4
"Design Principles for Component Broker Applications" on page 3

RELATED TASKS
"Open a Project" on page 6
"Getting Started with Object Builder" on page 39
"Filter the Tasks and Objects Pane" on page 9
"Using Rational Rose with Object Builder" on page 73
"Create a Component for Transient Data" on page 101
"Create a Component for New DB Data" on page 101
"Create a Component for Existing DB Data" on page 104
"Create a Component for PA Data" on page 115

What's New

The following major changes have taken place in Object Builder since R1.3:

Team development
Team environments, including cross-project references and builds, are now easier to set up and maintain. A general process for team development is documented, from set up to maintenance.

Troubleshooting
A model consistency checker helps you locate problems in your project model. An XML compare and merge tool lets you resolve differences between out-of-synch project versions.

General Ease-of-Use
You can now rename most elements in Object Builder (for example, files, interfaces, implementations, attributes, methods - but not PA schemas). You can search and filter the Tasks and Objects pane.

Cross-Platform Development
You can constrain development options for multi-platform objects, and generate code simultaneously for multiple platforms. You can develop for multiple platforms at once, and dynamically switch your view between the different target platforms.

Extended OS/390 Support
You now have more PA development options available for your OS/390 applications. You can configure and start remote builds from within Object Builder.

Rational Rose Integration
You can use Rational Rose in a team environment, exporting to multiple projects at once. You can selectively export elements of your design. You can also import from Object Builder, creating a Rose design based on an Object Builder project.

FlowMark Integration
You can integrate FlowMark client applications with Component Broker components, mapping data structures of the client application to attributes and methods of your Component Broker components.

Design Your Own Wizard
You can extend Object Builder's functionality by creating your own wizards. Create a sample component or file in Object Builder, then export the XML and use the SmartGuide Customizer for XML to define a wizard that creates components or files, using your sample as a template.

Additional Minor Changes
There have been numerous smaller changes throughout Object Builder. For example, protected and private attributes are now defined only on the implementation level, not as part of a business object interface.

Design Principles for Component Broker Applications

A typical design for a Component Broker application has three architectural layers:

- Application
  Consists of components that provide business logic for the application, but do not deal directly with data persistence.
- Consolidation
  Consists of components that consolidate the interfaces of a number of related components, through relationships or references. These components fill the same role as composite components (which are not currently definable in Rose).
- Base
  Consists of components that provide base-level behavior and data persistence.

Within each of these layers, you can have categories of related components. These categories, or packages, are the basic organizing principle in UML, and can map to

separate projects in a team environment. You would typically have multiple packages for each layer, with each package containing a number of related components. A package should not mix components from different layers.

RELATED CONCEPTS

"Rose" on page 74
"The Rose Bridge" on page 76
"Object Builder" on page 1
"Components" on page 15

RELATED TASKS

"Using Rational Rose with Object Builder" on page 73
"Export a Design from Rose" on page 89
"Export a Rose Design to a Team Environment" on page 204

Projects and Models

A project provides the directory structure that organizes your work. It can contain any number of components, organized into applications and application families. Each project contains a single model, which can be used to generate code for multiple platforms. When you create a new project, you need to name the model, and also identify any dependencies your work will have on other, existing, projects. The model name you provide will be used to identify the project for team environment builds.

Within the project directory, your work is stored in several subdirectories:

- *project/*Model
  Contains the .uni files that Object Builder uses to store your work between sessions, or when you select **File - Save**. Each model directory includes the following files:
  - obp.uni
    The project file. Defines project metadata. Defines the models that the project accesses in read-write and read-only modes. May itself be accessed by other projects in read-write mode (so that projects can exchange dependency information).

    At minimum, the project model has a dependency on obfram.uni and obprim.uni, models that define Component Broker framework interfaces and Component Broker primitive elements.

    This file should not be deleted, except as part of the entire project directory, or directly edited.
  - project.cfg
    If you are using the OBModelPath environment variable to manage your dependencies, then this file is not used. If you are **not** using the environment variable, then this file provides an ASCII version of the **depends** and **usedby** relationships this project's model maintains with other projects' models. You can manually change the dependencies by directly editing this file. Generally you should manage cross-project dependencies through the Project Dependencies page of the Open Project wizard, and through the OBMODELPATH environment variable.
  - obm.uni
    The model file that contains your work. Accessed in read-write mode by the project. Other obm.uni files in other project/Model directories may be accessed in read-only mode.

This file should not be deleted, except as part of the entire project directory, or directly edited.

If you are using external files to provide the implementations for some methods, these external files are also stored here.

- *project*/Working
  Contains the platform subdirectories for generated source files by platform (for example, *project*/Working/NT, *project*/Working/AIX). Source files are generated for the platforms selected on the **Platform** menu of Object Builder. You can generate source files by selecting **Generate - Selected** or **Generate - All** from the pop-up menus of folders or objects in the Tasks and Objects pane.
- *project*/Export
  Contains any exported model elements, in XML format. You can import these files into other projects, using the **Import** menu item on folder pop-up menus in the Tasks and Objects pane.
- *project*/Import
  Contains any XML files that were used by the Rose Bridge to export a Rose model into Object Builder.
- *project*/XMI
  Contains the file xmi.xml, which holds any model information that is not directly translatable between Rose and Object Builder. When you import or export from Rose, this file maintains the extra information that would otherwise be lost in the transfer.

RELATED CONCEPTS
“Components” on page 15
“Rose” on page 74
“The Rose Bridge” on page 76
“Chapter 8. Team Development” on page 201

RELATED TASKS
“Open a Project” on page 6

DBCS and Binary Data Support

Double Byte Character Set (DBCS) is an encoding scheme for Asian characters such as Japanese. DB2 allows you to store both database meta-data (for example: table names and column names) and database data in DBCS format. It also supports binary data storage.

Object Builder R2.0 enables the following storage patterns:

- Database meta-data names are in DBCS format
- Database data is stored in either DBCS or Binary format

Database meta-data names are in DBCS format
When you create a persistent object from a schema that was imported, if the given column name in the schema is an ASCII name (a legal C++ identifier), Object Builder will use the same name as the attribute name for the persistent object; otherwise, Object Builder generates names such as POAttribute1, POAttribute2. You can change these tool-generated names.

Database data is stored in either DBCS or Binary format
Object Builder uses the following data encoding schemes (page 109) for data of *string* type that is stored in database meta-data:

- DBCS
- Single Byte Character Set (SBCS) or Multi Byte Character Set (MBCS)
- Binary data

Note the following points when you do database queries:

- You can do database queries using DBCS or binary data just as you do queries with any other data type.
- You cannot do queries over large object types such as `LONG VARCHAR` and `LONG VARGRAPHIC` if they are used as either primary or foreign keys.

RELATED CONCEPTS

"Schema" on page 20
"Persistent Object" on page 19

RELATED TASKS

"Add a Persistent Object and Schema" on page 313

RELATED REFERENCES

"Data Encoding Schemes" on page 109
"DB2 Data Type Mappings" on page 110
"Oracle Data Type Mappings" on page 113

Set up Object Builder

Open a Project

WIN To start Object Builder from the Windows NT desktop, select **Start - Programs - IBM Component Broker - Object Builder**. The Open Project wizard opens to the Specify Project Directory Page.

To start Object Builder on AIX (or on Windows NT from a command prompt), type the command

```
ob <project_directory>
```

A project directory has a model subdirectory, where Object Builder stores an internal model of your work. The project directory also has a working directory, where Object Builder generates the code (such as .idl and .cpp files) for your work.

If you are creating a new project, you will be prompted to provide a model name. The name you provide will be used to identify the project in a team environment, regardless of any changes in directory structure.

If you are working with a large project (more than 30 components), you may need to increase the maximum heap size of the Java virtual machine. You can do so by editing the ob.bat file:

1. Make sure Object Builder is closed.
2. Edit \Cbroker\bin\ob.bat
3. Change the parameter -mx255m, increasing the number by 5m for each additional component in your project (this number is approximate, and assumes components of average complexity).

For example, if your project contains 100 component, change the parameter to -mx605m (70 additional components multiplied by 5m each, plus the original 255m).

4. Start Object Builder. The new parameter is used, and the maximum size of the Java virtual machine is increased.

Note: Do not alter the contents of the directories: **<path>\Cbroker\obprime** and **<path>\Cbroker\obframe**. These directories contain definitions for IDL primitive types and for the Component Broker frameworks, both of which are used by the project models you create in Object Builder.

RELATED CONCEPTS
"Object Builder" on page 1
"Projects and Models" on page 4

RELATED TASKS
"Migrate Old Projects"

Set Object Builder Preferences

You can customize the appearance and behavior of Object Builder using the Preferences notebook. To access the notebook and set preferences for Object Builder, follow these steps:

1. Click **File - Preferences**. The Preferences notebook opens.
2. Click on a folder or node in the tree view on the left. The General folder organizes general settings for Object Builder's appearance and behavior. The other folders organize specific settings for the different panes in Object Builder (for example, the Tasks and Objects pane). You can select from the following folders or nodes:
   - General Page
   - Appearance Page
   - Toolbars Page
   - Help Page
   - Tasks and Objects Page
   - Source Page
   - Text Style Page
   - Keyboard Support Page
3. Specify the settings you want.
4. Click **OK** to apply your settings and return to Object Builder.

RELATED CONCEPTS
"Object Builder" on page 1

Migrate Old Projects

You can work with projects created with Object Builder 1.3. However, when you save your changes, the project can no longer be worked with in Object Builder 1.3.

Note: There is no direct way to migrate from 1.0, 1.1, or 1.2 to 2.0. You can only migrate through immediately subsequent versions, until you reach 2.0 (for example: from 1.1 to 1.2 to 1.3 to 2.0).

For information on migrating existing NT or AIX projects to OS/390 targets, consult the *OS/390 Component Broker Planning and Installation Guide.*

To migrate your work from 1.3 to 2.0, follow these steps:

1. Start Object Builder.
2. Specify the location of your old project (for example, Cbroker\my1.3Project) on the Specify Project Directory Page of the Open Project wizard.
3. Click **Finish**.
4. Save the project. The Model Conversion dialog opens.

   You have several options in how you convert the model:
   - By default, the model is saved in the 2.0 format, replacing the old version.
   - If you check the **Compress 2.0 version** option, Object Builder will save the model in compressed form. This may take some time, but can result in considerable space savings.
   - If you check the **Retain 1.3 version** option, Object Builder will create a backup version of the old model, in the directory \Model13, before saving the model in the new format.
5. Click **Save**.

   The saved project is now updated to the 2.0 version.

If you open a 1.3 project as a project dependency from a 2.0 project (by selecting it on the Project Dependencies Page of the Open Project wizard), then it remains at the 1.3 level, until you open and save it directly.

RELATED CONCEPTS
"Projects and Models" on page 4

RELATED TASKS
"Set Object Builder Preferences" on page 7
"Open a Project" on page 6

Requirements for Java Development

To develop Java business objects, you need JDK version 1.1.6.

WIN The configuration to support Java business objects includes setting the CLASSPATH environment variable to include the path to the classes.zip file of the JDK.

The CLASSPATH environment variable must be set in the User Variables section of the user ID specified to be used by system management during the Component Broker install. To insure that this variable is correctly set:

1. Logon to Windows NT as the user specified for system management.
2. Set the CLASSPATH variable in the User Variables section of the environment setting to the value:

   ```
   x:\1.1.6\lib\classes.zip;.;%classpath%
   ```

   where *x:\1.1.6* is the JDK base install directory.
3. Stop the CBConnector service.
4. Restart the CBConnector service.
5. Reactivate the configuration using the System Manager User Interface.

AIX If the path does not already exist, you may need to add a path to libjava.a.

RELATED CONCEPTS
Programming Languages and Conventions (*Programming Guide*)

Work with Object Builder

Filters

You can use filters in Object Builder to exclude information from the Tasks and Objects pane. You can use the filters provided with Object Builder, or define your own.

Object Builder provides the following filters or views:

- **Business Objects View**
  Displays the User-Defined Business Objects folder, the Non-IDL Types folder, and the Build Configuration folder. Use this view if you are creating or working with components for new data, and do not need to work with data objects separately or import existing DB or PA schemas.
- **Data Objects View**
  Displays the User-Defined Data Objects folder, the Non-IDL Types folder, and the Build Configuration folder. Use this view if you are working primarily with data objects, and do not need to create business objects or import existing DB or PA schemas.
- **Schema View**
  Displays the DBA-Defined Schemas folder, the Non-IDL Types folder, and the Build Configuration folder. Use this view if you are working primarily with existing DB schemas, and do not need to work with data objects separately or create business objects.

The views hide other information, but do not prevent reference to it. For example, while using the data objects view, you can still define a data object attribute with the type of a hidden business object interface.

You can customize these views, or add new ones. For example, you could create a new view for working primarily with PA schemas, or for packaging (showing only the Build Configuration and Application Configuration folders).

RELATED CONCEPTS
"Object Builder" on page 1
"Component Assembly" on page 16

RELATED TASKS
"Filter the Tasks and Objects Pane""Create a Filter for the Tasks and Objects Pane" on page 10
"Search the Tasks and Objects Pane" on page 10

Filter the Tasks and Objects Pane

You can apply a filter to the Tasks and Objects pane in Object Builder, to show only the tasks that you are doing or the objects that you are using.

To apply a filter, follow these steps:

1. From Object Builder’s menu bar, select **View - Set Filter**.
2. Select a filter from the cascade of available views. Your selection is indicated with a checkmark.
3. Select **View - Turn Filter On**.

   The filter you selected is applied to the Tasks and Objects pane. You can switch filters at any time, and turn the currently selected filter off and on.

You can customize the existing filters, or add new filters as required.

While a filter is in effect, some menu items may be unavailable. This product’s task documentation assumes an unfiltered view.

RELATED CONCEPTS
“Object Builder” on page 1
“Filters” on page 9

RELATED TASKS
“Create a Filter for the Tasks and Objects Pane”

Create a Filter for the Tasks and Objects Pane

You can create your own filter for the Tasks and Objects pane, to hide folders or objects that you are not using. Once you create a filter, it is available from the **View - Set Filter** menu, and you can turn it on or off with the **View - Turn Filter On/Off** menu choice.

To create a filter, follow these steps:

1. From Object Builder’s menu bar, select **View - Set Filter - Create New**. The Filter Tree window opens.
2. Select one of the existing filters from the filter list, to use as a starting point.
3. Select the folders and objects you want included in the new view. Folders or objects that are **not** checked will be excluded by the filter.
4. Click **Save As**. The Save Filter Scheme window opens.
5. Name the filter.
6. Click **OK**.
7. Click **Finish**.

The new filter now appears in the **View - Set Filter** menu, and is automatically selected and applied.

RELATED CONCEPTS
“Object Builder” on page 1
“Filters” on page 9

RELATED TASKS
“Filter the Tasks and Objects Pane” on page 9

Search the Tasks and Objects Pane

When you are working with a large or complex application, it can be difficult to locate a particular component object or element of the application in Object Builder. To find a particular item in the Tasks and Objects pane, follow these steps:

1. Select **Edit - Find**. The Find dialog box opens.
2. Type the name of the item in the **Find Next** field.
3. Set any search options you want in effect:
   - **Match case**
     Only find items with names that have the same capitalization as the name you specified.
   - **Whole word**
     Only find items with the exact name you specified. Do not find items whose names merely include the string you specified.
   - **Wrap**
     Search the entire pane. If you do not select **Wrap**, the search only occurs from the currently selected item to the bottom of the pane.
4. Click **Find Next**.

   The first item with a matching name is selected in the Tasks and Objects pane. The tree view is expanded as necessary to show the item.
5. When you are finished searching, click **Cancel** in the Find dialog box.

The **Find** function will not find any items that do not appear in the Tasks and Objects pane. If you have applied a filter to the pane, the search will not find items that are excluded from the pane by the filter.

RELATED CONCEPTS
"Object Builder" on page 1

RELATED TASKS
"Filter the Tasks and Objects Pane" on page 9

Run Object Builder in Batch Mode

You can generate code from Object Builder from the command line, using the obgen.bat utility. You can also import and export XML files in batch mode. Those tasks are described in the Import XML and Export XML tasks.

The utility has the following command-line syntax:

obgen -p<*project_directory*> [-d <*destination_directory*>] -a<*object_type*> [-changed] [-linked] [-t<*platform_name*>]

The options are as follows:
- -ProjectDirectory (required)
  -DestinationDirectory (optional)
- -Artifact (required)
- -Changed (optional)
- -Linked (optional)
- -Target (optional)

-ProjectDirectory
The project directory you want to generate code for.

For example: -pE:\myproject

-DestinationDirectory
The directory you want to generate code into. By default, code is generated into the listed project's \Working directory.

-Artifact
The type of objects you want to generate. The object type must be one of the following:

- **All**
  Generate all objects in the project (equivalent to selecting **Generate - All** from the pop-up menus of the User-Defined Business Objects folder, the Build Configuration folder, and the Application Configuration folder).
- **BO**
  Generate all business objects (equivalent to selecting **Generate - Selected** from the pop-up menu of each business object interface and business object implementation).
- **DO**
  Generate all data objects (equivalent to selecting **Generate - All** from the pop-up menu of the User-Defined Data Objects folder).
- **Make**
  Generate all makefiles (equivalent to selecting **Generate Makefiles** from the pop-up menu of the Build Configuration folder).
- **SM**
  Generate all SM DDL (equivalent to selecting **Generate - All** from the pop-up menu of the Application Configuration folder).

-Changed
Generate only code for files that have changed. By default, all files are regenerated.

-Linked
Generate code for the current project only (not for projects listed as dependencies). Generate makefiles that refer to the \Working directories of other projects for any dependencies on other projects. This is equivalent to setting the **Team Environment** option in the Object Builder Preferences notebook (on the Tasks and Objects page of the notebook).

If you do **not** specify the **-linked** option, then code is generated for objects in the specified project, and for projects listed as dependencies, and for their dependencies, and so on. This is **not** equivalent to the **Standalone Environment** option in Object Builder: when you generate code from within Object Builder, for a standalone environment, only objects in the current project, and their direct dependencies, are included.

For example, the command:

```
obgen -pe:\myproject -aBO
```

generates the code for all business objects in the project e:\myproject , code for business objects in projects it depends on, and code for projects they depend on, until all dependencies are fulfilled. The generated code for all projects is placed in the e:\myproject\Working directory.

-Target
Specify platforms to generate code for. By default, code is generated for the current platform only. Options are **NT**, **AIX**, and **390**.

RELATED CONCEPTS
“Projects and Models” on page 4

RELATED TASKS
“Generate Code” on page 363
“Generate a Makefile” on page 367
“Generate the Install Image” on page 379
“Export XML” on page 224
“Import XML” on page 225
“Set Object Builder Preferences” on page 7

Import C++ or Java Classes

If you have an existing C++ or Java class that you want your component to use, and you do not want to create an IDL interface for it, you can import the class into Object Builder as a non-IDL type.

Once the class is imported, you can select it as an interface type within Object Builder (that is, as a method return type, method parameter, or attribute type).

Note: Because this is not an IDL type, it cannot be accessed through the distributed environment. The component’s managed object will not expose methods or attributes that use this type. Methods or attributes that use the type should be added to the business object implementation, not the business object interface.

To import a non-IDL type, follow these steps:
1. In the Tasks and Objects pane, find the Non-IDL Types folder.
2. From the folder’s pop-up menu, click **Import Non-IDL Type**. The Import Non-IDL Type wizard opens to the Name and Language Page.
3. Type the name of the class you want to import.
4. Select whether the class is implemented in C++ or Java.
5. Provide implementation details for the class:
   - For a C++ class, provide the name of the header file that defines the class and the name of the library file that contains its object code. Include the file extensions.
   - For a Java class, provide the name of the package.
6. Click **Finish**.

The class now appears in the Non-IDL Types folder, and you can select it as the type of a method parameter, a method return type, or an attribute type.

RELATED CONCEPTS
Progamming Languages and Conventions (*Programming Guide*)

RELATED TASKS
“Add a Business Object Interface” on page 283

Chapter 2. Component Overview

Components

In Component Broker, a component consists of a distributed set of objects that client applications access as a single entity. To a client application, a component appears to be a single class, with methods and attributes and relationships like any other class. Behind this single interface, however, each component consists of multiple objects on both the client and the server. This separation provides flexibility and control in the way data is stored and accessed, and in the way that business processes are distributed. The objects can exist on any number of different servers and databases, but to the client they present a single interface, with a single set of attributes.

Typically, a component consists of the following objects in Object Builder:
- business object interface
- business object implementation
- data object interface
- data object implementation
- persistent object and schema
- key
- copy helper
- managed object

At execution time, a call to the component (for example, a call to a CarPolicy component to get the value of the attribute make) resolves in the following order:
1. The managed object accepts the call, and calls its associated container for object services before passing the call on to the business object.
2. The business object accepts the call, and either returns the value of the attribute based on a cached copy of the data, or delegates the call to the data object.
3. The data object accepts the call, and either returns the value of the attribute based on a cached copy of the data, or delegates the call to the persistent object.
4. The persistent object retrieves the value from a database, and returns it.
5. The value is returned up the component tree until it reaches the managed object, which calls the container again for object services before returning the value to the caller.

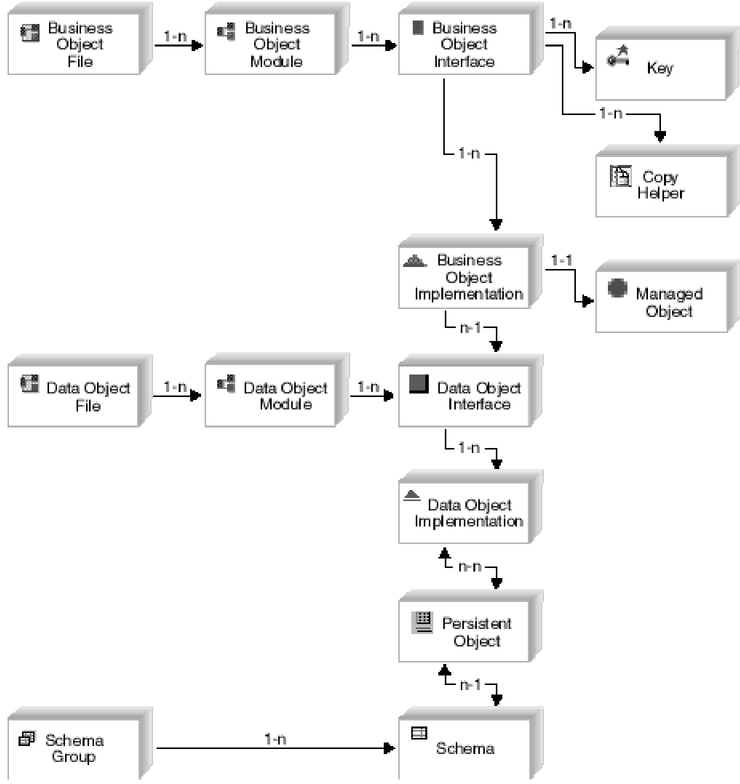
A component has no real existence: it is merely a convenient way to refer to a set of related objects.

RELATED CONCEPTS
"Object Builder" on page 1
"Business Object" on page 17
"Data Object" on page 18
"Persistent Object" on page 19
"Schema" on page 20
"Key" on page 21
"Copy Helper" on page 21
"Managed Object" on page 22

© Copyright IBM Corp. 1997, 1998

Component Assembly

When you assemble a component, you can start from the business object, data object, or schema. The following diagram shows the relationships among component objects as you assemble them. When you configure the objects into a unified component, you select a subset of these objects to form a particular component on the server. The deployed component is accessed through its managed object, by client applications or other components that require access to the server data.



RELATED CONCEPTS

"Components" on page 15
"Component Execution"

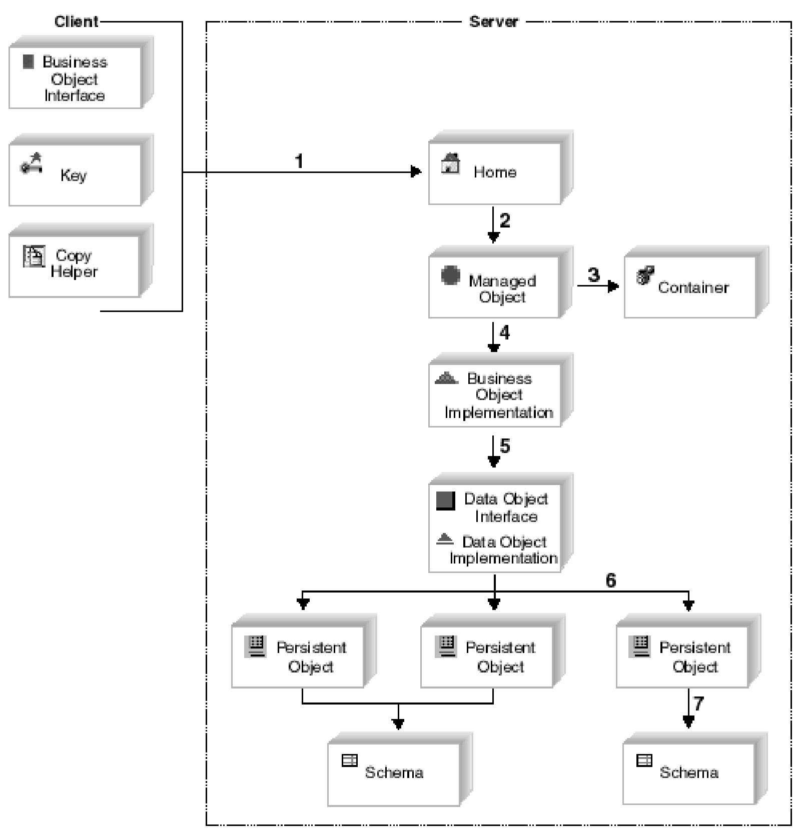
Component Execution

At execution time, a call to the component on the server (for example, a call to a CarPolicy component to get the value of the attribute make) resolves in the following order:

1. The managed object accepts the call (from the client or from another component), and calls its associated container for object services before passing the call on to the business object.
2. The business object accepts the call, and either returns the value of the attribute based on a cached copy of the data, or delegates the call to the data object.
3. The data object accepts the call, and either returns the value of the attribute based on a cached copy of the data, or delegates the call to the persistent object.

4. The persistent object retrieves the value from a database, and returns it.
5. The value is returned up the component tree until it reaches the managed object, which calls the container again for object services before returning the value to the caller.

Execution



RELATED CONCEPTS

"Components" on page 15
"Component Assembly" on page 16

Objects

Business Object

A business object represents a business function. Business objects contain attributes that define the state of the object, and methods that define the behavior of the object. A business object also has relationships with other business objects. It can cooperate with other business objects to perform a specific task. Business objects are independent of any individual application. They can be used in any combination to perform a desired task. Typical examples of business objects are: Customer, Invoice, or Account.

In Component Broker, a business object functions as part of a component, which is a collection of related objects that work together to represent the logic and data relationships of the business function.

A business object's interface is defined in an IDL file. Its implementation can be in either C++ or Java. You can set the default implementation language for business objects in the Preferences notebook, on the Tasks and Objects page. You can set the implementation language for a specific business object in the Business Object Implementation wizard, on the Implementation Language page.

RELATED CONCEPTS
"Components" on page 15

RELATED TASKS
"Add a Business Object Interface" on page 283

State Data

Every object has a state and a behavior, and presents an interface. An object's behavior is manifested in the implementation of the methods on the object's private and public interfaces. An object's state is manifested in its public and private data members and can be divided into two categories: essential and non-essential. The essential state makes up the state data of an object and consists of data that is persistent and not calculated or derived from other data members. The non-essential state consists of transient data that can be recreated as required. The non-essential state is usually derived from other state data and complements the essential state.

RELATED CONCEPTS
"Data Object"

RELATED TASKS
"Add a Business Object Implementation and Data Object Interface" on page 284

Data Object

A data object is responsible for managing the persistence of a component's essential state information (state data). It provides an interface for getting and setting the state data.

A data object isolates its business object from having to:

1. Know which of many datastores to use to make its state persistent.
2. Know how to access the datastore.
3. Manage access to the datastore.

A data object has two parts: the data object interface, which defines the state data of the component, and the data object implementation, which defines the form of persistence and access patterns for the data.

RELATED CONCEPTS
"Components" on page 15

RELATED TASKS
"Add a Business Object Interface" on page 283
"Add a Data Object Implementation" on page 299

Persistent Object

A persistent object is a C++ object that provides a mechanism for storing a component's state in a datastore. Every persistent object has an identifier or a key that is used for locating its corresponding record within the datastore.

There are two kinds of persistent objects: DB persistent objects and PA persistent objects.

Database (DB) Persistent Objects
This type of persistent object represents a record of a table or a view in a relational database. The component's state data that is stored in the relational database by means of a persistent object lasts longer than the execution time of the application that calls the component.

In a relational database such as an SQL database, all records are persistent because they are stored on disk in the form of database tables. A datastore, whether it is an Object-Oriented Database Management System (OODBMS) or a Relational Database Management System (RDBMS), stores an object's persistent data.

There are two kinds of DB persistent object implementations:

- Persistent objects that use embedded static SQL to access and update the data they represent in a database. The generated files are of type .hpp and .sqx. You can edit the embedded SQL clauses that Object Builder provides for the methods of these objects, but Object Builder will not validate your entries.
- Persistent objects that use the caching capabilities of the server for accessing and updating data from the datastores they represent. The generated files are of type .hpp and .cpp.

In Object Builder, you can create these type of objects at the time of schema creation (top-down) or after a schema has been imported into Object Builder (bottom-up).

Procedural Adaptor (PA) Persistent Objects
This type of persistent object encapsulates not only the data associated with an application but also the application's transaction logic. It is usually an embodiment of IMS and CICS transactions and data. A PA persistent object too is responsible for storing the data and transactions of a reusable application (which is bundled together as a component), much longer than the execution time of the application that calls the component.

Object Builder creates a PA persistent object for every Procedural Adaptor bean that is imported. The Procedural Adaptor bean exists as the PA schema in Object Builder and you can create additional PA persistent objects for a PA schema.

The generated files for a PA persistent object are of type .hpp and .cpp.

RELATED CONCEPTS
"Schema" on page 20
"Procedural Adaptor Bean (PA Bean)" on page 117
Cache Service (*Advanced Programming Guide*)
Session Service (*Advanced Programming Guide*)
Transaction Service (*Advanced Programming Guide*)

RELATED TASKS

"Add a Persistent Object and Schema" on page 313
"Add a Persistent Object from a DB Schema" on page 316"Customize Referential Integrity" on page 108
"Add a Persistent Object from a PA Schema" on page 334

Schema Group

A schema group is an organization of the different database schemas that you either create for a data object or import from the DDL file.

When the generate action is used on a schema group, an SQL file is created. It contains the subset of the definitions of the schemas that Object Builder requires to do table to persistent object mapping.

Note the following points about schema groups in Object Builder:

- All schemas in a schema group must belong to the same database type. That is, within a schema group, you cannot have some schemas that are of the DB2 database type and others that are of the Oracle database type. However, you can arrange schemas that exist in the same database, into different schema groups.
- Schema groups cannot be nested.

Note the following points about schema group names:

- Group names must contain only alphanumeric characters, the blank space, and the underscore.
- They are case sensitive.

RELATED CONCEPTS

"Schema"
"Persistent Object" on page 19

RELATED TASKS

"Work with DB Schema Groups" on page 318

Schema

A schema is a description of the structure of a table or a view in a relational database. It is a structural and behavioral abstraction of the real physical data, and focuses on information relevant to users of the applications that use the database.

In CBConnector, DB2 is the relational datastore supported, and the schemas are described using the SQL language.

In Object Builder, there are two kinds of schemas: database (DB) schemas and procedural adaptor (PA) schemas.

DB schemas can either be created from data object implementations (the top-down approach) or imported from SQL DDL files (the bottom-up approach). For organizational purposes, DB schemas are found in schema groups. All schemas within a group must be of a single database type. That is, they must all either be of the DB2 database type, or of the Oracle database type.

Follow these rules when you name a DB schema:

- The name must not exceed 18 characters for DB2; 30 characters for Oracle.
- All alphanumeric and DBCS characters are allowed, and there's no case sensitivity for names containing these characters.
- Non-alphanumeric names must be enclosed in double-quotes, and their case is maintained internally.

PA schemas are created when a procedural adaptor bean is imported into Object Builder from Enterprise Access Builder (EAB).

RELATED CONCEPTS
"Schema Group" on page 20
"Procedural Adaptor Bean (PA Bean)" on page 117

RELATED TASKS
"Add a Persistent Object and Schema" on page 313
"Create a DB Schema by Importing an SQL File" on page 321
"Edit a DB Schema Group" on page 319
"Create a PA Schema by Importing a PA Bean" on page 337

Key

A component's key object defines which attributes are to be used to find a particular instance of the component on the server. The key consists of one or more of the business object attributes, which must contain enough information to uniquely identify an instance.

The key is defined as a separate class (rather than simply flagging one or more of the attributes directly on the business object) for two reasons. First, CORBA does not permit passing a mix of different data types in one call; by defining the key in a separate class, you can mix multiple attributes and data types in whatever combination you require. Second, you gain the flexibility of changing the key, or having more than one key for different situations, without affecting the rest of the component.

When you define a key in Object Builder, its implementations are generated in both Java and C++.

RELATED CONCEPTS
"Components" on page 15
"Composite Key" on page 176

RELATED TASKS
"Work with Keys" on page 292

Copy Helper

A copy helper is an optional object that provides an efficient way for the client application to create new instances of the component on the server. The copy helper contains the same attributes as the business object, or a subset of them. Without a copy helper, the client might need to make many calls to the server for each new instance: one call to create the instance, and then an additional call to initialize each of the instance's attributes. With a copy helper, the client can create a local instance of the copy helper, set values for its attributes, and then create the server component and initialize its attributes in one call, by passing it the copy helper.

When you define a copy helper in Object Builder, its implementations are generated in both Java and C++.

Copy helper instances are created using the _create function. The client developer sets values locally, then creates a managed object from the copy helper using the createFromCopy function.

RELATED CONCEPTS
"Components" on page 15

RELATED TASKS
"Add a Copy Helper" on page 294

Managed Object

A class of objects that defines the set of methods that must be implemented by the business object to work with the appropriate application adaptor. Managed objects are enabled to work with two level storage containers and delegate to the data object the attributes of the object.

A managed object represents the component to the client application, and handles all calls from the client to the component on the server.

An application is defined by adding and configuring managed objects. By creating a managed object for a business object, you specify that it will be installed on the server. The managed object handles communication with other classes, and initialization, de-initialization, activation, and passivation of the business object.

RELATED CONCEPTS
"Components" on page 15

RELATED TASKS
"Work with Managed Objects - Overview" on page 339

Key Assistant

A key assistant is a new key helper class. It is a concrete subclass that is associated with a managed object assembly and has knowledge of the primary key that is configured for the assembly.

The interface of the key assistant supports creating keys from various existing objects. Currently, Object Builder supports creation of primary keys from a copy helper or a data object.

Object Builder optimizes performance by creating multiple copies of proxies of an object that every client can access. The way it does this is by generating a set of new IManagedServer::IKeyAssistant objects which introduce new sets of idl, ih and .cpp files.

RELATED CONCEPTS
"Key" on page 21
"Copy Helper" on page 21"Data Object" on page 18
"Managed Object"

Methods and Attributes

User-Defined Methods

You can define methods on the following objects:

- Business object interfaces
  Methods you define here are available to other components and applications, through the managed object. The method implementation is defined in the business object implementation.
- Business object implementations
  Methods you define here are specific to the implementation, and not exposed in the component's interface.
- Data object interfaces
  Methods you define here are available to other objects, but are not exposed in the managed object. The method implementation is defined in the data object implementation.
- Data object implementations
  Methods you define here are specific to the implementation, and not exposed in the component's interface.

Once you define a method on an object, it appears in the Methods pane when you click on the object.

When you define a method for an interface, its definition is automatically added to any associated implementation objects.

To provide the method body for a method, click on the business object implementation or data object implementation, and then click the method in the Methods list. You can now type the method body directly in the Source pane.

You can also provide a method body by referencing an external file in the method's Method Implementation wizard (accessed from the method's pop-up in the Methods pane). The external file can be a template, with macros that you can substitute values for.

RELATED CONCEPTS
"Business Object" on page 17
"External Files for Method Bodies" on page 273

RELATED TASKS
"Add a Business Object Interface" on page 283
"Add Code for User-Defined Methods" on page 267
"Add an Initializer Method" on page 268
"Edit a User-Defined Method" on page 269
"Delete a Method" on page 277

Get and Set Methods

Object Builder adds get and set methods to objects for each public attribute you define, as follows:

- Business object implementation:
  Has get and set methods for each public attribute in the business object interface.

- Key:
  Has get and set methods for each attribute that makes up the key.
- Copy helper:
  Has get and set methods for each attribute that makes up the copy helper.
- Data object implementation:
  Has get and set methods for each attribute defined in the data object interface.
- Persistent object:
  Has get and set methods for each attribute in the data object that it provides persistence for.

The implementations for get and set methods are provided by Object Builder, although you can edit them if necessary.

RELATED CONCEPTS
"Attributes" on page 26
"Special Framework Methods"

RELATED TASKS
"Edit Get and Set Methods" on page 270

Framework Methods

Framework methods are added to an object by Object Builder. Generally, framework methods are only called by other framework methods, or by Component Broker services.

Framework methods provide the functionality your objects need to work in a Component Broker distributed environment.

There are also special framework methods, which are a particular kind of framework method that let a component access its persistent data.

RELATED CONCEPTS
"Special Framework Methods"

RELATED TASKS
"Edit Framework Methods" on page 270

Special Framework Methods

Data objects and persistent objects that access a schema have the special framework methods insert, update, retrieve, del, and setConnection. The implementations for these methods are calculated based on the mapping between the persistent object methods and the data object methods.

The order in which the data object methods call their equivalent methods in persistent objects can affect the integrity of the references.

RELATED TASKS
"Edit Special Framework Methods" on page 271
"Customize Referential Integrity" on page 108

Push-Down Methods

Push-down methods are those that are “pushed down” from the persistent object to the data object and finally to the business object. Depending on whether they are used along with DB persistent objects, or PA persistent objects, it is either the Relational Database Application Adaptor (RDBAA), or the Procedural Application Adaptor (PAA) that handles the pass-through processing.

In Object Builder, push-down methods are used to expose functionality to the client. In particular, they are used for the following purposes:

- To transmit transactional data of existing applications (when they are used with PA persistent objects)
- To transmit data that is contained in databases used by existing applications (when they are used with DB persistent objects, or when they are used as stored procedures).

Note: Push-down methods in Object Builder are editable at the data object implementation level.

In Enterprise Access Builder (EAB), a push-down method is one that is written on the procedural adaptor bean. In Object Builder, it is a mapping to the implementation of the method in EAB.

When these methods are executed using the HOD mechanism for accessing IMS applications, the changes resulting from their execution are visible to other sessions.

When these methods are executed using the ECI mechanism for accessing CICS applications, the changes resulting from their execution are not visible to other sessions.

RELATED CONCEPTS

“Enterprise Access Builder (EAB)” on page 116
“Persistent Object” on page 19
“Data Object” on page 18
An Overview of Application Adaptors (*Programming Guide*)

RELATED TASKS

“Use Push-Down Methods with PA Persistent Objects” on page 274

Relationship Methods

When you define a relationship from one business object to another, there is a set of methods created for the relationship. They are the add, list, and remove methods, that enable you to access the relationship.

For example, if you have a one to many relationship between the Policy business object and the Claim object, you can use the add method to add claims for a policy, the list method to list claims in the policy, and the remove method to remove claims from a policy.

You can customize the implementation of the list method by providing your own OO-SQL code for it. (Use the OO-SQL Customization Page of the Method Implementation wizard.)

RELATED CONCEPTS

"Business Object" on page 17

RELATED TASKS

"Add a Business Object Implementation and Data Object Interface" on page 284
"Create a Relationship" on page 129
"Customize Business Object OO-SQL Implementation Methods" on page 275

Attributes

Public attributes of a component are defined in the business object interface. You can define protected or private attributes in the business object implementation. When you change an attribute in a business object interface, the change is applied automatically to the business object implementation, and is applied to the key, copy helper, and managed object the next time you edit them (open and finish their properties wizard). When you change an attribute in a data object interface, the change is applied automatically to the data object implementation.

You can also define implementation-only attributes in a Business Object Implementation wizard or Data Object Implementation wizard. These attributes are not available in the IDL and are not exposed in the managed object for the component.

Attributes are defined in component objects as follows:

- Behavior:
  - Business object interface: IDL attributes
  - Business object implementation: get and set methods, in C++ or Java
  - Key: get and set methods, in C++ and Java
  - Copy helper: get and set methods, in C++ and Java
- Data:
  - Data object interface: IDL attributes
  - Data object implementation: get and set methods, in C++
  - Persistent object: get and set methods, in C++
  - Schema: table columns in a database, or methods of a procedural adaptor bean.

RELATED CONCEPTS

"Components" on page 15
"Get and Set Methods" on page 23

RELATED TASKS

"Work with Attributes" on page 247

Constructs

In Object Builder, the following items, are referred to throughout as constructs:

- Constant
- Enumeration
- Exception
- Typedef
- Structure

- Union

They can be defined at the file, module, or interface level of a business object interface or data object interface, or at the file or module level of a composition.

RELATED TASKS
"Work with Constructs" on page 277

Business Object Behavior

Business object behavior encompasses the pattern to be used to handle the essential state of the business object, how the business object handles object references, whether the endResource() method is to be called on the object if it is a sessional business object, whether the object is to be associated with a data object, and the platform on which the business object is to be deployed.

A business object interface can have multiple implementations, depending on the quality of service required.

The following reference topics deal with business object behavior:
- "Pattern for Handling State Data"
- "Object Reference" on page 29
- "Data Object Interface" on page 29
- "Session Service" on page 30

You can specify all these details when you define an implementation for the business object, on the Name and Data Access Pattern Page of the Business Object Implementation wizard. This page has the same sections as the reference topics listed. In addition, besides providing the names for the business object implementation's file, module and interface, you can also specify the platform on which the object is to be deployed. You can select from one or more of Windows NT, AIX, and OS/390.

RELATED CONCEPTS
"Business Object" on page 17
"Data Object" on page 18
Object Relationships (*Programming Guide*)
RELATED TASKS
"Add a Business Object Implementation and Data Object Interface" on page 284

Pattern for Handling State Data

The implementation of the business object must have the specifications as to how the object has access to state data (data that is persistent). A business object can either have a part to play in the maintenance of its state, or it can pass on that responsibility entirely to its associated data object.

You can make this decision using the Name and Data Access Pattern page of the Business Object Implementation wizard when you add a business object implementation (**Add Implementation** from the pop-up menu of the business object interface in the User-Defined Business Objects folder, in the Tasks and

Objects pane), or when you edit the object's implementation that you have defined (**Properties** from the pop-up menu of the business object implementation in the User-Defined Business Objects folder).

The Pattern for Handling State Data section on this page has the following options:

- None
- Delegating
- Caching
- Same as parent's

None
This pattern implies that there is no data object supporting the business object, and no essential state in the business object.
Note: This option is not yet available.

Delegating
Select this radio button if you want to use the IManagedObjectWithDataObject class as the data access pattern. The maintenance of the business object's state is delegated to the data object. The essential state is passed to the data object, which sends it back to the business object. All non-derived, non-essential state is still stored (cached) in the business object.

Caching
This is the default pattern. This pattern uses the IManagedObjectWithCachedDataObject class for data access. Both the business object and the data object cache a local copy of the essential state. All essential state, and all non-derived, non-essential state, is cached in the business object. The business object does not delegate any calls to the data object. Additional framework methods syncToDataObject() (which is used to load the business object with data contained in the data object), and syncFromDataObject() (which is used to send data from the business object back to the data object) are used to keep the cached copy of the attributes in correspondence with the data object attributes. Once this option is selected, the **Object Reference** section is activated.

Same as parent's
The pattern for handling state data, which is used by the parent of this interface, will be used for this implementation.
Note: This option is selected by default if the interface for this business object inherits from another business object interface. However, you still have to indicate the parent on the Implementation Inheritance Page of this wizard, after you delete the default parent for business object implementations, which is IManagedClient IManagedClient::IManageable.

RELATED CONCEPTS
"Business Object" on page 17
Object Relationships (*Programming Guide*)
Cache Service (*Advanced Programming Guide*)

RELATED TASKS
"Add a Business Object Implementation and Data Object Interface" on page 284
"Add a Business Object Interface" on page 283
"Create a Customized Home" on page 343
"Create a Container Instance" on page 346

Object Reference

When you specify **Caching** as the pattern to be used for handling the essential state of the business object, you can select the **Use lazy evaluation** check box if you want the first copy of object references in the essential state to be fetched only when it is required, rather than automatically at startup.

You can make this decision using the Name and Data Access Pattern page of the Business Object Implementation wizard when you add a business object implementation (**Add Implementation** from the pop-up menu of the business object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the object's implementation that you have defined (**Properties** from the pop-up menu of the business object implementation in the User-Defined Business Objects folder).

RELATED CONCEPTS

"Business Object" on page 17
Object Relationships (*Programming Guide*)
Cache Service (*Advanced Programming Guide*)

RELATED TASKS

"Add a Business Object Implementation and Data Object Interface" on page 284
"Add a Business Object Interface" on page 283
"Create a Customized Home" on page 343
"Create a Container Instance" on page 346

Data Object Interface

You can choose to have Object Builder create a data object interface along with the business object implementation you are defining, or you can create, or select one later.

You can make this decision using the Name and Data Access Pattern page of the Business Object Implementation wizard when you add a business object implementation (**Add Implementation** from the pop-up menu of the business object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the object's implementation that you have defined (**Properties** from the pop-up menu of the business object implementation in the User-Defined Business Objects folder).

You can select one of the following radio buttons:

- Create a new one now
- Add or select one later

Create a new one now
Select this option if you want the data object to be derived from the business object. The data object is automatically created when you add a business object implementation. This is the default option.

Add or select one later
Select this option when you want to reuse an existing data object, which is stand-alone and not derived from a business object. This option enables you to match the interface and function requirements of the newly created top-down model with the classes developed from existing data.

RELATED CONCEPTS
"Business Object" on page 17
RELATED TASKS
"Add a Business Object Implementation and Data Object Interface" on page 284
"Add a Business Object Interface" on page 283

Session Service

390: This section is not applicable, and therefore not available when the development platform is OS/390.

Use this section if you plan to make the business object sessional. When a business object uses Session Service, you can provide your own code to be called during some of the normal processing for those services. To do this, select the **Provides end resource** check box in this section.

Provides end resource
Select this check box to indicate that the business object implementation inherits from ISessions::Resource, the class that has the endResource() method in it. Object Builder creates the endResource() method on the business object, and you can provide your own code for it. Your code will be called when the endResource() method is called on the managed object's mixin.

RELATED CONCEPTS
"Business Object" on page 17
Session Service (*Advanced Programming Guide*)

RELATED TASKS
"Add a Business Object Implementation and Data Object Interface" on page 284
"Add a Business Object Interface" on page 283
"Create a Customized Home" on page 343
"Create a Container Instance" on page 346
"Add endResource() to a Sessional Business Object" on page 117

Data Object Behavior

The behavior of a data object depends on various factors such as the environment for the business object, the implementation type of the data object and its storage options, and the pattern used by the data object for data access and storage of references.

The following reference topics deal with different aspects of the behavior of data objects:

- "Environment" on page 31
- "Form of Persistent Behavior and Implementation" on page 32
- "Data Access Pattern" on page 34
- "Handle for Storing Pointers" on page 35

You can specify all these details when you define an implementation for the data object on the Behavior Page of the Data Object Implementation wizard. This page has the same sections as the reference topics listed.

RELATED CONCEPTS
"Data Object" on page 18
"Persistent Object" on page 19
Data Object Customization

RELATED TASKS
"Work with Data Objects - Overview" on page 296

Environment

The environment for a component has to be either conducive to testing, or to production or deployment. The Unit test environment is for testing the business object; the other environments that use the Business Object Application Adaptor (previously known as BOIM) are for production, when we implement a real application with persistent data.

You can select the environment for your component in the Behavior Page of the Data Object Implementation wizard either when you add a data object implementation (**Add Implementation** from the pop-up menu of the data object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the data object implementation you have defined (**Properties** from the pop-up menu of the data object implementation in either the User-Defined Business Objects folder, or the User-Defined Data Objects folder).

You have the following environment options:
- Unit test
- BOIM with UUID key
- BOIM with any key
- Same as parent's

Unit test
Select this environment to unit test the data object without configuring and installing applications on the server. The business object is not loaded into the CBConnector server; it can be tested on the CBConnector client machine. The form of persistent behavior and implementation is automatically set to **Transient**, and cannot be changed. This is so that the essential state of the business object is not saved across executions of the unit test program. A local copy of the essential state is used for data access. The unit test reference collection is used to store the object.

390: This option is not available when the target platform includes OS/390.

BOIM with UUID key
This environment uses the Business Object Instance Manager (BOIM), commonly known as Business Object Application Adaptor, with the Universally Unique Identity (UUID) key. Select this implementation to create unique server data objects with transient data for supporting the business object. This option is useful for short-lived business objects that do not have to persist after your application has finished executing. The form of persistent behavior and implementation is automatically set to **Transient**, and cannot be changed. A local copy of the essential state is used for data access.

BOIM with any key
Select this implementation to create server data objects with persistent data for supporting the business object. The data object will be installed in a business object

application adaptor, and instances of the object will be located using keys. This is the option to select if you want to use a relational back-end datastore, as shown in the Life Insurance example. The set methods for the data object's attributes have the markDirty() method, which informs the application adaptor when the underlying datastore has to be updated. This environment enables you to create a persistent object or use an existing one. All the options in the **Form of Persistent Behavior and Implementation** section are available for selection. The default form for persistent behavior is set to **Embedded SQL**.

Note: If you use this option and create any persistent objects for this data object, you must use a customized container instance. The default container instances are only appropriate for objects with transient data.

Same as parent's
Select **Same as parent's** when you want to use the implementation type that is specified for the parent of this interface. The datastore defined in the parent is used. If the parent has no persistent object, this newly created data object has no persistent back-end. However, if the parent uses **Embedded SQL,** the newly created data object inherits that behavior. A local copy of the essential state is used for data access.

Note: If you are defining an implementation that inherits from another, this option will be selected.

RELATED CONCEPTS
"Data Object" on page 18
"Persistent Object" on page 19
Application Adaptor (*Programming Guide*)
Data Object Customization (*Programming Guide*)
"Container" on page 345
"State Data" on page 18
Cache Service (*Advanced Programming Guide*)
Using Sets of Objects (Using Reference Collections) (*Programming Guide*)

RELATED TASKS
"Work with Data Objects - Overview" on page 296
"Add a Persistent Object and Schema" on page 313
"Customize Referential Integrity" on page 108
"Create a Container Instance" on page 346
"Configure a Managed Object" on page 377

RELATED REFERENCES
"Form of Persistent Behavior and Implementation"
"Data Object Implementation Inheritance" on page 36

Form of Persistent Behavior and Implementation

The data object implementations you define differ from one another based on whether the associated data object is persistent or not, and on the type of service they use.

You can select the form of persistent behavior and implementation in the Behavior Page of the Data Object Implementation wizard either when you add a data object implementation (**Add Implementation** from the pop-up menu of the data object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the data object implementation you have defined

(**Properties** from the pop-up menu of the data object implementation in either the User-Defined Business Objects folder, or the User-Defined Data Objects folder). Each form of persistent behavior and implementation has a unique impact in terms of application performance, allocation of resources, and so on.

To be able to select any one of the different types of implementations, you must first select, on the same page of the wizard, the **BOIM with any key** environment. See "Environment" on page 31**.**

You have a choice of the following implementations:

- Transient
- Embedded SQL
- DB2 Cache Service
- Oracle Cache Service
- Procedural Adaptors

Transient
This option is automatically selected and is the only one available if the data object does not have to be persistent, as in the case of the **Unit test** environment, or when the data object implementation uses **BOIM with UUID key**. If you selected **BOIM with any key** in step 4, select this option if you want to write your own implementations for a persistent data object.

Embedded SQL
Select this option if embedded (static) SQL is to be used by the persistent object to access the database.

Cache Service
Use the **Cache Service** options if you want the CBConnector server to hold cached copies (instances) of the data object in memory, accessing the corresponding rows in the relational database only when necessary. This results in improved performance when the values in the accessed row need to be read frequently but not updated as frequently. You can select one of the following types of Cache services:

- **DB2 Cache Service**
  Select this option to access rows in a DB2 database.
- **Oracle Cache Service**
  Select this option to access rows in an Oracle database.

390
All **Cache Service** options are not available when the target platform is OS/390.

Procedural Adaptors
Select this option if the data object implementation is to be connected to a persistent object that is created for an imported procedural adaptor (PA) bean.

Note the following points when you configure a managed object for your application:

- You can select only those containers that match the data object implementation (and the managed object). For example, if your data object implementation uses **Embedded SQL**, only those containers that use embedded SQL (those without caching services) are shown. Similarly, if you defined the data object implementation to use **Procedural Adaptors**, and the related persistent object

uses **Session Service**, the selection you made on the Names and Services Page of the Import Procedural Adaptor Bean wizard, only containers that are configured for sessions are shown.

- You will be able to select only those data object implementations in the model (on the Data Object Implementations Page of the Configure Managed Object wizard) that use the service you specified on the Names and Connectors Page of the Import Procedural Adaptor Bean wizard.

RELATED CONCEPTS
"Data Object" on page 18
"Persistent Object" on page 19
Application Adaptor (*Programming Guide*)
Data Object Customization (*Programming Guide*)
"Container" on page 345
"State Data" on page 18
Cache Service (*Advanced Programming Guide*)
Using Sets of Objects (Using Reference Collections) (*Programming Guide*)

RELATED TASKS
"Work with Data Objects - Overview" on page 296
"Add a Persistent Object and Schema" on page 313
"Customize Referential Integrity" on page 108
"Create a Container Instance" on page 346
"Configure a Managed Object" on page 377

RELATED REFERENCES
"Data Object Implementation Inheritance" on page 36

Data Access Pattern

The data access pattern determines how the data object accesses data with the help of its persistent object.

You can set the data access pattern in the Behavior Page of the Data Object Implementation wizard either when you add a data object implementation (**Add Implementation** from the pop-up menu of the data object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the data object implementation you have defined (**Properties** from the pop-up menu of the data object implementation in either the User-Defined Business Objects folder, or the User-Defined Data Objects folder).

Data access patterns for some data object implementations are predestined by Object Builder, depending on the type of service used, or the transient or persistent nature of the implementation.

If the form of persistent behavior and implementation of the data object is **Embedded SQL**, you can select one of the following access patterns:

- Delegating
- Local copy

Delegating
The data object uses the attributes of its associated persistent object. The get and set methods of the data object call the corresponding get and set methods of the persistent object, which, in turn, access the persistent object attributes. A local copy of the data object attributes is maintained in the private members of the data object

class. If the implementation uses either of the **Cache Service** options, or **Procedural Adaptors**, the data access pattern is automatically set to **Delegating**. This setting cannot be changed.

If **Delegating** is used, the essential state is passed from the business object to the data object, which sends it back to the business object. If you select **Local copy**, the data object caches a local copy of the essential state.

Local copy
The data object has its own local copy of its attributes. They are private members of the data object class that can be accessed directly by the data object's methods. That is, the get and set methods are applied to private copies of the attributes. The attributes of the persistent object are set only when you make a call that invokes the database: when you implement any of the special framework methods, for example insert(). The local copy is used for type conversion purposes as in the case when a mapping helper is used to map the attributes of the data object to the attributes of an associated persistent object. If the implementation is **Transient**, the data access pattern is automatically set to **Local copy**, and cannot be changed.

RELATED CONCEPTS
"Data Object" on page 18
"Persistent Object" on page 19
Application Adaptor (*Programming Guide*)
Data Object Customization (*Programming Guide*)
"Container" on page 345
"State Data" on page 18
Cache Service (*Advanced Programming Guide*)
Using Sets of Objects (Using Reference Collections) (*Programming Guide*)

RELATED TASKS
"Work with Data Objects - Overview" on page 296
"Add a Persistent Object and Schema" on page 313
"Customize Referential Integrity" on page 108
"Create a Container Instance" on page 346
"Configure a Managed Object" on page 377

RELATED REFERENCES
"Data Object Implementation Inheritance" on page 36

Handle for Storing Pointers

The design pattern you use for the data object implementation determines how object references are stored (made persistent rather than transient) for later retrieval and use. Object references are in a format that can be stored in a database. These persistent storage forms, when converted back to in-memory pointers, need no further transformation in order to point to the right object. These patterns are implemented using handles that Object Builder generates. The choice of a pattern is based on factors such as speed of execution and storage overhead.

You can select the handle to be used for storing pointers in the Behavior Page of the Data Object Implementation wizard either when you add a data object implementation (**Add Implementation** from the pop-up menu of the data object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the data object implementation you have defined

(**Properties** from the pop-up menu of the data object implementation in either the User-Defined Business Objects folder, or the User-Defined Data Objects folder).

You can select one of the following handles:
- Default
- Stringified object reference
- Object name
- Home name and key

Default
Select this option if you prefer to have the default handle used as the design pattern for swizzling pointers. The default handle is the one you select when you define the implementation for the corresponding business object on the Handle Selection Page of the Business Object Implementation wizard is used.

Stringified object reference (SOR)
Select this option to distribute the object reference in the CORBA environment. This is the string form of an object reference. It helps in externalizing an object to a stream.

Object name
Select this option only for objects named using the Naming Service. An object thus named provides an interface that returns its name.

Home name and key
Select this option to implement specific relationships among CBConnector objects. The handle that the data object uses to store references to other objects is composed of a home that stores the instance of the referenced object and a key that identifies the instance. This option is sometimes preferred over a stringified object reference (SOR) because it takes less storage space, and can be maintained more efficiently: transferring a home from one server to another will not break a home name and key reference, but it would an SOR.

RELATED CONCEPTS
"Data Object" on page 18
"Persistent Object" on page 19
Using Handles (*Programming Guide*)
Naming Service (*Advanced Programming Guide*)
Application Adaptor (*Programming Guide*)
Data Object Customization for Cardinality Relationships (*Programming Guide*)
Object Relationships (*Programming Guide*)
Using Sets of Objects (Using Reference Collections) (*Programming Guide*)

RELATED TASKS
"Work with Data Objects - Overview" on page 296

RELATED REFERENCES
"Data Object Implementation Inheritance"

Data Object Implementation Inheritance

Persistence	Default Parent Implementation	Platforms
Transient	IBOIMExtLocalToServer IBOIMExtLocalToServer::IDataObjectBase	All

DB2 Cache service	IRDBIMExtLocalToServer IRDBIMExtLocalToServer::ICachingServiceDataObject	All
Oracle Cache service	IRDBIMExtLocalToServer IRDBIMExtLocalToServer::ICachingServiceDataObject	All
Procedural Adaptors	IPAAExtLocalToServer IPAAExtLocalToServer::IDataObject	All
Embedded SQL	RDBIMExtLocalToServer IRDBIMExtLocalToServer::IDataObject	NT, AIX
Embedded SQL	IBOIM390LocalToServer IBOIM390LocalToServer::IDataObject	OS/390
Transient(BOIM with UUID Key)	IBOIMExtLocalToServer IBOIMExtLocalToServer::IUUIDDataObject	All

RELATED CONCEPTS

"Data Object" on page 18
"Persistent Object" on page 19

RELATED TASKS

"Work with Data Objects - Overview" on page 296
"Add a Data Object Implementation" on page 299
"Edit a Data Object Implementation" on page 310

Chapter 3. Getting Started with Object Builder

Getting Started with Object Builder

The following scenarios introduce you to some of Object Builder's functionality, in the course of developing and deploying a component with data stored in a DB2 database. These scenarios are derived from the scenarios "Getting Started with C++ Business Objects" and "Getting Started with Java Business Objects", formerly in the Component Broker *Quick Beginnings* book.

You should follow these scenarios in order.

The introductory scenarios are:

1. "Create a Component - Scenario"
2. "Build DLLs or Shared Library Files - Scenario" on page 47
3. "Package an Application - Scenario" on page 50
4. "Install and Run an Application Using InstallShield - Scenario" on page 57
5. "Install and Run an Application - Scenario" on page 61
6. "Trace and Debug an Application - Scenario" on page 65
7. "Uninstall an Application Using InstallShield - Scenario" on page 70
8. "Uninstall an Application - Scenario" on page 71

For additional scenarios, search the online information for "scenario", or look in the book index under "scenario".

RELATED CONCEPTS

"Object Builder" on page 1

Create a Component - Scenario

Objectives
To create a C++ or Java component for new database (DB) data.
To write two methods that manipulate the data.
To generate the code for the component, including the DB schema that defines the data.

Before You Begin
You need the following installed on your system:
- CBToolkit, including Samples
- DB2 Universal Database
- WIN VisualAge for C++ and (for Java applications) VisualAge for Java

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide.*

If you are developing a Java component, make sure your CLASSPATH is set correctly, as described in the following topic:
- "Requirements for Java Development" on page 8

© Copyright IBM Corp. 1997, 1998

Description

This exercise defines the objects required to create a component named “Claim”. For this exercise, you will:

1. Create a new business object file
2. Define a business object
3. Define a data object implementation
4. Define a persistent object and schema
5. Define a managed object
6. Generate the code

Once you have defined the objects and generated the code for them, you can continue to the next scenario:

- “Build DLLs or Shared Library Files - Scenario” on page 47

Creating a New Business Object File

To create a new business object file to hold the interface, the Business Object File wizard is used. This wizard contains pages where you can:

- Specify the name of the business object interface.
- Define any associated constructs that will operate on a file-scope level.
- Define any associated files.
- Include any comments.

For this exercise, the file being added is ClaimFile. The only information required is the file name.

To define a business object file:

1. From the Task and Objects pane, select the **User-Defined Business Objects** folder.
2. Open the pop-up menu of User-Defined Business Objects, and select **Add File**. The Business Object File wizard is opened.
3. Type `ClaimFile` in the **Name** field.
4. Click the **Next** button to continue to the Constructs page.
5. Click the **Next** button to accept the defaults and to continue to the Files to Include page.
6. Click the **Next** button to accept the defaults and to continue to the Comments page.
7. Click the **Finish** button.

The ClaimFile file is displayed in the User-Defined Business Objects folder.

Defining a Business Object

After creating the new business object file, the business object needs to be defined. A fully configured business object consists of:

- A business object interface
- An associated key object
- An associated copy helper object (optional)
- A business object implementation

Defining a Business Object Interface

To define a business object interface for a component, the Business Object Interface wizard is used. This wizard contains pages where you can:

- Specify the name of the business object interface
- Define any associated constructs that will operate on an interface-scope level
- Define any parent classes
- Define any user-defined attributes
- Define any user-defined methods
- Define any relationships to other objects
- Include any comments

For this exercise, the interface being added is Claim. This interface will define:

- The claimNo and state attributes.
- The approve and deny methods.

To define the business object interface:

1. From the U**ser-Defined Business Objects** folder, select the **ClaimFile** business object file.
2. Open the pop-up menu of the ClaimFile business object file, and select **Add Interface**. This opens the Business Object Interface wizard.
3. Type `Claim` in the **Name** field.
4. Click the **Next** button to continue to the Constructs page.
5. Click the **Next** button to accept the defaults and continue to the Interface Inheritance page.
6. Click the **Next** button to accept the defaults and to continue to the Attributes page.
7. Define the user-defined attributes.
   a. Click the **Add Another** button.
   b. Type `claimNo` in the **Attribute Name** field, and ensure that `long` is selected as the attribute type.
   c. Click the **Add Another** button. The `claimNo` attribute is added to the tree, and the fields are filled with default values.
   d. Type `state` in the Attribute Name field, and ensure that `long` is selected as the attribute type.
   e. Click the **Next** button to continue to the Methods page.
8. Define the user-defined methods.
   a. Click the **Add Another** button.
   b. Type `approve` in the Method Name field, and ensure that `void` is selected as the return type.
   c. Click the **Add Another** button. The approve method is added to the tree, and the fields are filled with default values.
   d. Type `deny` in the Method Name field, and ensure that `void` is selected as the return type.
   e. Click the N**ext** button to continue to the Object Relationships page.
9. Click the **Next** button to accept the defaults and to continue to the Comments page.
10. Click the **Finish** button.

The Claim interface is displayed in the User-Defined Business Objects folder.

Defining a Key Object

After defining the Claim interface, define the key client object of the component. The key object defines the attributes needed to find a particular instance of the component on the server. It consists of one or more business object attributes, but these attributes must provide enough information to uniquely identify the instance.

To create a key object, the Key wizard is used. By using this wizard, a key object is defined by assigning a default name and a file name based on the equivalent interface names. This wizard contains pages where you can:

- Specify the key.
- Define any parent classes.
- Modify any existing framework methods.
- Define any additional framework methods.

For this exercise, a key object is added for the Claim business object interface. The ClaimFileKey and ClaimKey objects are created. The attribute used to create the key is the claimNo attribute. This exercise assumes that each claim instance will have a different claim number.

To define the key object:

1. From the **User-Defined Business Objects** folder, select the **Claim** interface.
2. Open the pop-up menu of Claim, and select **Add Key**. This opens the Key wizard.
3. Select the `claimNo` attribute from the Business Object Attributes list.
4. Click the >> button to move this attribute to the Key Attributes list.
5. Click the **Next** button to continue to the Implementation Inheritance page.
6. Click the **Next** button to accept the defaults and to continue to the Summary of Framework Methods page.
7. Click the **Next** button to continue to the Optional Framework Methods page.
8. Click the **Finish** button to accept the defaults.

The ClaimKey key is displayed in the User-Defined Business Objects folder.

Defining a Copy Helper Object

A component can have an optional copy helper class to provide an efficient way for the client application to create new instances of the component on the server. The copy helper object contains all or a subset of the attributes of a business object. Without a copy helper, the client may need to make multiple calls to the server for each new instance.

To create a copy helper object, the Copy Helper wizard is used. This wizard contains pages where you can:

- Specify the copy helper.
- Define any parent classes.
- Modify any existing framework methods.

For this exercise, a copy helper object is being added for the Claim business object interface. The ClaimFileCopy and ClaimCopy are created and the attributes associated with the defined key object are defined.

To define the copy helper object:

1. From the U**ser-Defined Business Objects** folder, select the **Claim** interface.

2. Open the pop-up menu of Claim, and select **Add Copy Helper**. This opens the Copy Helper wizard.
3. Click the **All >>** button to move the attributes from the Business Object Attributes list to the Copy Helper Attributes list.
4. Click the **Next** button to continue to the Implementation Inheritance page.
5. Click the **Next** button to accept the defaults and to continue to the Summary of Framework Methods page.
6. Click the **Finish** button to accept the defaults.

The ClaimCopy copy helper is displayed in the User-Defined Business Objects folder.

Defining a Business Object Implementation
After defining the client objects (business object interface, key object, and copy helper object), the business object implementation needs to be defined. The business object implementation is the first server object. This implementation defines a full class (interface and implementation).

As part of the implementation definition, identify which attributes need to be made persistent. These attributes define the interface to the data object. Each business object implementation can have only one data object interface.

For this task, the Business Object Implementation wizard is used. This wizard contains pages where you can:

- Specify the name and access implementation patterns.
- Define any parent classes.
- Specify the implementation language (C++ or Java).
- Define the key selection.
- Define how references are stored.
- Define the data object interface.
- Review the framework methods implemented by Object Builder for this object.

For this exercise, a business object implementation is added for the Claim business object interface. The ClaimFileBO file and ClaimBO business object implementation are created. You will define the implementation to handle the state data by caching. This implementation creates the syncToDataObject() and syncFromDataObject() methods.

To define the business object implementation and its data object interface:

1. From the **User-Defined Business Objects** folder, select the Claim interface.
2. Open the pop-up menu of Claim, and select **Add Implementation**. This opens the Business Object Implementation wizard to the Name and Data Access Pattern page.
3. Define the implementation.
    a. Set the **Caching** radio button in the Patterns for Handling State Data section. The business object implementation maintains a local copy of the data object attributes instead of delegating calls to the data object.
    b. Set the **Use lazy evaluation** check box. For attributes that are object references, the cached copy of the attributes is synchronized with the attributes of the data object at first use instead of during instantiation.

c. Set the C**reate a new one now** radio button in the Data Object Interface section to add a page to the wizard. This page will be used to define the business object attributes that need to be preserved in the data object.

Note the deployment platform options. By default, the deployment platforms match those selected in the **Platform - Constrain** menu. By selecting a single deployment platform, you can take advantage of a broader range of development options (because you are not limited to cross-platform options). In this exercise, you can ignore these options.

4. Click the **Next** button to continue to the Implementation Inheritance page.
5. Verify that `IManagedClient IManagedClient::IManageable` is selected as a parent.
6. Click the **Next** button to continue to the Implementation Language page.
7. Select **C++** as the implementation language.
8. Click the **Next** button to continue to the Attributes page.
9. Click the **Next** button to continue to the Methods page.
10. Click the **Next** button to continue to the Key and Copy Helper page.
11. Verify that the `ClaimKey` key and the `ClaimCopy` copy helper are selected.
12. Click the **Next** button to continue to the Handle Selection page.
13. Click the **Next** button to continue to the Attributes to Override page.
14. Click the **Next** button to accept the defaults and to continue to the Data Object Interface page.
15. Click the **All >>** button to move the attributes in the Business Object Attributes list to the State Data list.
16. Click the **Next** button to continue to the Data Objects Methods page.
17. Click the **Next** button to accept the defaults and to continue to the Summary of Framework Methods page.
18. Verify that the syncToDataObject() and syncFromDataObject() methods are added.
19. Click the **Finish** button.

The ClaimBO business object implementation and the ClaimDO data object interface are displayed in the User-Defined Business Objects folder.

The majority of the business object implementation is now defined, but the code for the approve() and deny() methods still needs to be provided. To provide this code:

1. From the Tasks and Objects pane, select the **ClaimBO** business object interface. The Method List pane contains the methods and attributes for the object.
2. Double-click **User-Defined Methods**, and select the **approve()** method. The skeleton implementation is displayed in the Source pane.
3. Type the following implementation for the approve() method:
   ```
   state(1);
   ```
   When a claim is approved, its state changes from `0` to `1`.
4. Again from the Methods pane, select the **deny()** method. The skeleton implementation is displayed in the Source pane.
5. Type the following implementation for the deny method:
   ```
   state(-1);
   ```

When a claim is denied, its state changes to `-1`.

Defining a Data Object Implementation

Each data object interface can have multiple implementations just as a business object interface can have multiple implementations.

For this exercise, a data object implementation is defined to access data from the database using embedded SQL. To create the data object implementation, the Data Object Implementation wizard is used. This wizard contains pages where you can:

- Define the environment.
- Define any parent classes.
- Specify the key and copy helper objects.

To define the data object implementation:

1. From the **User-Defined Business Objects** folder, select the **ClaimDO** data object interface.
2. Open the pop-up menu of ClaimDO, and select **Add Implementation**. This opens the Data Object Implementation wizard.
3. Set the environment.
   a. Set the **BOIM with any key** radio button in the Environment section to indicate that the data object is part of a component installed in a business object application adaptor with instances being located by key objects.
   b. Set the **Embedded SQL** radio button in the Form of Persistent Behavior and Implementation section.
   c. Set the **Local copy** radio button in the Data Access Pattern section.
   d. Set the **Home name and key** radio button in the Handle for Storing Pointers section.

   Note the deployment platform options. By default, the deployment platforms match those selected in the **Platform - Constrain** menu. By selecting a single deployment platform for the data object, you can take advantage of a broader range of development options (because you are not limited to cross-platform options). In this exercise, you can ignore these options.
4. Click the **Next** button to continue to the Implementation Inheritance page.
5. Click the **Next** button to accept the defaults and to continue to the Attributes page.
6. Click the **Next** button to continue to the Methods page.
7. Click the **Next** button to continue to the Key and Copy Helper page.
8. Verify that the following fields contain their associated values:
   - Key contains `ClaimKey`.
   - Copy Helper contains `ClaimCopy`.
9. Click the **Finish** button. The remaining pages in this wizard are not needed for this exercise.

The ClaimDOImpl data object implementation is displayed in the User-Defined Business Objects folder. Because BOIM with any key is specified on the Name and Behavior page, a persistent object needs to be added to the implementation. The framework methods cannot be implemented unless a persistent object is created. If you view the method body for any of the framework method, you will see a comment stating that a persistent object needs to be defined.

Defining a Persistent Object and Schema

The persistent object and schema are the final layer of the component. The database columns and SQL data types are mapped to C++ attributes and data types. To create the persistent object and schema, the Add Persistent Object and Schema wizard is used.

To define the persistent object and schema:

1. Expand **User-Defined Data Objects - ClaimFileDO - ClaimDO**, and select the **ClaimDOImpl** data object implementation.
2. Open the pop-up menu of ClaimDOImpl, and select **Add Persistent Object and Schema**. This opens the Add Persistent Object and Schema wizard to the Name and Attributes page.
3. Type `ClaimDBGroup` in the Group Name field.
4. Type `ClaimDB` in the Database field.
5. Click the **Finish** button.

The ClaimDBGroup schema group, ClaimDB schema, and ClaimPO persistent object appear in the DBA-Defined Schemas folder.

Defining a Managed Object

The managed object is the top layer of the component. The managed object gets the required server objects, provides the activation controls, and ensures that data is accessed and set reliably and securely. The managed object is defined from the business object implementation. If multiple business object implementations are defined, each implementation could have its own managed object.

To create a managed object, the Managed Object wizard is used. This wizard contains pages where you can:

- Specify the file name and name of the object.
- Define any parent classes.

To add a managed object:

1. Expand **User-Defined Business Objects - ClaimFile - Claim**, and select the **ClaimBO** business object implementation.
2. Open the pop-up menu of the ClaimBO business object implementation, and select **Add Managed Object**. This opens the Managed Object wizard to the Name and Application Adaptor page.

   Note the deployment platform options. By default, the deployment platforms match those selected in the **Platform - Constrain** menu. By selecting a single deployment platform for the managed object, you can take advantage of a broader range of development options (because you are not limited to cross-platform options). In this exercise, you can ignore these options.
3. Click the **Next** button to accept the defaults and continue to the Implementation Inheritance page.
4. Click the **Finish** button.

The ClaimMO managed object is displayed in the User-Defined Business Objects folder.

Generating the Code

Generating the code creates:

- IDL files for the interfaces.

- IDL and C++ or Java files for the implementations, the key object, and the copy helper object.
- An .sql file for the schema that can be changed to define the table in a database.
- An .sqx and an .hpp file for the persistent object.

Until you generate the code, all the information for the objects is maintained in an Object Builder data model. Save this data model by selecting File - Save. The data model is saved to the current project's model directory (for example, `f:\CBroker\MyProject\model` for Windows NT or `$HOME/MyProject/Model` for AIX). When you generate the code, the resulting files are placed in the current project's working directory (for example, `f:\CBroker\MyProject\working`\NT\ for Windows NT or `$HOME/MyProject/Working/AIX` for AIX).

To generate the code:

1. Expand **User-Defined Business Objects**, and select the **ClaimFile** business object file.
2. Open the pop-up menu of ClaimFile, and select **Generate - All** to generate code and to generate the .sql, .sqx, and .hpp files.

You can view the source code for any defined objects, but you cannot directly edit this source. If you want to change the source, use the wizards. The only code you should edit directly is the implementation code for methods, which you can access by clicking on a method in the Methods list.

For example, to view the source code for the ClaimBO business object implementation:

1. Open the pop-up menu of the ClaimBO business object implementation, and select **View Source**. The source files (.idl, .ih, and .cpp for C++, or .idl and .java for Java) for the object are displayed in the Source pane.
2. Click the drop-down arrow of the Source pane title bar to display the list of currently loaded files. You can switch among these files.

Before the code can be compiled, the ClaimDB database needs to be defined and a .mak file needs to be created. The .mak file defines build options, a set of source files that will be built into the target DLL files, and dependencies.

Summary
You have defined the objects that make up the Claim component, implemented two methods for changing the status of the Claim, and generated the code for the Claim objects.

You are ready to continue to the next scenario, in which you build the generated code into DLLs or shared library files:

- "Build DLLs or Shared Library Files - Scenario"

Build DLLs or Shared Library Files - Scenario

Objectives
To configure the database that will hold your data.
To define the makefiles to build a C++ or Java component with access to DB2 data.
To build the components into DLLs (also known as shared library files).

Before You Begin
This scenario is a continuation of the Create a Component (page 39) scenario. You should complete the previous scenario before attempting this one.

You need the following installed on your system:

- CBToolkit, including Samples
- DB2 Universal Database
- WIN VisualAge for C++ and (for Java applications) VisualAge for Java
- (optional) InstallShield

If you are developing a Java component, make sure your CLASSPATH is set correctly, as described in the following topic:
- “Requirements for Java Development” on page 8

Description
This exercise provides the necessary steps to build your generated code into DLLs or shared library files. For this exercise, you will:

1. Configure the ClaimDB database
2. Define the client DLL
3. Define the server DLL
4. Build the DLLs

Once you have created the DLLs and (for Java) .jar files, you can continue to the next scenario:
- “Package an Application - Scenario” on page 50

Configuring the ClaimDB Database
You need to define (in DB2) the ClaimDB database and Claim table that your component will access. You should have a database administrator perform this procedure.

To configure the database and table, you need to enter the following commands from a DB2 command prompt.

```
create database ClaimDB
connect to ClaimDB
create table Claim (claimNo integer not null, state integer, primary key(claimNo))
```

The syntax for the last command is provided by Object Builder in the generated .sql file for the ClaimDBGroup schema.

Creating Client and Server DLL Files
The defined objects need to be built into two separate DLL files:

- A client DLL, which provides the client application with an interface to the component on the server, along with a key and copy helper to simplify location and creation of the component on the server.
- A server DLL, which contains the actual component implementation on the server.

The client DLL file needs to be defined before the server DLL file. When the server DLL file is defined, it needs to link to the client DLL file. After defining the objects that comprise each DLL file, the DLL files can be built.

AIX only:
The DLL files are called “shared library” files and are in the format lib*.so. For any reference to a DLL file, substitute shared library file.

Defining the Client DLL File
To define the client DLL file:

1. From the Tasks and Objects pane, select the **Build Configuration** folder.
2. Open the pop-up menu of the Build Configuration folder, and select **Add Client DLL**. This opens the Client DLL wizard to the Name and Option page.
3. Type ClaimC in the Name field.
4. Click the **Next** button to continue to the Client Source Files page.
5. Click the **All >>** button to move the client Claim files to the Items chosen list.
6. Click the **Next** button to continue to the Libraries to Link With page.
7. Click the **Finish** button to accept the defaults.

The ClaimC client DLL file is displayed in the Build Configuration folder.

Defining the Server DLL File
To define the server DLL file:

1. From the Tasks and Objects pane, select the **Build Configuration** folder.
2. Open the pop-up menu of the Build Configuration folder, and select **Add Server DLL**. This opens the Server DLL wizard.
3. Provide the following information on this page.
   a. Type ClaimS in the **Name** field.
   b. WIN Type IVB_TRACE_DEBUG=1 in the **Make Options** field to enable remote tracing and debugging of server code.
   c. AIX Type -DCBS_TRACE_DEBUG -g in the **CPP Compiler Options** field, and -livbtr10 in the **Link Options** field.
4. Click the **Next** button to continue to the Server Source Files page.
5. Click the **All >>** button to move the server Claim objects to the Items chosen list.
6. Click the **Next** button to continue to the Libraries to Link With page.
7. Select ClaimC in the Items available list.
8. Click the **>>** button to move the DLL file to the Items chosen list. The library file for ClaimC is included as a link for the ClaimS DLL file.
9. Click the **Finish** button.

The ClaimS server DLL file is displayed in the Build Configuration folder.

Building the DLL Files
To generate the makefiles and build the DLL files:

1. From the Tasks and Objects pane, select the **Build Configuration** folder.
2. Open the pop-up menu of the Build Configuration folder, and select **Generate - All - C++ Default Targets**. This generates makefiles for all the DLL files defined in the folder and generates an all.mak file that calls the DLL makefiles. Note that the choice of C++ targets determines what the default build target is, but does not prevent you from using all.mak to build other targets.

For a C++ component, build as follows:

1. From the Build Configuration folder's pop-up menu, select **Build - Out-of-Date Targets - Default**.

   WIN The ClaimC.dll and ClaimS.dll files are stored in your Object Builder working\NT directory.

   AIX The libClaimC.so and libClaimS.so files are stored in your Object Builder working/AIX directory.

   If your application contained Java components as well as C++ components, you would also select **Build - Out-of-Date Targets - Java**, to create ClaimC.jar in your \working*platform* directory. The .jar file allows Java components on the server to interact with your C++ components in the DLLs. For this exercise, your application contains just a single C++ component, so you do not need to build the .jar file.

For a Java component, build as follows:

1. From the Build Configuration folder's pop-up menu, select **Build - Out-of-Date Targets - Java**.

   This creates ClaimC.jar and ClaimS.jar in your working*platform* directory.

If you had a Java client application, regardless of the language your component is implemented in, you would also select **Build - Out-of-Date Targets - Java Client Bindings** to generate Java client bindings (working*platform*\JCB\JCBClaimC.jar). For this exercise, the client application is in C++, and you do not need to build Java client bindings.

Summary
You have configured the database that will hold your data, created the makefiles that define the client and server DLLs or shared library files for your component, and built the DLLs or .jar files.

You are ready to continue to the next scenario, in which you package your client application and server component into an application family:

- "Package an Application - Scenario"

Package an Application - Scenario

Objectives
To build the client application from provided sample files.
To define application families for the client and server applications.
To define a container for the component on the server.
To configure the component with a container and home on the server.
To generate the install information for the application family.

Before You Begin
This scenario is a continuation of the scenario series:

1. "Create a Component - Scenario" on page 39
2. "Build DLLs or Shared Library Files - Scenario" on page 47

You should complete the previous scenarios before attempting this one.

You need the following installed on your system:

- CBToolkit, including Samples

- DB2 Universal Database
- WIN VisualAge for C++ and (for Java applications) VisualAge for Java
- (optional) InstallShield

If you are developing a Java component, make sure your CLASSPATH is set correctly, as described in the following topic:
- “Requirements for Java Development” on page 8

Description
This exercise provides the necessary steps for you to package your client application and server components and prepare them for installation on the client and server. For this exercise, you will:

1. Build the client application from provided sample files.
2. Define a client application family.
3. Add the client application to the family.
4. Define a server application family.
5. Add the server application in the family.
6. Define a container for the component on the server.
7. Configure the Claim component created in previous scenarios with the container, and add it to the server application.
8. Generate the installation information for both application families.

Once you have defined the application and generated the installation information, you can continue to the next scenario:
- WIN “Install and Run an Application Using InstallShield - Scenario” on page 57
- “Install and Run an Application - Scenario” on page 61

Building the Client Application
Before packaging an application, you would normally write a fully functional client application. However, writing this application is beyond the scope of this exercise. For information on writing a client application, see “MOFW Client Programming Model” in the *IBM Component Broker Programming Guide*.

For this exercise, the source for the client application is called “ClaimApp” and is shipped as part of the samples.

WIN To build ClaimApp:

1. Copy the ClaimApp.cpp and the ClaimApp.mak files from
   `x:\CBroker\samples\InstallVerification\ProgrammingModel\Applications\C++\Claim` to your Object Builder working directory (`x:\CBroker\MyProject\working\NT`).
2. From a command prompt, change directory to your Object Builder working directory.
3. Enter:
   ```
   nmake -f ClaimApp.mak
   ```

AIX To build ClaimApp:

1. Copy the ClaimApp.cpp and ClaimApp.mak files from
   `$HOME/samples/InstallVerification/ProgrammingModel/Applications/C++/Claim` to your Object Builder working directory (`$HOME/MyProject/Working/AIX`).

2. From a command prompt, change directory to your Object Builder working directory.
3. Enter:

```
make -f ClaimApp.mak
```

Creating the Server Application Family
The first step in packaging the DLL files for your application is defining its *application family*. An application family consists of a number of related applications that work together and need to be running at the same code level.

Each application family has a single installation process that handles all the applications defined in it. During installation, you can select the applications to install or remove from a system.

In this exercise, you will create two application families. The first application family will hold ClaimAppS. The ClaimAppS application defines the component on the server that the client application will access.

The installation checks the version of applications in the same family and ensures that, at the end of the installation, all applications in the family are at the same version.

To create the application family:
1. From the Tasks and Object pane, select the **Application Configuration** folder.
2. Open the pop-up menu of the Application Configuration folder, and select **Add Application Family**. This opens the Add Application Family wizard to the Name page.
3. Type `ClaimAppFam` in the **Name** field. The default version number (1.0.0) is acceptable and the description is optional.
4. Click the **Next** button to accept the other defaults and to continue to the Installation Information page.
5. Click the **Finish** button.

The ClaimAppFam object is displayed in the Application Configuration folder.

On the Installation Information page, you could specify any additional disk space requirements. If specified, the installation process checks to ensure that the necessary space is available before the installation can proceed. On this page, you could also specify the location of a README file you wanted to include, and the installation process would include the option to open the README file as a final step.

Defining a Server Application
To define the server application:
1. From the Tasks and Objects pane, expand the **Application Configuration** folder and select the **ClaimAppFam** application family.
2. Open the pop-up menu of the ClaimAppFam application family, and select **Add Application**. This opens the Add Application wizard to the Name and Environment page.
3. Type `ClaimAppS` in the **Name** field.
4. Click the **Next** button to continue to the **Additional Executables** page.
5. Select the platform you are configuring the application for.
6. Add the file Claim.sql:

a. Click the **Find** button to open the Executables to Include dialog.
b. Locate your Object Builder working directory.
c. Select Claim.sql
d. WIN Click the **Open** button.
e. AIX Click the **OK** button.
7. Add the file ClaimPO.bnd, in the same manner.
8. Click the Finish button.

The ClaimAppS application is displayed in the ClaimAppFam folder.

Creating the Client Application Family
Even though your client application is written and built outside of Object Builder, you can still use Object Builder to package the application and to generate the installation process for it.

You are now ready to define the client application family, that will hold the ClaimAppC application. The ClaimAppC application will hold the client executable, as well as its interface to the server component.

To create the application family:
1. From the Tasks and Object pane, select the **Application Configuration** folder.
2. Open the pop-up menu of the Application Configuration folder, and select **Add Application Family**. This opens the Add Application Family wizard to the Name page.
3. Type `ClaimAppClientFam` in the **Name** field. The default version number (1.0.0) is acceptable and the description is optional.
4. Click the **Next** button to accept the other defaults and to continue to the Installation Information page.
5. Click the **Finish** button.

The ClaimAppClientFam object is displayed in the Application Configuration folder.

On the Installation Information page, you could specify any additional disk space requirements. If specified, the installation process checks to ensure that the necessary space is available before the installation can proceed. On this page, you could also specify the location of a README file you wanted to include, and the installation process would include the option to open the README file as a final step.

Defining a Client Application
To define the client application:
1. From the Tasks and Objects pane, select the **Application Configuration** folder and select the **ClaimAppClientFam** application family.
2. Open the pop-up menu of the ClaimAppClientFam application family, and select **Add Application**. This opens the Add Application wizard to the Name and Environment page.
3. Define the application and its environment.
   a. Type `ClaimAppC` in the **Name** field.
   b. Click the **Next** button to continue to the Additional Executables page.
   c. Select the platform you are configuring the application for.
   d. Add the file ClaimApp.exe:

1) Click the **Find** button to open the Executables to Include dialog.
2) Locate your Object Builder working directory.
3) Select ClaimApp.exe
4) WIN Click the **Open** button.
5) AIX Click the **OK** button.

e. WIN Add the file ClaimC.dll, in the same manner.

Note:
If you do not see the ClaimC.dll file in the Object Builder file dialog, ensure that the file is not hidden. Using Windows NT Explorer, select View - Options. In the Options dialog, click the View tab and set the Show All Files check box. Close the Object Builder file dialog and click the Browse button again. The DLL file should now be displayed.

f. AIX Add the file libClaimC.so in the same manner.

4. Click the Finish button.

The ClaimAppC application appears in the Application Configuration folder.

Creating a Container Instance

Before you can add the Claim component to the server application, you need to define a new container instance that will provide object services for that component. Although there are default containers provided with Object Builder, these default containers are for components whose data is transient. Because Claim has persistent data, you must define a new container.

To define a container instance:

1. From the Tasks and Objects pane, select the **Container Definition** folder.
2. Open the pop-up menu for the Container Definition folder, and select **Add Container Instance**. This opens the Container wizard to the Name of Container and Number of Components page.

   Note the deployment platform options. By default, the deployment platforms match those selected in the **Platform - Constrain** menu. By selecting a single deployment platform for the container, you can take advantage of a broader range of development options (because you are not limited to cross-platform options). In this exercise, you can ignore these options.
3. Type `ContainerOfClaims` in the **Name** field.
4. Click the **Next** button to accept the other defaults and continue to the Workload Management page.
5. Click the **Next** button to accept the defaults and continue to the Service page.
6. Define the policies:
   a. Set the **Use RDB Transaction Services** check box.
   b. Click the **Next** button to continue.
7. On the Service Details page:
   a. Set the **Throw an exception and abandon call** radio button in the Behavior for Methods Called Outside a Transaction field.
   b. Click the **Next** button to accept the other defaults and to continue to the Data Access Patterns page.
8. Define the data access pattern.
   a. Set the **Caching** radio button in the Business Object field. This matches the data access pattern you set when you created the business object.

b. Set the **Local copy** radio button in the Data Object field. This matches the data access pattern you set when you created the data object implementation.
c. Click the **Finish** button.

The ContainerOfClaims container is added to the Container Definition folder.

Configuring the Managed Object
To add the Claim component to the application, you configure the component's managed object with the application. It also serves to confirm the objects that make up the component (for example, if you defined more than one data object implementation for the component, this is the point at which you choose which data object implementation to package).

To configure the managed object:

1. From the Tasks and Object pane, expand **Application Configuration - ClaimAppFam**, and select the **ClaimAppS** server application.
2. Open the pop-up menu of ClaimAppS, and select **Add Managed Object**. This opens the Configure Managed Object wizard.
3. Ensure that `ClaimFileMO ClaimMo` is selected in the Managed Object field.

   The Selection page should show:
   - For the managed object, `ClaimMO` in the ClaimS DLL.
   - For the key, `ClaimKey` in the ClaimC DLL.
   - For the copy helper, `ClaimCopy` in the ClaimC DLL.
4. Click the **Next** button to continue to the Data Object Implementations page.
5. Click the **Add Another** button. The ClaimDOImpl implementation and the ClaimS DLL file are selected.
6. Click the **Next** button to continue to the Container page.
7. Click the **Next** button to accept the defaults and continue to the Home page. For the Claim component, the default home selected and the default system management settings are sufficient.
8. Click the **Finish** button.

The ClaimMO managed object is displayed in the Application Configuration folder.

The Container and Home pages are used to define how the managed object will be installed in the Component Broker run-time environment. The container selected for the managed object determines which object services are available. The home you define determines the interface to the home that will be used to locate and create instances of the component. The combination of a home and a particular component's key uniquely identifies the component.

Generating the DDL Files
To generate the DDL files:

1. From the Tasks and Object pane, expand **Application Configuration** and select the **ClaimAppFam** application family.
2. Open the pop-up menu for the ClaimAppFam application family, and select **Generate**.

The ClaimAppFam.ddl and the SpecificClaimAppFam.ddl are generated and placed in a subdirectory of your working directory. For example:

WIN x:\CBroker\MyProject\working\NT\ClaimAppFam

AIX $HOME/MyProject/Working/AIX/ClaimAppFam

When generation is completed, the Method Implementation pane contains the ClaimAppFam.ddl file. You can now close Object Builder.

WIN Create the Install Image (optional)
If you are using InstallShield on Windows NT, you can create an install image, that you can burn onto a CD-ROM and distribute.

To set the location for InstallShield:
1. From Object Builder menu bar, select **File - Preferences**. The Preferences window is opened.
2. From the tree view, select **Tasks and Objects**.
3. Specify the version of InstallShield you are using, and provide the path for the directory in which it is installed.
4. Click **OK**.

To generate the install image:
1. From the pop-up menu of ClaimAppFam, select **Generate**. The install scripts are generated and placed in the working directory under a subdirectory of the same name as the application family (for example, `x:\CBroker\MyProject\Working\NT\ClaimAppFam`).

   When generation is completed, the Method Implementation pane contains the ClaimAppFam.ddl file. This DDL file is used to set up your application with Component Broker system management.
2. From the same pop-up menu, select **Build**. This calls the build.bat file that creates the Disk1 subdirectory for the Application Family directory (for example, `x:\CBroker\MyProject\Working\NT\ClaimAppFam\Disk1`).

   The progress of the build is displayed in a Command Window. When the build is finished, the following message is displayed at the bottom of the window:
   ```
   Ended (exit code 0)
   ```
3. Review the build record, and close the Command Window.

The Disk1 directory now contains the install image (including setup executables) that can be burned onto a compact disc.

Summary
You have built the client application, defined an application family and server application, defined a container for the component on the server, and configured the component with the server application. You have generated the installation information for the application family.

You are ready to continue to one of the next scenarios, in which you install and configure your server application, and run the client application:
- WIN “Install and Run an Application Using InstallShield - Scenario” on page 57
- “Install and Run an Application - Scenario” on page 61

Install and Run an Application Using InstallShield - Scenario

Objectives
To bind a server application to a database.
To install the server application on a server machine, using InstallShield
To configure the server application with System Management.
To run a client application that accesses the server data.

Before You Begin
This scenario is a continuation of the scenario series:

1. “Create a Component - Scenario” on page 39
2. “Build DLLs or Shared Library Files - Scenario” on page 47
3. “Package an Application - Scenario” on page 50

You should complete the previous scenarios before attempting this one.

This scenario runs on a server machine. While the previous scenarios can be run on a dedicated development machine, this scenario requires that you have CBConnector System Manager installed. For the sake of exercises that follow this one, it will be simpler if the client and server are on the same machine.

You need the following installed:

- CBConnector (System Management)
- DB2 Universal Database
- WIN VisualAge for C++ and (for Java applications) VisualAge for Java

If you are deploying a Java component, make sure your CLASSPATH is set correctly, as described in the following topic:

- “Requirements for Java Development” on page 8

Description
This exercise describes how to load, configure, and run your application, using InstallShield. If you are not using InstallShield, see the “Install and Run an Application - Scenario” on page 61 (without using InstallShield) scenario.

For this exercise, you will:

1. Bind the server application to a database.
2. Load the server application into System Management.
3. Configure the server application with System Management.
4. Run the client application.

Once you have installed the application and run the client successfully, you can continue to one of the next scenarios:

- “Trace and Debug an Application - Scenario” on page 65
- WIN “Uninstall an Application Using InstallShield - Scenario” on page 70

Binding the Application
WIN To bind an application to the ClaimDB database:

1. Change to the working directory that contains ClaimPO.bnd (`x:\CBroker\MyProject\working\NT`).

2. Enter:

```
obdatapr ClaimPO.bnd ClaimDB bind
```

This step is necessary because the Claim component uses the Embedded SQL pattern for persistence (as set in the data object implementation).

If your component had used the Cache Service pattern instead, you would change to the directory `x:\CBroker\etc\` and bind the files `db2cntcs.bnd` and `db2cntrr.bnd`, with the commands:

```
obdatapr db2cntcs.bnd ClaimDB bind
obdatapr db2cntrr.bnd ClaimDB bind
```

AIX To bind an application to the ClaimDB database:

1. Change to the working directory that contains ClaimPO.bnd (`$HOME/MyProject/Working/AIX`).
2. Enter:

```
db2 connect to claimdb
db2 bind ClaimPO.bnd
```

This step is necessary because the Claim component uses the Embedded SQL pattern for persistence (as set in the data object implementation).

If your component had used the Cache Service pattern instead, you would change to the directory `/usr/lpp/CBConnector/etc` and bind the files `db2cntcs.bnd` and `db2cntrr.bnd`, with the commands:

```
db2 bind db2cntcs.bnd
db2 bind db2cntrr.bnd
db2 connect reset
```

Running the Installation Setup

This procedure assumes that the initial activation is completed. To run the installation setup:

1. Ensure that the name server is running.
2. From the command line, enter:

```
x:\CBroker\MyProject\Working\NT\ClaimAppFam\Disk1\setup.exe
```

The InstallShield installation begins and displays a copyright screen.

3. Click the **Next** button.
4. Accept the default destination directory (`x:\CBroker`). The applications must be installed to the `ntapps\ClaimAppFam\bin` subdirectory; if you selected another destination, then the installation process would create this subdirectory off the specified directory.
5. Click the **Next** button. The Application Installation page is opened.
6. Select **ClaimAppS**. If you provided a description for the application (when you defined it in Object Builder), the description is displayed.
7. Click the **Next** button. The Start Installing page is opened.
8. Review the installation settings. You should be installing the ClaimAppS application.
9. Click the **Next** button. The install process begins and can take several minutes (depending on the speed of the computer).

After using the compact disk for a site installation, you still need to configure the client application and the component using the System Manager User Interface.

Configuring the Application with System Management

To configure the application:

1. Configure the application with a management zone.
   a. For ClaimAppS:
      1) Expand **Available Applications**, and select the ClaimAppS.
      2) Open the pop-up menu of ClaimAppS, and select **Drag**.
      3) Expand M**anagement Zones - Sample Cell and Workgroup Zone - Configurations**, and select **Sample Configuration**.
      4) Open the pop-up menu of Sample Configuration, and select **Add Application**.
   b. For SpecificClaimAppS:
      1) Expand **Available Applications**, and select **SpecificClaimAppS**.
      2) Open the pop-up menu of SpecificClaimAppS, and select **Drag**.
      3) Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations**, and select **Sample Configuration**.
      4) Open the pop-up menu of Sample Configuration, and select **Add Application**.

   The ClaimAppS and SpecificClaimAppS applications were added to the Applications folder within the Configurations folder.
2. Configure the server.
   - Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations**, and select **Sample Configuration**.
   - Open the pop-up menu of Sample Configuration, and select **New - Server Group**. This opens a dialog box.
   - Type `ClaimServerGroup` as the name for the server group.
   - Click the **OK** button. The ClaimServerGroup is displayed under Server Groups.
   - Open the pop-up menu of ClaimServerGroup, and select **New - Server (member of group)**. A dialog box is displayed.
   - Type `ClaimServer` as the name for the server.
   - Click the **OK** button. The ClaimServer is displayed under ClaimServerGroup.
3. Associate the configured application with the server.
   a. For ClaimAppS:
      1) **Expand Management Zones - Sample Cell and Workgroup Zone - Configurations - Sample Configuration - Applications**, and select C**laimAppS**.
      2) Open the pop-up menu of ClaimAppS, and select **Drag**.
      3) Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Sample Configuration - Server Groups**, and select C**laimServerGroup**.
      4) Open the pop-up menu of ClaimServerGroup, and select **Configure Application**.
   b. For SpecificClaimAppS:
      1) Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Sample Configuration - Applications**, and select **SpecificClaimAppS**.
      2) Open the pop-up menu of SpecificClaimAppS, and select **Drag**.

3) Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Sample Configuration - Server Groups**, and select **ClaimServerGroup**.
4) Open the pop-up menu of ClaimServerGroup, and select **Configure Application**.

A Configured Applications folder is displayed under ClaimServer. You can expand the folder to display the entries for ClaimAppS and SpecificClaimAppS.

4. Associate the iDB2IMServices application with the server.
   a. Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Sample Configuration - Applications, and select iDB2IMServices**.
   b. Open the pop-up menu of iDB2IMServices, and select **Drag**.
   c. Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Server Group**s, and select **ClaimServerGroup**.
   d. Open the pop-up menu of ClaimServerGroup, and select **Configure Application**.
5. Configure the server with the host.
   a. Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Sample Configurations - Server Groups - ClaimServerGroup - Servers (member of group)**, and select **ClaimServer**.
   b. Open the pop-up menu of ClaimServer, and select **Drag**.
   c. Under Hosts, select *myhost* for your current system.
   d. Open the pop-up menu of *myhost*, and select **Configure Server (member of group)**.

   Under the *myhost* folder, there is now a folder called Configured Servers (members of group) that contains an entry for the ClaimServer server.
6. Enable security services (optional).
   a. Expand **Management Zones - Sample Cell and Work Group Zone - Configurations - Sample Configurations - Server Groups**, and select **ClaimServerGroup**.
   b. Open the pop-up menu of ClaimServerGroup, and select **Edit**. This opens the Object Editor.
   c. Click the **Security Service** tab.
   d. Change the security enabled field from `no` to `yes`.
   e. Click the **OK** button.
7. Activate the configuration.
   a. Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations**, and select **Sample Configuration**.
   b. Open the pop-up menu of Sample Configuration, and select **Activate**. A window should appear, confirming that the configuration has been activated.

Running the Application

To run the client application:

1. From a command prompt, change directory to where the ClaimApp executable is stored:

   WIN `x:\CBroker\MyProject\working\NT`

AIX $HOME/MyProject/Working/AIX

2. Enter:

```
ClaimApp
```

When the application finishes running, the new Claim component is created (with values for its ClaimNo and state stored in the ClaimDB database) and the get and set accessor methods of the component.

Summary
You have bound the server application to the database, installed and configured the server application, and run the client application.

You are ready to continue to one of the next scenarios, in which you either debug your client and server applications, or remove the application from the server:

- WIN Debug an Application
- WIN Uninstall an Application Using InstallShield (page 70)

Install and Run an Application - Scenario

Objectives
To bind a server application to a database.
To install the server application on a server machine, without using InstallShield
To configure the server application with System Management.
To run a client application that accesses the server data.

Before You Begin
This scenario is a continuation of the scenario series:

1. "Create a Component - Scenario" on page 39
2. "Build DLLs or Shared Library Files - Scenario" on page 47
3. "Package an Application - Scenario" on page 50

You should complete the previous scenarios before attempting this one.

This scenario runs on a server machine. While the previous scenarios can be run on a dedicated development machine, this scenario requires that you have CBConnector System Manager installed. For the sake of exercises that follow this one, it will be simpler if the Component Broker client application and the Component Broker Application Server are on the same machine.

You need the following installed:

- CBConnector (System Management)
- DB2 Universal Database
- WIN VisualAge for C++ and (for Java applications) VisualAge for Java

If you are deploying a Java component, make sure your CLASSPATH is set correctly, as described in the following topic:

- "Requirements for Java Development" on page 8

Description
This exercise describes how to load, configure, and run your application, without

using InstallShield. If you are using InstallShield, see the “Install and Run an Application Using InstallShield - Scenario” on page 57 scenario.

For this exercise, you will:

1. Bind the server application to a database.
2. Load the server application into System Management.
3. Configure the server application with System Management.
4. Run the client application.

Once you have installed the application and run the client successfully, you can continue to one of the next scenarios:

- “Trace and Debug an Application - Scenario” on page 65
- “Uninstall an Application - Scenario” on page 71 (without using InstallShield)

Binding the Application

WIN To bind an application to the ClaimDB database:

1. Change to the working directory that contains ClaimPO.bnd (`x:\CBroker\MyProject\working\NT`).
2. Enter:

   ```
   obdatapr ClaimPO.bnd ClaimDB bind
   ```

   This step is necessary because the Claim component uses the Embedded SQL pattern for persistence (as set in the data object implementation).

   If your component had used the Cache Service pattern instead, you would change to the directory `x:\CBroker\etc\` and bind the files `db2cntcs.bnd` and `db2cntrr.bnd`, with the commands:

   ```
   obdatapr db2cntcs.bnd ClaimDB bind
   obdatapr db2cntrr.bnd ClaimDB bind
   ```

AIX To bind an application to the ClaimDB database:

1. Change to the working directory that contains ClaimPO.bnd (`$HOME/MyProject/Working/AIX`).
2. Enter:

   ```
   db2 connect to claimdb
   db2 bind ClaimPO.bnd
   ```

   This step is necessary because the Claim component uses the Embedded SQL pattern for persistence (as set in the data object implementation).

   If your component had used the Cache Service pattern instead, you would change to the directory `/usr/lpp/CBConnector/etc` and bind the files`db2cntcs.bnd` and `db2cntrr.bnd`, with the commands:

   ```
   db2 bind db2cntcs.bnd
   db2 bind db2cntrr.bnd
   ```
3. Once you have bound the files, reset the connection:

   ```
   db2 connect reset
   ```

Loading the Application onto System Management

Once you have bound the application to the database, you can load it onto System Management.

To load the application onto System Management:

1. Start the System Manager User Interface.
2. Become an Expert user (**View - User Level - Expert**).
3. Expand **Host Images**, and select **myhost.austin.ibm.com**.
4. Open the pop-up menu of myhost.austin.ibm.com, and select **Load Application**.
5. Browse for and select **ClaimAppFam.ddl**, in the following directory:

   WIN `x:\CBroker\MyProject\working\NT\ClaimAppFam`

   AIX `$HOME/MyProject/Working/AIX/ClaimAppFam`

6. Again open the pop-up menu for myhost.austin.ibm.com, and select **Load Application**.
7. Browse for and select **SpecificClaimAppFam.ddl**

Configuring the Application with System Management

To configure the application:

1. Configure the application with a management zone.
   a. For ClaimAppS:
      1) Expand **Available Applications**, and select **ClaimAppS**.
      2) Open the pop-up menu of ClaimAppS, and select **Drag**.
      3) Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations**, and select **Sample Configuration**.
      4) Open the pop-up menu of Sample Configuration, and select **Add Application**.
   b. For SpecificClaimAppS:
      1) Expand **Available Applications**, and select **SpecificClaimAppS**.
      2) Open the pop-up menu of SpecificClaimAppS, and select **Drag**.
      3) Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations**, and select **Sample Configuration**.
      4) Open the pop-up menu of Sample Configuration, and select **Add Application**.

   The ClaimAppS and SpecificClaimAppS applications were added to the Applications folder within the Configurations folder.
2. Configure the server.
   - Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations**, and select **Sample Configuration**.
   - Open the pop-up menu of Sample Configuration, and select **New - Server Group**. This opens a dialog box.
   - Type `ClaimServerGroup` as the name for the server group.
   - Click the **OK** button. The ClaimServerGroup is displayed under Server Groups.
   - Open the pop-up menu of ClaimServerGroup, and select **New - Server (member of group)**. A dialog box is displayed.
   - Type `ClaimServer` as the name for the server.
   - Click the **OK** button. The ClaimServer is displayed under ClaimServerGroup.
3. Associate the configured application with the server.
   a. For ClaimAppS:

1) Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Sample Configuration - Applications**, and select ClaimAppS.
2) Open the pop-up menu of ClaimAppS, and select **Drag**.
3) Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Sample Configuration - Server Groups**, and select **ClaimServerGroup**.
4) Open the pop-up menu of ClaimServerGroup, and select **Configure Application**.

b. For SpecificClaimAppS:

1) Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Sample Configuration - Applications**, and select **SpecificClaimAppS**.
2) Open the pop-up menu of SpecificClaimAppS, and select **Drag**.
3) Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Sample Configuration - Server Groups**, and select **ClaimServerGroup**.
4) Open the pop-up menu of ClaimServerGroup, and select **Configure Application**.

A Configured Applications folder is displayed under ClaimServer. You can expand the folder to display the entries for ClaimAppS and SpecificClaimAppS.

4. Associate the iDB2IMServices application with the server.
   a. Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Sample Configuration - Applications**, and select **iDB2IMServices**.
   b. Open the pop-up menu of iDB2IMServices, and select **Drag**.
   c. Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Server Groups**, and select **ClaimServerGroup**.
   d. Open the pop-up menu of ClaimServerGroup, and select **Configure Application**.
5. Configure the server with the host.
   a. Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations - Sample Configurations - Server Groups - ClaimServerGroup - Servers (member of group)**, and select **ClaimServer**.
   b. Open the pop-up menu of **ClaimServer**, and select **Drag**.
   c. Under **Hosts**, select **myhost** for your current system.
   d. Open the pop-up menu of myhost, and select **Configure Server (member of group)**.

   Under the myhost folder, there is now a folder called Configured Servers (members of group) that contains an entry for the ClaimServer server.
6. Enable security services (optional).
   a. Expand **Management Zones - Sample Cell and Work Group Zone - Configurations - Sample Configurations - Server Groups**, and select **ClaimServerGroup**.
   b. Open the pop-up menu of ClaimServerGroup, and select **Edit.** This opens the Object Editor.
   c. Click the **Security Service** tab.

d. Change the security enabled field from `no` to `yes`.
e. Click the **OK** button.

7. Activate the configuration.
   a. Expand **Management Zones - Sample Cell and Workgroup Zone - Configurations**, and select **Sample Configuration**.
   b. Open the pop-up menu of Sample Configuration, and select **Activate.** A window should appear, confirming that the configuration has been activated.

Running the Application
To run the client application:

1. Make sure the SOMCBENV environment variable is correctly set (for example, myhost.myintranet.com Default Client)
2. From a command prompt, change directory to where the ClaimApp executable is stored:

   WIN `x:\CBroker\MyProject\working\NT`

   AIX `$HOME/MyProject/Working/AIX`

3. Enter:

```
ClaimApp
```

When the application finishes running, the new Claim component is created (with values for its ClaimNo and state stored in the ClaimDB database) and the get and set accessor methods of the component.

Summary
You have bound the server application to the database, installed and configured the server application, and run the client application.

You are ready to continue to one of the next scenarios, in which you either debug your client and server applications, or remove the application from the server:

- WIN "Trace and Debug an Application - Scenario"
- "Uninstall an Application - Scenario" on page 71 (without using InstallShield)

RELATED TASKS
Load a new Application
Add a Client Application into a Configuration of your Application Environment
Configure Applications onto a Client Style

Trace and Debug an Application - Scenario

Objectives
To debug a fully deployed Component Broker application, on a client and server (both on the same machine, for this exercise).

Before You Begin
This scenario is a continuation of the scenario series:

1. "Create a Component - Scenario" on page 39
2. "Build DLLs or Shared Library Files - Scenario" on page 47
3. "Package an Application - Scenario" on page 50

4. “Install and Run an Application Using InstallShield - Scenario” on page 57 or “Install and Run an Application - Scenario” on page 61 (without using InstallShield)

You should complete the previous scenarios before attempting this one.

This scenario runs on a server machine. While the previous scenarios can be run on a dedicated development machine, this scenario requires that you have CBConnector System Manager installed.

The following steps assume that you are running Object Level Trace and the client application on a single NT or AIX workstation. You must interact with the Debugger on NT. For information on deploying Object Level Trace across multiple workstations, please see the online documentation.

You need the following installed:

- CBConnector (System Management)
- CBToolkit
- DB2 Universal Database
- WIN VisualAge for C++ and (for Java applications) VisualAge for Java
- Java SDK

If you are deploying a Java component, make sure your CLASSPATH is set correctly, as described in the following topic:

- “Requirements for Java Development” on page 8

Description

The CBToolkit provides two debugging tools:

- A local debugger for testing client-side code or for debugging business objects prior to deployment.
- Object Level Trace (OLT) which enables you to monitor the flow of a distributed application and to debug client and server code seamlessly from a single workstation.

To trace and debug a distributed application using OLT, you need to:

1. Enable remote tracing and debugging using the System Manager User Interface.
2. Start Object Level Trace.
3. Start the OLT Client Controller.
4. Run your client application to produce a trace display
5. Set breakpoints on the trace display
6. Run the client application a second time to debug client and server code.

If you did not compile your server DLL with OLT flags, you must do so now, then regenerate your code.

Once you have debugged the program, you can continue to one of the next scenarios, depending on whether you used InstallShield to install your application:

- WIN “Uninstall an Application Using InstallShield - Scenario” on page 70
- “Uninstall an Application - Scenario” on page 71 (without using InstallShield)

Enabling Remote Tracing and Debugging

Your name and application servers should already be running. To enable remote debugging, you need to stop the application server and edit your server and client properties as follows:

1. From the System Manager User Interface, expand Host Images - *myhost.austin.ibm.com* - Server Images, and select *myserver*. Where *myserver* is your application server image.
2. Open the pop-up menu of *myserver*, and select **Stop**. You can verify that the application server has stopped using the NT Task Manager.
3. Again open the pop-up menu of *myserver*, and select **Edit**. The Object Editor notebook is opened.
4. Select the **Main** tab.
5. Change the **debug enabled** field to yes.
6. Select the **Orb** tab.
7. Change the **request timeout** field to 0.
8. Click the **OK** button to exit this notebook.
9. Expand Host Images - *myhost.austin.ibm.com* Client Style Images, and select *myclient* image. Where *myclient* is your application client image.
10. Open the pop-up menu of *myclient*, and select **Edit.** The Object Editor notebook is opened.
11. Select the **Main** tab.
12. Change the **debug enabled** field to yes.
13. Select the **Orb** tab.
14. Change the **request timeout** field to 0.
15. Click the **OK** button to exit this notebook.
16. From the System Manager User Interface, expand Host Images - *myhost.austin.ibm.com* - Server Images, and select *myserver*.
17. Open the pop-up menu of *myserver*, and select **Run Immediate**. Monitor the Action Console window for completion status.

Note:
The claimapp client application contains a transaction timeout value. You must set this value to 0 and recompile before continuing.

1. In the claimapp.cpp file located in your Object Builder working directory, edit the transaction timeout value as follows:
2. Change line 55 to:
   ```
   current_transaction->set_timeout(0);
   ```
3. Change line 108 to:
   ```
   current_transaction->set_timeout(0);
   ```
4. Save the file.
5. In your working directory, enter:
   ```
   set IVB_TRACE_DEBUG=1
   ```
6. To recompile the makefile, enter:
   ```
   nmake
   ```

Starting Object Level Trace

To start OLT:

1. From the Windows NT **Start** menu, select **Programs - IBM Component Broker for Windows NT - Object Level Trace (OLT)**. This starts the Server process and opens the Viewer window. (On AIX, type `ivbtrsrv`.)
2. In the Viewer window, select **Options - Online mode**. An information message is displayed.
3. Click the **OK** button.

Starting the OLT Client Controller
The OLT Client Controller contains settings that allow the various OLT components (Server, Viewer, and so forth) to communicate with each other. You need to start the OLT Client Controller before running your application.

To start the OLT Client Controller:

1. From the Windows NT **Start** menu, select **Programs - IBM Component Broker for Windows NT - OLT Client Controller**. This opens the Client Controller window. (On AIX, type `ivbtrc`.)
2. Minimize the window.

Running the Client Application

C++ Client Application:

In the OLT Viewer, select **File - Start process** and browse for your executable, or start the application from the command line. On AIX, you should start the application in a new shell.

Java Client Application (Trace only):

```
java
-Dcom.ibm.CORBA.BootstrapHost=labadie01.torolab.ibm.com
-Dcom.ibm.CORBA.EnableApplicationOLT=true
-Dcom.ibm.CORBA.ApplicationOLTHome=c:/winnt/profiles/labadie01
claimapp
```

where:
labadie01.torolab.ibm.com = your server application host name
c:/winnt/profiles/labadie01 = directory pointed to by IVB_HOME environment variable

The client application begins calling methods on the Component Broker server. The OLT Viewer should begin showing trace lines and event symbols. Once your application is finished, you can use the trace display to set breakpoints on server events.

Setting breakpoints on the trace display
Debuggable methods are represented on the display by filled circles. To set a breakpoint:

1. Select any filled circle and click mouse button two.
2. From the popup menu, select **Add to breakpoint list**.
3. Repeat for each event you want to add.

To see a list of your breakpoints, select **Breakpoints - Create breakpoints**.

Starting the OLT Debugger Daemon
From a Windows NT **Start** menu, select **Programs - IBM Component Broker for Windows NT - OLT Debugger Daemon**.

The Debugger Daemon is started in a shell window. Minimize it, but do not close it.

Changing to Debug Mode

In the OLT Client Controller:

1. Select **Monitoring mode** from the tree view
2. Select **Trace and debug with prompt**.
3. Click **Apply**, then minimize the Client Controller.
4. If you started the OLT Debugger Daemon on another workstation, select Remote Debugger from the tree view and enter the host name of the machine where you started the daemon.

In the OLT Viewer window, deselect **Options - Step by step debug mode** (otherwise, the debugger will open on every debuggable event, instead of stopping only at your breakpoint).

Debugging Server and Client Code
You are now ready to rerun the application.The Java command is slightly different this time because the Java debugger must be invoked.

C++ Client Application:

In the OLT Viewer, select **File - Start process** and browse for your executable, or start the application from the command line.

Java Client Application (Trace and Debug):

```
java_g -debug
-Dcom.ibm.CORBA.BootstrapHost=labadie01.torolab.ibm.com
-Dcom.ibm.CORBA.EnableApplicationOLT=true
-Dcom.ibm.CORBA.ApplicationOLTHome=c:/winnt/profiles/labadie01
claimapp
```

where:
labadie01.torolab.ibm.com = your server application host name
c:/winnt/profiles/labadie01 = directory pointed to by IVB_HOME environment variable

When your program reaches the first breakpoint, the debugger opens and steps into your server method. To continue, click the **Run** button on the debugger toolbar. If you execute a **Step over** command at the completion of your method, you can follow the transaction back to the client code and continue debugging there.

Note:
Do not close the debugger window while your application is running. Closing this window brings down your application server. When your application has finished running, and you have completed your trace and debugging activities, you can close the various OLT and debugger windows.

Summary
You have debugged your client and server application.

You are ready to continue to one of the next scenarios, depending on whether you used InstallShield to install your application:

- WIN "Uninstall an Application Using InstallShield - Scenario"
- "Uninstall an Application - Scenario" on page 71 (without using InstallShield)

Uninstall an Application Using InstallShield - Scenario

Objectives
To remove a CB application from System Management, using InstallShield.

Before You Begin
This scenario is a continuation of the scenario series:

1. "Create a Component - Scenario" on page 39
2. "Build DLLs or Shared Library Files - Scenario" on page 47
3. "Package an Application - Scenario" on page 50
4. WIN "Install and Run an Application Using InstallShield - Scenario" on page 57
5. "Trace and Debug an Application - Scenario" on page 65 (optional)

You should complete the previous scenarios before attempting this one.

You need the following installed:

- CBConnector (System Management)
- DB2 Universal Database
- WIN VisualAge for C++ and (for Java applications) VisualAge for Java
- Java SDK

Description
Use the following procedure to uninstall a client application. Before an application can be uninstalled, the Name server and the application server must be running, in System Manager.

Uninstalling the Application
To clean up the results of the previous exercises and remove the Claim component and the ClaimApp application from your system:

1. From the System Manager User Interface, expand **Host Images - myhost.austin.ibm.com - Configured Servers (member of group)**, and select **ClaimServer**.
2. From the pop-up menu for ClaimServer, select **Stop Immediate**.
3. From a DB2 command prompt, enter: `drop database ClaimDB`
4. Run the InstallShield uninstall.
   a. Run `x:\CBroker\working\NT\ClaimAppFam\Disk1\setup.exe`.
   b. Click the **Next** button to continue to the Application Installation window.
   c. Select `ClaimAppS`.
   d. Click the **Next** button.
   e. Click the **Next** button on the confirmation screen.
   f. Click the **OK** button.

g. Click the **Cancel** button; otherwise, the uninstall process will begin again. The applications are removed for system management.

Summary
You have removed the application from the server, and have completed all the scenarios in this sequence.

For additional development scenarios, consult the index, or search the online Component Broker information for the keyword “Scenario”.

Uninstall an Application - Scenario

Objectives
To remove a CB application from System Management, without using InstallShield.

Before You Begin
This scenario is a continuation of the scenario series:

1. “Create a Component - Scenario” on page 39
2. “Build DLLs or Shared Library Files - Scenario” on page 47
3. “Package an Application - Scenario” on page 50
4. “Install and Run an Application - Scenario” on page 61 (without using InstallShield)
5. “Trace and Debug an Application - Scenario” on page 65 (optional)

You should complete the previous scenarios before attempting this one.

You need the following installed:

- CBConnector (System Management)
- DB2 Universal Database
- WIN VisualAge for C++ and (for Java applications) VisualAge for Java
- Java SDK

Description
Use the following procedure to uninstall a client application. Before an application can be uninstalled, the Name server and the application server must be running, in System Manager.

WIN If you installed your application on Windows NT using InstallShield, use the procedure “Uninstall an Application Using InstallShield - Scenario” on page 70 to remove the application.

Uninstalling the Application
Use the following procedure to uninstall a client application. Before an application can be uninstalled, the Name server and the application server must be running.

To uninstall an application:

1. Start the System Manager User Interface and set the User Level of Expert.
2. Expand **Host Images - myhost - Application Family Installs**, and select **myapplicationfamily**.
3. Open the pop-up menu for myapplicationfamily, and select **Uninstall Family**.

Due to the number of objects that need to be accessed, updated and deleted by this operation, you may see the application server cycle several times. Monitor the Uninstall Action Console for a success statement before continuing. If the application server fails and does not recycle, try restarting it. To restart the application server:

1. Expand **Host Images - myhost - Server Images**, and select **myserver**.
2. Open the pop-up menu of myserver, and select **Run Immediate**.

If you wanted only to uninstall the application, you can reactivate your configuration. To reactivate:

1. Expand **Management Zones - mymanagementzone - Configurations**, and select **myconfiguration**.
2. Open the pop-up menu of myconfiguration, and select **Activate**.

Important:
If there are two application families (a basic and a specific family), only the basic family needs to be uninstalled. The System Manager will uninstall the specific family first, then the basic family. For example, if the specific family is SpecificClaimAppFam and the basic family is ClaimAppFam, you should uninstall only ClaimAppFam.

Summary
You have removed the application from the server, and have completed all the scenarios in this sequence.

For additional development scenarios, consult the index, or search the online Component Broker information for the keyword “Scenario”.

Chapter 4. Working with Rose

Using Rational Rose with Object Builder

You can design your application in Rose, and then export the design to Object Builder. You can also import an Object Builder application back into Rose. You need to modify Rose 98 to support the import and export process, which uses the Rose Bridge.

If you export an incomplete design to Object Builder and make changes to it in its Object Builder form, make sure you import the Object Builder project back into Rose before doing any more work with the Rose model. If you do not import the changed project and continue work with the new model, your changes will be lost the next time you export.

You can export and import the following design elements, using the Rose Bridge:
- Classes (mapped either to component objects or to IDL constructs)
- Packaging information (mapped to projects and name scoping)
- One-to-one relationships (mapped to object references)
- One-to-many relationships (mapped either to an object relationship stored in a collection, or to a sequence attribute).
- Class inheritance (mapped to component inheritance)

By default, classes in your design are mapped to components in Object Builder.

The main design tasks are as follows:
1. “Import Component Broker Frameworks” on page 86
2. “Export a Design from Rose” on page 89
3. “Export a Rose Design to a Team Environment” on page 204
4. Import a Rose Design into Object Builder
5. “Work with an Exported Design” on page 91
6. “Import a Project into Rose” on page 92
7. “Import Projects from a Team Environment” on page 212

RELATED CONCEPTS
“The Rose Bridge” on page 76
“Design Principles for Component Broker Applications” on page 3
“Components” on page 15
“IDL Name Scoping in Rose” on page 77
“Constructs You Can Export from Rose” on page 79
“Class Properties You Can Export from Rose” on page 81
“Class Relationships You Can Export from Rose” on page 84
“Component Broker Frameworks in Rose” on page 89

RELATED TASKS
“Set up Rose 98” on page 74

© Copyright IBM Corp. 1997, 1998

Rose

Rational Rose is an object-oriented analysis and design modeling tool. You can use it to design your application, and then export the design to Object Builder, where you can finish its implementation. You can also import Object Builder projects into Rose, to work with an existing design.

Note: In order to export to and import from Object Builder, you must use the full version of Rose 98 (Enterprise edition, Professional C++ edition, or Professional Java edition), not just the Rose Modeller. Only the full version supports code generation properties, which are required by the export process.

AIX Until Rose 98 (Enterprise edition, Professional C++ edition, or Professional Java edition) becomes available on AIX, you must do your design work on Windows NT, then export to Object Builder on Windows NT, before you can move your model to AIX and complete your work on it.

To use Rose with Object Builder, you must first customize it, and then import the Component Broker frameworks. You can then use the Component Broker frameworks in your design.

When you export, the classes and relationships you defined in Rose are mapped to IDL equivalents in Object Builder. You can also define additional properties in Rose, to have the Rose Bridge create additional Component Broker objects for your design during the export process.

When you import, the elements of the Object Builder project are mapped to their equivalents in Rose.

RELATED CONCEPTS
"The Rose Bridge" on page 76
"IDL Name Scoping in Rose" on page 77
"Constructs You Can Export from Rose" on page 79
"Class Properties You Can Export from Rose" on page 81
"Class Relationships You Can Export from Rose" on page 84
"Mapping Rules: Object Builder to Rose" on page 87
"Component Broker Frameworks in Rose" on page 89

RELATED TASKS
"Set up Rose 98"
"Export a Design from Rose" on page 89
"Export a Rose Design to a Team Environment" on page 204
"Import a Project into Rose" on page 92
"Import Projects from a Team Environment" on page 212

Set up Rose 98

You can use Rose to create a design for your application, which you can then export to Object Builder. You can also import Object Builder projects into Rose, to work with an existing design.

Note: In order to export to and import from Object Builder, you must use the full version of Rose 98 Enterprise edition, Professional C++ edition, or Professional Java edition, not just the Rose Modeler. Only the full version supports code generation properties, which are required by the export process.

AIX Until Rational Rose 98 (Enterprise edition, Professional C++ edition, or Professional Java edition) becomes available on AIX, you must do your design work with Rose on Windows NT, then export to Object Builder and move the project to AIX, before you can complete your work on the project.

Before you can use Rose with Object Builder, you must configure its import and export facility, the Rose Bridge. Once the Rose Bridge is configured, you can load Component Broker frameworks into Rose, create your design, and export to and import from Object Builder.

To configure the Rose Bridge for Rose 98, follow these steps:

1. Add the export and import options to the Rose **File** menu, as follows:
   a. Create a backup copy of the file rose.mnu (for example, rose.bak).
   b. Add the following lines to the rose.mnu file:
   ```
   Menu File
   {
      Separator
      option "Export to Object Builder"
      {
         RoseScript $BOSS_PATH\r982c.ebx
      }
      option "Import from Object Builder"
      {
         RoseScript $BOSS_PATH\c2r98.ebx
      }
   }
   ```
2. Create the Rose path map BOSS_PATH, which will point to the directory path that contains the Component Broker model files. This allows you to import the model files, and specifies the location of the export script (r982c.ebx) and import script (c2r98.ebx). .
   a. Click **File - Edit Path Map**.
   b. Set the BOSS_PATH variable. For example, if you installed the product into <path>\Cbroker, set BOSS_PATH to <path>\Cbroker\rose, which is the directory that contains the *.cat files for the Component Broker model.

Some additional Component Broker-specific properties have been added to Rose to enable more information exchange between Rose and Object Builder. To enable the use of these additional properties, you must replace the roseidl.pty and roseddl.pty files that come with Rose 98 with the Component Broker versions. To replace these files, do the following steps:

1. Change to the directory where Rose 98 is installed
2. Create a backup copy of the file roseidl.pty (for example, roseidl.bak)
3. Create a backup copy of the file roseddl.pty (for example, roseddl.bak)
4. Copy the Component Broker versions of these files to the current directory from the rose subdirectory of your Component Broker install:
   ```
   copy <path>\CBroker\rose\*.pty
   ```

RELATED CONCEPTS

"Rose" on page 74
"The Rose Bridge" on page 76

RELATED TASKS

"Import Component Broker Frameworks" on page 86

"Using Rational Rose with Object Builder" on page 73
"Export a Design from Rose" on page 89
"Import a Project into Rose" on page 92

The Rose Bridge

The Rose Bridge provides the ability to export from Rose to Object Builder, and to import from Object Builder to Rose.

Loading the Component Broker Frameworks
Before you design in Rose, you need to load the Component Broker frameworks. The Component Broker Toolkit includes Rose .cat files for the Component Broker frameworks. Once you import the .cat files into Rose, you can incorporate the Component Broker Framework interfaces in your design and analysis work using Rose.

The Rose .cat files contain the following frameworks:
- services.cat: Object Services
- boim.cat: Business Object Application Adaptor Frameworks (BOIMs)
- managed.cat: Managed Object Framework (MOFW)

Exporting a Design
Once you have completed a design, you can export it to an Object Builder project. You can export a design for use in a single Object Builder project, or break up the design into separate projects by top-level package name. The export process updates the \Model subdirectory of the selected project or projects, and also creates the XML files for the model in the project's \Import directory. Once the export is complete, you can use Object Builder to further refine the model and to generate code

You should not restructure your design after exporting. If you restructure your design (for example, move a class from one package to another), the export process will treat the change as a combination add and delete, rather than a move. This would result in two definitions of the class in Object Builder (a new class definition for its new position, and the old class definition for its old position), which is not valid.

When you export from Rose, the export process generates a file named xmi.xml in the \XMI subdirectory of the target project, or the parent directory of a multiple-project target. This file allows the export process to track changes to design elements. For example, if you change the name of a method in Rose and re-export, the information in the xmi.xml file will ensure that the change will be applied to the appropriate method in Object Builder.

Importing a Design
You can make changes to the design in Object Builder, and then apply the changes to the original Rose model. If you are doing work in both Object Builder and Rose, make sure you keep the two versions synchronized. For example, if you change the Object Builder model, import the changes into Rose before doing any more work on the Rose model.

Re-Exporting a Design
When you re-export a model, the export process will add new elements to Object

Builder, or update existing elements, but will not delete elements that already exist in Object Builder. To delete existing elements, you must work directly in Object Builder.

Directories Used

The Rose Bridge process uses subdirectories of the targetted Object Builder projects to store information in, as follows:

- *project*\Model
  Contains the target Object Builder project model.
- *project*\Import
  Contains the udbo.xml file created by the export from Rose. This file defines all the elements that are importable into Object Builder, and is used to create the project model.
- *project*\XMI
  Contains the xmi.xml file created by an export or import from or to Rose. This file stores all the elements that cannot be mapped between a Rose model and an Object Builder model.

RELATED CONCEPTS

"Rose" on page 74
"IDL Name Scoping in Rose"
"Constructs You Can Export from Rose" on page 79
"Class Properties You Can Export from Rose" on page 81
"Class Relationships You Can Export from Rose" on page 84
"Mapping Rules: Object Builder to Rose" on page 87
"Component Broker Frameworks in Rose" on page 89

RELATED TASKS

"Import Component Broker Frameworks" on page 86
"Set up Rose 98" on page 74
"Export a Design from Rose" on page 89
"Export a Rose Design to a Team Environment" on page 204
"Work with an Exported Design" on page 91
"Import a Project into Rose" on page 92
"Import Projects from a Team Environment" on page 212

IDL Name Scoping in Rose

The name scoping used by Component Broker is based on CORBA IDL, where a containment relationship exists among IDL files, modules, and interfaces. In the Rose model, a containment relationship exists among packages (categories), subpackages (subcategories), and classes (interfaces or data types). The export process from Rose supports the same containment relationship as Object Builder. For this to work, some restrictions on what gets mapped into Object Builder are necessary.

When the Rose model is exported to Object Builder, containment relationships are mapped in the following manner (this refers to the Logical View only):

- A top-level package can be mapped to a project, if you are exporting the design to multiple projects. A package is top-level if it is contained directly by the Logical View. If you are exporting the design to a single project, then the top-level packages are not mapped to anything.

- An intermediate package is not mapped to anything. The package is considered to be just a grouping construct in Rose. A package is intermediate if it contains other subpackages.
- A Rose bottom-level subpackage is mapped to an IDL file that contains a module in the Object Builder model. A bottom-level subpackage is a subpackage that contains no other subpackages.
- A class in Rose is mapped to either an interface or an IDL complex type (such as a struct or enumeration) in Object Builder. If the class is contained in a bottom-level subpackage, then the class it maps to will be contained in the corresponding module, otherwise, the class will be directly contained in a file with the same name.

When an Object Builder model is imported into Rose, the containment relationships are mapped as follows:

- Business object files, modules, and interfaces that already have a mapping (because they were created by export from Rose) maintain that mapping.
- New business object files, modules, and interfaces (added directly to Object Builder, not by export from Rose) are mapped to packages, subpackages, and classes.

Simple Example of Export Mapping
An Insurance Application package contains the four classes Policy, PayoutFraction, Customer, and Agent.

When you export this design, the Insurance Application package is considered a top-level package or grouping mechanism, and is not mapped to anything. The four classes become four business object interfaces, each with its own IDL file.

Export Example with Subpackages
An Insurance Application package contains two subpackages: Policy (containing the classes Policy and PayoutFraction) and People (containing the classes Customer and Agent).

When you export this design, the Insurance Application package is again ignored. The Policy subpackage is a bottom-level subpackage, and becomes the Policy IDL file, which contains a Policy module, which contains the interfaces Policy and PayoutFraction. The People subpackage (also bottom-level) becomes the People IDL file, which contains a People module, which contains the interfaces Customer and Agent.

Policy.idl will contain the following declarations:

```
module Policy
{
        interface Policy
        {
        };
        interface Payout Fraction
        {
        };
};
```

People.idl will contain the following declarations:

```
module People
{
        interface Customer
        {
        };
```

```
        interface Agent
        {
        };
};
```

RELATED CONCEPTS

"The Rose Bridge" on page 76
"Constructs You Can Export from Rose"
"Class Properties You Can Export from Rose" on page 81
"Class Relationships You Can Export from Rose" on page 84

RELATED TASKS

"Export a Design from Rose" on page 89

Constructs You Can Export from Rose

You can specify constructs in Rose by defining classes with the IDLSpecificationType appropriate to the construct.

The IDLSpecificationType is set on the IDL page of the Class Specification notebook. By default, it is set to Interface (so the class will become a business object interface in Object Builder). You can change the default to one of:

- Struct
- Enumeration
- Typedef
- Union
- Const
- Exception

The export process preserves or maps information for the following types of constructs:

Struct
On the IDL page of the Class Specification notebook, set the IDLSpecificationType to Struct. You can specify the following information for a struct:

- **Name**
  Becomes the name of the struct. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
- **Attributes**
  Map to members of the struct. The members have names and types:
  - **Name**
    Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
  - **Type**
    Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.

Enumeration
On the IDL page of the Class Specification notebook, set the IDLSpecificationType to Enumeration. You can specify the following information for an enumeration:

- **Name**
  Becomes the name of the enumeration. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
- **Attributes**
  Map to members of the enumeration. The members have names and types:
  - **Name**
    Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
  - **Type**
    Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.

Typedef
On the IDL page of the Class Specification notebook, set the IDLSpecificationType to Typedef, and set the ImplementationType to the type this is a typedef for.

You can specify the following information for a typedef:

- **Name**
  Becomes the name of the typedef. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.

Union
On the IDL page of the Class Specification notebook, set the IDLSpecificationType to Union, and set the ImplementationType to the type of the union switch.

You can specify the following information for a union:

- **Name**
  Becomes the name of the union. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
- **Attributes**
  Map to members of the union. The members have names and types:
  - **Name**
    Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
  - **Type**
    Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.

Const
On the IDL page of the Class Specification notebook, set the IDLSpecificationType to Const, set the ImplementationType to the type of the const, and set the ConstValue to the value of the const.

You can specify the following information for a const:

- **Name**
  Becomes the name of the const. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.

Exception
On the IDL page of the Class Specification notebook, set the IDLSpecificationType to Exception. You can specify the following information for an exception:

- **Name**
  Becomes the name of the exception. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
- **Attributes**
  Map to members of the exception. The members have names and types:
  - **Name**
    Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
  - **Type**
    Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.

RELATED CONCEPTS
"The Rose Bridge" on page 76
"IDL Name Scoping in Rose" on page 77
"Class Properties You Can Export from Rose"
"Class Relationships You Can Export from Rose" on page 84

RELATED TASKS
"Export a Design from Rose" on page 89

Class Properties You Can Export from Rose

When you define a class in Rose, it can be mapped either to a business object interface, or to a type of construct, in Object Builder. For constructs, the name, documentation, and attributes (when appropriate) are preserved by the export process. For business object interfaces, both class properties and class relationships are preserved.

A class is exported as a business object interface by default, based on the setting of the IDLSpecificationType (which is set to Interface by default). The IDLSpecificationType is set on the IDL page of the Class Specification notebook. You can set additional properties of the class and of its attributes to create additional component objects for the class (such as the business object implementation) when you export.

The export process preserves or maps the following class information:

Name
The name you give the class becomes the name of the business object interface in Object Builder. If, during the export process, you select to generate additional

component objects (such as a business object implementation and data object interface), the additional objects are given names derived from the class name.

You should specify a valid C++ class name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores. For example, My Class Name would become My_Class_Name.

Documentation
Any documentation you enter for the class becomes comments in Object Builder, where they can be accessed from the last page of the Business Object Interface wizard. Do not include /* or */ in the documentation text: the generated code from Object Builder will provide C++ comment tags around your entries by default.

IsQueryable
Defines whether the specified interface is queryable. You can set the IsQueryable property to true on the IDL page of the Class Specification notebook.

Component Objects
Component objects, in addition to the business object interface, are created in Object Builder as follows:

- **CreateImplementation**
  Defines whether a business object implementation, and its accompanying data object interface, will be created for the specified interface. You can set the CreateImplementation property to true on the IDL page of the Class Specification notebook.
- **CreateKey**
  Defines whether a key will be created for the specified interface. You can set the CreateKey property to true on the IDL page of the Class Specification notebook.
- **CreateCopyHelper**
  Defines whether a copy helper will be created for the specified interface. You can set the CreateCopyHelper property on the IDL page of the Class Specification notebook

Attributes
Attributes of the class map to attributes of the interface, as follows:

- **Name**
  Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores. For example, my Data Name would become my_Data_Name.
- **Type**
  Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
- **Initial Value**
  Value placed in this field is transferred to the Initializer field for the attribute in Object Builder. This value should be consistent with the type defined for the attribute.
- **Access Control**
  Can be one of public, protected, or private, and maps as follows:
  - Public attributes map to attributes of the business object interface. The business object implementation will have get and set methods defined for these attributes.

– Protected attributes map to protected attributes of the business object implementation. They do not appear in the business object interface.
– Private attributes map to private attributes of the business object implementation. They do not appear in the business object interface.

- **Length**
  Defines the string length if the attribute is of type string. You can set the Length property to the appropriate value on the DDL page of the Attribute Specification notebook.
- **Key Attribute**
  Defines whether the attribute is included in the key object for the component (if you set the CreateKey property for the class). You can set the PrimaryKey property to true on the DDL page of the Attribute Specification notebook.

  Key attributes for parents will automatically be included in the child's key.

  Note: Do not specify complex types (such as structures or unions) as keys.
- **Copy Helper Attribute**
  Defines whether the attribute is included in the copy helper for the component (if you set the CreateCopy property for the class). By default, all public attributes are included in the copy helper. You can set the IsIncludedInCopyHelper property to false on the DDL page of the Attribute Specification notebook to exclude an attribute.
- **Override**
  If you define an attribute in a child class that has the same name and type as an attribute in its parent class, the attribute will be defined as an override in Object Builder, in the Business Object Implementation wizard, Attributes to Override Page.
- **Read-Only**
  Defines whether the attribute is read-only. By default the attribute is not read-only. You can set the IsReadOnly property on the IDL page of the Attribute Specification notebook.

Operations
Operations of the class map to methods of the interface, as follows:

- **Name**
  Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores. For example, my Method Name would become my_Method_Name.
- **Return Class**
  Maps to the return type for the method. Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
- **Export Control**
  Can be one of public, protected, or private, and maps as follows:
  – Public operations map to methods of the business object interface.
  – Protected operations map to protected methods of the business object implementation. They do not appear in the business object interface.
  – Private operations map to private methods of the business object implementation. They do not appear in the business object interface.
- **Override**
  If you define an operation in a child class that has the same name and signature

as an operation in its parent class, the operation will be defined as an override in Object Builder, in the Business Object Implementation wizard, Methods to Override Page.

- **Argument Name**
  Maps to a parameter of the method. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores. For example, first Parameter would become first_Parameter.
- **Argument Type**
  Maps to the parameter type. Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
- **Argument Default**
  Should be one of:
  - `in`
  - `out`
  - `inout`

  If you specify a different value, it will be ignored, and `in` will be used.
- **Exceptions**
  Map to exceptions raised by the method. You can specify these exceptions in Rose on the Details page of the Operation Specification notebook. The exceptions should be of a valid type (either one defined in the current Rose model, or one previously defined in the target Object Builder model).
- **One-Way Property**
  Maps to the one-way property in Object Builder. Set the OperationIsOneWay property on the IDL page of the Operation Specification notebook. The property defaults to False.

RELATED CONCEPTS
"The Rose Bridge" on page 76
"IDL Name Scoping in Rose" on page 77
"Constructs You Can Export from Rose" on page 79
"Class Relationships You Can Export from Rose"

RELATED TASKS
"Export a Design from Rose" on page 89

Class Relationships You Can Export from Rose

When you export your object model from Rose into Object Builder, the class relationships you have defined are mapped as follows:

Inheritance
Inheritance relationships you define in Rose are preserved by the export process, and applied to the business object interfaces that the exported classes are mapped to. If you are generating additional component objects for a class (an option of the export process), then the inheritance for the additional components parallels the inheritance for the business object interface.

For example, if ChildClass inherits from ParentClass, then after the export the ChildClass business object interface inherits from the ParentClass business object

interface. If you added business object implementations during the export, then in addition the ChildClassBO business object implementation inherits from the ParentClassBO business object implementation.

Associations and Aggregations
Associations and aggregations map to attributes, object relationships, or sequences, as explained below. Associations are only mapped if they are navigable. Aggregations are always mapped.

The export process preserves or maps the following information about the relationship:

- **Role A and Role B**
  Role A and Role B are the terms in Rose that define the two ends of an association. In the Association Specification notebook, the names you specify for the roles (on the Role A General and Role B General pages) determine the names of the attributes or relationships that each class has to represent its association with the other.

  The names you specify should be valid C++ names. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores. If you do not specify names for the roles, then default names based on the names of the referenced interfaces are used.

  For example, if a class named Agent is in Role A and a class named Customer is in Role B, the relationship is 1..1 and no names are specified, then Agent gets an attribute named `the_Customer` of type Customer, and Customer gets an attribute named `the_Agent` of type Agent.
- **Cardinality**
  The cardinality of the relationship is set on the Role A Detail or Role B Detail page of the Association Specification notebook. If the cardinality is set to one of 0..n, 1..n, or n, then it is considered to be a cardinality of 'many', and the relationship will be mapped to either an object relationship (stored in a reference collection) or an attribute of type sequence *ClassName* (where *ClassName* is the name of the class in the n role). With all other cardinalities, the relationship will be mapped to attributes of type *ClassName*.

  When the cardinality is 'many', you can choose whether to map as an object relationship or a sequence with the MapAsObjectRelationship property.
- **Class Relationship Mapping**
  For class relationships with role cardinality set to 'many', the MapAsObjectRelationship property defines whether the class relationship is exported as an object relationship or a sequence. By default, relationships are exported as object relationships. To export a relationship as a sequence, set the MapAsObjectRelationship property to false on the IDL A or IDL B page of the Association Specification notebook.
- **Relationship Implementation**
  If the class relationship has been set to export as an object relationship (MapAsObjectRelationship set to true in the appropriate IDL A or IDL B page of the association notebook), you can specify the implementation type for this object relationship. The RelationshipImplementation property on the IDL A or IDL B page of the Association Specification notebook can be set to one of the three following values:
  - Local Persistent Reference
  - User-defined OO_SQL Query
  - Reference Resolved by Foreign Key

- **Read Only or Read/Write**
  You can specify that a role in an association be read-only. When the association is exported, then any corresponding attribute is marked accordingly. You can specify whether an attribute is read-only on the appropriate page for the role (IDL A and IDL B). You can set the isReadOnly property on these pages to true or false. By default, the property is set to false (attributes have read/write access).

  For example, if Role A (Agent) is read-only, then Role B's attribute (Customer's attribute the_Agent of type Agent) is read-only.

  When one-to-many associations are exported as relationships, the read-only mark does not apply to the relationship. For example, if Agent has a one-to-many relationship to Customer, and Customer's role is read-only, the one-to-many relationship is still read-write.

  When one-to-many relationships are exported as sequence attribute, the read-only mark does apply. For example, if Customer's role is marked read-only, Agent can have a read-only sequence attribute of type Customer.
- **Access Control**
  If the relationship is being exported as an attribute, then the access control you set is applied. For example, if Agent's role is marked protected, then Customer's attribute of type Agent will be protected. The access control can be one of public, protected, or private, and maps as follows:
  - Public attributes map to attributes of the business object interface. The business object implementation will have get and set methods defined for these attributes.
  - Protected attributes map to protected attributes of the business object implementation. They do not appear in the business object interface.
  - Private attributes map to private attributes of the business object implementation. They do not appear in the business object interface.

RELATED CONCEPTS
"The Rose Bridge" on page 76
"IDL Name Scoping in Rose" on page 77
"Constructs You Can Export from Rose" on page 79
"Class Properties You Can Export from Rose" on page 81

RELATED TASKS
"Export a Design from Rose" on page 89

Import Component Broker Frameworks

In order to use Component Broker concepts in your analysis and design, you can import Component Broker frameworks, such as the Managed Object Framework, into Rose.

To import Component Broker frameworks into Rose, follow these steps:
1. Start Rose.
2. Create a new model (using the **File - New** menu option) or load an existing model.
3. Import the Component Broker Framework .cat files (managed.cat, services.cat, boim.cat):
   a. Click **File - Units - Load**. A dialog box opens, in which you can specify the model files you want to import.
   b. In the **Files of type** field, type *.cat.

c. Navigate to the <path>\Cbroker\rose directory.

d. Select one of the framework .cat files.

e. Click **Open**. The model files are loaded into Rose, and presented in the current view as a category or package that is selected by default.

f. Click elsewhere in the current view (to avoid importing the next .cat file into the selected category).

g. Load the other .cat files, using the same procedure.

Note: Imported categories all appear in the same spot in the current view, which means you often only see the last-imported category in the view. You can drag and drop a category to reveal the categories beneath.

You can now use the framework concepts in your design.

4. Perform the analysis and design of your application in Rose and save the design model.

If you are using Rose 98 with a model which was created in Rose 4.0, any Component Broker-specific properties you have customized (for example,. IDLSpecificationType) will be moved to the appropriate pages of the Rose 98 specification notebooks the first time the model is exported.

RELATED CONCEPTS

"Rose" on page 74
"Component Broker Frameworks in Rose" on page 89

RELATED TASKS

"Using Rational Rose with Object Builder" on page 73
"Export a Design from Rose" on page 89

Mapping Rules: Object Builder to Rose

When you import an Object Builder model into Rose, elements in the Object Builder model map to elements in a Rose model as follows:

- Business object files, modules, and interfaces that already have a mapping (because they were created by export from Rose) maintain that mapping.
- New business object files, modules, and interfaces (added directly to Object Builder, not by export from Rose) are mapped to packages, subpackages, and classes.

Business Object File

Business object files in Object Builder that already have a mapping (because they were created by export from Rose) maintain that mapping. New business object files are mapped to packages. Constructs defined with file scope map to top-level classes (for an existing mapping) or classes in the package (for new mappings). The IDLSpecification property of the class on the IDL page of its Class Specification notebook is set to one of:

- Struct
- Enumeration
- TypeDef
- Union
- Const
- Exception

Business Object Module
Business object modules in Object Builder that already have a mapping (because they were created by export from Rose) maintain that mapping. New business object modules are mapped to subpackages of the file package. Properties of the module map as follows:

- **Name**
  Stored as the subpackage name in the Package Specification notebook.
- **Constructs**
  Constructs with module scope map to classes contained in the subpackage.

  The IDLSpecification property of the class on the IDL page of its Class Specification notebook is set to one of:
  - Struct
  - Enumeration
  - TypeDef
  - Union
  - Const
  - Exception
- **Comments**
  Stored as documentation for the subpackage in the Package Specification notebook.

Business Object Interface
Business object interfaces Object Builder that already have a mapping (because they were created by export from Rose) maintain that mapping. New business object interfaces are mapped to classes in a file package or module subpackage. The IDLSpecification property of the class on the IDL page of its Class Specification notebook is set to Interface. Properties of the interface map to properties of the class as follows:

- **Name**
  Stored as the class name in the Class Specification notebook.
- **Constructs**
  Constructs with interface scope are not imported. Construct information is stored in the xmi.xml file defined in the project's \XMI directory.
- **Interface Inheritance**
  Parent interfaces map to generalize elements in Rose 98, on the Relations page of the Class Specification notebook.
- **Attributes**
  Attributes map to attribute elements of the class in Rose 98.
- **Sequence Attributes**
  If an attribute of type sequence was created in Object Builder by the Rose Bridge (by the export of an association with the MapAsObjectRelationship property set to false), then the import preserves this original mapping. If the attribute was created directly in Object Builder and does not exist in Rose, it maps to an attribute of the class.

  Note: If you defined the association in Rose, exported to Object Builder to create the sequence attribute, and then deleted the attribute in Object Builder, the import will **not** delete the association in Rose, but will set the is_navigable property of the Role to FALSE.
- **Methods**
  Methods map to operation elements of the class in Rose 98.
- **Object Relationships**

If an object relationship was created by exporting an association in Rose, the import preserves the original mapping, to a Role in a many-to-many association or one-to-many association in Rose 98. If the object relationship was created directly in Object Builder and does not have an equivalent in Rose, it is not imported, but is stored in the xmi.xml file in the project's \XMI directory.

Note: If you defined the association in Rose, exported to Object Builder to create the object relationship, and then deleted the relationship in Object Builder, the import will **not** delete the association in Rose, but will set the is_navigable property of the Role to FALSE.

- **Comments**
  Stored as class documentation in class specification notebook.

RELATED CONCEPTS

"Object Builder" on page 1
"Rose" on page 74
"The Rose Bridge" on page 76

RELATED TASKS

"Import a Project into Rose" on page 92
"Import Projects from a Team Environment" on page 212

Component Broker Frameworks in Rose

The structure of the Component Broker Frameworks will appear in Rose in accordance with the Rose-to-IDL name scoping relationship. For example, in IDL, the Managed Object Framework contains several modules. In Rose, the Managed Object Framework appears as a package with subpackages corresponding to the modules that it contains. When you expand any of these subpackages in Rose, the classes (corresponding to the interfaces) that it contains are shown. For example, if you expand the IManagedClient subpackage, the IManageable and IHome classes are displayed.

RELATED TASKS

"Import Component Broker Frameworks" on page 86

Export a Design from Rose

Once you have completed your design work in Rose, you can export your design to an Object Builder project or projects. Rose must first be set up to work with Object Builder, and Object Builder must be set up to use the Component Broker frameworks. When the export is complete, each class in your design will be mapped to a component in Object Builder. A component consists of a number of related objects, and, at minimum, a business object interface.

You can export the entire Rose model, or selected top-level classes or packages in the model. The export process only deals with information in your model's Logical view.

This task describes how to export to a single Object Builder project. You can also export to multiple projects in a team environment, as described in a separate task.

To export the entire model to a single project, follow these steps:

1. Load your design in Rose.

2. Select **File** - **Export to Object Builder**. The Rose Bridge wizard opens to the Export from Rose 98 to Object Builder Page.
3. Specify the Rose model you want to export, and add any necessary virtual symbols and associated actual path mappings to the Virtual Path Mapping listbox.
4. Select the destination directory you want to store your project in. The directory becomes an Object Builder project directory.
5. Select the **One Project** radio button.
6. Select the **Entire Model** radio button.
7. Click **Finish.**

Your model is exported to a project in the specified directory. Your model is saved in Rose as part of the export process.

To export selected top-level classes or packages to a single project, follow these steps:

1. Load your design in Rose.
2. Select **File** - **Export to Object Builder**. The Rose Bridge wizard opens to the Export from Rose 98 to Object Builder Page.
3. Specify the Rose model you want to export, and add any necessary virtual symbols and associated actual path mappings to the Virtual Path Mapping listbox.
4. Select the destination directory you want to store your project in. The directory becomes an Object Builder project directory.
5. Select the **One Project** radio button.
6. Select the **Selected Packages or Classes** radio button.
7. Click **Next** to turn to the Export from Rose 98 to Object Builder, Selection Page.
8. Use standard selection techniques (**Click**, **Shift-Click** and **Ctrl-Click**) to select the top-level packages and classes you want to export.

   As items are selected in the tree view on the left, the tree view on the right displays the component objects or elements which will be created for that item. The export process creates component objects according to the properties in the Class Specification notebook.
9. Click **Finish.**

   Your model is exported to a project in the specified directory. Your model is saved in Rose as part of the export process.

If you are using Rose 98 with a model which was created in Rose 4.0, any Component Broker-specific properties you have customized (for example, IDLSpecificationType) will be moved to the appropriate pages of the Rose 98 specification notebooks the first time the model is exported.

Once you have completed the export process, your design is applied to the \Model directory of the selected project. The interchange file udbo.xml used in the export process is stored in the \Import directory of the selected project. The classes and relationships you defined in Rose have been mapped to their Object Builder equivalents, and any component objects you specified have been defined in skeleton form.

The export process maintains the following design elements:

- The classes in your design
- Class inheritance

- Class relationships
- Attributes
- Methods

The export process adds the following elements:

- File and module objects (the mapping of classes to files and modules depends on the packaging structure used in Rose).
- Read and write methods, for each public attribute.
- Public attributes (get and set methods), to support aggregations of classes and navigable associations among classes.
- Component Broker objects (business object implementation, data object interface, copy helper, key), as specified during the import, in skeleton form.

When you export from Rose, the export process generates a file named xmi.xml in the target *project*\XMI subdirectory. This file allows the export process to track changes to design elements, so that if you change the name of a method in Rose and re-export, the change will be applied to the appropriate method in Object Builder. It also keeps track of any elements that do not have equivalents in both models, so that these elements are not simply lost in the bridging process.

You are now ready to work in Object Builder.

RELATED CONCEPTS

"The Rose Bridge" on page 76
"Components" on page 15

RELATED TASKS

"Using Rational Rose with Object Builder" on page 73
"Export a Rose Design to a Team Environment" on page 204
"Work with an Exported Design"

Work with an Exported Design

Once the export process is complete, you can start working with your design in Object Builder. To work with an exported design, complete the following steps in Object Builder:

1. Review the exported business object interfaces and their equivalent files.
2. Complete the skeleton objects created by the export.
3. Create data object implementations.
4. Create persistent objects and schemas (for communicating with databases).
5. Create managed objects (for packaging and instance management).

Review and edit exported objects as necessary:

1. Select the object in the Tasks and Objects pane.
2. From its pop-up menu click **Properties**. A wizard opens.
3. Click **Next** to go through all the pages of the wizard, and review its properties. Complete or change the contents of the wizard as you require.
4. Click **Finish**. Any changes you made are applied to the current model.

RELATED CONCEPTS

"The Rose Bridge" on page 76

RELATED TASKS

"Using Rational Rose with Object Builder" on page 73
"Add a Data Object Implementation" on page 299
"Add a Persistent Object and Schema" on page 313
"Add a Managed Object" on page 340

Import a Project into Rose

You can import an Object Builder project into Rose. If the imported project was originally created by a Rose export, then the new Rose model created by the import will mirror the information in the original, exported Rose model's Logical view. If your original model has additional information in other views, you can consolidate the two models (the original exported one, and the newly imported one) using the Rose 98 Visual Differencing tool.

If your Object Builder project was created directly in Object Builder (not by export from Rose), then the Rose model is based on default import mappings of Object Builder elements to Rose elements.

Once the import is complete, you can work with the design in Rose, and export the changes back to Object Builder.

To import an Object Builder project into Rose 98, follow these steps:

1. Select **File** - **Import from Object Builder**. The Rose Bridge wizard opens to the Import from Object Builder to Rose 98 Page.
2. Enter the directory of the Object Builder project you are importing.
3. Enter the name of the new Rose model file you are importing to. If you know your project will be imported into a category file (.cat) on Rose 98, then you need to specify the virtual path mapping information by entering the symbol and actual path data.
4. Select the **Import from: One Project** option.
5. Click **Finish**.

The project you selected is imported into the Rose model file you specified.

The import process works as follows:

1. The import process calls the obexport command to generate an XML file for the project (\Export\udbo.xml).
2. The import process checks to see if there is an \XMI\xmi.xml file in the project directory. This file is created by the Rose Bridge to preserve any information that would otherwise be lost during transfer between Rose and Object Builder.
3. The import process generates a Rose model file, based on the udbo.xml file and the xmi.xml file.
4. The import process updates the xmi.xml file to contain any Object Builder information that cannot be imported. For example, details of the implementation, key, and copy helper for a component, that cannot be stored as elements in Rose 98.
5. The import process loads the generated Rose model into Rose 98.

The import process maps Object Builder elements as follows:

- Business object files, modules, and interfaces that already have a mapping (because they were created by export from Rose) maintain that mapping.

- New business object files, modules, and interfaces (added directly to Object Builder, not by export from Rose) are mapped to packages, subpackages, and classes.
- IDL constructs with file or module scope become classes in Rose
- Attributes of an interface become attributes of a class in Rose
- Methods of an interface become operations of a class in Rose
- Parent interfaces become class relations in Rose
- Object relationships that were created by export from Rose are imported as the role of an association.
- Sequence attributes of the interface that were created by export from Rose are imported as the role of an association.

The import process keeps the following elements in the xmi.xml file:

- Component objects other than the business object interface (business object implementation, data object interface, copy helper, key)
- The method bodies
- Object relationships that were created directly in Object Builder (not by export from Rose)
- IDL constructs with interface scope

The import process updates the following properties in the Rose specification notebooks:

- Class Specification, IDL page, CreateImplementation
  property is set if the business object interface has a business object implementation
- Class Specification, IDL page, CreateKey
  property is set if the business object interface has a key
- Class Specification, IDL page, CreateCopyHelper
  property is set if the business object interface has a copy helper
- Class Specification, IDL page, IsQueryable
  property is set if the business object interface has the option **The interface is queryable** checked, in its properties notebook
- Attribute Specification, IDL page, length
  property is set if the attribute is of type string, and has associated size information.
- Attribute Specification, DDL page, IsIncludedInCopyHelper
  property is set if the attribute is part of the component's copy helper
- Attribute Specification, DDL page, PrimaryKey property is set if the attribute is part of the component's key.
- Association Specification, IDL A/B pages, MapAsObjectRelationship
  property is set if an object relationship or sequence attribute in the business object interface was created by exporting the role of an association from Rose
- Association Specification, IDL A/B pages, RelationshipImplementation
  property is set if the object relationship has a selected implementation type in the business object implementation

In order to track changes between Component Broker objects and Rose elements, the import process uses the UUID of an element as an identifier. The UUID is stored as the uuid property of IDL page in Rose for each package, class, attribute, operation, and role of association.

You have now imported an Object Builder project into a Rose model. If the imported project was created by export from Rose, and the original Rose model contains information in other views besides the Logical view, then you should consolidate the new model with the original model before doing any more design work.

To merge the new model with the original model, follow these steps:

1. Click **File-Save** to save the new model.
2. Click **Tools-Visual Differencing** to start the merging process.
3. When the **Give reference model** dialog opens, specify the original .mdl file.

   The Visual Differencing tool will load both models and generate a list of differences
4. In the Visual Differencing interface, click on the **+** next to the **Difference found** item to expand the tree one level.
5. Since no changes have been made to any information in the Use Case View, merge the information from the original model into the new model:
   a. Click **Use Case View** to select it
   b. Click **Merge** in the Use Case View pop-up
   c. Make sure the **Replace with reference** option is selected, and click **Merge**.
6. Repeat the same procedure for all other views that contain differences, except for the Logical view.
7. In the Logical view, you do not need to merge the entire view, only selected diagrams:
   a. Click **+** next to the 'Logical View' item to expand one level.
   b. Click **+** for all Logical View subtree members until the entire subtree is exposed.
   c. In each place a blue **+** exists in the Logical View subtree (all diagrams in the subtree):
      1) Click the item to select it.
      2) Click **Merge** in its pop-up menu.
      3) Make sure the **Replace with reference** option is selected, and click **Merge**.

   You have now completed merging information from your original model into the new model which contains the changes from Object Builder.
8. Click **File-Save** in the Visual Differencing tool and save the updated model to a new file.
9. Click **File-Exit** to close the Visual Differencing tool.
10. Click **File-Open** in Rose 98 and open the updated .mdl file.

Your model now contains the entire updated design, and you can continue your design work. When you are ready to switch back to Object Builder, you can export the design back to Object Builder by selecting **File** - **Export to Object Builder**.

RELATED CONCEPTS

"Object Builder" on page 1
"Projects and Models" on page 4
"Rose" on page 74
"The Rose Bridge" on page 76
"Mapping Rules: Object Builder to Rose" on page 87

RELATED TASKS
"Export a Design from Rose" on page 89
"Import Projects from a Team Environment" on page 212

Export from Rose - Scenario

Objectives
To create a class in Rose.
To specify class and attribute properties that will affect how the class is exported.
To export a sample application from Rose into an Object Builder project.

Before You Begin
You need Rational Rose 98 installed and set up to work with Object Builder, as described in the task "Set up Rose 98" on page 74.

You need Object Builder installed.

You should be familiar with Rational Rose 98. If you are not familiar with the tool, take the Rose tutorial included with the software.

You should be familiar with Object Builder. If you are not familiar with the tool, run through the "Getting Started with Object Builder" on page 39 scenarios.

Description
This exercise describes how to create a class in Rose, prepare it for export to Object Builder, complete the export, and open the exported project. A follow-on exercise, Import into Rose - Scenario (page 98), describes how to reverse the process, importing the project into Rose and updating the model with any changes that have been made to the project. A team development version, Team Development with Rose - Scenario (page 218), describes how to use a single Rose model with multiple Object Builder projects.

For this exercise, you will complete the following tasks:

1. Create a class in Rose.
2. Add Component Broker properties to the class and its attributes.
3. Export the model to an Object Builder project.
4. Open the project in Object Builder.

Create the Claim Class
Start Rose 98, and add a simple class with two attributes and two methods (Claim). This is the same as the class described in the Getting Started scenarios for Object Builder.

Add the Claim class:

1. Start Rose 98. It opens to a Class Diagram for Logical/Main.
2. From the pop-up menu of the class diagram, click **Class Wizard** to open the Class wizard.
3. Name the class Claim.
4. Click **Next** through the remaining wizard pages, then **Finish**.

   A class named Claim is added to the Logical View folder.

Add the attributes claimNo and state:

1. From the pop-up menu of Claim, click **New Attribute**. A placeholder attribute is added (named name, type of type, initial value of initval).
2. Type over each of the values for the new attribute, naming it claimNo, with type Integer and initial value of 0.
3. Click elsewhere in the diagram to apply the changes.
4. Add another attribute in the same way, and name it state, with type Integer and initial value of 0.

Add the operations approve() and deny():

1. From the pop-up menu of Claim, click **New Operation**. A placeholder operation is added (named opname, argument argname, return type return).
2. Type over each of the values for the new operation, naming it deny, with no arguments, and return value void.
3. Click elsewhere in the diagram to apply the changes.
4. Add another operation in the same way, and name it approve, with no arguments, and return value void.

You now have a class named Claim, with attributes claimNo and state, and methods approve() and deny().

Add Component Broker Properties to the Class
You can specify properties of the class and its attributes that will affect the way it is exported to Object Builder. Some of these properties are standard Rose properties that have meaning for the export process, others are specific to Component Broker, and were made available in Rose when you copied over customized .pty files during the Rose 98 setup.

Customize the way the class will be exported, to create the following component objects in Object Builder: business object interface, business object implementation, key, copy helper:

1. From the pop-up menu of the class in the class diagram, click **Open Specification** to open the Class Specification notebook.
2. Turn to the IDL page. The following properties map to component objects:
   - IDLSpecificationType
     By default, it is set to Interface. A business object interface is created for the class. Other values you can set for this property would make the class export as a construct (for example, a struct or enum).
   - CreateImplementation
     By default, it is set to False. A business object implementation and its accompanying data object interface are not created for the class.
   - CreateKey
     By default, it is set to False. A key is not created for the class.
   - CreateCopyHelper
     By default, it is set to False. A copy helper is not created for the class.
3. Click on CreateImplementation, then click the value False, and change it to True.
4. A business object implementation and its accompanying data object interface will be created for the class.
5. Set the values for CreateKey and CreateCopyHelper to True as well.

   A key and copy helper will now be created for the class.
6. Click **OK**.

Add Component Broker Properties to the Attributes

Customize the way the attributes will be exported, to make claimNo and state public attributes of the component, and make claimNo part of the key:

1. In the tree view, under the Logical View, select the attribute claimNo.
2. From its pop-up menu, click **Open Specification** to open its Class Attribute Specification notebook.
3. On the General page, set its Export Control to public.
4. Turn to the DDL page, and set the PrimaryKey property to True.
5. Turn to the IDL page, and set the isReadOnly property to True.
6. Click **OK**.

   The attribute claimNo will now be generated as a public read-only attribute that is part of the business object, key, and copy helper.
7. In the tree view, under the Logical View, select the attribute state.
8. From its pop-up menu, click **Open Specification** to open its Class Attribute Specification notebook.
9. On the General page, set its Export Control to public.

   The attribute state will now be generated as a public attribute that is part of the business object and copy helper.

You have added properties that specify how the class maps to component objects and attributes in Object Builder. You are ready to export to an Object Builder project.

Export to Object Builder

To export to Object Builder, follow these steps:

1. Save and name your model (for example, e:\scenarios\rosemodels\claim.mdl). You cannot export an unnamed model.
2. Click **File - Export to Object Builder**. The Rose Bridge wizard opens.
3. Specify a directory to export to (for example, e:\scenarios\roseclaim\). The Rose Bridge will create the directory if necessary, and turn it into an Object Builder project directory.
4. Accept the defaults for the other settings (you do not need to select what to export, and you are exporting to a single project, not a team environment).
5. Click **Finish**.

The Rose Bridge starts by saving your current Rose model. The Rose Bridge then exports an XML version of the model, consisting of two files: *project*\Import\udbo.xml, to become the Object Builder project, and *project*\XMI\xmi.xml, to hold any information that would otherwise be lost in the transfer. Finally, the Rose Bridge then imports udbo.xml into Object Builder to create the new Object Builder model files.

You can now open the project in Object Builder and review the results of the export.

Open the Object Builder Project

Open the Object Builder project and review the exported component:

1. Start Object Builder.
2. In the Open Project wizard, type the name of the directory you exported to (for example, e:\scenarios\roseclaim\).
3. Click **Finish**. The project opens.

4. In the Tasks and Objects pane, expand the User-Defined Business Objects folder. You can see the business object file Claim.
5. Expand the file to show the Claim interface, expand the interface to show ClaimKey, ClaimCopy, and ClaimBO, and expand ClaimBO to show ClaimDO. These objects were created according to the property settings of the Claim class in Rose, as follows:
   - Claim file and Claim interface
     Created because IDLSpecificationType was set to Interface.
   - ClaimKey
     Created because CreateKey was set to True.
   - ClaimCopy
     Created because CreateCopyHelper was set to True.
   - ClaimBO and ClaimDO
     Created because CreateImplementation was set to True.
6. Click on the Claim interface. You can see its attributes and methods in the Methods pane.
7. Click on the ClaimBO implementation. You can see the get and set methods for the attributes, and the method signatures, in the Methods pane.
8. Click on the ClaimKey key. You can see the get and set methods for claimNo, which you set to be part of the key with the PrimaryKey DDL property.
9. Click on the ClaimCopy copy helper. You can see the get and set methods for both attributes, which are part of the copy helper by default.

You can review the skeleton signatures for the methods, the default implementations for get and set methods, and the framework methods added by Object Builder, by clicking on the attribute or method in the Methods pane.

Summary
You have created a class in Rose named Claim, defined its attributes and operations, and exported the result as a set of component objects to Object Builder. You can now continue to the "Import into Rose - Scenario", in which you customize the class and then import the changes into Rose. If you want, you can skip that scenario and continue on to the "Team Development with Rose - Scenario" on page 218, in which you add a second class with an object relationship, and export the two classes into separate interdependent projects.

Import into Rose - Scenario

Objectives
To add attributes to a component in Object Builder.
To edit an existing attribute in Object Builder.
To apply the change to a Rose model.

Before You Begin
This scenario is a continuation of the "Export from Rose - Scenario" on page 95. You must complete the previous scenario before attempting this one.

You need Rational Rose 98 installed and set up to work with Object Builder, as described in the task "Set up Rose 98" on page 74.

You need Object Builder installed.

Description

In this exercise, you will extend the Claim component created in the previous exercise, by adding the attributes date and explanation, and change the name of the claimNo attribute to claimNumber. You will then apply the changes to the original Rose model, by importing your Object Builder project into Rose. Once you are done, you can continue on to the Team Development with Rose - Scenario (page 218), in which you add a second class with an object relationship, and export the two classes into separate interdependent projects.

For this exercise, you will complete the following tasks:

1. Open the project.
2. Edit the Claim component attributes.
3. Import the changes into Rose.

Open the Project

Open the project you created in the previous exercise:

1. Start Object Builder.
2. In the Open Project wizard, specify the project to open (for example, e:\scenarios\roseclaim\)
3. Click **Finish**.

Add and Edit Attributes

Add the two new attributes to the business object interface, and change the name of the existing key attribute:

1. Locate the Claim interface in the User-Defined Business Objects folder (defined under the Claim file).
2. From the Claim interface's pop-up menu, click **Properties** to open the Business Object Interface wizard.
3. Click the title and turn to the Attributes page.
4. Click **Add Another**.
5. Define an attribute named dateOpened, of type string, with size 10.
6. Click **Add Another**.
7. Define an attribute named explanation, of type string, with size 200.
8. Click on the claimNo attribute.
9. Type over its name, changing it to claimNumber.
10. Click **Refresh**.
11. Click **Finish**.

The new attributes are automatically added to the business object implementation, and the name change from claimNo to claimNumber is applied automatically to the key, copy helper, and implementation.

Add the two new attributes to the copy helper:

1. Locate the ClaimCopy copy helper, under the Claim interface.
2. From ClaimCopy's pop-up menu, click **Properties** to open the Copy Helper wizard.
3. Move the two new attributes from the Business Object Attributes list to the Copy Helper Attributes list.
4. Click **Finish**.

Save your changes and close Object Builder:

1. Click **File - Save**.
2. Click **File - Exit**.

You have made your changes to the project, saved them, and closed Object Builder. You are ready to import the project into Rose.

Import the Project into Rose

Import the changed Object Builder project to a new Rose model:

1. Start Rose.
2. Click **File - Open**.
3. Specify a new model file to create (for example e:\scenarios\rosemodels\importclaim.mdl).
4. Click **File - Import from Object Builder**. The Rose Bridge wizard opens.
5. In the Input Directory field, type the Object Builder project directory path (for example e:\scenarios\roseclaim\).
6. In the Output field, make sure the new Rose model is listed (for example e:\scenarios\rosemodels\importclaim.mdl).
7. Accept the defaults for the other options.
8. Click **Finish**.

The Rose Bridge generates the udbo.xml file in the project's \Export directory, updates the xmi.xml file in the project's \XMI directory with any project information it cannot preserve in the transfer, and then imports the two files into Rose to create a new model file.

Review the Changes

Under the Logical View, you can see that Claim now has two new attributes, dateOpened and explanation, and that the claimNo attribute has become claimNumber.

Open the Attribute Specification notebook for dateOpened. On the IDL page, you can see that the Length property has the value 10.

Save and close the model.

Because the model in the previous scenario contained information only in the Logical View, with no additional diagrams, the new model can simply replace the previous model. However, if your original model had contained information in other views, or additional diagrams within the Logical view, you could consolidate that information with the newly imported model by using the the Rose 98 Visual Differencing tool, as described in the topic "Import a Project into Rose" on page 92.

Summary

You have changed your component in Object Builder, and then applied the changes to the component design in Rose. You can now continue working in Rose as part of the Team Development with Rose - Scenario (page 218), in which you add a second class with an object relationship, and export the two classes into separate interdependent projects.

Chapter 5. Creating Components in Object Builder

Create a Component for Transient Data

If your component has data that does not need to be stored, or you are providing customized persistence rather than using Component Broker services, you can create a component for transient data.

You can create a component for transient data in much the same way you create a component for new DB data, starting from the business object file and working down to the data object implementation. Because the data is transient, you do not need a persistent object or schema.

A component is identified as containing transient data by the setting on its data object implementation. When you create the data object implementation, set its Persistent Behavior and Implementation (page 32) to **Transient**.

If you set the data object implementation's "Environment" on page 31 to **BOIM with UUID key**, you do not require a key for the component.

To create a component for transient data, complete these tasks:

1. "Create a Business Object File" on page 282
2. "Add a Business Object Module" on page 282
3. "Add a Business Object Interface" on page 283
4. "Add a Key" on page 292
5. "Add a Copy Helper" on page 294
6. "Add a Business Object Implementation and Data Object Interface" on page 284
7. "Add Code for User-Defined Methods" on page 267
8. "Add a Data Object Implementation" on page 299

RELATED CONCEPTS
"Components" on page 15

RELATED TASKS
"Create a Component for New DB Data"
"Create a Component for Existing DB Data" on page 104
"Create a Component for PA Data" on page 115

Create a Component for New DB Data

If you are creating a new component, which connects to a database that does not yet exist, you can create the entire component in Object Builder, starting with the business object interface and working your way down.

To create a new component directly in Object Builder, follow these steps:

1. "Create a Business Object File" on page 282
2. "Define Constructs with File Scope" on page 278

© Copyright IBM Corp. 1997, 1998

3. “Add a Business Object Module” on page 282
4. “Define Constructs with Module Scope” on page 279
5. “Add a Business Object Interface” on page 283
6. “Define Constructs With Interface Scope” on page 279
7. “Add a Key” on page 292
8. “Add a Copy Helper” on page 294
9. “Add a Business Object Implementation and Data Object Interface” on page 284
10. “Add Code for User-Defined Methods” on page 267
11. “Add a Data Object Implementation” on page 299
12. “Add a Persistent Object and Schema” on page 313
13. “Add a Managed Object” on page 340

RELATED CONCEPTS
“Components” on page 15

RELATED TASKS
Create a Component - Overview

Create a Component for New DB Data - Scenario

In this scenario you define a simple component with database persistence, starting from the component’s business object interface and working down to the component’s DB schema.

After you complete the scenario, you will have defined the Person component, including its business object, key and copy helper, data object, persistent object, and DB schema. Once you have defined the component, you can export the result in XML format, so you can easily retrieve and re-use the work in later scenarios that build on this one.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizards. If you are experiencing problems, click the Help button within a wizard, or go to the Help pulldown in Object Builder.

Create the Project
Create a sample project to hold your work.
1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory (for example, e:\scenarios\person).
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Create the Business Object Interface
Define the Person interface:
1. From the User-Defined Business Objects folder’s pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file PFile.

3. Click **Finish**. The file now appears under the folder.
4. From the file's pop-up menu, click **Add Module** to open the Business Object Module wizard.
5. Name the module PModule.
6. Click **Finish**. The module now appears under the file.
7. From the module's pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
8. Name the interface Person.
9. Click the title bar and turn to the Attributes page.
10. Add the following attributes:
    - readonly string ssNo (size 20)
    - readonly string name (size 100)
    - string street
    - string town
11. Click **Finish**. The interface now appears under the module.

Add the Key and Copy Helper
Add PersonKey:
1. From the interface's pop-up menu, click **Add Key** to open the Key wizard.
2. Select ssNo and name as the key attributes.
3. Click **Finish**. The key now appears under the interface.

Add PersonCopy:
1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Select all attributes to be part of the copy helper.
3. Click **Finish**. The copy helper now appears under the interface.

Add the Business Object Implementation and Data Object Interface
Add PersonBO and PersonDO:
1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Click the title bar and turn to the Key and Copy Helper page.
3. Select PersonKey and PersonCopy.
4. Click the title bar and turn to the Data Object Interface page.
5. Select all attributes as state data (to be preserved in the data object).
6. Click **Finish**. The business object implementation appears under the business object interface, and the data object interface appears under the implementation.

Add the Data Object Implementation
Add PersonDOImpl:
1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. Set the following patterns:
   - **Environment - BOIM with any key**
   - **Form of Persistent Behavior and Implementation - Embedded SQL**
   - **Data Access Pattern - Delegating**

3. Click the title bar and turn to the Key and Copy Helper page.
4. Select PersonKey and PersonCopy.
5. Click **Finish**. The data object implementation appears under the data object interface.

Add the Persistent Object and Schema
Add PersonPO and its associated schema:
1. From the data object implementation's pop-up menu, click **Add Persistent Object and Schema** to open the Add Persistent Object and Schema wizard.
2. Name the schema group and database.
3. Click **Next** and review the attribute mappings.
4. Click **Finish**.

The persistent object and schema appear under the data object implementation.

In the DBA-Defined Schemas folder, they appear under the schema group you named.

Add the Managed Object
Add PersonMO:
1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard.
2. Click **Finish**. The managed object now appears under the business object implementation.

Export as XML
You can now export the component in XML format, for re-use in other scenarios.

From the pop-up menu of the business object file (PFile), click **Export**. The file PFile.xml, which holds all the component objects defined under PFile in the User-Defined Business Objects folder, is placed in the \Working\Export directory.

This scenario does not cover build or application configuration. Refer to the other scenarios that use Person for these additional steps.

Create a Component for Existing DB Data

You can create a component for accessing existing or legacy database information by importing the database schema into Object Builder, and deriving a component from it, as follows:
1. "Create a DB Schema by Importing an SQL File" on page 321
2. "Edit a DB Schema Group" on page 319
3. "Edit a Generated SQL File" on page 331
4. "Add a Persistent Object from a DB Schema" on page 316
5. "Add a Data Object from a DB Persistent Object" on page 304
6. "Add a Business Object from a Data Object" on page 287
7. "Add Code for User-Defined Methods" on page 267
8. "Add a Key" on page 292
9. "Add a Copy Helper" on page 294

10. “Add a Managed Object” on page 340

RELATED CONCEPTS

“Components” on page 15
“Schema” on page 20
“DDL” on page 114

RELATED TASKS

“Create a Component for Transient Data” on page 101
“Create a Component for New DB Data” on page 101
“Create a Component for PA Data” on page 115

Mapping Helper

A mapping helper is a class that contains mapping methods that are responsible for type conversion between attributes of the two objects being mapped. Every mapping helper class contains at least two static methods that always return void. These methods must be declared public members of the class.

Type conversion is required for greater flexibility. For example, an attribute of type *string* may be required to map to an attribute of type VARCHAR, so that the length of the string is not a fixed, predetermined quantity; rather, it has the ability to take on different values, depending on the run-time allotment of the string’s contents.

Object Builder provides the default mapping helper file (DB2MappingHelper.hpp, which contains the mapping helper class and its methods) in the following cases:

- When a Stringified Object Reference (SOR) of the data object is mapped to a persistent object of type *char.* This happens if there exists an object reference between the selected object and another object.
- When a Stringified Object Reference (SOR) of the data object is mapped to a persistent object of type VARCHAR. This happens if there exists an object reference between the selected object and another one.
- When a data object attribute of type *string* is mapped to a persistent object attribute of type VARCHAR. (A data object attribute of type *string* is normally mapped to a persistent object attribute of C++ string type. For example, a string of length 20 is mapped to *char[21].*)
- When a data object attribute of type *wstring* is mapped to a persistent object attribute of type DB2VARGRAPHIC.
  390 When one of the constrain platforms is **390** (you select **Platform - Constrain - 390**), *wchar* and *wstring* are not available for selection as an attribute type for your object.
- When a data object attribute of type *ByteString* is mapped to a persistent object attribute of type DB2VARCHAR.
- When a data object attribute of type *ByteString* is mapped to a persistent object attribute of type *char[]* (length greater than 0).

Note: Whenever Object Builder provides the mapping helper, it is recommended that you use it rather than provide your own.

Restriction: Object Builder does not provide the default mapping between complex data types (*any, Object, wchar* and *wstring* and types defined as constructs, which include typedefs, structures, and unions) and DB2 database types. You must provide your own helper class for these mappings.

The mapping helper information can be viewed on the Attributes Mapping Page of the Data Object Implementation wizard. The .cpp file generated from the data object implementation contains the mapping helper (DB2MappingHelper.hpp) in its include section.

If you want to provide your own mapping helper, you must create (outside Object Builder) a .hpp file, which contains the mapping helper class. When you define the mapping helper, follow these rules:

- Define both the mapping methods: from the persistent object to the data object, and from the data object to the persistent object, in the mapping helper class.
- Declare both mapping methods as public members of the class.
- Define both methods as inline methods to avoid linker errors.
- Define both methods as static methods.
- Define the return type of both methods as void.
- Pass the input arguments for both methods by const reference.
- For the persistent object to data object mapping method, use the following signature:

  inline static void PO_to_DO_ mapping_method_name(att1, att2, ...attn, attribute_of_the_data_object)

  where att1, att2,... attn are the persistent object attributes that are mapped to the data object attribute, and require the mapping helper.
- For the data object to persistent object mapping method, use the following signature:

  inline static void DO_to_PO_ mapping_method_name(attribute_of_the_data_object, att1, att2,..., attn)

  where att1, att2,... attn are the persistent object attributes that are mapped to the data object attribute, and require the mapping helper.
- Ensure that the .cpp and the .hpp files have the same name as the mapping helper class name.

  Note: When you use a mapping helper when a foreign key is used for the mapping, the mapping methods must be defined from the key to the persistent object, and from the persistent object to the key.

Note the following points when you provide your own mapping helper file:

- Object Builder assumes that the name you provide as the class name is the same as the name of the mapping helper file (.hpp file) that you include in the file adornment's prolog. If the names are not the same, and you have all the mapping helper classes in a separate file, you must include this file in the prolog of the data object implementation's file adornment (click on the prolog or epilog object in the File Adornments folder, and type the #include statement at the beginning of the .cpp file in the editor pane), and regenerate the DOImpl_I.cpp file (**Generate - Selected - .cpp**, or **Generate - All** from the data object implementation).
- When you map a data object attribute to a persistent object attribute using a mapping helper you provide, you have to specify the name of the mapping helper file (without the extension) as the class name, and the names of the methods used for the mapping.

RELATED CONCEPTS

"Data Object" on page 18

“Persistent Object” on page 19
Data Object Customization for Cardinality Relationships (Additional Customizations) (*Programming Guide*)

RELATED TASKS

“Map Attributes Using a Mapping Helper” on page 260
“Map a Data Object to a DB Persistent Object” on page 251
Map a Data Object to a PA Persistent Object
“Add a Data Object Implementation” on page 299
“Add a Persistent Object and Schema” on page 313

Design Patterns and Iterators

Design Patterns

A design pattern describes a problem that occurs repeatedly in our environment. It then describes the core of the solution to the problem, in such a way that you can use this solution innumerable times without doing it the same way twice.

Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. One object’s pattern can be another one’s building block.

Design patterns are less specialized and smaller architectural elements than frameworks, but they are not frameworks, though they are more abstract than them.

Several design patterns can be contained in a framework, but the reverse is never true.

Design patterns must be implemented each time they are used - they are just code examples, whereas you can embody frameworks in code and use them directly.

Design patterns can be used in any kind of application; frameworks always have a particular application domain.

Examples of design patterns are object factories, iterators, mediators, proxies and bridges.

Iterators

A collection is a group of objects, and objects model real-world entities. So, very often, you need to access either references to objects, or the objects themselves (their references or indirection is hidden).

An iterator is a design pattern that defines three operations to traverse a collection (access objects directly or indirectly in that collection):
- *reset* points to the start of a collection
- *next* increments the iterator’s position
- *more* enables you to test if there are elements left in the iteration. This method returns *true* if there are more elements that you can access in the collection; it returns *false* if you have reached the end of the collection.

Iterators, like all design patterns must be implemented every time they are used.

A data object iterator supports data objects that are backed directly by DB2 queries.

Customize Referential Integrity

When every value of each foreign key of a database is valid, the database is in a state of referential integrity. A foreign key is a subset of columns in a table whose values match at least one primary key, or unique key value of a row of the parent table. For a database to be in a state of referential integrity, a referential constraint must be met. This referential constraint is that the values of the foreign key are valid only if one of the following statements is true:

- The values of the foreign key appear as values of a parent key (the key of the parent table).
- Some component of the foreign key is null.

Referential constraints are optional and can be defined in `CREATE TABLE` and `ALTER TABLE` statements. Referential constraints are enforced by the database manager during the execution of `INSERT`, `UPDATE`, `DELETE`, `ALTER TABLE ADD` *constraint*, and `SET CONSTRAINTS` statements. Corresponding to these statements, the data object has the set of special framework methods: insert(), update(), retrieve(), del(), and setConnection() that perform, respectively, the same tasks as the SQL statements:

- insert() is called when a table is created or altered.
- update() puts data back into the database when a table is altered.
- retrieve() gets data from the database.
- del() deletes a row in the table.
- setConnection() defines the database that is affected by the SQL statements in the insert(), update(), retrieve(), and del() methods. This method is implemented only if the data object implementation uses Embedded SQL.

You can customize referential integrity by specifying the processing order of methods so that they conform to constraints applied by the database.

Note: You can access the Methods Mapping Page to specify the processing order only if there is a persistent object associated with the data object. The insert(), update(), retrieve(), and del() methods are not implemented for a transient data object implementation.

To specify the order of the persistent object methods, follow these steps:

1. Select the data object implementation that corresponds to the persistent object whose methods you want to arrange in a specific processing order.
2. From the data object implementation's pop-up menu, select **Properties**. The Data Object Implementation wizard opens.
3. Click the arrow to the left of the page name, and select Methods Mapping Page from the list. The page opens.
4. The Special Framework Methods folder contains the framework methods you can customize: insert(), update(), retrieve(), del(), and setConnection().

   Note: The setConnection() method is available only if you specified the form of persistent behavior and implementation as **Embedded SQL** on the Name and Platform Page of the Data Object Implementation wizard.
5. Select the method you want customized, display its pop-up menu and select **Add Mapping**. The **Persistent Object Method** field appears with the del() method selected by default. The method name has the form: *POInstance_name.Method _name*.

6. Click the list button and select the persistent object methods in the order you want them executed for the selected framework method. For each of the methods insert(), update(), retrieve(), del(), and setConnection(), you can select the **Always complete calling sequence (ignore warnings)** check box if you want the next method to be implemented even if a warning message is issued.
7. Click **Finish.** The ordered list of methods is saved. You can view it later by opening the same wizard. You can also view the order you specified by examining the method body in the Source pane after selecting the special framework method in the Methods pane.

RELATED CONCEPTS
“Persistent Object” on page 19

RELATED TASKS
“Add a Data Object Implementation” on page 299
“Map a Data Object to a DB Persistent Object” on page 251

Data Encoding Schemes

Object Builder uses the following data encoding schemes for database data:

DBCS encoding scheme

Attribute Type (IDL Type)	Attribute is a Key	SQL Type	PO Type	Size	Dele- gation
wchar	Yes, No	GRAPHIC[1]	wchar_t[]		ESQL
wstring	Yes, No	VARGRAPHIC[n]	_DB2VARGRAPHIC		ESQL
wstring	No	LONG VARGRAPHIC	_DB2VARGRAPHIC		ESQL
string		VARCHAR	char[]		Caching
string	No	LONG VARCHAR	char[]	2000	Caching
wchar		GRAPHIC	wchar_t[]		Caching
wstring		VARGRAPHIC	wchar_t[]		Caching
wstring	No	LONG VARGRAPHIC	wchar_t[]	2000	Caching

Binary Data encoding scheme

Attribute Type (IDL Type)	Attribute is a Key	SQL Type	PO Type
ByteString	Yes, No	VARCHAR FOR BIT DATA	::ByteString or DB2VARCHAR
ByteString	Yes, No	VARCHAR	DB2VARCHAR
ByteString	No	LONG VARCHAR FOR BIT DATA	::ByteString or DB2VARCHAR
ByteString	No	LONG VARCHAR	DB2VARCHAR
ByteString	Yes, No	CHAR FOR BIT DATA	::ByteString

SBCS/MBCS encoding scheme

Attribute Type (IDL Type)	Attribute is a Key	SQL Type	PO Type	Delegation
any void Object string struct typedef union	No	LONG VARCHAR	DB2VARCHAR[2000]	Embedded SQL
		LONG VARCHAR	char[]	Caching Services

RELATED CONCEPTS

"Data Object" on page 18
"Persistent Object" on page 19
"DBCS and Binary Data Support" on page 5

RELATED TASKS

"Add a Persistent Object from a DB Schema" on page 316
"Add a Data Object from a DB Persistent Object" on page 304

RELATED REFERENCES

"DB2 Data Type Mappings"
"Oracle Data Type Mappings" on page 113

DB2 Data Type Mappings

The tables on this page show the mappings among IDL, PO and SQL data types in different situations, assuming a DB2 backend database.

The following mappings are used when you create a schema and persistent object from a data object implementation:

IDL Type	PO Type	SQL Type	Encoding scheme
boolean	short	SMALLINT	
char	char[]	CHARACTER	
string[n] {string length fixed, 0< n <255}	DB2VARCHAR	VARCHAR[n]	SBCS or MBCS
string[n] {varying length, n>255}	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
string (if it represents a decimal number)	char[]	DECIMAL	
double	double	DOUBLE	
	double	DECIMAL	
float	double	DOUBLE	
long	long	INTEGER	
unsigned long	long	INTEGER	
octet	short	INTEGER	

short	short	SMALLINT	
unsigned short	long	INTEGER	
date	char[]	DATE	
time			
any	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
void	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
Object	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
string	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
string (if it represents a decimal number)	char[]	DECIMAL	
wstring	DB2VARGRAPHIC	LONG VARGRAPHIC	DBCS
wstring[n] {fixed string length, 1<n<128}	DB2VARCHAR[2000]	VARGRAPHIC[n]	DBCS
wstring[n] {varying length, n>128}	DB2VARCHAR[2000]	LONG VARGRAPHIC	DBCS
wchar	wchar_t	GRAPHIC(1)	DBCS
struct	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
typedef	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
union	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
interface	DB2VARCHAR[2000]	VARCHAR[2000]	SBCS or MBCS
enum	long	INTEGER	
wstring	DB2VARGRAPHIC [2000]	GRAPHIC[n]	DBCS
string[n+1]	char[n+1]	CHAR[n]	SBCS or MBCS
IManagedClient ByteString	::ByteString	VARCHAR for bit data	Binary
IManagedClient ByteString	::ByteString	LONG VARCHAR for bit data	Binary
IManagedClient ByteString	::ByteString	VARGRAPHIC for bit data	Binary
IManagedClient ByteString	::ByteString	LONG VARGRAPHIC for bit data	Binary
All other types	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS

You can map each of the IDL types in the table below with each of the PO types listed, without using a mapping helper:

IDL Type	PO Type
char enum boolean double float long unsigned long short unsigned short octet	char long short float double

Note: You can also map an IDL type string to a PO type char without using a mapping helper.

Object Builder provides the mapping helper for the following IDL to PO type mappings:

IDL Type	PO Type	Name of Mapping Helper	DO to PO Mapping Method	PO to DO Mapping Method
string	_DB2VARCHAR[]	DB2MappingHelper	stringToVarChar	varCharToString
interface	_DB2VARCHAR[]	DB2MappingHelper	byteStringToVar Char	varCharToByte String
interface	char[]	DB2MappingHelper	byteStringToString	stringToByteString
wchar	wchar_t[]	DB2MappingHelper	wStringToVar Graphic()	varGraphicTo WString()

The following mappings are used when you create a persistent object from a schema:

SQL Type	Length	PO Type	Size	IDL Type	Size
CHARACTER	n	char[n]	n+1	string	n+1
CHARACTER[1]	1	char	1	char	1
INTEGER		long		integer	
SMALLINT		short		short	
DOUBLE		double		double	
DECIMAL NUMERIC		double		double	
	n	char[]	n+2	string	(n+2)*Scale
BLOB	n	char[]	n	string	n*Scale
CLOB	n	char[]	n	string	n*Scale
DBCLOB	n	char[]	n	string	n*Scale
GRAPHIC	1	wchar_t[]	1	wchar	1
GRAPHIC	n	wchar_t[]	n	wstring	n-1
DATE		char[]	11	date	
TIME		char[]	9	time	
TIMESTAMP		char[]	27	timestamp	
VARCHAR	n	DB2VARCHAR[]	n	string	n

VARGRAPHIC	n	DB2VARGRAPHIC[]	n	wstring	n
LONG VARCHAR		DB2VARCHAR[]	2000	string	2000
LONG VARGRAPHIC		DB2VARGRAPHIC[]	2000	wstring	2000

The following mappings are used when you create a data object from a persistent object:

PO Type	IDL Type	Size
char	char	
char[n]	string	n-1
wchar_t	wchar	1
wchar_t[n]	wstring	n-1
short	short	
long	long	
double	double	
float	float	
DB2VARCHAR[n]	string	n
DB2VARGRAPHIC[n]	wstring	n
All other types	string	256

RELATED CONCEPTS
"Persistent Object" on page 19

RELATED TASKS
"Add a Persistent Object and Schema" on page 313
"Add a Persistent Object from a DB Schema" on page 316
"Add a Data Object from a DB Persistent Object" on page 304

Oracle Data Type Mappings

Object Builder uses the Oracle Application Adaptor (OAA) to access data in Oracle databases on Windows NT platforms.

Restrictions:

- Only Oracle 8.0.4.0 databases are supported.
- Support for Oracle backend databases is limited to data objects that use the Oracle Caching services only. That is, data objects that use embedded SQL, or any other form of persistent behavior and implementation will not be able to access data stored in Oracle databases.
- Reference collections are not supported in conjunction with Oracle backends for Component Broker Release 1.3.
- For Oracle, only optimistic caching is supported.
- In the current release of Component Broker, only the Oracle VARCHAR2 and NUMBER data types are supported, along with those Oracle data types that have an equivalent type in DB2. That is, Object Builder accepts all SQL/DS and DB2 types and the Oracle NUMBER, NUMBER(p), NUMBER(p,s) and VARCHAR2 types. It will not accept any other Oracle types such as RAW(n), LONG RAW, NCHAR(n), NVARCHAR2, and ROWID.

- Object Builder will not accept the Oracle data type NUMBER with a negative scale.

The following table shows the mapping to the persistent object type, Interface Definition Language type, SQL type, and the equivalent DB2 type.

Oracle SQL type	precision (p)	scale (s)	PO type	IDL type	SQL type
NUMBER(p,s)	0	0	double	double	double
NUMBER(p,s)	1..4	0	short	short	smallint
NUMBER(p,s)	5..9	0	long	long	integer
NUMBER(p,s)	>=10	0	double / string	string	decimal(p,0)
NUMBER(p,s)***	p	<0	double / string	string	decimal(p,0)
NUMBER(p,s)	p	>38	double	double	double
NUMBER(p,s)	p	>p	double		double
NUMBER(p,s)*	p	s	double / string	string	decimal(p,s)
VARCHAR2(n)			string	string	varchar(n)
DATE**			string**	string	timestamp
RAW***			::ByteString	::ByteString	varchar for bit data****
LONG RAW***			::ByteString	::ByteString	varchar for bit data****

* Consider NUMBER(p) = NUMBER(p,0) and NUMBER = NUMBER(38,0).

** Length 27, not 11 as in DB2.

*** Not supported by the **Import SQL** action in Object Builder Release 1.3.

**** Both varchar for bit data and varchar(n) for bit data are valid. If it has a maximum length (n), you must provide it. If you do not specify n, Object Builder allocates a buffer of 32K.

RELATED CONCEPTS

"Persistent Object" on page 19
Application Adaptor *(Programming Guide)*

RELATED TASKS

"Add a Persistent Object and Schema" on page 313
"Add a Persistent Object from a DB Schema" on page 316
"Add a Data Object from a DB Persistent Object" on page 304

DDL

There are two types of DDLs (Data Description Languages): System Management DDL and SQL DDL.

System Management DDL: a scripting language that defines the structure of an application on both client and server. Object Builder can generate a DDL script for your application family that defines the structure of the applications in the family. This generated DDL file is found in your working directory, under a subdirectory that

has the same name as the application family. It is this file that provides the System Manager with information about the applications during the installation process.

SQL DDL: a language that describes data and their relationships in a database. It is composed of data definition statements that create, alter, or destroy database objects such as tables, aliases, views, and indexes.

A data definition is a program statement that describes the features of, specifies relationships of, and establishes the context of data. It has information that describes the contents and characteristics of a field, a record, or a file. A data definition can include field names, lengths, locations, and data types.

In Object Builder, you can import an SQL DDL file to create schemas within a schema group.

RELATED TASKS
"Generate the Install Image" on page 379
"Create a DB Schema by Importing an SQL File" on page 321

Create a Component for PA Data

You can create a component for accessing existing transactional information by importing the relevant PA bean into Object Builder, and deriving a component from it, as follows:

1. "Create a PA Schema by Importing a PA Bean" on page 337
2. "Add a Persistent Object from a PA Schema" on page 334
3. "Add a Data Object from a PA Persistent Object" on page 305
4. "Add a Business Object from a Data Object" on page 287
5. "Add Code for User-Defined Methods" on page 267
6. "Add a Key" on page 292
7. "Add a Copy Helper" on page 294
8. "Add a Managed Object" on page 340

You can then build DLLs and package the application.

RELATED CONCEPTS
"Components" on page 15
"Schema" on page 20
"Procedural Adaptor Bean (PA Bean)" on page 117
Session Service (*Advanced Programming Guide*)
Transaction Service (*Advanced Programming Guide*)
Connections to a Tier-3 System (*System Management*)

RELATED TASKS
Create a Component
Edit a Component
"Build DLLs - Overview" on page 363
"Package an Application" on page 375
Configure a new ECI Connection to a Tier-3 CICS Region (*System Management*)
Configure a new HOD Connection to a Tier-3 System (*System Management*)
Configure a new APPC Connection to a Tier-3 System (*System Management*)

Configure the iPAAServices Application onto an Application Server (*System Management*)
Configure an Application to use a Connection to a Tier-3 System (*System Management*)

Enterprise Access Builder (EAB)

Enterprise Access Builder (EAB) is a set of class frameworks and development tools in VisualAge for Java 2.0 that enable you to move your applications from a front-end transaction system such as Customer Information Control System (CICS), or Information Management System (IMS), to an object-oriented programming environment. Procedural Adaptor (PA) beans that are created using EAB can be imported into Object Builder as PA schemas and PA persistent objects.

Enterprise Access Builder (EAB) used to be referred to as CICON, which stood for Customer Information Control System (CICS) and Information Management System (IMS) Connection, in previous releases of Component Broker.

RELATED CONCEPTS
“Persistent Object” on page 19
“Schema” on page 20“Procedural Adaptor Bean (PA Bean)” on page 117

RELATED TASKS
“Create a PA Schema by Importing a PA Bean” on page 337
“Add a Persistent Object from a PA Schema” on page 334

Transaction Object

In Enterprise Access Builder (EAB), a transaction object is a container that encapsulates the sequence of screen panels you would navigate in order to complete a CICS or IMS transaction. All panel states, input fields, and output fields are modeled in the transaction object.

Note: An EAB transaction object has no connection with CORBA transactions, neither with any of the classes defined in cosTransactions.

RELATED CONCEPTS
“Enterprise Access Builder (EAB)”
“Procedural Adaptor Bean (PA Bean)” on page 117
“Transaction Record”

Transaction Record

In Enterprise Access Builder (EAB), a transaction record is an element of the transaction object. One transaction record models a single panel state in a CICS or IMS transaction, including all input and output fields on that panel.

RELATED CONCEPTS
“Enterprise Access Builder (EAB)”
“Procedural Adaptor Bean (PA Bean)” on page 117
“Transaction Object”

Procedural Adaptor Bean (PA Bean)

A PA bean is a bean in VisualAge for Java that inherits from the CBProceduralAdapterObject class. PA beans, built using Enterprise Access Builder (EAB), wrap existing transactions for reuse in Component Broker.

PA beans are imported into Object Builder as PA schemas. By default, a PA persistent object is generated for each bean that you import, but you can create one yourself, for the PA schema. The PA persistent object uses the definition of the PA schema to make calls to the PA bean.

RELATED CONCEPTS
"Persistent Object" on page 19
"Schema" on page 20"Enterprise Access Builder (EAB)" on page 116
RELATED TASKS
"Create a PA Schema by Importing a PA Bean" on page 337
"Add a Persistent Object from a PA Schema" on page 334

Add endResource() to a Sessional Business Object

When a business object uses Session Service, you can provide your own code to be called during some of the normal processing for those services. You can do this by calling the endResource() method that you define on the business object, in both C++ and Java implementations.

390 You cannot call endResource() when the target platform is OS/390.

Follow these steps:

1. From the pop-up menu of the business object implementation, select **Properties.** The Business Object Implementation wizard opens to the Name and Data Access Pattern Page.
2. Under the **Session Services** section, select the **Provides end resource** check box.
   By selecting it, you indicate that you want the endResource() method on the business object.

   When you select the business object implementation in the Tasks and Objects pane, you see the endResource() method that was created for the implementation by Object Builder, in the Framework Methods folder. It has an empty method body.
3. Select the method from the Framework Methods folder, and from its pop-up menu, select **Properties**.
4. On the Implementation Page of the Method Implementation wizard, select the **Use the implementation defined in the editor pane** radio button, and click **Finish.**
   The endResource() method is now editable in the Source pane, when you select the method in the Methods pane.
   Note: You can also select the **Use an external file** option, if you have the code stored in either an external template file, or a normal file.
5. Provide your own code for the method body in the Source pane.
   The business object implementation's endResource() method that contains your code is called by the framework when the endResource() method is called on the managed object's mixin.

6. If you have not yet added a managed object for your component, add one now: From the pop-up menu of the business object implementation, select **Add Managed Object**. Select **Session Service** as the service to be used by the business object, and specify parents, if any, for the implementation.
7. Generate code for the managed object: From the pop-up menu of the object, select **Generate - Selected - All Files**, or **Generate - Selected - .cpp**
   The .cpp file that is generated contains the endResource() method that contains your code. If you want to write a separate method to contain your code, you must call this method from endResource().

RELATED CONCEPTS
"Business Object" on page 17
Session Service (*Advanced Programming Guide*)

RELATED TASKS
"Generate Code" on page 363

Create a Component for PA Data - Scenario

This scenario assumes that you have successfully installed and configured Component Broker. You will create a component with procedural adaptor persistence.

Restriction: Only beans created using VisualAge for Java Release 2.0 are compatible with this release (2.0) of Component Broker.

Enable the IBM Component Broker CICS and IMS application adaptor functionality
For the bean to be found during import, ensure that the JAR file (beans.jar), which contains the bean class you are to import, is in your system CLASSPATH variable.

Restriction: Assuming you installed Component Broker in a directory such as x:\Cbroker, you cannot have your CLASSPATH variable contents longer than 1780 characters. If the installation directory path is longer, you must have a correspondingly shorter CLASSPATH value. You get a run-time error if you exceed this limit. This is because commands (such as ob.bat), which invoke the Object Builder functions prepend the Object Builder .jar files to the class path, and then invoke the java code to run Object Builder.

Create a New Project
1. Start Object Builder.
2. The Open Project wizard opens to the Project Directory Page. Type a name and path for the project directory (for example, e:\scenarios\ABeCashAcct).
3. Click **Finish**.
   Note: If the project directory has never been used before, and contains no models, Object Builder confirms with you if you want to create a model in the directory. It then prompts you for a new model name. It shows you a default model name, which it assumes is the same as the directory name for the project. You can either accept that name, or change it. Click **OK**.
4. Click **Yes**, to create a new project.

Import the PA Bean
The class name for this bean is paa.samples.cics.appc.acct.ABeCashAcctPAO.

1. In the Tasks and Objects pane, select the User-Defined PA Schemas folder, and from its pop-up menu, select **Import - Bean**. The Import Procedural Adaptor Bean wizard opens to the Bean Selection Page.
2. You can choose to either type the name of the bean class, or select the JAR file containing the file, and then select the bean class. Select the **Enter bean name** radio button, and type the name of the class (paa.samples.cics.appc.acct.ABeCashAcctPAO) in the field.
3. Click **Next**. The Names and Connectors Page opens. Type the name of the module and the persistent object to be associated with the PA schema. You can also select the connector type to be used to access objects. Select ECI as the connector type. This is the type of connector ABeCashAcctPAO uses.
   390 When you choose OS/390 as the development (target) platform (**Platform - Constrain - 390**), only the EXCI, OTMA and Generic connector types are available for selection.

   Note: When you select either NT and 390, or AIX and 390 as the development platforms, all the connector types are available for selection. However, in this scenario you must not select 390 either alone, or in combination with one of NT or AIX, as the sample bean is for an ECI connector, and is not valid on OS/390: if you select 390, you will not be able to select ABeCashAcctPAOPO as the type of your persistent object.
4. Click **Next**. Select `res_type` and `account_ID` (two of the properties of the bean) from the **Properties** box, and move them to the **Key Attributes** box, by clicking the **>>** button.
5. Click **Next**. The Attribute Type Specification Page opens. Accept the defaults.
6. Click **Finish**, and the bean will be imported into Object Builder. The ABeCashAcctPAO schema and its associated persistent object ABeCashAcctPAOPO appear in the tree under User-Defined PA Schemas folder.

Connecting the Imported Bean with an Application

We can connect the imported bean with either existing applications, or those created after the bean is imported. We will create a new application.

Creating the application objects (business object, data object, managed object)

Create the CashAcct business object file

1. From the pop-up menu of the User-Defined Business Objects folder, select **Add File.**
2. The Business Object File wizard opens to the Name Page.
3. Type CashAcct in the **Name** field, and click **Finish**.
4. The CashAcct file appears in the User-Defined Business Objects folder.

Create the CashAcct interface

1. From the pop-up menu of CashAcct, select **Add Interface.**
2. The Business Object Interface wizard opens to the Name Page.
3. Type CashAcct as the name of the interface in the **Name** field.
4. Click the arrow to the left of the page name, and select Attributes from the list. The page opens.
5. From the pop-up menu of the Attributes folder, select **Add**.
6. In the **Attribute Name** field, type `res_type` as the name of an attribute.
7. For the data type of the attribute, select *string* from the **Type** field.

8. Type 0 in the **Size** field.
9. Use the **Add Another** button to add the next attribute.
10. Add the attribute `balance`, of type *long* and the *string* attributes `account_ID`, `acct_type`, and `utilities`, using steps 6 - 9.
11. Click **Refresh** instead of **Add Another**, after you add the last attribute.
12. Click **Next**. The Methods Page opens.
13. Click **Finish**. The CashAcct interface appears under the CashAcct file, in the folder.

Add the key

1. From the pop-up menu of the CashAcct interface, select **Add Key**.
2. The Key wizard opens to the Name and Key Attributes Page. From the **Business Object Attributes** box, select res_type and account_ID, and click the **>>** button to move them to the **Key Attributes** box.
3. Click **Finish**. The key CashAcctKey appears beneath the CashAcct interface.

Add the copy helper

1. From the pop-up menu of the CashAcct interface, select **Add Copy Helper**.
2. The Copy Helper wizard opens to the Name and Attributes Page.
3. Click the **All>>** button to select all the business object interface attributes as attributes of the copy helper.
4. Click **Finish**. The copy helper, CashAcctCopy appears under the CashAcct interface.

Add the business object implementation and the data object interface

1. From the pop-up menu of the CashAcct interface, select **Add Implementation**.
2. The Business Object Implementation wizard opens.
3. Select **Delegating** as the **Pattern for Handling State Data.**
4. From the **Data Object Interface** section, make sure that **Create a new one now** is selected.
5. Click the arrow to the left of the page name, and select Key and Copy Helper from the list. The page opens. Make sure that CashAcctKey is selected as the key, and CashAcctCopy is selected as the copy helper.
6. Turn to the Data Object Interface Page.
7. Click the **All>>** button to select all the attributes of the business object as state data for the data object.
8. Click **Finish.** The business object implementation CashAcctBO appears under the CashAcct interface, and the data object interface CashAcctDO appears as a node beneath the implementation.

Add the data object implementation and connect the BeCashAcctPAO persistent object

1. From the pop-up menu of the CashAcctDO interface, select **Add Implementation**.
2. The Data Object Implementation wizard opens to the Name and Platform Page.
3. Accept the default names, and select NT and AIX as the deployment platforms.
4. Click **Next**. The Behavior Page opens.
5. From the **Environment** section, select **BOIM with any key**.

6. From the **Form of Persistent Behavior and Implementation** section, select **Procedural Adaptors**.
7. Click **Next**. The Implementation Inheritance Page opens.
8. Verify that the class IPAAExtLocalToServer appears under the Parents folder.
9. Click the arrow to the left of the page name, and select Associated Persistent Objects from the list. The page opens. From the pop-up menu of the Persistent Object Instances folder, select **Add**.
10. Type iABeCashAcctPAOPO in th**e Instance Name** field.
11. Click **Next**. The Attributes Mapping Page opens.
12. Select the attribute *res_type* of the data object from the Attributes folder, and from its pop-up menu, select **Primitive.**
13. Click the list button, and select the attribute iABeCashAcctPAOPO.phone of the persistent object from the **Persistent Object Attribute** field. You have just defined a one-to-one mapping between the data object and the persistent object.
14. Repeat steps 12 and 13 for all the other attributes in the folder, mapping them one-to-one.
15. Click **Next.** The Methods Mapping Page opens.
16. Select the insert() special framework method from the folder, and from its pop-up menu, select **Add Mapping**.
17. Click the list box, and select iABeCashAcctPAOPO.insert() from the **Persistent Object Method** field.
18. Repeat steps 16 and 17 for all the methods update(), retrieve(), del(), and setConnection(), using a one-to-one mapping.
19. Click **Finish**.

The data object implementation, CashAcctDOImpl will now appear under the CashAcctDO interface, and the ABeCashAcctPAOPO persistent object will appear under the CashAcctDOImpl data object implementation.

Add the managed object

1. From the pop-up menu of the CashAcctBO business object implementation, select **Add Managed Object.**
2. Under **Service to Use**, select **Session Service** if the development platform is either Windows NT, or AIX. If the platform is OS/390, the **Session Service** button is disabled, and **Transaction Service** is automatically selected.
3. Click **Finish**.

The managed object appears under the business object implementation.

Export as XML

If you want to reuse the component that you just created in other scenarios, you can export it in XML format:

From the pop-up menu of the business object file (CashAcct), select **Export**. The file Acct.xml is created and placed in the \Working\Export directory. This file contains all the component objects defined under the file CashAcct in the User-Defined Business Objects folder.

Generate the application code

From the pop-up menu of the CashAcct file in the User-Defined Business Objects

folder, select **Generate - All**. Code generation will begin, and you can monitor the progress in the bottom left corner of Object Builder's window.

Configure the Build
Add the client DLL

1. From the pop-up menu of the Build Configuration folder, select **Add Client DLL.** The Client DLL wizard opens.
2. Type CashAcctC in the **Name** field.
3. Click **Next.**
4. Click the **All >>** button to select all the client source files.
5. Click **Finish.**

The CashAcctC DLL will appear in the Build Configuration folder.

Add the server DLL

1. From the pop-up menu of the Build Configuration folder, select **Add Server DLL.** The Server DLL wizard opens.
2. Type CashAcctS in the **Name** field.
3. Click **Next.**
4. Click the **All >>** button to select all the server source files.
5. Click **Next.**
6. Click the **>>** button to add the CashAcctC dll to the list of Libraries to link with.
7. Click **Finish.**

The CashAcctS DLL will appear in the Build Configuration folder.

Build the DLLs
Generate the configuration

From the pop-up menu of the Build Configuration folder, select **Generate - All**. Code generation will begin.

Create a Container Instance

1. From the pop-up menu of the Container Definition folder, select **Add Container Instance.** The Container wizard opens.
2. Type CashAcctContainer in the **Name** field.
   390: If you are developing an application intended for deployment on OS/390 (the **Platform - Constrain - 390** menu choice is checked), you are now done. The rest of the container definition is handled through the System Management user interface.
3. Click the arrow to the left of the page name, and select Service from the list. The Service page opens. Select **Use PAA Session Service**.
4. On the Service Details Page, specify a name of your choice for the connection. Select **ECI** for the connector type used by the session.
5. Click **Finish.**

The CashAcctContainer will appear in the Container Definition folder.

Configure the Application
Add an application family

1. From the pop-up menu of the Application Configuration folder, select **Add Application Family.** The Application Family wizard opens.
2. Type CashAcctApp in the **Name** field.
3. Click **Finish.**

The CashAcctApp family will appear in the Application Configuration folder.

Add an application

1. From the pop-up menu of the CashAcctApp application family, select **Add Application.** The Application wizard opens.
2. Type CashAcct in the **Name** field.
3. Click **Finish.**

The CashAcct application will appear under the AcctApp family.

Add the application's managed object

1. From the pop-up menu of the CashAcct application, select **Add Managed Object.** The Managed Object Configuration wizard opens.
2. Click the list box of the **Managed Object** field, and select CashAcctMO CashAcctMO from the list.
3. Click **Next.**
4. From the pop-up menu of the Implementations folder, select **Add.**
5. Click the list box of the **Data Object Implementation** field, and select CashAcctDOImpl from the list.
6. Click **Next.**
7. Click the list box of the **Name** field, and select CashAcctContainer from the list.
8. Click **Finish.**

The CashAcctMO managed object will appear under the Acct application.

Generate the application family

From the pop-up menu of the CashAcctApp family, select **Generate.**

Build the CashAcct Application (Client and Server)
Set up the environment

You had added the location of the .jar file that contains the bean you import to your system class path variable. So, reboot your system for the new environment variables to take effect. The server will then be able to find the bean.

Start the Build

1. Go to the NT directory within the Object Builder Working directory. This should be located in e:\scenarios\ABeCashAcct, under the Object Builder source directory.
2. Type nmake -f all.mak
3. The CashAcct application should be built.
4. Copy CashAcctS.dll and CashAcctC.dll to the CBroker\bin directory to place them in your system path.

Start the System Management Tool
Select **Start - Programs/IBM Component Broker/System Manager User Interface**

Install the CashAcct Server.

1. Click on the tool bar button to set the user level to Super User.
2. From the pop-up menu of **Host Images/<your host name>**, select**Load DDL File.**
3. Type e:\scenarios\ABeCashAcct\Working\NT\CashAcctApp\CashAcctApp.ddl.
4. From the pop-up menu of **Management Zones/Sample Cell and Work Group Zone/Configurations/Sample Configuration/Server Groups**, select **New**, and then **Server Group**.
5. Type CashAcctServerGroup, and click **OK.**
6. From the pop-up menu of **CashAcctServerGroup** (under Server Groups), select **New**, and then **Server**.
7. Type CashAcctServer, and click **OK**.
8. From the pop-up menu of **Host Images/<your host name>/Application Family Installs/CashAcctApp/ApplicationInstalls/CashAcct**, select **Drag**.
9. From the pop-up menu of **Management Zones/Sample Cell and Work Group Zone/Configurations/Sample Configuration**, select **Add Application**.
10. From the pop-up menu of **Management Zones/Sample Cell and Work Group Zone/Configurations/Sample Configuration/Applications/CashAcct**, select **Drag**.
11. From the pop-up menu of **Management Zones/Sample Cell and Work Group Zone/Configurations/Sample Configuration/Server Groups/CashAcctServerGroup**, select **Configure Application**.
12. Drag iPAAServices and configure it as well.
13. From the pop-up menu of **Management Zones/Sample Cell and Work Group Zone/Configurations/Sample Configuration/Server Groups/CashAcctServerGroup/Servers/CashAcctServer**, select **Drag**.
14. From the pop-up menu of **Hosts/<your host name>**, select **Configure Server**.
15. From the pop-up menu of **Management Zones/Sample Cell and Work Group Zone/Configurations/Sample Configuration**, select **Activate**.
16. The above step will take some time to complete. Once it has completed, from the pop-up menu of **Host Images/<your host name>/Server Images/CashAcctServer**, select **Run Immediate**.

Build and Run the Test Application

1. Copy CashAcctCli.cpp and its associated makefile, CashAcctCli.mak from e:\CBroker\samples\InstallVerification\PAA\Application\CashAcctCli into the e:\scenarios\ABeCashAcct\Working\NT directory under the current Object Builder source directory, and go to that directory.
2. Type set APP=CashAcct;
3. Type nmake - f CashAcctCli.mak to build the application.
4. When the build has finished, type CashAcctCli to run the application.

Build the CashAcct Application (Client and Server)
Set up the environment

You had added the .jar file containing your bean (ABeCashAcctPAO) to your CLASSPATH variable. That location is required for import, and for the server to find the bean. So, reboot your system for the new environment variables to take effect.

Start the Build

1. Go to the NT directory within the Object Builder Working directory. This should be located in e:\scenarios\ABeCashAcct, under the Object Builder source directory.
2. Type nmake -f all.mak
3. The CashAcct application should be built.
4. Copy CashAcctS.dll and CashAcctC.dll to the CBroker\bin directory to place them in your system path.

Start the System Management Tool
Select **Start - Programs/IBM Component Broker/System Manager User Interface**

Install the CashAcct Server.

1. Click on the tool bar button to set the user level to Super User.
2. From the pop-up menu of **Host Images/<your host name>**, select**Load DDL File.**
3. Type e:\scenarios\ABeCashAcct\Working\NT\CashAcctApp\CashAcctApp.ddl.
4. From the pop-up menu of **Management Zones/Sample Cell and Work Group Zone/Configurations/Sample Configuration/Server Groups**, select **New**, and then **Server Group**.
5. Type CashAcctServerGroup, and click **OK.**
6. From the pop-up menu of **CashAcctServerGroup** (under Server Groups), select **New**, and then **Server**.
7. Type CashAcctServer, and click **OK**.
8. From the pop-up menu of **Host Images/<your host name>/Application Family Installs/CashAcctApp/ApplicationInstalls/CashAcct**, select **Drag**.
9. From the pop-up menu of **Management Zones/Sample Cell and Work Group Zone/Configurations/Sample Configuration**, select **Add Application**.
10. From the pop-up menu of **Management Zones/Sample Cell and Work Group Zone/Configurations/Sample Configuration/Applications/CashAcct**, select **Drag**.
11. From the pop-up menu of **Management Zones/Sample Cell and Work Group Zone/Configurations/Sample Configuration/Server Groups/CashAcctServerGroup**, select **Configure Application**.
12. Drag iPAAServices and configure it as well.
13. From the pop-up menu of **Management Zones/Sample Cell and Work Group Zone/Configurations/Sample Configuration/Server Groups/CashAcctServerGroup/Servers/AcctServer**, select **Drag**.
14. From the pop-up menu of **Hosts/<your host name>**, select **Configure Server**.
15. From the pop-up menu of **Management Zones/Sample Cell and Work Group Zone/Configurations/Sample Configuration**, select **Activate**.
16. The above step will take some time to complete. Once it has completed, from the pop-up menu of **Host Images/<your host name>/Server Images/CashAcctServer**, select **Run Immediate**.

Build and Run the Test Application

1. Copy CashAcctCli.cpp and its associated makefile, CashAcctCli.mak from e:\CBroker\samples\InstallVerification\PAA\Application\CashAcctCli into the e:\scenarios\ABeCashAcct\Working\NT directory under the current Object Builder source directory, and go to that directory.
2. Type set APP=CashAcct;
3. Type nmake - f CashAcctCli.mak to build the application.
4. When the build has finished, type CashAcctCli to run the application.

RELATED CONCEPTS

"Business Object" on page 17
Session Service (*Advanced Programming Guide*)

RELATED TASKS

"Generate Code" on page 363

Unit Test for Procedural Adaptors - Scenario

The stand-alone session support is used to provide a similar test environment to that provided by the Component Broker run time.

When testing a PA bean outside of Component Broker, a stand-alone session service is provided as part of the stand-alone Tier-3 communications (t3-comm) classes (that is, as part of the com.ibm.ivj.communications package), and is therefore available when you use those classes.

To use this stand-alone session service, the unit test case of the PA bean needs to perform the following actions:

- invoke the static method com.ibm.ivj.communications.Session.startSession() before the PA bean is constructed. This means that this method must be called before the construction of communication objects (that is before the setConnection method is called on the transaction object (step 3 below)), but it can even be called before the PA bean is even created (step 1 below).
- invoke the static method `com.ibm.ivj.communications.Session.endSession(tf)` when the PA bean is no longer needed (that is, just before the end of the unit test program, or as appropriate if the unit test needs to perform more comprehensive testing with sessions).
  (The parameter *tf* is a boolean parameter. If its value is set to *true*, it indicates that the session is to be checkpointed (which means that all changes are committed and are to be kept); if it is set to *false*, it means that the session is to be reset.)

The unit test scenario requires the following steps to be done in VisualAge for Java:

1. In the constructor of the PA bean, create the desired connectionSpec (for example, HODConnectionSpec) and set appropriate values for the host name and port.
2. Set the BplConnectionSpec attribute to the newly created connectionSpec. This can be done since the the PA bean extends CBProceduralAdapterObject. (In the CBProceduralAdapterObject class, there is a protected com.ibm.bpl.cicon.connection.BplConnectionSpec attribute.) This is done in the PA bean constructor also.

3. Call the setConnection method on the transaction object immediately after transaction objects are created in the CRUD methods of the PA bean.
   Note:
   - The connectionSpec passed into the TO.setConnection(connSpec) method is the one set in the protected BplConnectionSpec attribute in CBProceduralAdapterObject class by the PA bean constructor.
   - The connectionSpec set in the CRUD methods will take precedence over any previous connectionSpec that may have been set.
4. Once the connectionSpec is set, you can make any calls on the transaction object, as desired.

This unit test scaffolding can be kept in place even when the PA bean is deployed in a Component Broker scenario because connectionSpec passed in from CB will be set in the protected BplConnectionSpec attribute of CBProceduralAdapterObject and take precedence over any previously set connectionSpec.

Note: The type com.ibm.ivj.communications.ConnectionSpec used in Component Broker scenarios inherits from com.ibm.bpl.cicon.connection.BplConnectionSpec.

RELATED CONCEPTS

"Enterprise Access Builder (EAB)" on page 116
"Procedural Adaptor Bean (PA Bean)" on page 117
"Transaction Record" on page 116
"Transaction Object" on page 116
Connections to a Tier-3 System (*System Management*)

RELATED TASKS

"Create a Component for PA Data" on page 115
"Create a Component for PA Data - Scenario" on page 118 "Create a Component for PA Data" on page 115
"Work with Container Instances - Overview" on page 345
Configure an Application to use a Connection to a Tier-3 System (*System Management*)

Chapter 6. Components Working Together

Create a Relationship

The following tasks cover the different types of relationship you can define between components, in Object Builder:

- “Define a One-to-Many Relationship” on page 131
- “Define a One-to-One Relationship” on page 130
- “Define a Circular Relationship” on page 132
- “Define a Foreign Key Pattern” on page 133
- “Store an Object Reference” on page 135

RELATED CONCEPTS

Object Relationships (*Programming Guide*)
“Foreign Key Patterns” on page 132
Model Details (Object Identity) (*Programming Guide*)
Data Object Customization for Cardinality Relations (*Programming Guide*)
Expanding the Client Programming Interface (Using Handles) (*Programming Guide*)
“Inheritance” on page 137

RELATED TASKS

“Create a Child Component” on page 136

Dependencies within an IDL File

When you add modules, interfaces, or constructs to an IDL file, they are automatically re-ordered if necessary to resolve any internal dependencies.

You can view and change this order by displaying the wizard for an existing IDL file (select a business object file or data object file in the Tasks and Objects pane, and select **Properties** from its pop-up menu). The order appears on the Contents Ordering page.

When a construct or interface references another construct or interface that comes after it in the file, the dependency is resolved in one of two ways:

- If the dependency is within the same scope (the referencing and referenced element are both at the file level, or both in the same module), then a forward declaration is automatically included to resolve the reference.
- If the dependency is cross-scope (the referencing and referenced element are at different scopes), then the order must be changed; a forward declaration in IDL cannot be cross-scope.

You can view the type and scope dependencies for an IDL element by selecting it on the Contents Ordering page:

- An interface dependency is listed when the interface has an attribute, method return type, method parameter type, method exception type, object relationship type, construct type, or construct member type that references another interface or construct in the same file. If the referenced interface or construct is in another module, then the dependency is listed as being on the module.
- A construct dependency is listed when the construct is of a type, or contains a member of a type, that references another interface or construct in the same

© Copyright IBM Corp. 1997, 1998

file.If the referenced interface or construct is in another module, then the dependency is listed as being on the module.
- A module dependency is listed when it contains an interface or construct that has a dependency.

The order of the contents is automatically checked for validity, and re-ordered if necessary, whenever you click **Finish** in the wizard for a module or interface contained in the file.

Note: You cannot have circular dependencies between constructs.

RELATED CONCEPTS
Interface Definition Language (*Programming Guide*)

RELATED TASKS
"Work with Constructs" on page 277

Define a One-to-One Relationship

When you add an attribute whose type is another business object, you create a cardinality-to-1 relationship between the first object (which has the attribute) and the second object (which is the type of the attribute).

To create an attribute that references an object, follow these steps:
1. Open the Business Object Interface wizard (either by adding a new business object interface to a file or module, or by selecting **Properties** from the pop-up menu of an existing business object interface).
2. Click the title bar and turn to the Attributes Page.
3. From the Attributes pop-up menu, select **Add**.
4. Type the name of the attribute (for example, currentClaim).
5. From the **Type** drop-down list, select the type for the object that you want to reference (for example, Claim).
6. Enter any other information that defines the attribute.
7. Complete the remaining wizard pages, or click **Finish**.

The object you reference should already exist in Object Builder, at least in skeleton form. If you have two objects that reference each other, create the references as follows:
1. Define the first interface (for example, Policy) in skeleton form (without methods or attributes).
2. Define the second interface (for example, Claim) with a reference to the first interface.
3. Go back and edit the first interface, to add a reference to the second interface.

RELATED CONCEPTS
Expanding the Client Programming Interface (Using Handles) (*Programming Guide*)
"Business Object" on page 17
Object Relationships (*Programming Guide*)

RELATED TASKS
"Define a One-to-Many Relationship" on page 131
"Add a Business Object Interface" on page 283

Define a One-to-Many Relationship

You can define a one-to-many (1 to n) relationship between business objects. When you define a relationship, a set of patterned methods are added to the first object to support the relationship, to allow adding, deleting, and listing of its related objects.

To create a relationship between objects, follow these steps:

1. Open the Business Object Interface wizard (either by adding a new business object interface to a file or module, or by selecting **Properties** from the pop-up menu of an existing business object interface).
2. Click the title bar and turn to the Object Relationships Page.
3. From the Relationships pop-up menu, select **Add**.
4. Type a name for the relationship.
5. Select the type for the objects that you want to define a relationship with.
6. Complete the remaining wizard pages (if this is a new interface), or click **Finish**.

The business object interface will now have methods for adding, deleting, or listing objects of the selected type. For example, if you established a 1-to-n relationship from Policy to Claim, Policy would now have the methods addClaim, deleteClaim, and listClaims. These methods allow a client to add and delete Claim instances through the Policy class, and to iterate through a list of the Claim instances to which Policy is related.

Now set the implementation of the relationship in the business object implementation:

1. Open the Business Object Implementation wizard (either by adding a new business object implementation to the previous interface, or by selecting **Properties** from the pop-up menu of its existing business object implementation).
2. Click the title bar and turn to the Object Relationships Page. This page lists the relationships defined in the business object interface.
3. Click on the relationship you want to implement.
4. Under **Reference Collection Implementation**, set the type of implementation:
   - **Local persistent reference**
     The relationship will be stored in a collection accessed through a local attribute. You should add the attribute to the data object for the component. The attribute has the same name as the relationship. You can add the attribute to the data object when you define the data object interface from the business object, or when you map the business object to an existing data object.
   - **User-Defined OO-SQL Reference**
     The relationship will be implemented using logic you provide. Only skeleton methods will be generated.
   - **Reference resolved using foreign key**
     The relationship is implemented using the foreign key pattern. This is described in a separate task.
5. Click **Finish**.

RELATED CONCEPTS

"Business Object" on page 17
Object Relationships (*Programming Guide*)
"Foreign Key Patterns" on page 132

RELATED TASKS
"Add a Business Object Interface" on page 283
"Define a One-to-One Relationship" on page 130
"Define a Foreign Key Pattern" on page 133

Define a Circular Relationship

When two components reference each other (through attributes or one-to-many relationships), the relationship is bidirectional, or circular.

Circular relationships cannot cross module boundaries. Both interfaces must be defined in the same module, or else they cannot be in modules at all (though they can be in separate files).

To create a circular relationship between two components, follow these steps:

1. Create the first interface, without its reference or relationship to the second interface.
2. Create the second interface, with its reference or relationship to the first.
3. Edit the first interface, and add its reference or relationship to the second.

A foreign key pattern is a specific case of a circular relationship. It is documented in full in the foreign key pattern task.

RELATED CONCEPTS
"Foreign Key Patterns"

RELATED TASKS
"Create a Relationship" on page 129
"Define a Foreign Key Pattern" on page 133

Foreign Key Patterns

When a schema contains a foreign key reference (for example, the schema for Customer has a foreign key reference to Agent), this allows for more efficient relationships on the component level. For example, if Agent has a one-to-many relationship with Customer, calls to find a particular customer can be resolved on the database level, instead of on the business object level.

To take advantage of a foreign key reference on the component level, you need to define a component with a foreign key attribute (based on the foreign key reference), and then edit the component referenced by the foreign key attribute, to add a one-to-many relationship in the other direction (resolving references by foreign key).

For example, the component Agent has a one-to-many relationship with the component Customer, and the component Customer has an inverse object reference to the component Agent (each agent can have multiple customers, but each customer is represented by only one agent).

Foreign key relationships give better performance with SQL queries, because the references resolve directly to a database table, rather than indirectly through business object and data object attributes.

Once you define these relationships on the component level (a one-to-many relationship with foreign key support, and inverse references based on foreign keys), the foreign key attribute (for example, Customer's inverse reference to Agent) can be mapped to a foreign key in the imported .sql for the component.

Object Builder currently will not identify foreign keys in .sql files it generates. It only allows you to build components with foreign key relationships, based on schemas that contain foreign key references.

RELATED CONCEPTS
Object Relationships (*Programming Guide*)
Data Object Customization for Cardinality Relations (*Programming Guide*)

RELATED TASKS
"Define a Foreign Key Pattern"
"Customize Referential Integrity" on page 108

Define a Foreign Key Pattern

When a schema contains a foreign key reference (for example, the schema for Customer has a foreign key reference to Agent), this allows for more efficient relationships on the component level. For example, if Agent has a one-to-many relationship with Customer, calls to find a particular customer can be resolved on the database level, instead of on the business object level.

To take advantage of a foreign key reference on the component level, you need to define a component with a foreign key attribute (based on the foreign key reference), and a component with a one-to-many relationship (resolving references by foreign key).

To define these relationships, follow these steps:

1. Import the SQL DDL files that define the schemas for the related components (for example, myDB.Customer and myDB.Agent).
2. Create persistent objects from the schemas (for example, CustomerPO and AgentPO).
3. Create skeleton business object interfaces (for example, Customer and Agent). Specify their names only, do not specify their attributes or relationships.

   Note: If the two interfaces are defined in separate files, they cannot be contained in modules. If they are defined in modules, they must be defined in the same file. They cannot be defined in separate modules of separate files.
4. Complete the business object interface that owns the foreign key reference. Make sure the interface includes an object reference equivalent to the foreign key reference.

   The object reference represents a many-to-one relationship (many Customers share one Agent). You create this relationship in the same way you would create a one-to-one relationship, by creating an attribute of the referenced type (for example, the business object interface is defined with an attribute myAgent of type Agent). This is the foreign key attribute. The foreign key attribute cannot be read-only.
5. Create the business object implementation, data object interface and implementation, key, and copy helper for the owner of the foreign key reference (for example, CustomerBO, CustomerDO, CustomerDOImpl, CustomerKey, CustomerCopy).

Make sure the foreign key attribute (for example myAgent) is part of the component's state data, and identified in the component's key.

6. Complete the business object interface and implementation referenced by the foreign key, and define its one-to-many relationship (for example, add a one-to-many relationship from Agent to Customer, so that each agent can have multiple customers).

   To define a one-to-many relationship with references resolved by foreign key, follow these steps:

   a. Open the Business Object Interface wizard by selecting **Properties** from the pop-up menu of the business object interface.
   b. Click the title bar and turn to the Object Relationships Page.
   c. From the Relationships pop-up menu, click **Add**.
   d. Type a name for the relationship.
   e. From the **Object Type** drop-down list, select the interface that has the foreign key reference (for example, Customer).
   f. Specify the name of the home that will hold the component that owns the foreign key reference (for example, CustomerMOHome).
   g. Click **Finish**.
   h. From the pop-up menu of the interface, click **Add Implementation** to open the Business Object Implementation wizard.
   i. Click the title bar and turn to the Object Relationships Page.
   j. Under the Relationships folder, click on the relationship you defined in the interface. You can now set the implementation behavior for the relationship.
   k. Under **Reference Collection Implementation**, click **Reference resolved using foreign key**.

      Note: This option is enabled only when the selected object type meets the criteria for a foreign key relationship (it is either defined in the same file or neither interface is defined in a module, and it has a reference to the current object).
   l. From the **Foreign Key Attribute** list, select the attribute of the object that you want to use as the foreign key in this relationship. The list only displays attributes with the same type as the current object (for example, Customer's attribute myAgent of type Agent).
   m. In the **Home to Query** field, specify the home that will be used on the server to find objects of the selected type. The home you select must be the same one you configure with the target component's managed object.

      Typically the home name is derived from the target managed object's name (for example, CustomerMOHome).
   n. Click **Finish**.

7. Complete the rest of the component objects (for example, AgentBO, AgentDO, AgentDOImpl, AgentKey, AgentCopy).

8. Complete the component that owns the one-to-many relationship by mapping the data object implementation of the component to its equivalent persistent object (for example, map AgentDOImpl to AgentPO):

   a. In the implementation's wizard, add an instance of the persistent object to the implementation's Associated Persistent Objects Page.
   b. Turn to the Attributes Mapping Page and map the data object attributes to the persistent object attributes.
   c. Turn to the Methods Mapping Page and map the framework methods there to methods of the persistent object.

9. Complete the component that owns the foreign key reference by mapping the data object implementation of the component to its equivalent persistent object (for example, map CustomerDOImpl to CustomerPO):
   a. In the implementation's wizard, add an instance of the persistent object to the implementation's Associated Persistent Objects Page.
   b. Turn to the Attributes Mapping Page and map the data object attributes to the persistent object attributes.
   c. Map the foreign key attribute using the **Key Home** option, and then map the key attributes to their equivalents in the persistent object.
   d. Turn to the Methods Mapping Page and map the framework methods there to methods of the persistent object.

The foreign key pattern is now established.

RELATED CONCEPTS

"Foreign Key Patterns" on page 132
"Components" on page 15
"Home" on page 342

RELATED TASKS

"Define a One-to-One Relationship" on page 130
"Define a One-to-Many Relationship" on page 131
"Map a Data Object to a DB Persistent Object" on page 251
"Map Attributes Using a Key" on page 258
"Customize Referential Integrity" on page 108

Store an Object Reference

Because an object reference is literally a memory address, it needs to be converted into a more permanent form before it can be stored persistently. Object Builder supports the following handle patterns for a persistent reference:

- Stringified Object Reference (SOR)
- Object Name
- Home Name and Key

The handle pattern used to store references to a particular object type is set in the business object implementation of that object. The handle pattern can be overridden, however, by the referencing object, as set in the data object implementation of the referencing object.

To set or change the default handle pattern for a particular object type, follow these steps:

1. Open the Business Object Implementation wizard (either by adding a new business object implementation to an interface, or by selecting **Properties** from the pop-up menu of an existing business object implementation).
2. Click the title bar and turn to the Handle Selection Page.
3. Select the handle that will be used by default to store references to this type of object.
4. Complete the remaining wizard pages (if this is a new business object implementation), or click **Finish**.

To override the default behavior and use a single storage pattern for references to all types of objects, follow these steps:

1. Open the Data Object Implementation wizard (either by adding a new data object implementation to an interface, or by selecting **Properties** from the pop-up menu of an existing data object implementation).
2. Turn to the Behavior page..
3. Under "Handle for Storing Pointers" on page 35, select the handle you want to use for swizzling pointers.
4. Complete the remaining wizard pages (if this is a new data object implementation), or click **Finish**.

RELATED CONCEPTS

Using Handles (*Programming Guide*)
Object Relationships (*Programming Guide*)
Data Object Customization for Cardinality Relations (*Programming Guide*)

RELATED TASKS

"Add a Business Object Implementation and Data Object Interface" on page 284
"Add a Data Object Implementation" on page 299

Create a Child Component

You can create a child component in any of the following ways:

- "Define a Child with Attributes Duplication" on page 142
- "Define a Child with Key Duplication" on page 149
- "Define a Child with a Single Datastore" on page 156
- "Define a Child with Views" on page 164

All of these patterns assume that you are **not** overriding attributes in the business object implementation.

These patterns differ primarily in the way the data object maps to the persistent object (in other words, the way that the object hierarchy is mapped to the datastore). The general tasks involved in that step are as follows:

- "Map a Data Object to the Parent's Persistent Object" on page 254
- "Map a Data Object to the Child's Persistent Object" on page 255

Once you have created the child component, you can build and package it:

1. Build a Child Component
2. Package a Child Component

RELATED CONCEPTS

"Inheritance" on page 137
"Inheritance and Overriding in Business Objects" on page 138
"Choosing an Inheritance Pattern for Persistence" on page 140
"Components" on page 15

RELATED TASKS

"Inheritance with Attributes Duplication - Scenario" on page 144
"Inheritance with Key Duplication - Scenario" on page 151
"Inheritance with a Single Datastore - Scenario" on page 158
"Inheritance with Views - Scenario" on page 165

Inheritance

You can inherit data and behavior between components in Object Builder.

You do not need to explicitly inherit between objects in the same component (for example, a business object and data object, or business object and copy helper). The relationship between the objects is handled by Object Builder.

You do not need to include any of the framework interface files for Component Broker frameworks that your components inherit from. This also is handled by Object Builder.

Child components can inherit full implementations from their parent component, or only the interface.

When you create a child component with interface inheritance, only the child business object interface needs to inherit from the parent. Then, in the child business object implementation, the inherited interfaces can be implemented (by selecting to override the parent methods and attributes in the Business Object Implementation wizard). The rest of the child component objects do not have inheritance.

When you create a child component with implementation inheritance, the child component objects generally inherit from their equivalent parent objects:

- The child business object file should include the parent business object file.
- The child business object interface should inherit from the parent interface.
- It may not be necessary to have a child key and copy helper. If the child has the same key attribute as the parent, it can re-use the parent's key. If the child does not add any new attributes, it can re-use the parent's copy helper. If you do add a child key and copy helper, then they can either inherit from their equivalents in the parent component, or they can contain selected attributes of the parent interface, without inheriting from the parent key or copy helper.
- The child business object implementation should inherit from the parent implementation.
- The child data object interface should inherit from the parent data object interface.
- The child data object implementation should inherit from the parent data object implementation.
- The child managed object should inherit from the parent managed object.

For data inheritance to work, the type of persistence provided by the parent and child data object implementations should be the same.

RELATED CONCEPTS

"Components" on page 15
"Inheritance and Overriding in Helper Objects" on page 138
"Inheritance and Overriding in Business Objects" on page 138
"Inheritance and Overriding in Data Objects" on page 139
"Abstract Base Class Inheritance" on page 140
"Choosing an Inheritance Pattern for Persistence" on page 140
"Inheritance and Overriding in Helper Objects" on page 138
"Inheritance with Attributes Duplication" on page 141

"Inheritance with Key Duplication" on page 147
"Inheritance with a Single Datastore" on page 155
"Inheritance with Views" on page 162

RELATED TASKS
"Create a Child Component" on page 136

Inheritance and Overriding in Helper Objects

When you create the key and copy helper for a child component, you have the option of including some or all of the parent's equivalent attributes as part of the helper.

For a child's key, you have three options:

- Use the parent's key.
  If the child has the same key attributes as the parent, there is no need to create a separate key; you can simply re-use the one created for the parent. In the child's Data Object Implementation wizard, on the Key and Copy Helper page, select the parent's key.
- Use a mix of parent key attributes and child key attributes.
  In the child's Key wizard, on the Name and Key Attributes page, you have parent key attributes available for selection. Select some or all of these, and then select additional identifying attributes that are unique to the child.
- Use all the parent key attributes and additional child key attributes
  In the child's Key wizard, on the Name and Key Attributes page, select the child attributes you want to be part of the key. Do **not** select any of the parent attributes. Click **Next** to turn to the Implementation Inheritance page, and select the parent key to inherit from. The child's key then inherits the parent's key attributes, in addition to having the child key attributes specified on the previous page.
- Use only child key attributes.
  If the parent object has no identity in common with the child, then there is no reason for their keys to be related. You can create an entirely new key to reflect the child's unique identity, which includes none of the parent's attributes, and has default inheritance only.

For a child's copy helper, you have the same choices. The choice that makes sense will depend on the creation scenarios in which you intend to use the copy helper.

RELATED CONCEPTS
"Inheritance" on page 137
"Choosing an Inheritance Pattern for Persistence" on page 140
"Key" on page 21
"Copy Helper" on page 21

RELATED TASKS
"Create a Child Component" on page 136

Inheritance and Overriding in Business Objects

You can inherit both business object interface and business object implementation from the parent. In the business object implementation, you can select which attributes and methods you want to override. Generally you would do so if you wanted to change the way these attributes mapped to, or interacted with, the data

object. If you override an attribute or method in the child business object, and also choose to push it down to the child data object, then the child data object should **not** inherit from the parent data object. Otherwise the overridden attributes or methods will be defined twice, once through its association with the business object, and once through its inheritance from the parent.

There are three main situations in which you would override in the child's implementation:

- Overriding all attributes and inheriting behavior
  You can inherit behavior (method implementations) from a parent class, while overriding all its attributes. This is only appropriate for parent classes that have no data in the data object. The parent will have a business object interface, business object implementation, and a data object interface that contains no data. The child will inherit from each of the parent objects.
- Overriding all attributes and behavior
  You can use a parent class for interface-only inheritance, by overriding all its attributes and methods in the business object implementation. The child will inherit from the parent business object interface only. This pattern also applies to abstract base class inheritance.
- Overriding no attributes, overriding some or all behavior
  There are no restrictions on overriding methods, except for PA push-down methods (which have the same restrictions as attributes).

RELATED CONCEPTS
"Inheritance" on page 137
"Inheritance and Overriding in Helper Objects" on page 138
"Inheritance and Overriding in Data Objects"
"Abstract Base Class Inheritance" on page 140
"Choosing an Inheritance Pattern for Persistence" on page 140

RELATED TASKS
"Create a Child Component" on page 136

Inheritance and Overriding in Data Objects

You can inherit both interface and implementation from the parent. In the data object implementation, you can selectively map both local attributes and inherited attributes to an associated persistent object. When you map an inherited attribute, the mapping overrides the parent's mapping. In other words, the parent's mapping will still be in effect for the parent, but will be overridden in the child.

If you map all the inherited attributes to the child's persistent object, you are using the attributes duplication pattern of inheritance. If you map only the parent's key attributes to the child's persistent object, you are using the key duplication pattern of inheritance.

RELATED CONCEPTS
"Inheritance" on page 137
"Inheritance and Overriding in Helper Objects" on page 138
"Inheritance and Overriding in Business Objects" on page 138
"Abstract Base Class Inheritance" on page 140
"Choosing an Inheritance Pattern for Persistence" on page 140
"Inheritance with Attributes Duplication" on page 141
"Inheritance with Key Duplication" on page 147

RELATED TASKS
"Create a Child Component" on page 136

Abstract Base Class Inheritance

Abstract base classes are not supported by either the Interface Definition Language Compiler (IDLC) or Object Builder. So, any business object interface that you specify as a parent for another business object interface must have an implementation, even if every method in that implementation only throws a NO_IMPLEMENT exception.

You can create the equivalent of abstract base class inheritance by defining a business object with a minimal implementation and an empty data object interface. Child components can inherit from the business object interface only, and then implement all the parent attributes and methods in the child business object implementation (by selecting to override them in the Business Object Implementation wizard).

A component that acts as an abstract base class for inheritance purposes should consist of the following objects:

- Business object interface
  Defines the interface to the base class. Child business object interfaces inherit from this class.
- Business object implementation
  Contains skeleton implementations for methods and attributes. Methods at minimum throw the NO_IMPLEMENT exception.
- Data object interface
  Contains no data. Allows framework inheritance to work correctly.

RELATED CONCEPTS
"Inheritance" on page 137
"Inheritance and Overriding in Helper Objects" on page 138
"Inheritance and Overriding in Business Objects" on page 138
"Inheritance and Overriding in Data Objects" on page 139
"Choosing an Inheritance Pattern for Persistence"

RELATED TASKS
"Create a Child Component" on page 136

Choosing an Inheritance Pattern for Persistence

There are four main patterns for inheritance with persistence. For any of these patterns to work, you must **not** be overriding attributes in the business object implementation.

Your choice of inheritance pattern is based on three concerns:

- Identity: whether parent and child have the same identity (that is, they share the same key)
- Performance tradeoffs: whether performance or space efficiency is more important.
- Form of persistence: whether the parent has data to be persisted, and where and how the parent's and child's data is persisted.

If the parent and child have different keys, you should probably use the attributes duplication pattern. This means that the child's datastore provides persistence for all of its data, including inherited data (that is, the parent's attributes are duplicated in the child's datastore). The parent's datastore only provides persistence for instances of the parent, never for instances of the child. If you do not use the attributes duplication pattern when there are different keys, the parent's datastore will have two primary keys: the parent's key for the parent's data, and the child's key for the child's inherited data. It then becomes problematic to determine which data belongs to which object type.

If the parent and child have the same key, you can choose between the key duplication pattern and the single datastore with views pattern. The key duplication pattern will generally be more efficient in its use of space (because the persistent objects for each component contain only the data required for that component, and only the parent's key is duplicated in the child), and the views pattern will generally provide faster look-up time (because both local and inherited data are mapped to the same underlying datastore). The views pattern is based on views of the underlying database table, and requires that there be some unique attribute of the child that can be used to select appropriate views of the database.

If the parent and child have the same key and the parent never actually exists on its own (for example, there are never any pure Person instances kept in the datastore), you can use the single datastore pattern instead of the views pattern. Views are only required to select out the different object types being stored, and if the datastore only provides persistence for child and inherited attributes, the views are unnecessary.

RELATED CONCEPTS

"Inheritance" on page 137
"Inheritance with Attributes Duplication"
"Inheritance with Key Duplication" on page 147
"Inheritance with a Single Datastore" on page 155
"Inheritance with Views" on page 162

RELATED TASKS

"Create a Child Component" on page 136
"Define a Child with Attributes Duplication" on page 142
"Define a Child with Key Duplication" on page 149
"Define a Child with a Single Datastore" on page 156
"Define a Child with Views" on page 164

Inheritance with Attributes Duplication

If you have or want completely separate datastores for pure parent objects and parent objects that are also child objects, you can duplicate the attributes of the parent in the child's datastore. For example, data for a Person who isn't a Beneficiary is stored in the Person datastore, and data for a Person who is a Beneficiary is stored in the Beneficiary datastore.

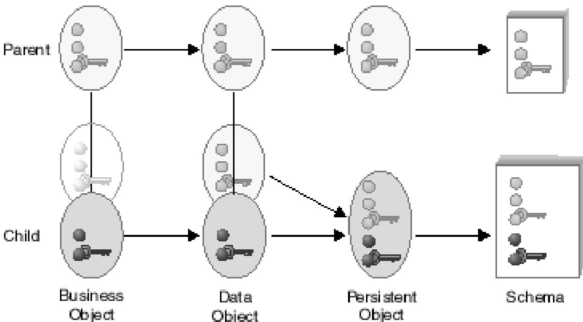
You can duplicate the parent's attributes in the child's datastore when you create the persistent object and schema from the data object. By mapping the parent's attributes to the child's persistent object, you implicitly override the parent's mapping. In other words, the parent's mapping will still be in effect for the parent, but will be overridden in the child.

For example, if Person has a child Beneficiary, then Person has a datastore that holds Person's attributes, and Beneficiary has a datastore that holds the total of Person's attributes and Beneficiary's attributes.

Advantages
The potential advantage to this approach is that you have a separate datastore for each type of object, regardless of its inheritance relationships. If it is important to maintain Person and Beneficiary in different datastores (for example, in different tables, different databases, or through different PA beans), then this approach can support that distinction, while still providing a unified object-oriented interface to the data.

This approach also allows the parent and child to use different keys to access their data, so the child does not have to use the parent's key

Disadvantages
This approach takes up more space than the other patterns, because of the duplication of attributes.



In this pattern:

- The parent's data object attributes and special framework methods are mapped to the parent's persistent object.
- The child's data object attributes, inherited attributes, and special framework methods are mapped to the child's persistent object.

RELATED CONCEPTS
"Inheritance" on page 137
"Choosing an Inheritance Pattern for Persistence" on page 140

RELATED TASKS
"Create a Child Component" on page 136
"Inheritance with Attributes Duplication - Scenario" on page 144

Define a Child with Attributes Duplication

This task covers the main steps necessary to create a component that inherits from another component already defined in Object Builder, and provides its own duplicated persistence for any inherited attributes. It does not cover every step; you should first be familiar with the tasks necessary to create a component without inheritance.

To create a child component in Object Builder, follow these steps:

1. Create the business object file.
2. Add the business object interface, and select the parent's business object interface on the Interface Inheritance Page.
3. If the child's identity differs from the parent's identity (in other words, it defines its own key attributes), you can add a key for the child. You can include attributes of the parent's key either by selecting specific attributes on the Name and Key Attributes Page, or include all the parent's attributes by selecting the parent key on the Implementation Inheritance Page. Do **not** do both.

   If the child has the same key attributes as the parent, then you do not need to create a key for the child. You can simply re-use the parent's key.
4. Add the copy helper. You can include attributes of the parent's copy helper either by selecting specific attributes on the Name and Attributes Page, or include all the parent's attributes by selecting the parent copy helper on the Implementation Inheritance Page. Do **not** do both.
5. Add the business object implementation:
   a. Under Data Object Interface, click **Add or select one later**. This allows you to add the data object interface in a separate step, and define its parent.
   b. Select the parent's business object implementation on the Implementation Inheritance Page.
   c. Do **not** override any attributes on the Attributes to Override page.
   d. Select any methods you want to override on the Methods to Override Page.
6. Add the managed object, and select the parent's managed object on the Implementation Inheritance Page.
7. Add the data object interface:
   a. From the business object implementation's pop-up menu, click **Add New Data Object Interface**.
   b. Select the attributes and methods of the business object you want represented in the data object.
   c. You should select the parent data object interface on the Interface Inheritance page.
8. Add the data object implementation, and select the parent data object implementation on the Implementation Inheritance page.
9. Add a persistent object and schema:
   a. On the Attributes Mapping page, click **Horizontal Partitioning** to map the child's attributes and inherited attributes to the child's persistent object.
   b. On the Methods Mapping page, map the special framework methods to the child's persistent object.

RELATED CONCEPTS

"Inheritance" on page 137
"Choosing an Inheritance Pattern for Persistence" on page 140
"Inheritance and Overriding in Helper Objects" on page 138
"Components" on page 15

RELATED TASKS

Create a Component - Overview
"Create a Child Component" on page 136
Build a Child Component
"Inheritance with Attributes Duplication - Scenario" on page 144

Inheritance with Attributes Duplication - Scenario

In this scenario you define a child component that provides its own persistence for inherited attributes (duplicating the persistence provided by its parent).

Before following these instructions, you should have the Person component defined and exported in XML format (PFile.xml), as described in the Simple Database Persistence - Scenario.

After you complete this scenario, you will have a component named Beneficiary that inherits from Person, and which provides persistence in a database table both for its own attributes, and for the attributes it inherits from Person.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizards. If you are experiencing problems, click the Help button within a wizard, or go to the Help pulldown in Object Builder.

Create the Project
Create a sample project to hold your work.

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory (for example, e:\scenarios\inheritadup).
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Import PFile.xml
Import the definition of the Person component, as created in the Database Persistence - Scenario:

1. From the User-Defined Business Objects folder's pop-up menu, click **Import - XML**.
2. Find and select PFile.xml.
3. Click **Finish**.

The component objects for Person appear in the folder.

Create the Business Object Interface
Define the Beneficiary interface:

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file BFile.
3. Click **Finish**. The file now appears under the folder.
4. From the file's pop-up menu, click **Add Module** to open the Business Object Module wizard.
5. Name the module BModule.
6. Click **Finish**. The module now appears under the file.
7. From the module's pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
8. Name the interface Beneficiary.
9. Click the title bar and turn to the Interface Inheritance page.
10. Add Person as a parent (replacing the default inheritance).
11. Click the title bar and turn to the Attributes page.

12. Add the following attributes:
    - readonly long id
    - float claimPayments
13. Click **Finish**. The interface now appears under the module.

Add the Key and Copy Helper
Add BeneficiaryKey:
1. From the interface's pop-up menu, click **Add Key** to open the Key wizard.
2. Select id as a key attribute.
3. Add ssNo and name as key attributes (so Beneficiary's identity includes the key attributes for its parent).

   Beneficiary's key now consists of the following:
   - long id (defined in Beneficiary)
   - string ssNo (defined in Person)
   - string name (defined in Person)
4. Click **Finish**. The key now appears under the interface.

Add BeneficiaryCopy:
1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Add all attributes to the copy helper.
3. Click **Finish**. The copy helper now appears under the interface.

Add the Business Object Implementation
Add BeneficiaryBO:
1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Set **Data Object Interface - Add or select one later** (you will create a new data object as a separate step).
3. Click **Next** to turn to the Implementation Inheritance page.
4. Add PersonBO as a parent.
5. Click the title bar and turn to the Key and Copy Helper page.
6. Select BeneficiaryKey and BeneficiaryCopy.
7. Click **Finish**. The business object implementation appears under the business object interface.

Add the Data Object Interface
Add BeneficiaryDO:
1. From the business object implementation's pop-up menu, click **Add New Data Object Interface** to open the Data Object Interface wizard.
2. Select all the business object attributes as state data (to be preserved in the data object).
3. Click the title bar and turn to the Interface Inheritance page.
4. Add PersonDO as a parent.
5. Click **Finish**. The data object interface appears under the business object implementation.

Add the Data Object Implementation
Add BeneficiaryDOImpl:

1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. Set the following patterns:
   - **Environment - BOIM with any key**
   - **Form of Persistent Behavior and Implementation - Embedded SQL**
   - **Data Access Pattern - Delegating**
3. Click **Next** to turn to the Implementation Inheritance page.
4. Add PersonDOImpl as a parent.
5. Click the title bar and turn to the Key and Copy Helper page.
6. Select BeneficiaryKey and BeneficiaryCopy.
7. Click **Finish**. The data object implementation appears under the data object interface.

Add the Persistent Object and Schema

Add BeneficiaryPO and its associated schema:

1. From the data object implementation's pop-up menu, click **Add Persistent Object and Schema** to open the Add Persistent Object and Schema wizard.
2. Type a name for the schema group that will hold the schema, and for the database.
3. Click **Next** to turn to the Attributes Mapping page. Both Beneficiary's attributes and Person's attributes are displayed.
4. Click **Horizontal Partitioning**. This maps all attributes (both Beneficiary's and Person's) to attributes of BeneficiaryPO.

   In this step, two things are happening:
   - Because BeneficiaryPO does not actually exist yet, this step defines what attributes BeneficiaryPO will contain.
   - By mapping Person's attributes to BeneficiaryPO, you are implicitly overriding the mapping in the Person component. BeneficiaryDOImpl now has its own copy of Person's attributes, which map to BeneficiaryPO instead of PersonPO.
5. Click **Finish**. The persistent object and schema appear under the data object implementation.

Add the Managed Object

Add BeneficiaryMO:

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard.
2. Click **Finish**. The managed object now appears under the business object implementation.

Configure the Build

You have now completed the definition of the Beneficiary component, and its inheritance from Person. The next step is to configure the client and server DLLs that will hold the components.

Define the Client DLL

Add the PBClient DLL:

1. From the Build Configuration folder's pop-up menu, click **Add Client DLL** to open the Client DLL wizard.
2. Name the DLL PBClient.

3. Click **Next** to turn to the Client Source Files page.
4. Select PFile, PFileKey, and PFileCopy (the Person client interfaces).
5. Select BFile, BFileKey, and BFileCopy (the Beneficiary client interfaces).
6. Click **Finish**. The client DLL appears under the folder.

Define the Server DLL
Add the PBServer DLL:

1. From the Build Configuration folder's pop-up menu, click **Add Server DLL** to open the Server DLL wizard.
2. Name the DLL PBServer.
3. Click **Next** to turn to the Server Source Files page.
4. Select PFileBO, PFileDO, PFileDOImpl, and PFileMO (the Person server interfaces).
5. Select BFileBO, BFileDO, BFileDOImpl, and BFileMO (the Beneficiary server interfaces).
6. Click **Next** to turn to the Libraries to Link With page.
7. Select the PBClient library file.
8. Click **Finish**. The server DLL appears under the folder.

Build the DLLs
Build the PBClient and PBServer DLLs:

1. From the pop-up menu of the User-Defined Business Objects folder, click **Generate - All**.
2. Wait for the code generation to complete. The generated source files are placed in the project's \Working directory.
3. From the pop-up menu of the Build Configuration folder, click **Generate - All - All Targets**.
4. From the same pop-up menu, click **Build - All Targets**. The DLLs are built and placed in the project's \Working directory.

Inheritance with Key Duplication

If your datastores are divided in a way that mirrors your component hierarchy, then inheritance works the same for persistent data as it does for data on the object level. In other words, a child has its own datastore for its own attributes, and uses its parent's datastore for inherited attributes. The only exception is for key attributes: in this pattern, the child typically uses the same key as the parent, and the parent's key is duplicated in the child's datastore.

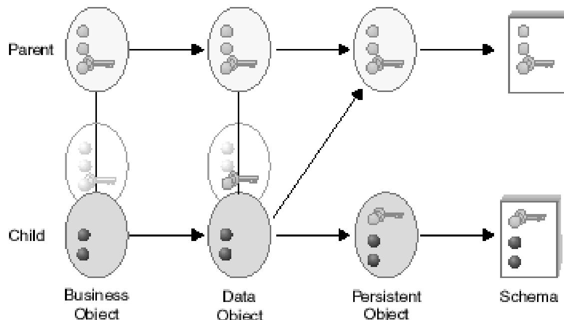
This is the default inheritance pattern in Object Builder. If you create new parent and child components (starting from the business object interface and working down to persistent objects and DB schemas), then each schema holds only the definitions for data defined in its component. The child component uses its own persistent object for its own data, and its parent's persistent object for inherited data.

Advantages
The advantage to this approach is its precision, and efficient use of space.

Disadvantages
Because data access can span multiple datastores, access time may be slower than with other patterns. Also, this approach is problematic if your parent and child use different keys. Because part of the child's data is stored in the parent's

datastore, the parent datastore needs to support both keys (the child's and the parent's), to ensure data for the right object type is returned. Generally, you should only use this pattern when the parent and the child use the same key.

For this pattern, the parent's table and the child's table must be in the same database.



In this pattern:

- The parent's data object attributes and special framework methods are mapped to the parent's persistent object.
- The child's data object attributes, and its inherited key, are mapped to the child's persistent object. This creates a duplicate of the parent's key in the child's persistent object, which allows it to locate the parent's persistent object when it needs to retrieve inherited attributes.
- The child's data object special framework methods are mapped in one of two ways, depending on the type of creation and deletion scenarios you want to support.

  If you want to support creation of a child with an existing parent entry, and deletion of a child without deletion of its parent entry, map as follows:

  – insert and update map to first the parent's and then the child's persistent objects, with the **Always complete calling sequence** option checked. (For example, insert maps to iPersonPO.insert and iBeneficiaryPO.insert.)

    Because they map to both, and the calling sequence will ignore errors, you can successfully create a Beneficiary that already exists as a Person: the parent insert will fail, but still proceed to the child insert, which is successful.

    You will not be able to set values for the attributes of an existing parent during creation of the child. If you create the child using a copy helper, any values you set for inherited attributes are ignored, since they are applied to the parent's existing records using insert, when they need to use update. You can change the inherited attributes in a separate update call after you create the child.

  – retrieve and setConnection map to first the child's and then the parent's persistent objects, with the **Always complete calling sequence** option **not** checked. (For example, retrieve maps to iBeneficiaryPO.retrieve and iPersonPO.retrieve.)

    Because Beneficiary stores its inherited attributes in Person's datastore, it **must** be able to retrieve the parent's data. If an error occurs on the parent's retrieve, it abandons the calling sequence and returns an error.

  – delete maps to the child's persistent object. (For example, iBeneficiaryPO.delete.)

Because the delete method maps only to the child's persistent object, when a child object is deleted, its record as a parent object remains. (For example, when you delete a Beneficiary, you retain an entry for the Person, even though the Person is no longer a Beneficiary.)

If you want to create only new parents and children, and delete the child and its parent in the same step, map as follows:

– insert and update map to first the parent's and then the child's persistent objects, with the **Always complete calling sequence** option **not** checked. (For example, insert maps to iPersonPO.insert and iBeneficiaryPO.insert.)

  This always creates a new parent along with the child.

  If you wanted to create a new child from an existing parent, you could still find the existing parent, create a copy of its attribute values, delete the parent, and then create the child as a new object with the values of the deleted parent.

– retrieve and setConnection map to first the child's and then the parent's persistent objects, with the **Always complete calling sequence** option **not** checked. (For example, retrieve maps to iBeneficiaryPO.retrieve and iPersonPO.retrieve.)

  Because Beneficiary stores its inherited attributes in Person's datastore, it **must** be able to retrieve the parent's data. If an error occurs on the parent's retrieve, it abandons the calling sequence and returns an error.

– delete maps to first the child's and then the parent's persistent objects, with the **Always complete calling sequence** option **not** checked. (For example, iBeneficiaryPO.delete and iPersonPO.delete.)

  This deletes the parent along with the child.

  If you wanted to delete the child and leave the parent entry, you could still copy the existing parent values, continue with the deletion of the child, and then recreate the parent with the copied values.

RELATED CONCEPTS

"Inheritance" on page 137
"Choosing an Inheritance Pattern for Persistence" on page 140

RELATED TASKS

"Create a Child Component" on page 136
"Inheritance with Key Duplication - Scenario" on page 151

Define a Child with Key Duplication

This task covers the main steps necessary to create a component that inherits from another component already defined in Object Builder, and duplicates its parent's key in the child's datastore, so it can look up its parent and use its parent's persistence for other inherited attributes. It does not cover every step; you should first be familiar with the tasks necessary to create a component without inheritance.

To use the key duplication pattern, the child must have the same key attributes as the parent. If the child has a different key, use the attributes duplication pattern. Also, if persistence is provided in a database, both the parent and child must use tables in the same database.

To create a child component in Object Builder, follow these steps:

1. Create the business object file.

2. Add the business object interface, and select the parent's business object interface on the Interface Inheritance Page.
3. Add the copy helper. You can include attributes of the parent's copy helper either by selecting specific attributes on the Name and Attributes Page, or include all the parent's attributes by selecting the parent copy helper on the Implementation Inheritance Page. Do **not** do both.
4. Add the business object implementation:
   a. Under Data Object Interface, click **Add or select one later**. This allows you to add the data object interface in a separate step, and define its parent.
   b. Select the parent's business object implementation on the Implementation Inheritance Page.
   c. Select the parent's key on the Key and Copy Helper page.
   d. Do **not** override any attributes on the Attributes to Override page.
   e. Select any methods you want to override on the Methods to Override Page.
5. Add the managed object, and select the parent's managed object on the Implementation Inheritance Page.
6. Add the data object interface:
   a. From the business object implementation's pop-up menu, click **Add New Data Object Interface**.
   b. Select the attributes and methods of the business object you want represented in the data object.
   c. You should select the parent data object interface on the Interface Inheritance page.
7. Add the data object implementation, and select the parent data object implementation on the Implementation Inheritance page.
8. Add a persistent object and schema, and map the attributes as follows:
   - On the Attributes Mapping page, click **Vertical Partitioning** to map the child's data object attributes, and its inherited key, to the child's persistent object.
9. Open the data object implementation's properties, and map the special framework methods as follows:
   - On the Methods Mapping page:
     - delete maps to the child's persistent object. (For example, iBeneficiary.delete.)
     - insert and update map to first the parent's and then the child's persistent objects, with the **Always complete calling sequence** option checked. (For example, insert maps to iPersonPO.insert and iBeneficiaryPO.insert.)
     - retrieve and setConnection map to first the child's and then the parent's persistent objects, with the **Always complete calling sequence** option **not** checked. (For example, retrieve maps to iBeneficiaryPO.retrieve and iPersonPO.retrieve.)

   These mappings support creation of a child when its entry as a parent already exists (for example, creation of a Beneficiary when a Person with the same key value already exists). If you wanted to restrict creation to entirely new objects, you could uncheck the **Always complete calling sequence** option on the insert and update mappings. This would mean that new children are always created with new parents.

   These mappings also support deletion of the child without deletion of its parent, leaving the parent entry behind (for example, deletion of a Beneficiary

does not affect its Person values). If you wanted to have deletion remove the parent along with the child, you could map the delete method to the parent's persistent object as well.

RELATED CONCEPTS

"Inheritance" on page 137
"Choosing an Inheritance Pattern for Persistence" on page 140
"Inheritance with Key Duplication" on page 147
"Components" on page 15

RELATED TASKS

Create a Component - Overview
"Create a Child Component" on page 136
Build a Child Component
"Inheritance with Key Duplication - Scenario"

Inheritance with Key Duplication - Scenario

In this scenario you define a child component that uses its parent's persistence for inherited attributes (so that each component in the object hierarchy provides persistence for its own attributes, plus its parent's key, which is used to look up the parent and find the value of inherited attributes). This inheritance pattern makes the most sense when parent and child share the same key. For a scenario where parent and child have different keys, see the Inheritance with Overriding Persistence - Scenario.

Before following these instructions, you should have the Person component defined and exported in XML format (PFile.xml), as described in the Simple Database Persistence - Scenario.

After you complete this scenario, you will have a component named Beneficiary that inherits from Person, and provides persistence in a database table for its own attributes (plus Person's key), and uses Person's database table to access inherited attributes. For example, a query on Beneficiary.name (an attribute inherited from Person) results in a lookup on Beneficiary's table to find the parent Person's key (which is duplicated in Beneficiary's table), and then a lookup on Person's table to find the value of the name attribute.

Note: For this pattern, the parent's table and the child's table must be in the same database.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizards. If you are experiencing problems, click the Help button within a wizard, or go to the Help pulldown in Object Builder.

Create the Project

Create a sample project to hold your work.

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory (for example, e:\scenarios\inheritoo).
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Import PFile.xml
Import the definition of the Person component, as created in the Database Persistence - Scenario:

1. From the User-Defined Business Objects folder's pop-up menu, click **Import - XML**.
2. Find and select PFile.xml.
3. Click **Finish**.

The component objects for Person appear in the folder.

Create the Business Object Interface
Define the Beneficiary interface:

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file BFile.
3. Click **Finish**. The file now appears under the folder.
4. From the file's pop-up menu, click **Add Module** to open the Business Object Module wizard.
5. Name the module BModule.
6. Click **Finish**. The module now appears under the file.
7. From the module's pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
8. Name the interface Beneficiary.
9. Click the title bar and turn to the Interface Inheritance page.
10. Add Person as a parent (replacing the default inheritance).
11. Click the title bar and turn to the Attributes page.
12. Add the following attribute:
    - float claimPayments
13. Click **Finish**. The interface now appears under the module.

Add the Copy Helper
Add BeneficiaryCopy:

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Add all attributes to the copy helper (both parent's and child's).
3. Click **Finish**. The copy helper now appears under the interface.

Add the Business Object Implementation
Add BeneficiaryBO:

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Set **Data Object Interface - Add or select one later** (you will create a new data object as a separate step).
3. Click **Next** to turn to the Implementation Inheritance page.
4. Add PersonBO as a parent.
5. Click the title bar and turn to the Key and Copy Helper page.
6. Select PersonKey and BeneficiaryCopy.
7. Click **Finish**. The business object implementation appears under the business object interface.

Add the Data Object Interface
Add BeneficiaryDO:

1. From the business object implementation's pop-up menu, click **Add New Data Object Interface** to open the Data Object Interface wizard.
2. Select all the business object attributes as state data (to be preserved in the data object).
3. Click the title bar and turn to the Interface Inheritance page.
4. Add PersonDO as a parent.
5. Click **Finish**. The data object interface appears under the business object implementation.

Add the Data Object Implementation
Add BeneficiaryDOImpl:

1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. Set the following patterns:
   - **Environment - BOIM with any key**
   - **Form of Persistent Behavior and Implementation - Embedded SQL**
   - **Data Access Pattern - Delegating**
3. Click **Next** to turn to the Implementation Inheritance page.
4. Add PersonDOImpl as a parent.
5. Click the title bar and turn to the Key and Copy Helper page.
6. Select PersonKey and BeneficiaryCopy.
7. Click **Finish**. The data object implementation appears under the data object interface.

Add the Persistent Object and Schema
Add BeneficiaryPO and its associated schema:

1. From the data object implementation's pop-up menu, click **Add Persistent Object and Schema** to open the Add Persistent Object and Schema wizard.
2. Type a name for the schema group that will hold the schema, and for the database.
3. Click **Next** to turn to the Attributes Mapping page. Both Beneficiary's attributes and Person's attributes are displayed.
4. Click **Vertical Partitioning**. This maps the child's attributes and the parent's key to the child's persistent object, creating a duplicate entry for the key in the child's persistent object.

   Because Beneficiary now has a record of the parent's key, a call to Beneficiary for an inherited attribute (such as town) can be delegated to the parent table. Beneficiary receives the call, then uses the parent's key to find the right row in the parent's table, and retrieve the called attribute. Contrast this with the Inheritance with Attributes Duplication - Scenario, in which all of the parent's data is persisted in the child's table.
5. Click **Finish**. The persistent object and schema appear under the data object implementation.

Map the Special Framework Methods
Map the way in which the data object's special framework methods will call the persistent objects' special framework methods:

1. From BeneficiaryDOImpl's pop-up menu, click **Properties** to open the Data Object Implementation wizard.
2. Click the title bar and turn to the Methods Mapping page.

   Because Beneficiary has its own data in one persistent object and inherited data in a separate persistent object, the special framework methods need to access both persistent objects in order to ensure all the right data is retrieved.
3. Map each method as follows:
   - insert and update map to first iPersonPO's methods and then iBeneficiaryPO's methods, with the **Always complete calling sequence** option checked.

     Because they map to both, and the calling sequence will ignore errors, you can successfully create a Beneficiary that already exists as a Person: the parent insert will fail, but still proceed to the child insert, which is successful.

     You will not be able to set values for the attributes of an existing parent during creation of the child. If you create the Beneficiary using a copy helper, any values you set for inherited attributes of Person are ignored, since they are applied to Person's existing records using insert, when they need to use update. You can change the inherited attributes in a separate update call after you create the child.
   - retrieve and setConnection map to first iBeneficiaryPO's methods and then iPersonPO's methods, with the **Always complete calling sequence** option **not** checked.

     Because Beneficiary stores its inherited attributes in Person's datastore, it **must** be able to retrieve the parent's data. If an error occurs on the parent's retrieve, it abandons the calling sequence and returns an error.

     By mapping to both the parent and the child persistent object, you allow a call to Beneficiary for its parent's data (for example, Beneficiary.town) to resolve as follows:
     a. Person's key is retrieved from iBeneficiaryPO
     b. The appropriate Person is located, and Person.town is retrieved from iPersonPO.
   - delete maps to iBeneficiaryPO.delete.

     Because the delete method maps only to the child's persistent object, when a Beneficiary is deleted, its record as a Person remains. (So you retain an entry for the Person, even though the Person is no longer a Beneficiary.)

   This scenario supports creation of a Beneficiary when its entry as a Person already exists. If you wanted to restrict creation to entirely new objects, you could uncheck the **Always complete calling sequence** option on the insert and update mappings. This would mean that new children are always created with new parents.

   This scenario also supports deletion of the Beneficiary without deletion of its parent Person, leaving the Person entry behind. If you wanted to have deletion remove the parent along with the child, you could map the delete method to the parent's persistent object as well.
4. Click **Finish**.

Add the Managed Object

Add BeneficiaryMO:

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard.

2. Click **Finish**. The managed object now appears under the business object implementation.

Configure the Build
You have now completed the definition of the Beneficiary component, and its inheritance from Person. The next step is to configure the client and server DLLs that will hold the components.

Define the Client DLL
Add the PBClient DLL:

1. From the Build Configuration folder's pop-up menu, click **Add Client DLL** to open the Client DLL wizard.
2. Name the DLL PBClient.
3. Click **Next** to turn to the Client Source Files page.
4. Select PFile, PFileKey, and PFileCopy (the Person client interfaces).
5. Select BFile and BFileCopy (the Beneficiary client interfaces).
6. Click **Finish**. The client DLL appears under the folder.

Define the Server DLL
Add the PBServer DLL:

1. From the Build Configuration folder's pop-up menu, click **Add Server DLL** to open the Server DLL wizard.
2. Name the DLL PBServer.
3. Click **Next** to turn to the Server Source Files page.
4. Select PFileBO, PFileDO, PFileDOImpl, and PFileMO (the Person server interfaces).
5. Select BFileBO, BFileDO, BFileDOImpl, and BFileMO (the Beneficiary server interfaces).
6. Click **Next** to turn to the Libraries to Link With page.
7. Select the PBClient library file.
8. Click **Finish**. The server DLL appears under the folder.

Build the DLLs
Build the PBClient and PBServer DLLs:

1. From the pop-up menu of the User-Defined Business Objects folder, click **Generate - All**.
2. Wait for the code generation to complete. The generated source files are placed in the project's \Working directory.
3. From the pop-up menu of the Build Configuration folder, click **Generate - All - All Targets**.
4. From the same pop-up menu, click **Build - All Targets**. The DLLs are built and placed in the project's \Working directory.

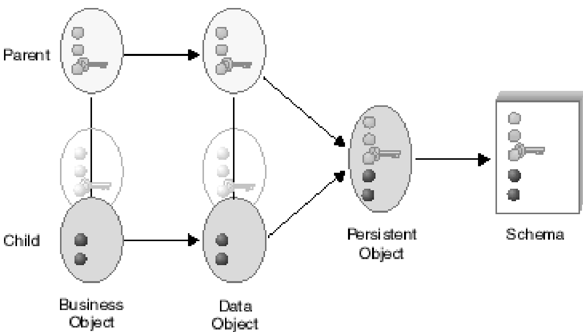
Inheritance with a Single Datastore

If all the components in an inheritance hierarchy share a single datastore (for example, both Person and its child Beneficiary store their data in the same database table), then you can represent the datastore with a single persistent object. You can then map the data object attributes of each component to selected persistent object attributes.

Essentially, this approach flattens the object hierarchy into a single datastore. There is only one entry for each unique attribute, and only one persistent object for both parent and child components.

Advantages
The advantage of this approach is faster access to the datastore, because both local and inherited attributes are in the same place.

Disadvantages
- This approach is problematic if you need to store pure parent objects (for example, a Person component that is not a Beneficiary). If you need to store both parent and child objects in the datastore, you should use the views pattern to select the relevant data for the different component types.
- This approach is not very efficient in its use of space: each component accesses only a small part of the datastore, leaving most of the persistent object and schema unused for any one specific task.
- This approach is problematic if your parent and child use different keys. Because both the parent's data and child's data is stored in a single datastore, the datastore needs to support both keys (the child's and the parent's), to ensure data for the right object type is returned. Generally, you should only use this pattern when the parent and the child use the same key.



In this pattern:
- The parent's data object attributes and special framework methods are mapped to the shared persistent object.
- The child's data object attributes and special framework methods are also mapped to the shared persistent object.

RELATED CONCEPTS
"Inheritance" on page 137
"Choosing an Inheritance Pattern for Persistence" on page 140

RELATED TASKS
"Create a Child Component" on page 136
"Inheritance with a Single Datastore - Scenario" on page 158

Define a Child with a Single Datastore

This task covers the main steps necessary to create a component that inherits from another component already defined in Object Builder, and shares a single datastore

with its parent. It does not cover every step; you should first be familiar with the tasks necessary to create a component without inheritance.

To use the single datastore pattern, the child must have the same key attributes as the parent, and the parent must be used for inheritance only (in other words, the only parent data in the datastore is for the child's inherited attributes). If the child has a different key, use the attributes duplication pattern. If the child has the same key but the parent is used as a real object (not just for inheritance), use the views pattern. The views pattern uses views of the datastore to select parent data of pure parent objects from parent data that is inherited by a child.

In the single datastore pattern, the parent's persistent object and schema include the attributes of the child component. Typically you would use this pattern after importing the data in a single large datastore into Object Builder, as part of a strategy to break up the data among several components in a class hierarchy.

To create the child component in Object Builder, follow these steps:

1. Create the business object file.
2. Add the business object interface, and select the parent's business object interface on the Interface Inheritance Page.
3. Add the copy helper. You can include attributes of the parent's copy helper either by selecting specific attributes on the Name and Attributes Page, or include all the parent's attributes by selecting the parent copy helper on the Implementation Inheritance Page. Do **not** do both.
4. Add the business object implementation:
   a. Under Data Object Interface, click **Add or select one later**. This allows you to add the data object interface in a separate step, and define its parent.
   b. Select the parent's business object implementation on the Implementation Inheritance Page.
   c. Select the parent's key on the Key and Copy Helper page.
   d. Do **not** override any attributes on the Attributes to Override page.
   e. Select any methods you want to override on the Methods to Override Page.
5. Add the managed object, and select the parent's managed object on the Implementation Inheritance Page.
6. Add the data object interface:
   a. From the business object implementation's pop-up menu, click **Add New Data Object Interface**.
   b. Select the attributes and methods of the business object you want represented in the data object.
   c. You should select the parent data object interface on the Interface Inheritance page.
7. Add the data object implementation:
   a. From the data object interface's pop-up menu, click **Add Implementation**.
   b. Select the parent data object implementation on the Implementation Inheritance page.
   c. Select the parent's key and the child's copy helper on the Key and Copy Helper page.
   d. Map the child's attributes to the parent's persistent object on the Attributes Mapping page.
8. Map the child's data object attributes and special framework methods to the parent's (now shared) persistent object.

RELATED CONCEPTS
"Inheritance" on page 137
"Choosing an Inheritance Pattern for Persistence" on page 140
"Inheritance with a Single Datastore" on page 155
"Components" on page 15

RELATED TASKS
Create a Component - Overview
"Create a Child Component" on page 136
Build a Child Component
"Inheritance with a Single Datastore - Scenario"

Inheritance with a Single Datastore - Scenario

In this scenario you define a parent component and child component that share a single datastore.

This inheritance pattern makes the most sense when parent and child share the same key. For a scenario where parent and child have different keys, see the Inheritance with Attributes Duplication - Scenario.

This scenario also does not use views, which means there is no easy way to determine when data is for a pure parent component, or part of the inherited data for a child component. While this is acceptable when there are no pure parent components to take into consideration (in other words, there are no pure parent instances to be persisted), it does not work well for datastores that contain a mix of objects. For a scenario using views on a mixed datastore, see the Inheritance with Views - Scenario.

Before following these instructions, you should have the Person component defined and exported in XML format (PFile.xml), as described in the Simple Database Persistence - Scenario.

After you complete this scenario, you will have a component named Beneficiary that inherits from Person, and a single database table that provides persistence for both Beneficiary's attributes and Person's attributes.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizards. If you are experiencing problems, click the Help button within a wizard, or go to the Help pulldown in Object Builder.

Create the Project
Create a sample project to hold your work.

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory (for example, e:\scenarios\inheritsh).
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Import PFile.xml
Import the definition of the Person component, as created in the Database Persistence - Scenario:

1. From the User-Defined Business Objects folder's pop-up menu, click **Import - XML**.

2. Find and select PFile.xml.
3. Click **Finish**.

The component objects for Person appear in the folder.

Create the Shared Table and Persistent Object

You need to create a schema that contains columns for all of Person's and Beneficiary's attributes.

1. Create a file with the following contents. If you are viewing this online, you can cut and paste these lines directly into an editor:
```
CREATE TABLE Shared
(
ssNo VARCHAR(20) NOT NULL ,
name VARCHAR(100) NOT NULL ,
street LONG VARCHAR ,
town LONG VARCHAR ,
claimPayments DOUBLE
, PRIMARY KEY
( ssNo, name )
);
```
2. From the pop-up menu of the DBA-Defined Schemas folder in Object Builder, click **Import - SQL**.
3. Select the file you created.
4. Name the database SharedDB (or provide the name of your own database).
5. Name the group MyGroup.
6. Click **Finish**.

   The schema appears in the folder, under the schema group.
7. From the pop-up menu of the schema, click **Add Persistent Object**.
8. Name the persistent object SharedPO, and name its package file SharedPkg.
9. Set its type of persistence to **Embedded SQL**, to match the type of persistence set in PersonDOImpl.
10. Click **Finish**.

The persistent object appears under the schema.

Map the Shared Table to the Parent

Map PersonDOImpl to SharedPO:

1. Delete PersonPO from under PersonDOImpl.
2. Delete Person's old schema from the DBA-Defined Schemas folder.
3. Delete PGroup from the DBA-Defined Schemas folder.
4. From the pop-up menu of PersonDOImpl, click **Properties** to open the Data Object Implementation wizard.
5. Click the title bar and turn to the Associated Persistent Objects page.
6. Add SharedPO as an associated persistent object, with the instance name iPersonPO.
7. Click **Next** to turn to the Attributes Mapping page.
8. Map Person's attributes to their equivalents in iPersonPO.
9. Click **Next** to turn to the Methods Mapping page.
10. Map Person's methods to their equivalents in iPersonPO.
11. Click **Finish**.

SharedPO and its schema now appear under PersonDOImpl.

You can now define the child component.

Create the Business Object Interface
Define the Beneficiary interface:

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file BFile.
3. Click **Finish**. The file now appears under the folder.
4. From the file's pop-up menu, click **Add Module** to open the Business Object Module wizard.
5. Name the module BModule.
6. Click **Finish**. The module now appears under the file.
7. From the module's pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
8. Name the interface Beneficiary.
9. Click the title bar and turn to the Interface Inheritance page.
10. Add Person as a parent (replacing the default inheritance).
11. Click the title bar and turn to the Attributes page.
12. Add the following attribute:
    - float claimPayments
13. Click **Finish**. The interface now appears under the module.

Add the Copy Helper
Add BeneficiaryCopy:

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Add all attributes to the copy helper (both parent's and child's).
3. Click **Finish**. The copy helper now appears under the interface.

Add the Business Object Implementation
Add BeneficiaryBO:

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Set **Data Object Interface - Add or select one later** (you will create a new data object as a separate step).
3. Click **Next** to turn to the Implementation Inheritance page.
4. Add PersonBO as a parent.
5. Click the title bar and turn to the Key and Copy Helper page.
6. Select PersonKey and BeneficiaryCopy.
7. Click **Finish**. The business object implementation appears under the business object interface.

Add the Data Object Interface
Add BeneficiaryDO:

1. From the business object implementation's pop-up menu, click **Add New Data Object Interface** to open the Data Object Interface wizard.
2. Select all the business object attributes as state data (to be preserved in the data object).
3. Click the title bar and turn to the Interface Inheritance page.

4. Add PersonDO as a parent.
5. Click **Finish**. The data object interface appears under the business object implementation.

Add the Data Object Implementation, and Map It to the Shared Persistent Object

Add BeneficiaryDOImpl, and map it to SharedPO:

1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. Set the following patterns:
   - **Environment - BOIM with any key**
   - **Form of Persistent Behavior and Implementation - Embedded SQL**
   - **Data Access Pattern - Delegating**
3. Click **Next** to turn to the Implementation Inheritance page.
4. Add PersonDOImpl as a parent.
5. Click the title bar and turn to the Key and Copy Helper page.
6. Select PersonKey and BeneficiaryCopy.
7. Click the title bar and turn to the Attributes Mapping page.
8. Map claimPayments to SharedPO.claimPayments. SharedPO is available for selection because it is associated with Beneficiary's parent.
9. Click **Finish**. The data object implementation appears under the data object interface.

Configure the Build

You have now completed the definition of the Beneficiary component, and its inheritance from Person. The next step is to configure the client and server DLLs that will hold the components.

Define the Client DLL

Add the PBClient DLL:

1. From the Build Configuration folder's pop-up menu, click **Add Client DLL** to open the Client DLL wizard.
2. Name the DLL PBClient.
3. Click **Next** to turn to the Client Source Files page.
4. Select PFile, PFileKey, and PFileCopy (the Person client interfaces).
5. Select BFile and BFileCopy (the Beneficiary client interfaces).
6. Click **Finish**. The client DLL appears under the folder.

Define the Server DLL

Add the PBServer DLL:

1. From the Build Configuration folder's pop-up menu, click **Add Server DLL** to open the Server DLL wizard.
2. Name the DLL PBServer.
3. Click **Next** to turn to the Server Source Files page.
4. Select PFileBO, PFileDO, PFileDOImpl, and PFileMO (the Person server interfaces).
5. Select BFileBO, BFileDO, BFileDOImpl, and BFileMO (the Beneficiary server interfaces).
6. Click **Next** to turn to the Libraries to Link With page.

7. Select the PBClient library file.
8. Click **Finish**. The server DLL appears under the folder.

Build the DLLs
Build the PBClient and PBServer DLLs:

1. From the pop-up menu of the User-Defined Business Objects folder, click **Generate - All**.
2. Wait for the code generation to complete. The generated source files are placed in the project's \Working directory.
3. From the pop-up menu of the Build Configuration folder, click **Generate - All - All Targets**.
4. From the same pop-up menu, click **Build - All Targets**. The DLLs are built and placed in the project's \Working directory.

Inheritance with Views

If your persistence is provided by a single datastore that stores both pure parent objects and child objects, you can use views to select out the appropriate data from the datastore. This combines an attributes duplication approach (one persistent object per component, with the parent's attributes duplicated in the child's persistent object) with a single datastore approach (a single datastore for all components in the hierarchy). The attributes duplication approach is used for retrieving data (allowing greater precision in selection of data), and the single datastore approach is used for changing (creating, updating, or deleting) data.

To accomplish this, the table must include a mechanism for identifying which data belongs to the parent, and which data belongs to the child. Typically, this can be accomplished by identifying a unique attribute for each component. For example, Beneficiary has an attribute claimPayment, and Person does not. So if the claimPayments column contains a value, then the component must be a Beneficiary.

Using the identifying attribute, you can create selective views of the table for each component type, and then create a persistent object for each view. These are the persistent objects that will be used to retrieve data, following the same pattern as the object-oriented approach. For example, Beneficiary could have a persistent object BeneficiaryPO, which represents a view of the table where claimPayments=notNull, and Person could have PersonPO, with a view of the table where claimPayments=Null.

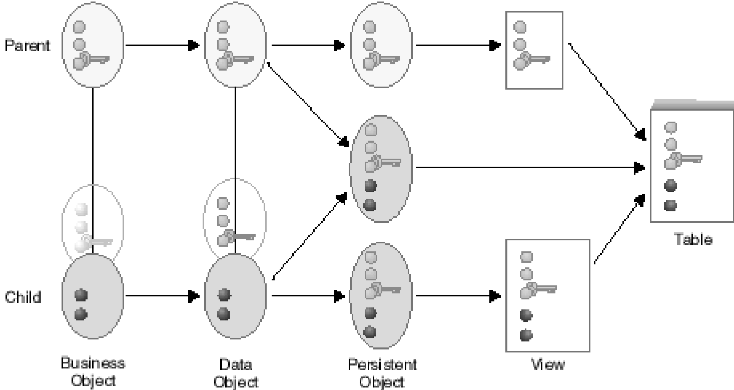
You also need to create a single persistent object that maps all the data in the table. This is the persistent object that all components will use to create, update, or delete data, following the same pattern as the shared approach. For example, both Person and Beneficiary could share the persistent object SharedPersonsPO, which represents the table directly.

Advantages
The advantage of this approach is that it takes up less space than the pure attributes duplication approach (because there is only one table for all attributes), with more precision than the single datastore approach (which cannot easily distinguish between pure parent data and inherited parent data).

Disadvantages
It is neither as efficient as the pure attributes duplication pattern, nor as fast as the

single datastore pattern. Also, like the single datastore pattern, the views pattern is problematic if your parent and child use different keys, because then the shared table would need to have two primary keys at once. Generally, you should only use this pattern when the parent and the child use the same key.

Single Datastore with Views Pattern



In this pattern:

- The parent's data object attributes are mapped to first the shared persistent object and then the parent's persistent object.
- The parent's retrieve method is mapped to the parent's persistent object.
- The parent's include, update, and delete methods are mapped to the shared persistent object.
- The parent's setConnection method is mapped to first the shared persistent object and then the parent's persistent object.
- The child's data object attributes and its inherited attributes are mapped to first the shared persistent object and the child's persistent object.
- The child's retrieve method maps to first the child's persistent object and then the parent's persistent object, with the **Always complete calling sequence** option **not** checked.
- The child's insert, update, and delete methods are mapped to the the shared persistent object.
- The child's setConnection method is mapped to first the shared persistent object and then the child's persistent object.

This mapping creates a new parent along with the child, and deletes the parent along with the child.

If you wanted to create a new child from an existing parent, you could find the existing parent, create a copy of its attribute values, delete the parent, and then create the child as a new object with the values of the deleted parent.

If you wanted to delete the child and leave the parent entry, you could copy the existing parent values, continue with the deletion of the child, and then re-create the parent with the copied values.

RELATED CONCEPTS

"Inheritance" on page 137
"Choosing an Inheritance Pattern for Persistence" on page 140

RELATED TASKS
"Create a Child Component" on page 136
"Inheritance with Views - Scenario" on page 165

Define a Child with Views

This task covers the main steps necessary to create a component that inherits from another component already defined in Object Builder, shares the same database table as its parent, and uses views to select the appropriate data out of the table. It does not cover every step; you should first be familiar with the tasks necessary to create a component without inheritance.

This approach is a variant of the single datastore pattern. To use this pattern, the child must have the same key attributes as the parent. If the child has a different key, use the attributes duplication pattern.

Typically you would use this pattern after importing the data in a single large datastore into Object Builder, as part of a strategy to break up the data among several components in a class hierarchy.

The parent maps to its data as follows:
- A single database table stores all the attributes for both parent and child.
- A shared persistent object maps all of the attributes in the table.
- A view of the database selects out those rows in which a unique child attribute is null (that is, the rows that do not contain data for a child component).
- A persistent object based on the view provides persistence for the parent's attributes.
- The parent's data object attributes, and its retrieve method, are mapped to the parent's persistent object.
- The parent's include, update, delete, and setConnection methods are mapped to the shared persistent object.

To create the child component in Object Builder, follow these steps:
1. Create a view of the shared table, selecting out those rows in which a unique child attribute is not null (that is, the rows that contain data for a child component).
2. Create a persistent object based on that view.
3. Create the business object file.
4. Add the business object interface, and select the parent's business object interface on the Interface Inheritance Page.
5. Add the copy helper. You can include attributes of the parent's copy helper either by selecting specific attributes on the Name and Attributes Page, or include all the parent's attributes by selecting the parent copy helper on the Implementation Inheritance Page. Do **not** do both.
6. Add the business object implementation:
   a. Under Data Object Interface, click **Add or select one later**. This allows you to add the data object interface in a separate step, and define its parent.
   b. Select the parent's business object implementation on the Implementation Inheritance Page.
   c. Select the parent's key on the Key and Copy Helper page.

d. Do **not** override any attributes on the Attributes to Override page.
e. Select any methods you want to override on the Methods to Override Page.

7. Add the managed object, and select the parent's managed object on the Implementation Inheritance Page.
8. Add the data object interface:
   a. From the business object implementation's pop-up menu, click **Add New Data Object Interface**.
   b. Select the attributes and methods of the business object you want represented in the data object.
   c. You should select the parent data object interface on the Interface Inheritance page.
9. Add the data object implementation, and select the parent data object implementation on the Implementation Inheritance page.
10. Map the data object implementation to the shared persistent object and the view-based persistent object, as follows:
    - The child's data object attributes and its inherited attributes are mapped to the child's persistent object.
    - the child's retrieve method maps to first the child's persistent object and then the parent's persistent object, with the **Always complete calling sequence** option **not** checked.
    - The child's insert, update, delete, and setConnection methods are mapped to the the shared persistent object.

      This always creates a new parent along with the child, and deletes the parent along with the child.

      If you wanted to create a new child from an existing parent, you could still find the existing parent, create a copy of its attribute values, delete the parent, and then create the child as a new object with the values of the deleted parent.

      If you wanted to delete the child and leave the parent entry, you could still copy the existing parent values, continue with the deletion of the child, and then re-create the parent with the copied values.

RELATED CONCEPTS
"Inheritance" on page 137
"Choosing an Inheritance Pattern for Persistence" on page 140
"Inheritance with Views" on page 162
"Components" on page 15

RELATED TASKS
Create a Component - Overview
"Create a Child Component" on page 136
Build a Child Component
"Inheritance with Views - Scenario"

Inheritance with Views - Scenario

In this scenario you define a parent component and child component that share the same database table, but map to it selectively using component-specific views. This inheritance pattern makes the most sense when parent and child share the same key. For a scenario where parent and child have different keys, see the Inheritance with Attributes Duplication - Scenario.

Before following these instructions, you should have the Person component defined and exported in XML format (PFile.xml), as described in the Simple Database Persistence - Scenario.

After you complete this scenario, you will have a component named Beneficiary that inherits from Person, a single database table that provides persistence for both Beneficiary's attributes and Person's attributes, and views of the table that provide component-specific schemas.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizard. If you are experiencing problems, click the Help button within a wizard, or go to the Help menu in Object Builder.

Create the Project

Create a sample project to hold your work.

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory (for example, e:\scenarios\inheritvw).
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Import PFile.xml

Import the definition of the Person component, as created in the Database Persistence - Scenario:

1. From the User-Defined Business Objects folder's pop-up menu, click **Import - XML**.
2. Find and select PFile.xml.
3. Click **Finish**.

The component objects for Person appear in the folder.

Create the Shared Table and Persistent Object

You need to create a schema that contains columns for all of Person's and Beneficiary's attributes.

1. Create a file with the following contents. If you are viewing this online, you can cut and paste these lines directly into an editor:
   ```
   CREATE TABLE Shared
   (
   ssNo VARCHAR(20) NOT NULL ,
   name VARCHAR(100) NOT NULL ,
   street LONG VARCHAR ,
   town LONG VARCHAR ,
   claimPayments DOUBLE
   , PRIMARY KEY
   ( ssNo, name )
   );
   ```
2. From the pop-up menu of the DBA-Defined Schemas folder in Object Builder, click **Import - SQL**.
3. Select the file you created.
4. Name the database SharedDB (or provide the name of your own database).
5. Name the group MyGroup.
6. Click **Finish**.

   The schema appears in the folder, under the schema group.
7. From the pop-up menu of the schema, click **Add Persistent Object**.

8. Name the persistent object SharedPO, and name its package file SharedPkg.
9. Set its type of persistence to **Embedded SQL**, to match the type of persistence set in PersonDOImpl.
10. Click **Finish**.

The persistent object appears under the schema.

Create the View for the Parent
Create SharedDB.PView:

1. From the pop-up menu of MyGroup, click **Add SQL View**. The SQL View Editor opens.
2. Name the view PView.
3. Click on the View Work Area tab.
4. Click on the Shared table in the Schemas pane.
5. In the Clauses pane, click the Selected Columns tab.
6. In the Columns pane, click on the columns you want represented in the view:
   - ssNo
   - name
   - street
   - town

   Their data appears in the fields of the Selected Columns page.
7. In the Clauses pane, click the Where tab.
8. In the Columns pane, click on claimPayments. Its data appears in the fields of the Where page.

   This is the column you are using to test whether the row contains data for the parent component, and to exclude rows that are for child components.
9. Click the list button of the Conditions field on the Where page, and select **Is NULL**.

   This ensures that only rows without claimPayments information appear in the view. Because the claimPayments column only contains information for Beneficiary components, this excludes child data from the view.

   If Person had additional child components, you could add additional **Is NULL** conditions, based on their unique attributes, to exclude them from the parent's view.
10. Click on the View Summary tab.
11. Review the SQL clauses that define the view, based on your selections on the previous page.
12. Click **OK**.

The view appears under MyGroup, in the DBA-Defined Schemas folder.

Create the Parent's Persistent Object
Re-create PersonPO, based on the new view of the shared table:

1. Delete PersonPO from under PersonDOImpl.
2. Delete Person's old schema from the DBA-Defined Schemas folder.
3. Delete PGroup from the DBA-Defined Schemas folder.
4. From the pop-up menu of SharedDB.PView, click **Add Persistent Object**.
5. Name the persistent object PersonPO, and name its package file PersonPkg.

6. Set its type of persistence to **Embedded SQL**, to match the type of persistence set in PersonDOImpl.
7. Click **Finish**.

The persistent object appears under the schema.

Map the Parent's Data Object and Persistent Objects

Map PersonDOImpl to SharedPO and PersonPO:

1. From the pop-up menu of PersonDOImpl, click **Properties** to open the Data Object Implementation wizard.
2. Click the title bar and turn to the Associated Persistent Objects page.
3. Add SharedPO as an associated persistent object, with the instance name iSharedPO.
4. Add PersonPO as an associated persistent object, with the instance name iPersonPO.
5. Click **Next** to turn to the Attributes Mapping page.
6. Map each attribute of Person to first its equivalent in iSharedPO, and then its equivalent in iPersonPO.
7. Click **Next** to turn to the Methods Mapping page.
8. Map Person's retrieve method to iPersonPO.retrieve.
9. Map Person's insert, update, and delete methods to iSharedPO.insert, iSharedPO.update, iSharedPO.delete, and iSharedPO.setConnection.
10. Map Person's setConnection method to first iSharedPO.setConnection, and then iPersonPO.setConnection.
11. Click **Finish**.

PersonPO and SharedPO now appear under PersonDOImpl.

You can now define the child component.

Create the View for the Child

Create SharedDB.BView:

1. From the pop-up menu of MyGroup, click **Add SQL View**. The View Editor opens.
2. Name the view BView.
3. Click on the View Work Area tab.
4. Click on the Shared table in the Schemas pane.
5. In the Clauses pane, click the Selected Columns tab.
6. In the Columns pane, click on the columns you want represented in the view:
   - ssNo
   - name
   - street
   - town
   - claimPayments

   Their data appears in the fields of the Selected Columns page.
7. In the Clauses pane, click the Where tab.
8. In the Columns pane, click on claimPayments. Its data appears in the fields of the Where page.

This is the column you are using to test whether the row contains data for the child component, and to exclude rows that are for parent components.

9. Click the list button of the Conditions field on the Where page, and select **Is Not NULL**.

   This ensures that only rows with claimPayments information appear in the view. Because the claimPayments column only contains information for Beneficiary components, this excludes data of pure parent components from the view.
10. Click on the View Summary tab.
11. Review the SQL clauses that define the view, based on your selections on the previous page.
12. Click **OK**.

The view appears under MyGroup, in the DBA-Defined Schemas folder.

Create the Child's Persistent Object
Create BeneficiaryPO, based on the new view of the shared table:

1. From the pop-up menu of SharedDB.BView, click **Add Persistent Object**.
2. Name the persistent object BeneficiaryPO, and name its package file BenPkg.
3. Set its type of persistence to **Embedded SQL**.
4. Click **Finish**.

The persistent object appears under the schema.

Create the Child's Business Object Interface
Define the Beneficiary interface:

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file BFile.
3. Click **Finish**. The file now appears under the folder.
4. From the file's pop-up menu, click **Add Module** to open the Business Object Module wizard.
5. Name the module BModule.
6. Click **Finish**. The module now appears under the file.
7. From the module's pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
8. Name the interface Beneficiary.
9. Click the title bar and turn to the Interface Inheritance page.
10. Add Person as a parent (replacing the default inheritance).
11. Click the title bar and turn to the Attributes page.
12. Add the following attribute:
    - float claimPayments
13. Click **Finish**. The interface now appears under the module.

Add the Child's Copy Helper
Add BeneficiaryCopy:

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Add all attributes to the copy helper (both parent's and child's).

3. Click **Finish**. The copy helper now appears under the interface.

Add the Child's Business Object Implementation

Add BeneficiaryBO:

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Set **Data Object Interface - Select or add one later** (you will create a new data object as a separate step).
3. Click **Next** to turn to the Implementation Inheritance page.
4. Add PersonBO as a parent.
5. Click the title bar and turn to the Key and Copy Helper page.
6. Select PersonKey and BeneficiaryCopy.
7. Click **Finish**. The business object implementation appears under the business object interface.

Add the Child's Data Object Interface

Add BeneficiaryDO:

1. From the business object implementation's pop-up menu, click **Add New Data Object Interface** to open the Data Object Interface wizard.
2. Select all the business object attributes as state data (to be preserved in the data object).
3. Click the title bar and turn to the Interface Inheritance page.
4. Add PersonDO as a parent.
5. Click **Finish**. The data object interface appears under the business object implementation.

Add the Child's Data Object Implementation, and Map It to Persistent Objects

Add BeneficiaryDOImpl, and map it to SharedPO:

1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. Set the following patterns:
   - **Environment - BOIM with any key**
   - **Form of Persistent Behavior and Implementation - Embedded SQL**
   - **Data Access Pattern - Delegating**
3. Click **Next** to turn to the Implementation Inheritance page.
4. Add PersonDOImpl as a parent.
5. Click the title bar and turn to the Key and Copy Helper page.
6. Select PersonKey and BeneficiaryCopy.
7. Click the title bar and turn to the Associated Persistent Objects page.

   SharedPO is already listed as an associated persistent object, because it is associated with Beneficiary's parent.
8. Add BeneficiaryPO as an associated persistent object, with the instance name iBeneficiaryPO.
9. Click **Next** to turn to the Attributes Mapping page.
10. Map Beneficiary's claimPayments attribute to first iSharedPO.claimPayments, and then iBeneficiary.claimPayments.
11. Map Beneficiary's inherited attributes to first iSharedPO and then iBeneficiaryPO.

For example, Person.ssNo maps to first iSharedPO.ssNo and then iBeneficiaryPO.ssNo

12. Click **Finish**.

Map the Special Framework Methods

Map the way in which the data object's special framework methods will call the persistent objects' special framework methods:

1. From BeneficiaryDOImpl's pop-up menu, click **Properties** to open the Data Object Implementation wizard.
2. Click the title bar and turn to the Methods Mapping page.
3. Map Beneficiary's retrieve method first to iBeneficiaryPO.retrieve, and then to PersonPO.retrieve, and make sure the **Always complete calling sequence** option is **not** checked.
4. Map Beneficiary's insert, update, and delete methods to iSharedPO.insert, iSharedPO.update, iSharedPO.delete.
5. Map Beneficiary's setConnection method to first iSharedPO.setConnection and then iBeneficiaryPO.setConnection.

   This creates a new parent along with the child, and deletes the parent along with the child.

   If you wanted to create a new child from an existing parent, you could find the existing parent, create a copy of its attribute values, delete the parent, and then create the child as a new object with the values of the deleted parent.

   If you wanted to delete the child and leave the parent entry, you could copy the existing parent values, continue with the deletion of the child, and then re-create the parent with the copied values.
6. Click **Finish**.

Configure the Build

You have now completed the definition of the Beneficiary component, and its inheritance from Person. The next step is to configure the client and server DLLs that will hold the components.

Define the Client DLL

Add the PBClient DLL:

1. From the Build Configuration folder's pop-up menu, click **Add Client DLL** to open the Client DLL wizard.
2. Name the DLL PBClient.
3. Click **Next** to turn to the Client Source Files page.
4. Select PFile, PFileKey, and PFileCopy (the Person client interfaces).
5. Select BFile and BFileCopy (the Beneficiary client interfaces).
6. Click **Finish**. The client DLL appears under the folder.

Define the Server DLL

Add the PBServer DLL:

1. From the Build Configuration folder's pop-up menu, click **Add Server DLL** to open the Server DLL wizard.
2. Name the DLL PBServer.
3. Click **Next** to turn to the Server Source Files page.
4. Select PFileBO, PFileDO, PFileDOImpl, and PFileMO (the Person server interfaces).

5. Select BFileBO, BFileDO, BFileDOImpl, and BFileMO (the Beneficiary server interfaces).
6. Click **Next** to turn to the Libraries to Link With page.
7. Select the PBClient library file.
8. Click **Finish**. The server DLL appears under the folder.

Build the DLLs
Build the PBClient and PBServer DLLs:
1. From the pop-up menu of the User-Defined Business Objects folder, click **Generate - All**.
2. Wait for the code generation to complete. The generated source files are placed in the project's \Working directory.
3. From the pop-up menu of the Build Configuration folder, click **Generate - All - All Targets**.
4. From the same pop-up menu, click **Build - All Targets**. The DLLs are built and placed in the project's \Working directory.

Create a Composite Component - Overview

A composite component provides access to the methods and data of its member components. The member components provide their own persistence for the data the composite accesses. The composite can also define its own original methods and data, and provide persistence for its key attributes and original data.

To create a composite component, you need to:
1. Group components into a composition.
2. Create a composite business object based on the composition.
3. Create a composite key for the component.
4. Complete the rest of the component.

The steps for creating a composite component are as follows:
1. "Create a Composition File" on page 349
2. "Add a Composition Module" on page 349
3. "Add a Composition" on page 350
4. "Create a Business Object File" on page 282
5. "Add a Business Object Module" on page 282
6. "Add a Composite Business Object Interface" on page 354
7. "Add a Composite Key" on page 360
8. "Add a Copy Helper" on page 294
9. "Add a Composite Business Object Implementation and Data Object Interface" on page 355
10. "Add a Data Object Implementation" on page 299
11. "Add a Managed Object" on page 340

RELATED CONCEPTS
"Composite Component" on page 173

RELATED TASKS
"Composite Component Creation - Scenario" on page 177
Create a Component - Overview
"Work with Compositions - Overview" on page 348
"Work with Composite Business Objects - Overview" on page 353
"Work with Composite Keys - Overview" on page 360

Composite Component

A composite component is an access point to the data and behavior of one or more other components, which the composite component's implementation delegates to. Typically, the other components are not directly accessible (in other words, the client cannot use the composite component to get a reference to one of the combined component instances); only specific data and behavior of the other components are available through the composite component's delegation of attribute and method calls. The composite component may have its own data and methods as well.

There are two kinds of composite component, based on the way its references to its constituent components are combined:

- Conjunction composite
  All of the composite component's references exist at once. In other words, at run-time the composite component has references to instances of each of its constituent components. All of the instances exist at the same time, and the composite combines their interfaces to provide a single access point to their data and behavior. This is the most common type of composite component.
- Disjunction composite
  Only one of the composite component's references exists at run-time. In other words, at run-time the composite component has a reference to an instance of only one of its constituent components. The composite component acts as a common interface for two or more mutually exclusive kinds of component, the choice of which is made when the composite component is created.

When you create a composite component, you start by defining the constituent components (in any of the standard ways, for example as a component for new DB data, legacy DB data, or PA data). Then you define the way in which the constituent components are combined in a composition object, and finally you create a new composite component based on that composition.

A composite component consists of the same objects as a normal component, with some differences to provide the compositing behavior:

- A (composite) business object, which is based on the composition.
- A (composite) key for the business object
- A data object, that stores the key attributes for the component, along with any attributes that are unique to this component (not derived from the constituent components).
- DB persistent object and DB schema (optional), that store the values of the key attributes, and the value of any attributes that are unique to the component.
- A copy helper and managed object.

RELATED CONCEPTS
"Composition" on page 174

“Composite Business Object” on page 175
“Composite Key” on page 176
“Components” on page 15

RELATED TASKS
“Composite Component Creation - Scenario” on page 177
“Create a Composite Component - Overview” on page 172

Composition

A composition defines a combined interface for a group of components. In addition, it describes the implementation of the attributes and methods in the combined interface, which delegate to attributes and methods of the components in the group. For example, we might define a composition, CompositeAccount, that combines two components, SavingsAccount and CheckingAccount. The CompositeAccount interface might include an attribute balance that is defined as the sum of a balance attribute on the SavingsAccount component and a balance attribute on the CheckingAccount component.

Once you have defined the composition, you can create composite business objects that are based on the composition.

A composition does not have its own managed object; it is only accessible as part of the business logic of a composite component based on the composition. It is an abstraction of the combining and delegating logic needed to access the data and behavior of the components being combined. This logic is implemented in a local-only helper object, for use by the composite business object that is based on the composition.

When you package a composite component, be sure to include the source files for the composition class in the component’s server DLL or shared library file. Otherwise, the composition logic contained in the helper object will not be available to the composite component.

You can create compositions under the User-Defined Compositions folder, in Object Builder’s Tasks and Objects pane. For each component that you add to a composition, the composition has:

- A managed object instance, of the same type as the component’s managed object
- Attributes that delegate to the component attributes.
- Methods that delegate to the component methods.

You can edit which attributes and methods are included, and what they delegate to. You can also define attributes and methods that contain logic or data that is unique to the composition, and does not simply delegate to a combined component. This is useful for adding private helper functions to hold user-defined logic. For example, a composition AllAccounts, which combines the components CheckingAccount and SavingsAccount, could have a private helper method addFloats, which can take the two original balances (CheckingAccount1.balance and SavingsAccount1.balance) as arguments, and return their sum. You can then map AllAccounts.balance to the helper method. When you add a new method, you can supply its implementation (for example, return arg1+arg2) in Object Builder’s Source pane (after you complete the composition, click on it in the Tasks and Objects pane; then select the method in the Methods pane, and complete its implementation in the Source pane).

RELATED CONCEPTS
“Composite Component” on page 173
“Composite Business Object”

RELATED TASKS
“Composite Component Creation - Scenario” on page 177
“Create a Composite Component - Overview” on page 172
“Work with Compositions - Overview” on page 348

Composite Business Object

A composite business object is part of a composite component. The business object is based on a composition, which defines the interface to one or more combined components.

When you base a business object on a composition, the business object automatically gets the attributes and methods defined in the composition (except for the composition’s references to its constituent components). The business object attributes and methods have implementations that delegate to their equivalents in the composition helper object. As with any other business object, you can also define other attributes and methods that are unique to the composite component, and do not delegate to a composition. You can make these attributes persistent through a DB schema.

The composition has a component instance for each component it composites. It does not, however, deal with managed object configuration issues such as how and when to find or create these instances. This information is instead provided in the composite business object. This allows you to re-use the pure combining logic of the composition in multiple versions of a composite component, each version providing different managed object configuration information. You provide the information for finding and creating the managed object instances in the composite business object implementation. The instances are then used by the business object, in conjunction with the logic in the composition helper, to delegate its attribute and method calls appropriately.

Each composite business object must have a composite key, in which the key attributes of the composite business object can be mapped to key attributes of the combined components. If the attributes have a simple mapping, you can define the mapping in the Key wizard and have the appropriate logic generated by Object Builder. If you require a more complex mapping, you can edit the provided mapping methods (for example, get_SavingsAccount_accountNo) and provide your own implementations.

You can use the composite component’s data object to store a secondary source for an attribute. If a delegating call to an attribute fails (for example, because the combined component that provides it is unavailable), the composite component will return the value in the data object instead of fail. This is particularly useful for composite components that use the disjunction pattern. In the disjunction pattern, only one of the combined component instances is available at run-time, which means that any unique attributes of the other combined components are unavailable. The data object can provide a secondary source for these unique attributes, which is used when the current component instance does not provide them.

RELATED CONCEPTS
“Composite Component” on page 173

"Composition" on page 174
"Composite Key"
"Business Object" on page 17

RELATED TASKS

"Composite Component Creation - Scenario" on page 177
"Create a Composite Component - Overview" on page 172
"Work with Composite Business Objects - Overview" on page 353

Composite Key

A composite key is the key object for a composite component. As with a regular key, the composite key defines attributes of its component that are to be used to find a particular instance of the component on the server. The key consists of one or more of the business object attributes, which must contain enough information to uniquely identify an instance. For a composite key, these business object attributes may optionally be used to identify the components that make up the composition.

A common pattern for locating the contributing components of a composition is to make the identity of the composite component the union of the identities of the contributing components. In other words, the composite key attributes are equivalent to the various key attributes of the components in the composition.

For example:

- A composite component AllAccounts is based on the composition AccountComposition, that combines two other components, SavingsAccount and CheckingAccount.
- The key attribute for SavingsAccount is accountNo.
- The key attribute for CheckingAccount is accountNo.
- The key attributes for AllAccounts are savingsAccountNo and checkingAccountNo, each of which is mapped to its equivalent accountNo attribute in SavingsAccount and CheckingAccount.

The composite key contains enough information to uniquely identify the AllAccounts component, and also to locate the equivalent SavingsAccount and CheckingAccount components. There is no need to maintain persistent references from the composite component to its constituent components; if you can find AllAccounts, you have enough information to find SavingsAccount and CheckingAccount.

When you use this pattern (the identity of the composite component as the union of the identities of its constituent components), you can provide a mapping between the attributes of the composite key and the attributes of keys for the combined components. You can define simple mappings between the two sets of attributes in the composite key's Key wizard.

For example, given the following objects and key attributes:

- AllAccountsKey is the composite key for AllAccounts, and has two key attributes:
  - savingsAccountNo
  - checkingAccountNo
- AccountComposition is the composition on which AllAccounts is based, and combines two components:
  - SavingsAccount, with the key attribute accountNo, defined in the key object SavingsAccountKey

– CheckingAccount, with the key attribute accountNo, defined in the key object CheckingAccountKey

The attributes in the composite key AllAccountsKey would be mapped as follows:

- savingsAccountNo maps to accountNo in SavingsAccountKey
- checkingAccountNo maps to accountNo in CheckingAccountKey

For simple mappings such as this one (where the attributes are of the same type, and the mapping is one-to-one), the mapping information will be used to generate implementations of the get_ methods (for example, get_SavingsAccount1_accountNo) in the composite business object implementation. If a mapping is complex or not provided at all, then you need to provide your own implementation for these methods.

RELATED CONCEPTS

"Composite Component" on page 173
"Composition" on page 174
"Composite Business Object" on page 175
"Key" on page 21

RELATED TASKS

"Composite Component Creation - Scenario"
"Create a Composite Component - Overview" on page 172
"Work with Composite Keys - Overview" on page 360
"Edit Get and Set Methods" on page 270

Composite Component Creation - Scenario

This scenario provides instructions for creating a composite component, that consolidates the interfaces of two other components.

After you complete this scenario, you will have two ordinary components, SavingsAccount and CheckingAccount, and a composite component, AllAcounts, that provides access to the data in SavingsAccount and CheckingAccount through a single combined interface.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizards. If you are experiencing problems, click the Help button within a wizard, or go to the Help pulldown in Object Builder.

Create the Project
Create a sample project to hold your work.

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory (for example, e:\scenarios\composite).
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Create the SavingsAccount Component
Create a simple component representing a savings account at a bank. For the sake of simplicity, you will be accepting the default for most of the object settings, and using transient data (no persistent objects or schemas).

Define the SavingsAccount interface:

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file SAFile.
3. Click **Finish**. The file now appears under the folder.
4. From the file's pop-up menu, click **Add Module** to open the Business Object Module wizard.
5. Name the module SAModule.
6. Click **Finish**. The module now appears under the file.
7. From the module's pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
8. Name the interface SavingsAccount.
9. Click the title bar and turn to the Attributes page.
10. Add the following attributes:
    - readonly long accountNo
    - readonly float balance
11. Click **Next** to turn to the Methods page.
12. Add the following methods:
    - void credit (in float amount)
    - void debit (in float amount)
13. Click **Finish**. The interface now appears under the module.

Add a key:

1. From the interface's pop-up menu, click **Add Key** to open the Key wizard.
2. Select accountNo as the key attribute.
3. Click **Finish**. The key now appears under the interface.

Add a copy helper:

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Select all the attributes to be part of the copy helper.
3. Click **Finish**. The copy helper now appears under the interface.

Add a business object implementation and data object interface:

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Click the title bar and turn to the Key and Copy Helper page.
3. Select SavingsAccountKey and SavingsAccountCopy.
4. Click the title bar and turn to the Data Object Interface page.
5. Select all attributes as state data (to be preserved in the data object).
6. Click **Finish**. The business object implementation appears under the business object interface, and the data object interface appears under the implementation.

Add a data object implementation:

1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.

2. For the sake of simplicity, set the environment to **BOIM with any key** and the form of persistence to **Transient**. This saves you the step of defining the database or procedural adaptor that would normally provide persistence for the data.
3. Click the title bar and turn to the Key and Copy Helper page.
4. Select SavingsAccountKey and SavingsAccountCopy.
5. Click **Finish**. The data object implementation appears under the data object interface.

Add a managed object:
1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard.
2. Click **Finish**. The managed object now appears under the business object implementation.

Create the CheckingAccount Component
Create another simple component, in the same way, that represents a Checking account. The steps are substantially the same as for the previous task, so only the differences are noted here.
1. Add the CAFile file and CAModule module.
2. Add the CheckingAccount interface, with the following attributes and methods:
   - readonly long accountNo
   - readonly float balance
   - long checkCount
   - void credit (in float amount)
   - void debit (in float amount)
3. Add a key, with accountNo as the key attribute.
4. Add a copy helper, with all attributes selected.
5. Add a business object implementation and data object interface, with all attributes represented in the data object interface, CheckingAccountKey selected as the key, and CheckingAccountCopy selected as the copy helper.
6. Add a data object implementation with the **BOIM with any key** setting, transient data, and CheckingAccountKey and CheckingAccountCopy selected as the key and copy helper.
7. Add a managed object.

Create the Composition
The composition defines the combined interface and delegating implementation for the composite component.

Add a file:
1. From the User-Defined Compositions folder's pop-up menu, click **Add File** to open the Composition File wizard.
2. Name the file ACFile.
3. Click **Finish**. The file appears under the folder.

Add a module:
1. From the file's pop-up menu, click **Add Module** to open the Composition Module wizard.
2. Name the module ACModule.

3. Click **Finish**. The module appears under the file.

Add the composition:

1. From the module's pop-up menu, click **Add Composition** to open the Composition Editor.
2. Click **Add** to display the Composition Palette.
3. Select SavingsAccountMO and CheckingAccountMO.
4. Click **Add** to add them to the **Objects to Composite** list.
5. Click **Close** to close the palette.
6. In the **Objects to Composite** list, you can see entries for both SavingsAccount1 and CheckingAccount1 (the default names for the SavingsAccountMO and CheckingAccountMO instances the composition will hold). Under each instance entry you can see its attributes and methods.

   Above the list, you can see the **Composition Style** that is being applied to the selected objects to produce the resulting composition in the **Results** list.
7. Try selecting some other composition styles, and review the results.
8. Return to the **Conjunction without name matching** style.
9. In this style, attributes with conflicting names (such as accountNo and balance) are made unique by combining them with their instance names (for example, SavingsAccount1_accountNo). Attributes that are already unique (such as checkCount) are not renamed.
10. Click on the checkCount attribute to see the the delegating implementation of its getter method in the Current Republished Value pane.
11. Click **Setter** to see its setter method's implementation.
12. You can use the pop-up menu of the current value to remap the method to another value. For this exercise, simply accept the defaults.
13. Double-click on the checkCount attribute to see its properties. You can change the name of the attribute, but you cannot change its implementation details.
14. In the **Results** pane, click on the parent folder (named **Untitled** by default). This folder represents the composition itself.
15. Click the **Properties** tab to display the properties of the composition.
16. Name the composition AccountComposition.
17. Click **OK**. The composition appears under the module.

The composition is a complete implementation object. You can generate its IDL and C++ code by selecting **Generate - All** from its pop-up menu.

Click on the composition in the Tasks and Objects pane to review its attributes and methods in the Methods pane. Note that the managed object instances appear as attributes under the User-Defined Attributes folder.

Add the Composite Component AllAccounts

Now that you have the composition, you can create a composite component based on the composition. This is similar to the procedure for creating a normal component, and only the differences are noted here.

Add an AAFile file and AAModule module, and then add the AllAccounts interface:

1. From the modules' pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
2. Name the interface AllAccounts.
3. Check the **Composite** choice.

4. From the **Composition to Use** list, select AccountComposition.
5. Click the title bar and turn to the Attributes page.
6. Review the list of attributes the component has received from its base.
7. You can delete or rename these attributes, and create new ones that are specific to the component (rather than taken from the composition). For this exercise, accept the default.
8. Click the title bar and turn to the Methods page.
9. Review the list of methods the component has received from its base.
10. You can delete or rename these methods, and create new ones that are specific to the component (rather than taken from the composition). For this exercise, accept the default.
11. Click **Finish**. The interface appears under the module.

Add a composite key:
1. From the interface's pop-up, click **Add Key** to open the Key wizard.
2. Select SavingsAccount1_accountNo and CheckingAccount1_accountNo as key attributes.
3. Click **Next** to turn to the Composite Key page. This page is added to the wizard for keys of composite components.
4. On this page, you map the selected attributes of the composite component back to the original attributes in the keys of the grouped components.
5. In the Composite Key list (on the right), click SavingsAccount1_accountNo to select it.
6. In the Composite Key Element list (on the left), click the accountNo attribute of the SavingsAccountKey object.
7. Click **Add**. The scoped attribute is added to the Composite Key list, under the selected attribute.
8. Perform the same mapping for CheckingAccount1_accountNo to CheckingAccountKey::accountNo.
9. Click **Finish**. The key appears under the interface.

Add a copy helper:
1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Select all attributes to be part of the copy helper.
3. Click **Finish**. The copy helper appears under the interface.

Add a business object implementation and data object interface:
1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Select **Caching** as the pattern for handling state data.
3. Click the title bar and turn to the Key page.
4. Select AllAccountsKey.
5. Click the title bar and turn to the Location page. This page is added to the wizard for business object implementations of composite components.
6. On this page, you define the component's relationship to the composited managed object instances (SavingsAccountMO and CheckingAccountMO), and provide information about the managed objects' locations.

7. Accept the default settings for both components (**Remove the instance when the composition is destroyed** is not checked, **Add the instance to the composition by: Find or create** is selected, and **Create the instance using its copy helper** is not checked).
8. The component will **not** destroy the composition's instances when the composition is destroyed. Any attempt to access a managed object instance will be resolved by finding it, if it exists, or creating it, if it doesn't. The instance will be created using its primary key (i.e., not its copy helper).
9. Accept the default pattern for locating the home (Factory Finder / Principal).
10. Accept the default location for the home (**Factory Finder Name**).
11. This information should match the **Name in Factory Finding Service Registry** for the managed object's home, in the application configuration information for the managed object (in the Managed Object Configuration wizard, Home page). Because the managed objects are not yet configured, you should accept the default for now.
12. Accept the default principal interface name.
13. Click the title bar and turn to the Data Object Interface page.
14. Add the key attributes (SavingsAccount1_accountNo and CheckingAccount1_accountNo) to the data object.
15. Click **Finish**. The business object implementation and data object interface appear under the business object interface.

Add a transient data object:

1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. Select **BOIM with any key** as the environment.
3. Select **Transient** as the form of persistence.
4. Click the title bar and turn to the Key and Copy Helper page.
5. Select AllAcountsKey and AllAccountsCopy.
6. Click **Finish**. The data object implementation appears under the data object interface.

Add a managed object:

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard.
2. Click **Finish**. The managed object appears under the business object implementation.

The composite component is now defined.

Edit the Composition

When you edit attributes or methods in the composition, the changes are automatically applied to the composite components based on the composition. In this task, you will consolidate the attributes SavingsAccount1_balance and CheckingAccount1_balance into a single balance attribute, that returns the sum of the two component's balances.

Consolidate the two balance attributes:

1. Locate the AccountComposition composition, in the User-Defined Compositions folder.

2. From the composition’s pop-up menu, click **Properties** to open the Composition Editor.
3. In the Results pane, select the attributes SavingsAccount1_balance and CheckingAccount1_balance. Select the second attribute using the Ctrl key plus right-click, to both select it and display its pop-up menu at the same time.
4. From the pop-up menu, select **Equate**. The two attributes are consolidated into a single balance attribute.
5. Click on the new balance attribute to display its delegating behavior in the Current Republished Value pane.

   By default, the consolidated attribute delegates to a sequence of the two source attributes. This means that it will return the balance of the last attribute in the sequence. To customize the method and have it return the sum of the two attributes, instead of just the last value in the sequence, you need to add some extra processing in the form of a private helper function.

Add a private helper function:

1. In the Results pane, display the pop-up menu for the User-Defined Methods folder and click **Add**. A new method with the default name newOperation1 appears.
2. Click on the Properties tab to display the properties for the method.
3. Change its name to addFloats.
4. Change its return type to float.
5. Change its implementation to Private.
6. In the Results pane, expand the method to show the Parameters folder underneath it.
7. From the pop-up of the Parameters folder, click **Add**. A parameter with the default name newParameter1 is added to the folder. The properties of the parameter appear on the Properties page.
8. Change the parameter’s name to arg1.
9. Change the parameter’s type to float.
10. Add a second parameter named arg2, type float.

Change the delegation for balance:

1. In the Results pane, click on the balance attribute you consolidated earlier. The delegating behavior of its Getter method (to a <sequence> of SavingsAccount1.balance and CheckingAccount1.balance) appears in the Current Republished Value pane. There is no delegation for the Setter method, because the source balance attributes are read-only.
2. Delete the <sequence> node. It is replaced by an <empty> node.
3. From the pop-up menu of the <empty> node, click **Set value**. A list of attributes and methods with type or return type float appear.
4. Select addFloats as the value to map to. It replaces the <empty> node, and two parameter nodes (labelled ??) appear beneath it.
5. From the pop-up menu of the first ?? node, click **Set value**. Map the parameter to SavingsAccount1.balance. The selected attribute replaces the ?? node.
6. Map the second parameter in the same way, to CheckingAccount.balance.
7. Click **OK** to apply your changes to the composition, and return to the Object Builder main window.

Add the implementation for balance:

1. Click on the composition in the User-Defined Compositions folder. Its attributes and methods appear in the Methods pane.
2. Select the addFloats method in the Methods pane. Its skeleton implementation appears in the Source pane.
3. Add the following implementation to the Source pane:

   return arg1+arg2;

   The method delegation is complete. Calls to the combined balance attribute are automatically delegated to addFloats, which takes the two source balance attributes as parameters and returns their sum.

Review the changes in the composite component:
1. In the User-Defined Business Objects folder, locate the composite business object implementation AllAccountsBO.
2. Click on AllAccountsBO to display its attributes and methods in the Methods list.

   It now has a balance attribute, that has replaced the SavingsAccount1_balance and CheckingAccount1_balance attributes.
3. Click on the balance attribute to display its implementation, which has already been filled in with appropriate delegation behavior.
4. Click **File - Save** to save your changes.

Configure the Build
Create client and server DLLs for the components. For this exercise, all the components will be configured into the same DLLs. The AllAccounts component is configured like a normal component. The AccountComposition composition is a server-only object.

If the AllAccounts component were configured into a separate DLL from the SavingsAccount and CheckingAccount components, then the AllAccounts DLLs would need to link with the other component's libraries (on the Libraries to Link With page of the wizards for the DLLs). This would be necessary to resolve the composite component's references to the composited managed objects.

Define the client DLL:
1. From the Build Configuration folder's pop-up menu, click **Add Client DLL** to open the Client DLL wizard.
2. Name the DLL AccountsClient.
3. Click **Next** to turn to the Client Source Files page.
4. Select SAFile, SAFileKey, and SAFileCopy (the SavingsAccount client interfaces).
5. Select CAFile, CAFileKey, and CAFileCopy (the CheckingAccount client interfaces).
6. Select AAFile, AAFileKey, and AAFileCopy (the AllAccounts client interfaces).
7. Click **Finish**. The client DLL appears under the folder.

Define the server DLL:
1. From the Build Configuration folder's pop-up menu, click **Add Server DLL** to open the Server DLL wizard.
2. Name the DLL AccountsServer.
3. Click **Next** to turn to the Server Source Files page.
4. Select SAFileBO, SAFileDO, SAFileDOImpl, and SAFileMO (the SavingsAccount server interfaces).

5. Select CAFileBO, CAFileDO, CAFileDOImpl, and CAFileMO (the CheckingAccount server interfaces).
6. Select AAFileBO, AAFileDO, AAFileDOImpl, and AAFileMO (the AllAccount server interfaces).
7. Select ACFile (the AccountComposition composition).
8. Click **Next** to turn to the Libraries to Link With page.
9. Select the AccountsClient library file.
10. Click **Finish**. The server DLL appears under the folder.

Build the DLLs:

1. From the pop-up menu of the User-Defined Business Objects folder, click **Generate - All**.
2. Wait for the code generation to complete. The generated source files are placed in the project's \Working directory.
3. From the pop-up menu of the User-Defined Compositions folder, click **Generate - All**. The code for the composition is added to the \Working directory.
4. From the pop-up menu of the Build Configuration folder, click **Generate - All - All Targets**.
5. From the same pop-up menu, click **Build - All Targets**. The DLLs are built and placed in the project's \Working directory.

Configure the Application

Create the application family:

1. From the Application Configuration folder's pop-up menu, click **Add Application Family** to open the Application Family wizard.
2. Name the application AccountFamily.
3. Click **Finish**. The application family appears under the folder.

Create the application:

1. From the application family's pop-up menu, click **Add Application** to open the Application wizard.
2. Name the application AccountApplication.
3. Click **Finish**. The application appears under the application family.

Configure the SavingsAccount managed object:

1. From the application's pop-up menu, click **Add Managed Object** to open the Managed Object Configuration wizard.
2. Select SavingsAccountMO as the managed object. The other fields become filled in with appropriate defaults.
3. Click **Next** to turn to the Data Object Implementations page.
4. Select SavingsAccountDOImpl.
5. Click **Finish**. The correct container and home are selected by default. The managed object configuration appears under the application.

Configure the CheckingAccount managed object using the same steps, with the following differences:

1. Select CheckingAccountMO as the managed object.
2. Select CheckingAccountDOImpl as the data object implementation.

The managed object configuration appears under the application.

Configure the AllAccounts managed object using the same steps, with the following differences:

1. Select AllAccountsMO as the managed object.
2. Select AllAccountsDOImpl as the data object implementation.
3. Click the title bar and turn to the Home page.
4. Review the **Name in Factory Finding Service Registry** path, and verify that it is the same as the location you provided in the Business Object Implementation wizard, Location page (in the **Factory Finder Name** field).
5. Click **Finish**. The managed object configuration appears under the application.

Create the install image:

1. From the application family's pop-up menu, click **Generate**. The install scripts are created and placed in the project's \Working*platform*\AccountFamily\ directory.
2. From the application family's pop-up menu, click **Build**. The install image is created and placed in the project's \Working*platform*\AccountFamily\Disk1\ directory.

You have now completely defined two components, a composition that combines their interfaces, and a composite component that allows access to the combined interfaces.

Chapter 7. Multi-Platform Development

You can use Object Builder to develop components for deployment on Windows NT, AIX, or OS/390 servers. Most development options are the same for all platforms: the main differences appear when you generate the code for your components. There are three mechanisms in place for dealing with these differences: platform-filtered views, platform-targetted code generation, and platform-specific development constraints. You can also implement different versions of your method implementations for different platforms.

Views
You can select a platform view from the **Platform - View** menu in Object Builder. Inheritance options, framework methods, and framework method implementations will be filtered for the selected platform. The information for all views is stored in the same project model; you can switch between views at any time.

Code Generation
You can select platforms to generate for from the **Platform - Generate** menu in Object Builder. For each platform you select, an equivalent subdirectory will be added to the project's \Working directory. For example, if you select AIX and 390, you will have code generated to the directories <*project*>\Working\AIX and <*project*>\Working\390 . Every time you select a **Generate** option from within the Tasks and Objects pane, code will be generated for all selected platforms. The more platforms you select to generate for, the longer code generation will take.

Constraints
You can set constraints to ensure that the components you develop will be deployable on your target platforms. Select platform constraints on the **Platform - Constrain** menu in Object Builder. By default, any components you develop will be deployable on the platforms you selected. You can override these defaults on a particular object, to create a platform-specific version of the object.

You can set object-specific platform constraints on data object implementations, managed objects, managed object configurations, containers, and DLLs. You can set the constraints when you create the object, or by editing its properties. On the first page of the object's wizard, under the heading **Set Deployment Platform**, you can select a subset of the platform constraints to apply. For example, if your platform constraints are set to AIX and 390, you can select to apply only 390 constraints, to develop a 390-specific version of the object.

Methods
In the Properties wizard for a method, you can define whether its implementation in the Source pane is to be used for all platforms, or to be defined separately for each platform. Access the wizard from the Methods pane, by selecting **Properties** from a method's pop-up menu. Once you have set to use different versions, you can use the **Platform - View** menu option to choose which platform-specific implementation to display and edit in the Source pane.

RELATED TASKS
"Set Platform Constraints" on page 189
"Generate Code" on page 363
"Develop a Multi-Platform Application - Scenario" on page 190

RELATED REFERENCES
"Platform Differences" on page 188

© Copyright IBM Corp. 1997, 1998

Platform Differences

Most development options are the same for all platforms. The main differences are between OS/390 and the workstation platforms, NT and AIX.

The following differences apply between OS/390 and the workstation platforms:

- Inheritance
  Different framework inheritance may apply for the different platforms. When you generate code for multiple platforms, the right inheritance will automatically be used. When you view a specific platform, the inheritance that applies to that platform is shown. Not all inheritance options have cross-platform equivalents.

  If you are developing an OS/390 component, you cannot select the parent:

  IBOIMExtLocal IBOIMExtLocal::IUUIDCopyHelperBase
- Framework methods
  Different framework methods may apply for the different platforms. When you generate code for multiple platforms, the right framework methods will automatically be implemented. When you view a specific platform, the framework methods that apply to that platform, and the appropriate method implementations, are shown.
- Wide types
  Wide types are not available on OS/390. Do not use when you develop for OS/390. They are not available for selection if you have **390** listed as a platform constraint.
- Services
  Cache Service are not available on OS/390. The Cache Service is intended to provide increased performance for workstation servers, and is not needed for OS/390 servers.
- Sessionable managed objects
  You cannot create sessionable managed objects for OS/390.
- PA development
  Procedural Adaptor persistent objects on OS/390 and the workstation platforms have mutually exclusive connection types. You cannot create common PA persistent objects for both OS/390 and any other platform. You must create platform-specific versions of the data object implementations and persistent objects for PA components.
- Container definition
  OS/390 servers do not use any of the container information you provide when you define a container, except for its name and description. If you are developing an OS/390-specific container, the additional pages are not available. If you are developing a container for multiple platforms including OS/390, the additional pages are available, but the information on them will be ignored by the OS/390 server.

RELATED CONCEPTS

"Chapter 7. Multi-Platform Development" on page 187

RELATED TASKS

"Set Platform Constraints" on page 189
"Generate Code" on page 363
"Develop a Multi-Platform Application - Scenario" on page 190

Set Platform Constraints

In Object Builder, you can develop components that will work on Windows NT, AIX, and OS/390. However, not all development options are available for every platform. To ensure that your components will run on the platforms you intend to deploy on, and to take advantage of all opportunities available for each platform, you can constrain your development options on both a project level and on an object level.

To ensure that objects you create will run on your deployment platforms, you can set project-wide constraints that allow access only to development options available on all your deployment platforms. To set project-wide platform constraints, follow these steps:

1. From the Object Builder menu bar, click **Platform - Constrain**.
2. From the cascade, select a platform constraint.
3. Add additional platform constraints in the same way.

Once you set these constraints, development options (such as framework inheritance and services) are filtered to ensure that the application you develop will be deployable to the platforms you select. This is a “least common denominator” approach; you may want to supplement it by developing some objects in multiple versions, to take advantage of some platform-specific options.

Within the project-wide constraints, you can develop multiple versions of an object for your different deployment platforms. For example, in a project to be deployed on AIX and OS/390, you could develop a data object implementation for AIX only, and another data object implementation for OS/390 only. This would allow you to use the Caching Service on AIX, which is unavailable for OS/390 (where the delegating pattern performs well enough to require no alternative).

You can set object-specific platform constraints on data object implementations, managed objects, managed object configurations, containers, and DLLs. You can set the constraints when you create the object, or by editing its properties. To set object-specific constraints, follow these steps:

1. Open the object’s wizard (either by creating the object, or by clicking **Properties** from its pop-up menu).

   On the first page of the wizard, the group box **Set Deployment Platform** contains checkboxes for NT, AIX, and 390. Only platforms that are listed in your project-wide constraints are available for selection.

   By default, all the platforms in the project-wide constraints are listed.
2. Deselect any platforms you are not deploying this object on. For example, if you are deploying the object for AIX only, make sure the NT and AIX checkboxes are **not** checked.

   The wizard will now allow access to all options available for the platforms you indicated. For example, if you are deploying the object for AIX only, all AIX-specific options will be available, even those not available on other platforms.
3. Complete your selections in the wizard, and click **Finish**.

To change your project-wide platform constraints, follow these steps:

1. From the Object Builder menu bar, click **Platform - Constrain**.
2. From the cascade, select or deselect a platform constraint.
3. Add or remove additional platform constraints in the same way.

The new constraints will affect the choices you have in developing new objects, but do not affect any existing objects created under different constraints. To check your application under the new constraints, run a consistency check on the project's model.

4. From the Object Builder menu bar, click **File - Check Model**.
5. Review the report, and save it if you want before closing it.
6. Edit objects as necessary to make your model consistent under the new constraints.

RELATED CONCEPTS

"Chapter 7. Multi-Platform Development" on page 187

RELATED TASKS

"Check a Model for Consistency" on page 412
"Develop a Multi-Platform Application - Scenario"

RELATED REFERENCES

"Platform Differences" on page 188

Develop a Multi-Platform Application - Scenario

Objectives
To create a component for deployment on two different platforms.
To add platform-specific method implementations.
To create platform-specific versions of a data object implementation.
To build DLLs for the different platforms.
To define containers for use on each platform.
To create application packages for each platform.

Before You Begin
You need the following installed on your system:

- CBToolkit, including Samples
- DB2 Universal Database
- VisualAge for C++ or VisualAge for Java

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide.*

You should be familiar with the steps involved in defining, building, and packaging components in Object Builder, as described in the scenario sequence starting with "Create a Component - Scenario" on page 39. For most tasks in this scenario, you will be given only general directions. For more specific instructions, you can refer to the referenced scenario or one of its sequels.

Description
This exercise defines the objects required to create a component named "Claim" for deployment on the AIX and OS/390 platforms. The component will have platform-specific versions of its data object implementation. For this exercise, you will:

1. Create the project
2. Create a business object interface
3. Add a key and copy helper

4. Add a business object implementation
5. Add a data object implementation for AIX
6. Define a data object implementation for OS/390
7. Define a persistent object and schema
8. Add a managed object
9. Generate the code
10. Define a client DLL and server DLL for AIX
11. Define a client DLL and server DLL for OS/390
12. Define an application family and application for AIX
13. Define an application family and application for OS/390
14. Configure the component with both applications

Create the Project

Create a sample project to hold your work.

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory (for example, e:\scenarios\multiplat\).
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Set platform constraints and code generation for AIX and 390:

1. Click **Platform - Constrain - AIX**
2. Click **Platform - Constrain - 390**

   By default, any objects that have platform-specific development options will only allow selection of options that exist on both AIX and OS/390. These constraints will be overridden when you create the data object implementation, to allow separate versions for each platform.
3. Click **Platform - Generate - AIX**
4. Click **Platform - Generate - 390**

   Code will be generated for both platforms, into the \Working\AIX and \Working\390 directories.
5. Click **Platform - View - AIX**

   When there are differences in an object's inheritance or framework methods for different platforms, you will see the AIX version.

Create the Business Object Interface

Define a business object file (ClaimFile):

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file ClaimFile.
3. Click **Finish**. The file now appears under the folder.

Add an interface (Claim):

1. From ClaimFile's pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
2. Name the interface Claim.
3. Click the title bar and turn to the Attributes page.
4. Add the following attributes:

- readonly long claimNo
- long state
5. Add the following methods:
   - void approve
   - void deny
6. Click **Finish**. The Claim interface now appears under the ClaimFile file.

Add a Key and Copy Helper
Add a key (ClaimKey):
1. From Claim's pop-up menu, click **Add Key** to open the Key wizard.
2. Accept the default name; select the claimNo attribute and add it to the **Key Attributes** list.
3. Click **Finish**. ClaimKey appears under Claim.

Add a copy helper (ClaimCopy):
1. From Claim's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Accept the default name; select all attributes and add them to the **Copy Helper Attributes** list.
3. Click **Finish**. ClaimCopy appears under Claim.

Add a Business Object Implementation and Data Object Interface
Add a business object implementation (ClaimBO) and data object interface (ClaimDO):
1. From Claim's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Accept the default name and behavior settings.
3. Click the title bar and turn to the Data Object Interface page.
4. Select all attributes and add them to the **State Data** list.
5. Click **Finish**. ClaimBO appears under Claim, and ClaimDO appears under ClaimBO.

Add Platform-Specific Method Implementations
For each method, you can specify whether to use a different method implementation for each platform, or share the same implementation on all platforms. By default, method implementations are shared.

Make the approve() method implementation platform-specific:
1. Click on ClaimBO in the Tasks and Objects pane. Its methods and attributes are listed in the Methods pane.
2. In the Methods pane, locate the approve() method.
3. From the pop-up method for the approve() method, click **Properties** to open the Method Implementation wizard.
4. Deselect the option **Method body is the same for all platforms**.
5. Click **Finish**.

Add an implementation for the approve() method on AIX:
1. Click on the approve() method in the Methods pane. The skeleton implementation appears in the Source pane.
2. Type the following implementation for the approve() method:

```
state(1);
```

When a claim is approved on AIX, its state changes from 0 to 1.

Add an implementation for the approve method on OS/390:

1. Click **Platform - View - 390**.
2. Click on the approve() method in the Methods pane. The method implementation you provided for AIX does not appear. Instead you see a skeleton implementation for the 390-specific version of the implementation.
3. Type the following implementation for the approve() method:

   ```
   state(2);
   ```

   When a claim is approved on OS/390, its state changes from 0 to 2.

When you generate code for the business object implementation, the code in the Working\AIX directory will use the AIX-specific implementation, and the code in the Working\390 directory will use the OS/390-specific implementation.

In most cases, you should be able to use the same implementation for all platforms. This example is intended to show the procedure, but is not intended as a model for you to follow.

Add a Shared Method Implementation

Add a shared implementation for the deny method on AIX and OS/390:

1. Click on the deny() method in the Methods pane. The skeleton implementation appears in the Source pane.
2. Type the following implementation for the approve() method:

   ```
   state(-1);
   ```

   When a claim is denied, its state changes from 0 to 1.

Because you did not change the default settings in the method's Method Implementation wizard, this implementation will apply to all platforms for which code is generated.

Add a Data Object Implementation for AIX

1. From ClaimDO's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.

   On the first page, the **Select Deployment Platforms** constraints are listed:

   - **NT** is greyed out and cannot be selected, because the project-wide platform constraints exclude it. This prevents you from creating an NT-specific object within project constraints for AIX and 390.
   - **AIX** is selected by default, based on the project-wide platform constraints.
   - **390** is selected by default. based on the project-wide platform constraints.
2. Deselect the **390** option. AIX-specific development options are now available.
3. Name the object ClaimAIXDOImpl, with the file name ClaimFileAIXDOImpl.
4. Click **Next** to turn to the Behavior page.
5. Set the following behaviors:
   - **Environment: BOIM with any key**
   - **Form of Persistent Behavior and Implementation: DB2 Cache Service**

The Cache Service is not used on OS/390. By making this object AIX-specific, it can take advantage of the Cache Service, while the 390-specific version can use the delegating pattern.

For more information on the Cache Service, see the *IBM Component Broker Advanced Programming Guide*.

- **Data Access Pattern: Delegating**

6. Click **Finish**. ClaimAIXDOImpl appears under ClaimDO.

Add a Data Object Implementation for OS/390

1. From ClaimDO's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.

   On the first page, the **Select Deployment Platforms** constraints are listed:

   - **NT** is greyed out and cannot be selected, because the project-wide platform constraints exclude it. This prevents you from creating an NT-specific object within project constraints for AIX and 390.
   - **AIX** is selected by default, based on the project-wide platform constraints.
   - **390** is selected by default. based on the project-wide platform constraints.
2. Deselect the **AIX** option.
3. Name the object Claim390DOImpl, with the file name ClaimFile390DOImpl.
4. Click **Next** to turn to the Behavior page.
5. Set the following behaviors:
   - **Environment: BOIM with any key**
   - **Form of Persistent Behavior and Implementation: Embedded SQL**

     You cannot select the Cache Service option here, because it is not available on OS/390. The embedded SQL option on OS/390 is fast enough not to require an alternative.
   - **Data Access Pattern: Delegating**
6. Click **Finish**. ClaimAIXDOImpl appears under ClaimDO.

Define a Persistent Object and Schema

Each version of the data object requires its own persistent object (one with the **Cache Service** type of persistence and one with the **Embedded SQL** type of persistence), but they can share the same schema definition because they are both accessing the same data.

Add a persistent object for AIX, and a common schema for both platforms:

1. From the pop-up menu of ClaimAIXDOImpl, click **Add Persistent Object and Schema** to open the Add Persistent Object and Schema wizard.
2. Type ClaimDBGroup in the **Group Name** field.
3. Type ClaimDB in the **Database** field.
4. Name the persistent object ClaimAIXPO.
5. Click the **Finish** button.

The ClaimDBGroup schema group, ClaimDB schema, and ClaimAIXPO persistent object appear in the DBA-Defined Schemas folder.

Add a persistent object for OS/390:

1. From the pop-up menu of ClaimDB.Claim in the DBA-Defined Schemas folder, click **Add Persistent Object** to open the Add Persistent Object wizard.
2. Make sure the type of persistence is set to **Embedded SQL**.

3. Review the mappings (from **SQL Type** INTEGER to **Attribute Type** long).
4. Name the persistent object Claim390PO.
5. Click **Finish**.

Claim390PO appears under ClaimDB.Claim in the DBA-Defined Schemas folder.

Map the OS/390 data object implementation and persistent object:

1. From the pop-up menu of Claim390DOImpl in the User-Defined Data Objects folder, click **Properties** to open the Data Object Implementation wizard.
2. Click the title bar and turn to the Associated Persistent Objects page.
3. Add a persistent object instance with the default instance name (iPO), and with type Claim390PO.
4. Click **Next** to turn to the Attributes Mapping page.
5. Map claimNo to iPO.claimNo, and map state to iPO.state.
6. Click **Next** to turn to the Methods Mapping page.
7. Map each method to its equivalent (insert to iPO.insert, retrieve to iPO.retrieve, and so on).
8. Click **Finish**.

Claim390PO appears under Claim390DOImpl, in the User-Defined Data Objects folder and User-Defined Business Objects folder.

Add a Managed Object

While you can create separate managed objects for both platforms, there is no need in this case. Both versions of the component can use the same managed object.

1. From the pop-up menu of ClaimBO in the User-Defined Business Objects folder, click **Add Managed Object** to open the Managed Object wizard.
2. Click **Finish**.

ClaimMO appears under ClaimBO, in the User-Defined Business Objects folder.

Generate the Code

You can generate all the code for the components, including the separate method versions and data object implementations for each platform, in one step:

1. From the pop-up menu of ClaimFile in the User-Defined Business Objects folder, click **Generate - All**.

This will take some time. When the code generation is complete, review the contents of the two directories (Working\AIX and Working\390).

Configure the ClaimDB Database

You need to define (in DB2) the ClaimDB database and Claim table that your component will access. You should have a database administrator perform this procedure. To build on both platforms, you will need to configure the database and table on both AIX and OS/390.

To configure the database and table, you need to enter the following commands from a DB2 command prompt.

```
create database ClaimDB
connect to ClaimDB
create table Claim (claimNo integer not null, state integer, primary key(claimNo))
```

The syntax for the last command is provided by Object Builder in the generated .sql file for the ClaimDBGroup schema.

Define a Common Client DLL
Because the client interfaces are the same on both platforms, there is no need to specify separate client DLLs (known on AIX as shared library files). You can define a single client DLL, using the same build configuration options. The appropriate makefile will be generated into both the Working\AIX and Working\390 directories.

1. From the pop-up menu of the Build Configuration folder, click **Add Client DLL** to open the Client DLL wizard.
2. Name the DLL ClaimC.
3. Set the deployment platforms to **AIX** and **390**.
4. Click **Next** to turn to the Client Source Files page.
5. Select all the client source files for Claim and add them to the **Items Chosen** list.
6. Click **Finish**.

ClaimC appears under the Build Configuration folder.

Define a Server DLL for AIX
Because you have different data object implementations for the two platforms, you need to define different server DLLs.

1. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.
2. Name the DLL ClaimAIXS.
3. Set the deployment platform to **AIX**.
4. Click **Next** to turn to the Server Source Files page.
5. Select all the server source files for Claim except for Claim390DOImpl, and add them to the **Items Chosen** list.
6. Click **Finish**.

Define a Server DLL for OS/390
In addition to defining a separate server DLL for OS/390, you can also choose to run a remote build.

1. From the pop-up menu of the Build Configuration folder, click **Remote OS/390 Options** to open the Remote OS/390 Options wizard.

   You can specify an OS/390 host on which to build the Claim DLLs for OS/390, using the generated source in the Working\390 subdirectory. When you build the DLLs, the OS/390 DLL will get built on the specified host.
2. Click **Finish** when you have completed the configuration. If you do not configure the remote build, then the DLLs will be built locally. You will still be able to debug the code, but you will not be able to run it.
3. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.
4. Name the DLL Claim390S.
5. Set the deployment platform to **390**.
6. Click **Next** to turn to the Server Source Files page.
7. Select all the server source files for Claim except for ClaimAIXDOImpl, and add them to the **Items Chosen** list.
8. Click **Finish**.

Build the DLLs
To generate the makefiles and build the DLL files:

1. From the pop-up menu of the Build Configuration folder, select **Generate > All > C++ Default Targets** to generate makefiles for all the DLL files defined in the folder and generate an all.mak file that calls the DLL makefiles.
2. From the same pop-up menu, select **Build > Out-of-Date Targets > C++** to call all.mak and display the progress of the build in a window.
3. Close this window after the build finishes.

For OS/390, the ClaimC.dll and ClaimS.dll files are stored in the specified directory on the specified host.

For AIX, the libClaimC.so and libClaimS.so files are stored in Working\AIX

Define a Container
Define a container to hold the component on the server. You can use the same container definition for both application families.

1. From the pop-up menu of the Container Definition folder, click **Add Container Instance** to open the Add Container wizard.
2. Accept the deployment constraints of **NT** and **390**
3. Name the container ContainerOfClaims.

   The name is the only information that will be used in the OS/390 installation. The rest of the information in the wizard will be ignored on OS/390, and can be AIX-specific.
4. Click the title bar and turn to the Service page.
5. Click **Use RDB Transaction Service**.
6. Click the title bar and turn to the Data Access Patterns page.
7. Set the following patterns:
   - **Business Object: Delegating**
   - **Data Object: Delegating**
   - **Cache Service**

   These are based on the settings in the ClaimBO business object implementation, and the ClaimAIXDOImpl data object implementation.
8. Click **Finish**.

ContainerOfClaims appears under the Container Definition folder.

Define an Application Family and Application for AIX
To define the application family and server application for AIX, follow these steps:

1. From the pop-up menu of the Application Configuration folder, click **Add Application Family** to open the Add Application Family wizard.
2. Name the family ClaimAppFamAIX.
3. Click **Finish**. ClaimAppFamAIX appears under the Application Configuration folder.
4. From the pop-up menu of ClaimAppFamAIX, click **Add Application** to open the Add Application wizard.
5. Name the application ClaimAppAIX.
6. Set the initial state of the application to **stopped**.
7. Click **Next** to turn to the Additional Executables page.

8. Click the **Browse** button to open the Executables to Include dialog.
9. Locate your Object Builder working directory.
10. From this directory, select:
    - Claim.sql
    - ClaimAIXPO.bnd
11. Click the **OK** button.
12. Click **Finish**. ClaimAppAIX appears under ClaimAppFamAIX.

Define an Application Family and Application for OS/390
To define the application family and server application for OS/390, follow these steps:

1. From the pop-up menu of the Application Configuration folder, click **Add Application Family** to open the Add Application Family wizard.
2. Name the family ClaimAppFam390.
3. Click **Finish**. ClaimAppFam390 appears under the Application Configuration folder.
4. From the pop-up menu of ClaimAppFam390, click **Add Application** to open the Add Application wizard.
5. Name the application ClaimApp390.
6. Set the initial state of the application to **stopped**.
7. Click **Next** to turn to the Additional Executables page.
8. Click the **Browse** button to open the Executables to Include dialog.
9. Locate your Object Builder working directory.
10. From this directory, select:
    - Claim.sql
    - Claim390PO.bnd
11. Click the **OK** button.
12. Click **Finish**. ClaimApp390 appears under ClaimAppFam390.

Configure the Component with Both Applications
Configure Claim for AIX:

1. From the pop-up menu of ClaimAppAIX in the Application Configuration folder, click **Add Managed Object** to open the Managed Object Configuration wizard.
2. Select ClaimFileMO ClaimMO. The rest of the fields should fill in with correct defaults.
3. Click **Next** to turn to the Data Object Implementations page.
4. Add ClaimAIXDOImpl.
5. Click **Next** to turn to the Container page.
6. Select ContainerOfClaims.
7. Click **Finish**.

ClaimMO appears under the ClaimAppAIX application.

Configure Claim for OS/390:

1. From the pop-up menu of ClaimApp390 in the Application Configuration folder, click **Add Managed Object** to open the Managed Object Configuration wizard.
2. Select ClaimFileMO ClaimMO. The rest of the fields should fill in with correct defaults.

3. Click **Next** to turn to the Data Object Implementations page.
4. Add Claim390DOImpl.
5. Click **Next** to turn to the Container page.
6. Select ContainerOfClaims.
7. Click **Finish**.

Generate the application installation information:

1. From the pop-up menu of the Application Configuration folder, click **Generate**.

   The DDL that defines the applications for System Management is generated into the Working\AIX\ClaimAppFamAIX and Working\390\ClaimAppFam390.

Summary

You have created a component for deployment on either AIX or OS/390, with different versions of some component objects to take advantage of platform-specific development options. You have defined separate build and packaging processes, and have created two separate application packages targeted at two different platforms, based on a single project model.

Chapter 8. Team Development

When you develop a standalone project, the entire application is contained in a single project, and its development cycle, from definition through build to packaging, is all handled through that single project. To develop this same application in a team environment, you simply distribute the application's components among a number of interdependent projects. Each project can then be worked on by a separate developer, with the code in the project built as required.

Typically, a team environment begins with a standalone project or Rose design, in which the basic structure of the application is defined. Then the standalone project is split out into multiple projects, that are accessed and edited through a change control system, and kept up-to-date with regular automated builds.

If you are working in Rose, then your design can be split out by package, with each package in Rose corresponding to a project in the team environment.

If you are starting with a standalone Object Builder project, then your design can be split out by package (that is, conceptual groupings of related components) or by layer (that is, different layers of component objects: business objects, data objects, or persistent objects).

If you want to define a set of standard interfaces for which other projects can provide implementations, you can do so by defining the standard interfaces as simple business objects with minimal implementations, and inherit from them as if they were abstract base classes.

Generally, a team environment consists of:

- A number of interdependent projects, which contain the component objects that make up the application.
- An integration project, which defines the build configuration and application packaging options for the application.
- A change control system, which holds all the projects, and controls access to them.
- An automated build process, which extracts all projects, generates and builds the code, on a daily or nightly basis.
- A project repository, which is the result of the automated build process, and that can be used to resolve dependencies when a team members checks out and updates a project.

RELATED CONCEPTS

"Projects and Models" on page 4
"Design Principles for Component Broker Applications" on page 3
"Change Control" on page 202
"Model Interchange with XML" on page 203

RELATED TASKS

"Set up a Team Environment" on page 204
"Work in a Team Environment" on page 212
"Maintain a Team Environment " on page 223

© Copyright IBM Corp. 1997, 1998

Change Control

When you set up a team environment, you will need to provide a change control mechanism to ensure that your team members are always working with the latest versions of their projects. Whatever change control mechanism you use, you need to set a consistent unit of change control that can be used throughout your change control system. The unit you choose will be checked out and locked while a team member is working with it, and then checked back in and released when the team member has finished working with it.

You can set up your change control system to manage information in any of the following units. Whichever unit you choose should be used consistently throughout your system.

- The model files for a project (the contents of a project's \Model directory). This is the recommended unit of change control.

  The model files can simply be opened and worked with. If you are using external files to provide method implementations, it may be easier to package the entire \Model directory (for example, in a .zip file) and use that as the unit of change control.
- The generated xml files for a project (the contents of a project's \Export directory).

  You can generate XML files for the entire project, or for selected components or component objects within the project. This allows you to store and exchange information at a more granular level, but involves extra work because you must import and export the XML files, rather than simply opening and saving the model files.
- The generated source files for a project (the contents of a project's \Working directory).

  This assumes that each team member is maintaining a single copy of their project model. The generated code is stored for backup purposes only, and for regular automated builds.

Once your change control system is in place, and your project information is stored in it, you can use it to manage your team environment.

Typically, you should use a daily build process to extract all projects, build them, and make the result available to team members. A team member who wants to make a change to a project can then check out the project from the change control system, and use the daily build structure to resolve the project's dependencies.

RELATED CONCEPTS

"Chapter 8. Team Development" on page 201
"Projects and Models" on page 4
"Model Interchange with XML" on page 203

RELATED TASKS

"Set up a Change Control Process" on page 209
"Import Changes to Methods" on page 272

Model Interchange with XML

You can exchange model information between projects using an exported XML format. This format should **not** be edited directly.

When you export XML, it is placed in the exporting project's \Export directory.

You can export information at the project, folder, component, or object level. The exported XML conforms to a DTD (document type definition) for Component Broker models. Only XML that conforms to the DTD can be imported.

You can export XML for each level as follows:

Project
You can export the entire project model by selecting **File - Export Model** in Object Builder. A set of XML files (udbo.xml, nidl.xml, dll.xml, appl.xml, cont.xml) that define the project model are generated to the \Export directory.

Folder
You can export the contents of a particular folder by selecting **Export** from the folder's pop-up menu. The XML file representing the folder is generated to the \Export directory.

You can generate XML for the following folders:

- User-Defined Business Objects folder (udbo.xml)
- User-Defined Data Objects folder (uddo.xml)
- DBA-Defined Schemas folder (udschema.xml)
- User-Defined PA Schemas folder (udpao.xml)
- Non-IDL Types folder (nidl.xml)
- Build Configuration folder (dll.xml)
- Application Configuration folder (appl.xml)
- Container Definition folder (cont.xml)

Component or Object
Select **Export** from the pop-up menu of a component elements, and then set what to export in the Export wizard, as follows:

- User-Defined Business Objects folder, business object file:
  Select the **Export data objects** option to export the entire component.
  Deselect the **Export data objects** option to export the business object layer only (business object interface and implementation, key and copy helper, managed object).
- User-Defined Data Objects folder, data object file:
  Select the **Export persistent objects** option to export both the data layer and the persistent layer (data object interface and implementation, persistent object and schema).
  Deselect the **Export persistent objects** option to export the data object layer only (data object interface and implementation).
- DBA-Defined Schemas folder, schema group:
  Always exports the schema group, along with any schemas and associated persistent objects.

The exported XML file has a name based on the name of the element you exported from (the business object file name, data object file name, or schema group name).

RELATED CONCEPTS
"Chapter 8. Team Development" on page 201

RELATED TASKS
"Export XML" on page 224
"Import XML" on page 225

Set up a Team Environment

To set up a team environment, you typically start by defining the structure of your application (either in a Rose design, or in a single Object Builder project), then divide the structure into working units (either by exporting from Rose, or by splitting up the single Object Builder project). Once you have the directory structure that holds your design defined, you can add an integration or build project. You can then store the structure in a change control system, set up an automated build process, and finally set up the individual development machines.

The tasks involved in setting up a team environment are as follows:
1. "Export a Rose Design to a Team Environment"
2. "Split up a Project for Team Development" on page 206
3. "Add an Integration Project to a Team Environment" on page 208
4. "Set up a Change Control Process" on page 209
5. "Set up an Automated Build Process" on page 210
6. "Set up a Team Development Environment" on page 211

RELATED CONCEPTS
"Chapter 8. Team Development" on page 201

RELATED TASKS
"Work in a Team Environment" on page 212

Export a Rose Design to a Team Environment

When you export a Rose design to Object Builder, you can either export the entire design to a single project, or export each top-level package or class as a separate project.

If you export as separate projects, then your target projects should all be subdirectories of the same root path, and the target project's names should match the names of the top-level packages in Rose.

You can export the entire Rose model, or selected top-level classes or packages in the model.

To export the entire model, follow these steps:
1. Organize your design into packages that represent your desired project setup.
2. Select **File** - **Export to Object Builder**. The Rose Bridge wizard opens to the Export from Rose 98 to Object Builder Page.
3. Add any necessary virtual symbols and associated actual path mappings to the Virtual Path Mapping list box.

4. Select the destination directory you want to store your projects in. The exported projects will all be stored in subdirectories of this directory.
5. Select the **Separate Project**s radio button.
6. Select the **Entire Model** radio button.
7. Click **Finish.**

Your model is exported to projects in the specified directory. Your model is saved in Rose as part of the export process.

To export selected top-level classes or packages in the model, follow these steps:

1. Organize your design into packages that represent your desired project setup.
2. Select **File** - **Export to Object Builder**. The Rose Bridge wizard opens to the Export from Rose 98 to Object Builder Page.
3. Add any necessary virtual symbols and associated actual path mappings to the Virtual Path Mapping listbox.
4. Select the destination directory you want to store your project in. The directory becomes an Object Builder project directory.
5. Select the **Separate Projects** radio button.
6. Select the **Selected Packages or Classes** radio button.
7. Click **Next** to turn to the Export from Rose 98 to Object Builder, Selection Page.
8. Use standard selection techniques (**Click**, **Shift-Click** and **Ctrl-Click**) to select the top-level packages and classes you want to export.

   As items are selected in the tree view on the left, the tree view on the right displays the component objects or elements which will be created for that item. The export process creates component objects according to the properties in the Class Specification notebook.
9. Click **Finish.**

   Your model is exported to projects in the specified directory. Your model is saved in Rose as part of the export process.

If you are using Rose 98 with a model which was created in Rose 4.0, any Component Broker-specific properties you have customized (for example, IDLSpecificationType) will be moved to the appropriate pages of the Rose 98 specification notebooks the first time the model is exported.

Once you have completed the export process, your design is applied to the \Model subdirectory in each of the selected projects. The interchange file udbo.xml used in the export process is stored in the \Import subdirectory in each of the selected projects. The classes and relationships you defined in Rose have been mapped to their Object Builder equivalents, and any component objects you specified have been defined in skeleton form.

The export process maintains the following design elements:

- The classes in your design
- Class inheritance
- Class relationships
- Attributes
- Methods

The export process adds the following elements:

- File and module objects (the mapping of classes to files and modules depends on the packaging structure used in Rose).

- Read and write methods, for each public attribute.
- Public attributes (get and set methods), to support aggregations of classes and navigable associations among classes.
- Component Broker objects (business object implementation, data object interface, copy helper, key), as specified during the import, in skeleton form.

When you export to separate projects, XML files are generated in the \Import directory of each project. These files are then imported into Object Builder using the equivalent of an obimport command with the -X option, which imports all the files at once and generates the appropriate dependencies in the various projects.

For example:

obimport -X -d=Import e:\myRBprojects\projA e:\myRBprojects\projB e:\myRBprojects\projC

This command looks in the Import subdirectory of each listed project directory, and imports the cont.xml, udbo.xml, uddo.xml, udschema.xml, dll.xml, and appl.xml files it finds there.

When you export from Rose, the export process generates a file named xmi.xml in the target *project*\XMI subdirectory. This file allows the export process to track changes to design elements, so that if you change the name of a method in Rose and re-export, the change will be applied to the appropriate method in Object Builder. It also keeps track of any elements that do not have equivalents in both models, so that these elements are not simply lost in the bridging process.

RELATED CONCEPTS
"The Rose Bridge" on page 76
"Projects and Models" on page 4
"Chapter 8. Team Development" on page 201

RELATED TASKS
"Set up a Team Environment" on page 204
"Export a Design from Rose" on page 89
Import a Rose Design into Object Builder
"Import XML" on page 225
"Work with an Exported Design" on page 91

Split up a Project for Team Development

You can split an existing project into a set of interdependent projects in a team environment. You can choose to split along package lines and component layers:

- Packages
  Your application design can be viewed in terms of categories or groupings of related components, which serve to partition the logical model of your application. In UML terms, these categories are packages. If you created your application design in Rational Rose, you can export the design directly to a team environment. The design will automatically be split into a number of different projects, based on the top-level package structure in Rose.

  If your team development roles align with divisions in the design, this is the main strategy you will use.
- Component layers
  Each component consists of a behavior layer (the business object), a data layer

(the data object), and a persistence layer (the persistent object). You can split out a component into its separate component objects, and maintain them in separate, interdependent projects.

If your team development roles align with component layers (for example, an object-oriented designer at one end versus a database administrator at the other), this is the main strategy you will use.

You can mix strategies within a team environment. You will also want an additional, separate project from which you can coordinate application-wide builds and application packaging.

To divide an existing project into separate projects, follow these steps:

1. Open the existing project.
2. Export XML for individual components or component layers. Later you will group the exported files into projects. You can select **Export** from the pop-up menus of the various component files, and set what to export in the Export wizard, as follows:
   - User-Defined Business Objects folder, business object file:
     Select the **Export data objects** option to export the entire component. You can then group the exported components into packages.
     Deselect the **Export data objects** option to export the business object layer only (business object interface and implementation, key and copy helper, managed object). You can then maintain the business object layer in one project, and its associated layers in other projects.
   - User-Defined Data Objects folder, data object file:
     Select the **Export persistent objects** option to export both the data layer and the persistent layer (data object interface and implementation, persistent object and schema). You can then maintain these two layers in one project, and the business object layer in a separate project.
     Deselect the **Export persistent objects** option to export the data object layer only (data object interface and implementation). You can then maintain all three layers in separate projects, once you export the schema layer.
   - DBA-Defined Schemas folder, schema group:
     Always exports the schema group, along with any schemas and associated persistent objects.

   The exported XML file has a name based on the name of the element you exported from (the business object file name, data object file name, or schema group name). The exported files are in the \Export subdirectory of the project directory.
3. Export XML for the Build Configuration and Application Configuration folders (click **Export** from each folder's pop-up menu). The resulting files (dll.xml, appl.xml) are in the \Export subdirectory of the project directory.
4. Create a set of project directories that represent the groupings of component objects you want, under a single parent directory.

   For example, you could create the project directories e:\allprojects\policyBO, e:\allprojects\carpolicyBO, e:\allprojects\allDOs, e:\allprojects\integration.
5. In each project directory, create a subdirectory (for example, \Import).
6. Place the component XML files for each grouping in the subdirectory of its equivalent project.

   You can place as many or as few XML files in each subdirectory as you want, depending on the way you want to organize the project contents.

7. From the command line, run the obimport command to create a set of interdependent project models based on the XML files in each project's subdirectory (for example \Import). For example:

   obimport -X -d=Import e:\allprojects\policyBO e:\allprojects\carpolicyBO e:\allprojects\allDOs e:\allprojects\integration

   This example would create four projects, by importing the XML files found in the \Import subdirectories of the listed directories.
8. Set up the environment variable OBMODELPATH to point to the parent directory that contains your project directories. For example:

   ```
   set OBMODELPATH=f:\allprojects;
   ```

Once you have split the project into a set of projects in a team environment, you can continue with your team environment set up. You need to choose a change control unit, and a system to store the units in. You need to set up the team environment, and the development machines, to support either local dependency resolution (with project dependencies extracted from the change control system) or remote dependency resolution (with project dependencies resolved by a project depository on a shared network drive). You need to set the makefile generation preferences for each Object Builder installation to reflect a team environment.

RELATED CONCEPTS

"Chapter 8. Team Development" on page 201
"Projects and Models" on page 4
"Design Principles for Component Broker Applications" on page 3
"Model Interchange with XML" on page 203

RELATED TASKS

"Export XML" on page 224
"Import XML" on page 225
"Add an Integration Project to a Team Environment"

Add an Integration Project to a Team Environment

If you created your team environment by splitting up an existing project, then your integration project already exists, and simply contains the build and application configuration information from the original project. However, if you created your team environment by exporting a model from Rose, you will need to add a new integration project, as described here.

The integration project will define the build configuration options and makefiles for all the components defined in your projects. Typically, it will also contain the application configuration information, when you reach the stage of packaging the application. To add an integration project, follow these steps:

1. Create a new project. It should be part of the same directory structure as your other projects (that is, they should all be under the same parent directory). In the Open Projects wizard, Project Dependencies page, list all the projects in your team environment as dependencies.
2. Once the new project is open, click **File - Preferences** to open the Preferences notebook.
3. Click on the Tasks and Objects node in the Preferences tree view.
4. Under **Makefile Generation**, set the **Team Environment** option. This allows the project's build process to locate the generated code in the dependency projects' \Working directories.

In order to locate the other projects' generated code, the integration project's makefiles will use absolute paths instead of relative ones. If your directory structure changes, you will need to regenerate the makefiles.

5. Add client and server DLLs for all the components in your application. If a DLL is already defined in one of the dependency projects, then you must use a different name for the DLL configuration node in the integration project, but you should use the same name for the built DLL file.
6. Add application families and applications, and configure the managed objects of the various components in the other projects.
7. Save and close the project.

The project will be used as the starting point for any regular automated builds, and for packaging the application.

RELATED CONCEPTS

"Chapter 8. Team Development" on page 201
"Projects and Models" on page 4

RELATED TASKS

"Set up a Team Environment" on page 204
"Split up a Project for Team Development" on page 206
"Set up an Automated Build Process" on page 210

Set up a Change Control Process

Once you have created your team directory structure and skeleton projects (either through export from Rose, or by splitting up an existing project), you can set up a change control process, to ensure that only one person makes changes to a project at a time.

Typically, you should use a daily build process to extract all projects, build them, and make the result available to team members. A team member who wants to make a change to a project can then check out the project from the change control system, and use the daily build structure to resolve the project's dependencies.

Setting up a change control process requires the following general steps:

1. Create the team directory structure (that is, a set of project directories that hold the elements of your application). You can create the structure either by exporting a design from Rose, or by taking a design in an existing Object Builder project, and splitting it up into multiple projects.
2. Add an integration project to the team directory structure.
3. Select a unit to use for change control (for example, the contents of each project's \Model directory).
4. Check all projects into the change control system.
5. Set up a daily or nightly build process, that will extract all projects, generate code for the entire application using the integration project, and build the application DLLs. The resulting project repository (all projects, with their generated and built code) needs to be available for team members to access (for example, as a zip file in the change control system, or as a directory structure on the LAN).

RELATED CONCEPTS
"Change Control" on page 202
"Chapter 8. Team Development" on page 201
"Projects and Models" on page 4

RELATED TASKS
"Set up a Team Environment" on page 204

Set up an Automated Build Process

You can use an automated build process to create and update a project repository, which can be used to resolve the dependencies of a project being edited.

The automated build process needs to do the following:

1. Extract all projects from the change control system.
2. Generate code for all projects, using obgen with the -linked option. For example:

   ```
   obgen -pF:\allprojects\projectA -aAll -tNT -linked
   obgen -pF:\allprojects\projectB -aAll -tNT -linked
   obgen -pF:\allprojects\projectC -aAll -tNT -linked
   ```

   The above commands generate the code for all objects (-aAll) in projectA, projectB, and projectC (the equivalent of selecting **Generate - All** from the pop-up menu of the User-Defined Business Objects folder in each project). The code is generated for the platform Windows NT (-tNT), and placed in each project's \Working\NT directory.
3. Generate the makefiles for the application integration project. For example:

   obgen -pF:\allprojects\Integration -aMake -linked -tNT

   The above command generates the makefiles (-aMake) defined in the Integration project. The -linked option generates the makefiles with absolute paths to the code found in the other projects' \Working\NT directories, rather than generating makefiles that assume the generated code is in the current project's \Working\NT directory.
4. Build all the DLLs defined in the application integration project. For example:

   nmake F:\allprojects\Integration\Working\NT\all.mak

   The DLLs, .jar files, and any other targets defined in the makefiles are built in the integration project's \Working\NT directory.
5. Make the full extracted directory structure available, including the \Model directories, and the \Working directories with the generated code and built DLLs.

   For example, you could export F:\allprojects on the network, or zip the contents of the directory and place the zip file in the change control system.

The following sample build script extracts the project model directories for four projects, including an integration project, generates their code, generates the makefiles, builds the code, and creates a zip file:

allprojectsbuild.bat

```
<Extract all projects from your change control system>
obgen -pF:\allprojects\projectA -aAll -tNT -linked
obgen -pF:\allprojects\projectB -aAll -tNT -linked
obgen -pF:\allprojects\projectC -aAll -tNT -linked
obgen -pF:\allprojects\Integration -aMake -linked -tNT
nmake F:\allprojects\Integration\Working\NT\prjall.mak
zip latest.zip F:\allprojects\* -r
<publish latest.zip>
```

RELATED CONCEPTS
"Chapter 8. Team Development" on page 201

RELATED TASKS
"Run Object Builder in Batch Mode" on page 11
"Add an Integration Project to a Team Environment" on page 208
"Set up a Team Environment" on page 204

Set up a Team Development Environment

Once the projects in the team environment have been stored in a change control system, and an automated build process has begun producing regularly updated project repositories, you can set up the development machines to be part of the team environment.

On each development machine, set up the team environment as follows:

1. Create a local directory that will hold the project repository created by the nightly build (for example, f:\allprojects\). Each team member will be responsible for updating their copy of the repository when required.
2. Create another local directory to hold any projects you check out from the change control system for editing (for example, f:\currentprojects\).
3. Set up OBMODELPATH to include first the current projects directory, and then the project repository directory.

   For example:

   ```
   set OBMODELPATH=f:\currentprojects;n:\allprojects;
   ```

   The directories, and their subdirectories, will be searched for project dependencies in the order they are listed. For example, if you check out three interdependent projects into f:\currentprojects\, then the duplicates of those projects in the f:\allprojects\ directory are ignored, because their dependencies on each other are resolved before the f:\allprojects\ directory is searched. Any additional dependencies, beyond just the three checked out projects, will be resolved in the f:\allprojects\ directory.

On each development machine, set up Object Builder for a **Team Environment**:

1. Open Object Builder, with any project.
2. Click **File - Preferences** to open the Preferences notebook.
3. Click on the Tasks and Objects node in the tree view.
4. Under **Makefile Generation**, set the **Team Environment** option. This allows the project's build process to locate code and makefiles in other project \Working directories, to resolve makefile dependencies correctly.

   In order to locate code in other projects' \Working directories, the generated makefiles will use absolute paths instead of relative ones. If your directory structure changes, you will need to regenerate the makefiles.
5. Save and close the project.

RELATED CONCEPTS
"Chapter 8. Team Development" on page 201
"Projects and Models" on page 4

RELATED TASKS

"Set up a Team Environment" on page 204
"Work in a Team Environment"
"Maintain a Team Environment " on page 223

Work in a Team Environment

The same rules that apply to developing an application within a single project apply to developing an application across multiple projects. You must define parent components before child components, and referenced interfaces before referencing interfaces.

There are some considerations that are specific to development in a team environment, including how you exchange information with a Rational Rose model, and how you work with references across projects. They are described in the following tasks:

1. "Import Projects from a Team Environment"
2. "Create a Project in a Team Environment" on page 215
3. "Edit a Project in a Team Environment" on page 216
4. "Delete a Project in a Team Environment" on page 217
5. "Build DLLs in a Team Environment" on page 217
6. "Package an Application in a Team Environment" on page 218

RELATED CONCEPTS

"Chapter 8. Team Development" on page 201

RELATED TASKS

"Set up a Team Environment" on page 204
"Work in a Team Environment"

Import Projects from a Team Environment

You can import Object Builder projects into Rose. If the imported projects were originally created by a Rose export, then the new Rose model created by the import will mirror the information in the original, exported Rose model's Logical View. If your original model has additional information in other views, you can consolidate the two models (the original exported one, and the newly imported one) using the Rose 98 Visual Differencing tool.

If the projects were created only in Object Builder, or the original design is unavailable, then the import process creates a new Rose design. When the import is completed, each project maps to a package in Rose, and each business object interface maps to a class in Rose. You can then work with the design in Rose, and export the changes back to Object Builder.

To import a set of interdependent Object Builder projects (that is, projects in a team environment) into Rose 98, follow these steps:

1. Select **File** - **Import from Object Builder**. The Rose Bridge wizard opens to the Import from Object Builder to Rose 98 Page
2. Enter the name of the directory that contains the Object Builder projects you are importing.

3. Enter the name of the Rose model file you are importing to. If you know your project will be imported into a category file (.cat) on Rose 98, then you need to specify the virtual path mapping information by entering the symbol and actual path data.
4. Select the **Import from: Separate Project**s option.
5. Click **Finish**.

The projects in the directory you selected are imported into the Rose model file you specified.

The import process works as follows:
1. The import process calls the obexport command to generate an XML file for the project (\Export\udbo.xml).
2. The import process checks to see if there is an \XMI\xmi.xml file in the project directory. This file is created by the Rose Bridge to preserve any information that would otherwise be lost during transfer between Rose and Object Builder.
3. The import process generates a Rose model file, based on the udbo.xml file and the xmi.xml file.
4. The import process updates the xmi.xml file to contain any Object Builder information that cannot be imported. For example, details of the implementation, key, and copy helper for a component, that cannot be stored as elements in Rose 98.
5. The import process loads the generated Rose model into Rose 98.

The import process maps Object Builder elements as follows:
- Business object files, modules, and interfaces that already have a mapping (because they were created by export from Rose) maintain that mapping.
- New business object files, modules, and interfaces (added directly to Object Builder, not by export from Rose) are mapped to packages, subpackages, and classes.
- IDL constructs with file or module scope become classes in Rose
- Attributes of an interface become attributes of a class in Rose
- Methods of an interface become operations of a class in Rose
- Parent interfaces become class relations in Rose
- Object relationships that were created by export from Rose are imported as the role of an association.
- Sequence attributes of the interface that were created by export from Rose are imported as the role of an association.

The import process keeps the following elements in the xmi.xml file:
- Component objects other than the business object interface (business object implementation, data object interface, copy helper, key)
- The method bodies
- Object relationships that were created directly in Object Builder (not by export from Rose)
- IDL constructs with interface scope

The import process updates the following properties in the Rose specification notebooks:

- Class Specification, IDL page, CreateImplementation
  property is set if the business object interface has a business object implementation
- Class Specification, IDL page, CreateKey
  property is set if the business object interface has a key
- Class Specification, IDL page, CreateCopyHelper
  property is set if the business object interface has a copy helper
- Class Specification, IDL page, IsQueryable
  property is set if the business object interface has the option **The interface is queryable** checked, in its properties notebook
- Attribute Specification, IDL page, length
  property is set if the attribute is of type string, and has associated size information.
- Attribute Specification, DDL page, IsIncludedInCopyHelper
  property is set if the attribute is part of the component's copy helper
- Attribute Specification, DDL page, PrimaryKey property is set if the attribute is part of the component's key.
- Association Specification, IDL A/B pages, MapAsObjectRelationship
  property is set if an object relationship or sequence attribute in the business object interface was created by exporting the role of an association from Rose
- Association Specification, IDL A/B pages, RelationshipImplementation
  property is set if the object relationship has a selected implementation type in the business object implementation

In order to track changes between Component Broker objects and Rose elements, the import process uses the UUID of an element as an identifier. The UUID is stored as the uuid property of IDL page in Rose for each package, class, attribute, operation, and role of association.

You have now imported an Object Builder project into a Rose model. If the imported project was created by export from Rose, and the original Rose model contains information in other views besides the Logical view, then you should consolidate the new model with the original model before doing any more design work.

To merge the new model with the original model, follow these steps:

1. Click **File-Save** to save the new model.
2. Click **Tools-Visual Differencing** to start the merging process.
3. When the **Give reference model** dialog opens, specify the original .mdl file.

   The Visual Differencing tool will load both models and generate a list of differences
4. In the Visual Differencing interface, click on the **+** next to the **Difference found** item to expand the tree one level.
5. Since no changes have been made to any information in the Use Case View, merge the information from the original model into the new model:
   a. Click **Use Case View** to select it
   b. Click **Merge** in the Use Case View pop-up
   c. Make sure the **Replace with reference** option is selected, and click **Merge**.
6. Repeat the same procedure for all other views that contain differences, except for the Logical view.

7. In the Logical view, you do not need to merge the entire view, only selected diagrams:
   a. Click **+** next to the 'Logical View' item to expand one level.
   b. Click **+** for all Logical View subtree members until the entire subtree is exposed.
   c. In each place a blue **+** exists in the Logical View subtree (all diagrams in the subtree):
      1) Click the item to select it.
      2) Click **Merge** in its pop-up menu.
      3) Make sure the **Replace with reference** option is selected, and click **Merge**.

   You have now completed merging information from your original model into the new model which contains the changes from Object Builder.
8. Click **File-Save** in the Visual Differencing tool and save the updated model to a new file.
9. Click **File-Exit** to close the Visual Differencing tool.
10. Click **File-Open** in Rose 98 and open the updated .mdl file.

Your model now contains the entire updated design, and you can continue your design work. When you are ready to switch back to Object Builder, you can export the design back to Object Builder by selecting **File** - **Export to Object Builder**.

RELATED CONCEPTS
"Object Builder" on page 1
"Projects and Models" on page 4
"Rose" on page 74
"The Rose Bridge" on page 76
"Chapter 8. Team Development" on page 201

RELATED TASKS
"Export a Design from Rose" on page 89
"Export a Rose Design to a Team Environment" on page 204

Create a Project in a Team Environment

To create a project in a team environment, follow these steps:
1. Create a project directory for the project, in your local project directory structure (for example, f:\currentprojects\mynewproject).
2. Identify any existing interfaces that will be required by the new project's component (for example, parent interfaces, or interfaces that will be used as attribute types or in method signatures).
3. List the projects that contain those interfaces as dependencies in the Open Project wizard, Project Dependencies Page (for example, f:\allprojects\projectA, f:\allprojects\projectB).
4. Open the project, and begin your work on the new application elements it will contain. Add components in the same way you would in a standalone environment (starting with the business object interface file, an imported DB or PA schema, or data object interface file).
5. Save the project, and check it into your change control system.

6. Check out the integration project, and add any build configuration nodes that apply (for example, client and server DLL configurations for any components you added to the new project).
7. Save your changes to the integration project, and check it back into your change control system.
8. Update the automated build process, to include code generation for the new project.

RELATED CONCEPTS

"Chapter 8. Team Development" on page 201

RELATED TASKS

"Work in a Team Environment" on page 212
"Edit a Project in a Team Environment"
"Set up a Change Control Process" on page 209
"Set up an Automated Build Process" on page 210

Edit a Project in a Team Environment

To edit a project in a team environment, follow these steps:

1. Ensure your local copy of the project repository (created by the automated build process) is up-to-date.
2. Check out the project you want to edit from the change control system.
3. Open the project. Its dependencies on other projects should resolve using the project repository (based on your OBMODELPATH settings).
4. Make any changes you want to the project. If your changes affect the way the project relates to other projects (for example, you want to add a reference that requires another project as a dependency, or you delete a reference that justifies an existing dependency), you will need to close and open the project again, to update the listed project dependencies (for example, add the dependency first, then add the new reference; or delete the reference, then remove the dependency).
5. Generate updated code for the project.

You can now build your code, using either a local definition of the DLLs you want to build, or using the integration project's DLL configurations.

Some of your changes may affect code in other projects. For example, if you rename an interface, any methods, attributes, constructs, or relationships that reference the interface have their type renamed automatically. If a referenced interface is deleted, the reference type becomes invalidType. For example, if Customer has an attribute custAgent of type Agent, and the Agent interface is deleted, Customer now has an attribute custAgent of type invalidType. You can locate all occurrences of invalidType within a project by running the model consistency checker.

RELATED CONCEPTS

"Chapter 8. Team Development" on page 201

RELATED TASKS

"Work in a Team Environment" on page 212
"Build DLLs in a Team Environment" on page 217
"Check a Model for Consistency" on page 412

Delete a Project in a Team Environment

To delete a project in a team environment, follow these steps:

1. Delete the project from your change control system.
2. Update the automated build process, to remove any reference to the project.
3. Check out the integration project, and remove any build configuration nodes for code in the project.
4. Check out any projects that have dependencies on the deleted project, and remove their dependencies on the Open Project wizard's Project Dependencies.

Any references within a project that depend on the deleted interface will be automatically modified to point to type invalidType. The following properties will be automatically modified:

- Attributes are modified when the interface whose type they are is deleted.
- Methods are modified when an interface used as their return type, or as a parameter, is deleted.
- Object relationships are modified when the interface they refer to is deleted.

You can find invalidType references within a project by checking the project model's consistency.

RELATED CONCEPTS
"Chapter 8. Team Development" on page 201

RELATED TASKS
"Work in a Team Environment" on page 212
"Check a Model for Consistency" on page 412

Build DLLs in a Team Environment

When your project is part of a team environment, typically the entire application will share a single integration project, that defines the build configuration options for all the components in the team environment, regardless of the project they are defined in.

When have edited a project and want to rebuild the code that was affected by your changes, you can use the integration project in the project repository to rebuild the affected DLLs. This will only affect your local copies of the DLLs; once you check an edited project back into your change control system, the DLLs will be rebuilt by your automated build process, and the updates will be made available in the next version of the project repository.

To build a DLL locally in a team environment, after you have made changes to a checked out project, follow these steps:

1. Regenerate the makefiles in your integration project (in your local copy of the project repository). This creates a version of the makefiles that correctly points to the updated code in your local check-out directory.
2. Build the updated makefiles, using the integration project's all.mak file.

If your project repository is editable, then you can regenerate the makefiles and build the code from within Object Builder:

1. Open the integration project.

2. From the Build Configuration folder's pop-up menu click **Generate - All Targets**
3. From the same pop-up menu click **Build - Out-of-Date Targets - Default**.

If your project repository is read-only, then you can regenerate the makefiles and build the code from the command line. For example:

1. obgen -pF:\allprojects\Integration -aMake -linked -tNT
2. nmake F:\allprojects\Integration\Working\NT\all.mak

You can also create a build configuration definition within the checked out project, without using the integration project. To do so, simply add client and server DLL definitions in the usual manner. You can use the same DLL file names as those in the integration project, but should define different DLL configuration node names.

RELATED CONCEPTS
"Chapter 8. Team Development" on page 201

RELATED TASKS
"Work in a Team Environment" on page 212
"Package an Application in a Team Environment"
"Build DLLs - Overview" on page 363
"Run Object Builder in Batch Mode" on page 11
"Set up an Automated Build Process" on page 210
"Add an Integration Project to a Team Environment" on page 208

Package an Application in a Team Environment

You can use the integration project of your team environment to do your application packaging. Add application families, applications, and managed object configurations in the usual manner. Before you create the install image, you will need to build the DLLs for the entire application, so that the built files in the project's \Working directories are up-to-date.

If you are packaging your application in a different project than your integration project, you will still need to build from the integration project first, and then copy the contents of the integration project's \Working directory (including subdirectories) to the application packaging project's \Working directory.

You can then configure the application, and create the install image, in the same way you would in a standalone project environment.

RELATED CONCEPTS
"Chapter 8. Team Development" on page 201

RELATED TASKS
"Work in a Team Environment" on page 212
"Package an Application" on page 375

Team Development with Rose - Scenario

Objectives
To create an object relationship in Rose.
To export from Rose to multiple projects.
To import from multiple projects to Rose.

Before You Begin
This scenario is a continuation of the scenario sequence:

1. "Export from Rose - Scenario" on page 95
2. "Import into Rose - Scenario" on page 98

You must complete the previous scenarios before attempting this one.

You need Rational Rose 98 installed and set up to work with Object Builder, as described in the task "Set up Rose 98" on page 74.

You need Object Builder installed.

Description
In this exercise, you will extend the Rose model for Claim to include a second component, Policy, which has a one-to-many relationship with Claim (each policy can have multiple claims). You will then export the modified model to a new set of interdependent project directories, modify the exported components in Object Builder, and import the changes.

Note that when you are creating more complicated team environments (with multiple projects, each containing multiple components), you will want to minimize the number of cross-project dependencies. In Rose terms, you would group your classes into packages, and minimize the cross-package relationships and references.

For this exercise, you will complete the following tasks:

1. Add the Policy class to your Rose model.
2. Export the model to Object Builder projects.
3. Edit the Claim component.
4. Edit the Policy component.
5. Import the projects and apply the changes to your Rose model.

Add the Policy Class
Start Rose 98, and add the Policy class, with one attribute and a one-to-many relationship with Claim.

Add the Policy class:

1. Start Rose 98.
2. Click **File - Open** and load the existing model for Claim (for example, e:\scenarios\rosemodels\claim.mdl).
3. From the pop-up menu of the class diagram, click **Class Wizard** to open the Class wizard.
4. Name the class Policy.
5. Click **Next** through the remaining wizard pages, then **Finish**.

   A class named Policy is added to the Logical View folder.

Add the attribute policyNo:

1. From the pop-up menu of Policy, click **New Attribute**. A placeholder attribute is added (named name, type of type, initial value of initval).
2. Type over each of the values for the new attribute, naming it policyNo, with type Integer and initial value of 0.

3. Click elsewhere in the diagram to apply the changes.

Add a one-to-many relationship from Policy to Claim:

1. Click **Tools - Create - Aggregate Association**. Your mouse pointer changes to an arrow.
2. Click and hold on Policy, and then drag to Claim, to draw the aggregation. When you release the mouse button, an arrow is drawn from Policy to Claim.
3. Double-click on the arrow to open Aggregation Specification notebook.
4. Turn to the Role A Detail page, and set the cardinality to n.
5. Turn to the IDL A or B page. Review the values for the aggregation properties:
   - MapAsObjectRelationship=True
     The aggregation will map to an object relationship in Object Builder. If this value were false, the aggregation would map to an attribute of type sequence in Object Builder.
   - RelationshipImplementation=Local Persistent Reference
     The relationship will be implemented in the business object implementation as a local persistent reference, rather than mapping to a database query.
6. Click **OK** to close the diagram and apply your changes.

You now have a class named Policy, with the attribute policyNo and a one-to-many object relationship to Claim.

Export to Object Builder
You are ready to export the classes to Object Builder. A separate project will be created for each class, and the project dependencies will be set accordingly.

To export to separate Object Builder projects, follow these steps:

1. Click **Save As** to save and rename your model (for example, as e:\scenarios\rosemodels\ClaimPolicy.mdl).
2. Click **File - Export to Object Builder**. The Rose Bridge Export wizard appears.
3. Specify an output directory to hold the project directories for this design (for example, e:\scenarios\roseteam\).
4. Set the **Export as: Separate Projects** option.
5. Accept the default for the other options.

   If you had exported to separate projects before, you could set the **Export Selected Packages or Classes** option to update only the projects that were affected by your change. However, when you are exporting for the first time to separate projects, you should export the entire model, to set up the project directory structure and project dependencies.
6. Click **Finish**. Your existing Rose model is automatically saved as part of the export process. A Claim project directory and a Policy project directory are created beneath the output directory you specified (for example, e:\scenarios\roseteam\claim\ and e:\scenarios\roseteam\policy\). The \Import directory under each project directory, and the \XMI directory under the output directory, contain the XML files used to transfer information from Rose to Object Builder. As part of the XML import process, dependencies are set up between the target projects, as necessary to resolve any references between the classes.

You are ready to open the projects in Object Builder, review the results of the export, and edit the components in Object Builder.

Work with Claim
Work with the exported Claim component:

1. Start Object Builder.
2. In the Open Project wizard, specify the project directory created by the export for Claim (for example, e:\scenarios\roseteam\claim\).
3. Click **Next** to turn to the Project Dependencies page. Note that there are no dependencies listed. Claim does not have any references to Policy, so it does not have a dependency on Policy's model. However, because you are now going to edit Claim and add a reference to Policy, you need to add Policy's project directory as a dependency.
4. Add Policy's directory as a dependency (for example, e:\scenarios\roseteam\policy\).
5. Click **Finish**. The project for Claim opens.
6. Expand the User-Defined Business Objects, and locate the Claim interface (under the Claim file).
7. From the pop-up menu of the Claim interface, click **Properties** to open the Business Object Interface wizard.
8. Click the title bar and turn to the Attributes page.
9. Add an attribute thePolicy of type Policy.
10. Click **Finish**.

Add the new attribute to Claim's copy helper:

1. Locate the ClaimCopy copy helper, under the Claim interface.
2. From ClaimCopy's pop-up menu, click **Properties** to open the Copy Helper wizard.
3. Move the new attribute from the Business Object Attributes list to the Copy Helper Attributes list.
4. Click **Finish**.

Save your changes and close Object Builder:

1. Click **File - Save**.
2. Click **File - Exit**.

You have made your changes to the project, saved them, and closed Object Builder. You are ready to work with Policy's project.

Work with Policy
Work with the exported Policy component, changing the implementation of its relationship, and adding a key and copy helper:

Edit the relationship implementation:

1. Start Object Builder.
2. In the Open Project wizard, specify the project directory created by the export for Policy (for example, e:\scenarios\roseteam\policy\).
3. Click **Next** to turn to the Project Dependencies page. Note the dependency on Claim, which was set up by the export process to support Policy's one-to-many relationship to Claim.
4. Click **Finish**. The project for Policy opens.
5. Expand the User-Defined Business Objects, and locate the PolicyBO implementation (under the Policy interface).

6. From the pop-up menu of PolicyBO, click **Properties** to open the Business Object Implementation wizard.
7. Click the title bar and turn to the Object Relationships page.
8. Change the relationship implementation to **Reference resolved by foreign key**.
9. Select Claim's new attribute thePolicy as the foreign key attribute for the relationship.
10. Type ClaimMOHome as the name of the home to query.

    This name is based on the name of Claim's managed object, and the home instance it is configured with. Because you have not yet created or configured Claim's managed object, the home name here is just a logical guess, which assumes you will accept the default name and home configuration options.
11. Click **Finish**.

You have created a foreign key pattern relationship between Policy and Claim. This allows the relationship to be implemented at the database level, using foreign key references, instead of at the business object level.

Add a key and copy helper:
1. From the pop-up menu of the Policy interface, click **Add Key** to open the Key wizard.
2. Move the policyNo attribute to the Key Attributes list.
3. Click **Finish**.
4. From the pop-up menu of the Policy interface, click **Add Copy Helper** to open the Copy Helper wizard.
5. Move the policyNo attribute to the Copy Helper Attributes list.
6. Click **Finish**.

You now have a key and copy helper for Policy. Save your changes and close Object Builder:
1. Click **File - Save**.
2. Click **File - Exit**.

You have made your changes to the project, saved them, and closed Object Builder. You are ready to import your changes from both projects into Rose, and review their effect on your Rose model.

Import into Rose

To import your changes into Rose, follow these steps:
1. Start Rose 98.
2. Click **File - Open** and load your Rose model for Policy and Claim (for example, e:\scenarios\rosemodels\ClaimPolicy.mdl). Your model opens, and the class diagram for Policy and Claim appears.
3. Click **File - Import from Object Builder**. The Rose Bridge Import wizard opens.
4. In the Input Directory field, type the path of the parent directory for your Object Builder projects, as defined during the export process (for example e:\scenarios\roseteam\).
5. In the Output field, make sure the current Rose model is selected (for example e:\scenarios\rosemodels\ClaimPolicy.mdl).
6. Set the **Import from: Separate Projects** option.
7. Click **Finish**.

The Rose Bridge updates the udbo.xml file for each project, updates the xmi.xml files with any project information it cannot preserve in the transfer, and then imports all four files into Rose to update the selected model file.

Review the Changes
Your changes in the Object Builder projects are applied, and have had the following effect:

- Claim has a new attribute thePolicy.
- Policy's attribute policyNo now has the DDL property isPrimaryKey set to True (reflecting your inclusion of the attribute in PolicyKey, in Object Builder).
- The aggregation association between Claim and Policy now has the IDL property RelationshipImplementation set to Reference Resolved by Foreign Key (reflecting your changes to the relationship implementation in PolicyBO, in Object Builder).

Summary
You have created a team environment with two interdependent project directories, created a cross-project foreign key pattern relationship, and applied changes in the team environment to your Rose model.

Maintain a Team Environment

After your team environment is defined (either through migration or evolution), maintenance of the environment must provide for the relocation of projects, the relocation of objects between projects, and the management of cross-project dependencies.

The main strategies for managing multiple projects are:

- Make changes in logical units (move parent and child components together when possible).
- Make changes in logical order (change parent components before child components, change referenced components before referencing components).
- Edit project divisions by selectively exporting the project's contents as XML files, deleting the content from the project, then importing the XML files into another project or projects.
- When you have two versions of a project, resolve the differences by exporting and merging their XML with the Compare and Merge Tool for XML.

The specific tasks for maintaining a team environment are as follows:

1. "Import XML" on page 225
2. "Move a Project" on page 227
3. "Change Project Divisions" on page 227
4. "Compare Files with the Compare and Merge Tool for XML" on page 228
5. "Merge Files with the Compare and Merge Tool for XML" on page 229
6. "Manage Cross-Project Dependencies" on page 230

RELATED CONCEPTS
"Chapter 8. Team Development" on page 201

RELATED TASKS
"Set up a Team Environment" on page 204
"Work in a Team Environment" on page 212

Export XML

You can export the data in a project's model in XML format. You can export at several different levels of granularity, either from within Object Builder, or from a command line. Once you have exported, you can import the data into another project's model.

You can export information from Object Builder at the following levels:

- The entire project (click **File - Export Model**)
- The entire contents of a folder that contains user-defined objects (from the pop-up menu of the folder)
- A branch of objects defined off a business object file, in the User-Defined Business Objects folder, or just the business object.
- A branch of objects defined off a data object file, in the User-Defined Data Objects folder, or just the data object.
- A schema and persistent object, in the DBA-Defined Schemas folder.

The exported XML file is placed in the project's /Export directory, and named according to the item selected. For example, if you exported the contents of the User-Defined Business Objects folder, the exported file is udbo.xml. If you exported a component with the business object file name ClaimFile, the exported file is ClaimFile.xml.

Once you have exported the file, you can import it into another project. The file conforms to a DTD (document type definition) for Object Builder models.

To export from the command line or batch interface, use the following command:

```
obexport -ProjectDirectory<project_directory> -D<export_directory>
```

If you do not specify an export directory, the \Export subdirectory of the project directory is used.

For example, the command:

```
obexport -Pe:\myproject -De:\NewExport
```

generates the following files into e:\NewExport\ :

- udbo.xml (the contents of the User-Defined Business Objects folder)
- uddo.xml (the contents of the User-Defined Data Objects folder)
- nidl.xml (the contents of the Non-IDL Types folder)
- udschema.xml (the contents of the DBA-Defined Schemas folder)
- dll.xml (the contents of the Build Configuration folder)
- appl.xml (the contents of the Application Configuration folder)
- cont.xml (the contents of the Container Definition folder)

RELATED CONCEPTS

"Model Interchange with XML" on page 203

RELATED TASKS

"Maintain a Team Environment " on page 223
"Import XML" on page 225

Import XML

You can import XML that has been exported from Object Builder. This allows you to transfer information from one project model to another.

This task covers several ways of importing XML:
- From within Object Builder
- Importing two files with circular references
- Importing from the command line to a single project
- Importing to multiple projects with cross-dependencies

XML can be exported from any of the main folders in Object Builder's Tasks and Objects pane. When you import XML, be sure to import it into the same type of folder it was exported from.

The exported XML conforms to a DTD (document type definition) for Component Broker models. When you import an XML file, it is parsed and checked against the DTD using TRLXML, an XML parser from IBM Tokyo Research Laboratory. Only XML that conforms to the DTD can be imported.

If you are importing on top of existing objects (for example, ClaimDO currently exists in Object Builder, and you import ClaimDO.xml), the import process will add to the existing objects (for example, if the XML defines extra attributes, they will be added to the objects), but will not subtract from them (for example, if the XML defines fewer attributes than the object currently has, the object keeps the attributes despite the import).

The current project must contain the information necessary to resolve any references in the XML file, or the references will not be imported (the rest of the XML will be).

For example:
- If you are importing XML that defines a child component, the child's parent component must already be defined in the current project.
- If you are importing XML that defines application configuration, the managed objects configured with the application must already be defined in the current project.
- If you are importing XML that defines a component Customer that references a component Account, the referenced Account component must already be defined in the current project.

Import in Object Builder
To import XML, follow these steps:
1. From a folder's pop-up menu, click **Import**.The Import XML wizard opens to the File Selection Page.
2. Specify the file you want to import. Unless you have moved it, it is in the /Export subdirectory of the project you exported it from.
3. Click **Finish**.

   The data in the XML file is loaded into the current project.
4. Select **File - Save**. The newly imported data is saved to the project's model.

Import Files with Circular References

If you are importing several XML files that contain circular references to each other, you can use the obimport command outside of Object Builder, with the -X option.

For example, to import Customer.xml and Agent.xml (one defines the Customer component, which has a reference to Agent, and the other defines the Agent component, which has a relationship to Customer), place the two XML files in your current project's \Import subdirectory, and then enter the following command:

obimport -X -d=Import e:\myproject

This will import all files in e:\myproject\Import into the project e:\myproject . The -X option is generally used when you are importing into multiple projects, as described later in this topic, but is appropriate for resolving circular references during import to a standalone environment.

Importing from the Command Line to a Single Project

To import from the command line to a single project, use the following command:

```
obimport -P<project_dir> -d<import_dir> <xmfile1 xmfile2...xmlfilen|ALL>
```

The parameters are as follows:

- -P
  The project directory you are importing into.
- -d
  The directory that contains the XML files you are importing. Defaults to the \Import subdirectory of the project directory you specified.
- ALL
  You can list the XML files you want to import, or specify ALL to import all XML files in the specified directory.

For example:

```
obimport -Pe:\myproject -de:\newimports ALL
```

Importing from the Command Line to Multiple Projects with Cross-Dependencies

An alternative syntax for obimport supports import into multiple projects at the same time. This is especially useful for projects that have cross-dependencies, where you would normally have to import some sets of XML files multiple times.

Use the following command syntax to import into multiple projects:

```
obimport -X -d<import_subdirectory> [project1 project2 ... projectn]
```

For each project, the XML files in the import subdirectory are imported, and any cross-project references will be resolved.

The parameters are as follows:

- -X
  Specifies that you want to use the cross-project import syntax.
- -d
  Optionally specifies the project subdirectory to search for XML files. By default, the import looks in the \Export subdirectory of each listed project.

- *project*
  Anything else you specify will be interpreted as the path to a project. The XML files in the \Export subdirectory (or the subdirectory you specified) of each listed project will be imported into the project.

RELATED CONCEPTS
"Model Interchange with XML" on page 203

RELATED TASKS
"Maintain a Team Environment " on page 223
"Export XML" on page 224

Move a Project

To change the location of a project, follow these steps:

1. Move the project directory, and its subdirectories, to its new location.
2. Update the OBMODELPATH environment variable to point to the new directory, so that any other projects that depend on the moved one will still be able to find it.

To set the OBMODELPATH environment variable, use the following command:

```
set OBMODELPATH=[directory1;directory2;...directoryn]
```

For example:

```
set OBMODELPATH=f:\project1;g:\project2
```

RELATED CONCEPTS
"Chapter 8. Team Development" on page 201

RELATED TASKS
"Maintain a Team Environment " on page 223

Change Project Divisions

You can move information from one project to another by exporting the information in XML format, and then importing it into the other project.

When possible, follow these guidelines for transferring information. Otherwise, relationships or references may be automatically deleted during the transfer.

- Move information in logical units (move parent and child components together when possible)
- Move information in logical order (move parent components before child components, move referenced components before referencing components)

To transfer information from one project to another, follow these steps:

1. Select the element that you want to transfer.

   You can transfer all the information in a project, all the information in a folder, information defined off of a business object file in the User-Defined Business Objects folder, or information defined off of a data object file in the User-Defined Data Objects folder.
2. From the element's pop-up menu, click **Export**.

   An XML file corresponding to the element is exported into the project's /Export subdirectory.

3. Delete the element from the project.
4. Save and close the project.
5. Open the target project.
6. From the pop-up menu of the target folder, click **Import XML**.
7. Click **Find** and select the XML file you had exported.
8. Click **Finish**. The information is imported, and appears in the folder.
9. Save and close the project.

RELATED CONCEPTS
"Chapter 8. Team Development" on page 201

RELATED TASKS
"Maintain a Team Environment " on page 223
"Manage Cross-Project Dependencies" on page 230

The Compare and Merge Tool for XML

You can use the Compare and Merge Tool for XML to compare the XML files generated from project models based on node identification, and then merge them. You can decide which differences to include in the resultant, merged file.

You can use the tool in two specific scenarios: you can review changes that you made to a file over a course of time, and you can merge the changes if you want to, or you can use the tool to review and consolidate changes made to a single XML file by different users, who essentially work on the project in a team development environment.

Note: To eliminate any inconsistencies that might exist in the resulting model, you can use another tool called the Model Consistency Checker.

RELATED CONCEPTS
"Chapter 8. Team Development" on page 201"Model Interchange with XML" on page 203

RELATED TASKS
"Compare Files with the Compare and Merge Tool for XML"
"Merge Files with the Compare and Merge Tool for XML" on page 229"Compare Files with the Compare and Merge Tool for XML""Merge Files with the Compare and Merge Tool for XML" on page 229
"Maintain a Team Environment " on page 223
"Import XML" on page 225
"Export XML" on page 224

Compare Files with the Compare and Merge Tool for XML

You can compare XML files at the element level (a level higher than the file level), based on node identification. Follow these steps:

1. Launch the Compare and Merge Tool for XML: use the command `xmldiff` from a command line.
2. Use the menu to import the files to be compared into the tool: from the File Menu, choose Open. The Select Base XML File dialog box opens. Type the name of the base (control) file against which to base your comparison. The tool parses the file.

3. The Select Modified XML File dialog box opens. Type the name of the modified file that you want to compare with the base file. The tool parses the file, and displays a preliminary, combined view of the two files in the Merged View pane. Symbols and color highlight the differences between the two files.

RELATED CONCEPTS
The Compare and Merge Tool for XML
"Model Interchange with XML" on page 203
"Chapter 8. Team Development" on page 201

RELATED TASKS
"Merge Files with the Compare and Merge Tool for XML"

Merge Files with the Compare and Merge Tool for XML

Once you have a display of the combined XML files in the Merged View pane, you can walk through the changed nodes and decide whether the change should be incorporated in the merged file from either the base file, or the phase file.
Every modified node in the tree has an associated pop-up menu, with choices that enable you to implement the decision whether to incorporate properties from either the base file, or the modified file.

The new nodes have the following pop-up menu choices:

- **Do not use new**: the new node is not incorporated in the merged file.
- **Use new element**: the merged file has the new node, and its children, if any, as they are in the modified file.

The deleted nodes have the following pop-up menu choices:

- **Do not delete**: the merged file has the node as it exists in the base file.
- **Delete from base file**: the merged file does not have the node that was deleted from the base file.

The changed nodes have the following pop-up menu choices:

- **Use old, where conflict**: the merged file has the nodes as they are in the base file for the current node, and any of its unresolved children (those modified child nodes for which you have not made a decision about incorporation in the merged file yet).
- **Use new, where conflict**: the merged file has the nodes as they are in the modified file for the current node, and any of its unresolved children.

These choices are also available from the Selected menu.
To merge the two XML files, follow these steps:

1. Select one of the highlighted nodes in the merged tree view.
2. Select **Use modified file for node and unresolved children** from the pop-up menu of the node if you want to have the merged file incorporate the changes that were made in the modified XML file. Select **Use base file for node and unresolved children** from the pop-up menu of the node if you want to have the merged file have the older version of the corresponding node.
3. Use **Edit - Undo** to undo all of your actions up to the last time you saved your work.

The menu selection you make on a node is applied to the node, as well as to all its unresolved child nodes.

For example, if you select a new node to be part of the merged file (**Use modified file for node and unresolved children**), then all its children will also be in the merged file; in addition, if this was the only node with a conflict under its parent, then the parent would be marked as resolved (its red cross-bar will disappear). Similarly, if you use the phase file as the source for change propagation for a deleted node, the node and all its children will not be present in the merged file. Whenever the changes of all the child nodes are resolved, the parent node and all its child nodes will have a check mark in front of them.

RELATED CONCEPTS
Compare and Merge Tool for XML

RELATED TASKS
"Compare Files with the Compare and Merge Tool for XML" on page 228

Manage Cross-Project Dependencies

Each project maintains its own list of dependencies. The list covers both the dependencies it has on other projects (displayed in its Open Project wizard, Project Dependencies Page), and the dependencies other projects have on it (displayed in the Project Dependencies Page of other projects).

When you create a dependency from one project on another, the dependency is added in the dependencies files for both projects.

When you open a project that has dependencies, the models of the projects depended on are opened in read-only mode. The dependency files for the projects are opened in read-write mode.

When you delete a dependency from a project, its listing is removed from the dependencies files for both projects (the dependent project and the depending project).

To avoid managing the dependency files when you move a project or change the directory structure, use the OBMODELPATH environment variable to list the directories of all projects in your application.

To set the OBMODELPATH environment variable, use the following command:

```
set OBMODELPATH=[directory1;directory2;...directoryn]
```

For example:

```
set OBMODELPATH=f:\project1;g:\project2
```

The directories you list, and their subdirectories, will be searched for project dependencies whenever you open an Object Builder project.

Note: The more directories Object Builder searches, the longer it will take to open projects. Try to achieve a compromise between completeness (searching all appropriate project directories) and speed (avoid listing the root directory of every drive).

If you are importing XML that contains cross-project references, you can use the obimport command with the -X option to import the XML for all the affected projects at once, while preserving the cross-project references.

For example:

```
obimport -X -d=Import e:\myRBprojects\projA e:\myRBprojects\projB
e:\myRBprojects\projC
```

This command looks in the Import subdirectory of each listed project directory, and imports the cont.xml, udbo.xml, uddo.xml, udschema.xml, dll.xml, and appl.xml files it finds there.

RELATED CONCEPTS

"Chapter 8. Team Development" on page 201

RELATED TASKS

"Maintain a Team Environment " on page 223
"Import XML" on page 225
"Move a Project" on page 227

Chapter 9. XML Wizards

When you create a complex XML document, one of the standard authoring strategies is to look at an example document first, and then re-use its structure and content, customizing only the parts that you need. In this way you start with a valid structure that roughly meets your needs, and then extend or change it only as necessary. This reduces the time you need to learn a DTD before working in it, and makes it both quicker and easier to create valid, useful XML documents.

This process can be made even simpler and more repeatable by creating a wizard as an interface to editing the example document. You can create an XML wizard, or SmartGuide, using the SmartGuide Customizer for XML. The wizard or SmartGuide allows you to selectively add and edit element types in the document.

Begin the process of creating an XML wizard by identifying or creating the sample XML file. You can then open the file in the SmartGuide Customizer for XML, and explicitly mark those sections you want to change or extend. When you are done, you can generate an XML wizard script. When you run the script, the wizard exposes the elements you chose to be editable, and applies your edits to create a new document based on the original. The new document is extended only in the ways you selected; the structure and context of the original file is preserved. All your changes are applied through the wizard, without editing the source directly.

Once you have created the wizard, anyone can use it to create new documents following the pattern you set, without having to understand the XML DTD at all, or ever work in XML directly. You can also add help text and fly-over text to the wizard to make it even easier to use.

RELATED CONCEPTS

"Model Interchange with XML" on page 203

RELATED TASKS

"Create an XML Wizard"
"Export XML" on page 224
"Import XML" on page 225

Create an XML Wizard

You can use an XML wizard, or SmartGuide, to allow selective editing and extension of an XML file through a wizard interface that provides constraints, descriptions, fly-over help, and HTML help. The wizard script can include customized default values, derivation relationships between values, and customized lists of selectable values.

To create an XML wizard, you first need an example XML file, that you can use as a template for the output the wizard will generate. Once you have the example XML file, open it in the SmartGuide Customizer for XML, and begin working with its elements.

To create an XML wizard, you can follow these steps:

1. "Start the SmartGuide Customizer for XML" on page 234
2. "Define XML Wizard Macros" on page 235

© Copyright IBM Corp. 1997, 1998

3. “Customize Value Lists in an XML Wizard” on page 237
4. “Derive Values in an XML Wizard” on page 237
5. “Propagate Values in an XML Wizard” on page 239
6. “Constrain Values in an XML Wizard” on page 240
7. “Define the Layout of an XML Wizard” on page 242
8. “Test an XML Wizard” on page 243

Once you have created the XML wizard, you can run it to produce new XML files based on your original template.

You can work with existing XML wizards in the following ways:

- “Run an XML Wizard” on page 243
- “Edit an XML Wizard” on page 244
- “Distribute an XML Wizard” on page 245

RELATED CONCEPTS
“Chapter 9. XML Wizards” on page 233
“Model Interchange with XML” on page 203

RELATED TASKS
“Export XML” on page 224

Start the SmartGuide Customizer for XML

You can use the SmartGuide Customizer for XML to build an XML wizard, or SmartGuide, for creating XML documents.

Before you start the Customizer, you should have a sample of the XML document type you want your XML wizard to create. You will use this sample as a template, which the XML wizard’s output will be based on.

To start the Customizer, follow these steps:

1. Locate or create the sample XML file you want to start with. The sample file must include its associated XML DTD, or point to a place where the DTD is available.
2. Run the SmartGuide Customizer. From the command line, type the command:
   ```
   xmlcustm
   ```
3. In the Customizer, click **File - Open** and select the example XML file.

   The Customizer parses the XML document, and displays its content as a tree of element nodes in the left-hand pane.

You are now ready to begin identifying the elements your XML wizard will work with.

RELATED CONCEPTS
“Chapter 9. XML Wizards” on page 233

RELATED TASKS
“Create an XML Wizard” on page 233
“Define XML Wizard Macros” on page 235

Define XML Wizard Macros

When you load an XML file in the SmartGuide Customizer, you see the structure of the document displayed in a tree view in the left-hand pane of the tool. This structure contains two types of node:

- Container element nodes
  These are XML elements that organize sub-elements with content, or have attributes with content, but do not have their own content aside from this.
- Element content or attribute nodes
  These are either the content of an XML element, or the value of an element attribute. They appear under a container element. Content nodes are labeled as Text in the tree view. Content nodes only appear when there is actual content in the sample XML file; if the element in the sample file has no content, then it does not have a content node in the SmartGuide Customizer.

When you click on a container element, the right-hand pane enables settings for you to define a wizard page for that element's contents. By default, none of the container elements have wizard pages associated with them.

You can also select whether the element structure is repeatable. If you check the **Repeatable** option, then the user will be able to add multiple instances of the selected element, using a tree view control on the wizard page. Each instance the user adds will have the editable properties you set for the element's content or attributes.

When you click on an element's content or attribute, the right-hand pane enables settings for you to make the content or attribute a macro. When you set an element's content or attribute to be a macro, its value becomes part of the wizard's XML script.

Macros are also automatically defined when you create a derivation or propagation relationship between elements.

To define an element's content or attribute as a simple wizard macro (without derivation or propagation), follow these steps:

1. In the left-hand pane, click on a content or attribute node. Its settings appear in the right-hand pane.
2. From the **Macro** pulldown, select one of the following:
   - **None**
     All settings for the content or attribute are ignored, and the original value from the source XML file is used instead.
   - **Hidden**
     The content or attribute value will not appear in the wizard interface, but will be used internally by the wizard (for example, it might have a derived value).
   - **Editable**
     The content or attribute value appears in the wizard interface, and is editable by the wizard user.
   - **Read-only**
     The content or attribute value appears in the wizard interface, but is not editable by the wizard user.
3. Type a label for the content or attribute in the **Label** field. If you are creating an Editable or Read-only macro, this label appears in the wizard interface as the label for the associated value (for example, **Name**:).

4. Type a default value for the content or attribute in the **Default** field's **Value** section. If you do not change the default value, the value from the original sample file is used.

   If the definition for the element or attribute in the XML DTD prescribes a list of valid values, then the **Value** section becomes a drop-down list which you can select valid values from. If the macro type is **Editable**, then the wizard user will be presented with this list as well. You can customize the terms used in the list (but should not change the underlying values). Click **Values** to customize the terms for the wizard user, as described in the customizing value lists task.

   Even if the definition for the element or attribute allows any content type, you can limit the user's choices to a set of values that make sense for the wizard's intended use. If the macro type is **Editable**, the user will be presented with a drop-down list that contains only the values you specify. You can create or customize the value list the user sees by clicking **Values**, as described in the customizing value lists task.

   Ignore the prefix and suffix sections: they are for use when you derive or propagate a value, as described in the derivation and propagation tasks.

   If the macro type is **Hidden** or **Read-only**, then this value is always applied in the wizard's output.

   If the macro type is **Editable**, then it appears in the wizard interface as a default, and can by typed over by the wizard user.
5. If the macro type is **Editable**, select the constraint (if any) you want to apply to the user's input. You can select from the following values:
   - NoSpace
   - C++
   - CORBA
   - SQL
   - LongFile
   - File83
   - File8
   - Any
   - Action

   You can also define your own constraints, as described in the constraining values task.
6. If the macro type is **Editable**, type a brief description of the element content or attribute in the **Fly-Over Help** field. This description will appear when the wizard user moves the mouse pointer over the field.

RELATED CONCEPTS

"Chapter 9. XML Wizards" on page 233

RELATED TASKS

"Create an XML Wizard" on page 233
"Customize Value Lists in an XML Wizard" on page 237
"Derive Values in an XML Wizard" on page 237
"Propagate Values in an XML Wizard" on page 239
"Constrain Values in an XML Wizard" on page 240

Customize Value Lists in an XML Wizard

When you define a macro of type **Editable** for an element content or attribute, the wizard user will be able to enter a value for that content or attribute in the wizard. You can limit the user's choices to a set of values that make sense for the wizard's intended use. These values will be displayed in a drop-down list in the wizard interface.

If an element content or attribute already has a set of acceptable values enumerated in the DTD, the SmartGuide Customizer for XML displays them in a drop-down list by default. You can customize the terms used in the list to make them more descriptive for your wizard users, but should not change the underlying values; otherwise the XML wizard will generate XML that is not valid.

To create or customize a value list, follow these steps:

1. In the SmartGuide Customizer tree view, click on the element's content or attribute whose value list you want to customize.

   In the right-hand pane, the properties of the selected node appear. The **Default** field's **Value** section should be a drop-down list. If the section is an entry field (no drop-down arrow), then the content or attribute does not have a list of acceptable values in the XML DTD, and there is no value selection list to customize.
2. Click the **Values** button. The Customize Value List dialog opens.

   The existing value appears in the list as the default. Any additional valid values defined in the DTD appear as well.
3. If the DTD defines a list of valid values, you can change the terms used in the list, or remove terms from the list, but should not change the underlying values or add new values that are not compliant with the DTD definition.
4. If the DTD does not define a list of valid values, you can change or add new values, and customize the terminology, as required.
5. Select which term you want to be selected by default.
6. Click **OK**. The **Value** section of the **Default** field becomes a drop-down list, if it wasn't already.

When you create the XML wizard, the wizard user will be able to select a value for this element content or attribute from the list you created. The user will see the terms you specified, which will map to the underlying values according to the mapping you created.

RELATED CONCEPTS

"Chapter 9. XML Wizards" on page 233

RELATED TASKS

"Create an XML Wizard" on page 233
"Define XML Wizard Macros" on page 235
"Constrain Values in an XML Wizard" on page 240

Derive Values in an XML Wizard

When you create an XML wizard with the SmartGuide Customizer for XML, you can derive the value for one element's content or attribute from the value given for another element's content or attribute, rather than identifying the values separately.

For example, you could make the first value editable, and then the second, derived value be a simple reflection of what the user enters for the first value, with the addition of a prefix or suffix.

This task deals with creating a derivation relationship, starting with the derived element value and specifying what its source should be. You can also work in the other direction to create a propagation relationship, starting with an existing value and propagating it to a number of deriving elements.

There are two ways you can create a derivation relationship:

- Using the implicit derivation rules in your XML sample document, based on the comparison of content strings in the different elements.
- Making an explicit association, regardless of the content strings in the sample document.

To create a derivation relationship based on the existing content of the sample document, follow these steps:

1. Select the element content or attribute for which you want to specify a source.
2. In the **Default** field’s **Value** section, type the substring of its value that was derived, in the original XML file.

   For example, if you know that the attribute’s current value PersonBO is derived from another attribute’s value Person, then decompose the value into **Value** of Person and **suffix** of BO. You can then search to find all other attributes or element content with the value Person, and select one of them as the source to be derived from.
3. In the toolbar, click the **Derive** button. A Derive Value dialog appears, and highlights the first node in the tree view that has the value you listed.
4. Click **Find Next** or **Find Previous** to navigate through all the nodes in the tree that have the listed value, and are valid sources for a derivation relationship.
5. When you have found the node you want to derive from, click the **Derive From** button.

   The derivation relationship is created. The **Derived value** option is checked, and the selected source node is marked as a macro. The **Default** field’s **Value** section is greyed out, to prevent you from editing the derived value.
6. Mark the content or attribute as a macro (**Hidden**, **Editable**, or **Read-only**). The relationship will be implemented in the XML wizard.

To create a derivation relationship regardless of existing content, follow these steps:

1. In the tree view, locate the element content or attribute that you want to specify as derived.
2. From the pop-up menu of the node, click **Derive Value From**.

   A Derivation Tree appears. You can select any node in the tree as the source to derive from. You should ensure that the node you select to derive from has a value associated with it, and that the value is of an appropriate type to act as the source for the deriving value.
3. Select the node you want to derive from.
4. Click **OK**.

   The derivation relationship is created. The **Derived value** option is checked, and the selected source node is marked as a macro. The **Default** field’s **Value** section is greyed out, to prevent you from editing the derived value.
5. Fill in any prefix or suffix that you want to apply to the derived value.

6. Mark the content or attribute as a macro (**Hidden**, **Editable**, or **Read-only**). The relationship will be implemented in the XML wizard.

RELATED CONCEPTS
"Chapter 9. XML Wizards" on page 233

RELATED TASKS
"Create an XML Wizard" on page 233
"Define XML Wizard Macros" on page 235
"Propagate Values in an XML Wizard"

Propagate Values in an XML Wizard

When you create an XML wizard with the SmartGuide Customizer for XML, you can derive the value for one element from the value given for another element, rather than identifying the values separately. In other words, you can base the value for an element content or attribute on the value given for another element's content or attribute within the same XML document. For example, you could make the first value editable, and then the second, derived value be a simple reflection of what the user enters for the first value, with the addition of a prefix or suffix.

This task deals with creating a propagation relationship, starting with the source content or attribute value and specifying what other values are derived from it. You can also work in the other direction to create a derivation relationship, starting with derived values and specifying where the value should be derived from.

There are two ways you can create a propagation relationship:
- Using the implicit propagation rules in your XML sample document, based on the comparison of content strings in the different elements.
- Making an explicit association, regardless of the content strings in the sample document.

To create a propagation relationship based on the existing content of the sample document, follow these steps:
1. Select the element content or attribute whose value you want to propagate.
2. In the toolbar, click the **Propagate** button. A Propagate Value dialog appears, and highlights the first node in the tree view that has the selected value.

   For example, if the current value is Person, then you can search through all other nodes whose value includes that substring (PersonBO, PersonDOImpl, iPersonPO, and so on).
3. Click **Find Next** or **Find Previous** to navigate through all the nodes in the tree that have the listed value, and are valid targets for a propagation relationship.
4. When you have found a node you want to propagate to, click the **Propagate To** button.

   A propagation relationship is created. The selected target's **Derived value** option is checked, and the source node is marked as a macro. The selected target's **Default** field's **Value** section is greyed out, to prevent you from editing the derived value.

   The Propagate Value dialog remains open, for you to select additional nodes to propagate to.
5. When you have finished propagating values, click **Cancel** to close the dialog.
6. Edit each propagation target:

a. Fill in any prefix or suffix that you want to apply to the derived value.
b. Mark the target node as a macro (**Hidden**, **Editable**, or **Read-only**). The relationship will be implemented in the XML wizard.

To create a propagation relationship regardless of existing content, follow these steps:

1. In the tree view, locate the element content or attribute whose value you want to propagate.
2. From the pop-up menu of the node, click **Propagate Value To**.

   A Propagation Tree appears. You can select any nodes in the tree as targets to propagate values to. You should ensure that the nodes you select to propagate to have values associated with them, and that the values are of an appropriate type to act as the target for your source value.
3. Select the nodes you want to propagate to.
4. Click **OK**.

   The propagation relationships are created. The selected targets' **Derived value** options are checked, and the source node is marked as a macro. The selected targets' **Default** fields' **Value** sections are greyed out, to prevent you from editing the derived value.
5. Edit each propagation target:
   a. Fill in any prefix or suffix that you want to apply to the derived value.
   b. Mark the target node as a macro (**Hidden**, **Editable**, or **Read-only**). The relationship will be implemented in the XML wizard.

RELATED CONCEPTS

"Chapter 9. XML Wizards" on page 233

RELATED TASKS

"Create an XML Wizard" on page 233
"Define XML Wizard Macros" on page 235
"Derive Values in an XML Wizard" on page 237

Constrain Values in an XML Wizard

When you define an XML wizard macro as **Editable** in the SmartGuide Customizer for XML, you can also select a constraint to apply to it. This will prevent the wizard user from entering a value outside the selected constraint.

The following constraints are provided by default:

- NoSpace
- C++
- CORBA
- SQL
- LongFile
- File83
- File8
- Any
- Action

You can provide your own constraint by creating a Java class that implements the Constraint interface, provided with the SmartGuide Customizer for XML. To apply

the constraint, select the **Actions** option in the **Constraints** field, and then type over the selection with the name of the class. When the XML wizard runs, it will look for a Java class with that name, and call its test() function with the value the user entered as a parameter.

RELATED CONCEPTS

"Chapter 9. XML Wizards" on page 233

RELATED TASKS

"Create an XML Wizard" on page 233
"Define XML Wizard Macros" on page 235
"Customize Value Lists in an XML Wizard" on page 237

XML Wizard Constraints

When you create a wizard, or SmartGuide, with the SmartGuide Customizer for XML, you set which elements will be editable in the wizard. These elements are exposed in the wizard as fields, in which the wizard user can enter values.

When you set the element as editable, you can also set constraints that will be applied to the field, to limit the user to certain value types or formats.

You can set one of the following constraints:

- **NoSpace**
  No spaces are allowed in the value.
- **C++**
  The value must be a valid C++ type.
- **CORBA**
  The value must be a valid CORBA type.
- **SQL**
  The value must be a valid SQL type.
- **LongFile**
  The value must be a valid file name, for a system that supports long file names.
- **File83**
  The value must be a valid file name, for systems that have a maximum file name length of 8, with a maximum file extension length of 3.
- **File8**
  The value must be a valid file name, for systems that have a maximum file name length of 8, and the value must not include a file extension.
- **Any**
  There are no constraints on the value the user enters.
- **Action**
  Replace the selection with the name of a Java class that provides a constraint you defined. The class must implement the Constraint interface, provided with the SmartGuide Customizer for XML. When the XML wizard runs, it will look for a Java class with that name, and call its test() function with the value the user entered as a parameter.

RELATED CONCEPTS

"Chapter 9. XML Wizards" on page 233

RELATED TASKS

"Create an XML Wizard" on page 233

"Define XML Wizard Macros" on page 235
"Constrain Values in an XML Wizard" on page 240

Define the Layout of an XML Wizard

The XML wizard you create, using the SmartGuide Customizer for XML, will walk through all the macros you identify, displaying entry fields or drop-down lists for macros that are **Editable** and displaying read-only text for macros that are **Read-only**.

Macros are generally grouped by the element that contains them. For each element that contains macros, you can select whether to start a new page for the contained and any subsequent macros.

To define an XML wizard page, follow these steps:

1. In the SmartGuide Customizer tree view, click on the element you want to define a page for.
2. In the properties pane, select whether the element is **Repeatable**.

   If you make an element repeatable, then the wizard will display a tree view for the element, to which the user can add instances of the element. Each element instance will have its own set of values, as defined in the SmartGuide Customizer. Values marked as **Editable** are editable by the user, on a per-instance basis.
3. In the properties pane, click the **Start new page** option.

   The values of the current element will now be on a new page. Values of subsequent elements will also appear on the current page, until the next element defined as the start of a page.

   Generally, if you make an element **Repeatable**, it should have its own page. In other words, it should have the **Start new page** option checked, and the next element that contains **Editable** or **Read-only** macros should also have the **Start new page** option checked.
4. Type a title for the new page in the **Title** field.
5. Type a description for the new page in the **Description** field. The description appears directly below the title, and above any editing controls for the element contents and attributes on the page.
6. Type a URL for an HTML file that provides help for the page in the **Help URL** field. The URL can be absolute (for example, http://mycompany.intranet/product/wizard1/NamePage.html) or relative to the location of the wizard macro script (for example, help/NamePage.html). This URL will be associated with the **Help** button on the wizard page, and the HTML file will be loaded in the user's default web browser when the user clicks **Help**.

Once you have defined the layout, you can view the results from within the SmartGuide Customizer by testing the XML wizard.

RELATED CONCEPTS
"Chapter 9. XML Wizards" on page 233

RELATED TASKS
"Create an XML Wizard" on page 233
"Define XML Wizard Macros" on page 235
"Test an XML Wizard" on page 243

Test an XML Wizard

Once you have selected the elements you want represented in the XML wizard, and customized the XML wizard's layout, you can save the XML macro file for the wizard, and test it through the SmartGuide Customizer's interface.

To generate the XML wizard script, follow these steps:

1. Select **File - Save**.
2. Select a location in which to save the file.
3. Type the name of the file as *name*.xml.
4. Save the file.

Once you have generated the script, and before you use it to create new files, you can test it from within the SmartGuide Customizer. To test the script, follow these steps:

1. Select **File - Test**.
2. Run through the wizard pages, and review the result of your layout selections in the SmartGuide Customizer.
3. Click **Finish**.

   You are prompted for the location of the original XML file (on which the wizard is based), and a path and file name for the resulting wizard-generated XML file.
4. Provide the names and click **Finish**, or click **Cancel** to return to the SmartGuide Customizer without saving the results of your test.

You can also run the wizard script from the command line, by running the SmartGuide Launcher for XML (type xmllaunch on the command line).

RELATED CONCEPTS

"Chapter 9. XML Wizards" on page 233

RELATED TASKS

"Create an XML Wizard" on page 233
"Define XML Wizard Macros" on page 235
"Define the Layout of an XML Wizard" on page 242
"Run an XML Wizard"

Run an XML Wizard

Once you have created an XML wizard script in the SmartGuide Customizer for XML, you can run the wizard using the SmartGuide Launcher for XML, and use the wizard to create new XML documents.

In order to run an XML wizard, you need the following:

- The XML wizard script, generated by the SmartGuide Customizer for XML.
- Any help files for the wizard script, if they are linked using a relative path (rather than, for example, an http address on your intranet).
- The original XML file on which the wizard script is based.
- The DTD for the original XML file, either contained in, or referenced by, the original file. If the DTD is referenced using a relative path, the path is resolved relative to the current directory (the directory from which the SmartGuide Launcher tool is run).

- Any classes that provide customized input constraints.
- The SmartGuide Launcher tool, which runs the script.

To run an XML wizard, follow these steps:

1. On the command line, enter the following command:

   xmllaunch

   The SmartGuide Launcher wizard opens.
2. Type the name of the wizard script you want to run.
3. Click **Finish**.

   The XML wizard opens.
4. In the XML wizard, follow the prompts to add elements and edit element values. Fly-over help, and HTML help for each page, are available if they were defined in the SmartGuide Customizer.
5. Click **Finish**.

   You are prompted for the location of the original file on which the script is based, and a location in which to save the new document generated by the wizard.
6. Provide the location of the original source XML file.
7. Provide a name and path in which to save the new XML document, which is based on that original.
8. Click **Finish**.

The XML file is created, with the name and path you specified.

RELATED CONCEPTS

"Chapter 9. XML Wizards" on page 233
"Model Interchange with XML" on page 203

RELATED TASKS

"Import XML" on page 225
"Create an XML Wizard" on page 233
"Edit an XML Wizard"
"Distribute an XML Wizard" on page 245

Edit an XML Wizard

You can customize an XML wizard by loading its XML script file into the SmartGuide Customizer for XML, selecting elements to expose in the wizard interface, setting how they will be exposed, and then re-generating the XML wizard script.

To open an existing XML wizard script in the SmartGuide Customizer, follow these steps:

1. Run the SmartGuide Customizer. From the command line, type the command:

   ```
   xmlcust
   ```
2. Open the script file in the SmartGuide Customizer. Click **File - Open** and select the file.

   The SmartGuide Customizer recognizes the XML file as a script or macro file, and displays it in terms of its original source document structure (rather than interpreting the contents of the file literally), with the macros applied as a set of modifications and selections.

3. Edit the macros, and create new ones, in the same you would when creating a new XML wizard.
4. Click **File - Save** to save the XML script with your changes applied.

RELATED CONCEPTS
"Chapter 9. XML Wizards" on page 233

RELATED TASKS
"Create an XML Wizard" on page 233
"Run an XML Wizard" on page 243

Distribute an XML Wizard

Once you have created and tested the XML wizard, you can package it for use by others.

Each package should include the following:
- The XML wizard script, generated by the SmartGuide Customizer for XML.
- Any help files for the wizard script, if you linked to the files using a relative path (rather than, for example, an http address on your intranet).
- The original XML file on which the wizard script is based.
- The DTD for the original XML file, either contained in, or referenced by, the original file. If the DTD is referenced using a relative path, the path is resolved relative to the current directory (the directory from which the SmartGuide Launcher tool is run).
- Any classes you defined to provide customized input constraints.
- The SmartGuide Launcher tool, which runs the script.

The relative paths from the wizard script's location to the original XML file and its DTD should be preserved in the package.

RELATED CONCEPTS
"Chapter 9. XML Wizards" on page 233

RELATED TASKS
"Create an XML Wizard" on page 233
"Constrain Values in an XML Wizard" on page 240
"Test an XML Wizard" on page 243
"Run an XML Wizard" on page 243

Chapter 10. Object Development Tasks

Work with Attributes

Component attributes are defined in the business object interface. You can also define attributes for specific component objects (business object implementations, data object interfaces, data object implementations).

The get and set methods for component attributes are defined in the business object implementation and data object implementation. The mapping between attributes and datastores is accomplished using special framework methods in the data object and persistent objects.

The following tasks deal with attributes:

- “Add an Attribute”
- “Edit an Attribute” on page 248
- “Map Data Object Attributes to Persistent Object Attributes” on page 256
- “Delete an Attribute” on page 249

RELATED CONCEPTS
“Attributes” on page 26

RELATED TASKS
“Work with Methods ” on page 267

Add an Attribute

You can explicitly define attributes in the business object interface, or in data object interfaces that are not associated with a business object. You can also add implementation-only attributes to a business object implementation or data object implementation. Implementation-only attributes are not exposed in the component’s managed object.

When you add objects from an interface, any elements necessary to support the interface’s attributes are added automatically. When you add a business object implementation and a data object interface in a single step, you do not need to define the data object attributes separately: you can select the data object attributes from a list of the existing business object attributes.

To define new attributes in an existing component, add them in the business object interface, edit the key and copy helper if you want the attribute to be used in those objects, and then edit the business object implementation to make it part of the data object. The changes are applied automatically to the implementations.

To add an attribute to an existing interface, follow these steps:

1. From the interface’s pop-up menu, click **Properties**.
2. In the interface’s wizard, click the title bar and turn to the Attributes page.
3. From the pop-up menu of the Attributes folder on that page, click **Add**.
4. Define the attribute.
5. Click **Refresh**. The attribute is added to the Attributes folder.
6. Click **Finish**.

© Copyright IBM Corp. 1997, 1998

To make the new attribute part of an associated data object, follow these steps:

1. From the business object implementation's pop-up menu, click **Properties**.
2. In the implementation's wizard, click the title bar and turn to the Data Object Interface page.
3. Move the attribute from the Business Object Attributes list to the State Data list.
4. Click **Finish**. The attribute is added to the data object, including its data object implementation.

If appropriate, you can also edit the key and copy helper associated with the interface, and add the new attribute to them.

RELATED CONCEPTS

"Attributes" on page 26

RELATED TASKS

"Work with Attributes" on page 247
"Edit an Attribute"
"Edit a Business Object Interface" on page 290
"Edit a Data Object Interface" on page 309

Edit an Attribute

When you edit attributes in an existing component, you must edit them in the business object interface. The change is automatically applied to the other objects in the component.

You can also edit an attribute in a data object interface, if it is not yet connected to a business object implementation.

To edit an attribute, follow these steps:

1. From the interface's pop-up menu, click **Properties**.
2. In the interface's wizard, click the title bar and turn to the Attributes page.
3. Select an existing attribute under the Attributes folder.
4. Edit the properties of the attribute.
5. Click **Refresh**. The changes are applied.
6. Click **Finish**.

The change is automatically applied to the equivalent attribute in any related key, copy helper, implementation, or data objects.

RELATED CONCEPTS

"Attributes" on page 26

RELATED TASKS

"Work with Attributes" on page 247
"Delete an Attribute" on page 249
"Edit a Business Object Interface" on page 290
"Edit a Data Object Interface" on page 309

Delete an Attribute

When you delete attributes from a component, you must delete them in the business object interface. References to the attribute in the rest of the component, and in other components will be automatically removed.

You can also delete an attribute from a data object interface, if it is not yet connected to a business object implementation.

To delete an attribute, follow these steps:

1. From the interface's pop-up menu, click **Properties**.
2. In the interface's wizard, click the title bar and turn to the Attributes page.
3. Select an existing attribute under the Attributes folder.
4. From the attribute's pop-up menu, click **Delete**. The attribute is removed.
5. Click **Finish**.

RELATED CONCEPTS

"Attributes" on page 26

RELATED TASKS

"Work with Attributes" on page 247
"Edit an Attribute" on page 248
"Edit a Business Object Interface" on page 290
"Edit a Data Object Interface" on page 309

Map a Data Object to a PA Persistent Object

Mapping a data object to a persistent object consists of mapping of attributes and methods from one object to the other. Mapping of attributes and methods is required to define the bonding between the objects. A data object attribute can be mapped to one or more persistent object attributes and each special framework method of the data object can be mapped to one or more persistent object methods.

Restrictions:

- You cannot map multiple data object attributes to the same persistent object attribute.
- When you map a data object to multiple persistent objects, you must map each key attribute of the data object directly to each of the key attributes of the different persistent objects.

These are the preliminary steps you must follow before you can map a data object to a persistent object:

1. Create a PA schema and its associated PA persistent object by importing a PA bean.
2. Add a customized PA persistent object to the PA schema if you do not want to use the one Object Builder provides.
3. Add a data object implementation (The environment for the implementation should be **Procedural Adaptors**.)

Note the following points:

- To map a data object to a persistent object, there must be an association between the two objects, which you specify on the Associated Persistent Objects Page of the Data Object Implementation wizard.
- As soon as you associate a persistent object with the data object, the Attributes Mapping Page and the Methods Mapping Page are dynamically added to the wizard.

To define the mapping between the attributes of the data object and the persistent object, follow these steps:

1. If you are in the process of defining the data object implementation, proceed with step 2. If you have already defined the data object implementation, from the data object implementation's pop-up menu, select **Properties**. The Data Object Implementation wizard opens.
2. Turn to the Attributes Mapping Page. Here, you can map the data object interface attributes to the attributes of the persistent object.

You can map a data object attribute to a persistent object attribute in one of the following ways:

- Using the primitive pattern
- Using the exploded mapping pattern (for structures, which are complex attributes)
- Using a foreign key
- Using a mapping helper

To define the mapping between the methods of the data object and the persistent object, follow these steps:

1. If you are in the process of defining the data object implementation, proceed with step 2. If you have already defined the data object implementation, from the data object implementation's pop-up menu, select **Properties**. The Data Object Implementation wizard opens.
2. Turn to the Methods Mapping Page. When you define the mapping between methods, you define the processing order of the persistent object methods that you associate with the data object's special framework methods insert(), update(), retrieve(), del(). These persistent object methods act directly on elements of transaction logic in the legacy business applications.
3. The methods that you defined for the data object appear in the User-Defined Methods folder. You can map each of them to a push-down method of the PA persistent object.

RELATED CONCEPTS

"Data Object" on page 18
"Persistent Object" on page 19
"Special Framework Methods" on page 24
"User-Defined Methods" on page 23
"Push-Down Methods" on page 25

RELATED TASKS

"Add a Data Object Implementation" on page 299
"Work with PA Schemas - Overview" on page 337
"Map Data Object Attributes to Persistent Object Attributes" on page 256
"Create a Relationship" on page 129
"Work with Methods " on page 267
"Use Push-Down Methods with PA Persistent Objects" on page 274

RELATED REFERENCES
"DB2 Data Type Mappings" on page 110
"Oracle Data Type Mappings" on page 113

Map a Data Object to a DB Persistent Object

There are certain stages in development in which you can map a data object to a persistent object:

- When you associate a persistent object in the model with the data object implementation you are creating (meet-in-the-middle).
  Note the following points:
  – The persistent object has to use the same type of persistence as the data object implementation.
  – You can customize the mapping of both attributes and special framework methods of the data object to relevant attributes and methods of the persistent object. Object Builder does not do the default mapping.
- When you create a persistent object and schema from a data object implementation (top-down).
  Note: In this case you can customize only the mapping of the attributes of the two objects. The default mapping of attributes is done for you.
- When you create a data object from a persistent object. (bottom-up)
  Note: As in the meet-in-the-middle case, you can customize the mapping of both attributes and special framework methods of the data object to relevant attributes and methods of the persistent object. In the bottom-up case, however, Object Builder does the default mappings for you.

Restrictions:

- Multiple data object attributes cannot be mapped to the same persistent object attribute.
- When you map a data object to multiple persistent objects, you must map each key attribute of the data object directly to each of the key attributes of the different persistent objects.

Meet-in-the-middle
These are the preliminary steps you must follow before you can map a data object to a DB persistent object, when you associate a persistent object in the model to the data object implementation being created.

1. Create a schema by importing an SQL file.
2. Add a persistent object to the schema.
3. Add a data object implementation. The environment for the implementation must be **BOIM with any key**, and the implementation must not be transient (select any option except the **Transient** option from the"Form of Persistent Behavior and Implementation" on page 32 section).

Note the following points:

- To map a data object to a persistent object, there must be an association between the two objects, which you specify on the Associated Persistent Objects Page of the Data Object Implementation wizard.
- As soon as you associate a persistent object with the data object, the Attributes Mapping Page and the Methods Mapping Page are dynamically added to the wizard.

To define the mapping between the attributes of the data object and the persistent object, follow these steps:

1. If you are in the process of defining the data object implementation, proceed with step 2. If you have already defined the data object implementation (you want to map the persistent object to a data object that exists in the model, and for which an implementation has been created), from the data object implementation's pop-up menu, select **Properties**. The Data Object Implementation wizard opens.
2. Go to the Associated Persistent Objects Page. Select the persistent object that you added to the schema that you just imported.
3. Go to the Attributes Mapping Page. Here, you can map the data object interface attributes to the attributes of the persistent object.

You can map a data object attribute to a persistent object attribute in one of the following ways:

- Using the primitive pattern
- Using the exploded mapping pattern (for structures, which are complex attributes)
- Using a foreign key
- Using a mapping helper

To define the mapping between the methods of the data object and the persistent object, follow these steps:

1. If you are in the process of defining the data object implementation, proceed with step 2. If you have already defined the data object implementation, from the data object implementation's pop-up menu, select **Properties**. The Data Object Implementation wizard opens.
2. Turn to the Methods Mapping Page. When you define the mapping between methods, you actually "Customize Referential Integrity" on page 108: you define the processing order of the persistent object methods that you associate with the data object's special framework methods insert(), update(), retrieve(), del() and setConnection(). These persistent object methods act directly on data in the persistent store (tables in the database).
3. Click **Finish**.

Attention:
If a persistent object is not created from this implementation but was created from another implementation and is used with this data object (you selected it on the Associated Persistent Objects Page), you have to define the mapping between the data object methods and the persistent object methods (on the Methods Mapping Page) for the special framework methods in the Methods pane to have implementations. The code for these methods gets modified according to the changes you make on the Methods Mapping page. Similarly, any changes you make to the mapping of attributes on the Attributes Mapping page, get recorded in the code for the attributes' get and set methods.

Top-down
These are the preliminary steps you must follow before you can map a data object to a DB persistent object, when you are defining the persistent object and the schema from the implementation.

1. Add a data object implementation to a data object interface.
2. Indicate that the environment for the implementation is **BOIM with any key**.

To define the mapping between the attributes of the data object and the persistent object, follow these steps:

1. Proceed to add the persistent object and schema from the implementation: from the pop-up menu of the data object implementation, select the **Add Persistent Object and Schema** option.
2. Turn to the Attributes Mapping Page of the Add Persistent Object and Schema wizard.
3. Change the mapping of the attributes, if you want to. Object Builder provides a default mapping between the attributes of the data object and those of the persistent object.
4. Click **Finish**.

Note: While you are defining the mapping of attributes using the Attributes Mapping Page, you can also change the defaults that Object Builder sets for both the persistent object (names and types of persistent object attributes) and the schema (column names and SQL types for the columns).

Bottom-up

Before you can map a data object to a persistent object in the bottom-up case, you must follow these steps:

1. Create a schema by importing an SQL file.
2. Add a persistent object to the schema.

To define the mapping between the attributes of the data object and the persistent object, follow these steps:

1. From the pop-up menu of the persistent object in the DBA-Defined Schemas folder, select **Add Data Object**. The Add Data Object wizard opens to the Names Page, where you can specify the names and the file names for the data object interface and its implementation which you are adding.
2. Click **Next**. The Methods Page opens, and you can define methods specific to the data object.
3. Click **Finish**.

The data object file, interface, and implementation are created in the User-Defined Data Objects folder, and are associated with the persistent object. At this point the default mapping exists between the data object and the persistent object. You can customize the mapping. Follow these steps:

1. From the data object implementation's pop-up menu, select **Properties**. The Data Object Implementation wizard opens.
2. Go to the Attributes Mapping Page. Here, you can change the mapping between the attributes of the data object interface and those of the persistent object. You can use one of the three mapping patterns, and for each pattern, elect whether to provide a mapping helper or not.
3. Click **Next**. The Methods Mapping Page opens, and you can override the default mappings of the special framework methods to the methods of the persistent object.
4. Click **Finish**.

Note:

At this point the data object is stand-alone (it is not associated with a business object). To render the data object functional, you can associate it with an existing business object: first delete the business object's associated data object interface,

and then, from its pop-up menu, select **Select Data Object Interface**, and specify the one that was created from the persistent object.

RELATED CONCEPTS
“Data Object” on page 18
“Persistent Object” on page 19
“Special Framework Methods” on page 24

RELATED TASKS
“Add a Data Object Implementation” on page 299
“Create a DB Schema by Importing an SQL File” on page 321
“Work with Data Objects - Overview” on page 296
“Work with DB Persistent Objects” on page 313
“Map Data Object Attributes to Persistent Object Attributes” on page 256
“Create a Relationship” on page 129
“Customize Referential Integrity” on page 108
“Work with Methods ” on page 267
“Create a Child Component” on page 136

RELATED REFERENCES
“DB2 Data Type Mappings” on page 110
“Oracle Data Type Mappings” on page 113

Map a Data Object to the Parent’s Persistent Object

If the data object implementation you are defining top-down inherits from another (you select a parent implementation for the current one on the Implementation Inheritance Page of the Data Object Implementation wizard), you can use one of two patterns (the flattening pattern or the partitioning pattern), to map the data object to a persistent object.

To map attributes of the data object to attributes of persistent objects that were created from the parent implementation, you use the flattening pattern. Follow these steps:

1. Turn to the Attributes Mapping Page.
2. Select an attribute from the Attributes folder.
3. From the pop-up menu of the attribute, select the pattern for the mapping, which can be one of **Primitive**, **Key Home**, or **Explode**.
4. Click the list button of the **Persistent Object Attribute** field and select an attribute that belongs to a persistent object that was created for the parent data object implementation.
5. Click **Finish.**

Note the following points:

- From the Attributes folder, you can select attributes that are specific to the data object implementation (those you define on the Attributes Page of this wizard), as well as those defined for the business object and specified as data object attributes (state data) on the Data Object Interface Page of the Business Object Implementation wizard.
- The **Persistent Object Attribute** field lists not only the attributes of the parent implementation’s persistent object, but also those of persistent objects belonging to the current implementation.
- You cannot select a parent persistent object for the current implementation on the Associated Persistent Objects Page. That page is used only for associating with

the implementation persistent objects that are at the same level of hierarchy as those that would be created directly from this implementation.

RELATED CONCEPTS

"Inheritance" on page 137
"Choosing an Inheritance Pattern for Persistence" on page 140
"Inheritance with a Single Datastore" on page 155
"Complex Attributes and Mapping Patterns" on page 263

RELATED TASKS

"Map a Data Object to a DB Persistent Object" on page 251
Map a Data Object to a PA Persistent Object
"Map Data Object Attributes to Persistent Object Attributes" on page 256
"Inheritance with a Single Datastore - Scenario" on page 158

Map a Data Object to the Child's Persistent Object

If the data object implementation you are defining top-down inherits from another (you select a parent implementation for the current one on the Implementation Inheritance Page of the Data Object Implementation wizard), you can use one of two patterns (the flattening pattern or the partitioning pattern), to map the data object to a persistent object.

When you use the partitioning pattern, you can use one of two sub-patterns to map the data object's attributes to its DB persistent object's attributes, depending on the type of inheritance between the current data object implementation and its parent:

- "Inheritance with Key Duplication" on page 147
- "Inheritance and Overriding in Helper Objects" on page 138

Inheritance with key duplication
When you use this pattern of mapping, you map attributes of the parent implementation, and all attributes of the current implementation to attributes of the persistent object that you are creating. Follow these steps:

1. From the pop-up menu of the data object implementation, select **Add Persistent Object and Schema.**
2. The Add Persistent Object and Schema opens to the Names Page. Type the identification for the schema and the persistent object you are defining, on this page.
3. Click **Next.** The Attributes Mapping Page opens. Click the **Vertical Partitioning** button. Object Builder maps only those inherited attributes of parent implementation which are key attributes of its business object, as well as all attributes of the current implementation, to attributes of the new persistent object. That is, for each of the key attributes of the parent implementation, and all attributes of the current implementation, it creates corresponding attributes in the persistent object, and does the mapping.
4. Click **Next.** The Columns and Attributes Page opens. You can view the definition of the persistent object attributes and the corresponding schema columns that are created.
5. Click **Next**, and add any comments you want to, on the Comments Page. You can type comments specific to the persistent object, the schema, and each of the schema columns.
6. Click **Finish.**

Inheritance with overriding persistence
When you use this pattern of mapping, you map all attributes of the parent implementation, both key attributes and non-key attributes, and all attributes of the current implementation to attributes of the persistent object that you are creating.

Follow the same steps as for **Inheritance with key duplication**, but in step 3, select the **Horizontal Partitioning** button instead of the **Vertical Partitioning** button.
Object Builder maps all inherited attributes of parent implementation - those which are key attributes of its business object as well as the non-key attributes, and all attributes of the current implementation, to attributes of the new persistent object. That is, for every one of the attributes that are inherited from the parent implementation, and all attributes of the current implementation, it creates corresponding attributes in the persistent object, and does the mapping.

Note: You can map a data object to persistent objects using these same patterns even in the meet-in-the-middle case (on the Attributes Mapping Page of the Data Object Implementation wizard), when you associate a data object implementation with one or more persistent objects with matching persistence type that exist in the model. However, you will have to do the entire mapping on your own.

RELATED CONCEPTS

"Data Object" on page 18
"Persistent Object" on page 19
"Schema" on page 20
"Inheritance" on page 137
"Choosing an Inheritance Pattern for Persistence" on page 140
"Inheritance and Overriding in Helper Objects" on page 138
"Inheritance with Key Duplication" on page 147
"Inheritance with Attributes Duplication" on page 141
"Complex Attributes and Mapping Patterns" on page 263

RELATED TASKS

"Add a Persistent Object and Schema" on page 313
"Map a Data Object to a DB Persistent Object" on page 251
"Map Data Object Attributes to Persistent Object Attributes"
"Create a Child Component" on page 136
"Define a Child with Key Duplication" on page 149
"Inheritance with Key Duplication - Scenario" on page 151
"Define a Child with Attributes Duplication" on page 142
"Inheritance with Attributes Duplication - Scenario" on page 144

Map Data Object Attributes to Persistent Object Attributes

You can map attributes of the data object to those of the persistent object using any one of the following methods:

- "Map Attributes Using the Default Mapping Pattern" on page 257
- "Map Attributes Using a Key" on page 258
- "Map Attributes Using a Mapping Helper" on page 260

The following tasks deal with mapping of complex attributes of the data object to persistent object attributes:

- Map Complex Attributes Using the Primitive Pattern

• “Map Complex Attributes Using the Explode Pattern” on page 265

Restrictions:

- You cannot map multiple data object attributes to the same persistent object attribute.
- When you map a data object to multiple persistent objects, you must map each key attribute of the data object directly to each of the key attributes of the different persistent objects.

RELATED CONCEPTS
“Attributes” on page 26
“Data Object” on page 18
“Persistent Object” on page 19

RELATED TASKS
“Work with Attributes” on page 247
“Work with Data Objects - Overview” on page 296
“Work with DB Persistent Objects” on page 313

Map Attributes Using the Default Mapping Pattern

When you map an attribute of the data object to an attribute of the persistent object of corresponding type, you do not have to use a mapping helper to provide the conversion, or use a key as an intermediate object to perform the mapping.

Note: Only attributes that use the default mapping between the data object and the persistent object will be capable of participating in an object query.

To define a mapping using the default mapping pattern, follow these steps:

1. In the Tasks and Objects pane, select the data object implementation of the object whose attributes you want to map to the attributes of a persistent object.

   Note: The data object implementation can be selected from either the User-Defined Business Objects folder or the User-Defined Data Objects folder.
2. From the implementation’s pop-up menu, select **Properties**. The Data Object Implementation wizard opens to the Name and Platform Page. Click **Next**, or click the arrow to the left of the page name, and select Attributes Mapping Page from the list. The page opens.

   Note: You can also define the mapping when you are defining the data object implementation, if you have selected an associated persistent object to be used with the data object on the Associated Persistent Objects Page, or when you add a persistent object and schema from a data object implementation.
3. From the Attributes folder, select the data object attribute that you want to map.
4. From the pop-up menu of the attribute, select **Primitive**.

Note: When you click **Finish**, if there are any mappings whose types are not suitable for the default mapping pattern, you will be notified.

RELATED CONCEPTS
“Data Object” on page 18
“Persistent Object” on page 19
“Mapping Helper” on page 105
Query Service (*Advanced Programming Guide*)

RELATED TASKS
"Map a Data Object to a DB Persistent Object" on page 251
"Add a Data Object Implementation" on page 299
"Add a Persistent Object and Schema" on page 313

RELATED REFERENCES
"DB2 Data Type Mappings" on page 110
"Oracle Data Type Mappings" on page 113

Map Attributes Using a Key

When there is a reference between two objects, you can use one or more foreign keys to define the mapping between the data object and the persistent object by mapping the key attributes to the persistent object attributes.

Restriction: An attribute that does not use the default mapping between the data object and the persistent object will not be capable of participating in an object query.

Note the following points:

- If the persistent object is created from the implementation itself, default attribute mapping that is done by Object Builder will be using the **Key Home** pattern, but you can override this by defining a **Primitive** mapping.
- You can map an attribute using both the key mapping, and the primitive mapping: multiple mapping is supported.

To define the mapping, follow these steps:

1. In the Tasks and Objects pane, select the data object implementation of the object which has a reference to another object (that is, at least one of the attributes of the data object must have as its type an interface of another object, and the other object must have at least one key defined).

   Note: The data object implementation can be selected from either the User-Defined Business Objects folder or the User-Defined Data Objects folder.
2. From the implementation's pop-up menu, select **Properties**. The Data Object Implementation wizard opens to the Name and Platform Page. Click **Next**, or click the arrow to the left of the page name, and select Attributes Mapping Page from the list. The page opens.

   Note: You can also define the mapping when you are defining the data object implementation, if you have selected an associated persistent object to be used with the data object on the Associated Persistent Objects Page, or when you add a persistent object and schema for the implementation, using the Add Persistent Object and Schema wizard.
3. From the **Attributes** folder, select the data object attribute that you want to map. From its pop-up menu, you have the following choices: **Primitive** (the default mapping), **Key Home**, or **Explode** (if the data object attribute is a complex attribute). The choices are not complementary: the mapping can be done using more than one pattern.
4. Select the **Key Home** option. The first key defined for the referenced object is taken as the default, and appears (with its attributes) in the folder, beneath the data object attribute.

   Restrictions:

- When you create a DB persistent object and DB schema from this data object implementation, it will not automatically create a foreign key in the DB schema.
- When you map a data object attribute that is also specified as an attribute of the key for the corresponding business object, to multiple persistent object attributes using a mapping helper, all the persistent object attributes that are mapped must be persistent object keys.

5. You can change the key you want to use to define the mapping: Select the key in the **Attributes** folder. Click the list button of the **Key** field and select a key from the list of keys defined for the object.
6. Type a name for the home, which is to contain the object that is referenced by the data object in the **Home to Query** field.
7. Select a key attribute. (From its pop-up menu, only the **Primitive** mapping pattern is available.) The **Mapping Helper Class** section appears. For each key attribute, you can optionally specify a mapping helper class and its methods to define the mapping to a persistent object attribute.

   Note the following points:

   - All key attributes have to be mapped, and only one mapping is permitted.
   - Key attributes can be mapped using only the Primitive pattern: the Key Home and Explode patterns are not supported.

   If you want to provide a mapping helper, follow these steps:

   a. Type the name of the mapping helper class in the **Class Name** field.
   b. Type the name of the method that does the mapping from the key attribute to the persistent object attribute in the **Key to PO Method** field.
   c. Type the name of the method that does the mapping from the persistent object attribute to the key attribute in the **PO to Key Method** field.
   d. From the key attribute's pop-up menu, select **Add Mapping**. The first of the defined persistent object attributes is mapped to the selected key attribute. You can change the persistent object attribute you want to use for the mapping: Click the list button of the **Persistent Object Attribute** field and select an attribute from the list of attributes defined for the persistent object. The **Type** field shows the type of the selected attribute. For *char* and *string* types, the **Size** field shows the size of the type.

Note: The mapping from a key attribute to a persistent object attribute is one-to-one. If the persistent object is associated with a schema, and the schema has at least one foreign key, the default one-to-one mapping between the key attributes and the persistent object attributes is provided by Object Builder.

RELATED CONCEPTS

"Data Object" on page 18
"Persistent Object" on page 19
"Mapping Helper" on page 105
Query Service (*Advanced Programming Guide*)
Object Relationships (*Programming Guide*)
"Foreign Key Patterns" on page 132
"Home" on page 342

RELATED TASKS

"Add a Data Object Implementation" on page 299
"Add a Persistent Object and Schema" on page 313
"Map a Data Object to a DB Persistent Object" on page 251

“Map Attributes Using a Mapping Helper”
“Work with Customized Homes - Overview” on page 342
“Define a Foreign Key Pattern” on page 133

Map Attributes Using a Mapping Helper

When you map an attribute of the data object to an attribute of the persistent object of corresponding type, you do not have to use a mapping helper to provide the conversion. When you map attributes of different types, a mapping helper is required. The mapping helper is a class that contains mapping methods. Mapping methods provide the conversion between the attribute types of the two objects. You can either use the mapping helpers provided by Object Builder, or you can define your own.

Restriction: An attribute that does not use the default mapping between the data object and the persistent object will not be capable of participating in an object query.

Note: Even if you map a single data object attribute to multiple persistent object attributes of the same type, it is recommended that you use a mapping helper so that any method that copies the persistent object attributes to the data object attribute copies all mapped persistent object attributes; not just the last one that is mapped.

Object Builder provides the default mapping helper (the class and its methods) in the following cases:

- When a Stringified Object Reference (SOR) of the data object is mapped to a persistent object attribute of type `VARCHAR`. To use this mapping helper, you should have followed these steps:
  1. Specified the type of one of the attributes to be a reference to an object on the Attributes Page of the Business Object Interface wizard. For example, if you wanted the selected object (say Claim) to reference the Policy object, you should have selected the type of one of the attributes of the Claim object to be of type Policy PolicyInterf, where PolicyInterf is the interface defined for the business object named Policy.
  2. Selected **Stringified object reference** as the handle for storing pointers on the Behavior Page of the Data Object Implementation wizard.
  3. Mapped the data object attribute that is of an object reference type to the persistent object attribute of type `VARCHAR` on the Attributes Mapping Page of the Data Object Implementation wizard.
- When a Stringified Object Reference (SOR) of the data object is mapped to a persistent object of type *char.*

  To use this mapping helper, you should have followed the same steps as in the previous case, only replacing the persistent object attribute of type `VARCHAR` with one of type *char*.
- When a data object attribute of type *string* is mapped to a persistent object attribute of type `VARCHAR` (A data object attribute of type *string* is normally mapped to a persistent object attribute of C++ string type. For example, a string of length 20 is mapped to *char[21].*) To use this mapping helper, you should have followed these steps:
  1. Specified one of the attributes of the business object to be of type *string* on the Attributes Page of the Business Object Interface wizard.

2. Changed the SQL type of the column name that corresponds to the business object attribute of type *string,* to VARCHAR on the Name and Attributes Page of the Add Persistent Object and Schema wizard.
3. Mapped the data object attribute that is of type *string* to the persistent object attribute of type VARCHAR on the Attributes Mapping Page of the Data Object Implementation wizard.

- When a data object attribute of type *wstring* is mapped to a persistent object attribute of IDL type DB2VARGRAPHIC (persistent object SQL type VARCHAR).
  390 When one of the constrain platforms is **390** (you select **Platform - Constrain - 390**), *wchar* and *wstring* are not available for selection as an attribute type for your object.
- When a data object attribute of type *ByteString* is mapped to a persistent object attribute of type DB2VARCHAR (persistent object SQL type VARCHAR).
- When a data object attribute of type *ByteString* is mapped to a persistent object attribute of type *char[]* (length greater than 0).

You can view the mapping helper information on the Attributes Mapping Page of the Data Object Implementation wizard when you select the mapped data object attribute in the folder. The .cpp file generated from the data object implementation contains the mapping helper file (DB2MappingHelper.hpp) in its include section.

Restrictions:

- Object Builder does not provide the default mapping between complex data types (*any, Object, wchar* and *wstring* and types defined as constructs, which include typedefs, structures, and unions) and DB2 database types. You must provide your own helper class for these mappings.
- You cannot use a mapping helper to map many data object attributes to either one or many persistent object attributes; you can use one to map many data object attributes to one persistent object attribute.

To map attributes using a mapping helper, follow these steps:

Notes:

- It is recommended that you follow steps 4 through 11 in the sequence laid out.
- If you want to provide your own mapping helper, follow steps 1 through 11.
- If you want to use the mapping helper provided by Object Builder, follow steps 4 through 7.

1. Create (outside Object Builder) a .hpp file, which contains the mapping helper class.

   When you define the mapping helper, follow these rules:

   - Ensure that the .cpp and the .hpp files have the same name as the mapping helper class name.
   - Define both the mapping methods: from the persistent object to the data object, and from the data object to the persistent object, in the mapping helper class.
   - Declare both mapping methods as public members of the class.
   - Define both methods as inline methods to avoid linker errors.
   - Define both methods as static methods.
   - Define the return type of both methods as void.
   - Pass the input arguments for both methods by const reference.

- For the persistent object to data object mapping method, use the following signature:

  inline static void PO_to_DO_ mapping_method_name(att1, att2, ...attn, attribute_of_the_data_object)

  where att1, att2,... attn are the persistent object attributes that are mapped to the data object attribute, and require the mapping helper.
- For the data object to persistent object mapping method, use the following signature:

  inline static void DO_to_PO_ mapping_method_name(attribute_of_the_data_object, att1, att2,..., attn)

  where att1, att2,... attn are the persistent object attributes that are mapped to the data object attribute, and require the mapping helper.>

  Note: The mapping helper file can be located in any directory that is in your include search path.

2. In the Tasks and Objects pane, select the data object implementation of the object whose attributes you want to map to the attributes of a persistent object.

   Note: The data object implementation can be selected from either the User-Defined Business Objects folder or the User-Defined Data Objects folder.
3. From the implementation's pop-up menu, select **Properties**. The Data Object Implementation wizard opens to the Name and Platform Page. Click **Next**, or click the arrow to the left of the page name, and select Attributes Mapping Page from the list. The page opens.
   Note: You can also define the mapping when you are defining the data object implementation, if you have selected an associated persistent object to be used with the data object on the Associated Persistent Objects Page.
4. From the **Attributes** folder, select the data object attribute that you want to map to the persistent object.
5. From the data object attribute's pop-up menu, select **Mapping**.
6. Click the list button of the **Persistent Object Attribute** field, and select the attribute of the persistent object that you want to map to this data object attribute. The persistent object attribute is added to the tree beneath the data object attribute.

   Note: The order in which you map the different persistent object attributes to the data object attribute must be the same as the order in which they are listed in the mapping helper method signatures.
7. Select the data object attribute from the folder.

   Note: If the mapped attributes meet the conditions for which Object Builder provides the default mapping helper, the **Map using helper class** option is automatically selected and the names of the mapping helper class and methods are shown in their respective fields. It is recommended that you use the default mapping helper provided. If you still want to provide your own mapping helper, follow steps 9 through 11.
8. Specify the mapping pattern: select the **Map using helper class** option.
9. Type the name of the mapping helper class in the **Class Name** field.
   Note: Object Builder assumes that the name you provide as the class name is the same as the name of the .hpp file that you include in the file adornment's prolog. If the names are not the same, and you have all the mapping helper

classes in a separate file, you must include this file in the prolog of the data object implementation's file adornment, and regenerate. Follow these steps:

a. Click on the prolog or epilog object in the File Adornments folder

b. Type the #include statement at the beginning of the .cpp file in the editor pane

c. Regenerate the DOImpl_I.cpp file: From the data object implementation's pop-up menu, select **Generate - Selected - .cpp**, or **Generate - All**.

10. Type the name of the method that does the mapping from the attribute of the data object to the attributes of the persistent object in the **DO to PO Mapping Method** field.

11. Type the name of the method that does the mapping from the attributes of the persistent object to the attributes of the data object in the **PO to DO Mapping Method** field.

Note: Mapping helpers are also used when you map an attribute of the data object to an attribute of the persistent object using a foreign key. If a key attribute and the persistent object attribute being mapped are of different types, the mapping helper includes the methods that map between the key and the persistent object.

RELATED CONCEPTS

"Data Object" on page 18
"Persistent Object" on page 19
"Mapping Helper" on page 105
Query Service (*Advanced Programming Guide*)

RELATED TASKS

"Add a Data Object Implementation" on page 299
"Add a Persistent Object and Schema" on page 313
"Map a Data Object to a DB Persistent Object" on page 251
Map a Data Object to a PA Persistent Object

RELATED REFERENCES

"DB2 Data Type Mappings" on page 110
"Oracle Data Type Mappings" on page 113

Complex Attributes and Mapping Patterns

An attribute that is made up of multiple entities is called a complex attribute. For example, a structure.

Restrictions:

- This release of Object Builder supports only structures (data type *struct*) as complex attributes.
- Nested structures are not supported. However, structures whose members are themselves other structures are supported.

When you make a complex attribute persistent using a relational backend datastore, you can use the following mapping patterns:

- Primitive
- Explode

Primitive: The structure is streamed out into a single column whose format is known to the client programmer. That is, you create a single attribute in the persistent object (and a corresponding column in the associated schema) to support a complex data object attribute.

Explode: Each of the primitive members of the attribute is mapped to a different column in the table. This is the same table the data object that contains the attribute is mapped to. You must select the complex attribute members as distinct items from which to create a persistent object and a schema.

The members of the complex attribute are exposed on the Attributes Mapping Page of the Data Object Implementation wizard. You can associate a member of a complex attribute with one or more persistent object attributes.

RELATED CONCEPTS

"Persistent Object" on page 19
"Schema" on page 20

RELATED TASKS

"Work with DB Persistent Objects" on page 313
"Work with DB Schemas" on page 320
"Add a Data Object Implementation" on page 299
"Map a Data Object to a DB Persistent Object" on page 251
"Edit a DB Schema" on page 329
"Edit a Generated SQL File" on page 331

RELATED REFERENCES

"DB2 Data Type Mappings" on page 110
"Oracle Data Type Mappings" on page 113

Map Complex Attributes Using the Primitive Pattern

Mapping of a complex attribute using the primitive pattern is similar to mapping an ordinary attribute of the data object to one in the persistent object.

Note: This release supports only structures (type *struct*) as complex attributes.

Top-down

1. From the pop-up menu of the data object implementation in either the User-Defined Business Objects or the User-Defined Data Objects folder, select Add Persistent Object and Schema.
2. The Add Persistent Object and Schema wizard opens to the Names Page. Name the persistent object and schema you are adding.
3. Click **Next**. The Attributes Mapping Page opens. By default, Object Builder maps all complex data object attributes using the Explode mapping pattern.
4. Delete the default mapping. From the pop-up menu of Explode in the Attributes folder, select **Delete**. The exploded form of the mapping is deleted.
5. The pop-up menu of the complex attribute has two options: **Primitive** and **Explode**. Select **Primitive**. The complex attribute is mapped directly to the persistent object attribute.
6. Click **Finish**. A message informs you that the mapping between the complex attribute of the data object and the persistent object requires a mapping helper.
7. Click **No** to have the mapping exist in its primitive form, without the mapping helper.

When you examine the properties of the persistent object (**Properties** from the pop-up menu of the persistent object), you will see that each of the complex attributes of the data object for which you defined the Primitive mapping is mapped to just one attribute of the persistent object, and therefore to the corresponding column in the schema (the backend database table).

Meet-in-the-middle

1. From the pop-up menu of the data object implementation in either the User-Defined Business Objects or the User-Defined Data Objects folder, select **Properties.**
2. As long as the environment for the data object implementation is **BOIM with any key,** you can associate persistent objects with the implementation. Click the arrow to the left of the page name, and select Associated Persistent Objects Page from the list. The page opens.
3. Add a persistent object instance, and click **Next**.
4. The Attributes Mapping Page opens. Object Builder does not provide the default mappings. For each attribute in the Attributes folder, you can provide a mapping. The simple attributes have only the **Primitive** mapping option available from their pop-up menus; the complex attributes have both the **Primitive** as well as the **Explode** mapping options.
5. For each of the complex attributes, select the **Primitive** mapping pattern. Each complex attribute is mapped directly to the persistent object attribute.
6. Click **Finish.** A message informs you that the mapping between the complex attribute of the data object and the persistent object requires a mapping helper.
7. Click **No** to have the mapping exist in its primitive form, without the mapping helper.

Bottom-up

Complex types in database columns are not supported. So, there's no mapping of complex attributes in the bottom-up case.

RELATED CONCEPTS

"Persistent Object" on page 19
"Schema" on page 20

RELATED TASKS

"Add a Data Object Implementation" on page 299
"Map a Data Object to a DB Persistent Object" on page 251
"Map Data Object Attributes to Persistent Object Attributes" on page 256

RELATED REFERENCES

"DB2 Data Type Mappings" on page 110
"Oracle Data Type Mappings" on page 113

Map Complex Attributes Using the Explode Pattern

The Explode pattern is the default mapping pattern provided by Object Builder when you map a complex attribute of a data object to an attribute of the persistent object. The complex attribute is exploded into its primitive component data elements and mapped across a set of columns in a table.

Note: This release supports only structures (type *struct*) as complex attributes.

To map attributes using the Explode pattern, follow these steps:

1. From the pop-up menu of the data object implementation in either the User-Defined Business Objects or the User-Defined Data Objects folder, select Add Persistent Object and Schema.
2. The Add Persistent Object and Schema wizard opens to the Names Page. Name the persistent object and schema you are adding.
3. Click **Next**. The Attributes Mapping Page opens. By default, Object Builder maps all complex data object attributes using the Explode mapping pattern. In this pattern, each member of the complex attribute is mapped to a different persistent object attribute that is associated with the same database table.
   Note: If you are editing the properties of the data object implementation (that is, you are redefining the mapping that is to be set up between the data object and any persistent objects that are to be created later from this data object implementation), from the pop-up menu of the complex attribute, select **Explode**. Then, continue with step 4. In this case, however, those persistent objects created before you edit the data object implementation retain their original mapping pattern.
4. Accept the default mapping, and click **Finish**. You can, however, change the attributes of the persistent object to which the members of the complex attribute of the data object are mapped.

When you examine the properties of the persistent object (**Properties** from the pop-up menu of the persistent object), you will see that instead of each of the complex attributes of the data object being mapped to a persistent object attribute, only the members of the complex attributes are mapped. So, each member of the data object's complex attributes has a counterpart in the corresponding database table.

Note the following points:

- For the Explode mapping, it is not necessary to provide your own implementation of the mapping helper: you can use the one provided by Object Builder.
- Mapping helpers known to Object Builder are selectable for reuse.

Meet-in-the-middle

1. From the pop-up menu of the data object implementation in either the User-Defined Business Objects or the User-Defined Data Objects folder, select **Properties.**
2. As long as the environment for the data object implementation is BOIM with any key, you can associate persistent objects with the implementation. Click the arrow to the left of the page name, and select Associated Persistent Objects Page from the list.
3. Add a persistent object instance, and click **Next**.
4. The Attributes Mapping Page opens. Object Builder does not provide the default mappings. For each attribute in the Attributes folder, you can provide a mapping. The simple attributes have only the **Primitive** mapping option available from their pop-up menus; the complex attributes have both the **Primitive** as well as the **Explode** mapping options.
5. For each of the complex attributes, select the **Explode** mapping pattern. Each of the component data elements of the complex attribute is mapped directly to a different persistent object attribute in the associated schema (table in the backend store: the database).
6. Click **Finish.**

The resulting mapping is the same as in the top-down case.

Bottom-up
Complex types in database columns are not supported. So, there's no mapping of complex attributes in the bottom-up case.

RELATED CONCEPTS
"Persistent Object" on page 19
"Schema" on page 20

RELATED TASKS
"Add a Data Object Implementation" on page 299
"Map a Data Object to a DB Persistent Object" on page 251
RELATED REFERENCES
"DB2 Data Type Mappings" on page 110
"Oracle Data Type Mappings" on page 113

Work with Methods

There are several different kinds of methods in Component Broker: user-defined methods (methods you define for a component or component object), the get and set methods automatically generated for attributes you define, framework methods automatically generated to support the server programming model, and special framework methods that data objects use to handle persistent data.

For most cases, you only need to provide implementations for user-defined methods. Default implementations are provided for other methods.

The following tasks deal with methods:
- "Add Code for User-Defined Methods"
- "Edit a User-Defined Method" on page 269
- "Edit Get and Set Methods" on page 270
- "Edit Framework Methods" on page 270
- "Edit Special Framework Methods" on page 271
- "Import Changes to Methods" on page 272
- "Delete a Method" on page 277

RELATED CONCEPTS
"User-Defined Methods" on page 23
"Get and Set Methods" on page 23
"Framework Methods" on page 24
"Special Framework Methods" on page 24
"External Files for Method Bodies" on page 273

RELATED TASKS
"Work with Attributes" on page 247

Add Code for User-Defined Methods

To add code for a method you have defined, follow these steps:

1. In the User-Defined Business Objects folder, find the business object implementation or data object implementation whose methods you want to implement (for example, CarPolicyBO).
2. Click on the object. The following folders appear in the Methods List pane:
   - User-Defined Methods
   - User-Defined Attributes
   - Framework Methods
   - File Adornments

   The User-Defined Methods folder is expanded by default, and under it appear the methods you have defined for the business object.
3. Click a method.

   The signature of the method appears in the editor pane. You can add an implementation to the signature directly in the editor pane, or you can edit the properties of the method and get its implementation from elsewhere.
4. From the method's pop-up menu, click **Properties**. The Method Implementation wizard opens to the Implementation Page.
5. Select whether to use the implementation defined in the editor pane, or get the implementation code for the method from an external file.

   If you select to get the information from an external file, you can specify it as a template file, in which case you can use substitution macros, as defined on the Template File Macros Page of the wizard.
6. Select whether to have a single implementation for all platforms, or use a separate implementation for each platform.

   If you select to have a different implementation for each platform, then the implementation you provide in the Source pane will apply only to the current platform selected in the **Platform - View** menu. You can switch the view to provide implementations for each of the platforms you intend to deploy on.
7. Click **Finish**. The selected behavior will be used the next time code is generated for the business object implementation.

RELATED CONCEPTS
"User-Defined Methods" on page 23
"Chapter 7. Multi-Platform Development" on page 187
"External Files for Method Bodies" on page 273

RELATED TASKS
"Work with Methods " on page 267
"Import Changes to Methods" on page 272

Add an Initializer Method

If there is code that you need called when a component is loaded (the equivalent of a static initializer method in Java), you can put the code in a static method that gets called by a static attribute. When the component is loaded, the attribute is initialized, and calls the initializer method.

To add an initializer method, that will contain code to be executed on the loading of the component, follow these steps:
1. Open the Business Object Implementation wizard (from the implementation's pop-up menu, click **Properties**).
2. Click the title bar and turn to the Methods page.

3. Add a static method that returns type int.
4. Click the title bar and turn to the Attributes page.
5. Add a static attribute of type int. In the attribute's initializer field, type a call to the static method.
6. Click **Finish**. The wizard closes.
7. In the Tasks and Objects pane, make sure the business object implementation is in focus.
8. In the Methods pane, expand the User-Defined Methods folder and select the method you defined. Its skeleton implementation appears in the Source pane.
9. In the Source pane, add to the method body any code you want called during initialization of the component. As the final step, return some int value to the calling attribute, to complete the attribute's initialization.
10. Click **File - Save**.

RELATED CONCEPTS

"User-Defined Methods" on page 23
"Attributes" on page 26

RELATED TASKS

"Add an Attribute" on page 247
"Add Code for User-Defined Methods" on page 267

Edit a User-Defined Method

To edit the signature of a user-defined method, follow these steps:
1. From the pop-up menu of the business object interface or data object interface where the method is defined, click **Properties** to open the Business Object Interface wizard.
2. Click the title bar and turn to the Methods Page.
3. Select the method under the Methods folder.
4. Make your changes to the method.
5. Click **Finish**.

You can edit the implementation of a user-defined method directly in the editor pane. If the editor pane is in read-only mode, then the implementation is being provided from an external file, as set in the method's wizard.

To access a method's wizard, follow these steps:
1. Select the business object implementation or data object implementation that implements the method.
2. In the Methods pane, select the user-defined method.
3. From the pop-up menu of the method, click **Properties** to open the Method Implementation wizard.
4. Make your changes in the wizard and click **Finish** to apply them.

RELATED CONCEPTS

"User-Defined Methods" on page 23

RELATED TASKS

"Add Code for User-Defined Methods" on page 267
"Import Changes to Methods" on page 272

Edit Get and Set Methods

Get and set methods provide access to attributes defined in a business object interface or data object interface.

To edit the signature of a get or set method, you must edit the attribute it represents, in the business object interface or data object interface.

By default, the get and set implementations are read-only. To edit the implementation of a get or set method (not recommended), follow these steps:

1. Click on the business object implementation or data object implementation in the Tasks and Objects pane.
2. In the Methods pane, locate the get or set method under the Attributes folder.
3. From the pop-up menu of the attribute, click **Properties**. The Method Implementation wizard appears, open to the Implementation Page.
4. Select the check box **Method body is the same for all platforms** if you plan to provide your own code for the method body, and you want it to be the same for all platforms.
5. Click **Use the implementation defined in the editor pane**.

   You could also click **Use an external file**, and select an external file that contained the method implementation.
6. Click **Finish**.

At any time, you can reset the implementation by opening the Method Implementation wizard and clicking **Return to Default**. If you want to return to using only the default, click **Use the implementation provided by Object Builder** to put the implementation back into read-only mode.

RELATED CONCEPTS

"Get and Set Methods" on page 23
"Attributes" on page 26

RELATED TASKS

"Work with Methods " on page 267
"Import Changes to Methods" on page 272
"Edit an Attribute" on page 248

Edit Framework Methods

Framework methods are added to an object by Object Builder. Generally, framework methods are only called by other framework methods, or by Component Broker services.

You cannot edit the signature of a framework method. Some methods are added or deleted based on your selections in the component wizards.

By default, the implementations of framework methods are read-only. To edit the implementation of a framework method (not recommended), follow these steps:

1. In the Tasks and Objects pane, click on the object whose framework methods you want to edit.
2. In the Methods pane, locate the framework method under the Framework Methods folder.

3. From the pop-up menu of the attribute, click **Properties**. The Method Implementation wizard appears, open to the Implementation Page.
4. Click **Use the implementation defined in the editor pane**.

   You could also click **Use an external file**, and select an external file that contained the method implementation.

   As long as this option is selected, the method's implementation will be determined by what is provided in the Source pane. The implementation will **not** be automatically updated in response to design changes. The implementation must be updated by hand.
5. Click **Finish**.

At any time, you can reset the implementation by opening the Method Implementation wizard and clicking **Return to Default**. If you want to return to using only the default, click **Use the implementation provided by Object Builder** to put the implementation back into read-only mode.

The framework methods create, retrieve, update, del, and setConnection are special framework methods of the data object implementation and persistent object. Unless you provide your own implementation, the special framework method implementations are defined based on the mapping of the data object to the persistent object.

RELATED CONCEPTS

"Framework Methods" on page 24
"Special Framework Methods" on page 24

RELATED TASKS

"Work with Methods " on page 267
"Import Changes to Methods" on page 272
"Edit Special Framework Methods"

Edit Special Framework Methods

Data objects and persistent objects that access a schema have the special framework methods insert, update, retrieve, del, and setConnection. The implementations for these methods are calculated based on the mapping between the persistent object methods and the data object methods.

By default, the method implementations are read-only in the editor pane. To override the calculated method implementations in the editor pane, follow these steps:

1. Locate the method in the Methods pane.
2. From the method's pop-up menu, click **Properties**.
3. In the Method Implementation wizard, select where you want to get the implementation from: the tool-provided implementation, the editor pane, or an external file.
4. If you select the editor pane as the source, the implementation becomes editable in the editor pane, and whatever changes you make will be preserved.

   Note: You can switch back to the calculated method body at any time, by changing the setting in the wizard.

For most cases, the tool-provided method bodies should be sufficient. However, if a persistent object represents a read-only view in the database, you will need to edit

the mappable framework method implementations. Most views are read-only but some can be updated. The Embedded SQL preprocessor (idatapre) will fail on any .sqx file generated from an embedded static persistent object, which you create for a read-only view in the database.

If you detect that a view is read-only (at DLL build time), for each of the framework methods insert(), update() and del() in the Methods List pane, follow these steps:

1. From the pop-up menu of the method, select Properties. The Method Implementation wizard opens to the Implementation Page.
2. Select the option **Use the implementation defined in the editor pane**, and make sure the method body is empty.

   For example,
   ```
   void insert()
   {
   }
   ```

To change the calculated method body for a special framework method, change the mapping of persistent object method to data object method in the Data Object Implementation wizard, Methods Mapping Page.

RELATED CONCEPTS

"Special Framework Methods" on page 24

RELATED TASKS

"Work with Methods " on page 267
"Import Changes to Methods"
"Customize Referential Integrity" on page 108

Import Changes to Methods

If you make changes to method implementations in the generated source code, you need to import the changes back into Object Builder, or the changes will be overwritten the next time you generate code.

When you import edited code, only changes to method implementations are applied. The import process recognizes method implementations by the comment block that delimits them:

```
// Insert method modifications here
...
// End method modifications here
```

This comment block is inserted by the code generation process. Any changes you make outside of these generated comment blocks are ignored.

To import code you have edited, follow these steps:

1. From the pop-up menu of either the User-Defined Business Objects folder or the User-Defined Data Objects folder, click **Import - Changes** to open the Import Changes wizard.
2. From the **Available files** list, select those that contain changes to method bodies that you want to import.
3. Click **>>** to move the files to the **Files to be imported** list.
4. Click **Finish**. The method body changes are applied.

5. Make any other changes necessary (for example, if the interface of a method has changed, you need to change its definition in the Business Object Interface wizard).

You can also import changes in batch mode, as follows:
1. Close Object Builder (to allow the import process access to the project model).
2. From a command window, type

   importimpl -p<*project_dir_name*> -f<*sourcefile1 sourcefile2*...>

   where *project_dir_name* is the name of the project directory that contains your model, and the file names that follow are the business object implementation files that contain your changes. For example:

   importimpl -pF:\MyProject -fClaimBO.cpp AgentBO.java

RELATED TASKS
"Edit a Business Object Implementation" on page 290
"Generate Code" on page 363
"Add Code for User-Defined Methods" on page 267

External Files for Method Bodies

Most of the editing you do in Object Builder is of method bodies. You can either create a method body in Object Builder using the Source pane editor, or define the method body in an external file that will get pulled into the generated code for the object.

You can select to use an external file for a particular method in its Method Implementation wizard. From a method's pop-up menu, click **Properties** to display the wizard. By default, new external files will be placed in the current project's \Model directory. The advantage of using external files is that you can do more work outside of Object Builder. You can edit the external files with your preferred editor, and then use the obgen command (with the -change option), outside of Object Builder, to generate the code for the relevant objects and pull together the code from the external file.

An alternative to the use of external files is direct editing of method bodies in the generated source files. While this removes the need to use obgen to pull in your changes, you do need to remember to import the changes back into Object Builder before you generate code again, or your changes will get over-written.

Template Files
Another advantage of external files is that you can use the same external file for multiple methods, by putting macros in the file and identifying the file as a template. Macros are identified within the file by the delimiter $.

For example, given the following template file:

```
GenericMethodBody.template
char* str = "This is a method of $classname$";
cout << str << endl;
return ::CORBA::string_dup (str);
```

In a method's wizard, for example the deny() method of a ClaimBO, specify the external file GenericMethodBody.template, specify that it is a template file, and on the next page, add a Template File Macro with the name classname and the substitution value ClaimBO. When you generate the code for ClaimBO, it contains a method body something like this:

```
::CORBA::Void ClaimBO_Impl::deny()
{
//Version identifier DCE:F3F30755-6F47-11d2-AF4E-000629B3CFEE:1
// Insert Method modifications here
char* str = "This is a method of ClaimBO";
cout << str << endl;
return ::CORBA::string_dup (str);
// End Method modifications here
}
```

RELATED CONCEPTS

"User-Defined Methods" on page 23
"Get and Set Methods" on page 23
"Framework Methods" on page 24
"Special Framework Methods" on page 24
"Push-Down Methods" on page 25

RELATED TASKS

"Import Changes to Methods" on page 272
"Run Object Builder in Batch Mode" on page 11
"Add Code for User-Defined Methods" on page 267

Use Push-Down Methods with PA Persistent Objects

When push-down methods are used to transmit transactional data of existing applications, they have to be used with PA persistent objects. The method of using them differs, depending on whether you are mapping a business object to a data object, or whether you are mapping a data object to a persistent object.

Method 1 (Map a Business Object to a Data Object)

1. Import the PA bean. The PA schema is created in the User-Defined PA Schemas folder along with the PA persistent object.
2. Add a data object to the persistent object.
3. On the Methods Page of the Add Data Object wizard, define the method *debit* (of type *long,* with parameter *amount* that maps to the push-down method associated with the PA persistent object, for example, iBeCashAcctPAOPO.debit)
   Note: The type of the data object method must be the same as the type of the method defined in the PA bean.
4. Create a business object.
5. At the business object interface level, define method *debit* (return type void) with a parameter *amount* (of type long).
   Note: You must define the method signature to match the one you created for your PA bean.
6. Add an implementation for the business object. Select the option **Add or select one later**.
7. From the pop-up menu of the business object implementation, select **Select a Data Object Interface.** On the Selection Page, ensure that you select the data object interface, which was created when you added the data object from the PA persistent object (BeCashAcctPAODO). The data object interface, implementation, and the PA persistent object and PA schema are added to the business object implementation in the User-Defined Business Objects folder.
8. Turn to the Methods Mapping Page. The Business Object Methods folder shows the method (*debit*).

9. Select the method, and from its pop-up menu, select **Add.** The Data Object Method field appears.
10. Click the list button, and select the data object method (*debit*) to be mapped to the business object method.
11. Click **Finish.**

Method 2 (Map a Data Object to a Persistent Object)

1. At the business object interface level, define method *debit* (return type void) with a parameter *amount* (of type long)
2. Add an implementation for the business object.
3. On the Data Object Interface Page, specify the attributes of the business object that are to be data object attributes.
4. Turn to the Data Object Methods Page, and select the methods of the business object to be pushed down to the data object.
5. Add a data object implementation, selecting **Procedural Adaptors** for the implementation.
6. Associate a PA persistent object with the implementation.
7. Turn to the Methods Mapping Page.
8. The method you defined for the data object (*debit*) appears in the User-Defined Methods folder, and you can map it to the corresponding persistent object push-down method.

RELATED CONCEPTS

"Enterprise Access Builder (EAB)" on page 116
"Push-Down Methods" on page 25
"Persistent Object" on page 19
"Data Object" on page 18
Application Adaptor *(Programming Guide)*

RELATED TASKS

"Map a Business Object to a Data Object" on page 288
Map a Data Object to a PA Persistent Object

Customize Business Object OO-SQL Implementation Methods

If you define a relationship from one business object to another, and you choose to implement the relationship using OO-SQL queries, which you indicate on the Object Relationships Page of the Business Object Implementation wizard, you can provide the OO-SQL code for the list() method of the business object implementation. Object Builder will not validate the code you provide. Your code will overwrite the default tool-generated code for this method.

Follow these steps to customize the implementation of the list method:

1. In the Tasks and Objects pane, select the business object implementation for which you have defined the relationship whose implementation is to use OO-SQL queries.
2. The User-Defined Relationships folder in the Methods pane shows the relationship you have defined. Expand the relationship node to view the add, list and remove methods for the relationship that are being implemented for the business object implementation.
3. Select the list method.

4. From its pop-up menu, select **Properties**. The Method Implementation wizard opens to the Implementation Page.
5. Accept the defaults and click **Next** to advance to the OO-SQL Customization Page.
6. Select the **Provide your own OO-SQL code** check box.
7. The tool-generated code is cleared from the panel, and it becomes editable. Type in your code for the method and click **Finish.**

You can view the code you provided in the editor pane, when you select the list method in the Methods pane.

RELATED CONCEPTS

"Business Object" on page 17
"Relationship Methods" on page 25

RELATED TASKS

"Add a Business Object Implementation and Data Object Interface" on page 284
"Create a Relationship" on page 129

Customize Persistent Object ESQL Framework Methods

You can use the ESQL Customization Page of the Method Implementation wizard to provide your own embedded SQL code for the special framework methods of any persistent object in the model that uses embedded SQL.

Note the following points:

- The persistent object's special framework methods that you can customize are the insert(), update(), retrieve(), and del() methods; not the setConnection() method.
- Object Builder will not validate the code you provide. Your code will overwrite the default tool-generated code for this method.

Follow these steps to customize the ESQL clauses for the special framework methods of the persistent object:

1. Select the persistent object that uses embedded SQL in the Tasks and Objects pane. (You can select it from any one of these folders: User-Defined Business Objects folder, User-Defined Data Objects folder, DBA-Defined Schemas folder.) You will see the persistent object's methods in the Methods pane.
2. Select the persistent object's special framework method that you want to customize from the Framework Methods folder.
3. From its pop-up menu, select **Properties**. The Method Implementation wizard opens to the Implementation Page.
4. Ensure that the Use the implementation provided by Object Builder option is selected.
5. Click **Next**, or click the arrow to the left of the page name, and select ESQL Customization Page from the list. The page opens.
6. Select the check box **Provide your own ESQL code.** This makes the panel that shows the code for the method editable, and you can either edit the code provided by Object Builder, or type in entirely different code for the method. Object Builder will not validate the code you provide.
7. Click **Finish**.

Note: You can overwrite the code you provide with Object Builder's code for the method by clearing the **Provide your own ESQL code** check box.

RELATED CONCEPTS
"Persistent Object" on page 19
"Special Framework Methods" on page 24

RELATED TASKS
"Work with Methods " on page 267

Delete a Method

To delete a user-defined method, follow these steps:

1. In the User-Defined Business Objects folder, locate the object (business object interface, data object interface, business object implementation, or data object implementation) that defines the method.
2. From the pop-up menu of the interface, click **Properties** to open the object's wizard.
3. Click the title bar and turn to the Methods Page.
4. Locate the method under the Methods folder.
5. From the pop-up menu of the method, click **Delete**.
6. Click **Finish**.

   If the method was defined in an interface, then it is automatically from any associated implementations.

Get and set methods are deleted automatically when the attribute they represent is deleted.

RELATED CONCEPTS
"User-Defined Methods" on page 23

RELATED TASKS
"Work with Methods " on page 267

Work with Constructs

Constructs (constants, enumerations, exceptions, typedefs, structures, and unions) can be defined at the file, module, or interface level of a business object interface or data object interface, or at the file or module level of a composition.

You can define them directly in Object Builder, or define them in Rose and export them to Object Builder.

The following tasks deal with constructs:

- "Define Constructs with File Scope" on page 278
- "Define Constructs with Module Scope" on page 279
- "Define Constructs With Interface Scope" on page 279
- "Edit a Construct" on page 280
- "Delete a Construct" on page 280

RELATED CONCEPTS

"Constructs" on page 26
"Constructs You Can Export from Rose" on page 79

RELATED TASKS

"Work with Methods " on page 267
"Work with Attributes" on page 247

Define Constructs with File Scope

You can define constructs with file scope using the Business Object File wizard, Data Object File wizard, or Composition File - wizard. You can add the constructs when you create a new file, or by modifying the properties of an existing file (from the file's pop-up menu, select **Properties**).

Warning: File scope constructs are not qualified. In C++ implementations, they pollute the namespace and may conflict with other definitions. In Java implementations they are placed in the "unnamed" package, and may produce compilation errors (processing of classes in the unnamed package is implementation dependent). It is recommended that all constructs be defined at module or interface scope.

To define a construct, follow these steps:

1. In the wizard, click the title bar and turn to the Constructs Page.
2. From the Constructs pop-up menu, select the construct you want to add. You can select from the following options:
   - Constant
   - Enumeration
   - Exception
   - Typedef
   - Structure
   - Union

   If the construct is an enumeration, exception, structure, or union, you must define at least one member for it. To define a member for a construct, follow these steps:

   a. Under the construct in the tree view, select the Members folder.
   b. From the Members pop-up menu, click **Add**.
   c. Type a name for the member and any other requested information. The information is saved when you click another item, select another action, or leave the page.

   Note: To use the construct as a type within another construct, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.
3. Complete the rest of the pages, or click **Finish**.

RELATED TASKS

"Create a Business Object File" on page 282
"Create a Data Object File" on page 303
"Create a Composition File" on page 349

Define Constructs with Module Scope

You can define constructs with module scope using the Business Object Module wizard, Data Object Module wizard, or Composition Module wizard. You can add the constructs when you create a new module, or by modifying the properties of an existing module (from the module's pop-up menu, select **Properties**).

To define a construct, follow these steps:

1. In the wizard, click the title bar and turn to the Constructs Page.
2. From the Constructs pop-up menu, select the construct you want to add. You can select from the following options:
   - Constant
   - Enumeration
   - Exception
   - Typedef
   - Structure
   - Union

   If the construct is an enumeration, exception, structure, or union, you must define at least one member for it. To define a member for a construct, follow these steps:

   a. Under the construct in the tree view, select the Members folder.
   b. From the Members pop-up menu, select **Add Member**.
   c. Type a name for the member and any other requested information. The information is saved when you click another item, select another action, or leave the page.

   Note: To use the construct as a type within another construct, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.
3. Complete the rest of the pages, or click **Finish**.

RELATED TASKS

"Add a Business Object Module" on page 282
"Add a Data Object Module" on page 304
"Add a Composition Module" on page 349

Define Constructs With Interface Scope

You can define constructs with interface scope using the Business Object Interface wizard or Data Object Interface wizard. You can add the constructs when you create a new interface, or by modifying the properties of an existing interface (from the interface's pop-up menu, select **Properties**).

To define a construct, follow these steps:

1. In the wizard, click the title bar and turn to the Constructs Page.
2. From the Constructs pop-up menu, select the construct you want to add. You can select from the following options:
   - Constant
   - Enumeration
   - Exception

- Typedef
- Structure
- Union

  If the construct is an enumeration, exception, structure, or union, you must define at least one member for it. To define a member for a construct, follow these steps:

  a. Under the construct in the tree view, select the Members folder.
  b. From the Members pop-up menu, select **Add Member**.
  c. Type a name for the member and any other requested information. The information is saved when you click another item, select another action, or leave the page.

3. Complete the rest of the pages, or click **Finish**.

   Note: To use the construct as the type of an attribute, method return, method exception, or construct member, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.

RELATED TASKS
"Add a Business Object Interface" on page 283
"Create a Data Object Interface" on page 297

Edit a Construct

To edit a construct, follow these steps:

1. Locate the file, module, or interface that defines the construct in the User-Defined Business Objects folder.
2. From the pop-up menu of the item, click **Properties**. The item's wizard opens.
3. Click the title bar and turn to the Constructs Page.
4. Under the Constructs folder, locate the construct.
5. Click on the construct.
6. Edit the construct. If the construct has a members folder, you can add, delete, or edit the members.
7. Click **Finish**.

The construct is changed. Any methods that used the construct as their method return type or as a parameter, and any attributes that had the construct as their type, are automatically updated.

RELATED CONCEPTS
"Constructs" on page 26

RELATED TASKS
"Work with Constructs" on page 277

Delete a Construct

To delete a construct, follow these steps:

1. Locate the file, module, or interface that defines the construct in the User-Defined Business Objects folder.
2. From the pop-up menu of the item, click **Properties**. The item's wizard opens.

3. Click the title bar and turn to the Constructs Page.
4. Under the Constructs folder, locate the construct.
5. From the pop-up menu of the construct, click **Delete**.
6. Click **Finish**.

The construct is deleted. Any methods that used the construct as their method return type or as a parameter, and any attributes that had the construct as their type, are automatically modified to refer to invalidType.

RELATED CONCEPTS
"Constructs" on page 26

RELATED TASKS
"Work with Constructs" on page 277

Work with Business Objects

Business objects are defined in the User-Defined Business Objects folder, and are presented in terms of four objects:

- The business object file (which contains one or more interfaces, optionally organized into modules)
- The business object module, if any (which contains one or more interfaces)
- The business object interface (which has one or more implementations)
- The business object implementation (which has its own file, defined on the first page of its wizard)

The four objects are created and edited separately, but collectively form a single business object. Each business object (each set of business object file, module, interface, and implementation) typically has its own data object.

The following tasks deal with business objects:

- "Create a Business Object File" on page 282
- "Add a Business Object Module" on page 282
- "Add a Business Object Interface" on page 283
- "Create a Business Object Interface by Importing an IDL File" on page 289
- "Add a Business Object Implementation and Data Object Interface" on page 284
- "Add a Business Object from a Data Object" on page 287
- "Map a Business Object to a Data Object" on page 288
- "Edit a Business Object Interface" on page 290
- "Edit a Business Object Implementation" on page 290
- "Delete a Business Object Interface" on page 291
- "Delete a Business Object Implementation" on page 291

RELATED CONCEPTS
"Business Object" on page 17
"Data Object" on page 18

Create a Business Object File

A business object file (IDL) is a container for your business object interfaces. Although a file can hold multiple business object interfaces, which you may organize into modules, you typically add one interface to each file.

To create a business object file, follow these steps:

1. From the Tasks and Objects pane, select the **User-Defined Business Objects** folder.
2. From the folder's pop-up menu, select **Add file**. The Business Object File wizard opens to the Name Page.
3. Type a name for the file (for example, an insurance application might have a file named Policy).
4. Click **Next**. The Constructs Page opens.

   Use the Constructs pop-up menu to add constants, enumerations, exceptions, structures, typedefs, and unions. Any constructs you add are scoped to every interface in the file.

   Note: To use the construct as a type within another construct, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.
5. Click **Next**. The Files to Include Page opens.

   IManagedClient is included by default. This is the correct choice for a component that represents a base class in your design. If your component had a parent, you would specify the business object file of the parent component in this field. For example, if the CarPolicy component inherits from the Policy component, then you would specify the business object file for Policy on this page. Also include the business object files for any referenced or related components. For example, if CarPolicy has an attribute of type Claim, you would need to include the business object file for Claim on this page.
6. Click **Next**. The Comments Page opens. Type any comments you want to include as comment lines in your generated IDL code.
7. Click **Finish**. The wizard closes, and your file is added to the User-Defined Business Objects folder. You can now add modules or interfaces to the file.

Once you have created the file, you can modify it by selecting **Properties** from its pop-up menu. The Business Object File wizard opens again, with your selections preserved.

RELATED CONCEPTS

"Business Object" on page 17

RELATED TASKS

"Define Constructs with File Scope" on page 278
"Add a Business Object Module"
"Add a Business Object Interface" on page 283

Add a Business Object Module

If you plan to add multiple business object interfaces to a single file, you may want to store the interfaces in separate modules. Any constructs you add to a module are scoped only to the interfaces within that module. To add a module to a file, follow these steps:

1. From the User-Defined Business Objects folder, select your business object file.
2. From the file's pop-up menu, select **Add Module**. The Business Object Module wizard opens to the Name Page.
3. Type a name for the module.
4. Click **Next**. The Constructs Page opens.

   Use the Constructs pop-up menu to add enumerations, exceptions, structures and so on.

   Note: To use the construct as a type within another construct, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.
5. Click **Next**. The Comments Page opens. Type any comments you want to include as comment lines in your generated code.
6. Click **Finish**. The wizard closes, and your module is added to the User-Defined Business Objects folder, underneath the file.

You can now add business object interfaces to the module.

RELATED TASKS

"Define Constructs with Module Scope" on page 279
"Add a Business Object Interface"

Add a Business Object Interface

To add a business object interface to a file (or module), follow these steps:

1. From the User-Defined Business Objects folder, select the file or module that will contain the interface.
2. From the pop-up menu for the file or module, select **Add Interface**. The Business Object Interface wizard opens to the Name Page.
3. Type a name for the interface (for example, CarPolicy).
4. Select whether the interface will be queryable or not.

   If you select this option, the generated code for the managed object contains the dynamic dispatch method callMethodByName, which allows the Query Service to call the methods of the managed object. You should also configure the managed object with a queryable home.
5. Click **Next**. The Constructs Page opens.

   Use the Constructs pop-up menu to add constants, enumerations, exceptions, typedefs, structures, or unions. Any constructs you add are scoped to this interface only.

   Note: To use the construct as the type of an attribute, method return, method exception, or construct member, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.
6. Click **Next**. The Interface Inheritance Page opens.

   By default, the interface inherits from IManagedClient::IManageable. This is the correct choice for a component that represents a base class in your design. If your component had a parent, you would specify the business object interface of the parent component on this page.
7. Click **Next**. The Attributes Page opens.

   To specify attributes for your interface, select **Add** from the Attributes pop-up menu (for example, the CarPolicy interface could have the attributes "make" and "model").

Note: For most attribute types, a default initializer value is provided. When there is no suitable default (for example, an attribute whose type is an enumeration), you should assign your own initializer value, if necessary.

If you specify an attribute as public, then it will be exposed in the interface's IDL file, and Object Builder will provide the appropriate get and set methods for the attribute in the business object implementation.

If you specify an attribute as protected or private, then it will not be exposed in the IDL file, but will be included in the business object implementation (as a protected or private attribute) when you add the implementation to the interface.

8. Click **Next**. The Methods Page opens.

   To specify methods for your interface, select **Add** from the Methods pop-up menu. For example, the CarPolicy interface could have the method "riskQuotient".
9. Click **Next**. The Object Relationships Page opens.

   To specify any relationships that this class has to other classes, select **Add** from the Objects pop-up menu. The relationships will be one-to-many, and will be stored in the collection you select.
10. Click **Next**. The Comments Page opens. Type any comments you want to include as comment lines in your generated code.
11. Click **Finish**. Your new interface is added to the User-Defined Business Objects folder, with the attributes and methods you specified.

You should now see your interface in the Inheritance pane, and any methods you defined for your interface should appear under the **User-Defined Methods** folder in the Methods pane.

RELATED CONCEPTS

"Business Object" on page 17
Query Service (*Advanced Programming Guide*)

RELATED TASKS

"Work with Business Objects" on page 281
"Define Constructs With Interface Scope" on page 279
"Add a Key" on page 292

Add a Business Object Implementation and Data Object Interface

Once you have created a business object interface, you can add one or more implementations for that business object, and also create the data object interface that provides your business object with access to data. You can accomplish both tasks using the Business Object Implementation wizard. Ensure that you have added a key and a copy helper to the business object interface before proceeding with this task.

To create the business object implementation, and its associated data object interface, follow these steps:

1. From the User-Defined Business Objects folder, select the business object interface you want to implement.
2. Display the pop-up menu for the interface and select **Add Implementation**. The Business Object Implementation wizard opens to the Name and Data Access Pattern Page.

3. Appropriate implementation names are filled in for you (the business object file name and interface name plus BO: for example, CPFile::CarPolicy gets an implementation named CPFileBO::CarPolicyBO). You can accept these defaults or replace them with your own names.
4. Select the pattern you want to use for handling state data. The following patterns are available:
   - **Delegating**
     The business object delegates every request for the essential state to the data object interface.
   - **Caching**
     Both the business object and the data object have their own copies of the essential state, which are synchronized. **Lazy evaluation** is the default synchronization method, meaning that cached copies of the attributes are synchronized at first use, rather than at instantiation.
   - **Same as parent's**
     The business object inherits its pattern from a parent interface.

     Note: This option is selected by default if the interface for this business object inherits from another business object interface. However, you still have to indicate the implementation parent on the Implementation Inheritance page of this wizard.

   There is also an option listed for **None**, which would generate a transient data object. This option is not available in this release.
5. Select whether to create a new data object now, or add or select one later.
6. If the component uses Session Services, select whether to provide end-of-session cleanup logic for the component.

   When you check this option, the endResource method is added to the data object, so you can add your own implementation for it. The data object's endResource method will be called at the end of a session, immediately before the managed object's endResource method.
7. Click **Next**. The Implementation Inheritance Page opens.
8. Make sure that IManagedClient::IManageable is listed as a parent under the Parent Class folder.

   You can also select any parent business object implementations you want to inherit behavior from.
9. Click **Next**. The Implementation Language page opens. Select the language you want the business object to be implemented in. You can select either Java or C++.

   The default for this page is set in the Preferences notebook, on the Tasks and Objects page.
10. Click **Next**. The Attributes Page opens. Specify any attributes you want to add to the business object implementation (in addition to the attributes you already specified in the business object interface).
11. Click **Next**. The Methods Page opens. Specify any methods you want to add to the business object implementation (in addition to the methods you already specified in the business object interface).
12. Click **Next**. The Key and Copy Helper Page opens. Select a key and, optionally, copy helper that you have created for this business object (for example, CarPolicyKey and CarPolicyCopy).
13. Click **Next**. The Handle Selection Page opens.

You can select a handle for the business object implementation. If you select a handle, then the framework method getHandleString is implemented, which overrides the getHandleString method of IManagedClient::IManageable. The method provides a way to encapsulate the business object implementation, by returning a string that represents a reference to the business object. The handle you select determines the pattern used to form the string (that is, to turn the reference into a string, or to swizzle the pointer).

14. Click **Next**. If the business object implementation has parent classes with overrideable attributes, then the Attributes to Override Page opens.

    You can use this page to select which of the parent class's attributes you want to override.
15. Click **Next**. If the business object implementation has parent classes with overrideable methods, then the Methods to Override Page opens.

    You can use this page to select which of the parent class's methods you want to override.
16. Click **Next**. If the business object interface defines one-to-many relationships, then the Object Relationships page opens.

    You can use this page to set the way that the object relationship will be implemented.
17. Click **Next**. The Data Object Interface Page opens. (Note: This page does not open if, on the first page, you chose not to create a new data object.)

    Appropriate data object names are filled in for you (the business object file name and interface name plus DO: for example, CPFile::CarPolicy gets the data object interface CPFileDO::CarPolicyDO). You can accept these defaults or replace them with your own names.
18. Select which attributes you want preserved in the data object. These attributes constitute the state data for the component.

    If you implemented a one-to-many relationship as a **Local persistent reference**, then an attribute representing it appears here, so you can select to preserve it in the data object.
19. Click **Next**. The Data Object Methods Page opens. (Note: This page does not open if, on the first page, you chose not to create a new data object.)
20. Select which business object methods you want to push down to the data object (that is, call equivalent methods to be defined in the data object).
21. Click **Next**. The Summary of Framework Methods Page opens.

    Based on your selections on the previous pages of the wizard, this page displays the methods that your object implements. For example, if you selected a caching pattern to handle the essential state of your business object (on the first page), this list includes the synchToDataObject method required to keep the two sets of attributes synchronized. No action is needed.
22. Click **Finish**. The business object implementation and data object interface appear in the User-Defined Business Objects folder, under your business object interface. The data object interface also appears in the User-Defined Data Objects folder.

Now that the business object implementation is defined, you can enter the implementation code for each user-defined method.

RELATED CONCEPTS

"Business Object" on page 17
"Data Object" on page 18
Session Service (*Advanced Programming Guide*)

RELATED TASKS

"Add Code for User-Defined Methods" on page 267
"Add endResource() to a Sessional Business Object" on page 117
"Add a Data Object Implementation" on page 299
"Define a One-to-Many Relationship" on page 131

Add a Business Object from a Data Object

Although you cannot add a business object directly to a data object, you can create a business object separately, and then map it to the data object.

To create a business object and map it to an existing data object, follow these steps:

1. Define the business object file, module (if any), and interface.

   When you define the interface, make sure you define attributes that you can map to the attributes of your data object.
2. From the pop-up menu of the business object interface, click **Add Implementation** to open the Business Object Implementation wizard to the Name and Data Access Pattern Page.
3. Under **Data Object Interface**, click **Add or select one later**.
4. Specify the rest of the properties of the business object implementation normally.
5. Click **Finish**. The business object implementation is added to the User-Defined Business Objects folder, under the business object interface.
6. From the pop-up menu of the business object implementation, click **Select Data Object Interface**. The Data Object Interface Connection wizard appears, open to the Selection page.
7. Type the name of the data object you want to map to, or select it from the drop-down list.
8. Click **Next** to show the Attributes Mapping page.
9. For each attribute in the business object implementation, add a mapping to an equivalent attribute in the data object interface. The mapping must be one-to-one, and the types of the mapped attributes must be identical.
10. Click **Finish**. The business object and data object are now associated, and the data object interface appears in the User-Defined Business Object's folder, underneath the business object implementation you mapped it to.

RELATED CONCEPTS

"Data Object" on page 18
"Business Object" on page 17

RELATED TASKS

"Create a Business Object File" on page 282
"Add a Business Object Module" on page 282
"Add a Business Object Interface" on page 283
"Map a Business Object to a Data Object" on page 288

Map a Business Object to a Data Object

Once you have added a business object and its implementation, and have specified that it uses the meet-in-the-middle approach, you can map it to an existing data object. You can map both attributes and methods of one object to the other. To do so, follow these steps:

1. From the business object implementation's pop-up menu, click **Select Data Object Interface**. The Data Object Interface Connection wizard opens to the Selection Page.
2. In the **Data Object Interface Name** field, type the name of an existing data object interface in the form *data_object_name data_object_interface_name,* or select one of the interface names from the list.

   Note: If you select a data object, which was created using the bottom-up approach (that is, one created from a persistent object), it is recommended that you initialize the attributes of the data object before you map the attributes of the business object to those of the data object. (They are not initialized by default.) So, follow these steps:

   a. Click **Finish.** This adds the selected data object interface to the User-Defined Business Objects folder, beneath the business object implementation.
   b. Initialize the attributes of the data object. Follow these steps:
      1) Select the data object interface in the User-Defined Data Objects folder.
      2) From the pop-up menu of the data object interface, select **Properties**. The Data Object Interface wizard opens.
      3) Click **Next**, or click the arrow to the left of the page name, and select Attributes Page from the list. The page opens.
      4) Select the data object attributes from the Attributes folder and type the initial value for the attribute in the **Initializer** field.
   c. Continue with the mapping: from the pop-up menu of the data object interface (which was selected for the business object), select **Properties**. The Data Object Interface Connection wizard opens.
   d. Click **Next**, or click the arrow to the left of the page name, and select Attributes Mapping Page from the list. The page opens.
   e. Follow step 4.
3. Click **Next.** The Attributes Mapping Page opens.
4. On this page, you can map the business object attributes to the data object attributes available from the business object interface. A business object attribute can map to a single data object attribute of the same data type: the mapping is one-to-one.

   To map the business object attributes to the data object attributes, follow these steps:

   a. From the pop-up menu of the Business Object Attributes folder, select **Add**.
   b. Type the name of an attribute of the data object interface you specified earlier, or select one from the **Data Object Attribute** field's list.

   Click **Next.** The Methods Mapping Page opens.
5. On this page, you can map the business object methods to the data object methods. You can only have a one-to-one mapping between any business object methods and data object methods, and the signatures of the two methods must be the same.

   To map a business object method to data object method, follow these steps:

a. From the pop-up menu of the Business Object Methods folder, select **Add**.
b. Select a method of the data object from the **Data Object Methods** field's list. This list contains only those methods of the data object that you defined for its interface.

6. Click **Next**, and add any comments about the mapping on the Comments Page.

The current mapping takes effect when you proceed to map the next business object attribute, or when you click **Finish**. The selected data object interface appears in the User-Defined Business Objects folder, under the business object implementation node.

Note: After you have mapped a business object to a data object, you can view and edit the mapping of the attributes on the Attributes Mapping Page and the Methods Mapping Page of the Business Object Implementation wizard. This wizard will no longer have the Data Object Interface Page.

RELATED CONCEPTS

"Business Object" on page 17
"Data Object" on page 18

RELATED TASKS

"Work with Business Objects" on page 281
"Work with Data Objects - Overview" on page 296

RELATED REFERENCES

"DB2 Data Type Mappings" on page 110
"Oracle Data Type Mappings" on page 113

Create a Business Object Interface by Importing an IDL File

If you have code already in IDL files, you can parse the code into Object Builder, and incorporate the classes, relationships, and code in the IDL files into your Object Builder application.

Note: The IDL must be CORBA 2.0-compliant without IDL extensions. You can make sure the IDL files you are importing are valid by compiling them first. Object Builder will only import IDL files that are considered valid by the compiler.

To import an existing IDL file, follow these steps:

1. Under Tasks and Objects, select the User-Defined Business Objects folder.
2. From the folder's pop-up menu, select **Import IDL**. The Import IDL wizard opens to the File Selection Page.
3. Browse for and select the files you want to import. The files you select, and any files they include, will be parsed and imported into Object Builder.
4. Click **Next**. The Search Paths for Nested Files Page opens.
5. From the Include directories pop-up menu, select **Add**. Browse for the directories you want searched.

   When you import a file that includes other files (that is, a file with nested files), the import process will search for the other files in the directories you specify here.
6. Click **Finish**. The selected files (and files they include) are parsed into Object Builder, and the information in the IDL is added to the current project model as business object files, business object modules, and business object interfaces.

Edit a Business Object Interface

Business object interfaces are defined in the User-Defined Business Objects folder, where they are shown under the file (and module, if any) in which they are defined. You can edit the file, module, and business object interface as separate objects, following these steps:

1. From the pop-up menu of the file, module, or interface, click **Properties** to display the appropriate wizard.
2. Click the title bar to select a page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

Note the following points:

- If you want to specify a parent for the interface after you have defined the implementation for the business object, follow these steps:
  1. Add the parent to the **Parents** folder on the Interface Inheritance page of the Business Object Interface wizard
  2. Open the Business Object Implementation wizard, and on the Name and Data Access Pattern page specify the pattern for handling state data as **Same as parent's**.
  3. Click **Next.**
  4. Add the implementation parent on the Implementation Inheritance page.
- If you want to change the order of the business object file contents, follow these steps:
  1. Open the file's wizard.
  2. Click the title bar and turn to the Contents Ordering page.
  3. Move elements into the new order.
  4. Click **Order by Dependency** to validate the new order.
  5. Click **Finish**.

RELATED CONCEPTS

"Business Object" on page 17
"Dependencies within an IDL File" on page 129

RELATED TASKS

"Work with Business Objects" on page 281

Edit a Business Object Implementation

Business object implementations are defined in the User-Defined Business Objects folder, where they are shown under the business object interface they were added to. You can edit a business object implementation by following these steps:

1. From the pop-up menu of the business object implementation, click **Properties**. The Business Object Implementation wizard opens to the Name and Data Access Pattern Page.
2. Click the title bar to select another page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

Note: The **Same as parent's** option is selected by default if the interface for this business object inherits from another business object interface. However, you still have to indicate the implementation parent on the Implementation Inheritance page of this wizard, after you delete the default parent for business object implementations, which is IManagedClient IManagedClient::IManageable.

RELATED CONCEPTS
"Business Object" on page 17

RELATED TASKS
"Work with Business Objects" on page 281

Delete a Business Object Interface

To delete a business object interface, follow these steps:

1. Locate the business object interface in the User-Defined Business Objects folder.
2. Delete any managed objects defined off of the interface's business object implementation.
3. Delete or remove the data object interface defined off of the business object implementation.
4. Delete the business object implementation.
5. Delete any keys and copy helpers defined off of the interface.
6. From the pop-up menu of the business object interface, click **Delete**.

When you delete a business object interface, any methods, attributes, constructs, or one-to-many relationships that use it as a type have the type changed to invalidType. For example, if you delete the interface of Agent, then an attribute Agent custAgent becomes attribute invalidType custAgent. You can find all occurences of invalidType by running the model consistency checker.

RELATED CONCEPTS
"Business Object" on page 17

RELATED TASKS
"Work with Business Objects" on page 281
"Check a Model for Consistency" on page 412

Delete a Business Object Implementation

To delete a business object implementation, follow these steps:

1. Locate the business object implementation in the User-Defined Business Objects folder.
2. Delete any managed objects defined off of the business object implementation.
3. Delete or remove the data object interface defined off of the business object implementation.
4. From the pop-up menu of the business object implementation, click **Delete**.

RELATED CONCEPTS
"Business Object" on page 17

RELATED TASKS
"Work with Business Objects" on page 281

"Delete a Managed Object" on page 341
"Delete a Data Object Interface" on page 312

Work with Keys

Keys are defined in the User-Defined Business Objects folder, where you can add them from the pop-up menu of a business object interface.

The key provides a way for the client to locate a specific instance of a component on the server.

You can add multiple keys to a business object interface, but each component you configure can only have one key.

The following tasks deal with keys:
- "Add a Key"
- "Edit a Key" on page 293
- "Delete a Key" on page 293

RELATED CONCEPTS
"Key" on page 21

Add a Key

Each business object must have a primary key class that contains enough information to uniquely identify the object. The key is used when new instances of the object are created or when existing instances need to be found.

To add a key, follow these steps:
1. From the User-Defined Business Objects folder, select your business object interface (for example, CarPolicy).
2. From the object's pop-up menu, select **Add Key**. The Key wizard opens to the Name and Key Attributes Page.
3. Appropriate key names are filled in for you (the business object file name and interface name plus Key: for example, CPFile::CarPolicy gets a key named CPFileKey::CarPolicyKey). You can accept these defaults or replace them with your own names.
4. Select the business object attributes that make up the primary key. If the business object has a parent interface, you can also select from the parent interface's attributes (you should **not** select attributes of the parent interface if you are planning to inherit from the parent interface's key).
5. Click **Next**. The Implementation Inheritance Page opens.

   On this page, you can specify the type of key (primary or unique), and inherit from the appropriate parent class (IPrimaryKey or IUniqueKey) .

   If the key has a parent, you can specify it here.

   Note: You should not inherit from a parent key if you also selected inherited attributes on the previous page.
6. Click **Next**. The Summary of Framework Methods Page opens. This page summarizes the framework methods this object implements. No action is needed.

7. Click **Next**. The Optional Framework Methods Page opens. Select any additional framework methods you want to implement. Object Builder will add signatures for the methods you select, but you must provide your own implementation code. The methods you implement will override the equivalent framework methods of the parent class.

   Note: The editor pane will not allow you to edit these methods until you set them as editable in the Method Implementation wizard. To set a method as editable, follow these steps:

   a. In the Methods pane, select the framework method.
   b. From its pop-up menu, click **Properties**.
   c. In the Method Implementation wizard, specify that you want to use the editor pane.
8. Click **Finish**. The key appears in the User-Defined Business Objects folder, under your business object interface.

In the Methods pane, you should see some items listed in the Framework Methods folder. Default implementation code is provided for these methods, which you can view in the edit pane by selecting a method. Normally, you will not want to edit this code (except for the code for the optional framework methods, as noted above). The code for framework methods is read-only by default.

RELATED CONCEPTS
"Key" on page 21

RELATED TASKS
"Work with Keys" on page 292
"Add a Copy Helper" on page 294

Edit a Key

Keys are defined in the User-Defined Business Objects folder, where they are shown under the business object interface they were added to. You can edit a key by following these steps:

1. From the pop-up menu of the key, click **Properties**. The Key wizard opens to the Name and Key Attributes Page.
2. Click the title bar to select a page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

RELATED CONCEPTS
"Key" on page 21

RELATED TASKS
"Work with Keys" on page 292

Delete a Key

To delete a key, follow these steps:

1. Remove the key from any business object implementations that are configured with it.
2. Remove the key from any data object implementations that are configured with it.

3. Remove the key from any managed object configuration that uses it.
4. Locate the key in the User-Defined Business Objects folder.
5. From the key's pop-up menu, click **Delete**.

RELATED CONCEPTS
"Key" on page 21

RELATED TASKS
"Work with Keys" on page 292
"Edit a Business Object Implementation" on page 290
"Edit a Data Object Implementation" on page 310
"Edit a Managed Object Configuration" on page 379

Work with Copy Helpers

Copy helpers are defined in the User-Defined Business Objects folder, where they are shown below the business object interface they were added to.

The copy helper is an optional object that provides a way to initialize multiple attributes of a component instance with a single call to the server.

You can add multiple copy helpers to a business object interface, but each component you configure can only have one copy helper.

The following tasks deal with copy helpers:
- "Add a Copy Helper"
- "Edit a Copy Helper" on page 295
- "Delete a Copy Helper" on page 295

RELATED CONCEPTS
"Copy Helper" on page 21

Add a Copy Helper

The copy helper is an optional component object that lets you initialize the attributes of a new component on the server with a single call. It embodies the business object attributes that you will want to initialize.

To add a copy helper, follow these steps:
1. From the User-Defined Business Objects folder, select your business object interface (for example, CarPolicy).
2. From the object's pop-up menu, select **Add Copy Helper**. The Copy Helper wizard opens to the Name and Attributes Page.
3. Appropriate copy helper names are filled in for you (the business object file name and interface name plus Copy: for example, CPFile::CarPolicy gets a copy helper named CPFileCopy::CarPolicyCopy). You can accept these defaults or replace them with your own names.
4. Select which business object attributes to externalize in the copy helper. If the business object has a parent interface, you can also select from the parent interface's attributes (you should **not** select attributes of the parent interface if you are planning to inherit from the parent interface's copy helper).

5. Click **Next**. The Implementation Inheritance Page appears.

   By default, the copy helper inherits from IManagedLocal IManagedLocal::INonManageable. This is the correct choice if the copy helper is for a component without parents.

   Note: You should not inherit from a parent copy helper if you also selected inherited attributes on the previous page.
6. Click **Next**. The Summary of Framework Methods Page opens. This page summarizes the framework methods this object implements. No action is needed.
7. Click **Finish**. The key appears in the User-Defined Business Objects folder, under your business object interface.

In the Methods pane, you should see some items listed in the Framework Methods folder. Default implementation code is provided for these methods, which you can view in the edit pane by selecting a method. By default, this code is read-only.

RELATED CONCEPTS
"Copy Helper" on page 21

RELATED TASKS
"Work with Copy Helpers" on page 294
"Add a Business Object Implementation and Data Object Interface" on page 284

Edit a Copy Helper

Copy helpers are defined in the User-Defined Business Objects folder, where they are shown under the business object interface they were added to. You can edit a copy helper by following these steps:
1. From the pop-up menu of the copy helper, click **Properties**. The Copy Helper wizard opens to the Name and Attributes.
2. Click the title bar to select a page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

RELATED CONCEPTS
"Copy Helper" on page 21

RELATED TASKS
"Work with Copy Helpers" on page 294

Delete a Copy Helper

To delete a copy helper, follow these steps:
1. Remove the copy helper from any data object implementations that are configured with it.
2. Remove the copy helper from any managed object configurations that use it.
3. From the pop-up menu of the copy helper, click **Delete**.

RELATED CONCEPTS
"Copy Helper" on page 21

RELATED TASKS

"Work with Copy Helpers" on page 294
"Edit a Data Object Implementation" on page 310
"Edit a Managed Object Configuration" on page 379

Work with Data Objects - Overview

A data object manages the state of a business object. It encapsulates the object's persistent behavior, if there is any.

In the User-Defined Data Objects folder, a data object is fully presented in terms of four objects:

- The data object file (which contains one or more interfaces, optionally organized into modules)
- The data object module, if any (which contains one or more interfaces)
- The data object interface (which has one or more implementations)
- The data object implementation (which has its own file, defined on the first page of its wizard)

You can create the four objects separately when you create a data object interface that is not connected to a business object. Collectively, these four objects form a single data object. Each data object (each set of data object file, module, interface, and implementation) typically has its own persistent object.

The four objects are created automatically in the User-Defined Data Objects folder when you perform one of the following actions:

- "Add a Data Object from a DB Persistent Object" on page 304
- "Add a Data Object from a PA Persistent Object" on page 305
- "Add a Data Object Implementation" on page 299

The following tasks deal with data objects:

- "Create a Data Object File" on page 303
- "Add a Data Object Module" on page 304
- "Create a Data Object Interface" on page 297
- "Add a Business Object Implementation and Data Object Interface" on page 284
- "Create a Data Object Interface by Importing an IDL File" on page 306
- "Add a Data Object Implementation" on page 299
- "Edit a Data Object Interface" on page 309
- "Edit a Data Object Implementation" on page 310
- "Delete a Data Object Interface" on page 312
- "Delete a Data Object Implementation" on page 313
- "Add a Data Object from a DB Persistent Object" on page 304
- "Add a Data Object from a PA Persistent Object" on page 305
- "Map a Data Object to a DB Persistent Object" on page 251

- Map a Data Object to a PA Persistent Object

- “Map Data Object Attributes to Persistent Object Attributes” on page 256
- “Add a Business Object from a Data Object” on page 287
- “Map a Business Object to a Data Object” on page 288

Note: In the User-Defined Business Objects folder, the DBA-Defined Schemas folder and the User-Defined PA Schemas folder, the data object is only presented in terms of its interface and implementation.

RELATED CONCEPTS
“Data Object” on page 18
“Persistent Object” on page 19
“Business Object” on page 17

Create a Data Object Interface

You can add a data object interfaces in different situations, from the following folders:

- User-Defined Business Objects folder
- User-Defined Data Objects folder
- DBA-Defined Schemas folder

From the User-Defined Business Objects folder

Once you have deleted a data object interface that was created along with a business object implementation, you can create a new one to be associated with the implementation. Follow these steps:

1. From the pop-up menu of the unassociated business object implementation in the User-Defined Business Objects folder, select **Add New Data Object Interface**. The Add New Data Object wizard opens to the Data Object Interface Page.
2. Accept the default data object file name and interface name, or rename them, and select those attributes of the business object to be used as state data in the data object.
3. Click **Next**. The Data Object Methods Page opens.
4. Select the methods of the business object that are to be delegated to the data object. The methods you select form part of the data object’s interface.
5. Click **Next**. The Constructs Page opens. Use the pop-up menu of the Constructs folder to add constants, enumerations, exceptions, typedefs, structures, or unions. Any constructs you add are scoped to this interface only.

   Note: To use the construct as the type of an attribute, method return, or method exception, you must first click **Finish** and then reopen the wizard and define the attribute. The construct is not added to the current model until you click **Finish**.
6. The Interface Inheritance Page opens. Here you can specify one or more classes from which the interface can inherit. Click the list button of the **Parent Interface** and select a parent from the list of available classes, or type the interface name using the following syntax: *filename interface_name*
7. When you have specified all the parents for this interface, click **Next.** The Comments Page opens. Type any comments you want to include as comment lines in your generated code.
8. Click **Finish.** The interface is added to the folder.

From the User-Defined Data Objects folder

In this folder, you can create a data object interface in the following ways:

- From a data object file
- From a data object module
- By importing an IDL file

From a data object file

1. Select the User-Defined Data Objects folder.
2. From its pop-up menu, select **Add File**. The Data Object File wizard opens to the Name Page.
3. Specify a name for the data object file.
4. Click **Finish**. The data object file is added to the folder.
5. Select the file, and from its pop-up menu, select **Add Interface**. The Data Object Interface wizard opens to the Name Page.
6. Specify a name for the interface.
7. Click **Next** if you want to define constructs at the interface level.
8. Go to the Interface Inheritance Page if you want this interface to inherit from an existing one.
9. Go to the Attributes Page if you want to define attributes that are specific to the data object interface.
10. Go to the Methods Page if you want to define methods for the interface.
11. Add any comments you want to, on the Comments Page.
12. Click **Finish**.

The data object interface is added to the folder, and appears as a node beneath the file.

From a data object module

If you want the data object interface to be scoped within a module, follow this method.

Follow steps 1 to 4 that are outlined in the previous method. Once the data object file is created, follow these steps:

1. Select the file, and from its pop-up menu, select **Add Data Object Module**. The Data Object Module wizard opens to the Name Page.
2. Type a name for the module.
3. Click **Next** if you want to add constructs at the module level.
4. Go to the Comments Page if you want to add comments about the module.
5. Click **Finish**. The data object module is created, and appears as a node beneath the data object file.
6. Select the module in the folder, and from its pop-up menu, select **Add Interface**. The Data Object Interface wizard opens to the Name Page.

Follow steps 6 to 12 as when you added the interface from the file. The data object interface is added to the folder, and appears as a node beneath the module.

By importing an IDL file

This is actually a method of reusing an existing data object interface. You add it, or create it within Object Builder by importing it. Follow these steps:

1. Select the User-Defined Data Objects folder.
2. From its pop-up menu, select **Import - IDL**. The Import IDL wizard opens to the IDL File Selection Page.
3. From the IDL Files folder's pop-up menu, select **Add**. You can then specify the IDL file to be imported in the **File Name** field. You can also use the Find button to open the File to Import dialog box. Use it to view the contents of the different drives and find the exact path for the IDL file to be imported.
   Note: The IDL must be CORBA 2.0-compliant without IDL extensions. You can make sure the IDL files you are importing are valid by compiling them first. Object Builder will only import IDL files that are considered valid by the compiler.
4. Click **Next**. The Search Paths for Nested Files Page opens.
   Include files (that is, those that are nested in another file), which are not already in the model, have to be imported. Other include files, (for example, IManagedClient), which exist in the model, do not have to be imported.
5. Indicate the directories that must be searched for the include files that are specified in the IDL file to be imported: from the pop-up menu of the Directories folder, select **Add**. Type the include directory in the **Directory** field.
6. Click **Finish**.

The data object interface appears in the folder beneath the file that contains it, or if it is scoped within a module, beneath the module that contains it.

From the DBA-Defined Schemas folder
This method assumes you have imported an SQL file into Object Builder, and added a persistent object from it. Follow these steps to add a data object interface from the persistent object:
1. Select the persistent object that you added to the imported schema.
2. From its pop-up menu, select **Add Data Object**.
3. The Add Data Object wizard opens to the Names Page.
4. Specify the names for the data object interface and its file, and for the associated data object implementation and its file.
5. Click **Next** to go to the Methods Page, if you want to add methods for the data object.
6. Click **Finish**.

The data object interface appears beneath its file in the User-Defined Data Objects folder, with the data object implementation beneath it, with the implementation connected to the persistent object, which in turn is connected to the scehma.

RELATED CONCEPTS
"Data Object" on page 18
Query Service (*Advanced Programming Guide*)

RELATED TASKS
"Work with Data Objects - Overview" on page 296
"Create a DB Schema by Importing an SQL File" on page 321
"Add a Persistent Object from a DB Schema" on page 316
"Create a Data Object Interface by Importing an IDL File" on page 306

Add a Data Object Implementation

Once you have either created or selected the data object interface for your business object, you can create a data object implementation. The business object

is dependent on the data object interface, but not on its implementation. However, for the business object to be of any use, the data object interface must be implemented. The data object implementation can emulate the real application environment, with either a local-only test environment or a full client-server setup.

To create a data object implementation, follow these steps:

1. From the Tasks and Objects pane, select the data object interface for which you want to create an implementation (for example, CarPolicyDO).
2. From the pop-up menu of the interface, select **Add Implementation**. The Data Object Implementation wizard opens to the Name and Platform Page.
3. Type the name of the implementation class and its file, or accept the default names (for example CarPolicyDOImpl for the implementation name and FileDOImpl for the file name). If the data object interface is contained in a module, specify the module name in the **Module Name** field.
4. Click **Next**. The Behavior Page opens.
5. Select the environment for testing the object in the Environment (page 31) section.
6. Select the form of persistent behavior and implementation from the Form of Persistent Behavior and Implementation (page 32) section of the same page. All options in this section are available for selection only if you have selected **BOIM with any key** in step 5.
   390 All Cache Service options are not available when the target platform is OS/390.
7. Select the data access pattern to be used in the data object implementation from the Data Access Pattern (page 34) section. This section is available for selection only if you have selected **BOIM with any key** in step 5. The access pattern can be either **Delegating**, which is the default option, or **Local copy**.
8. Select the handle for storing references from the Handle for Storing Pointers (page 35) section.
9. Click **Next**. The Implementation Inheritance Page opens. Click the list button of the **Parent Class** field and select a parent, or accept the default, which is provided by Object Builder.
10. Click **Next**. The Attributes Page opens. You can use this page to add more attributes for the data object. These attributes will be specific to this implementation.
11. Click **Next**. The Methods Page opens. Here, you can add implementation-specific methods for the data object. These methods can access the implementation-specific attributes you added on the previous page. These methods can also be called from within other methods that you define for the data object interface.
12. Click **Next**. The Select Key and Copy Helper Page opens. Select the key and the copy helper to use with this implementation.
    Note: If you selected **BOIM with any key** in step 5, and there is at least one persistent object in the model that has the same type of persistence as that for the implementation, which you specify in step 6, the Associated Persistent Objects Page is added to the wizard, and you can continue with step 13; if not, continue with step 16.
13. Click **Next.** The Associated Persistent Objects Page opens. You can specify existing persistent object instances that are to be associated with the data

object. These instances are stored as protected members of the data object implementation.

Note the following points:

- At this point, you can also finish the wizard and add a persistent object and schema from the data object implementation. The persistent object you create is automatically associated with the data object and appears in the Persistent Object Instances folder on the Associated Persistent Objects Page.
- If you associated one or more existing persistent object instances with the data object, the Attributes Mapping Page and the Methods Mapping Page are added to the wizard. You can map the data object to the persistent object: continue with step 14. If you did not associate any persistent object with the data object, continue with step 16.

14. Click **Next.** The Attributes Mapping Page opens. You can specify the mapping between the data object attributes and the persistent object attributes.
15. Click **Next.** The Methods Mapping Page opens. For each of the special framework methods associated with the implementation, you can specify the persistent object methods that it calls. The order of the methods in the tree determines the order in which they are called.

    Note: If you have associated any Procedural Adaptor (PA) persistent objects with this implementation, this page has the User-Defined Methods folder as well. This folder contains the methods you defined for the data object using the Methods Page of the Data Object Interface wizard. You can map each of these methods to the corresponding push-down method of the persistent object, if you want these methods to be called immediately.
16. Click **Next**. The Summary of Framework Methods Page opens. On this page you can review the framework methods that are implemented by Object Builder for this class. These methods have to be implemented for the data object to work properly in the server.
17. Click **Finish**. The data object implementation appears in the Tasks and Objects pane, with the appropriate methods added in the Method List pane. You can now add your own code to implement those methods. None of the framework methods have implementations at this point. Object Builder provides the code for these methods as soon as there is a persistent object associated with this data object.

RELATED CONCEPTS

“Data Object” on page 18
“Persistent Object” on page 19
Object Relationships (*Programming Guide*)
Application Adaptor (*Programming Guide*)
Data Object Customization for Cardinality Relationships (*Programming Guide*)
“State Data” on page 18
Data Object Customization (*Programming Guide*)
“Container” on page 345
“Home” on page 342
Using Handles (*Programming Guide*)
Naming Service (*Advanced Programming Guide*)
Cache Service (*Advanced Programming Guide*)
“Special Framework Methods” on page 24
“Framework Methods” on page 24
Using Sets of Objects (Using Reference Collections) (*Programming Guide*)

RELATED TASKS

“Work with Data Objects - Overview” on page 296

"Add a Persistent Object and Schema" on page 313
"Create a Component for New DB Data - Scenario" on page 102
"Customize Referential Integrity" on page 108
"Create a Container Instance" on page 346
"Configure a Managed Object" on page 377
"Work with Attributes" on page 247"Work with Methods " on page 267
"Add Code for User-Defined Methods" on page 267
"Add endResource() to a Sessional Business Object" on page 117

RELATED REFERENCES
"Data Object Implementation Inheritance" on page 36

Add a Data Object From a Business Object

A business object is normally created with its own data object. If however, you delete the associated data object, or select the **Add or select one option later** option when you add the business object implementation (in the "Data Object Interface" on page 29 section on the Name and Data Access Pattern Page of the Busisness Object Implementation wizard), you can either select a data object interface that exists in the model, or define an entirely new one for the business object.

To add a data object from a business object, follow these steps:

1. From the pop-up menu of the business object, select Add Data Object Interface. The Add New Data Object wizard opens to the Data Object Interface Page.

   Appropriate data object names are filled in for you (the business object file name and interface name plus DO: for example, CarPolicy gets the data object interface CarPolicyDO). You can accept these defaults or replace them with your own names.

   Select the business object attributes that you want preserved in the data object. These attributes constitute the state data for the component.
2. Click **Next** to open the Constructs Page. Use the Constructs pop-up menu to add constants, enumerations, exceptions, typedefs, structures, or unions. Any constructs you add are scoped to this interface only.

   Note: To use the construct as a type within another construct, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.
3. Click **Next** to open the Interface Inheritance Page.Here you can specify one or more classes from which the interface can inherit. Click the list button of the **Parent Interface** and select a parent from the list of available classes, or type the interface name using the following syntaxes:
   filename interface_name (if the interface is stand-alone)
   filename module_name::interface_name (if the interface derives from a module)
4. When you have specified all the parents for this interface, click **Next**.
5. Type any comments you need to add, on the Comments Page.
6. Click **Finish.** The interface is added to the folder.

RELATED CONCEPTS
"Business Object" on page 17
"Data Object" on page 18

RELATED TASKS
"Work with Data Objects - Overview" on page 296

Create a Data Object File

A data object file (IDL) can hold multiple data object interfaces, which you may organize into modules. However, you typically add one interface to each file.

To create a data object file, follow these steps:

1. From the Tasks and Objects pane, select the **User-Defined Data Objects** folder.
2. From the folder's pop-up menu, select **Add File**. The Data Object File wizard opens to the Name Page.
3. Type a name for the file.
4. Click **Next**. The Constructs Page opens.

   Use the Constructs pop-up menu to add constants, enumerations, exceptions, typedefs, structures, or unions. Any constructs you add are scoped to this file only.

   Note: To use the construct as the type of an attribute, method return, method exception, or as a type within another construct, you must first click **Finish** and then reopen the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.
5. Click **Next**. The Files to Include Page opens.

   If your component had a parent, you would specify the data object file of the parent component in this field. For example, if the CarPolicy component inherits from the Policy component, then you would specify the data object file for Policy on this page. The data object files for any referenced or related components are automatically added to the Include Files folder. For example, if the CarPolicy interface has an attribute of type Claim, the data object file for Claim is automatically included on this page as soon as the attribute is defined for the interface.
6. Click **Next**.
   Note: If the file has constructs or other interfaces defined in it, continue with step 6; otherwise, continue with step 7.
7. The Contents Ordering Page opens. This page enables you to view the order of elements within the IDL file. You can also change the order of these constructs and interfaces within the file by using either the **Move Up** and **Move Down** buttons, or the **Order by Dependency** button. When you are satisfied with the order of elements, click **Next.**
8. The Comments Page opens. Type any comments you want to include as comment lines in your generated IDL code.
9. Click **Finish**. The wizard closes, and your file is added to the User-Defined Data Objects folder. You can now add modules or interfaces to the file.

Once you have created the file, you can modify it by selecting **Properties** from its pop-up menu. The Data Object File wizard opens again, with your selections preserved. You can change the name of the file and the construct names or their types only if the file was not defined in another model. (If the file was defined in another model, and you specify that model as a project dependency when you open the current project, the project dependency model is read-only, and can be used for inheritance purposes, and for reuse of interfaces, attribute types and constructs.)

RELATED CONCEPTS
"Data Object" on page 18

RELATED TASKS
"Add a Business Object Module" on page 282
"Create a Data Object Interface" on page 297

Add a Data Object Module

If you plan to add multiple data object interfaces to a single file, you may want to store the interfaces in separate modules. To add a module to a file, follow these steps:

1. From the User-Defined Data Objects folder, select your data object file.
2. From the file's pop-up menu, select **Add Module**. The Data Object Module wizard opens to the Name Page.
3. Type a name for the module.
4. Click **Next**. The Constructs Page opens.

   Use the Constructs pop-up menu to add constants, enumerations, exceptions, typedefs, structures, or unions. Any constructs you add are scoped to this module only.

   Note: To use the construct as the type of an attribute, method return, method exception, or as a type within another construct, you must first click **Finish** and then reopen the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.
5. Click **Next**. The Comments Page opens. Type any comments you want to include as comment lines in your generated code.
6. Click **Finish**. The wizard closes, and your module is added to the User-Defined Data Objects folder, underneath the file.

You can now add data object interfaces to the module.

RELATED TASKS
"Create a Component for New DB Data" on page 101
"Create a Data Object Interface" on page 297

Add a Data Object from a DB Persistent Object

To add a data object to a persistent object, follow these steps:

1. From the pop-up menu of the persistent object in the DBA-Defined Schemas folder, select **Add Data Object.** The Add Data Object wizard opens to the Names Page.
2. Type a name for the interface file in the **Interface File Name** field, or accept the default.
3. Type a name for the data object interface in the **Interface Name** field, or accept the default.
4. Type a filename for the data object implementation in the **Implementation File Name** field, or accept the default.
5. Type a name for the data object implementation in the **Implementation Name** field, or accept the default.
6. Click **Finish**.

The data object appears in the User-Defined Data Objects folder under the data object filename you provided. The data object interface exists in the tree as a child of the data object file, and the data object implementation exists as a child of the

interface. The data object implementation has the persistent object as its child node and the persistent object has the schema as its child node.

Note the following points:

- A default mapping is generated between the attributes of the data object and the persistent object. If there are mappings generated that need a mapping helper, Object Builder will inform you for which pairs of attributes it is required. You can then follow these steps:
  1. Select the data object implementation that was just created.
  2. Select **Properties** from its pop-up menu. The Data Object Implementation wizard opens.
  3. Click **Next**, or click the arrow to the left of the page name, and select. Attributes Mapping Page from the list. The page opens.
  4. Select **Map using a helper class** and provide the mapping helper class and method names for each pair of attributes.
- You can initialize the attributes of the data object interface that is created. (They are not initialized by default.) Follow these steps:
  1. Select the data object interface in the User-Defined Data Objects folder.
  2. From the pop-up menu of the data object interface, select **Properties**. The Data Object Interface wizard opens.
  3. Click **Next**, or click the arrow to the left of the page name, and select Attributes Page from the list. The page opens.
  4. Select the data object attributes from the Attributes folder and type the initial value for the attribute in the **Initializer** field.

RELATED CONCEPTS

"Data Object" on page 18
"Persistent Object" on page 19
"Schema" on page 20

RELATED TASKS

"Work with DB Persistent Objects" on page 313
"Map Attributes Using a Mapping Helper" on page 260

RELATED REFERENCES

"DB2 Data Type Mappings" on page 110
"Oracle Data Type Mappings" on page 113

Add a Data Object from a PA Persistent Object

To add a data object from a PA persistent object, follow these steps:

1. From the pop-up menu of the PA persistent object in the User-Defined PA Schemas folder, select **Add Data Object.** The Add Data Object wizard opens to the Names Page.
2. Type a name for the interface file in the **Interface File Name** field, or accept the default.
3. Type a name for the data object interface in the **Interface Name** field, or accept the default.
4. Type a filename for the data object implementation in the **Implementation File Name** field, or accept the default.
5. Type a name for the data object implementation in the **Implementation Name** field, or accept the default.

6. Click **Next**. The Methods Page opens. To specify methods for your interface, select **Add** from the Methods folder's pop-up menu.

The data object appears in the User-Defined Data Objects folder under the data object filename you provided. The data object interface exists in the tree as a child of the data object file, and the data object implementation exists as a child of the interface. The data object implementation has the PA persistent object as its child node and the persistent object has the PA schema object as its child node.

Note the following points:

- A default mapping is generated between the attributes of the data object and the persistent object. If there are mappings generated that need a mapping helper, Object Builder will inform you for which pairs of attributes it is required. You can then follow these steps:
  1. Select the data object implementation that is just created.
  2. Select **Properties** from its pop-up menu. The Data Object Implementation wizard opens.
  3. Click the arrow to the left of the page name, and select Attributes Mapping Page from the list.
  4. Select **Map using a helper class** and provide the mapping helper class and method names for each pair of attributes.
- You can initialize the attributes of the data object interface that is created. (They are not initialized by default.) Follow these steps:
  1. Select the data object interface in the User-Defined Data Objects folder.
  2. From the pop-up menu of the data object interface, select **Properties**. The Data Object Interface wizard opens.
  3. Click **Next**, or click the arrow to the left of the page name, and select Attributes Page from the list. The page opens.
  4. Select the data object attributes from the Attributes folder and type the initial value for the attribute in the **Initializer** field.

RELATED CONCEPTS

"Data Object" on page 18
"Persistent Object" on page 19
"Schema" on page 20

RELATED TASKS

"Create a Component for PA Data" on page 115
"Work with PA Persistent Objects - Overview" on page 333
"Map Attributes Using a Mapping Helper" on page 260

Create a Data Object Interface by Importing an IDL File

If you have code already in IDL files, you can parse the code into Object Builder, and incorporate the classes, relationships, and code in the IDL files into your Object Builder application.

Note: The IDL must be CORBA 2.0-compliant without IDL extensions. You can make sure the IDL files you are importing are valid by compiling them first. Object Builder will only import IDL files that are considered valid by the compiler.

To import an existing data object interface IDL file, follow these steps:

1. Under Tasks and Objects, select the User-Defined Data Objects folder.

2. From the folder's pop-up menu, select **Import IDL**. The Import IDL wizard opens to the IDL File Selection Page.
3. Browse for, and select the files you want to import. The files you select, and any files they include, will be parsed and imported into Object Builder.
   Note: The files will be imported under the User-Defined Business Objects folder. You will have to delete the data object interface files from the business object interface.
4. Click **Next**. The Search Paths for Nested Files Page opens.
5. From the **Directories** pop-up menu, select **Add**. Browse for the directories you want searched.

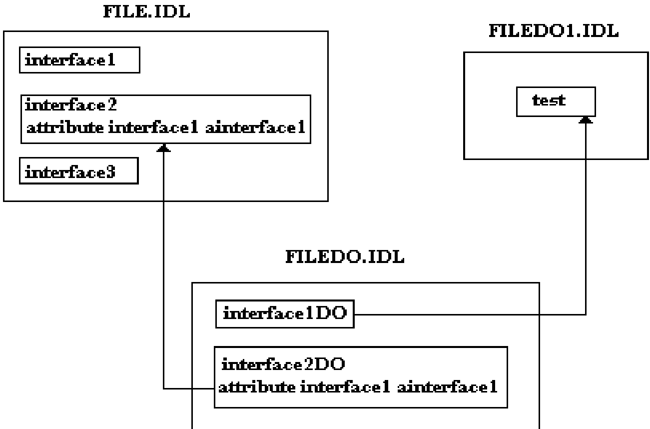
   When you import a file that includes other files (that is, a file with nested files), the import process will search for the other files in the directories you specify here.
6. Click **Finish**. The selected files (and files they include) are parsed into Object Builder, and the information in the IDL is added to the current project model as data object files, data object modules, and data object interfaces.

Consequences of importing an interface that has dependencies on other interfaces
If the data object interface that you import inherits from another object - either a business object or another data object, the interfaces of those objects are imported into the model as well, and appear under the User-Defined Business Objects folder.

Follow these steps to complete the import process:
1. Specify the directory the parent interfaces are in, on the Include Files Page of this wizard.
2. Delete the extraneous data object interfaces from the User-Defined Business Objects folder (from the pop-up menu of the interface, select **Delete**)
3. Import the same data object interfaces again into the User-Defined Data Objects folder.

   Note: Even after you import the data object interfaces, inheritance will no longer work because the definition of the interface changes as soon as the objects are deleted from the User-Defined Business Objects folder. The IDL file you generate from this interface will not contain the include statements. So, follow step 4.
4. Open the Data Object Interface wizard and add the parent interfaces on the Interface Inheritance Page.

Example
1. You have the following IDL files:



1. `FileDO.idl` has an interface that inherits from the test interface of the IDL file `FileDO1.idl` , and another one that has an attribute of a type defined in an interface of the file `File.idl` .
2. You try to import `FileDO.idl` (Select **Import IDL** from the pop-up menu of the User-Defined Data Objects folder and specify `FileDO.idl` as the file to be imported.)
3. The business object IDL file `File` is imported into the User-Defined Business Objects folder, along with its three interfaces. The data object IDL file `FileDO` is also imported into the User-Defined Business Objects folder, along with its interface.
4. You must delete the `FileDO1` from the User-Defined Business Objects folder.
5. Select **Import IDL** from the pop-up menu of the User-Defined Data Objects folder and this time, specify `FileDO1.idl` as the file to be imported.
6. Open `FileDO`'s Data Object Interface wizard. Turn to the Interface Inheritance Page and select `interface2` and `test` as the parent interfaces.
7. Click **Finish.**

Work-around

If you have an IDL file you want imported, you know the interfaces it depends on (the other data object interfaces it inherits from), and the other .idl files from which it wants to use information (for example, you want to use the exceptions defined in some business object interface IDL file), you can import the data object IDL file without the business object interface and other supporting data object interfaces being imported into the User-Defined Business Objects folder, if you follow these steps:

1. Import the data object interfaces from which this interface is to inherit, before importing the interface itself (on the IDL File Selection Page).
2. Include the business object interface IDL files which have the information required for the interface, on the Include Files Page of the same wizard.

Note: If a data object has an interface and you create a new interface for it, when you generate code for the data object (file level), the interface you just created will appear last in the code. If you want the old interface to inherit from the new one, and you specify this on the Interface Inheritance Page of the Data Object Interface wizard, and then generate code from the data object file, Object Builder places the definition of the new interface in the beginning, before the definition of the interfaces that inherit from it.

RELATED TASKS

"Create a Business Object Interface by Importing an IDL File" on page 289

Edit a Data Object Interface

You can edit a data object interface, whether it was created with a business object or created by itself.

To edit a data object interface that was created with a business object, follow these steps:

1. From the pop-up menu of the data object interface in the **User-Defined Business Objects** folder, select **Properties**. The Data Object Interface wizard opens to the Name Page. You cannot change the name of the data object file, or module. You can rename the data object interface: specify a new name in the **Name** field.
   Restriction: You cannot change the name of the interface if it inherits from another.
2. Click **Next**. The Interface Inheritance Page opens. You can add new parents for the interface, delete the ones specified earlier, or rename them.
3. Click **Next**. The Methods Page opens. Make changes as required.
4. Click **Next**. The Comments Page opens. Type any remarks, if required.
5. Click **Finish**. The changes you made to the interface are saved and can be viewed later by examining the same wizard using the same option from the data object interface's pop-up menu.

Note the following points:

- To ensure that valid code is generated after a rename, use **Generate - All** instead of **Generate - Selected** from the pop-up menu of the object.
- If the data object has dependent objects such as a business object, key, or copy helper, and you change an attribute of the data object, you must make the change in the dependent objects as well. For each of the objects, follow these steps:
-
  –
    1. Open the object's wizard.
    2. Click **Finish**.

To edit a stand-alone data object interface, follow these steps:

1. From the pop-up menu of the data object interface in the **User-Defined Data Objects** folder, select **Properties**. The Data Object Interface wizard opens to the Name Page. You can rename the data object interface if it does not inherit from another.
2. Click **Next**. The Interface Inheritance Page opens. You can add new parents for the interface, delete the ones specified earlier, or rename them.
3. Click **Next**. The Attributes Page opens. You can add new attributes for the interface, or delete or rename the ones specified earlier.

   Note the following points:

   - You can only access the Attributes Page when you are viewing the properties of the data object interface from the **User-Defined Data Objects** folder.

- If there is a data object implementation created from this data object interface follow these steps to refresh the model after you modify the interface in any way:
  a. From the pop-up menu of the data object implementation, select **Properties.** The Data Object Implementation wizard opens to the Name and Platform Page.
  b. Click **Finish,** or turn to any other page of the wizard and click**Finish.**

4.
   - If there is a persistent object associated with the data object that has a mapping defined between the attributes of these objects, and you change any of the data object's attributes, these attributes lose the defined mapping. You will either have to remap the attributes of the data object to those of the persistent object (on the Attributes Mapping Page of the Data Object Implementation wizard), or you can delete the persistent object and schema, and create a new one, which will automatically use the changed attributes.
5. Click **Next**. The Methods Page opens. Add new methods or make changes as required.
6. Click **Next**. The Comments Page opens. Type any remarks, if required.
7. Click **Finish**. The changes you made to the interface are saved and can be viewed later by opening the same wizard using the same option from the data object interface's pop-up menu.

Whichever data object interface you are editing, if you want to change the order of the file's contents (modules, interfaces and constructs within the file), follow these steps:

1. Open the data object file's wizard.
2. Click the arrow to the left of the page name, and select Contents Ordering Page from the list.
3. Move elements into the new order.
4. Click **Order by Dependency** to validate the new order.
5. Click **Finish**.

RELATED CONCEPTS
"Data Object" on page 18
"Dependencies within an IDL File" on page 129

RELATED TASKS
"Work with Data Objects - Overview" on page 296

Edit a Data Object Implementation

To edit a data object implementation, follow these steps:

1. Select the data object implementation in either the User-Defined Business Objects folder or the User-Defined Data Objects folder. From its pop-up menu, select **Properties.** The Data Object Implementation wizard opens to the Name and Platform Page. Change the names of the data object implementation's file, interface, or module, if you want to. You can also specify a different combination of deployment platforms.
   Restriction: You cannot change the names of the objects if the implementation inherits from another.
   Note: To ensure that valid code is generated after a rename, use **Generate - All** instead of **Generate - Selected** from the pop-up menu of the object.

2. Click **Next** to turn to the Behavior Page.
3. You can modify the type of implementation. You can select to test your business object either in a distributed environment, or as a stand-alone. Select one of the different options in the Environment (page 31) section.
4. You can change the data object behavior and its implementation in the Form of Persistent Behavior and Implementation (page 32) section of the same page. All options in this section are available for selection only if you have selected **BOIM with any key** in step 3.
390 All Cache Service options are not available when the target platform is OS/390.
5. The choices available in the next section, Data Access Pattern (page 34) section are determined by your selection of the form of persistent behavior and implementation in step 4. The access pattern is either **Delegating** or **Local copy**.
6. Select the handle for storing references to objects in the Handle for Storing Pointers (page 35) section.
7. Click **Next**. The Implementation Inheritance Page opens. You can specify new parent classes for the implementation to inherit from, or delete existing ones.
8. Click **Next**. The Attributes Page opens. You can use this page to add more attributes for the data object. These attributes will be specific to this implementation.
9. Click **Next**. The Methods Page opens. Here, you can add implementation-specific methods for the data object.
10. Click **Next**. The Select Key and Copy Helper Page opens. Select a different key and copy helper, if necessary.

    Note: If you selected **BOIM with any key** in step 3, and there is at least one persistent object in the model that has the same type of persistence as that for the implementation, which you specify in step 4, the Associated Persistent Objects Page is added to the wizard, and you can follow steps 11 through 15; if not, continue with step 14.
11. Click **Next.** The Associated Persistent Objects Page opens. You can specify new persistent object instances that are to be stored as protected members of the data object implementation, and edit or delete existing ones.
12. Click **Next.** The Attributes Mapping Page opens. You can specify or change the mapping between the data object attributes and the persistent object attributes.
13. Click **Next.** The Methods Mapping Page opens. For each of the special framework methods associated with the implementation, you can specify the persistent object methods that it calls. You can change the mappings by adding new persistent object methods, deleting existing ones or changing the order of the methods in the tree. Their order determines the order in which they are called.

    Note: If you have associated any PA persistent objects with this implementation, this page has the User-Defined Methods folder as well. This folder contains methods you defined at the data object interface level. If you want to call your own methods at the end of a Procedural Adaptor session for some clean-up work, you can map each of these methods to the corresponding push-down method of the persistent object.
14. Click **Next.** The Summary of Framework Methods Page opens. This page shows you the framework methods that Object Builder implements for this data object implementation. You cannot edit this page.

15. Click **Finish**. The data object implementation appears in the Tasks and Objects pane, with the changes to its properties.

Note: If there is no persistent object associated with the data object, none of the special framework methods will have implementations. You can add code for methods you want to implement, or for those for which you do not want to use the implementation provided by Object Builder.

RELATED CONCEPTS

"Data Object" on page 18
"Persistent Object" on page 19
Application Adaptor (*Programming Guide*)
Data Object Customization for Cardinality Relationships (*Programming Guide*)
Object Relationships (*Programming Guide*)
Data Object Customization (Storage Options) (*Programming Guide*)
"Container" on page 345
"Home" on page 342
"State Data" on page 18
Using Handles (*Programming Guide*)
Naming Service (*Advanced Programming Guide*)
Cache Service (*Advanced Programming Guide*)
"Special Framework Methods" on page 24
"Framework Methods" on page 24
Using Sets of Objects (Using Reference Collections) (*Programming Guide*)

RELATED TASKS

"Work with Data Objects - Overview" on page 296
"Add a Persistent Object and Schema" on page 313
"Customize Referential Integrity" on page 108
"Create a Container Instance" on page 346
"Configure a Managed Object" on page 377
"Work with Methods " on page 267
"Add Code for User-Defined Methods" on page 267
"Add endResource() to a Sessional Business Object" on page 117

RELATED REFERENCES

"Data Object Implementation Inheritance" on page 36

Delete a Data Object Interface

To delete a data object interface, follow these steps:

1. If there is a data object implementation created for the data object, you must first delete the implementation before you can delete the data object interface.
2. Select the data object interface in either the User-Defined Business Objects folder or the User-Defined Data Objects folder.
3. From the pop-up menu of the data object interface, select **Delete**. The interface is deleted from both folders.

RELATED CONCEPTS

"Data Object" on page 18

RELATED TASKS

"Work with Data Objects - Overview" on page 296

Delete a Data Object Implementation

To delete a data object implementation, follow these steps:

1. Select the data object implementation in either the User-Defined Business Objects folder or the User-Defined Data Objects folder.
2. From the pop-up menu of the data object implementation, select **Delete**.

Note: If the data object implementation has a persistent object and schema associated with it, these objects are deleted as well from these folders. However, the persistent object and schema still exist in the DBA-Defined Schemas folder.

RELATED CONCEPTS
"Data Object" on page 18

RELATED TASKS
"Work with Data Objects - Overview" on page 296

Work with DB Persistent Objects

DB persistent objects are defined in the DBA-Defined Schemas folder. A DB schema is created when you import an SQL DDL file into Object Builder. You can create multiple DB persistent objects for every DB schema. You can also create multiple data objects from the DB persistent object. Further, you can also associate a DB persistent object with a data object implementation, or create a DB persistent object and its associated schema from the implementation.

The following tasks deal with DB persistent objects:

- "Add a Persistent Object and Schema"
- "Add a Persistent Object from a DB Schema" on page 316
- "Edit a DB Persistent Object" on page 317
- "Add a Data Object from a DB Persistent Object" on page 304
- "Map a Data Object to a DB Persistent Object" on page 251
- "Delete a DB Persistent Object" on page 317

RELATED CONCEPTS
"Persistent Object" on page 19
"Schema" on page 20
"DDL" on page 114

Add a Persistent Object and Schema

Once you have added a data object implementation to the data object interface, you can either create schemas and persistent objects for the implementation, or map existing persistent objects and schemas to the implementation.

To create a schema and a persistent object, follow these steps:

1. From the User-Defined Data Objects folder, select the data object implementation for which you want to create the persistent object.

2. From the data object implementation's pop-up menu, select **Add Persistent Object and Schema**. The Add Persistent Object and Schema wizard opens to the Names Page.
   a. Type a name for the group.

      Note: Group names can contain only alphanumeric characters, the blank space, and the underscore, and they are case sensitive.
   b. Type a name for the database file.

      Note the following points about database names:
      - They must not exceed 8 characters.
      - They can contain any of the following characters: the letters a-z and A-Z, 0-9, #, @, $.
      - The first character of the name must be an alphabetic character, or one of #, @, or $. They must not contain must not contain characters from European or Asian character sets (for example, umlauts are not allowed).
      - They are not case sensitive.
   c. Type the table name, or accept the default.
   d. Type the user name.
   e. Type a name for the schema file, or accept the default.

      Follow these rules when you name a DB schema:
      - The name must not exceed 18 characters for DB2; 30 characters for Oracle.
      - All alphanumeric characters from your database character set and the characters _, $, #, @ are allowed. Characters include those from DBCS or European sets (including umlauts).
      - There's no case sensitivity for names containing these characters, unless they are surrounded by double quotation marks.
      - Non-alphanumeric names must be enclosed in double-quotes, and their case is maintained internally.
   f. If you had selected **Embedded SQL** as the type of persistence, you must type a name in the **Package File** field, or accept the default.

      Note: The name of the package file must not exceed 8 characters. It must be unique for each of the persistent objects that you create, if they are to operate under the same server at run time.

      You can either type the names for the persistent object class and instance, or accept the default names. The persistent object class name must not exceed 8 characters.

      390 If OS/390 is one of the deployment platforms for the data object implementation, the persistent object class name must not exceed 8 characters. Object Builder validates the length of the persistent object class when you create a persistent object from a data object implementation, but if you change the deployment platform after you have created the persistent object, be sure that you follow the rule. If not, Object Builder will truncate the name to the 8.3 format. This may result in two persistent object file names becoming identical after truncation, since Object Builder assumes the object's file name to be the same as the persistent object class name.

      Note: A schema must have a database key specified.
3. Click **Next**. The Attributes Mapping Page opens. Here, you can map attributes of the data object to those of the persistent object. You can also change the names of the attributes of the persistent object and the corresponding columns

of the schema, and their data types, and specify the persistent object keys if necessary, and the database keys for the table.
Note: The persistent object attribute name must not exceed 26 characters in length.
4. Click **Next**. The Columns and Attributes Page opens. Use this page to view the mapping between the schema and the persistent object.
5. Click **Next**. The Comments Page opens. Use it to save any comments about the schema, any of the schema columns, or the persistent object.

The persistent object is automatically associated with the data object implementation: the persistent object instance is added to the folder on the Associated Persistent Objects Page. Object Builder provides the default mapping of both attributes and methods of the data object to the persistent object. You can change the default mappings.

The persistent object appears as a child of the data object implementation, and the schema appears as a child of the persistent object in both the User-Defined Business Objects folder and the User-Defined Data Objects folder. In the DBA-Defined Schemas folder, the schema exists (with its persistent object child) outside in the schema group you named.

Restriction: Even if there is a key defined for a business object and it is designated as a foreign key, when you create a persistent object and schema for a business object referenced by the other object, it will not automatically create a foreign key in the schema.

Note: In some RDBMS configurations, the .sql files that Object Builder generates from the schemas must be processed by a database administrator using a design tool such as Logic Works' ERWin version 3.5 or 3.0, before they can be used to create tables in the database catalog. In others, you may be able to bypass the design tool, and instead use command line or other procedures to populate the database catalog.

Example:
- In the DB2 NT 5.0 single-user environment, you can use the following sequence of commands from the DB2 command window:
  db2 connect to *"name of working database"*
  db2 -t -f *"SQL filename with the path"*
- In Oracle 8.0.4.0 script center, you can import the .sql files.

ERWin 3.0 does not support the following database systems that ERWin 3.5 supports:
- DB2 / 390 5
- DB2 / CS 2
- DB2 / UDB 5
- Oracle 8.x

If you are using ERWin 3.0 or 3.5 to generate SQL files to be imported into Object Builder, you cannot use the default options provided by ERWin for the Oracle DBMS. In ERWin, when you select **Tasks - Forward Engineer/Schema Generation**, you must change the Referential Integrity Options for the Primary Key and Foreign Key to use the `CREATE` statements instead of the `ALTER` statements.

RELATED CONCEPTS

"Persistent Object" on page 19
"Schema" on page 20

RELATED TASKS

"Work with DB Persistent Objects" on page 313
"Work with DB Schemas" on page 320
"Add a Data Object Implementation" on page 299
"Map a Data Object to a DB Persistent Object" on page 251
"Edit a DB Schema" on page 329
"Edit a Generated SQL File" on page 331
"Create a Component for New DB Data - Scenario" on page 102

RELATED REFERENCES

"DB2 Data Type Mappings" on page 110
"Oracle Data Type Mappings" on page 113

Add a Persistent Object from a DB Schema

To add a persistent object to an existing schema, follow these steps:

1. From the DBA-Defined Schemas folder, select the schema to which you want to add a persistent object.
2. From the schema's pop-up menu, select **Add Persistent Object**. The Add Persistent Object wizard opens.
3. Type a name for the persistent object in the **Name** field.
4. Change the setting of the **Table is updatable** check box, if you want. Your selection determines if the schema is read-only or if it can be updated. For schemas, this check box is selected by default; for views, it is not selected.
5. Select **DB2 Cache Service** or **Embedded SQL** to specify the type of persistence for the object.
6. If you select **Embedded SQL**, you must type a name for the package file, or accept the default.
7. Indicate whether a particular schema column is to be mapped to the corresponding persistent object attribute: click the **Mapped** field and select the check box for the column.

   Restriction: You must map all schema columns to their corresponding persistent object attributes; otherwise you may get exceptions thrown at run time if you use the Query Service.
8. Modify the name of the persistent object attribute, if required.
9. Specify whether a schema column's corresponding persistent object attribute is to be the key for the persistent object by selecting the **PO Key** check box for the column.
10. Click **Next**. The Comments Page opens. Use it to add any comments about the persistent object.

Restrictions:

- If your schema uses Oracle Cache service, you can create a persistent object from it only if the schema columns are of the `NUMBER` or `VARCHAR2` data types, or any of the IBM DB2 data types.

- Even if there is a key defined for a business object and it is designated as a foreign key, when you create a persistent object and schema for a business object referenced by the other object, it will not automatically create a foreign key in the schema.

RELATED CONCEPTS
"Persistent Object" on page 19
"Schema" on page 20
Cache Service (*Advanced Programming Guide*)
Query Service (*Advanced Programming Guide*)

RELATED TASKS
"Create a Component for Existing DB Data" on page 104
"Create a DB Schema by Importing an SQL File" on page 321
"Add a Persistent Object and Schema" on page 313
"Work with DB Persistent Objects" on page 313
"Add a Data Object from a DB Persistent Object" on page 304

RELATED REFERENCES
"DB2 Data Type Mappings" on page 110
"Oracle Data Type Mappings" on page 113

Edit a DB Persistent Object

To modify a persistent object, follow these steps:

1. Select the persistent object in the Tasks and Objects pane.
2. From its pop-up menu, select **Properties**.
3. The Persistent Object wizard opens to the Persistent Object Page. You can rename the persistent object and provide a new name for the package file. In the panel, you can rename the persistent object attribute: double-click in the field, and type in the new name. You can also specify different persistent object attributes as the keys for the object. Click in the **PO Key** field, and select or clear the check box.

   Restriction: A persistent object attribute name cannot exceed 26 characters in length.
4. Click **Next** if you want to change any comments about the persistent object.

Note: To ensure that valid code is generated after a rename, use **Generate - All** instead of **Generate - Selected** from the pop-up menu of the object.

RELATED CONCEPTS
"Persistent Object" on page 19
"Schema" on page 20

RELATED TASKS
"Work with DB Persistent Objects" on page 313
"Edit a DB Schema" on page 329

Delete a DB Persistent Object

To delete a DB persistent object, follow these steps:

1. Select the persistent object from either the User-Defined Business Objects folder, the User-Defined Data Objects folder, or the DBA-Defined Schemas folder.

2. From the pop-up menu of the persistent object, select **Delete**.

If the persistent object is not connected to a data object implementation, it is deleted from the DBA-Defined Schemas folder.

If the persistent object is associated with a data object implementation, the following deletions take place:

- the persistent object and its underlying schema are deleted from the User-Defined Business Objects folder and the User-Defined Data Objects folder.
- the persistent object is deleted from the DBA-Defined Schemas folder

RELATED CONCEPTS

“Persistent Object” on page 19

RELATED TASKS

“Work with DB Persistent Objects” on page 313

Work with DB Schema Groups

All DB schemas in Object Builder exist in schema groups for organizational purposes.

The following tasks deal with schema groups:

- “Create a DB Schema Group”
- “Edit a DB Schema Group” on page 319
- “Re-import an SQL File” on page 330
- “Delete a DB Schema Group” on page 320

RELATED CONCEPTS

“DDL” on page 114
“Schema” on page 20
“Schema Group” on page 20

Create a DB Schema Group

You can create a schema group in Object Builder in the following ways:

- by specifying the name of the schema group when you add a persistent object and schema (page 313) for a data object implementation
- by specifying the name of the schema group when you create schemas by importing an SQL file (page 321) into Object Builder
- by selecting **Add Schema Group** from the pop-up menu of the DBA-Defined Schemas folder

To create an empty schema group, follow these steps:

1. From the pop-up menu of the DBA-Defined Schemas folder, select **Add Schema Group**. The Schema Group wizard appears open to the Schema Group Name Page.
2. Type a name for the schema group in the **Schema Group Name** field.
3. Type a name of the database to be associated with this schema group in the **Database Name** field.

4. Select the type of the relational database backend for which you are creating the schema group. You can select either **DB2** or **Oracle**.

Keep the following points in mind:
- Whenever you create a schema group in Object Builder, along with the schema group name, you must specify the name of the database to be associated with the schema group.
- You can import SQL files into any of the existing schema groups.

RELATED CONCEPTS
"DDL" on page 114
"Schema" on page 20
"Schema Group" on page 20

RELATED TASKS
"Work with DB Schema Groups" on page 318

Edit a DB Schema Group

You can edit a schema group by editing either the properties of the group or its contents.

To edit the properties of a schema group, follow these steps:
1. From the pop-up menu of the schema group, select **Properties**.
2. The Schema Group wizard opens to the Schema Group Page.
3. You can rename the group and the database to be associated with the group. The name of the group must be unique.
4. In the **File to Open in Editor** section, indicate the file you want to view when you use the **Open in Editor** option from the pop-up menu of either the schema group, or any schema within the group. You can change it from the source file (the original SQL DDL file that was imported), which is the default, to the generated file, which is the SQL file that Object Builder generates (when you select **Generate** from the pop-up menu of the schema group).
   Note the following points:
   - **Generated** is the only option available for schema groups that are created top-down.
   - For a schema group, using **Generate - Selected** is the only way to emit a .sql file for the group.
   - The generated file exists in the working directory and has the same name as the name of the schema group. If you want to preserve these generated files, you must rename the existing generated file before you select the **Generate** option from the schema group's pop-up menu. This is particularly important if you want to re-import the SQL source file and this file exists in the working directory, and has the same name as the group. You can re-import an SQL file using the Statements to Import Page of the Import SQL DDL File wizard.

New and existing schemas within the group will be associated with the new database name.

The following tasks deal with editing the contents of a schema group:
- "Create a DB Schema by Importing an SQL File" on page 321
- "Re-import an SQL File" on page 330

- “Create a View with the SQL View Editor” on page 324
- “Edit a View with the SQL View Editor” on page 325

RELATED CONCEPTS
“DDL” on page 114
“Schema” on page 20
“Schema Group” on page 20

RELATED TASKS
“Create a Component for Existing DB Data” on page 104
“Work with DB Schema Groups” on page 318
“Work with DB Schemas”

Delete a DB Schema Group

To delete a schema group, follow these steps:

1. Delete any schemas belonging to another group that reference schemas within this group.
2. From the DBA-Defined Schemas folder, select the schema group.
3. From the pop-up menu of the schema group, select **Delete**. You get a warning message informing you that if you delete the group, any associated persistent objects that are contained within that group will be deleted as well.
4. To continue with the deletion process, click **Yes**. The entire schema group is removed from the DBA-Defined Schemas folder, and any schemas and their persistent objects that were members of the group are deleted from the User-Defined Business Objects folder and the User-Defined Data Objects folder as well.

RELATED CONCEPTS
“Schema” on page 20
“Schema Group” on page 20“Persistent Object” on page 19

RELATED TASKS
“Work with DB Schema Groups” on page 318
“Work with DB Schemas”

Work with DB Schemas

A schema is a structural and behavioral composition that defines data storage and data access mechanisms within the database. A schema is always related to storage. A persistent object is usually associated with the schema, and provides persistence of the data beyond the execution time of the application that instantiated the object.

A schema can be created based on a data object, or it can be created from the database definitions stored in a DDL file.

The following tasks deal with DB schemas:

- “Add a Persistent Object and Schema” on page 313
- “Create a DB Schema by Importing an SQL File” on page 321
- “Create a View with the SQL View Editor” on page 324

- “Use Complex Relationships in SQL Clauses” on page 326
- “Edit a View with the SQL View Editor” on page 325
- “Edit a View” on page 328
- “Edit a DB Schema” on page 329
- “Edit a Generated SQL File” on page 331
- “Delete a DB Schema” on page 333

RELATED CONCEPTS
“Schema” on page 20
“Persistent Object” on page 19

RELATED TASKS a
“Add a Persistent Object and Schema” on page 313
“Add a Persistent Object from a DB Schema” on page 316“Edit a DB Schema Group” on page 319
“Edit a DB Schema” on page 329

Create a DB Schema by Importing an SQL File

When you import an SQL DDL file, you import the description of the database schema as it is defined in a relational database. Schemas imported from an SQL file exist in Object Builder within a schema group.

The DDL file is an ASCII file containing SQL statements that describe the schema. A schema can have tables. Each row must be uniquely identifiable: each table must have a unique primary key. A row of a table (or of a view) is represented as a single persistent object instance. All rows in the table can be represented by a single persistent object class but each row is a separate instance of that class.

Restrictions:
- This release supports SQL DDL files for Oracle 8.0.4.0 and DB2 MVS 4.1 databases, and tolerates UDB V5 syntax. This means that you may not be able to import 5.0 DDL, but if you have a DDL file containing 4.1 DDL with a few 5.0-specific lines, you will be able to import the 4.1 DDL lines from that file.
- SQL files larger than 2M are not recommended.

To import an existing DDL file, follow these steps:
1. From the pop-up menu of DBA-Defined Schemas folder or the schema group, select **Import SQL**. The Import SQL DDL File wizard opens to the SQL File Selection Page.
2. Type the name of the DDL file (.sql file) or click **Find** to specify the path and select from a list of files.
   Note: It is recommended that the SQL source file be placed in a directory other than the Working directory. This is to avoid having the file overwritten when you select either **Generate - Selected** or **Generate - All** from the schema group’s pop-up menu.
3. Type a name for the database to be associated with the schema being imported or accept the default in the **Database Name** field.
4. Type a name for the group to contain the schemas being imported or accept the default in the **Group Name** field. The schemas appear in the DBA-Defined Schemas folder beneath the the group.

5. Click **Next**. The Statements to Import Page opens with all the SQL statements in the imported file selected for parsing. To deselect all the statements, click **Deselect All**. You can select the specific ones you want parsed. Multiple selections are possible. To select all the statements, click **Select All**.

   Note: At least one `CREATE TABLE` statement must be selected for the import process to succeed.

   Restriction: Currently, the only SQL statements supported are `DROP`, `CREATE TABLE`, `CREATE VIEW`, `ALTER TABLE`, and `COMMENT ON`. None of these statements must contain expressions or column functions. The `CREATE VIEW` statement must contain only a simple query (`SELECT` statement). Currently there is no support for unnamed columns, expressions, functions, or sub-selects in `CREATE VIEW`.
6. Click **Finish.** Schemas thus imported into Object Builder, appear in the DBA-Defined Schemas folder, within a group that gets its name from the .sql file.

Note: You can re-import a schema group to include different tables, or to modify or delete existing tables. To do so, follow these steps:
1. Select the schema group in the DBA-Defined Schemas folder. From the pop-up menu of the schema group, select Import SQL. The Import SQL DDL File wizard opens to the Statements to Import Page.
2. All the SQL statements in the imported file are selected for parsing. To deselect all the statements, click **Deselect All**. You can select the specific ones you want parsed. Multiple selections are possible. To select all the statements, click **Select All**. The schema group is overwritten.

Importing an SQL file is the first step in the bottom-up scenario, when you can reuse existing data. The scenario continues with the following steps:
1. Using the schema information, create a persistent object.
2. Create a data object that corresponds to the persistent object.
3. Create a business object and select the data object (created in step 2) to be used by the business object. The data object uses the mapping information you provide when you select it, to manage the business object's persistent state.

Restrictions:
- Double-Byte Character Set (DBCS) is not supported for the English version of Component Broker.
- A table that is associated with a schema of one schema group cannot reference a foreign key defined in a table within another schema group.
- Most views are read-only but some can be updated. The Embedded SQL preprocessor (idatapre) will fail on any .sqx file generated from an embedded static persistent object, which you create for a read-only view in the database.

  If you detect that a view is read-only (at DLL build time), for each of the special framework methods insert(), update() and del() in the Methods pane, follow these steps:
  1. From the pop-up menu of the method, select **Properties**. The Method Implementation wizard opens to the Implementation Page.
  2. Select the option **Use the implementation defined in the editor pane**, and make sure the method body is empty.

For example,

```
void insert()
{
}
```

- Object Builder lets you import schemas for which no primary keys have been defined. However, these schemas can result in exceptions thrown at run time if you use Query Services. To avoid this happening, you can follow any one of these steps:
  1. After you import the SQL file, select **Properties** from the pop-up menu of the schema, and select any of the schema columns as the database key. (Select the **DB Key** check box.)
  2. Before you import the SQL file, edit the source file and add a `PRIMARY KEY` constraint for at least one of the tables.
  3. Edit the primaryKey entry in the table MappedType.*tablename*_Table, in the System Management Data Definition Language (SM DDL) file generated from the Application Family. This file's primary name will have the term *Specific* before the family name you specified and its extension will be .ddl. For tables that do not have at least one primary key defined, the primaryKey entry will be `void ("")`. Edit it to include the names of all the table columns that comprise the primary key you want to define. For example, if you want to indicate that the columns `COMP`, `PLAT`, `SEQ` and `ATTEMPT` comprise the primary key, this would be the entry: `primaryKey = "\"COMP\",\"PLAT\",\"SEQ\",\"ATTEMPT\"";`

RELATED CONCEPTS

"DDL" on page 114
"Schema" on page 20
"Persistent Object" on page 19
"Special Framework Methods" on page 24

RELATED TASKS

"Create a Component for Existing DB Data" on page 104
"Work with DB Schemas" on page 320
"Work with DB Persistent Objects" on page 313
"Edit a DB Schema Group" on page 319
"Edit Special Framework Methods" on page 271
"Generate Code" on page 363

SQL View Editor

The SQL View Editor is a tool that enables you to create and modify views from within Object Builder. You can invoke it from the pop-up menu of a schema group in the DBA-Defined Schemas folder when you want to create a view, or from the pop-up menu of a view when you want to edit the view.

The View Editor has the following notebook pages:

- View Properties
- View Work Area
- View Summary

View Properties
This page enables you to provide identification for the view you are adding. You can review details about the view, and edit any comments you added when you created the view, when you use the Editor to edit a view.
View Work Area

This page is where most of your interaction with the View Editor takes place. This area is further subdivided into the following panes:

- **Schemas**: This pane lists the schemas that belong to the schema group. In the Graphic view, it also shows foreign key relationships among the schemas.
- **Columns**: This pane lists the columns of the schema that you select in the Schemas pane. You can select columns these columns for inclusion in SQL clauses.
- **Clauses**: This pane enables you to construct or modify clauses to define the view, with the columns you select. It is further divided into the following panes for defining the clauses:
  - Selected Columns
  - Where
  - Group By
  - Having

View Summary
Use this section to view the SQL code for the different clauses you defined using the **Clauses** pane in the **View Work Area**. You can view the code either as a single SQL statement, or clause by clause.

RELATED CONCEPTS
“Schema” on page 20
“Schema Group” on page 20

RELATED TASKS
“Create a View with the SQL View Editor”
“Edit a View with the SQL View Editor” on page 325
“Use Complex Relationships in SQL Clauses” on page 326

RELATED REFERENCES
Query Service

Create a View with the SQL View Editor

To create an SQL view in Object Builder, follow these steps:

1. Select the schema group from which you want to create the view, in the DBA-Defined Schemas folder.
   Note: The schema group must contain the schemas from which you want to create the view.
2. From its pop-up menu, select **Add SQL View**. The SQL View Editor opens.
3. Click the **View Properties** tab of the editor. On the View Properties Page, type a name for the view, and optionally specify a userid and add any comments.
4. Click the **Selected Columns** tab of the Clauses pane.
5. Select a schema to be used for the view in the Schemas pane. The schema columns and their details appear in the Columns pane.
6. Select the columns you require for the view from the Columns pane. As you select each column, the column name and the table it belongs to appear in the Selected Columns Page.
7. Repeat steps 4, 5 and 6 for each of the schemas whose columns you want to include in the view.

8. Click the **Where** tab of the Clauses pane. On the Where Page, you can specify conditions that have to be met by the various schema columns, for inclusion in the view.
9. Click the **Group By** tab of the Clauses pane. On the Group By Page, you can specify the columns, based on whose values the order of occurrence of the view's rows is determined: Click the **Select All** button. All columns that you specified as selected columns on the Selected Columns Page are set as the grouping columns. The first grouping column determines the initial grouping. Subsequent grouping columns are used to resolve the order of the rows when there is a tie within a group of rows formed by its predecessor.

   Note: You cannot select a subset of the selected columns to group the view by. You have to use the **Select All** button.
10. Click the **Having** tab of the Clauses pane. On the Having Page, you can apply a qualifying condition to the groups created with the `GROUP BY` clause. Only those groups that meet the `HAVING` condition are included in the view.
    Note: From the Where Page and the Having Page, you can bring up the Organize Logical Combination dialog box, where you can manually arrange the predicates to be combined for the view.
11. Click the **View Summary** tab to open the View Summary Page of the Clauses pane. Use this page to view the SQL clauses you defined for the view.
12. When you have finished reviewing your definition, click **Finish**.

The view appears in the schema group in the DBA-Defined Schemas folder.

RELATED CONCEPTS
"Schema" on page 20
"Schema Group" on page 20

RELATED TASKS
"Create a DB Schema Group" on page 318
"Edit a View with the SQL View Editor"

Edit a View with the SQL View Editor

To edit an SQL view in Object Builder, follow these steps:
1. Select the view in the DBA-Defined Schemas folder.
2. From its pop-up menu, select **SQL View Editor**. The SQL View Editor opens.
3. Click the **View Properties** tab. You cannot change the name and userid of the view. However, you can modify the comments on the View Properties Page.
4. Click **Next**, or click the **View Work Area** tab. Columns of the view appear in the Selected Columns page of the Clauses pane. As you select the different schemas in the Schemas pane, their schema columns and corresponding details appear in the Columns pane. To add a new column to the view, click on it in the Columns pane. To remove a column from the view, right-click on any field in column's row in the Clauses pane, and select **Remove** from the pop-up menu. To rename a view column, right-click on the name in the **View Column** of the Selected Columns page, and select **Change Value** from the pop-up menu. The Change Column Name dialog box appears, and you can type a new name in the field.
   Note: On all the other pages of this pane, you can view the previous settings and make changes if you want. To remove an entry in a field on either the Where Page or the Having Page, right click in the field and select **Remove**. To

specify a new entry for the Table/Column field in the predicate section, select the column from the Columns pane; to specify a new entry in the search conditions' Table/Column section, first click in the field, and then, select a column from the Columns pane.

5. Click the **Where** tab of the Clauses pane. On the Where Page, you can view the conditions that were previously set for the various schema columns, for inclusion in the view.
6. Click the **Group By** tab of the Clauses pane. On the Group By Page, if no `GROUP BY` clause had been specified for the view, you can specify the columns, based on whose values the order of occurrence of the view's rows is determined: Click the **Select All** button. All columns that you specified as selected columns on the Selected Columns Page are set as the grouping columns. If the view's previous definition included a `GROUP BY` clause, the only modification you can make on this page is to deselect all grouping columns: click on **Clear All**. The view will not have an orderly grouping for its rows.
7. Click the **Having** tab of the Clauses pane. On the Having Page, you can apply a qualifying condition to the groups created with the `GROUP BY` clause. Only those groups that meet the `HAVING` condition are included in the view.

   Note: From the Where Page and the Having Page, you can bring up the Organize Logical Combination dialog box, where you can manually arrange the predicates to be combined for the view.
8. Click the **View Summary** tab to open the View Summary Page of the Clauses pane. Use this page to view the SQL clauses you redefined for the view.
9. When you have finished reviewing your definition, click **Finish.**

The view will be redefined according to your modifications.

RELATED CONCEPTS

"Schema" on page 20
"Schema Group" on page 20

RELATED TASKS

"Create a DB Schema Group" on page 318
"Create a View with the SQL View Editor" on page 324

Use Complex Relationships in SQL Clauses

In the SQL View Editor, you can specify conditions or relationships that must exist among rows of the various schemas, for them to be included in the view. You can do this on the Where Page and the Having Page of the Clauses pane. These conditions are also called predicates, and they can be combined in the following ways:

- All predicates must be satisfied ("AND")
- At least one predicate must be satisfied ("OR")
- A more complex arrangement of "AND", "OR" and "NOT" conditions must be satisfied

Each of these conditions can be expressed by selecting the corresponding button (**And**, **Or**, or **Use Complex Relationships**) at the bottom of the page. Whichever condition you specify, you can see a graphical representation of the logic behind the condition by clicking the **Edit Conditional Relationships** button.

To add a complex condition to the SQL clause when creating or editing a view, follow these steps:

1. Click the **Edit Conditional Relationships** button.
   The Organize Logical Combination dialog opens. Each predicate is represented by either an entry in the list box on the left, or by a rectangle in the graph on the right. Predicates in the list box do not belong to any logical combination; those in the graph do.
2. You can manually change the logical combination of the predicates in the graph view by doing one of the following tasks:
   - Adding a predicate to the graph
   - Removing a predicate from the graph
   - Negating a predicate
   - Negating a combination of predicates

To add a predicate to the graph, follow these steps:

1. Select the predicate from the list in the left-hand frame. The statement of the predicate appears below the list box.
   Note: If there are any other elements in the graph, you must select at least one with which the new predicate will be combined. For example, if you choose predicate 'A' from the list, and then choose predicate 'B' from the graph, you are then allowed to combine 'A' with 'B', with either an "AND" condition or an "OR" condition. You may choose as many predicates from the graph as you wish, and the editor will add the new predicate according to the combination you requested.
2. Once you have selected the predicates, you can select one of two buttons below the list box: **Add as And>>** and **Add as Or>>**. Your selection determines how the new predicate will be combined with the selected ones.

To remove a predicate from the graph, follow these steps:

1. Select the predicate with the mouse.
   Note: As you move the mouse over a predicate in the graph, the text of the predicate appears.
2. Click the **<<Remove** button.

Notice the following indications on the graph:

- To the left of each predicate is a white NOT indicator. You can use it to negate a condition. As you move the mouse over this symbol, a red outline appears around the predicate. This implies that the NOT operator is associated with that predicate. Click the NOT operator to negate the predicate indicated by the rectangle. Once a predicate is negated, its associated NOT operator appears red.
- To the left of each combination of predicates, there is also a yellow tilde. When you move the mouse over this tilde, a red outline appears around the entire logical combination.

To negate a predicate (specify "NOT" conditions), follow these steps:

1. Move your mouse over the predicate and click the NOT operator closest to it. The yellow tilde turns red, indicating that the predicate is negated.

To negate a logical combination, follow these steps:

1. Move your mouse over the combination of predicates and click the outermost NOT operator. The yellow tilde turns red, indicating that the combination as a

whole is negated. For example, if the combination shows 'A' AND 'B', then by selecting the tilde for this combination, it becomes NOT ('A' AND 'B').

Note: When you close the dialog, one of the radio buttons: **And**, **Or**, or **Use Complex Relationships** will be selected according to the state of the arrangement. If **Use Complex Relationships** is selected, it implies that one of the following conditions exists in the arrangement:

- There are predicates left in the list box that are not yet placed in combination
- The picture in the graph is a combination of AND, OR and NOT conditions, and cannot be reflected using either a simple AND or a simple OR combination.

RELATED TASKS

"Create a View with the SQL View Editor" on page 324
"Edit a View with the SQL View Editor" on page 325

Edit a View

You can modify a view that exists in Object Builder using the Schema Page of the Schema wizard the same way you modify a schema in Object Builder.

To change the identification for a view, follow these steps:

1. In the DBA-Defined Schemas folder, select the view. From the pop-up menu of the view, select **Properties**. The Schema Page opens.
2. You can modify the user ID, table name and schema filename.
3. Click **Finish**.
   A new, stand-alone view is created only when the old view is referenced under another view. If the view is not under another view, it is renamed. So, even a view that has associated persistent objects will be renamed, not copied. A copy of the view, with the new name and properties is created only for views that are under other views.

Note: To ensure that valid code is generated after a rename, use **Generate - All** instead of **Generate - Selected** from the pop-up menu of the object.

You can modify the structure of a view by editing the Schema Page, and the Clause Summary Page.

To change the structure of a view using the Schema Page, follow these steps:

1. From the pop-up menu of the view object, select **Properties**. The Schema Page opens.
2. You can edit the **ForBitData**, **DB Key** and **Not Null** fields. The existing view is overwritten with the changes.

To change the structure of a view using the Clause Summary Page, follow these steps:

1. From the pop-up menu of the view object, select **Properties**. The Schema wizard opens to the Schema Page. You can edit the **ForBitData**,**DB Key** and **Not Null** fields.
2. Click the arrow to the left of the page name, and select Clause Summary Page from the list. By default, the text panel on this page is read-only, and you can select the radio buttons associated with the different SQL clauses to see their definition.
   Attention: It is not recommended that you edit the Object Builder-generated

SQL clauses for the view definition. The changes you make affect the DDL that Object Builder generates and you can access the original code only by redefining the view, or importing once again into Object Builder the SQL file that contains the definition of the view. However, if you must edit some of the view's definition clauses, follow step 3; otherwise proceed with step 4.

3. Select the **Provide your own SQL for the clause** check box. Once you select this box, the text panel containing the clauses becomes editable, and you can overwrite the definition provided by Object Builder. You can overwrite one clause at a time.
4. Turn to the Comments Page. Here, you can type comments for the view, as well as for the schema columns that are used in the definition of the view.
5. Click **Finish.**

The existing view is overwritten with the changes.

Note: To delete a view that has an associated persistent object, you must first delete the persistent object. To delete a view that is used to create other views, you must first delete the view that is created from it.

RELATED CONCEPTS

"Schema" on page 20
"Persistent Object" on page 19

RELATED TASKS

"Add a Persistent Object and Schema" on page 313
"Add a Persistent Object from a DB Schema" on page 316
"Work with DB Schemas" on page 320
"Edit a DB Schema Group" on page 319

Edit a DB Schema

You can modify any DB schema that exists in Object Builder, whether it was created from a data object or imported from an SQL DDL file.

To change the identification for a schema, follow these steps:

1. From the pop-up menu of the schema object, select **Properties**. The Schema Page opens.
2. You can modify the user ID, table name and schema filename.
3. Click **Next**, and modify the comments about the schema (Comments Page), if you choose to.
   Note: In the case of a view, besides all the above modifications, you can also modify the clauses that define the view (Clause Summary Page).
   A new, stand-alone schema is created only when the old schema is referenced under another view. If the schema is not under another view, it is renamed. So, even a schema that has foreign key relationships or associated persistent objects will be renamed, not copied. A copy of the schema, with a new name and new properties is created only for schemas that are under other views.

Note: To ensure that valid code is generated after a rename, use **Generate - All** instead of **Generate - Selected** from the pop-up menu of the object.

You can modify the structure of a schema by either editing the Schema Page or by re-importing the SQL file.

To change the structure of a schema using the Schema Page, follow these steps:

1. From the pop-up menu of the schema object, select **Properties**. The Schema Page opens.
2. You can edit the **Schema Column**, **ForBitData**, **DB Key** and **Not Null** fields. The existing schema is overwritten with the changes. To rename a schema column, double-click in the field, and type in the new name. You can also allocate different columns as DB keys, and change the Not Null specification for the different columns as well, and modify the ForBitData for those schema columns for which this information can be specified. Click in the field, and select or clear the check box.

   Note: All columns that you indicate as DB keys have to be not null. This specification cannot be changed.

To change the structure of a schema by re-importing the SQL file, follow these steps:

1. From the pop-up menu of the schema group, select **Import SQL**. The Statements to Import Page opens.
2. Select those `ALTER TABLE` statements that refer to the schema that you want to modify in this group.

RELATED CONCEPTS
"Schema" on page 20
"Persistent Object" on page 19
RELATED TASKS
"Add a Persistent Object and Schema" on page 313
"Add a Persistent Object from a DB Schema" on page 316"Work with DB Schemas" on page 320
"Edit a DB Schema Group" on page 319

Re-import an SQL File

To re-import an SQL file, follow these steps:

1. From the pop-up menu of DBA-Defined Schemas folder or the schema group, select **Import SQL**. The Import SQL DDL File wizard opens to the SQL File Selection Page.
2. The name of the DDL file (.sql file) previously imported appears in the **Last File Name Imported** field.

   Note: It is recommended that the SQL source file be placed in a directory other than the subdirectories of the Working directory, which are named according to the platform for which you are generating code. This is to avoid having the file overwritten when you select either **Generate - Selected** or **Generate - All** from the schema group's pop-up menu.
3. The name of the database previously associated with the schema being imported is shown in the **Database Name** field. This entry cannot be changed.
4. The name of the group shown in the **Group Name** field too cannot be changed. The schemas appear in the DBA-Defined Schemas folder beneath the group.
5. Click **Next**. The Statements to Import Page opens with all the SQL statements in the imported file selected for parsing. To deselect all the statements, click **Deselect All**. You can select the specific ones you want parsed. Multiple selections are possible. To select all the statements, click **Select All**.

Restriction: Currently, the only SQL statements supported are `DROP`, `CREATE TABLE`, `CREATE VIEW`, `ALTER TABLE`, and `COMMENT ON`. None of these statements must contain expressions or column functions. The `CREATE VIEW` statement must contain only a simple query (`SELECT` statement). Currently there is no support for unnamed columns, expressions, functions, or sub-selects in `CREATE VIEW`.

6. Click **Finish.**

If you use **Import SQL** from the folder, the schemas imported into Object Builder appear in the DBA-Defined Schemas folder, within a group whose default name is the name of the .sql file. You can change the name of the group and the default name of the database as well. If you use **Import SQL** from a schema group, you can neither change the name of the group nor that of the database.

If you select statements that act on existing tables, Object Builder warns you that the tables will be overwritten.

RELATED CONCEPTS

"DDL" on page 114
"Schema" on page 20
"Schema Group" on page 20
SQL View Editor

RELATED TASKS

"Create a DB Schema by Importing an SQL File" on page 321
"Work with DB Schemas" on page 320
"Add a Persistent Object from a DB Schema" on page 316
"Edit a DB Schema" on page 329
"Edit a Generated SQL File"

Edit a Generated SQL File

When you generate a schema that is created from a data object, using either **Generate - Selected** or **Generate - All** from the pop-up menu of the schema, or **Generate - All** from the pop-up menu of its containing schema group, the resulting .sql file cannot be used as such by DB2 to create tables.

Note the following points:

- In some RDBMS configurations, the .sql files that Object Builder generates from the schemas must be processed by a database administrator using a design tool such as Logic Works' ERWin version 3.5 or 3.0, before they can be used to create tables in the database catalog. In others, you may be able to bypass the design tool, and instead use command line or other procedures to populate the database catalog.

  Example:

  In the DB2 NT 5.0 single-user environment, you can use the following sequence of commands from the DB2 command window:

  db2 connect to *"name of working database"*
  db2 -t -f *"SQL filename with the path"*

  In Oracle 8.0.4.0 script center, you can import the .sql files.
- ERWin 3.0 does not support the following database systems that ERWin 3.5 supports:
  – DB2 / 390 5

– DB2 / CS 2
– DB2 / UDB 5
– Oracle 8.x

If you are using ERWin 3.0 or 3.5 to generate SQL files to be imported into Object Builder, you cannot use the default options provided by ERWin for the Oracle DBMS. In ERWin, when you select Tasks - Forward Engineer/Schema Generation, you must change the Referential Integrity Options for the Primary Key and Foreign Key to use the `CREATE` statements instead of the `ALTER` statements.

- If the design tool you use includes a startup command that takes a DDL file as an argument, you can place that command in the file named sqlparse.cmd (on NT) and sqlparse.sh (on AIX). The **Open in Editor** option from the pop-up menu of the schema, would then launch the tool.
- The SQL DDL files that you create using a database design tool can be imported into Object Builder.

To launch ERWin from Object Builder, follow these steps:

1. Open the file sqledit.bat (which is located in the bin subdirectory of Component Broker ToolKit's installation directory, which is usually \CBroker\bin in your current drive).
2. Comment out (add `rem` before) the command that launches the LPEX editor (`@start evfxlxpm %sqleditargs%`).
3. Delete `rem`, which is at the beginning of the command that launches ERWin (`@start mmopn32 %sqleditargs%`).

   The sqledit.bat file should have the following entries:
   ```
   rem @start evfxlxpm %sqleditargs%
   @start mmopn32 %sqleditargs%
   ```
4. From the DBA-Defined Schemas folder or the User-Defined Data Objects folder, select the schema.
5. From the pop-up menu of the schema, select **Open in Editor**. This launches ERWin .

AIX ERWin does not run on AIX, but if you still want to use it as the design tool, follow these steps:

1. Transfer the generated .sql file from AIX to NT.
2. Process the .sql file using ERWin against an NT DB2 client installation, which is backed by an AIX DB2 server.

With AIX, too you can specify the editor of your choice. This is done in the file sqledit.sh.

1. Comment out (add `#` before) the command line that launches the vi editor (`dtterm -e vi $* &`)
2. Include the editor of your choice instead of vi (for example, LPEX: `lpex $* &`)

ERWin runs only on Windows NT.

Note: LPEX is available only if SDE/6000 is installed.

The sqledit.sh file should have the following entries:

```
# dtterm -e vi $* &
```

```
lpex $* &
```

Hint: When you use ERWin to create table columns that will hold object references, you can specify the `VARCHAR FOR BIT DATA` type.

RELATED CONCEPTS
"Schema" on page 20
Object Relationships

RELATED TASKS
"Create a Component for Existing DB Data" on page 104
"Edit a DB Schema Group" on page 319
"Store an Object Reference" on page 135

Delete a DB Schema

To delete a DB schema, follow these steps:

1. Delete any persistent objects that are associated with this DB schema.

   Note: If you delete the persistent objects from the the User-Defined Data Objects folder, or the User-Defined Business Objects folder, the schemas are automatically deleted from these folders. You still have to delete it from the DBA-Defined Schemas folder. (Follow step 4.)
2. Delete any views that use this schema.
3. Delete any other schemas that reference this schema.
4. From the pop-up menu of the schema in the DBA-Defined Schemas folder, select **Delete**.

RELATED CONCEPTS
"Schema" on page 20
"Persistent Object" on page 19

RELATED TASKS
"Work with DB Schemas" on page 320
"Delete a DB Persistent Object" on page 317
Delete Component Objects

Work with PA Persistent Objects - Overview

PA persistent objects are defined in the User-Defined PA Schemas folder. A PA persistent object is created along with every PA schema that is created in Object Builder, as a result of importing a PA bean.

You can create additional PA persistent objects for every PA schema. You can also create multiple data objects from the PA persistent object. Further, you can also associate a PA persistent object with a data object implementation.

The following tasks deal with PA persistent objects:

- "Add a Persistent Object from a PA Schema" on page 334
- "Edit a PA Persistent Object" on page 336
- "Add a Data Object from a PA Persistent Object" on page 305

- Map a Data Object to a PA Persistent Object
- “Delete a PA Persistent Object” on page 336

RELATED CONCEPTS
“Persistent Object” on page 19
“Schema” on page 20“Procedural Adaptor Bean (PA Bean)” on page 117

RELATED TASKS
Work with Components

Add a Persistent Object from a PA Schema

To add a persistent object to an existing PA schema, follow these steps:

1. From the User-Defined PA Schemas folder, select the schema to which you want to add a persistent object.
2. From the schema’s pop-up menu, select **Add Persistent Object**. The Add Procedural Adaptor Persistent Object wizard opens to the Attributes Mapping Page.
3. Type a name for the persistent object in the **Name** field.
4. Indicate which of the attributes are to be keys for the persistent object.
5. Modify the names of the persistent object attributes, if required.

RELATED CONCEPTS
“Persistent Object” on page 19
“Schema” on page 20

RELATED TASKS
“Create a Component for PA Data” on page 115
“Work with PA Persistent Objects - Overview” on page 333
“Add a Data Object from a PA Persistent Object” on page 305

Map a Data Object to a PA Persistent Object

Mapping a data object to a persistent object consists of mapping of attributes and methods from one object to the other. Mapping of attributes and methods is required to define the bonding between the objects. A data object attribute can be mapped to one or more persistent object attributes and each special framework method of the data object can be mapped to one or more persistent object methods.

Restrictions:

- You cannot map multiple data object attributes to the same persistent object attribute.
- When you map a data object to multiple persistent objects, you must map each key attribute of the data object directly to each of the key attributes of the different persistent objects.

These are the preliminary steps you must follow before you can map a data object to a persistent object:

1. Create a PA schema and its associated PA persistent object by importing a PA bean.
2. Add a customized PA persistent object to the PA schema if you do not want to use the one Object Builder provides.

3. Add a data object implementation (The environment for the implementation should be **Procedural Adaptors**.)

Note the following points:

- To map a data object to a persistent object, there must be an association between the two objects, which you specify on the Associated Persistent Objects Page of the Data Object Implementation wizard.
- As soon as you associate a persistent object with the data object, the Attributes Mapping Page and the Methods Mapping Page are dynamically added to the wizard.

To define the mapping between the attributes of the data object and the persistent object, follow these steps:

1. If you are in the process of defining the data object implementation, proceed with step 2. If you have already defined the data object implementation, from the data object implementation's pop-up menu, select **Properties**. The Data Object Implementation wizard opens.
2. Turn to the Attributes Mapping Page. Here, you can map the data object interface attributes to the attributes of the persistent object.

You can map a data object attribute to a persistent object attribute in one of the following ways:

- Using the primitive pattern
- Using the exploded mapping pattern (for structures, which are complex attributes)
- Using a foreign key
- Using a mapping helper

To define the mapping between the methods of the data object and the persistent object, follow these steps:

1. If you are in the process of defining the data object implementation, proceed with step 2. If you have already defined the data object implementation, from the data object implementation's pop-up menu, select **Properties**. The Data Object Implementation wizard opens.
2. Turn to the Methods Mapping Page. When you define the mapping between methods, you define the processing order of the persistent object methods that you associate with the data object's special framework methods insert(), update(), retrieve(), del(). These persistent object methods act directly on elements of transaction logic in the legacy business applications.
3. The methods that you defined for the data object appear in the User-Defined Methods folder. You can map each of them to a push-down method of the PA persistent object.

RELATED CONCEPTS

"Data Object" on page 18
"Persistent Object" on page 19
"Special Framework Methods" on page 24
"User-Defined Methods" on page 23
"Push-Down Methods" on page 25

RELATED TASKS

"Add a Data Object Implementation" on page 299
"Work with PA Schemas - Overview" on page 337
"Map Data Object Attributes to Persistent Object Attributes" on page 256

“Create a Relationship” on page 129
“Work with Methods ” on page 267
“Use Push-Down Methods with PA Persistent Objects” on page 274

RELATED REFERENCES
“DB2 Data Type Mappings” on page 110
“Oracle Data Type Mappings” on page 113

Edit a PA Persistent Object

In this release of Object Builder, you cannot edit a PA persistent object that was created for you along with the PA bean that you imported into Object Builder.

However, you can add another persistent object to the schema (from the pop-up menu of the PA schema, select **Add Persistent Object**), and you can change the names of the attributes, if you want to. Once the persistent object is created, it is not editable.

RELATED CONCEPTS
“Persistent Object” on page 19

RELATED TASKS
“Work with PA Persistent Objects - Overview” on page 333
Edit Component Objects

Delete a PA Persistent Object

To delete a PA persistent object, follow these steps:

1. Select the persistent object from either the User-Defined Business Objects folder, the User-Defined Data Objects folder, or the User-Defined PA Schemas folder.
2. From the pop-up menu of the persistent object, select **Delete**.

If the persistent object is not connected to a data object implementation, the following deletions take place:

- the persistent object and its underlying schema are deleted from the User-Defined Data Objects folder.
- the persistent object is deleted from the User-Defined PA Schemas folder (the schema is not removed from this folder)

If the persistent object is associated with a data object implementation, the following deletions take place:

- the persistent object and its underlying schema are deleted from the User-Defined Business Objects folder and the User-Defined Data Objects folder.
- the persistent object is deleted from the User-Defined PA Schemas folder (the schema is not removed from this folder)

RELATED CONCEPTS
“Persistent Object” on page 19

RELATED TASKS
“Work with DB Persistent Objects” on page 313
Delete Component Objects

Work with PA Schemas - Overview

You can create a component for existing transactional information by importing the PA bean into Object Builder, and deriving a component from it.

The following tasks deal with PA schemas:

- “Create a PA Schema by Importing a PA Bean”
- “Add a Persistent Object from a PA Schema” on page 334
- “Edit a PA Schema” on page 339
- “Delete a PA Schema” on page 339

RELATED CONCEPTS
“Persistent Object” on page 19
“Schema” on page 20“Procedural Adaptor Bean (PA Bean)” on page 117

RELATED TASKS
Work with Components

Create a PA Schema by Importing a PA Bean

To import a procedural adaptor schema (PA schema), follow these steps:

1. In the Tasks and Objects pane, select the **User-Defined PA Schemas** folder.
2. From its pop-up menu, select **Import - Bean**. The Import Procedural Adaptor Bean wizard opens to the Bean Selection Page.
3. You can import the bean using one of the following methods:
   a. Specify the name of a Component Broker Procedural Adaptor bean class. Follow these steps:
      1) Select the **Enter bean name** radio button, and type the name of the class in the field. For example, to import the BeCashAcct bean, type paa.samples.cics.eci.acct.BeCashAcctPAO.
   b. Specify a JAR file. Follow these steps:
      1) Select the **Select a bean from an existing JAR file** radio button. The **Find JAR file** button becomes active, and you can use it to locate the file.
      Restriction: Only beans created using VisualAge for Java Release 2.0 are compatible with this release (2.0) of Component Broker.
      2) Once you have selected the JAR file (for example BeCashAcct.jar), the panel lists the classes contained in the file and you can select the one you want imported. The field just below the **Enter bean name** button shows the name of the selected object class.

   Note: Typically, the following types of classes are referenced in the bean:
   1) The PAA run-time file sompart.zip, and the ivjeab.jar file.
   2) User-defined classes (any JAR files corresponding to the PA beans you created using Enterprise Access Builder (EAB), and any other classes you defined that are used by the PA bean)

   If you get a message about files not being in the class path, follow these steps:

1) Ensure that the IBM Component Broker CICS and IMS Application Adaptor component is installed.
2) Ensure that all the user classes that are referenced in the bean are included in your system environment variable CLASSPATH. The class path must either contain the JAR file (if you archived the classes into a JAR file), or the directory under which the class files exist (if you did not create a JAR file). CLASSPATH must also include the path for the PA bean class.

4. Click **Next**. The Names and Connectors Page opens. Here, you can name the module and the persistent object to be associated with the PA schema. You can also select the connector type to be used to access objects. The connector type must match the one you used when you created the procedural adaptor bean. **390** When you choose OS/390 as the development (target) platform (**Platform - Constrain**), only the EXCI, OTMA, and Generic connector types are available for selection.

   When you select either NT and 390, or AIX and 390 as the development platforms, all the connector types (LU 6.2, HOD and ECI, besides the types available for OS/390) are available for selection.
5. Click **Next**. The Key Selection Page opens with the properties of the PA schema listed in the **Properties** box.
6. Select any of the properties you want as the key for the object from **Properties** box, and move them to the **Key Attributes** box.
7. Click **Next**. The Attribute Type Specification Page opens, and you can specify whether the attributes of the PA bean that are of character or string type are either single-byte, or multi-byte.
8. Click **Next**. The Method and Parameter Type Specification Page opens, and you can specify whether the return type of the methods defined on the PA bean that are of character or string type are either single-byte, or multi-byte. You can also specify the same for the character and string types of these method's parameters.
9. Click **Finish**. The bean will be imported into Object Builder. The PA schema (for example, BeCashAcctPAO) and its associated persistent object (BeCashAcctPAOPO) will now appear in the Tasks and Objects pane under the **User-Defined PA Schemas** folder.

Note: When you select the PA persistent object in the Tasks and Objects pane, the Methods pane shows you the the attributes and methods defined on the PA schema, based on those that you had defined on the PA bean that you imported. The names of method parameters may not appear as you specified them in the VA Java IDE. However, this is only superficial, and it does not affect the behavior when the method is called on the bean.

RELATED CONCEPTS

"Persistent Object" on page 19
"Schema" on page 20

RELATED TASKS

"Create a Component for PA Data" on page 115
"Work with PA Schemas - Overview" on page 337
"Add a Persistent Object from a PA Schema" on page 334
"Add a Data Object Implementation" on page 299

Edit a PA Schema

Note: You cannot rename the PA schema.

To edit a PA schema, follow these steps:

1. Select the PA persistent object in the User-Defined PA Schemas folder.
2. From its pop-up menu, select **Properties**. The PA Schema wizard opens to the Attributes Page. You can change the connector type information for the schema. You can select from HOD, ECI, LU 6.2, Generic, EXCI, or OTMA.

   390 If the deployment platform is OS/390, you can only select from among EXCI, OTMA, and Generic.

Note: If you had associated the PA persistent object that is connected with this PA schema, with a data object implementation, you must first disassociate this PA persistent object from the data object implementation (delete the persistent object from the Persistent Object Instances folder on the Associated Persistent Objects Page of the Data Object Implementation wizard) before you can change the connector type.

RELATED CONCEPTS

"Schema" on page 20

RELATED TASKS

"Work with PA Schemas - Overview" on page 337
Edit Component Objects

Delete a PA Schema

To delete a PA schema, follow these steps:

1. Delete any persistent objects that are associated with this PA schema.
   Note: If you delete the persistent objects from the the User-Defined Data Objects folder, or the User-Defined Business Objects folder, the schemas are automatically deleted from these folders. You still have to delete it from the User-Defined PA Schemas folder. (Follow step 2.)
2. From the pop-up menu of the schema in the User-Defined PA Schemas folder, select **Delete**.

RELATED CONCEPTS

"Schema" on page 20
"Persistent Object" on page 19

RELATED TASKS

"Work with PA Schemas - Overview" on page 337
"Delete a PA Persistent Object" on page 336
Delete Component Objects

Work with Managed Objects - Overview

Managed objects are defined in the User-Defined Business Objects folder, where they are shown under the business object implementation they were added to.

Once you configure a managed object with an application, an object representing that configuration appears in the Application Configuration folder, where it is shown under the application it was added to.

You can add multiple managed objects to each business object implementation, but each component you configure will have only one managed object. In fact, the component is defined by the configuration of the managed object.

The following tasks deal with managed objects:

- “Add a Managed Object”
- “Configure a Managed Object” on page 377
- “Edit a Managed Object” on page 341
- “Edit a Managed Object Configuration” on page 379
- “Delete a Managed Object” on page 341
- “Delete a Managed Object Configuration” on page 379

RELATED CONCEPTS
“Managed Object” on page 22

RELATED TASKS
Work with Components - Overview

Add a Managed Object

For a component to be installed on the server, it must have a managed object. End-user applications primarily interact with the managed object, which inherits the interface of the business object. The managed object controls the key and the copy helper, the relationship between the business object and its data object, and so on. You must create a new managed object for each of your business object implementations.

To add a managed object, follow these steps:

1. From the User-Defined Business Objects folder, select your business object implementation (for example, CarPolicyBO).
2. From the object’s pop-up menu, select **Add Managed Object**. The Managed Object wizard opens to the Name and Services Page.

   Appropriate names are filled in for you (the business object file name and interface name plus MO: for example, CPFile::CarPolicy gets a managed object CPFileMO::CarPolicyMO). You can accept these defaults or replace them with your own names.
3. Set the deployment platforms (the platforms on which this managed object will be deployed). This determines the development options that are selectable (you can only select options that are available on all selected platforms). By default, the managed object is deployable to the set of platforms defined in the **Platforms - Constrain** menu. You cannot select platforms that are **not** already selected in the **Platforms - Constrain** menu.
4. Make sure the correct service is selected. The services should be appropriate for the form of persistence provided by the component’s data object implementations.

   Note: Transactional Services is the appropriate choice for every form of persistence except Procedural Adaptors (which can use either service).

390: If one of your deployment platforms is OS/390, you can only select Transactional Services.

5. Click **Next**. The Implementation Inheritance Page opens.

   By default, no inheritance is selected. If, however, the managed object is for a component that already has an inheritance tree (for example, this is the managed object for the interface CarPolicy, which inherits from Policy), then the managed object should follow the same inheritance pattern (CarPolicyMO should inherit from PolicyMO).

   Note: The options available in the **Parent class** drop-down list are for defining a container or home (specialized forms of managed objects). There are separate instructions for these tasks. Do not use the list's options when creating a simple managed object.
6. Click **Finish**. The managed object appears in the User-Defined Business Objects folder, under your business object implementation.

RELATED CONCEPTS
"Managed Object" on page 22
An Overview of Application Adaptors (*Programming Guide*)

RELATED TASKS
"Work with Managed Objects - Overview" on page 339
"Set Platform Constraints" on page 189
"Build DLLs - Overview" on page 363
"Configure a Managed Object" on page 377

Edit a Managed Object

Managed objects are defined in the User-Defined Business Objects folder, where they are shown under the business object implementation they were added to. The configuration of a managed object with an application is represented by a separate object, in the Application Configuration folder, where it is shown under the application it was configured with.

You can change the services used by the managed object by following these steps:

1. From the pop-up menu of the managed object, click **Properties**. The Managed Object wizard opens to the Name and Services Page.
2. Change your selections as necessary.
3. Click **Finish** to apply your changes.

RELATED CONCEPTS
"Managed Object" on page 22

RELATED TASKS
"Work with Managed Objects - Overview" on page 339

Delete a Managed Object

To delete a managed object, follow these steps:

1. Delete its configuration, if any, in the Application Configuration folder.

   If the managed object is a customized home, then you must also remove it from any configurations of other managed objects that use it.
2. From the pop-up menu of the managed object, click **Delete**.

RELATED CONCEPTS
"Managed Object" on page 22
"Home"

RELATED TASKS
"Work with Managed Objects - Overview" on page 339
"Delete a Managed Object Configuration" on page 379
"Edit a Managed Object Configuration" on page 379

Work with Customized Homes - Overview

Customized homes, also known as specialized homes, are shown in the User-Defined Business Objects folder, and are presented in terms of five objects:

- The business object file (which contains one or more interfaces, optionally organized into modules), customized to be the file for a home.
- The business object module, if any (which contains one or more interfaces).
- The business object interface (which has one or more implementations), customized to be the interface for a home.
- The business object implementation (which has its own file, defined on the first page of its wizard), customized to be the implementation for a home.
- The managed object (which acts as an access point for the business object), customized to serve as a home.

The following tasks deal with customized homes:

- "Create a Customized Home" on page 343
- "Edit a Customized Home" on page 344
- "Delete a Customized Home" on page 345

RELATED CONCEPTS
"Home"

RELATED TASKS
Work with Components - Overview

Home

A home is the birthplace of managed objects. It serves as both a factory and a collection for managed objects. It is like a factory designed to manufacture only objects of a specific type.

Component Broker provides some default instances of homes, and most managed objects will use home instances based on these default ones. However, you can create a customized home (also known as a specialized home) in Object Builder if your managed objects require home instances with additional or specific behaviors.

When you add a managed object to an application, you select the type of home that will be used to create it on the Add Managed Object wizard, Home Page. When you generate the install image for the application, the generated DDL defines the home instance that will be responsible for finding and creating instances of the managed object.

RELATED CONCEPTS
"Managed Object" on page 22
"DDL" on page 114
Creating Specialized Homes (*Programming Guide*)

RELATED TASKS
"Create a Customized Home"
"Configure a Managed Object" on page 377

Create a Customized Home

When you configure a managed object with an application, you define a home instance that will be used to create and find instances of the managed object. Component Broker provides default home instances for you to base home instances on, which should be sufficient for most managed objects. However, you may want to create a home instance based on a customized home class for the needs of a particular Application Adaptor type, for example, by adding customized create and find methods.

To create a customized home class, follow these steps:

1. From the pop-up menu of the User-Defined Business Objects folder, click **Add File** to open the Business Object File wizard.
2. Click the title bar and turn to the Files to Include Page.
3. Under the Include Files folder, click the existing file to display its information.
4. Change the default include file:
   - If you want a queryable home, select IManagedAdvancedClient from the list.
   - If you don't need a queryable home, click the **Component Broker Customized Home** button to include the appropriate file.
5. Complete the remaining wizard pages and click **Finish**.
6. From the pop-up menu of the file you just added, click **Add Interface** to open the Business Object Interface wizard.
7. Click the title bar and turn to the Interface Inheritance Page.
8. Under the Parents folder, click the existing parent to display its information.
9. Change the default parent interface:
   - If you want a queryable home, select IManagedAdvancedClient::IQueryableIterableHome from the list.
   - If you don't need a queryable home, click the **Component Broker Customized Home** button to include the appropriate file.
10. Complete the remaining wizard pages and click **Finish**.
11. From the pop-up menu of the interface you just added, click **Add Implementation** to open the Business Object Implementation wizard.

    On the first page, the data access and data object options are absent because this is a customized home. The data object interface pages of the wizard are also absent.
12. Click the title bar and turn to the Implementation Inheritance Page.

    If you want a queryable home, change the default parent implementation to IManagedAdvancedServer::ISpecializedQueryableIterableHome
13. Complete the remaining wizard pages and click **Finish**.
14. From the pop-up menu of the implementation you just added, click **Add Managed Object** to open the Managed Object wizard.

15. Click the title bar and turn to the Implementation Inheritance Page.
16. From the Parents pop-up menu, click **Add**.
17. Select the appropriate Component Broker home class from the **Parent class** drop-down list.
18. Complete the remaining wizard pages and click **Finish**.
19. Configure the customized home, as a managed object, with your application.

    Note: Make sure that the customized home and its associated managed objects are configured with different containers. If a managed object and its home are configured with the same container, the server will not activate.

You now have a customized home class.

Customized homes do not require data object interfaces, data object implementations, copy helpers, or key classes. When you configure managed objects for an application, you can associate them with your customized home (on the Managed Object Configuration wizard, Home Page). An instance of the customized home class will be defined in the generated DDL for the managed object, and used on the server to create and find instances of the managed object.

You **must** package the customized home class in the same application as the managed objects that use it.

RELATED CONCEPTS
"Home" on page 342
"DDL" on page 114

RELATED TASKS
"Configure a Managed Object" on page 377
"Package an Application" on page 375

Edit a Customized Home

Customized homes are defined in the User-Defined Business Objects folder, where they are shown as a tree of business object file, module (if any), business object interface, business object implementation, and managed object. You can edit these objects by following these steps:

1. From the pop-up menu of the object, click **Properties** to display the appropriate wizard.
2. Click the title bar to select a page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

   Note: When you click **Finish**, the framework methods for the business object implementation are recalculated. If you made any changes to the framework method implementations (not recommended), those changes are lost.

RELATED CONCEPTS
"Home" on page 342

RELATED TASKS
"Work with Customized Homes - Overview" on page 342

Delete a Customized Home

To delete a customized home, follow these steps:

1. Remove the customized home from any managed object configurations that use it.
2. Delete the managed object configuration for the customized home's managed object.
3. Delete the customized home's managed object.
4. Delete its business object implementation.
5. Delete its business object interface.
6. If it is the only interface defined in the file, delete the business object file.

RELATED CONCEPTS

"Home" on page 342

RELATED TASKS

"Work with Customized Homes - Overview" on page 342

Work with Container Instances - Overview

Containers provide object services for components. Default containers are provided for objects with transient data. If you have objects with persistent data, or want to customize the types of service that are provided by a container, you need to define your own container instance.

The following tasks deal with containers:

- "Create a Container Instance" on page 346
- "Edit a Container Instance" on page 348
- "Delete a Container Instance" on page 348

RELATED CONCEPTS

"Container"

RELATED TASKS

Work with Components - Overview
"Configure a Managed Object" on page 377

Container

A container is a configured version of a particular application adaptor that represents a physical boundary around objects. It can be thought of as where the objects exist. A container can provide some level of isolation between it and other containers. A container can also provide some isolation among objects within the container.

When you add a managed object to an application, you configure it with a container that will be responsible for handling object services for the managed object.

RELATED CONCEPTS

"Managed Object" on page 22

RELATED TASKS

"Configure a Managed Object" on page 377
"Create a Container Instance"

Create a Container Instance

The Component Broker frameworks provide a number of default containers, which are appropriate for components with transient data (that is, without persistent objects). If your component has data that you want to be persistent, you must define a container for the particular needs of the component. You can define new containers in the Container Definition folder.

To add a container instance, follow these steps:

1. Under Tasks and Objects, select the Container Definition folder.
2. From the folder's pop-up menu, click **Add Container Instance**. The Container wizard opens to the Name Page.
3. Type a name and description for the container.

   390: If you are developing an application intended for deployment on OS/390 (the **Platform - Constrain - 390** menu choice is checked), then you are now done. The rest of the container definition is handled through the System Management user interface.
4. In the **Number of Components** field, type an estimate for the number of managed objects this container will hold. This sets a lower limit on the size of the container's hash table; additional space will be allocated when it is needed.
5. Click **Next**. The Workload Management page opens.
6. Specify whether the container is workload managing. If you check this option, you must also specify the policy group it will be configured with. For new policy groups, accept the default <New> entry.
7. Click **Next**. The Service Page opens.
8. Select the policies and services the container will provide for its components. You can select from the following:
   - **Use no Object Services**
     The container will provide no object services. Select this option if the container will contain components with transient data only, or if data persistence is provided without object services.
   - **Use Home Services**
     The container will use Home Services. Select this option if the container will contain customized homes.
   - **Use RDB Transaction Services**
     The container will use Transaction Services. Select this option if the container will contain components with database persistence, or if you want to provide transaction support for components with transient data.
   - **Use PAA Transaction Services**
     The container will use PAA Services. Select this option if the container will contain components with persistence provided by a procedural adaptor, for which you want to provide transaction services.
   - **Use PAA Session Services**
     The container will use PAA Services. Select this option if the container will contain components with persistence provided by a procedural adaptor, or if you want to provide transaction support for components with transient data.
9. If you select not to provide Transaction Services, select how the container should handle object data when the server stops running.

10. Select whether the container should passivate objects not in use, or keep objects in memory at all times.
11. If you select not to provide any object services (**Use no Object Services**), select whether to enable persistent references.

    The **Enable persistent references** option is only available if you select **Use no Object Services**. By default, object references are not made persistent when there are no object services enabled (the data is assumed to be transient), and instances held by the container are dropped when the server stops. Check this option if your components have persistent data, that you are accessing without the Object Services.
    If you check this option, then an attempt to find a component in the container will first try looking in current memory, and if that fails, then try calling the retrieve method of the component's data object implementation. If the retrieve method does not throw an exception, the retrieve is assumed to be successful, and the container returns an object reference.
    To force the retrieve method to fail for a particular data object (when, for example, there is no datastore to access), you can modify the code of the retrieve method to return exception IBOIMException::IDataKeyNotFound.
    If you do not check this option, then an attempt to find a component in the container will succeed only if the component is currently in memory. The component will be in memory if it has been created and added to the container since the server was last started. Check this option if your components have only transient data.
12. Click **Next**.
13. If you selected **RDB Transaction Services**, **PAA Transaction Services**, or **PAA Session Services**, then the Services Details Page now opens. Specify the behavior you want for methods called outside the scope of a transaction or session, and for sessions specify the type of session. For PAA Session Services, specify the name of the connection.
14. Click **Next** when you are done.
15. The Data Access Patterns Page opens. Select the options on this page according to the options set for the objects the container will hold.
16. Under **Business Object**, click **Delegating** or **Caching** according to the option selected for the business object implementation's **Pattern for Handling State Data** (Business Object Implementation wizard, Name and Data Access Pattern Page).
17. Under **Data Object**, click **Delegating** or **Local copy** according to the option selected for the data object implementation's **Data Access Pattern** (Data Object Implementation wizard, Behavior Page).
18. If you select **Delegating**, then you need to indicate whether or not the data object uses the Cache Service. Select the **Cache Service** check box if the data objects have their **Form of Persistent Behavior and Implementation** set to either **DB2 Cache Service** or **Oracle Cache Service** (Data Object Implementation wizard, Behavior Page). Otherwise, click **No**.
19. Click **Finish**. The new container is added to the Container Definition folder.

RELATED CONCEPTS
"Container" on page 345
Workload Management
Transaction Service
Session Service
Cache Service

RELATED TASKS
"Configure a Managed Object" on page 377
"Work with Container Instances - Overview" on page 345

Edit a Container Instance

To edit a container instance you have defined, follow these steps:
1. From the pop-up menu of your container, click **Properties**. The Container Definition wizard opens to the Name of Container and Number of Components Page.
2. Click the title bar to select another page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

RELATED CONCEPTS
"Container" on page 345

RELATED TASKS
"Work with Container Instances - Overview" on page 345

Delete a Container Instance

You cannot delete the default container instances. To delete a container instance that you have defined, follow these steps:
1. Remove the container from any managed object configurations that use it.
2. Locate the container in the Container Definition folder.
3. From the pop-up menu of the container, click **Delete**.

RELATED CONCEPTS
"Container" on page 345

RELATED TASKS
"Work with Container Instances - Overview" on page 345
"Package an Application" on page 375
"Edit a Managed Object Configuration" on page 379

Work with Compositions - Overview

A composition defines a combined interface for a group of components. In addition, it describes the implementation of the attributes and methods in the combined interface, which delegate to attributes and methods of the components in the group. Once you have combined the components into a composition, you can create composite components that are based on the composition.

The following tasks deal with compositions:
- "Create a Composition File" on page 349
- "Add a Composition Module" on page 349
- "Add a Composition" on page 350
- "Edit a Composition" on page 352

RELATED CONCEPTS
"Composition" on page 174

RELATED TASKS
"Create a Composite Component - Overview" on page 172
"Work with Composite Business Objects - Overview" on page 353
"Work with Composite Keys - Overview" on page 360

Create a Composition File

A composition file (IDL) is a container for your compositions. Although a file can hold multiple compositions, which you may organize into modules, you typically add one composition to each file.

To create a composition file, follow these steps:

1. From the Tasks and Objects pane, select the **User-Defined Compositions** folder.
2. From the folder's pop-up menu, select **Add File**. The Composition File wizard opens to the Name Page.
3. Type a name for the file (for example, CGFile).
4. Click **Next**. The Constructs Page opens.

   Use the Constructs pop-up menu to add enumerations, exceptions, structures and so on. Any constructs you add are scoped to every interface in the file.
5. Click **Next**. The Files to Include Page opens.

   IManagedClient is included by default. Do not change this.
6. Click **Next**. The Comments Page opens. Type any comments you want to include as comment lines in your generated IDL code.
7. Click **Finish**. The wizard closes, and your file is added to the User-Defined Compositions folder. You can now add modules or interfaces to the file.

Once you have created the file, you can modify it by selecting **Properties** from its pop-up menu. The Composition File wizard opens again, with your selections preserved.

RELATED CONCEPTS
"Composition" on page 174

RELATED TASKS
"Create a Composite Component - Overview" on page 172
"Work with Compositions - Overview" on page 348
"Add a Composition Module"
"Add a Composition" on page 350

Add a Composition Module

If you plan to add multiple compositions to a single file, you may want to store the compositions in separate modules. Any constructs you add to a module are scoped only to the compositions within that module. To add a module to a file, follow these steps:

1. From the User-Defined Compositions folder, select your composition file.
2. From the file's pop-up menu, select **Add Module**. The Composition Module wizard opens to the Name Page.

3. Type a name for the module.
4. Click **Next**. The Constructs Page opens.

   Use the Constructs pop-up menu to add enumerations, exceptions, structures and so on.
5. Click **Next**. The Comments Page opens. Type any comments you want to include as comment lines in your generated code.
6. Click **Finish**. The wizard closes, and your module is added to the User-Defined Compositions folder, underneath the file.

You can now add compositions to the module.

RELATED CONCEPTS
"Composition" on page 174

RELATED TASKS
"Create a Composite Component - Overview" on page 172
"Work with Compositions - Overview" on page 348
"Add a Composition"

Add a Composition

The composition is a server-side implementation object that provides a composite business object with access to its member components' methods and data. It can also define its own methods and attributes for use by the composite business object.

To add a composition to a file (or module), follow these steps:

1. From the User-Defined Compositions folder, select the file or module that will contain the interface.
2. From the pop-up menu for the file or module, click **Add Composition**. The Composition Editor opens.
3. Click **Add** to open the Composition Palette.
4. Select the components you want to add to the composition.
5. Click **Add**, then **Close**. The components are added to the composition in the form of managed object instances, with default names based on the original component interface (for example, SavingsAccount component becomes SavingsAccount1).
6. Review the list of objects to composite in the Objects to Composite list.
7. You can rename a managed object instance by clicking on its name and then clicking **Rename** or by double-clicking on its name.
8. Select a composition style to apply to the objects. The result is applied to the list of composited attributes and methods in the Results pane. For conjunction composites you should choose the variant (that is, with or without name matching) that produces a result that is closest to what you want.

   If you choose the Conjunction with name matching style, but would still like certain attributes or methods to remain separate, you can selectively reverse the name-matching and split the combined attribute or method into its separate elements. To split a combined attribute or method, select **Split** from the pop-up menu of the attribute or method.

   If you choose the Conjunction without name matching style, but would still like certain attributes or methods to be combined (as if you had chosen the

Conjunction with name matching style), you can selectively match and combine attributes or methods. To combine multiple attributes or methods, select them by holding down the Ctrl key and clicking the left mouse button, then click on the last one with the Ctrl key plus right mouse button to display its pop-up menu, and select **Equate**.

You can use the **Equate** command to join attributes or methods with the same type. They do not need to have the same names.

9. Click on an attribute or method to view its republishing (delegating) behavior, in the Current Republish Value pane.
10. Click on the Properties tab to display the properties of the currently selected attribute and method. Double-clicking on an attribute or method will also display its properties.
11. Only the name of the attribute or method is editable, because their definitions are based on their equivalents in the combined components.
12. Add any new attributes or methods you want to be part of the composition, that may or may not be based on combined components. You can add new attributes or methods from the pop-up menus of the folders in the Results pane. These new methods could, for example, provide extra processing of the information being combined (beyond simple delegation).

    For example, a composition AllAccounts, which combines the components CheckingAccount and SavingsAccount, could have a private helper method addFloats, which can take the two original balances (CheckingAccount1.balance and SavingsAccount1.balance) as arguments, and return their sum. You can then map AllAccounts.balance to the helper method.

    When you add a new method, you can supply its implementation (for example, return arg1+arg2) in Object Builder's Source pane (after you complete the composition, click on it in the Tasks and Objects pane; then select the method in the Methods pane, and complete its implementation in the Source pane).
13. Edit republishings using the pop-up menu of the current value in the pane. You can also change a republish value by simply double-clicking on it, and then selecting a new value from the drop-down list that appears.

    For conjunction composites, you can map attributes and methods either to attributes and methods of the combined components, or to a *sequence* of attributes and methods (in which all listed attributes or methods are called in sequence, and the result of the last one is returned). The delegating attribute or method must have a type or return type that matches the result of the last call in the sequence.

    Disjunction attributes and methods usually map to a *select* of attribute and methods (that is, a list of mutually exclusive attributes and methods, only one of which will exist at run time and be called).

    You can map to attributes or methods of the combined components, or to other attributes and methods that are unique to the composition.

    For example, if the composition AllAccounts combines CheckingAccount and SavingsAccount with the Conjunction with name matching style, then by default AllAccounts.balance returns a **sequence** of CheckingAccount1.balance and SavingsAccount1.balance (which simply returns the second value in the sequence). You can replace this default mapping with a more useful one that returns their sum, by adding a private helper method addFloats (as described in the previous step), and changing the mapping to call the helper method, with the two original balance attributes as arguments.
14. Click on the parent folder (representing the composition as a whole) in the **Results** pane. By default, its name is **Untitled**.

15. Click on the **Properties** tab.
16. Type a name for the composition. The name is reflected in the **Results** pane.
17. Set the implementation language (C++ or Java).
18. Click **OK**.

RELATED CONCEPTS
"Composition" on page 174

RELATED TASKS
"Create a Composite Component - Overview" on page 172
"Work with Compositions - Overview" on page 348
"Add a Composite Business Object Interface" on page 354

Edit a Composition

Compositions are defined in the User-Defined Compositions folder, where they are shown under the file (and module, if any) in which they are defined. You can edit the file and module as separate objects, following these steps:

1. From the pop-up menu of the file or module, click **Properties** to display the appropriate wizard.
2. Click the title bar to select a page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

To edit the attributes or methods of the composition, follow these steps:

1. From the pop-up menu of the composition group, click **Properties** to open the Composition Editor.

   You can edit the delegating behavior of the methods or attributes currently in the composition, add new methods or attributes that are unique to the composition, or change what components are combined in the composition.
2. When you are done editing, click **OK**.

From within the editor, you can turn to the Compositions page to change which components make up the composition:

To delete a component from the composition, follow these steps:

1. Select its managed object instance in the Objects to Composite list.
2. Click **Delete**.

To add a component to the composition, follow these steps:

1. Click **Add** to open the Composition Palette.
2. Select a managed object.
3. Click **Add**, then **Close**.

To rename a component in the composition, follow these steps:

1. Select its managed object instance in the Objects to Composite list.
2. Click **Rename**.
3. Type over the old name.
4. Click elsewhere in the list to apply the new name.

Once you are done editing, click **OK** to apply your changes.

If your changes are limited to renaming or deleting elements, then your changes will automatically be reflected in the other composite component objects that are based on the composition (for example, the business object and key). If, however, you added new components to the composition, you need to provide these objects with the information necessary to locate or create instances of the new component.

When you have added new components to a composition, follow these steps for each composite component based on the composition:

1. From the pop-up menu of the composite component's key, click **Properties** to open the Key wizard.
2. Add any new key attributes that may be required and provide the composite key to component key mappings if possible.
3. Click **Finish**.
4. From the pop-up menu of the composite component's business object implementation, click **Properties** to open the Business Object Implementation wizard.
5. Click the title bar and turn to the Location page.
6. Review and update the location information for any new or edited components of the composition.
7. Click the title bar and turn to the Data Object Interface page.
8. Add any new key attributes to the data object interface.
9. Click **Finish**.

RELATED CONCEPTS

"Composition" on page 174

RELATED TASKS

"Work with Compositions - Overview" on page 348

Work with Composite Business Objects - Overview

Composite business objects are defined in the User-Defined Business Objects folder, and are presented in terms of four objects:

- The business object file (which contains one or more interfaces, optionally organized into modules)
- The business object module, if any (which contains one or more interfaces)
- The business object interface (which has one or more implementations)
- The business object implementation (which has its own file, defined on the first page of its wizard)

The four objects are created and edited separately, but collectively form a single business object. Each business object (each set of business object file, module, interface, and implementation) typically has its own data object.

A business object is composite when it is based on a composition, as set by the business object interface. A composite business object has attributes and methods based on those in the composition, which are in turn based on the composited components that make up the composition.

The following tasks deal with composite business objects:

- "Add a Composite Business Object Interface" on page 354

- “Add a Composite Business Object Implementation and Data Object Interface” on page 355
- “Edit a Composite Business Object Interface” on page 359
- “Edit a Composite Business Object Implementation” on page 360

RELATED CONCEPTS
“Composite Business Object” on page 175

RELATED TASKS
“Create a Composite Component - Overview” on page 172
“Work with Composite Keys - Overview” on page 360

Add a Composite Business Object Interface

Once you have defined a composition, you can create composite components that are based on the composition, starting with the business object.

First, add a business object file (and optionally module) to the User-Defined Business Objects folder.

To add a composite business object interface to a file (or module), follow these steps:

1. From the User-Defined Business Objects folder, select the file or module that will contain the interface.
2. From the pop-up menu for the file or module, select **Add Interface**. The Business Object Interface wizard opens to the Name Page.
3. Type a name for the interface (for example, CompositeCustomer). Do not use the same name as the composition unless one or both are nested in modules.
4. Select the **Composite** check box.
5. From the **Composition to Use** list, select the composition you want to base the interface on.
6. Click **Next**. The Constructs Page opens.
7. Use the Constructs pop-up menu to add enumerations, exceptions, structures and so on. Any constructs you add are scoped to this interface only.

   Note: To use the construct as the type of an attribute, method return, or method exception, you must first click **Finish** and then re-open the wizard and define the attribute. The construct is not added to the current model until you click **Finish**.
8. Click **Next**. The Interface Inheritance Page opens.

   By default, the interface inherits from IManagedClient::IManageable. This is the correct choice for a component that represents a base class in your design. If your component had a parent, you would specify the business object interface of the parent component on this page.
9. Click **Next**. The Attributes Page opens.

   The public attributes (except for the attributes that represent references to instances of the combined components) of the composition you selected appear here, but are not editable. You can edit them (for example, change their names or delegating behavior) in the composition where they are defined. Changes to the composition are applied to the composite business object automatically.

When you add the composite business object implementation, the get and set methods for these attributes will be implemented, and delegate to the composition helper object.

To specify additional attributes for your interface, select **Add** from the Attributes pop-up menu. When you add the business object implementation, these attributes will receive default implementations in the usual manner.

10. Click **Next**. The Methods Page opens.

    The public methods of the composition you selected appear here, but are not editable. You can edit them in the composition where they are defined. Changes to the composition are applied to the composite business object automatically.

    When you add the composite business object implementation, the implementations for these methods will be implemented, and delegate to the composition helper object.

    To specify additional methods for your interface, select **Add** from the Methods pop-up menu. When you add the business object implementation, you can provide your own implementations for the methods in the usual manner.
11. Click **Next**. The Object Relationships Page opens.

    To specify any relationships that this class has to other classes, select **Add** from the Objects pop-up menu. You can specify how the relationship will be implemented when you add the business object implementation.
12. Click **Next**. The Comments Page opens. Type any comments you want to include as comment lines in your generated code.
13. Click **Finish**. Your new interface is added to the User-Defined Business Objects folder, with the attributes and methods of the composition you selected, as well as any additional attributes and methods you specified.

You should now see your interface in the Tasks and Objects pane. Any methods defined for your interface should appear under the **User-Defined Methods** folder in the Methods pane, and any attributes defined for your interface should appear under the **User-Defined Attributes** folder in the Methods pane.

RELATED CONCEPTS

“Composite Business Object” on page 175
“Business Object” on page 17

RELATED TASKS

“Create a Composite Component - Overview” on page 172
“Work with Composite Business Objects - Overview” on page 353
“Add a Composite Key” on page 360

Add a Composite Business Object Implementation and Data Object Interface

Once you have created a composite business object interface, you must add one or more implementations for that composite business object, and also create its data object interface. You can accomplish both tasks using the Business Object Implementation wizard. Ensure that you have added a key to the composite business object interface before proceeding with this task.

To create the composite business object implementation, and its associated data object interface, follow these steps:

1. From the User-Defined Business Objects folder, select the composite business object interface you want to implement.
2. Display the pop-up menu for the interface and select **Add Implementation**. The Business Object Implementation wizard opens to the Name and Data Access Pattern Page.

   Appropriate implementation names are filled in for you (the business object file name and interface name plus BO: for example, AAFile::AllAccounts gets an implementation named AAFileBO::AllAccountsBO).You can accept these defaults or replace them with your own names.
3. Select the pattern you want to use for handling the component's state data (that is, any attributes of the component that are **not** derived from the composition it is based on). The following patterns are available:
   - **Delegating**
     The business object delegates every request for the essential state to the data object interface.
   - **Caching**
     Both the business object and the data object have their own copies of the essential state, which are synchronized. **Lazy evaluation** is the default synchronization method, meaning that cached copies of the attributes are synchronized at first use, rather than at instantiation.
   - **Same as parent's**
     The business object inherits its pattern from a parent interface.
     Note: This option is selected by default if the interface for this business object inherits from another business object interface. However, you still have to indicate the implementation parent on the Implementation Inheritance page of this wizard.

   There is also an option listed for **None**, which would generate a transient data object. This option is not available in this release.

   The pattern you select will apply for any attributes you created that are unique to the composite component. It does not apply to any attributes derived from the component's composition. Attributes derived from the composition are always implemented as delegation calls to their equivalents in the composition helper object, regardless of the pattern selected here.
4. Select whether to create a new data object now, or add or select one later.
5. Click **Next**. The Implementation Inheritance Page opens.
6. Make sure that IManagedClient::IManageable is listed as a parent under the Parent Class folder.
7. You can also select any parent business object implementations you want to inherit behavior from.
8. Click **Next**. The Implementation Language page opens. Select the language you want the business object to be implemented in. You can select either Java or C++.
9. The default for this page is set in the Preferences notebook, on the Tasks and Objects page.
10. Click **Next**. The Attributes Page opens.

    A private attribute with the same name as the composition being used is automatically included. This attribute is used to access the composition helper

object in the delegating implementations of composite attributes and methods (for example, the composite component method. *debit* calls the composition's method *iCompositeAccount.debit*).

You can also specify any attributes you want to add to the business object implementation (in addition to the attributes you already specified in the business object interface).

11. Click **Next**. The Methods Page opens.
12. Several private methods related to composition are automatically included:
    - A method of the form *loc_<instance name>* is included for each component instance of the composition being used (e.g. loc_SavingsAccount1 and loc_CheckingAccount1). These methods are called during activation to locate or create the managed objects that are used to initialize the composition helper object when it is created. The implementation of these methods is automatically generated by ObjectBuilder using the information provided on the Location Page (see below).
    - A method of the form *get_<instance name>_<key attribute name>* is included for every key attribute of each component instance of the composition being used (for example,. get_SavingsAccount1_accountNo and get_CheckingAccount1_accountNo). These methods are called by the loc_ methods to get the values used to initialize the primary key attributes of the component instances. These methods will be automatically generated by ObjectBuilder if simple key attribute mappings were supplied for the composite key.

    You can also specify any methods you want to add to the business object implementation (in addition to the methods you already specified in the business object interface).
13. Click **Next**. The Key and Copy Helper Page opens. Select a key and, optionally, copy helper that you have created for this business object (for example, AllAccountsKey and AllAccountsCopy).
14. Click **Next**. The Handle Selection Page opens.

    You can select a handle for the business object implementation. If you select a handle, then the framework method getHandleString is implemented, which overrides the getHandleString method of IManagedClient::IManageable. The method provides a way to encapsulate the business object implementation, by returning a string that represents a reference to the business object. The handle you select determines the pattern used to form the string (that is, to turn the reference into a string, or to swizzle the pointer).
15. Click **Next**. The Location Page opens.

    On this page, you set the composite business object's relationship to the managed objects being combined in the composition.
16. For each managed object in the composition, provide the following information:
    a. Indicate whether it should be destroyed when the composition is destroyed, or have its destruction managed independently.
    b. Indicate the expected state of the managed object:
       - Click **Find or create** if the managed object might or might not already exist.
       - Click **Find** if the managed object must already exist (and should not be created if it doesn't).
       - Click **Create** if the managed object must **not** already exist (and should not be returned if it does).

c. Indicate whether the managed object should be created using a copy helper instead of its primary key. When creating a component using a copy helper, the attributes that are also key attributes will be initialized as usual (by calling get_ methods). The other attributes on the copy helper will be set to the initial values specified in the “Interface wizard” of the component being created.

   Note: Components using PAA Services (i.e. CICS components) can only be created using a copy helper.

d. Select the way the managed object should be located.

e. Provide the information necessary to implemented the selected location pattern.

17. Click **Next**. If the business object implementation has parent classes with overrideable attributes, then the Attributes to Override Page opens.

    You can use this page to select which of the parent class’s attributes you want to override.

18. Click **Next**. If the business object implementation has parent classes with overrideable methods, then the Methods to Override Page opens.

    You can use this page to select which of the parent class’s methods you want to override.

19. Click **Next**. If the business object interface defines one-to-many relationships, then the Object Relationships page opens.

    You can use this page to set the way that the object relationship will be implemented.

20. Click **Next**. The Data Object Interface Page opens. (Note: This page does not open if, on the first page, you chose not to create a new data object.)

    Appropriate data object names are filled in for you (the business object file name and interface name plus DO: for example, AAFile::AllAccounts gets the data object interface AAFileDO::AllAccountsDO). You can accept these defaults or replace them with your own names.

    If you implemented a one-to-many relationship as a **Local persistent reference**, then an attribute representing it appears here, so you can select to preserve it in the data object.

21. Select the attributes you want preserved in the data object. Because this component is a composite one, state for all of the composite attributes is already preserved in the referenced components. In other words, composite attributes are already preserved in the data objects of their originating components. You only need to select the key attributes here, and any non-composite (not derived from the composition) attributes you defined for the business object.

22. Click **Next**. The Data Object Methods Page opens. (This page does not open if, on the first page, you chose not to create a new data object.)

23. Select which business object methods you want to push down to the data object (that is, call equivalent methods to be defined in the data object).

24. Click **Next**. The Summary of Framework Methods Page opens.

    Based on your selections on the previous pages of the wizard, this page displays the methods that your object implements. For example, if you selected a caching pattern to handle the essential state of your business object (on the first page), this list includes the synchToDataObject method required to keep the two sets of attributes synchronized.

Because this business object is a composite, this list also includes two composition methods, initializeComposition and uninitComposition. These two methods are also automatically generated by Object Builder.

You can review the framework methods before closing the wizard.

25. Click **Finish**. The business object implementation and data object interface appear in the User-Defined Business Objects folder, under your business object interface. The data object interface also appears in the User-Defined Data Objects folder.

Now that the business object implementation is defined, you can enter the implementation code for any new methods you defined.

RELATED CONCEPTS

"Composite Business Object" on page 175
"Business Object" on page 17
"Data Object" on page 18

RELATED TASKS

"Create a Composite Component - Overview" on page 172
"Work with Composite Business Objects - Overview" on page 353
"Add Code for User-Defined Methods" on page 267
"Add a Data Object Implementation" on page 299
"Define a One-to-Many Relationship" on page 131

Edit a Composite Business Object Interface

Composite business object interfaces are defined in the User-Defined Business Objects folder, where they are shown under the file (and module, if any) in which they are defined.

Composite business objects are based on compositions, from which they derive attributes and methods. These derived methods are not editable in the business object interface. You can edit them (for example, change their names or delegating behavior) in the composition where they are defined. Changes to the composition are applied to the composite business object automatically.

You can edit the file, module, and the non-composite attributes and methods of the business object interface as follows:

1. From the pop-up menu of the file, module, or interface, click **Properties** to display the appropriate wizard.
2. Click the the title bar to select a page to turn to.
3. Change your selections as necessary.

   If you want to specify a parent for the interface after you have defined the implementation for the business object, follow these steps:

   a. Add the parent to the **Parents** folder on the Interface Inheritance page of the Business Object Interface wizard
   b. Open the Business Object Implementation wizard, and on the Name and Data Access Pattern page specify the pattern for handling state data as **Same as parent's**.
   c. Click **Next.**
   d. Add the implementation parent on the Implementation Inheritance page.
4. Click **Finish** to apply your changes.

RELATED CONCEPTS
"Composite Business Object" on page 175

Edit a Composite Business Object Implementation

Composite business object implementations are defined in the User-Defined Business Objects folder, where they are shown under the composite business object interface they were added to. You can edit a composite business object implementation by following these steps:

1. From the pop-up menu of the business object implementation, click **Properties**. The Business Object Implementation wizard opens to the Name and Data Access Pattern Page.
2. Click the title bar to select another page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

Note: The **Same as parent's** option is selected by default if the interface for this business object inherits from another business object interface. However, you still have to indicate the implementation parent on the Implementation Inheritance page of this wizard.

RELATED CONCEPTS
"Composite Business Object" on page 175

Work with Composite Keys - Overview

A composite key object defines which attributes are to be used to find a particular instance of the composite component on the server. The key consists of one or more of the business object attributes, which must contain enough information to uniquely identify an instance.

The following tasks deal with composite keys:

- "Add a Composite Key"
- "Edit a Composite Key" on page 362

RELATED CONCEPTS
"Composite Key" on page 176
"Composite Component" on page 173

RELATED TASKS
"Create a Composite Component - Overview" on page 172
"Work with Composite Business Objects - Overview" on page 353

Add a Composite Key

Each composite component must have a primary key class that contains enough information to uniquely identify the component. The key is used when new instances of the component are created or when existing instances need to be found.

Once you have created a composite business object interface, you can define its composite key.

To add a composite key, follow these steps:

1. From the User-Defined Business Objects folder, select your composite business object interface.
2. From the object's pop-up menu, select **Add Key**. The Key wizard opens to the Name and Key Attributes Page.
3. Appropriate key names are filled in for you (the business object file name and interface name plus Key: for example, AAFile::AllAccounts gets a key named AAFileKey::AllAccountsKey). You can accept these defaults or replace them with your own names.
4. Select the composite business object attributes that make up the primary key.

   If possible, you will want to select attributes that were part of the keys for the original combined components.

   For example, given the following situation:

   - A composite component AllAccounts is based on a composition of two other components, SavingsAccount and CheckingAccount.
   - The primary keys of both SavingsAccount and CheckingAccount contain a single attribute accountNo (the account number).
   - The two account numbers are exposed in the composite business object as attributes savingsAccountNo and checkingAccountNo.

   If you select the attributes savingsAccountNo and checkingAccountNo for the primary key of AllAccounts, the composite key then includes all the information needed not only to uniquely identify the AllAccounts component, but to identify the SavingsAccount and CheckingAccount components as well. This eliminates the need to maintain persistent references from the composite component to the original combined components.

   If the composite business object has a parent business object (specified on the Interface Inheritance page of its wizard), you can also select from the parent interface's attributes (you should **not** select attributes of the parent interface if you are planning to inherit from the parent interface's key).
5. Click **Next**. The Composite Key Page opens. Here you are given the opportunity to provide mappings between the composite key attributes and the attributes of keys for the grouped components.
6. For each key attribute you selected that corresponds directly to an attribute of a component key, describe the mapping:
   a. Select an attribute in the Composite Key list (for example, checkingAccountNo).
   b. Select an attribute of a key in the Composite Key Elements list (for example, the accountNo attribute of CheckingAccountKey).
   c. Click **Add**.
7. Click **Next**. The Implementation Inheritance Page opens.

   On this page, you can specify the type of key (primary or unique), and inherit from the appropriate parent class (IPrimaryKey or IUniqueKey).
8. Verify that the primary key type is selected.

   If the key has a parent, you can specify it here.

   Note: You should not inherit from a parent key if you also selected inherited attributes on the previous page.
9. Click **Next**. The Summary of Framework Methods Page opens. This page summarizes the framework methods this object implements. No action is needed.

10. Click **Next**. The Optional Framework Methods Page opens. Select any additional framework methods you want to implement. Object Builder will add signatures for the methods you select, but you must provide your own implementation code. The methods you implement will override the equivalent framework methods of the parent class.

    Note: The editor pane will not allow you to edit these methods until you set them as editable in the Method Implementation wizard. To set a method as editable, follow these steps:

    a. In the Methods pane, select the framework method.
    b. From its pop-up menu, click **Properties**.
    c. In the Method Implementation wizard, specify that you want to use the implementation defined in the editor pane. This lets you use the Source pane editor to edit the method implementation.
11. Click **Finish**. The key appears in the User-Defined Business Objects folder, under your composite business object interface.

In the Methods pane, you should see some items listed in the Framework Methods folder. Default implementation code is provided for these methods, which you can view in the edit pane by selecting a method. Normally, you will not want to edit this code (except for the code for the optional framework methods, as noted above). The code for framework methods is read-only by default.

RELATED CONCEPTS

"Composite Key" on page 176
"Key" on page 21

RELATED TASKS

"Create a Composite Component - Overview" on page 172
"Work with Composite Keys - Overview" on page 360
"Add a Composite Business Object Implementation and Data Object Interface" on page 355

Edit a Composite Key

Composite keys are defined in the User-Defined Business Objects folder, where they are shown under the composite business object interface they were added to. You can edit a key by following these steps:

1. From the pop-up menu of the key, click **Properties**. The Key wizard opens to the Name and Key Attributes Page.
2. Click the title bar to select a page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

RELATED CONCEPTS

"Composite Key" on page 176

Chapter 11. Configuration Tasks

Build DLLs - Overview

Once you have defined your components in Object Builder, you are ready to build the components into client and server dynamic link libraries (DLLs, also known as shared library files). The client DLLs contain the component interfaces, and helper classes, which allow your client applications to locate and use the components on the server. The server DLLs contain the implementations and data objects for the component.

To build your DLLs, complete the following steps:

1. "Generate Code"
2. "Define a Client DLL" on page 364
3. "Define a Server DLL" on page 366
4. "Generate a Makefile" on page 367
5. "Build the DLLs" on page 368

Once you have built the DLLs, you can debug them, or package them as part of an application.

RELATED TASKS
Develop Applications in Object Builder - Overview
"Build DLLs in a Team Environment" on page 217
"Package an Application" on page 375

Generate Code

Before building an application, you must generate source code for the objects you have created. By selecting **Generate - Selected** or **Generate - All** from an object's pop-up menu, you can generate code for that object only, or for that object *and* all objects below it in the tree. You can also generate code for a project from a command line, using the obgen command.

Until you generate code, all information for your objects is maintained in an Object Builder model (for example, MyProject/Model/*.uni). When you generate, the resulting files are placed in the /Working/*platform* subdirectory of the project directory you specified, ready to be compiled (for example, MyProject\Working\NT*.idl, *.cpp, *.java). Java versions of the key and copy helper, for use by Java client applications, are generated into subdirectories with names based on the module names of the key or copy helper (for example, MyProject\Working\NT\ClaimModuleCopy\ClaimCopyHelper.java).

You can select which platforms you generate code for using the **Platforms - Generate** menu on the Object Builder main menu bar. You can also select which platform to view information for, and constrain your development options to a particular set of platforms. You can only view one platform at a time, but you can generate code for multiple platforms at a time.

© Copyright IBM Corp. 1997, 1998

You can generate source code for any object in the **User-Defined Business Objects** folder, **User-Defined Data Objects** folder, **DBA-Defined Schemas** folder, and **User-Defined Compositions** folder. To generate code for an object, follow these steps:

1. Select an object.
2. From the object's pop-up menu, click **Generate - Selected - All**. The appropriate code for the object is generated into the working directory. You can also select to generate only a particular type of code, from the **Selected** choices. These choices display the list of file types that can be generated for the selected object (for example, .ih, .cpp, .java)

Note: Because a business object interface is physically contained in a business object file, you generate the code for the interface by generating the code for the file (from the business object file's pop-up menu, click **Generate - Selected**). The same applies to data object interfaces in the User-Defined Data Objects folder.

You can generate the code for all the objects in a folder by selecting **Generate - All** from the folder's pop-up menu.

The generation process is tracked by a progress indicator, and may take some time. The more platforms you are generating code for, the longer the generation process will take.

To view the source code for any of the objects you defined, select **View Source** from the object's pop-up menu. The .idl, .ih, and .cpp or .java files for the object are loaded in the editor pane. Click the drop-down arrow on the right end of the editor pane's title bar to access a list of currently loaded files and switch between them. You cannot edit the source code directly: if you want to change the source code, do so by changing the selections in the wizards, or editing the code associated with your methods in the Methods pane. The next time you generate the source code, your changes are applied.

Note: Outside of Object Builder, you can edit the source code with the editor of your choice. Changes to method bodies should be imported back into Object Builder, or your changes will be over-written the next time code is generated.

You can now generate the makefiles that will set your build options and define your target DLLs.

RELATED CONCEPTS
"Chapter 7. Multi-Platform Development" on page 187

RELATED TASKS
"Import Changes to Methods" on page 272
"Generate a Makefile" on page 367
"Build the DLLs" on page 368
"Run Object Builder in Batch Mode" on page 11

RELATED REFERENCES
Objects to Source Files Mapping

Define a Client DLL

Your application will typically consist of both client and server shared libraries, or dynamic link libraries (DLLs). To define a client DLL, follow these steps:

1. Under Tasks and Objects, select the **Build Configuration** folder.
2. From the pop-up menu of the folder, select **Add Client DLL**. The Client DLL wizard opens to the Name and Options Page.
3. Type a name for the configuration. This is a unique identifier for the build configuration that creates the DLL. If you want, you can also type a description of the configuration.
4. Set the platforms for which you want to build DLLs (**Applicable Platforms**).
5. Set the options for each platform:
   a. Select a platform from the **Platforms** list. All the options you enter below will apply to the DLL built for this platform.
   b. Type a name for the library (DLL), without the file extension.

      Note the following points:
      - You cannot have spaces in the DLL file name. When you click **Finish** to close the wizard, the program strips out any spaces. It also removes the file extension, if you happened to include it.
      - If you do not specify a file name, the name of the configuration will be used (with a .dll extension).
      - **390**: The file name cannot exceed 8 characters.
   c. In the **Make Options** field, type any options you want to call the DLL's makefile with. The options are added to the all.mak file that calls the DLL makefiles.

      There are several options specific to Component Broker that you can enter in this field:
      - DEBUG=1 (page 370)
      - IVB_TRACE_DEBUG=1 (page 370) (not appropriate for a client DLL)
      - IVB_UNOPTIMIZE=1 (page 370)
      - IVB_DYNAMIC_LINK=1 (page 371)
      - IVB_BUILD_VERBOSE=1 (page 371)
      - all (page 371)
      - java (page 371)
      - jcb (page 371)
      - cpp (page 371)
   d. In the **IDL Compile Options**, **IOM Java Compile Options**, **JCB Java Compile Options**, and **CPP Compile Options** fields, specify any options you want passed to the IDL, Java, and C++ compilers by the makefile.
   e. In the **Link Options** field, specify any linker options you want to build the DLL with.
   f. Also enter any non-Object Builder user-defined libraries for any DLLs that are referenced by this DLL.
6. Click **Next**. The Client Source Files Page opens.
7. Select the files you want to use as source for the DLL. Only files that are candidates for a client DLL (for example, key and copy interfaces) are available for selection.

   If you are building a composition or a composite component, you need to include the client interface files of the member components in the composition (business object file, key file, and copy helper file).
8. Click **Next**. The Libraries to Link With Page opens.

9. Select the names of the import libraries for any other DLLs you have defined in Object Builder that are referenced by this DLL.

   For example, if this DLL contains a child interface whose parent is defined in another DLL, you need to select the import library for the parent's DLL here.
10. Click **Finish**. The client DLL object appears in the Build Configuration folder and you are ready to generate the makefile that will build it.

RELATED TASKS
"Define a Server DLL"
"Generate a Makefile" on page 367

Define a Server DLL

Your application will typically consist of both client and server shared libraries, or dynamic link libraries (DLLs). To define a server DLL, follow these steps:

1. Under Tasks and Objects, select the **Build Configuration** folder.
2. From the pop-up menu of the folder, select **Add Server DLL**. The Server DLL wizard opens to the Name and Options Page.
3. Type a name for the configuration. This is a unique identifier for the build configuration that creates the DLL. If you want, you can also type a description of the configuration.
4. Set the platforms for which you want to build DLLs (**Applicable Platforms**).
5. Set the options for each platform:
   a. Select a platform from the **Platforms** list. All the options you enter below will apply to the DLL built for this platform.
   b. Type a name for the library (DLL), without the file extension.

      Notes:
      - You cannot have spaces in the DLL file name. When you click **Finish** to close the wizard, the program strips out any spaces. It also removes the file extension, if you happened to include it.
      - If you do not specify a file name, the name of the configuration will be used (with a .dll extension).
      - **390**: The file name cannot exceed 8 characters.
   c. In the **Make Options** field, type any options you want to call the DLL's makefile with. The options are added to the all.mak file that calls the DLL makefiles.

      There are several options specific to Component Broker that you can enter in this field:
      - DEBUG=1 (page 370)
      - IVB_TRACE_DEBUG=1 (page 370)
      - IVB_UNOPTIMIZE=1 (page 370)
      - IVB_DYNAMIC_LINK=1 (page 371)
      - IVB_BUILD_VERBOSE=1 (page 371)
      - all (page 371)
      - java (page 371)
      - cpp (page 371)

d. In the **IDL Compile Options**, **IOM Java Compile Options**, **JCB Java Compile Options**, and **CPP Compile Options** fields, specify any options you want passed to the IDL, Java, and C++ compilers by the makefile.
e. In the **Link Options** field, specify any linker options you want to build the DLL with.
f. Also enter any non-Object Builder user-defined libraries for any DLLs that are referenced by this DLL.

6. Click **Next**. The Server Source Files Page opens.
7. Select the IDL files you want to use as source for the DLL. Only files that are candidates for a server DLL (for example, business object implementations and managed objects) are available for selection.

   When you select the source file for a data object implementation, the source files for its associated persistent objects are automatically included.
8. Click **Next**. The Libraries to Link With Page opens.
9. Select the name of the import library (.lib file) for the corresponding client DLL. Also select the names of the import libraries for any other DLLs you have defined in Object Builder that are referenced by this DLL.

   For example, if this DLL contains a child interface whose parent is defined in another DLL, you need to select the import library for the parent's DLL here.
10. Click **Finish**. The server DLL object appears in the Build Configuration folder. You are now ready to generate the makefile that will build it.

RELATED TASKS
"Define a Client DLL" on page 364
"Generate a Makefile"

Generate a Makefile

To generate the makefiles that will build the shared libraries or dynamic link libraries (DLLs) in the Build Configuration folder, follow these steps:
1. Select the Build Configuration folder.
2. From the folder's pop-up menu, select one of the options under **Generate - All** (as described below). The makefiles (all.mak, and the makefiles for the DLLs in the folder) are generated into your working directory.

Once the makefiles have been generated, you can view them by clicking **View Source** from the folder's pop-up menu.

When you generate from the folder's pop-up menu, the option you select under **Generate - All** determines what is included in the makefiles. You can select from the following:

- **C++ Default Targets**
  The makefile will build all C++ DLLs.
- **Java Default Targets**
  The makefile will build all Java JAR files.
  AIX Not available on AIX.
- **Java Client Bindings Default Targets**
  The makefile will build all Java client bindings
  AIX Not available on AIX.
- **All Targets**
  The makefile will build all DLLs and associated objects defined in the folder.

The menu item you select determines what the **Build - Default Targets** action will build. The makefile you generate can still be used to build other targets, through the folder pop-up menu's **Build** actions.

The makefile for each DLL includes any IDL compile, Java compile, CPP compile, and link options you specified for the DLL. Do **not** use these files directly. Use the all.mak file, which calls the makefiles for each DLL, and includes any make options you specified for each DLL. Using the all.mak file ensures that the DLLs are built in the correct order.

Notes:

- If an interface defined using an Object Builder wizard or imported from an .idl file "includes" other interfaces, the "included" interface or header files *does not appear in the makefile as dependencies* of the "including" interface.

  Prior to re-building the generated source, you should either manually edit the makefile to add the missing dependencies, or clean and rebuild all targets.
- You should build the DLLs on a server development machine (typically, the one on which you are using Object Builder). If you move the makefiles to another machine without the server SDK installed, the DLLs may not compile.

RELATED TASKS

"Define a Client DLL" on page 364
"Define a Server DLL" on page 366

Build the DLLs

Before packaging an application, you must build your client and server DLLs (that is, compile and link the generated code). You can do this by running the **all.mak** file you generated. Do **not** run the makefiles for the individual DLLs directly. Using all.mak ensures that the DLLs are built in the correct order.

You should build the DLLs on a server development machine (typically, the one on which you are using Object Builder). If you move the makefiles to another machine without the server SDK installed, the DLLs may not compile.

If you are building for OS/390, you can use the OS/390 remote build process to build on a specified remote host.

To run all.mak, follow these steps:

1. Under Tasks and Objects, select the **Build Configuration** folder.
2. From the folder's pop-up menu, select **Build** and then one of the following sub-options:
   - **Out-of-Date Targets**
     You can select the type of out-of-date targets to build:
     - **C++**
       Builds C++ client and server DLLs.
     - **Java**
       Builds Java JAR files for Java business objects and for components with PA-based persistence.
     - **Java Client Bindings**
       Builds Java client bindings that allow a Java client application to access the equivalent components in the server application.

- **Default**
  Builds whatever was selected when the makefile was generated (for example, if **Generate - All - C++ Default Targets** was used to generate the makefiles, then selecting **Build - Out-of-Date Targets - Default** will build the C++ targets).
- **All Targets**
  All targets are built.
- **Rebuild All Targets**
  A build clean is performed, followed by a build all targets.
- **Clean**
  Performs a build clean, but does not perform a build.

3. When the build has finished, you can review the record of the build in the command window.

You can also make all.mak from a command line, with the following flags:

- all (page 371)
- java (page 371)
- cpp (page 371)
- jcb (page 371)

Once you have built all applicable targets, your DLLs and .jar files exist on your hard drive, in the project working directory defined in Object Builder, and you can package them into an application.

For C++ components, the DLLs (*MyClientDLL*.dll and *MyServerDLL*.dll) are built with the file name you specify and placed in the project working directory. If you have Java components in the same application, then you will need a .jar file for each C++ component that provides Java components on the server with access to the C++ components: *MyClientDLL*.jar.

If you are supporting a Java client application, then Java client bindings also need to be built: JCB*MyClientDLL*.jar, in the working*platform*\JCB\ directory.

For Java components, three .jar files are created:

- *MyClientDLL*.jar
  Supports access to the component by other components on the server.
  - Source: \Working*platform*\
  - Compiled classes: \Working*platform*\JAVACLS*MyClientDLL*\
  - JAR file location: \Working*platform*\
- *MyServerDLL*.jar
  Supports and implements the Java business object on the server.
  - Source: \Working*platform*\
  - Compiled classes: \Working*platform*\JAVACLS*MyServerDLL*\
  - JAR file location: \Working*platform*\
- JCB\JCB*MyClientDLL*.jar
  Java client bindings that support access to the component by the client application.
  - Source: \Working*platform*\JCB\
  - Compiled classes: \Working*platform*\JCBCLS*MyClientDLL*\
  - JAR file location: \Working*platform*\JCB\

The DLLs and JAR files are automatically pulled into an application when you configure the managed object with the application.

RELATED CONCEPTS

"Chapter 13. Troubleshooting" on page 411

RELATED TASKS

"Launch a Remote OS/390 Build" on page 373
"Generate a Makefile" on page 367
"Package an Application" on page 375

Build Configuration Options

When you configure the build process for your application, you can specify options for the way your DLLs are compiled and linked in the Client DLL wizard and Server DLL wizard, on the Name and Options Page.

You can either set options directly (using their command-line syntax, in the **Compile Options** and **Link Options** fields), or set them through the make command (by specifying macros in the **Options for Make** field). If you intend to use the same Object Builder model on both Windows NT and AIX, we recommend using the macros. The macros will set the appropriate compile and link options for the specified build type, regardless of the current platform.

The predefined macros provided by Object Builder function as follows:

DEBUG=1
The DLLs are compiled without optimization, and enabled for source-level debugging. Options set:

Windows NT
CPP Compile:

- /O-
- /Ti+
- /Tm+

IVB_TRACE_DEBUG=1
Same as DEBUG=1, but also defines the CBS_TRACE_DEBUG preprocessor macro, which then includes code that allows the DLL to send trace data to the Object Level Trace tool for remote debugging. Options set:

Windows NT
CPP Compile:

- /O-
- /Ti+
- /Tm+
- /DCBS_TRACE_DEBUG

IVB_UNOPTIMIZE=1
The DLLs are compiled without optimization. Option set:

Windows NT
CPP Compile:

- /O-

IVB_DYNAMIC_LINK=1
The DLLs are linked dynamically with the VisualAge for C++ runtime DLLs. This reduces the size of your DLLs, but makes them dependent on the presence of the VisualAge for C++ DLLs. You must then package the runtime DLLs with your application, as follows (procedure applies to Windows NT):

1. Use the DLLRNAME utility to rename the VisualAge for C++ DLLs and update the references to them in your executables. See *Packaging the VisualAge for C++ Runtime DLLs* in the *VisualAge for C++ for Windows User's Guide*.
2. When you define the application, include the renamed DLLs on the Additional Executables Page of the Add Application wizard.

AIX For AIX, consult the C Set++ documentation for information on packaging shared libraries.

The default is to link to the runtime DLLs statically, which means any necessary code is built directly into your DLLs and there are no special packaging concerns. In most cases, the default (static linkage) is preferable.

Option set:

Windows NT
CPP Compile:

- /Gd+

IVB_BUILD_VERBOSE=1
The DLLs are compiled and linked with the maximum amount of feedback generated. Options set:

Windows NT
CPP Compile:

- /Q- (actually the Q+ option is just removed)

Link:

- /VERBOSE

all
Builds IDL, C++, and Java
AIX On AIX, Builds IDL and C++ only.

java
Builds IDL and Java.
AIX Not available on AIX.

jcb
Builds Java client bindings.
AIX Not available on AIX.

cpp
Builds IDL and C++. This is the default behavior (when make is run without flags, or run directly from Object Builder).

RELATED TASKS
"Define a Client DLL" on page 364
"Define a Server DLL" on page 366
"Generate a Makefile" on page 367
"Build the DLLs" on page 368

Remote Build Configuration (OS/390) Remote Build

Remote Build

A build that is activated on another computer that is distant from a central site, usually over a network connection. The remote computer may be stationary and non-portable, or it may be portable.

RELATED CONCEPTS
"Profile""Pass Ticket"

RELATED TASKS
"Launch a Remote OS/390 Build" on page 373
Launch a Remote OS/390 Build - Scenario

Pass Ticket

In Resource Access Control Facility (RACF) secured sign-on, and for the OS/390 secure server, a pass ticket is a dynamically generated, random, one-time-use, password substitute that a workstation or other client can use to sign on to the host rather than sending a RACF password across the network.

This pass ticket is composed of 8 characters, which can be any of the letters A to Z, and the digits 0 to 9.

A pass ticket can be used only once in the ten-minute period from its generation. It acts as a secure bridge from legacy applications to the modern world, though it is not as secure as digital certificates.

RELATED CONCEPTS
"Remote Build"
"Profile"

RELATED TASKS
"Launch a Remote OS/390 Build" on page 373
Launch a Remote OS/390 Build - Scenario

Profile

In Object Builder, when you are specifying the options for a remote OS/390 build, you can optionally specify the name of a profile file. This is a shell file that contains initializations of the OS/390 environment variables.

RELATED CONCEPTS
"Remote Build"
"Pass Ticket"

RELATED TASKS
"Launch a Remote OS/390 Build"
Launch a Remote OS/390 Build - Scenario

Launch a Remote OS/390 Build

Preliminary steps:

- Ensure that the rexec daemon is running on the OS/390 host machine.
- Have Object Builder for Windows NT up and running.
- Change the platform view to OS/390. Select **Platform - View - 390** from Object Builder's main menu.

To launch a remote build, follow these steps:

1. Select the Build Configuration folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Remote OS/390 Options**. The Remote Build wizard opens to the OS/390 Options Page.
3. Specify the name of the OS/390 machine on which you want to run the remote build in the**Host Name** field.
4. Type the user ID and password by which you will access the host machine.
5. Type the full directory path on the OS/390 host machine, which is to contain the files generated by Object Builder in the Host Directory field. This is the directory that will contain the files and directories that are normally contained in Object Builder's Working\390 directory after file generation.
6. You can optionally specify the name of a shell profile file, to be used to initialize environment variables.
7. Indicate the format in which data is to be returned by the host. You can choose between the American Standard Code for Information Interchange (ASCII) and the Extended Binary Coded Decimal Interchange Code (EBCDIC). Your selection determines the data translations, if any, that are required.
8. Click **Finish**.
   Your user ID and password are stored in memory, but the host name, host directory, and profile name (if you provide it) are saved.
9. Select the Build Configuration folder again, and from its pop-up menu, select**Build**.
   The remote build will be activated if the host name, user ID, password, and host directory are properly set.
   Note: As long you use the same Object Builder session, you do not have to retype your user ID and password each time you want to do a remote build; you can just execute step 9. You will have to follow the preliminary steps, and steps 1 to 4, 6, 8 and 9, if you close Object Builder, and restart it.

RELATED CONCEPTS
"Remote Build" on page 372
RELATED TASKS
Launch a Remote OS/390 Build - Scenario

Launch a Remote OS/390 Build - Scenario

Preliminary steps:

- You must ensure that the rexec daemon is running on the OS/390 host machine.

- Optionally, create an NFS read/write mount of your OS/390 host directory. You can then generate code directly into the NFS mounted directory. Instead, you can generate the files onto your local file system, and then use the File Transfer Protocol (FTP) to transfer them over to an Open Edition for OS/390 system.
- Set up Object Builder for Windows NT.
- Once it is running, change the platform view to OS/390. Select **Platform - View - 390** from Object Builder's main menu.

To launch a remote build, follow these steps:

1. Select the Build Configuration folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Remote OS/390 Options**. The Remote Build wizard opens to the OS/390 Options Page.
3. Specify machine.host.com as the name of the OS/390 machine on which you want to run the remote build in the **Host Name** field.
4. Type the user ID and password by which you will access the host machine.
5. Type .../Working/390 as the full directory path on the OS/390 host machine, which is to contain the files generated by Object Builder in the Host Directory field. This is the directory that will contain the files and directories that are normally contained in Object Builder's Working\390 directory after file generation.
6. If you maintain the settings of the Component Broker Toolkit environment variables such as CLASSPATH, PATH, and so on in a shell profile file similar to .profile in AIX, specify its name.
7. Accept the default (ASCII - the American Standard Characters for Information Interchange) format in which data is to be returned by the host. The other choice is the Extended Binary Character Digital Interchange Code (EBCDIC). Your selection determines the data translations, if any, that are required.
8. Click **Finish**.
   Your user ID and password are stored in memory, but the host name, host directory, and profile name are saved.
9. Select the Build Configuration folder again, and from its pop-up menu, select**Build**.
   The remote build will be activated if the host name, user ID, password, and host directory are properly set.
   Note: As long you use the same Object Builder session, you do not have to retype your user ID and password each time you want to do a remote build; you can just execute step 9. You will have to follow the preliminary steps, and steps 1 to 4, 6, 8 and 9, if you close Object Builder, and restart it.

You can now make incremental changes to your files, and you will not have to use the File Transfer Protocol.

RELATED CONCEPTS

"Remote Build" on page 372

RELATED TASKS

"Launch a Remote OS/390 Build" on page 373

Package an Application

When your application is ready to ship, you can package it for easy installation at a customer site. Typically you will create an application family for your server applications, and a separate application family to hold your client applications. Server applications consist of managed object configurations, which define the component objects and DLLs you want installed on the server. Client applications consist of the client DLLs for your components, plus the client EXEs (built outside of Object Builder) that will access the components.

To package an application, follow these steps:

1. "Create an Application Family"
2. "Add a Client Application" on page 376
3. "Add a Server Application" on page 377
4. "Create a Container Instance" on page 346
5. "Configure a Managed Object" on page 377
6. "Generate the Install Image" on page 379

RELATED CONCEPTS
"DDL" on page 114

RELATED TASKS
Develop Applications in Object Builder - Overview

Create an Application Family

An application family consists of one or more applications that are packaged together on a CD and need to run at the same code level. There is a single installation process for each application family you define. You can group applications in a family to ensure version compatibility. The installation checks each application's version, and at the end of the installation ensures that all applications in the family are at the same version.

When you install an application family, you cannot select which applications you want to install. You must install all or none of the applications in the family.

Application families consist of either client applications (which include an EXE file and one or more DLLs used to access the server applications), or server applications (which include DLLs, defined in Object Builder, that contain the components accessed by the client application).

To create an application family, follow these steps:

1. Under Tasks and Objects, select the Application Configuration folder.
2. From the folder's pop-up menu, select **Add Application Family**. The Add Application Family wizard opens to the Name Page.
3. Enter a name, description, and version number for the application family.
4. Click **Next**. The Installation Information Page appears.
5. Specify if any additional disk space is required by the application family. By default, the InstallShield program calculates the disk space needed to install your applications based on the size of the disk image. If your applications

require additional disk space (for example, to store temporary files your application generates when it runs), type the additional amount here.
6. Find the Readme file, if you have one. The Readme file could document any hardware and software prerequisites for the application, and troubleshooting and recovery information for the installation.
7. Click **Finish**. The application family is added to the Application Configuration folder.

You can now add applications to the application family.

RELATED TASKS
"Add a Server Application" on page 377
"Add a Client Application"

Add a Client Application

An application is a complete, self-contained program that performs a specific function for a user. In Object Builder terms, a client application consists of an EXE file, and one or more client DLLs defined in Object Builder that define the component interfaces the client application can access. The client application then works with the server applications (which provide DLLs that contain the components the client uses). For Java applications, .jar files serve the same function as DLLs.

Do not add client and server applications to the same application family. Because you cannot selectively install within an application family, grouping the two together would mean you could not perform a client-only installation.

To add an application to your application family, follow these steps:
1. From the Application Configuration folder, select your application family.
2. From the family's pop-up menu, select **Add Application**. The Add Application wizard opens to the Name and Environment Page.
3. Enter a name, description, and version number for the application.

   You can also specify a Java virtual machine name.
4. Click **Next**. The Additional Executables Page opens.
5. Browse for and select the following files:
   - For Java client applications, the Java .jar files that contain your client application, and any additional .jar files your application requires.
   - For C++ applications, the client EXE file (which you created outside of Object Builder) or Java files, and any supporting DLLs it requires.
   - Any client DLL or .jar files (defined in Object Builder) that contain the definitions of components your client application uses.
6. Click **Finish**. The application appears under your application family in the Application Configuration folder.

Note: Do **not** configure any managed objects with a client application, or it will be installed incorrectly. Managed objects can only be configured with server applications, and accessed from client applications through the client DLLs that define the component's client interfaces.

RELATED TASKS
"Add a Server Application" on page 377
"Define a Client DLL" on page 364

Add a Server Application

A server application consists of components, which encapsulate distributed data and resources for the use of a client application.

Do not add client and server applications to the same application family. Because you cannot selectively install within an application family, grouping the two together would mean you could not perform a client-only installation.

To add a server application to your application family, follow these steps:

1. From the Application Configuration folder, select your application family.
2. From the family's pop-up menu, select **Add Application**. The Add Application wizard opens to the Name and Environment Page.
3. Enter a name, description, and version number for the application.

   You can also specify a Java virtual machine name.
4. Click **Next**. The Additional Executables Page opens.
5. Browse for and select the following files:
   - Any client DLL or .jar files (defined in Object Builder) for components in other application families that are referenced by your application.

     DLLs and .jar files for components in this application are automatically included when you configure the component managed objects with the application.
   - Any additional DLLs or .jar files (not defined in Object Builder) that contain code required by your application.
   - Any bind files for components that use embedded SQL (that is, the component's data object implementation has the **Embedded SQL** option set on the Behavior Page of its wizard).

     Bind files are the compiled form of a persistent object .sqx file (for example, ClaimPO.sqx becomes ClaimPO.bnd).
   - Any SQL files for components that connect to new (as opposed to pre-existing) database tables.

     When the server application is installed, the SQL files can be used to configure the database for use by the application's components.
6. Click **Finish**. The application appears under your application family in the Application Configuration folder.

You can now configure managed objects with your application and, if you want, create a container that handles object services for the managed objects.

RELATED TASKS

"Add a Client Application" on page 376
"Create a Container Instance" on page 346
"Configure a Managed Object"

Configure a Managed Object

Once you have defined a server application, you can add and configure the managed objects you want your application to consist of.

To add a managed object to an application, follow these steps:

1. From the Application Configuration folder, select your application.

2. From the application's pop-up menu, select **Add Managed Object**. The Configure Managed Object wizard opens to the Selection Page.
3. Select the managed object from the drop-down list.

   If the managed object has been added to a DLL, and is associated with a key and a copy helper, then the primary key, copy helper, and DLL fields are filled in for you. You can type over these automatic selections, or make alternative selections from the drop-down lists.
4. Click **Next**. The Data Object Implementations Page opens.
5. From the Implementations pop-up menu, select **Add**.
6. Select the data object implementations that will be available to the application, and associated DLLs. Note that this is a packaging statement, and not a configuration statement.

   You can only select data object implementations whose type of persistence matches the service provided by the managed object (transactional services for DB persistence, session services for PA persistence).
7. Click **Next**. The Container Page opens.
8. Specify whether you want to use a workload managing container. If you check this option, then only workload managing containers are available in the Container list.
9. Select the container to use with this managed object. The container determines the quality of service (that is, how objects are instantiated, terminated, and so on). If you select a workload managing container, then the component will be workload managed.

   The only containers listed are those that are appropriate for the current managed object and selected data object implementations.

   390: If you are developing an application intended for deployment on OS/390 (the **Platform - Constrain - 390** menu choice is checked), then all containers are listed, and you need to make an appropriate choice based on the kind of managed object you are configuring, and the services it requires. The rest of the container definition is handled through the System Management user interface.

   Note: Make sure that the managed object is configured with a different container than that used by its home. If necessary, create a separate container instance for the managed object. If a managed object and its home are configured with the same container, the server will not activate.
10. Click **Next**. The Home Page page appears.
11. Define the home to use with this managed object. You can define a home instance of a default home provided with Component Broker, or define a home instance of a customized home you created. If you specify a customized home, you must also specify which DLL contains it.
12. Select any other configuration options for the home
13. Click **Finish**. You have configured the managed object by choosing a copy helper and a key for it to work with, data object implementations for it to use, a container, a home, and the DLLs that contain it and the other objects. The managed object now appears in the Application Configuration folder, underneath the application you configured it for.

Once you have finished adding managed objects to your server applications, and have completed the configuration of the applications in your application family, you can generate the installation image for your application family.

RELATED CONCEPTS
“Home” on page 342
“Container” on page 345
“Data Object” on page 18
Naming Service
Life Cycle Service
Workload Management

RELATED TASKS
“Create a Container Instance” on page 346
“Create a Customized Home” on page 343
“Add a Server Application” on page 377
“Generate the Install Image”
“Work with Managed Objects - Overview” on page 339

Edit a Managed Object Configuration

To edit a managed object configuration, follow these steps:

1. Locate the managed object configuration in the Application Configuration folder.
2. From the pop-up menu of the configuration, click **Properties** to open the Managed Object Configuration wizard.
3. Click the title bar to display the contents of the guide, and turn to a particular page.
4. Make your changes.
5. Click **Finish**.

RELATED CONCEPTS
“Managed Object” on page 22

RELATED TASKS
“Work with Managed Objects - Overview” on page 339

Delete a Managed Object Configuration

To delete a managed object configuration, click **Delete** from its pop-up menu.

Note: If the managed object configuration is part of a customized home, then you must first remove it from any other managed object configurations that use it as their home.

RELATED CONCEPTS
“Managed Object” on page 22

RELATED TASKS
“Work with Managed Objects - Overview” on page 339
“Work with Customized Homes - Overview” on page 342

Generate the Install Image

Once you have created an application family, added applications to the family and configured your managed objects, you can generate an install image to burn onto a CD-ROM. The install image includes the DDL file that defines your data object to the server, and an InstallShield setup file that starts the install from the CD-ROM.

Before you can generate the install image you must have installed the following products:

- Component Broker base services
- InstallShield

To generate the install image, follow these steps:

1. From the Application Configuration folder, select your application family.
2. From the family's pop-up menu, select **Generate**.

   If you have multiple application families, you can generate the installation scripts for all of them at once. Select **Generate - All** from the pop-up menu of the Application Configuration folder to generate images for all the families in the folder. You will still need to build the image for each application family individually.

   The following files are generated and placed in your working directory, under a subdirectory that has the same name as the application family (for example, myProject\Working\MyApplicationFamily\):

   - <AppFamilyName>.ddl
     The DDL script that provides information about your application family to System Manager. Editable with the DDL Editor tool.
   - <AppFamilyName>.auto.ddl
     A backup version of the generated DDL.
   - setup.rul, setupmsg.h, setup.lst
     InstallShield scripts used by build.bat.
     AIX Not generated for AIX.
   - build.bat
     Builds the install image.
     AIX Not generated for AIX.
   - AIX build.sh
     Builds the install image, on AIX.
3. Again select your application family.
4. From the family's pop-up menu, select **Build**. This runs the build, and generates an install image in the following format:

   Windows NT
   An install image is created as the contents of a directory called **Disk1**, located in a subdirectory below the build file (for example, myProject\Working\MyApplicationFamily\Disk1\).

   You can test the install image by changing to the image directory and typing **setup**.

   AIX
   An install image is created in the AIX backup file format, with the name of the application family and the extension .bff (for example, myProject\Working\MyApplicationFamily\myApplicationFamily.bff).

   Use the smit utility on AIX to install the image on a server.

   390
   If you have developed your code for OS/390 (as specified in the **Platform** menu), the generated DDL for the application family includes the statement:

   ```
   targetplatform="390"
   ```

   This statement prevents the application family from being accidentally installed on an incompatible System Management platform.
5. Burn the contents of the directory onto a CD-ROM.

The CD-ROM contains everything necessary to install the product. Once the application is installed, you can configure it with system management, and run it to make the components available to client applications.

Note: Before you run a Java client application, you need to add the following JAR files to the beginning of your classpath:

- somojor.zip
  Contains classes to support the client-side Java ORB. If this is not at the beginning of the classpath, the wrong classes will be found, and your application will not run.
- The JAR files that contain your Java client bindings (located in the JCB subdirectory, with the naming convention JCB*MyObject*C.jar). These contain classes to support a client application accessing the equivalent Java component on the server.

RELATED CONCEPTS
"DDL" on page 114

RELATED TASKS
"Create an Application Family" on page 375
"Add a Server Application" on page 377
"Configure a Managed Object" on page 377
Edit an Application DDL File
Install and Configure a New Application

Application DDL Files

The installation package for an application family contains a **DDL file** that describes the contents of the application family. It describes the applications in the family and the objects, attributes, and relationships that make up each application. For example, the DDL file for an application family defines the following:

- The applications to run on servers and their relationships to objects that they provide
- The applications to run on clients and their relationships to objects that they provide
- The classes, DLLs, homes, containers, and other objects provided by the applications, and appropriate relationships between such objects
- Appropriate attributes of the applications and other objects in the application family

The application family installation program uses the information in the DDL file to create *Install objects* that the System Manager can use to define and configure the applications.

When you use Object Builder to create an application family, it generates a DDL file for the application family. Before you generate the install image for an application family package, you can add other objects to the DDL file.

You do not normally change DDL files after the application family has been installed into Component Broker. When you load an application family into Component Broker, each application in the DDL file is represented as an **available application** through the System Manager user interface. If you need to customize the

application within Component Broker, you normally do so by changing model objects for the application through the System Manager user interface.

RELATED CONCEPTS
"The DDL Editor"
The files and process used by the DDL Editor (page 383)

RELATED TASKS
Edit a DDL File

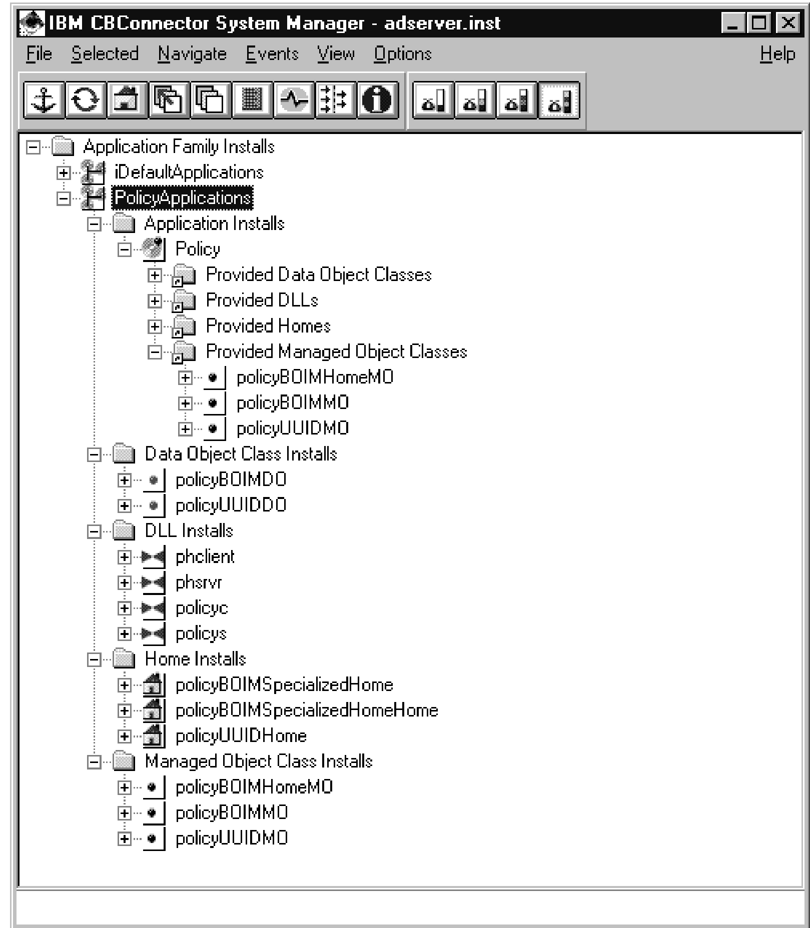
The DDL Editor

The DDL Editor can be used to display and edit the objects, object attributes and relationships in a DDL file. The DDL Editor is used after a DDL file has been created by the CBToolkit Object Builder to complete the DDL file before adding it to an application installation package.

Editing DDL files is not usually of interest for system administration. Configuration tasks are normally performed on model objects within Configurations of your application Management Zones.

The DDL Editor uses a version of the standard System Manager user interface to display the objects defined in the DDL file, as shown below. Through the DDL Editor you can add new objects to your application family in the DDL file, edit the objects in your application family, create relationships between the objects, and change existing relationships.

The DDL Editor window



An overview of the files and process used by the DDL Editor
For an application ddl file called *filename.ddl*, the Object Builder creates the following ddl files, used as input to the DDL Editor (as shown in **Files used by the DDL Editor**):

filename.auto.ddl
: This is used as a reference so that the DDL Editor can tell what objects and relationships have been generated automatically.

filename.ddl
: This is the ddl file that is to be used for an application, and which you want to edit. Initially this file is identical to the *filename.auto.ddl* file, but the DDL Editor overwites the *filename.ddl* file when you choose to save the changes that you have made. The *filename.ddl* output by the DDL Editor includes all the changes made to objects, attributes, and relationships by editing the input *filename.ddl*.

If you save any changes that you have made to the ddl file, This creates the following files for an edited ddl file:

filename.additions.ddl
: This contains the extra objects and relationships that you added to the edited ddl file called *filename.ddl*

filename.ddl
: This contains the complete new ddl file, including the objects and relationships from the edited ddl file and the extra objects and relationships

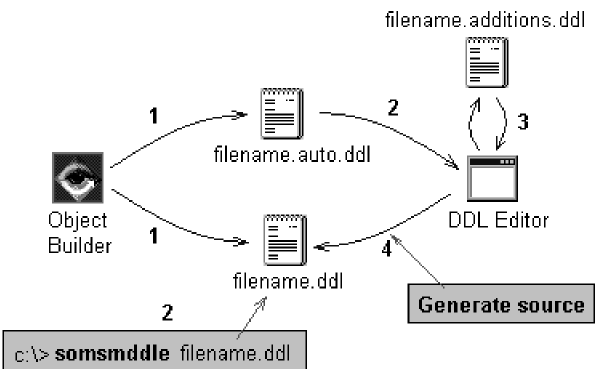
also stored in the *filename.ddl.additions* file. This file overwrites the original version specified on the command used to start the DDL Editor.

If you edit the *filename.ddl* again, the *filename.additions.ddl* file is used to reapply the changes that you have made using the DDL Editor on previous occasions. Each time you use the DDL Editor to edit the same original ddl file, the *filename.additions.ddl* and *filename.ddl* files are overwritten, but the original *filename.auto.ddl* file is not affected.

The *filename.additions.ddl* file forms a cumulative log of additions that you have made to the original ddl file. Therefore, keep that file until you have completed editing the original ddl file (until you no longer need to edit the original ddl file again with the changes that you have made previously).

If the Object Builder outputs the original *filename.ddl* filename again, you should edit that file again and regenerate it to ensure that the changes are included from the *filename.auto.ddl*. Do this even if you do not want to change the ddl file immediately. You should also check the application families displayed by the DDL Editor, to see if the changes made by the Object Builder have any side effects on the changes that you had made to the ddl file.

The files and process used by the DDL Editor



Tips
When a DDL file *filename.ddl* is first edited, the *filename.auto.ddl* file is identical. If you do not have a *filename.auto.ddl* file, you can copy and rename the DDL file to be edited. If you later need to recreate the original DDL file to be edited, without regenerating it from Object Builder, you can copy and rename the *filename.auto.ddl* file. (Normally, you would use Object Builder to regenerate the DDL file to be edited.)

RELATED CONCEPTS
“Application DDL Files” on page 381

RELATED TASKS
Edit a DDL File

Creating and Editing DDL Files

When you use Object Builder to create an application family it generates a DDL file for the application family. This generated DDL file is found in your working directory,

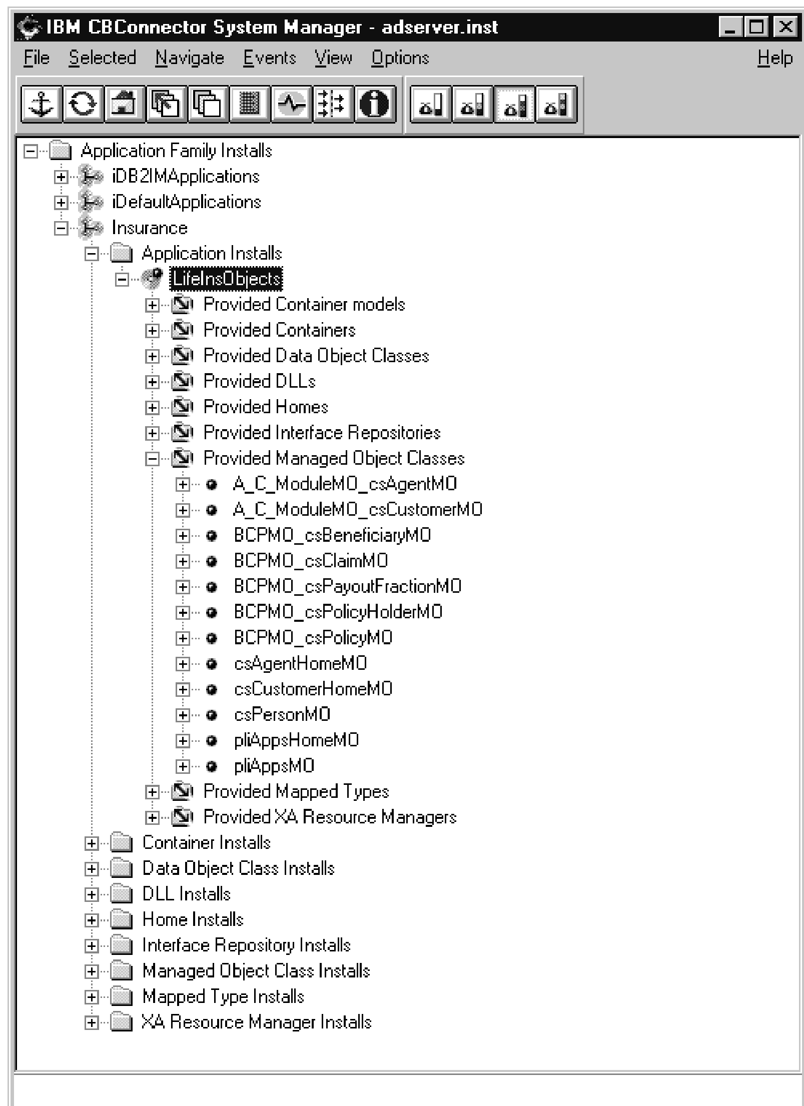
under a subdirectory that has the same name as the application family. The DDL file also has the same name as the application family.

When you use Object Builder to add objects to the application family package, it adds entries for those objects into the DDL file with appropriate attributes and relationships.

Before you generate the install image for an application family package, you can add other objects to the DDL file. Such objects are sometimes needed to configure special application functions; for example, when packaging an application family for a controlled server group, you can add policy groups, bind policies and their associated C++ classes. After you have completed your Object Builder output, if you need to add more to the DDL file you should use the **DDL Editor**.

The DDL Editor provides the same interface to the objects in the DDL file as the System Manager user interface provides to equivalent system management objects installed on your hosts. For each class of object in the DDL file, the DDL Editor presents you with only valid actions, attributes, and relationships. Therefore, you can only create valid types of objects, edit appropriate attributes, and create appropriate relationships between objects. Further, the DDL Editor provides normal graphical user interface actions for you to act on the DDL file, so preventing syntax errors and other problems associated with editing a DDL file directly.

The structure of a DDL file, as presented by the DDL Editor is shown in the following figure:



Objects in a DDL file
The main folder in a DDL file, **Application Family Installs**, contains the object for your application family and one or more other application families that are provided by Component Broker. *You should only change the contents and attributes of your application family. Do not change other application families provided by Component Broker.*

Within your application family object there is a range of folders for all the objects within that family. All these folders are at the same level within the application family object, and are displayed in alphabetical order. To find an object, expand the folder for your application family, then expand the folder for the object class.

Server applications are defined in the **Application Installs** folder. Each application contains relationships with the objects that it provides. Through the DDL Editor, these relationships are grouped into folders; for example, **Provided Managed Object Classes**, as shown in the above figure.

Any client applications are defined in the **Client Application Installs** folder. Each client application contains relationships with the objects that it provides.

Other objects in the application family are defined in the folder for the object class. For example, to see containers provided by an application, expand the **Provided Managed Object Classes** relationship folder or the **Managed Object Class Installs** folder (as shown above). You can edit the same objects from either folder.

If the folder for an object class is not visible, it is most likely empty. You can use the **View** menu bar option to change the filter to show empty folders. Also check that the user-level is set to **Expert**.

Each object contains appropriate relationships with other objects.

For more information about editing DDL files, see the task description in the topic Edit a DDL File.

RELATED CONCEPTS
"The DDL Editor" on page 382
The files and process used by the DDL Editor (page 383)

RELATED TASKS
Edit a DDL File

Edit an Application DDL File

Using the **DDL Editor**, you can add new objects to a DDL file, edit the objects in the file, create new relationships between the objects, and change existing relationships.

This topic gives a task overview of how to edit a DDL file. Specific instructions required by other tasks are given in related task topics and in information provided by applications.

The DDL Editor uses a version of the standard System Manager user interface, so to edit DDL files you act on objects and their relationships in the same way as you would using the System Manager user interface, as described in the related topics.

Prerequisites
You must have the files **defaultApplications.ddl**, **somdb2im.ddl**, and **ddleditor.dict** in the Component Broker data subdirectory; for example; on Windows NT, e:\cbroker\data. These files are normally stored there automatically when Component Broker is installed.

To edit a DDL file, complete the following steps:

1. Display the DDL Editor:
   a. Open a command line window
   b. On the command line, type the following command

      somsmddle *ddl_filename*

      where, *ddl_filename* is the name of the ddl file to be edited. If the file is not in the current directory, type the full pathname of the ddl file. If the pathname name includes directory names with blank spaces, enclose the pathname within quotes. For example, to edit the file **c:\Program Files\Component Broker\Data\Myapplication.ddl**, you could type the following command:

      somsmddle "c:\Program Files\Component Broker\Data\Myapplication.ddl"

2. Change the objects and relationships *for your application family* in the DDL file. *You cannot delete any objects or relationships input from the Object Builder*.
3. To save any changes that you have made to the DDL file, on the pop-up menu for the Application Family Install click on **Generate source**. This saves any changes that you have made to the *filename.additions.ddl* file and recreates the *filename.ddl* with the input objects and relationships combined with the changes you have made.
4. To exit the DDL Editor, click the **X** icon in the top right corner of the window or Press Alt+F4.

Note: You cannot delete objects or relationships generated by the Object Builder (the objects provided in the *filename.ddl* input to the DDL Editor).

You can edit a DDL file in the following ways:

To display objects and folders and move around the object tree view displayed by the DDL Editor, you use the same tools and actions as for the standard SM user interface. For example, the following are some of the methods that you can use to display and act on objects:

- To expand the tree structure, click on the + symbol next to a folder or object
- To display objects from the session history or your hotlist, click the history or hotlist icons on the Tool bar and select an entry from the window displayed
- To change objects through relationships, click on the pop-up menu of the relationship shortcut icons

Objects within the application family are grouped into object class folders, displayed in alphabetical order. To find an object, expand the folder for your application family, then expand the folder for the object class.

If the folder for an object class is not visible, it is most likely empty. You can use the **View** menu bar option to change the filter to show empty folders. Also check that the user-level is set to **Expert**.

When you have displayed the required object, you can act on it using its pop-up menu or select it to use the menu bar choices or Tool bar.

To create a new object, you insert the object into the folder for the object class, using either of the following methods:

- On the pop-up menu of the application family, click **New**, then select the class of object to be created. For example, click **New - Application Install**, to create a new Application Install object.
- Insert the new object directly into its folder, by completing the following steps:
  1. Display the object class folder
  2. On the folder’s pop-up menu, click **Insert**
  3. Type an appropriate name for the new object
  4. To create the object, click the **OK** button

Both methods display a dialog box for you to identify the new object. If you type a valid name, the new object is created in the folder. To display the object, and perhaps act on it, expand the folder.

To display and edit object attributes, on an object's pop-up menu click **Edit**. This displays the Object Editor window, which you can use to change the attributes of objects.

To delete an object that you have added to the DDL file, click **Delete** on the pop-up menu of the object.

To create new relationships, drag one object and drop it onto either another object or a relationship folder.

Drag and drop tasks involve two actions; clicking **Drag** for the object to be dragged, and clicking an appropriate context-sensitive action for the target object or relationship. The **Drag** action causes context-sensitive actions to be added to the pop-up menus of the target objects and relationships.

To change relationships, you can use either of the following procedures:

- Drag and drop a new object onto the relationship folder. This either creates a new relationship or replaces the existing relationship with the new dragged object or new target object.
- Delete the shortcut icon in a relationship folder. This deletes the relationship, but does not delete the object.

RELATED CONCEPTS

"Application DDL Files" on page 381
"The DDL Editor" on page 382
Features of the User Interface
User-Level Settings and Object-Level Filters

RELATED TASKS

Display Objects
Create Objects
Select and Deselect Objects
Act on Objects
Drag and Drop Objects
Edit Objects

The Structure of a DDL file

This topic describes the general internal structure of an application DDL file. It is intended as a review aid in case you need to look within a DDL file after it has been generated by Object Builder. Usually, if you need to look at objects in a DDL file, you should use the **DDL Editor**. If you need to edit a DDL file after it has been generated by Object Builder, you should also use the DDL Editor.

The structure of a DDL file, as presented by the DDL Editor, is shown and described in the topic Objects in a DDL file.

The DDL Editor provides the same interface to the objects in the DDL file as the System Manager user interface provides to equivalent system management objects installed on your hosts. For each class of object in the DDL file, the DDL Editor presents you with only valid actions, attributes, and relationships. Therefore, you can only create valid types of objects, edit appropriate attributes, and create appropriate relationships between objects. Further, the DDL Editor provides normal graphical user interface actions for you to act on the DDL file, so preventing syntax errors and other problems associated with editing a DDL file directly.

The following description of the DDL file structure is based on an extract of the Insurance.ddl file provided with Component Broker. The application family defined within the DDL file is referred to as “your application family”. Other application families provided by Component Broker are referred to by name. Some lines have been missed out, and replaced with ellipsis (...), where they do not add any significant value to the description.

Declaration of objects supplied by Component Broker that are used by your application family
At the top of the DDL file is a set of lines that declare the objects supplied by Component Broker that are used by your application family. Most, if not all, of these objects are defined in the **iDefaultApplications** application family provided by Component Broker. The DDL Editor displays these objects within the **iDefaultApplications** Application Family object. You should not change the definitions for these objects.

```
//*******************************************************************
// Top of DDL file
//*******************************************************************
//Pre-Declare objects used by the Application which are Supplied by CB
//*******************************************************************
ApplicationFamily.iDefaultApplications;
ApplicationFamily.iDefaultApplications/Dll.somibl1i; ...
ApplicationFamily.iDefaultApplications/
ManagedObjectClass.IBOIMManagedObject_IViewCollectionImpl;
```

DDL Editor representation of application families provided by Component Broker

Declaring objects
Objects within a DDL file are identified by their class and object name, in the following format:

```
object_class.object name;
```

If an object exists in a different DDL file, the name is prefixed with the DDL file name; for example, *ApplicationFamily.iDefaultApplications/Dll.somibl1i*. Note that the DDL file name is joined to the object name by a forward slash character (/) and each declaration line ends in a semi-colon (;). Several declarations, separated by commas, can be grouped on the same line.

Object names
If the name of an object is to contain embedded blanks or any of the following characters, the name must be enclosed in double quotes:

```
{ } , ; . /
```

(Open brace, close brace, comma, semi-colon, period, and forward slash.).Otherwise, the use of double quotes is optional.

The System Manager expands the name of a DLL object into a fully-qualified path name when it creates the corresponding DLL Image. The DLL object name is prefixed with the install path for the application family, and has the file type (.dll or .a) appended. For example, for the DLL object **myappinit** and its application family installed in **c:\Cbroker\appfamily** on Windows NT, the name of the DLL Image becomes **c:\Cbroker\appfamily\myappinit.dll**.

Definition of your application family
A DDL file defines one application family only. Everything about that application family if contained within the definition of that application family, which is delimited by the **ApplicationFamily.family_name** statement and its opening and closing braces **{ ... }**. When you use Object Builder to create an application family, it creates this definition. The DDL Editor displays your application family as the **family_name** Application Family object. You can use the DDL Editor to change the attributes of your application family. Normally, your application family and all its objects have the same value for the **version** attribute.

```
// Describe the application family named "Insurance".
ApplicationFamily."Insurance"
{ // Set the attributes of the application family.
description = ""; version = "1.0.0"; ... }
//***********************************************************
// Bottom of DDL file
//***********************************************************
```

Object attributes
All object attribute statements have the general form **attribute_name = value;**

Text string values must be enclosed in doubles quotes. If an attribute has several values (at the same time), the sequence of values is enclosed within braces and each value separated by commas; for example, `{value1,value2,value3}`

When needed, attribute statements are created automatically by Object Builder. Other attributes do not need to be defined in a DDL file, and are left to assume their default values. Using the DDL Editor, you can display and change attribute values by selecting the **Edit** action from the object's pop-up menu. (This invokes the Object Editor, as used by the System Manager user interface.) Note that using the DDL Editor, only appropriate attributes for an object can be changed and the syntax for names and values are validated.

Forward declaration of objects that are needed later
At the top of your application family definition is a set of entries that declare the objects that are defined later within your application family. For each entry, the object class and name must match its later definition.

```
Foward declarations of objects which will be needed later.
Xarm."LifeIns"; MappedType.BCPBO_csClaimBOBO_DO; ...
Container.InsuranceContainer;
```

Definition of objects within your application family
Within your application family definition there are separate definitions for all the objects of the family, as declared at the top of your application family definition.

All these object definitions are at the same level within your application family definition, and have the same general format, as shown below:

```
 // Define the Xarm image
for "LifeIns". Xarm."LifeIns" { openString = "LifeIns";
switchLoadFile = "db2slf"; }
```

An object definition is delimited by its **class_name.object_name** statement and its opening and closing braces **{ ... }**.

Definition of applications within your application family

An application within an application family is defined like any other object. A server application is delimited by its **Application.application_name** statement and its opening and closing braces **{ ... }**. A client application is delimited in the same way by its **ClientApplication.application_name** statement and braces.

Within the braces are statements that define appropriate attributes of the application and the "provides" relationships to objects that the application provides.

```
// Define applications.
Application."LifeInsObjects"
{ // Set the attributes of the application.
description = ""; version = "1.0.0"; runControl = stop;
requiredJavaVMName = ""; ProvidesXarm -> { Xarm."LifeIns" };
ProvidesManagedObjectClass -> { ManagedObjectClass."BCPMO_csClaimMO",
ManagedObjectClass."BCPMO_csPayoutFractionMO", ...
ManagedObjectClass."A_C_ModuleMO_csAgentMO" }; ... }
// End definition of application LifeInsObjects.
```

Application "provides" relationships

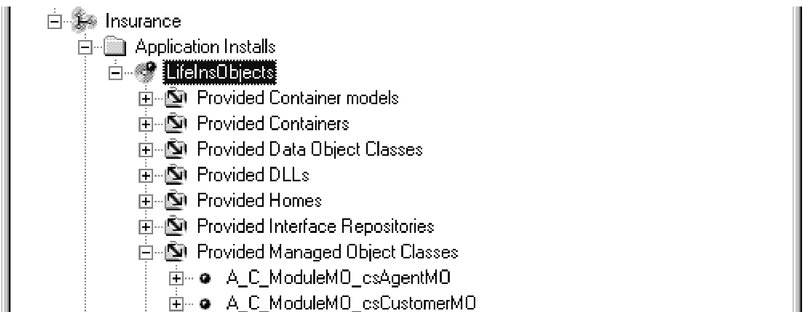
For each non-application object within your application family there should be a "provides" relationship with at least one application. (An object can have "provides" relationships with more than one application.) These relationships are created automatically by Object Builder or using the DDL Editor by dragging an object and using the **Configure object_class** action from the application's pop-up menu. The DDL Editor displays the relationships in "provides" folders within the application object.

The "provides" relationships have the same form:

```
 relationship_name -> {
object_class.object_name };
```

The "arrow" (->) indicates a forward relationship to the object that the application provides. If an application provides several objects of the same class, the sequence of object identifiers is enclosed within braces and each identifier separated by a comma.

DDL Editor representation of application "provides" relationships



Other relationships between objects

Some objects need to contain relationships with other objects. A relationship should exist in only one of the two related objects. (If an object in your application family needs a relationship to an object in the default application family, it must be defined in the object in your DDL file.) These relationships are created automatically by

Object Builder or using the DDL Editor by dragging an object and using the **Configure object_class** action from another object's pop-up menu. The DDL Editor displays the relationships in folders within the object. Note that using the DDL Editor, you do not need to be concerned about the relationship name and direction, nor by which relationships are valid for an object.

The relationships of an object are normally listed after the objects attributes; for example:

```
// Define the MO class 'BCPMO_csClaimMO'.
ManagedObjectClass."BCPMO_csClaimMO"
{ // Define the attributes. description =
"Description of the class named csClaim."; ... interfaceName =
"BCP::csClaim";
// Define the relationships. This defines the DLLs containing
//this class's information. ContainsManagedObjectImplementation
<- dll.b_s; containsmanagedobjectkeyimplementation
<- dll.b_c; containsmanagedobjectcopyhelperimplementation
<- dll.b_c; }
```

The direction of the relationship, defined by the arrow (<- or ->), must be appropriate for the object that the relationship is defined in. You can get a clue about the correct direction from the relationship name: relationship names starting "Uses" or "Provides" are forward relationships (->); relationship names starting "Contains" or "Collects" are backward relationships (<-).

For example, the relationships for a home define the managed object class, data object class, and container used by the home (as forward relationships). It also defines the home provided by Component Broker that "collects" this home (as a backward relationship) and the home of view objects provided by Component Broker that this home uses (as forward relationships).

```
// Define a home for the "LifeInsObjects_BCPMO
csClaimMO_BCPDOImpl_csClaimDOImpl" class. Home."LifeInsObjects_BCPMO
csClaimMO_BCPDOImpl_csClaimDOImpl"
{ // Define the attributes. ...
// Define therelationships.
UsesManagedObjectClass -> ManagedObjectClass.BCPMO_csClaimMO;
UsesDataObjectClass -> DataObjectClass."LifeInsObjects_BCPDOImpl_csClaimDOImpl";
UsesContainer -> Container.InsuranceContainer; CollectsHome
<- applicationfamily.idefaultapplications/home.iboimhomeofregqihomes; homeofviews ->
ApplicationFamily.iDefaultApplications/Home.iBOIMViewCollection; }
```

RELATED CONCEPTS

"Application DDL Files" on page 381
Creating and Editing DDL Files
"The DDL Editor" on page 382
The files and process used by the DDL Editor (page 383)

RELATED TASKS

Edit Application DDL Files

Chapter 12. Access a Component through FlowMark

FlowMark

IBM FlowMark has a workflow manager that enables you to automate your business processes. The workflow manager usually runs as a distributed application on local area networks that consist of several workstations, but you can also have the FlowMark clients and servers on a single, stand-alone workstation.

You can maintain as many FlowMark databases as you need. The workflow manager lets you access them either one at a time, or simultaneously. Data in each database must be replicated in other databases to ensure that the information required to execute a process is available on each server. The Import and Export utilities help in data replication.

The workflow manager maintains FlowMark Definition Language (FDL) files, which are ASCII text files that store definitions that are contained in FlowMark databases in an external format called FlowMark Definition Language. FDL files are created when you export data from FlowMark, and you can import them into FlowMark as well.

RELATED CONCEPTS
Activity

FDL

FlowMark Definition Language (FDL) is an external format for defining programs, data structures, and workflow models in a flat, ASCII text file. Definitions in an FDL file can be imported into a FlowMark database.

The FDL file name usually has the .fdl extension. The name can have a maximum of 8 alphanumeric characters (0-9 and A-Z).

The name of the FDL file is used by Object Builder as the default file for the **Import FDL** and **Export FDL** actions: when you select **Import FDL** from the pop-up menu of the bag's folder, you import from a file with this name. Similarly, when you select the **Export FDL** action from the bag, definitions of the bag are stored in a file with the same name

RELATED CONCEPTS
"Bag"

RELATED TASKS
"Work with Bags - Overview" on page 406

Bag

A FlowMark bag is a container for data structures and programs. The data structures contain the same data as IBM FlowMark data structures. Programs contain data from the FlowMark program registrations as well as data required to enable FlowMark to invoke a method of a CBConnector managed business object.

© Copyright IBM Corp. 1997, 1998

RELATED CONCEPTS
Data Structure
Program

RELATED TASKS
"Work with Bags - Overview" on page 406
"Work with Data Structures - Overview" on page 408
"Work with Programs - Overview" on page 409

Add a Bag

A bag contains FlowMark programs and data structures. Associated with every bag is a FlowMark Definition Language (FDL) file. This file is created along with the bag.

You can create a bag that holds data structures and programs, directly from the FlowMark folder. Once a bag is created, you can import a FlowMark Definition Language (FDL) file, which has the definitions of the data and programs stored in a flat text format. If the FDL file that you import has the same name as that associated with the bag, it overwrites the definitions within the bag. If the two FDL files have different names, new data structures and programs are created within the bag.

To add a bag, follow these steps:
1. Select the FlowMark folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Add Bag**. The FlowMark Bag wizard opens to the Name Page, where you can specify the name of the bag to hold your programs and data structures.

The bag is created as a folder within the FlowMark folder, and it contains a folder each for the data structures and the programs.
RELATED CONCEPTS
"Bag" on page 395
Data Structure
Program

RELATED TASKS
"Work with Bags - Overview" on page 406

Data Structure

A FlowMark data structure is the description of any data that is used as either input or output, or that is referenced in either exit or transition conditions.

A data structure is an ordered list of variables (called members), each of which has a name and a data type. The members of a data structure can be one-dimensional arrays of the following data types:
- long
- floating point
- string
- a defined data structure (the data structure is nested)

For example, a data structure to define an address can have members of the string data type for the name of the street and the city.

An array can have a maximum of 512 elements. Each element of an array counts as a member item. In the case of nested data structures, each of the member items of the nested data structure count as one member item of the containing data structure. The maximum number of members a data structure can have is 512, assuming none of them are arrays, but the maximum number of member items a data structure can contain is also limited to 512. So, if a data structure has a member, which is an array of 512 elements, the data structure is limited to just that one member.

Follow these rules when you name a data structure:
- The name must not exceed 32 characters in length.
- You can include blanks except for leading blanks, trailing blanks, and consecutive blanks.
- You can use any character above ASCII character 31 in the name except for the following characters: * ? " . : ;

Your description for a data structure (optional) can be free-form, with a maximum length of 1024 characters.

RELATED CONCEPTS
"FlowMark" on page 395

RELATED TASKS
"Work with Data Structures - Overview" on page 408

Add a Data Structure

To add a data structure to a bag, follow these steps:
1. Select the Data Structures folder within the bag.
2. From its pop-up menu, select **Add Data Structure**. The Data Structure wizard opens to the Names Page, where you can specify a name for the data structure, and provide a description of it, if you want to.
3. Click **Next**, or use the page list arrow to the left of the page name to go to the Members Page. Here, you must define at least one member for the data structure.

The data structure is added as an object within the Data Structures folder.

RELATED CONCEPTS
Data Structure

RELATED TASKS
"Work with Data Structures - Overview" on page 408

Program

In FlowMark, a program is a computer-based application program that supports the work to be done in an activity. Program activities reference executable programs using the logical names associated with the programs in the FlowMark program registrations.

A program registration is the identification of a program to a FlowMark database so that it can be assigned to a program activity in a workflow model. Program registrations can contain run-time parameters for executable programs.

A FlowMark program corresponds to method invocation. FlowMark creates the flow of action: it drives the order in which methods are called. FlowMark acts as the sole client program.

RELATED CONCEPTS
Activity
RELATED TASKS
"Work with Programs - Overview" on page 409

Activity

An activity is a step within a process. In FlowMark, a process is a sequence of activities that must be completed to accomplish a task. The process is invoked when the activity is started. It represents a piece of work that an assigned person can complete by starting a program or another process.

A FlowMark workflow model has the following types of activities:

Program Activity: has a program assigned to perform it. The program is invoked when the activity is started. Output from the program can be used in the exit condition for the program activity and for the transition conditions to other activities.

Process Activity: has a process assigned to perform it. A process is a sequence of activities that must be completed to accomplish a task. The process is invoked when the activity is started. A process activity represents a way to reuse a set of activities that are common to different processes. Output from the process can be used in the exit condition for the process activity and for the transition conditions to other activities.

RELATED CONCEPTS
"FlowMark" on page 395Program

RELATED TASKS
"Work with Programs - Overview" on page 409

Add a Program

A FlowMark program is contained within a FlowMark bag, along with data structures. Once you create a bag in Object Builder, it contains the Data Structures folder and the Programs folder. Before you add a program, you must have at least one data structure defined.

To add a program, follow these steps:
1. From the pop-up menu of the Programs folder, select **Add Program**. The FlowMark Program wizard opens to the Names Page.
2. Specify the name of the program registration, the input and output data structures to be used by the program. You can also specify whether the same data structures that you indicated as input and output data structures are to be used for program activity.
3. Click **Next** to turn to the Business Object Page. Click the list button, and select the name of the business object to be associated with the program in the **Business Object** field. This field lists all configured managed objects currently in the model.

4. Select the **Create**, **Execute Method**, or **Delete** radio button, to indicate the action to be taken on the business object. This selection determines the set of methods that can be called on the business object, and consequently the remaining pages available in the wizard. You can also name the file that is to be created on program generation, and indicate whether to propagate the input data structure to the output data structure.
   Note the following points:
   - The availability of the Find Parameters Page, the Input Parameters Page, and the Output Parameters Page depend on your selection in step 4. If you select **Create** in step 4, continue with steps 8, 9, and then, 12. If you select **Execute Method** in step 4, follow all the remaining steps. If you select **Delete** in step 4, continue with steps 5 through 7, and then, 12.
   - Object Builder Release 2.0 does not support invocation of attributes (get and set methods) on the configured business object instance; it only supports method calls.
5. Click **Next**. The Find Parameters Page opens. The Find Parameters folder contains all attributes of the business object that are key attributes.
6. Select a find parameter from the folder.
7. Click the list button, and select one of the members of the input data structure from the **Member Name** field. The selected find parameter is mapped to the member you choose.
8. Click **Next**. The Input Parameters Page opens. If the action that you selected for the business object on the Business Object Page is **Create**, the Input Parameters folder contains all attributes of the business object that are key attributes. If the action that you selected for the business object is **Execute Method**, this folder contains all input parameters that are defined for the business object's methods. They include all those parameters that are designated as either In or In/Out on the Methods Page of the Business Object Interface wizard.
9. You can view the default mapping of the parameters to the data structure members by selecting each of the parameters in this folder. To change an input parameter map, select it and make the change in the **Member Name** field.
10. Click **Next**. The Output Parameters Page opens. The Output Parameters folder contains all output parameters that are defined for the business object's methods. They include all those parameters that are designated as either **Out** or **In/Out** on the Methods Page of the Business Object Interface wizard. The return type of the method (if other than void) is shown too.
11. You can view the default mapping of the parameters to the data structure members by selecting each of the parameters in this folder. To change an output parameter mapping, select it and make the change in the **Member Name** field.
12. Click **Finish.**

Note the following points when you do the mapping:

- Find parameters of the types *wchar* and *void* cannot be mapped to data structure members of the following data types:
  – long
  – float
  – string
- Find parameters of the types *wstring* and *Object* cannot be mapped to data structure members of the following data types:

- – long
  – float
- If the input data structure member you select for the mapping is an array, you must specify in the **Array Entry** field a value greater than 1 and less than or equal to the array size.

RELATED CONCEPTS
Data Structure
Program

RELATED TASKS
"Work with Data Structures - Overview" on page 408
"Work with Programs - Overview" on page 409
Work with FlowMark Business Objects

Map a Component to a Data Structure

To associate a FlowMark program with an instance of a component in Object Builder, you have to map the input, the output, and the find parameters, or a combination of them to the members of the input or output data structures.

The following tasks deal with mapping of the business object to either input or output data structures:
- Map Input Parameters to the Input Data Structure
- Map Output Parameters to the Output Data Structure
- Map Find Parameters to the Input Data Structure

RELATED CONCEPTS
"Business Object" on page 17
Data Structure
"Bag" on page 395

RELATED TASKS
Work with FlowMark Business Objects

Map Input Parameters to the Input Data Structure

When you add a program to your model, you must associate an instance of a configured business object with it, and map the parameters of the business object's method to either input or output data structures, or both, depending on the action to be taken on the business object.

You can do the mapping either when you are adding a new program or when you are editing the program. You must define at least one data structure.

To map the input parameters of the business object instance to the input data structure, follow these steps:
1. If you are adding a new program, from the pop-up menu of the Programs folder, select **Add Program**. If you are editing a program, from the pop-up menu of the program in the Programs folder, select **Properties.** The FlowMark Program wizard opens to the Names Page.

2. From the **Input Data Structure** field, select the data structure to be used by the program as the input data structure.
3. Click **Next** to go to the Business Object Page.
4. Select the **Execute Method** radio button to indicate that you want a method defined on the business object to be executed.
5. Click **Next**. The Input Parameters Page opens. The Input Parameters folder contains all input parameters that are defined for the business object's methods. They include all those parameters that are designated as either **In** or **In/Out** on the Methods Page of the Business Object Interface wizard.
6. Select an input parameter from the folder.
7. Click the list button, and select one of the members of the input data structure from the **Member Name** field. The selected input parameter is mapped to the member you select.

Note the following points when you do the mapping:

- Input parameters of the types *wchar* and *void* cannot be mapped to data structure members of the following data types:
  - long
  - float
  - string
- Input parameters of the types *wstring* and *Object* cannot be mapped to data structure members of the following data types:
  - long
  - float
- If the input data structure member you select for the mapping is an array, you must specify in the **Array Entry** field a value greater than 1 and less than or equal to the array size.

RELATED CONCEPTS

Data Structure
Program

RELATED TASKS

"Configure a Managed Object" on page 377
"Work with Data Structures - Overview" on page 408
"Work with Programs - Overview" on page 409
Work with FlowMark Business Objects
Call a Component Method from FlowMark

Map Output Parameters to the Output Data Structure

When you add a program to your model, you must associate an instance of a configured business object with it, and map the parameters of the business object's method to either input or output data structures, or both, depending on the action to be taken on the business object.

You can do the mapping either when you are adding a new program or when you are editing the program. You must define at least one data structure.

To map the output parameters of the business object instance to the output data structure, follow these steps:

1. If you are adding a new program, from the pop-up menu of the Programs folder, select **Add Program**. If you are editing a program, from the pop-up menu of the program in the Programs folder, select **Properties.** The FlowMark Program wizard opens to the Names Page.
2. From the **Output Data Structure** field, select the data structure to be used by the program as the output data structure.
3. Click **Next** to go to the Business Object Page.
4. Select the **Execute Method** radio button, to indicate that you want a method defined on the business object to be executed.
5. Go to the Output Parameters Page. The Output Parameters folder contains all output parameters that are defined for the business object's methods. They include all those parameters that are designated as either Out or In/Out on the Methods Page of the Business Object Interface wizard.
6. Select an output parameter from the folder.
7. Click the list button, and select one of the members of the output data structure from the **Member Name** field. The selected output parameter is mapped to the member you select.

Note the following points when you do the mapping:

- Output parameters of the types *wchar* and *void* cannot be mapped to data structure members of the following data types:
  - long
  - float
  - string
- Output parameters of the types *wstring* and *Object* cannot be mapped to data structure members of the following data types:
  - long
  - float
- If the output data structure member you select for the mapping is an array, you must specify in the **Array Entry** field a value greater than 1 and less than or equal to the array size.

RELATED CONCEPTS
Data Structure
Program

RELATED TASKS
"Configure a Managed Object" on page 377
"Work with Data Structures - Overview" on page 408
"Work with Programs - Overview" on page 409
Work with FlowMark Business Objects
Call a Component Method from FlowMark

Map Find Parameters to the Input Data Structure

When you add a program to your model, you have to associate an instance of a configured business object with it, and map the parameters of the business object's method to either input or output data structures, or both, depending on the action to be taken on the business object.

You can do the mapping either when you are adding a new program, or when you are editing the program. At least one data structure must be defined.

To map the output parameters of the business object instance to the input data structure, follow these steps:

1. If you are adding a new program, from the pop-up menu of the Programs folder, select **Add Program**. If you are editing a program, from the pop-up menu of the program in the Programs folder, select **Properties.** The FlowMark Program wizard opens to the Names Page.
2. From the **Input Data Structure** field, select the data structure to be used by the program as the input data structure.
3. Click **Next** to go to the Business Object Page.
4. You can select either the **Execute Method** radio button to indicate that you want a method defined on the business object to be executed, or the **Delete** radio button to indicate that you want to delete the business object.
5. Click **Next.** The Find Parameters Page opens. The Find Parameters folder contains all attributes of the business object that are key attributes.
6. Select a find parameter from the folder.
7. Click the list button, and select one of the members of the input data structure from the **Member Name** field. The selected find parameter is mapped to the member you select.

Note the following points when you do the mapping:

- Find parameters of the types *wchar* and *void* cannot be mapped to data structure members of the following data types:
  - long
  - float
  - string
- Find parameters of the types *wstring* and *Object* cannot be mapped to data structure members of the following data types:
  - long
  - float
- If the input data structure member you select for the mapping is an array, you must specify in the **Array Entry** field a value greater than 1 and less than or equal to the array size.

RELATED CONCEPTS

Data Structure
Program

RELATED TASKS

"Configure a Managed Object" on page 377
"Work with Data Structures - Overview" on page 408
"Work with Programs - Overview" on page 409
Work with FlowMark Business Objects

Work with FlowMark Business Objects - Overview

To associate a FlowMark program with a business object in Object Builder, you must map the input parameters, the output parameters, and the find parameters, or a combination of these, to the members of the input or output data structures in FlowMark.

The following tasks deal with FlowMark business objects:

- Create a Component Instance through FlowMark
- Call a Component Method from FlowMark
- Delete a Component Instance through FlowMark

RELATED CONCEPTS
"Business Object" on page 17
Data Structure
"Bag" on page 395

RELATED TASKS
"Map a Component to a Data Structure" on page 400

Create a Component Instance through FlowMark

You can create an instance of a configured business object when you add or edit a FlowMark program.

Follow these steps:

1. If you are adding a new program, from the pop-up menu of the Programs folder, select **Add Program**. If you are editing a program, from the pop-up menu of the program in the Programs folder, select **Properties.** The FlowMark Program wizard opens to the Names Page.
2. Specify the name of the program registration, and the input and output data structures to be used by the program. You can also specify whether the same data structures that you indicated as input and output data structures are to be used for a program activity.
3. Click **Next** to go to the Business Object Page. Click the list button of the **Business Object** field, and select the name of the business object to be associated with the program. This field lists all configured business objects currently in the model.
4. Select **Create** to indicate that you want to create a business object. You can either create a business object from a copy helper or from the primary key. You can also name the file that is to be created on program generation, and indicate whether you want the generated program to propagate the input data structure to the output data structure.
5. Click **Next.** The Input Parameters Page opens, and you can use it to map the input parameters of the business object's methods to the input data structure.

RELATED CONCEPTS
"Business Object" on page 17
"Bag" on page 395

RELATED TASKS
"Work with Programs - Overview" on page 409
Access a Component through FlowMark

Call a Component Method from FlowMark

Restriction: Object Builder, Release 2.0 does not support invocation of attributes (get and set methods) on the configured business object instance; it only supports method calls.

You can indicate that a method is to be executed on an instance of a component in Object Builder when you add or edit a FlowMark program.

Follow these steps:

1. If you are adding a new program, from the pop-up menu of the Programs folder, select **Add Program**. If you are editing a program, from the pop-up menu of the program in the Programs folder, select **Properties.** The FlowMark Program wizard opens to the Names Page.
2. Specify the name of the program registration, and the input and output data structures to be used by the program. You can also specify whether the same data structures that you indicated as input and output data structures are to be used for program activity.
3. Click **Next** to go to the Business Object Page. Click the list button of the **Business Object** field and select the name of the business object to be associated with the program. This field lists all configured business objects currently in the model.
   Note: If you are using a program that was created by importing an FDL file, you must explicitly follow step 3, though a business object appears to be associated with the program, when you examine the properties of the program using the Business Object Page. If you do not, the .cpp and .mak files will not be created when you generate the program in step 9.
4. Select the **Execute Method** radio button. This enables you to call methods that are defined on the business object, on the Component Broker server business object. Once you select this option, you can also name the file that is to be created on program generation, and indicate whether to propagate the input data structure to the output data structure.
5. Click **Next**. The Find Parameters Page opens, and you can use it to map the find parameters of the business object to the input data structure.
6. Click **Next.** The Input Parameters Page opens, and you can use it to map the input parameters of the business object's methods to the input data structure.
7. Click **Next.** The Output Parameters Page opens, and you can use it to map the output parameters of the business object's methods to the output data structure.
8. Click **Finish**. If you are defining a new program, it is added to the Programs folder. If you are changing the definition of an existing program, the new definitions take effect.
9. Generate the program. The corresponding .cpp and .mak files are created.
10. Compile the generated file. This results in an executable file.
11. Launch the executable (.exe) file from within IBM FlowMark. The methods will be executed on the business object in the Component Broker server.

RELATED CONCEPTS

"Business Object" on page 17
"Bag" on page 395

RELATED TASKS

"Work with Programs - Overview" on page 409
Access a Component through FlowMark

Delete a Component Instance through FlowMark

You can delete an instance of a component that you created through FlowMark when you add or edit a program.

Follow these steps:

1. If you are adding a new program, from the pop-up menu of the Programs folder select **Add Program**. If you are editing a program, from the pop-up menu of the program in the Programs folder, select **Properties.** The FlowMark Program wizard opens to the Names Page.
2. Specify the name of the program registration, and the input and output data structures to be used by the program. You can also specify whether the same data structures that you indicated as input and output data structures are to be used for program activity.
3. Click **Next** to turn to the Business Object Page. Click the list button of the **Business Object** field, and select the name of the business object to be associated with the program. This field lists all configured business objects currently in the model.
4. Select **Delete** to indicate that you want to delete a business object. The remove method acts on the object. You can also name the file that is to be created on program generation, and indicate whether you want the generated program to propagate the input data structure to the output data structure.
5. Click **Next**. The Find Parameters Page opens, and you can use it to map the find parameters of the business object to the input data structure.

RELATED CONCEPTS
“Business Object” on page 17
“Bag” on page 395

RELATED TASKS
“Work with Programs - Overview” on page 409
Access a Component through FlowMark

Work with Bags - Overview

A bag contains FlowMark programs and data structures. Associated with every bag is a FlowMark Definition Language (FDL) file. This file is created when you add a bag.

You can also create a new bag by creating new data structures and programs within an existing bag when you import an FDL file.

The following tasks deal with FlowMark bags:

- “Add a Bag” on page 396
- “Edit a Bag”
- “Delete a Bag” on page 407

RELATED CONCEPTS
“Bag” on page 395
Data Structure
“Bag” on page 395

Edit a Bag

A bag contains FlowMark programs and data structures. Associated with every bag is a FlowMark Definition Language (FDL) file. This file is created along with the bag.

You cannot rename the bag, but you can specify a different FDL file to be associated with the bag. When you select **Import FDL** from the pop-up menu of the bag's folder, you import from a file with this name. Similarly, when you select the **Export FDL** menu item from the pop-up menu of the bag, definitions of the bag are stored in a file with the same name. You must specify a different FDL file to be associated with the bag if you do not want to overwrite the FDL file that is to be imported, when you select the **Export FDL** action from the bag.

To edit a bag, follow these steps:

1. Select the bag from the FlowMark folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Properties**. The FlowMark Bag wizard opens to the Name Page, where you can specify a different FDL file in the **FDL File Name** field. Type a description, or modify the existing description if you want to.
3. Click **Finish**.

   Note the following points:

   - The name of the FDL file can contain only alphanumeric characters, and cannot exceed eight characters in length. The description must not exceed 1024 characters.
   - Object Builder uses the FDL file you specify as the default for both the **Import FDL** and **Export FDL** actions from the bag.

Your specifications overwrite the existing bag definitions in Object Builder.

You can also edit a bag by importing an FDL file with the same name as the one associated with the bag, but which has been modified. You can also import a new FDL file into Object Builder (**Import FDL** from the pop-up menu of the bag in the FlowMark folder): the current bag shows additional programs and data structures, as defined in the imported FDL file.

RELATED CONCEPTS
"Bag" on page 395

RELATED TASKS
"Work with Bags - Overview" on page 406

Delete a Bag

To delete a bag from the FlowMark folder, follow these steps:

1. Select the bag in the FlowMark folder.
2. From its pop-up menu, select **Delete**.

You are asked to confirm that you want to delete the bag, and if you select **Yes**, the bag and its associated data structures and programs are deleted from the folder.

RELATED CONCEPTS
"Bag" on page 395
RELATED TASKS
"Work with Bags - Overview" on page 406

Work with Data Structures - Overview

Restriction: Flowmark does not support renaming of objects. That is, if you map any object's attributes, methods, or parameters to a FlowMark program, and later change their names in the User-Defined Business Objects folder, the new names are not automatically propagated to the objects within the Flowmark folder. It is recommended that you finish your work with components you want to associate with FlowMark, and then create the FlowMark objects within Object Builder.

A FlowMark data structure is the description of any data that is used as either input or output, or that is referenced in either exit or transition conditions.

A data structure is an ordered list of variables (called members), each of which has a name and a data type.

The following tasks deal with data structures:
- "Add a Data Structure" on page 397
- "Edit a Data Structure"
- "Map a Component to a Data Structure" on page 400
- "Delete a Data Structure"

RELATED CONCEPTS
Data Structure
"Components" on page 15

Edit a Data Structure

To edit a data structure, follow these steps:
1. Select the data structure from its folder within the FlowMark folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Properties**. The Data Structure wizard opens to the Name Page, where you can change the description of the data structure. You cannot rename the data structure.
3. Click **Next.** The Members Page opens. You can rename the members, change their type, and type a new description for each one.
4. Click **Finish.**

The new definitions take effect.

RELATED CONCEPTS
Data Structure

RELATED TASKS
"Work with Data Structures - Overview"

Delete a Data Structure

To delete a data structure from a bag, follow these steps:
1. Select the data structure from the Data Structures folder within the FlowMark folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Delete**.

You are asked to confirm that you want to delete the data structure. If you selectYes the data structure, and its members are deleted from the folder.

RELATED CONCEPTS
Data Structure
RELATED TASKS
"Work with Data Structures - Overview" on page 408

Work with Programs - Overview

A FlowMark program consists of all the information necessary to generate an executable application program in order to allow FlowMark to invoke a CBConnector business object's method.

The following tasks deal with programs:

- "Add a Program" on page 398
- "Edit a Program"
- Delete a Program

RELATED CONCEPTS
"Bag" on page 395

Edit a Program

To edit a program, follow these steps:

1. Select the program from its folder within the FlowMark folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Properties**. The FlowMark Program wizard opens to the Name Page. You cannot rename the program, but you can select different data structures for input and output, indicate whether they are to be used for program activity, and change the description.
3. Click **Next** to go to the Business Object Page. You can select a different business object to be associated with the program from the **Business Object** field. This field lists all configured business objects currently in the model.
4. You can change the action to be taken on the business object: select from **Create**, **Execute Method**, or **Delete**. Your selection determines the set of methods that can be called on the business object; the FlowMark Program wizard adjusts according to your selection. You can also rename the file that is to be created on program generation, and change your decision on whether you want the generated program to propagate the input data structure to the output data structure.
5. Use the **Next** button to advance to the remaining pages of the wizard.
6. On the Find Parameters Page, the Input Parameters Page, and the Output Parameters Page, you cannot change the data type of the parameters listed in the folder, but you can map them to different data structure members.
7. Click **Finish.**

Your new definitions take effect.

RELATED CONCEPTS
Program

RELATED TASKS
"Work with Programs - Overview" on page 409

Delete a Program

To delete a program from a bag, follow these steps:

1. Select the program from the Programs folder within the FlowMark folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Delete**.

You are asked to confirm that you want to delete the program. If you select **Yes**, the program is deleted from the folder.

RELATED CONCEPTS
Program

RELATED TASKS
"Work with Programs - Overview" on page 409

Chapter 13. Troubleshooting

If you encounter problems when you build your code into DLLs, the following tips may help you. This section covers the following problems:

- Memory Problems
- Generally Odd Behavior
- Cannot Start Object Builder on AIX

Memory Problems
If you are working with a large project (more than thirty components), you may need to increase the maximum heap size of the Java virtual machine. You can do so by editing the ob.bat file:

1. Make sure Object Builder is closed.
2. Edit \Cbroker\bin\ob.bat
3. Change the parameter -mx255m, increasing the number by five for each additional component in your project (this number is approximate, and assumes components of average complexity).

   For example, if your project contains one hundred components then change the parameter to -mx605m (seventy additional components multiplied by 5m each, plus the original 255m).
4. Start Object Builder. The new parameter is used, and the maximum size of the Java virtual machine is increased.

Generally Odd Behavior
Not all exceptions are displayed in the user interface. After major actions such as saving a project, check Object Builder's command window for any exceptions. The command window is the window from which you started Object Builder, or the window that appeared in the background if you started Object Builder from the **Start** menu.

Cannot Start Object Builder on AIX
If you receive the operating system message "Killed" when you try to start Object Builder, you need to increase the amount of paging space on your machine. Object Builder requires a minimum of 200MB of paging space in order to run on AIX.

BAD_OPERATION Exception with Composite Components
If the client program receives a BAD_OPERATION exception while using a composite component, the most probable cause is inaccurate location information in the business object implementation's properties. Look in the activity log of the server to determine the cause of the exception. If the problem is a failure to locate one of the member components in the composition, check the Location page of the composite component's Business Object Implementation wizard.

Java Server Fails to Run with Composite Components
If the Java client and Java server are installed on the same machine, make sure the CLASSPATH has the files ibmcbjs.zip and somshor.zip listed before somojor.zip. Otherwise the Java business object will attempt to use the ORB interfaces in somojor.zip, instead of the server-side ORB interfaces it requires in ibmcbjs.zip and somshor.zip.

Error on Running Object Builder: "The input line is too long"
If the classpath environment variable is too long (approximately 1700 characters or more), then Object Builder cannot be run. You will need to shorten the classpath, by

© Copyright IBM Corp. 1997, 1998

removing directories and .jar or .zip files (besides those added by the Component Broker install, and besides any PA beans and their dependencies being used in your projects), before running Object Builder.

Check a Model for Consistency

While Object Builder does perform regular consistency checks on your model, there are circumstances in which the model can become internally inconsistent. If you are experiencing consistency problems with your generated code (for example, type mismatches between an attribute and its referenced interface), run the consistency checker to diagnose the problem, and generate a report on the state of your model.

To check a model for consistency, follow these steps:

1. Open the project whose model you want to check (select **File - Open New Project**). The project opens, and the project model is loaded into Object Builder.
2. From Object Builder's menu bar, click **File - Check Model**.

   The consistency checker dialog opens.
3. Select the types of consistency problem you want to check for.
4. Click **Run**. The consistency check runs, and its output is displayed in a report window.
5. Review the report.

   You can save the report for later review by clicking **Save**.

   Each error, warning, or information message includes the file, module, object type, and object name to which the message applies.
6. Click **OK** to close the report window and return to Object Builder.

RELATED CONCEPTS

"Projects and Models" on page 4
"Chapter 13. Troubleshooting" on page 411

RELATED REFERENCES

"Consistency Checker Errors"

Consistency Checker Errors

No.	Message Text	Description
1	Unexpected interface type: %1.	Internal error
2	Exception %1 caught while processing.	Internal error
3	A configured managed object does not have a mapped application managed object.	A managed object configuration in the Application Configuration folder exists, but the original managed object in the User-Defined Business Objects folder does not exist. Re-create the managed object in the User-Defined Business Objects folder, and then edit the managed object configuration and reselect the managed object.

No.	Message Text	Description
4	A configured managed object does not have a mapped data object.	A managed object configuration in the Application Configuration folder exists, but does not have an associated data object. Delete the managed object configuration, and then re-create it with a mapping to the data object.
5	A configured managed object has more than one mapped data object. Only the first one will be used.	A managed object configuration in the Application Configuration folder exists, but has more than one data object associated with it. Only a single data object can be associated with any instance of a managed object, and the first data object listed will be used.
6	The data access pattern in the container does not match that in the contained business object.	The data access pattern (Caching or Delegating) in the business object must match the selected data access pattern for the associated container. Select a different container, edit the container, or edit the business object.
7	The data access pattern in the container does not match that in the contained data object.	The data access pattern (Local Copy or Delegating) in the data object must match the selected data access pattern for the associated container. Select a different container, edit the container, or edit the data object.
8	The selected keys do not match in the business object and managed object.	The key selected in the business object implementation must match the key selected in the configured managed object. Edit the implementation or the managed object configuration and make sure they both refer to the same key.
9	The selected copy helpers do not match in the business object and managed object.	The copy helper selected in the business object implementation must match the copy helper selected in the configured managed object. Edit the implementation or the managed object configuration and make sure they both refer to the same copy helper.
10	A key must be specified for a data object that uses either the ’BOIM with any key’ or ’unit test’ environment.	A key must be defined, and selected in the data object implementation, for this type of data object. Define and select a key, or edit the environment of the data object.
11	No key must be specified for a data object that uses the ’BOIM with UUID key’ environment.	A ’BOIM with UUID key’ data object cannot have a user-defined key. Either clear the key selection for the data object or change the environment of the data object to ’BOIM with Any Key’.
12	Queryable interfaces must use a home such as the system-provided ’BOIMHomeOfRegQIHomes’.	
13	Non-queryable interfaces must use a home such as the system-provided ’BOIMHomeOfRegHomes’.	

No.	Message Text	Description
14	The selected keys do not match in the business object and data object.	The component must use the same key consistently in all its objects. If a key is associated with the business object then the same key must be associated with the data object implementation. Edit the business object or data object and make sure they both refer to the same key.
15	The selected copy helpers do not match in the business object and data object.	The component requires at most a single copy helper definition. If a copy helper is associated with the business object then the same copy helper must be associated with the data object implementation. Edit the business object or data object and make sure they both refer to the same copy helper.
16	Non-queryable interfaces must use a home derived from ’IHome’.	IHome is the base interface for all homes. Non-queryable interfaces cannot use homes that derive from IQueryableIterableHome.
17	Queryable interfaces should use a home derived from ’IQueryableIterableHome’.	Queries are supported only against homes the derive from IQueryableIterableHome. Queryable interfaces can be placed in a home that derives from just IHome, however, queries against the home for the interface will not be permitted.
18	When a business object interface inherits from ’IManagedClient IManageable’, the business object implementation must also inherit from ’IManagedClient IManageable’.	This is a basic requirement of the programming model for all components that inherit directly from the programming framework (that is, are base classes in your design), rather than inheriting from other components.
19	When a business object implementation inherits from ’ManagedClient IManageable’, the data access pattern must be either Caching or Delegating.	A business object implementation that inherits directly from the programming model framework must use the Caching or Delegating data access pattern. A business object implementation that inherits from a another business object implementation must use the Same as Parent’s access pattern.
20	When a business object interface inherits from another interface, the business object implementation should also inherit from the other implementation.	
21	When a business object implementation inherits from another implementation, the data access pattern must be ’Same As Parent’s’.	A business object implementation that inherits directly from the programming model framework must use the Caching or Delegating data access pattern. A business object implementation that inherits from a another business object implementation must use the Same as Parent’s access pattern. This helps to avoid mixing access patterns within the hierarchy of business object implementations.

No.	Message Text	Description
22	When a business object implementation inherits from more than one implementation, all the parent's data access patterns must be the same.	Access patterns cannot be mixed within the hierarchy of business object implementations that define a business object.
23	When a data object implementation inherits from '%1', the environment must be '%2', and the persistent behavior must be '%3'.	The programming model interfaces from which a data object implementation may derive are tightly coupled to the choice of environment and form of persistent behaviour. Consult the Component Broker Toolkit online documentation for details.
24	When a data object implementation inherits from another implementation, the environment must be 'Same as parent's'.	A data object implementation that inherits from another data object implementation must use the Same as Parent's environment. This helps to avoid mixing environments within the hierarchy of data object implementations.
25	When a business object implementation inherits from another business object implementation, the managed object should inherit in the same way.	The business object implementation inheritance and associated managed object inheritance hierarchies should be parallel, otherwise there may unexpected results at runtime.
26	When a home inherits from 'IManagedClient IHome', then the home implementation should inherit from 'IManagedAdvancedServer ISpecializedHome'.	
27	When a home inherits from 'IManagedClient IHome', then the home managed object should inherit from 'IManagedAdvancedServer ISpecializedHomeManagedObject'.	
28	When a home inherits from 'IManagedAdvancedClient IQueryableIterableHome', then the home implementation should inherit from 'IManagedAdvancedServer ISpecializedQueryableIterableHome'.	
29	When a home inherits from 'IManagedAdvancedClient IQueryableIterableHome', then the home managed object should inherit from 'IManagedAdvancedServer ISpecializedQueryableIterableHomeManagedObject'.	
30	In the relationship described below, the type '%1' cannot be resolved.	Ensure the type of the relationship object is correctly defined.
31	In a 'one to many' relationship that is resolved by a foreign key, the business objects that are on the 'many' side of the relationship must be queryable.	The 'list' method of a foreign key relationship on the referencing object is satisfied by running a query against the foreign key attributes of the referenced object. This query will not be possible if the referenced object is not queryable.

No.	Message Text	Description
32	An invalid type, caused by the deletion of the original type definition, is in use.	Change the type of the identified element to one that is currently available, or remove the element.
33	The listed types must be the same.	Two related elements (such as an attribute in an interface and in the implementation) must have the same type, but do not. Either change the type in the interface (the change will propagate to the implementation), or delete and re-create the implementation.
34	An unresolved (forward declared) type is in use.	This is typically caused by an incorrect type selection (such as a mistake in the type field when creating an attribute or method). Ensure that the type is specified correctly, in the format "file module::interface"
35	The attribute shown is defined in an implementation, but there is no corresponding definition in the interface.	Internal model error. While you can add implementation-only attributes, this attribute was derived from its associated interface, but no longer exists there. Define the attribute as specified on the interface object. Alternatively, delete and re-create the object implementation.
36	The method shown is defined in an implementation, but there is no corresponding definition in the interface.	Internal model error. While you can add implementation-only methods, this method was derived from its associated interface, but no longer exists there. Define the method as specified on the interface object. Alternatively, delete and re-create the object implementation.
37	The parameters for the listed methods do not match.	
38	The attribute shown identifies the incorrect interface as containing the original definition.	
39	The identifier syntax shown is illegal.	Internal model error. This can occur if incorrect type specifications are use for attribute or methods, or by importing incorrect XML. Export the model to XML files, remove the offending definition, and create a new instance of the model by importing the XML again.
40		
41	The file shown is not included in any DLL.	The file must be included in a DLL before it can be built, configured into an application and deployed. Under the Build Configuration folder, add the file to either an existing DLL or a new DLL.
42	Constructs defined at file scope are not qualified, and may produce compilation errors for Java business objects.	It is recomended that all constructs (typedefs, structs, etc) be defined at module or interface scope.

No.	Message Text	Description
43	The method shown has a non-void return type, but an empty method body.	A method with a non-void return type and an empty method body will fail to compile in C++ and Java. Provide an implementation that returns a value of the correct return type or throws an exception.
44	The model contains internal inconsistencies. Use the '-i -f' options to fix the damaged parts.	A correctable model error has been detected. Use the "fix model" option to fix the damaged parts. No data will be lost as a result of this operation.
45	The model contained internal inconsistencies. The damaged parts of the model were corrected.	
46	The model contains links to other models that are from the previous release.	It takes longer to open a project if it has dependencies on other projects that have not yet been migrated. You can decrease the time it takes to open this project by opening the projects it depends on and saving them in the newest format.
47	The 'home to query' in a relationship resolved by foreign key cannot be found.	
48	The interface identified as the 'home to query' in a relationship resolved by foreign key is not a home.	
49	String attributes used in keys should either strip or pad trailing spaces.	Strings that do not strip or pad are subject to subtle transformations by the backing store. For example, DB2 will pad strings mapped to CHAR(n) column types. Such transformations may inhibit the proper functioning of Queries and findByPrimaryKeyString operations.
50	The identifier listed ends with a restricted suffix (%1).	Using a restricted suffix will generally result in a compilation error. Change the suffix.
51	The first identifier listed ends with a restricted suffix (%1), and will conflict with the second identifier listed.	The suffix listed is reseved by CORBA. The two identifiers listed will conflict with each other, due to the use of the restricted suffix by the first identifier. Change the suffix.
52	The method '%1' should be overridden with caution; See the programming reference for details.	
53	The data object implementation listed has no inheritance defined.	This can occur when you deploy an object on multiple platforms. For example, if the 390 platform is selected as a deployment platform for the data object, then a parent that is appropraite for the 390 platform must be selected on the implementation inheritance page. You must change your view to 390 (select **View - 390**) before the appropriate parents appear on the page, and can be selected.

No.	Message Text	Description
54	The data object implementation and the associated container are configured incompatibly; See the programming reference for details.	
55	All persistent objects configured against a common container must share a common database name.	A Component Broker container cannot manage connections to more than one database in the current release. Insure that all the schemas under all the data objects configured into the container are associated to the same database.
56	An attribute used in the selected key has not been mapped to the DO.	The key attributes of a business object are part of the essential state of the object and must be persisted in the data object. To correct the problem, add the missing key attributes to the data object.
57	An attribute used in the selected copy helper has not been mapped to the DO.	The copy helper attributes of a business object are part of the essential state of the object and must be persisted in the data object. To correct the problem, add the missing key attributes to the data object.
58	When a business object interface inherits from 'IManagedClient IManageable', it should not also inherit from any other interfaces.	A business object interface must inherit directly from either the framework or another business object, but not both.
59	In a 'one to many' relationship that is resolved by a foreign key, the business objects that are on the 'many' side of the relationship must include a back referencing attribute that is read/write.	The back-referencing attribute represents the set of foreign key attributes of the referenced object at the 'many' side of the 'one to many' relationship. A foreign key relationship is not possible without the back reference. To correct the problem, add a read/write attribute of the type of the referencing object to the referenced object
60	Persistent object attribute names should not be longer than 26 characters, as some embedded SQL preprocessors cannot tolerate longer lengths.	To correct this problem, open the Persistent Object properties wizard and rename the attributes.
61	A managed object must not inherit from 'IManagedClient IManageable'.	
62	A makefile is targetted for a platform that is not selected by a contained interface.	This can occur if a makefile is targetted for more platforms than is really required. Ensure that the makefile is targetted correctly, and contains only those files that are applicable to the target platforms.
63	The container behavior for methods called outside a session called 'Ignore condition and complete the call' is not supported in this release.	
64	The container behavior for methods called outside a transaction called 'Ignore the condition and complete the call' is not supported in this release.	

No.	Message Text	Description
65	The container memory management policy called 'Passivate a component at the end of a transaction' must be selected whenever 'Cache Service' is selected for your data object. Failure to do so will cause severe memory leaks.	
66	The container memory management policy called 'Passivate a component at the end of a session' must be selected whenever 'use PAA Session Service' is selected. Failure to do so will cause severe memory leaks.	
67	The container memory management policy called 'Passivate a component at the end of a transaction' must be selected whenever 'use PAA Transaction Service' is selected. Failure to do so will cause severe memory leaks.	

RELATED CONCEPTS
"Components" on page 15
"Component Assembly" on page 16

RELATED TASKS
"Check a Model for Consistency" on page 412

Restrictions for R2.0

The following restrictions apply to Object Builder in R2.0:

CLASSPATH Restriction
Assuming that you installed Component Broker in a directory such as x:\Cbroker, you cannot have your CLASSPATH variable contents longer than 1780 characters. If the installation directory path is longer, you must have a correspondingly shorter CLASSPATH value. You get a run-time error if you exceed this limit. This is because commands (such as ob.bat), which invoke the Object Builder functions prepend the OB jar files to the class path, and then invoke the java code to run Object Builder.

Rose Design Restriction
You should not restructure your design after exporting. If you restructure your design (for example, move a class from one package to another), the export process will treat the change as a combination add and delete, rather than a move. This would result in two definitions of the class in Object Builder (a new class definition for its new position, and the old class definition for its old position), which is not valid.

Opening a Project with Disconnected Network Drives
If you open a project in Object Builder while there are disconnected network drives on your system, when you click **Browse** the network drives will be accessed and reconnected. This may take some time.

Heap Size for Java Virtual Machine
If your project has thirty components or more, you may need to manually edit the ob.bat file to increase the maximum heap size for the Java Virtual Machine used by Object Builder. The default heap size is 255m (-mx255m): increase the heap size by 5m for each component beyond thirty.

Opening in Editor Puts Model in Use
If you open a schema or schema group in an editor (the **Open in Editor** pop-up menu choice), the model will be locked by the editor. When you close the editor, the model remains locked until you do one of the following:

- Log off of Windows NT and log back on again.
- Use the Windows NT Task Manager to terminate the processes EVFXLXPM.EXE and IWFWTV35.EXE

Silent Exceptions
Not all exceptions are displayed in the user interface. After major actions such as saving a project, check Object Builder's command window for any exceptions. The command window is the window from which you started Object Builder, or the window that appeared in the background if you started Object Builder from the **Start** menu.

Restrictions when Adding Comments
When you add comments for the different objects in Object Builder (for example, a business object module), the comments are generated within language comment delimiters. You cannot use these delimiters: /* and */ within your comments.

Naming Restrictions for Interfaces, Modules, Constructs, Attributes, Methods, and Relationships
The name must include only alphanumeric characters (letters and numbers), and must start with a letter: for example, a1bc23 is acceptable; a#bc23 and 1abc23 are not acceptable.

You cannot use the following keywords to name the interface:

- Java keywords
- IDL keywords

None of the following names can be used as interface names:

- any method name in Java.lang.Object. These include names such as *clone, finalize, hashCode, notifyAll, wait, equals, getClass, notify*, and *toString*
- any name that is suffixed with *Package, Holder, Helper*, *Ref*, *_var*, or *_ptr*.
- *goto*

Note: For attributes, the following additional restriction applies:
If you use OO-SQL keywords like KEY, REF, TYPE and WORK as attribute names, Object Builder generates objects that cannot be used with the Query Service, and you will not be able to perform OO-SQL queries. See OO-SQL Keywords for a complete list.

Restrictions for Attributes and Methods at the Interface Level
You cannot select a non-IDL type class that you have imported into Object Builder as the type of an attribute, or a method return type if you are defining attributes and methods for an object's interface; you can use the non-IDL type class only if you are defining the attributes or methods for the object's implementation.
An attribute that you define for an object's interface can only have a public

implementation. If you want to define attributes that are either private or protected, you must define them at the object's implementation level.

Type of Attributes Available for Use in Keys and Copy Helpers
Business object attributes of the following data types are available for use in the copy helper:

- All the CORBA standard data types
- All business object interfaces defined in the model (You can define a business object interface and use it as a key attribute's data type for another business object.)

Business object attributes of complex data types are excluded from use as copy helper attributes. These include the following types:

- *any* and *wstring*
- Typedefs, structures, and unions, which are defined as constructs.

Requirement for Copy Helper Attributes
Any attributes you include in the copy helper must also be included in the data object.

Business Object Interface Inheritance Restriction
Abstract base classes are not supported by either the Interface Definition Language Compiler (IDLC) or Object Builder. So, any business object interface that you specify as a parent for another business object interface must have an implementation, even if every method in that implementation only throws a NO_IMPLEMENT exception.

Key and Copy Helper Inheritance Restriction
If the interface has one or more parents, the parent interface attributes are also available for selection. If the key or copy helper will inherit from the parent's key or copy helper, you should **not** select any of the parent interface attributes.

Restrictions when Mapping Data Object Attributes to Persistent Object Attributes
Multiple data object attributes cannot be mapped to the same persistent object attribute.

You cannot map multiple data object attributes to the same persistent object attribute.

When you map a data object to multiple persistent objects, you must map each key attribute of the data object directly to each of the key attributes of the different persistent objects.

Mapping Pattern Restrictions

Primitive Mapping Pattern

- An attribute that does not use the primitive mapping between the data object and the persistent object will not be capable of participating in an object query.

Map as a Key

- When you map attributes using a foreign key and create a persistent object and schema from the data object implementation, it will not automatically create a foreign key in the schema.

- If you are mapping attributes using a key, you must select only those attributes of the persistent object that are defined as PO keys, to map to the key attribute. To check which attributes are PO keys, select the persistent object in either the User-Defined Data Objects folder or the DBA-Defined Schemas folder and use **Properties** from its pop-up menu to view the Persistent Object Page.

Map using helper class

- Object Builder does not provide the default mapping between complex data types (*any, wchar* and *wstring* and types defined as constructs, which include typedefs, structures, and unions) and DB2 database types. You must provide your own helper class for these mappings. No other kind of mapping is permitted. Note however, that you can map the members of structures that are of an object interface type using the **Key Home** mapping, but you must map the structure itself using a mapping helper.
- When you map a data object attribute that is also specified as an attribute of the key for the corresponding business object, to multiple persistent object attributes using a mapping helper, all the persistent object attributes that are mapped must be persistent object keys.

Complex Attributes and Mapping Patterns

- This release of Object Builder supports only structures (structs) as complex attributes.
- Nested structs are not supported. However, structs whose members are other structs are supported.

Restrictions when Adding a Persistent Object and a Schema

Foreign Key Not Propagated down to the Schema
Even if there is a key defined for another business object and it is designated as a foreign key, when you create a persistent object and schema for a business object referenced by the other object , it will not automatically create a foreign key in the schema.

Restrictions when Adding a Persistent Object from a Schema
You must map all schema columns to their corresponding persistent object attributes; otherwise you may get exceptions thrown at runtime if you use the Query Service.

If your schema uses the Oracle Cache Service, you can create a persistent object from it only if the schema columns are of the VARCHAR2 or NUMBER data types, or any of the IBM DB2 data types.

Restrictions when Naming a Persistent Object Attribute
A persistent object attribute name must not exceed 26 characters in length.

Restriction when Mapping a Persistent Object to a Schema
You must map all schema columns to their corresponding persistent object attributes; otherwise, you may get exceptions at runtime, if you use the query service.

SQL Files Supported for Import

- This release supports SQL DDL files for DB2 MVS 4.1 databases. If an SQL DDL file contains a mix of supported and unsupported statements, the supported statements will be imported.

- This release supports SQL DDL files only from the following DBMSs:
  - DB2 MVS 4.1, DB2 V5 (UDB).
  - Oracle 8.0.4.0

    Oracle SQL files can be imported, but language elements that have no analog in DB2 will not be parsed correctly except for columns of the NUMBER or VARCHAR2 data type. Object Builder does not support any non-ANSI syntax construction such as Oracle comments (/*...*/), SQL commands.
- SQL files larger than 2 MB are not recommended.

Restrictions when Importing Oracle SQL Files

- Only Oracle 8.0.4.0 databases are supported.
- Support for Oracle backend databases is limited to data objects that use the Oracle Cache Service only. That is, data objects that use embedded SQL, or any other form of persistent behavior and implementation will not be able to access data stored in Oracle databases.
- Reference collections are not supported in conjunction with Oracle backends for Component Broker Release 2.0.
- For Oracle, only optimistic caching is supported.
- In the current release of Component Broker, only the Oracle VARCHAR2 and NUMBER data types are supported, along with those Oracle data types that have an equivalent type in DB2. That is, Object Builder accepts all SQL/DS and DB2 types and the Oracle NUMBER, NUMBER(p), NUMBER(p,s) and VARCHAR2 types. It will not accept any other Oracle types such as RAW(n), LONG RAW, NCHAR(n), NVARCHAR2, and ROWID. See “Oracle Data Type Mappings” on page 113 for a complete list.
- Object Builder will not accept the Oracle data type NUMBER with a negative scale.

SQL Statements Supported for Import

Currently, the only SQL statements supported are the following:

- CREATE TABLE
- ALTER TABLE
- DROP
- CREATE VIEW
- COMMENT ON

None of these statements must contain expressions or column functions. The CREATE VIEW statement must contain only a simple query (SELECT statement). Currently there is no support for unnamed columns, expressions, functions, or sub-selects in CREATE VIEW.

Most views are read-only but some can be updated. The Embedded SQL preprocessor (idatapre) will fail on any .sqx file generated from an embedded static persistent object, which you create for a read-only view in the database.

If you detect that a view is read-only (at DLL build time), you must ensure that every one of the framework methods insert(), update() and del() for the persistent object has an empty method body.

Double-Byte Character Set (DBCS) is not supported for the English version of Component Broker.

Importing Schemas with no Primary Keys

Object Builder lets you import schemas for which no primary keys have been defined. However, these schemas can result in query exceptions at runtime.

To avoid this happening, you can either select **Properties** from the pop-up menu of the schema, and select any of the schema columns as the database key (Select the **DB Key** check box.), or before you import the SQL file, edit the source file and add a PRIMARY KEY constraint for at least one of the tables.

Schema Group Restrictions

A table associated with a schema of one schema group cannot reference a foreign key defined in a table within another schema group.

Procedural Adaptor (PA) Bean Restrictions

Importing Beans

Only beans created using VisualAge for Java Release 2.0 are compatible with this release (2.0) of Component Broker.

Attributes and Push-Down Method Parameter Types of the PA Bean

Only the Java types *int, float, double, boolean, char, short* and *java.lang.String* are supported as attribute types, and push-down method parameter types for the PA bean.

Linking to Non-IDL Types on AIX

When you define a shared library (client or server DLL) on AIX that contains references to a non-IDL type, the shared library statically links the code for the non-IDL type. For example, if some of your code uses the IString class, then the shared library that contains your code links statically to the library file that contains the IString object code.

Static linkage can multiply the size of your shared library file by a factor of 20 or more. The advantage of static linkage is that you do not need to ship the shared library file that defines the non-IDL type.

To link dynamically, and reduce the size of your shared library file, edit the makefile for your shared library file (DLL) and change the referenced file libibmcl.a to libibmcls.a . You must then ship libibmcls.a with your shared library (include it on the Additional Executables page of the DLL wizard) in accordance with the Licensed Program Specifications for C Set++ for AIX.

FlowMark Restrictions

Object Builder Release 2.0 does not support invocation of attributes (get and set methods) on the configured business object instance; it only supports method calls.

Mappings in the FlowMark folder are not automatically updated when a mapping element is renamed. That is, if you map any object’s attributes, methods, or parameters to a flowmark program, and later change their names in the User-Defined Business Objects folder, the new names are not automatically propagated to the objects within the Flowmark folder. It is recommended that you finish your work with components you want to associate with FlowMark, and then create the FlowMark objects within Object Builder.

Before generating a program that was created by importing an FDL file, you must explicitly associate a business object with the program. (Use the Business Object Page of the FlowMark Program wizard.) If you do not, the .cpp and .mak files will not be created on generation.

OS/390 Platform Restrictions
When one of the constraint platforms is 390 (you select **Platform - Constrain - 390**), wchar and wstring are not available for selection as either attribute types, method return types, or method parameter types for your objects.

When you define a data object implementation, all Cache Service options are not available when the target platform is OS/390.

If OS/390 is one of the deployment platforms for the data object implementation, the persistent object class name must not exceed 8 characters. Object Builder validates the length of the persistent object class when you create a persistent object from a data object implementation, but if you change the deployment platform after you have created the persistent object, be sure that you follow the rule. If not, Object Builder will truncate the name to the 8.3 format. This may result in two persistent object file names becoming identical after truncation, since Object Builder assumes the object's file name to be the same as the persistent object class name.

When you import a procedural adaptor bean, and have OS/390 as the deployment (target) platform (Platform - Constrain - 390), only the EXCI, OTMA, and Generic connector types are available for selection (LU 6.2, HOD, and ECI are not available).

When you use Procedural Adaptors, you cannot call endResource() on the business object when the target platform is OS/390.

RELATED CONCEPTS
"Chapter 13. Troubleshooting" on page 411

Composition Restrictions

One-to-Many Relationships
The one-to-many relationships of components you add to a composition will not be available in the composition, or in the interface of any composite components based on the composition. One-to-many relationships, and the methods that support them (for example, addRel, removeRel, listRel), will not be included or republished in the composites you create.

Composition Include Files
The include files for composited components are included automatically. You do **not** need to add them to the Composition File wizard, Include Files page.

The Composition File wizard, Include Files page shows IManagedClient as an include file, even though compositions inherit from IManagedLocal. This include file is required for code generation, and should not be deleted.

DB2 Column Name Limitations
DB2 limits column names to 18 characters. Because of the mapping of attribute names as they are added to a composite, the attribute names may be longer than

18 characters. The attribute names may have to be edited in the Add Persistent Object and Schema wizard to shorten them to less than 18 characters.

Republishing Methods
Attributes in the composition should not delegate to methods that throw User Exceptions. The code generated will not compile. CORBA attributes do not support exceptions.

Attributes in a composition should not delegate to methods that have “out” or “inout” parameters. It is not possible to republish “out” or “inout” parameters on attributes.

Adding or Renaming a Managed Object After a Composite Is Built
When you add or rename a composited component in the Composition Editor, Composition page (Objects to Composite list), you must do the following:

1. Open any business object interfaces that are based on the group, and select **Refresh from Composition**.
2. Update the key, if necessary.

Chapter 14. Debug Local Applications

Write Programs for Debugging

You can make your programs easier to debug by following these simple guidelines:

- Where possible, do not put multiple statements on a single line, because some debugger features operate on a line basis. For example, you cannot step over or set line breakpoints on more than one statement on the same line.
- Assign intermediate expression values to temporary variables to make it easier to verify intermediate results. For example, you will not be able to display the substrings of IString objects in the first C++ code fragment below, but you will in the second:

```
// Can't see the substrings in this one
if (StrA.subString(x,y)==StrB.subString(m,n)) dups++;
// Can see the substrings here
IString SubA=StrA.subString(x,y);
IString SubB=StrB.subString(m,n)
if (SubA==SubB) dups++;
```

To be able to debug your programs at the level of source code statements, you must specify C++ compiler options that generate debug information, and in some cases you must specify options that enable the debugger to work properly with your code.

RELATED REFERENCES
IBM VisualAge for C++ Compiler Options

Compile a Program for Debugging

Note: This section does not apply to programs that you plan to debug with Object Level Trace. To compile Component Broker applications for distributed tracing and debugging, see “Compile Application Code with OLT Flags” on page 486

In order to be able to debug your program at the source code level, you need to compile your program with certain compiler options that instruct the compiler to generate symbolic information in the object file. The Related Topics section below points to information on how to compile your program for a specific environment.

RELATED REFERENCES
“IBM VisualAge C++ Compiler Options”

RELATED TASKS
“Debug Optimized Code” on page 467
Write Programs for Debugging
“Compile Application Code with OLT Flags” on page 486

IBM VisualAge C++ Compiler Options

Compile your C++ programs with the IBM VisualAge C++ **/Ti+** option (to generate debugging information) if you want to be able to debug your program at the source code statement level. You should also consider using the following options:

Option
Purpose

© Copyright IBM Corp. 1997, 1998

/Tm+ Enable debug memory management support. Use this option if you want to do heap debugging (using the Storage monitor and **Check heap when stopping**).

/O- Compiles your program with optimization off. This is the default. (Some optimizations reorder the execution sequence of your program, while others may eliminate expressions whose result is never used. You may find it confusing to debug a program compiled with optimization, because statements may execute in a nonsequential fashion or not at all.)

/Oi- Compiles your program with inlining off. This is the default.

/DEbug
Use this option with the ilink command when linking objects that were compiled with debug information but are being separately linked. When you specify the /Ti+ option for a source file, the compiler passes the /DE linker option to the linker automatically.

RELATED TASKS
"Invoke the Debugger" on page 433
"Debug Heap Use" on page 465

Interpreted Java Compiler Options

If you use the **javac** compiler to compile your code for debugging, you can set breakpoints and step through your source code without using any compiler options. Use the **-g** option if you want to examine local, class instance and static variables while debugging.

Here is a partial list of compiler options to consider when compiling your classes:

Option
Purpose

-g Compiles your code with debug information. Use this option if you want to examine the contents of local variables when debugging your classes. You can still set breakpoints and step through your code if you do not compile your classes with this option.

-O Compiles and optimizes your code. Do not use this option if you want to debug your classes. If you compile your code with this option **all** debugging information is removed from the class during optimization.

-classpath <path>
Overrides the **CLASSPATH** environment variable with the path specified by <path- . Use this option when you want to try compiling something without modifying the **CLASSPATH** environment variable

-d <dir>
Determines the root directory where compiled classes are stored. This is useful since classes are often organized in a hierarchical directory structure. With this option, the directories are created below the directory specified by <dir- .

For a complete list of compiler options, refer to documentation provided with the JDK.

Environment Variables

The debugger uses the following environment variables. If you want to place multiple entries in any of the variables that contain path names, separate the entries with a semicolon.

- "IVB_DBG_CASESENSITIVE Environment Variable" on page 430
- "IVB_DBG_LANG Environment Variable" on page 430
- "IVB_DBG_LOCAL_PATH Environment Variable" on page 430
- "IVB_DBG_NUMBEROFELEMENTS Environment Variable" on page 431
- "IVB_DBG_OVERRIDE Environment Variable" on page 431
- "IVB_DBG_PATH Environment Variable" on page 431
- "IVB_DBG_REMOTE_SEARCH_PATH Environment Variable" on page 431
- "IVB_DBG_TAB Environment Variable" on page 431
- "IVB_DBG_TABGRID Environment Variable" on page 432
- "Other Environment Variables" on page 432
- "INCLUDE Environment Variable" on page 432
- "Other Environment Variables" on page 432
- "CLASSPATH Environment Variable (Java Only)" on page 432

RELATED TASKS

"Set Environment Variables for the Debugger"

Set Environment Variables for the Debugger

The debugger user interface running on the workstation uses certain environment variables to determine the dominant language, the location of online help files, and so on. See the Related Topics below for help on the environment variables themselves.

To set an environment variable for a given session of the debugger, do the following on your workstation:

1. Open a command shell window.
2. Use the SET command to set each environment variable to the required value:
   ```
   SET VARNAME=VAL1;VAL2;VAL3
   ```
   where VARNAME is the name of the environment variable, and VAL1, VAL2, and VAL3 are values assigned to it (normally multiple values for a variable are separated by semicolons).

   If you want to add more values to an existing variable, use the following syntax:
   ```
   SET VARNAME=%VARNAME%;VAL4;VAL5
   ```
   This adds both the existing contents of VARNAME and the new values to the variable.
3. Invoke the debugger user interface from that command shell.

RELATED REFERENCES
"Environment Variables" on page 429

IVB_DBG_CASESENSITIVE Environment Variable

The IVB_DBG_CASESENSITIVE environment variable, if set to a non-null value (for example, "yes", 1, "true", etc.) tells the debugger to compare part names and module names on a case sensitive basis. By default the debugger converts all names to uppercase for comparison purposes. Note that this does not affect filesystem accesses which are opererating system dependent and not affected by IVB_DBG_CASESENSITIVE.

RELATED REFERENCES
"Environment Variables" on page 429

IVB_DBG_LANG Environment Variable

The IVB_DBG_LANG environment variable sets the dominant language for a debugging session. The setting of IVB_DBG_LANG determines the display style for windows, dialogs, and menus throughout the debugging session, regardless of what language the program your are debugging was written in. If IVB_DBG_LANG is not set, the default language is C++. Available choices are:

CPP The dominant language is C++.

JAVA The dominant language is Java.

The following aspects of debugger behavior are affected by the setting of IVB_DBG_LANG:

Value of IVB_DBG_LANG	CPP	JAVA
Term used in dialogs and windows to indicate a C++ function or a Java method	The term "function" is used	The term "method" is used
Heap checking	You can perform heap checks using Run - Check heap when stopping	Heap checking is not available.
Startup	Startup runs to the first statement in main after program initialization, unless you chose to debug program initialization.	Startup runs until the first debuggable statement in the application.

RELATED REFERENCES
"Environment Variables" on page 429

IVB_DBG_LOCAL_PATH Environment Variable

The IVB_DBG_LOCAL_PATH environment variable is used to locate executables and DLLs on the debuggee machine.

RELATED REFERENCES
"Environment Variables" on page 429
"Search Order" on page 439

IVB_DBG_NUMBEROFELEMENTS Environment Variable

The IVB_DBG_NUMBEROFELEMENTS environment variable can be set to an integer value to tell the debugger the maximum number of elements to display for an array, structure, or object in a Program, Private, Local Variables, or Popup monitor.

RELATED REFERENCES
"Environment Variables" on page 429

IVB_DBG_OVERRIDE Environment Variable

The IVB_DBG_OVERRIDE environment variable takes precedence over IVB_DBG_PATH. If you set your IVB_DBG_PATH variable in your system settings, but you want to temporarily add another path that takes precedence over IVB_DBG_PATH, set IVB_DBG_OVERRIDE. To restore IVB_DBG_PATH as the path used to locate executables and DLLs, clear IVB_DBG_OVERRIDE, for example by using:

```
set IVB_DBG_OVERRIDE=
```

RELATED REFERENCES
"IVB_DBG_PATH Environment Variable"
"Search Order" on page 439
"Environment Variables" on page 429

IVB_DBG_PATH Environment Variable

The IVB_DBG_PATH environment variable is used to locate debug source files on your workstation that are not stored in the same location as the executable being debugged. For example, if your debug executable is stored in F:\BUILDS\SANDDUNE\TEST but your source code is stored in F:\SOURCE and F:\SOURCE\INCLUDE, you should set your IVB_DBG_PATH variable as follows:

```
set IVB_DBG_PATH=F:\SOURCE;F:\SOURCE\INCLUDE
```

You can set the IVB_DBG_PATH environment variable on both client and server systems.

The search order used to search for source files depends on the settings of other environment variables as well.

IVB_DBG_REMOTE_SEARCH_PATH Environment Variable

The IVB_DBG_REMOTE_SEARCH_PATH environment variable is used to search specified paths on the remote host for a requested source file.

RELATED REFERENCES
"Environment Variables" on page 429
"Search Order" on page 439

IVB_DBG_TAB Environment Variable

The IVB_DBG_TAB environment variable affects how the debugger expands tab characters in a source or mixed view within a Source window, when IVB_DBG_TABGRID is set to 0 (or is not set). The value for this variable is an integer, indicating the number of spaces to convert a tab character into. Unlike IVB_DBG_TABGRID, IVB_DBG_TAB does not cause the debugger to place tabbed

information in specific columns; it simply results in each tab in the displayed files being converted to the indicated number of spaces.

RELATED REFERENCES
"IVB_DBG_TABGRID Environment Variable"

IVB_DBG_TABGRID Environment Variable

The IVB_DBG_TABGRID environment variable affects how the debugger uses tab characters to align tabs to columns in a source or mixed view within a Source window. The value of this variable is an integer indicating the starting position and frequency of the tab. For example, if you set IVB_DBG_TABGRID=6, the debugger sets tab stops at 6, 12, 18, 24, and so on. If IVB_DBG_TABGRID is set to a nonzero value, the setting of IVB_DBG_TAB has no effect.

RELATED REFERENCES
"Environment Variables" on page 429
"IVB_DBG_TAB Environment Variable" on page 431

INCLUDE Environment Variable

The INCLUDE environment variable is used by both the compiler and the debugger. It specifies the path the compiler and debugger use to locate C++ include files (files included in your source code with the **#include** directive.)

RELATED REFERENCES
"Environment Variables" on page 429

CLASSPATH Environment Variable (Java Only)

The CLASSPATH environment variable tells the debugger, as well as the Java Virtual Machine and other Java applications, where to find your class libraries.

This variable must be set correctly for any of your Java applications to work.

RELATED REFERENCES
"Search Order" on page 439

Other Environment Variables

The debugger also uses the following standard environment variables on the Windows or OS/2 workstation. These variables all contain one or more directory names separated by semicolons:

PATH
The PATH environment variable is used to locate the debugger executable and the executable programs to be debugged, as well as any other executables being run on the workstation. On Windows platforms the PATH environment variable is also used to locate DLLs.

DPATH
The DPATH environment variable is used to locate message files, which the debugger needs to display messages and the text of menus and dialogs.

RELATED REFERENCES
"Environment Variables" on page 429

Start or Stop Debugging a Program

To start or stop debugging a program, you will need to know how to perform some of the following subtasks:

- Attach to a Process
- Start debugging a DLL
- Set breakpoints in your program
- Step through, run, or halt a program

RELATED TASKS
"Invoke the Debugger"
"Attach to a Process" on page 435
"Debug a DLL" on page 451
"Set Breakpoints" on page 446
"Run, Step Through, or Stop a Program" on page 453
"Terminate a Debug Session" on page 456

Invoke the Debugger

You can invoke the debugger remotely (where the program being debugged, and the debugger user interface, are on different machines or operating systems), or locally.

RELATED CONCEPTS
"Remote Debugging" on page 442

RELATED TASKS
"Start the Debugger"
"Debug a Distributed Application" on page 494

Start the Debugger

Note: This information does not apply to starting the debugger in conjunction with Object Level Trace. To debug with OLT (and to debug AIX clients), see "Debug a Distributed Application" on page 494.

You can start a local debug session on Windows NT from the command shell using the following syntax:

`bdbug` [*debugger-options*] [*program-name*[*program-parameters*]]

where:

debugger-options
: includes zero or more valid options supported by the debugger.

program-name
: is the name of an executable file, with a valid path (or no path, if the executable is locatable through the PATH environment variable), and the extension .EXE (optional)

program-parameters
: includes zero or more parameters your program expects; for C++ programs, these parameters are usually accessed by your program through the argc and argv arguments to the **main** function.

If you do not specify a program name, the debugger opens a **Startup** dialog in which you can choose what you want to debug.

RELATED REFERENCES

"Debugger Options"

RELATED TASKS

"Attach to a Process" on page 435
"Debug a DLL" on page 451

Debugger Options

The debugger supports the following options. These options should be specified *after* the `bdbug` command, but before the name of the program you want to debug. For example, to debug the program myprog.exe using the /p- option, use the following command line:

```
bdbug /p- myprog.exe
```

Option
Purpose

/a *process_id*
Attach to the already running process *process_id*. Note that you cannot attach to an already running process on OS/2.

/c *child_process_id*
Start debugging the specified child process of the program being debugged. This option only applies to debuggee programs running on OS/2, and is ignored on other platforms.

/h or /?
Display help for the bdbug command.

/i
Start the debugger in the system initialization code that precedes the call to the main entry point for the program. (C++ only): This can be useful if you need to debug the constructors for static class objects.

/p+
Use program profile information (this is the default). If the debugger has saved a profile containing information on window, breakpoint, and monitor settings from a previous debug session for this program, the profile is used to restore those settings.

/p-
Do not use program profile information. The debugger ignores any program profile information, and opens the debugger in a default appearance with no breakpoints set and only the Session Control window and one Source window.

The bdbugd command on the workstation starts the remote debug daemon and supports the following options:

-v
Display connection information. The default is not to display this information.

-qdebugger=*<debugger name>*
The debugger that the daemon will invoke. The default is the name of the daemon without the last letter. For example, if the daemon name is bdbugd, the default debugger is bdbug.

-qservice=*<service name>*
The service name that is used by the daemon to get the port number. The default is the name of the daemon without the last letter.

-qport=<*port number*>
: The port number on which the daemon listens for incoming requests. If you do not specify a port, the daemon looks up the port number for the specified service from the system services file on the workstation (for example the file \etc\services).

RELATED TASKS
"Attach to a Process"

Debug a Microsoft Visual C++ Program

Note: This section applies only to programs being debugged on Windows NT.

You can use the debugger to debug programs compiled by the Microsoft Visual C++ Compiler Version 5.0, provided the debug information is imbedded in the executable. Both C and C++ programs are supported. To debug such a program follow these steps:

1. Use the Visual C++ compiler to compile the program with the /Z7 option, to produce Microsoft C 7.0 debug information.
2. Link the .obj files with the /DEBUG and /PDB:NONE linker options to generate the necessary debug information and imbed it in the executable.
3. Start debugging the program.

For example, to compile, link, and debug the program hello.c, use the following commands:

```
cl /Z7 /c hello.c
link /DEBUG /PDB:NONE hello.obj /OUT:hello.exe
bdbug hello.exe
```

If you do not compile and link with these options, debug information, if generated, is stored in a separate .PDB file which the debugger cannot read. In this case the debugger treats the executable as if it had no debug information, and only disassembly views of the code are available.

RELATED TASKS
"Start the Debugger" on page 433

RELATED REFERENCES
"Limitations when Debugging Visual C++ Programs" on page 476

Attach to a Process

You can attach to an already running process from the **Startup** dialog, or from the **Process list** item on the **File** menu of the **Source** or **Session Control** windows. When you are debugging a Component Broker method, the Startup dialog and Process list items are disabled.

The Startup dialog appears when you start the debugger and do not specify a program to debug. You can also get to this dialog by choosing **File - Startup** from the Source or Session Control windows. Choose **Process list** from the Startup dialog, and select a process to debug.

Select a process from the process list. See under "Related Topics" for guidance on *when* to attach to a process, and for more details on using the Process List dialog.

When you attach to a process, the **Run - Restart** menu item is disabled.

RELATED TASKS
"Invoke the Debugger" on page 433

RELATED REFERENCES
"When to Use the Process List Dialog"

When to Use the Process List Dialog

Note: The Process List dialog, menu choice, and pushbutton are not available when you are debugging Component Broker programs.

If you close the debugger after attaching to a process on Windows NT, the process terminates.

You can use the **Process List** dialog to attach the debugger to an already running program where an error or failure has occurred. There are two main reasons for attaching the debugger to an already running process:

- You anticipate a problem at a particular point in your program, and you do not want to step through the program or set breakpoints. In this situation, you can run your program, and at a program pause shortly before the anticipated failure (for example, while the program is waiting for keyboard input), you attach to the process. You can then provide the input, and debug from that point on.
- You are developing or maintaining a program that hangs sporadically, and you want to find out why it is hanging. In this situation, you can attach the debugger to the hung process, and look for infinite loops or other problems that might be causing your program to hang.

You can also use the Debug on Demand feature to invoke the debugger when an application running on your system throws an exception that is not handled.

RELATED REFERENCES
"Debug on Demand" on page 438

Specify Command-Line Parameters for Your Program

You can specify command-line parameters for your program either from the command shell where you invoke the debugger, or in the Startup dialog of the debugger.

Specify Parameters from a Command Shell
To pass arguments to your program from a command shell, make sure that they appear after the name of the program on the command line:

```
bdbug [ debugger-options] [program-name [program-parameters]]
```

where *debugger-options* are any options supplied to the debugger itself, *program-name* is the name of the executable you want to debug, and *program-parameters* are the arguments or parameters you want to pass to your program.

Specify Parameters from the Startup Dialog
To pass arguments to your program from a startup dialog, do the following:

1. Open the startup dialog, either by invoking the debugger without a program name, or by choosing **File - Startup** from the Session Control window or a Source window.
2. Enter the name of the program you want to debug in the **Program** entry field, or select a file from the pulldown list.
3. Enter the program parameters in the **Parameters** field.
4. Click on **OK** to start debugging the program with the supplied command-line parameters.

RELATED TASKS
"Invoke the Debugger" on page 433
"Start or Stop Debugging a Program" on page 433

RELATED REFERENCES
"Debugger Options" on page 434

Attach to a Running Java Virtual Machine

You can attach to an already running Java Virtual Machine (JVM) if you start your Java application with the `java -debug` command.

When you start your application with the `java -debug` command, an agent password is printed. Take note of this password because it will be needed to attach to the running JVM.

Once your application is running and you have the agent password:
1. Start the debugger for the host JVM. The command is:
   `jdbug -qhost=<port> -host=<hostname> -password=<password>`
2. Start the debugger interface in attach mode. The command is:
   `jdebug -qhost=<hostname> -qport=<port> /a0`

RELATED TASKS
"Invoke the Debugger" on page 433
"Start the Debugger and the Remote Java Program" on page 444

RELATED REFERENCES
"When to Use the Process List Dialog" on page 436

Start Debugging a Java Applet

When debugging an applet, you must begin by debugging the **sun.applet.AppletViewer** class. After you have begun debugging this class, you can open the source for any class which is part of your applet and set breakpoints.

You can start to debug an applet by following these steps:
1. On the machine where the applet to be debugged exists, issue the `bjdbug -qport=<port>` command.
2. On the machine where the debugger will run, issue the `bdbug -qhost=<host> -qport=<port>` command.
3. In the Program field of the Startup Information dialog box, enter "sun.applet.AppletViewer"

4. In the Parameters field of the Startup Information dialog box, enter the name of the HTML file where the applet is embedded. The file name must be entered as one of the following:
   - a URL. The URL must begin with `http://` or `file:/` (only one slash for URLs beginning with `file`.)
   - a file name only. The file must exist in the directory where you issued the bjdbug command.
5. Click **OK**.
6. In the Source or Session Control window, select **Open New Source** from the **File** menu.
7. Enter the name of your applet class you are debugging.
8. Click **OK**.
9. Set your breakpoint and run.

We recommend that you set a breakpoint on the `init{}` or `start{}`methods since these are the first methods that are called by the applet viewer.

Once you have set this breakpoint you can debug your applet as you would debug a Java application.

RELATED CONCEPTS
"When You Start Debugging" on page 439

RELATED TASKS
"Set Breakpoints" on page 446
"Start the Debugger and the Remote Java Program" on page 444

Debug on Demand

Note: Debug on Demand is available only on Windows NT, and only for local debugging.

Debug on Demand enables you to open a debugging session whenever an unhandled exception or other unrecoverable error occurs in your application. The debugger starts and attaches to your application at the point of fault. This can save you time for two reasons: you do not have to recreate errors, and your application can run at full speed without interference from the debugger until the exception is encountered.

Debug on demand can be started for any application that fails while it is running, even if the application does not contain debug information. With debug on demand, you can even find and fix a problem in your application and let the application continue running.

To enable this feature, type the following at a command shell:

`bdod` *path_name*

where *path_name* is the path where the debugger is installed, for example e:\ibmcppw\bin.

To disable Debug on demand, type the following at a command shell:

```
bdod /u
```

When You Start Debugging

The first time you debug a program, the debugger opens the following windows:

- A Source window. This window contains the source code (or disassembly code, if the program was compiled without debug information) for the **main** function or method of your program.
- The Session Control window. Use this window to access other windows, to control the debugging of threads and functions or methods, and to perform various debugger commands.
- When the program being debugged (the “debuggee”) is running locally, the debugger also raises a text-mode window known as the Debug Application window. This window is used for any console input and output your program may require. This window is only available when debugging C++ programs. For programs using a graphical user interface (such as the User Interface classes of IBM OpenClass), the Debug Application window usually stays blank throughout the debugging session.

The debugger behavior at startup depends upon the dominant language, as specified by the IVB_DBG_LANG environment variable.

IVB_DBG_LANG=CPP
: The debugger runs up to the start of main. If you checked the **Debug program initialization** check box on the Startup dialog, the debugger starts at the first line of disassembly code in your program. Use this check box when you want to debug initialization code such as the constructors for class objects declared at global scope.

IVJ_DBG_LANG=JAVA
: The debugger runs up to the method with a **public static void main(String[])** signature in the selected class.

As you step through or run your program, the debugger may raise additional Source windows for other object files or class files that are executed. If you exit the debugger, then debug the same program later, these other windows appear on reload.

When you start debugging a program for the first time, no breakpoints are set and no variables or expressions are being monitored. During the debug session, you may set breakpoints, or add variables or expressions to a monitor. When you exit the debugger, these breakpoints,variables and expressions are saved in the program profile, and the monitors will be activated the next time you debug this program.

Search Order

The debugger uses a different search order for finding source files, depending on whether you are debugging locally or remotely. It searches through each location in the lists below until it finds a file that matches the requested name.

Local debugging: The debugger searches for source files in:

1. The executable directory
2. The current directory
3. Paths in the IVB_DBG_PATH environment variable

Remote debugging: The debugger searches for source files in the above directories on the debuggee machine, then in:

1. The current directory of the debugger machine
2. The record of directories on the debugger machine
3. Paths in the IVB_DBG_PATH environment variable on the debugger machine

If the source file cannot be located in any of the above directories, a dialog box opens requesting the path name for the source file. The path name you enter is searched for, first on the debuggee machine, then on the debugger machine.

RELATED CONCEPTS
"Record of Directories"

RELATED REFERENCES
"IVB_DBG_PATH Environment Variable" on page 431

Record of Directories

The **record of directories** is a cumulative list of all directories you have entered in source path dialog boxes that opened when the debugger could not locate a source file. This list is only cumulative for the duration of the debugger session; when you exit the debugger, the list is not saved.

The debugger assumes that any directory in the record of directories may be on either the debugger machine or the debuggee machine.

Debugger Windows

Source Window
The Source window displays the source code for the program you are debugging.

When debugging C++ and your program was compiled with debugging information, you have three choices as to how to view it: by its source code, its disassembled machine code, or a combination of the two.

When debugging interpreted Java, the Source window will display an error message if the source code can not be found. A dissassemble machine code view and combination view are not available.

Call Stack Window
The Call Stack window displays information about active functions for a single thread. The thread number is indicated in the window title.

Breakpoints List Window
Use the Breakpoints List window to view breakpoints set in your program, change their characteristics, delete them, or add new ones.

Session Control Window
The Session Control window is the control window of the debugger, and is displayed during the entire debugging session. It contains a status line that indicates what the debugger is doing (for example, **Ready**).

When you are debugging a Component Broker application, the title for the Session Control window consists of the client host id, process id, and thread id of the remote object being debugged.

RELATED CONCEPTS
"Debugger Monitors" on page 457
"Source Window Views"

Source Window Views

If your program was compiled with debugging information, you have three choices as to how to view it in the Source window:

Source View
For C++, Source views are only available for components that were compiled with debug information. For any file that can be viewed with a source view, all views (source, disassembly, and mixed) are available.

In a C++ Source view, the Source window displays the source code for an object file within your program. If the object file was made from several source files, and **Options - Window settings - Notebook** is checked, the source view is displayed in notebook format, with a tab for each source file that is included in the object file.

For interpreted Java, Source views are only available for classes that are in paths listed in the **CLASSPATH** environment variable. Both the source and class file must be accessible through the **CLASSPATH** environment variable. For example, for a class file `a.b.c.d`, the debugger will look for the source as `a/b/c/d.java` under each entry in the **CLASSPATH** environment variable.

In an interpreted Java Source view, the Source window displays the source code for an class file within your program.

Disassembly View
For objects that were *not* compiled with debug information, Disassembly view is the only view available.

In a Disassembly view, the object code for an object used by your program is disassembled into assembly language. For objects that were compiled with debug information, you can switch between disassembly view and the other views (source and mixed).

Mixed View
Mixed views are available only for objects that were compiled with debug information.

A Mixed view is a combination of a source view and a disassembly view. In this view, the **Source** window displays each line of source code followed by the resulting assembly language instructions. If the object file was made from several source files, and **Options - Window settings - Notebook** is checked, the mixed view is displayed in notebook format, with a tab for each source file that is included in the object file.

RELATED CONCEPTS
"Debugger Windows" on page 440

Problems Getting a Source or Mixed View

If you are in the **disassembly** view of a section of code, you may find that you cannot obtain a source or mixed view of your code (either through the **View** menu

or when you click on the Source View icon). There are several likely reasons for such problems. Click on links in the list below for proposed solutions:

- The code you are debugging was not compiled with debug information, because the debugger is finding a different version of the executable than the one you compiled with debug information.
- The code you are debugging was not compiled with debug information, because it is not code you wrote, but code in a DLL or other object that your program uses.
- The code you are debugging was compiled with debug information, but the debugger cannot locate the source code. If you try to switch to a source or mixed view in this situation, a dialog opens so that you can enter the path of the source file. If you select Cancel, the view remains a disassembly view.

RELATED REFERENCES

"Debugger Is Using a Different Executable Version" on page 478
"Debugger Cannot Find Source Code" on page 478
"Environment Variables" on page 429
"Source Views for Code You Did Not Write"

Source Views for Code You Did Not Write

In the Source window, you may not be able to obtain a source view of an object if the current code being executed is not part of your program. (For example, you may have halted execution while in a DLL that does not contain debug information.) Check the object name that appears at the top of the Source window. If you do not recognize this name as an object of your own executable, you are probably debugging system code used by, but not compiled with, your program. If you have access to the source code for this object, you could recompile it with debug information so that you can obtain a source view of it.

How Step Commands Work in Different Views

In the **Source** window, the current view of your program affects how step commands work. In a source view, they operate on the basis of lines of source code (typically, one step per line of source code that contains executable code). In a mixed view, the debugger treats source code lines as comments; in both **Mixed and Disassembly** views, step commands operate on disassembly instructions (typically, one step per line of disassembled code).

When you step from a function displayed in source view into a call to a function that was compiled without debug information, the Source window for the called function appears in disassembly view, and therefore step commands in that window will operate on a disassembly-instruction basis.

RELATED REFERENCES

"Stepping and Functions" on page 454

Remote Debugging

Remote debugging lets you debug programs that are running on one system, using a VisualAge debugger running on another system.

Why Use Remote Debugging
You might want to use remote debugging for the following reasons:

- The program you are debugging is running on another user's system, and is behaving differently on that system than on your own. You can use the remote debug feature to debug this program on the other system, from your system. The user on the system running that program interacts with the program as usual (except where breakpoints or step commands introduce delays). You interact with the debugger, but not with the I/O of the program being debugged.
- It is easier to debug an application that uses graphics or has a GUI when you keep the debugger user interface separate from the application's GUI. Your interaction (or another user's interaction) with the application occurs on the remote system, while your interaction with the debugger occurs on the local system.
- The program you are debugging was compiled for a platform that the debugger user interface does not run on. You can use the remote debug feature to take advantage of the debugger user interface while debugging the remote application

Supported Communications Protocols and Platforms
The debugger supports the TCP/IP protocol to establish the communications link between the debugger and a debuggee program running on different systems.

Limitations of Remote Debugging
Remote debugging imposes the following limitations:
- **Halt** is not supported when remote debugging.
- **Browse** is not available when prompted for a source file path.

RELATED TASKS
"Start the Debugger and the Remote Program"
"Debug a Distributed Application" on page 494

Start the Debugger and the Remote Program

Note: This section describes how to debug a program running on one workstation from a debugger user interface on another workstation. To do remote, distributed debugging with Object Level Trace, see "Debug a Distributed Application" on page 494

To remotely debug a program, follow these steps:
1. On the machine where the program to be debugged will run, issue the brmtdbg command.
2. On the machine where the debugger will run, issue the bdbug command and provide the remote host name. Either specify the program you want to debug as the last argument of the command, or enter the program name in the Startup dialog that appears.

The brmtdbg command has the following syntax:

```
brmtdbg [-qprotocol=tcpip] [-qport=port] [-qsession=single|multi]
```

where:

-qprotocol=tcpip
: Specifies the communications protocol to use. Only the TCP/IP communications protocol is supported. This is the default protocol.

-qport=*port*
: Specifies the TCP/IP port used for the connection. If you specify

-qprotocol=tcpip but you do not specify a port, or if you do not specify a protocol (so that TCP/IP becomes the default protocol), the default port is 8000.

-qsession=single|multi
Specifies whether to support single session debugging or multiple session debugging. The default is single session.

The command for invoking the debugger has the following syntax when used for remote debugging:

bdbug [-qprotocol=tcpip] [-qport=*port*] -qhost=*remotehost* *remote-program* [*program-parameters*]

where:

-qhost=*remotehost*
Specifies the TCP/IP name or address of the host to connect to. This argument is required.

remote-program
Is the name of the executable program on the remote machine (including an optional path)

program-parameters
are any parameters you want to pass to the executable program

The communications options for the bdbug command are the same (and have the same defaults) as those for the brmtdbg command. Note that you do not specify a host name on the brmtdbg command because the remote machine initiates the connection.

Start the Debugger and the Remote Java Program

Note: This section describes how to debug a Java program running on one workstation from a debugger user interface on another workstation. To do remote, distributed debugging with Object Level Trace, see “Debug a Distributed Application” on page 494

To remotely debug an interpreted Java program, follow these steps:

1. On the machine where the program to be debugged will run, issue a bjdbug -qport=<service port> command. If you do not use the -qport option, the debugger will attempt to run on the local machine.
2. On the machine where the debugger will run, issue the bdbug -qhost=<remotehost> -port=<port> command. Either specify the program you want to debug as the last argument of the command, or enter the program name in the Startup dialog that appears.

The bjdbug command has the following syntax:

```
bjdbug [-help] [-multi] [-host=<hostname>] [-password=<password>]
[-qport=<service port>] [-qhost=<uidhost>] [-quidport=<uidport>]
[-quiport=<uidport>][-qtitle=<uidtitle>]
[-jvmargs=<args>][classname][parameters]
```

where:

-help Gives a list of options for the command

-multi Allows multiple debuggers to connect to the program being debugged.

-host=<hostname>
Sets the name of the host machine on which the Java interpreter session to attach to is running.

-password=<password>
Sets the password used to log in to the active Java interpreter session on the host machine. This is the password printed by the Java interpreter when it is invoked with the `-debug` option.

-qport=<service port>
Sets the port on the local machine where the debugger should

-qhost=<uidhost>
Sets the host name where the debugger will run. This will be a different host from that in the `-host` option when debugging

-quiport=<uidport>
Sets the port on the host set by the `-qhost` option for communication between the program being debugged and the debugger. The default is `8001`.

-qtitle=<uidtitle>
Sets the title to appear on the debugger.

-jvmargs=<args>
Overrides environment variable settings inherited from the command line session where you issued the `bjdbug` command. This is useful if you would like the debugger to use a different `CLASSPATH` than the one existing in your command line session.

classname
The fully qualify class name of the the class on the local machine. This class must be in a path listed in the **CLASSPATH** environment variable on the local machine.

parameters
Any parameters you want to pass the class.

The command for invoking the debugger has the following syntax when used for remote debugging:

`bdbug -qhost=`*<remotehost>* [`-qport=`*<port>*] [*remote-program*]
[*program-parameters*]

where:

-qhost=*<remotehost>*
Specifies the TCP/IP name or address of the host to connect to. This argument is required.

-qport=<port>
Specifies the TCP/IP port on the host to connect to. The default is `8001`. This should be the same as that set with the `-qport` option of the `bjdbug` command.

remote-program
: The fully qualify class name of the the class on the remote machine. This class must be in a path listed in the **CLASSPATH** environment variable on the remote machine.

program-parameters
: Any parameters you want to pass to the executable program.

Breakpoints

Breakpoints are markers you place in your program to tell the debugger to stop whenever execution reaches that point. For example, if a particular statement in your program is causing problems, you could set a breakpoint on the line containing the statement, then run your program. Execution stops at the breakpoint, before the statement is executed, and you can check the contents of variables, registers, storage, and the stack, then either step over (execute) the statement to see how the problem arises, or jump over (not execute) the statement to temporarily circumvent the problem.

For C++, the debugger supports the following types of breakpoints:

- **Line breakpoints** are triggered before the code at a particular line in a source file is executed.
- **Function breakpoints** are triggered when the function they apply to is reached.
- **Address breakpoints** are triggered before the disassembly instruction at a particular address is executed.
- **Storage change** breakpoints are triggered when the storage within a particular address range is changed. The range is typically a small power of 2 (for example, 4 bytes).
- **Load occurrence** breakpoints are triggered when a DLL is loaded into an application. This happens the first time a reference is made to a function within the DLL.

For interpreted Java, the debugger supports the following types of breakpoints:

- **Line breakpoints** are triggered before the code at a particular line in a source file is executed.
- **Method breakpoints** are triggered when the method they apply to is reached.

RELATED TASKS

"Set a Deferred Breakpoint" on page 449
"Set Multiple Breakpoints" on page 449
"Delete Breakpoints" on page 450
"Delete All Breakpoints" on page 450
"Modify Breakpoint Characteristics" on page 450
"Enable and Disable Breakpoints" on page 450
"Set and Delete Breakpoints from a Source Window" on page 447

Set Breakpoints

You can set breakpoints from the following windows:

- Breakpoints List window
- Source window
- Session Control window

RELATED CONCEPTS
"Breakpoints" on page 446

RELATED TASKS
"Set Multiple Breakpoints" on page 449
"Set Breakpoints in the Breakpoints List Window"
"Set and Delete Breakpoints from a Source Window"
"Set Function or Method Breakpoints from the Session Control Window" on page 448

Set Breakpoints in the Breakpoints List Window

To set breakpoints in the Breakpoints List window, open the window (if it is not already open), using one of the following methods:

- Type Ctrl+X from any main debugger window or monitor
- Select **List** from the **Breakpoints** menu of the Source window
- Click on the button in the Source window
- Position the cursor in the prefix area, click the right mouse button, and select **List** from the popup menu.

Then do the following:

1. From the Breakpoints List window, select the type of breakpoint you want to set from the **Set** menu.
2. Enter the information for the breakpoint in the dialog, and click on **OK** or press the **Enter** key.

You can set breakpoints from the Source window and the Session Control window as well. Unless you want to set multiple breakpoints with the same nonstandard optional parameters, you may find it easier to use the Source window for setting line breakpoints and address breakpoints, and the Session Control window for setting function or method breakpoints. Storage change breakpoints and load occurrence breakpoints can be set from any of the three windows.

RELATED CONCEPTS
"Breakpoints" on page 446

RELATED TASKS
"Set Multiple Breakpoints" on page 449
"Set and Delete Breakpoints from a Source Window"
"Set Function or Method Breakpoints from the Session Control Window" on page 448

Set and Delete Breakpoints from a Source Window

You can set and delete breakpoints from a **Source** window in the following ways:

- Toggle a breakpoint on or off for a line. There are three ways to do this:
  - Double-click on the prefix area for that line.
  - Highlight the line using either the up and down cursor keys or the mouse, then press the space bar.
  - Highlight the line, then select **Breakpoints - Toggle at current line**.
- From the prefix area, or while the pointer is over a function name, click the right mouse button and select a breakpoint option from the pop-up menu.

- Open a Breakpoints List window, and set or delete the breakpoint from there.
- Select a type of breakpoint from the **Breakpoints** menu, and enter the appropriate information.

Note: If you delete a breakpoint, you must recreate it if you want to use it again. If you plan to use a breakpoint again later, it is better to disable the breakpoint instead.

RELATED TASKS
"Open Other Debugger Windows from a Source Window" on page 459
"Enable and Disable Breakpoints" on page 450

Set Function or Method Breakpoints from the Session Control Window

You can set **function breakpoints** or **method breakpoints** in the **Session Control** window by either of the following methods:

- Go to the **Breakpoints** menu and choose **Set function breakpoint** or **Set method breakpoint**
- Select a function in the **Components** pane, click the right mouse button to bring up a popup menu, and choose **Set function breakpoint** or **Set method breakpoint**.

RELATED TASKS
"Set Breakpoints" on page 446

Set a Line Breakpoint

You can set a line breakpoint from a **Source** window, or from the Breakpoints List window.

There are three ways to set a line breakpoint from the **Source** window:

1. Make sure the appropriate line is visible in the window (use the scroll bar or cursor keys to locate it), then double-click on the line number in the prefix area of the line.
2. Type the line number in the source window. A **Scroll to Line Number** dialog opens up. The line number you entered is placed in the dialog entry field. Click on the **Breakpoint** button.
3. Select **Breakpoints - Set line** from the menu bar, and fill in appropriate fields in the resulting Line breakpoint dialog box as described below for the Breakpoints List window.

To set a line breakpoint in a Breakpoints List window, do the following:

1. Select **Set - Line** from the menu bar.
2. From the **Executable** pulldown list in the Line breakpoint dialog, choose the DLL you want to debug.
3. From the **Source** pulldown list choose, the source file containing the code you want to debug.
4. In the **Line** entry field, enter the line number within the source file you want to place a breakpoint in.
5. Click on the **Defer breakpoint** check box.
6. Click **OK** to set the breakpoint and dismiss the Line breakpoint dialog.
7. Issue the **Run** command by pressing Ctrl+R.

RELATED CONCEPTS
"Breakpoints" on page 446

RELATED TASKS
"Set Breakpoints" on page 446
"Set Breakpoints in the Breakpoints List Window" on page 447
"Set and Delete Breakpoints from a Source Window" on page 447

Set a Deferred Breakpoint

A deferred breakpoint is a breakpoint set in a DLL that is not currently loaded.

To set a deferred line breakpoint, do the following:

1. Check the **Defer breakpoint** check box.
2. Open a Line breakpoint dialog in one of the following ways:
   - From a Source window or the Session Control window, select **Breakpoints - Set line**
   - From a Breakpoints List window, select **Set - Set line**.
3. Choose the DLL from the Executable pull-down field.
4. Choose the source file from the Include File pull-down field.
5. Enter the line number where you want the breakpoint set.
6. Specify any additional information you wish in the **Optional Parameters** group box.
7. Click on **OK** or press **Enter**.

RELATED TASKS
"Debug a DLL" on page 451
"Start Debugging a DLL from a Load Occurrence Breakpoint Dialog" on page 451
"Start Debugging a DLL from a Source Window" on page 452
"Start Debugging a DLL from the Session Control Window" on page 452

Set Multiple Breakpoints

You can set several breakpoints with the same optional parameters from a *Type* Breakpoint dialog, where *Type* corresponds to the type of breakpoint (Line, Function, Address, Storage change, Load occurrence). To do this, follow these steps:

1. From the breakpoint dialog, enter the information for the first breakpoint. Change any fields in the **Optional Parameters** section of the window, as desired.
2. Click on **Set**. The settings are saved for the current breakpoint.
3. For each additional breakpoint, enter only the changed information (for example, the new line number, new function name, new address), and again click on **Set**.
4. After you have set the last breakpoint, click on **Cancel** to dismiss the dialog.

RELATED TASKS
"Set Breakpoints" on page 446

Delete Breakpoints

To delete all breakpoints at once, select **Edit - Delete all** from the Breakpoints List window, or **Breakpoints - Delete all** from the Session Control or Source window. To delete individual breakpoints, position the pointer over the desired breakpoint, click the right mouse button, and select **Delete**.

When you delete a breakpoint, all information about it is lost. If you want to temporarily deactivate a breakpoint and activate it later, you should disable the breakpoint instead.

RELATED TASKS
"Enable and Disable Breakpoints"

Delete All Breakpoints

You can delete all breakpoints for your program from the Breakpoints List window in three ways:

- Click on the toolbar button
- Select **Delete all** from the **Edit** menu
- Position the pointer on a breakpoint in the list, click the right mouse button, and select **Delete all** from the popup menu.

Once you have deleted breakpoints, all information on them is lost. If you want to temporarily prevent breakpoints from stopping execution, but you may want to activate them later, disable them instead, by selecting **Edit - Disable all**.

RELATED TASKS
"Enable and Disable Breakpoints"

Enable and Disable Breakpoints

You can disable a breakpoint so that it does not stop execution, and then later enable it again. The advantage of disabling a breakpoint instead of deleting it is that it is easier to enable a breakpoint than to recreate it. The debugger lets you:

- Enable breakpoints one at a time
- Disable breakpoints one at a time
- Enable all breakpoints at once
- Disable all breakpoints at once.

RELATED TASKS
"Delete Breakpoints"
"Modify Breakpoint Characteristics"
"Set Breakpoints" on page 446

Modify Breakpoint Characteristics

You can change the following characteristics of a breakpoint:

- Which threads the breakpoint applies to
- How often the debugger should skip the breakpoint (the frequency)
- Whether to stop on the breakpoint only when a given expression is true

- Whether to defer the breakpoint (for use with DLLs)

You can also change the **Required parameters** fields for a breakpoint, which amounts to deleting the existing breakpoint and setting a new one.

To change a breakpoint's characteristics, select the breakpoint in the Breakpoints List window and choose **Edit - Modify**, or place your pointer over the breakpoint, click the right mouse button, and select **Modify**.

A *Type* Breakpoint window opens, where *Type* corresponds to the type of breakpoint (Line, Function, Address, Storage change, Load occurrence), with the current settings for the breakpoint already placed in the window's fields. For information on individual fields within each window, see that window's help.

RELATED CONCEPTS
"Breakpoints" on page 446

RELATED TASKS
"Set a Deferred Breakpoint" on page 449

Debug a DLL

To debug functions within a DLL, you first need to do the following:

1. Compile the DLL source files with symbolic information, if you have access to them and you want to be able to debug DLL functions at the source code level.
2. If you do not already have an executable program that calls the DLL functions you want to debug, write such a program and link it to the DLL. See your compiler documentation for information on how to do this.

RELATED TASKS
"Set a Deferred Breakpoint" on page 449
"Start Debugging a DLL from a Load Occurrence Breakpoint Dialog"
"Start Debugging a DLL from a Source Window" on page 452
"Start Debugging a DLL from the Breakpoints List Window" on page 452
"Start Debugging a DLL from the Session Control Window" on page 452

Start Debugging a DLL from a Load Occurrence Breakpoint Dialog

You can set a breakpoint in a DLL that causes execution to stop when the DLL is first loaded by your application. The DLL is generally loaded by your application the first time a function contained within the DLL is called by another function within your application. This type of breakpoint is called a **load occurrence breakpoint**.

To set a load occurrence breakpoint, do the following:

1. From the **Breakpoints** menu of the **Session Control** or **Source** windows, or from the **Set** menu of the **Breakpoints List** window, choose **Set load occurrence breakpoint**.
2. In the **Load Occurrence Breakpoint** dialog, enter the name of the DLL you want to set the breakpoint in.
3. Click on **OK** to set the breakpoint and close the **Load Occurrence Breakpoint** dialog, or click on **Set** if you want to set more load occurrence breakpoints.

When you run your application, execution will stop when the DLL is first loaded.

RELATED TASKS
"Debug a DLL" on page 451

Start Debugging a DLL from a Source Window

You can start debugging a DLL used by your application from the **Source** window containing the current execution point, provided you can use step commands to reach a statement that calls a function within your DLL.

1. In the **Source** window, use a combination of breakpoints and **Run**, **Step Debug**, and **Jump to Location** commands to execute or skip over portions of your code, up to the call to the DLL function you want to debug.
2. Issue a **Step Into** or **Step Debug** command to enter the DLL function.

A Source window showing the DLL function is raised, and the current line is the first line within the function.

RELATED TASKS
"Debug a DLL" on page 451
"Set Breakpoints" on page 446
"Run, Step Through, or Stop a Program" on page 453

Start Debugging a DLL from the Breakpoints List Window

You can start debugging a DLL from the **Breakpoints List** window by setting a line breakpoint in that DLL, then issuing the Run command. You can only do this if the compile unit you want to place the breakpoint in was compiled with debug information. To set a line breakpoint in a DLL and then start debugging the DLL, do the following:

1. Open a Breakpoints List window.
2. From the **Set** menu, choose **Set line.**
3. From the **Executable** pulldown list in the **Line breakpoint** dialog, choose the DLL you want to debug.
4. From the Source pulldown list choose, the source file containing the code you want to debug.
5. In the **Line** entry field, enter the line number within the source file you want to place a breakpoint in.
6. Click on the **Defer breakpoint** check box.
7. Click **OK** to set the breakpoint and dismiss the **Line breakpoint** dialog.
8. Issue the **Run** command by pressing Ctrl+R.

RELATED TASKS
"Debug a DLL" on page 451
"Set a Deferred Breakpoint" on page 449
"Set a Line Breakpoint" on page 448

Start Debugging a DLL from the Session Control Window

You can use the **Session Control** window to access a Source window containing the source for a DLL you want to debug.

1. In the **Session Control** window, expand the Components Pane entry for the DLL you want to debug.

2. Continue expanding appropriate items in the list until you find the source file or function you want to debug. If the source file or function is not listed, you may not have compiled the function with debug information.
3. Double-click on the source file or function to raise a Source window for it.
4. Set a line, function, or other breakpoint at an appropriate point in that **Source** window.
5. Run your program.

RELATED TASKS
"Debug a DLL" on page 451
"Run a Program"
"Set Breakpoints" on page 446

Run, Step Through, or Stop a Program

You can use debugger toolbar buttons, accelerator keys, and menu commands to accomplish the following subtasks:

- Run your program
- Step through your program
- Halt execution of your program
- Restart your program
- Terminate a debug session

RELATED TASKS
"Run a Program"
"Step Commands"
"Halt Execution of a Debuggee Program" on page 455
"Restart Your Program" on page 455
"Terminate a Debug Session" on page 456

Run a Program

After you have set appropriate breakpoints in your program, you are ready to issue the **Run** command. The debugger runs the program from the current execution point to the first enabled breakpoint it encounters. A Source window indicates the line where execution of the program stopped. You can then examine the contents of storage, variables, the call stack, or processor registers; step through your program; or issue the Run command again to run to the next enabled breakpoint.

RELATED TASKS
"Set Breakpoints" on page 446
"View Variables, Memory, Registers, and the Stack" on page 460

RELATED REFERENCES
"Step Commands"

Step Commands

You can use **step commands** to step through your program a single line or disassembly instruction at a time. You can issue step commands from the Source window that contains the current execution point.

The following types of step commands are available:

- **Step Over** - executes the current line, without stopping in any functions called within the line
- **Step Into** - executes the current line. If the current line contains a call to a function, execution stops in the first source line or disassembly instruction of the called function. If the called function was not compiled with debug information, the function is shown in a disassembly view.
- **Step Debug** - executes the current line. Execution stops at the next line encountered for which debug information is available. This could be in the current function, in the called function, or in a function called within the called function.
- **Step Return** - executes from the current execution point up to the line immediately following the line that called this function. If you issue a Step Return command from the **main** function, the program runs to completion.

Note that execution may stop earlier than indicated above, if the debugger encounters a breakpoint or an exception.

If the step command involves any calls to Object Request Broker methods on a server, a new set of debugger windows for the server code opens (or an existing set of debugger windows has its focus raised) and execution stops at the first statement in the method invoked on the server.

You can use combinations of step commands to step through multiple calls on a single line.

RELATED REFERENCES
"Stepping and Functions"

Stepping and Functions

Note: This section does not apply to statements containing method calls to remote Object Request Broker methods. Any step command that involves a call to an Object Request Broker method for which debugging is requested by the Object Level Trace, causes a different debugger user interface to start up (or be raised if it was started on a previous such call), and execution stops, in that debugger user interface, at the first source statement of the called method.

In a source code line that contains multiple calls, you can choose to step over all the calls, or step through the calls individually. Given a complex C++ call such as func1(func2(), func3());, you can do the following:

- Step over the entire line with a single **Step Over** command.
- Step through each called function with a series of **Step Into** commands. You can then step through the function, or, to return to the original statement so that you can step into the next function, issue a **Step Return** command.
- Step into each called function for which debug information is available, with a series of **Step Debug** commands. Each time you use Step Debug to step into such a function, you can then step through the function, or issue a Step Return command to return to the original statement.

You can also use a combination of step commands and function or method breakpoints to more finely control which functions called from a given line are stepped into and which are stepped over.

RELATED TASKS
"Set and Delete Breakpoints from a Source Window" on page 447

Skip over Sections of Code

You can change the current execution point in a debugging session to another location in the current Source window, so that certain lines are executed again, or are skipped over when they otherwise would not be. From within the Source window containing the current execution point, do the following:

1. Scroll to the line number you want to jump to, if it is not already visible.
2. Issue the **Jump to location** command by doing one of the following:
   - Click on the prefix area for the line, click, and select **Jump to location** from the popup menu
   - Select the line, then select **Run - Jump to location** from the menu bar
   - Select the line, then press the **N** key, which is the accelerator for the Jump to location command.

Note that using Jump to location can cause unpredictable results if you jump outside the current function, jump over code that has side-effects (for example, calls to functions whose results are assigned to variables, or functions that change the contents of variables passed by reference), or jump into the middle of a block such as a **for** loop.

Halt Execution of a Debuggee Program

To halt execution of a debuggee program that is currently running, do one of the following:

- From a Source window or the Session Control window, select **Run - Halt**
- From any window that has toolbar buttons displayed, click on the button

You may find that execution halts in a function other than the one you are debugging (for example, a system library function). To run to the end of that function and stop in your own code, do one of the following:

- Issue the **Step return** command from the Source window execution stopped in
- If the previous technique results in the debugger displaying the message "Cannot determine return address", issue the **Step debug** command until execution returns to your code
- If you know what line in your program will be the next to execute after the current function returns, go to the source window containing that line, set a breakpoint on it, and issue the **Run** command.

RELATED TASKS
"Set a Line Breakpoint" on page 448
"Terminate a Debug Session" on page 456

Restart Your Program

Note: You cannot restart a program that uses Component Broker objects. You cannot restart a program that you attached to with the debugger.

You can start debugging your program again from the beginning (the start of the **main** function) by doing the following:

1. If the program is currently executing within the debugger, issue a Halt command by selecting Run - Halt from a Source window or the Session Control window, or pressing the button.
2. Set a breakpoint at the location you want to run to, if it is not the start of the **main** function and you have not already set a breakpoint there.
3. If the previous run of your program performed file output and the program logic will be changed by the existence of such files from a previous debug session, you may want to erase these files before restarting.
4. Select Run - Restart from a Source window or the Session Control window.
5. If you want to run up to a breakpoint, issue the Run command by pressing Ctrl+R from any window (or R from a Source window) or pressing the button.

RELATED TASKS
"Halt Execution of a Debuggee Program" on page 455
"Set Breakpoints" on page 446

Terminate a Debug Session

To terminate a debug session and exit the debugger, do one of the following:

- Select **File - Close debugger** from any debugger window that has a **File** menu.
- Press **F3** from any debugger window that is not a dialog
- Switch to the Session Control window and double-click on the upper left corner of the window or press **Alt+F4**.

To terminate a debug session and start another one (local debugging only) do *one* of the following:

- If you want to start the same program executing again, select **Run - Restart** from a Source window or the Session Control window.
- Run the current program to completion, if this is feasible. A message window with the text "Program has run to completion" appears. Click on **OK**. A Startup dialog then appears.
- Select **File - Startup** from a Source window or the Session Control window.

Note: You cannot use Run - Restart or Startup when running a program that uses Object Request Broker objects.

The debugger locks the load modules for your program from write updates by the compiler until execution completes. If you want to recompile your program and debug it again, you should run the program to completion, exit the debugger, or switch to debugging a different program before recompiling.

RELATED TASKS
"Halt Execution of a Debuggee Program" on page 455

Debugger Monitors

Local Variables Monitor
The Local Variables monitor helps you monitor all variables within the current scope of a Source window. This monitor is associated with a particular thread, and closes automatically when that thread terminates. It is updated, after each Step or Run command, to show what variables are currently in scope and the contents of those variables.

Popup Monitor
A Popup monitor displays a variable or expression you select for monitoring. This monitor is associated with a specific Source window and closes when the associated window closes. Each time you add a variable or expression to a Popup monitor, a new Popup monitor opens. The contents of each Popup monitor are updated after each Step or Run command (except for disabled variables or expressions within such windows).

Private Monitor
A Private monitor lets you monitor variables and expressions that you select from a specific Source window. Private monitors help you keep track of local variables and expressions in programs with multiple compilation units. They are particularly useful in cases where the number of variables or expressions is large, or where variables with the same name appear in different compilation units.

Program Monitor
The Program Monitor shows variables and expressions that you select from Source windows. This monitor is not associated with any particular Source window, and remains open until you close it directly or exit the debugger. Use it to monitor global variables or variables you want to see at all times during your debugging session.

Registers Monitor
The Registers monitor shows the contents of processor registers for a particular thread in your program. If you are debugging multiple threads, you can display a separate Registers monitor for each thread. Although all threads share the same set of registers, the operating system saves the register contents of each thread as the thread is suspended, and restores that thread's processor contents when the thread resumes.

Storage Monitor
The Storage monitor lets you view and update the contents of storage areas used by your program. You can specify a variable, array, class object (C++ only), expression, or storage address to view. You can also change the address range to view, modify the contents of storage, and change the representation the debugger uses to display storage, for example, from hexadecimal to floating-point.

RELATED TASKS
"Add Expressions and Variables to a Monitor" on page 458
"Change the Contents of Storage, Variables, and Registers" on page 463
"Debug Heap Use" on page 465
"Edit Variable Contents" on page 461
"Open a New Storage Monitor" on page 458
"View a Location in Storage" on page 461
"View Variable Contents" on page 460
"View Variables, Memory, Registers, and the Stack" on page 460

RELATED CONCEPTS
"Differences between Program and Private Monitors"
"Debugger Windows" on page 440

Differences between Program and Private Monitors

The Program and Private monitors are very similar, but have the following differences:

- The Program monitor can be used to monitor all variables, while a Private monitor is associated with a single Source window.
- The Program monitor remains open at all times unless you close it or exit the debugger. A Private monitor closes whenever the Source window it is associated with closes, and is hidden whenever its Source window becomes inactive (for example, when a statement in that Source window returns control to code in another Source window).
- Because a Private monitor is associated with a specific Source window, whose focus is raised whenever you select the monitor, it does not have a **Windows** menu.

RELATED TASKS
"Add Expressions and Variables to a Monitor"
"Change the Contents of Storage, Variables, and Registers" on page 463
"View a Location in Storage" on page 461
"Open a New Storage Monitor"
"View Variables, Memory, Registers, and the Stack" on page 460
"Debug Heap Use" on page 465
"View Variable Contents" on page 460

Add Expressions and Variables to a Monitor

From the **Source** window, you can add a variable or expression to a monitor, so that you can keep track of how the variable's contents or the expression's value changes during program execution. You can use any of the following methods:

- Position the pointer over the variable, click the right mouse button, and select the monitor you want from the popup menu.
- Highlight the variable or expression, then select **Monitors - Monitor expression**. A dialog opens containing the variable or expression; select a monitor from there.
- Select **Monitors - Monitor expression**. In the dialog, enter the variable or expression and choose a monitor.
- Double-click on a variable to add it to the Program monitor.

RELATED REFERENCES
"C++ Expressions Supported" on page 474
Right Mouse Button Behavior

Open a New Storage Monitor

You can bring up a new **Storage** monitor in a number of ways:

- Click on the toolbar button from a Source window or the Session Control window.
- Select **Monitors - Storage** from a Source window or the Session Control window.

- Highlight an expression in a Source window, click the right mouse button, and select **Add to storage monitor** from the popup menu.
- Select **Monitors - Monitor expression** from a Source window or the Session Control window, enter the expression you want to monitor, and select the **Storage monitor** radio button.

A new storage monitor opens each time you do one of the above, even if you already opened another storage monitor for the same Source window, variable or expression.

RELATED TASKS
"Open Other Debugger Windows from a Source Window"
"Change the Storage Monitor Address Range"

Change the Storage Monitor Address Range

You can change the address range you want displayed in a **Storage** monitor to a specified address. Double-click on an entry in one of the address columns (for example, the "Flat" column), or move the cursor to that entry and press Enter; then enter a new address. You can also scroll through the range using the cursor keys (including Page Up and Page Down), or using the scrollbar.

If the debugger cannot determine the contents of storage at a particular location, it displays the contents as a series of question marks (?). This can occur when you display storage that your program does not own or when the address range does not point to a valid storage area.

If you want to view the storage associated with a different variable, adding that variable to a new Storage monitor is easier because the debugger determines the variable's address in storage for you. You may want to close the old Storage monitor first if its contents are no longer needed.

RELATED TASKS
"Add Expressions and Variables to a Monitor" on page 458
"Change the Contents of Storage, Variables, and Registers" on page 463
"Change the Representation of Storage" on page 462
"Open a New Storage Monitor" on page 458
"View a Location in Storage" on page 461

RELATED CONCEPTS
"Debugger Monitors" on page 457

Open Other Debugger Windows from a Source Window

You can open other Source windows from a Source window or from the Session Control window.

To open a **monitor** window from a Source window, click on a **monitor toolbar** button or select the desired monitor from the **Monitors** menu. You can also open a monitor window by adding a variable to that monitor, in one of the following ways:

- Click the right mouse button on the variable in the Source window, then select the appropriate monitor from the popup menu.
- Double-click the left mouse button on the variable to add the variable to the Program monitor.

- Click the left mouse button on the variable, press Ctrl+M, and select the desired monitor from the Monitor Expression dialog.

To open the **Breakpoints list** window, select **Breakpoints - List** from the menu bar, or press Ctrl+X.

To change the focus from a Source window to another window that is already opened (or minimized), select that window from the **Windows** menu.

RELATED TASKS
View Different Source Files
“Add Expressions and Variables to a Monitor” on page 458

View Variables, Memory, Registers, and the Stack

Follow one or more of the tasks below:
- Add a variable or expression to a monitor so you can view or change its contents
- View variable contents
- View storage contents
- View the contents of registers
- View the contents of the call stack

View Variable Contents

To view the contents of a variable you have already added to a monitor, do the following:
1. Select a window that contains a **Windows** menu (for example, a Source window or the Session Control window).
2. Select the monitor containing the variable from that **Windows** menu.
3. If necessary, use the scroll bars or PageUp and PageDown keys to scroll the monitor until the variable is visible.
4. If necessary, change the representation of the variable: click on the variable with the right mouse button and select **Next representation** from the popup menu.

To view the contents of a variable you have not yet added to a monitor, do one of the following:
- Add the variable to a monitor (see under Procedures below).
- If the variable is in scope in the current Source window, select **Monitors - Local variables** from the menu bar of the Source window to open a Local variables monitor. This monitor should contain all local variables that are currently in scope.

RELATED TASKS
“Add Expressions and Variables to a Monitor” on page 458
“Change the Contents of Storage, Variables, and Registers” on page 463
“View a Location in Storage” on page 461
“Open a New Storage Monitor” on page 458
“View Variables, Memory, Registers, and the Stack”
“Debug Heap Use” on page 465

Edit Variable Contents

To edit the contents of a variable in a monitor, do one of the following:

- Select the line containing the variable, click the right mouse button, and select **Edit** from the popup menu
- Double-click on the content of the variable
- Use the cursor keys or the mouse to highlight the line containing the variable and press Enter.

Then enter a new value and press Enter. This value must be valid for the type of variable.

RELATED TASKS
Change the Contents of Storage, Variables, and Registers

View a Location in Storage

To view a particular location in storage (for example, the storage used by a variable), either select **Options - Monitor expression** from the Storage monitor, or open a new Storage monitor from a Source window or the Session Control window, and specify an expression to monitor. See the Reference section below for information on what types of expressions the debugger supports.

C or C++: To view the storage for a class object or a variable, specify the address of the object or variable name by preceding the name with an ampersand (&).

RELATED REFERENCES
“C++ Expressions Supported” on page 474

RELATED TASKS
“Change the Contents of Storage, Variables, and Registers” on page 463
“Add Expressions and Variables to a Monitor” on page 458
“Change the Representation of Storage” on page 462

View the Contents of Registers

You may want to view the contents of a single register, or of many registers at once.

View Contents of a Single Register
If you only want to view the contents of a small number of registers, enter the names of those registers as expressions in a Monitor Expression dialog and select what monitor you want them to appear in. For example, if you want to monitor the EAX register on an Intel machine, do the following:

1. From a Source window, select **Monitors - Monitor expression** from the menu bar, or press the Ctrl+M accelerator key, to open a **Monitor expression** dialog.
2. Enter the register name as the expression (EAX)
3. Choose what monitor you want the expression to appear in. Do not choose the storage monitor.
4. Click on the OK button or press Enter.

The register is displayed in the monitor you selected. Note that if the name of the register corresponds to the name of a variable in your program, the variable may be displayed instead of the register depending on the monitor you choose, the current scope, and the scope of the variable.

View Contents of Many Registers at Once
If you want to open a monitor showing the contents of all or most processor registers of the stopped thread, do the following:

1. Raise a **Source** window or the **Session Control** window.
2. Select **Monitors - Registers** or click on the pushbutton.
3. A **Registers Monitor** displays the contents of processor registers for the current thread.

RELATED TASKS
"Add Expressions and Variables to a Monitor" on page 458
"Change the Contents of Registers" on page 463
"Change Which Registers Are Displayed" on page 464
"Change the Layout of the Registers Monitor" on page 464

RELATED CONCEPTS
"Debugger Monitors" on page 457

View the Contents of the Call Stack

You can view information for the active functions on a thread's stack. A function is on the stack from the time it is called until after it returns.

To view stack information for functions, open or raise a Call Stack window in one of the following ways:

- If a Call Stack window is already open, access it from the **Windows** menu of any debugger window that contains this menu.
- Select **Monitors - Call stack** from a Source or Session Control window, or press **Ctrl+K**, or click on the toolbar button.

RELATED CONCEPTS
"Debugger Windows" on page 440

Change the Representation of Storage

To change the representation of storage in the **Storage** monitor, do one of the following:

- Select a representation from **Options - Display style**
- Click on the appropriate display style toolbar button: .

The change in representation does not affect any other open storage monitors, only the one in which you make the change.

RELATED TASKS
"Change the Contents of Storage, Variables, and Registers" on page 463
"Change the Storage Monitor Address Range" on page 459

Change the Contents of Storage, Variables, and Registers

To change the contents of **storage** in a Storage monitor, do the following:

1. Raise or open a Storage monitor (select **Storage monitor** from the **Monitors** menu of the Source or Session Control windows, or press Ctrl+G, or click on the button).
2. If the address whose contents you want to change is not shown, use the PageUp and PageDown keys to scroll to that address, or click on an entry in the address column and enter a new address.
3. Double-click on an entry in a data column in the monitor (or click on the entry and press Enter).
4. Type a value that is valid for the shown representation .
5. Press Enter.

To change the contents of a **variable** in a Local Variables, Program, Private, or Popup monitor, or a **register** in the Registers monitor, do the following:

1. Raise or open the appropriate monitor (from the **Monitors** menu of a Source window or the Session Control window, or using the appropriate accelerator key or toolbar button).
2. Double-click on the entry field that shows the contents of the variable or register you want to change.
3. Type a value that is valid for the current representation of that variable or register.
4. Press Enter.

RELATED REFERENCES

"Values that Are Valid for the Current Representation" on page 479

Change the Contents of Registers

Caution: Changing the contents of registers that affect program flow (for example, registers used to manipulate the stack) can destabilize the program you are debugging and may cause it to terminate abnormally.

You can change the contents of most registers in the **Registers** monitor as follows:

1. Either double-click on the current value of the register or move the cursor to that register and press Enter.
2. Type in a new value and press Enter.

If updating of the register is allowed and your entry can be evaluated to a value the register supports, the register is updated.

Note: On Intel platforms, you cannot change the contents of the CS (code segment) register, and you cannot enter values into floating-point registers when they display "Not used" instead of a value.

For flags, you can only enter values within the range supported by the flag, or expressions that evaluate within such a range. Flags usually only have two valid values: 0 and 1.

RELATED REFERENCES
Valid Entries for Registers
"C++ Expressions Supported" on page 474

Change Which Registers Are Displayed

You can choose what groups of registers are displayed in the **Registers** monitor by selecting **Options - Display style**. In the dialog that appears, select what items you want to display.

RELATED CONCEPTS
"Debugger Monitors" on page 457

RELATED TASKS
"Change the Contents of Registers" on page 463
"Change the Layout of the Registers Monitor"
"Display Floating-Point Register Contents"
"View the Contents of Registers" on page 461

RELATED REFERENCES
"Registers Monitor Split Bars" on page 465

Display Floating-Point Register Contents

If you are debugging an Intel-based application and you step over a source line containing floating-point arithmetic, you may find that the values of floating-point registers in the **Registers** monitor are not displayed. Instead, "Not used" appears beside each register. In fact, one or more of these registers is being used during execution of the source line, but once you have stepped over the source line, the register's contents have been written to a variable and the register is no longer in use. If you want to step over a floating-point statement and see a floating-point register's value before it is written to the variable, do the following:

1. Change from source view to mixed view (select **View - Mixed**).
2. Locate the source line containing the floating-point instructions. Look for a disassembly instruction between this source line and the next that contains a floating-point store instruction (for example FSTP), and place a breakpoint on that line.
3. Change back to source view.
4. Now when you step over the source line containing the floating-point arithmetic, you must issue two Step Over commands for the line instead of one (because the first Step Over command stops at the breakpoint you set in the mixed view). After the second Step Over command, you should see the value of the floating point register as it was before it was stored.

RELATED TASKS
"Change the Contents of Registers" on page 463
"Change the Layout of the Registers Monitor"
"Change Which Registers Are Displayed"

Change the Layout of the Registers Monitor

The layout of the **Registers** monitor can be changed in a number of ways. You can:

- Change the amount of space given to each group of registers using the Registers monitor split bars

- Change the following settings from the **Options - Display style** menu choice:
  – Which groups of registers are displayed
  – Whether register groups are displayed in columns or rows
  – Whether group titles are displayed
  – Whether split bar positions between groups are saved

RELATED TASKS
"Change the Contents of Registers" on page 463
"Change Which Registers Are Displayed" on page 464

RELATED REFERENCES
"Registers Monitor Split Bars"

Registers Monitor Split Bars

You can change the amount of space given to each group of registers in the **Registers** monitor, as follows:

1. Place the pointer on the split bar (a division between two panes of the window). The pointer's shape changes to an icon with two arrows.
2. Click and hold.
3. Drag the mouse up or down (for register groups displayed in rows) or left or right (for register groups displayed in columns) until you reach the desired size for the pane.
4. Release the mouse button.

To save the position of the split bars, select **Options - Display style** and click on the **Save Split Bar Positions** check box.

Debug Heap Use

If you suspect problems with heap use in your program, you can pinpoint likely causes of heap errors by following these steps:

1. Compile your program with the /Tm option so that the debug versions of memory management functions are used.
2. Run your program. Any errors detected by the debug memory management programs are written to standard error, with the source file and line number where the error was detected, and the source file and line number where the heap was last known to be uncorrupted. (You can also run your program within the debugger; in this case, the errors are shown in a popup window instead of on standard error.)
3. To further isolate a heap corruption error, debug the program, set a breakpoint at the source file and line number where the heap was last known to be uncorrupted, and run the program.
4. When execution stops at the breakpoint, enable the debugger's own heap checking functions by enabling **Run - Check heap when stopping** from a Source window or the Session Control window.
5. Step through your code, or set frequent breakpoints and run it. Each time the debugger stops, it causes your application to call the heap checking functions. When heap corruption is detected, an error message displays in a popup menu.
6. Each time you see a heap error message and are unable to pinpoint the exact location of the error, note the two line numbers provided, restart the program,

set a breakpoint at the line where the heap was last found to be uncorrupted, and use step commands or breakpoints and the **Run** command to further narrow down the location of the error.

RELATED REFERENCES
"Heap Errors"

Heap Errors

Heap errors can occur when your code inadvertently overwrites control information that the memory management functions use to control heap usage. Each block of allocated storage within a heap consists of a data area, which starts at the address returned by the allocating function, as well as a control area adjacent to the data area, which is needed by the memory management functions to free the storage properly when you deallocate the storage. If you overwrite a control structure in the heap (for example, by writing to elements outside the allocated bounds of an array, or by copying a string into too small a block of allocated storage), the control information is corrupted and may cause incorrect program behavior even if the data areas of other allocated blocks are not overwritten.

You should consider the following points when you are trying to locate heap errors:

Finding heap errors outside the debugger
To detect heap errors, you can compile your program to use the heap-checking versions of memory management functions (use the /Tm option). When you run a program compiled with this option, each call to a memory management function causes a heap check to be performed on the default heap. This heap check involves checking the control structures for each allocated block of storage within the heap, and ensuring that none were overwritten. If an error is encountered, the program terminates and information is written to standard error including the address where heap corruption occurred, the source file and line number at which a valid heap state was last detected, and the source file and line number at which the memory error was detected.

Heap checking for default and other heaps
Heap checking is only enabled for the default heap used by each executable. If the debug versions of the memory management functions do not report heap corruption and you still suspect a problem, you may be using additional heaps and corrupting them. You can debug usage of nondefault heaps by adding calls to the **_uheapchk** C Library function to your source code. See your compiler documentation for more information.

Pinpointing heap errors within the debugger
You can pinpoint the cause of a heap error from within the debugger, provided the heap causing the error is known to be the default heap, by continually narrowing down the gap between the last line at which the heap was valid, and the first line at which corruption occurred. From within the Source window, use a combination of run commands, step commands, line and function breakpoints, and the **Check heap when stopping** setting on the **Run** menu, to narrow the scope of your search.

Check heap when stopping may expose other coding errors
For semantically incorrect programs, **Check heap when stopping** is intrusive in that it may cause different results where a program is incorrectly accessing data on the stack. This is because **Check heap when stopping** causes the process and

thread being debugged to call a heap check function each time execution stops, and this heap check function affects the safe area of the stack by overwriting part of that area with its stack frame. For example, if a called function returns the address of a local variable, that local variable's contents will be accessible from the calling function, and will not change, as long as the stack frame used by the called function is not overwritten by a subsequent call. However, if you issue a Step return command from the called function while **Check heap when stopping** is enabled, the heap checking function is called immediately on return from the called function, and the storage pointed to by the returned pointer may overwritten by the stack frame of the heap checking function.

Check heap when stopping affects performance
Heap checking within the debugger has a high overhead cost for step commands, because the heap is checked after each step. If you are stepping through large sections of code, or frequently stopping at breakpoints, and you find debug performance too slow, try turning on **Check heap when stopping** only in those areas you suspect are causing heap errors.

Notes on Check Heap when Stopping

- For the Check heap when stopping choice to work, you have to compile your application using the VisualAge C++ /Tm+compiler option.
- If you enable the Check heap when stopping choice and run your application to termination, and the application contains a heap error, the heap check is not made. To check the heap just before termination, set a breakpoint on the last line of your application.

RELATED TASKS
"Debug Heap Use" on page 465

Debug Optimized Code

Problems that only surface during optimization are often an indication of logic errors that are exposed by optimization, for example using a variable that has not been initialized. If you encounter an error in your program that only occurs in the optimized version, you can usually find the cause of the error using a binary search technique to find the failing module:

1. Begin by optimizing half the modules and see if the error persists.
2. After each change in the number of optimized modules, if the error persists, optimize fewer modules; if the error goes away, optimize more modules. Eventually you will have narrowed the error down to a single module or a small number of modules.
3. Debug the failing module. If possible, turn off the instruction scheduling optimizations for that module. Look for problems such as reading from a variable before it has been written to, and pointers or array indices exceeding the bounds of storage allocated for the pointer or array.

RELATED REFERENCES
"Notes on Debugging Optimized Code" on page 468

Notes on Debugging Optimized Code

When you debug optimized code, information in debugger panes may lead you to suspect logic problems that do not actually exist. You should bear in mind the points below.

Values in some monitors may not be current
Do not rely on monitors such as the Local Variables or Popup monitors to show the current values of variables. Numeric and char values may be kept in processor registers, as may pointers to other types of variables such as strings and class objects. In the optimized program, these values and pointers are not always written out to memory; in some cases, they may be discarded because they are not needed.

Static and external variables are not always current
Static or external variables can be monitored at function entry and exit points. Within an optimized function, their values may be optimized out of existence.

Register and Storage monitors are always current
The register and storage monitors are correct. Unlike monitors that show actual variables, such as the Local Variables or Popup monitors, the Register and Storage Monitors are always up-to-date as of the last time execution stopped.

Source statements may be optimized away
Use the disassembly view of your program to see whether source statements whose result you were relying on have been optimized away (via dead code elimination, where code that performs no useful work is removed). You may find, for example, that an assignment to a variable in your source code does not result in any disassembly code being produced; this may indicate that the variable's value is never used after the assignment.

RELATED TASKS
"Debug Optimized Code" on page 467

Debugging Threads

Note: The sections on debugging threads are intended primarily for users with limited knowledge of developing multithreaded programs, in particular, those whose programming experience is mainly with single-threaded environments such as Windows 3.1.

Multithreaded programs may behave differently in the debugger than they do when run normally. Because debugger features such as single-stepping and certain kinds of breakpoints involve processing overhead, running a multithreaded program within the debugger may affect the timing of thread switches.If you are experiencing problems, you can use the Threads pane of the Session Control window to enable or disable threads so that you can debug problems related to thread timing.

The main problems you are likely to encounter in debugging multithreaded programs are timing and deadlock problems. You should not assume that a timing or deadlock problem that seems only to occur when your program is running within the debugger will never occur outside of the debugger. It is far more likely that the processing overhead of the debugger is merely increasing the frequency with which coding problems lead to deadlocks or thread timing errors.

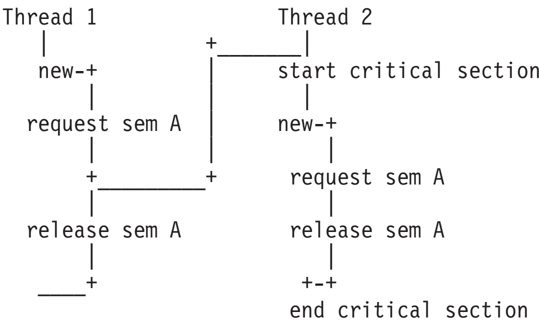
RELATED CONCEPTS

"Critical Sections"
"Deadlocks and Timing Problems" on page 470
"Must Complete Sections" on page 471
"Race Conditions" on page 471
"Threads and C++ Class Members" on page 472
"Threads and Load Occurrence Breakpoints" on page 472
"Threads and Source Language Statements" on page 473
"Windowing System Lockups" on page 473

Critical Sections

Windows supports the use of critical sections to mark particular sections of code that should never be timesliced out within a multithreaded application. The purpose of these critical sections is to prevent problems such as loss of data integrity (where two threads are simultaneously reading and then writing to the same variable or file). You should avoid using critical sections except under very limited circumstances. Critical sections can cause deadlocks and major timing problems, because:

- They block every thread within your application, other than the thread containing the critical section, from doing anything. This prevents an efficient distribution of system resources to all threads.
- If you call a function within a critical section (explicitly through a call, or implicitly by using the C++ **new**, **delete**, or user-defined operators for a variable of class type, such as an IString object) the called function may request a semaphore that is already locked. This thread then locks, but because it is declared as a critical section, the thread that owns the semaphore can never be timesliced back in to release the semaphore. For example:



In this example, Thread 1 calls the **new** operator, which requests semaphore A. The operating system timeslices it out while it owns semaphore A. Thread 2 gets timesliced in, declares a critical section, and calls **new**, which requests the same semaphore A (this semaphore is an operating system semaphore used to prevent two threads from simultaneously allocating storage). Because semaphore A is already owned by thread 1, thread 2 waits for the semaphore indefinitely. Thread 1 can never be timesliced back in to release semaphore A, because thread 2 is in a critical section; and thread 2 can never exit its critical section, because it is frozen waiting for semaphore A.

Semaphores and critical sections provide some of the same functionality, but using only semaphores is a better programming practice and leads to more threadsafe programs. Avoid the use of critical sections in your programs wherever possible.

RELATED CONCEPTS

"Debugging Threads" on page 468

Deadlocks and Timing Problems

Thread deadlocks can occur in a multithreaded program running inside the debugger, even though they never seem to occur when the program runs outside the debugger. Consider a program with two threads running, in which the threads both request two mutex semaphores but in different orders:

```
THREAD 1                                  THREAD 2
Lots of code                              A bit of code, then request semaphore Y
Request semaphore X                       Request semaphore X
Lots of Code                              Release semaphores Y and X
Requests semaphore Y
Some code
Release sempahores X and Y
```

These two threads may deadlock. However, because the operating system may timeslice your program and the threads within it in an unpredictable fashion, such deadlocks may not become obvious until you try to debug the program. In the example, you may find that when run outside the debugger, Thread 2 can request and release both semaphores before Thread 1 has even finished its large initial code section; both semaphores may already be released by Thread 2 by the time they are requested by Thread 1. However, within the debugger, if you are setting breakpoints, stepping through code, or otherwise slowing down one thread compared to another, you may wind up finding the requests for the semaphores clashing and thereby blocking each other. For example, if you step through Thread 2, its short initial code section may take longer to execute than the long initial code section of Thread 1:

```
Thread 1: Lots of code
     Thread 2: A bit of code
.
     .
.
     .
Thread 1: Lots of code completes
     Thread 2: A bit of code completes
Thread 1: Request semaphore X (okay)
     Thread 2: Request semaphore Y (okay)
Thread 1: Lots of code, then request semaphore Y (waits)
     Thread 2: Request semaphore X (deadlock)
```

Because each thread is requesting a semaphore owned by the other thread, the two threads lock up.

To avoid this kind of lockup, always request a group of semaphores in the same order from every thread that uses them. It is also a good idea to release them in the reverse order from the request order. For example:

```
THREAD 1                 THREAD 2
Lots of code             A bit of code
Request semaphore X      Request semaphore X (waits)
Lots of code             (still waiting)
Request semaphore Y      (still waiting)
Release semaphore Y      (still waiting)
Release semaphore X      (obtains semaphore X)
                         Request semaphore Y (okay)
...                      ...
```

Regardless of which thread completes first, there is no deadlock because whichever thread requests X last will be forced to wait until X is freed, and X will not be freed until the other thread no longer requires either semaphore.

The debugger may expose semaphore-related deadlocks that do not normally occur, because of the following factors:

- When you step through a thread or run to a breakpoint within that thread, you affect the point at which thread swapping occurs
- The debugger needs to be informed of thread starting and stopping, exceptions, module loads, and so on, and its mere presence affects the dynamics of the operating system.

RELATED CONCEPTS
"Debugging Threads" on page 468

Must Complete Sections

When a program calls **exit()**, all threads in that program are normally terminated at their current execution point. If a thread holds a mutex semaphore, that semaphore changes state to "Owner died". Any code that subseqently tries to request that semaphore will fail.

RELATED CONCEPTS
"Debugging Threads" on page 468

Race Conditions

Note: The examples below assume the program in question was written in C++.

A **race** condition can occur in a multithreaded program when you do not use semaphores. A race condition is a situation where two threads are "racing" towards use of the same variable or some other data structure. For example, suppose Thread 1 contains the statement i=foo(i); and Thread 2 contains the statement a=b=i;. If you do not use a semaphore to block one thread, timeslicing could, in theory, result in either of the following sequences of events, among others:

```
Sequence 1:
Thread 2: Load value of i into register
Thread 1: Load value of i into register
Thread 2: Store value of i in register to b
Thread 1: Call foo(i) using value of i in register
Thread 2: Store value of i in register to a
Thread 1: Store result of foo(i) to i
Sequence 2:
Thread 1: Load value of i into register
Thread 1: Call foo(i) using value of i in register
Thread 1: Store result of foo(i) to i
Thread 2: Load value of i into register
Thread 2: Store value of i in register to b
Thread 2: Store value of i in register to a
```

If i is an integer with a value of 3 before the sequences begin, and foo(x) is an integer function that returns 3*x+1, the variables at the end of sequence 1 will be: a=1, b=1, i=10; the variables at the end of sequence 2 will be: a=10, b=10, i=10.

A race condition can occur even when you *do* use semaphores. In such a case, it may be an indication of incorrect program logic. Suppose, for example, that two threads both assign a value to a variable, and that a third thread reads the value of that variable:

```
Thread 1:        Thread 2:        Thread 3:
i=3;             i=4;             j=i;
```

Even if you request a semaphore before each assignment to i in threads 1 and 2, and release the semaphore after the assignment, there is no way of predicting whether j will be assigned the value 3 or 4 (or even the value of i before Threads 1 and 2 assigned to it). In this example, the race condition is simply poor programming logic.

You may want to use a race condition to determine which of two or more threads completed a given task first. For example, if the statement in Thread 3 was:

```
if (i==3) cout << "Thread 1 completed first" << endl;
else cout << "Thread 2 completed first" << endl;
```

and you had protected each assignment to i with a semaphore, the statement in Thread 3 would be reliable.

Remember to use semaphores not only on pointers to objects, but on the objects themselves. If two pointers point to the same object and you only use semaphores to lock the pointers, two different threads using different pointers can access the same object simultaneously.

You can use a Storage change breakpoint to find race conditions such as those shown above. By placing a Storage change breakpoint on the address of a variable, you can find all statements that change the variable and make note of the order in which different threads change it.

Race conditions are another example of a timing problem that may only occur when you are debugging your program, because the debugger may affect the order in which threads are accessing shared data.

RELATED CONCEPTS
"Breakpoints" on page 446
"Debugging Threads" on page 468

Threads and C++ Class Members

If you use the same C++ class in two different threads, you may have thread problems such as the following:

- If the class contains static variables, you may have re-entrancy problems.
- If you are using the same instance from two separate threads, the instance may be accessed by both threads at once, resulting in a loss of data integrity.

In both cases you need re-entrancy protection. For example, use a semaphore whenever you access the static variables or the common class instance.

RELATED CONCEPTS
"Debugging Threads" on page 468

Threads and Load Occurrence Breakpoints

If you set a load occurrence breakpoint for a DLL that has not been loaded, you may find that, in a multithreaded program, the DLL never triggers the breakpoint, even though it must have been loaded by a call to a function within it. The usual cause of a load occurrence breakpoint not triggering is that you associated the breakpoint with a particular thread.

To avoid this problem, choose “every” in the Optional Parameters group box in the Load Occurrence Breakpoint window, to have the load occurrence breakpoint trigger regardless of which thread first calls a function within it. The debugger tells you which thread triggered the load.

For all breakpoint types, you can specify a breakpoint at the same location but with different conditions for different threads. For example, you can set a breakpoint that is triggered in thread 1 when the variable a has the value 3, and a breakpoint at the same location triggered in thread 2 when the variable a has the value 4. Conditions on thread-specific breakpoints can be useful for determining whether you have thread data integrity problems.

RELATED CONCEPTS
“Debugging Threads” on page 468

Threads and Source Language Statements

It is possible for a single source-language statement to be interrupted in mid-statement by another thread. The statement: i++; might involve three machine language instructions: loading a variable from storage into a register (if it is not already in a register); incrementing the register contents; and storing the result back to memory. If i is a **double**, or a pointer to **struct**, for example, the increment itself may be broken up into several machine language instructions. The thread may be interrupted at any instruction’s completion point by another thread that also uses or changes the same variable, if you have not used a semaphore to lock during the increment.

Even if the machine code for a simple statement is a single instruction, you should avoid relying on this fact to provide data integrity. An increment of an integer variable requires a load, increment, and store; the load may have occurred on an earlier use of the variable within the same thread, and the store may occur some time later after another use of the variable. Thus the increment statement may only have a single instruction associated with it in the assembly listing, but another thread’s modifying that variable between the load and increment, or the increment and store, affects the data integrity of the variable.

RELATED CONCEPTS
“Debugging Threads” on page 468

Windowing System Lockups

Multithreaded programs have a greater tendency than single-thread programs to cause your windowing system to lock up, because of such problems as semaphore deadlocks and live threads waiting for results from threads that have inadvertently died. When the debugging of a multithreaded program hangs your windowing system, you have no way of continuing to debug the faulty program, because the debugger uses the windowing system as well.

To solve such problems, you can use the debugger’s remote debug feature. Install the debugger on both machines, start a remote debug server session on the machine you want to run the program on, and run the debugger from the other machine. If your program hangs the windowing system on its own machine (the remote machine), you can still step through it because the windowing system on your local machine is still operational.

RELATED CONCEPTS
"Debugging Threads" on page 468
"Remote Debugging" on page 442

Troubleshooting and Limitations

C++ Expressions Supported

The expression language that is supported by the debugger for C++ programs is a subset of the C/C++ language. You can only monitor expressions with:

- A "C++ Supported Expression Operands"
- A "C++ Supported Expression Operators" on page 475
- A "C++ Supported Data Types" on page 476

C++ Supported Expression Operands

You can monitor an expression that uses the following types of operands only:

Operand
: **Definition**

Variable
: A variable used in your program.

Constant
: The constant can be one of the following types:
  - Fixed or floating-point constant within the ranges supported by the system the debuggee is running on.
  - A string constant, enclosed in double quotation marks (for example, "mystring")
  - A character constant, enclosed in single quote marks (for example, 'x')

Register
: Any of the processor registers that can be displayed in the Registers Monitor. In the case of conflicting names, program variable names take precedence over register names. For conversions that are done automatically when the registers display in mixed-mode expressions, general-purpose registers are treated as unsigned arithmetic items with a length appropriate to the register. For example, on Intel platforms EAX is 32-bits, AX is 16-bits, and AL is 8-bits.

If you monitor an enumerated variable, a comment appears to the right of the value. If the value of the variable matches one of the enumerated types, the comment contains the name of the first enumerated type that matches the value of the variable. If the length of the enumerated name does not fit in the monitor, the contents appear as an empty entry field.

The comment (empty or not) lets you distinguish between a valid enumerated value and an invalid value. An invalid value does not have a comment to its right.

You *cannot* update an enumerated variable by entering an enumerated type. You must enter a value or expression. If the value is a valid enumerated value, the comment to the right of it is updated.

Bit fields are supported for C/C++ compiled code only. You can display and update bit fields, but you cannot use them in expressions. You cannot look at variables that have been defined using the `#define` preprocessor directive.

C++ Supported Expression Operators

You can monitor an expression that uses the following operators only:

Operator
: **Coded as**

Global scope resolution
: ::*a*

Class or namespace scope resolution
: *a*::*b*

Subscripting
: *a*[b]

Member selection
: *a.b* or *a - b*

Size
: `sizeof` *a* or `sizeof` (*type*)

Logical not
: !*a*

Ones complement
: ~*a*

Unary minus
: -*a*

Unary plus
: +*a*

Dereference
: **a*

Type cast
: (*type*) *a*

Multiply
: *a* * *b*

Divide
: *a* / *b*

Modulo
: *a* % *b*

Add
: *a* + *b*

Subtract
: *a* - *b*

Left shift
: *a* << *b*

Right shift
: *a* >> *b*

Less than
: *a* < *b*

Greater than
: *a* > *b*

Less than or equal to
a <= *b*

Greater than or equal to
a > = *b*

Equal *a* == *b*

Not equal
a != *b*

Bitwise AND
a & *b*

Bitwise OR
a | *b*

Bitwise exclusive OR
a ˆ *b*

Logical AND
a && *b*

Logical OR
a || *b*

C++ Supported Data Types

You can monitor an expression that includes a cast to any of the following types:
- 8-bit signed byte
- 8-bit unsigned byte
- 16-bit signed integer
- 16-bit unsigned integer
- 32-bit signed integer
- 32-bit unsigned integer
- 64-bit signed integer
- 64-bit unsigned integer
- Single-precision floating-point floating-point
- Double-precision floating-point floating-point
- Pointers
- User-defined types

These data types include **int**, **short**, **char** and so on.

Limitations when Debugging Visual C++ Programs

Note: This section applies only to programs being debugged on Windows NT.

You can debug C and C++ programs compiled with the Microsoft Visual C++ compiler, provided you have compiled and linked your program with the appropriate options. The following limitations apply when debugging such programs:

Enumerated types: Enumeration member name information for enumerated types in C programs compiled with Visual C++ is not shown in monitors that display variable contents, such as the Local Variables monitor. This information is available for C++ programs, however. The following shows a code fragment with enumerated types, and the values displayed in the Local Variables monitor of the debugger:

```
typedef enum { One=1, Two, Three} TypeX;
TypeX a=One;
TypeX b=Two;
TypeX c=Three;
```

If the program is compiled as a C program with Visual C++:

```
a: 1
b: 2
c: 3
```

If the program is compiled as a C or C++ program with VisualAge C++, or a C++ program with Visual C++

```
a: 1 /* One */
b: 2 /* Two */
c: 3 /* Three */
```

Constants: A statement such as `const int i = 42;` in a Visual C++ program does not generate any debug information for the variable i. Therefore the debugger does not display any value for it. The IBM VisualAge C++ compiler does generate information for this symbol.

Namespaces: You cannot debug Visual C++ namespaces because the Visual C++ compiler does not generate the necessary debug information for namespaces in the executable.

RELATED TASKS
"Debug a Microsoft Visual C++ Program" on page 435

Interpreted Java Expressions Supported

Only expressions using the dot (.) operator are supported. The dot operator is used to access instance and static variables within objects and classes.

Limitations When Debugging Interpreted Java

Limitations exist in the JDK that may cause problems when debugging your interpreted Java classes. Be aware of the following limitations and problems when debugging:

- The Java debugger does not accept input from stdin. You may want to replace use of stdin with a dialog box.
- `System.out` may not print Chinese, Korean, or Japanese characters correctly when run under debugger control.
- Stepping behavior may be erratic when stepping into constructors, or when stepping into or over `SystemLoad` library functions.
- You can not suspend, start, or stop threads.
- The debugger cannot halt an applet or application that has all of its thread blocked.
- You can not modify the contents of monitored variables.
- Breakpoints set on static initializers or static blocks will be ignored.
- Breakpoints set on `try` statements are ignored.
- Execution will not stop inside catch blocks for thrown errors.
- The debugger will not notify the user when classes extending from `java.lang.Error` are thrown. The debugger will notify the user when classes extending from `java.lang.Exception` are thrown.

- When debugging JAR files, the source code for classes contained in a JAR file must available outside of the JAR file.
- After exiting a program block, variables now out of scope may still be shown in a monitor.
- Debug-agent error messages may appear intermittently while debugging your classes.
- If you step into a class that has not been registered with the debugger, you may receive a **"Cannot find source for null"** message**.** If this happens, issue a step return command to continue debugging. To avoid this problem, register the appropriate class source file or package containing the class source with the debugger before you start debugging.
- When debugging remotely, communication between the debugger and the program being debugged may be terminated prematurely by the JVM.

Debugger Is Using a Different Executable Version

In the **Source** window, you may not be able to obtain a source view of an object if the debugger is finding a different version of the executable than the one you created with debug information. Set the **Show module path** check box in the **Options - Window Settings - Display Style** dialog of the Session Control window, so that full pathnames are displayed in the components pane of the Session Control window. Check that the modules listed are in the correct location. If they are not, you can exit the debugger, remove or rename the executable or DLL in the incorrect path so that the one in the correct path is accessed, and start debugging again. (For example, if there is an obsolete version of your executable in one directory, and an updated version in another, and the obsolete directory's entry in PATH precedes that of the updated directory, remove the executable from the obsolete directory.)

You can also specify an absolute path to the version you know was compiled with debug information, when you invoke the debugger or in the **Startup** dialog. To bring up the **Startup** dialog, choose **File - Startup** from within the Source or Session Control window, or start the debugger with no parameters.

Debugger Cannot Find Source Code

The debugger searches the workstation for source files using a search path based on environment variables, which you can set before starting your debug session. If you anticipate frequently having your source files on the workstation, you should set these environment variables.

In the **Source** window, you may not be able to obtain a source view of an object, even though the code was compiled with debug information, if the debugger cannot find the source files for it. When you start debugging such a program, or when execution lands in a part of the program that was compiled with debug information but the debugger cannot find the source code for it, the debugger normally opens a **Source Filename** dialog in which you can enter the location and name of the source file. If you choose **Cancel** when this dialog appears, the debugger displays a disassembly view of the code, because it has no source code to display. If the source file has been moved or renamed, select **View - Change text file**, and enter the correct path and name in the **Change text file** dialog. (The debugger searches the workstation for source files using a search path based on environment variables, which you can set before starting your debug session. If you anticipate

frequently having your source files in a different directory from your executables, you should set these environment variables.)

If you are debugging remotely, the debugger searches for the files in the path you specify, first on the workstation where the program being debugged is running, then on the workstation where the debugger user interface is running.

RELATED REFERENCES
"Environment Variables" on page 429

Values that Are Valid for the Current Representation

When you are entering a value in an entry field such as a register in the Registers monitor, a column of storage in the Storage monitor, or the contents of a variable in another monitor, the debugger checks that the value you enter is valid for the current representation of that column or entry field.

A value is valid for the current representation if it contains only the characters used for that representation, and does not exceed the length of the variable or register involved. For example, if you want to change the contents of storage, and the Content style setting for a particular Storage monitor is 32-bit integer, the value you enter must be a valid 32-bit integer, not a floating-point value or other value. Or, if you want to change the contents of a character string and the current representation is a text string, you must enter a new string in double quotes, and the length of the string must not exceed the declared array size.

RELATED REFERENCES
"C++ Expressions Supported" on page 474

RELATED TASKS
"Change the Contents of Storage, Variables, and Registers" on page 463

Valid Addresses and Expressions

Here are some examples of addresses or expressions you can enter in the **Addresses or expression**s field of the Set Storage Change dialog:

MyVariable (C++)
: The address pointed to by MyVariable, if such a variable exists in your program, and if its value is a valid storage address (for example, if the variable is a pointer to another variable or to an offset within an array).

&MyVariable (C++)
: The storage of the variable MyVariable.

A1FCC
: The hex address A1FCC, unless you have a variable declared as A1FCC, in which case it is treated the same as MyVariable above.

0xA1FCC
: The hex address A1FCC.

RegisterName
: The name of a processor register on the system the debuggee is running on. If the expression evaluates to a processor register, that register's contents are used in place of the register name.

RELATED REFERENCES
"C++ Expressions Supported" on page 474

Right Mouse Button Behavior

The behavior of the right mouse button depends on the **Mouse button 2 behavior** group box in the Debugger properties dialog group box.

One of three possible events occur when clicking the right mouse button:
1. Display a popup menu, where one is available; otherwise do nothing.
2. Perform a step over command.
3. Display a popup menu, when the pointer is over certain text (for example, a variable or a line number in the prefix area), and perform a step over command when the pointer is over white space or over text for which no popup menu is available.

C++ example: If **Popup menus and step in white space** is checked, and you click the right mouse button on the C++ source line int myVar=18;, a popup menu appears if the pointer is over int, myVar, or 18, and a step command is performed if the pointer is on white space or the = or ; symbols.

RELATED TASKS
"Change Right Mouse Button Behavior"

Change Right Mouse Button Behavior

To change the behavior of the right mouse button, select **Options - Debugger settings - Debugger properties** from the Source or Session Control window menus.

Then, select one of the following check boxes to determine how the right mouse button will behave in the Source window.

Popup menus
: Displays a popup menu, where one is available; otherwise does nothing.

Step always
: Performs a step over command.

Popup menus and step in white space
: Displays a popup menu, when the pointer is over certain text (for example, a variable or a line number in the prefix area), and performs a step over command when the pointer is over white space or over text for which no popup menu is available.

RELATED REFERENCES
Right Mouse Button Behavior

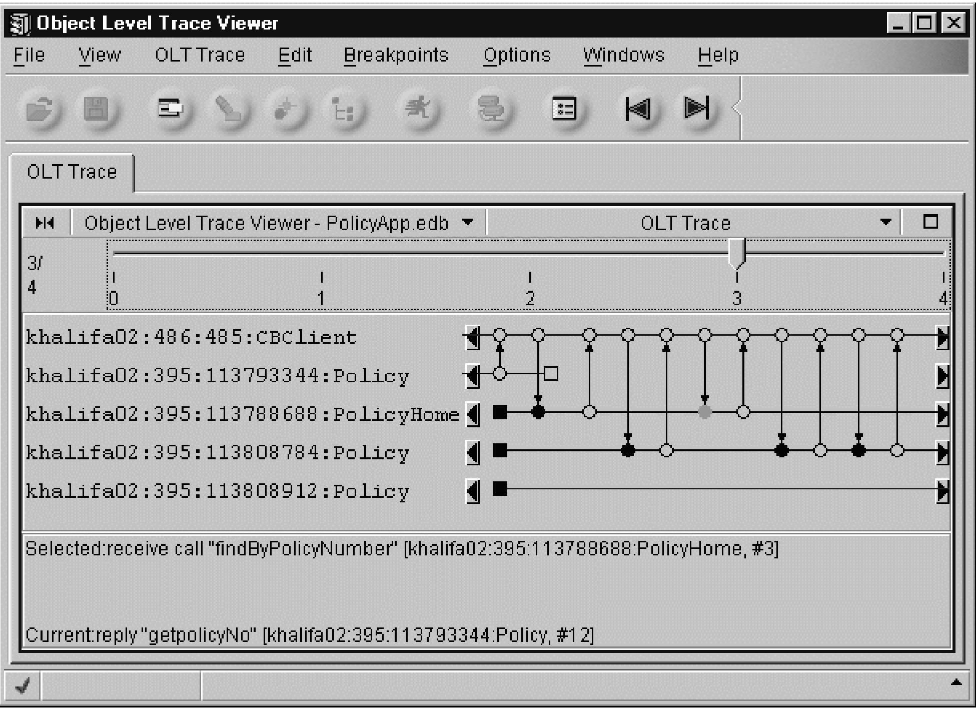
Chapter 15. Trace and Debug Distributed Applications

Object Level Trace

Object Level Trace (OLT) enables you to trace and debug multilingual, distributed applications from a single workstation. OLT works in conjunction with enhanced versions of the VisualAge for C++ and VisualAge for Java debuggers, and with a remote Debugger Daemon, to enable you to debug server and client code seamlessly from a single workstation.

Tracing Client-Server Communication
The trace facility consists of three components:

- The **OLT Server** records method calls from the client to distributed objects residing on Component Broker (CB) application servers.
- The **OLT Viewer** receives method call information from the OLT Server, and displays it in a graphical form (see the example below).
- The **OLT Client Controller** runs on the same workstation as the client application. It enables the setting of parameters that allow the OLT Server and Viewer to communicate with each other, and with the appropriate debugger.



The Trace
Each trace line displayed in the OLT Viewer represents either a CB server object, or the client application that initiated a method call to an object. Object method calls are shown as circles. If the method call is debuggable, the circle is filled-in. Start and exit events are shown as squares. An arrow connects paired events, with the arrowhead representing the direction of data flow.

Because distributed applications are often multi-threaded, it can be difficult to determine the relationship between threads and events. The trace uses “partial-order” representation to provide you with a clearer picture of this

© Copyright IBM Corp. 1997, 1998

relationship. Partial order concentrates on the causal relationships between events, rather than their chronological order. In some cases, however, you may prefer a "real-time" display. The OLT Viewer enables you to switch between partial-order and real-time display (**File - Preferences - OLT - Display**).

You can use the trace to analyze performance, isolate communication errors, and set debugger breakpoints.

Debugging Distributed Applications

A typical CB application is composed of three parts:

- Application code residing on a client
- Business objects residing on an application server
- Data objects that implement access to back-end data stores

Traditionally, this kind of distributed application was difficult to debug for two reasons:

1. You had to run a local debugger on both the client and the server. This often meant interacting with two or more machines in different locations.
2. If multiple clients were interacting with the server, it was difficult to know which client was calling the server.

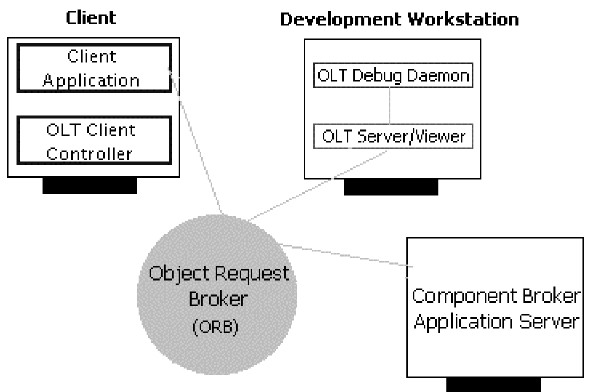
OLT solves the first problem by enabling you to debug both client and server code seamlessly from one workstation, as if both client and server code resided on a single machine. Secondly, the trace provides you with a clear picture of the communication between client and server. Each debuggable event on the trace has a pop-up menu from which you can set a debugger breakpoint.

(Note that OLT currently allows you to trace and debug client applications and business objects, but not data access code.)

You can debug your application from the client machine, the server, your development workstation, or any other machine on which the CBToolkit, or the client-server software development kit (SDK) is installed. Before running your application, start the OLT Debugger Daemon on the machine where you want to debug. Tell the OLT Client Controller where the Debugger Daemon is located, then rerun your client application. When it reaches the first breakpoint, OLT automatically launches the appropriate Java or C++ debugger on the designated workstation.

A Typical OLT Configuration

In the development stage, you typically run the OLT components and the debugger from your client machine. If necessary, however, you can deploy these tools across multiple hosts. This is especially useful when your application fails on a particular machine. Using OLT, you can run the application on the problem machine while tracing and debugging from your development workstation, as shown in the following diagram:



RELATED CONCEPTS

"Supported Platforms and Languages" on page 484
"Partial-order Display" on page 502
"Real-time Display" on page 503

RELATED TASKS

"Trace a Distributed Application" on page 489

What's New

The following changes have taken place in Object Level Trace since Release 1.3:

Performance Analysis
A new performance analysis feature has been added to the OLT Viewer. When performance analysis is enabled (from **File - Preferences - OLT Display**), calls that take more than 9 seconds are highlighted on the trace. This enables you to analyze the performance of your application and isolate bottlenecks.

Default Monitoring Mode is "Trace Only"
Previously, the default mode for running OLT was "Trace and debug with prompt". In that mode, the debugger attached immediately to your running process, and OLT opened a debugger window when it encountered the first debuggable event. Because the debugger uses extra system resources, your results will be more efficient if you produce a trace first, set breakpoints on the trace, then run the application a second time in "Trace and debug with prompt" mode. For this reason, the default monitoring mode in the Client Controller has been changed to "Trace Only".

Distributed Debugging
The CB Toolkit includes enhanced versions of the VisualAge for C++ and VisualAge for Java debuggers. You can use these debuggers in the traditional way, to test local applications, or you can run them in conjunction with OLT, to debug distributed applications. The Component Broker debugger includes the following enhancements:

- Remote debugging
- Local source capability (source code can reside on the debugger back-end or front-end workstation)
- Support for Java business objects
- Unhandled exceptions support
- AIX support

- Deep-step debugging
- Codeview 5 support
- Debug -on-demand capability

Usability Improvements in the OLT Viewer
The following improvements have been made to the “look and feel” of the OLT Viewer:

- Toolbar buttons have been added to provide easy access to common functions.
- A pop-up menu is now available from the selected event. From the pop-up menu, you can tag the selected event, or add it to the breakpoint list.
- The horizontal scrolling mechanism has been replaced by a slide bar that enables you to move quickly to any event on a particular trace, and toolbar buttons have been added for bi-directional “far scrolling”.
- The “Home” and “End” keys now take you to the first or last event on a trace.

Supported Platforms and Languages

The following matrix shows the client-server configurations currently supported by Object Level Trace. Several scenarios are included to guide you through using OLT with the most common configurations:

	Server (Business Objects):					
	NT	NT	AIX	AIX	OS/390	OS/390
	VisualAge for C++ BO	**Java BO**	**VisualAge for C++ BO**	**Java BO**	**VisualAge for C++ BO**	**Java BO**
NT Client:						
ActiveX	T	TD	X	X	X	X
VisualAge for C++	TD (Scenario 1)	TD	TD	TD	TD	TD
Java	TD (Scenarios 2 & 3)	TD	TD	TD	TD	TD
AIX Client:						
VisualAge for C++	TD	TD	TD	TD	TD	TD
Java	TD	TD	TD	TD	TD	TD
OS/390 Client:						
VisualAge for C++	TD	TD	TD (Scenario 4)	TD	TD	TD

TD: Full tracing and debugging support
T: Trace only support
X: No support

Scenarios:

1. See “Debug a C++ Client and C++ BO in Step by Step Mode - Scenario” on page 518
2. See “Trace and Debug a Java Client and C++ BO - Scenario” on page 512

3. See “Debug a Java Client from Startup - Scenario” on page 515
4. See “Trace and Debug a C++ Client and C++ BO on AIX - Scenario” on page 522

Debugger Limitations:

1. The debugger interface runs only on Windows NT. You can debug AIX or OS/390 applications by running the OLT Debugger Daemon on a Windows NT machine, and indicating the location of that machine in the OLT Client Controller. See the topic “Debug Business Objects” on page 495 for more information.
2. You cannot trace or debug Java applets. You must first port your applet into a Java application.

RELATED CONCEPTS
“Monitoring Modes”

RELATED TASKS
“Prepare for Distributed Tracing and Debugging” on page 486

Monitoring Modes

Object Level Trace (OLT) combines a graphical tracing tool with a distributed debugger. You can run the trace facility and the debugger separately or together; you can choose from five monitoring modes in the OLT Client Controller:

No trace and debug
Use this mode when you do not want to trace or debug your application.

Trace only (default)
As your application runs, the OLT Server monitors events and sends corresponding information to the Viewer. The Viewer creates a trace. Trace only mode has a lesser impact on memory and performance than any of the debugging modes. Once you have a trace to analyze, you can set breakpoints on selected events, then change to a debugging mode before rerunning your application.

Debug only
This mode enables you to debug your application without producing a trace. In this mode, the debugger steps into *every* debuggable business object method without first prompting you.

Trace and debug with prompt
This mode provides access to both the trace facility and the distributed debugger. It also gives you the greatest control over the debugging process. If **Options - Step by step debug mode** is selected in the OLT Viewer (it is selected by default), OLT stops your application at each debuggable method. At that point, you can choose to step into or over the debuggable code. If you turn off **Options - Step by step debug mode**, you can set your own breakpoints on the trace, by clicking on a server event, and selecting **Add to breakpoint list** from its pop-up menu.When you run your application again, the debugger opens on every occurrence of the server event you selected.

Trace and debug without prompt
In this mode (unlike trace and debug *with* prompt), the debugger steps into *every* debuggable business object method without first prompting you.

RELATED CONCEPTS
"Supported Platforms and Languages" on page 484

RELATED TASKS
"Prepare for Distributed Tracing and Debugging"
"Debug a Distributed Application" on page 494

Prepare for Distributed Tracing and Debugging

Object Level Trace (OLT) enables you to monitor the flow of a distributed application, and to seamlessly debug client and server code from a single workstation. You must have the client Software Development Kit (SDK) installed on the machine where you plan to trace or debug the client application.

In a typical OLT session, you first create a graphical trace of your application, then set breakpoints and rerun your application in conjunction with the debugger.

To start an OLT session, follow these steps:

1. On your development workstation, "Compile Application Code with OLT Flags".
2. Using System Manager, install your application (System Administration Guide).
3. Using System Manager, "Enable Remote Tracing and Debugging" on page 488.
4. Using OLT, "Trace a Distributed Application" on page 489.

RELATED CONCEPTS
"Supported Platforms and Languages" on page 484

RELATED TASKS
"Debug a Distributed Application" on page 494
"OLT Scenarios" on page 509

Compile Application Code with OLT Flags

In order to trace or debug with OLT, you must compile your code using the IVB_TRACE_DEBUG option. This option sets flags in the makefile, which adds tracing and debugging statements to your server DLL. The flags are not set by default because these statements significantly increase the size of your DLLs. Code compiled with OLT flags will continue to run normally outside of OLT.

(For OS/390 applications, refer to the topic "Use OLT with OS/390" on page 492.)

WIN **Compile Server Code on Windows NT:**
To recompile from a command line, complete the following steps:

1. Enter `set IVB_TRACE_DEBUG=1`
2. Enter `nmake -f yourfile.mak clean`
3. Enter `nmake -f yourfile.mak all`

To recompile your code in Object Builder, complete the following steps:

1. From the tree view, select the **Build Configuration** folder with the right mouse button.
2. From the pop-up menu, select **Add Server DLL**. The Server DLL wizard opens.

3. On the **Name and Options page**, in the **MAKE Options** field, type `IVB_TRACE_DEBUG=1`.
4. Click **Finish**.
5. Click the **Build Configuration** folder with the right mouse button.
6. From the pop-up menu, select **Generate - All - All Targets** (or specify C++ or Java targets).

AIX Compile Server Code on AIX:

To recompile from a command line, complete the following steps

1. For tracing, edit the obadll.mk file to remove the comment marks from the following lines:

   # CONST_CC_FLAGS_TRACE = -DCBS_TRACE_DEBUG
   # CONST_LD_FLAGS_TRACE = -livbtr10

   Add comment marks to the following lines:

   CONST_CC_FLAGS_TRACE =
   CONST_LD_FLAGS_TRACE =
2. For debugging, remove the comment marks from the following lines:

   #CONST_CC_FLAGS_DEBUG = -g
   #CONST_LD_FLAGS_DEBUG =
   #CONST_JAVAC_FLAGS_DEBUG = -g

   Add comment marks to the following lines:
   CONST_CC_FLAGS_DEBUG =
   CONST_LD_FLAGS_DEBUG =
   CONST_JAVAC_FLAGS_DEBUG =

   Remove the comment marks from the following lines:

   # CONST_CC_FLAGS_OPTIMIZE = -qnooptimize
   # CONST_JAVAC_FLAGS_OPTIMIZE =

   Add comment marks to the following lines:

   CONST_CC_FLAGS_OPTIMIZE = -O -Q
   CONST_JAVAC_FLAGS_OPTIMIZE = -O
3. Clean-up any previous make by deleting any *.o, *.so, and *.a files.
4. Enter `make yourfile.mak`

To recompile your code within Object Builder, complete the following steps:

1. From the tree view, select the **Build Configuration** folder with the right mouse button.
2. From the pop-up menu, select **Add Server DLL**. The Server DLL wizard opens.
3. On the **Name and Options page**, type the following options:
   - **CPP Compiler Options**: `-DCBS_TRACE_DEBUG -g`
   - **Link Options:** `-livbtr10`
4. Click **Finish**.
5. Click the **Build Configuration** folder with the right mouse button.
6. From the pop-up menu, select **Generate - All - All Targets** (or specify C++ or Java targets)

Compiling a Single Business Object

If you want to debug only one business object, you can save time by recompiling only that object's makefile. Before doing so, delete the DLL and OBJ files associated with your object (for example, *yourobject*S.dll, *yourobject*MO_.obj, and

y*ourobject*BO_I.obj on Windows NT, or the equivalent *.o, *.so, and *.a files on AIX), then recompile the object's make file (*yourobject*S.mak).

In Object Builder, select your business object file, and from its pop-up menu, select **Generate - All**.

Compile Client Applications for Debugging
OLT enables you to debug server and client code seamlessly. If you intend to debug a C++ client application, you must set IVB_TRACE_DEBUG=1 and recompile with the debug option: `-d`. Make sure that Optimization is not enabled (remove `-O`).

For Java applications, compile with the **javac -g** option.

Next, use System Manager to install your application.

RELATED TASKS
Install an Application (System Administration Guide)
"Enable Remote Tracing and Debugging"

Enable Remote Tracing and Debugging

Before running Object Level Trace, use System Manager to enable remote tracing and debugging, and change the request timeout value on both your server and client images.

To enable remote tracing and debugging, follow these steps:
1. In System Manager, select **View - User Level - Expert**.
2. Open the **Host Images** folder and expand the host image that corresponds to the name of your server.
3. Expand **Server Images** and select the server image where your application resides. If the application server is already running, select **Stop** from the server image's pop-up menu before proceeding to the next step.
4. From the pop-up menu, select **Edit**. A notebook opens.
5. Select the **Main** tab.
6. Change the **debug enabled** attribute to **yes**.
7. AIX For AIX servers only, change the **Health monitor polling interval** value to **0**.
8. Select the **ORB** tab.
9. Change the **request timeout** value to **0** (or a value that cannot be easily reached when using the debugger, such as 1800 seconds).
10. Click **OK** to close the server image notebook.
11. If your client application resides on the *same host* as the server, or on a different host with no System Manager installed, follow these steps:
    a. Expand **Client Style Images** and select the host name that corresponds to the machine where your client application resides.
    b. From the client style image's pop-up menu, select **Edit**. A notebook opens.
    c. On the **Main** tab, change the **debug enabled** attribute to **yes**.
    d. Select the **ORB** tab.
    e. Change the **request timeout** value to 0 (or a value that cannot be easily reached when using the debugger, such as 1800 seconds).

f. Click **OK** to close the client image notebook.

If your client application resides on a different host *and* you have System Manager installed on that host, complete steps a-f on the *client machine*.

You are now ready to start the Component Broker name and application servers. To do so, follow these steps:

1. From the **Host Images** folder, select the host image that corresponds to the name of your server machine.
2. From the host image's pop-up menu, select **Activate**. This starts the communication daemon and the name server. Monitor the Action Console window for completion status.
3. When activation is complete, select your **application server**.
4. From the application server's pop-up menu, select **Run Immediate**. Monitor the Action Console window for completion status.

Note:
If your client-side code includes transaction timeout values, you must set these values to 0 and recompile before continuing.

You are now ready to trace your application using OLT.

RELATED TASKS
"Trace a Distributed Application"

Trace a Distributed Application

Before you trace or debug an application using OLT, you must first complete these steps:

1. "Compile Application Code with OLT Flags" on page 486
2. Use System Manager to install your application (System Administration Guide).
3. Use System Manager to "Enable Remote Tracing and Debugging" on page 488.

To start an OLT session, and create a trace, follow these steps:

1. From the Windows NT or the AIX machine where you want to view the trace, select **Start - Programs - IBM Component Broker - Object Level Trace (OLT)** (or enter `ivbtrsrv` on a command line). The OLT Server process starts and the Viewer opens.

   Note: You may see a large number of messages in the OLT Server window. As long as the OLT Server is running, you can ignore these messages, but do not close the window until you have finished using OLT.
2. In the OLT Viewer, select **Options - Online mode** . An information message appears showing the host name and TCP/IP port number where the OLT Server is listening for events. If you plan to run your client application on a different machine, you should make note of this information so that you can enter it in the OLT Client Controller (next step). Click **OK** on the message dialog box.
3. On the machine where you intend to run the client application (that is, where the *ClientC*.dll or *ClientC*.so is installed), select **Start - Programs - IBM Component Broker - OLT Client Controller** (or enter `ivbtrc` on a command line). A settings window opens.

If OLT is running on the same machine as your client application, you can accept the default settings; if OLT is running on a different machine, you need to tell the Client Controller where the OLT Server is running:

a. Select the **OLT Server** page.

b. Enter the host name where you started the OLT Server (this information was provided to you when you selected **Options - Online mode**). In the event of a port conflict, you would also change the port number.

4. Click **Apply**, then minimize the Client Controller window.
5. Start your client application.

C++ Client Application:

Start your C++ application from a command prompt.

WIN Alternatively, if the OLT Viewer is running on your client machine, you can start the application from the Viewer by selecting **File - Start process** . The Start Process dialog box maintains a list of previously-run applications, making it easier to run your application a second time.

Java Client Application:

At a command prompt, enter this command:

```
java
-Dcom.ibm.CORBA.BootstrapHost=labadie01.torolab.ibm.com
-Dcom.ibm.CORBA.EnableApplicationOLT=true
-Dcom.ibm.CORBA.ApplicationOLTHome=c:\winnt\profiles\labadie01
PolicyApp
```

where:
labadie01.torolab.ibm.com = your server application host name
c:\winnt\profiles\labadie01 = %userprofile% directory on Windows NT; $HOME directory on AIX

As your client calls objects on the CB application server, or servers, trace lines and event symbols should appear in the OLT Viewer. Once the application has finished running, you can use the trace to set breakpoints on any debuggable server events (debuggable events are represented by filled circles).

RELATED CONCEPTS
"Reading the Trace" on page 499

RELATED TASKS
"Debug a Distributed Application" on page 494
"Navigate the Trace" on page 504
"OLT Scenarios" on page 509

RELATED REFERENCES
"OLT Troubleshooting" on page 527

Start the OLT Server and Viewer on Separate Machines

When you start Object Level Trace (OLT) from the Windows NT Start menu, the Server and Viewer are automatically started together. You would typically run these components on the same machine, together with the debugger. The OLT Server, however, requires a significant amount of memory. When tracing and debugging a large application, you may want to run the Server apart from the Viewer, on a more powerful machine.

Before you can trace or debug an application using OLT, you must first complete these steps:

1. “Compile Application Code with OLT Flags” on page 486.
2. Use System Manager to install your application (System Administration Guide).
3. Use System Manager to “Enable Remote Tracing and Debugging” on page 488.

To create a trace, with the OLT Server and Viewer running on separate machines, follow these steps:

1. From the Windows NT or the AIX workstation where you want to view the trace, select **Start - Programs - IBM Component Broker - Object Level Trace (OLT)** (or type `ivbtrsrv` on a command line). The Server process starts and the Viewer window opens.
2. In the OLT Viewer, select **File - Preferences** . A settings window opens.
3. Select **OLT**. In the **OLT Server host name** field, type the host name of the machine on which you plan to start the OLT Server. Click **OK**.
4. In the OLT Viewer, select **File - Exit**. By closing the window, you save the OLT Server location to an environment file (ivbtrenv.dat).
5. On the machine where you want to start the OLT Server, enter `ivbtrsrv -standalone`. The OLT Server process starts in a shell window (or kornshell on AIX).
6. On the machine where you want the OLT Viewer, enter `ivbtrvwt`. A Viewer window opens.
7. In the OLT Viewer, select **Options - Online mode** . When an information message appears, ensure that the host name provided is the host name of the machine on which the OLT Server is running. Click **OK**.
8. On the machine where your client application is installed, select **Start - Programs - IBM Component Broker - OLT Client Controller** (or type `ivbtrc` on a command line). A settings window opens.
9. On the **OLT Server** page, type the host name of the machine on which you started the OLT Server.
10. Click **Apply**, then minimize the Client Controller window.
11. Start your client application.

    C++ Client Application:

    Start your C++ application from a command prompt.

    WIN Alternatively, if the OLT Viewer is running on your client machine, you can start the application from the Viewer by selecting **File - Start process** .

    The Start Process dialog box maintains a list of previously-run applications, making it easier to run your application a second time.

    Java Client Application:

    At a command prompt, enter this command:

    ```
    java
    -Dcom.ibm.CORBA.BootstrapHost=labadie01.torolab.ibm.com
    -Dcom.ibm.CORBA.EnableApplicationOLT=true
    -Dcom.ibm.CORBA.ApplicationOLTHome=c:\winnt\profiles\labadie01
    PolicyApp
    ```

    where:
    labadie01.torolab.ibm.com = your server application host name
    c:\winnt\profiles\labadie01 = %userprofile% directory on Windows NT; $HOME directory on AIX

Note:
In step 3, you entered a remote location for the OLT Server. This information was saved, in step 4, to your OLT environment file (ivbtrenv.dat). In future, the OLT application looks for the OLT Server in the location you specified. If you decide to run the OLT Server and Viewer together again on this machine (or to run the OLT Server in a location other than the one specified above), you must edit the ivbtrenv.dat file to point to the correct OLT Server location. The ivbtrenv.dat file is located in your %userprofile% directory on Windows NT, and the $HOME directory on AIX.

RELATED CONCEPTS
"Reading the Trace" on page 499

RELATED TASKS
"Debug a Distributed Application" on page 494
"Navigate the Trace" on page 504
"OLT Scenarios" on page 509

RELATED REFERENCES
"OLT Environment File" on page 525
"OLT Troubleshooting" on page 527

Use OLT with OS/390

To use Object Level Trace with an OS/390 server, follow these following steps:

1. Compile server application code (page 492) with OLT flags enabled
2. Using System Manager, install your application (System Administration Guide)
3. Prepare your 390 server environment for OLT (page 492)
4. Prepare your 390 or Windows NT client environment for OLT (page 492)
5. Trace your application (page 493)

Compile 390 code with OLT flags
To compile your server code for OLT, complete the following steps:

1. Set the OLT compile option:
   ```
   export IVB_TRACE_DEBUG=1
   ```
2. Compile your business object with the compiler debugging option:
   ```
   make -f yourfile.mak yourfilebo_I.o debug=1 COMPILEOPS=NOCSE
   ```
3. Compile all other files:
   ```
   make -f yourfile.mak
   ```

Copy your source files to the x:\CBroker\bin directory on the Windows NT workstation where you plan to debug.

Prepare the OS/390 Server Environment for OLT
Add the following environment variable to the server JCL:

```
IVB_DEBUG_ENABLED=1
```

Prepare a OS/390 Client Environment for OLT
Add the following environment variables to the client JCL (ensure that your environment file contains "end of string characters" at the end of each line):

- `IVB_HOME=/.`
  (this can be any existing, writable directory)
- `IVB_DEBUG_ENABLED=1`
- `data.ctrlport=5000`
- `data.ctrlhost=9.21.39.181`
  (replace 9.21.39.181 with the actual IP address of the workstation where you plan to run the client controller)

Ensure that the OLT DLLs are accessible from your OS/390 server and client. That is, **ivbtr10i.dll** and **ivbtrmsg.dll** must be authorized.

Prepare a Windows NT Client Environment for OLT
Use System Manager to enable remote tracing and debugging and change the request timeout values on your client image. You must have completed a Component Broker client installation on this workstation.

To enable remote tracing and debugging on the client image, follow these steps:

1. In System Manager, select **View - User Level - Expert**.
2. Open the **Host Images** folder, then expand the host image that corresponds to your machine name.
3. Expand **Client Style Images** and select the host name that corresponds to the machine where your client application resides.
4. From the client style image's pop-up menu, select **Edit**. A notebook opens.
5. On the **Main** tab, change the **debug enabled** attribute to **yes**.
6. Select the **ORB** tab.
7. Change the **request timeout** value to 0 (or a value that cannot be easily reached when using the debugger, such as 1800 seconds).
8. Click **OK** to close the client image notebook.

Note:
If your client-side code includes transaction timeout values, you must set these values to 0 (or a value that cannot be easily reached when using the debugger) and recompile before continuing.

You should now start your Component Broker name and application servers.

Trace a Distributed OS/390 Application
To start an OLT session, and create a trace, follow these steps:

1. From the Windows NT machine where you want to view the trace, select **Start - Programs - IBM Component Broker - Object Level Trace (OLT**). The OLT Server process starts and the Viewer opens.
2. In the OLT Viewer, select **Options - Online mode**. An information message appears showing the host name and TCP/IP port number where the OLT Server is listening for events. Click **OK** on the message dialog box.
3. If your client application is installed on OS/390 machine, start the OLT Client Controller on the Windows NT workstation that you previously set as your **data.ctrlhost**. If your client application is installed on Windows NT, start the Client Controller on the client workstation:
   a. Select **Start - Programs - IBM Component Broker - OLT Client Controller**. A settings window opens.

b. If this is the same machine on which you started the OLT Server, accept the default settings. If the OLT Server is running on a different machine, enter the host name of that machine on the **OLT Server** page.

c. Click **Apply**, then minimize the Client Controller.

If your OS/390 client environment is properly configured for OLT, the file **ivbtr11j.properties** is created in the directory defined by the IVB_HOME variable, and is updated each time you make a change in the Client Controller.

4. Start your client application.

As your client calls objects on the CB application server, or servers, trace lines and event symbols should appear in the OLT Viewer.

RELATED CONCEPTS

“Supported Platforms and Languages” on page 484

RELATED TASKS

“OLT Scenarios” on page 509
“Set Breakpoints on the Trace”

Debug a Distributed Application

Object Level Trace (OLT) works in conjunction with an enhanced VisualAge debugger. This debugger is capable of stepping seamlessly from server code to client code, as if the server and client were a single application. Furthermore, the debugger steps over any non-debuggable “glue” code found in CB server objects.

To debug using OLT, complete the following steps:

1. “Compile Application Code with OLT Flags” on page 486
2. Using System Manager, install your application (System Administration Guide).
3. Using System Manager, “Enable Remote Tracing and Debugging” on page 488.
4. Using OLT, “Trace a Distributed Application” on page 489.
5. Using the trace, “Set Breakpoints on the Trace”.
6. “Debug Business Objects” on page 495

Note:
When debugging Java classes, make sure that the source files for your classes are accessible from the CLASSPATH environment variable. That is, if the source for my.package.MyClass resides in x:\source\my\package\MyClass.java, you must add x:\source to the CLASSPATH. Otherwise, the debugger cannot find the source and you will have to enter the location manually.

RELATED TASKS

“OLT Scenarios” on page 509
“Debug Client Applications from Startup” on page 497

Set Breakpoints on the Trace

Before setting breakpoints, ensure that **Options - Step by step debug mode** is deselected. You can set a breakpoint on any debuggable server event. Debuggable

events are represented on the trace by filled circles. You cannot set a breakpoint on a "crash" event (represented by an unfilled circled with an X through it).

To set a breakpoint, follow these steps:

1. Select the filled circle that represents the method you want to debug.
2. From the circle's pop-up menu, select **Add to breakpoint list**.
3. To view a list of your breakpoints, select Breakpoints - Create breakpoints .

Alternatively, you can manually enter breakpoint information using the **Create Breakpoints dialog box:**

1. Select **Breakpoints - Create breakpoints** .
2. Enter the names of the object and method being called, and the host name of the application server on which the object resides. To obtain this information, move the mouse pointer over the event you want to debug. That event's host, object, and method names are shown on the second status line (the "current event") at the bottom of the OLT Viewer window.

When you have finished adding breakpoints, follow the steps to "Debug Business Objects".

RELATED TASKS

"Disable or Re-enable Breakpoints" on page 499
"Debug in Step by Step Mode" on page 498

Debug Business Objects

Before debugging your application, complete the following steps to create a trace and set breakpoints:

1. "Compile Application Code with OLT Flags" on page 486.
2. Using System Manager, install your application (System Administration Guide).
3. Using System Manager, "Enable Remote Tracing and Debugging" on page 488.
4. Using OLT, "Trace a Distributed Application" on page 489.
5. Using the trace, "Set Breakpoints on the Trace" on page 494.

The source code you intend to debug must be accessible from the CB application server, or from the client machine (see the topic "Search Order" for more information).

To debug C++ or Java business objects, complete the following steps:

1. On the Windows NT workstation where you want the debugger interface to open, select **Start - Programs - IBM Component Broker - OLT Debugger Daemon**. The daemon starts in a shell window. Minimize this window.
2. OLT should already be running (see step 4, above). Deselect **Options - Step by step debug mode**.
3. The OLT Client Controller should already be open on the client machine (see step 4, above). On the **Monitoring Mode** page, click **Trace and debug with prompt**, then click **Apply**.
4. If you started the Debugger Daemon on a different machine, you need to tell the Client Controller where the Debugger Daemon is running:
   a. Select the **Remote Debugger** page.

b. Enter the host name where you started the Debugger Daemon

c. Click **Apply**.

5. Start your client application:

C++ Client Application:

Start your C++ application from a command prompt.

WIN Alternatively, if the OLT Viewer is running on your client machine, you can start the application from the Viewer by selecting **File - Start process** . The Start Process dialog box maintains a list of previously-run applications, making it easier to run your application a second time.

Java Client Application:

At a command prompt, enter this command:

```
java_g -debug
-Dcom.ibm.CORBA.BootstrapHost=labadie01.torolab.ibm.com
-Dcom.ibm.CORBA.EnableApplicationOLT=true
-Dcom.ibm.CORBA.ApplicationOLTHome=c:\winnt\profiles\labadie01
PolicyApp
```

where:
labadie01.torolab.ibm.com = your server application host name
c:\winnt\profiles\labadie01 = %userprofile% directory on Windows NT; $HOME directory on AIX

AIX On AIX, replace `java_g` with `java`

As your application runs, trace lines and symbols are added to the OLT Viewer. When OLT encounters a breakpoint, the debugger automatically attaches to the process and finds the server event on which you set the breakpoint. At the same time, the debugger interface opens wherever you started the debugger deamon.

Once you have stepped through the object method call on the server, the application runs until the next breakpoint, or the end of the program, is reached.

Alternatively, you can step the debugger out of the server function and into your client code. This opens a second debugger window, and places you in the client code, immediately after the server call. Thus, you are able to debug both server and client seamlessly, as if they were one application.

While debugging a C++ business object, do not close the debugger window. Doing so shuts down the application server (this is a Windows NT limitation only). When you finish debugging, stop your application server using System Manager, then close the OLT and debugger windows.

Note:
When debugging Java classes, make sure that the source files for your classes are accessible from the CLASSPATH environment variable. That is, if the source for my.package.MyClass resides in x:\source\my\package\MyClass.java, you must add x:\source to the CLASSPATH. Otherwise, the debugger cannot find the source and you will have to enter the location manually.

RELATED CONCEPTS
"Reading the Trace" on page 499

RELATED TASKS
"Debug Client Applications from Startup" on page 497

RELATED REFERENCES
"Search Order" on page 439
"OLT Troubleshooting" on page 527

Debug Client Applications from Startup

By issuing a slightly different command when you start your client application, you can have the debugger attach to the client process from startup. This is useful if your application fails before reaching the server.

To debug the client from startup, complete the steps under the topic "Debug Business Objects" on page 495, but substitute one of the following commands to start your client application:

C++ Application:
WIN To debug a client running on Windows NT, where the client application and debugger are to run on the same workstation, enter:

```
bdbug yourapp.exe
```

If you want the debugger to open on a different workstation, follow these steps:

1. On the workstation where you want the debugger interface to open, select **Start - Programs - IBM Component Broker - OLT Debugger Daemon**. The daemon starts in a shell window. Minimize this window.
2. On the client machine, enter:
   ```
   brmtdbg -qhost=NThostname yourapp
   ```

AIX To debug a client running on AIX, follow these steps:

1. On your AIX workstation, enter:
   ```
   irmtdbg -qsession=multi -qport=8001
   ```
2. On the Windows NT workstation where you plan to use the debugger, enter:
   ```
   bdbug -qhost=dadttp2 -qport=8001 -options program args
   ```
   You can repeatedly connect using the **bdbug** command. When you are finished debugging, stop the **brmtdbg** process on AIX.

Java Application:
At a command prompt, enter this command:

```
java com.ibm.debug.engine.Jde -qhost=labadie01 -jvmargs="-
Dcom.ibm.CORBA.BootstrapHost=labadie01.torolab.ibm.com
-Dcom.ibm.CORBA.EnableApplicationOLT=true
-Dcom.ibm.CORBA.ApplicationOLTHome=c:\winnt\profiles\labadie01"
yourapp
```

where:
labadie01 = host name of the machine where the Debugger Daemon is running
labadie01.torolab.ibm.com = your server application host name
c:\winnt\profiles\labadie01 = %userprofile% directory on Windows NT; $HOME directory on AIX

Alternatively, if your application server and client are installed on the same machine, you can enter a simpler command which executes a batch file that starts your debugger and application:

```
bjdbug PolicyApp
```

AIX To use the bjdbug batch command on AIX, you must set the IVB_DBG_HOST and IVB_DBG_PORT variables to point to an OLT Debugger Daemon host and port. The script returns an error message if either of these environment variables have not been set.

Stepping from Client to Server:
In order to step from the client application into the business object, you should set your client breakpoints in the server stub, as shown below. For a C++ client, you would set the breakpoint at line 251. For Java, set the breakpoint at line 255 (in the invoke statement):

```
Source: Policy_S.cpp - Thread :9 - Policy_C.cpp
File  View  Breakpoints  Monitors  Run  Options  Windows  Help
248
249  ::CORBA::Void Policy_ORBProxy::premium(::CORBA::Float premium)
250  {
251    CORBA::Request _req(this, "_set_premium", 0);
252   try {
253
254      _req << premium;
255      _req.invoke();
256      CORBA::RequestInterceptor::run_client_filters(&_req);
257      return;
258    }
259    catch(::CORBA::SystemException &sys_ex) {
260        ::CORBA::RequestInterceptor::run_client_exception_filters(&_req, sys_ex);
261    }
262  }
```

RELATED TASKS
"Start the Debugger and the Remote Program" on page 443
"Start the Debugger and the Remote Java Program" on page 444

RELATED REFERENCES
"Search Order" on page 439
"OLT Troubleshooting" on page 527

Debug in Step by Step Mode

Step by step debugging is the alternative to setting your own breakpoints. In this mode, OLT halts your application at the entry to *every* debuggable server event. A **Breakpoints** dialog box opens and prompts you to step into the debuggable code. If you select the breakpoint, and click **OK**, a debugger window opens and displays the first executable line in the business object method. (The debugger steps over any non-debuggable "glue" code.) You cannot go back and bring up the debugger on a previous call. To do this, you must rerun the application, or clear **Options - Step by step debug mode**. While the debugger is open, it is in complete control of the CB application server.

Note: When your application is calling objects on multiple servers, the dialog box may list multiple breakpoints. Select all those that you want to step into, then click **OK.**

Step by step debug mode is selected by default. To set your own breakpoints, you must first deselect **Options - Step by step debug mode**, or clear this option in the **Breakpoints** dialog box.

RELATED TASKS
"Set Breakpoints on the Trace" on page 494
"Disable or Re-enable Breakpoints"

Disable or Re-enable Breakpoints

To disable breakpoints previously set, follow these steps:

1. Select **Breakpoints - Create breakpoints** . A dialog box opens.
2. From the **Available breakpoints** list, select all the breakpoints that you want to disable.
3. Click **Disable**.

The Set Debugger Breakpoints dialog box maintains a history of all the breakpoints you previously set, including those you have disabled. It maintains this list even after you shut down and restart OLT. To re-enable a previously disabled breakpoint, select it from the list and click **Enable**. To delete a breakpoint, select it from the list and click **Delete**.

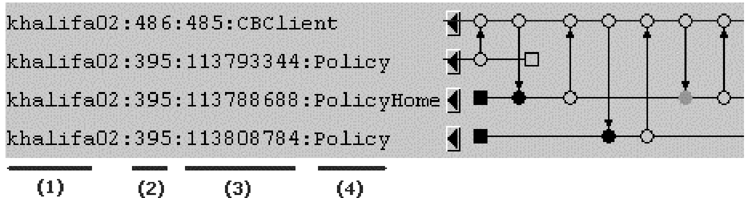
Hint:
If Breakpoints - Create breakpoints is unavailable, make sure that you have deselected **Options - Step by step mode**, make sure that you have selected Options - Online mode .

RELATED TASKS
"Debug in Step by Step Mode" on page 498

Reading the Trace

The trace consists of trace lines and event symbols. The colors referred to below are the OLT Viewer's default colors. You can change these by selecting **File - Preferences - OLT - Display - Colors**.

Trace Lines
A trace is a horizontal line connecting a sequence of events (object method calls) that run under a single thread of execution. Each trace line represents either a Component Broker object residing on the application server, or the client application that initiated a method call to an object. The name of each trace is shown to the left of the line:



Each trace line name consists of four parts, representing the following information:

1 host name
2 process ID
3 **on a client trace:** thread ID
on an object trace: object ID

4	**on a client trace:** CBClient **on an object trace:** object name

To help you navigate on the trace, every fourth line is colored blue.

Events

An "event" is a call to a business object method, a return from a method call, or the start or end of a process. Object method calls are shown on the trace as circles. Start and exit events are shown as squares. An arrow connects paired events, with the arrowhead representing the direction of data flow.

Identifying Predecessor and Successor Events

Predecessor and successor events are those events that follow or precede a particular event in its own thread, and related events in other threads. To identify all predecessors and successors to a particular event, position the mouse pointer over an event and hold down **Ctrl**, while right-clicking on the event. All predecessor events are colored red, and all successor events are colored green.

Status Lines

The status lines at the bottom of the window identify the location of a "selected" event and a "current" event. The selected event (highlighted green by default) is the last event you clicked with the left mouse button. The current event is the event that your mouse pointer is currently positioned over. As you move your pointer, the current event changes.

1 2 3 4 5 6 7

Selected: call "findByPrimaryKeyString" [mfok01:418:464: CBClient, #6]
Current: reply "syncFromDataObject" [mfok01:475:110520928:Policy, #2]

This textual representation of each event consists of seven parts, representing the following information:

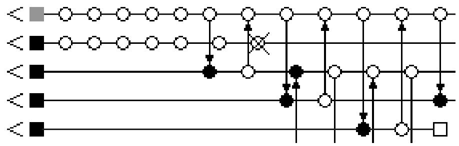
1	event type
2	method name
3	host name
4	process ID
5	**on a client trace:** thread ID **on an object trace:** object ID
6	**on a client trace:** CBClient **on an object trace:** object name
7	position on the trace

RELATED CONCEPTS

"Trace Symbols"
"Selected Event" on page 501
"Partial-order Display" on page 502
"Real-time Display" on page 503
"Performance Analysis" on page 504

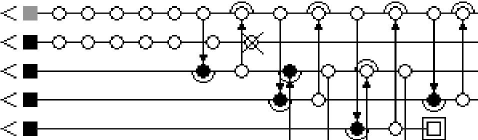
Trace Symbols

The table below explains what the trace symbol and their corresponding status line text represent.



Symbol	Status Line Text	Type of Event
■	start	object created or retrieved
	call	method call from client
	receive call	entry point to debuggable method of an object
	reply	method call completed from an object
	receive reply	method call completed from client
	receive call	call from an object to itself (nested)
	reply	call from an object to itself (nested)
	one-way call	method call from client, no reply expected
	receive one-way call	entry point to an object method
	untraceable call	call from client, recipient is untraceable
□	exit	stop, or object destroyed or released
	receive call	application exception
	call or reply	event waiting for partner to arrive

You also have the option of adding "decorations" to the trace (**File - Preferences - OLT - Display**). This option adds an outline to all events that complete a call, as well as exit events, as shown below:

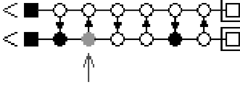


RELATED CONCEPTS
"Selected Event"
"Partial-order Display" on page 502
"Real-time Display" on page 503

RELATED TASKS
"Prepare for Distributed Tracing and Debugging" on page 486
"Navigate the Trace" on page 504

Selected Event

The trace always highlights one event. This is the "selected" event (green is the default highlighting color).



Information about the selected event is shown on the first status line:

1 2 3 4 5 6 7

Selected: call "findByPrimaryKeyString" [mfok01:418:464: CBClient, #6]
Current: reply "syncFromDataObject" [mfok01:475:110520928:Policy, #2]

This textual representation of each event consists of seven parts, representing the following information:

1 event type
2 method name
3 host name
4 process ID
5 **on a client trace:** thread ID
on an object trace: object ID
6 **on a client trace:** CBClient
on an object trace: object name
7 position on the trace

You can change which event is selected by clicking another event with the left mouse button.

Most OLT actions apply to the selected event. In that sense, the selected event acts as a kind of cursor on the trace. For example, pressing the arrow keys on your keyboard moves the selected status to the next event in the direction of the arrow. To access the pop-up menu for a selected event, click the right mouse button. From the pop-up menu, you can tag the selected event, or add it to the breakpoint list.

A selected event is always visible. If you scroll a selected event past the edge of the trace, that event loses its selected status and the OLT Viewer automatically selects another event to take its place.

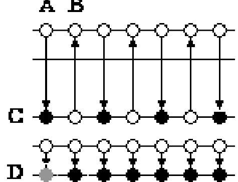
RELATED TASKS
"Navigate the Trace" on page 504
"Scroll the Trace" on page 505

Partial-order Display

Partial order is the default display mode. In this mode, the OLT Viewer attempts to represent as many events as possible on screen, while at the same time respecting causal relationships among events.

In partial-order mode, as opposed to real-time mode, events are not always drawn in the sequence in which they occurred. Partial order recognizes that just because an event occurred first does not mean that it *had* to occur first.

In the following graphic, the highlighted event D actually occurred 6 seconds *after* event A, but because A and its successors have no precedence relationship to D, partial ordering allows them to be drawn on the same vertical. In real-time mode, you would have to scroll many screens to the right in order to see event D.



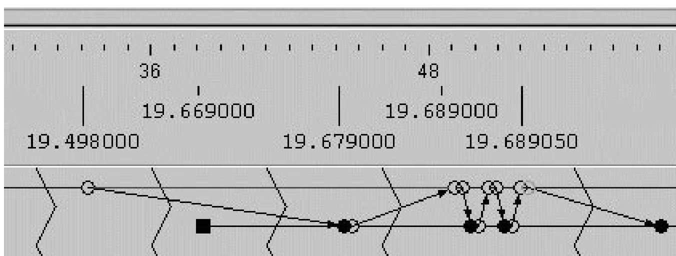
By default, “time” advances horizontally from left to right (though you can change to a vertical orientation under **File - Preferences - OLT - Display**). If event C causally precedes event B, then C is *always* placed to the left of B, never to the right and never on the same vertical. Every time you scroll the trace, the Viewer redraws events according to this partial-ordering principle. For that reason, scrolling can often appear uneven because the Viewer does whatever reordering is necessary to keep as many events as possible on screen.

All Component Broker communication is synchronous. The OLT Viewer aligns communication pairs vertically and connects them with an arrow, indicating the direction of data flow. While waiting for a partner event to arrive, the event that initiated the call is drawn on the trace with an “x” through it, to indicate that its position in the partial order is still uncertain.

RELATED CONCEPTS
“Real-time Display”
“Performance Analysis” on page 504
“Reading the Trace” on page 499

Real-time Display

You can change the OLT Viewer mode to real-time display by selecting **File - Preferences - OLT - Display - Display real time**. This display shows events as they actually occurred in real time (as opposed to partial ordering, which respects causal relationships but does not necessarily show events in chronological order):



The default time scale (measured in microseconds) is designed to fit a reasonable number of events on a single screen. The jagged lines between events represent lapses in time, during which the call might be stepping through non-debuggable code or be caught in network traffic.

When viewing events in real time, be aware of the following issues:

Overlapping events
In real time, events can occur almost simultaneously. This means that event symbols and connection arrows frequently overlap. You can adjust the time scale on the trace by selecting **Options - Change scale**, but even when you set the scale to its smallest interval (1 tic of the clock, or microsecond, per pixel), events can still overlap.

Cumulative time
The clock starts at 0 when you first run your client application. If you run the application again, the clock continues from the time reached on the first run, rather than restarting at 0.

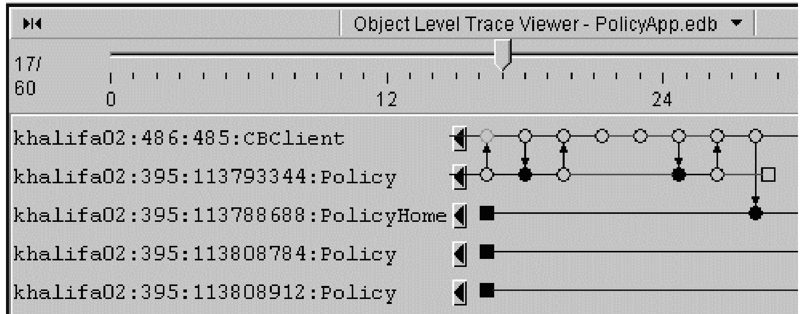
Synchronization
OLT performs clock synchronization between machines in order to avoid anomalies (for example, a received call being drawn before a call). Thus, times shown on the real-time scale may not match system clocks exactly. Also, if some events on the trace have been collected with real-time information, and some have not, OLT will “fake” the time for those that do not have real-time information.

RELATED CONCEPTS
“Partial-order Display” on page 502
“Performance Analysis”

Performance Analysis

When performance analysis is enabled (under **File - Preferences - OLT - Display**), the trace line between any calls that take more than 9 seconds is turned to red (this color can be changed). This includes, calls made within the client, within the server or client-to-server calls.

You can change the time interval size under Performance Analysis settings (**File - Preferences - OLT - Display).**



This highlighting enables you to analyze the performance of your application, isolate bottlenecks, and determine which functions are slower than others.

If you encounter performance problems, you may want to switch to a real-time display, by selecting **File - Preferences - OLT - Display**. Under Display Style, click **Real time**.

RELATED CONCEPTS
“Real-time Display” on page 503
“Partial-order Display” on page 502

Navigate the Trace

When working with a trace, think of the selected event (highlighted green by default) as your cursor. Most of your actions depend on the position of this event. The selected event is the event that you last clicked with the left mouse button (unless scrolling has moved that event off the trace, in which case the program selects another event).

The following keys can help you navigate the trace:

- **P** moves the selected status along the communication arrow to the partner event. If the selected event has no partner (as with a start or an exit call), "P" has no effect. This key is particularly useful when working in real-time display, where partner events are not vertically aligned.
- **Arrow Keys** move selected status to the closest event in the direction of the arrow.
- **Page Up** moves the selected event to the far right of the screen
- **Page Down** moves the selected event to the far left of the screen.
- **Home** takes you to the first event of the line on which the selected event is currently positioned.
- **End** takes you to the last event of the line on which the selected event is currently positioned. This is helpful if you want to verify that all your threads completed successfully.

You can locate specific events in one of two ways:

Scrolling
Scrolling horizontally across the trace is much more complex than simply moving the cursor along the trace lines. There are three mechanisms for scrolling: slide bar, trace line arrows, and scroll buttons. As you scroll, the trace is continually refreshed to reflect precedence relationships to the selected event. This is done so that as many events as possible can be shown on the screen at any one time.

Tagging
Tagging adds a bookmark-type identifier to an event so that you can return directly to it at any time.

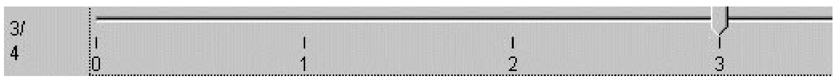
RELATED CONCEPTS
"Reading the Trace" on page 499
"Selected Event" on page 501
"Real-time Display" on page 503

RELATED TASKS
"Scroll the Trace"
"Tag an Event" on page 507

Scroll the Trace

In most cases, the OLT Viewer cannot display all events simultaneously. OLT provides three scrolling mechanisms for navigating along the trace:
- slide bar
- trace line arrows
- scroll buttons

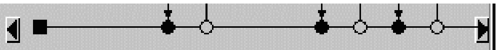
Slide Bar



The slide bar moves the selected event along its trace line. The total number of events on the trace is shown on the slide ruler. To bring a particular event into view,

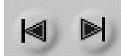
slide the pointer along the ruler. The numbers to the left of the ruler (3/4 in the example above) show the number of the selected event, over the total number of events on the line.

Trace line arrows



The trace lines themselves have horizontal scroll arrows on each end. Click on these arrows to move to the right or left along a single line. As you go, the trace is refreshed according to the partial-order display principle.

Scroll buttons



The scroll buttons move screen-by-screen through the trace. Clicking on a scroll button places the selected event on the right or left edge of the trace, and refreshes the other events according to partial order.

Undo Scrolling
To undo the most recent scroll operation, select **Edit - Undo scrolling**. This returns you to the previously selected event, but may not produce exactly the same trace that existed before scrolling.

RELATED CONCEPTS
"Selected Event" on page 501
"Partial-order Display" on page 502

Reorder Trace Lines

By selecting **Edit - Reorder traces by...**, you can reorder trace lines in the following ways:

Precedence
This option reorders trace lines according to precedence relationships with the selected event (which is highlighted green by default). Traces containing events with a precedence relationship are moved nearer to the selected event, while those with no relationship are pushed farther away. Only events currently visible are considered.

(You can also show precedence relationships by pressing **Ctrl**, while clicking an event with the right mouse button).

Local Optimization
This option reorders traces to minimize the length of the arrows that connect binary event pairs (such as between a call and a received call.) Local optimization is useful when the trace appears cluttered with arrows. This option only considers currently visible events. If you frequently scroll to other parts of the trace, you should instead reorder by global optimization, which considers all event connections.

Global Optimization
This option reorders every trace line to minimize the length of arrows connecting binary event pairs (such as between a call and a received call). Global optimization is useful if the trace appears cluttered with arrows and you find yourself frequently scrolling to different sections of the trace.

Original Order
This option returns traces to their original order (the order that was in place before you selected an **Edit - Reorder traces by...** option).

Move Trace
You can also reorder trace lines by moving an individual line to any point in the trace. For example, you may want to position a pair of client and object traces side by side. To move a trace, follow these steps:

1. Select **Edit - Move trace**.
2. Click anywhere on the trace line you want to move. The line is highlighted green.
3. Drag and drop the line to its new position. The relative positions of all other lines are unaffected.

RELATED CONCEPTS
"Reading the Trace" on page 499
"Selected Event" on page 501

RELATED TASKS
"Navigate the Trace" on page 504

Tag an Event

In a large trace, you can attach a descriptive, bookmark-type tag to any event that you may want to find again quickly (for example, an event immediately preceding a crash event).

To tag an event, follow these steps:

1. Select the event by with the left mouse button. The event is highlighted green (by default).
2. Click the right mouse button. From the pop-up menu, select **Tag event** . A dialog box opens.
3. Enter a descriptive name to identify the event, then click **OK**. A circle (red, by default) is added to the tagged event.

To find a tagged event, follow these steps:

1. Select **Edit - Locate event tag** . A dialog box opens and lists all tagged events.
2. Select the tag name of the event you want to find, then click **OK**. The Viewer scrolls to the tagged event.

To delete a tag, follow these steps:

1. Select **Edit - Delete event tag** . A dialog box opens and lists all tagged events.
2. Select the tag name, or names, that you want to delete, then click **OK**.

Save the Current Trace to a File

You may want to save your trace to a file in order to analyze it at a later date, or on another machine (with the same operating system). The Save As action does not simply take a snapshot of the current screen. All events recorded in the current session (both before and after the Save As action) are saved to the file you specify. Once you specify a filename, you cannot change that name until the next session. Similarly, once you have completed the original save, as described above, there is no need to save again.

To save the current trace to a file, follow these steps:

1. Ensure that online mode is selected (**Options - Online mode**).
2. Select **File - Save OLT File as** . A dialog box opens.
3. Type a name for the new file, then click **OK**. The file is saved when you close OLT.

By default, event trace files are saved to the directory defined by IVB_DRIVER_PATH%\temp (for example, x:\CBroker\temp) on NT, or /tmp on AIX. To change the directory to which files are saved, follow these steps:

1. Select **File - Preferences** . A settings window opens.
2. From the tree view, select **OLT**.
3. Enter a new path in the **OLT Read/Save file path** field.
4. Click **OK**. This change takes effect the next time you open the OLT Server.

Note: The file is saved to the machine where the OLT Server is running.

RELATED TASKS

"Open an Existing Trace File"

Open an Existing Trace File

To open a previously saved trace file, start OLT and follow these steps:

1. Select **File - Open OLT File** . A dialog box opens.
2. Select an event file name from the list, then click **OK**. The selected OLT file opens in the Viewer.

OLT files can only be saved to, and opened from, one specific directory on your workstation. The default directory for storing trace files is the directory defined by the %IVB_DRIVER_PATH%\temp environment variable (for example, x:\CBroker\temp) on Windows NT, or /tmp on AIX. To change this directory, follow these steps:

1. Select **File - Preferences** . A settings window opens.
2. From the tree view, select **OLT**.
3. Enter a new path in the **OLT Read/Save file path** field.
4. Click **OK**. This change takes effect the next time you start OLT.

Note:
Once you have opened a trace file in the OLT Viewer, you cannot toggle back to online mode in order to trace a running application. You must first close the OLT Viewer (and Server) and restart.

RELATED TASKS
"Save the Current Trace to a File" on page 508

OLT Scenarios

The following scenarios use the Policy sample to illustrate some common OLT tasks:

- Windows NT Client
  - "Trace and Debug a Java Client and C++ BO - Scenario" on page 512
  - "Debug a Java Client from Startup - Scenario" on page 515
  - "Debug a C++ Client and C++ BO in Step by Step Mode - Scenario" on page 518
- AIX Client
  - "Trace and Debug a C++ Client and C++ BO on AIX - Scenario" on page 522

To use these scenarios, ensure that you have the Component Broker run time, C++ client SDK, Server SDK, CB Toolkit, and tookit samples installed on your system, then complete the following steps:

1. Set OLT compile options (explained below).
2. Compile the Policy application according to the instructions in the sample documentation (x:\CBroker\samples\InstallVerification\ProgrammingModel\Docs\Policy.html).
3. Install the Policy application using System Manager (explained below).

Set OLT Compile Options
Enable the trace and debug flags before compiling:

WIN This procedure for Windows NT assumes that Component Broker is installed at x:\CBroker.

1. From a command prompt, change to your working directory:
   ```
   x:\CBroker\samples\InstallVerification\ProgrammingModel\
   BusinessObjects\Policy\Working\NT
   ```
2. Enter:
   ```
   set IVB_TRACE_DEBUG=1
   ```

AIX To compile the sample on AIX, follow these steps:

1. Edit the obdll.mk file as follows:

   Remove the comment marks from the following lines:

   # CONST_CC_FLAGS_TRACE = -DCBS_TRACE_DEBUG
   # CONST_LD_FLAGS_TRACE = -livbtr10

   Add comment marks to the following lines:

   CONST_CC_FLAGS_TRACE =
   CONST_LD_FLAGS_TRACE =

   Remove the comment marks from the following lines:

   #CONST_CC_FLAGS_DEBUG = -g
   #CONST_LD_FLAGS_DEBUG =
   #CONST_JAVAC_FLAGS_DEBUG = -g

Add comment marks to the following lines:
CONST_CC_FLAGS_DEBUG =
CONST_LD_FLAGS_DEBUG =
CONST_JAVAC_FLAGS_DEBUG =

Remove the comment marks from the following lines:

CONST_CC_FLAGS_OPTIMIZE = -qnooptimize
CONST_JAVAC_FLAGS_OPTIMIZE =

Add comment marks to the following lines:

CONST_CC_FLAGS_OPTIMIZE = -O -Q
CONST_JAVAC_FLAGS_OPTIMIZE = -O

Compile the Sample Application
Follow the instructions in this file:
(x:\CBroker\samples\InstallVerification\ProgrammingModel\Docs\Policy.html)

Install the Sample Application
After successfully compiling the Policy sample, install the application by following these steps:

1. Define the sample application to System Management.
2. Configure an application server.
3. Load the sample application on a server.

Define the Sample Application to System Management
Load the application definition on your Host Image as follows:

1. In System Manager, select **View - User Level - Expert**.
2. Expand **Host Images**.
3. Select a host image. There is a default image that corresponds to the name of the system on which the Server with System Manager configuration was installed.
4. Open the pop-up menu, and select **Load Application**.
5. Browse and select **PolicyFamily.ddl**.
6. Click **OK**. The Action Console window indicates when the application has been successfully loaded.
7. Close the Action Console window.

Configure an Application Server
Create a Server Group and Server in the Sample Configuration, then load the Policy application onto the server.

WIN On Windows NT, you can configure the application server using System Management wizards:

1. Select **Wizards - Create Servers**.
2. Click **Next** to accept the defaults on the first two pages.
3. On the **Server Group** page, type a server group name. For this exercise, use *myservergroup*.
4. Click **Next**.
5. On the **Server** page, type a server name. For this exercise, use *myserver.*
6. Click **Finish**.
7. Select **Wizards - Configure Server**.

8. On the **Select Applications to Configure** page, select **PolicyApp**, then click **Add**.
9. Click **Next** to accept the defaults on the next two pages.
10. On the **Select Server to Configure Applications On** page, select **myservergroup**, then click **Add.**
11. **Click Finish**.

AIX On AIX, follow these steps to configure the application server:

1. In System Manager, expand **Management Zones - Sample Cell and Work Group Zone - Configurations**.
2. Select **Sample Configuration**. From its pop-up menu, select **New - Server Group**. A dialog box is displayed.
3. Type a server group name. For this exercise, use *myservergroup*.
4. Click **OK**.
5. Select **myservergroup**. From its pop-up menu, select **New - Server (member of group)**.
6. Type a server name. For this exercise, use *myserver*.
7. Click **OK**.
8. Select **myserver**. From its pop-up menu, select **Drag**.
9. Collapse Management Zones.
10. Expand **Hosts**.
11. Select y*ourhost*. From its pop-up menu, select **Configure Server (member of group)**.
12. Expand **Available Applications**.
13. Select **Policy**. From its pop-up menu, select **Drag**.
14. Expand **Management Zones - Sample Cell and Work Group Zone - Configurations**.
15. Select **Sample Configuration**. From its pop-up menu, select **Add Application**. This places the application in the configuration.
16. Select **Policy**. From its pop-up menu, select **Drag**.
17. Expand **Management Zones - Sample Cell and Work Group Zone - Configurations - Sample Configuration - Server Groups**.
18. Select **myservergroup**. From its pop-up menu, select **Configure Application**.

Enable Remote Tracing and Debugging

Enable remote tracing and debugging on both your client and server images, follow these steps:

1. Expand **Management Zones - Sample Cell and Workgroup Zone - Configuration** - **Sample Configuration** - **Server Groups** folder and expand the host image that corresponds to the name of your server.
2. Expand **Groups** and select **myservergroup.** From its pop-up menu, select **Edit**. A notebook opens.
3. Select the **Main** tab.
4. Change the **debug enabled** attribute to **yes**.
5. AIX For AIX servers only, change the **Health monitor polling interval** value to **0**.
6. Select the **ORB** tab.
7. Change the **request timeout** value to **0**.

8. Click **OK** to close the server image notebook.
9. Expand **Client Style** and select the host name that corresponds to the machine where your client application resides.
10. From the client image's pop-up menu, select **Edit**. A notebook opens.
11. On the **Main** tab, change the **debug enabled** attribute to **yes**.
12. Select the **ORB** tab.
13. Change the **request timeout** value to 0.
14. Click **OK** to close the client image notebook.

Activate Your Configuration
To activate your configuration:

1. Expand **Management Zones - Sample Cell and Workgroup Zone - Configuration**, and select **Sample Configuration**.
2. Click Sample Configuration with the right mouse button, and select **Activate** from the pop-up menu.
3. Monitor the Action Console window for a completion status.

You are now ready to trace and debug the Policy sample using one of the OLT scenarios.

Trace and Debug a Java Client and C++ BO - Scenario

Objective
To trace and debug a distributed application in which the client code is written in Java and the business object is written in C++. Both are installed on Windows NT.

Before You Begin
You must complete the steps to compile and install the Policy sample for Windows NT, including the Java client application (as explained in the "OLT Scenarios" on page 509).

For this exercise, you should run OLT and the distributed debugger on the same Windows NT machine as your client application. The CB application server can either reside on this same machine, or on a remote host.

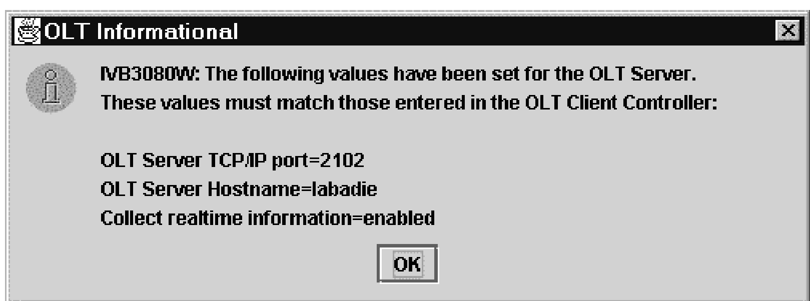
Description
In this exercise, you will complete these steps:

1. Start Object Level Trace
2. Start the OLT Client Controller
3. Run your application to produce a trace
4. Set a breakpoint on the business object
5. Start the OLT Debugger Daemon
6. Rerun your application
7. Debug the server method
8. Step from server to client code

Start Object Level Trace:

1. From the Windows NT Start menu, select **Programs - IBM Component Broker - Object Level Trace (OLT)**. The OLT Server process starts and the Viewer window opens.

2. In the OLT Viewer, select **Options - Online mode** . An information message is displayed.



3. Click **OK**.

Start the OLT Client Controller:

1. From the Windows NT Start menu, select **Programs -IBM Component Broker - OLT Client Controller**. A settings window opens:
2. Minimize **OLT Client Controller** window.

Run your application:

1. From a command prompt, change to the directory where you compiled the Java client version of the Policy sample
2. Enter the following command:

```
java
-Dcom.ibm.CORBA.BootstrapHost=labadie01.torolab.ibm.com
-Dcom.ibm.CORBA.EnableApplicationOLT=true
-Dcom.ibm.CORBA.ApplicationOLTHome=c:\winnt\profiles\labadie01
PolicyApp
```

where:
labadie01.torolab.ibm.com = your server application host name
c:\winnt\profiles\labadie01 = %userprofile% directory

A trace is created, showing the calls from your client to the Policy and PolicyHome objects on the server.

Set a breakpoint on the trace:

1. Select the event that represents the “setpremium” method (the fourth event on the Policy trace).
2. From this event’s pop-up menu, select **Add to breakpoint list**.

Change to a debugging mode:

1. In the OLT Client Controller, select **Monitoring Mode**.
2. Click **Trace and debug with prompt**, then click **Apply**.
3. From the Windows NT Start menu, select **Programs - IBM Component Broker - OLT Debugger Daemon**. The daemon starts in a shell window. Minimize this window.
4. In the OLT Viewer, deselect **Options - Step by step debug mode**.

Rerun your application:

1. From a command prompt, change to the directory where you compiled the Java client version of the Policy sample. Ensure that somojor.zip the first file in your classpath.
2. Enter the following command:

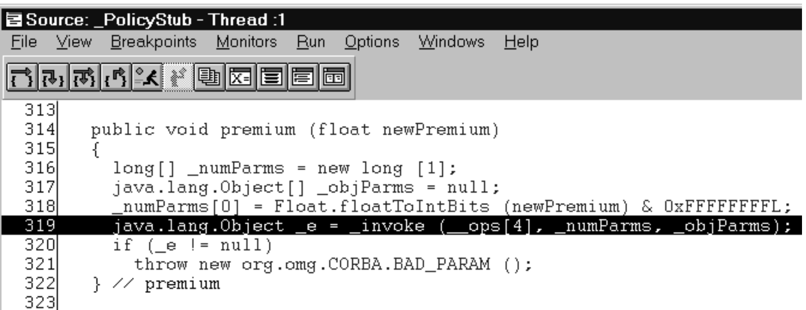
```
java_g -debug
-Dcom.ibm.CORBA.BootstrapHost=labadie01.torolab.ibm.com
-Dcom.ibm.CORBA.EnableApplicationOLT=true
-Dcom.ibm.CORBA.ApplicationOLTHome=c:\winnt\profiles\labadie01
PolicyApp
```

where:
labadie01.torolab.ibm.com = your server application host name
c:\winnt\profiles\labadie01 = %userprofile% directory on Windows NT

The program halts at your breakpoint. The debugger opens and steps into the "setpremium" method on the server:

```
Source: PolicyBO_I.obj - Thread :9 - POLICYBO_I.CPP
File View Breakpoints Monitors Run Options Windows Help
57 ::CORBA::Void PolicyBO_Impl::premium( ::CORBA::Float  premium)
58
59 {
60   //Version identifier DCE:9BF7CAB0-8962-11d2-B5AE-400011532396:1
61   // Insert Method modifications here
62   iDataObject->premium(premium);
63
64   // End Method modifications here
65 }
66
```

3. Click **Step over** twice. This opens a second debugger window for the client, and places you in the client code, immediately past the call to the "setpremium" method:



4. On the client debugger toolbar, click **Run** .

5. When you see the following dialog box, click **OK**:

Your trace should now be complete.

Important Note:
While running your application, do not close the debugger window you are using to debug server code. Doing so shuts down the application server (this is a Windows NT limitation). When you finish debugging, stop your application server using System Manager, then close the OLT and debugger windows.

Debug a Java Client from Startup - Scenario

Objective
To debug Java client code from startup, then step into the C++ server code.

Before You Begin
You must complete the steps to compile and install the Policy sample for Windows NT, including the Java client application (as explained in the “OLT Scenarios” on page 509).

For this exercise, you should run OLT and the distributed debugger on the same Windows NT machine as your client application. The CB application server can either reside on this same machine, or on a remote host.

Description
In this exercise, you will complete these steps:

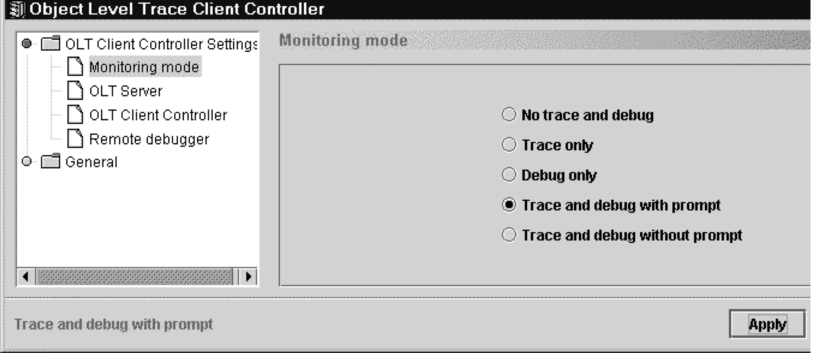
1. Start the OLT Debugger Daemon.
2. Start Object Level Trace in “trace and debug with prompt” mode.
3. Start the Java client debugger and client application.
4. Step through a client call to the server.
5. Debug the server method.

Start the OLT Debugger Daemon:

1. From the Windows NT Start menu, select **Programs - IBM Component Broker - OLT Debugger Daemon**. The daemon starts in a shell window. Minimize this window.

Start Object Level Trace:

1. From the Windows NT Start menu, select **Programs - IBM Component Broker - Object Level Trace (OLT)**. The OLT Server process starts and the Viewer window opens.
2. In the OLT Viewer, select **Options - Online mode** . An information message is displayed. Click **OK**.
3. Deselect **Options - Step by step debug mode**.
4. From the Windows NT Start menu, select **Programs - IBM Component Broker - OLT Client Controller**. A settings window opens.
5. From the tree view, select **Monitoring Mode**.
6. Select **Trace and debug with prompt**, then click **Apply**:



7. Minimize the Client Controller window.

Start the Java client debugger and client application:

1. From a command prompt, change to the directory where you compiled the Java version of the Policy sample. Ensure the somojor.zip is the first file in your classpath.
2. Enter the following command, which starts both the policy application and the Java debugger:

   ```
   java com.ibm.debug.engine.Jde -qhost=labadie01 -jvmargs="-
   Dcom.ibm.CORBA.BootstrapHost=labadie01.torolab.ibm.com
   -Dcom.ibm.CORBA.EnableApplicationOLT=true
   -Dcom.ibm.CORBA.ApplicationOLTHome=c:\winnt\profiles\labadie01"
   PolicyApp
   ```

   where:
   labadie01 = host name of the machine where the OLT Debugger Daemon is running
   labadie01.torolab.ibm.com = your server application host name
   c:\winnt\profiles\labadie01 = %userprofile% directory

   Alternatively, if the application server and client are installed on the same machine, you can enter a simpler command which executes a batch file that starts your debugger and application:

   ```
   bjdbug PolicyApp
   ```

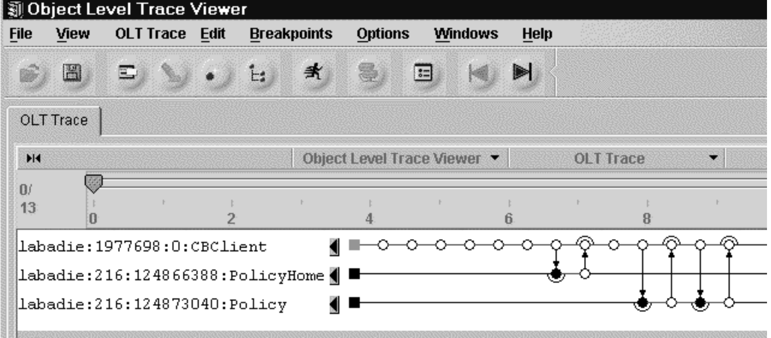
The Java debugger opens to the first executable line in the Policy client application.

Set a breakpoint on the client:

1. Scroll to line 451, and set a breakpoint by double-clicking on that line:

   ```
   418  //==================================================================
   419  // Attempt to set amount and premium values on policy object.
   420  //==================================================================
   421  void set_policy_values()
   422  {
   423    try
   424    {
   425      System.out.println("\t (set the premium value)  policy.premium((flc
   426      policy.premium((float) 200.00);
   427      System.out.println("\t (set the amount value)   policy.amount((floa
   428      policy.amount((float) 20000.00);
   429      System.out.println("\t policy values: premium = " + policy.premium(
   430    }
   ```

2. On the debugger toolbar, click **Run** . The OLT Viewer should begin creating a trace:

   The application stops at the breakpoint you set in your client code.
3. On the debugger toolbar, click **Step into** . This places you in the “amount” method on the client:

```
Source: _PolicyStub - Thread :1
File  View  Breakpoints  Monitors  Run  Options  Windows  Help
283
284   public void amount (float newAmount)
285   {
286     long[] _numParms = new long [1];
287     java.lang.Object[] _objParms = null;
288     _numParms[0] = Float.floatToIntBits (newAmount) & 0xFFFFFFFFL;
289     java.lang.Object _e = _invoke (__ops[1], _numParms, _objParms);
290     if (_e != null)
291       throw new org.omg.CORBA.BAD_PARAM ();
292   } // amount
293
```

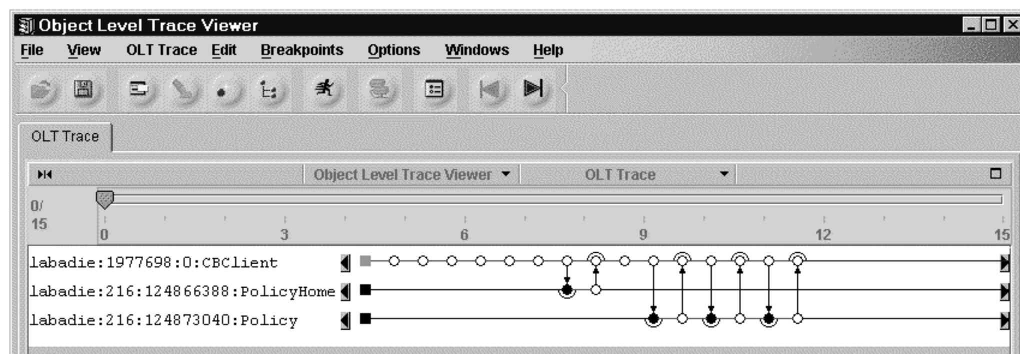
4. Set a breakpoint at line 289, then click **Run** .
5. On the debugger toolbar, click **Step debug** . The debugger follows the client request to the server. A new debugger window opens and places you in the server code, at the “amount” method of the selected object:

```
Source: PolicyBO_I.obj - Thread :9 - POLICYBO_I.CPP
File  View  Breakpoints  Monitors  Run  Options  Windows  Help
34  {
35    //Version identifier DCE:9BF7C8F4-8962-11d2-B5AE-400011532396:1
36    // Insert Method modifications here
37    iDataObject->amount(amount);
38
39    // End Method modifications here
40  }
41  ::CORBA::Long PolicyBO_Impl::policyNo()
```

6. On the debugger toolbar, click **Step over** to step over line 35. Click **Step over** a second time to go back to the next executable line on the client:

```
Source: _PolicyStub - Thread :1
File  View  Breakpoints  Monitors  Run  Options  Windows  Help
283
284   public void amount (float newAmount)
285   {
286     long[] _numParms = new long [1];
287     java.lang.Object[] _objParms = null;
288     _numParms[0] = Float.floatToIntBits (newAmount) & 0xFFFFFFFFL;
289     java.lang.Object _e = _invoke (__ops[1], _numParms, _objParms);
290     if (_e != null)
291       throw new org.omg.CORBA.BAD_PARAM ();
292   } // amount
293
```

At this point, the OLT trace looks like this:



7. On the client debugger toolbar, click **Run.** When you see the following dialog box, click **OK**:

The trace should now be complete.

Important Note:
While running your application, do not close the debugger window that you are using to debug server code. Doing so shuts down your application server. When you finish debugging, stop your application server using System Manager, then close the OLT and debugger windows.

Debug a C++ Client and C++ BO in Step by Step Mode - Scenario

Objective
To trace and debug a distributed application in which both server and client code are written in C++ and installed on Windows NT.

Before You Begin
You must complete the steps to compile and install the C++ Policy Sample for Windows NT (as explained in the “OLT Scenarios” on page 509).

For this exercise, you should run OLT and the distributed debugger on the same Windows NT machine as your client application. The CB application server can reside either on this same machine, or on a remote host.

Description
In this exercise, you will complete these steps:

1. Start the OLT Debugger Daemon.
2. Start Object Level Trace.
3. Change the monitoring mode to “Trace and debug with prompt”.
4. Debug a method on the server.
5. Step from the server method into your client code.

Start the OLT Debugger Daemon:

1. From the Windows NT Start menu, select **Programs - IBM Component Broker - OLT Debugger Daemon**. The daemon starts in a shell window.
2. Minimize the window.

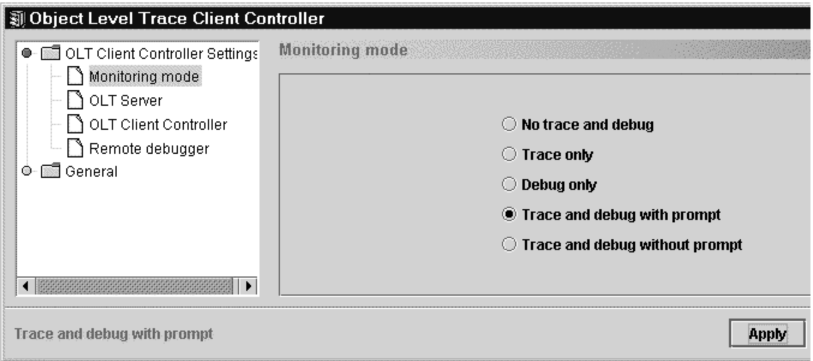
Start Object Level Trace:

1. From the Windows NT Start menu, select **Programs - IBM Component Broker - Object Level Trace (OLT)**. The OLT Server process starts and the Viewer window opens.
2. In the OLT Viewer, select **Options - Online mode** . An information message is displayed:

3. Click **OK**.
4. Ensure that **Options - Step by step debug mode** is selected:

In **Step by step debug mode**, OLT stops each time a debuggable method is encountered on the CB application server. OLT then asks whether you want to step into, or over, your server code.

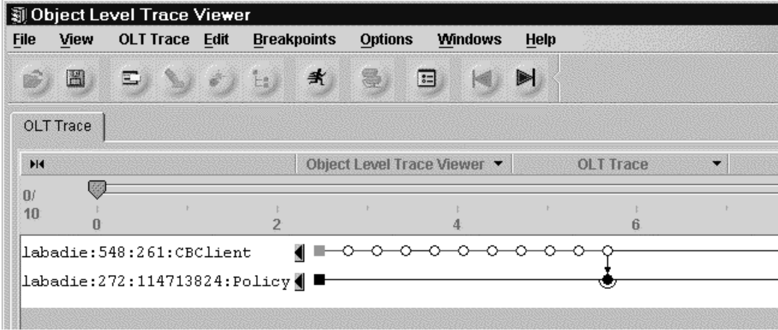
5. From the Windows NT Start menu, select **Programs - IBM Component Broker - OLT Client Controller**. A settings window opens:
6. Select **Monitoring Mode** from the tree view:
7. Select **Trace and debug with prompt**:



Run your client application:

1. In the OLT Viewer, select **File - Start process** and browse for PolicyApp.exe.

   The complete path is:
   `x:\CBroker\samples\InstallVerification\ProgrammingModel\`
   `BusinessObjects\Policy\Working\NT\PolicyApp.exe`
2. Click **OK**. The client application starts in a command shell. The OLT Viewer should soon display trace lines and event symbols:



Debug the “getpolicyNo” method:

1. The first debuggable event that OLT encounters is a call to the “getpolicyNo” method. At this point, the application halts and a dialog box opens:

2. Click **OK**. The debugger opens on the server code and steps into “getpolicyNo”.

Source: PolicyBO_I.obj - Thread :10 - POLICYBO_I.CPP

File View Breakpoints Monitors Run Options Windows Help

```
42 {
43   //Version identifier DCE:9BF7CAAF-8962-11d2-B5AE-400011532396:1
44   // Insert Method modifications here
45   return iDataObject->policyNo();
46
47   // End Method modifications here
48 }
49 ::CORBA::Float PolicyBO_Impl::premium()
```

Step into the client code:

1. On the debugger toolbar, click **Step Over** . This starts a second instance of the debugger and places you in the client application source code:

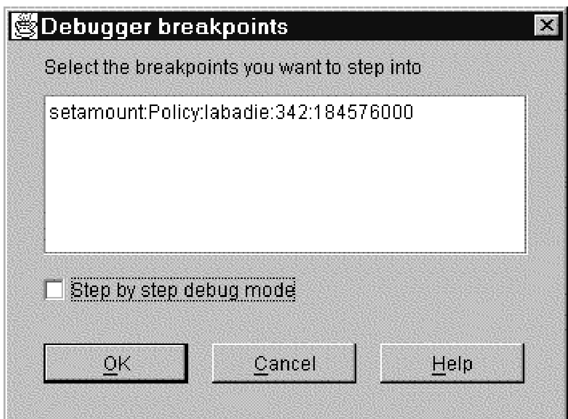
Source: Policy_S.obj - Thread :1 - POLICY_C.CPP

File View Breakpoints Monitors Run Options Windows Help

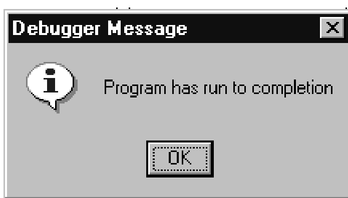
```
214   CORBA::Request _req(this, "_get_policyNo", 0);
215  try {
216
217    _req.invoke();
218    ::CORBA::Long _result;
219    _req >> _result;
220     CORBA::RequestInterceptor::run_client_filters(&_req);
221     return _result;
222   }
```

2. On the debugger toolbar, click **Run** . The application runs until it encounters the next debuggable server event (“setamount”).

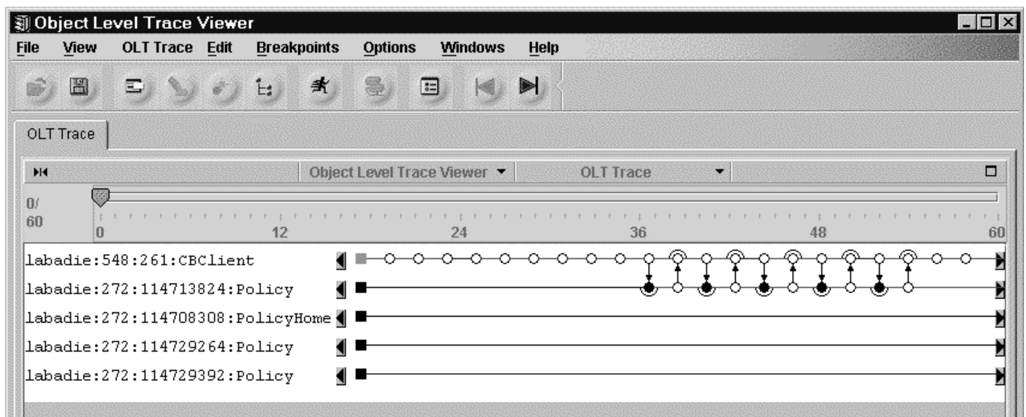
3. In the Debugger Breakpoints dialog box, clear the **Step by step debug mode** checkbox:



4. Click **OK**. The application runs through to completion, without stopping at debuggable events.
5. When you see the following dialog box, click **OK**:



The OLT Viewer should now contain a complete trace:



At this point, you can set breakpoints on any of the debuggable server events (shown as filled circles on the Policy trace). If you rerun PolicyApp.exe, the debugger opens only on your breakpoints.

Important Note:
While running your application, do not close the debugger window you are using to debug server code. Doing so shuts down the application server (this is a Windows NT limitation). When you finish debugging, stop your application server using System Manager, then close the OLT and debugger windows.

Trace and Debug a C++ Client and C++ BO on AIX - Scenario

Objective
To trace and debug a distributed application in which both server and client code are written in C++ and installed on AIX.

Before You Begin
You must complete the steps to compile and install the C++ Policy Sample for AIX (as explained in the OLT Scenarios overview).

For this exercise, you should run Object Level Trace on the same AIX machine as your client application. The CB application server can reside either on this same machine, or on a remote host. You also need a Windows NT workstation on which to interact with the debugger. This workstation must have the CB Toolkit installed, and have access to the source code.

Description
In this exercise, you will complete these steps:

1. Reset usage limits on your AIX machine.
2. Start Object Level Trace.
3. Run the client application and create a trace.
4. Set a breakpoint on the trace.
5. Start the OLT Debugger Daemon.
6. Rerun the application.
7. Debug the server method.
8. Step from server to client code.

Reset Usage Limits on the AIX machine:

1. To avoid memory errors when debugging, add the following lines in the login script file (.profile):

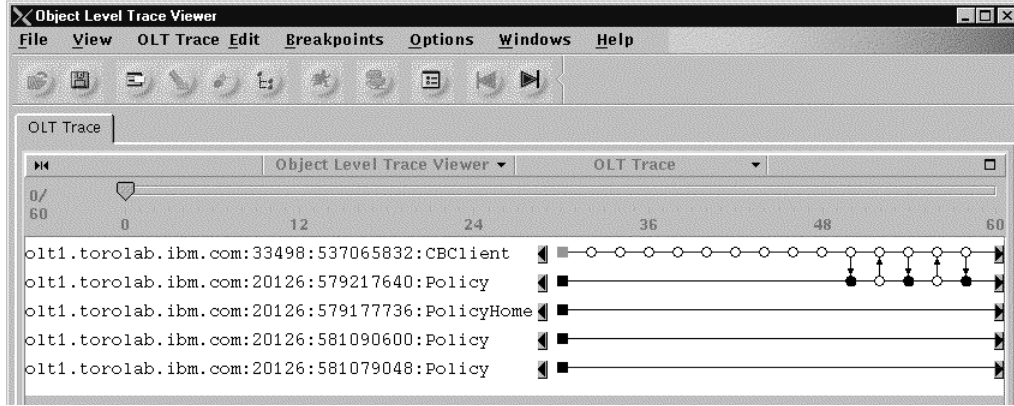
   ```
   ulimit -d unlimited    # to reset limits on data size
   ulimit -m unlimited    # to reset limits on physical memory
   ulimit -s unlimited    # to reset limits on stack size
   ```

   In addition, you should keep your virtual memory paging space as large as possible. To check current paging space, enter `lsps -a` on a command line, and increase if possible.

Start Object Level Trace:

1. From a kornshell on your AIX client, enter `ivbtrsrv`. The Server process starts and the Viewer window opens.
2. In the Viewer window, select **Options - Online mode** .
3. From a kornshell, enter `ivbtrc`. The OLT Client Controller opens.
4. Click **Apply**, then minimize the Client Controller window.
5. From a kornshell, type `PolicyApp`.

   Once PolicyApp has run to completion, the OLT Viewer should contain a trace, similar to the following:



Create a breakpoint:

1. On the trace, select the server event that represents the “setpremium” method (the fourth event on the Policy trace).
2. From the event’s pop-up menu, select **Add to breakpoint list**.

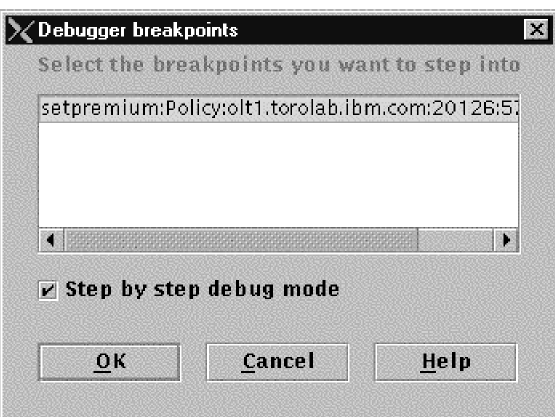
Prepare for debugging:

1. In the OLT Viewer, deselect **Options - Step by step debug mode**.
2. On a Windows NT workstation, start the OLT Debugger Daemon (**Programs - IBM Component Broker - OLT Debugger Daemon**). The daemon starts in a shell window. Minimize this window.
3. In the Client Controller on your AIX workstation, select **Remote Debugger**.
4. Type the host name of the Windows NT machine on which you started the Debugger Daemon.
5. Select **Monitoring mode**. On the Monitoring Mode page, select **Trace and debug with prompt**:
6. Click **Apply**, then minimize the Client Controller.

Run your application:

1. From a kornshell, enter `PolicyApp` to rerun the application.

   The program halts at your specified breakpoint, and the debugger prompts you to step into the “setpremium” method:

2. Select **setpremium**, then click **OK.** The debugger opens on the “setpremium” method:

```
Source: PolicyBO_I.cpp - Thread :12 - PolicyBO_I.cpp
File View Breakpoints Monitors Run Options Windows Help
53
54   // End Method modifications here
55 }
56 ::CORBA::Void PolicyBO_Impl::premium( ::CORBA::Float  premium)
57
58 {
59   //Version identifier DCE:99FE008F-DEC6-11d1-B431-08005ACE0236:2
60   // Insert Method modifications here
61   iDataObject->premium(premium);
62
63   // End Method modifications here
64 }
```

Step from server to client code:

1. Click Step return . A new debugger is opened for the client. Notice that you stop immediately after the _req.invoke() call that invokes the method on the server:

```
Source: Policy_S.cpp - Thread :9 - Policy_C.cpp
File View Breakpoints Monitors Run Options Windows Help
248
249  ::CORBA::Void Policy_ORBProxy::premium(::CORBA::Float premium)
250  {
251    CORBA::Request _req(this, "_set_premium", 0);
252   try {
253
254      _req << premium;
255      _req.invoke();
256      CORBA::RequestInterceptor::run_client_filters(&_req);
257      return;
258    }
259    catch(::CORBA::SystemException &sys_ex) {
260        ::CORBA::RequestInterceptor::run_client_exception_filters(&_req, sys_ex);
261    }
262  }
```

2. Click Step return .again to return immediately past the setpremium method called by the client:

```
Source: PolicyApp.cpp - Thread :9 - PolicyApp.cpp
File View Breakpoints Monitors Run Options Windows Help
284       else {
285          cout << "Amount set correctly..." << endl;
286          }
287
288       //****************************************************
289       ::CORBA::Float prem = 555.00;
290       floatResult = 0.0;
291       policyPtr->premium(prem);
292       floatResult = policyPtr->premium();
293       if (prem != floatResult) {
294          cout << "Policy premium not set" << endl;
295          return 1;
296          }
297       else {
298          cout << "Premium set correctly..." << endl << endl;
299          }
```

After two step-returns, the debugger is now in the client application. The client application calls the getter method for the premium attribute on the Policy object (line 292).

3. To step back into the Policy object, click Step into at line 292 three times. When you step into`policyPtr->premium()`, the debugger stops at the server stub:

```
Source: PolicyApp.cpp - Thread :9 - PolicyApp.cpp
File  View  Breakpoints  Monitors  Run  Options  Windows  Help
284        else {
285           cout << "Amount set correctly..." << endl;
286           }
287
288        //*****************************************************
289        ::CORBA::Float prem = 555.00;
290        floatResult = 0.0;
291        policyPtr->premium(prem);
292        floatResult = policyPtr->premium();
293        if (prem != floatResult) {
294           cout << "Policy premium not set" << endl;
295           return 1;
296           }
297        else {
298           cout << "Premium set correctly..." << endl << endl;
299           }
```

4. Step over to line 233.
5. Step-debug at line 233. This places you in the premium getter method of the Policy Business Object:

```
Source: PolicyApp.cpp - Thread :9 - PolicyApp.cpp
File  View  Breakpoints  Monitors  Run  Options  Windows  Help
284        else {
285           cout << "Amount set correctly..." << endl;
286           }
287
288        //*****************************************************
289        ::CORBA::Float prem = 555.00;
290        floatResult = 0.0;
291        policyPtr->premium(prem);
292        floatResult = policyPtr->premium();
293        if (prem != floatResult) {
294           cout << "Policy premium not set" << endl;
295           return 1;
296           }
297        else {
298           cout << "Premium set correctly..." << endl << endl;
299           }
```

6. Click **Run** . When the client application completes, you should see the following dialog box:

7. Click **OK**. The debugger process for the client application ends. The debugger for the Policy business object stays active until you stop the application server using System Manager.

Important Note:
While running your application, do not close the debugger window that you are using to debug server code. Doing so shuts down your application server (this is a Windows NT limitation). When you finish debugging, stop your application server using System Manager, then close the OLT and debugger windows.

OLT Reference

OLT Environment File

The OLT environment file (ivbtrenv.dat) is stored on the machine where the OLT Server is running. On Windows NT, the file can be found in your %userprofile% directory. On AIX, the file is in your $HOME directory.

The ivbtrenv.dat file contains the following variables, which are explained below:

```
IVB_TRC_SRV_HOST=
IVB_TRC_SRVAPP_PORT=2102
IVB_TRC_SRVCLT_PORT=2202
IVB_TRC_EV_DIR=%IVB_DRIVER_PATH%/temp
IVB_TRC_OLT_VER=OLT20
IVB_TRC_SRV_UID1=6666
IVB_TRC_SRV_UID2=9999
IVB_TRC_SRV_TGTDIR=%IVB_DRIVER_PATH%\\bin;
```

WIN **IVB_TRC_SRV_HOST=**

AIX **IVB_TRC_SRV_HOST=$HOSTNAME**

When you select **Options - Online mode** in the OLT Viewer, an information message will display the host name of the OLT Server. You must ensure that this host name matches the host name entered on the **OLT Server page** in the **OLT Client Controller**.

If you run the OLT Server *separately* from the OLT Viewer, the IVB_TRACE_HOST value in the ivbtrenv.dat file on the Viewer machine must point to the location of the OLT Server.

IVB_TRC_SRVAPP_PORT=2102
IVB_TRC_SRVCLT_PORT=2202

The OLT Server attempts to listen for events on TCP/IP port 2102. If it cannot connect to both this port and to port 2202, the OLT Server will not come up. If you change these numbers (for example, if the ports are busy and you cannot start the OLT Server), you must edit the the ivbtrenv.dat file on any other machine that is running an OLT component. This does not apply if you are running the debugger, Server, and Viewer on the same machine.

WIN **IVB_TRC_EV_DIR=%IVB_DRIVER_PATH%/temp**

AIX **IVB_TRC_EV_DIR=/TMP**

This value determines which directory the trace files are stored to and retrieved from. By default, event trace files are stored in the directory defined by %IVB_DRIVER_PATH%/temp (or /tmp on AIX). If you are unsure about the directory, enter `set IVB_DRIVER_PATH` on a command line (`echo $IVB_DRIVER_PATH` on AIX). To save and retrieve event trace files from a different directory, enter a new path in the Viewer, from **File - Preferences - OLT**. To open any previously-saved files, you must first move them to your new default directory.

IVB_TRC_OLT_VER=OLT20

This value represents the OLT version number (for example, Release 2.0).

IVB_TRC_SRV_UID1=6666
IVB_TRC_SRV_UID2=9999

These values are PIN numbers that validate the OLT Viewer and Server pair.

WIN **IVB_TRC_SRV_TGTDIR=%IVB_DRIVER_PATH%\\bin**

AIX **IVB_TRC_SRV_TGTDIR=$IVB_DRIVER_PATH/bin**

This value specifies the location of the file ivbtrdsc. This file tells the OLT Viewer how to display events.. By default, this location is the bin directory, under your Component Broker installation.

Note:
If you change a value in the ivbtrenv.dat file, it does not take effect until you restart the OLT Server and Viewer.

OLT Command-line Arguments

Object Level Trace supports the following command-line arguments (arguments can be combined):

Start the OLT Server and Viewer	`ivbtrsrv`
Start the OLT Server without the Viewer	`ivbtrsrv -standalone`
Start the OLT Server with a specific configuration file	`ivbtrsrv -dat` *path_and_filename*
Start the OLT Server and Viewer, and open a saved trace file	`ivbtrsrv` *filename*
Start the OLT Viewer	`ivbtrvwt`
Start the OLT Client Controller	`ivbtrc`

OLT Troubleshooting

Troubleshooting information is included for the following OLT problems.

- **OLT Startup**
  - OLT Server or Viewer fails to start (page 528)
  - OLT Server abends (page 528)
  - OLT Client Controller fails to start (page 528)
  - Client application fails to run (page 528)
  - Client application runs, but OLT appears to stop the application server (page 528)
  - Events do not appear in the Viewer (page 528)
  - Unusual program behavior (page 529)
- **Java Clients**
  - Out-of-memory errors when starting a Java client application (page 529)
  - Visual Age for Java clients cannot be traced with OLT (page 529)
- **Distributed Debugging**
  - Debugger Daemon fails to start (page 530)
  - Debugger interface fails to open (page 530)
  - Debugger ignores second application
  - Cannot set breakpoints in the OLT Viewer (page 530)
  - Debugger fails when debugging AIX client (page 531)
  - Debugger is extremely slow when using Loopback Adaptor (page 531)
- **Real-time Display**
  - Real-time information is not collected (page 531)
- **OS/390**
  - Confirm that your OS/390 client is properly connected to the Client Controller (page 531)

RELATED REFERENCES
"Limitations when Debugging Visual C++ Programs" on page 476
"Limitations When Debugging Interpreted Java" on page 477
"Limitations When Debugging Interpreted Java" on page 477

OLT Troubleshooting - Startup

OLT Server or Viewer fails to start
If OLT appears to start, then closes prematurely, try starting OLT from the command line by typing `ivbtrsrv`. This should provide you with an error message. Ensure that the directory defined by the %IVB_DRIVER_PATH%\temp exists, and that you have "write" permission for this directory. If the problem persists, enter the following command to pipe your error message to a file, then send this file to your IBM representative:

```
ivbtrsrv >olt_err.log 2>&1
```

OLT Server abends
The OLT Server can handle a maximum of 300 processes. If you reach this limit, and continue running the application (for example, if you are running a continuous loop), the OLT Server aborts. If this happens, close and reopen the OLT Viewer. Do not forget to select Options - Online mode before rerunning your application.

Before closing the Viewer, you can save your previous trace to a file (**File - Save OLT file as**).

OLT Client Controller fails to start
In the directory defined by %userprofile% on Windows NT, or $HOME on AIX, delete the ivbtr11j.properties file, then try starting the OLT Client Controller again. If the problem persists, use the following command to start the Client Controller from a command line and pipe any startup errors to a file. Send this file to your IBM representative:

```
ivbtrc >olt_err1.log 2>&1
```

Client application fails to run
Try running the application from a command shell. If the application fails to start, a detailed error message should appear. For help interpreting the message, see the *Problem Determination Guide*.

Client application runs, but OLT appears to stop the application server
Check the transaction timeout values in your application code. These values must be set to zero, or a value not easily reached while using the debugger, such as 1800 seconds.

Events do not appear in the Viewer
If your application runs cleanly, but events do not appear in the OLT Viewer, ensure that you have completed the necessary startup steps:

1. Using the `set IVB_TRACE_DEBUG=1` option, compile your code to include OLT flags.
2. Using System Manager:
   a. Install your application.
   b. On your server and client-style images, enable remote tracing and debugging and set request timeout values to zero (or a value not easily reached while using the debugger, such as 1800 seconds).
   c. Activate the server host image.
   d. Start the application server.
3. On your client machine:

a. Start Object Level Trace, then put the Viewer in online mode by selecting **Options - Online mode** (do not close the OLT Server window).
b. Start the OLT Client Controller (click **Apply**, then minimize the window but do not close it).
c. Run the client application.

Remember to start the OLT Client Controller on every client that is running an application.

Also, you should verify that the Server settings in the Client Controller (hostname and port number) match those displayed when you selected **Options - Online** mode in the OLT Viewer.

Unusual program behavior
Check your temporary directory (as set by the %IVB_DRIVER_PATH%/TEMP environment variable on Windows NT, or the /tmp directory on AIX). If you have a .chk file in your temporary directory, delete it and try running your application again. During normal operation, Object Level Trace creates files in the temporary directory and, if the program terminates unexpectedly, some of these files may not be properly deleted.

OLT Troubleshooting - Java Clients

Out-of-memory errors when starting a Java client application
If you encounter memory errors when starting a Java client application, modify the start command to include the following string before the application name: -mx255m -ms30m -oss75m. For example:

```
java com.ibm.debug.engine.Jde -qhost=labadie01 -jvmargs="-
Dcom.ibm.CORBA.BootstrapHost=labadie01.torolab.ibm.com
-Dcom.ibm.CORBA.EnableApplicationOLT=true
-Dcom.ibm.CORBA.ApplicationOLTHome=c:\winnt\profiles\labadie01"
-mx255m -ms30m -oss75m myapp
```

Visual Age for Java clients cannot be traced with OLT
On your Visual Age for Java client code, set the OLT properties inside the code:

```
java.util.Properties props = new java.util.Properties();
props.put("com.ibm.CORBA.EnableApplicationOLT","true");
props.put("com.ibm.CORBA.ApplicationOLTHome","c:/winnt/profiles/labadie");
props.put("com.ibm.CORBA.BootstrapHost","labadie.torolab.ibm.com");
// If we got the host and port from some other source besides the command
line
// arguments passed by the PolicyApp invocation, we could set the values
using
// properties like this below and then use the CBSeriesGlobal.Initialize
// method that takes as parms, host and port.
// props.put("com.ibm.CORBA.BootstrapHost", host);
// props.put("com.ibm.CORBA.BootstrapPort", port);

System.out.println("1) About to call ORB.init passing Bootstrap information
passed in on command line");
orb = ORB.init (args, props);
```

You must now use use the VA Java debugger on your client. Debugging your business object brings up the Component Broker debugger. You should not step

from the BO back to the client using Step over or Step debug because the VA Java debugger is already attached to your client.

OLT Troubleshooting - Distributed Debugging

Debugger Daemon fails to start
When you installed CBConnector, port 8001 was designated for the remote debugger. If you change this port, you must change the **IVB_DBG_PORT** environment variable, and the **bdbug** entry in the services file on all clients and servers affected by the change, and on those machines running the application you want to debug. Any change you make to the debugger port number must also be reflected on the **Remote Debugger** page in the **OLT Client Controller**.

WIN On Windows NT, the *services* file is found in Winnt\system32\drivers\etc\.
AIX On AIX, the *services* file is located in /etc/.

If you do not have a port conflict and the Debugger Daemon still fails to start, check the services file and ensure that **bdbug 8001/tcp** is on a separate line.

Debugger interface fails to open
Check your Windows NT Task Manager for the OLT Debugger Daemon (bdbugd.exe). If the daemon is not running, follow these steps:

1. Close the OLT Server, Viewer, and Client Controller windows.
2. Select **Start - Programs - IBM Component Broker - OLT Debugger Daemon.** A blank command shell window opens. Minimize this window, but do not close it.
3. Start **Object Level Trace** and put the Viewer in online mode (select **Options - Online mode**).
4. Open the **OLT Client Controller** window and select **Monitoring mode** from the tree view.
5. On the **Monitoring mode** page, ensure that one of the debug modes is selected.
6. Click **Apply**. Minimize the Client Controller window, but do not close it.
7. Run your client application.

If the debugger still does not appear, you might have a port conflict. The following three values must match:

- **TCP/IP port** entry on the **Remote Debugger** page in the OLT Client Controller.
- **bdbug** entry in the Winnt\system32\drivers\etc\services file on Windows NT, or the /etc/services file on AIX.
- **IVB_DBG_PORT** environment variable.

If you have multiple clients interacting with the same server, you must start the OLT Client Controller on *each* client, and specify **the same host name** on the Remote Debugger page (this should be the host name of the machine on which you started the OLT Debugger Daemon).

Debugger ignores second application
When you have a multi-user host talking to a single Component Broker application server, the first client application to gain control of the debugger retains control throughout its run. The debugger ignores the other application. To debug object method calls from the second client, you must bring down the application server and start the second application on its own.

Cannot set breakpoints in the OLT Viewer

“Step by step” debug mode is enabled by default in the OLT Viewer. In this mode, OLT stops at every instance of debuggable server code. You cannot set your own breakpoints until you deselect **Options - Step by step debug mode**.

Once step by step mode is turned off, you can right-click a debuggable event (represented by a filled circle), and select **Add to breakpoint list** from the pop-up menu. If you click a non-debuggable event, this option is disabled.

Debugger fails when debugging AIX client

When debugging an AIX client directly, memory limitations may cause the debugger to fail. You can avoid this problem by adding the following lines in the login script file (.profile):

```
ulimit -d unlimited    # to reset limits on data size
ulimit -m unlimited    # to reset limits on physical memory
ulimit -s unlimited    # to reset limits on stack size
```

this ensures that usage limits are cleared in each window you open. In addition, you should keep your virtual memory paging space as large as possible. To check current paging space, enter `lsps -a` on a command line.

Debugger is extremely slow when using Loopback Adaptor

If you are using the Microsoft Loopback Adaptor on a laptop, and the debugger takes 10 minutes or more to open on every request, remove any DNS entries from your TCP/IP settings.

OLT Troubleshooting - Real-time Display

Real-time information not collected

If you are running multiple clients, and you select **collect real time information** for one client but not another, OLT obeys whichever client was first to send a request to the OLT Server. In other words, if the first client to send a request did *not* have **collect real time information** selected, real-time information is *not* collected for *any* of the clients.

Therefore, when running multiple client applications, set the same real-time option (in the OLT Client Controller) for every client.

OLT Troubleshooting - OS/390

Confirm that your OS/390 client is connected to the Client Controller

If your OS/390 client environment is properly configured for OLT, the file**ivbtr11j.properties** is created in the directory defined by the IVB_HOME variable, and is updated each time you make a change in the OLT Client Controller.

Chapter 16. IR Browser

Start the IR Browser

The interface repository (IR) browser is part of the CBToolkit development environment. The IR browser enables you to examine and modify the contents of the Component Broker interface repository. Use the IR Browser to:

- Navigate through the various repository views.
- Locate type definitions.
- Understand calling relationships among interfaces and operations.
- Delete objects.

To start the IR Browser:

- at the command prompt, enter `irbrowser`, or
- on a Windows NT desktop, select **Start - Programs - IBM Component Broker - Interface (IR) Repository Browser.**

To exit the IR Browser, select **Repository - Quit** from the Repository menu or double-click the IR Browser icon in the title bar.

RELATED CONCEPTS
Interface Repository (*Advanced Programming Guide*)

Configure Online Help

If the online help does not appear when you select **Help - Topics**, complete the following configuration steps in the IR Browser:

1. Select **Options - Help Setup**
2. Enter the path to your web browser executable (for example, `x:\netscape\netscape.exe`)
3. Enter the path to the documentation:

   ```
   http://localhost:49213/cgi-
   bin/cbwebx.exe/en_US/cbdoc/Extract/0/irb/hgirb.htm
   ```

Try invoking the online help by selecting **Help - Topics**.

View Objects in the Repository

View the Definition of an Object

To view the definition (contents) of an object, double-click on the object in the **Containment**view. A textual representation (such as the IDL definition or IR dump output) appears in the **IDL** view.

View Relationships Between Objects

To view the ancestors or children of an interface, double-click the object in the **Containment** or **Inheritance** view. A graphical representation appears in the **Inheritance** view, showing the following relationships to the highlighted object:

© Copyright IBM Corp. 1997, 1998

- direct base (parent) interfaces
- direct derived (child) interfaces

The flow of the **Inheritance** view is from left to right, that is, the base or parent interfaces are to the left and the derived or child interfaces are to the right of the selected interface.

To view the siblings of an interface, double-click the direct base interface in the **Containment** view. The tree expands to show a hierarchical representation. A container such as a module can be expanded to show sibling interfaces for the interface.

RELATED TASKS
"Find An Object"

View the Operations of an Interface

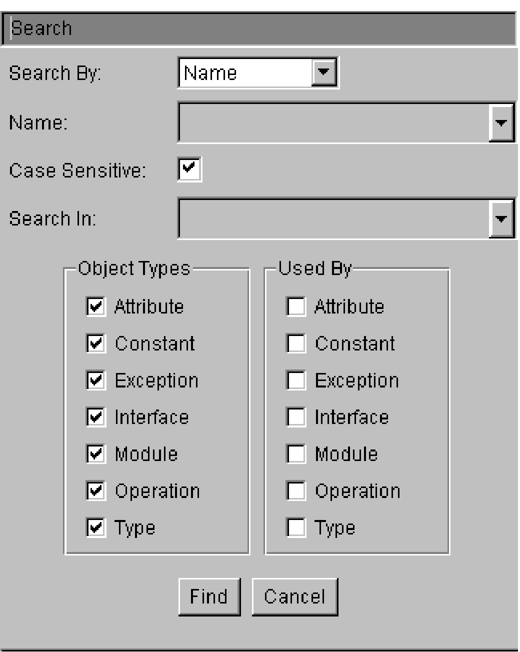
To view the operations of an interface, double-click the object in the **Containment** or **Inheritance** view. The hierarchical representation for the object (showing the container relationships) appears in the left pane of the window.

Search the Repository

Find An Object

To find an object in the interface repository, follow these steps:

1. Select **Search - Find**. The **Find** window opens.



The **Search By:** drop-down menu allows you to search for an object by name, or by its unique repository ID. Both methods accept wildcard characters as input.

2. Specify your Search Criteria.

3. Press the **Find** button. All objects that match the search criteria are listed in the **Result** scrollbox.
4. Double-click on an object to have it become the focus for the **Containment**, **IDL**, and **Inheritance** views.
5. Close the **Find** window.

Note: Due to the size and complexity of the interface repository, some searches might take several minutes. You can click the **Cancel** button to stop the search and narrow your search criteria.

RELATED TASKS
"Find an Interface's Referencing Operations"
"Search Using Wildcards"
"Search by Object Type"

Search Using Wildcards

The **Find** window uses a string-matching facility to find object types within the selected containment, or entire repository.

- Use an asterisk (*) to match any number of characters.
- Use a question mark (?) to match one character.

RELATED TASKS
"Find An Object" on page 534

Find an Interface's Referencing Operations

To find the operations that reference a particular interface, follow these steps:

1. In the IR Browser, choose **Search - Find**.
2. Enter the name of the interface.
3. Under **Object Types**, select **interface**.
4. Under **Used By**, select the listed operations you are interested in.
5. Press the **Find** button.

RELATED TASKS
"Search Using Wildcards"
"Search by Object Type"

Search by Object Type

To find an attribute, constant, exception, interface, module, operation or type:

1. Select **Search - Find**
2. In the **Object Name** field, enter the name of the attribute, constant, exception, interface, module, operation or type.
3. Narrow the scope of the search by the selecting the appropriate **Object Type**.
   Note: Use the **Used By** buttons to restrict the search to a list of objects that *reference* the input objects.
4. Press the **Find** button.

All objects that match the search criteria are listed in the **Result** scrollbox. When you select an item from the list, that object is highlighted in the **Containment**view, and displayed in the **IDL** and **Inheritance** views.

RELATED TASKS

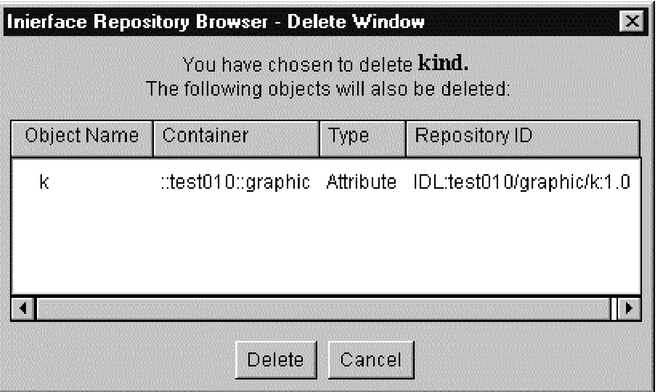
"Find an Interface's Referencing Operations" on page 535
"Search Using Wildcards" on page 535

Modify the Repository

Delete Objects from the Repository

Be aware that objects deleted from the interface repository cannot be restored. To permanently delete an object, follow these steps:

1. Allow updating of the interface repository by selecting **Options - Allow Updating Interface Repository**.
2. In either the Containment or Inheritance view, select the object you want to delete.
3. Select **Edit - Delete**. The IR Browser returns a dialog box listing any objects that will *also* be deleted as a result of your action.



4. Verify that you want to delete all of these objects by pressing the **Delete** button. Otherwise, press the **Cancel** button.

Index

A

activity
  in FlowMark
    defined 398
addresses valid 479
application DDL files
  defined 381
  editing 387
application family
  creating 375
applications
  packaging 375
  packaging in team environment 218
attributes 26
  adding 247
  deleting 249
  editing 248

B

bag
  adding 396
  deleting 407
  editing 406
  working with 406
breakpoints 499
  deferred, setting 449
  deleting 450
  disabling 450
  enabling 450, 499
  line breakpoint, setting 448
  modifying characteristics 450
  setting from the source window 447
  setting in breakpoints list window 447
  setting multiple 449
  supported in interpreted Java 446
  types of
    load occurrence 472
build configuration options 370
build process
  automated
    setting up 210
building DLLs 363
  scenario 368
business object
  adding from a data object 287
  behavior 27
  defined 17
  interface 17
  OO-SQL implementation methods
    customizing 275
  relationship 17
  setting implementation language 17
  working with 281
business object file
  creating 282
business object implementation
  adding, with data object interface 284
  composite
    adding, with data object interface 355
    editing 360
  deleting 291
  editing 290
business object interface
  adding 283
  composite
    adding 354
    editing 359
  creating
    by importing an IDL file 289
  deleting 291
  editing 290
business object module
  adding 282

C

C++
  debugging, supported data types 476
  debugging, supported expression operands 474
  debugging, supported expression operators 475
  debugging of 435
  debugging of class members 472
  limitations when debugging 476
  VisualAge C++ compiler options for debugging 427
call stack, view 462
change control
  defined 202
  managing a team environment 202
  managing information 202
  process
    setting up 209
  system 202
child component
  with attributes duplication
    defining 142
  with key duplication
    defining 149
  with single datastore
    defining 156
  with views
    defining 164
classes
  C++, Java
    importing 13
client application
  adding 376
client DLL
  defining 364
code
  generating 363
Compare and Merge Tool for XML 228
  comparing files with 228
  merging files with 229

© Copyright IBM Corp. 1997, 1998

compiling
programs for debugging 427
programs for OLT 486
complex attributes
associating with persistent objects 263
defined 263
mapping patterns
Explode 263
Primitive 263
component
assembly 16
creating
for existing DB data 104
for new DB data 101
for PA data 115
for transient data 101
execution 16
Component Broker
applications 3
architectural layers 3
design principles for 3
frameworks
importing 86
component instance
creating
through FlowMark 404
deleting
through FlowMark 405
component method
calling from FlowMark 404
components
calling methods on 15
defined 15
objects 15
composite business objects
attributes 175
defined 175
helper objects 175
key 175
methods 175
working with 353
composite component
Conjunction 173
creating 173
overview 172
defined 173
Disjunction 173
objects, composed of 173
composite keys
adding 360
defined 176
editing 362
using for location of composition components 176
working with 360
composition
adding 350
class source files 174
creating composite business objects from 174
defined 174
editing 352
helper objects 174
composition *(continued)*
modules
adding 349
objects, composed of 174
restrictions 425
working with 348
composition file
creating 349
constructs
constant 26
defined 26
deleting 280
editing 280
enumeration 26
exception 26
structure 26
typedef 26
union 26
with file scope
defining 278
with interface scope
defining 279
with module scope
defining 279
working with 277
container 345
container instances
creating 346
deleting 348
editing 348
working with 345
copy helpers
adding 294
attributes 21
defined 21
deleting 295
editing 295
implementations 21
instances 21
critical sections 469
customized homes
creating 343
deleting 345
editing 344
working with 342

D

data access pattern 34
data encoding schemes 109
binary 5
double byte character set 5
data object
adding
from a DB persistent object 304
from a PA persistent object 305
adding from a business object 302
behavior 30
defined 18
implementation 18
interface 18

data object *(continued)*
  using 18
data object file
  creating 303
data object implementation
  adding 299
  deleting 313
  editing 310
data object interface 29
  creating 297
    by importing an IDL file 306
  deleting 312
  editing 309
data object module
  adding 304
data objects
  working with 296
data structures
  adding
    input 397
  deleting 408
  editing 408
  FlowMark 396
  input 396
  member 396
  output 396
  working with 408
data type mappings
  DB2 110
  Oracle 113
DB (database) persistent objects
  deleting 317
DB (database) schemas
  deleting 333
DB persistent objects
  editing 317
DB schema group
  deleting 320
  editing 319
DB schemas
  creating
    by importing an SQL file 321
  editing 329
DDL
  file
    structure 389
  files
    objects 389
  SQL 114
DDL (Data Definition Language)
  system management 114
DDL Editor
  defined 382
  file 382
  process 382
  using 382
DDL files 384
  applications 381
  creating, editing 384
deadlocks 470
debug on demand 438
debugger monitors explained 457
debugger windows explained 440
debugging
  attaching to a running process 435
  behaviour of the debugger at startup 439
  client applications from startup 497
  command-line parameters 436
  compiling programs 427
  halting execution 455
  invoking the debugger 433
  monitoring expressions and variables 458
  of distributed applications 495
  options 434
  race conditions 471
  remote 442
  remote, starting program 443
  restarting a program 455
  running a program 453
  search order 439
  skipping sections of code 455
  source window views, explained 441
  starting the debugger 433
  step commands 442, 453
  stepping and functions 454
  terminating a debug session 456
  troubleshooting
    code you did not write 442
    debugger cannot find source code 478
    debugger is using different executable version 478
    problems getting a source or mixed view 441
  view a location in storage 461
  window system lockups 473
  writing programs 427
defined 395
dependencies
  cross-project
    managing 230
design patterns
  defined 107
  iterators 107
designing in Rose 73
Development
  Java
    requirements for 8
  Multi-platform
    code generation 187
    constraints 187
    method implementation 187
    views 187
DLLs
  debugging 451
    from the breakpoints list 452
    from the load occurrence dialog 451
    from the session control window 452
    from the source window 452

E

Enterprise Access Builder (EAB)
  defined 116

Enterprise Access Builder (EAB) *(continued)*
system
managing a team environment 116
managing information 116
environment 31
environment variables
debugger 432
CLASSPATH 432
INCLUDE 432
IVB_DBG_CASESENSITIVE 430
IVB_DBG_LANG 430
IVB_DBG_LOCAL_PATH 430
IVB_DBG_NUMBEROFELEMENTS 431
IVB_DBG_OVERRIDE 431
IVB_DBG_PATH 431
IVB_DBG_REMOTE_SEARCH_PATH 431
IVB_DBG_TAB 431
IVB_DBG_TABGRID 432
setting for the debugger 429
exported design
working with 91

F

filters
available 9
creating
for viewing objects pane 10
creating new 9
FlowMark 395
databases 395
Definition Language 395
workflow manager 395
FlowMark Bag
bag
defined 395
programs 395
FlowMark business objects
working with 403
foreign key
patterns 132
relationships 132
foreign key pattern
defining 133
form of persistent behavior, implementation 32
framework methods
calling 24
defined 24
editing 270
special 24

G

get and set methods
defined 23
editing 270

H

handles
for storing pointers 35
heap use
debugging of 465, 466
home
defined 342
instance 342
specialized (customized) 342

I

IDL (Interface Definition Language) files
dependencies within 129
inheritance
abstract base class 140
data object implementation 36
defined 137
recommended, for component objects 137
with attributes duplication 141
with key duplication 147
with single datastore 155
with views 162
inheritance and overriding
in business objects 138
in data objects 139
inheritance pattern
for persistence 140
initializer method
adding 268
input parameters
mapping to input data structure 400
install image
generating 379
integration project
adding to team environment 208
Interface Repository Browser 533

J

Java
applets, debugging 437
attaching debugger to a running JVM 437
breakpoints supported 446
compiler options for debugging 428
expressions supported when debugging 477
limitations when debugging interpreted 477

K

key
adding 292
defined 21
deleting 293
editing 293
implementations 21
using 21
key and copy helper
inheritance 138
overriding 138
key assistant
defined 22
interface 22

L

load occurrence breakpoints 472

M

makefiles
  generating 367
managed object
  adding 22, 340
  configuring 22, 377
  defined 22
  deleting 341
  editing 341
  using 22
  working with 339
managed object configuration
  deleting 379
  editing 379
mapping
  attributes
    using a key 258
    using a mapping helper 260
    using the default mapping pattern 257
  business object
    to data object 288
  complex attributes
    using the Explode pattern 265
  component
    to data structure 400
  data object
    to child's persistent object 255
    to DB persistent object 251
    to parent's persistent object 254
mapping helper
  class 105
  file
    default 105
  methods 105
  providing your own 105
  using 105
mapping rules
  Object Builder to Rose 87
method body
  external files for 273
methods
  deleting 277
  for public attributes 23
  get 23
  importing changes 272
  push-down
    in Enterprise Access Builder 25
    in Object Builder 25
    using 25
    using ECI 25
    using HOD 25
  relationship
    using 25
  set 23
  User-Defined
    defining 23
    providing method bodies 23
model
  checking for consistency 412

O

Object Builder
  components 1
  defined 1
  getting started with 39
  panes 1
  preferences
    setting 7
  running in batch mode 11
  starting 1
Object Level Trace
  command-line parameters 527
  environment file 525
  languages and platforms supported 484
  monitoring modes 485
  opening a trace file 508
  overview 481
  reorder trace lines 506
  saving the display 508
  scenarios 509
    C++ client and BO on AIX 522
    debug Java client from startup 515
    Java client and C++ BO 512
    step by step debug mode 518
  Start the components on separate machines 490
  trace display symbols, explained 500
  troubleshooting
    distributed debugging 530
    java clients 529
    OS/390 531
    real time display 531
    startup 528
object reference 29
  storing 135
optimized code
  debugging of 467, 468
OS/390
  tracing applications with OLT 492
output parameters
  mapping to input data structure 402
  mapping to output data structure 401

P

PA (Procedural Adaptor) persistent objects
  deleting 336
PA persistent objects
  editing 336
  using push-down methods with 274
  working with 333
PA schema
  creating
    by importing a PA bean 337
PA schemas
  deleting 339
  editing 339
  working with 337

partial order display 502
pass ticket
composition 372
for OS/390 372
in RACF 372
using 372
performance analysis 504
persistent object
adding from DB schema 316
adding from PA schema 334
database (DB) 19
ESQL framework methods
customizing 276
implementation
database caching 19
database embedded SQL 19
procedural adaptor (PA) 19
persistent object and schema, adding 313
platform constraints
setting 189
platform differences 188
procedural adaptor (PA)
bean
importing 117
persistent object 117
schema 117
process activity 398
process list dialog 436
profile
file 372
for remote OS/390 build 372
program
activity 397
adding 398
deleting 410
editing 409
in FlowMark 397
registration 397
working with 409
program activity 398
project
creating
in a team environment 215
editing
in a team environment 216
importing into Rose 92
moving 227
splitting
for team development 206
starting 6
project divisions
changing 227
projects
directories 4
files 4
migrating, old 7
model name
using 4
organization 4
subdirectories 4

R

Rational Rose
class properties
exporting 81
class relationships
associations and aggregations 84
exporting 84
inheritance 84
classes
mapping to Object Builder classes 81
Component Broker Frameworks in 89
constructs 79
exporting 79
defined 74
exporting from Rose 89
IDL name scoping in 77
setting up 74
using 74
Rational Rose design
exporting to team environment 204
real time display 503
record of directories 440
referential integrity
customizing 108
registers
changing the contents of 463
changing which are displayed 464
floating-point 464
viewing the contents of 461
registers monitor 464
relationship
circular
defining 132
one-to-many
defining 131
one-to-one
defining 130
remote build 372
launching 373
remote debugging 488
restrictions
Object Builder 419
Rose Bridge
exporting design to Object Builder project 76
Rose Bridge, the
importing design into Rose 76
loading Component Broker frameworks 76
re-exporting design 76

S

scenarios
building DLLs or shared library files 47
creating component 39
for new DB data 102
for PA data 118
creating composite component 177
develop multi-platform applications 190
exporting from Rose 95
importing attribute changes into Rose 98

scenarios *(continued)*
inheriting with attributes duplication 144
inheriting with key duplication 151
inheriting with single datastore 158
inheriting with views 165
installing and running applications 61
installing and running applications with InstallShield 57
launching remote OS/390 build 373
packaging an application 50
team development with Rose 218
tracing and debugging applications 65
uninstalling an application 71
uninstalling an application using InstallShield 70
unit testing for procedural adaptors 126
schema
DB
creating 20
naming 20
group
naming 20
PA 20
schema groups
creating 318
selected event 501
server application
adding 377
server DLL
defining 366
Session service 30
sessional business object
adding endResource() to 117
SmartGuide Customizer for XML
starting 234
SmartGuides
constraining values in 240
constraints 241
creating 233
deriving values in 237
distributing 245
editing 244
in SmartGuide Customizer for XML
using 233
layout
defining 242
macros
defining 235
propagating values in 239
running 243
testing 243
value lists
customizing 237
source language statements, debugging of 473
special framework methods
del() 24
editing 271
insert() 24
retrieve() 24
setConnection() 24
update() 24
SQL clauses
using complex relatonships in 326
SQL file
generated
editing 331
re-importing 330
SQL View Editor 323
state data 18
pattern for handling 27
step by step debugging 498
storage monitor, change address displayed 459
storage monitor, opening 458

T

tagging events 507
Tasks and Objects pane
filtering 9
searching 10
team development
defined 201
environment
working with a Rose package 201
working with an Object Builder project 201
setting up 211
team environment
building DLLs in 217
deleting projects in 217
importing projects from 212
maintaining 223
setting up 204
working in 212
threads, debugging of 468, 473
tracing
distributed applications 489
Transaction Object 116
Transaction Record 116
troubleshooting
inability to start Object Builder on AIX 411
memory problems 411
odd behavior 411

U

user-defined methods
adding code for 267
editing 269

V

variable contents, viewing 460
view
creating
with SQL View Editor 324
editing 328
with SQL View Editor 325

W

What's New
Compare and Merge Tool for XML 2

What's New *(continued)*
filtering 2
FlowMark support 2
miscellaneous product changes 2
Model Consistency Checker 2
OS/390 support, extended 2
SmartGuide Customizer for XML 2
Tasks and Objects pane, finding objects in 2
team development, easier use of 2
wizards 2

X

XML
exporting 224
importing 225
model interchange with 203
XML wizards
constraining values in 240
constraints 241
creating 233
deriving values in 237
distributing 245
editing 244
in SmartGuide Customizer for XML
using 233
layout
defining 242
macros
defining 235
propagating values in 239
running 243
testing 243
value lists
customizing 237

IBM

Printed in the United States of America

SC09-2705-03