
CORBA services: Common Object Services Specification

Revised Edition - March 31, 1995
Updated: March 28, 1996
Updated: July 15, 1996
Updated: November 22, 1996
Updated: March 1997
Updated: July 1997

Copyright 1996, AT&T/Lucent Technologies, Inc.
Copyright 1995, 1996 AT&T/NCR
Copyright 1995, 1996 BNR Europe Limited
Copyright 1996, Cooperative Research Centre for Distributed Systems Technology (DSTC Pty Ltd).
Copyright 1995, 1996 Digital Equipment Corporation
Copyright 1996, Gradient Technologies, Inc.
Copyright 1995, 1996 Groupe Bull
Copyright 1995, 1996 Hewlett-Packard Company
Copyright 1995, 1996 HyperDesk Corporation
Copyright 1995, 1996 ICL plc
Copyright 1995, 1996 Ing. C. Olivetti & C.Sp
Copyright 1995, 1996 International Business Machines Corporation
Copyright 1996, International Computers Limited
Copyright 1995, 1996 Iona Technologies Ltd.
Copyright 1995, 1996 Itasca Systems, Inc.
Copyright 1996, Nortel Limited
Copyright 1995, 1996 Novell, Inc.
Copyright 1995, 1996 O2 Technologies
Copyright 1995, 1996 Object Design, Inc.
Copyright 1995, 1996 Object Management Group, Inc.
Copyright 1995, 1996 Objectivity, Inc.
Copyright 1995, 1996 Ontos, Inc.
Copyright 1995, 1996 Oracle Corporation
Copyright 1995, 1996 Persistence Software
Copyright 1995, 1996 Servio, Corp.
Copyright 1995, 1996 Siemens Nixdorf Informationssysteme AG
Copyright 1995, 1996 Sun Microsystems, Inc.
Copyright 1995, 1996 SunSoft, Inc.
Copyright 1996, Sybase, Inc.
Copyright 1996, Taligent, Inc.
Copyright 1995, 1996 Tandem Computers, Inc.
Copyright 1995, 1996 Teknekron Software Systems, Inc.
Copyright 1995, 1996 Tivoli Systems, Inc.
Copyright 1995, 1996 Transarc Corporation
Copyright 1995, 1996 Versant Object Technology Corporation

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version.

Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright, in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

NOTICE

The information contained in this document is subject to change without notice.

The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The Object Management Group and the companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013

OMG® and Object Management are registered trademarks of the Object Management Group, Inc.
Object Request Broker, OMG IDL, ORB, CORBA, CORBAfacilities, CORBAservices, and COSS are trademarks of the Object Management Group, Inc.

X/Open is a trademark of X/Open Company Ltd.

Table of Contents

0.1	About This Document	xli
0.1.1	Object Management Group	xli
0.1.2	X/Open	xlii
0.2	Intended Audience	xlii
0.3	Need for Object Services	xlii
0.3.1	What Is an Object Service Specification?	xliii
0.4	Associated Documents	xliii
0.5	Structure of this Manual	xliv
0.6	Acknowledgements	xliv
1.	Overview	1-1
1.1	Summary of Key Features	1-1
1.1.1	Naming Service	1-1
1.1.2	Event Service	1-2
1.1.3	Life Cycle Service	1-2
1.1.4	Persistent Object Service	1-3
1.1.5	Transaction Service	1-3
1.1.6	Concurrency Control Service	1-3
1.1.7	Relationship Service	1-4
1.1.8	Externalization Service	1-4
1.1.9	Query Service	1-5
1.1.10	Licensing Service	1-5
1.1.11	Property Service	1-5
1.1.12	Time Service	1-6
1.1.13	Security Service	1-6
1.1.14	Object Trader Service	1-7

2.	General Design Principles.	2-1
2.1	Service Design Principles.	2-1
2.1.1	Build on CORBA Concepts	2-1
2.1.2	Basic, Flexible Services	2-2
2.1.3	Generic Services.	2-2
2.1.4	Allow Local and Remote Implementations.	2-2
2.1.5	Quality of Service is an Implementation Characteristic	2-2
2.1.6	Objects Often Conspire in a Service.	2-2
2.1.7	Use of Callback Interfaces	2-4
2.1.8	Assume No Global Identifier Spaces	2-4
2.1.9	Finding a Service is Orthogonal to Using It	2-4
2.2	Interface Style Consistency	2-4
2.2.1	Use of Exceptions and Return Codes	2-4
2.2.2	Explicit Versus Implicit Operations	2-5
2.2.3	Use of Interface Inheritance	2-5
2.3	Key Design Decisions	2-5
2.3.1	Naming Service: Distinct from Property and Trading Services.	2-5
2.3.2	Universal Object Identity	2-5
2.4	Integration with Future Object Services.	2-5
2.4.1	Archive Service	2-6
2.4.2	Backup/Restore Service	2-6
2.4.3	Change Management Service.	2-6
2.4.4	Data Interchange Service	2-6
2.4.5	Internationalization Service	2-6
2.4.6	Implementation Repository	2-7
2.4.7	Interface Repository	2-7
2.4.8	Logging Service	2-7
2.4.9	Recovery Service	2-8
2.4.10	Replication Service.	2-8
2.4.11	Startup Service	2-9
2.4.12	Data Interchange Service	2-9
2.5	Service Dependencies	2-9
2.5.1	Event Service	2-9
2.5.2	Life Cycle Service	2-9
2.5.3	Persistent Object Service	2-9
2.5.4	Relationship Service.	2-10
2.5.5	Externalization Service.	2-10
2.5.6	Transaction Service	2-10

2.5.7	Concurrency Control Service	2-11
2.5.8	Query Service	2-11
2.5.9	Licensing Service	2-11
2.5.10	Property Service	2-12
2.5.11	Time Service	2-12
2.5.12	Security Service	2-12
2.6	Relationship to CORBA	2-12
2.6.1	ORB Interoperability Considerations: Transaction Service	2-12
2.6.2	Life Cycle Service	2-13
2.6.3	Naming Service	2-13
2.6.4	Relationship Service	2-13
2.6.5	Persistent Object Service	2-13
2.6.6	General Interoperability Requirements	2-13
2.7	Relationship to Object Model	2-13
2.8	Conformance to Existing Standards	2-14
3.	Naming Service Specification	3-1
3.1	Service Description	3-1
3.1.1	Overview	3-1
3.1.2	Names	3-2
3.1.3	Names Library	3-3
3.1.4	Example Scenarios	3-3
3.1.5	Design Principles	3-4
3.1.6	Resolution of Technical Issues	3-5
3.2	The CosNaming Module	3-6
3.2.1	Binding Objects	3-8
3.2.2	Resolving Names	3-9
3.2.3	Unbinding Names	3-10
3.2.4	Creating Naming Contexts	3-11
3.2.5	Deleting Contexts	3-11
3.2.6	Listing a Naming Context	3-12
3.2.7	The BindingIterator Interface	3-12
3.3	The Names Library	3-13
3.3.1	Creating a Library Name Component	3-14
3.3.2	Creating a Library Name	3-15
3.3.3	The LNameComponent Interface	3-15
3.3.4	The LName Interface	3-15
	Destroying a Library Name Component Pseudo Object 3-16	
	Inserting a Name Component 3-16	

Getting the i^{th} Name Component	3-16
Deleting a Name Component	3-17
Number of Name Components	3-17
Testing for Equality	3-17
Testing for Order	3-17
Producing an IDL form	3-18
Translating an IDL Form	3-18
Destroying a Library Name Pseudo-Object	3-18

4. Event Service Specification	4-1
4.1 Service Description	4-1
4.1.1 Overview	4-1
4.1.2 Event Communication	4-2
4.1.3 Example Scenario	4-2
4.1.4 Design Principles	4-4
4.1.5 Resolution of Technical Issues	4-4
4.1.6 Quality of Service	4-6
4.2 Generic Event Communication	4-6
4.2.1 Push Model	4-6
4.2.2 Pull Model	4-7
4.3 The CosEventComm Module	4-8
4.3.1 The PushConsumer Interface	4-8
4.3.2 The PushSupplier Interface	4-9
4.3.3 The PullSupplier Interface	4-9
4.3.4 The PullConsumer Interface	4-10
4.4 Event Channels	4-10
4.4.1 Push-Style Communication with an Event Channel	4-10
4.4.2 Pull-Style Communication with an Event Channel	4-11
4.4.3 Mixed Style Communication with an Event Channel	4-11
4.4.4 Multiple Consumers and Multiple Suppliers	4-12
4.4.5 Event Channel Administration	4-13
4.5 The CosEventChannelAdmin Module	4-15
4.5.1 The EventChannel Interface	4-16
4.5.2 The ConsumerAdmin Interface	4-17
4.5.3 The SupplierAdmin Interface	4-17
4.5.4 The ProxyPushConsumer Interface	4-17
4.5.5 The ProxyPullSupplier Interface	4-18
4.5.6 The ProxyPullConsumer Interface	4-18
4.5.7 The ProxyPushSupplier Interface	4-19
4.6 Typed Event Communication	4-19

4.6.1	Typed Push Model	4-19
4.6.2	Typed Pull Model	4-20
4.7	The CosTypedEventComm Module	4-21
4.7.1	The TypedPushConsumer Interface	4-22
4.7.2	The TypedPullSupplier Interface	4-23
4.8	Typed Event Channels	4-23
4.9	The CosTypedEventChannelAdmin Module	4-24
4.9.1	The TypedEventChannel Interface	4-26
4.9.2	The TypedConsumerAdmin Interface	4-26
4.9.3	The TypedSupplierAdmin Interface	4-27
4.9.4	The TypedProxyPushConsumer Interface	4-28
4.9.5	The TypedProxyPullSupplier Interface	4-28
4.10	Composing Event Channels and Filtering	4-28
4.11	Policies for Finding Event Channels	4-29
Appendix A	Implementing Typed Event Channels	4-30
Appendix B	An Event Channel Use Example	4-32
5.	Persistent Object Service Specification	5-1
5.1	Introduction	5-1
5.2	Goals and Properties	5-3
5.2.1	Basic Capabilities	5-3
5.2.2	Object-oriented Storage	5-3
	Interfaces to Data 5-4	
	Self-description 5-4	
	Abstraction 5-4	
5.2.3	Open Architecture	5-4
5.2.4	Views of Service	5-5
	Client 5-5	
	Object Implementation 5-6	
	Persistent Data Service 5-6	
	Datastore 5-6	
5.3	Service Structure	5-7
5.4	The CosPersistencePID Module	5-8
5.4.1	PID Interface	5-9
5.4.2	Example PIDFactory Interface	5-11
5.5	The CosPersistencePO Module	5-11
5.5.1	The PO Interface	5-12
5.5.2	The POFactory Interface	5-14
5.5.3	The SD Interface	5-14
5.6	The CosPersistencePOM Module	5-15
5.7	Persistent Data Service (PDS) Overview	5-18

5.8	The CosPersistencePDS Module	5-19
5.9	The Direct Access (PDS_DA) Protocol	5-21
5.10	The CosPersistencePDS_DA Module	5-21
5.10.1	The PID_DA Interface	5-23
5.10.2	The Generic DAObject Interface	5-24
5.10.3	The DAObjectFactory Interface	5-24
5.10.4	The DAObjectFactoryFinder Interface	5-25
5.10.5	The PDS_DA Interface.	5-25
5.10.6	Defining and Using DA Data Objects	5-26
5.10.7	The DynamicAttributeAccess Interface	5-28
5.10.8	The PDS_ClusteredDA Interface	5-29
5.11	The ODMG-93 Protocol.	5-30
5.12	The Dynamic Data Object (DDO) Protocol	5-30
5.13	The CosPersistenceDDO Module.	5-31
5.14	Other Protocols	5-33
5.15	Datstores: CosPersistenceDS_CLI Module	5-34
5.15.1	The UserEnvironment Interface	5-36
5.15.2	The Connection Interface	5-37
5.15.3	The ConnectionFactory Interface	5-37
5.15.4	The Cursor Interface.	5-38
5.15.5	The CursorFactory Interface.	5-38
5.15.6	The PID_CLI Interface.	5-38
5.15.7	The Datastore_CLI Interface	5-40
5.16	Other Datstores.	5-43
5.17	Standards Conformance	5-43
5.18	References	5-43
6.	Life Cycle Service Specification	6-1
6.1	Service Description	6-1
6.1.1	Overview	6-1
6.1.2	Organization of this Chapter.	6-3
6.1.3	Client's Model of Object Life Cycle.	6-4
	Client's Model of Creation 6-4	
	Client's Model of Deleting an Object 6-6	
	Client's Model of Copying or Moving an Object 6-6	
6.1.4	Factory Finders.	6-7
	Multiple Factory Finders	6-8
6.1.5	Design Principles	6-8
6.1.6	Resolution of Technical Issues	6-9
6.2	The CosLifeCycle Module	6-10

6.2.1	The LifecycleObject Interface	6-11
	copy 6-11	
	move 6-12	
	remove 6-13	
6.2.2	The FactoryFinder Interface	6-13
	find_factories 6-13	
6.2.3	The GenericFactory Interface	6-14
	create_object 6-15	
	supports 6-16	
6.2.4	Criteria	6-17
6.3	Implementing Factories	6-18
6.3.1	Minimal Factories	6-19
6.3.2	Administered Factories	6-19
6.4	Target's Use of Factories and Factory Finders	6-21
6.5	Summary of Life Cycle Service	6-21
6.5.1	Summary of Life Cycle Service Structure ..	6-22
Appendix A	Addendum to Life Cycle Service: Compound Life Cycle Specification	6-23
Appendix B	Filters	6-47
Appendix C	Administration	6-51
Appendix D	Support for PCTE Objects	6-59
7.	Concurrency Control Service	7-1
7.1	Service Description	7-1
7.1.1	Basic Concepts of Concurrency Control.	7-1
	Clients and Resources 7-1	
	Transactions as Clients 7-2	
	Locks 7-2	
	Lock Modes 7-2	
	Lock Granularity 7-2	
	Conflict Resolution 7-3	
	Conflict Resolution for Transactions 7-3	
	Lock Duration 7-3	
7.2	Locking Model	7-3
7.2.1	Lock Modes	7-4
	Read, Write, and Upgrade Locks 7-4	
	Intention Read and Intention Write Locks 7-4	
	Lock Mode Compatibility 7-5	
7.2.2	Multiple Possession Semantics	7-5
7.3	Two-Phase Transactional Locking	7-6
7.4	Nested Transactions	7-6
7.5	CosConcurrencyControl Module	7-7
7.5.1	Types and Exceptions	7-9
7.5.2	LockCoordinator Interface	7-9

	7.5.3	LockSet Interface	7-10
	7.5.4	TransactionalLockSet Interface	7-11
	7.5.5	LockSetFactory Interface	7-13
8.		Externalization Service Specification	8-1
8.1		Service Description	8-1
8.2		Service Structure	8-2
	8.2.1	Client's Model of Object Externalization . . .	8-2
	8.2.2	Stream's Model of Object Externalization . .	8-3
	8.2.3	Object's Model of Externalization	8-4
	8.2.4	Object's Model of Internalization	8-5
8.3		Object and Interface Hierarchies	8-7
8.4		Interface Summary	8-10
		Externalization Service Architecture: Audience/Bearer Mapping 8-11	
8.5		CosExternalization Module	8-12
	8.5.1	StreamFactory Interface	8-12
		Creating a Stream Object 8-12	
	8.5.2	FileStreamFactory Interface	8-13
		Creating a Stream Object Associated with a File 8-13	
	8.5.3	Stream Interface	8-13
		Externalizing an Object 8-13	
		Externalizing Groups of Objects 8-14	
		Internalizing an Object 8-14	
8.6		CosStream Module	8-15
	8.6.1	The StreamIO Interface	8-16
	8.6.2	The Streamable Interface	8-17
		Writing the Object's State to a Stream 8-18	
		Reinitializing the Object's State from a Stream 8-18	
	8.6.3	The StreamableFactory Interface	8-19
		Creating a Streamable Object 8-19	
8.7		CosCompound Externalization Module	8-19
	8.7.1	The Node Interface	8-21
		Externalizing a Node 8-21	
		Internalizing a Node 8-21	
	8.7.2	The Role Interface	8-22
		Externalizing a Role 8-22	
		Internalizing a Role 8-23	
		Getting a Propagation Value 8-23	
	8.7.3	The Relationship Interface	8-23
		Externalizing the Relationship 8-23	
		Internalizing the Relationship 8-23	
		Getting a Propagation Value 8-24	

8.7.4	The PropagationCriteriaFactory Interface . . . Create a Traversal Criteria Based on Externalization Propagation 8-24	8-24
8.8	Specific Externalization Relationships	8-25
8.9	The CosExternalizationContainment Module	8-26
8.10	The CosExternalizationReference Module	8-28
8.11	Standard Stream Data Format	8-29
8.11.1	OMG Externalized Object Data	8-29
8.11.2	Externalized Repeated Reference Data	8-30
8.11.3	Externalized NIL Data	8-31
8.12	References	8-31
9.	Relationship Service Specification	9-1
9.1	Service Description	9-1
9.1.1	Key Features of the Relationship Service	9-2
9.1.2	The Relationship Service vs. CORBA Object References Relationships that Are Multidirectional 9-4 Relationships that Allow Third Party Manipulation 9-4 Traversals that Are Supported for Graphs of Related Objects 9-4 Relationships and Roles that Can Be Extended with Attributes and Behavior 9-4	9-3
9.1.3	Resolution of Technical Issues Modeling and Relationship Semantics 9-4 Managing Relationships 9-5 Constraining Relationships 9-5 Referential Integrity 9-5 Relationships and Roles as First Class Objects 9-5 Different Models for Navigating and Constructing Relationships 9-6 Efficiency Considerations 9-6	9-4
9.2	Service Structure	9-7
9.2.1	Levels of Service Level One: Base Relationships 9-7 Level Two: Graphs of Related Objects 9-8 Level Three: Specific Relationships 9-9	9-7
9.2.2	Hierarchy of Relationship Interface	9-10
9.2.3	Hierarchy of Role Interface	9-10
9.2.4	Interface Summary	9-11
9.3	The Base Relationship Model	9-13
9.3.1	Relationship Attributes and Operations Rationale	9-14 9-15
9.3.2	Higher Degree Relationships Rationale	9-15 9-15

9.3.3	Operations	9-17
	Creation 9-17	
	Navigation 9-18	
	Destruction 9-18	
9.3.4	Consistency Constraints	9-18
9.3.5	Implementation Strategies	9-19
9.3.6	The CosObjectIdentity Module	9-19
	The IdentifiableObject Interface 9-19	
	constant_random_id9-20	
	is_identical 9-20	
9.3.7	The CosRelationships Module	9-20
	Example of Containment Relationships9-23	
	The RelationshipFactory Interface 9-23	
	The Relationship Interface 9-25	
	Destroying a Relationship 9-26	
	The Role Interface 9-26	
	The RoleFactory Interface 9-30	
	The RelationshipIterator Interface 9-32	
9.4	Graphs of Related Objects	9-33
9.4.1	Graph Architecture	9-33
	Nodes 9-35	
9.4.2	Traversing Graphs of Related Objects	9-35
	Detecting and Representing Cycles 9-35	
	Determining the Relevant Nodes and Edges 9-36	
9.4.3	Compound Operations	9-36
9.4.4	An Example Traversal Criteria	9-37
	Propagation 9-37	
9.4.5	The CosGraphs Module	9-38
	The TraversalFactory Interface 9-41	
	The Traversal Interface 9-42	
	The TraversalCriteria Interface 9-43	
	The Node Interface 9-44	
	The NodeFactory Interface 9-46	
	The Role Interface 9-46	
	The EdgeIterator Interface 9-47	
9.5	Specific Relationships	9-47
9.5.1	Containment and Reference	9-48
9.5.2	The CosContainment Module	9-48
9.5.3	The CosReference Module	9-50
9.6	References	9-51
10.	Transaction Service Specification	10-1
10.1	Service Description	10-1
10.1.1	Overview of Transactions	10-2
10.1.2	Transactional Applications	10-2
10.1.3	Definitions	10-3
	Transactional Client 10-4	

	Transactional Object	10-4
	Recoverable Objects and Resource Objects	10-5
	Transactional Server	10-6
	Recoverable Server	10-6
10.1.4	Transaction Service Functionality	10-6
	Transaction Models	10-6
	Transaction Termination	10-7
	Transaction Integrity	10-8
	Transaction Context	10-8
10.1.5	Principles of Function, Design, and Performance	10-8
	Functional Requirements	10-8
	Design Requirements	10-10
	Performance Requirements	10-11
10.2	Service Architecture.	10-12
10.2.1	Typical Usage	10-12
10.2.2	Transaction Context	10-13
10.2.3	Context Management	10-14
10.2.4	Datatypes	10-15
10.2.5	Exceptions	10-15
	Standard Exceptions	10-15
	Heuristic Exceptions	10-16
	WrongTransaction Exception	10-17
	Other Exceptions	10-17
10.3	Transaction Service Interfaces	10-17
10.3.1	Current Interface.	10-18
	begin	10-18
	commit	10-19
	rollback	10-19
	rollback_only	10-19
	get_status	10-19
	get_transaction_name	10-20
	set_timeout	10-20
	get_control	10-20
	suspend	10-20
	resume	10-20
10.3.2	TransactionFactory Interface	10-21
	create	10-21
10.3.3	Control Interface.	10-21
	get_terminator	10-22
	get_coordinator	10-22
10.3.4	Terminator Interface	10-22
	commit	10-23
	rollback	10-23
10.3.5	Coordinator Interface	10-24
	get_status	10-24
	get_parent_status	10-24
	get_top_level_status	10-25
	is_same_transaction	10-25
	is_ancestor_transaction	10-25
	is_descendant_transaction	10-25

	is_related_transaction	10-25
	is_top_level_transaction	10-25
	hash_transaction	10-25
	hash_top_level_tran	10-26
	register_resource	10-26
	register_subtran_aware	10-26
	rollback_only	10-26
	get_transaction_name	10-27
	create_subtransaction	10-27
10.3.6	Recovery Coordinator Interface	10-27
	replay_completion	10-27
10.3.7	Resource Interface	10-27
	prepare	10-28
	rollback	10-29
	commit	10-29
	commit_one_phase	10-29
	forget	10-29
10.3.8	Subtransaction Aware Resource Interface	10-29
	commit_subtransaction	10-30
	rollback_subtransaction	10-30
10.3.9	TransactionalObject Interface.	10-30
10.4	The User's View.	10-31
10.4.1	Application Programming Models	10-31
	Direct Context Management: Explicit Propagation	10-31
	Indirect Context Management: Implicit Propagation	10-31
	Indirect Context Management: Explicit Propagation	10-32
	Direct Context Management: Implicit Propagation	10-32
10.4.2	Interfaces	10-32
10.4.3	Checked Transaction Behavior	10-32
10.4.4	X/Open Checked Transactions	10-33
	Reply Check	10-34
	Commit Check	10-34
	Resume Check	10-34
10.4.5	Implementing a Transactional Client: Heuristic Completions	10-34
10.4.6	Implementing a Recoverable Server.	10-35
	Transactional Object	10-35
	Resource Object	10-35
	Reliable Servers	10-35
10.4.7	Application Portability	10-36
	Flat Transactions	10-36
	X/Open Checked Transactions	10-36
10.4.8	Distributed Transactions.	10-36
10.4.9	Applications Using Both Checked and Unchecked Services	10-36

10.4.10	Examples	10-37
	A Transaction Originator: Indirect and Implicit 10-37	
	Transaction Originator: Direct and Explicit 10-38	
	Example of a Recoverable Server 10-39	
	Example of a Transactional Object 10-40	
10.4.11	Model Interoperability	10-41
	Importing Transactions 10-41	
	Exporting Transactions 10-42	
	Programming Rules 10-43	
10.4.12	Failure Models	10-43
	Transaction Originator 10-43	
	Transactional Server 10-44	
	Recoverable Server 10-44	
10.5	The Implementors' View	10-44
10.5.1	Transaction Service Protocols	10-45
	General Principles 10-45	
	Normal Transaction Completion 10-46	
	Failures and Recovery 10-52	
	Transaction Completion after Failure 10-53	
10.5.2	ORB/TS Implementation Considerations	10-55
	Transaction Propagation 10-55	
	Transaction Service Interoperation 10-57	
	Transaction Service Portability 10-60	
10.5.3	Model Interoperability	10-63
10.6	The CosTransactions Module	10-65
10.6.1	The CosTSInteroperation Module	10-69
10.6.2	The CosTSPortability Module	10-69
Appendix A	Relationship of Transaction Service to TP Standards	10-70
Appendix B	Transaction Service Glossary	10-81
11.	Query Service Specification	11-1
11.1	Service Description	11-1
11.1.1	Overview	11-1
11.1.2	Design Principles	11-1
11.1.3	Architecture	11-2
	Query Evaluators: Nesting and Federation 11-3	
	Collections 11-4	
	Queryable Collections for Scope and Result 11-5	
	Query Objects 11-5	
11.1.4	Query Languages	11-6
	SQL Query 11-7	
	OQL 11-7	
	SQL Query = OQL 11-8	
11.1.5	Key Features	11-9
11.2	Service Structure	11-10

11.2.1	Overview	11-10
	Type One: Collections 11-10	
	Type Two: Query Framework 11-10	
11.2.2	Collection Interface Structure.	11-10
11.2.3	Query Framework Interface Hierarchy/ Structure	11-10
11.2.4	Interface Overview	11-11
11.3	The Collection Model	11-12
11.3.1	Common Types of Collections	11-12
11.3.2	Iterators	11-12
11.4	The CosQueryCollection Module.	11-14
11.4.1	The CollectionFactory Interface.	11-15
	Creating a Collection 11-16	
11.4.2	The Collection Interface	11-16
	Determining the Cardinality 11-16	
	Adding an Element 11-16	
	Adding Elements from a Collection 11-17	
	Inserting an Element 11-17	
	Replacing an Element 11-17	
	Removing an Element 11-17	
	Removing all Elements 11-18	
	Retrieving an Element 11-18	
	Creating an Iterator 11-18	
11.4.3	The Iterator Interface	11-18
	Accessing the Current Element 11-18	
	Resetting the Iteration 11-19	
	Testing for Completion of an Iteration 11-19	
11.5	The Query Framework Model	11-19
11.5.1	Query Evaluators	11-19
11.5.2	Queryable Collections	11-20
11.5.3	Query Managers	11-21
11.5.4	Query Objects.	11-21
11.6	The CosQuery Module	11-23
11.6.1	The QueryLanguageType Interfaces.	11-24
11.6.2	The QueryEvaluator Interface	11-25
	Determining the Supported Query Language Types 11-25	
	Determining the Default Query Language Type 11-25	
	Evaluating a Query 11-25	
11.6.3	The QueryableCollection Interface.	11-25
11.6.4	The QueryManager Interface	11-25
	Creating a Query Object 11-26	
11.6.5	The Query Interface	11-26
	Determining the Associated Query Manager 11-26	
	Preparing the Query for Execution 11-26	

	Executing the Query 11-26	
	Determining the Query Status 11-27	
	Obtaining the Query Result 11-27	
11.7	References	11-27
12.	Licensing Service Specification	12-1
12.1	Background On Existing License Management Products	12-1
12.1.1	Business Policy	12-2
12.1.2	License Types	12-2
12.1.3	A History of License Types	12-3
12.1.4	Asset Management	12-3
12.1.5	License Usage Practices	12-4
12.1.6	Scalability	12-4
12.1.7	Reliability	12-4
12.1.8	Legacy Applications	12-5
12.1.9	Security	12-6
12.1.10	Client/Server Authentication	12-6
12.1.11	Example: Application Acquiring and Releasing a Concurrent License	12-6
12.2	Service Description	12-7
12.2.1	Overview	12-7
12.2.2	Key Components of a Licensing System ...	12-8
	License Attributes 12-8	
	Licensing Policy 12-8	
	Interfaces Isolated From Business Policies 12-10	
12.2.3	Licensing in the CORBA Environment	12-10
12.2.4	Design Principles	12-12
12.2.5	Licensing Service Interfaces	12-13
	Interfaces are Mandatory 12-13	
	Constraints on Object Behavior 12-13	
12.2.6	Licensing Event Trace Diagram	12-14
12.3	The CosLicensing Module	12-16
12.3.1	LicenseServiceManager Interface	12-19
12.3.2	ProducerSpecificLicenseService Interface ..	12-19
12.4	References	12-21
Appendix A	Licensing Service Glossary	12-22
Appendix B	Use of Other Services	12-23
Appendix C	Producer Client Implementation Issues	12-27
Appendix D	Challenge Mechanism	12-30
13.	Property Service	13-1
13.1	Overview	13-1

13.1.1	Service Description	13-1
	Client's Model of Properties 13-2	
	Object's Model of Properties 13-2	
13.1.2	OMG IDL Interface Summary	13-3
13.1.3	Summary of Key Features	13-3
13.2	Service Interfaces.	13-4
13.2.1	CosPropertyService Module.	13-4
	Data Types 13-5	
	Exceptions 13-7	
13.2.2	PropertySet Interface	13-9
	Defining and Modifying Properties 13-9	
	define_properties 13-10	
	Listing and Getting Properties 13-11	
	get_all_property_names 13-11	
	get_property_value 13-11	
	get_properties 13-11	
	get_all_properties 13-12	
	Deleting Properties 13-12	
	delete_property 13-12	
	delete_properties 13-13	
	delete_all_properties 13-13	
	Determining If a Property Is Already	
	Defined 13-14	
13.2.3	PropertySetDef Interface	13-14
	Retrieval of PropertySet Constraints 13-15	
	get_allowed_properties 13-15	
	Defining and Modifying Properties with	
	Modes 13-15	
	define_properties_with_modes 13-16	
	Getting and Setting Property Modes 13-17	
	get_property_modes 13-18	
	set_property_mode 13-18	
	set_property_modes 13-19	
13.2.4	PropertiesIterator Interface.	13-19
	next_one, next_n 13-19	
	Destroying the Iterator 13-20	
13.2.5	PropertyNamesIterator Interface	13-20
	Resetting the Position in an Iterator 13-20	
	next_one, next_n 13-20	
	Destroying the Iterator 13-21	
13.2.6	PropertySetFactory Interface	13-21
13.2.7	PropertySetDefFactory Interface	13-22
Appendix A	Property Service IDL	13-29
14.	Time Service Specification	14-1
14.1	Introduction	14-1
14.1.1	Time Service Requirements	14-1
14.1.2	Representation of Time.	14-1
14.1.3	Source of Time	14-2

14.1.4	General Object Model	14-3
14.1.5	Conformance Points	14-4
14.2	Basic Time Service.	14-4
14.2.1	Object Model	14-4
14.2.2	Data Types	14-5
	Type TimeT 14-6	
	Type InaccuracyT 14-6	
	Type Tdft 14-6	
	Type UtcT 14-6	
	Type IntervalT 14-6	
	Enum ComparisonType 14-7	
	Enum TimeComparison 14-7	
	Enum OverlapType 14-7	
14.2.3	Exceptions	14-8
	TimeUnavailable 14-8	
14.2.4	Universal Time Object (UTO).	14-8
	Readonly attribute time 14-9	
	Readonly attribute inaccuracy 14-9	
	Readonly attribute tdf 14-9	
	Readonly attribute utc_time 14-9	
	Operation absolute_time 14-9	
	Operation compare_time 14-10	
	Operation time_to_interval 14-10	
	Operation interval 14-10	
14.2.5	Time Interval Object (TIO).	14-10
	Readonly attribute time_interval 14-10	
	Operation spans 14-11	
	Operation overlaps 14-11	
	Operation time 14-11	
14.2.6	Time Service.	14-11
	Operation universal_time 14-12	
	Operation secure_universal_time 14-12	
	Operation new_universal_time 14-12	
	Operation uto_from_utc 14-12	
	Operation new_interval 14-12	
14.3	Timer Event Service.	14-13
14.3.1	Object Model	14-13
14.3.2	Usage	14-13
14.3.3	Data Types	14-14
	Enum TimeType 14-14	
	Enum EventStatus 14-14	
	Type TimerEventT 14-15	
14.3.4	Exceptions	14-15
14.3.5	Timer Event Handler	14-15
	Attribute status 14-16	
	Operation time_set 14-16	
	Operation set_timer 14-16	
	Operation cancel_timer 14-16	
	Operation set_data 14-16	

14.3.6	Timer Event Service	14-16
	Operation register 14-17	
	Operation unregister 14-17	
	Operation event_time 14-17	
14.4	Conformance	14-17
Appendix A	Implementation Guidelines	14-18
Appendix B	Consolidated OMG IDL	14-20
Appendix C	Notes for Users	14-23
Appendix D	Extension Examples	14-25
Appendix E	References	14-28
15.	Security Service Specification	15-1
15.1	Introduction to Security	15-1
15.1.1	Why Security?	15-1
15.1.2	What Is Security?	15-1
15.1.3	Threats in a Distributed Object System	15-2
15.1.4	Summary of Key Security Features	15-3
15.1.5	Goals	15-3
	Simplicity 15-4	
	Consistency 15-4	
	Scalability 15-4	
	Usability for End Users 15-4	
	Usability of Administrators 15-5	
	Usability for Implementors 15-5	
	Flexibility of Security Policy 15-5	
	Independence of Security Technology 15-5	
	Application Portability 15-6	
	Interoperability 15-6	
	Performance 15-6	
	Object Orientation 15-6	
	Specific Security Goals 15-7	
	Security Architecture Goals 15-7	
15.2	Introduction to the Specification	15-8
15.2.1	Conformance to CORBA Security	15-9
15.2.2	Specification Structure	15-10
	Normative and Non-normative Material 15-10	
	Section Summaries 15-11	
	Proof of Concept 15-12	
15.3	Security Reference Model	15-12
15.3.1	Definition of a Security Reference Model	15-12
15.3.2	Principals and Their Security Attributes	15-14
15.3.3	Secure Object Invocations	15-15
	Establishing Security Associations 15-16	
	Message Protection 15-17	
15.3.4	Access Control Model	15-19
	Object Invocation Access Policy 15-20	

	Application Access Policy	15-20
	Access Policies	15-21
15.3.5	Auditing	15-23
15.3.6	Delegation	15-25
	Privilege Delegation	15-26
	Overview of Delegation Schemes	15-27
	Facilities Potentially Available	15-27
	Specifying Delegation Options	15-30
	Technology Support for Delegation Options	15-30
15.3.7	Non-repudiation	15-31
15.3.8	Domains	15-33
	Security Policy Domains	15-34
	Security Environment Domains	15-36
	Security Technology Domains	15-37
	Domains and Interoperability	15-38
15.3.9	Security Management and Administration	15-40
	Managing Security Policy Domains	15-40
	Managing Security Environment Domains	15-41
	Managing Security Technology Domains	15-41
15.3.10	Implementing the Model	15-41
15.4	Security Architecture	15-42
15.4.1	Different Users' View of the Security Model	15-42
	Enterprise Management View	15-42
	End User View	15-43
	Application Developer View	15-43
	Administrator's View	15-44
	Object System Implementor's View	15-45
15.4.2	Structural Model	15-46
	Application Components	15-47
	ORB Services	15-47
	Security Services	15-49
	Security Policies and Domain Objects	15-49
15.4.3	Security Technology	15-51
15.4.4	Basic Protection and Communications	15-52
	Environment Domains	15-52
	Component Protection	15-52
15.4.5	Security Object Models	15-54
	The Model as Seen by Applications	15-54
	Administrative Model	15-71
	The Model as Seen by the Objects	
	Implementing Security	15-75
	Summary of Objects in the Model	15-82
15.5	Application Developer's Interfaces	15-84
15.5.1	Introduction	15-84
	Security Functionality Conformance	15-85
	Introduction to the Interfaces	15-86
15.5.2	Finding Security Features	15-92
	Description of Facilities	15-92
	Interfaces	15-92

	Portability Implications 15-92	
15.5.3	Authentication of Principals	15-92
	Description of Facilities 15-92	
	Interfaces 15-93	
	Portability Implications 15-95	
15.5.4	Credentials	15-96
	Description of Facilities 15-96	
	Interfaces 15-96	
	Portability Implications 15-100	
15.5.5	Object Reference	15-100
	Description of Facilities 15-100	
	Interfaces 15-101	
	Portability Implications 15-104	
15.5.6	Security Operations on Current	15-104
	Description 15-104	
	Interfaces 15-105	
15.5.7	Security Audit	15-109
	Description of Facilities 15-109	
	Interfaces 15-109	
	Portability Implications 15-111	
15.5.8	Administering Security Policy	15-111
15.5.9	Use of Interfaces for Access Control	15-111
	Description of Facilities 15-111	
	Interfaces 15-112	
	Portability Implications 15-113	
15.5.10	Use of Interfaces for Delegation 15-113	
	Description of Facilities 15-113	
	Interfaces 15-114	
	Portability Implications 15-114	
15.5.11	Non-repudiation	15-115
	Description of Facilities 15-115	
	Interfaces 15-116	
15.6	Administrator's Interfaces	15-123
15.6.1	Concepts	15-124
	Administrators 15-124	
	Policy Domains 15-124	
	Security Policies 15-125	
15.6.2	Domain Management	15-125
	Policy 15-126	
	Domain Manager 15-126	
	Construction Policy 15-127	
	Extensions to the Object Interface 15-127	
15.6.3	Security Policies Introduction	15-128
15.6.4	Access Policies	15-129
	Rights 15-129	
	AccessPolicy Interface 15-131	
	Specific Invocation Access Policies 15-132	
	DomainAccessPolicy Interface 15-132	
15.6.5	Audit Policies	15-138
	Audit Administration Interfaces 15-138	

15.6.6	Secure Invocation and Delegation Policies . . .	15-140
	Secure Invocation Policies	15-141
	Invocation Delegation Policy	15-144
15.6.7	Non-repudiation Policy Management	15-145
15.7	Implementor's Security Interfaces	15-147
15.7.1	Generic ORB Services and Interceptors	15-148
	Request-Level Interceptors	15-149
	Message-Level Interceptors	15-149
	Selecting Interceptors	15-150
	Interceptor Interfaces	15-150
15.7.2	Security Interceptors.	15-150
	Invocation Time Policies	15-152
	Secure Invocation Interceptor	15-152
	Access Control Interceptor	15-154
15.7.3	Implementation-Level Security Object Interfaces	15-155
	Vault	15-156
	Security Context Object	15-158
	Access Decision Object	15-161
	Audit Objects	15-162
	Principal Authentication	15-163
	Non-repudiation	15-163
15.7.4	Replaceable Security Services	15-163
	Replacing Authentication and Security Association Services	15-163
	Replacing Access Decision Policies	15-163
	Replacing Audit Services	15-164
	Replacing Non-repudiation Services	15-164
	Other Replaceability	15-164
	Linking to External Security Services	15-164
15.8	Security and Interoperability	15-165
15.8.1	Interoperability Model	15-166
	Security Information in the Object Reference	15-167
	Establishing a Security Association	15-168
	Protecting Messages	15-168
	Security Mechanisms for Secure Object Invocations	15-168
	Security Mechanism Types	15-169
	Interoperating between Underlying Security Services	15-170
	Interoperating between Security Policy Domains	15-170
	Secure Interoperability Bridges	15-171
15.8.2	Protocol Enhancements	15-171
15.8.3	CORBA Interoperable Object Reference with Security	15-171
	Security Components of the IOR	15-172
	Operational Semantics	15-175
15.8.4	Secure Inter-ORB Protocol (SECIOP)	15-177

	SECIOP Message Header 15-177	
	SECIOP 15-178	
	ContextId 15-178	
	ContextIdDefn 15-178	
	Message Definitions 15-179	
	SECIOP Protocol State Tables 15-182	
15.8.5	DCE-CIOP with Security	15-185
	Goals of Secure DCE-CIOP 15-185	
	Secure DCE-CIOP Overview 15-186	
	IOR Security Components for DCE-CIOP 15-186	
	DCE RPC Security Services 15-191	
	Secure DCE-CIOP Operational Semantics 15-192	
Appendix A	Consolidated OMG IDL	15-196
Appendix B	Summary of CORBA 2 Core Changes	15-217
Appendix C	Relationship to Other Services	15-232
Appendix D	Conformance Details	15-235
Appendix E	Guidelines for a Trustworthy System	15-245
Appendix F	Conformance Statement	15-267
Appendix G	Facilities Not in This Specification	15-273
Appendix H	Interoperability Guidelines	15-281
Appendix I	Glossary	15-286
16.	Trading Object Service Specification.	16-1
16.1	Overview	16-2
16.1.1	Diversity and Scalability	16-3
16.1.2	Linking Traders	16-3
16.1.3	Policy	16-3
16.1.4	Additional ObjectID	16-4
16.2	Concepts and Data Types	16-4
16.2.1	Exporter	16-4
16.2.2	Importer	16-4
16.2.3	Service Types	16-4
	Service Type Model 16-5	
16.2.4	Properties	16-7
16.2.5	Service Offers	16-7
	Modifiable Properties 16-8	
	Dynamic Properties 16-8	
16.2.6	Offer Identifier	16-9
16.2.7	Offer Selection	16-9
	Standard Constraint Language 16-9	
	Preferences 16-10	
	Links 16-11	
	Policies 16-12	
	Trader Policies 16-16	
	Link Follow Behavior 16-16	

	Importer Policies 16-17	
	Exporter Policies 16-18	
	Link Creation Policies 16-18	
16.2.8	Interworking Mechanisms	16-18
	Link Traversal Control 16-18	
	Federated Query Example 16-19	
	Proxy Offers 16-20	
16.2.9	Trader Attributes	16-21
16.3	Exceptions	16-23
16.3.1	For CosTrading module	16-23
	Exceptions used in more than one interface 16-23	
	Additional Exceptions for Lookup Interface 16-24	
	Additional Exceptions For Register Interface 16-25	
	Additional Exceptions for Link Interface 16-26	
	Additional Exceptions for Proxy Offer Interface 16-27	
16.3.2	For CosTradingDynamic module	16-27
16.3.3	For CosTradingRepos module	16-27
16.4	Abstract Interfaces	16-28
16.4.1	TraderComponents	16-28
16.4.2	SupportAttributes	16-29
16.4.3	ImportAttributes	16-29
16.4.4	LinkAttributes	16-30
16.5	Functional Interfaces	16-30
16.5.1	Lookup	16-30
	Query Operation 16-31	
16.5.2	Offer Iterator	16-35
	Signature 16-35	
	Function 16-36	
16.5.3	Register	16-36
	Export Operation 16-39	
	Withdraw Operation 16-41	
	Describe Operation 16-41	
	Modify Operation 16-42	
	Withdraw Using Constraint Operation 16-44	
	Resolve Operation 16-45	
16.5.4	Offer Id Iterator	16-45
	Signature 16-45	
	Function 16-46	
16.5.5	Admin.	16-46
	Attributes and Set Operations 16-48	
	List Offers Operation 16-48	
	List Proxies Operation 16-48	
16.5.6	Link	16-49
	Add_Link Operation 16-51	
	Remove Link Operation 16-52	
	Describe Link Operation 16-52	
	List Links Operation 16-53	

	Modify Link Operation	16-53	
16.5.7	Proxy	16-54	16-54
	Export Proxy Operation	16-55	
	Withdraw Proxy Operation	16-58	
	Describe Proxy Operation	16-58	
16.6	Service Type Repository	16-59	16-59
	Add Type Operation	16-62	
	Remove Type Operation	16-64	
	List Types Operation	16-64	
	Describe Type Operation	16-65	
	Fully Describe Type Operation	16-65	
	Mask Type Operation	16-66	
	Unmask Type Operation	16-66	
16.7	Dynamic Property Evaluation interface	16-67	16-67
16.8	Conformance Criteria	16-68	16-68
16.8.1	Conformance Requirements for Trading Interfaces as Server	16-69	16-69
	Lookup Interface	16-69	
	Register Interface	16-69	
	Admin Interface	16-70	
	Link Interface	16-70	
	Proxy Interface	16-70	
16.8.2	Conformance Requirements for Implementation Conformance Classes	16-71	16-71
	Query Trader	16-71	
	Simple Trader	16-72	
	Stand-alone Trader	16-72	
	Linked Trader	16-72	
	Proxy Trader	16-73	
	Full-service Trader	16-73	
Appendix A	CORBA OMG IDL based Specification of the Trading Function	16-74	16-74
Appendix B	OMG Constraint Language BNF	16-93	16-93
Appendix C	OMG Constraint Recipe Language	16-99	16-99
17.	Object Collection Specification		17-1
17.1	Overview	17-2	17-2
17.2	Service Structure	17-2	17-2
17.2.1	Combined Property Collections	17-3	17-3
	Restricted Access Collections	17-4	
	Collection Factories	17-5	
17.2.2	Iterators	17-5	17-5
17.2.3	Function Interfaces	17-7	17-7
	Collectible Elements and Type Safety	17-7	
	Collectible Elements and the Operations Interface	17-7	
	Collectible Elements of Key Collections	17-8	
17.2.4	List of Interfaces Defined	17-8	17-8

17.3	Combined Collections	17-10
17.3.1	Combined Collections Usage Samples	17-10
	Bag, SortedBag 17-10	
	EqualitySequence 17-11	
	Heap 17-11	
	KeyBag, KeySortedBag 17-11	
	KeySet, KeySortedSet 17-12	
	Map, SortedMap 17-12	
	Relation, SortedRelation 17-13	
	Set, SortedSet 17-13	
	Sequence 17-13	
17.4	Restricted Access Collections	17-14
17.4.1	Restricted Access Collections Usage Samples	17-14
	Deque 17-14	
	PriorityQueue 17-14	
	Queue 17-15	
	Stack 17-15	
17.5	The CosCollection Module	17-15
17.5.1	Interface Hierarchies	17-15
	Collection Interface Hierarchies 17-15	
	Iterator Hierarchy 17-18	
17.5.2	Exceptions and Type Definitions	17-19
17.5.3	Abstract Collection Interfaces	17-21
	The Collection Interface 17-21	
	The OrderedCollection Interface 17-28	
	The SequentialCollection Interface 17-31	
	The SortedCollection Interface 17-37	
	The EqualityCollection Interface 17-37	
	The KeyCollection Interface 17-42	
	The EqualityKeyCollection Interface 17-50	
	The KeySortedCollection Interface 17-51	
	The EqualitySortedCollection Interface 17-53	
	The EqualityKeySortedCollection Interface 17-55	
	The EqualitySequentialCollection Interface 17-55	
17.5.4	Concrete Collections Interfaces	17-57
	The KeySet Interface 17-57	
	The KeyBag Interface 17-57	
	The Map Interface 17-57	
	The Relation Interface 17-61	
	The Set Interface 17-61	
	The Bag Interface 17-62	
	The KeySortedSet Interface 17-62	
	The KeySortedBag Interface 17-63	
	The SortedMap Interface 17-63	
	The SortedRelation Interface 17-63	
	The SortedSet Interface 17-63	
	The SortedBag Interface 17-64	
	The Sequence Interface 17-64	
	The EqualitySequence Interface 17-64	
	The Heap Interface 17-64	
17.5.5	Restricted Access Collection Interfaces	17-65

17.5.6	Abstract RestrictedAccessCollection Interface	17-65
	The RestrictedAccessCollection Interface	17-65
17.5.7	Concrete Restricted Access Collection	
	Interfaces	17-66
	The Queue Interface	17-66
	The Dequeue Interface	17-67
	The Stack Interface	17-67
	The PriorityQueue Interface	17-69
17.5.8	Collection Factory Interfaces	17-70
	The CollectionFactory and CollectionFactories	
	Interfaces	17-71
	The RACollectionFactory and	
	RACollectionFactories Interfaces	17-74
	The KeySetFactory Interface	17-75
	The KeyBagFactory Interface	17-75
	The MapFactory Interface	17-76
	The RelationFactory Interface	17-76
	The SetFactory Interface	17-77
	The BagFactory Interface	17-77
	The KeySortedSetFactory Interface	17-78
	The KeySortedBagFactory Interface	17-78
	The SortedMapFactory Interface	17-79
	The SortedRelationFactory Interface	17-79
	The SortedSetFactory Interface	17-80
	The SortedBagFactory Interface	17-80
	The SequenceFactory Interface	17-81
	The EqualitySequence Factory Interface	17-81
	The HeapFactory Interface	17-82
	The QueueFactory Interface	17-82
	The StackFactory Interface	17-83
	The DequeFactory Interface	17-83
	The PriorityQueueFactory Interface	17-83
17.5.9	Iterator Interfaces	17-84
	Iterators as pointer abstraction	17-84
	Iterators and support for generic	
	programming	17-84
	Iterators and performance	17-85
	The Managed Iterator Model	17-85
	The Iterator Interface	17-86
	The OrderedIterator Interface	17-97
	The SequentialIterator Interface	17-106
	The KeyIterator Interface	17-108
	The EqualityIterator Interface	17-110
	The EqualityKeyIterator Interface	17-111
	The SortedIterator Interface	17-112
	The KeySortedIterator Interface	17-112
	The EqualitySortedIterator Interface	17-114
	The EqualityKeySortedIterator Interface	17-116
	The EqualitySequentialIterator Interface	17-117
17.5.10	Function Interfaces	17-118
	The Operations Interface	17-118
	The Command and Comparator Interface	17-122

	Identification and Justification of Differences	17-124
	CosQueryCollection Module Detailed Comparison	17-126
	Containers	17-133
	Algorithms	17-134
	Iterators	17-134
	Consideration on choice	17-135
Appendix A	OMG Object Query Service	17-124
Appendix B	Relationship to Other Relevant Standards	17-133
Appendix C	References	17-138
Index		Index-1

List of Figures

<i>Figure 2-1</i>	An event channel as a collection of objects conspiring to manage multiple simultaneous consumer clients.	2-3
<i>Figure 3-1</i>	A Naming Graph	3-2
<i>Figure 3-2</i>	The CosNaming Module	3-6
<i>Figure 3-3</i>	The Names Library Interface in PIDL	3-14
<i>Figure 4-1</i>	Push-style Communication Between a Supplier and a Consumer	4-7
<i>Figure 4-2</i>	Pull-style Communication Between a Supplier and a Consumer	4-7
<i>Figure 4-3</i>	The OMG IDL Module CosEventComm	4-8
<i>Figure 4-4</i>	Push-style Communication Between a Supplier and an Event Channel, and a Consumer and an Event Channel	4-11
<i>Figure 4-5</i>	Pull-style communication between a supplier and an event channel and a consumer and the event channel	4-11
<i>Figure 4-6</i>	Push-style Communication Between a Supplier and an Event Channel, and Pull-style Communication Between a Consumer and an Event Channel	4-12
<i>Figure 4-7</i>	An Event Channel with Multiple Suppliers and Multiple Consumers	4-12
<i>Figure 4-8</i>	A newly created event channel. The channel has no suppliers or consumers	4-13
<i>Figure 4-9</i>	State diagram of a proxy	4-14

<i>Figure 4-10</i>	The CosEventChannelAdmin Module	4-16
<i>Figure 4-11</i>	Typed Push-style Communication Between a Supplier and a Consumer	4-20
<i>Figure 4-12</i>	Typed Pull-style Communication Between a Supplier and a Consumer	4-21
<i>Figure 4-13</i>	The IDL Module CosTypedEventComm	4-22
<i>Figure 4-14</i>	The CosTypedEventChannelAdmin Module	4-25
<i>Figure 5-1</i>	Roles in the Persistent Object Service	5-1
<i>Figure 5-2</i>	Major Components of the POS and their Interactions	5-8
<i>Figure 5-3</i>	The CosPersistencePID Module	5-9
<i>Figure 5-4</i>	TheCosPersistencePO Module	5-12
<i>Figure 5-5</i>	The CosPersistencePOM Module	5-15
<i>Figure 5-6</i>	Example to illustrate POMFunctions	5-18
<i>Figure 5-7</i>	The CosPersistencePDS Module	5-20
<i>Figure 5-8</i>	Direct Access Protocol Interfaces	5-21
<i>Figure 5-9</i>	The CosPersistencePDS_DA Module	5-22
<i>Figure 5-10</i>	Structure of a DDO	5-31
<i>Figure 5-11</i>	The CosPersistenceDDO Module	5-32
<i>Figure 5-12</i>	The CosPersistenceDS_CLI Module	5-35
<i>Figure 6-1</i>	Life Cycle service defines how a client can create an object “over there”.	6-1
<i>Figure 6-2</i>	Life Cycle Service defines how a client can move or copy an object over there.	6-2
<i>Figure 6-3</i>	The object life cycle problem for graphs of objects is to determine the boundaries of a graph of objects and operate on that graph. In the above example, a document contains a graphic and a logo, refers to a dictionary and is contained in a folder.	6-3
<i>Figure 6-4</i>	To create an object “over there” a client must possess an object reference to a factory over there. The client simply issues a request on the factory.	6-4
<i>Figure 6-5</i>	An example of a document factory interface. This interface is defined for clients as a part of application development.	6-5
<i>Figure 6-6</i>	To delete an object, a client must possess an object reference supporting the <i>LifeCycleObject</i> interface and issues a <i>remove</i> request on the object.	6-6

<i>Figure 6-7</i>	Life cycle services define how a client can move or copy an object from here to there. 6-7
<i>Figure 6-8</i>	The <i>FactoryFinder</i> interface can be “mixed in” with interfaces of more powerful finding services. 6-8
<i>Figure 6-9</i>	The CosLifeCycle Module 6-10
<i>Figure 6-10</i>	The Life Cycle service provides a generic creation capability. Ultimately, implementation specific creation code is invoked by the creation service. The implementation specific code also supports the <i>GenericFactory</i> interface. 6-15
<i>Figure 6-11</i>	Factories assemble resources for the execution of an object. A minimal implementation achieves this with a single factory implementation. 6-19
<i>Figure 6-12</i>	In an administered environment, factory implementations can delegate the creation problem to a generic factory. The generic factory can apply resource allocation policies. Ultimately the creation service communicates with implementation specific code that assembles resources for the object 6-20
<i>Figure 6-13</i>	The copy and move operations are passed a <i>FactoryFinder</i> to represent "there." The implementation of the target uses the <i>FactoryFinder</i> to find a factory object for creation over there. The protocol between the object and the factory is private. They can communicate and transfer state according to any implementation-defined protocol. 6-21
<i>Figure 8-1</i>	Externalization control flow when streamable object is not in a graph of related objects 8-4
<i>Figure 8-2</i>	Externalization control flow when streamable object is a node in a graph of related objects 8-5
<i>Figure 8-3</i>	Internalization control flow when object is not in a graph of related objects 8-6
<i>Figure 8-4</i>	Internalization control flow when object is in a graph of related objects 8-7
<i>Figure 8-5</i>	Object Externalization Service Booch Class (=Interface) Diagram 8-9
<i>Figure 8-6</i>	Client Functional Interfaces support client’s model of externalization 8-10

<i>Figure 8-7</i>	Service Construction Interfaces support service implementation's model of externalization	8-10
<i>Figure 8-8</i>	Compound Externalization Interfaces support service implementation's model of graph externalization	8-11
<i>Figure 8-9</i>	The CosStream module	8-15
<i>Figure 8-10</i>	The CosCompoundExternalization Module	8-20
<i>Figure 8-11</i>	Internalizing a node returns the new object and the corresponding roles.	8-22
<i>Figure 8-12</i>	The CosExternalizationContainment module	8-26
<i>Figure 8-13</i>	The CosExternalizationReference module	8-28
<i>Figure 9-1</i>	Base relationships	9-7
<i>Figure 9-2</i>	Navigation functionality of base relationships	9-8
<i>Figure 9-3</i>	An example graph of related objects	9-9
<i>Figure 9-4</i>	Relationship interface hierarchy	9-10
<i>Figure 9-5</i>	Role interface hierarchy	9-10
<i>Figure 9-6</i>	Simple relationship type: documents reference books . . .	9-14
<i>Figure 9-7</i>	Simple relationship instance: my document references the book "War and Peace"	9-14
<i>Figure 9-8</i>	A ternary check-out relationship type between books, libraries and persons.	9-15
<i>Figure 9-9</i>	An unsatisfactory representation of the ternary check-out relationship using binary relationships	9-16
<i>Figure 9-10</i>	Another unsatisfactory representation	9-16
<i>Figure 9-11</i>	Creating a role for an object	9-17
<i>Figure 9-12</i>	A fully established binary relationship	9-17
<i>Figure 9-13</i>	The CosObjectIdentity Module	9-19
<i>Figure 9-14</i>	The CosRelationships Module	9-21
<i>Figure 9-15</i>	Two binary one-to-many containment relationships.	9-23
<i>Figure 9-16</i>	An example graph of related objects	9-34
<i>Figure 9-17</i>	A traversal of a graph for compound copy operation . . .	9-37
<i>Figure 9-18</i>	How deep, shallow and none propagation values affect nodes, roles and relationships.	9-38
<i>Figure 9-19</i>	The CosGraphs Module	9-39
<i>Figure 9-20</i>	The CosContainment Module	9-48

<i>Figure 9-21</i>	The CosReference Module	9-50
<i>Figure 10-1</i>	This figure illustrates the major components and interfaces of the Transaction Service	10-12
<i>Figure 10-2</i>	X/Open client	10-41
<i>Figure 10-3</i>	X/Open server	10-42
<i>Figure 10-4</i>	Example	10-42
<i>Figure 10-5</i>	Model interoperability example	10-64
<i>Figure 11-1</i>	Query Evaluators: Nesting and Federation	11-3
<i>Figure 11-2</i>	Queryable Collections	11-5
<i>Figure 11-3</i>	SQL Query = OQL	11-8
<i>Figure 11-4</i>	Collection interface structure	11-10
<i>Figure 11-5</i>	Query Framework interface hierarchy/structure	11-11
<i>Figure 11-6</i>	CosQueryCollection Module	11-14
<i>Figure 11-7</i>	Query Evaluator and Queryable Collection	11-20
<i>Figure 11-8</i>	Query Manager and Query Object	11-21
<i>Figure 11-9</i>	QueryLanguageType Interface Hierarchy	11-24
<i>Figure 12-1</i>	Licensing Service Relationships	12-7
<i>Figure 12-2</i>	Licensing Service Instance Diagram	12-14
<i>Figure 12-3</i>	Licensing Event Trace Diagram	12-16
<i>Figure 12-4</i>	CosLicensingManager Module	12-17
<i>Figure 13-1</i>	Data types	13-5
<i>Figure 13-2</i>	PropertySet interface exceptions	13-7
<i>Figure 13-3</i>	Operations used to define new properties or set new values	13-9
<i>Figure 13-4</i>	Operations used to retrieve property names and values	13-11
<i>Figure 13-5</i>	Operations used to delete properties	13-12
<i>Figure 13-6</i>	is_property_defined operation	13-14
<i>Figure 13-7</i>	Operations used to retrieve information related to constraints.	13-15
<i>Figure 13-8</i>	Operations used to define new properties or values.	13-16
<i>Figure 13-9</i>	Operations used to get and set property mode	13-18
<i>Figure 13-10</i>	reset operation	13-19
<i>Figure 13-11</i>	next_one and next_n operations (properties)	13-20

<i>Figure 13-12</i>	destroy operation	13-20
<i>Figure 13-13</i>	reset operation	13-20
<i>Figure 13-14</i>	next_one, next_n operations (PropertyNames)	13-21
<i>Figure 13-15</i>	destroy operation	13-21
<i>Figure 13-16</i>	PropetySetFactory interface	13-21
<i>Figure 13-17</i>	PropertySetDefFactory interface	13-22
<i>Figure 14-1</i>	General Object Model for Service	14-3
<i>Figure 14-2</i>	Object Model for Time Service	14-5
<i>Figure 14-3</i>	Illustration of Interval Overlap	14-8
<i>Figure 14-4</i>	Object Model of Timer Event Service	14-13
<i>Figure 14-5</i>	Time Service and Proxies	14-19
<i>Figure 15-1</i>	A Security model for object systems	15-13
<i>Figure 15-2</i>	Credential containing security attributes	15-15
<i>Figure 15-3</i>	Target Object via ORB	15-15
<i>Figure 15-4</i>	Message protection	15-18
<i>Figure 15-5</i>	Access control model	15-19
<i>Figure 15-6</i>	Authorization model	15-21
<i>Figure 15-7</i>	Auditing model	15-24
<i>Figure 15-8</i>	Delegation model	15-25
<i>Figure 15-9</i>	No delegation	15-28
<i>Figure 15-10</i>	Simple delegation	15-28
<i>Figure 15-11</i>	Composite delegation	15-29
<i>Figure 15-12</i>	Combined privileges delegation	15-29
<i>Figure 15-13</i>	Traced delegation	15-29
<i>Figure 15-14</i>	Proof of receipt	15-32
<i>Figure 15-15</i>	Non-repudiation services	15-32
<i>Figure 15-16</i>	Security policy domains	15-34
<i>Figure 15-17</i>	Policy domain hierarchies	15-35
<i>Figure 15-18</i>	Federated policy domains	15-35
<i>Figure 15-19</i>	System- and application-enforced policies	15-36
<i>Figure 15-20</i>	Overlapping policy domains	15-36
<i>Figure 15-21</i>	Framework of domains	15-39

<i>Figure 15-22</i>	Structural model	15-46
<i>Figure 15-23</i>	ORB services	15-47
<i>Figure 15-24</i>	Object reference	15-49
<i>Figure 15-25</i>	Domain objects	15-50
<i>Figure 15-26</i>	Controlled relationship	15-53
<i>Figure 15-27</i>	Object encapsulation	15-53
<i>Figure 15-28</i>	Authentication	15-55
<i>Figure 15-29</i>	Multiple credentials	15-56
<i>Figure 15-30</i>	Changing security attributes	15-57
<i>Figure 15-31</i>	Making a secure invocation	15-59
<i>Figure 15-32</i>	Target object security	15-60
<i>Figure 15-33</i>	Security-unaware intermediate object	15-61
<i>Figure 15-34</i>	Security-aware intermediate object	15-62
<i>Figure 15-35</i>	access_allowed application	15-63
<i>Figure 15-36</i>	get_policy application	15-64
<i>Figure 15-37</i>	audit_write application	15-65
<i>Figure 15-38</i>	Audit decision object	15-65
<i>Figure 15-39</i>	set_NR_features operation	15-66
<i>Figure 15-40</i>	generate_token operation	15-67
<i>Figure 15-41</i>	Non-repudiation service	15-69
<i>Figure 15-42</i>	verify_evidence operation	15-70
<i>Figure 15-43</i>	Proof of origin message	15-70
<i>Figure 15-44</i>	Managing security policies	15-74
<i>Figure 15-45</i>	Securing invocations	15-76
<i>Figure 15-46</i>	get_policy operation	15-77
<i>Figure 15-47</i>	ORB Security Services	15-78
<i>Figure 15-48</i>	Access decision object	15-79
<i>Figure 15-49</i>	Target objects sharing security names	15-81
<i>Figure 15-50</i>	Object created by application or factory	15-82
<i>Figure 15-51</i>	Relationship between main objects	15-83
<i>Figure 15-52</i>	Interceptors Called During Invocation Path	15-149

<i>Figure 15-53</i>	Security Functionality Implemented by Security Service Objects	15-151
<i>Figure 15-54</i>	Secure Interoperability Model	15-167
<i>Figure 15-55</i>	New CORBA 2.0 Protocol	15-177
<i>Figure 16-1</i>	Interactions between a trader and its clients	16-1
<i>Figure 16-2</i>	Property Strength	16-5
<i>Figure 16-3</i>	Pipeline View of Trader Query Steps and Cardinality Constraint Application	16-15
<i>Figure 16-4</i>	Flow of a query through a trader graph	16-19
<i>Figure 17-1</i>	Collections Interfaces Hierarchy	17-17
<i>Figure 17-2</i>	Restricted Access Collections Interface Hierarchy	17-17
<i>Figure 17-3</i>	Iterator Interface Hierarchy	17-18
<i>Figure 17-4</i>	Inheritance Relationships	17-126

List of Tables

<i>Table 3-5</i>	Exceptions Raised by Binding Operations	3-9
<i>Table 3-6</i>	Exceptions Raised by Resolve Operation	3-10
<i>Table 3-7</i>	Exceptions Raised by Unbind Operation	3-10
<i>Table 3-8</i>	Exceptions Raised by Creating New Contexts	3-11
<i>Table 6-1</i>	Suggested Conventions for Factory Finder Keys	6-14
<i>Table 6-2</i>	Suggested Conventions for Generic Factory Keys	6-16
<i>Table 6-3</i>	Suggested Criteria	6-17
<i>Table 8-1</i>	Tag Byte Values and Data Formats for Basic CORBA Data Types	8-30
<i>Table 9-1</i>	Interfaces Defined in the CosObjectIdentity Module	9-11
<i>Table 9-2</i>	Interfaces Defined in the CosRelationships Module	9-11
<i>Table 9-3</i>	Interfaces Defined in the CosGraphs Module	9-12
<i>Table 9-4</i>	Interfaces Defined in the CosContainment Module	9-12
<i>Table 9-5</i>	Interfaces Defined in the CosReference Module	9-13
<i>Table 10-1</i>	Use of Transaction Service Functionality	10-32

<i>Table 11-1</i>	Interfaces Defined in the CosQueryCollection Module	11-12
<i>Table 12-1</i>	Exceptions Raised by Licensing Service Operations.	12-19
<i>Table 13-1</i>	Property Service Interfaces.	13-3
<i>Table 13-2</i>	Exceptions Raised by Define Operations	13-10
<i>Table 13-3</i>	Exceptions Raised by List and Get Properties Operations	13-12
<i>Table 13-4</i>	Exceptions Raised by delete_properties Operations.	13-13
<i>Table 13-5</i>	Exceptions Raised by define Operations	13-17
<i>Table 13-6</i>	Exceptions Raised by Get and Set Mode Operations.	13-19
<i>Table 15-1</i>	DomainAccessPolicy	15-133
<i>Table 15-2</i>	User Privilege Attributes (Not Defined by This Specification)	15-133
<i>Table 15-3</i>	DomainAccessPolicy (with Privilege Attributes).	15-134
<i>Table 15-4</i>	DomainAccessPolicy (with Delegate Entry)	15-134
<i>Table 15-5</i>	Interface Instances	15-135
<i>Table 15-6</i>	DomainAccessPolicy (with Required Rights Mapping).	15-135
<i>Table 15-7</i>	RequiredRights for Interfaces c1, c2 and c3	15-136
<i>Table 15-8</i>	Standard Audit Policy.	15-137
<i>Table 15-9</i>	Option Definitions	15-174
<i>Table 15-10</i>	IOR Example	15-175
<i>Table 15-11</i>	Client State Table	15-182
<i>Table 15-12</i>	Target State Table	15-184
<i>Table 15-13</i>	Association Option Mapping to DCE Security.	15-189
<i>Table 15-14</i>	Relationship between Identifiers	15-191
<i>Table 16-1</i>	Preferences	16-10
<i>Table 16-2</i>	Scoping Policies	16-13

<i>Table 16-3</i>	Capability Supported Policies	16-15
<i>Table 16-4</i>	Trader Attributes.	16-21
<i>Table 16-5</i>	Primary/Secondary Policy Parameters	16-56
<i>Table 17-1</i>	Interfaces derived from combinations of collection properties	17-4
<i>Table 17-2</i>	Iterators and Collections	17-19
<i>Table 17-3</i>	Collection interfaces and the iterator interfaces supported	17-27
<i>Table 17-4</i>	Implementation Category Examples	17-72
<i>Table 17-5</i>	Required element and key-type specific user-defined information for KeySetFactory. []- implied by key_compare.	17-75
<i>Table 17-6</i>	Required element and key-type specific user-defined information for KeyBagFactory. []- implied by key_compare.	17-76
<i>Table 17-7</i>	Required element and key-type specific user-defined information for MapFactory. []- implied by key_compare.	17-76
<i>Table 17-8</i>	Required element and key-type specific user-defined information for RelationFactory. []- implied by key_compare.	17-77
<i>Table 17-9</i>	Required element and key-type specific user-defined information for SetFactory. []- implied by compare.	17-77
<i>Table 17-10</i>	Required element and key-type specific user-defined information for BagFactory. []- implied by compare.	17-78
<i>Table 17-11</i>	Required element and key-type specific user-defined information for KeySortedSetFactory. []- implied by key_compare.	17-78
<i>Table 17-12</i>	Required element and key-type specific user-defined information for KeySortedBagFactory. []- implied by key_compare.	17-79
<i>Table 17-13</i>	Required element and key-type specific user-defined information for SortedMapFactory. []- implied by key_compare.	17-79
<i>Table 17-14</i>	Required element and key-type specific user-defined information for SortedRelationFactory. []- implied by key_compare.	17-80

<i>Table 17-15</i>	Required element and key-type specific user-defined information for SortedSetFactory. []- implied by compare.	17-80
<i>Table 17-16</i>	Required element and key-type specific user-defined information for SortedBagFactory. []- implied by compare.	17-81
<i>Table 17-17</i>	Required element and key-type specific user-defined information for EqualitySequenceFactory.	17-82
<i>Table 17-18</i>	Required element and key-type specific user-defined information for PriorityQueueFactory. [] - implied by key_compare.	17-83

Preface

0.1 About This Document

Under the terms of the collaboration between OMG and X/Open Co Ltd, this document is a candidate for endorsement by X/Open, initially as a Preliminary Specification and later as a full CAE Specification. The collaboration between OMG and X/Open Co Ltd ensures joint review and cohesive support for emerging object-based specifications.

X/Open Preliminary Specifications undergo close scrutiny through a review process at X/Open before publication and are inherently stable specifications. Upgrade to full CAE Specification, after a reasonable interval, takes place following further review by X/Open. This further review considers the implementation experience of members and the full implications of conformance and branding.

0.1.1 Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 750 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

0.1.2 X/Open

X/Open is an independent, worldwide, open systems organization supported by most of the world's largest information system suppliers, user organizations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

0.2 Intended Audience

The specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for object services; the benefits of compliance are outlined in the following section, "Need for Object Services."

0.3 Need for Object Services

To understand how Object Services benefit all computer vendors and users, it is helpful to understand their context within OMG's vision of object management. The key to understanding the structure of the architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in *CORBA: Common Object Request Broker Architecture and Specification*.
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. For example, the Life Cycle Service defines conventions for creating, deleting, copying, and moving objects; it does not dictate how the objects are implemented in an application. Specifications for Object Services are contained in this manual.
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility. Information about Common Facilities is contained in *CORBAfacilities: Common Facilities Architecture*.
- **Application Objects**, which are products of a single vendor or in-house development group which controls their interfaces. Application Objects correspond to the traditional notion of applications, so they are not standardized by OMG. Instead, Application Objects constitute the uppermost layer of the Reference Model.

The Object Request Broker, then, is the core of the Reference Model. Nevertheless, an Object Request Broker alone cannot enable interoperability at the application semantic level. An ORB is like a telephone exchange: it provides the basic mechanism for making and receiving calls but does not ensure meaningful communication between

subscribers. Meaningful, productive communication depends on additional interfaces, protocols, and policies that are agreed upon outside the telephone system, such as telephones, modems and directory services. This is equivalent to the role of Object Services.

0.3.1 What Is an Object Service Specification?

A specification of an Object Service usually consists of a set of interfaces and a description of the service's behavior. The syntax used to specify the interfaces is the OMG Interface Definition Language (OMG IDL). The semantics that specify a services's behavior are, in general, expressed in terms of the OMG Object Model. The OMG Object Model is based on objects, operations, types, and subtyping. It provides a standard, commonly understood set of terms with which to describe a service's behavior.

(For detailed information about the OMG Reference Model and the OMG Object Model, refer to the *Object Management Architecture Guide*).

0.4 Associated Documents

The CORBA documentation set includes the following books:

- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBAservices: Common Object Services Specification* contains specifications for the object services.
- *CORBAfacilities: Common Facilities Architecture* contains information about the design of Common Facilities; it provides the framework for Common Facility specifications.
- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote.

To obtain books in the documentation set, or other OMG publications, refer to the enclosed subscription card or contact the Object Management Group, Inc. at:

OMG Headquarters
492 Old Connecticut Path
Framingham, MA 01701
USA
Tel: +1-508-820-4300
pubs@omg.org
<http://www.omg.org>

0.5 *Structure of this Manual*

In addition to this preface, *CORBAservices: Common Object Services* contains the following chapters:

Overview provides an introduction to the CORBA object services, including a summary of features for each service.

General Design Principles provides information about the principles that were used in designing each service; explains the dependencies among services; and explains how Object Services relate to each other, CORBA, and industry standards in general.

Chapters 3 through 16 each contain a specification for the following Object Services:

- Naming
- Event
- Persistent Object
- Life Cycle
- Concurrency Control
- Externalization
- Relationship
- Transaction
- Query
- Licensing
- Property
- Time
- Security
- Trading
- Collections

0.6 *Acknowledgements*

The following companies submitted parts of the specifications that were approved by the Object Management Group to become *CORBAservices*:

AT&T/Lucent Technologies

AT&T/NCR

BNR Europe Limited

Cooperative Research Centre for Distributed Systems Technology (DTSC Pty Ltd.)

Digital Equipment Corporation

Expersoft Corporation

Gradient Technologies, Inc.

Groupe Bull

Hewlett-Packard Company

HyperDesk Corporation

ICL PLC

Ing. C. Olivetti & C.Sp

International Business Machines Corporation
International Computers Limited
Iona Technologies Ltd.
Itasca Systems, Inc.
Nortel Limited
Novell, Inc.
O2 Technologies, SA
Object Design, Inc.
Objectivity, Inc.
Odyssey Research Associates, Inc.
Ontos, Inc.
Oracle Corporation
Persistence Software, Inc.
Servio Corporation
Siemens Nixdorf Informationssysteme AG
Sun Microsystems, Inc.
SunSoft, Inc.
Sybase, Inc.
Taligent, Inc.
Tandem Computers, Inc.
Teknekron Software Systems, Inc.
Tivoli Systems, Inc.
Transarc Corporation
Versant Object Technology Corporation

1.1 Summary of Key Features

1.1.1 Naming Service

- The Naming Service provides the ability to bind a name to an object relative to a *naming context*. A naming context is an object that contains a set of name bindings in which each name is unique. To resolve a name is to determine the object associated with the name in a given context.
- Through the use of a very general model and dealing with names in their structural form, naming service implementations can be application specific or be based on a variety of naming systems currently available on system platforms.
- Graphs of naming contexts can be supported in a distributed, federated fashion. The scalable design allows the distributed, heterogeneous implementation and administration of names and name contexts.
- Because name component attribute values are not assigned or interpreted by the naming service, higher levels of software are not constrained in terms of policies about the use and management of attribute values.
- Through the use of a “names library,” name manipulation is simplified and names can be made representation-independent thus allowing their representation to evolve without requiring client changes.
- Application localization is facilitated by name syntax-independence and the provision of a name “kind” attribute.

1.1.2 Event Service

- The Event Service provides basic capabilities that can be configured together in a very flexible and powerful manner. Asynchronous events (decoupled event suppliers and consumers), event “fan-in,” notification “fan-out,” and (through appropriate event channel implementations) reliable event delivery are supported.
- The Event Service design is scalable and is suitable for distributed environments. There is no requirement for a centralized server or dependency on any global service.
- The Event Service interfaces allow implementations that provide different qualities of service to satisfy different application requirements. In addition, the event service does not impose higher level policies (e.g., specific event types) allowing great flexibility on how it is used in a given application environment.
- Both push and pull event delivery models are supported: that is, consumers can either request events or be notified of events, whichever is needed to satisfy application requirements. There can be multiple consumers and multiple suppliers events.
- Suppliers can generate events without knowing the identities of the consumers. Conversely, consumers can receive events without knowing the identities of the suppliers.
- The event channel interface can be subtyped to support extended capabilities. The event consumer-supplier interfaces are symmetric, allowing the chaining of event channels (for example, to support various event filtering models). Event channels can be chained by third-parties.
- Typed event channels extend basic event channels to support typed interaction.
- Because event suppliers, consumers and channels are objects, advantage can be taken of performance optimizations provided by ORB implementations for local and remote objects. No extension is required to CORBA.

1.1.3 Life Cycle Service

- The Life Cycle Service defines conventions for creating, deleting, copying and moving objects. Because CORBA-based environments support distributed objects, life cycle services define services and conventions that allow clients to perform life cycle operations on objects in different locations.
- The client’s model of creation is defined in terms of factory objects. A factory is an object that creates another object. Factories are *not* special objects. As with any object, factories have well-defined OMG IDL interfaces and implementations in some programming language.
- The Life Cycle Service defines an interface for a generic factory. This allows for the definition of standard creation services.
- The Life Cycle Service defines a *LifeCycleObject* interface. This interface defines remove, copy and move operations.

- The Life Cycle Service has been extended to support compound life cycle operations on graphs of related objects. Compound objects (graphs of objects) rely on the Relationship Service for the definition of object graphs.

1.1.4 Persistent Object Service

- The Persistent Object Service (POS) provides a set of common interfaces to the mechanisms used for retaining and managing the persistent state of objects.
- The object ultimately has the responsibility of managing its state, but can use or delegate to the Persistent Object Service for the actual work. A major feature of the Persistent Object Service is its openness. In this case, that means that there can be a variety of different clients and implementations of the Persistent Object Service, and they can work together. This is particularly important for storage, where mechanisms useful for documents may not be appropriate for employee databases, or the mechanisms appropriate for mobile computers do not apply to mainframes.

1.1.5 Transaction Service

- The Transaction Service supports multiple transaction models, including the flat (mandatory in the specification) and nested (optional) models.
- The Object Transaction Service supports interoperability between different programming models. For instance, some users want to add object implementations to existing procedural applications and to augment object implementations with code that uses the procedural paradigm. To do so in a transaction environment requires the object and procedural code to share a single transaction.
- Network interoperability is also supported, since users need communication between different systems, including the ability to have one transaction service interoperate with a cooperating transaction service using different ORBs.
- The Transaction Service supports both implicit (system-managed transaction) propagation and explicit (application-managed) propagation. With implicit propagation, transactional behavior is not specified in the operation's signature. With explicit propagation, applications define their own mechanisms for sharing a common transaction.
- The Transaction Service can be implemented in a TP monitor environment, so it supports the ability to execute multiple transactions concurrently, and to execute clients, servers, and transaction services in separate processes.

1.1.6 Concurrency Control Service

- The Concurrency Control Service enables multiple clients to coordinate their access to shared resources. Coordinating access to a resource means that when multiple, concurrent clients access a single resource, any conflicting actions by the clients are reconciled so that the resource remains in a consistent state.

- Concurrent use of a resource is regulated with locks. Each lock is associated with a single resource and a single client. Coordination is achieved by preventing multiple clients from simultaneously possessing locks for the same resource if the client's activities might conflict. Hence, a client must obtain an appropriate lock before accessing a shared resource. The Concurrency Control Service defines several lock modes, which correspond to different categories of access. This variety of lock modes provides flexible conflict resolution. For example, providing different modes for reading and writing lets a resource support multiple concurrent clients on a read-only transaction. The Concurrency Control Service also defines Intention Locks that support locking at multiple levels of granularity.

1.1.7 Relationship Service

- The Relationship Service allows entities and relationships to be explicitly represented. Entities are represented as CORBA objects. The service defines two new kinds of objects: relationships and roles. A role represents a CORBA object in a relationship. The Relationship interface can be extended to add relationship-specific attributes and operations. In addition, relationships of arbitrary degree can be defined. Similarly, the *Role* interface can be extended to add role-specific attributes and operations.
- Type and cardinality constraints can be expressed and checked: exceptions are raised when the constraints are violated.
- The Life Cycle Service defines operations to copy, move, and remove graphs of related objects, while the Relationship Service allows graphs of related objects to be traversed without activating the related objects.
- Distributed implementations of the Relationship Service can have navigation performance and availability similar to CORBA object references: role objects can be located with their objects and need not depend on a centralized repository of relationship information. As such, navigating a relationship can be a local operation.
- The Relationship Service supports the compound life cycle component of the Life Cycle Service by defining object graphs.

1.1.8 Externalization Service

- The Externalization Service defines protocols and conventions for externalizing and internalizing objects. Externalizing an object is to record the object state in a stream of data (in memory, on a disk file, across the network, and so forth) and then be internalized into a new object in the same or a different process. The externalized object can exist for arbitrary amounts of time, be transported by means outside of the ORB, and be internalized in a different, disconnected ORB. For portability, clients can request that externalized data be stored in a file whose format is defined with the Externalization Service Specification.
- The Externalization Service is related to the Relationship Service and parallels the Life Cycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for facilities, directory services, and file services.

1.1.9 Query Service

- The purpose of the Query Service is to allow users and objects to invoke queries on collections of other objects. The queries are declarative statements with predicates and include the ability to specify values of attributes; to invoke arbitrary operations; and to invoke other Object Services.
- The Query Service allows indexing; maps well to the query mechanisms used in database systems and other systems that store and access large collections of objects; and is based on existing standards for query, including SQL-92, OQL-93, and OQL-93 Basic.
- The Query Service provides an architecture for a nested and federated service that can coordinate multiple, nested query evaluators.

1.1.10 Licensing Service

- The Licensing Service provides a mechanism for producers to control the use of their intellectual property. Producers can implement the Licensing Service according to their own needs, and the needs of their customers, because the Licensing Service does not impose its own business policies or practices.
- A license in the Licensing Service has three types of attributes that allow producers to apply controls flexibly: *time*; *value mapping*, and *consumer*. Time allows licenses to have start/duration and expiration dates. Value mapping allows producers to implement a licensing scheme according to units, allocation (through concurrent use licensing), or consumption (for example, metering or allowance of grace periods through “overflow licenses.”) Consumer attributes allow a license to be reserved or assigned for specific entities; for example, a license could be assigned to a particular machine. The Licensing Service allows producers to combine and derive from license attributes.
- The Licensing Service consists of a *LicenseServiceManager* interface and a *ProducerSpecificLicenseService* interface: these interfaces do not impose business policies upon implementors.

1.1.11 Property Service

- Provides the ability to dynamically associate named values with objects outside the static IDL-type system.
- Defines operations to create and manipulate sets of name-value pairs or name-value-mode tuples. The names are simple OMG IDL strings. The values are OMG IDL *anys*. The use of type *any* is significant in that it allows a property service implementation to deal with any value that can be represented in the OMG IDL-type system. The modes are similar to those defined in the *Interface Repository AttributeDef* interface.
- Designed to be a basic building block, yet robust enough to be applicable for a broad set of applications.

- Provides “batch” operations to deal with sets of properties as a whole. The use of “batch” operations is significant in that the systems and network management (SNMP, CMIP, ...) communities have proven such a need when dealing with “attribute” manipulation in a distributed environment.
- Provides exceptions such that *PropertySet* implementors may exercise control of (or apply constraints to) the names and types of properties associated with an object, similar in nature to the control one would have with CORBA attributes.
- Allows *PropertySet* implementors to restrict modification, addition and/or deletion of properties (readonly, fixed) similar in nature to the restrictions one would have with CORBA attributes.
- Provides client access and control of constraints and property modes.
- Does not rely on any other object services.

1.1.12 Time Service

- Enables the user to obtain current time together with an error estimate associated with it.
- Ascertains the order in which “events” occurred.
- Generates time-based events based on timers and alarms.
- Computes the interval between two events.
- Consists of two services, hence defines two service interfaces:
 - Time Service manages Universal Time Objects (UTOs) and Time Interval Objects (TIOs), and is represented by the *TimeService* interface.
 - Timer Event Service manages Timer Event Handler objects, and is represented by the *TimerEventService* interface.

1.1.13 Security Service

The security functionality defined by this specification comprises:

- **Identification** and **authentication** of principals (human users and objects which need to operate under their own rights) to verify they are who they claim to be.
- **Authorization** and **access control** - deciding whether a principal can access an object, normally using the identity and/or other privilege attributes of the principal (such as role, groups, security clearance) and the control attributes of the target object (stating which principals, or principals with which attributes) can access it.
- **Security auditing** to make users accountable for their security related actions. It is normally the human user who should be accountable. Auditing mechanisms should be able to identify the user correctly, even after a chain of calls through many objects.

- **Security of communication** between objects, which is often over insecure lower layer communications. This requires trust to be established between the client and target, which may require **authentication of clients to targets** and **authentication of targets to clients**. It also requires **integrity protection** and (optionally) **confidentiality protection** of messages in transit between objects.
- **Non-repudiation** provides irrefutable evidence of actions such as proof of origin of data to the recipient, or proof of receipt of data to the sender to protect against subsequent attempts to falsely deny the receiving or sending of the data.
- **Administration** of security information (for example, security policy) is also needed.

1.1.14 Object Trader Service

The Object Trader Service provides a matchmaking service for objects.

The Service Provider registers the availability of the service by invoking an export operation on the trader, passing as parameters information about the offered service. The export operation carries an object reference that can be used by a client to invoke operations on the advertised services, a description of the type of the offered service (i.e., the names of the operations to which it will respond, along with their parameter and result types), information on the distinguishing attributes of the offered service.

The offer space managed by traders may be partitioned to ease administration and navigation. This information is stored persistently by the Trader. Whenever a potential client wishes to obtain a reference to a service that does a particular job, it invokes an import operation, passing as parameters a description of the service required. Given this import request, the Trader checks appropriate offers for acceptability. To be acceptable, an offer must have a type that conforms to that requested and have properties consistent with the constraints specified by an imported.

Trading service in a single trading domain may be distributed over a number of trader objects. Traders in different domains may be federated. Federation enables systems in different domains to negotiate the sharing of services without losing control of their own policies and services. A domain can thus share information with other domains with which it has been federated, and it can now be searched for appropriate service offers.

1.1.15 Object Collections Service

Collections are groups of objects which, as a group, support some operations and exhibit specific behaviors that are related to the nature of the collection rather than to the type of object they contain. Examples of collections are sets, queues, stacks, lists, binary, and trees. The purpose of the Collection Object Service is to provide a uniform way to create and manipulate the most common collections generically.

Examples of collections are sets, queues, stacks, lists, binary, and trees. For example, sets might support the following operations: insert new element, membership test, union, intersection, cardinality, equality test, emptiness test, etc. One of the defining

semantics of a set is that, if an object *O* is a member of a set *S*, then inserting *O* into *S* results in the set being unchanged. This property would not hold for another collection type called a bag.

This chapter discusses the principles that were considered in designing Object Services and their interfaces. It also addresses dependencies between Object Services, their relationship to CORBA, and their conformance to existing standards.

2.1 Service Design Principles

2.1.1 Build on CORBA Concepts

The design of each Object Service uses and builds on CORBA concepts:

- Separation of interface and implementation
- Object references are typed by interfaces
- Clients depend on interfaces, not implementations
- Use of multiple inheritance of interfaces
- Use of subtyping to extend, evolve and specialize functionality

Other related principles that the designs adhere to include:

- Assume good ORB and Object Services implementations. Specifically, it is assumed that CORBA-compliant ORB implementations are being built that support efficient local and remote access to “fine-grain” objects and have performance characteristics that place no major barriers to the pervasive use of distributed objects for virtually all service and application elements.
- Do not build non-type properties into interfaces

A discussion and rationale for the design of object services was included in the HP-SunSoft response to the OMG Object Services RFI (OMG TC Document 92.2.10).

2.1.2 *Basic, Flexible Services*

The services are designed to do one thing well and are only as complicated as they need to be. Individual services are by themselves relatively simple yet they can, by virtue of their structuring as objects, be combined together in interesting and powerful ways.

For example, the event and life cycle services, plus a future relationship service, may play together to support graphs of objects. Object graphs commonly occur in the real world and must be supported in many applications. A functionally-rich Folder compound object, for example, may be constructed using the life cycle, naming, events, and future relationship services as “building blocks.”

2.1.3 *Generic Services*

Services are designed to be generic in that they do not depend on the type of the client object nor, in general, on the type of data passed in requests. For example, the event channel interfaces accept event data of any type. Clients of the service can dynamically determine the actual data type and handle it appropriately.

2.1.4 *Allow Local and Remote Implementations*

In general the services are structured as CORBA objects with OMG IDL interfaces that can be accessed locally or remotely and which can have local library or remote server styles of implementations. This allows considerable flexibility as regards the location of participating objects. So, for example, if the performance requirements of a particular application dictate it, objects can be implemented to work with a Library Object Adapter that enables their execution in the same process as the client.

2.1.5 *Quality of Service is an Implementation Characteristic*

Service interfaces are designed to allow a wide range of implementation approaches depending on the quality of service required in a particular environment. For example, in the Event Service, an event channel can be implemented to provide fast but unreliable delivery of events or slower but guaranteed delivery. However, the interfaces to the event channel are the same for all implementations and all clients. Because rules are not wired into a complex type hierarchy, developers can select particular implementations as building blocks and easily combine them with other components.

2.1.6 *Objects Often Conspire in a Service*

Services are typically decomposed into several distinct interfaces that provide different views for different kinds of clients of the service. For example, the Event Service is composed of *PushConsumer*, *PullSupplier* and *EventChannel* interfaces. This simplifies the way in which a particular client uses a service.

A particular service implementation can support the constituent interfaces as a single CORBA object or as a collection of distinct objects. This allows considerable implementation flexibility. A client of a service may use a different object reference to communicate with each distinct service function. Conceptually, these “internal” objects *conspire* to provide the complete service.

As an example, in the Event Service an event channel can provide both *PushConsumer* and *EventChannel* interfaces for use by different kinds of client. A particular client sends a request not to a single “event channel” object but to an object that implements either the *PushConsumer* and *EventChannel* interface. Hidden to all the clients, these objects interact to support the service.

The service designs also use distinct objects that implement specific service interfaces as the means to distinguish and coordinate different clients without relying on the existence of an object equality test or some special way of identifying clients. Using the event service again as an example, when an event consumer is connected with an event channel, a new object is created that supports the *PullSupplier* interface. An object reference to this object is returned to the event consumer which can then request events by invoking the appropriate operation on the new “supplier” object. Because each client uses a different object reference to interact with the event channel, the event channel can keep track of and manage multiple simultaneous clients. This is shown graphically in Figure 2-1.

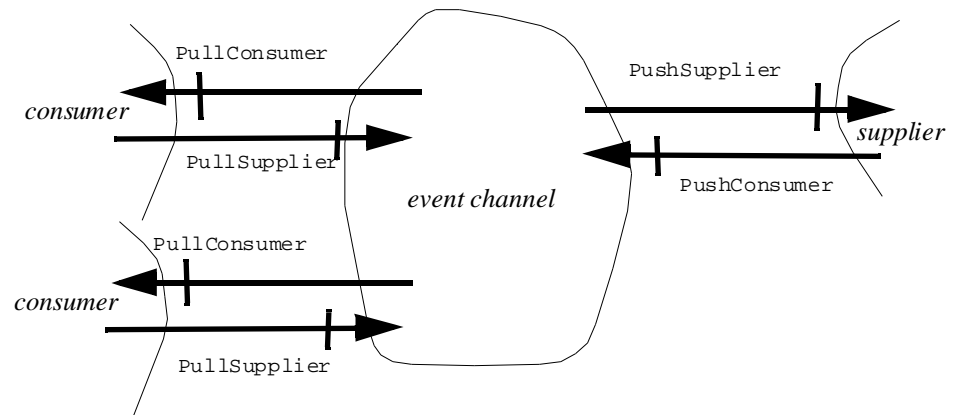


Figure 2-1 An event channel as a collection of objects conspiring to manage multiple simultaneous consumer clients.

The graphical notation shown in Figure 2-1 is used throughout this document and in the full service specifications. An arrow with a vertical bar is used to show that the target object supports the interface named below the arrow and that clients holding an object reference to it of this type can invoke operations. In shorthand, one says that the object reference (held by the client) supports the interface. The arrow points from the client to the target (server) object.

A blob (misshapen circle) delineates a conspiracy of one or more objects. In other words, it corresponds to a conceptual object that may be composed of one or more CORBA objects that together provide some coordinated service to potentially multiple clients making requests using different object references.

2.1.7 Use of Callback Interfaces

Services often employ callback interfaces. Callback interfaces are interfaces that a client object is required to support to enable a service to *call back* to it to invoke some operation. The callback may be, for example, to pass back data asynchronously to a client.

Callback interfaces have two major benefits:

- They clearly define how a client object participates in a service
- They allow the use of the standard interface definition (OMG IDL) and operation invocation (object reference) mechanisms

2.1.8 Assume No Global Identifier Spaces

Several services employ identifiers to label and distinguish various elements. The service designs do not assume or rely on any global identifier service or global id spaces in order to function. The scope of identifiers is always limited to some context. For example, in the naming service, the scope of names is the particular naming context object.

In the case where a service generates ids, clients can assume that an id is unique within its scope but should not make any other assumption.

2.1.9 Finding a Service is Orthogonal to Using It

Finding a service is at a higher level and orthogonal to using a service. These services do not dictate a particular approach. They do not, for example, mandate that all services must be found via the naming service. Because services are structured as objects there does not need to be a special way of finding objects associated with services - general purpose finding services can be used. Solutions are anticipated to be application and policy specific.

2.2 Interface Style Consistency

2.2.1 Use of Exceptions and Return Codes

Throughout the services, exceptions are used exclusively for handling exceptional conditions such as error returns. Normal return codes are passed back via output parameters. An example of this is the use of a DONE return code to indicate iteration completion.

2.2.2 *Explicit Versus Implicit Operations*

Operations are always explicit rather than implied e.g. by a flag passed as a parameter value to some “umbrella” operation. In other words, there is always a distinct operation corresponding to each distinct function of a service.

2.2.3 *Use of Interface Inheritance*

Interface inheritance (subtyping) is used whenever one can imagine that client code should depend on less functionality than the full interface. Services are often partitioned into several unrelated interfaces when it is possible to partition the clients into different roles. For example, an administrative interface is often unrelated and distinct in the type system from the interface used by “normal” clients.

2.3 *Key Design Decisions*

2.3.1 *Naming Service: Distinct from Property and Trading Services*

The Naming Service is addressed separately from property and trading services.

Naming contexts have some similarity to property lists (that is, lists of values associated with objects though not necessarily part of the object’s state). The Naming Service in general also has elements in common with a trading service. However, following the “Bauhaus” principle of keeping services as simple and as orthogonal as possible, these services have been kept distinct and are being addressed separately.

2.3.2 *Universal Object Identity*

The services described in this manual do not require the concept of object identity.

2.4 *Integration with Future Object Services*

This section discusses how the Object Services could evolve to integrate with future services, such as:

- Archive
- Backup/Restore
- Change Management (Versioning)
- Data Interchange
- Implementation Repository
- Internationalization
- Logging
- Recovery
- Replication
- Startup

2.4.1 *Archive Service*

Persistent Object Service. The Archive Service copies objects from an active/persistent store to a backup store and vice versa. This service should be able to archive objects stored with the Persistent Object Service.

Externalization Service. The Archive Service copies objects from an active/persistent store to a backup store and vice versa. This service could use the Externalization Service to get the internal state of objects for saving and to subsequently recreate objects with this stored state. If only persistent objects need to be archived, then the Object Persistence Service could be used instead.

2.4.2 *Backup/Restore Service*

Externalization Service. The Backup/Restore Service provides recovery after a system failure or a user error. This service could use the Object Externalization Service as an underlying mechanism for objects regardless of whether they are persistent.

Persistent Object Service. The Backup/Restore Service provides recovery after a system failure or a user error. This service could use the Persistent Object Service as an underlying mechanism for persistent objects.

Transaction Service. The permanence of effect property of a transaction implies that the state established by the commitment of a transaction will not be lost. To guarantee this property, the storage media on which the objects updated by the transaction are stored must be backed-up to secondary storage to ensure that they are not lost should the primary storage media fail. Similarly, the storage media used by the logging service must be restorable should the media fail. Since there are multiple components which require backup services, a single interface would be advantageous.

2.4.3 *Change Management Service*

Persistent Object Service. The Change Management Service supports the identification and consistent evolution of objects including version and configuration management. This service should work with the Persistent Object Service to allow persistent objects to evolve from the old to new versions.

2.4.4 *Data Interchange Service*

Persistent Object Service. The Data Interchange Service enables objects to exchange some or all of their associated state. This service should work with Persistent Object Service to allow state to be exchanged when one or more of the objects are persistent.

2.4.5 *Internationalization Service*

Naming Service. Naming Service interfaces may also need to be extended (for example, the structure of names extended, additional name resolution operations added) to better support representing and resolving names for some languages and cultures.

2.4.6 *Implementation Repository*

Persistent Object Service. The Implementation Repository supports the management of object implementations. The Persistent Object Service may depend on this to determine what persistent data an object contains. This dependency is at the implementation level.

2.4.7 *Interface Repository*

Persistent Object Service. The Interface Repository supports runtime access to OMG IDL-specified definitions such as object interfaces and type definitions. The Persistent Object Service depends on this to determine if a persistent object supports certain interfaces.

2.4.8 *Logging Service*

Transaction Service. A logging service implements the abstract notion of an infinitely long, sequentially-accessible, append-only file. It typically supports multiple log files, where each log file consists of a sequence of log records. New log records are written to the end of a log file, old log records can be read from any position in the file. To stop log files from growing too large for the underlying storage medium, a log service must provide an operation to archive old log records to allow the log file to be truncated.

Various components of a transaction processing system may require the services of a log service:

- **Transaction Service:** during the two-phase commit protocol the Transaction Service must log its state to ensure that the outcome of the committing transaction can be determined should there be a failure.
- **Recoverable (transactional) objects:** a log can be used to record old and new versions of a recoverable object for the purposes of supporting recovery.
- **Locking service:** a log can be used to record the locks held on an object at prepare time to facilitate recovery.

Since there are multiple components within a distributed transaction processing system that require the services of a log service, a single log service interface (and potentially server) that is shared between the components is clearly advantageous.

The correctness of a transaction service depends upon the services of a log service, for this reason, the log service must meet the following requirements:

1. Restart.

A restart facility allows rapid recovery from the cold start of an application. The recovery service used by the application (indirectly through the application's use of recoverable objects) would use the restart facility to establish a *checkpoint*: a consistent point in the execution state of the application from which the recovery process can proceed. In the absence of a checkpoint the recovery service would have to scan the entire log to ensure restart recovery occurs correctly.

2. Buffering and forcing operations.

A log service should provide two classes of operation for writing log records:

- a. An operation to buffer a log record (the record is not written directly to the underlying storage medium). Used during the execution of a transaction. Since the log record is buffered the write is inexpensive.
- b. An operation to force a log record to the underlying storage medium. Used during the two-phase commit protocol to guarantee the correctness of the transaction. Forcing a log record also flushes all previously written, but buffered, log records.

3. Robustness.

The log service should ensure the consistency of the underlying storage medium in which log files are stored. This usually involves the log service employing protocols that update the storage in a manner that would not result in the loss of any existing data (i.e. careful updates), along with support for mirroring the storage media to tolerate media failures.

4. Archival.

A log service should provide support for archiving log records. Archival is necessary to allow the log to be truncated to ensure that it does not grow without bounds.

5. Efficiency.

Since the log service may be written to by multiple components within a transaction, the addition of log records must be efficient to avoid the bandwidth of log from becoming a bottleneck in the system.

2.4.9 Recovery Service

Transaction Service. As recoverable objects are updated during a transaction, they (as resource managers) keep a record of the changes made to their state that is sufficient to undo the updates should the transaction rollback. The component responsible for this task is termed the recovery service. Various different forms of recovery are possible, however the most common form is called value logging and involves the recoverable object recording both the old and new values of the object. When a transaction is recovered due to failure, the old value of an object is used to undo changes made to the object during the transaction. Most recovery services employ the services of a logging service (described in this section) to maintain the “undo” information. The definition of a standard recovery service interface is one possible additional CORBA-compliant object service.

2.4.10 Replication Service

Persistent Object Service. The Replication Service provides explicit replication of objects in a distributed environment and manages the consistency of replicated copies. This service could use the Persistent Object Service to manage persistent replicas.

2.4.11 *Startup Service*

Persistent Object Service. The Startup Service supports bootstrapping and termination of the Persistent Object Service.

2.4.12 *Data Interchange Service*

Externalization Service. The Data Interchange Service enables objects to exchange some or all of their associated state. This service could use the Object Externalization Service to allow state to be exchanged regardless of whether the objects are persistent.

2.5 *Service Dependencies*

The interface designs of all the services are general in nature and do not presume or require the existence of specific supporting software in order to implement them. An implementation of the Name Service, for instance, could use naming or directory services provided in a general-purpose networking environment. For example, an implementation may be based on the naming services provided by ONC or DCE. Such an implementation could provide enterprise-wide naming services to both object-based and non-object-based clients. Object-based software would see such services through the use of NamingContext objects.

Although the Object Services do not depend upon specific software, some dependencies and relationships do exist between services.

2.5.1 *Event Service*

The Event Service does not depend upon other services.

2.5.2 *Life Cycle Service*

Interfaces for the Life Cycle Service depend on the Naming Service.

The Life Cycle Service also defines compound operations that depend on the Relationship Service for the definition of object graphs. Appendix A describes the topic of compound life cycle, and its dependence on the Relationship Service, in detail.

2.5.3 *Persistent Object Service*

The Externalization Service provides functions that provide for the transformation of an object into a form suitable for storage on an external media or for transfer between systems. The Persistent Object Service uses this service as a POS protocol.

The Life Cycle Service provides operations for managing object creation, deletion, copy and equivalence. The Persistent Object Service depends on this service for creating and deleting all required objects.

The Naming Service provides mappings between user-comprehensible names and CORBA object references. The Persistent Object Service depends on this service to obtain the object reference of, say, a PDS from its name or id.

2.5.4 Relationship Service

The Relationship Service does not depend on other services. Note especially that the Relationship Service does not depend on any common storage service.

For guidelines about when to use the Relationship Service and when to use CORBA object references, refer to the section “The Relationship Service vs CORBA Object References,” in Chapter 9.

2.5.5 Externalization Service

The Externalization Service works with the Life Cycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for graphs of related objects that support compound operations. Specifically, this service uses the Life Cycle Service to create and remove Stream and StreamFactory objects. ORB services may be used in Stream implementations to identify InterfaceDef and ImplementationDef objects corresponding to an externalized object, and to support finding an appropriate factory for recreating that object at internalization time.

The Externalization Service can also work with the Relationship Service. Implementations of Stream and StreamIO operations could use the Relationship Service to ensure that multiple references to the same object or circular references don't result in duplication of objects at internalization time or in the external representation.

In addition, the Externalization Service adds compound externalization semantics to the containment and reference relationships in the Relationship Service. Detailed information is provided in “Specific Externalization Relationships” on page 8-25.

2.5.6 Transaction Service

As concurrent requests are processed by an object a mechanism is required to mediate access. This is necessary to provide the transaction property of isolation. The Concurrency Control Service is one possible implementation of a locking service.

The Transaction Service depends upon the Concurrency Control Service in the following ways:

- Concurrency Control Service must support transaction duration locks, which provide isolation of concurrent requests by different transactions.
- Concurrency Control Service must record transaction duration locks on persistent media, such as a log, as part of the prepare phase of commitment.
- If nested transactions are supported by the Transaction Service then the Concurrency Control Service must also support locks that provide isolation between siblings in a transaction family and provide inheritance of locks owned by a subtransaction to its parent when the subtransaction commits.

- Transactional clients of the Concurrency Control Service are responsible for ensuring that all locks held by a transaction are dropped after all recovery or commitment operations have taken place. The drop-locks operation is provided by the LockCoordinator interface for this purpose.

The Transaction Service supports atomicity and durability properties through the Persistent Object Service (POS). The Transaction Service can work with the POS to support atomic execution of operations on persistent objects. Transactions and persistence are not provided by the same service. When coordination of multiple state changes are required to persistent data, a persistence service requires a transaction service. The POS can provide persistence, but its implementation needs to be changed to support transactional behavior. There are no changes to the interfaces of the POS to support transactions. The following discussion applies to support of persistence when a transaction service is required.

Support for persistence can be built from other specialized services that can also be shared by other object services. Examples include:

- Recovery service: this supports the atomicity and durability properties.
- Logging service: this is used by the recovery service to assist in supporting the atomicity and durability properties. It is also used by the Transaction Service to support the two-phase commit protocol.
- Backup and restore service: this supports the isolation property.

This view is consistent with the X/Open DTP (Distributed Transaction Processing) model which separates the transaction manager service (i.e. the implementation of a generalized two-phase commit protocol) from a resource manager that provides services for data with a life beyond process execution. This permits both transactions on transient objects and on persistent objects without transactions.

2.5.7 Concurrency Control Service

The Concurrency Control Service does not depend on any other service per se. Nevertheless, it is designed to work with the Transaction Service.

2.5.8 Query Service

The Query Service does not depend on other service but is closely related to these Object Services: Life Cycle; Persistent Object; Relationship; Concurrency Control; Transaction; Property; and Collection.

2.5.9 Licensing Service

The Licensing Service depends on the Event Service. It may depend on the Relationship, Property, and Query Services for some implementations. This dependency is determined by an implementation's policy definition and entry capability. The Licensing Service also depends on the Security Service, because the Licensing Service interface can use unforgeable and secure events. The Licensing Service will use Security Service interfaces to support the requirements addressed by the challenge mechanism.

2.5.10 Property Service

The Property Service does not depend upon other services; however, it is closely related to Collection Service.

2.5.11 Time Service

The Time Service does not depend upon other services.

2.5.12 Security Service

The Security Service does not depend upon other services.

2.5.13 Trader Service

The Trader Service does not depend upon other services.

2.5.14 Collections Service

The Collections Service does not depend upon other services; however, it is closely related to these services: Concurrency, Naming, Persistent Object, Property, and Query.

2.6 Relationship to CORBA

This section provides information about the relationship of other services to the CORBA specification.

2.6.1 ORB Interoperability Considerations: Transaction Service

Some implementations of the Transaction Service will support:

- The ability of a single application to use both object and procedural interfaces to the Transaction Service. This is described as part of the specification, particularly in the sections “The User’s View” and “The Implementor’s View.”
- The ability for different Transaction Service implementations to interoperate across a single ORB. This is provided as a consequence of this specification, which defines IDL interfaces for all interactions between Transaction Service implementations.
- The ability for the same Transaction Service to interoperate with another instance of itself across different ORBs. (This ability is supported by the Interoperability specification of CORBA 2.0.)
- The ability for different Transaction Services implementations to interoperate across different ORBs. (This ability is supported by the Interoperability specification of CORBA 2.0.)

- A critical dependency for Transaction Service interoperation across different ORBs is the handling of the `propagation_context` between ORBs. This includes the following:
 - Efficient transformation between different ORB representations of the `propagation_context`.
 - The ability to carry the ID information (typically an X/Open XID) between interoperating ORBs.
 - The ability to do interposition to ensure efficient local execution of the `is_same_transaction` operation.

2.6.2 *Life Cycle Service*

The Life Cycle Service assumes CORBA implementations support object relocation.

2.6.3 *Naming Service*

Entities that are not CORBA objects - that is to say, not objects accessed via an Object Request Broker - are used for names (in the guise of pseudo objects). In both cases the interfaces to these entities conform as closely as possible to OMG IDL while satisfying the specific service design requirements, in order to enable maximum flexibility in the future. Specifically, in the Naming Service, name objects are pseudo objects with interfaces defined in pseudo IDL (PIDL). These objects look like CORBA objects but are specifically designed to be accessed using a programming language binding. This is done for reasons based on the expected use of these objects.

2.6.4 *Relationship Service*

The Relationship Service requires CORBA Interface Repositories to support the ability to dynamically determine if an `InterfaceDef` conforms to another `InterfaceDef`, that is, if it is a subtype. This is needed to implement type constraints for particular relationships.

2.6.5 *Persistent Object Service*

The Persistent Object Service requires CORBA Interface Repositories.

2.6.6 *General Interoperability Requirements*

Interoperability between Object Services and users of Object Services implemented on different ORBs requires common `RepositoryIDs` be used to identify types in both systems. The types identified by these `RepositoryIDs` must also be consistently defined. As described in Common Object Request Broker: Architecture and Specification, Pragma Directives for Repository Id section, all CORBA service IDL presented in this specification is implicitly preceded at file scope by the following directive:

```
#pragma prefix "omg.org"
```

Object Service Implementations that choose to extend the standard interfaces must do so by deriving new interfaces rather than by modifying the standard interfaces.

2.7 *Relationship to Object Model*

All specifications contained in this manual conform to the OMG Object Model. No additional components or profiles are required by any service.

2.8 *Conformance to Existing Standards*

In general, existing relevant standards do not have object-oriented interfaces nor are they structured in a form that is easily mapped to objects. These specifications have been influenced by existing standards, and services have been designed which minimize the difficulty of encapsulating supporting software. The naming service specification is believed to be compatible with X.500, DCE CDS and ONC NIS and NIS+.

These specifications are broadly conformant to emerging ISO/IEC/CCITT ODP standards:

- CCITT Draft Recommendations X.900, ISO/IEC 10746 Basic Reference Model for Open Distributed Computing
- ISO/IEC JTC1 SC21 WG7 N743 Working Document on Topic 9.1 - ODP Trader

14.1 Introduction

14.1.1 Time Service Requirements

The requirements explicitly stated in the RFP ask for a service that enables the user to obtain current time together with an error estimate associated with it.

Additionally, the RFP suggests that the service also provide the following facilities:

- Ascertain the order in which “events” occurred.
- Generate time-based events based on timers and alarms.
- Compute the interval between two events.

Although the RFP mentions specification of a synchronization mechanism, the submitters deemed it inappropriate to specify a single such mechanism as discussed in Section 14.1.3, Source of Time.

14.1.2 Representation of Time

Time is represented many ways in programs. For example the *X/Open DCE Time Service* [1] defines three binary representations of absolute time, while the UNIX SVID defines a different representation of time. Other systems use time represented in myriads of different ways. It is not a goal of the service defined in this submission to deal with all these different representations of time or to propose a new unifying representation of time.

To satisfy the set of requirements that are addressed, we have chosen to use only the Universal Time Coordinated (UTC) representation from the *X/Open DCE Time Service*. Global clock synchronization time sources, such as the UTC signals broadcast by the WWV radio station of the National Bureau of Standards, deliver time, which is relatively easy to handle in this representation. UTC time is defined as follows.

Time units	100 nanoseconds (10 ⁻⁷ seconds)
Base time	15 October 1582 00:00:00.
Approximate range	AD 30,000

UTC time in this service specification always refers to time in Greenwich Time Zone. The corresponding binary representations of relative time is the same one as for absolute time, and hence with similar characteristics:

Time units	100 nanoseconds (10 ⁻⁷ seconds)
Approximate range	+/- 30,000 years

In order to ease implementation on existing systems, migration from them and interoperation with them, care has been taken to ensure that the representation of time used interoperates with *X/Open DCE Time Service* [1], and that the operation for getting current time is easy to implement on *X/Open DCE Time Service*, *NTP* [2] (and for that matter any other reasonable distributed time synchronization algorithm that one might come up with, e.g. ones presented in [3]) with appropriate values for inaccuracies.

14.1.3 Source of Time

The services defined in this chapter depend on the availability of an underlying Time Service that obtains and synchronizes time as required to provide a reasonable approximation of the current time to these services. The following assumptions are made about the underlying time synchronization service:

- The Time Service is able to return current time with an associated error parameter.
- Within reasonable interpretation of the terms, the Time Service is available and reliable. The time provided by the underlying service can be trusted to be within the inaccuracy window provided by the underlying system.
- The time returned by the Time Service is from a monotonically increasing series.

Additionally, if the underlying Time Service meets the criteria to be followed for secure time presented in Appendix A, Implementation Guidelines, then the Time Service object is able to provide trusted time.

No additional assumptions are made about how the underlying service obtains the time that it delivers to this service. For example it could utilize a range of techniques whether it be using a Cesium clock attached to each node or some hardware/software time synchronization method. It is assumed that the underlying service may fail occasionally. This is accounted for by providing an appropriate exception as part of the interface. The availability and accuracy of trusted time depends on what is provided by the underlying Time Service.

14.1.4 General Object Model

The general architectural pattern used is that a service object manages objects of a specific category as shown in Figure 14-1.

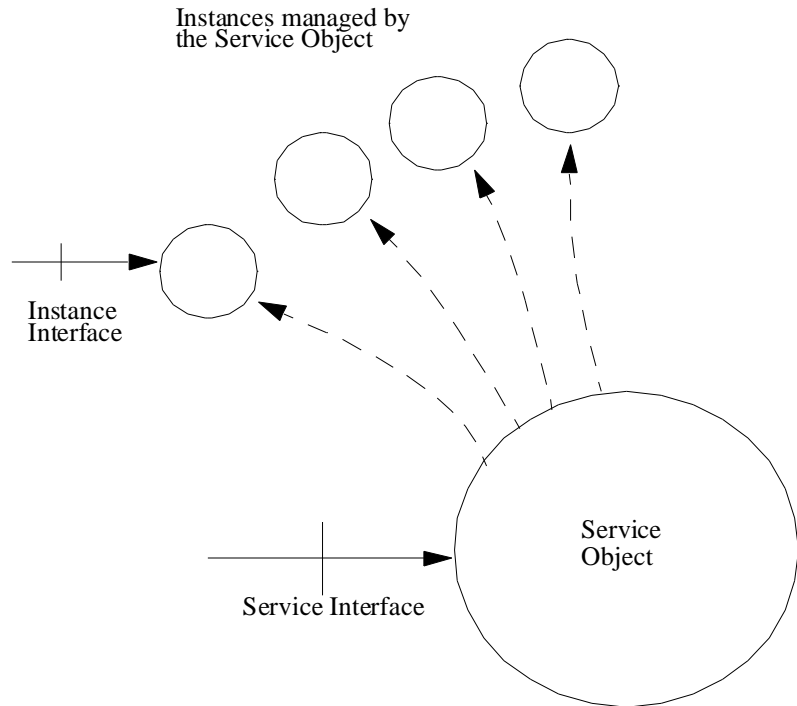


Figure 14-1 General Object Model for Service

The service interface provides operations for creating the objects that the service manages and, if appropriate, also provides operations for getting rid of them.

The Time Service object consists of two services, and hence defines two service interfaces:

- Time Service manages Universal Time Objects (UTOs) and Time Interval Objects (TIOs), and is represented by the *TimeService* interface.
- Timer Event Service manages Timer Event Handler objects, and is represented by the *TimerEventService* interface.

The underlying facility that delivers time is associated with the **UniversalTime** and **SecureUniversalTime** operation of the *TimeService* interface as described in Section 14.2, Basic Time Service.

14.1.5 Conformance Points

There are two conformance points for this service.

- *Basic Time Service.* This service consists of all data types and interfaces defined in the TimeBase and CosTime modules in Section 14.2, Basic Time Service. It provides operations for getting time and manipulating time. A complete implementation of the TimeBase and the CosTime modules is necessary and sufficient to conform to the Time Service object standard. An implementation of the CosTime module in which the **universal_time** operation always raises the **TimeUnavailable** exception is not acceptable for satisfying this conformance point.
- *Timer Event Service.* This service consists of all data types and interfaces defined in the CosTimerEvent module in Section 14.3, Timer Event Service. It provides operations for managing time-triggered event handlers and the events that they handle. A complete implementation of this module is necessary to conform to the optional Timer Event Service component of the Time Service object. Since the CosTimerEvent module depends on the CosTime module, it is not possible to conform just to the Timer Event Service without conforming to Basic Time Service. To claim conformance to Timer Event Service, both Timer Event Service and Time Service must be provided.

14.2 Basic Time Service

All data structures pertaining to the basic Time Service, Universal Time Object, and Time Interval Object are defined in the TimeBase module so that other services can make use of these data structures without requiring the interface definitions. The interface definitions and associated enums and exceptions are encapsulated in the CosTime module.

14.2.1 Object Model

The object model of this service is depicted in Figure 14-2. The Time Service object manages Universal Time Objects (UTOs) and Time Interval Objects (TIOs). It does so by providing methods for creating UTOs and TIOs. Each UTO represents a time, and each TIO represents a time interval, and reference to each can be freely passed around, subject to the caveats discussed in Appendix A, Implementation Guidelines.

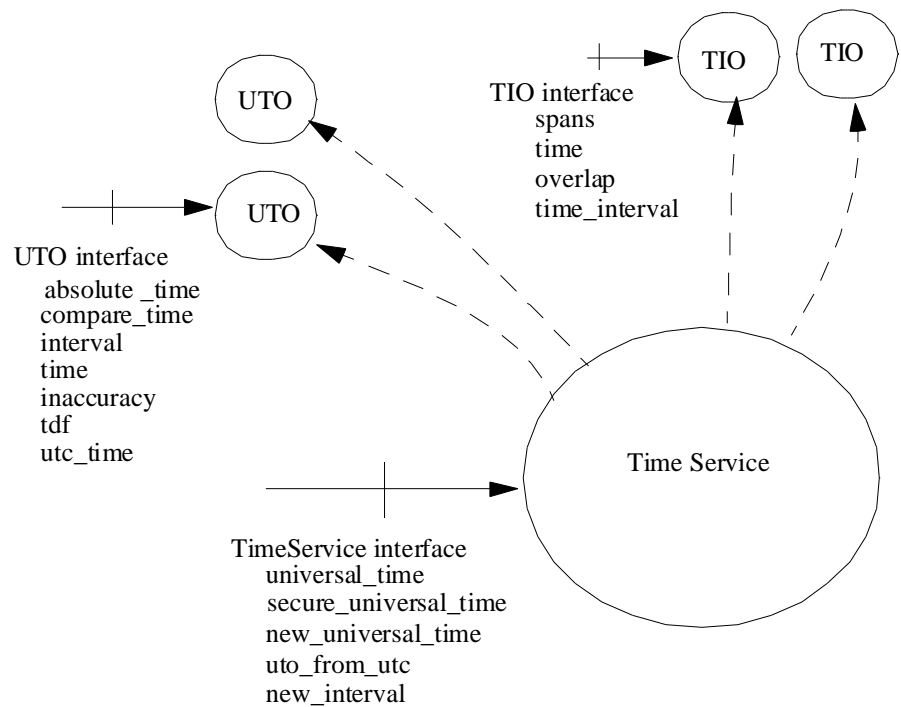


Figure 14-2 Object Model for Time Service

14.2.2 Data Types

A number of types and interfaces are defined and used by this service. All definitions of data structures are placed in the TimeBase module. All interfaces, and associated enum and exception declarations are placed in the CosTime module. This separation of basic data type definitions from interface related definitions allows other services to use the time data types without explicitly incorporating the interfaces, while allowing clients of those services to use the interfaces provided by the Time Service to manipulate the data used by those services.

```

module TimeBase {

    typedef unsigned long long    TimeT;
    typedef TimeT                InaccuracyT;
    typedef short                TdfT;
    struct UtcT {
        TimeT                time;        // 8 octets
        unsigned long        inacclo;    // 4 octets
        unsigned short       inacchi;    // 2 octets
        TdfT                 tdf;        // 2 octets
        // total 16 octets.
    };
    struct IntervalT {
        TimeT                lower_bound;
    };
}

```

```

        TimeT          upper_bound;
    };
};

```

Type TimeT

`TimeT` represents a single time value, which is 64 bits in size, and holds the number of 100 nanoseconds that have passed since the base time. For absolute time the base is 15 October 1582 00:00.

Type InaccuracyT

`InaccuracyT` represents the value of inaccuracy in time in units of 100 nanoseconds. As per the definition of the inaccuracy field in the *X/Open DCE Time Service* [1], 48 bits is sufficient to hold this value.

Type TdfT

`TdfT` is of size 16 bits short type and holds the time displacement factor in the form of minutes of displacement from the Greenwich Meridian. Displacements East of the meridian are positive, while those to the West are negative.

Type UtcT

`UtcT` defines the structure of the time value that is used universally in this service. The basic value of time is of type `TimeT` that is held in the time field. Whether a `UtcT` structure is holding a relative or absolute time is determined by its history. There is no explicit flag within the object holding that state information. The `iacclo` and `inacchi` fields together hold a 48-bit estimate of inaccuracy in the time field. These two fields together hold a value of type `InaccuracyT` packed into 48 bits. The `tdf` field holds time zone information. Implementation must place the time displacement factor for the local time zone in this field whenever they create a UTO.

The contents of this structure are intended to be opaque, but in order to be able to marshal it correctly, at least the types of fields need to be identified.

Type IntervalT

This type holds a time interval represented as two `TimeT` values corresponding to the lower and upper bound of the interval. An `IntervalT` structure containing a lower bound greater than the upper bound is invalid. For the interval to be meaningful, the time base used for the lower and upper bound must be the same, and the time base itself must not be spanned by the interval.

```

module CosTime {
    enum TimeComparison {
        TCEqualTo,
        TCLessThan,
    }
}

```

```

        TCGreaterThan,
        TCIndeterminate
    };

    enum ComparisonType {
        IntervalC,
        MidC
    };

    enum OverlapType {
        OTContainer,
        OTContained,
        OTOverlap,
        OTNoOverlap
    };
};

```

Enum ComparisonType

`ComparisonType` defines the two types of time comparison that are supported. `IntervalC` comparison does the comparison taking into account the error envelope. `MidC` comparison just compares the base times. A `MidC` comparison can never return `TCIndeterminate`.

Enum TimeComparison

`TimeComparison` defines the possible values that can be returned as a result of comparing two UTOs. The values are self-explanatory. In an `IntervalC` comparison, `TCIndeterminate` value is returned if the error envelopes around the two times being compared overlap. For this purpose the error envelope is assumed to be symmetrically placed around the base time covering time-inaccuracy to time+inaccuracy. For `IntervalC` comparison, two UTOs are deemed to contain the same time only if the `Time` attribute of the two objects are equal and the `Inaccuracy` attributes of both the objects are zero.

Enum OverlapType

`OverlapType` specifies the type of overlap between two time intervals. Figure 14-3 depicts the meaning of the four values of this enum. When interval A wholly contains interval B, then it is an `OTContainer` of interval B and the overlap interval is the same as the interval B. When interval B wholly contains interval A, then interval A is `OTContained` in interval B and the overlap region is the same as interval A. When neither interval is wholly contained in the other but they overlap, then the `OTOverlap` case applies and the overlap region is the length of interval that overlaps. Finally, when the two intervals do not overlap, the `OTNoOverlap` case applies.

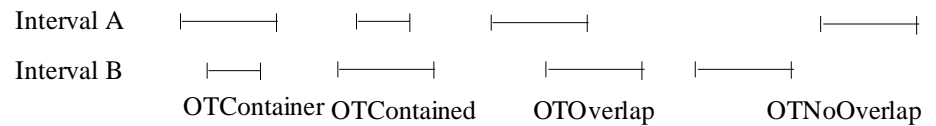


Figure 14-3 Illustration of Interval Overlap

14.2.3 Exceptions

This service returns standard CORBA exceptions where specified in addition to the service-specific exception described in this section.

```
module CosTime {
    exception TimeUnavailable {};
}
```

TimeUnavailable

This exception is raised when the underlying trusted time service fails, or is unable to provide time that meets the required security assurance.

14.2.4 Universal Time Object (UTO)

The UTO provides various operations on basic time. These include the following groups of operations:

- Construction of a UTO from piece parts, and extraction of piece parts from a UTO (as read only attributes).
- Comparison of time.
- Conversion from relative to absolute time, and conversion to an interval.

Of these, the first operation is required for completeness, since in its absence it would be difficult to provide a time input to the timer event handler, for example. The second operation is required by the RFP, and the third is required for completeness and usability.

```
module CosTime {
    interface TIO;           // forward declaration
    interface UTO {
        readonly attribute TimeBase::TimeT      time;
        readonly attribute TimeBase::InaccuracyTinaccuracy;
        readonly attribute TimeBase::TdfT      tdf;
        readonly attribute TimeBase::UtcT      utc_time;

        UTO absolute_time();

        TimeComparison compare_time(
```

```

        in      ComparisonType  comparison_type,
        in      UTO             uto
    );

    TIO time_to_interval(
        in      UTO             uto
    );

    TIO interval();
};
};

```

The *UTO* interface corresponds to an object that contains `utc` time, and is the means for manipulating the time contained in the object. This interface has operations for getting a `UtcT` type data structure containing the current value of time in the object, as well as operations for getting the values of individual fields of `utc` time, getting absolute time from relative time, and comparing and doing bounds operations on `UTOs`. The *UTO* interface does not provide any operation for modifying the time in the object. It is intended that `UTOs` are immutable.

ReadOnly attribute time

This is the time attribute of a `UTO` represented as a value of type `TimeT`.

ReadOnly attribute inaccuracy

This is the inaccuracy attribute of a `UTO` represented as a value of type `InaccuracyT`.

ReadOnly attribute tdf

This is the time displacement factor attribute `tdf` of a `UTO` represented as a value of type `TdfT`.

ReadOnly attribute utc_time

This attribute returns a properly populated `UtcT` structure with data corresponding to the contents of the `UTO`.

Operation absolute_time

This attribute returns a `UTO` containing the absolute time corresponding to the relative time in object. Absolute time = current time + time in the object. Raises **CORBA::DATA_CONVERSION** exception if the attempt to obtain absolute time causes an overflow.

Operation compare_time

Compares the time contained in the object with the time given in the input parameter `uto` using the comparison type specified in the `in` parameter `comparison_type`, and returns the result. See the description of `TimeComparison` in Section 14.2.2, Data Types, for an explanation of the result. See the explanation of `ComparisonType` in Section 14.2.2 for an explanation of comparison types. Note that the time in the object is always used as the first parameter in the comparison. The time in the `utc` parameter is used as the second parameter in the comparison.

Operation time_to_interval

Returns a TIO representing the time interval between the time in the object and the time in the UTO passed in the parameter `uto`. The interval returned is the interval between the midpoints of the two UTOs and the inaccuracies in the UTOs are not taken into consideration. The result is meaningless if the time base used by the two UTOs are different.

Operation interval

Returns a TIO representing the error interval around the time value in the UTO as a time interval. `TIO.upper_bound = UTO.time+UTO.inaccuracy`. `TIO.lower_bound = UTO.time - UTO.inaccuracy`.

14.2.5 Time Interval Object (TIO)

The TIO represents a time interval and contains operations relevant to time intervals.

```
module CosTime {
  interface TIO {
    readonly attribute TimeBase::IntervalT time_interval;

    OverlapType spans (
      in UTO                time,
      out TIO                overlap
    );
    OverlapType overlaps (
      in TIO                interval,
      out TIO                overlap
    );

    UTO time ();
  }
}
```

Readonly attribute time_interval

This attribute returns an `IntervalT` structure with the values of its fields filled in with the corresponding values from the TIO.

Operation spans

This operation returns a value of type `OverlapType` depending on how the interval in the object and the time range represented by the parameter `UTO` overlap. See the definition of `OverlapType` in Section 14.2.2, Data Types. The interval in the object is interval A and the interval in the parameter `UTO` is interval B. If `OverlapType` is not `OTNoOverlap`, then the `out` parameter `overlap` contains the overlap interval, otherwise the `out` parameter contains the gap between the two intervals. The exception **CORBA::BAD_PARAM** is raised if the `UTO` passed in is invalid.

Operation overlaps

This operation returns a value of type `OverlapType` depending on how the interval in the object and interval in the parameter `TIO` overlap. See the definition of `OverlapType` in Section 14.2.2, Data Types. The interval in the object is interval A and the interval in the parameter `TIO` is interval B. If `OverlapType` is not `OTNoOverlap`, then the `out` parameter `overlap` contains the overlap interval, otherwise the `out` parameter contains the gap between the two intervals. The exception **CORBA::BAD_PARAM** is raised if the `TIO` passed in is invalid.

Operation time

Returns a `UTO` in which the inaccuracy interval is equal to the time interval in the `ITO` and time value is the midpoint of the interval.

14.2.6 Time Service

The *TimeService* interface provides operations for obtaining the current time, constructing a `UTO` with specified values for each attribute, and constructing a `TIO` with specified upper and lower bounds.

```
module CosTime {
  interface TimeService {
    UTO universal_time()
      raises(TimeUnavailable
    );
    UTO secure_universal_time()
      raises(TimeUnavailable
    );
    UTO new_universal_time(
      in TimeBase::TimeT      time,
      in TimeBase::InaccuracyT inaccuracy,
      in TimeBase::TdfT      tdf
    );
    UTO uto_from_utc(
      in TimeBase::UtcT      utc
    );
  };
};
```

```

        TIO new_interval(
            in TimeBase::TimeT      lower,
            in TimeBase::TimeT      upper
        );
    };
};

```

Operation universal_time

The **universal_time** operation returns the current time and an estimate of inaccuracy in a UTO. It raises `TimeUnavailable` exceptions to indicate failure of an underlying time provider. The time returned in the UTO by this operation is not guaranteed to be secure or trusted. If any time is available at all, that time is returned by this operation.

Operation secure_universal_time

The **secure_universal_time** operation returns the current time in a UTO only if the time can be guaranteed to have been obtained securely. In order to make such a guarantee, the underlying Time Service must meet the criteria to be followed for secure time, presented in Appendix A, Implementation Guidelines. If there is any uncertainty at all about meeting any aspect of these criteria, then this operation must return the `TimeUnavailable` exception. Thus, time obtained through this operation can always be trusted.

Operation new_universal_time

The **new_universal_time** operation is used for constructing a new UTO. The parameters passed in are the time of type `TimeT` and `inaccuracy` of type `InaccuracyT`. This is the only way to create a UTO with an arbitrary time from its components. This is expected to be used for building UTOs that can be passed as the various time arguments to the Timer Event Service, for example.

CORBA::BAD_PARAM is raised in the case of an out-of-range parameter value for `inaccuracy`.

Operation uto_from_utc

The **uto_from_utc** operation is used to create a UTO given a time in the `UTC` form. This has a single `in` parameter `UTC`, which contains a time together with `inaccuracy` and `tdf`. The UTO returned is initialized with the values from the `UTC` parameter. This operation is used to convert a UTC received over the wire into a UTO.

Operation new_interval

The **new_interval** operation is used to construct a new TIO. The parameters are `lower` and `upper`, both of type `TimeT`, holding the lower and upper bounds of the interval. If the value of the `lower` parameter is greater than the value of the `upper` parameter, then a **CORBA::BAD_PARAM** exception is raised.

14.3 Timer Event Service

The module CosTimerEvent encapsulates all data type and interface definitions pertaining to the Timer Event Service.

14.3.1 Object Model

The TimerEventService object manages Timer Event Handlers represented by Timer Event Handler objects as shown in Figure 14-4. Each Timer Event Handler is immutably associated with a specific event channel at the time of its creation. The Timer Event Handler can be passed around as any other object. It can be used to program the time and content of the events that will be generated on the channel associated with it. The user of a Timer Event Handler is expected to notify the Timer Event Service when it has no further use for the handler.

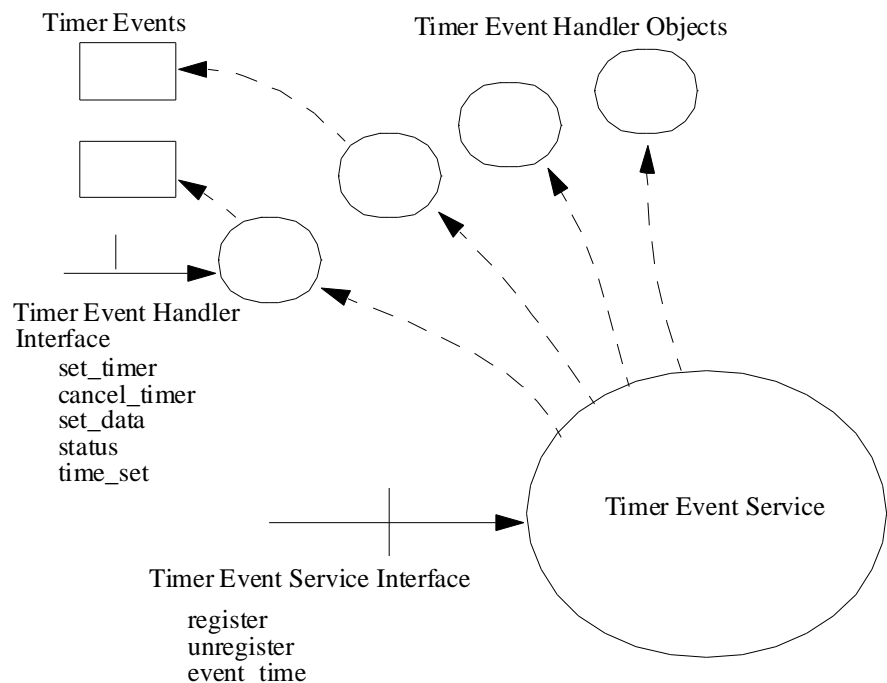


Figure 14-4 Object Model of Timer Event Service

14.3.2 Usage

In a typical usage scenario of this service, the user must first create an event channel of the “push” type (see *CORBA Service: Event Service Specification* [Chapter 4]). The user must then register this event channel as the sink for events generated by the timer event handler that is returned by the registration operation. The user can then use the

timer event handler object to set up timer events as desired. The service will cause events to be pushed through the event channel within a reasonable interval around the requested event time. The implementor of the service will document what the expected interval is for their implementation. The data associated with the event includes a timestamp of the actual event time with the error envelope including the requested event time.

14.3.3 Data Types

All declarations pertaining to this service is encapsulated in the `CosTimerEvent` module.

```
module CosTimerEvent{
    enum TimeType {
        TTAbsolute,
        TTRelative,
        TTPeriodic
    };

    enum EventStatus {
        ESTimeSet,
        ESTimeCleared,
        ESTriggered,
        ESFailedTrigger
    };

    struct TimerEventT{
        TimeBase::UtcT    utc;
        any                event_data;
    };
};
```

Enum TimeType

`TimeType` is used to specify whether a time is `TTRelative`, `TTAbsolute`, or `TTPeriodic` in operations for setting timer intervals for the event-triggering mechanism. The `TTRelative` value is used to specify that the time provided is relative to current time, `TTAbsolute` is used to specify that the time provided is absolute, and `TTPeriodic` is used to specify that the time provided is a period (and hence a relative time) between successive events. If `TTPeriodic` is used, then the same event continues to be triggered repeatedly at the completion of the time interval specified, until the timer is reset.

Enum EventStatus

`EventStatus` defines the state of a `TimerEventHandler` object. The state `ESTimeSet` means that the event has been set with a time in the future, and will be triggered when that time arrives. `ESTimeCleared` means that the event is not set to go off, and the time was cleared before the previously set triggering time arrived.

`ESTriggered` means that the event has already triggered and the appropriate data has been sent the event channel. `ESFailedTrigger` means that the event did trigger, but data could not be delivered over the event channel.

In case of `TTPeriodic` events, the status `ESTriggered` never occurs. Upon successful triggering of a `TTPeriodic` event, the status is set to `ESTimeSet`.

Type TimerEventT

This is the structure that is returned to the event requester by the time-driven event-triggering mechanism. It has two fields. The first field, `utc`, contains the actual time at which the event was triggered. This value is set in the time field of `utc`. The inaccuracy fields `inacclo` and `inacchi` of `utc` are set to the difference between the requested event time and the actual event time.

The second field, `event_data`, contains the data that the requester of the event had asked to be sent when the event was triggered.

14.3.4 Exceptions

Timer Event Service raises standard CORBA exceptions as specified in OMG IDL for the service. It does not have any service-specific exceptions.

14.3.5 Timer Event Handler

Timer Event Handlers are created and managed by the Timer Event Service. A `TimerEventHandler` object holds information about an event that is to be triggered at a specific time and action that is to be taken when the event is triggered. It provides operations for setting, resetting, and canceling the timer event associated with it, as well as for changing the event data that is sent back as a part of a `TimeEventT` structure on the event channel upon the triggering of the event. The only thing that cannot be changed is the event channel associated with that event handler. An attribute named `status` holds the current status of the event handler.

```
module CosTimerEvent {
    interface TimerEventHandler {
        readonly attribute EventStatus      status;
        boolean time_set(
            out CosTime::UTO                uto
        );
        void set_timer(
            in TimeType                      time_type,
            in CosTime::UTO                  trigger_time
        );
        boolean cancel_timer();
        void set_data(
            in any                            event_data
        );
    };
};
```

Attribute status

`status` is a readonly attribute that reflects the current state of the `TimerEventHandler`. See the definition of `EventStatus` enumerator in Section 14.3.1, Object Model, for details.

Operation time_set

Returns **TRUE** if the time has been set for an event that is yet to be triggered, **FALSE** otherwise. In addition, it always returns the current value of the timer in the event handler as the `out_uto` parameter.

Operation set_timer

Sets the triggering time for the event to the time specified by the `uto` parameter, which may contain `TTRelative`, `TTAbsolute` or `TTPeriodic` time. The `time_type` parameter specifies what type of time is contained in the `uto` parameter. The previous trigger, if any, is canceled and a new trigger is enabled at the time specified if `absolute`, or at current time + time specified if `relative`. If a `relative` time value of zero is specified (i.e. the time attribute of `utc = 0LL`), then the last `relative` time that was specified is reused. If no `relative` time was previously specified, then a **CORBA::BAD_PARAM** exception is raised. If a `periodic` time is specified (`time_type == periodic`), then the time parameter is interpreted as a `relative` time and the time trigger is set at the periodicity defined by the time (i.e. at current time + time, current time + 2 * time, etc.).

Operation cancel_timer

Cancels the trigger if one had been set and had not gone off yet. Returns **TRUE** if an event is actually canceled, **FALSE** otherwise.

Operation set_data

The data that will be passed back through the event channel in a `TimerEventT` structure for all future triggering of the event handler is set to `event_data`.

14.3.6 Timer Event Service

The Timer Event Service provides operations for registering and unregistering events.

```
module CosTimerEvent {
    interface TimerEventService {

        TimerEventHandler register(
            in CosEventComm::PushConsumer event_interface,
            in any data
        );
        void unregister(
            in TimerEventhandler timer_event_handler
        );
    };
};
```

```

        );
        CosTime::UTO event_time(
            in TimerEventT                timer_event
        );
    };
};

```

Operation register

The **register** operation registers the event handler specified by the data and the `event_interface` parameters. When the event handler is triggered, the data is delivered using the **push** operation (of the *PushConsumer* interface in Chapter 4, Event Service Specification, Section 4.3, CosEventComm Module) specified in the `event_interface` parameter. Only the *Push Model* is supported for timer event delivery. Note that the event handler needs to be primed with a triggering time using the **set_time** operation of the *TimerEventHandler* interface in order for an actual event to be triggered. At initialization, the time in the handler is set to current time and its state is set to `ESTimeCleared`, and no event is scheduled. Raises **CORBA::NO_RESOURCE** exception if lack of resources causes it to fail to register the event handler.

Operation unregister

The **unregister** operation notifies the service that the `timer_event_handler` will not be used any more and all resources associated with it can be destroyed. Subsequent attempts to use that object reference will raise **CORBA::INV_OBJREF**.

Operation event_time

The **event_time** operation returns a UTO containing the time at which the event contained in the `timer_event` structure was triggered.

14.4 Conformance

It is sufficient to provide just the Time Service (module TimeBase and CosTime) to claim conformance with the Time Service object as described in Section 14.1.5, Conformance Points. To claim conformance with the Timer Event Service, both Time Service and Timer Event Service (module CosTimerEvent) must be provided.

In order to conform to the Basic Time Service, the semantics of the **secure_universal_time** operation must be strictly adhered to. In order to return a valid time from this operation, the vendor must provide a statement about how the security assurance criteria specified in Appendix A, Implementation Guidelines, are met in their product. To conform to the object Time Service, in all other cases, i.e. when the security assurance criteria are not satisfied, the **secure_universal_time** operation must raise the `TimeUnavailable` exception.

Appendix A *Implementation Guidelines*

A.1 Introduction

This appendix contains advice to implementors. Appropriate documented handling of the criteria presented here is mandatory for conformance to the Basic Time Service conformance point.

A.2 Criteria to Be Followed for Secure Time

The following criteria must be followed in order to assure that the time returned by the **secure_universal_time** operation is in fact secure time. If these criteria are not satisfactorily addressed in an ORB, then it must return the `TimeUnavailable` exception upon invocation of the **secure_universal_time** operation of the *TimeService* interface.

Administration of Time

Only administrators authorized by the system security policy may set the time and specify the source of time for time synchronization purposes.

Protection of Operations and Mandatory Audits

The following types of operations must be protected against unauthorized invocation. They must also be mandatorily audited:

- Operations that set or reset the current time
- Operations that designate a time source as authoritative
- Operations that modify the accuracy of the time service or the uncertainty interval of generated timestamps

Synchronization of Time

Synchronization of time must be transmitted over the network. This presents an opportunity for unauthorized tampering with time, which must be adequately guarded against. Time Service implementors must state how time values used for time synchronization are protected while they are in transit over the network.

Time Service implementors must state whether or not their implementation is secure. Implementors of secure time services must state how their system is secured against threats documented in Chapter 15, Security Service Specification. They must also document how the issues mentioned in this section are addressed adequately.

A.3 Proxies and Time Uncertainty

The Time Service object returns a timestamp, which contains both a time and an associated uncertainty interval. These values are considered valid at the instant they are returned by the Time Service object; however, if these values are not delivered to the caller immediately, they may no longer be reliable by the time the caller receives them.

In a CORBA system, the use of proxy objects can render time values unreliable by introducing unpredictable and uncorrected latency between the time the time server object generates a timestamp and the time the caller's time server proxy receives the timestamp and returns it to the caller (see Figure 14-5 below).

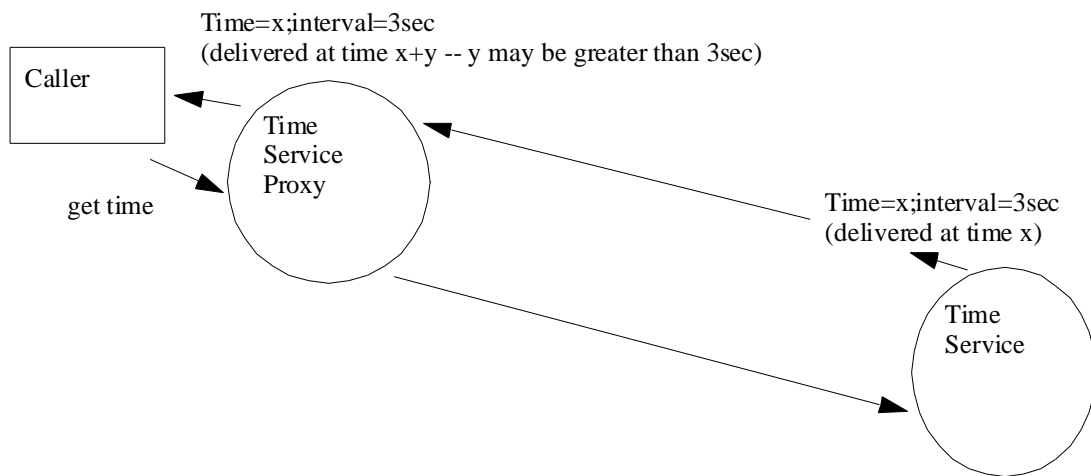


Figure 14-5 Time Service and Proxies

Implementors of the Time Service must prevent this problem from occurring. Two possible ways of preventing proxy latency are:

- Prohibit proxies of the time server object (i.e. require a Time Service implementation in every address space that will need to make Time Service calls).
- Create a special time server proxy, which measures latency between the Time Service object and the proxy, recalculates the time interval's uncertainty, and adjusts the interval value before returning the timestamp to the caller.

Other approaches probably exist; the two above are intended as examples only.

Appendix B Consolidated OMG IDL

B.1 Introduction

This appendix contains a summary of the OMG IDL defined in this document.

B.2 Time Service

This section contains the OMG IDL definitions pertaining to the Time Service, which is encapsulated in the TimeBase and CosTime modules. The TimeBase module contains the basic data type declarations that can be used by others without pulling in the Time Service interfaces. The *Time Service* interface and associated enums and exceptions are declared in the CosTime module.

```

module TimeBase {
    typedef unsigned long long    TimeT;
    typedef TimeT                InaccuracyT;
    typedef short                TdfT;
    struct UtcT {
        TimeT                time;        // 8 octets
        unsigned long        inacclo;    // 4 octets
        unsigned short       inacchi;    // 2 octets
        TdfT                 tdf;        // 2 octets
                                        // total 16 octets.
    };

    struct IntervalT {
        TimeT                lower_bound;
        TimeT                upper_bound;
    };
};

module CosTime {

    enum TimeComparison {
        TCEqualTo,
        TCLessThan,
        TCGreaterThan,
        TCIndeterminate
    };

    enum ComparisonType{
        IntervalC,
        MidC
    };

    enum OverlapType {
        OTContainer,
        OTContained,
        OTOverlap,
        OTNoOverlap
    };
};

```

```

exception TimeUnavailable {};
interface TIO;           // forward declaration

interface UTO {
    readonly attribute TimeBase::TimeTtime;
    readonly attribute TimeBase::InaccuracyTinaccuracy;
    readonly attribute TimeBase::TdfT    tdf;
    readonly attribute TimeBase::UtcT    utc_time;
    UTO absolute_time();
    TimeComparison compare_time(
        in      ComparisonType comparison_type,
        in      UTO          uto
    );
    TIO time_to_interval(
        in      UTO          uto
    );
    TIO interval();
};

interface TIO {
    readonly attribute TimeBase::IntervalT time_interval;
    boolean spans (
        in UTO          time,
        out TIO         overlap
    );
    boolean overlaps (
        in TIO          interval,
        out TIO         overlap
    );
    UTO time ();
};

interface TimeService {
    UTO universal_time()
        raises(TimeUnavailable)
    );
    UTO secure_universal_time()
        raises(TimeUnavailable)
    );
    UTO new_universal_time(
        in TimeBase::TimeT    time,
        in TimeBase::InaccuracyT inaccuracy,
        in TimeBase::TdfT    tdf
    );
    UTO uto_from_utc(
        in TimeBase::UtcT    utc
    );
    TIO new_interval(
        in TimeBase::TimeT    lower,
        in TimeBase::TimeT    upper
    );
};
};
};

```

B.3 Timer Event Service

This section contains all the OMG IDL definitions pertaining to the Timer Event Service, which are encapsulated in the CosTimerEvent module. This module depends on TimeBase, CosTime, CosEventComm and CORBA.

```

module CosTimerEvent{
    enum TimeType {
        TTAbsolute,
        TTRelative,
        TTPeriodic
    };

    enum EventStatus {
        ESTimeSet,
        ESTimeCleared,
        ESTriggered,
        ESFailedTrigger
    };

    struct TimerEventT {
        TimeBase::UtcT          utc;
        any                     event_data;
    };

    interface TimerEventHandler {
        readonly attribute EventStatus status;
        boolean time_set(
            out CosTime::UTO          uto
        );
        void SetTimer(
            in TimeType                time_type,
            in CosTime::UTO            trigger_time
        );
        boolean cancel_timer();
        void set_data(
            in any                     event_data
        );
    };

    interface TimerEventService {
        TimerEventHandler register(
            in CosEventComm::PushConsumer event_interface,
            in any                          data
        );
        void unregister(
            in TimerEventHandler          timer_event_handler
        );
        CosTime::UTO event_time(
            in TimerEventT                timer_event
        );
    };
};

```

Appendix C Notes for Users

C.1 Introduction

This appendix contains notes covering the following matters:

- Guarding against proxy-related inaccuracies in time contained in UTO.
- How to transmit time and time intervals across the network and recover the corresponding UTO and TIO at the other end.

C.2 Proxies and Time

As explained in Appendix B, Consolidated OMG IDL, indiscriminate use of remote proxies to obtain value of current time can lead to obtaining values of time in which the inaccuracy is incorrect due to transmission delays. Consequently, care should be taken to ensure that the local Time Service is used to obtain the value of current time.

C.3 Sending Time Across the Network

When passing small objects such as UTO and TIO from one location to another, one should be aware that each time the passed object reference is used by the recipient it causes an object invocation to take place across the network and is inherently inefficient. The preferred way of dealing with this problem is to pass small objects by value instead of by reference. Unfortunately, due to various reasons, OMG IDL does not allow specification of passing of object parameters by value. Consequently, the user has to explicitly take action to avoid this problem.

The interfaces defined contain features that make it possible for the user to explicitly send the value of time, and time interval across from one location to another and then reconstruct the appropriate object at the receiving end. This is done as follows:

- The signature of the operation that passes time or time interval as a parameter across the network should specify that time is passed as the data type and not as an object reference. For example, for passing universal time, a signature such as

```
void foo(in TimeBase::UtcT);
```

should be used instead of

```
void foo(in CosTime::UTO);
```

- The invoker should use the data attribute of the UTO as the `in` parameter. In pseudo-code, something such as the following should be done by the invoker:

```
CosTime::UTO uto = CosTime::universal_time();
foo(uto.data);
```

- At the server end, the time data received can be converted to a UTO as follows:

```
foo(in TimeBase::UtcT utc) {  
    CosTime::UTO uto = CosTime::TimeService::uto_from_utc(utc);  
  
    .....  
  
};
```

It would be nice to say in the definition of the **foo** operation something such as:

```
foo(in byvalue UTO uto);
```

and have the system take care of doing essentially what is described above. However, there are difficult model- and paradigm-related issues that need resolution before such a change can be coherently proposed.

Appendix D Extension Examples

D.1 Introduction

The process of constructing the contents of a `TimeBase::TimeT` value can be quite tedious, involving many 64-bit multiplications and additions. The CORBA Facility for Time Representation is going to provide user-friendly ways of creating `TimeT` data and displaying them. However, if one is planning to use only the Time Service, it will be necessary to construct some rudimentary facility to build `TimeT` things. This appendix shows one way of doing this as an example of how to extend this service in useful ways.

D.2 Object Model

Following the design pattern used in the rest of this service definition, the basic extension is to define a `TimeI` object corresponding to the `TimeT` structure, and extend `TimeService` to provide an operation for creating such objects. The `TimeI` object has attributes corresponding to the user-friendly representation of time such as year, month, day, hour, minute, second, microsecond, etc.

D.3 Summary of Extensions

The additions are encapsulated in the `FriendlyTime` module. The changes are as follows:

- Data type declaration for components of time.
- Definition of the `TimeI` interface, consisting mostly of attributes.
- Definition of the `FriendlyTime::TimeService` interface derived from the `CosTime::TimeService` interface, for adding the operation to create `TimeI` objects.

D.4 Data Types

The data types are self-explanatory for the purposes of setting up this example. A complete specification should state more specific properties of each of these data types.

```
module FriendlyTime {
    typedef unsigned short    YearT; // must be > 1581
    typedef unsigned short    MonthT; // 1 - 12
    typedef unsigned short    DayT; // 1 - 31
    typedef unsigned short    HourT; // 0 - 24
    typedef unsigned short    MinuteT; // 0 - 59
    typedef unsigned short    SecondT; // 0 - 59
    typedef unsigned short    MicrosecondT;
}
```

D.5 Exceptions

No exceptions are defined in this module.

D.6 Friendly Time Object

The time object provides a friendly interface to the various components usually used to represent time in normal human discourse. The set of attributes used in this example are by no means exhaustive, and is used only for illustrative purposes.

```
module FriendlyTime {
  interface TimeI {
    attribute YearT          year;
    attribute MonthT        month;
    attribute DayT          day;
    attribute HourT         hour;
    attribute MinuteT       minute;
    attribute SecondT       second;
    attribute MicrosecondT  microsecond;
    attribute TimeBase::TimeT time;
    void reset(); // set all attributes to zero
  };
};
```

The `TimeI` object can be viewed as a representation conversion object. The general technique for using it is to create one using the operation **CosFriendlyTime::TimeService::time** introduced in Section D.7, Extended Time Service. This creates a `TimeI` object with time set to zero in it. Then the `_set` operation can be used to set the values of the various attributes. Finally, the attribute `time` can be used to get the corresponding `TimeT` value.

Conversely, one can set any `TimeT` value in the `time` attribute and then get the year, month, etc. from the appropriate attributes.

The **reset** operation facilitates reuse of time objects.

D.7 Extended Time Service

CosTime::TimeService is extended by derivation to provide an operation for creating `TimeI` objects.

```
module FriendlyTime {
  interface TimeService : CosTime::TimeService {
    TimeI time();
  };
};
```


D.8 Epilogue

The extension provided in this appendix makes the Time Service defined in the normative part of the document more easily usable. This leads one to wonder why this extension is not part of the main body of this submission. The reason is that there is no agreement on what the most useful representative components of time are, and the feeling that in general this should be dealt with at the Common Facilities level in general. We still felt that it would be useful to illustrate how easy it is to extend the basic service to provide this ease-of-use facility, thus this appendix.

Appendix E References

- X/Open DCE Time Service, X/Open CAE Specification C310, November 1994.
- RFC 1119 Network Time Protocol, D. Mills, September 1989.
- Probabilistic Clock Synchronization, Flaviu Cristian, Distributed Computing (1989) 3: Pg. 146-158.
- OMG IDL type Extensions RFP, Andrew Watson Ed., OMG Doc. No. 95-1-35.
- CORBAServices: Common Object Service Specification, OMG Doc. No. 95-3-31, March 31 1995 revision, Chapter 4, Event Service Specification, Section 4.2 Pg. 4-6.
- CORBAServices: Common Object Service Specification, OMG Doc. No. 96-10-1, October 1996 revision, Chapter 15, Security Service Specification.

This chapter provides complete documentation for the Object Collection Service specification.

Contents

This chapter contains the following sections.

Section Title	Page
“Overview”	17-2
“Service Structure”	17-2
“Combined Collections”	17-10
“Restricted Access Collections”	17-14
“The CosCollection Module”	17-15
Appendix A, “OMG Object Query Service”	17-124
Appendix B, “Relationship to Other Relevant Standards”	17-133
Appendix C, “References”	17-138

17.1 Overview

Collections support the grouping of objects and support operations for the manipulation of the objects as a group. Common collection types are queues, sets, bags, maps, etc. Collection types differ in the “nature of grouping” exposed to the user. “Nature of grouping” is reflected in the operations supported for the manipulation of objects as members of a group. Collections, for example, can be ordered and thus support access to an element at position “i” while other collections may support associative access to elements via a key. Collections may guarantee the uniqueness of elements while others allow multiple occurrences of elements. A user chooses a collection type that matches the application requirements based on manipulation capabilities.

Collections are foundation classes used in a broad range of applications; therefore, they have to meet the general requirement to be able to collect elements of arbitrary type. On the other hand, a collection instance usually is a homogenous collection in the sense that all elements collected are of the same type, or support the same single interface.

Sometimes you may not want to do something to all elements in a collection, but only treat an individual object or traverse a collection explicitly (not implicitly via a collection operation). To enable this, a pointer abstraction often called an iterator is supported with collections. For example, an iterator points to an element in a collection and processes the element pointed to. Iterators can be moved and used to visit elements of a collection in an application defined manner. There can be many iterators pointing to elements of the same collection instance.

Normally, when operating on all elements of a collection, you want to pass user-defined information to the collection implementation about what to do with the individual elements or which elements are to be processed. To enable this, function interfaces are used. A collection implementation can rely on and use the defined function interface. A user has to specialize and implement these interfaces to pass the user-defined information to the implementation. A function interface can be used to pass element type specific information such as how to compare elements or pass a “program” to be applied to all elements.

17.2 Service Structure

The purpose of an Object Collection Service is to provide a uniform way to create and manipulate the most common collections generically. The Object Service defines three categories of interfaces to serve this purpose.

1. **Collection interfaces** and **collection factories**. A client chooses a collection interface which offers grouping properties that match the client’s needs. A client creates a collection instance of the chosen interface using a collection factory. When creating a collection, a client has to pass element type specific information such as how to compare elements, how to test element equality, or the type checking desired. A client uses collections to manipulate elements as a group. When a

collection is no longer used it may be destroyed - this includes removing the elements collected, destroying element type specific information passed, and the iterators pointing to this collection.

2. **Iterator interfaces.** A client creates an iterator using the collection for which it is created as factory. A client uses an iterator to traverse the collection in an application defined manner, process elements pointed to, mark ranges, etc. When a client no longer uses an iterator, it destroys the iterator.
3. **Function interfaces.** A client creates user-defined specializations of these interfaces using user-defined factories. Instances are passed to a collection implementation when the collection is created (element type specific information) or as a parameter of an operation (for example, code to be executed for each element of the collection). Instances of function interfaces are used by a collection implementation rather than by a client.

17.2.1 Combined Property Collections

The Object Collection Service (or simply Collection Service) defined in this specification aims at being a complete and differentiated offering of interfaces supporting the grouping of objects. It enables a user to make a choice when following the rule “pay only for what you use.” With this goal in mind, a very systematic approach was chosen.

Groups, or collections of objects, support operations and exhibit specific behaviors that are mainly related to the nature of the collection rather than the type of objects they collect.

“Nature of the collection” can be expressed in terms of well defined properties.

Ordering of elements

A *previous* or *next* relationship exists between the elements of an *ordered collection* which is exposed in the interface.

Ordering can be sequential or sorted. A sequential ordering can be explicitly manipulated; however, a sorted ordering is to be maintained implicitly based on a sort criteria to be defined and passed to the implementation by the user.

Access by key

A *key collection* allows associative access to elements via a *key*. A key can be computed from an element value via a user-defined key operation. Furthermore, key collections require key equality to be defined.

Element equality

An *equality collection* exploits the property that a test for element equality is defined (i.e., it can be tested whether an element is equal to another in terms of a user-defined element equality operation). This enables a test on containment, for example.

Uniqueness of entries

A collection with *unique* entries allows exactly one occurrence of an element key value, not *multiple* occurrences.

Meaningful combinations of these basic properties define “collections of differing nature of grouping.” Table 17-1 provides an overview of meaningful combinations. The listed combinations are described in more detail in the following section.

Table 17-1 Interfaces derived from combinations of collection properties

		Unordered		Ordered		
				Sorted		Sequential
		Unique	Multiple	Unique	Multiple	Multiple
Key (Key equality must be specified)	Element Equality	Map	Relation	Sorted Map	Sorted Relation	
	No Element Equality	KeySet	KeyBag	Key Sorted Set	Key SortedBag	
No Key	Element Equality	Set	Bag	SortedSet	Sorted Bag	Equality Sequence
	No Element Equality		Heap			Sequence

Properties are mapped to interfaces - each interface assembling operations that exploit these properties. These interfaces are combined via multiple inheritance and form an *abstract interface hierarchy*. Abstract means that no instance of such a class can be instantiated, an attempt to do so may raise an exception at run-time. Leaves of this hierarchy represent concrete interfaces listed in the table above and can be instantiated by a user. They form a complete and differentiated offering of collection interfaces.

Restricted Access Collections

Common data structures based on these properties sometimes restrict access such as queues, stacks, or priority queues. They can be considered as restricted access variants of *Sequence* or *KeySortedBag*. These interfaces form their own hierarchy of *restricted access interfaces*. They are not incorporated into the hierarchy of combined properties because a user of restricted access interfaces should not be bothered with inherited operations which cannot be used in these interfaces. Nevertheless, to support several “views” on an interface, a restricted users view of a queue and an unrestricted system administrators view to the same queue instance, the restricted access collections are defined in a way that allows combining them with the combined properties collections via multiple inheritance.

All collections are unbounded (there is no explicit bound set) and controlled by the collections; however, it depends on the quality of service delivered whether there are “natural” limits such as the size of the paging space.

Collection Factories

For each concrete collection interface specified in this specification there is one corresponding collection factory defined. Each such factory offers a typed create operation for the creation of collection instances supporting the respective collection interface.

Additionally, a generic extensible factory is specified to enable the usage of many implementation variants for the same collection interface. This extensible generic factory allows the registration of implementation variants and their user-controlled selection at collection creation time.

Information to be passed to a collection at creation time is the element and key type specific information that a collection implementation relies on. That is, one passes the information how to compare element keys, how to test equality of element keys, type checking relevant information, etc. Which type of information needs to be passed depends on the respective collection interface.

17.2.2 Iterators

Iterators, as defined in this specification, are more than just simple “pointing devices.”

Iterator hierarchy

The service defines a hierarchy of iterators which parallels the collection hierarchy.

The top level iterator is generic in the sense that it allows iteration over all collections, independent of the collection type because it is supported by all collection types. The ordered iterator adds some capabilities useful for all kinds of ordered collections. Iterators further down in the hierarchy add operations exploiting the capabilities of the corresponding collection type. Each iterator type is supported by each collection type. For example, a **KeyIterator** is supported only by collection interfaces derived from **KeyCollection**.

Iterators are tightly intertwined with collections. An iterator cannot exist independently of a collection (i.e., the iterator life time cannot exceed that of the collection for which it is created). A collection is the factory for *its* iterators. An iterator is created for a given collection and can be used for this, and only this, collection.

Generic and iterator centric programming

Iterators on the one hand are pointer abstractions in the sense of simple pointing devices. They offer the basic capabilities you can expect from a pointer abstraction. One can reset an iterator to a start position for iteration and move or position it in different ways depending on the iterator type.

There are essentially two reasons to embellish an iterator with more capabilities.

1. To support the processing of very large collections to allow for delayed instantiation or incremental query evaluation in case of very large query results. These are scenarios where the collection itself may never exist as instantiated main memory collection but is processed in “fine grains” via an iterator passed to a client.
2. To enrich the iterator with more capabilities is to strengthen the support for the generic programming model as introduced with ANSI STL to the C++ world.

One can retrieve, replace, remove, and add elements via an iterator. One can test iterators for equality, compare ordered iterators, clone an iterator, assign iterators, and destroy them. Furthermore, an iterator can have a `const` designation which is set when created. A `const` iterator can be used for access only.

The `reverse` iterator semantics is supported. No extra interfaces are specified to support this but a `reverse` designation is set at creation time. An ordered iterator for which the `reverse` designation is set reinterprets the operations of a given iterator type to work in reverse.

Iterators and performance

To reduce network traffic, combined operations and bulk operations are offered.

- Combined operations are combinations of simple iterator operations often used in loops.
- Bulk operations support retrieving, replacing, and adding many elements within one operation.

Managed Iterators

All iterators are managed in the sense that iterators never become undefined; therefore, they do not lead to undefined behavior. Common behavior of iterators in class libraries today is that iterators become undefined when the collection content is changed. For example, if an element is added the side effect on iterators of the collection is unknown. Iterators do not “know” whether they are still pointing to the same element as before, still pointing to an element at all, or pointing “outside” the collection. One cannot even test the state. This is considered unacceptable behavior in a distributed environment.

The iterator model used in this specification is a managed iterator. Managed iterators are “robust” to modifications of the collection. A managed iterator is always in one of the following defined testable states:

- *valid* (pointing to an element of the collection)
- *invalid* (pointing to nothing; comparable to a NULL pointer)
- *in-between* (not pointing to an element, but still “remembering” enough state to be valid for most operations on it).

A valid managed iterator remains valid as long as the element it points to remains in the collection. As soon as the element is removed, the according managed iterator enters a so-called *in-between* state. The *in-between* state can be viewed as a vacuum within the collection. There is nothing the managed iterator can point to. Nevertheless,

managed iterators remember the next (and for ordered collection, also the previous) element in iteration order. It is possible to continue using the managed iterator (in a `set_to_next_element()` for example) without resetting it first. For more information, see “The Managed Iterator Model” on page 17-85.

17.2.3 Function Interfaces

The Object Collection service specifies function interfaces used to pass user-defined information to the collection implementation (either at creation time or as parameters of operations). The most important is the **Operations** interface discussed in more detail below.

Collectible Elements and Type Safety

Collections are foundation classes used in a broad range of applications. They have to be able to collect elements of arbitrary type and support keys of arbitrary type. Instances of collections are usually homogenous collections in the sense that all elements have the same element type.

Because there is no template support in CORBA IDL today, the requirement “collecting elements of arbitrary type” is met by defining the element type and the key type as a CORBA `any`. In doing so, compile time type checking for element and key type is impossible.

As collections are often used as homogenous collections, dynamic type checking is enabled by passing relevant information to the collection at creation time. This is done by specialization of the function interface **Operations**. This interface defines attributes `element_type` and `key_type` as well as defines operations `check_element_type()` and `check_key_type()` which have to be implemented by the user. Implementations may range from “no type checking at all,” “type code match,” “checking an interface to be supported,” up to “checking constraints in addition to a simple type code checking.” Using the **Operations** interface allows user-defined customization of the dynamic type checking.

Collectible Elements and the Operations Interface

The function interface **Operations** is used to pass a number of other user-defined element type specific information to the collection implementation.

The type checking of relevant information is one sample.

Depending on the properties represented by a collection interface, a respective implementation relies on some element type specific or key type specific information passed to it. For example, one has to pass the information “element comparison” to implement a **SortedSet** or “key equality” to guarantee uniqueness of keys in a **KeySet**. The **Operations** interface is used to pass this information.

The third use of this interface is to pass element or key type specific information that the different categories of implementations rely on. For example, tree-like implementations for a `KeySet` rely on the “key comparison” information and hashing based implementations rely on the information how to hash key values. This information is passed via the `Operations` interface.

A user has to customize the `Operations` interface and to implement the appropriate operations dependent on the collection interface to be used. An instance of the specialized `Operations` interface is passed at collection creation time to the collection implementation.

Collectible Elements of Key Collections

Key collections offer associative access to collection elements via a key. A key is computed from the element value and is user-defined element type specific information to be passed to a collection. The `Operations` interface has an operation `key()` which returns the user-defined key of a given element.

For a specific element type, a user has to implement the element type specific `key()` operation in an interface derived from `Operations`. The key type is a CORBA `any`. Again this is designed to accommodate generality. Computable keys reflect the data base view on elements of key collections as “keyed elements” where a key is a component of a tuple or is “composed” from several components of a tuple.

17.2.4 List of Interfaces Defined

The Object Collection service offers the following interfaces:

Abstract interfaces representing collection properties and their combinations

- `Collection`
- `OrderedCollection`
- `KeyCollection`
- `EqualityCollection`
- `SortedCollection`
- `SequentialCollection`
- `EqualitySequentialCollection`
- `EqualityKeyCollection`
- `KeySortedCollection`
- `EqualitySortedCollection`
- `EqualityKeySortedCollection`

Concrete collections and their factories

- CollectionFactory, CollectionFactories
- KeySet, KeySetFactory
- KeyBag, KeyBagFactory
- Map, MapFactory
- Relation, RelationFactory
- Set, SetFactory
- Bag, BagFactory
- KeySortedSet, KeySortedSetFactory
- KeySortedBag, KeySortedBagFactory
- SortedMap, SortedMapFactory
- SortedRelation, SortedRelationFactory
- SortedSet, SortedSetFactory
- SortedBag, SortedBagFactory
- Sequence, SequenceFactory
- EqualitySequence, EqualitySequenceFactory
- Heap, HeapFactory

Restricted access collections and their factories

- RestrictedAccessCollection, RACollectionFactory
- Stack, StackFactory
- Queue, QueueFactory
- Deque, DequeFactory
- PriorityQueue, PriorityFactory

Iterator interfaces

- Iterator
- OrderedIterator
- SequentialIterator
- SortedIterator
- KeyIterator
- EqualityIterator
- EqualityKeyIterator

- KeySortedIterator
- EqualitySortedIterator
- EqualitySequentialIterator
- EqualityKeySortedIterator

Function interfaces

- Operations
- Command
- Comparator

17.3 Combined Collections

The overview introduced *properties* and listed the meaningful combinations of these properties that result in consistently defined collection interfaces forming a differentiated offering. In the following sections, the semantics of each combination will be described in more detail and demonstrated by an example.

17.3.1 Combined Collections Usage Samples

Bag, SortedBag

A Bag is an unordered collection of zero or more elements with no key. Multiple elements are supported. As element equality is supported, operations which require the capability “test of element equality” (e.g., test on containment) can be offered.

Example: The implementation of a text file compression algorithm. The algorithm finds the most frequently occurring words in sample files. During compression, the words with a high frequency are replaced by a code (for example, an escape character followed by a one character code). During re-installation of files, codes are replaced by the respective words.

Several types of collections may be used in this context. A Bag can be used during the analysis of the sample text files to collect isolated words. After the analysis phase you may ask for the number of occurrences for each word to construct a structure with the 255 words with the highest word counts. A Bag offers an operation for this, you do not have to “count by hand,” which is less efficient. To find the 255 words with the highest word count, a SortedRelation is the appropriate structure (see “Relation, SortedRelation” on page 17-13). Finally, a Map may be used to maintain a mapping of words to codes and vice versa. (See “Map, SortedMap” on page 17-12).

A SortedBag (as compared to a Bag) exposes and maintains a sorted order of the elements based on a user-defined element comparison. Maintained elements in a sorted order makes sense when printing or displaying the collection content in sorted order.

EqualitySequence

An EqualitySequence is an ordered collection of elements with no key. There is a first and a last element. Each element, except the last one, has a next element and each element, except the first one, has a previous element. As element equality is supported, all operations that rely on the capability “test on element equality” can be offered, for example, locating an element or test for containment.

Example: An application that arranges wagons to a train. The order of the wagons is important. The trailcar has to be the first wagon, the first class wagons are arranged right behind the trailcar, the restaurant has to be arranged right after the first class and before the second class wagons, and so on. To check whether the wagon has the correct capacity, you may want to ask: “How many open-plan carriages are in the train?” or “Is there a bistro in the train already?”

Heap

A Heap is an unordered collection of zero or more elements without a key. Multiple elements are supported. No element equality is supported.

Example: A “trash can” on a desktop which memorizes all objects moved to the trashcan as long as it is not emptied. Whenever you move an object to the trashcan it is added to the heap. Sometimes you move an object accidentally to the trashcan. In that case, you iterate in some order through the trashcan to find the object - not using a test on element equality. When you find it, you remove it from the trashcan. Sometimes you empty the trashcan and remove all objects from the trashcan.

KeyBag, KeySortedBag

A KeyBag is an unordered collection of zero or more elements that have a key. Multiple keys are supported. As no element equality is assumed, operations such as “test on collection equality” or “set theoretical operation” are not offered.

A KeySortedBag is sorted by key. In addition to the operations supported for a KeyBag, all operations related to ordering are offered. For example, operations exploiting the ordering such as “set_to_previous / set_to_next” and “access via position” are supported.

A license server maintaining floating licenses on a network may be implemented using a KeyBag to maintain the licenses in use. The key may be the LicenseId and additional element data may be, for example, the user who requested the license. As usual, more than one floating license is available per product; therefore, many licenses for the same product may be in use. A LicenseId may occur more than once. A user may request a license multiple times, it may also occur that the same LicenseId with the same user occurs multiple times. If a user of the product requests and receives the license, the LicenseId, together with the request data, is added to the licenses in use. If the license is released, it is deleted from the Bag of licenses in use. Sometimes you may want to ask for the number of licenses of a product in use, that is ask for the number of the licenses in use with a given LicenseId.

Access to licenses in use is via the key `LicenseId`. This sample application does not require operations such as testing two collections for equality or set theoretical operations on collections. It is not exploiting element equality; therefore, it can use a `KeyBag` instead of a `Relation` (which would force the user to define element equality).

If you want to list the licenses in use with the users holding the licenses sorted by `LicenseId`, you could make use of a `KeySortedBag` instead of a `KeyBag`.

KeySet, KeySortedSet

A `KeySet` is an unordered collection of zero or more elements that have a key. Keys must be unique. Defined element equality is not assumed; therefore, operations and semantics which require the capability “element equality test” are not offered.

A `KeySortedSet` is sorted by key. In addition to the operations supported for a `KeySet`, all operations related to ordering are offered. For example, operations exploiting the ordering, such as “`set_to_previous / set_to_next`” and “access via position” are supported.

Example: A program that keeps track of cancelled credit card numbers and the individuals to whom they are issued. Each card number occurs only once and the collection is sorted by card number. When a merchant enters a customer’s card number into the point-of-sales terminal, the collection is checked to determine whether the card number is listed in the collection of cancelled cards. If it is found, the name of the individual is shown and the merchant is given directions for contacting the card company. If the card number is not found, the transaction can proceed because the card is valid. A list of cancelled cards is printed out each month, sorted by card number, and distributed to all merchants who do not have an automatic point-of-sale terminal installed.

Map, SortedMap

A `Map` is an unordered collection of zero or more elements that have a key. Keys must be unique. As defined, element equality is assumed access via the element value and all operations which need to test on element equality, such as a test on containment for an element, test for equality, and set theoretical operations can be offered for maps.

A `SortedMap` is sorted by key. In addition to the operations supported for a `Map`, all operations related to ordering are offered. For example, operations exploiting the ordering like “`set_to_previous / set_to_next`” and “access via position” are supported.

Example: Maintaining nicknames for your mailing facility. The key is the nickname. Mailing information includes address, first name, last name, etc. Nicknames are unique; therefore, adding a nickname/ mailing information entry with a nickname that is already available should fail, if the mailing information to be added is different from the available information. If it is exactly the same information, it should just be ignored. You may define more than one nickname for the same person; therefore, the same element data may be stored with different keys. If you want to update address

information for a given nickname, use the `replace_element_with_key()` operation. To create a new nickname file from two existing files, use a union operation which assumes element equality to be defined.

Relation, SortedRelation

A Relation is an unordered collection of zero or more elements with a key. Multiple keys are supported. As defined element equality is assumed, test for equality of two collections is offered as well as the set theoretical operations.

A SortedRelation is sorted by key. In addition to the operations supported for a Relation, all operations related to ordering are offered. For example, operations that exploit ordering such as “`set_to_previous / set_to_next`” and “access via position” are supported.

A SortedRelation may be used in the text file compression algorithm mentioned previously in the Bag, Sorted Bag example to find the 255 words with the highest frequency. The key is the word count and the additional element data is the word. As words may have equal counts, multiple keys have to be supported. The ordering with respect to the key is used to find the 255 highest keys.

Set, SortedSet

A set is an unordered collection of zero or more elements without a key. Element equality is supported; therefore, operations that require the capability “test on element equality” such as intersection or union can be offered.

A SortedSet is sorted with respect to a user-defined element comparison. In addition to the operations supported for a Set, all operations related to ordering are offered. For example, operations that exploit ordering such as “`set_to_previous / set_to_next`” and “access via position” are supported.

Example: A program that creates a packing list for a box of free samples to be sent to a warehouse customer. The program searches a database of in-stock merchandise, and selects ten items at random whose price is below a threshold level. Each item is added to the set. The set does not allow an item to be added if it already is present in the collection; this ensures that a customer does not get two samples of a single product.

Sequence

A Sequence is an ordered collection of elements without a key. There is a first and a last element. Each element (except the last one) has a next element and each element (except the first one) has a previous element. No element equality is supported; therefore, multiples may occur and access to elements via the element value is not possible. Access to elements is possible via position/index.

Example: A music editor. The Sequence is used to maintain tokens representing the recognized notes. The order of the notes is obviously important for further processing of the melody. A note may occur more than once. During editing, notes are accessed by position and are removed, added, or replaced at a given position. To print the result, you may iterate over the sequence and print note by note.

A Sequence may also be used to represent how a book is constructed from diverse documents. It is obvious that ordering is important. It may be the case that a specific document is used multiple times within the same book (for example, a specific graphic). Reading the book, you may want to access a specific document by position.

17.4 *Restricted Access Collections*

17.4.1 *Restricted Access Collections Usage Samples*

Deque

A double ended queue may be considered as a sequence with restricted access. It is an ordered collection of elements without a key and no element equality. As there is no element equality, an element value may occur multiple times. There is a first and a last element. You can only add an element as first or last element and only remove the first or the last element from the Deque.

A Deque may be used in the implementation of a pattern matching algorithm where patterns are expressed as regular expressions. Such an algorithm can be described as a non-deterministic finite state machine constructed from the regular expression. The implementation of the regular-pattern matching machine may use a deque to keep track of the states under consideration. Processing a null state requires a stack-like data structure - one of two things to be done is postponed and put at the front of the not being postponed forever list. Processing the other states requires a queue-like data structure, since you do not want to examine a state for the next given character until you are finished with the current character. Combining the two characteristics results in a Deque.

PriorityQueue

A PriorityQueue may be considered as a KeySortedBag with restricted access. It is an ordered collection with zero or more elements. Multiple key values are supported. As no element equality is defined, multiple element values may occur. Access to elements is via key only and sorting is maintained by key. Accessing a PriorityQueue is restricted. You can add an element relative to the ordering relation defined for keys and remove only the first element (e.g., the one with highest priority).

PriorityQueues may be used for implementing a printer queue. A print job's priority may depend on the number of pages, time of queuing, and other characteristics. This priority is the key of the print job. When a user adds a print job it is added relative to its priority. The printer daemon always removes the job with the highest priority from the queue.

PriorityQueues also may be used as special queues in workflow management to prioritize work items.

Queue

A queue may be considered as a sequence with restricted access. It is an ordered collection of elements with no key and no element equality. There is a first and a last element. You can only add (enqueue) an element as last element and only remove (dequeue) the first element from the Queue. That is, a queue exposes FIFO behavior.

You would use a queue in tree traversal to implement a breadth first search algorithm.

Queues may be used for the implementation of all kinds of buffered communication where it is important that the receiving side handles messages in the same order as they were sent. Queues may be used in workflow management environments where queues collect messages waiting for processing.

Stack

A Stack may be considered as a sequence with restricted access. It is an ordered collection of elements with no key and no element equality. There is a first and a last element. You can only add (push) an element as last element (at the top) and only remove (pop) the last element from the Stack (from the top). That is, a Stack exposes LIFO behavior. The classical application for a stack is the simulation of a calculator with Reverse Polish Notation. The calculator engine may get an arithmetic expression. Parsing the expression operands are pushed on to the stack. When an operator is encountered, the appropriate number of operands is popped off the stack, the operation performed, and the result pushed on the stack.

A Stack also may be used in the implementation of a window manager to maintain the order in which the windows are superimposed.

17.5 The CosCollection Module

17.5.1 Interface Hierarchies

Collection Interface Hierarchies

The collection interfaces of the Collection Services are organized in two separate hierarchies, as shown in Figure 17-1 on page 17-17 and Figure 17-2 on page 17-17. The inner nodes of the hierarchy may be thought of as abstract views. They represent the basic properties and their combinations. Leaf nodes may be thought of as concrete interfaces for which implementations are provided and from which instances can be created via a collection factory. The organization of the interfaces as a hierarchy enables reuse and the polymorphic usage of the collections from typed languages such as C++.

Each abstract view is defined in terms of operations and their behavior. The most abstract view of a collection is a container without any ordering or any specific element or key properties. This view allows adding elements to and iterating over the collection.

In addition to the common collection operations, collections whose elements define equality or key equality provide operations for locating and retrieving elements by a given element or key value.

Ordered collections provide the notion of well-defined explicit positioning of elements, either by element key ordering relation or by positional element access.

Sorted collections provide no further operations, but introduce a new semantics; namely, that their elements are sorted by element or key value. These properties are combined through multiple inheritance.

The fourth property, uniqueness/multiplicity of elements and keys, is not represented by a separate abstract view for combination with other properties. This was done to reduce the complexity of the hierarchy. Instead, operations related to multiplicity are provided in the base interface from which the interface specializations with multiplicity are derived.

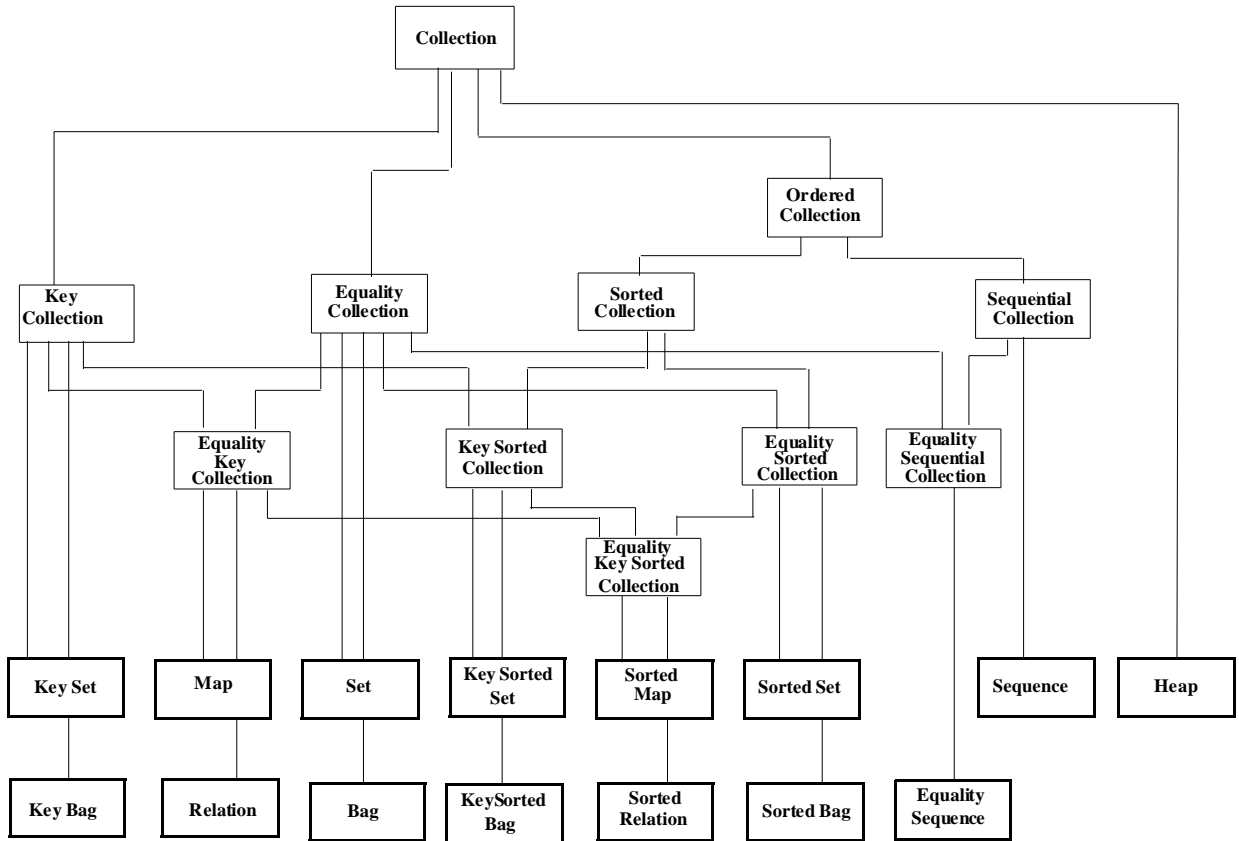


Figure 17-1 Collections Interfaces Hierarchy

The restricted access collections form their own hierarchy as shown in Figure 17-2 on page 17-17. This abstract view defines the operations that all restricted access collections have in common.

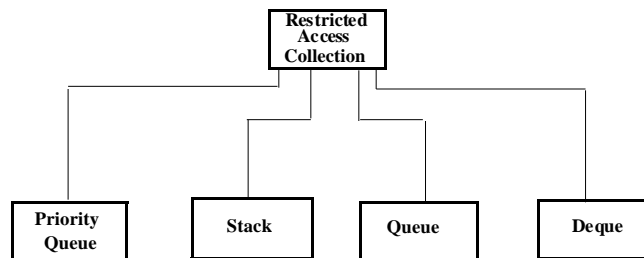


Figure 17-2 Restricted Access Collections Interface Hierarchy

Iterator Hierarchy

The iterator interface hierarchy parallels the Collection interface hierarchy shown in Figure 17-3 on page 17-18. The defined interfaces support the fine-grain processing of very large collections via an iterator only and support a generic programming model similar to what was introduced with ANSI STL to the C++ world. Concepts like constness of iterators, reverse iterators, bulk and combined operations are offered to strengthen the support for the generic programming model.

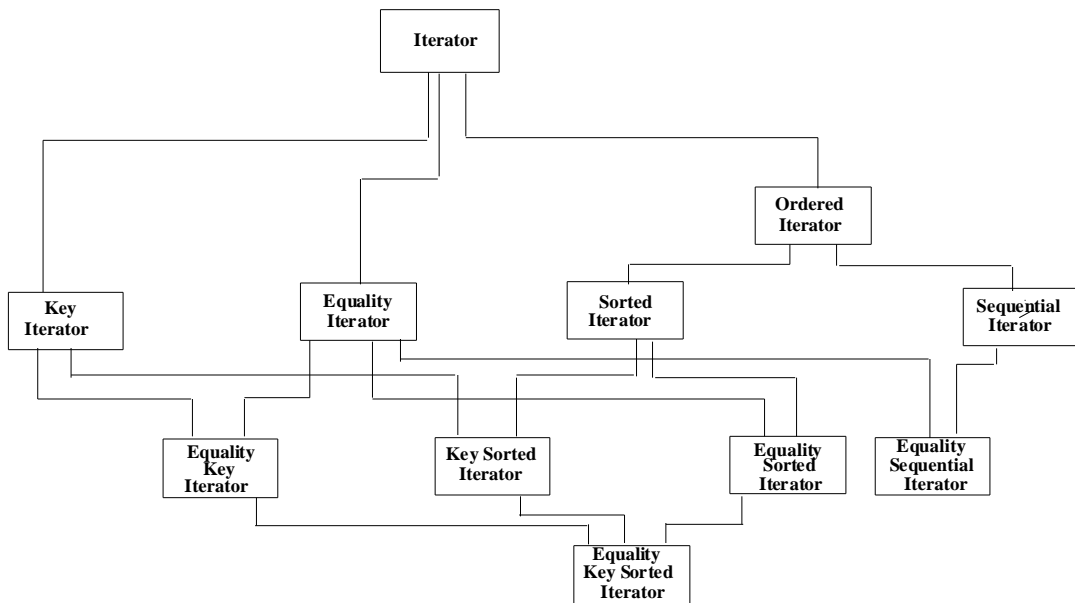


Figure 17-3 Iterator Interface Hierarchy

The top level `Iterator` interface represents a generic iterator that can be used for iteration over and manipulation of all collections independent of their type. The top level iterator allows you to add, retrieve, replace, and remove elements. There are operations to clone, assign, and test iterators for equality. There are tests on the iterator state and you can check whether an iterator is *const*, created for a given collection, or created for the same collection as another iterator.

The `OrderedIterator` interface adds those operations which are useful on collections with an explicit notion of ordering (all those collections inheriting from the `OrderedCollection` interface). An ordered iterator can be moved forward and backward, set to a position, and its position can be computed. Only ordered iterators can be used with “reverse” semantics. The `SequentialIterator` is used with sequentially ordered collections where it is possible to add elements at a user-defined position so that the iterator offers the capability to add elements relative to its position.

The `KeyIterator` and `EqualityIterator` interface add operations for positioning an iterator by key or element value. The sorted versions of these interfaces add respective backward movements and the capability to define lower and upper bounds in sorted collections.

An iterator is always created for a collection using the collection as iterator factory. Each iterator type is supported by each collection type. The Iterators and the Collections that are supported by all interfaces derived from those collections are listed in Table 17-2 on page 17-19.

Table 17-2 Iterators and Collections

	Supported by all interfaces derived from:
Iterator	Collection
OrderedIterator	OrderedCollection
SequentialIterator	SequentialCollection
EqualitySequentialIterator	EqualitySequentialCollection
KeyIterator	KeyCollection
EqualityIterator	EqualityCollection
EqualityKeyIterator	EqualityKeyCollection
SortedIterator	SortedCollection
KeySortedIterator	KeySortedCollection
EqualitySortedIterator	EqualitySortedCollection
EqualityKeySortedIterator	EqualityKeySortedCollection

17.5.2 Exceptions and Type Definitions

The following exceptions are used by the subsequently defined interfaces.

```

module CosCollection {
  // Type definitions
  typedef sequence<any> AnySequence;
  typedef string Istring;
  struct NVPair {Istring name; any value;};
  typedef sequence<NVPair> ParameterList;

  // Exceptions
  exception EmptyCollection{};

```

```
exception PositionInvalid{};
enum IteratorInvalidReason {is_invalid, is_not_for_collection,
is_const};
exception IteratorInvalid {IteratorInvalidReason why;};
exception IteratorInBetween{};
enum ElementInvalidReason {element_type_invalid,
positioning_property_invalid, element_exists};
exception ElementInvalid {ElementInvalidReason why;};
exception KeyInvalid {};
exception ParameterInvalid {unsigned long which; Istring why;};
```

AnySequence

A type definition for a sequence of values of type **any** used in bulk operations.

Istring

A type definition used as place holder for a future IDL internationalized string data type.

ParameterList

A sequence of name-value pairs of type **NVPair** and used as a generic parameter list in a generic collection creation operation.

EmptyCollection

Raised when an operation to remove an element is invoked on an empty collection.

PositionInvalid

Raised when an operation on an ordered collection passes a position out of the allowed range, that is less than 1 or greater than the number of elements in the collections.

IteratorInvalid

Raised when an operation uses an iterator pointing to nothing, that is, using an *invalid* iterator (**in_valid**) or when an operation uses an iterator which was not created for the collection (**is_not_for_collection**) or if one tries to modify a collection via an iterator that is created with **const** designation (**is_const**).

IteratorInBetween

Raised when an operation uses an iterator in a way that does not allow the state *in-between* such as all “..._at” operations.

ElementInvalid

Raised when one of the operations passes an element that is for one of several reasons invalid. It is raised

- when the element is not of the expected element type (`element_type_invalid`).
- if one tries to replace an element by another element changing the positioning property (`positioning_property_invalid`).
- when an element is added to a Map and the key already exists (`element_exists`).

KeyInvalid

Raised when one of the operations passes a key that is not of the expected type.

ParameterInvalid

Raised when a parameter passed to the generic collection creation operation of the generic `CollectionFactory` is invalid.

17.5.3 Abstract Collection Interfaces

The Collection Interface

The `Collection` interface represents the most abstract view of a collection. Operations defined in this top level interface can be supported by all collection interfaces in the hierarchy. Each concrete collection interface offers the appropriate operation semantics dependent on the collection properties. It defines operations for:

- adding elements
- removing elements
- replacing elements
- retrieving elements
- inquiring collection information
- creating iterators

```
// Collection
interface Iterator;
interface Command;

interface Collection {

// element type information
readonly attribute CORBA::TypeCode element_type;
```

```
// adding elements
boolean add_element (in any element) raises (ElementInvalid);
boolean add_element_set_iterator (in any element, in Iterator where)
raises (IteratorInvalid, ElementInvalid);
void add_all_from (in Collection collector) raises (ElementInvalid);

// removing elements
void remove_element_at (in Iterator where) raises (IteratorInvalid,
IteratorInBetween);
unsigned long remove_all ();

// replacing elements
void replace_element_at (in Iterator where, in any element)
raises (IteratorInvalid, IteratorInBetween, ElementInvalid);

// retrieving elements
boolean retrieve_element_at (in Iterator where, out any element)
raises (IteratorInvalid, IteratorInBetween);

// iterating over the collection
boolean all_elements_do (in Command what) ;

// inquiring collection information
unsigned long number_of_elements ();
boolean is_empty ();

// destroying collection
void destroy();

// creating iterators
Iterator create_iterator (in boolean read_only);
};
```

Type checking information

readonly attribute CORBA::TypeCode element_type;

Specifies the element type expected in the collection. See also “The Operations Interface” on page 17-118.

Adding elements

boolean add_element (in any element) raises (ElementInvalid);

Description

Adds an element to the collection. The exact semantics of the add operations depends on the properties of the concrete interface derived from the **Collection** that the collection is an instance of.

If the collection supports unique elements or keys and the element or key is already contained in the collection, adding is ignored. In sequential collections, the element is always added as last element. In sorted collections, the element is added at a position determined by the element or key value.

If the collection is a **Map** and contains an element with the same key as the given element, then this element has to be equal to the given element; otherwise, the exception **ElementInvalid** is raised.

Return value

Returns **true** if the element is added.

Exceptions

The element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

Side effects

All iterators keep their state.

boolean add_element_set_iterator(in any element, in Iterator where) raises (IteratorInvalid, ElementInvalid);

Description

Adds an element to the collection and sets the iterator to the added element. The exact semantics of the add operations depends on the properties of the concrete interface derived from the **Collection** that the collection is an instance of.

If the collection supports unique elements or keys and the element or key is already contained in the collection, adding is ignored and the iterator is just set to the element or key already contained. In sequential collections, the element is always added as last element. In sorted collections, the element is added at a position determined by the element or key value.

If the collection is a **Map** and contains an element with the same key as the given element, then this element has to be equal to the given element; otherwise, the exception **ElementInvalid** is raised.

Return value

Returns **true** if the element is added.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All other iterators keep their state.

```
void add_all_from (in Collection elements) raises (ElementInvalid);
```

Adds all elements of the given collection to this collection. The elements are added in the iteration order of the given collection and consistent with the semantics of the `add` operation. Essentially, this operation is a sequence of `add` operations.

Removing elements

```
void remove_element_at (in Iterator where) raises(IteratorInvalid);
```

Description

Removes the element pointed to by the given iterator. The given iterator is set to *in-between*.

Exceptions

The iterator must belong to the collection and must point to an element of the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

Iterators pointing to the removed element go *in-between*. Iterators which do not point to the removed element keep their state.

```
unsigned long void remove_all();
```

Description

Removes all elements from the collection.

Return value

Returns the number of elements removed.

Side effects

Iterators pointing to removed elements go *in-between*. All other iterators keep their state.

Replacing elements

void `replace_element_at` (in Iterator where, in any element) raises (IteratorInvalid, IteratorInBetween, ElementInvalid)

Description

Replaces the element pointed to by the iterator by the given element. The given element must have the same positioning property as the replaced element.

- For collections organized according to element properties such as ordering relation, the replace operation must not change this element property.
- For key collections, the new key must be equal to the key replaced.
- For non-key collections with element equality, the new element must be equal to the replaced element as defined by the element equality relation.

Sequential collections have a user-defined positioning property and heaps do not have positioning properties. Element values in sequences and heaps can be replaced freely.

Exceptions

The given element must not change the positioning property; otherwise, the exception `ElementInvalid` is raised.

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The iterator must belong to the collection and must point to an element of the collection; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

Retrieving elements

boolean `retrieve_element_at` (in Iterator where, out any element) raises (IteratorInvalid, IteratorInBetween);

Description

Retrieves the element pointed to by the given iterator and returns it via the output parameter `element`.

Return value

Returns `true` if an element is retrieved.

Exceptions

The given iterator must belong to the collection and must point to an element of the collection; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

Note – Whether a copy of the element is returned or the element itself depends on the element type represented by the `any`. If it is an object, a reference to the object in the collection is returned. If the element type is a non-object type, a copy of the element is returned. In case of element type object, do not manipulate the element or the key of the element in the collection in a way that changes the positioning property of the element.

Iterating over a collection

`boolean all_elements_do (in Command what);`

Description

Calls the “do_on()” operation of the given `Command` for each element of the collection until the “do_on()” operation returns `false`. The elements are visited in iteration order (see “The Command and Comparator Interface” on page 17-122).

- The “do_on()” operation must not remove elements from or add elements to the collection.
- The “do_on()” operation must not manipulate the element in the collection in a way that changes the positioning property of the element.

Return value

Returns `true` if the “do_on()” operation returns `true` for each element it is applied to.

Inquiring collection information

The collection operations do have preconditions which when violated raise exceptions. There are operations for testing those preconditions to enable the user to avoid raising exceptions.

`unsigned long number_of_elements ();`

Return value

Returns the number of elements contained in the collection.

`boolean is_empty ();`

Return value

Returns `true` if the collection is empty.

Destroying a collection

void destroy();

Description

Destroys the collection. This includes:

- removing all elements from the collection
- destroying all iterators created for this collection
- destroying the instance of **Operations** passed at creation time to the collection implementation.

Note – Removing elements in case of objects means removing object references, not destroying the collected objects.

Object references to iterators of the collections become invalid.

Creating iterators

Iterator create_iterator (in boolean read_only);

Creates and returns an iterator instance for this collection. The type of iterator that is created depends on the interface type of this collection. The following table describes the type of iterator that is created for the type of concrete collection.

Table 17-3 Collection interfaces and the iterator interfaces supported

Ordered	Collection Interfaces	Supported Iterator Interface
	Bag	EqualityIterator
yes	SortedBag	EqualitySortedIterator
yes	EqualitySequence	EqualitySequentialIterator
	Heap	Iterator
	KeyBag	KeyIterator
yes	KeySortedBag	KeySortedIterator
	KeySet	KeyIterator
yes	KeySortedSet	KeySortedIterator
	Map	EqualityKeyIterator
yes	SortedMap	EqualityKeySortedIterator
	Relation	EqualityKeyIterator
yes	Sequence	SequentialIterator

Table 17-3 Collection interfaces and the iterator interfaces supported

yes	SortedRelation	EqualityKeySortedIterator
	Set	EqualityIterator
yes	SortedSet	EqualitySortedIterator
yes	Sequence	SequentialIterator

After creation, the iterator is initialized with the state *invalid*, that is, “pointing to nothing.”

If the given parameter `read_only` is true, the iterator is created with `const` designation (i.e., a trial to modify the collection content via this iterator is rejected and raises the exception `IteratorInvalid`).

Note – Collections serve as factories for *their* iterator instances. An iterator is created in the same address space as the collection for which it is created. An iterator instance can only point to elements of the collection for which it was created.

The OrderedCollection Interface

```
interface OrderedIterator;
// OrderedCollection
interface OrderedCollection: Collection {

// removing elements
void remove_element_at_position (in unsigned long position) raises
(PositionInvalid);
void remove_first_element () raises (EmptyCollection);
void remove_last_element () raises (EmptyCollection);

// retrieving elements
boolean retrieve_element_at_position (in unsigned long position, out
any element) raises (PositionInvalid);
boolean retrieve_first_element (out any element) raises
(EmptyCollection);
boolean retrieve_last_element (out any element) raises
(EmptyCollection);

// creating iterators
OrderedIterator create_ordered_iterator(in boolean read_only, in
boolean reverse_iteration);
};
```

Ordered collections expose the ordering of elements in their interfaces. Elements can be accessed at a position and forward and backward movements are possible (i.e., ordered collection can support ordered iterators). Ordering can be implicitly defined via the ordering relationship of the elements or keys (as in sorted collections) or ordering can be user-controlled (as in sequential collections).

In addition to those inherited from the `Collection` Interface, which all ordered collections have in common, the `OrderedCollection` interface provides operations for

- removing elements,
- retrieving elements, and
- creating ordered iterators.

Removing elements

`void remove_element_at_position` (in unsigned long position) raises (`PositionInvalid`);

Description

Removes the element from the collection at a given position. The first element of the collection has position 1.

Exceptions

The value of "position" must be a valid position in the collection; otherwise, the exception `PositionInvalid` is raised. A position is valid if it is greater than or equal to 1 and less than or equal to `number_of_elements()`.

Side effects

All iterators pointing to the removed element go *in-between*. Iterators that do not point to the removed element keep their state.

`void remove_first_element` () raises (`EmptyCollection`);

Description

Removes the first element from the collection.

Exceptions

The collection must not be empty; otherwise, the exception `EmptyCollection` is raised.

Side effects

All iterators pointing to the removed element go *in-between*. Iterators that do not point to the removed element keep their state.

void remove_last_element () raises (EmptyCollection);

Description

Removes the last element from the collection.

Exceptions

The collection must not be empty; otherwise, the exception `EmptyCollection` is raised.

Side effects

All iterators pointing to the removed element go *in-between*. Iterators that do not point to the removed element keep their state.

Retrieving elements

boolean retrieve_element_at_position (in unsigned long position, out any element) raises (PositionInvalid);

Description

Retrieves the element at the given position in the collection and returns it via the output parameter `element`. Position 1 specifies the first element.

Return value

Returns `true` if an element is retrieved.

Exceptions

The value of "position" must be a valid position in the collection; otherwise, the exception `PositionInvalid` is raised.

boolean retrieve_first_element (out any element) raises (EmptyCollection);

Description

Retrieves the first element in the collection and returns it via the output parameter `element`.

Return value

Returns `true` if an element is retrieved.

Exceptions

The collection must not be empty; otherwise, the exception `EmptyCollection` is raised.

boolean `retrieve_last_element` (out any element) raises (`EmptyCollection`);

Description

Retrieves the last element in the collection and returns it via the output parameter element.

Return value

Returns `true` if an element is retrieved.

Exceptions

The collection must not be empty; otherwise, the exception `EmptyCollection` is raised.

Creating iterators

`OrderedIterator` `create_ordered_iterator` (in boolean `read_only`, in boolean `reverse_iteration`);

Description

Creates and returns an ordered iterator instance for this collection.

Which type of ordered iterator actually is created depends on the interface type of this collection. Table 17-1 on page 17-4 describes which type of ordered iterator is created for which type of concrete ordered collection.

After creation, the iterator is initialized with the state invalid, that is, “pointing to nothing.”

Exceptions

If the given parameter `read_only` is `true`, the iterator is created with `const` designation (i.e., a trial to modify the collection content via this iterator is rejected and raises the exception `IteratorInvalid`).

Side effects

If the given parameter `reverse_iteration` is `true`, the iterator is created with reverse iteration semantics. Only ordered iterators can be created with reverse semantics.

The SequentialCollection Interface

```
interface Comparator;
interface SequentialCollection: OrderedCollection {
// adding elements
void add_element_as_first (in any element) raises (ElementInvalid);
```

```

void add_element_as_first_set_iterator (in any element, in Iterator
where) raises (ElementInvalid, IteratorInvalid);

void add_element_as_last (in any element) raises (ElementInvalid);
void add_element_as_last_set_iterator (in any element, in Iterator
where) raises (ElementInvalid, IteratorInvalid);

void add_element_as_next (in any element, in Iterator where) raises
(ElementInvalid, IteratorInvalid);
void add_element_as_previous (in any element, in Iterator where)
raises (ElementInvalid, IteratorInvalid);
void add_element_at_position (in unsigned long position, in any
element) raises (PositionInvalid, ElementInvalid);
void add_element_at_position_set_iterator (in unsigned long
position, in any element, in Iterator where) raises
(PositionInvalid, ElementInvalid, IteratorInvalid);

// replacing elements
void replace_element_at_position (in unsigned long position, in any
element) raises (PositionInvalid, ElementInvalid);
void replace_first_element (in any element) raises (ElementInvalid,
EmptyCollection);
void replace_last_element (in any element) raises (ElementInvalid,
EmptyCollection);

// reordering elements
void sort (in Comparator comparison);
void reverse();
};

```

Sequential collections expose user-controlled sequential ordering. Determine where elements are added by comparing to sorted collections where the “where an element is added” is determined implicitly by the defined element or key comparison.

The `SequentialCollection` interface adds all those operations to the `OrderedCollection` interface. “The `SequentialCollection` Interface” on page 17-31 describes operators that are unique for positional element access for

- adding elements,
- replacing elements, and
- re-ordering elements.

Adding elements

```
void add_element_as_first (in any element) raises (ElementInvalid);
```

Description

Adds the element to the collection as the first element in sequential order.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All iterators keep their state.

`void add_element_as_first_set_iterator` (in any element, in Iterator where)
raises (`ElementInvalid`,`IteratorInvalid`);

Description

Adds the element to the collection as the first element in sequential order and sets the iterator to the added element.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All iterators keep their state.

`void add_element_as_last` (in any element) raises (`ElementInvalid`);

Description

Adds the element to the collection as the last element in sequential order.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All iterators keep their state.

`void add_element_as_last_set_iterator` (in any element, in Iterator where)
raises (`ElementInvalid`,`IteratorInvalid`);

Description

Adds the element to the collection as the last element in sequential order. Sets the iterator to the added element.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All other iterators keep their state.

`void add_element_as_next(in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);`

Description

Adds the element to the collection after the element pointed to by the given iterator. Sets the iterator to the added element. If the iterator is in the state *in-between*, the element is added before the iterator's "potential next" element.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The iterator must belong to the collection and be valid; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All iterators keep their state.

`void add_element_as_previous (in any element, in Iterator where) raises (IteratorInvalid, ElementInvalid);`

Description

Adds the element to the collection as the element previous to the element pointed to by the given iterator. Sets the iterator to the added element. If the iterator is in the state *in-between*, the element is added after the iterator's "potential previous" element.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The iterator must belong to the collection and must be valid; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All iterators keep their state.

`void add_element_at_position` (in unsigned long position, in any element) raises(`PositionInvalid`, `ElementInvalid`);

Description

Adds the element at the given position to the collection. If an element exists at the given position, the new element is added as the element preceding the existing element.

Exceptions

The position must be valid (i.e., greater than or equal to 1 and less than or equal to `number_of_elements() + 1`); otherwise, the exception `PositionInvalid` is raised.

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All iterators keep their state.

`void add_element_at_position_set_iterator` (in unsigned long position, in any element, in `Iterator` where) raises (`PositionInvalid`, `ElementInvalid` `IteratorInvalid`);

Description

Adds the element at the given position to the collection and sets the iterator to the added element. If an element exists at the given position, the new element is added as the element preceding the existing element.

Exceptions

The position must be valid (i.e., greater than or equal to 1 and less than or equal to `number_of_elements() + 1`); otherwise, the exception `PositionInvalid` is raised.

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All iterators keep their state.

Replacing elements

void `replace_element_at_position` (in unsigned long position, in any element) raises (`PositionInvalid`, `ElementInvalid`);

Description

Replaces the element at a given position with the given element. The given position must be valid (i.e., greater than or equal to 1 and less than or equal to `number_of_elements()`).

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

void `replace_first_element` (in any element) raises (`ElementInvalid`, `EmptyCollection`);

Description

Replaces the first element with the given element.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The collection must not be empty; otherwise, the exception `EmptyCollection` is raised.

void `replace_last_element` (in any element) raises (`ElementInvalid`, `EmptyCollection`);

Description

Replaces the last element with the given element.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The collection must not be empty; otherwise, the exception `EmptyCollection` is raised.

Re-ordering elements

void sort (in Comparator comparison);

Description

Sorts the collection so that the elements occur in ascending order. The relation of two elements is defined by the “compare” method, which a user provides when implementing an interface derived from Comparator. See “The Command and Comparator Interface” on page 17-122.

Side effects

All iterators in the state *in-between* go *invalid*.

All other iterators keep their state.

void reverse ();

Description

Orders elements in reverse order.

Side effects

All iterators in the state *in-between* go *invalid*.

All other iterators keep their state.

The SortedCollection Interface

```
interface SortedCollection: OrderedCollection{}
```

Sorted collections currently do not provide further operations but define a more specific behavior; namely, that the elements or their keys are sorted with respect to a user-defined element or key compare. See “The OrderedCollection Interface” on page 17-28.

The EqualityCollection Interface

```
interface EqualityCollection: Collection {
```

```
    // testing element containment
```

```
    boolean contains_element (in any element) raises(ElementInvalid);
```

```
    boolean contains_all_from (in Collection collector)
    raises(ElementInvalid);
```

```
    // adding elements
```

```
boolean locate_or_add_element (in any element) raises
(ElementInvalid);
boolean locate_or_add_element_set_iterator (in any element, in
Iterator where) raises (ElementInvalid, IteratorInvalid);

// locating elements
boolean locate_element (in any element, in Iterator where) raises (
ElementInvalid, IteratorInvalid);
boolean locate_next_element (in any element, in Iterator where)
raises (ElementInvalid, IteratorInvalid);
boolean locate_next_different_element (in Iterator where) raises
(IteratorInvalid, IteratorInBetween);

// removing elements
boolean remove_element (in any element) raises (ElementInvalid);
unsigned long remove_all_occurrences (in any element) raises
(ElementInvalid);

// inquiring collection information
unsigned long number_of_different_elements ();
unsigned long number_of_occurrences (in any element)
raises(ElementInvalid);
};
```

Collections whose elements define equality introduce operations which exploit the defined element equality. These operations are for finding elements by element value (and adding if not found), for testing containment of a given element, and inquiring the collection about how many elements of a given value were collected.

Testing element containment

```
boolean contains_element (in any element) raises (ElementInvalid);
```

Return value

Returns **true** if the collection contains an element equal to the given element.

Exceptions

The given elements must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

```
boolean contains_all_from (in Collection collector) raises (ElementInvalid);
```


Return value

Returns **true** if all the elements of the given collection are contained in the collection. The definition of containment is given in “contains_element.”

Exceptions

The elements in the given collection must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Adding elements

`boolean locate_or_add_element (in any element) raises (ElementInvalid);`

Description

Locates an element in the collection that is equal to the given element. If no such element is found, the element is added as described in `add`.

Return value

Returns **true** if the element was found.

Returns **false** if the element had to be added.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All iterators keep their state.

`boolean locate_or_add_element_set_iterator (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);`

Description

Locates an element in the collection that is equal to the given element. If no such element is found, the element is added as described in `add`. The iterator is set to the found or added element.

Return value

Returns **true** if the element was found.

Returns **false** if the element had to be added.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All other iterators keep their state.

Locating elements

`boolean locate_element` (in any element, in `Iterator` where) raises (`ElementInvalid`, `IteratorInvalid`);

Description

Locates an element in the collection that is equal to the given element. Sets the iterator to point to the element in the collection, or invalidates the iterator if no such element exists. If the collection contains several such elements, the first element in iteration order is located.

Return value

Returns `true` if an element is found.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All iterators keep their state.

`boolean locate_next_element` (in any element, in `Iterator` where) raises (`ElementInvalid`, `IteratorInvalid`);

Description

Locates the next element in iteration order in the collection that is equal to the given element, starting at the element next to the one pointed to by the given iterator. Sets the iterator to point to the located element. The iterator is invalidated if the end of the collection is reached and no more occurrences of the given element are left to be visited. If the iterator is in the state *in-between*, locating is started at the iterator's "potential next" element.

Return value

Returns **true** if an element was found.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The iterator must belong to the collection and must be valid; otherwise, the exception `IteratorInvalid` is raised.

`boolean locate_next_different_element (in Iterator where) raises (IteratorInvalid, IteratorInBetween);`

Description

Locates the next element in iteration order that is different from the element pointed to by the given iterator. If no more elements are left to be visited, the given iterator will no longer be valid.

Return value

Returns **true** if the next different element was found.

Exception

The iterator must belong to the collection and point to an element of the collection; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

Removing elements

`boolean remove_element (in any element) raises (ElementInvalid);`

Description

Removes an element in the collection that is equal to the given element. If no such element exists, the collection remains unchanged. In collections with non-unique elements, an arbitrary occurrence of the given element will be removed.

Return value

Returns **true** if an element was removed.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

If an element was removed, all iterators pointing to this element go *in-between*.

All other iterators keep their state.

unsigned long remove_all_occurrences (in any element) raises (ElementInvalid);

Description

Removes all elements from the collection that are equal to the given element and returns the number of elements removed.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

All iterators pointing to elements removed go *in-between*.

All iterators keep their state.

Inquiring collection information

unsigned long number_of_different_elements ();

Return value

Returns the number of different elements in the collection.

unsigned long number_of_occurrences (in any element) raises (ElementInvalid);

Return value

Returns the number of occurrences of the given element in the collection.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The KeyCollection Interface

```
interface KeyCollection: Collection {  
    // Key type information  
    readonly attribute CORBA::TypeCode key_type;  
  
    // testing containment
```

```
boolean contains_element_with_key (in any key) raises(KeyInvalid);
boolean contains_all_keys_from (in KeyCollection collector)
raises(KeyInvalid);

// adding elements
boolean locate_or_add_element_with_key (in any element)
raises(ElementInvalid);
boolean locate_or_add_element_with_key_set_iterator (in any
element, in Iterator where) raises (ElementInvalid,
IteratorInvalid);

// adding or replacing elements
boolean add_or_replace_element_with_key (in any element)
raises(ElementInvalid);
boolean add_or_replace_element_with_key_set_iterator (in any
element, in Iterator where) raises (ElementInvalid,
IteratorInvalid);

// removing elements
boolean remove_element_with_key(in any key) raises(KeyInvalid);
unsigned long remove_all_elements_with_key (in any key)
raises(KeyInvalid);

// replacing elements
boolean replace_element_with_key (in any element)
raises(ElementInvalid);
boolean replace_element_with_key_set_iterator (in any element, in
Iterator where) raises (ElementInvalid, IteratorInvalid);

// retrieving elements
boolean retrieve_element_with_key (in any key, out any element)
raises (KeyInvalid);
// computing the keys
void key (in any element, out any key) raises (ElementInvalid);
void keys (in AnySequence elements, out AnySequence keys) raises
(ElementInvalid);

// locating elements
boolean locate_element_with_key (in any key, in Iterator where)
raises (KeyInvalid, IteratorInvalid);
boolean locate_next_element_with_key (in any key, in Iterator where)
raises (KeyInvalid, IteratorInvalid);
boolean locate_next_element_with_different_key (in Iterator where)
raises (IteratorInBetween, IteratorInvalid);
```

```
// inquiring collection information
unsigned long number_of_different_keys ();
unsigned long number_of_elements_with_key (in any key)
raises(KeyInvalid);
};
```

A **KeyCollection** is a collection which offers associative access to its elements via a key. All elements of such a collection are keyed elements (i.e., they do have a key which is computed from the element value). How to compute the key from an element value is user-defined. A user specializes the **Operations** interface and implements the operation **key()** as desired (see “The Operations Interface” on page 17-118). This information is passed to the collection at creation time.

Type checking information

readonly attribute CORBA::TypeCode key_type;

Specifies the key type expected in the collection. See also “The Operations Interface” on page 17-118.

Testing containment

boolean contains_element_with_key (in any key) raises (KeyInvalid);

Return value

Returns **true** if the collection contains an element with the same key as the given key.

Exceptions

The given key has to be of the expected type; otherwise, the exception **KeyInvalid** is raised.

boolean contains_all_keys_from (in KeyCollection collector) raises(KeyInvalid);

Return value

Returns **true** if all of the keys of the given collection are contained in the collection.

Exceptions

The keys of the given collection have to be of the expected type of this collection; otherwise, the exception **KeyInvalid** is raised.

Adding elements

boolean locate_or_add_element_with_key (in any element)
raises(ElementInvalid);

Description

Locates an element with the same key as the key in the given element. If no such element exists the element is added; otherwise, the collection remains unchanged.

Return value

Returns **true** if the element is located.

Exceptions

The given element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

Side effects

All iterators keep their state.

boolean locate_or_add_element_with_key_set_iterator (in any element, in
Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates an element with the same key as the key in the given element and sets the iterator to the located elements (see **locate_element_with_key()**). If no such element exists, the element is added and the iterator is set to the element added.

Return value

Returns **true** if the element is located.

Exceptions

The given element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

The given iterator must belong to the collection; otherwise, the exception **IteratorInvalid** is raised.

Side effects

All iterators keep their state.

boolean add_or_replace_element_with_key (in any element) raises
(ElementInvalid);

Description

If the collection contains an element with the key equal to the key in the given element, the element is replaced with the given element; otherwise, the given element is added to the collection.

Return value

Returns **true** if the element was added.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All iterators keep their state.

boolean `add_or_replace_element_with_key_set_iterator` (in any element, in Iterator where) raises (`ElementInvalid`, `IteratorInvalid`);

Description

If the collection contains an element with the key equal to the key in the given element, the iterator is set to that element and the element is replaced with the given element; otherwise, the given element is added to the collection, and the iterator set to the added element.

Return value

Returns **true** if the element was added.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All iterators keep their state.

Removing elements

boolean `remove_element_with_key` (in any key) raises (`KeyInvalid`);

Description

Removes an element from the collection with the same key as the given key. If no such element exists, the collection remains unchanged. In collections with non-unique elements, an arbitrary occurrence of such an element will be removed.

Exceptions

The given key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

Side effects

If an element was removed, all iterators pointing to the element go *in-between*.

All other iterators keep their state.

unsigned long `remove_all_elements_with_key` (in any key) raises(`KeyInvalid`);

Description

Removes all elements from the collection with the same key as the given key.

Exceptions

The given key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

Side effects

Iterators pointing to elements removed go *in-between*.

All other iterators keep their state.

Replacing elements

boolean `replace_element_with_key` (in any element) raises (`ElementInvalid`);

Description

Replaces an element with the same key as the given element by the given element. If no such element exists, the collection remains unchanged. In collections with non-unique elements, an arbitrary occurrence of such an element will be replaced.

Return value

Returns `true` if an element was replaced.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

boolean `replace_element_with_key_set_iterator` (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Replaces an element with the same key as the given element by the given element, and sets the iterator to this element. If no such element exists, the iterator is invalidated and the collection remains unchanged. In collections with non-unique elements, an arbitrary occurrence of such an element will be replaced.

Return value

Returns `true` if an element was replaced.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Computing keys

void `key` (in any element, out any key) raises(`ElementInvalid`);

Description

Computes the key of the given element and returns it via the output parameter `key`.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

void `keys` (in Any Sequence elements, out Any Sequence keys) raises(`ElementInvalid`);

Description

Computes the keys of the given elements and returns them via the output parameter `keys`.

Exceptions

The given elements must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

An implementation may rely on the key operation of a user supplied interface derived from `Operations`. An instance of this interface is passed to a collection at creation time and can be used in the collection implementation.

Locating elements

boolean `locate_element_with_key` (in any key, in Iterator where) raises (`KeyInvalid`, `IteratorInvalid`);

Description

Locates an element in the collection with the same key as the given key. Sets the iterator to point to the element in the collection, or invalidates the iterator if no such element exists.

If the collection contains several such elements, the first element in iteration order is located.

Return value

Returns `true` if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

boolean `locate_next_element_with_key` (in any key, in Iterator where) raises (`KeyInvalid`, `IteratorInvalid`);

Description

Locates the next element in iteration order with the key equal to the given key, starting at the element next to the one pointed to by the given iterator. Sets the iterator to point to the element in the collection. The given iterator is invalidated if the end of the collection is reached and no more occurrences of such an element are left to be visited. If the iterator is in the *in-between* state, locating starts at the iterator's "potential next" element.

Return value

Returns `true` if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

The given iterator must belong to the collection and must be valid; otherwise, the exception `IteratorInvalid` is raised.

`boolean locate_next_element_with_different_key (in Iterator where)
raises(IteratorInvalid, IteratorInBetween)`

Description

Locates the next element in the collection in iteration order with a key different from the key of the element pointed to by the given iterator. If no such element exists, the given iterator is no longer valid.

Return value

Returns `true` if an element was found.

Exceptions

The given iterator must belong to the collection and must point to an element; otherwise, the exception `IteratorInvalid` respectively `IteratorInBetween` is raised.

Inquiring collection information

`unsigned long number_of_different_keys ();`

Return value

Returns the number of different keys in the collection.

`unsigned long number_of_elements_with_key (in any key) raises(KeyInvalid);`

Return value

Returns the number elements with key specified.

Exceptions

The key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

The EqualityKeyCollection Interface

`interface EqualityKeyCollection : EqualityCollection, KeyCollection{};`

Description

This interface combines the interfaces representing the properties “key access” and “element equality.” See “The EqualityCollection Interface” on page 17-37 and “The KeyCollection Interface” on page 17-42.

The KeySortedCollection Interface

```
interface KeySortedCollection : KeyCollection, SortedCollection {
// locating elements
boolean locate_first_element_with_key (in any key, in Iterator
where) raises (KeyInvalid, IteratorInvalid);
boolean locate_last_element_with_key(in any key, in Iterator where)
raises (KeyInvalid, IteratorInvalid);
boolean locate_previous_element_with_key (in any key, in Iterator
where) raises (KeyInvalid, IteratorInvalid);
boolean locate_previous_element_with_different_key(in Iterator
where) raises (IteratorInBetween, IteratorInvalid);
};
```

This interface combines the interfaces representing the properties “key access” and “ordering.” See “The KeyCollection Interface” on page 17-42 and “The SortedCollection Interface” on page 17-37.

Locating elements

```
boolean locate_first_element_with_key (in any key, in Iterator where)
raises (KeyInvalid, IteratorInvalid);
```

Description

Locates the first element in iteration order in the collection with the same key as the given key. Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

```
boolean locate_last_element_with_key(in any key, in Iterator where) raises
(KeyInvalid, IteratorInvalid);
```

Description

Locates the last element in iteration order in the collection with the same key as the given key. Sets the given iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

`boolean locate_previous_element_with_key` (in any key, in `Iterator` where) raises (`KeyInvalid`, `IteratorInvalid`);

Description

Locates the previous element in iteration order with a key equal to the given key, beginning at the element previous to the one specified by the given iterator and moving in reverse iteration order through the elements. Sets the iterator to the located element or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, locating begins at the iterator's "potential previous" element.

Return value

Returns `true` if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

The given iterator must belong to the collection and be valid; otherwise, the exception `IteratorInvalid` is raised.

`boolean locate_previous_element_with_different_key`(in `Iterator` where) raises (`IteratorInBetween`, `IteratorInvalid`);

Description

Locates the previous element in iteration order with a key different from the key of the element pointed to, beginning at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

The given iterator must point to an element; otherwise, the exception `IteratorInBetween` or `IteratorInvalid` is raised.

The EqualitySortedCollection Interface

This interface combines the interfaces representing the properties “element equality” and “ordering.” See “The EqualityCollection Interface” on page 17-37 and “The SortedCollection Interface” on page 17-37. It adds those methods which exploit the combination of both properties.

```
interface EqualitySortedCollection : EqualityCollection,
SortedCollection {
// locating elements
boolean locate_first_element (in any element, in Iterator where)
raises (ElementInvalid, IteratorInvalid);
boolean locate_last_element (in any element, in Iterator where)
raises (ElementInvalid, IteratorInvalid);
boolean locate_previous_element (in any element, in Iterator where)
raises
raises
(ElementInvalid, IteratorInvalid);
boolean locate_previous_different_element (in Iterator where) raises
(IteratorInvalid);
};
```

Locating elements

```
boolean locate_first_element (in any element, in Iterator where) raises
(ElementInvalid, IteratorInvalid);
```

Description

Locates the first element in iteration order in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

boolean `locate_last_element` (in any element, in `Iterator` where) raises (`ElementInvalid`, `IteratorInvalid`);

Description

Locates the last element in iteration order in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

boolean `locate_previous_element` (in any element, in `Iterator` where) raises (`ElementInvalid`, `IteratorInvalid`);

Description

Locates the previous element in iteration order that is equal to the given element, beginning at the element previous to the one specified by the given iterator and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, the search begins at the iterator's "potential previous" element.

Return value

Returns `true` if an element was found.

Exceptions

The given element must be of the expected type otherwise the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

boolean locate_previous_different_element (in Iterator where) raises (IteratorInBetween, IteratorInvalid);

Description

Locates the previous element in iteration order with a value different from the element pointed to, beginning at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The given iterator must point to an element; otherwise, the exception IteratorInBetween or IteratorInvalid is raised.

The EqualityKeySortedCollection Interface

```
interface EqualityKeySortedCollection: EqualityCollection, KeyCollection,
SortedCollection {};
```

This interface combines the interface representing the properties “element equality,” “key access,” and “ordering.”

The EqualitySequentialCollection Interface

This interface combines the interface representing the properties “element equality” and “(sequential) ordering” and offers additional operations which exploit this combination.

```
interface EqualitySequentialCollection: EqualityCollection,
SequentialCollection
{
// locating elements
boolean locate_first_element_with_value (in any element, in Iterator
where) raises (ElementInvalid, IteratorInvalid);
boolean locate_last_element_with_value (in any element, in Iterator
where) raises (ElementInvalid, IteratorInvalid);
boolean locate_previous_element_with_value (in any element, in
Iterator where) raises (ElementInvalid, IteratorInvalid);
};
```

Locating elements

boolean locate_first_element_with_value (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates the first element in iteration order in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

boolean locate_last_element_with_value (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates the last element in iteration order in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

boolean locate_previous_element_with_value (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates the previous element in iteration order that is equal to the given element, beginning at the element previous to the one specified by the given iterator and moving in reverse iteration order through the elements. Sets the iterator to the located element or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, locating begins at the iterators “potential previous” element.

Return value

Returns `true` if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The iterator must belong to the collection and be valid; otherwise, the exception `IteratorInvalid` is raised.

17.5.4 Concrete Collections Interfaces

The previously listed “abstract views” on collections combine the properties “key access,” “element equality,” and “ordering relationship” on elements. The subsequent interfaces add “uniqueness” support for “multiples.” To reduce the complexity of the hierarchy, this fourth property is not represented by a separate interface.

The KeySet Interface

```
interface KeySet: KeyCollection {};
```

The `KeySet` offers an interface representing the property “key access” with the semantics of “unique keys required.” See “The `KeyCollection` Interface” on page 17-42.

The KeyBag Interface

```
interface KeyBag: KeyCollection {};
```

The `KeyBag` offers the interface representing the property “key access” with multiple keys allowed. See “The `KeyCollection` Interface” on page 17-42.

The Map Interface

```
interface Map : EqualityKeyCollection {
  // set theoretical operations
  void difference_with (in Map collector) raises (ElementInvalid);
  void add_difference (in Map collector1, in Map collector2) raises
  (ElementInvalid);
```

```

void intersection_with (in Map collector) raises (ElementInvalid);
void add_intersection (in Map collector1, in Map collector2) raises
(ElementInvalid);
void union_with (in Map collector) raises (ElementInvalid);
void add_union (in Map collector1, in Map collector2)raises
(ElementInvalid);

// testing equality
boolean equal (in Map collector) raises (ElementInvalid);
boolean not_equal (in Map collector) raises(ElementInvalid);
};

```

The Map offers the interface representing the combination of the properties “element equality testable” and “key access” and supports the semantics “unique keys required” (which implies unique elements). See “The EqualityKeyCollection Interface” on page 17-50.

With element equality defined, a test on equality for collections of the same type is possible as well as a meaningful definition of the set theoretical operations.

Set theoretical operations

```
void difference_with (in Map collector) raises(ElementInvalid);
```

Description

Makes this collection the difference between this collection and the given collection. The difference of A and B (A minus B) is the set of elements that are contained in A but not in B.

The same operation is defined for other collections, too. The following rule applies for collections with multiple elements: If collection P contains the element X m times and collection Q contains the element X n times, the difference of P and Q contains the element X m-n times if “m > n,” and zero times if “m <= n.”

Exceptions

Elements of the given collection must have the expected type of this collection; otherwise, the exception `ElementInvalid` is raised.

Side effects

Valid iterators pointing to removed elements go *in-between*. All other iterators keep their state.

```
void add_difference (in Map collector1, in Map collector2) raises
(ElementInvalid);
```

Description

Creates the difference between the two given collections and adds the difference to this collection.

Exceptions

Elements of the given collections must be of the expected type in this collection; otherwise, the exception `ElementInvalid` is raised.

Side effects

Adding the difference takes place one by one so the semantics for `add` applies here for raised exceptions and iterator state.

`void intersection_with (in Map collector) raises (ElementInvalid);`

Description

Makes this collection the intersection of this collection and the given collection. The intersection of A and B is the set of elements that is contained in both A and B.

The same operation is defined for other collections, too. The following rule applies for collections with multiple elements: If collection P contains the element X m times and collection Q contains the element X n times, the intersection of P and Q contains the element X “MIN(m,n)” times.

Exceptions

Elements of the given collection must have the expected type of this collection; otherwise, the exception `ElementInvalid` is raised.

Side effects

Valid iterators of this collection pointing to removed elements go *in-between*.

All other iterators keep their state.

`void add_intersection (in Map collector1, in Map collector2) raises (ElementInvalid);`

Description

Creates the intersection of the two given collections and adds the intersection to this collection.

Exceptions

Elements of the given collections must have the expected type of this collection; otherwise, the exception `ElementInvalid` is raised.

Side effects

Adding the intersection takes place one by one so the semantics for `add` apply here for raised exceptions and iterator state.

`void union_with (in Map collector) raises (ElementInvalid);`

Description

Makes this collection the union of this collection and the given collection. The union of A and B are the elements that are members of A or B or both.

The same operation is defined for other collections, too. The following rule applies for collections with multiple elements: If collection P contains the element X m times and collection Q contains the element X n times, the union of P and Q contains the element X m+n times.

Exceptions

Elements of the given collection must have the expected type of this collection; otherwise, the exception `ElementInvalid` is raised.

Side effects

Adding takes place one by one so the semantics for `add` applies here for raised exceptions and iterator state.

`void add_union (in Map collector1, in Map collector2) raises (ElementInvalid);`

Description

Creates the union of the two given collections and adds the union to the collection.

Exceptions

Elements of the given collections must have the expected type of this collection; otherwise, the exception `ElementInvalid` is raised.

Side effects

Adding the intersection takes place one by one; therefore, the semantics for `add` applies here for validity of iterators and raised exceptions.

Testing equality

`boolean equal (in Map collector) raises(ElementInvalid);`

Return value

Returns `true` if the given collection is equal to the collection.

This operation is defined for other collections, too. Two collections are equal if the number of elements in each collection is the same and if the following conditions (depending on the collection properties) are fulfilled.

- **Collections with unique elements:** If the collections have unique elements, any element that occurs in one collection must occur in the other collections, too.
- **Collections with non-unique elements:** If an element has *n* occurrences in one collection, it must have exactly *n* occurrences in the other collection.
- **Sequential collections:** They are sequential collections if they are lexicographically equal based on element equality defined for the elements of the sequential collection.

Exceptions

Elements of the given collections must have the expected type of this collection; otherwise, the exception `ElementInvalid` is raised.

`boolean not_equal (in Map collector) raises (ElementInvalid);`

Return value

Returns `true` if the given collection is not equal to this collection.

The Relation Interface

```
interface Relation : EqualityKeyCollection {
// equal, not_equal, and the set-theoretical operations as defined
for Map
};
```

The `Relation` interface offers the interface representing the combination of the properties “element equality testable” and “key access” and supports the semantics “multiple elements allowed.” See “The `EqualityKeyCollection` Interface” on page 17-50. For a definition of the set-theoretical operation see “The `Map` Interface” on page 17-57.

The Set Interface

```
interface Set : EqualityCollection {
// equal, not_equal, and the set theoretical operations as defined
for Map
};
```

The `Set` offers the interface representing the property “element equality testable” with the semantics of “unique elements required.” See “The `EqualityCollection` Interface” on page 17-37.

The Bag Interface

```
interface Bag : EqualityCollection {  
    // equal, not_equal, and the set theoretical operations as defined  
    // for Map  
};
```

The **Bag** offers the interface representing the property “element equality testable” with the semantics of “multiples allowed.” See “The EqualityCollection Interface” on page 17-37.

The KeySortedSet Interface

```
interface KeySortedSet : KeySortedCollection {  
    long compare (in KeySortedSet collector, in Comparator comparison);  
};
```

The **KeySortedSet** offers the sorted variant of **KeySet**. See “The KeySortedCollection Interface” on page 17-51.

The sorted variant of **KeySet** introduces a new operation **compare** which can be supported only when there is “ordering.” This operation takes an instance of a user-defined **Comparator** as given parameter. See “The Command and Comparator Interface” on page 17-122.

The **Comparator** defines the comparison to be used for the elements in the context of this **compare** operation. Comparison on two **KeySortedSets** then is a lexicographical comparison based on this element comparison.

long compare (in KeySortedSet collector, in Comparator comparison) raises (**ElementInvalid**);

Description

Compares this collection with the given collection. Comparison yields:

- <0 if this collection is less than the given collection,
- 0 if the collection is equal to the given collection, and
- >0 if the collection is greater than the given collection.

Comparison is defined by the first pair of corresponding elements, in both collections, that are not equal. If such a pair exists, the collection with the greater element is the greater one. If such a pair does not exist, the collection with more elements is the greater one.

The “compare” operation of the user’s comparator (interface derived from **Comparator**) must return a result according to the following rules:

- >0 if (element1 > element2)
- 0 if (element1 = element2)


```
<0    if (element1 < element2)
```

Return value

Returns the result of the collection comparison.

The KeySortedBag Interface

```
interface KeySortedBag : KeySortedCollection {
long compare (in KeySortedBag collector, in Comparator comparison);
};
```

The `KeySortedBag` is the sorted variant of the `KeyBag`. See “The `KeySortedCollection` Interface” on page 17-51. The additional operation `compare` is offered. See “The `KeySortedSet` Interface” on page 17-62.

The SortedMap Interface

```
interface SortedMap : EqualityKeySortedCollection {
// equal, not_equal, and the set theoretical operations
long compare (in SortedMap collector, in Comparator comparison);
};
```

The `SortedMap` interface is the sorted variant of a `Map`. See “The `EqualityKeySortedCollection` Interface” on page 17-55. The additional operation `compare` is offered. See “The `KeySortedSet` Interface” on page 17-62.

The SortedRelation Interface

```
interface SortedRelation : EqualityKeySortedCollection {
// equal, not_equal, and the set theoretical operations
long compare (in SortedRelation collector, in Comparator
comparison);
};
```

The `SortedRelation` interface is the sorted variant of a `Relation`. See “The `EqualitySortedCollection` Interface” on page 17-53. The additional operation `compare` is offered. See “The `KeySortedSet` Interface” on page 17-62.

The SortedSet Interface

```
interface SortedSet : EqualitySortedCollection {
// equal, not_equal, and the set theoretical operations
long compare (in SortedSet collector, in Comparator comparison);
};
```

The `SortedSet` interface is the sorted variant of a `Set`. The additional operation `compare` is offered. See “The `KeySortedSet` Interface” on page 17-62.

The SortedBag Interface

```
interface SortedBag: EqualitySortedCollection {  
    // equal, not_equal, and the set theoretical operations  
    long compare (in SortedBag collector, in Comparator comparison);  
};
```

The **SortedBag** interface is the sorted variant of a Bag. See “The EqualitySortedCollection Interface” on page 17-53. The additional operation **compare** is offered. See “The KeySortedSet Interface” on page 17-62.

The Sequence Interface

```
interface Sequence : SequentialCollection {  
    // Comparison  
    long compare (in Sequence collector, in Comparator comparison);  
};
```

The **Sequence** supports the interface representing the property “sequential ordering.” This property enables the definition of comparison on two Sequences; therefore, the operation **compare** is offered. See “The SequentialCollection Interface” on page 17-31.

The EqualitySequence Interface

```
interface EqualitySequence : EqualitySequentialCollection {  
    // test on equality  
    boolean equal (in EqualitySequence collector);  
    boolean not_equal (in EqualitySequence collector);  
    // comparison  
    long compare (in EqualitySequence collector, in Comparator  
comparison);  
};
```

The **EqualitySequence** supports the combination of the properties “sequential ordering” and “element equality testable.” See “The EqualitySequentialCollection Interface” on page 17-55. This allows the operations **equal**, **not_equal** and **compare**.

The Heap Interface

```
interface Heap : Collection {};
```

The **Heap** does not support any property at all. It just delivers the basic **Collection** interface. See “The Collection Interface” on page 17-21.

17.5.5 *Restricted Access Collection Interfaces*

Common data structures, such as a stack, may restrict access to the elements of a collection. The restricted access collections support these data structures. `Stack`, `Queue`, and `Dequeue` are essentially restricted access Sequences. `PriorityQueue` is essentially a restricted access `KeySortedBag`. For convenience, these interfaces offer the commonly used operation names such as `push`, `pop`, etc. rather than `add_element`, `remove_element_at`. Although the restricted access collections form their own hierarchy, the naming was formed in a way that allows mixing-in with the hierarchy of the combined property collections.

This may be useful to support several views on the same instance of a collection. For example, a “user view” to a job queue with restricted access of a `PriorityQueue` and an “administrator view” to the same print job queue with the full capabilities of a `KeySortedBag`.

17.5.6 *Abstract RestrictedAccessCollection Interface*

The RestrictedAccessCollection Interface

```
// Restricted Access Collections
interface RestrictedAccessCollection {

    // getting information on collection state
    boolean unfilled ();
    unsigned long size ();

    // removing elements
    void purge ();
};
```

`boolean unfilled ();`

Return value

Returns `true` if the collection is empty.

`unsigned long size ();`

Return value

Returns the number of elements in the collection.

`void purge ();`

Description

Removes all elements from the collection. See “The Collection Interface” on page 17-21.

17.5.7 Concrete Restricted Access Collection Interfaces

The Queue Interface

```
interface Queue : RestrictedAccessCollection {  
  
    // adding elements  
    void enqueue (in any element) raises (ElementInvalid);  
  
    // removing elements  
    void dequeue () raises (EmptyCollection);  
    boolean element_dequeue (out any element) raises (EmptyCollection);  
};
```

A **Queue** may be considered as a restricted access **Sequence**. Elements are added at the end of the queue only and removed from the beginning of the queue. FIFO behavior is delivered.

Adding elements

```
void enqueue (in any element) raises (ElementInvalid);
```

Description

Adds the element as last element to the Queue.

Exceptions

The given element must be the expected type; otherwise, the exception **ElementInvalid** is raised.

Removing elements

```
void dequeue () raises (EmptyCollection);
```

Description

Removes the first element from the queue.

Exceptions

The queue must not be empty; otherwise, the exception **EmptyCollection** is raised.

boolean element_dequeue(out any element) raises (EmptyCollection);

Description

Retrieves the first element in the queue, returns it via the output parameter element, and removes it from the queue.

Return value

Returns true if an element was retrieved.

Exceptions

The queue must not be empty; otherwise, the exception EmptyCollection is raised.

The Dequeue Interface

```
interface Deque : RestrictedAccessCollection {

    // adding elements
    void enqueue_as_first (in any element) raises (ElementInvalid);
    void enqueue_as_last (in any element) raises (ElementInvalid);

    // removing elements
    void dequeue_first () raises (EmptyCollection);
    boolean element_dequeue_first (out any element) raises
    (EmptyCollection);
    void dequeue_last () raises (EmptyCollection);
    boolean element_dequeue_last (out any element) raises
    (EmptyCollection);
};
```

The Dequeue may be considered as a restricted access Sequence. Adding and removing elements is only allowed at both ends of the double-ended queue. The semantics of the Dequeue operation is comparable to the operations described for the Queue interface. See “The Queue Interface” on page 17-66.

The Stack Interface

```
interface Stack: RestrictedAccessCollection {

    // adding elements
    void push (in any element) raises (ElementInvalid);

    // removing and retrieving elements
    void pop () raises (EmptyCollection);
    boolean element_pop (out any element) raises (EmptyCollection);
```

```
boolean top (out any element) raises (EmptyCollection);  
};
```

The **Stack** may be considered as a restricted access **Sequence**. Adding and removing elements is only allowed at the end of the queue. LIFO behavior is delivered.

Adding elements

```
void push (in any element) raises (ElementInvalid);
```

Description

Adds the element to the stack as the last element.

Exceptions

The given element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

Removing elements

```
void pop () raises (EmptyCollection);
```

Description

Removes the last element from the stack.

Exceptions

The stack must not be empty; otherwise, the exception **EmptyCollection** is raised.

```
boolean element_pop (out any element) raises (EmptyCollection);
```

Description

Retrieves the last element from the stack and returns it via the output parameter **element** and removes it from the stack.

Return value

Returns **true** if an element is retrieved.

Exceptions

The stack must not be empty; otherwise, the exception **EmptyCollection** is raised.

Retrieving elements

```
boolean top (out any element) raises (EmptyCollection);
```

Description

Retrieves the last element from the stack and returns it via the output parameter `element`.

Return value

Returns `true` if an element is retrieved.

Exceptions

The stack must not be empty; otherwise, the exception `EmptyCollection` is raised.

The PriorityQueue Interface

```
interface PriorityQueue: RestrictedAccessCollection {
  // adding elements
  void enqueue (in any element) raises (ElementInvalid);

  // removing elements
  void dequeue () raises (EmptyCollection);
  boolean element_dequeue (out any element) raises (EmptyCollection);
};
```

The `PriorityQueue` may be considered as a restricted access `KeySortedBag`. The interface is identical to that of an ordinary `Queue`, with a slightly different semantics for adding elements.

Adding elements

```
void enqueue (in any element) raises (ElementInvalid);
```

Description

Adds the element to the priority queue at a position determined by the ordering relation provided for the key type.

Exceptions

The `Element` must be the expected type; otherwise, the exception `ElementInvalid` is raised.

Removing elements

```
void dequeue () raises (EmptyCollection);
```

Description

Removes the first element from the collection.

Exceptions

The priority queue must not be empty; otherwise, the exception `EmptyCollection` is raised.

`boolean element_dequeue (out any element)` raises (`EmptyCollection`);

Description

Retrieves the first element in the priority queue and returns it via the output parameter `element`, removes it from the priority queue, and returns the copy to the user.

Return value

Returns `true` if an element is retrieved.

Exceptions

The priority queue must not be empty; otherwise, the exception `EmptyCollection` is raised.

17.5.8 Collection Factory Interfaces

There is one collection factory defined per concrete collection interface which offers a typed operation for the creation of collection instances supporting the respective collection interface as its principal interface.

The information passed to a collection implementation at creation time is:

1. Element type specific information required to implement the correct semantics. For example, to implement `Set` semantics one has to pass the information how to test the equality of elements.
2. Element type specific information that can be exploited by the specific implementation variants. For example, a hashtable implementation of a `Set` would exploit the information how the hash value for collected elements is computed.

This element type specific information is passed to the collection implementation via an instance of a user-defined specialization of the `Operations` interface.

3. An implementation hint about the expected number of elements collected. An array based implementation may use this hint as an estimate for the initial size of the implementation array.

To enable the support for, and a user-controlled selection of implementation variants, there is a generic extensible factory defined. This allows for registration of implementation variants and their user-defined selection at creation time.

The CollectionFactory and CollectionFactories Interfaces

```
interface Operations;
interface CollectionFactory {
Collection generic_create (in ParameterList parameters) raises
(ParameterInvalid);
};
```

CollectionFactory defines a generic collection creation operation which enables extensibility and supports the creation of collection instances with the very basic capabilities.

Collection generic_create (in ParameterList parameters) raises (ParameterInvalid);

Returns a new collection instance which supports the interface **Collection** and does not offer any type checking. A sequence of name-value pairs is passed to the create operation. The only processed parameter in the given list is “expected_size,” of type “unsigned long.”

This parameter is optional and gives an estimate of the expected number of elements to be collected.

Note – All collection interface specific factories defined in this specification inherit from the interface **CollectionFactory** to enable their registration with the extensible generic **CollectionFactories** factory specified below.

```
interface CollectionFactories : CollectionFactory {
boolean add_factory (in Istring collection_interface, in Istring
impl_category, in Istring impl_interface, in CollectionFactory
factory);
boolean remove_factory (in Istring collection_interface, in Istring
impl_category, in Istring impl_interface);
};
```

The interface **CollectionFactories** specifies a generic extensible collection creation capability. It maintains a registry of collection factories. The create operation of the **CollectionFactories** does not create collection instances itself, but passes the requests through to an appropriate factory registered with it and passes the result through to the caller. Note that only factories derived from **CollectionFactory** can be registered with **CollectionFactories**.

boolean add_factory (in Istring collection_interface, in Istring impl_category, in Istring impl_interface, in CollectionFactory factory);

Registers the factory with three pieces of information:

1. `collection_interface` specifies the collection interface (directly or indirectly derived from `Collection`) supported by the given factory. That is, a collection instance created via the given factory has to support the given interface `collection_interface`.
2. `impl_interface` specifies the implementation interface (directly or indirectly derived from the interface specified in `collection_interface`) supported by the registered factory. Collection instances created via this factory are instances of this implementation interface.
3. `impl_category` specifies a named group of equivalent implementation interfaces to which the implementation interface supported by the registered factory belongs. A group of implementation interfaces of a given collection interface are equivalent if they:
 - rely on the same user-defined implementation support, that is, the same operations defined in the user-defined specialization of the `Operations` interface.
 - are based on essentially the same data structure and deliver comparable performance characteristics.

The following table lists *examples* of implementation categories (representing common implementations).

Table 17-4 Implementation Category Examples

Implementation Category	Description
ArrayBased	User-defined implementation specific operations do not have to be defined. The basic data structure used is an array.
LinkedListBased	User-defined implementation specific operations do not have to be defined. The basic data structure used is a simple linked list.
SkipListsBased	A compare operation has to be defined for the key element values that depend on whether or not the collection is a <code>KeyCollection</code> derived from <code>KeyCollection</code> . The basic data structure are skip lists.
HashTableBased	A hash-function has to be defined for key element values that depend on whether or not the interface implemented is a <code>KeyCollection</code> derived from <code>KeyCollection</code> . The basic data structure is a hashtable based on the hash-function defined.
AVLTreeBased	A compare operation has to be defined for the key element values that depend on whether or not the collection is a <code>KeyCollection</code> derived from <code>KeyCollection</code> . The basic data structure is an AVL tree.
BStarTreeBased	A compare operation has to be defined for key values. The basic data structure is a B*tree.

The operation does not check the validity of the registration request in the sense that it checks any of the restrictions on the parameters described above, but just registers the given information with the factory. It is the responsibility of the user to ensure that the registration is valid.

The entry is added if there is not already a factory registered with the same three pieces of information; otherwise, the registration is ignored. Returns true if the factory is added.

boolean remove_factory (in Istring collection_interface, in Istring impl_category, in Istring impl_interface)

Description

Removes the factory registered with the given three pieces of information from the registry.

Return value

Returns true if an entry with that name exists and is removed.

create (ParameterList parameters) raises (ParameterInvalid)

The create operation of the **CollectionFactories** interface does not create instances itself, but passes through creation requests to factories registered with it. The factory is passed a sequence of name-value pairs of which the only mandatory one is "collection_interface" of type Istring.

"collection_interface" of type Istring

A string which specifies the name of the collection interface (directly or indirectly derived from **Collection**) the collection instance created has to support.

This name-value pair corresponds to the collection_interface parameter of the add_factory() operation.

The following name-value pairs are optional:

"impl_category" of type Istring

A string which denotes the desired implementation category. This name-value pair corresponds to the impl_category parameter of the add_factory() operation.

"impl_interface" of type Istring

A string which specifies a desired implementation interface. This name-value pair corresponds to the impl_interface parameter of the add_factory() operation.

If one or both of these name-value pairs are given, it is searched for a best matching entry in the factory registry and the request is passed through to the respective factory. “Best matching” means that if an implementation interface is given, it is searched for a factory supporting an exact matching implementation interface first. If no factory supporting the desired implementation interface is registered, it is searched for a factory supporting an implementation interface of the same implementation category.

If none of the two name-value pairs are given, the request is passed to a factory registered as default factory for a given “collection_interface.” For each concrete collection interface specified in this specification, there is one collection specific factory defined which serves as default factory and is assumed to be registered with CollectionFactories.

There must be a name-value pair with name “collection_interface” given and a factory must be registered for “collection_interface;” otherwise, the exception `ParameterInvalid` is raised.

If a desired implementation interface and/or an implementation category is given, a factory with matching characteristics must be registered; otherwise, the exception `ParameterInvalid` is raised.

For factories specified for each concrete collection interface in this specification, the following additional name-value pairs are relevant:

“operations” of type <code>Operations</code>	An instance of a user-defined specialization of <code>Operations</code> which specifies element- and/or key-type specific operations.
“expected_size” of type unsigned long	is an unsigned long and gives an estimate about the expected number of elements to be collected.

Those parameters are not processed by the create operation of `CollectionFactories` itself, but just passed through to a registered factory.

The RACollectionFactory and RACollectionFactories Interfaces

```
interface RACollectionFactory {
    RestrictedAccessCollection generic_create (in ParameterList
    parameters) raises (ParameterInvalid);
};
```

The interface `RACollectionFactory` corresponds to the interface `CollectionFactory`, but defines an abstract interface.

```
interface RACollectionFactories : RACollectionFactory {
```

```

boolean add_factory (in Istring collection_interface, in Istring
impl_category, in Istring impl_interface, in RACollectionFactory
factory);

boolean remove_factory (in Istring collection_interface, in Istring
impl_category, in Istring impl_interface);
};

```

The interface `RACollectionFactories` corresponds to the `CollectionFactories` interface. It enables the registration and deregistration of collections with restricted access as well as control over the implementation choice for a given restricted access collection at creation time.

The KeySetFactory Interface

```

interface KeySetFactory : CollectionFactory {
KeySet create (in Operations ops, in unsigned long expected_size);
};

```

`KeySet create (in Operations ops, in unsigned long expected_size);`

Creates and returns an instance of `KeySet`. The given instance of `Operations` passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-5 Required element and key-type specific user-defined information for `KeySetFactory`. []- implied by `key_compare`.

KeySet						
equal	compare	hash	key	key_equal	key_compare	key_hash
			x	[x]	x	

The KeyBagFactory Interface

```

interface KeyBagFactory : CollectionFactory {
KeyBag create (in Operations ops, in unsigned long expected_size);
};

```

`KeyBag create (in Operations ops, in unsigned long expected_size);`

Creates and returns an instance of **KeyBag**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-6 Required element and key-type specific user-defined information for **KeyBagFactory**. []- implied by **key_compare**.

KeyBag						
equal	compare	hash	key	key_equal	key_compare	key_hash
			x	[x]	x	

The MapFactory Interface

```
interface MapFactory : CollectionFactory {
Map create (in Operations ops, in unsigned long expected_size);
};
```

Map create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **Map**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-7 Required element and key-type specific user-defined information for **MapFactory**. []- implied by **key_compare**.

Map						
equal	compare	hash	key	key_equal	key_compare	key_hash
x			x	[x]	x	

The RelationFactory Interface

```
interface RelationFactory : CollectionFactory {
Relation create (in Operations ops, in unsigned long expected_size);
};
```

Relation create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **Relation**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-8 Required element and key-type specific user-defined information for **RelationFactory**[- implied by **key_compare**.

Relation						
equal	compare	hash	key	key_equal	key_compare	key_hash
x			x	[x]	x	

The SetFactory Interface

```
interface SetFactory : CollectionFactory {
Set create (in Operations ops, in unsigned long expected_size);
};
```

Set create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **Set**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation.

The following table defines the requirements for the element key operations to be implemented.

Table 17-9 Required element and key-type specific user-defined information for **SetFactory**[- implied by **compare**.

Set						
equal	compare	hash	key	key_equal	key_compare	key_hash
[x]	x					

The BagFactory Interface

```
interface BagFactory {
Bag create (in Operations ops, in unsigned long expected_size);
};
```

Bag create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **Bag**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-10 Required element and key-type specific user-defined information for **BagFactory**.[]- implied by **compare**.

Bag						
equal	compare	hash	key	key_equal	key_compare	key_hash
[x]	x					

The KeySortedSetFactory Interface

```
interface KeySortedSetFactory {
    KeySortedSet create (in Operations ops, in unsigned long
        expected_size);
};
```

KeySortedSet create (in **Operations** ops, in unsigned long expected_size)

Creates and returns an instance of **KeySortedSet**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-11 Required element and key-type specific user-defined information for **KeySortedSetFactory**.[]- implied by **key_compare**.

KeySortedSet						
equal	compare	hash	key	key_equal	key_compare	key_hash
			x	[x]	x	

The KeySortedBagFactory Interface

```
interface KeySortedBagFactory : CollectionFactory {
    KeySortedBag create (in Operations ops, in unsigned long
        expected_size);
};
```

KeySortedBag create (in **Operations** ops, in unsigned long expected_size);

Creates and returns an instance of **KeySortedBag**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation.

The following table defines the requirements for the element key operations to be implemented.

Table 17-12 Required element and key-type specific user-defined information for KeySortedBagFactory.[]- implied by key_compare.

KeySortedBag						
equal	compare	hash	key	key_equal	key_compare	key_hash
			x	[x]	x	

The SortedMapFactory Interface

```
interface SortedMapFactory : CollectionFactory {
SortedMap create (in Operations ops, in unsigned long
expected_size);
};
```

SortedMap create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **SortedMap**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-13 Required element and key-type specific user-defined information for SortedMapFactory.[]- implied by key_compare.

SortedMap						
equal	compare	hash	key	key_equal	key_compare	key_hash
x			x	[x]	x	

The SortedRelationFactory Interface

```
interface SortedRelationFactory : CollectionFactory {
SortedRelation create (in Operations ops, in unsigned long
expected_size);
};
```

SortedRelation create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **SortedRelation**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-14 Required element and key-type specific user-defined information for **SortedRelationFactory**.[]- implied by **key_compare**.

SortedRelation						
equal	compare	hash	key	key_equal	key_compare	key_hash
x			x	[x]	x	

The SortedSetFactory Interface

```
interface SortedSetFactory : CollectionFactory {
SortedSet create (in Operations ops, in unsigned long
expected_size);
};
```

SortedSet create (in **Operations** ops, in unsigned long expected_size);

Creates and returns an instance of **SortedSet**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-15 Required element and key-type specific user-defined information for **SortedSetFactory**. []- implied by **compare**.

SortedSet						
equal	compare	hash	key	key_equal	key_compare	key_hash
[x]	x					

The SortedBagFactory Interface

```
interface SortedBagFactory {
SortedBag create (in Operations ops, in unsigned long
expected_size);
};
```

SortedBag create (in **Operations** ops, in unsigned long expected_size);

Creates and returns an instance of **SortedBag**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation.

The following table defines the requirements for the element key operations to be implemented.

Table 17-16 Required element and key-type specific user-defined information for SortedBagFactory. []- implied by compare.

SortedBag						
equal	compare	hash	key	key_equal	key_compare	key_hash
[x]	x					

The SequenceFactory Interface

```
interface SequenceFactory : CollectionFactory {
Sequence create (in Operations ops, in unsigned long expected_size);
};
```

Sequence create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **Sequence**. No requirements on the element respectively key operations to be implemented is specified for a **Sequence**. Nevertheless one still has to pass an instance of **Operations** as type checking information has to be passed to the collection implementation.

Note – As the **Sequence** interface represents array as well as linked list implementation of sequentially ordered collections, a service provider should offer at least two implementations to meet the performance requirements of the two most common access patterns. That is, a service provider should offer an array based implementation and a linked list based implementation.

The EqualitySequence Factory Interface

```
interface EqualitySequenceFactory : CollectionFactory {
EqualitySequence create (in Operations ops, in unsigned long
expected_size);
};
```

EqualitySequence create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of `EqualitySequence`. The given instance of `Operations` passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-17 Required element and key-type specific user-defined information for `EqualitySequenceFactory`.

Equality Sequence						
equal	compare	hash	key	key_equal	key_compare	key_hash
x						

Note – As the `EqualitySequence` interface represents array as well as linked list implementations of sequentially ordered collections, a service provider should offer at least two implementations to meet the performance requirements of the two most common access patterns. That is, a service provider should offer an array based implementation and a linked list based implementation.

The HeapFactory Interface

```
interface HeapFactory : CollectionFactory {
Heap create (in Operations ops, in unsigned long expected_size);
};
```

Heap create (in Operations ops, in unsigned long expected_size);

Returns an instance of a `Heap`. No requirements for the element key operations to be implemented is specified for a `Heap`. Nevertheless, one still has to pass an instance of `Operations` as type checking information must pass to the collection implementation.

The QueueFactory Interface

```
interface QueueFactory : RACollectionFactory {
Queue create (in Operations ops, in unsigned long expected_size);
};
```

Queue create (in Operations ops, in unsigned long expected_size);

Returns an instance of a `Queue`. No requirements for the element key operations to be implemented is specified for a `Queue`. Nevertheless, one still has to pass an instance of `Operations` as type checking information must pass to the collection implementation.

The StackFactory Interface

```
interface StackFactory : RACollectionFactory {
Stack create (in Operations ops, in unsigned long expected_size);
};
```

Stack create (in Operations ops, in unsigned long expected_size);

Returns an instance of a **Stack**. No requirements for the element key operations to be implemented is specified for a **Stack**. Nevertheless, one still has to pass an instance of **Operations** as type checking information must pass to the collection implementation.

The DequeFactory Interface

```
interface DequeFactory : RACollectionFactory {
Deque create (in Operations ops, in unsigned long expected_size);
};
```

Deque create (in Operations ops, in unsigned long expected_size);

Returns an instance of a **Deque**. No requirements on the element key operations to be implemented is specified for a **Deque**. Nevertheless, one still has to pass an instance of **Operations** as type checking information must pass to the collection implementation.

The PriorityQueueFactory Interface

```
interface PriorityQueueFactory : RACollectionFactory {
PriorityQueue create (in Operations ops, in unsigned long
expected_size);
};
```

PriorityQueue create (in Operations ops, in unsigned long expected_size);

Returns an instance of a **PriorityQueue**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-18 Required element and key-type specific user-defined information for PriorityQueueFactory. [] - implied by key_compare.

PriorityQueue						
equal	compare	hash	key	key_equal	key_compare	key_hash
			x	[x]	x	

17.5.9 Iterator Interfaces

Iterators as pointer abstraction

An **iterator** is in a first approximation of a pointer abstraction. It is a movable pointer to elements of a collection. Iterators are tightly intertwined with collections. An iterator cannot exist independently of a collection (i.e., the iterator life time cannot exceed that of the collection for which it is created). A collection is the factory for *its* iterators. An iterator is created for a given collection and can be used for this and only this collection.

The iterators specified in this specification form an interface hierarchy which parallels the collection interface hierarchy. The supported iterator movements reflect the capabilities of the corresponding collection type.

The top level **Iterator** interface defines a generic iterator usable for iteration over all types of collections. It can be set to a start position for iteration and moved via a series of forward movements through the collection visiting each element exactly once.

The **OrderedIterator** is supported by ordered collections only. It “knows about ordering;” therefore, it can be moved in forward and backward direction.

The **KeyIterator** exploits the capabilities of key collections. It can be moved to an element with a given key value, advanced to the next element with the same key value, or advanced to the next element with a different key value in iteration order.

The **KeySortedIterator** is created for key collections sorted by key. The iterator can be advanced to the previous element with the same key value or the previous element with a different key value.

The **EqualityIterator** exploits the capabilities of equality collections. It can be moved to an element with a given value, advanced to the next element with the same element value, or advanced to the next element with a different element value in iteration order.

The **EqualitySortedIterator** is created for equality collections sorted by element value. The iterator can be advanced to the previous element with the same value or the previous element with a different value.

Iterators and support for generic programming

Iterators go far beyond being simple “pointing devices.” There are essentially two reasons to extend the capabilities of iterators.

1. To support the processing of very large collections which allows for delayed instantiation or incremental query evaluation in case of very large query results. These are scenarios where the collection itself may never exist as instantiated main memory collection but is processed in “finer grains” via an iterator passed to a client.
2. To enrich the iterator with more capabilities strengthens the support for the generic programming model, as introduced with ANSI STL to the C++ world.

You can retrieve, replace, remove, and add elements via an iterator. You can test iterators for equality, compare ordered iterators, clone an iterator, assign iterators, and destroy them. Furthermore an iterator can have a `CONST` designation which is set when created. A `CONST` iterator can be used for access only.

The `reverse` iterator semantics is supported. No extra interfaces are specified to support this, but a `reverse` designation is set at creation time. An ordered iterator for which the `reverse` designation is set reinterprets the operations of a given iterator type to work in reverse.

Iterators and performance

To reduce network traffic, *combined* operations and *batch* or *bulk* operations are offered.

Combined operations are combinations of simple iterator operations often used in loops. These combinations support generic algorithms. For example, a typical combination is “test whether range end is reached; if not `retrieve_element`, advance iterator to next element.”

Batch or *bulk* operations support the retrieval, replacement, addition, and removal of many elements within one operation. In these operations, the “many elements” are always passed as a `CORBA::sequence` of elements.

The Managed Iterator Model

All iterators are managed. The real benefit of being managed is that these iterators never become undefined. Note that “undefined” is different from “invalid.” While “invalid” is a testable state and means the iterator points to nothing, “undefined” means you do not know where the iterator points to and cannot inquiry it. Changing the contents of a collection by adding or deleting elements would cause an unmanaged iterator to become “undefined.” The iterator may still point to the same element, but it may also point to another element or even “outside” the collection. As you do not know the iterator state and cannot inquiry which state the iterator has, you are forced to newly position the unmanaged iterator, for example, via a `set_to_first_element()`.

This kind of behavior, common in collection class libraries today, seems unacceptable in a distributed multi-user environment. Assume one client removes and adds elements from a collection with side effects on the unmanaged iterators of another client. The other client is not able to test whether there have been side effects on its unmanaged iterators, but would only notice them indirectly when observing strange behavior of the application.

Managed iterators are intimately related to the collection they belong to, and thus, can be informed about the changes taking place within the collection. They are always in a defined state which allows them to be used even though elements have been added or removed from the collection. An iterator may be in the state *invalid*, that is pointing to nothing. Before it can be used it has to be set to a valid position. An iterator in the state

valid may either point to an element (and be valid for all operations on it) or it may be in the state *in-between*, that is, not pointing to an element but still “remembering” enough state to be valid for most operations on it.

A valid managed iterator remains valid as long as the element it points to remains in the collection. As soon as the element is removed, the according managed iterator enters a so-called *in-between* state. The *in-between* state can be viewed as a vacuum within the collection. There is nothing the managed iterator can point to. Nevertheless, managed iterators remember the next (and for ordered collection, also the previous) element in iteration order. It is possible to continue using the managed iterator (in a `set_to_next_element()` for example) without resetting it first.

There are some limitations. Once a managed iterator no longer points to an element, it remembers the iteration order in which the element stood before it was deleted. However, it does not remember the element itself. Thus, there are some operations which cannot be performed even though a managed iterator is used.

Consider an iteration over a **Bag**, for example. If you iterate over all different elements with the iterator operation `set_to_next_different_element()`, then removing the element the iterator points to leads to an undefined behavior of the collection later on. By removing the element, the iterator becomes *in-between*. The `set_to_next_different_element()` operation then has no chance to find the next different element as the collection does not know what is different in terms of the current iterator state. Likewise, for a managed iterator in the state *in-between* all operations ending with “..._at” are not defined. The reason is simple: There is no element at the iterator’s position - nothing to retrieve, to replace, or to remove in it. This situation is handled by raising an exception `IteratorInvalid`.

Additionally, all operations that (potentially) destroy the iteration order of a collection invalidate the corresponding managed iterators that have been in the state *in-between* before the operation was invoked. These are the `sort()` and the `reverse()` operation.

The Iterator Interface

```
// Iterators

interface Iterator {

    // moving iterators
    boolean set_to_first_element ();
    boolean set_to_next_element() raises (IteratorInvalid);
    boolean set_to_next_nth_element (in unsigned long n) raises
    (IteratorInvalid);

    // retrieving elements
    boolean retrieve_element (out any element) raises (IteratorInvalid,
    IteratorInBetween);
```



```
boolean retrieve_element_set_to_next (out any element, out boolean
more) raises (IteratorInvalid, IteratorInBetween);
boolean retrieve_next_n_elements (in unsigned long n, out
AnySequence result, out boolean more) raises (IteratorInvalid,
IteratorInBetween);
boolean not_equal_retrieve_element_set_to_next (in Iterator test,
out any element) raises (IteratorInvalid, IteratorInBetween);

// removing elements
void remove_element() raises (IteratorInvalid, IteratorInBetween);
boolean remove_element_set_to_next() raises (IteratorInvalid,
IteratorInBetween);
boolean remove_next_n_elements (in unsigned long n, out unsigned
long actual_number) raises (IteratorInvalid, IteratorInBetween);
boolean not_equal_remove_element_set_to_next (in Iterator test)
raises (IteratorInvalid, IteratorInBetween);

// replacing elements
void replace_element (in any element) raises (IteratorInvalid,
IteratorInBetween, ElementInvalid);
boolean replace_element_set_to_next (in any element)
raises(IteratorInvalid, IteratorInBetween, ElementInvalid);
boolean replace_next_n_elements (in AnySequence elements, out
unsigned long actual_number) raises (IteratorInvalid,
IteratorInBetween, ElementInvalid);
boolean not_equal_replace_element_set_to_next (in Iterator test, in
any element) raises(IteratorInvalid,IteratorInBetween,
ElementInvalid);

// adding elements
boolean add_element_set_iterator (in any element)raises
(ElementInvalid);
boolean add_n_elements_set_iterator (in AnySequence elements, out
unsigned long actual_number) raises (ElementInvalid);

// setting iterator state
void invalidate ();
// testing iterators
boolean is_valid ();
boolean is_in_between ();
boolean is_for(in Collection collector);
boolean is_const ();
boolean is_equal (in Iterator test) raises (IteratorInvalid);

// cloning, assigning, destroying an iterators
```

```
Iterator clone ();  
void assign (in Iterator from_where) raises (IteratorInvalid);  
void destroy ();  
};
```

Moving iterators

```
boolean set_to_first_element ();
```

Description

The iterator is set to the first element in iteration order of the collection it belongs to. If the collection is empty, that is, if no first element exists, the iterator is invalidated.

Return value

Returns **true** if the collection it belongs to is not empty.

```
boolean set_to_next_element () raises (IteratorInvalid);
```

Description

Sets the iterator to the next element in the collection in iteration order or invalidates the iterator if no more elements are to be visited. If the iterator is in the state *in-between*, the iterator is set to its “potential next” element.

Return value

Returns **true** if there is a next element.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

```
boolean set_to_next_nth_element (in unsigned long n) raises (IteratorInvalid);
```

Description

Sets the iterator to the element *n* movements away in collection iteration order or invalidates the iterator if there is no such element. If the iterator is in the state *in-between* the movement to the “potential next” element is the first of the *n* movements.

Return value

Returns **true** if there is such an element.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

Retrieving elements

boolean `retrieve_element` (out any element) raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Retrieves the element pointed and returns it via the output parameter `element`.

Return value

Returns `true` if an element was retrieved.

Exceptions

The iterator must point to an element of the collection; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

Note – Whether a copy of the element is returned or the element itself depends on the element type represented by the `any`. If it is an object, a reference to the object in the collection is returned. If the element type is a non-object type, a copy of the element is returned. In case of element type object, do not manipulate the element or the key of the element in the collection in a way that changes the positioning property of the element.

boolean `retrieve_element_set_to_next` (out any element) raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Retrieves the element pointed to and returns it via the output parameter `element`. The iterator is moved to the next element in iteration order. If there is a next element `more` is set to `true`. If there are no more next elements, the iterator is invalidated and `more` is set to `false`.

Return value

Returns `true` if an element was retrieved.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

boolean `retrieve_next_n_elements` (in unsigned long `n`, out `AnySequence` `result`, out boolean `more`) raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Retrieves at most the next `n` elements in iteration order of the iterator's collection and returns them as `sequence` of `anys` via the output parameter `result`. Counting starts with the element the iterator points to. The iterator is moved behind the last element retrieved. If there is an element behind the last element retrieved, `more` is set to `true`. If there are no more elements behind the last element retrieved or there are less than `n` elements for retrieval, the iterator is invalidated and `more` is set to `false`. If the value of `n` is 0, all elements in the collection are retrieved until the end is reached.

Return value

Returns `true` if at least one element is retrieved.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

boolean `not_equal_retrieve_element_set_to_next` (in `Iterator` `test`, out any element) raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Compares the given iterator `test` with this iterator.

- If they are not equal, the element pointed to by this iterator is retrieved and returned via the output parameter `element`, the iterator is moved to the next element, and `true` is returned.
- If they are equal, the element pointed to by this iterator is retrieved and returned via the output parameter `element`, the iterator is not moved to the next element, and `false` is returned.

Return value

Returns `true` if this iterator is not equal to the test iterator at the beginning of the operation.

Exceptions

The iterator and the given iterator `test` each must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

Removing elements

void `remove_element` () raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Removes the element pointed to by this iterator and sets the iterator *in-between*.

Exceptions

The iterator must be valid and point to an element of the collection; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The iterator must not have the `const` designation; otherwise, the exception `IteratorInvalid` is raised.

Side effects

Other valid iterators pointing to the removed element go *in-between*.

All other iterators keep their state.

`boolean remove_element_set_to_next()` (`IteratorInvalid`, `IteratorInBetween`);

Description

Removes the element pointed to by this iterator and moves the iterator to the next element.

Return value

Returns `true` if a next element exists.

Exceptions

The iterator must be valid and point to an element of the collection; otherwise, the exception `IteratorInvalid` is raised.

The iterator must not have the `const` designation; otherwise, the exception `IteratorInvalid` is raised.

Side effects

Other valid iterators pointing to the removed element go *in-between*.

All other iterators keep their state.

`boolean remove_next_n_elements` (in unsigned long `n`, out unsigned long `actual_number`) raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Removes at most the next `n` elements in iteration order of the iterator's collection. Counting starts with the element the iterator points to. The iterator is moved to the next element behind the last element removed. If there are no more elements behind the last element removed or there are less than `n` elements for removal, the iterator

is invalidated. If the value of `n` is 0, all elements in the collection are removed until the end is reached. The output parameter `actual_number` is set to the actual number of elements removed. If the value of `n` is 0, all elements in the collection are removed until the end is reached.

Return value

Returns `true` if the iterator is not invalidated.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The iterator must not have the `const` designation; otherwise, the exception `IteratorInvalid` is raised.

Side effects

Other valid iterators pointing to removed elements go *in-between*.

All other iterators keep their state.

`boolean not_equal_remove_element_set_to_next(in iterator test)`
(`IteratorInvalid`, `IteratorInBetween`);

Description

Compares this iterator with the given iterator `test`. If they are not equal the element this iterators points to is removed and the iterator is set to the next element, and `true` is returned. If they are equal the element pointed to is removed, the iterator is set *in-between*, and `false` is returned.

Return value

Returns `true` if this iterator and the given iterator `test` are not equal when the operations starts.

Exception

This iterator and the given iterator `test` must be valid otherwise the exception `IteratorInvalid` or `IteratorInBetween` is raised.

This iterator and the given iterator `test` must not have a `const` designation otherwise the exception `IteratorInvalid` is raised.

Side effects

Other valid iterators pointing to removed elements go *in-between*.

All other iterators keep their state.

Replacing elements

void `replace_element` (in any element) raises (`IteratorInvalid`, `IteratorInBetween`, `ElementInvalid`);

Description

Replaces the element pointed to by the given element.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The iterator must not have a `CONST` designation; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected element type; otherwise, the `ElementInvalid` exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception `ElementInvalid` is raised.

For positioning properties, see “The Collection Interface” on page 17-21.

boolean `replace_element_set_to_next`(in any element) raises (`IteratorInvalid`, `IteratorInBetween`, `ElementInvalid`);

Description

Replaces the element pointed to by this iterator by the given element and sets the iterator to the next element. If there are no more elements, the iterator is invalidated.

Return value

Returns `true` if there is a next element.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The iterator must not have a `CONST` designation; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected element type; otherwise, the `ElementInvalid` exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception `ElementInvalid` is raised.

For positioning properties, see “The Collection Interface” on page 17-21.

boolean `replace_next_n_elements`(in AnySequence `elements`, out unsigned long `actual_number`) raises (`IteratorInvalid`, `IteratorInBetween`, `ElementInvalid`);

Description

Replaces at most as many elements in iteration order as given in `elements` by the given elements. Counting starts with the element the iterator points to. If there are less elements in the collection left to be replaced than the given number of elements as many elements as possible are replaced and the actual number of elements replaced is returned via the output parameter `actual_number`.

The iterator is moved to the next element behind the last element replaced. If there are no more elements behind the last element replaced or the number of elements in the collection to be replaced is less than the number given `elements`, the iterator is invalidated.

Return value

Returns `true` if there is another element behind the last element replaced.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The elements given must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

For each element the positioning property of the replaced element must be the same as that of the element replacing it; otherwise, the exception `ElementInvalid` is raised.

For positioning property see “The Collection Interface” on page 17-21.

boolean `not_equal_replace_element_set_to_next` (in `Iterator` `test`, in any element) raises (`IteratorInvalid`, `IteratorInBetween`, `ElementInvalid`);

Description

Compares this iterator and the given iterator `test`. If they are not equal, the element pointed to by this iterator is replaced by the given element, the iterator is set to the next element, and `true` is returned. If they are equal, the element pointed to by this iterator is replaced by the given element, the iterator is not set to the next element, and `false` is returned.

Return value

Returns `true` if this iterator and the given iterator `test` are not equal before the operations starts.

Exceptions

This iterator and the given iterator must be valid and point to an element each; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

This iterator must not have a `const` designation; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected element type; otherwise, the `ElementInvalid` exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception `ElementInvalid` is raised.

For positioning property, see “The Collection Interface” on page 17-21.

Adding elements

`boolean add_element_set_iterator` (in any element) (`ElementInvalid`);

Description

Adds an element to the collection that this iterator points to and sets the iterator to the added element. The exact semantics depends on the properties of the collection for which this iterator is created.

If the collection supports unique elements or keys and the element or key is already contained in the collection, adding is ignored and the iterator is just set to the element or key already contained. In sequential collections, the element is always added as last element. In sorted collections, the element is added at a position determined by the element or key value.

Return value

Returns `true` if the element was added. The element to be added must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Exceptions

If the collection is a `Map` and contains an element with the same key as the given element, then this element has to be equal to the given element; otherwise, the exception `ElementInvalid` is raised.

Side effects

All other iterators keep their state.

`void add_n_elements_set_iterator` (in `AnySequence` elements, out unsigned long actual_number) (`ElementInvalid`);

Description

Adds the given elements to the collection that this iterator points to. The elements are added in the order of the input sequence of elements and the delivered semantics is consistent with the semantics of the `add_element_set_iterator` operation. It is essentially a sequence of `add_element_set_iterator` operations. The output parameter `actual_number` is set to the number of elements added.

Setting iterator state

```
void invalidate ();
```

Description

Sets the iterator to the state *invalid*, that is, “pointing to nothing.” You may also say that the iterator, in some sense, is set to “NULL.”

Testing iterators

Whenever there is a precondition for an iterator operation to be checked, there is a test operation provided that enables the user to avoid raising an exception.

```
boolean is_valid ();
```

Return value

Returns `true` if the Iterator is *valid*, that is points to an element of the collection or is in the state *in-between*.

```
boolean is_for (in Collection collector);
```

Return value

Returns `true` if this iterator can operate on the given collection.

```
boolean is_const ();
```

Return value

Returns `true` if this iterator is created with “const” designation.

```
boolean is_in_between ();
```

Return value

Returns `true` if the iterator is in the state *in-between*.

boolean is_equal (in Iterator test) raises (IteratorInvalid);

Return value

Returns true if the given iterator points to the identical element as this iterator.

Exceptions

The given iterator must belong to the same collection as the iterator; otherwise, the exception IteratorInvalid is raised.

Cloning, Assigning, Destroying iterators

Iterator clone();

Description

Creates a copy of this iterator.

void assign (in Iterator from_where) raises (IteratorInvalid)

Description

Assigns the given iterator to this iterator.

Exceptions

The given iterator must be created for the same collection as this iterator; otherwise, the exception IteratorInvalid is raised.

void destroy();

Description

Destroys this iterator.

The OrderedIterator Interface

```
interface OrderedIterator: Iterator {
```

```
    // moving iterators
```

```
    boolean set_to_last_element ();
```

```
    boolean set_to_previous_element() raises (IteratorInvalid);
```

```
    boolean set_to_nth_previous_element(in unsigned long n) raises  
    (IteratorInvalid);
```

```

void set_to_position (in unsigned long position) raises
(PositionInvalid);

// computing iterator position
unsigned long position () raises (IteratorInvalid);

// retrieving elements
boolean retrieve_element_set_to_previous(out any element, out
boolean more) raises (IteratorInvalid, IteratorInBetween);
boolean retrieve_previous_n_elements (in unsigned long n, out
AnySequence result, out boolean more) raises (IteratorInvalid,
IteratorInBetween);
boolean not_equal_retrieve_element_set_to_previous (in Iterator
test, out any element) raises (IteratorInvalid, IteratorInBetween);

// removing elements
boolean remove_element_set_to_previous() raises (IteratorInvalid,
IteratorInBetween);
boolean remove_previous_n_elements (in unsigned long n, out unsigned
long actual_number) raises (IteratorInvalid, IteratorInBetween);
boolean not_equal_remove_element_set_to_previous(in Iterator test)
raises (IteratorInvalid, IteratorInBetween);

// replacing elements
boolean replace_element_set_to_previous(in any element) raises
(IteratorInvalid, IteratorInBetween, ElementInvalid);
boolean replace_previous_n_elements(in AnySequence elements, out
unsigned long actual_number) raises (IteratorInvalid,
IteratorInBetween, ElementInvalid);
boolean not_equal_replace_element_set_to_previous (in Iterator
test, in any element) raises (IteratorInvalid,IteratorInBetween,
ElementInvalid);

// testing iterators
boolean is_first ();
boolean is_last ();
boolean is_for_same (in Iterator test);
boolean is_reverse ();
};

```

Moving iterators

```
boolean set_to_last_element();
```

Description

Sets the iterator to the last element of the collection in iteration order. If the collection is empty (if no last element exists) the given iterator is invalidated.

Return value

Returns **true** if the collection is not empty.

`boolean set_to_previous_element()` raises (`IteratorInvalid`);

Description

Sets the iterator to the previous element in iteration order, or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, the iterator is set to its “potential previous” element.

Return value

Returns **true** if a previous element exists.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

`boolean set_to_nth_previous_element (in unsigned long n)` raises (`IteratorInvalid`);

Description

Sets the iterator to the element *n* movements away in reverse collection iteration order or invalidates the iterator if there is no such element. If the iterator is in the state *in-between*, the movement to the “potential previous” element is the first of the *n* movements.

Return value

Returns **true** if there is such an element.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

`void set_to_position (in unsigned long position)` raises (`PositionInvalid`);

Description

Sets the iterator to the element at the given position. Position 1 specifies the first element.

Exceptions

Position must be a valid position (i.e., greater than or equal to 1 and less than or equal to `number_of_elements()`); otherwise, the exception `PositionInvalid` is raised.

Computing iterator position

`unsigned long position ()` raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Determines and returns the current position of the iterator. Position 1 specifies the first element.

Exceptions

The iterator must be pointing to an element of the collection; otherwise, the exception `IteratorInvalid` respectively `IteratorInBetween` is raised.

Retrieving elements

`boolean retrieve_element_set_to_previous (out any element, out boolean more)` raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Retrieves the element pointed to and returns it via the output parameter `element`. The iterator is set to the previous element in iteration order. If there is a previous element, `more` is set to `true`. If there are no more previous elements, the iterator is invalidated and `more` is set to `false`.

Return value

Returns `true` if an element was returned.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

`boolean retrieve_previous_n_elements(in unsigned long n, out AnySequence result, out boolean more)` raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Retrieves at most the *n* previous elements in iteration order of this iterator's collection and returns them as **sequence** of anys via the output parameter **result**. Counting starts with the element the iterator is pointing to. The iterator is moved to the element before the last element retrieved.

- If there is an element before the last element retrieved, **more** is set to **true**.
- If there are no more elements before the last element retrieved or there are less than *n* elements for retrieval, the iterator is invalidated and **more** is set to **false**.
- If the value of *n* is 0, all elements in the collection are retrieved until the end is reached.

Return value

Returns **true** if at least one element is retrieved.

Exceptions

The iterator must be valid and pointing to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

boolean `not_equal_retrieve_element_set_to_previous` (in `Iterator` test, out any element) raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Compares the given iterator **test** with this iterator.

- If they are not equal, the element pointed to by this iterator is retrieved and returned via the output parameter **element**, the iterator is moved to the previous element, and **true** is returned.
- If they are equal, the element pointed to by this iterator is retrieved and returned via the output parameter **element**, the iterator is not moved to the previous element, and **false** is returned.

Return value

Returns **true** if this iterator is not equal to the test iterator at the beginning of the operation.

Exceptions

The iterator and the given iterator **test** each must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

Replacing elements

boolean `replace_element_set_to_previous`(in any element) raises (`IteratorInvalid`, `IteratorInBetween`, `ElementInvalid`);

Description

Replaces the element pointed to by this iterator by the given element and sets the iterator to the previous element. If there are no previous elements, the iterator is invalidated.

Return value

Returns **true** if there is a previous element.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The iterator must not have a **const** designation; otherwise, the exception `IteratorInvalid` is raised.

The element must be the expected element type; otherwise, the `ElementInvalid` exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception `ElementInvalid` is raised.

For positioning properties, see “The Collection Interface” on page 17-21.

`boolean replace_previous_n_elements(in AnySequence elements, out unsigned long actual_number) raises (IteratorInvalid, IteratorInBetween, ElementInvalid);`

Description

At most, replaces as many elements in reverse iteration order as given in **elements**. Counting starts with the element the iterator points to. If there are less elements in the collection left to be replaced than the given number of elements as many elements as possible are replaced and the actual number of elements replaced is returned via the output parameter **actual_number**.

The iterator is moved to the element before the last element replaced. If there are no more elements before the last element replaced or the number of elements in the collection to be replaced is less than the number of given elements, the iterator is invalidated.

Return value

Returns **true** if there is an element before the last element replaced.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The elements given must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

For each element the positioning property of the replaced element must be the same as that of the element replacing it; otherwise, the exception `ElementInvalid` is raised.

For positioning property, see “The Collection Interface” on page 17-21.

boolean `not_equal_replace_element_set_to_previous` (in `Iterator` test, in any element) raises (`IteratorInvalid`, `IteratorInBetween`, `ElementInvalid`);

Description

Compares this iterator and the given iterator `test`.

- If they are not equal, the element pointed to by this iterator is replaced by the given element, the iterator is set to the previous element, and `true` is returned.
- If they are equal, the element pointed to by this iterator is replaced by the given element, the iterator is not set to the previous element, and `false` is returned.

Return value

Returns `true` if this iterator and the given iterator `test` are not equal before the operations starts.

Exceptions

This iterator and the given iterator each must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

This iterator must not have a `const` designation; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected element type; otherwise, the `ElementInvalid` exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception `ElementInvalid` is raised.

For positioning property, see “The Collection Interface” on page 17-21.

Removing elements

boolean `remove_element_set_to_previous()` raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Removes the element pointed to by this iterator and moves the iterator to the previous element.

Return value

Returns **true** if a previous element exists.

Exceptions

The iterator must be valid and point to an element of the collection; otherwise, the exception `IteratorInvalid` is raised.

The iterator must not have the `const` designation; otherwise, the exception `IteratorInvalid` is raised.

Side effects

Other valid iterators pointing to the removed element go *in-between*.

All other iterators keep their state.

`boolean remove_previous_n_elements` (in unsigned long `n`, out unsigned long `actual_number`) raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Removes at most the previous `n` elements in reverse iteration order of the iterator's collection. Counting starts with the element the iterator points to. The iterator is moved to the element before the last element removed.

- If there are no more elements before the last element removed or there are less than `n` elements for removal, the iterator is invalidated.
- If the value of `n` is 0, all elements in the collection are removed until the beginning is reached. The output parameter `actual_number` is set to the actual number of elements removed.

Return value

Returns **true** if the iterator is not invalidated.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The iterator must not have the `const` designation; otherwise, the exception `IteratorInvalid` is raised.

Side effects

Other valid iterators pointing to removed elements go *in-between*.

All other iterators keep their state.

`boolean not_equal_remove_element_set_to_previous`(in `Iterator test`) raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Compares this iterator with the given iterator `test`.

- If they are not equal, the element this iterator points to is removed, the iterator is set to the previous element, and `true` is returned.
- If they are equal, the element pointed to is removed, the iterator is set *in-between*, and `false` is returned.

Return value

Returns `true` if this iterator and the given iterator `test` are equal when the operation starts.

Exceptions

This iterator and the given iterator `test` must be valid; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

This iterator and the given iterator `test` must not have a `const` designation; otherwise, the exception `IteratorInvalid` is raised.

Side effects

Other valid iterators pointing to the removed element go *in-between*.

All other iterators keep their state.

Testing iterators

```
boolean is_first ();
```

Return value

Returns `true` if the iterator points to the first element of the collection it belongs to.

```
boolean is_last ();
```

Return value

Returns `true` if the iterator points to the last element of the collection it belongs to.

```
boolean is_for_same (in Iterator test);
```

Return value

Returns `true` if the given iterator is for the same collection as this.

```
boolean is_reverse();
```

Return value

Returns **true** if the iterator is created with “reverse” designation.

The SequentialIterator Interface

```
interface SequentialIterator : OrderedIterator {
// adding elements
boolean add_element_as_next_set_iterator (in any element)
raises(IteratorInvalid, ElementInvalid);
void add_n_elements_as_next_set_iterator(in AnySequence elements)
raises(IteratorInvalid, ElementInvalid);

boolean add_element_as_previous_set_iterator(in any element)
raises(IteratorInvalid, ElementInvalid);
void add_n_elements_as_previous_set_iterator(in AnySequence
elements) raises(IteratorInvalid, ElementInvalid);
};
```

Adding elements

```
boolean add_element_as_next_set_iterator (in any element)
raises(IteratorInvalid, ElementInvalid);
```

Description

Adds the element to the collection that this iterator points to (in iteration order) behind the element this iterator points to and sets the iterator to the element added. If the iterator is in the state *in-between*, the element is added before the “potential next” element.

Return value

Returns **true** if the element is added.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The element added must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All other iterators keep their state.

```
void add_n_elements_as_next_set_iterator(in AnySequence elements)
raises(IteratorInvalid, ElementInvalid);
```

Description

Adds the given elements to the collection that this iterator points to behind the element the iterator points to. The behavior is the same as *n* times calling the operation `add_element_as_next_set_iterator()`.

If the iterator is in the state *in-between*, the elements are added before the “potential next” element.

The elements are added in the order given in the input sequence.

```
boolean add_element_as_previous_set_iterator(in any element)
raises(IteratorInvalid, ElementInvalid)
```

Description

Adds the element to the collection that this iterator points to (in iteration order) before the element that this iterator points to and sets the iterator to the element added. If the iterator is in the state *in-between*, the element is added after the “potential previous” element.

Return value

Returns `true` if the element is added.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The element added must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All other iterators keep their state.

```
void add_n_elements_as_previous_set_iterator(in AnySequence elements)
raises(IteratorInvalid, ElementInvalid);
```

Description

Adds the given elements to the collection that this iterator points to previous to the element the iterator points to. The behavior is the same as *n* times calling the operation `add_element_as_previous_set_to_next()`.

If the iterator is in the state *in-between*, the elements are added behind the “potential previous” element.

The elements are added in the reverse order given in the input sequence.

The KeyIterator Interface

```
interface KeyIterator : Iterator {  
    // moving the iterators  
    boolean set_to_element_with_key (in any key) raises(KeyInvalid);  
    boolean set_to_next_element_with_key (in any key)  
    raises(IteratorInvalid, KeyInvalid);  
    boolean set_to_next_element_with_different_key() raises  
    (IteratorInBetween, IteratorInvalid);  
  
    // retrieving the keys  
    boolean retrieve_key (out any key) raises (IteratorInBetween,  
    IteratorInvalid);  
    boolean retrieve_next_n_keys (out AnySequence keys) raises  
    (IteratorInBetween, IteratorInvalid);  
};
```

Moving iterators

boolean set_to_element_with_key (in any key) raises (KeyInvalid);

Description

Locates an element in the collection with the same key as the given key. Sets the iterator to the element located or invalidates the iterator if no such element exists.

If the collection contains several such elements, the first element in iteration order is located.

Return value

Returns **true** if an element was found.

Exceptions

The key must be of the expected type; otherwise, the exception **KeyInvalid** is raised.

boolean set_to_next_element_with_key (in any key) raises (IteratorInvalid, KeyInvalid);

Description

Locates the next element in iteration order with the same key value as the given key, starting search at the element next to the one pointed to by the iterator. Sets the iterator to the element located.

- If there is no such element, the iterator is invalidated.
- If the iterator is in the state *in-between*, locating starts at the iterator's "potential next" element.

Return value

Returns **true** if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

`boolean set_to_next_element_with_different_key ()` raises (`IteratorInBetween`, `IteratorInvalid`)

Description

Locates the next element in iteration order with a key different from the key of the element pointed to by the iterator, starting the search with the element next to the one pointed to by the iterator. Sets the iterator to the located element.

If no such element exists, the iterator is invalidated.

Return value

Returns **true** if an element was found.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInBetween` respectively `IteratorInvalid` is raised.

Retrieving keys

`boolean key (out any key)` raises(`IteratorInvalid`,`IteratorInBetween`);

Description

Retrieves the key of the element this iterator points to and returns it via the output parameter `key`.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

`boolean retrieve_next_n_keys (in unsigned long n, out AnySequence keys)`
raises(`IteratorInvalid`, `IteratorInbetween`)

Description

Retrieves the keys of at most the next *n* elements in iteration order, sets the iterators to the element behind the last element from which a key is retrieved, and returns them via the output parameter *keys*. Counting starts with the element this iterator points to.

- If there is no element behind the last element from which a key is retrieved or there are less than *n* elements to retrieve keys from the iterator is invalidated.
- If the value of *n* is 0, the keys of all elements in the collection are retrieved until the end is reached.

Return value

Returns **true** if at least one key is retrieved.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The EqualityIterator Interface

```
interface EqualityIterator : Iterator {
  // moving the iterators
  boolean set_to_element_with_value(in any element)
  raises(ElementInvalid);
  boolean set_to_next_element_with_value(in any element)
  raises(IteratorInvalid, ElementInvalid);
  boolean set_to_next_element_with_different_value() raises
  (IteratorInBetween, IteratorInvalid);
};
```

Moving iterators

```
boolean set_to_element_with_value (in any element) raises(ElementInvalid);
```

Description

Locates an element in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists. If the collection contains several such elements, the first element in iteration order is located.

Return value

Returns **true** if an element is found.

Exceptions

The element must be of the expected type; otherwise, the expected `ElementInvalid` is raised.

`boolean set_to_next_element_with_value(in any element)` raises (`IteratorInvalid`, `ElementInvalid`);

Description

Locates the next element in iteration order in the collection that is equal to the given element, starting at the element next to the one pointed to by the iterator. Sets the iterator to the located element in the collection.

- If there is no such element, the iterator is invalidated.
- If the iterator is in the state *in-between*, locating is started at the iterator's "potential next" element.

Return value

Returns `true` if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

`boolean set_to_next_different_element ()` raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Locates the next element in iteration order that is different from the element pointed to. Sets the iterator to the located element, or if no such element exists, the iterator is invalidated.

Return value

Returns `true` if the next different element was found.

Exceptions

The iterator must be valid and point to an element of the collection; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The EqualityKeyIterator Interface

```
interface EqualityKeyIterator : EqualityIterator, KeyIterator {};
```

This interface just combines the two interfaces `EqualityIterator` (see “The EqualityIterator Interface” on page 17-110) and `KeyIterator` (see “The KeyIterator Interface” on page 17-108).

The SortedIterator Interface

```
interface SortedIterator : OrderedIterator {};
```

This interface does not add any new operations but new semantics to the operations.

The KeySortedIterator Interface

```
// enumeration type for specifying ranges
enum LowerBoundStyle {equal_lo, greater, greater_or_equal};
enum UpperBoundStyle {equal_up, less, less_or_equal};
interface KeySortedIterator : KeyIterator, SortedIterator
{
// moving the iterators
boolean set_to_first_element_with_key (in any key, in
LowerBoundStyle style) raises(KeyInvalid);
boolean set_to_last_element_with_key (in any key, in UpperBoundStyle
style) raises (KeyInvalid);
boolean set_to_previous_element_with_key (in any key)
raises(IteratorInvalid, KeyInvalid);
boolean set_to_previous_element_with_different_key() raises
(IteratorInBetween, IteratorInvalid);
// retrieving keys
boolean retrieve_previous_n_keys(out AnySequence keys) raises
(IteratorInBetween, IteratorInvalid);
};
```

Moving iterators

```
boolean set_to_first_element_with_key (in any key, in LowerBoundStyle style)
raises (KeyInvalid);
```

Description

Locates the first element in iteration order in the collection with key:

- equal to the given key, if style is `equal_lo`
- greater or equal to the given key, if style is `greater_or_equal`
- greater than the given key, if style is `greater`

Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

`boolean set_to_last_element_with_key(in any key, in UpperBoundStyle style);`

Description

Locates the last element in iteration order in the collection with key:

- equal to the given key, if `style` is `equal_up`
- less or equal to the given key, if `style` is `less_or_equal`
- less than the given key, if `style` is `less`

Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

`boolean set_to_previous_element_with_key (in any key) raises(IteratorInvalid, KeyInvalid);`

Description

Locates the previous element in iteration order with a key equal to the given key, beginning at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, the search begins at the iterator's "potential previous" element.

Return value

Returns **true** if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

`boolean set_to_previous_element_with_different_key() raises (IteratorInBetween, IteratorInvalid);`

Description

Locates the previous element in iteration order with a key different from the key of the element pointed to, beginning search at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInBetween` or `IteratorInvalid` is raised.

Retrieving keys

`boolean retrieve_previous_n_keys (in unsigned long n, out AnySequence keys)
raises(IteratorInvalid, IteratorInbetween)`

Description

Retrieves the keys of at most the previous `n` elements in iteration order, sets the iterators to the element before the last element from which a key is retrieved, and returns them via the output parameter `keys`. Counting starts with the element this iterator points to.

- If there is no element previous the one from which the `n`th key is retrieved or if there are less than `n` elements to retrieve keys from, the iterator is invalidated.
- If the value of `n` is 0, the keys of all elements in the collection are retrieved until the beginning is reached.

Return value

Returns `true` if at least one key is retrieved.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The EqualitySortedIterator Interface

```
interface EqualitySortedIterator : EqualityIterator, SortedIterator
{
    // moving the iterator
    boolean set_to_first_element_with_value (in any element, in
    LowerBoundStyle style) raises (ElementInvalid);
    boolean set_to_last_element_with_value (in any element, in
    UpperBoundStyle style) raises (ElementInvalid);
}
```

```

boolean set_to_previous_element_with_value (in any elementally)
raises (IteratorInvalid, ElementInvalid);
boolean set_to_previous_element_with_different_value() raises
(IteratorInBetween, IteratorInvalid);
};

```

Moving iterators

```

boolean set_to_first_element_with_value (in any element, in LowerBoundStyle
style) raises(ElementInvalid);

```

Description

Locates the first element in iteration order in the collection with value:

- equal to the given element value, if `style` is `equal_lo`
- greater or equal to the given element value, if `style` is `greater_or_equal`
- greater than the given element value, if `style` is `greater`

Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

```

boolean set_to_last_element_with_value(in any element, in UpperBoundStyle
style) raises (ElementInvalid);

```

Description

Locates the last element in iteration order in the collection with value:

- equal to the given element value, if `style` is `equal_up`
- less or equal to the given element value, if `style` is `less_or_equal`
- less than the given element value, if `style` is `less`

Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

`boolean set_to_previous_element_with_value(in any element)
raises(IteratorInvalid, ElementInvalid);`

Description

Locates the previous element in iteration order with a value equal to the given element value, beginning search at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, the search begins at the iterator's "potential previous" element.

Return value

Returns `true` if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

`boolean set_to_previous_element_with_different_value() raises
(IteratorInBetween, IteratorInvalid);`

Description

Locates the previous element in iteration order with a value different from the value of the element pointed to, beginning search at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInBetween` or `IteratorInvalid` is raised.

The EqualityKeySortedIterator Interface

```
interface EqualityKeySortedIterator: EqualitySortedIterator,  
KeySortedIterator {};
```

This interface combines the interfaces `KeySortedIterator` and `EqualitySortedIterator`. This interface does not add any new operations, but new semantics.

The EqualitySequentialIterator Interface

```
interface EqualitySequentialIterator : EqualityIterator,
SequentialIterator
{
// locating elements
boolean set_to_first_element_with_value (in any element) raises
(ElementInvalid);
boolean set_to_last_element_with_value (in any element) raises
(ElementInvalid);
boolean set_to_previous_element_with_value (in any element) raises
(ElementInvalid);
};
```

Moving Iterators

```
boolean set_to__first_element_with_value (in any element)
raises(ElementInvalid);
```

Description

Sets the iterator to the first element in iteration order in the collection that is equal to the given element or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

```
boolean set_to__last_element (in any element) raises(ElementInvalid);
```

Description

Sets the iterator to the last element in iteration order in the collection that is equal to the given element or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

`boolean set_to_previous_element_with_value` (in any element) raises (`IteratorInvalid`, `ElementInvalid`);

Description

Sets the iterator to the previous element in iteration order that is equal to the given element, beginning search at the element previous to the one specified by the iterator and moving in reverse iteration order through the elements. Sets the iterator to the located element or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, search starts at the “potential previous” element.

Return value

Returns `true` if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

17.5.10 Function Interfaces

The Operations Interface

```
Interface Operations {  
  
    // element type specific information  
    readonly attribute CORBA::TypeCode element_type;  
    boolean check_element_type (in any element);  
    boolean equal (in any element1, in any element2);  
    long compare (in any element1, in any element2);  
    unsigned long hash (in any element, in unsigned long value);  
  
    // key retrieval  
    any key (in any element);  
  
    // key type specific information  
    readonly attribute CORBA::TypeCode key_type;  
    boolean check_key_type (in any key);
```



```

boolean key_equal (in any key1, in any key2);
long key_compare (in any key1, in any key2);
unsigned long key_hash (in any thisKey, in unsigned long value);

// destroying
void destroy();
};

```

The function interface **Operations** is used to pass a number of other user-defined element type specific information to the collection implementation.

The first kind of element type specific information passed is used for typechecking. There are attributes specifying the element and key type expected in a given collection. In addition to the type information there are two typechecking operations which allow customizing the typechecking in a user-defined manner. The “default semantics” of these operations is a simple check on whether the type code of the given element or key exactly matches the type code specified in the element key type attribute.

Dependent on the properties as represented by a collection interface the respective implementation relies on some element type specific or key type specific information to be passed to it. For example one has to pass the information “element comparison” to implementation of a **SortedSet** or “key equality” to the implementation of a **KeySet** to guarantee uniqueness of keys. To pass this information, the **Operations** interface is used.

The third use of this interface is to pass element or key type specific information relevant for different categories of implementations. (Performing) implementations of associative collections essentially can be partitioned into the categories comparison-based or hashing-based. An AVL-tree implementation for a **KeySet** (for example) is key-comparison-based; therefore, it relies on key comparison defined and a hash table implementation of **KeySet** hashing-based (which relies on the information how a hash key values). Passing this information is the third kind of usage of the **Operations** interface.

The operations defined in the **Operations** interface are in summary:

- element type checking and key type checking
- element equality and the ordering relationship on elements
- key equality and ordering relationship on keys
- key access
- hash information on elements and keys

In order to pass this information to the collection, a user has to derive and implement an interface from the interface **Operations**. Which operations you have to implement depends on the collection interface and the implementation category you want to use. An instance of this interface is passed to a collection at creation time and then can be used by the implementation.

Ownership for an `Operations` instance is passed to the collection at creation time. That is, the same instance of `Operations` respectively a derived interface cannot be used in another collection instance. The collection is responsible for destroying the `Operations` instance when the collection is destroyed.

`Operations` only defines an abstract interface. Specialization and implementation are part of the application development as is the definition and implementation of respective factories and are not listed in this specification.

Element type specific operations

readonly attribute `CORBA::TypeCode` `element_type`;

Description

Specifies the type of the element to be collected.

boolean `check_element_type` (in any element);

Description

A collection implementation may rely on this operation being defined to use it for its type checking. A default implementation may be a simple test whether the type code of the given element exactly matches `element_type`. For object references, sometimes a check on equality of the type codes is not desired but a check on whether the type of the given element is a specialization of the `element_type`.

Return value

Returns `true` if the given element passed the user-defined element type-checking.

boolean `equal` (in any element1, in any element2);

Return value

Returns `true` if `element1` is equal to `element2` with respect to the user-defined semantics of element equality.

Note – If case `compare` is defined, the `equal` operation has to be consistently defined (i.e., is implied by the defined element comparison).

long `compare` (in any element1, in any element2);

Return value

Returns a value less than zero if `element1 < element2`, zero if the values are equal, and a value greater than zero if `element1 > element2` with respect to the user-defined ordering relationship on elements.

unsigned long hash (in any element, in unsigned long value);

Return value

Returns a user-defined hash value for the given `element`. The given `value` specifies the size of the hashtable. This information can be used for the implementation of more or less sophisticated hash functions. Computed hash values have to be less than `value`.

Note – The definition of the hash function has to be consistent with the defined element equality (i.e., if two elements are equal with respect to the user-defined element equality they have to be hashed to the same hash value).

Computing the key

any key (in any element);

Description

Computes the (user-defined) key of the given element.

Key type specific information

readonly attribute CORBA::TypeCode key_type;

Description

Specifies the type of the key of the elements to be collected.

boolean check_key_type (in any key);

Return value

Returns `true` if the given key passed the user-defined element type-checking.

boolean key_equal (in any key1, in any key2);

Return value

Returns **true** if **key1** is equal to **key2** with respect to the user-defined semantics of key equality.

Note – If case **key_compare** is defined, the **key_equal** operation has to be consistently defined (i.e., is implied by the defined key comparison). When both key and element equality are defined, the definitions have to be consistent in the sense that element equality has to imply key equality.

key_compare (in any **key1**, in any **key2**);

Return value

Returns a value less than zero if **key1** < **key2**, zero if the values are equal, and a value greater than zero if **key1** > **key2** with respect to the user-defined ordering relationship on keys.

unsigned long **key_hash** (in any **key**, in unsigned long **value**);

Return value

Returns a user defined hash value for the given **key**. The given **value** specifies the size of the hashtable. This information can be used for the implementation of more or less sophisticated hash functions. Computed hash values have to be less than **value**.

Note – The definition of the hash function has to be consistent with the defined key equality (i.e., if two elements are equal with respect to the user defined element equality they have to be hashed to the same hash value).

Destroying the Operations instance

void **destroy**();

Destroys the operations instance.

The Command and Comparator Interface

Command and Comparator are auxiliary interfaces.

A collection service provider may either provide the interfaces only or a default implementation that raises an exception whenever an operation of these interfaces is called. In either case, a user is forced to provide his/her implementation of either the interfaces or a derived interface to make use of them in the operations `all_elements_do`, and `sort`.

The Command Interface

An instance of an interface derived from `Command` is passed to the operation `all_elements_do` to be applied to all elements of the collection.

```
interface Command {
boolean do_on (in any element);
};
```

The Comparator Interface

An instance of a user defined interface derived from `Comparator` is passed to the operation `sort` as sorting criteria.

```
interface Comparator {
long compare (in any element1, in any element2);
};
```

The `compare` operation of the user's comparator (interface derived from `Comparator`) must return a result according to the following rules:

- >0 if (element1 > element2)
- 0 if (element1 = element2)
- <0 if (element1 < element2)

Appendix A *OMG Object Query Service*

A.1 *Object Query Service Differences*

Identification and Justification of Differences

The relationship between the Object Collection Service (OCS) and the Object Query Service (OQS) is two-fold. The Object Query Service uses collections as *query result* and as scope of query evaluation.

The `get_result` operation of `CosQuery::Query` for example and the `evaluate` operation of `CosQuery::QueryEvaluator` may return a collection as result or may return an iterator to the query result.

There may be a `QueryEvaluator` implementation that takes a collection instance passed as input parameter to evaluate a query on this collection which specifies the scope of evaluation. The query evaluator implementation relies on the `Collection` interface and the generic `Iterator` being supported by the collection passed.

A `CosQuery::QueryableCollection` is a special case of query evaluator which allows a collection to serve directly as the scope to which a query may be applied. As `QueryableCollection` is derived from `Collection` a respective instance can serve to collect a query result to which further query evaluation is applied.

Both usages of collections - as query result and as scope of evaluation - rely on the fact that a minimum collection interface representing a generic aggregation capability is supported as a common root for all collections. Further, they rely on a generic iterator that can be used on collections independent of their type.

Summarizing, Object Query Service essentially depends on a generic collection service matching some minimal requirements. As Object Query Service was defined when there was not yet any Object Collection Service specification available a generic collection service was defined as part of the Query Service specification.

The `CosQueryCollection` module defines three interfaces:

- `CollectionFactory`: provides a generic creation capability
- `Collection`: defines a generic aggregation capability
- `Iterator`: offers a minimal interface to traverse a collection.

Those interfaces specify the minimal requirements of OQS to a generic collection service. The following discusses whether it is possible to replace `CosQueryCollection` module by respective interfaces in the `CosCollection` module as defined in this specification. Differences are identified and justified.

In anticipation of the details given in the next paragraph we can summarize:

- The `CosCollection::Collection` top level collection interface matches the `CosQueryCollection::Collection` interface except for minor differences. Collections as defined in the `CosCollection` module can be used with Query Service.
- The `CosCollection::Collection` top level collection interface proposes an operation which one may consider as an overlap with the Object Query Service function. The operation `all_elements_do` which can be considered a special case of query evaluation.
- The `CosCollection::Iterator` top level iterator interface is consistent with `CosQueryCollection::Iterator` interface in the sense that operations defined in `CosQueryCollection::Iterator` are supported in `CosCollection::Iterator`. In addition a managed iterator semantics is defined which is reflected in the specified side effects on iterators for modifying collection operations. This differs from the iterator semantics defined in the Object Query Service specification but is considered a requirement in a distributed environment.
- There are a number of operations in the `CosCollection::Iterator` interface you do not find in the `CosQueryCollection::Iterator` interface. They are defined in the `CosCollection::Iterator` interface to provide support for performing distributed processing of very large collections and to support the generic programming model as introduced with ANSI STL to the C++ world.
- The restricted access collections which are part of this proposal do not inherit from the top level `CosCollection::Collection` interface. They cannot be used with Object Query Service as they are. But this is in the inherent nature of the restricted access semantics of these collections and is not considered to be a problem. Nevertheless, the interfaces of the restricted access collections allow combining them with the collections of the combined property collections hierarchy via multiple inheritance to enable usage of restricted access collections within the Object Query Service. In doing so, the restricted access collections lose the guarantee for restricted access, but only support interfaces offering the commonly used operation names for convenience.
- The `CosQueryCollection::CollectionFactory` defines the exact same interface as `CosCollection::CollectionFactory`.

Replacing the interfaces defined in the Object Query Service `CosQuery::Collection` module by the respective interface defined in this specification, the Object Collection Service enables the following inheritance relationship:

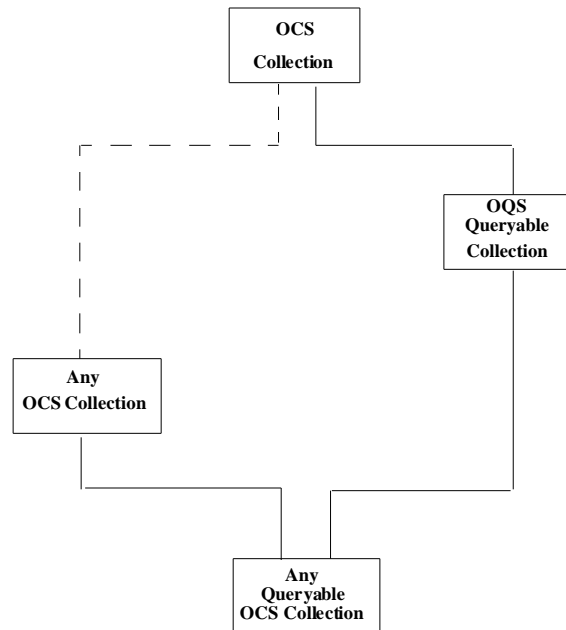


Figure 17-4 Inheritance Relationships

A detailed comparison of the interfaces is given in the following sections and is outlined along the `CosQueryCollection` module definitions.

CosQueryCollection Module Detailed Comparison

Exception Definitions

The following mapping of exceptions holds true:

- `CosQueryCollection::ElementInvalid` maps to `CosCollection::ElementInvalid`
- `CosQueryCollection::IteratorInvalid` maps to `CosCollection::IteratorInvalid` (with `IteratorInvalidReason` `not_for_collection`)
- `CosQueryCollection::PositionInvalid` maps to `CosCollection::IteratorInvalid` (with `IteratorInvalidReason` `is_invalid`) and `CosCollection::IteratorInBetween`

Type Definitions

There are a number of type definitions in the `CosQueryCollection` module for the mapping of SQL data types and for defining the type `Record`. These types are Object Query Service specific; therefore, they are not part of the Object Collection Service defined in this specification. Object Query Service may move these definitions to the `CosQuery` module.

CollectionFactory Interface

The `CosQueryCollection::CollectionFactory` interface defines the same interface as `CosCollection::CollectionFactory` and with it the same generic creation capability.

While the generic create operations of `CosQueryCollection::CollectionFactory` do not raise any exceptions, the respective operation in the `CosCollection::CollectionFactory` raises exception “ParameterInvalid.”

Collection Interface

The `CosQueryCollection::Collection` interface defines a basic collection interface, without restricting specializations to any particular type such as equality collections or ordered collections.

Collection Element Type

The element type of Object Query Service collections is a CORBA any to meet the general requirement that collections have to be able to collect elements of arbitrary type. The same holds true for the proposed Object Collection Service defined in this specification.

Using the CORBA any as element type implies the loss of compile time type checking. The Object Collection Service as defined here-in considers support for run-time type checking as important; therefore, it offers respective support. In the interface `Collection` this is reflected by introducing a read-only attribute “element_type” of type `TypeCode` which enables a client to inquire the element type expected.

This differs from Object Query Service collections which do not define any type checking specific support.

Collection Attributes

The following attribute is defined in the OQS Collection interface:

cardinality

This read-only attribute maps to the operation `number_of_elements()` in `CosCollection::Collection`. This is semantically equivalent. The name of the operation was chosen consistently with the overall naming scheme of the Collection Service.

Collection Operations

The following operations are defined in the Object Query Service Collection interface.

void add_element (in any element) raises (ElementInvalid)

This operation maps - except for side effects on iterators due to managed iterator semantics - to

`boolean add_element(in any element) raises (ElementInvalid)`

void add_all_elements (in Collection elements) raises (ElementInvalid)

This operation maps - except for side effects on iterators due to managed iterator semantics - to

void add_all_from (in Collection collector) raises (ElementInvalid).

void insert_element_at (in any element, in Iterator where) raises (IteratorInvalid, ElementInvalid)

This operation maps - except for side effects on iterators due to managed iterator semantics - to

boolean add_element_set_iterator(in any element, in Iterator where) raises (IteratorInvalid, ElementInvalid).

void replace_element_at (in any element, in Iterator where) raises (IteratorInvalid, PositionInvalid, ElementInvalid);

This operations maps to

void replace_element_at (in Iterator where, in any element) raises (IteratorInvalid, IteratorInBetween,ElementInvalid).

void remove_element_at (in Iterator where) raises (IteratorInvalid, PositionInvalid)

This operation maps - except for side effects on iterators due to managed iterator semantics - to

void remove_element_at (in Iterator where) raises (IteratorInvalid, IteratorInBetween).

void remove_all_elements ()

This operation maps - except for side effects on iterators due to managed iterator semantics - to

unsigned long remove_all ().

any retrieve_element_at (in Iterator where) raises (IteratorInvalid, PositionInvalid)

This operation maps to

`boolean retrieve_element_at` (in `Iterator` where, out any element) raises (`IteratorInvalid`, `IteratorInBetween`).

`Iterator create_iterator ()`

This operation maps to

`Iterator create_iterator` (in `boolean read_only`).

The parameter “`read_only`” parameter is used to support `const` iterators. This is introduced to support the iterator centric ANSI STL like programming model.

Where different operation names are used in the Object Collection Service defined here-in this is done to maintain consistency with the Collection Service overall naming scheme.

Side effects to iterators specified differ from those specified in the Query Service collection module as the Object Collection Service defined here-in specifies a managed iterator model which we consider necessary in a distributed environment. For more details in the managed iterator semantics see chapter “Iterator Interfaces.”

The top-level `CosCollection::Collection` interface proposes all the methods defined in `CosQueryCollection::Collection`. There are some few additional operations defined in `CosCollection::Collection`:

`boolean is_empty()`

This operation is provided as there are collection operations with the precondition that the collection must not be empty. To avoid an exception, the user should have the capability to test whether the collection is empty.

`void destroy()`

This operation is defined for destroying a collection instance without having to support the complete `LifeCycleObject` interface.

`void all_elements_do`(in `Command command`)

This operation is added for convenience; however, it seems to be an overlap with OQS functionality. This frequently used trivial query should be part of the collection service itself. A typical usage of this operation may be, for example, iterating over the collection to print all element values. Note that the `Command` functionality is very restricted to enable an efficient implementation. That is, the command is not allowed to change the positioning property of the element applied to and must not remove the element.

Iterator Interface

The `CosQueryCollection::Iterator` corresponds to `CosCollection::Iterator`. `CosCollection::Iterator` is supported for all collection interfaces of the Object Collection Service derived from `Collection`. The Object Collection Service iterator interfaces defined in this specification are designed to support an iterator centric and generic programming model as introduced with ANSI STL. This implies very powerful iterators which go far beyond simple pointing devices as one needs to be able to retrieve, add, remove elements from/to a collection via an iterator. In addition iterator interfaces are enriched with bulk and combined operations to enable an efficient processing of collections in distributed scenarios. Subsequently, the `CosCollection::Iterator` is much more powerful than the `CosQueryCollection::Iterator`.

Iterator Operations

The following operations are defined in the `CosQueryCollection::Iterator` interface:

- any `next ()` raises (`IteratorInvalid`, `PositionInvalid`)

This operation maps to

`boolean retrieve_element_set_to_next (out any element)` raises (`IteratorInvalid`, `IteratorInBetween`)

- `void reset ()`

This operation maps to

`boolean set_to_first_element()` of the Object Collection Service `Iterator` interface.

- `boolean more ()`

This operation maps to

`boolean is_valid() && ! is_inbetween()`

Due to the support for iterator centric and generic programming there are number of additional operations in the `CosCollection::Iterator` interface:

- `set_to_next_element`, `set_to_next_nth_element`
- `retrieve_element`, `retrieve_next_n_elements`, `not_equal_retrieve_element_set_to_next`
- `remove_element`, `remove_element_set_to_next`, `remove_next_n_elements`, `not_equal_remove_element_set_to_next`
- `replace_element`, `replace_element_set_to_next`, `replace_next_n_elements`, `not_equal_replace_element_set_to_next`
- `add_element_set_iterator`, `add_n_elements_set_iterator`
- `invalidate`
- `is_in_between`, `is_for`, `is_const`, `is_equal`
- `clone`, `assign`, `destroy`

Most of the operations can be implemented as combinations of other basic iterator operations so that the burden put on Object Query Service providers who implement such an interface should not be too high.

A.2 *Other OMG Object Services Defining Collections*

There are several object services that define collections, that is Naming Service, Property Service, and the OMG RFC "System Management: Common Management Facility, Volume 1" submission, for example.

These services define very application specific collections. The Naming Service for example defines the interface `NamingContext` or the Property Service an interface `PropertySet`. Both are very application specific collections and may be implemented using the Object Collection Service probably wrapping an appropriate Object Collection Service collection rather than specializing one of those collection interfaces.

The collections defined in the System Management RFC form a generic collection service. But the service defines collection members that need to maintain back references to collections in which they are contained to avoid dangling references in collections. This was considered as inappropriate heavyweight for a general object collection service. The collections in the System Management RFC may use Object Collection Service collections for their implementation up to some extent even reuse interfaces.

A.3 *OMG Persistent Object Services*

Collections as persistent objects in the sense defined by the Persistent Object Service

- may support the `CosPersistencePO::PO` interface. This interface enables a client being aware of the persistent state to explicitly control the PO's relationship with its persistent data (connect/disconnect/store/restore)
- may support the `CosPersistence::SD` interface which allows objects to synchronize their transient and persistent data
- have to support one of protocols used to get persistent data in and out of an object, like DA, ODMG, or DDO.

Support for these interfaces does not effect the collection interface.

Persistent *queryable* collections may request index support for collections. "Indexing of collections" enables to exploit underlying indices for efficient query evaluation. We do not consider "indexed collections" as part of the Object Collection Service but think that indexing support can be achieved via composing collections defined in the Object Collection Service proposed.

A.4 *OMG Object Concurrency Service*

Any implementation of the Object Collection Service probably will have to implement concurrency support. But we did not define any explicit concurrency support in the collection interfaces as part of the Object Collection Service because we consider that

as an implementation issue that can be solved by specialization. This also would allow to reuse the respective interfaces of the Object Concurrency Service rather than introducing a collection specific support for concurrency.

Appendix B Relationship to Other Relevant Standards

B.1 ANSI Standard Template Library

The ISO/ANSI C++ standard, as defined by ANSI X3J16 and OSI WG21, contains three sections defining the Containers library, the Iterators library and the Algorithms library, which form the main part of the **Standard Template Library**. Each section describes in detail the class structure, mandatory methods and performance requirements.

Containers

The standard describes two kinds of container template classes, sequence containers and so called associative containers. There is no inheritance structure relating the container classes.

Sequence containers organize the elements of a collection in a *strictly linear* arrangement. The following sequence containers are defined

- **vector**: Is a generalization of the concept of an ordinary C++ array the size of which can be dynamically changed. It's an indexed data structure, which allows fast, that is, constant time random access to its elements. Insertion and deletion of an element at the end of a vector can be done in constant time. Insertion and deletion of an element in the middle of the data structure may take linear time.
- **deque**: Like a vector it is an indexed structure of varying size, allowing fast, that is, constant time random access to its elements. In addition to what a vector offers a deque also offers constant time insertion and deletion of an element at the beginning.
- **list**: Is a sequence of varying size. Insertion and deletion of an element at any position can be done in constant time. But only linear-time access to an element at an arbitrary position is offered.

Associative containers provide the capability for fast, $O(\log n)$, retrieval of elements from the collections by "contents", that is, key value. The following associative containers are provided:

- **set**: Is a collection of unique elements which supports fast access, $O(\log n)$, to elements by element value.
- **multiset**: Allows multiple occurrences of the same element and supports fast access, $O(\log n)$, to elements by value.
- **map**: Is a collection of (key, value) pairs which supports unique keys. It is an indexed data structure which offers fast, $O(\log n)$, access to values by key.
- **multimap**: Is a collection of (key, value) pairs which allows multiple occurrences of the same key.

Container adapters are the well known containers with restricted access, that is:

- **stack**

- queue
- priority_queue

As roughly sketched ANSI STL specifies performance requirements for container operations. Those enforce up to some extent the kind of implementation. If you look at the performance requirements for vector, deque and list they correspond to array and list like implementations.

This differs from what the here-in discussed Object Collection Service proposes. The collection classes vector, deque, and list all map to the same interface Sequence. The different performance profiles are delivered via the implementation choice.

Algorithms

Different from other container libraries ANSI STL containers offer a very limited set of operations at the containers themselves. Instead, all higher level operations like union, find, sort, and so on are offered as so called generic algorithms. A generic algorithm is a global template function that operates on all containers - supporting the appropriate type of iterator. There are approximately 50 algorithms offered in ANSI STL.

There are:

- non-mutating sequence algorithms
- mutating sequence algorithms
- sorting and related algorithms
- generalized numeric algorithms

The basic concept here is the separation of data structures and algorithms. Instead of implementing an algorithm for each container in the library you provide a generic one operating on all containers.

If one implements a new container and ensures that an appropriate iterator type is supported one gets the respective algorithms “for free”. One may also implement new generic algorithms working on iterators only which will apply to all containers supporting the iterator type.

In addition, because the algorithms are coded as C++ global template functions, reduction of library and executable size is achieved (selective binding).

Iterators

The key concept in ANSI STL that enables flexibility of STL are Iterator classes. Iterator classes in ANSI STL are C++ pointer abstractions. They allow iteration over the elements of a container.

Their design ensures, that all template algorithms work not only on containers in the library but also on built-in C++ data type array. Algorithms work on iterators rather than on the containers themselves. An algorithm does not even “know” whether it is working with an ordinary C++ pointer or an iterator created for a container of the library.

There are:

- input iterator, output iterator
- forward iterator
- bidirectional iterator
- random access iterator
- const, reverse, insert iterators

Consideration on choice

The collection class concept as defined by the ANSI standard is designed for optimal, local use within programs written in C++. In some sense they are extensions of the language and heavily exploit C++ language features. No considerations, of course, are given to distribution of objects or language neutrality.

Some of the advantages clearly visible in a local C++ environment cannot be carried over into a distributed and language neutral environment. Some of them are even counterproductive.

In summary, the following list of issues are the reason why the ANSI collection class standard has not been considered as a basis for this proposal:

- Aiming with its design at high performance and small code size of C++ applications ANSI STL seems to have avoided inheritance and virtual functions. As no inheritance is defined, polymorphic use of the defined collection classes is not possible.
- The ANSI STL programming model of generic programming is very C++ specific one. ANSI STL containers, iterators, and algorithms are designed as C++ language extension. Containers are smooth extensions of the built-in data type array and iterators are smooth extensions of ordinary C++ pointers. Containers in the library are processed by generic algorithms via iterators in the same way as C++ arrays via ordinary pointers. Rather than subclassing and adding operations to a container one extends a container by writing a new generic algorithm. This is a programming model just introduced to the C++ world with ANSI STL and for sure not the programming model Smalltalk programmers are used to.
- As a consequence of the separation of data structures and algorithms containers in ANSI STL up to some extent expose implementation. As an example consider the two sequential containers `list` and `vector`. The algorithms `sort` and `merge` are methods of the `list` container. `vector` on the other hand can support efficient random access and therefore use the generic

algorithms sort and merge. Subsequently you do not find them as methods in the vector interface. This requires rework of clients when server implementations changes from list to vector or deque because of changing access patterns.

- The IDL concept has no notion of global (template) functions. The only conceivable way to organize the algorithms is by collecting them in artificial algorithm object(s). The selective binding advantage is lost in a CORBA environment and careful placement of the algorithm object(s) near the collection must be exercised.
- In the ANSI STL approach the reliance on generic programming as algorithms is substantial. We believe that this concept is not scalable. It is difficult to imagine a generic sort in a CORBA environment is effective without the knowledge of underlying data structures. Each access to a container has to go via an iterator mediated somehow by the underlying request broker, which is not a satisfactory situation. Object Collection Services will be used in a wide variety of environments, ranging from simple telephone lists up to complex large stores using multiple indices, exhibiting persistent behavior and concurrently accessed via Object Query Service. We do not believe that generic algorithms scale up in such environments.

B.1.1 ODMG-93

Release 1.1 of the ODMG specification defines a set of collection templates and an iterator template class.

An abstract base class `Collection<T>` is defined from which all concrete collections classes are derived. The concrete collection classes supported are `Set<T>`, `Bag<T>`, `List<T>`, `Varray<T>`. In addition an Iterator class `Iterator<T>` is defined for iteration over the elements of the collection.

`Set` and `Bag` are unordered collections and `Bag` allows multiples. `List` is an ordered collection that allows multiples. The `Varray<T>` is a one dimensional array of varying length.

`Collection<T>` offers the test `empty()` and allows to ask for the current number of elements, `cardinality()`. Further the tests `is_ordered()` and `allows_duplicates()` are offered. There is a test on whether an element is contained in a given collection. Operations for insertion, `insert_element()`, and removal, `remove_element()` are provided. Last not least there is a `remove_all()` operation.

Each of the derived classes provides an `operator==` and an `operator!=` and an operation `create_iterator()`.

A `Set<T>` is derived from `Collection<T>` and offers in addition operations `is_subset_of()`, `is_proper_subset_of()`, `is_superset_of()`, or `is_proper_superset_of()` a suite of set-theoretical operations to form the union, difference, intersection of two sets.

A `Bag<T>` offers the same interface as `Set<T>` but allows multiples.

A `List<T>` offers specific operations to retrieve or remove the first respectively last element in the list or to insert an element as first respectively last element. Retrieving, removing, and replacing an element at a given position is supported. Inserting an element before or after a given position is possible.

`Varray<T>` exposes the characteristics of a one dimensional array of varying length. An array can be explicitly re-sized. The `operator[]` is supported. The operations to find, remove, retrieve, and replace an element at a given position are supported.

An instance `Iterator<T>` is created to iterate over a given collection. The `operator=` and `operator ==` are defined. There is a `reset()` operation moving an iterator to the beginning of the collection. There is an operation `advance()` and overloaded the `operator++` to move the iterator to the next element. Retrieving and replacing the element currently “pointed to” is possible. A check on whether iteration is not yet finished is offered, `not_done()`. For convenience in iteration there is an operation `next()`, combining “check end of iteration, retrieval of an element, and moving to the next element”.

ODMG-93 structure is very similar to the proposed Object Collections Service. ODMG-93 `Set <T>` and `Bag<T>` correspond very well to `Set` and `Bag` as defined herein. `List<T>` maps one-to-one to an `EqualitySequence`. A `Varray<T>` maps to an `EqualitySequence` too. That the interfaces `List<T>` and `Varray <T>` map to the same interface in the Object Collection Service proposed reflects that `List<T>` and `Varray<T>` somehow expose the underlying kind of implementation structure assumed - namely a list like structure respectively a table like structure. In the Object Collection Service proposed the different kinds of implementation of a sequence like interface are not reflected in the interface but only in the delivered performance profile. This is the reason why `List<T>` and `Varray<T>` map to the same interface `EqualitySequence`. The `Iterator` interface maps to the top level `Iterator` interface of the iterator hierarchy of the Object Collection Service.

In summary the Object Collection Service proposed is a superset of the ODMG-93 proposed collections and iterators.

Appendix C References

C.1 List of References

OMG, *CORBAservices: Common Object Services Specification*, Volume 1, March 1996.

A

abort
 see rollback
 absolute_time 14-9
 Abstract Collection Interfaces 17-21
 Abstract interface hierarchy 17-4
 Abstract Interfaces 16-28
 Abstract RestrictedAccessCollection Interface 17-65
 Access by key 17-3
 Access Control 15-111
 access control 15-3
 Access Control Interceptor 15-154
 Access Control Model 15-19
 Access Decision Object 15-161
 Access Decision Policies 15-163
 Access Decision Time 15-155
 access identity 15-14
 Access Policies 15-21, 15-129
 Access Policies Supported by This Specification 15-22
 AccessDecision Use of AccessPolicy and RequiredRights 15-134
 accountability 15-2
 Add Type Operation 16-62
 Add_Link Operation 16-51
 Additional ObjectID 16-4
 adjudication 15-71
 Admin Interface 16-70
 Administering Security Policy 15-111
 Administration of security information 15-3
 Administration of Time 14-18
 administrative interfaces 15-51
 Administrative Model 15-71
 Administrator's Interfaces 15-123
 Administrator's View 15-44
 AlreadyBound 3-9, 3-11
 ANSI Standard Template Library 17-133
 Application Access Policies 15-63
 Application Access Policy 15-20
 application access policy 15-19
 Application Activities 15-64
 application audit policies 15-23
 Application Components 15-47
 Application Developer View 15-43
 Application Developer's Interfaces 15-84
 Application Interfaces - Security Functionality Level 1 15-201
 Application Interfaces - Security Functionality Level 2 15-201
 Application Interfaces for Non-repudiation 15-207
 application object xlii, 4-1
 Asymmetric key technology 15-38
 atomicity 10-45, 10-48, 10-52
 glossary definition 10-81
 Attribute status 14-16
 Attributes and Set Operations 16-48
 Audit Administration Interfaces 15-138
 Audit Channel Objects 15-163
 Audit Decision Objects 15-162
 Audit Event Families and Types 15-215
 audit identity 15-14
 audit objects 15-162
 Audit Policies 15-138
 Audit Services 15-164
 audit_channel 15-110

audit_needed 15-109
 audit_write 15-110
 Auditing 15-23
 Auditing Application Activities 15-64
 authenticate 15-93
 authentication 15-3
 Authentication of principals 15-92
 Authorization 15-3
 authorization_service Field 15-188
 availability 15-2

B

Bag Interface 17-62
 Bag, SortedBag 17-10
 BagFactory Interface 17-77
 Basic Time Service 14-4
 Bind Time 15-154
 Bind Time - Client Side 15-153
 Bind Time - Target Side 15-154
 Binding 15-220
 binding 15-48
 Binding and Interceptor 15-221
 Binding Handle 15-194
 BindingIterator interface 3-12
 next_n operation 3-12
 next_one operation 3-12
 Bindings and Object Reference 15-48
 Bridges 15-171

C

callback interface
 described 58
 call-back object 8-24
 cancel_timer 14-16
 CannotProceed 3-10
 Changes to Support the Current Pseudo-Object 15-230
 CLI 5-34
 Client and Target Invoke 15-224
 Client Side 15-175
 ClientSecureInvocation 15-141
 Client-Target Binding 15-220
 Collectible elements and the operations interface 17-7
 Collectible elements and type safety 17-7
 Collectible elements of key collections 17-8
 collection 11-4, 11-10
 model 11-12
 Collection factories 17-2, 17-5
 Collection Factory Interfaces 17-70
 Collection Interface 17-21
 Collection interface 11-14
 add_all_elements operation 11-17
 add_element operation 11-16
 create_iterator operation 11-18
 insert_element_at operation 11-17
 remove_all_elements 11-18
 remove_element_at operation 11-17
 replace_element_at operation 11-17
 retrieve_element_at operation 11-18
 Collection Interface Hierarchies 17-15
 Collection interfaces 17-2
 CollectionFactory and CollectionFactories Interfaces 17-71

- CollectionFactory interface 11-14
- Collections 17-2
- Combined Collections 17-10
- combined privileges delegation 15-29
- Command and Comparator Interface 17-122
- Common collection types 17-2
- Common Facilities 15-234
- common facilities xlii
- compare_time 14-10
- Complete evidence 15-67
- Component Protection 15-52
- Components 15-188
- composite delegation 15-29
- compound copy request 6-27
- compound externalization 64, 8-25
- compound life cycle 63, 9-3, 9-36, 9-37
 - and containment roles 6-42
 - and relationship service 6-37, 6-39, 6-41
 - copy operation example 6-27–6-30
 - copying, moving relationships 6-39–6-41
 - copying, moving roles 6-37–6-39
 - copying, moving, removing nodes 6-35–6-37
 - copying, moving, removing objects 6-33–6-35
- compound name 3-1, 3-2, 3-11, 3-17
- compound object 56
- compound operations 9-36
 - propagation 9-37
- Concepts 15-124
- concepts of 55
- Concrete Restricted Access Collection Interfaces 17-66
- concurrency control service
 - overview 49, 7-1
- ConcurrencyControl module
 - OMG IDL 7-8–7-9
- Confidentiality 15-17
- confidentiality 15-1
- Conformance Criteria 16-68
- Conformance Details 15-235
- Conformance Requirements for Implementation Conformance
 - Classes 16-71
- Conformance Requirements for Trading Interfaces as Server 16-69
- connect 4-18
- Connection interface 5-37
 - operations 5-37
- ConnectionFactory interface 5-37
 - operations 5-37
- Consolidated OMG IDL 14-20, 15-196, 16-74, 16-93, 16-99
- Constraint Language 16-93
- Constraint Language BNF 16-95
- Constraint Recipe Language 16-99
- consumer 4-2
- ConsumerAdmin interface 4-16, 4-17, 4-26
 - for_consumers operation 4-16
 - obtain_pull_supplier operation 4-17
 - obtain_push_supplier operation 4-17
- ContainedInRole interface 8-26
- containment relationship 9-1, 9-9
 - defining 9-49–9-50
 - example 9-23
 - overview 9-47
- ContainsRole interface 8-26
- ContextId 15-178
- continue_authentication 15-94
- Control Attributes 15-22
- Control interface 10-21
- control object 10-21, 10-27, 10-56
- Control of privileges delegated 15-27
- Control of privileges used 15-28
- Control of target restrictions 15-28
- Controls Used Before Initiating Object Invocations 15-27
- Coordinator interface 10-24
 - create_subtransaction operation 10-27
 - get_parent_status operation 10-24
 - get_status operation 10-24
 - get_top_level_status operation 10-25
 - get_transaction_name operation 10-27
 - hash_top_level_tran operation 10-26
 - hash_transaction operation 10-25
 - is_ancestor_transaction operation 10-25
 - is_descendant_transaction operation 10-25
 - is_related_transaction operation 10-25
 - is_same_transaction operation 10-25
 - is_top_level_transaction operation 10-25
 - register_resource operation 10-26
 - register_subtran_aware operation 10-26
 - rollback_only operation 10-26
- coordinator object 10-28, 10-29, 10-38, 10-39, 10-49, 10-56
 - glossary definition 10-81
- copy 15-96
- CORBA 55
 - documentation set xliii
 - object references 64
 - standard requests 4-1
- CORBA Interoperable Object Reference with Security 15-171
- CORBA Module Changes for Replaceability Conformance 15-229
- CORBA Module Changes to Support Security Level 1 15-226
- CORBA Module Changes to Support Security Level 2 15-227
- CORBA Module Deprecated Interfaces 15-231
- CORBA OMG IDL based Specification of the Trading
 - Function 16-74
- CosCompoundExternalization
 - Node interface 8-6
- CosCompoundExternalization module
 - OMG IDL 8-20–8-21
- CosCompoundExternalizationNode interface 8-5
- CosCompoundLifeCycle module
 - OMG IDL 6-30–6-33
- CosCompoundLifeCycleOperations interface 6-26
- CosConcurrencyControl module
 - overview 7-7
- CosContainment module
 - attributes and operations 9-49–9-50
 - OMG IDL 9-48
- CosEventChannelAdmin module
 - OMG IDL 4-15–4-16
- CosEventComm module
 - OMG IDL 4-8
- CosExternalization module
 - OMG IDL 8-12
- CosExternalizationContainment module
 - OMG IDL 8-26
 - see also CosCompoundExternalization module 8-26

- see also CosContainment module 8-26
- CosExternalizationReference module
 - OMG IDL 8-28
 - see also CosCompoundExternalization module 8-28
 - see also CosReference module 8-28
- CosGraphs
 - TraversalCriteria interface 6-41
- CosGraphs module 8-24
 - OMG IDL 9-39–9-41
- CosLicensingManager module
 - OMG IDL for 12-17
- CosLifeCycle module
 - OMG IDL 6-10–6-11
- CosLifeCycleContainment module
 - andCosCompoundLifeCycle and CosContainment modules 6-42
 - OMG IDL 6-42
- CosLifeCycleLifeCycleObject interface 6-37
- CosLifeCycleReference module
 - OMG IDL 6-44
- CosNaming module
 - OMG IDL 3-6–3-8
- CosPersistenceDDO module 5-31–5-33
 - OMG IDL 5-31
- CosPersistenceDS_CLI module
 - OMG IDL 5-35–5-36
- CosPersistencePDS module
 - OMG IDL 5-20
- CosPersistencePDS_DA module 5-21–5-29
 - OMG IDL 5-22
- CosPersistencePID module
 - OMG IDL 5-9
- CosPersistencePO module
 - OMG IDL 5-12
- CosPropertyService 13-4
- CosQuery module
 - OMG IDL for 11-23
- CosQueryCollection module
 - OMG IDL for 11-14
- CosReference module
 - attributes and operations 9-50–9-51
- CosRelationships module
 - OMG IDL 9-20–9-23
- CosStream module
 - OMG IDL 8-15–8-16
- CosTime 14-4, 14-5
- CosTransactions module
 - datatypes defined by 10-15
 - OMG IDL 10-65–10-68
- CosTSInteroperation module
 - PIDL 10-58, 10-69
- CosTSPortability module
 - PIDL 10-69
- CosTypedEventComm module
 - OMG IDL 4-22
- Creating iterators 17-27
- Credentials 15-56, 15-96
- cryptographic keys 15-4
- Curren 15-217
- Current 15-56
- Current interface 10-37
- Cursor interface 5-38
 - operations 5-38
- CursorFactory interface 5-38
 - operations 5-38
- D**
- DA protocol 5-19
 - compared to ODMG-93 protocol 5-30
- DADO 5-26
- DAObject interface 5-24
 - boolean dado_same (inDAObject d) operation 5-24
 - DataObjectID dado_oid() operation 5-24
 - PID_DA dado_pid() operation 5-24
 - void dado_free() operation 5-24
 - void dado_remove() operation 5-24
- DAObjectFactory interface 5-24
 - DAObjectFactory create() operation 5-25
- DAObjectFactoryFinder interface 5-25
 - find_factory operation 5-25
- Data Definition Language
 - see DDL
- data objects 5-27, 5-28
 - and dynamic access to attributes 5-28
- Data Types 15-86
- datastore 5-7, 5-13, 5-17, 5-18, 5-26, 5-34, 5-43
 - and DDO protocol 5-31
- Datastore_CLI interface 5-40
 - and CLI 5-43
 - operations 5-41–5-43
- DCE Association Options Reduction Algorithm 15-193
- DCE Authorization Services 15-191
- DCE RPC Authentication Services 15-192
- DCE RPE Protection Levels 15-192
- DCE Security Parameters 15-193
- DCE Security Services 15-191
- DCEAuthorizationDCE 15-191
- DCEAuthorizationName 15-191
- DCEAuthorizationNone 15-191
- DCE-CIOP 15-186
- DCE-CIOP Operational Semantic 15-192
- DCE-CIOP with Security 15-185
- DDL 5-21, 5-26, 5-27, 5-28
- DDO
 - storing,restoring,deleting 5-40
- DDO interface
 - attributes 5-32
 - short add_data() operation 5-32
 - short add_data_property (in short data_id) operation 5-32
 - short get_data_count() operation 5-32
 - short get_data_property_count (in short data_id) operation 5-33
 - void get_data operation 5-33
 - void get_data_property operation 5-33
 - void set_data operation 5-33
 - void set_data_property operation 5-33
- DDO protocol 5-19, 5-30
- define 13-10, 13-16
- Defining 13-9, 13-15
- defining and modifying properties 13-9
- Delegation 15-25, 15-113
- Delegation Options 15-30
- Delegation Policies 15-140

Index

- Delegation Schemes 15-27
 - Delegation State 15-134
 - delete 9-30, 13-12, 13-13
 - Deleting 13-12
 - deleting properties 13-12
 - Deque 17-14
 - DequeFactory Interface 17-83
 - Dequeue Interface 17-67
 - Describe Link Operation 16-52
 - Describe Operation 16-41
 - Describe Proxy Operation 16-58
 - Describe Type Operation 16-65
 - design goals, of event service interfaces 48
 - destroy 3-18
 - destroy operation 3-13
 - Destroying 13-21
 - Destroying a collection 17-27
 - destroying the iterator 13-20, 13-21
 - Determining 13-14
 - determining defined property 13-14
 - direct access protocol
 - see PDS_DA protocol
 - direct attribute protocol
 - see DA protocol
 - distributed objects 6-3
 - Domain 15-218
 - Domain Management 15-125
 - Domain Manager 15-126
 - Domain Managers 15-74
 - Domain objects 15-49
 - DomainAccessPolicy 15-132, 15-136
 - DomainAccessPolicy Use of Privilege Attributes 15-133
 - DomainAccessPolicy Use of Rights and Rights Families 15-134
 - Domains 15-33, 15-132
 - Domains and Interoperability 15-38
 - Domains at Object Creation 15-73
 - dynamic data object protocol
 - see DDO protocol
 - Dynamic Property Evaluation interface 16-67
 - Dynamic Property Module 16-88
 - DynamicAttributeAccess interface 5-28
 - any attribute_get(in string name) operation 5-28
 - AttributeNames attribute_names() operation 5-28
 - void attribute_set(in string name, in any value) operation 5-28
 - E**
 - edge structure 9-46
 - EdgeIterator interface 9-47
 - destroy operation 9-47
 - next_n operation 9-47
 - next_one operation 9-47
 - encryption 15-17
 - End User View 15-43
 - Enhancements to the CORBA Module 15-226
 - Enterprise Management View 15-42
 - Enum ComparisonType 14-7
 - Enum EventStatus 14-14
 - Enum OverlapType 14-7
 - Enum TimeComparison 14-7
 - Enum TimeType 14-14
 - Environment Domains 15-52
 - Equality collection 17-3
 - EqualityCollection Interface 17-37
 - EqualityIterator Interface 17-110
 - EqualityKeyCollection Interface 17-50
 - EqualityKeyIterator Interface 17-111
 - EqualityKeySortedCollection Interface 17-55
 - EqualityKeySortedIterator Interface 17-116
 - EqualitySequence 17-11
 - EqualitySequence Factory Interface 17-81
 - EqualitySequence Interface 17-64
 - EqualitySequentialCollection Interface 17-55
 - EqualitySequentialIterator Interface 17-117
 - EqualitySortedCollection Interface 17-53
 - EqualitySortedIterator Interface 17-114
 - Establishing a Security Association 15-168
 - Establishing Credentials 15-54
 - Establishing the Binding and Interceptors 15-221
 - event channel 48, 56, 57, 4-5, 4-13
 - adding consumers 4-16
 - adding consumers to 4-17
 - adding consumers to typed 4-26
 - adding pull consumer to typed 4-28
 - adding pull consumers to 4-18
 - adding pull suppliers to 4-18
 - adding push consumers to 4-19
 - adding push suppliers to 4-17
 - adding push suppliers to typed 4-28
 - adding suppliers 4-16
 - adding suppliers to 4-17
 - adding suppliers to typed 4-27
 - and CORBA requests 4-10
 - decoders 4-31
 - defined 4-2, 4-10
 - encoders 4-31
 - filtering 4-28–4-29
 - implementing typed 4-30–4-31
 - sample use 4-32–4-33
- event communication
 - mixed 4-11
 - multiple 4-12
 - pull model 48, 4-2, 4-7, 4-11
 - push model 48, 4-2, 4-6, 4-10
 - typed pull model 4-20
 - typed push model 4-19
- event consumer 4-2, 4-6, 4-10
 - proxy 4-13
- Event Service 15-233
- event service
 - and CORBA scoping 4-5
 - and license service 12-13, 12-15
 - design goal of interfaces 48
 - overview 48, 4-1
- event supplier 4-2, 4-6, 4-10
 - proxy 4-13
- event_time 14-17
- EventChannel interface 56, 4-13, 4-16
- exception 4-27
- Exceptions 16-23
 - Additional Exceptions for Link Interface 16-26
 - Additional Exceptions for Lookup Interface 16-24
 - Additional Exceptions for Proxy Offer Interface 16-27

- Additional Exceptions For Register Interface 16-25
 - For CosTrading module 16-23
 - exceptions
 - described 58
 - InvalidName 3-10
 - Exceptions and Type Definitions 17-19
 - export 16-2
 - Export Operation 16-39
 - Export Proxy Operation 16-55
 - Exporter 16-4
 - Exporter Policies 16-18
 - Extended Time Service 14-26
 - Extension to the Use of Current 15-217
 - Extensions to CORBA for Domains and Policies 15-218
 - Extensions to Object Interfaces for Security 15-218
 - Extensions to the Object Interface 15-127
 - External Security Services 15-164
 - externalization
 - defined 8-1
 - externalization service
 - and compound life cycle 8-6
 - and inheritance and use of objects 8-7
 - and life cycle service 64
 - and persistent object service 8-17
 - and relationship service 64, 8-5, 8-24
 - and transaction service 8-17
 - interface summary 8-10
 - overview 50
 - externalizing a node 8-21
 - externalizing a relationship 8-23
 - externalizing a role 8-22
- F**
- Facilities Used on Accepting Object Invocations 15-30
 - factory finder 6-7, 6-13, 6-21, 8-3
 - factory keys
 - and kind field 6-14, 6-16
 - factory object 48, 6-4
 - definition 6-18
 - FactoryFinder interface 6-8, 6-13–6-14
 - find_factories operation 6-13
 - Features (security) 15-92
 - Federated Policy Domains 15-35
 - Federated query example 16-19
 - FileStreamFactory interface 8-8, 8-12, 8-13
 - create operation 8-13
 - Final target 15-26
 - Finding Domain Managers 15-74
 - Finding the Policies 15-74
 - Finding What Security Facilities Are Supported 15-217
 - framework 11-10
 - Friendly Time Object 14-26
 - Full-service Trader 16-73
 - Fully Describe Type Operation 16-65
 - Function Interfaces 17-3, 17-118
 - Functional Interfaces 16-30
- G**
- General Security Data Module 15-196
 - generic factory
 - criteria parameters 6-17–6-18
 - generic factory interface 6-5
 - GenericFactory interface 6-14–6-18, 6-22
 - and criteria parameter 6-17
 - and criteria parameters 6-17
 - create_object operation 6-15, 6-17
 - supports operation 6-16
 - get 13-11, 13-12, 13-15, 13-18
 - get_active_credentials 15-102
 - get_all_properties 13-12
 - get_all_property_names 13-11
 - get_attributes 15-99, 15-105
 - get_component operation 3-16
 - get_credentials 15-107
 - get_number_of_properties 13-11
 - get_policy 15-103, 15-108
 - get_properties 13-11
 - get_property_value 13-11
 - get_security_features 15-97, 15-102
 - get_security_mechanisms 15-103
 - get_security_names 15-104
 - Getting 13-17
 - global identifier 58
 - Goals
 - Consistency 15-4
 - Scalability 15-4
 - Goals of Secure DCE-CIOP 15-185
 - graphical notation 57
 - graphs of related objects 9-3
 - copying to 6-33
 - creating traversal criteria for 8-24
 - destroying 6-35
 - examples 9-33
 - moving 6-34
 - removing 6-34
 - traversal of 9-35, 9-37
 - traversing 9-36
 - Guidelines for a Trustworthy System 15-245
- H**
- Handling Multiple Credentials 15-56
 - Heap 17-11
 - Heap Interface 17-64
 - HeapFactory Interface 17-82
- I**
- IDAPI standard 5-34
 - Identification 15-3
 - Identity domains 15-37
 - Immediate invoker 15-26
 - Implementation-Level Security Object Interfaces 15-155
 - Implementor's Security Interfaces 15-147
 - Implementor's View of Secure Invocations 15-76
 - Implementor's View of Secure Object Creation 15-81
 - Implications of Assurance 15-226
 - import 16-2
 - ImportAttributes 16-29
 - Importer 16-4
 - Importer Policies 16-17
 - Initiator 15-26
 - Integrity 15-17
 - integrity 15-1

Index

- Interceptor 15-148
- Interceptor Interfaces 15-150, 15-223
- Interceptors 15-219, 15-221
- Interface Changes Required for Interceptors 15-225
- Interface Hierarchies 17-15
- interface inheritance.see subtyping
- interface repository 61
- Interfaces 15-92
- Intermediate 15-26
- Intermediate Objects in a Chain of Objects 15-60
- internalization
 - object's model 8-5
- internalizing a node 8-21, 8-22
- internalizing a relationship 8-23
- internalizing a role 8-23
- Interoperability 15-225
- Interoperability Model 15-166
- Interoperating between ORB Technology Domains 15-39
- Interoperating between Security Policy Domains 15-170
- Interoperating between Security Technology Domains 15-39
- Interoperating between Underlying Security Services 15-170
- Interoperating with Multiple Security Mechanisms 15-169
- interval 14-10
- InvalidName exception 3-10
- Invocation Delegation Policy 15-144
- Invocation Time Policies 15-152
- IOR Security Components for DCE-CIOP 15-186
- is_valid 15-99
- Iterating over a collection 17-26
- Iterator Hierarchy 17-18
- Iterator interface 11-14
 - any next operation 11-18
 - boolean more operation 11-19
 - void reset operation 11-19
- Iterator Interfaces 17-3, 17-84
- Iterators 17-5
- Iterators and performance 17-6, 17-85
- Iterators and support for generic programming 17-84
- Iterators as pointer abstraction 17-84

K

- Key collection 17-3
- Key collections 17-8
- KeyBag Interface 17-57
- KeyBag, KeySortedBag 17-11
- KeyBagFactory Interface 17-75
- KeyIterator Interface 17-108
- KeySet Interface 17-57
- KeySet, KeySortedSet 17-12
- KeySetFactory Interface 17-75
- KeySortedBag Interface 17-63
- KeySortedBagFactory Interface 17-78
- KeySortedCollection Interface 17-51
- KeySortedIterator Interface 17-112
- KeySortedSet Interface 17-62
- KeySortedSetFactory Interface 17-78

L

- Legal Property Value Types 16-94
- library names
 - PIDL operations 3-18

- license service
 - and event service 12-13, 12-15
 - and life cycle service 12-19
 - and properties service 12-23
 - and relationship service 12-26
 - and security service 12-26
 - example implementation 12-27
 - exceptions 12-19
 - overview 12-8
 - sample implementation 12-14
- LicenseServiceManager interface 12-13, 12-17
 - check_use operation 12-13
 - end_use operation 12-13
 - obtain_producer_specific_license_service operation 12-19, 12-27
 - start_use operation 12-13
- licensing attributes
 - examples of 12-24
- life cycle service
 - and license service 12-19
 - and naming service 63, 6-15
 - and relationship service 63
 - client's model 6-4
 - overview 48, 6-1, 6-21
- LifeCycleObject interface 48, 6-6, 6-11–6-13, 6-22, 6-25
 - and criteria parameter 6-17
 - copy operation 6-11
 - move operation 6-12
 - NoFactory exception for copy operation 6-11
 - remove operation 6-13
- Link 16-49
- Link Creation Policies 16-18
- Link Interface 16-70
- Link Traversal Control 16-18
- LinkAttributes 16-30
- Linked Trader 16-72
- Linking to External Security Services 15-164
- Linking Traders 16-3
- Links 16-11
- List Offers Operation 16-48
- List Proxies Operation 16-48
- List Types Operation 16-64
- Listing 13-11
- listing and getting properties 13-11
- LName interface 3-3, 3-15
 - delete_component operation 3-17
 - destroy operation 3-16
 - equal operation 3-17
 - insert_component operation 3-16
 - less_than operation 3-17
 - num_components operation 3-17
- LNameComponent interface 3-3, 3-13, 3-15
 - get_id operation 3-15
 - get_kind attribute 3-3
 - get_kind operation 3-15
 - set_id operation 3-15
 - set_kind operation 3-15
- LockCoordinator interface 7-9
 - drop_locks operation 7-10
- locks 50, 61, 7-1, 7-2–7-7
 - and nested transactions 7-6

- intention read and write 7-4
- mode compatibility 7-5
- multiple possession semantics 7-5
- read,write,upgrade 7-4
- transaction-duration 7-6
- LockSet interface 7-9, 7-10–7-11
 - change_model operation 7-11
 - get_coordinator operation 7-11
 - lock operation 7-11
 - try_lock 7-11
 - unlock operation 7-11
- LockSetFactory interface 7-13
 - create operation 7-13
 - create_related operation 7-13
 - create_transactional operation 7-13
 - create_transactional_related operation 7-13
- Lookup 16-30
- Lookup Interface 16-69
- M**
- Making a Secure Invocation 15-58
- Managed Iterator Model 17-85
- Managed iterators 17-6
- Managing Security Environment Domains 15-41
- Managing Security Policy Domains 15-40
- Managing Security Technology Domains 15-41
- Map Interface 17-57
- Map, SortedMap 17-12
- MapFactory Interface 17-76
- Mask Type Operation 16-66
- MD5 message digest algorithm 12-30
- Message Definitions 15-179
- Message Protection 15-17, 15-154
- Message protection domains 15-37
- Message-Level Interceptors 15-149, 15-223
- Messages 15-18, 15-168
- messages 15-70
- meta-policy 15-13
- Modify Link Operation 16-53
- Modify Operation 16-42
- MTCompleteEstablishContext 15-179
- MTContinueEstablishContext 15-180
- MTDiscardContext 15-180
- MTEstablishContext 15-179
- MTMessageError 15-181
- MTMessageInContext 15-181
- Multiple Credentials 15-56
- Multiple Security Mechanisms 15-169
- N**
- name 3-2
 - binding 3-1
 - binding operations 3-8
 - component attributes 3-2
 - components 3-2
 - compound 3-2
 - resolution 3-1
 - simple 3-2
 - structure 3-18
- name binding 3-1
- name component
 - attributes 3-15
- names library 47, 3-3, 3-13
 - PIDL 3-13–3-14
- namespace administration 3-5
- name-to-object association 3-1
- naming context 47, 3-1, 3-5, 3-6
 - and property lists 59
 - deleting 3-11
- naming graph 3-1
 - example 3-2
- Naming Service 15-233
- naming service
 - and internationalization 3-3, 3-6
 - design of 3-4
 - overview 47
- NamingContext interface 3-8, 3-13, 3-18
 - bind operation 3-8
 - bind_context operation 3-9
 - bind_new_context operation 3-11
 - destroy operation 3-11
 - list operation 3-12
 - new_context operation 3-11
 - rebind operation 3-8
 - rebind_context operation 3-9
 - resolve operation 3-9
 - unbind operation 3-10
- nested queries 11-20
- nested transaction 64
- new_interval 14-12
- new_universal_time 14-12
- next 13-19, 13-20
- no delegation 15-28
- Node interface 6-35, 9-35, 9-44
 - add_role operation 9-45
 - copy operation 6-35
 - externalize_node operation 8-21
 - internalize_node operation 8-21, 8-22
 - move operation 6-36
 - related_object attribute 9-45
 - remove operation 6-37
 - remove_role operation 9-46
 - roles_of_node attribute 9-45
 - roles_of_type operation 9-45
- NodeFactory interface 9-46
 - create_node operation 9-46
- nodes
 - creating 9-46
- NoFactory 6-40
- Non-repudiation 15-3, 15-31, 15-66, 15-115, 15-163
- Non-repudiation credentials and policies 15-66
- non-repudiation evidence 15-31
- non-repudiation for receipt of messages 15-70
- non-repudiation policy 15-31
- Non-repudiation Policy Management 15-145
- Non-repudiation Service Data Types 15-116
- Non-repudiation Service Operations 15-117
- Non-repudiation services 15-32
- non-repudiation services 15-67
- non-repudiation services for adjudication 15-71
- NoProtection 15-191
- NotCopyable 6-40

Index

NotMovable 6-40
NotRemovable 6-37

O

Object Interfaces for Security 15-218
Object Invocation Access Policy 15-20
Object Management Group xli
 address of xliii
object model xliii
Object Reference 15-100
object request broker xlii
Object Security Services 15-49
object service
 context xlii
 specification defined xliii
Object System Implementor's View 15-45
Objects 15-60
ODBC standard 5-34
ODMG-93 17-136
ODMG-93 protocol 5-19, 5-30, 5-43, 10-79
 integration with transaction service 10-80
Offer Id Iterator 16-45
Offer Identifier 16-9
Offer Iterator 16-35
Offer Selection 16-9
OMG 13-3
OMG Constraint Language BNF 16-93
OMG Constraint Recipe Language 16-99
OMG IDL xliii, 56, 3-3
OMG Trading Function Module 16-74
Operation Access 15-75
operational interfaces 15-50
Operational Semantics 15-175
OperationFactory interface
 create_compound_operations operation 6-33
operations 3-15
Operations Interface 17-7, 17-118
Operations interface 6-33
 copy operation 6-33
 destroy operation 6-35
 move operation 6-34
 remove operation 6-34
OperationsFactory interface 6-33
Operator Restrictions 16-94
OQL-93 Basic Query Language 11-7
OQL-93 Query Language 11-6
ORB Core and ORB Services 15-219
ORB Interoperability 15-225
ORB Security Services 15-76
ORB Services 15-47, 15-219
ORB Services and Interceptors 15-148
Ordering of elements 17-3
OSI TP protocol 10-76
 exported transactions 10-78
 imported transactions 10-77
 transaction identifiers 10-77
Overlapping Policy Domains 15-36
overlaps 14-11
override_default_credentials 15-101
override_default_mechanism 15-103
override_default_QOP 15-101

P

PDS 5-43
 see persistent data service
PDS interface 5-19–5-20
 and DA protocol 5-25
 PDS connect operation 5-20
 void delete operation 5-20
 void disconnect operation 5-20
 void restore operation 5-20
 void store operation 5-20
PDS_ClusteredDA interface 5-29
 ClusterID cluster_id() operation 5-29
 ClusterIDs clusters_of() operation 5-29
 PDS_ClusteredDA copy_cluster(in PDS_DA source)
 operation 5-29
 PDS_ClusteredDA create_cluster(in string kind) operation 5-29
 PDS_ClusteredDA open_cluster(in ClusterID cluster)
 operation 5-29
 string cluster_kind() operation 5-29
PDS_DA interface 5-21, 5-25
 and ODMG-93 protocol 5-30
 DAObject get_data() operation 5-25
 DAObject lookup(in DAObjectID id) operation 5-25
 DAObjectFactoryFinder data_factories() operation 5-26
 PID_DA get_object_pid(in DAObject dao) operation 5-25
 PID_DA get_pid() operation 5-25
 void set_data(in DAObject new_data) operation 5-25
PDS_DA protocol 5-21, 5-25
 and data objects 5-26
persistent data service 5-7, 5-17, 5-26, 5-27
 overview 5-18
persistent data service interface
 see PDS interface
persistent identifier 5-7
 compared to CORBA object reference 5-9
persistent object interface
 see PO interface
persistent object manager 5-11
 and PO interface 5-13
 purpose of 5-17
Persistent Object Service 15-233
persistent object service
 and clients 5-5
 and CORBA accessor operations 5-27
 and CORBA Dynamic Invocation interface 5-28
 and CORBA persistent reference handling 5-2, 5-3
 and datastore 5-6
 and factory finders 5-25
 and factory objects 5-24
 and object implementation 5-6
 and persistent data service 5-6
 and query service 5-42
 and transaction service 5-42
 overview 49
PID
 see persistent identifier
PID interface 5-8
PID_CLI interface 5-38
 attributes 5-39
PID_DA interface 5-23
 DAObjectID attribute 5-23

- PIDL 67, 3-3
 - PO interface 5-12–5-13
 - ... connect operation 5-13
 - void delete operation 5-13
 - void disconnect operation 5-13
 - void restore operation 5-13
 - void store operation 5-13
 - Policies 15-74, 15-218
 - Policy Details 15-75
 - Policy Domain Hierarchies 15-34
 - Policy domain managers 15-50
 - Policy Domains 15-124
 - POM interface
 - ...connect operation 5-16
 - OMG IDL 5-16
 - void delete operation 5-16
 - void disconnect operation 5-16
 - void restore operation 5-16
 - void store operation 5-16
 - Preferences 16-10
 - Principal Authentication 15-163
 - Principal authenticator 15-55
 - principal_authenticator 15-108
 - Principals 15-92
 - Principals and Their Security Attributes 15-14
 - PriorityQueue 17-14
 - PriorityQueue Interface 17-69
 - PriorityQueueFactory Interface 17-83
 - Privilege Attributes 15-21, 15-133
 - Privilege Delegation 15-26
 - privilege delegation 15-25
 - ProducerSpecificLicenseService interface 12-13, 12-14, 12-17
 - check_use operation 12-20, 12-21, 12-27
 - end_use operation 12-20, 12-27
 - start_use operation 12-20, 12-27
 - proof of delivery 15-32
 - proof of origin 15-32
 - propagation 10-30–10-34, 10-38, 10-41, 10-55, 10-59, 10-62
 - deep 9-37
 - glossary definition 10-83
 - none 9-38
 - shallow 9-37
 - propagation context 67
 - PropagationCriteriaFactory interface 8-24
 - create operation 6-41, 8-24
 - Properties 16-7
 - Dynamic 16-8
 - modifiable 16-8
 - properties
 - defining and modifying with modes 13-15
 - properties service
 - and license service 12-23
 - PropertiesIterator 13-19
 - PropertiesIterator interface 13-19
 - Property 13-23
 - property list 4-1, 12-23
 - property modes
 - getting and setting 13-17
 - property service
 - object classification 13-1
 - object usage count 13-1
 - Property service IDL 13-23
 - PropertyNamesIterator 13-20
 - PropertyNamesIterator interface 13-20
 - PropertySet 13-9
 - PropertySetDef 13-14
 - PropertySetDef interface 13-14
 - PropertySetDefFactory 13-22
 - PropertySetDefFactory interface 13-22
 - PropertySetFactory 13-21
 - PropertySetFactory interface 13-21
 - Protecting Messages 15-168
 - Protection boundaries 15-53
 - Protocol Enhancements 15-171
 - proxies and Time 14-23
 - Proxy 16-54
 - Proxy Interface 16-70
 - Proxy Trader 16-73
 - ProxyPullConsumer interface 4-18
 - connect_pull_supplier operation 4-18
 - ProxyPullSupplier 4-18
 - ProxyPullSupplier interface 4-3, 4-18
 - connect_pull_consumer operation 4-18
 - ProxyPushConsumer interface 4-3, 4-17
 - connect_push_supplier operation 4-18
 - disconnect_push_supplier operation 4-18
 - ProxyPushSupplier interface 4-19
 - connect_push_consumer operation 4-19
 - pseudo object 67, 3-3, 3-13, 3-18
 - creating library name 3-14
 - Public 15-14
 - Public key technology 15-38
 - PullConsumer interface 4-3, 4-10, 4-21
 - disconnect_pull_consumer operation 4-7
 - PullSupplier interface 56, 4-7, 4-9
 - disconnect_pull_supplier operation 4-7, 4-10
 - pull operation 4-9
 - try_pull operation 4-9
 - PushConsumer interface 56, 4-6, 4-8, 12-27
 - disconnect_push_consumer operation 4-9
 - push operation 4-8
 - PushSupplier interface 4-3, 4-9
 - disconnect_push_supplier operation 4-7, 4-9
- Q**
- quality of service 56, 4-3, 4-4, 4-6, 4-12
 - query collection 11-10
 - query evaluator 11-3
 - defined 11-19
 - Query Example 16-19
 - query framework 11-10
 - query framework interfaces
 - overview of 11-10
 - Query interface
 - execute operation 11-26
 - get_result operation 11-27
 - get_status operation 11-27
 - prepare operation 11-26
 - readonly attribute 11-26
 - query object
 - defined 11-21
 - Query Operation 16-31

Index

- query service
 - and transaction service 11-2
 - list of interfaces for 11-23
 - Query Trader 16-71
 - queryable collection
 - defined 11-20
 - QueryableCollection interface 11-25
 - QueryEvaluator interface
 - attributes for 11-25
 - QueryManager interface
 - create operation 11-26
 - Queue 17-15
 - Queue Interface 17-66
 - QueueFactory Interface 17-82
- R**
- RACollectionFactory and RACollectionFactories Interfaces 17-74
 - ReadOnly attribute inaccuracy 14-9
 - ReadOnly attribute tdf 14-9
 - ReadOnly attribute time 14-9
 - ReadOnly attribute time_interval 14-10
 - ReadOnly attribute utc_time 14-9
 - received_credentials 15-107
 - received_security_features 15-107
 - Recipe Syntax 16-99
 - recoverable object 10-5
 - and nested transactions 10-29
 - recoverable server 10-6, 10-39
 - glossary definition 10-83
 - implementing 10-35
 - RecoveryCoordinator interface 10-27
 - replay_completion operation 10-27
 - reference model xlii
 - reference relationship 9-1, 9-9
 - defining 9-50–9-51
 - overview 9-47
 - reference restriction 15-25
 - refresh 15-99
 - Register 16-36
 - register 14-17
 - Register Interface 16-69
 - Relation Interface 17-61
 - Relation, SortedRelation 17-13
 - RelationFactory Interface 17-76
 - relationship
 - and nodes, defined 9-35
 - creating 9-24
 - destroying 9-26
 - determining roles 9-26
 - Relationship between implementation objects for associations 15-80
 - relationship between main objects 15-82
 - relationship factory attributes 6-42, 6-45
 - Relationship interface 6-39, 8-23, 8-26, 9-25
 - copy operation 6-39
 - destroy operation 9-26
 - externalize_role operation 8-23
 - internalize_relationship operation 8-23
 - life_cycle_propagation operation 6-41
 - move operation 6-40
 - named_roles attribute 9-26
 - propagation_for operation 8-24
 - relationship service
 - and base level operations 9-17
 - and cardinality 9-2, 9-18
 - and compound life cycle 9-3
 - and containment relationship 9-47–9-48
 - and CORBA object references 64
 - and degree 9-2
 - and entity 9-2
 - and levels of service 9-3, 9-7–9-10
 - and license service 12-26
 - and reference relationship 9-47–9-48
 - and semantics 9-2
 - and type 9-1, 9-14
 - attribute and operation rationale 9-15
 - interface summary 9-11–9-13
 - overview 50
 - Relationship to Object Services and Common Facilities 15-232
 - Relationship to Other Relevant Standards 17-133
 - RelationshipFactory interface 9-23
 - create operation 9-24
 - degree attribute 9-25
 - named_role_types attribute 9-25
 - relationship_type attribute 9-25
 - RelationshipIterator interface 9-32
 - destroy operation 9-32
 - next_n operation 9-32
 - next_one operation 9-32
 - relationships
 - and defining role attributes 9-30
 - and operations on roles 9-26–9-30
 - containment 8-25
 - reference 8-25
 - Remove Link Operation 16-52
 - Remove Type Operation 16-64
 - Replaceable Security Service 15-163
 - Replaceable Security Services 15-78
 - Replacing Access Decision Policies 15-163
 - Replacing Audit Services 15-164
 - Representation of Literals 16-95
 - representation of Time 14-1
 - Request-Level Interceptors 15-149, 15-222
 - required_rights_object 15-108
 - RequiredRights 15-134
 - RequiredRights Interface 15-130
 - Resetting 13-19
 - resetting
 - position in an iterator 13-20
 - resetting position in iterator 13-20
 - Resetting the position in an iterator 13-19
 - Resolve Operation 16-45
 - Resource interface 10-27
 - commit operation 10-29
 - commit_one_phase operation 10-29
 - forget operation 10-29
 - prepare operation 10-28
 - rollback operation 10-29
 - resource manager 10-9, 10-64, 10-74
 - mappings to 10-72
 - resource object
 - defined 10-5

- Restricted Access Collection Interfaces 17-65
- Restricted Access Collections 17-4, 17-14
- RestrictedAccessCollection Interface 17-65
- Retrieval 13-15
- retrieval of PropertySet constraints 13-15
- Rights 15-22, 15-129
- Rights Families 15-130, 15-134
- Rights Families and Values 15-215
- RM
 - see resource manager
- role factory attributes 6-43, 6-45
- Role interface 6-37, 8-22, 9-26, 9-46
 - check_minimum_cardinality operation 9-29
 - copy operation 6-38
 - destroy operation 9-29
 - destroy_relationships operation 9-28
 - externalize_propagation operation 8-23
 - externalize_role operation 8-22
 - get_edges operation 9-47
 - get_other_related_object operation 9-27
 - get_other_role operation 9-27
 - get_relationships operation 9-28
 - how_many operation 9-28
 - internalize_role operation 8-23
 - life_cycle_propagation operation 6-39
 - link operation 9-29
 - move operation 6-38
 - related_object attribute 9-27
 - unlink operation 9-30
- RoleFactory interface 9-27, 9-30
 - and max_cardinality attribute 9-31
 - and min_cardinality attribute 9-31
 - and role_type attribute 9-31
 - create_role operation 9-30
 - related_object_type attribute 9-32
- roles
 - and cardinality 9-29, 9-31
- rollback
 - glossary definition 10-84
- S**
- Scoping Policies 16-13
- SD interface 5-11
- SECIOP 15-178
- SECIOP Message Header 15-177
- SECIOP Protocol State Tables 15-182
- Secure DCE-CIOP 15-186
- Secure DCE-CIOP Operational Semantics 15-192
- Secure Interoperability 15-243
- Secure Interoperability Bridges 15-171
- Secure Inter-ORB Protocol (SECIOP) 15-177
- Secure Invocation and Delegation Policies 15-140
- Secure Invocation Interceptor 15-152
- Secure Object Invocations 15-15, 15-168
- Secure Time 14-18
- secure_universal_time 14-12, 14-17
- SecureUniversalTime 14-3
- Securing the Binding Handle to the Target 15-194
- Security 15-1
 - Goals 15-3
- Security Administration Interfaces 15-205
- Security and Interoperability 15-165
- Security Architecture 15-42
- Security Association 15-168
- security association 15-16
- Security at the Target 15-59
- Security Attributes 15-57
- Security Audit 15-109
- Security auditing 15-3
- Security Components of the IOR 15-172
- Security context 15-80
- Security Context Object 15-158
- Security Data Modul 15-196
- security domains 15-4
- Security environment domain 15-33
- Security Environment Domains 15-36, 15-41
- Security Facilities 15-217
- Security Features 15-3, 15-92
- Security Functionality Conformance 15-85
- Security Functionality Level 1 15-85, 15-236
- Security Functionality Level 2 15-85, 15-238
- Security Information in the Object Reference 15-167
- Security Interceptors 15-150
- Security Mechanism Types 15-169
- Security Mechanisms 15-63, 15-216
- Security Mechanisms for Secure Object Invocations 15-168
- security name 15-15
- Security Object Models 15-54
- Security of communication 15-3
- Security Operations on Current 15-104
- Security Policies 15-65, 15-72, 15-125, 15-128
- Security policies and domain objects 15-49
- Security Policy 15-76
- Security Policy Domains 15-34, 15-40, 15-170
- Security Reference Model 15-12
- Security Replaceability 15-241
- Security Replaceability Ready 15-85
- Security Replaceable Service Interfaces 15-210
- Security Service 15-1
 - security service
 - and license service 12-26
 - security specification 15-2
- Security Technology 15-51
- Security technology domain 15-33
- Security Technology Domains 15-37, 15-41
- see also data objects
- Selecting Security Attributes 15-57
- Selection of ORB Services 15-47
- Send and Receive Message 15-224
- sending Time across the network 14-23
- Sequence 17-13
- Sequence Interface 17-64
- SequenceFactory Interface 17-81
- SequentialCollection Interface 17-31
- Service Offers 16-7
- Service Type Repository 16-59
- Service Type Repository Module 16-89
- set 13-18, 13-19
- set_security_features 15-97
- Set, SortedSet 17-13
- set_credentials 15-106
- set_data 14-16

Index

- set_privileges 15-58, 15-98
 - set_security_features 15-58
 - set_timer 14-16
 - SetFactory Interface 17-77
 - Setting Security Policy Details 15-75
 - simple delegation 15-28
 - simple name 3-2
 - Simple Trader 16-72
 - SNA LU protocol 10-76, 10-78
 - incoming communication 10-79
 - outgoing communication 10-79
 - transaction identifiers 10-78
 - SortedBag Interface 17-64
 - SortedCollection Interface 17-37
 - SortedIterator Interface 17-112
 - SortedMap Interface 17-63
 - SortedMapFactory Interface 17-79
 - SortedRelation Interface 17-63
 - SortedRelationFactory Interface 17-79
 - SortedSet Interface 17-63
 - SortedSetFactory Interface 17-80
 - source of Time 14-2
 - spans 14-11
 - Specific ORB Security Services and Replaceable Security Services 15-78
 - Specifying Delegation Options 15-30
 - Specifying Use of Rights for Operation Access 15-75
 - SQL Query Language 11-6
 - Stack 17-15
 - Stack Interface 17-67
 - StackFactory Interface 17-83
 - Stand-alone Trader 16-72
 - Standard Data Type 15-213
 - Standardized Capability Supported Policies 16-15
 - Stream interface 8-12, 8-13
 - begin_context operation 8-14
 - end_context operation 8-14
 - externalize operation 8-13
 - flush operation 8-14
 - internalize operation 8-13, 8-14
 - internalize_from_stream operation 8-15
 - stream object
 - creating 8-12, 8-13
 - data format 8-29–8-31
 - externalizing 8-13
 - externalizing group 8-14
 - internalizing 8-13, 8-14
 - stream service 8-3
 - and begin_context request 8-3
 - and externalize_to_stream request 8-3, 8-4
 - and internalize_from_stream request 8-3
 - and readonly key attribute 8-3
 - Streamable interface 8-4, 8-7, 8-17
 - externalize_to_stream operation 8-18
 - internalize_from_stream 8-18
 - is_identical operation 8-17
 - streamable object
 - and inheritance 8-17
 - creating
 - StreamableFactory interface
 - create_uninitialized operation 8-19
 - creation key 8-17
 - StreamableFactory interface 8-19
 - StreamFactory interface 8-8, 8-12
 - create operation 8-12
 - StreamIO interface 8-4, 8-8, 8-16
 - read_operation 8-19
 - read_object operation 8-18, 8-19
 - read_t operation 8-16
 - write_operation 8-18, 8-30
 - write_object operation 8-18
 - write_operation 8-16
 - SubtransactionAwareResource interface 10-29
 - commit_subtransaction operation 10-30
 - commit_subtransaction operation 10-30
 - subtransactions 10-7, 10-12, 10-52, 10-54, 10-55, 10-59, 10-63
 - subtyping 55, 59
 - Summary of CORBA 2 Core Changes 15-217
 - supplier 4-2
 - SupplierAdmin interface 4-3, 4-16, 4-17
 - for_suppliers operation 4-16
 - obtain_pull_consumer operation 4-17
 - obtain_push_consumer operation 4-17
 - SupportAttributes 16-29
 - Symmetric key technology 15-38
 - synchronization of Time 14-18
 - synchronized data interface
 - see SD interface
 - System- and Application-Enforced Policies 15-35
 - system audit policies 15-23
- ## T
- TAG_ASSOCIATION_OPTIONS 15-193
 - Target 15-59
 - Target Side 15-176
 - target_requires field 15-190
 - target_supports field 15-189
 - TargetSecureInvocation 15-141
 - Technology Support for Delegation Options 15-30
 - Terminator interface
 - rollback operation 10-23
 - terminator object 10-38
 - Threats in a Distributed Object System 15-2
 - time 14-11
 - Time Interval Object (TIO) 14-10
 - Time Interval Objects (TIOs) 14-3
 - Time Service 15-233
 - Time Service interface 14-11
 - Time Service Requirements 14-1
 - Time Service requirements 14-1
 - time_set 14-16
 - time_to_interval 14-10
 - TimeBase 14-4, 14-5
 - Timer Event Handler 14-3, 14-15
 - Timer Event Service 14-3, 14-4, 14-13, 14-16, 14-22
 - TimeUnavailable 14-4, 14-8
 - traced delegation 15-29
 - Trader Attributes 16-21
 - Trader Policies 16-16
 - trading object service 16-2
 - transactions

- resource manager 10-64
 - transaction abort
 - see Resource interface
 - rollback operation 10-29
 - transaction context 10-18
 - management of 10-21
 - propagation of 10-21
 - transaction originator 10-13, 10-19, 10-22, 10-43
 - glossary definition 10-84
 - Transaction Service 15-232
 - transaction service
 - and concurrency control service 64
 - and orb interoperability 66
 - and persistent object service 65
 - application use of 10-31
 - transactional client 10-4, 10-34
 - glossary definition 10-84
 - transactional object 10-4
 - example 10-40
 - transactional server
 - defined 10-6
 - TransactionalLockSet interface 7-9
 - TransactionalLockSet interface operations 7-12
 - TransactionalObject interface 10-30
 - TransactionFactory interface 10-38
 - transactions
 - checked 10-32–10-34, 10-36
 - consistency property 10-53
 - consistency property, glossary definition 10-81
 - coordinator object 10-28, 10-29, 10-38, 10-39, 10-49, 10-56
 - distributed 10-36
 - durability 10-52
 - durability, glossary definition 10-82
 - flat 10-6, 10-7, 10-9, 10-36
 - flat, glossary definition 10-82
 - implicit propagation 10-37
 - interposition 10-45, 10-56, 10-59
 - interposition, glossary definition 10-82
 - isolation 10-7, 10-9, 10-13, 10-23
 - isolation, glossary definition 10-82
 - propagation 10-30–10-34, 10-38, 10-41, 10-55, 10-59, 10-62, 10-83
 - propagation to resource manager 10-74
 - recoverable object 10-5, 10-29
 - recoverable server 10-6, 10-35
 - recoverable server, glossary definition 10-83
 - recoverable server, example 10-39
 - resource manager 10-9, 10-74
 - terminator object 10-38
 - two-phase commit protocol 65, 10-12, 10-27, 10-45, 10-48, 10-53, 10-57, 10-64, 10-76, 10-79
 - two-phase commit, glossary definition 10-85
 - TraveralCriteria interface
 - next_n operation 9-44
 - traversal criteria
 - creating 6-41, 9-36
 - example of 9-37
 - Traversal interface
 - destroy operation 9-43
 - next_n operation 9-43
 - next_one operation 9-42
 - ScopedEdge structure 9-42
 - traversal object 9-35, 9-36
 - creating 9-41
 - TraversalCriteria interface 9-36, 9-43
 - destroy operation 9-44
 - next_one operation 9-43
 - visit_node operation 9-44
 - Weighted_Edge structure 9-43
 - TraversalFactory interface 9-41
 - create_traversal_on operation 9-42
 - Trusted Computing Base 15-53
 - Trustworthy System 15-245
 - Type checking information 17-22
 - Type Definitions 17-19
 - Type InaccuracyT 14-6
 - Type IntervalT 14-6
 - Type safety 17-7
 - Type TdfT 14-6
 - Type TimerEventT 14-15
 - Type TimeT 14-6
 - Type UtcT 14-6
 - TypedConsumerAdmin interface
 - obtain_typed_pull_supplier operation 4-26
 - obtain_typed_push_supplier operation 4-26
 - TypedProxyPullSupplier interface 4-28
 - TypedProxyPushConsumer interface 4-28
 - TypedPullSupplier interface 4-21
 - TypedPushConsumer interface 4-20
 - TypedSupplierAdmin interface 4-27
 - obtain_typed_pull_consumer operation 4-27
 - obtain_typed_push_consumer operation 4-27
- U**
- Unique entries (collections) 17-4
 - universal object identity 59
 - Universal Time Coordinated (UTC) 14-1
 - Universal Time Object (UTO) 14-8
 - Universal Time Objects (UTOs) 14-3
 - universal_time 14-4, 14-12
 - UniversalTime 14-3
 - Unmask Type Operation 16-66
 - unregister 14-17
 - Use of AccessPolicy and RequiredRights 15-134
 - Use of Interfaces for Access Control 15-111
 - Use of Interfaces for Delegation 15-113
 - Use of Privilege Attributes 15-133
 - Use of Rights and Rights Families 15-134
 - User sponsor 15-55
 - UserEnvironment interface
 - operations 5-37
 - Users' View of the Security Model 15-42
 - Using Interceptors 15-222
 - uto_from_utc 14-12
- V**
- Values for Standard Data Types 15-213
 - Vault 15-79, 15-156
 - View of the Security Model 15-42
- W**
- Withdraw Operation 16-41

Index

Withdraw Proxy Operation 16-58
Withdraw Using Constraint Operation 16-44

X

X/Open xlii
X/Open CLI standard 5-34
X/Open TX interface 10-70–10-72
X/Open XA interface 10-64

CORBAservices: Common Object Services Specification

TO: *CORBAservices* Readers
FROM: OMG Headquarters
RE: Update package for *CORBAservices*
DATE: July 30, 1997

In addition to the usual update pages, this update package contains the following new or changed information:

- Overview (chapter 1) - added Object Collections Service
Note: print complete chapter
- General Design Principles (chapter 2) - added Object Collections Service on page 2-12 and General Interoperability Requirements on page 2-13.
Note: print complete chapter
- Time Service (chapter 14) - replaced the type definition of type TimeT from “ulonglong” to “unsigned long long” (and associated text changes) and substituted the word “minutes” in place of “seconds” in the description of the type Tdft.
Note: print complete chapter
- Object Collection Specification (chapter 17) - new specification
Note: print complete chapter

Refer to the next page for complete update instructions.

Pages to remove from CORBA services (March 1997)	Pages to add from this update package (footer July 1997)
Title and copyright	Title and copyright
Table of Contents (footer reads March 1997)	Table of Contents (footer reads July 1997)
List of Figures (footer reads March 1997)	List of Figures (footer reads July 1997)
List of Tables (footer reads March 1997)	List of Tables (footer reads July 1997)
Preface (footer reads March 1997)	Preface (footer reads July 1997)
Chapter 1 - Overview (footer reads March 1997)	Chapter 1 - Overview (footer reads July 1997)
Chapter 2 - General Design Principles (footer reads March 1995)	Chapter 2 - General Design Principles (footer reads July 1997)
Chapter 14 - Time Service (footer reads November 1996)	Chapter 14 - Time Service (footer reads July 1997)
---	Chapter 17 - Object Collection Service (footer reads July 1997)
Index (footer reads March 1997)	Index (footer reads July 1997)