

EDS Business Objects Facility

This document is a response to the Object Management Group (OMG) Business Objects Domain Task Force (BODTF) Request for Proposals (RFP) number 1 (originally issued as Common Facilities RFP-4), Common Business Objects and Business Objects Facility

Submitted by Electronic Data Systems Corporation

January 17, 1997

OMG Document Number: bom/97-01-07

Copyright 1997 by Electronic Data Systems Corporation

All rights reserved.

The companies listed above hereby grant a royalty-free license to the Object Management Group, Inc. (OMG) for worldwide distribution of this document or any derivative works thereof, so long as the OMG reproduces the copyright notices and the below paragraphs on all distributed copies.

The material in this document is submitted to the OMG for evaluation. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, 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 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 information contained in this document is subject to change without notice.

This document contains information which is protected by copyright. All Rights Reserved. Except as otherwise provided herein, no part of this work 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 the permission of one of the copyright owners. All copies of this document must include the copyright and other information contained on this page.

The copyright owners grant member companies of the OMG permission to make a limited number of copies of this document (up to 50 copies) for their internal use as part of the OMG evaluation process.

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

Preface

Submitter

Electronic Data Systems Corporation (EDS) has prepared this submission in response to the Object Management Group, Common Facilities Request for Proposal Number 4 issued in January, 1996.

Scope of Submission

The Request for Proposals solicits submissions addressing either Common Business Objects or a Business Objects Facility or both. This submission addresses only the Business Objects Facility. As a practical matter, EDS believes that the development of viable common business objects depends on the availability of a stable Business Objects Facility so that the common business objects can, in fact, be marketed as components to be shared in a large market. Therefore, EDS has deferred any proposal for common business objects until after OMG has adopted a Business Object Facility specification.

Proof of Concept

EDS has developed many applications of object technology over the past ten years, including applications incorporating products that meet OMG CORBA specifications. EDS has also prototyped a Business Object Facility and is in the process of implementing a production Business Object Facility for use in large-scale, commercial applications. This production version will be available to support production applications by EDS later in 1997.

Submission Contact Point

Please address all inquiries regarding this submission to the following contact person:

Fred A. Cummins
Intelligent & Object Systems
Electronic Data Systems
5555 New King Street
Troy, Michigan 48098
email: cummins@ae.eds.com
Phone: (810) 696-2016
FAX: (810) 696-2325

Table of Contents

| | |
|---|------------|
| Preface..... | iii |
| Submitter..... | iii |
| Scope of Submission..... | iii |
| Proof of Concept..... | iii |
| Submission Contact Point..... | iii |
| Table of Contents..... | v |
| Introduction..... | xi |
| Part 1, EDS Approach..... | 1 |
| 1.1 Potential of Distributed Objects Technology..... | 1 |
| 1.1.1 Consistent..... | 1 |
| 1.1.2 Adaptable..... | 2 |
| 1.1.3 Integrated..... | 2 |
| 1.1.4 Accessible..... | 2 |
| 1.1.5 Component-Based..... | 2 |
| 1.1.6 Vendor Independent..... | 2 |
| 1.1.7 Reliable..... | 2 |
| 1.1.8 Responsive..... | 2 |
| 1.1.9 Secure..... | 3 |
| 1.1.10 Easy to Use..... | 3 |
| 1.1.11 Intelligent..... | 3 |
| 1.2 Challenges of Large-Scale, Distributed Objects Systems..... | 3 |
| 1.2.1 Technical Complexity..... | 3 |
| 1.2.2 Consistent Business Model..... | 3 |
| 1.2.3 Security..... | 4 |
| 1.2.4 Performance..... | 4 |
| 1.2.5 Legacy Migration..... | 4 |
| 1.2.6 Integration of Purchased Components..... | 4 |
| 1.3 A Strategic Architecture..... | 5 |
| 1.3.1 Architecture Overview..... | 5 |
| 1.3.1.1 Enterprise layer..... | 6 |
| 1.3.1.2 Application layer..... | 6 |
| 1.3.1.3 User interface layer..... | 6 |
| 1.3.1.4 Agents layer..... | 7 |
| 1.3.2 Separation of Layers..... | 7 |
| 1.3.2.1 Views and adaptors..... | 7 |
| 1.3.2.2 Change notification..... | 9 |
| 1.3.2.3 Commit validation..... | 9 |

| | |
|--|-----------|
| 1.3.3 Separation of Responsibility..... | 9 |
| 1.3.4 The Business Object Abstraction..... | 10 |
| 1.3.5 The Development Process..... | 11 |
| 1.4 The EDS Business Object Facility Implementation..... | 12 |
| 1.4.1 The Application Framework..... | 13 |
| 1.4.2 The Development tool..... | 13 |
| 1.4.3 The Workload Management Tool..... | 14 |
| 1.5 Summary..... | 14 |
| Part 2, Proposed Specification..... | 15 |
| 2.1 Introduction | 15 |
| 2.2 Business Object Facility Overview..... | 15 |
| 2.2.1 Business Solution Perspective..... | 16 |
| 2.2.2 System Architect's Perspective..... | 17 |
| 2.2.3 Alternative Configurations..... | 18 |
| 2.3 Business Object Facility General Objectives..... | 20 |
| 2.3.1 Business Object Interoperability..... | 21 |
| 2.3.1.1 Conceptual business object model..... | 21 |
| 2.3.1.2 Instance variable encapsulation..... | 21 |
| 2.3.1.3 Relationships..... | 22 |
| 2.3.1.4 Change notification..... | 22 |
| 2.3.1.5 Exceptions..... | 22 |
| 2.3.1.6 Transaction context..... | 22 |
| 2.3.2.7 Commit protocol..... | 23 |
| 2.3.1.8 Life cycle manager..... | 23 |
| 2.3.1.9 Unique external identifier..... | 23 |
| 2.3.2 Run-Time Sharing of Objects..... | 23 |
| 2.3.2.1 Consistent business information..... | 23 |
| 2.3.2.2 Collaborative computing..... | 24 |
| 2.3.2.3 Propagation of change..... | 24 |
| 2.3.2.4 Legacy integration..... | 24 |
| 2.3.3 Application Development Productivity..... | 24 |
| 2.3.3.1 Life Cycle..... | 24 |
| 2.3.3.2 Relationships..... | 24 |
| 2.3.3.3 Concurrency control..... | 24 |
| 2.3.3.4 Transactions..... | 25 |
| 2.3.3.5 Persistence..... | 25 |
| 2.3.3.6 Query..... | 25 |
| 2.3.3.7 Workload distribution..... | 25 |
| 2.3.4 Plug-and-Play Applications..... | 25 |
| 2.3.4.1 Adaptation..... | 26 |
| 2.3.4.2 Security..... | 26 |
| 2.3.4.3 Validate method..... | 27 |
| 2.3.4.4 Change notification..... | 27 |
| 2.3.4.5 Deployment of executables..... | 27 |
| 2.3.5 Heterogeneity..... | 28 |
| 2.3.5.1 Languages..... | 28 |
| 2.3.5.2 Paradigms..... | 28 |

| | |
|---|-----------|
| 2.3.5.3 Computer and communication networks..... | 28 |
| 2.3.5.4 User interfaces..... | 29 |
| 2.3.5.5 Desktop applications..... | 29 |
| 2.3.5.6 Development tools and techniques..... | 29 |
| 2.4 Business Object Interoperability Interfaces..... | 29 |
| 2.4.1 Base Business Object interface..... | 30 |
| 2.4.1.1 The identifier..... | 31 |
| 2.4.1.2 The type..... | 31 |
| 2.4.1.3 Bind..... | 31 |
| 2.4.2 Specialized Business Objects Interface..... | 31 |
| 2.4.2.1 Attributes..... | 31 |
| 2.4.2.2 Relationships..... | 32 |
| 2.4.2.3 Operations..... | 33 |
| 2.4.3 Adaptors..... | 34 |
| 2.4.4 Business Object Identifiers..... | 34 |
| 2.4.4.1 Persistent identifier..... | 34 |
| 2.4.4.2 External identifier..... | 35 |
| 2.4.5 Life Cycle Services Interface..... | 35 |
| 2.4.5.1 Remove..... | 36 |
| 2.4.5.2 Copy..... | 36 |
| 2.4.5.3 Move..... | 36 |
| 2.4.6 Change Notification Interfaces..... | 37 |
| 2.4.6.1 Supplier interface..... | 38 |
| 2.4.6.2 Consumer interface..... | 39 |
| 2.4.7 Transactional Resource Interface..... | 39 |
| 2.4.7.1 Validate method..... | 40 |
| 2.4.8 View Request Interface..... | 40 |
| 2.4.9 View Interface..... | 41 |
| 2.4.10 Workload Management Service Interfaces..... | 41 |
| 2.4 10.1 Workload manager..... | 41 |
| 2.4 10.2 Workload manager based finder..... | 43 |
| 2.4 10.3 Finder factory..... | 43 |
| 2.4 10.4 Workload logger..... | 43 |
| 2.4.11 Transaction Coordinator Interface..... | 44 |
| 2.4.12 Life Cycle Manager Interface..... | 44 |
| 2.4.13 Name Service Interface..... | 45 |
| 2.4.14 Query Evaluator Interface..... | 46 |
| 2.4.15 Exceptions..... | 46 |
| 2.4.16 Business Object State Data..... | 47 |
| 2.4.17 Business Objects Iterator Interface..... | 48 |
| 2.5 Business Object Persistence Management Interfaces..... | 48 |
| 2.5.1 Database Service Manager Interface..... | 49 |
| 2.5.2 Database Service Interface..... | 50 |
| 2.6 External Interfaces..... | 51 |
| 2.7 Extensions to IDL and DDL..... | 51 |
| 2.7.1 View Declaration..... | 51 |
| 2.7.2 Relationship Declaration..... | 51 |

| | |
|--|-----------|
| Part 3, Compliance with RFP Requirements..... | 53 |
| 3.1 Compatibility with Existing OMG Standards..... | 53 |
| 3.1.1 Object Request Broker..... | 53 |
| 3.1.2 Transaction Service..... | 53 |
| 3.1.3 Security..... | 53 |
| 3.1.4 Naming Service..... | 53 |
| 3.1.5 Relationship Service..... | 54 |
| 3.1.6 Collections..... | 54 |
| 3.1.7 Event Service..... | 54 |
| 3.1.8 Persistence Service..... | 55 |
| 3.1.9 Query Service..... | 55 |
| 3.1.10 Concurrency Service..... | 56 |
| 3.1.11 Life Cycle Service..... | 56 |
| 3.1.12 Exceptions..... | 56 |
| 3.2 Compliance with RFP General Requirements..... | 56 |
| 3.2.1 Interoperability..... | 56 |
| 3.2.2 Separation of Technology Issues..... | 56 |
| 3.2.3 Extensibility of Business Objects..... | 56 |
| 3.2.4 Reusability..... | 57 |
| 3.2.5 Scalability..... | 57 |
| 3.2.6 Ease of Development and Deployment..... | 57 |
| 3.2.7 Application Integration..... | 58 |
| 3.2.8 Security..... | 58 |
| 3.3 Compliance with RFP Optional Requirements..... | 58 |
| 3.3.1 How business objects implement the business model..... | 58 |
| 3.3.2 Legacy applications..... | 59 |
| 3.3.3 Flexibility and longevity..... | 59 |
| 3.3.4 Generality and desktop integration..... | 60 |
| 3.3.5 Proof of commonality..... | 60 |
| 3.3.6 Specification of business objects and metadata..... | 60 |
| 3.3.7 Multilingual use..... | 61 |
| 3.4 Compliance with RFP Technical Criteria..... | 61 |
| 3.4.1 Change and event notification..... | 61 |
| 3.4.2 Active views..... | 61 |
| 3.4.3 Transparent persistence..... | 61 |
| 3.4.4 Search mechanism..... | 61 |
| 3.4.5 Backout..... | 61 |
| 3.4.6 Concurrency and Serialization..... | 62 |
| 3.4.7 Nested transactions..... | 62 |
| 3.4.8 Referential integrity and garbage collection..... | 62 |
| 3.4.9 Encapsulated attributes and relationships..... | 62 |
| 3.4.10 Constraints, rules and policies..... | 62 |
| 3.4.11 Relationship management..... | 63 |
| 3.4.12 External name management..... | 63 |

| | |
|--|-----------|
| 3.4.13 Exception/fault resolution..... | 63 |
| 3.4.14 Configuration management..... | 63 |
| 3.4.14.1 Object upgrade installation..... | 63 |
| 3.4.14.2 Configuration of executables..... | 63 |
| 3.4.14.3 Adaptation of application components..... | 63 |
| 3.4.14.4 Local extensions to shared objects..... | 64 |
| 3.4.14.5 Implementation and location transparency..... | 64 |
| 3.4.14.6 Dynamic workload balancing..... | 64 |
| 3.4.15 Composite object bounds..... | 64 |
| 3.4.16 External resource representation..... | 64 |
| 3.4.17 User attributes and preferences..... | 64 |
| 3.4.18 Textual representation..... | 64 |
| 3.4.19 Executable object expressions..... | 65 |
| 3.4.20 Loose binding..... | 65 |
| 3.4.21 Instance specialization..... | 65 |
| 3.4.22 Reflection..... | 65 |
| 3.4.23 External interfaces..... | 65 |
| 3.5 Compliance with End User Requirements..... | 66 |
| 3.5.0 Completeness..... | 66 |
| 3.5.1 Primary requirements..... | 66 |
| 3.5.1.1 Interoperability..... | 66 |
| 3.5.1.2 Portability..... | 66 |
| 3.5.1.3 Substitutability..... | 67 |
| 3.5.1.4 Life cycle..... | 67 |
| 3.5.2 Administrative requirements..... | 67 |
| 3.5.2.1 Installation/de-installation..... | 67 |
| 3.5.2.2 Upgrade..... | 67 |
| 3.5.2.3 Performance management..... | 67 |
| 3.5.3 Additional requirements..... | 67 |
| 3.5.3.1 Testing and problem determination/resolution..... | 67 |
| 3.5.3.2 Version compatibility..... | 68 |
| 3.6 Conformance with Existing Industry Standards..... | 68 |
| Appendix A: Consolidated IDL Specifications..... | 71 |
| Common..... | 71 |
| Exceptions..... | 71 |
| Business Object..... | 72 |
| Life Cycle Manager..... | 73 |
| Change Notification..... | 75 |
| Naming Service..... | 76 |
| Transaction Service..... | 76 |
| Persistence Management..... | 77 |
| Query Service..... | 78 |
| Workload Management..... | 78 |
| Relationships..... | 81 |

| | |
|---|-----------|
| Appendix B: Example of Relationship IDL..... | 83 |
| Glossary | 87 |

Introduction

This document is a response to the Object Management Group (OMG), Business Objects Domain Task Force Request for Proposals (RFP) Number 1, (originally issued as Common Facilities RFP Number 4), issued in January, 1996. The RFP requested submissions on Common Business Objects and a Business Objects Facility. Submitters were asked to respond to either one or both of these subjects. Electronic Data Systems has elected to respond with a specification for the Business Objects Facility.

This document is presented in three parts. Part 1 describes the EDS perspective on the role of the Business Objects Facility and the approach EDS has taken to the implementation of such a facility. This information is provided as background for evaluation of the submission and is not expected to become a part of the OMG specification for a Business Objects Facility.

Part 2 contains the proposed Business Objects Facility specification. This part begins with an overview of the Business Object Facility to provide potential implementers with a full appreciation of anticipated functionality of a Business Objects Facility and its interfaces. This is followed by details of the proposed specification which includes a number of interfaces, extensions to several existing OMG interface specifications and extensions to the IDL syntax that provide more concise specifications for business objects implemented using a Business Objects Facility.

Part 3 addresses the compliance of this proposal with specific requirements set forth in the RFP. This includes compliance with the RFP's specific requirements, RFP's optional requirements, RFP's technical criteria, OMG's general technical requirements, and End Users' requirements, along with compatibility with existing OMG specifications, .

The appendix contains a complete listing of the IDL proposed by this document followed by a glossary of terms.

Part 1, EDS Approach

EDS is a large systems integrator and information services provider serving many, diverse corporations, institutions and governments, large and small, in many countries throughout the world. Our customers expect EDS to deliver solutions to their information systems needs making use of the most appropriate technology. EDS has identified the OMG Object Management Architecture and CORBA-based products as the foundation for a strategic architecture appropriate for many EDS customers.

While this technology offers great promise, there are few examples to prove its potential, and there is considerable cost and risk involved in transitioning an enterprise to such an architecture. The current effort by the OMG to define industry specifications for a Business Objects Facility is key to the adoption of this technology by many corporations, institutions and governments. It will provide industry direction and provide users of the technology the assurance that investment in such an architecture will have long-term benefits.

This part of the EDS submission will describe both the value of the Business Objects Facility to users of the technology and the approach EDS plans to take to apply the Business Objects Facility specifications proposed in Part 2 for the benefit of EDS customers. This part is organized into four major sections: (1) potential of distributed objects technology, (2) challenges of large-scale, distributed objects systems, (3) a strategic enterprise architecture and (4) EDS Business Object Facility implementation.

1.1 Potential of Distributed Objects Technology

EDS customers, end users of the technology, want information systems that support their business and, where possible, give them a competitive advantage. In today's world, remaining competitive generally means reducing costs, improving quality and becoming more responsive to customer demands and new business opportunities.

While advances in computing technology offer reduced cost and more sophisticated solutions, application of the technology often brings new risks and difficulties adapting and integrating the computing systems to meet business needs.

Most customers do not want to manage the technology. Customers need a computing and communications infrastructure that allows them to operate and evolve their enterprises with the support of information technology. They want information systems that offer the basic qualities described below.

1.1.1 Consistent

Information systems should provide consistent information and functionality and support consistent business processes throughout the enterprise. Inconsistencies are a major source of difficulties in improving business operations, making sound business decisions and responding to business opportunities.

1.1.2 Adaptable

Systems must be designed to adapt to changing business needs. Too often changes to information systems are the primary impediment to the ability of an enterprise to implement fundamental changes or respond to new opportunities.

1.1.3 Integrated

Users should have access to the information they need from anywhere in the enterprise without encountering barriers to communication or technical incompatibilities. They should be able to incorporate this information into their local applications on an ad hoc basis when necessary, based on an integrated model of the business. This integration of information systems will help people work together and use information more effectively.

1.1.4 Accessible

Systems must be accessible on an ad hoc basis. Internet technology is key to providing access anywhere, at anytime so users do not need to be explicitly connected to an application to use the information or obtain information. Accessibility must extend to business partners, customers and prospective customers. Users must be able to access information and system functionality from a variety of platforms with various modes of connection.

1.1.5 Component-Based

The architecture should be component-based so that moderately-sized components can be purchased or redeveloped without disrupting the rest of the business. Users should be able to purchase the latest solutions to business problems, gaining the benefits of the market place in cost and quality, and they should be able to easily integrate those solutions.

1.1.6 Vendor Independent

Compatible system components should be available from multiple vendors. Users have found it necessary to commit to the products of particular vendors in order to develop a consistent computing and communications infrastructure. Unfortunately, a consistent infrastructure may be out of date before it is fully implemented.

1.1.7 Reliable

Systems must operate so that information is accurate and up to date. Downtime must be minimal and failures that do occur should have minimal consequence to overall operation of the business.

1.1.8 Responsive

When users request information or enter commands, the systems must respond in a timely manner. Users should be able to easily perform ad hoc searches of enterprise information, display the information in various formats, and monitor the effects of business decisions. Individual productivity must be enhanced, not hindered, by system performance.

1.1.9 Secure

Information and system functionality must be adequately protected from improper disclosure or modification. Networking and desktop computing have increased security risks. The enterprise should be able to incorporate the internet, interoperate with business partners and interact directly with customers without compromising security.

1.1.10 Easy to Use

Systems must be easy to use to minimize training requirements, support individual productivity and facilitate the introduction of new system facilities or business processes.

1.1.11 Intelligent

Systems should do what they should reasonably be expected to do. Displays should include appropriate information for the tasks they support. Commands should provide intuitive defaults and raise questions when users take unusual or unforgiving actions. Systems should automate computations and processes that are accepted solutions. In some cases, intelligent components should provide the benefit of computerized expertise to the solution of difficult problems.

1.2 Challenges of Large-Scale, Distributed Objects Systems

Distributed objects technology has the potential to address the user needs described above, but current implementations have costs and risks that are major deterrents. The following paragraphs describe the major challenges.

1.2.1 Technical Complexity

While the currently defined OMG specifications provide basic building blocks for a distributed objects architecture, the integration of these components and the development of applications is still very complex. The development effort requires highly skilled people, the time and cost to create an appropriate architecture is a major investment and the risk that it will not meet expectations is high.

Too often the architecture development is driven by a single application with limited budget and challenging delivery dates. Compromises must be made in the design and implementation of the architecture that may later be major shortcomings as the scope of use of the architecture expands.

1.2.2 Consistent Business Model

Many years ago, industry leaders were in pursuit of corporate data models that would allow widespread sharing of information through integrated databases. These efforts met with only limited success because it became apparent that the modeling effort was very difficult and, once implemented, the database and applications using the model were difficult to change when new requirements emerged.

Although objects are more easily adapted, the development of an enterprise object model is just as difficult. Large-scale implementations of the technology must be designed to enable the enterprise model to evolve while continuing to provide reliable access to enterprise information.

1.2.3 Security

Security facilities have been late in coming to object technology. It is a major concern, particularly when applications execute on desktop computers or communicate over the internet.

Enterprise objects cannot be loaded into insecure environments. When enterprise objects are held in a secure environment, access to their data and functionality must be restricted so users can only access the subset for which they have authority.

Communications must also be protected. The internet offers major benefits in lower costs and flexibility but with security risks. Encryption must be integrated into the communication facilities.

1.2.4 Performance

Performance is a problem, to a great extent, because it is difficult to predict. A particular business process might be performed by a few objects on a single server, but there is a great possibility, particularly when systems scale up to the enterprise level, that processing will branch out to many servers in unpredictable ways as objects collaborate with other objects.

In addition to better design and simulation techniques, what is needed are tools and mechanisms for workload balancing so that the distribution of objects can be quickly adapted to the workload without disruption of service.

1.2.5 Legacy Migration

Most enterprises already have major investments in their current systems. Many of these systems were either developed as large monoliths, or have become tightly integrated over time. Most managers do not want to make the investment or endure the upheaval and risks of re-developing these major systems.

There must be ways, supported by the new infrastructure, to migrate the business systems a portion at a time. For the most part, this will be addressed by various forms of legacy system encapsulation. At the same time, ties to legacy systems must not condemn the new systems to ill-conceived business models.

1.2.6 Integration of Purchased Components

Part of the vision of object technology and distributed objects environments in particular, is that purchased components and applications can be plugged into the enterprise infrastructure. This would create a new market in which innovative solutions could be easily adopted with innovation and pricing driven by large market opportunities.

This vision will not be realized without the benefit of standards. The development of interchangeable components requires standards to assure compatibility of components and

architectures. Investments in the implementation of enterprise architectures are at risk without standards to assure compatibility.

While current OMG specifications provide for the interconnection of distributed objects architectural components, they do not address the need for consistent interfaces for business objects and applications comprised of business objects.

1.3 A Strategic Architecture

The challenges described above call for an integration of OMG facilities and services that (1) resolves the problems of managing distributed objects in a consistent and reliable way, (2) provides a consistent protocol for interoperation of business objects and (3) supports a consistent strategy for adaptation and integration of components, applications and legacy systems. These are the general objectives of the strategic architecture supported by the EDS Business Objects Facility. The following paragraphs describe the strategic architecture and key capabilities of the EDS Business Object Facility (BOF).

1.3.1 Architecture Overview

Traditionally, an enterprise architecture was viewed as a selection of hardware and software products which could be assembled in sufficiently diverse configurations to support the various information access, processing and communications needs of the enterprise. When the focus of computing was large computers in corporate data processing centers, this was, for the most part, achievable. However, with the advent of desktop computing and local servers along with rapid advances in the technology, consistency of hardware and software components and configurations is nearly impossible to achieve for large enterprises. Fortunately, with CORBA-based products and services it will be no longer necessary.

The enterprise architecture should comprehend a heterogeneous combination of languages, operating systems, computers, communications facilities and databases. The architecture should expand and evolve incrementally, not by massive upgrade or replacement. Changes should be driven by business benefits from improved cost, performance or functionality that can be implemented locally and integrated into the enterprise infrastructure, over time, with no disruption to the rest of the enterprise.

The key to this flexibility is to adopt standards that provide for the integration of alternative technological components, and to establish a standard, higher level abstraction that supports the implementation and interoperation of business objects and business solutions. Current CORBA specifications provide standards for the integration of technological components. The BOF specifications will provide the abstraction to support business objects and solutions.

Figure 1.1, The Enterprise Computing Model, depicts an Enterprise Computing Model based on the desired abstraction. The model does not show computers, databases or networks. Instead, the model depicts a layered architecture of objects that are implemented on a distributed computing environment.

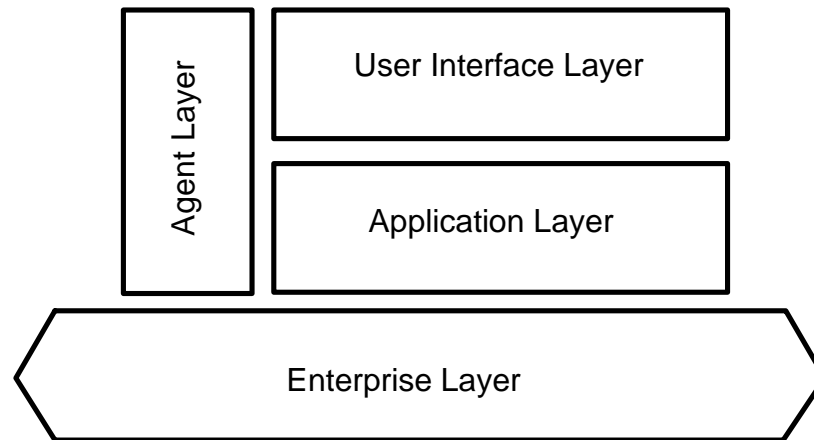


Figure 1.1, The Enterprise Computing Model

The boundaries between the layers will correspond to CORBA interfaces and will often correspond to physical separation of processing so that computers for the enterprise layer objects will typically be different from the computers for application layer objects, and so on.

1.3.1.1 Enterprise layer

The enterprise layer contains the business objects that serve interests across the enterprise. They are widely shared. The enterprise layer will likely be supported by a number of computers which may be geographically distributed. The important distinction of the enterprise layer is that its objects are accessible from anywhere in the enterprise, and possibly by authorized users outside the enterprise. But the objects are also highly secure, and accesses will be restricted to assure that requirements for protection of confidentiality and proprietary interests are met, and users can only perform operations on these objects for which they are authorized.

1.3.1.2 Application layer

The application layer contains business objects for solving particular business problems. To the extent the business concepts of these objects correspond to the business concepts of enterprise objects, they will be directly linked. Changes to objects in the application layer can be immediately reflected in the enterprise objects so that they can be seen throughout the enterprise. The objects in the application layer are essentially abstractions of the enterprise objects since they will only reflect a restricted view of the corresponding enterprise objects. At the same time, they may have additional data and functionality that is only of local interest.

There may be many applications interfaced to the enterprise layer. For the most part, applications do not communicate directly with each other, but interact through the shared objects in the enterprise layer. Applications can be attached and detached without disrupting other operations.

1.3.1.3 User interface layer

The user interface layer implements another level of abstraction where aspects of the business objects in the application layer are represented as graphical objects (or possibly other multi-media elements) to present information to the user and support interactions. The same mechanism can

be employed for interactions between user interface objects and application objects as are used for interactions between application objects and enterprise objects. In fact, when the functionality of an application reduces to viewing enterprise information, the user interface may be directly linked to the enterprise objects.

Separation of the user interface also allows different ad hoc configurations to be employed. In the work environment, the application and its user interface may run on a desktop computer, while the same user might access the application remotely over the internet, with a network-browser-based user interface. Similarly, customers and business partners may use different modes to access the applications they need.

1.3.1.4 Agents layer

The agents layer implements another dimension of business logic. These components monitor and sometimes direct the activity of objects in the other layers. For example, workflow management should be implemented in the agents layer. The workflow manager monitors activity in the enterprise objects, and possible application objects, to determine when information or a request for action should be routed to a person or organization. Other agents might provide interactive assistance to users in the form of tutorials or expert advisors. Agents are more likely than mainstream applications to be written in specialized languages and implement different abstractions of the business.

1.3.2 Separation of Layers

Key to this Enterprise Computing Model is a mechanism for separation of the layers. This mechanism must provide abstraction from the more general and comprehensive to more specific representations, it must protect the integrity of a layer being updated by the actions of another layer, and it must provide for the propagation of changes so that the effects will be reflected in related abstractions.

The EDS BOF and the proposed specification provide three principle components to implement this separation of layers: (1) views and adaptors, (2) change notification and (3) commit validation.

1.3.2.1 Views and adaptors

Users and applications should only see limited views of enterprise objects. These views should reflect their need for information and their authority to read or update it. The information they are not allowed to read or update must remain in a secure environment beyond their control.

At the same time, there is a need for flexibility between the representation of concepts at the enterprise level and the implementations of particular business solutions. Shared objects must be more stable and reliable than local applications; changes must be carefully designed, tested and coordinated.

Views and adaptors provide the controlled access and flexibility needed between the architectural layers. Figure 1.2, Use of Views and Adaptors, illustrates the use of views and adaptors. The example depicts an enterprise-level inventory object that is shared by two applications-- production updates and production planning. The enterprise layer is on the right and the application layer is on the left.

Access to the enterprise inventory object is restricted. The applications access the inventory object through different views. Each view provides an interface that is restricted to those operations that are authorized for that application (or user). The CORBA security service is used to determine and enforce the use of only authorized views.

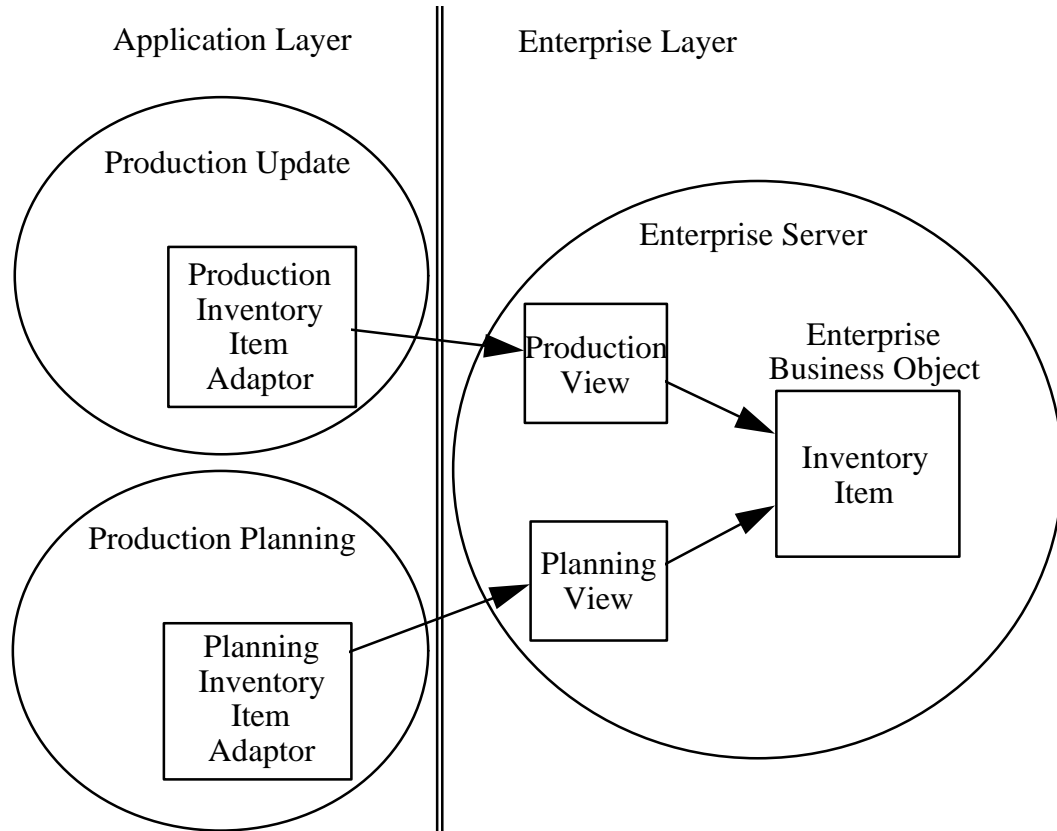


Figure 1.2, Use of Views and Adaptors

The applications on the left incorporate business objects that represent an inventory item from their individual perspectives. The inventory item is represented in the applications on the left by adaptor objects. An adaptor object is the same as other business objects but with a special capability.

The adaptor object is linked, directly or indirectly, to the object for which it provides a local abstraction. When the adaptor receives a message that is implemented by the enterprise object, the message will be forwarded to the associated view object in the enterprise layer. If the view interface does not include a specification for the message, then the message will be rejected. If the message is acceptable, the view will, in turn, forward it to the enterprise object.

The adaptor may translate the message if necessary to conform to the enterprise object interface. Translation may be necessary because the application was developed independently, or because the local perspective calls for different semantics from the enterprise-wide perspective.

The adaptor can also have local methods and instance variables that are not of interest at an enterprise level. This allows the application to be implemented without requiring changes to the enterprise objects.

The same concepts may be applied at other boundaries of the Enterprise Computing Model. Views will define restricted interfaces if the objects are shared, and adaptors will provide flexibility in the linkage between different perspectives.

1.3.2.2 Change notification

The propagation of effects from lower layers to upper layers is provided by another mechanism: change notification. In particular, change notification can be used to cause displays to be continuously updated, to detect conditions requiring special attention or to initiate action by a workflow management system.

All business objects are capable of providing ad hoc change notification. A change notification request to a business object specifies the aspects (attributes and relationships) of interest. Whenever one of the specified aspects changes, a notice will be sent to the designated consumer.

Change notification is specific to the object instance receiving the request and the designated consumer. A business object may provide different notices for a number of different consumers. At any point in time, most business objects will not be sending any notices since most of the time they are not subjects of interest.

1.3.2.3 Commit validation

The separation of responsibility is further supported by the integrity protection provided by **validate** methods on business objects. Applications incorporate enterprise objects through adaptors. Views limit access, but do not control the quality of changes made to the enterprise objects. The application developer may not be aware of all the policies and integrity rules applicable to the enterprise objects. Since the enterprise objects are widely shared and the information can be critical to the operation of the business, there is a need for additional assurance that updates are appropriate. This protection is provided by **validate** methods.

Each business object has a validate method. The method evaluates the current state of the business object, and possibly related business objects, to determine if it is acceptable. Acceptability may be in terms of business policies or in terms of business model integrity.

The validate method is always invoked before a changed object is committed. If the state is not valid, the associated transaction will be rolled back and the change will not be committed. The validate method may also be used by the application to evaluate changes and provide feedback to a user.

The validate method is a critical element in maintaining system integrity. It should be used at both the enterprise level and the application level to assure that results of actions by other levels and even within the same level are appropriate.

1.3.3 Separation of Responsibility

The separation of layers in the Enterprise Computing Model also corresponds to appropriate separation of responsibilities in the development of business systems. The enterprise layer should be developed through modeling the enterprise. It should be tightly controlled and the implementation must be highly secure.

The application layer is the responsibility of application developers whose attention is focused on solving particular business problems. They are able to model the solution in a manner and with semantics most suitable to the local business perspective.

The user interface, while usually associated with a specific application, requires a different skill set. In addition, ad hoc displays may be developed after the application is completed. In some cases, generalized user interfaces may be developed to provide standard graphics that are independent of particular applications. Adaptors allow these application-independent visualizations to be mapped to particular applications or enterprise objects.

Agents are yet another area of responsibility. While, in a sense, they are just additional applications, they may employ special tools or programming paradigms so that the developers require special skills.

Perhaps most importantly, the business object abstraction supports separation of the business system development responsibility from responsibility for the distribution of processing and the implementation of persistence (i.e., database interface and schema management).

1.3.4 The Business Object Abstraction

The Enterprise Computing Model is based on a business object abstraction that conceals most of the CORBA infrastructure mechanisms. This abstraction simplifies the development of business objects so that a developer can focus on solving the business problem instead of worrying about developing mechanisms to manage distributed processing. This abstraction is implemented by the EDS BOF.

For the most part, the developer works with objects as if they are in a single address space. Potential conflicts with other applications or processes are resolved automatically. Changes to business objects must always be performed in the context of a transaction. When a business process begins, it must start a transaction. When it completes, it must request the transaction to commit. If it encounters a problem and cannot proceed, it must request the transaction to roll back the changes. The identity of the transaction, the transaction context is carried implicitly with the actions of the business process so the developer is only concerned with beginning and terminating the transaction..

Transaction serialization and updates to persistent storage are handled implicitly. If a transaction encounters an object already in use, it will be suspended until the object is released. If a transaction is using an object, other transactions that attempt to use it will be suspended until it is completed. Persistent storage is only updated when the business process declares that the transaction should be committed.

The developer need not know where an object is located to interact with it. The object will be identified through an external identifier known to the user, or through its relationship to another business object. The Naming Service provides the translation of an external identifier to an object reference that can be used for message-sending.

When a new object is created, it is automatically assigned to an appropriate address space according to architectural specifications independent of the business processing. Objects may be moved for workload balancing, but these are not issues of concern to the business developer. They are the responsibility of persons who manage the infrastructure.

Storage and retrieval of persistent objects is handled automatically. When a message is sent to an object, it may be active or inactive. If it is inactive it will be automatically retrieved from the database. When the associated transaction commits, changed objects will be updated in the database. When there is no longer a need for the object to be in memory, then it is deactivated to be restored from persistent storage when it is needed again. Deactivation occurs automatically based on performance criteria.

The particular database used and the mapping of objects to the database storage structure, is encapsulated by the persistence management service. The database interface is defined by persons who specialize in this aspect of the system implementation, so the business developer is not distracted by these technical and architectural details. The same persistence management service can interface to different BOF's and a BOF may interface to multiple, heterogeneous databases, but this is not a concern of the business developer.

The business developer will only program business operations on the business objects. The methods for access to attributes and relationships are provided by the BOF. These attribute and relationship methods incorporate mechanisms that support the abstraction.

By concealing the distributed objects implementation details, the business object abstraction not only simplifies the solution of business problems, but it improves the flexibility of the infrastructure. Assumptions and decisions that affect the distribution of processing or the implementation of databases are not part of the business solution. The decisions are made and can be changed independent of the development of the business solution. Consequently, when advances in technology offer improvements in cost or performance, they can be implemented without changing the business systems. And when changes in workload require reconfiguration of the distributed processing network, these changes can be made without changes to the business systems and may often be accomplished without interrupting system operation.

In summary, the business object abstraction allows the business developers to concentrate on the business problems, while other specialists manage the infrastructure.

1.3.5 The Development Process

The development process for business objects is different from conventional object-oriented programming because the code for attributes, relationships and other mechanisms that the BOF encapsulates is either generated from business object specifications or provided by a BOF framework. In addition, business developers do not write code to interface with the database.

Figure 1.3, The Development Process, illustrates how business object components are developed. Analysis and design are fundamentally the same. Design will be affected by the particular language and programming paradigm supported by the BOF. However, the fundamental result of analysis and design will be interface definitions and class specifications. There are two categories of interface specifications: (1) business object interfaces expressed in IDL and (2) data specifications for the exchange of data with persistence management expressed in DDL (Data Definition Language). Both are defined by OMG. The class specifications define implementation inheritance.

The Programming Facility accepts these inputs along with the methods code which expresses the business operations logic. It produces expanded source code in the particular programming language. The code for business objects is then configured by the Configuration Facility into business object components, the executable modules which will determine the grouping of

business objects in the run-time address spaces. The business object component will include code to register it with the distributed computing environment as a resource or to link to another level (e.g., an application linking to enterprise objects).

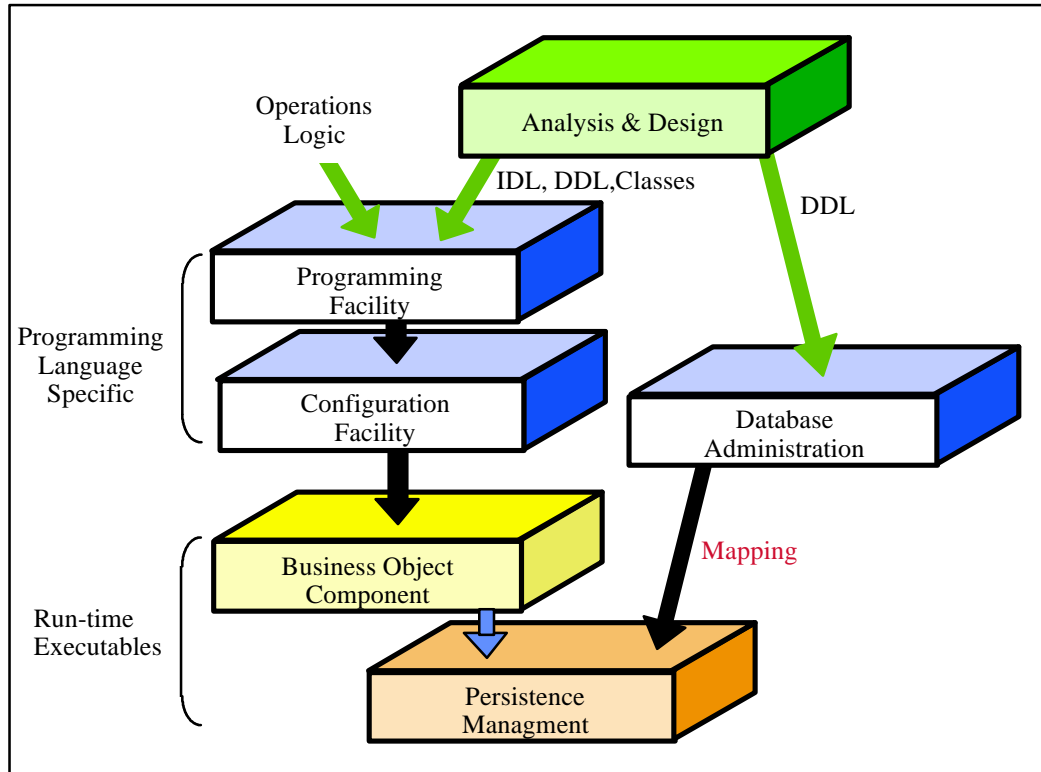


Figure 1.3, The Development Process

In parallel with this activity, the DDL is used to reconcile the business object data storage requirements with the database design. In the case of relational databases in particular, there will be a mapping of object attributes and relationships to relational table elements. The DDL defines the naming and format of data elements exchanged between the Business Object Component and Persistence Management at run time.

The specifications presented in this proposal define the interfaces to be supported in the run-time environment. The implementation of these run-time facilities, as a practical matter, requires complementary development tools to produce code that is reliable and consistent. Consequently, a Business Object Facility consists of the Programming, Configuration and Database Administration facilities along with the supporting logic which is incorporated in the Business Object Component and Persistence Management at run time. The diagram illustrates how the business development and the database development can be separated and, in fact, how Persistence Management can be defined as a separate architectural component from BOF support for business object implementations.

1.4 The EDS Business Object Facility Implementation

The proposed BOF specifications define interfaces for working with business objects supported by a Business Object Facility. There may be a variety of ways of implementing a BOF to comply

with these specifications. The implementation may depend upon the programming language, the operating environment, database interfaces, or other particular objectives sought by the BOF developer. The goal of the specification is to enable interoperability of business objects, not to constrain possible implementations.

Here we will briefly describe the EDS BOF implementation as an example approach which supports the goals and strategy described above. This BOF is implemented in C++ using Iona's Orbix object request broker and RogueWave's class libraries.

There are three fundamental components to the EDS BOF: an application framework, a development tool, and a workload management tool.

1.4.1 The Application Framework

The application framework provides BOF components that are incorporated into the production environment. These fall into two general categories: abstract classes from which business objects inherit functionality, and classes which implement supporting services, some of which are abstract classes and some of which are instantiated directly.

Business objects inherit functionality to support a number of BOF mechanisms. These include transaction management, concurrency management, change notification, and persistence. This inherited functionality is generic. Adaptors, which are specialized business objects, also inherit the same functionality plus adaptor functionality. Views inherit from an abstract view class.

Supporting services are implemented with other classes. Many of these are extensions of currently defined CORBA services such as the Naming Service, the Transaction Service and Life Cycle Services.

None of this functionality is application-specific.

1.4.2 The Development tool

The development tool generates application-specific code. This code incorporates BOF functionality in the following areas:

- Attribute accessor methods
- Relationship accessor methods
- Object instantiation and activation/deactivation
- Mapping of objects to relational tables
- View methods to provide for pass-through of authorized messages.
- Configuration of executable components to define domains and servers.

While business programmers could write this code and invoke inherited methods, it would be very tedious and have a high risk of coding errors. In addition, if the BOF functionality is

enhanced, the code can be regenerated from the object specifications without developer involvement.

The business developer writes the code for operation methods on the business objects. These methods must use the attribute and relationship accessor methods so that instance variable access is properly managed. Otherwise, the business process methods simply involve normal message sending.

The development tool may eventually be interfaced to an analysis tool or business model repository as the source of business object specifications. The tool will still be needed to generate the BOF code and create the executable configurations.

1.4.3 The Workload Management Tool

The workload management tool allows the system administrator to monitor system activity at run time and make adjustments to the allocation of work. The activity level of host computers, address spaces, life cycle managers (create, activate and deactivate activity by class) and even individual objects can be observed to identify bottlenecks. The system administrator can then move objects into different address spaces and control the allocation of objects to alleviate the bottlenecks. In addition, error conditions will be reported so that corrective action can be taken promptly.

1.5 Summary

The distributed objects architecture has the potential for substantially improving the consistency, integration and flexibility of enterprise systems. EDS has identified a strategic architecture to exploit this technology. However, there are number of challenges to be overcome.

A Business Object Facility, based on industry standards is a key enabler. This proposal by EDS addresses the challenges and reflects an EDS strategic architecture. The proposed specification reflects technical requirements based on a BOF implementation under development which will address the challenges and implement the strategic architecture.

Part 2, Proposed Specification

2.1 Introduction

This part of the EDS submission defines proposed Business Object Facility (BOF) specifications. A BOF facilitates development, supports interoperability and provides an environment for business objects. It is software that integrates other CORBA infrastructure components and services to provide a higher-level abstraction for the development of interoperable business objects and applications for a distributed, heterogeneous computing environment. These specifications define interfaces for BOFs that would be interoperable even though they might be implemented by different vendors, using different designs, languages, computing environments and databases.

A BOF supports *business objects*. A business object is an object which represents a concept inherent in a business enterprise as opposed to an object which represents a computational mechanism or elementary data value. A business enterprise, in this context, is broadly construed to include any corporation, institution, or government agency or any joint endeavor of these which might function as an enterprise. Business objects represent customers, employees, parts, organizations, invoices, accounts, specifications, machines, business procedures, locations, and many other business concepts. Some business concepts, and consequently some business objects, will only be relevant to a particular enterprise. Others will only be relevant to particular industries. Still others will occur in many different enterprises and represent concepts inherent in the conduct of any enterprise.

The general objective of this specification is to define the interfaces that will enable the integration and interoperation of business objects and applications developed by independent developers, running in different computing environments and using different languages and databases. These interfaces will enable object technology to scale up to enterprise-level integration of systems. It will also enable development of a marketplace in interoperable components that will reduce costs to users of the technology and result in components of better quality than those developed for individual applications.

The following section provides an overview description of a BOF and its role in support of distributed objects applications. This is followed by sections which detail the associated interface specifications as listed below.

- Business object interoperability interfaces,
- Interfaces to a supporting persistence management facility,
- External interfaces, and
- Extensions to IDL.

2.2 Business Object Facility Overview

Figure 2.1, Common Business Objects and the Business Objects Facility, illustrates the relationship of a BOF to business objects and the CORBA infrastructure. At the top of the diagram are specific applications. These will be built by incorporating and extending the

business objects depicted below them in the diagram. Some of these shared business objects will be industry-specific and some will be objects that implement concepts shared by many industries—the Common Business Objects.

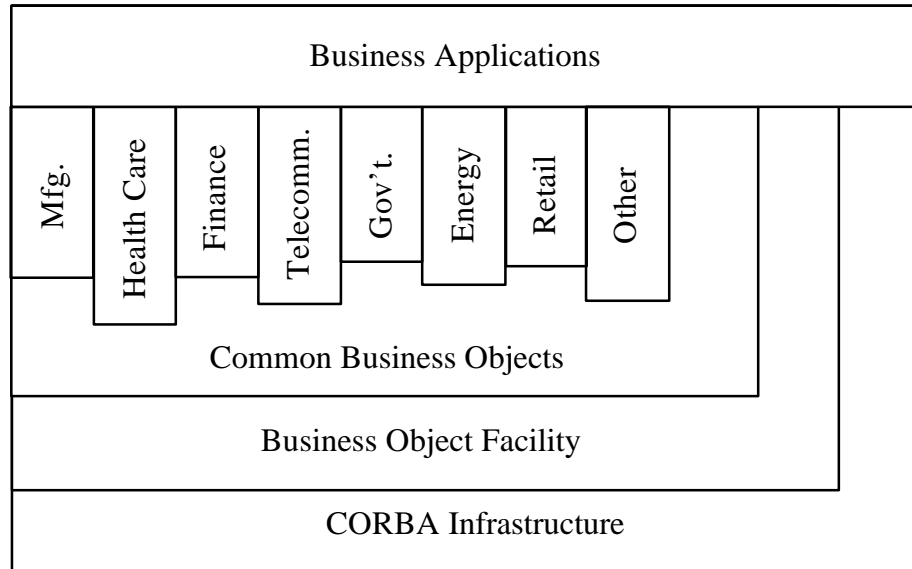


Figure 2.1, Common Business Objects and the Business Objects Facility

The BOF fills the gap between the business objects and the CORBA infrastructure at the bottom of the diagram. The CORBA infrastructure provides the general purpose mechanisms for interoperation of objects in a heterogeneous computing environment.

Without the BOF specifications, it is very difficult to develop business objects that can be used by multiple enterprises. First, there is a need for consistency in the interfaces to the business objects and their use of the CORBA infrastructure. This consistency requires generally accepted specifications. Second, there is a need for software that simplifies the development of business objects for a distributed objects environment and supports the consistent interoperability protocols. These specifications provide the basis for development of that software, the BOF, in a manner that will allow business objects and applications built on different BOF's to interoperate.

2.2.1 Business Solution Perspective

The abstraction provide by the BOF greatly simplifies the development of business solutions for a distributed objects environment. Developers define business objects to represent business concepts and then program methods to perform business logic on those objects. The methods are executed within the context of a transaction so that changes will be persistent or can be rolled back in the event of a unresolvable error.

The developer sees the objects as if they are in a single work space and for the exclusive use of the particular process being performed as depicted in Figure 2.2, Business Solution Perspective. Objects are located by asking the Naming Service to translate an external identifier to an internal object reference, the relationships of these objects will provide access to other objects.

The BOF will automatically retrieve objects from the database when they are needed. Business processes are initiated by messages from the user interface. When processing starts, a message to the transaction service starts a transaction. The information displayed in the user interface is obtained by sending messages to the business objects to access attributes and relationships. The change notification mechanism will also provide messages to update the user interface for attributes and relationships that change in objects of interest. When a process is complete, a message to the transaction service will commit the result; this will automatically update any affected databases.

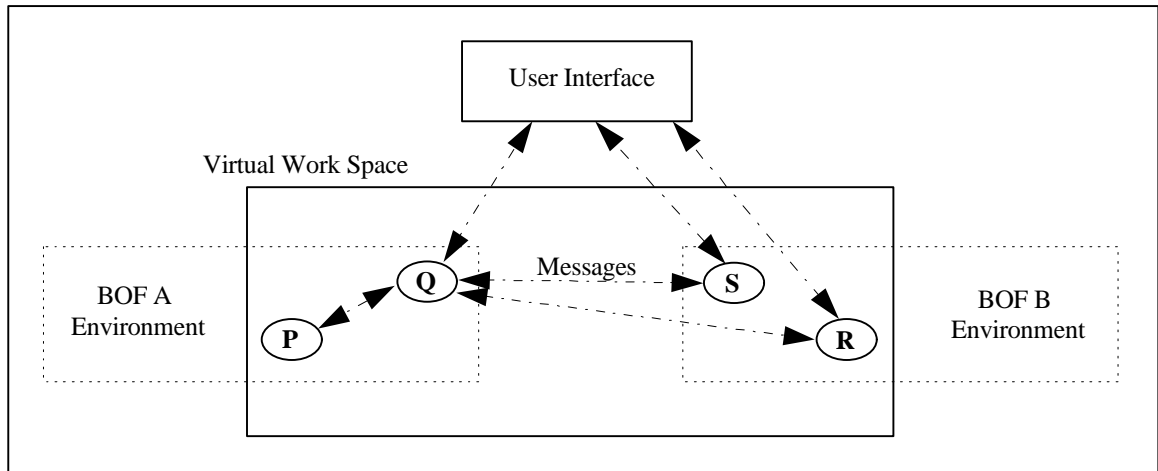


Figure 2.2, Business Solution Perspective

All these business operations are performed within the virtual work space defined by the scope of the active transition. There may be other business objects executing on the same computers in use by other business processes. As depicted in the diagram, the business objects may actually be implemented using two or more BOF's from different vendors, running on different platforms with objects stored in different types of databases. The business objects could be implemented in different languages--of course the business methods would then be written in those different languages. The user interface may be implemented in a specialized environment as well, such as a graphical programming environment or an internet browser.

2.2.2 System Architect's Perspective

The system architect sees the business system from a somewhat different perspective. The architect is interested in the distribution of work and data storage and its affect on system resource requirements and performance. Figure 2.3, The Architect's Perspective, depicts this perspective.

The architect will see the multiple user interfaces interacting with objects in multiple BOF address spaces and supported by multiple databases. The objects may send messages to each other, some of which will go between address spaces and over the network.

The architect will be involved in defining the configurations of address spaces, i.e., which business objects can run in which processes on which machines, as well as the specification of databases and where business objects will be stored.

The BOF and Persistence Management protocols will allow business objects running in different BOF's from the same or different vendors to be stored in a variety of databases. In fact, business objects can be dynamically moved from one BOF to another to balance the workload at run time. In the diagram, business object R has been moved from BOF A to BOF B. Its persistent state is still stored and updated in Database X. Thus the system architect can manage the distribution of processing and data storage both in configuring the system and through run-time workload balancing mechanisms.

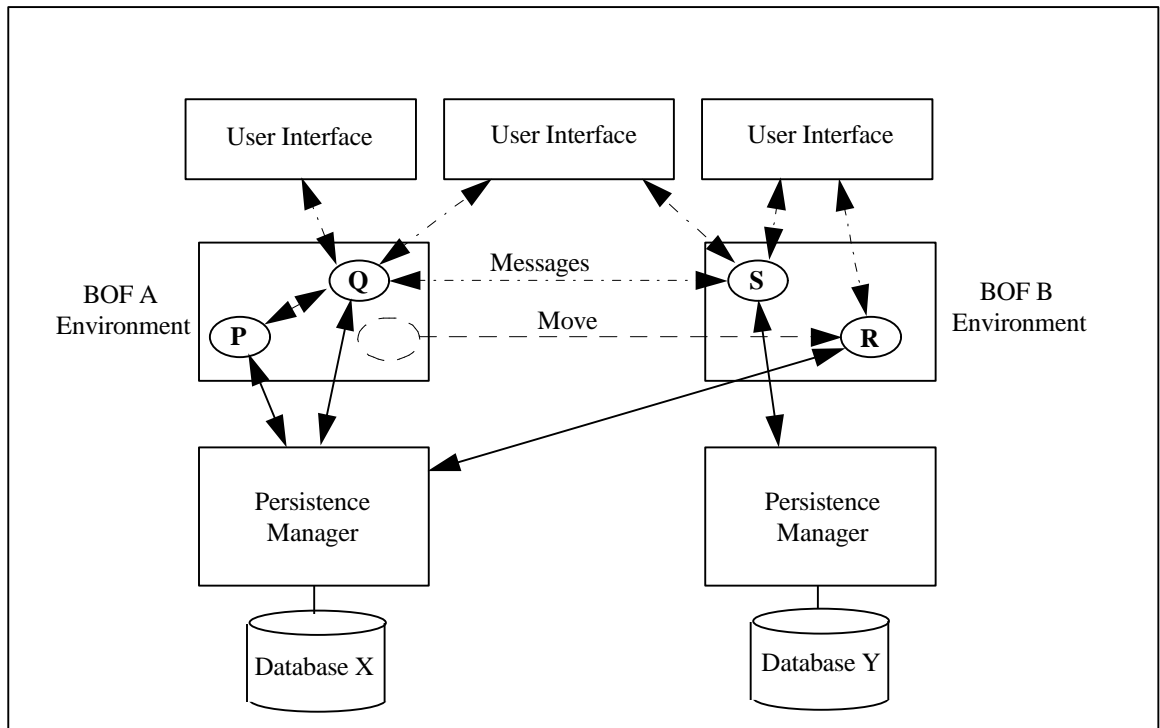


Figure 2.3, The Architect's Perspective

The databases may be different types of databases, relational or object-oriented, and they can store objects executed in different BOF's. While the business object interfaces are specified in IDL, the database storage requirement for business objects is expressed in DDL (Data Definition Language), a form similar to IDL, but limited to expression of attribute and relationship data storage requirements. The data storage requirements of a particular class of business object will be defined with the same DDL for all databases that store that class of business object. Note that not all attributes and relationships expressed in the IDL are necessarily stored in the business object. It is the responsibility of the Persistence Manager to perform the translation of the business object data to the specific database storage structures based on implementation-specific mapping specifications. This also allows mapping of object data to heterogeneous, legacy databases.

2.2.3 Alternative Configurations

The BOF allows configuring systems in a variety of ways within the same enterprise. Adaptors allow business objects to be shared at different levels. Several configurations are illustrated in Figure 2.5, Alternative Configurations Supported by BOF Specifications. This diagram shows a

3. Configuration 3 combines configurations 1 and 2 so that the user interface is separated from the application. This provides a “thin client” configuration where the user interface may be in a desktop application or an internet browser while the application runs on a more powerful server. This configuration also allows multiple users to have interfaces to a shared application for collaborative computing. The application business objects would be implemented with a BOF and may have local state and functionality along with local persistence.
4. Configuration 4 illustrates an intelligent agent component which may be implemented with business objects and a BOF or in a unique paradigm with a specialized BOF. The agent interfaces to shared objects using the adaptor protocol. Such a configuration might operate autonomously or have its own user interface. Workflow management, for example, should be implemented as intelligent agents, monitoring enterprise activity for certain conditions requiring action, and applying policies and work routings defined in shared objects to place items in appropriate work queues. Its operation could be displayed and controlled by viewing and acting on objects it shares at a global or application level. This is similar to configurations 2 and 3, except that the agent is not driven by a user through a user interface, it is driven by events that occur in the shared objects.
5. Configuration 5 illustrates encapsulation of a legacy system as a large-grained object. Here the large-grained object appears in the shared objects layer as an adaptor supported by a BOF. The adaptor encapsulates the legacy application state and functionality.
6. The last configuration illustrates a legacy interface through a shared database. Here the BOF shares the legacy database for persistent storage of shared objects which represent the same business concepts as those inherent in the legacy application. The database management system mediates between the BOF and the legacy system to resolve concurrency conflicts.

These do not represent all possible configurations, but they illustrate the potential diversity. The key is encapsulation of functionality to support distributed objects and the implementation of standard protocols that enable business objects to be interfaced and interoperate in a consistent fashion.

2.3 Business Object Facility General Objectives

This section will describe the following general objectives of a Business object facility:

- Business object interoperability
- Run-time sharing of objects
- Application development productivity
- Plug-and-play applications
- Heterogeneity

2.3.1 Business Object Interoperability

Business objects developed by different teams for different environments must be interoperable--they must be able to interact to perform business operations through message sending in a distributed environment. To achieve interoperability, business objects implemented with different BOF implementations must comply with certain consistent protocols.

2.3.1.1 Conceptual business object model

To begin with, they must be consistent with a general conceptual business object model. The business object model underlying the BOF specifications is very similar in concept to the basic OMG object model. The primary difference is in the integration of related services to provide a higher level abstraction for business systems development. Message-sending to invoke operations or access attributes remains the same. The model applies to fine-grained or large-grained implementations. In either case, a business object is a distinct entity in the computing environment which is accessed through sending messages using an object reference.

However, the business object abstraction does extend the generic OMG object interface in several ways: .

- Business object instance variables are fully encapsulated so that their values are only accessed through methods specifically for that purpose even by methods on the same object.
- Business objects have relationships, which are similar to attributes, but define references to other business objects.
- All business objects provide an ad hoc change notification service.
- Exceptions raised by methods on business objects have a specific structure and are returned in an unbounded sequence.
- Business objects operate in a transaction context, register as transaction resources, implement prepare, commit and rollback methods, and support transaction serialization.
- Business objects implement **validate** method used to validate the object's state prior to commit.
- A business object has a primary external identifier which is unique within a naming context.

The implications of these extensions are discussed in the subsections that follow.

2.3.1.2 Instance variable encapsulation

Encapsulation of instance variables is key to the integration of services and development of a higher level of abstraction for business objects. The access protocol for attributes and relationships hides their implementation so that attribute and relationship values may be stored or computed within the same interface protocol. The encapsulating code incorporates concurrency control and change notification mechanisms which are key to integration of shared, distributed objects. To assure consistent operation of these mechanisms, all accesses to instance variables

must go through the defined accessor methods, even accesses in methods written on the same object

2.3.1.3 Relationships

While attributes still have implicitly defined **get** and **set** methods, relationships must be distinguished from attributes. Relationships are references between business objects. They require additional methods. While these methods can be explicitly defined in IDL, this specification defines a relationship declaration as an IDL extension to provide for more concise expression of business object interfaces.

Relationships may be one-to-one or one-to-many. The BOF must automatically provide methods defined by a standard protocol for access to related objects and update of complementary relationships on related objects. Since a one-to-many relationship will contain a collection of related objects, a relationship method will return an iterator for access to the members of the relationship. This preserves the encapsulation of the relationship and provides the mechanism for query support as well.

2.3.1.4 Change notification

Business objects support ad hoc change notification. A business object can be requested to provide notification of a change in one or more of its aspects. An aspect is an object state value which may be either an attribute or a relationship. The request for notification will designate the aspect(s) of interest and the recipient of the notification along with other arguments. Whenever the specified aspect(s) change, the business object must send notices.

Three levels of service are provided. The first is not persistent and provides immediate notice regarding un-committed changes. The second is persistent, but will be discontinued if the consumer becomes unavailable. The third is persistent and if the consumer is unavailable, the notices will be held for later delivery. Both of the persistent levels of service deliver notices only when the changes are committed.

2.3.1.5 Exceptions

These specifications define a particular form for business object exceptions that will provide consistent information for processing of exceptions. In addition, exceptions are returned in an unbounded sequence so that multiple exceptions can be captured and returned. This allows applications to report multiple exceptions to allow an end user to take corrective action.

2.3.1.6 Transaction context

A business object operates as a resource in a transaction context. The context is passed implicitly with any message to a business object so that the BOF logic within the business object can identify the transaction and coordinate its activity. The first time a resource is locked for a transaction, it will register itself with the transaction coordinator.

The business object will automatically resolve concurrency control. If a second transaction attempts to operate on a business object when it is still in use by a first transaction, the business object will cause the second transaction to be terminated or suspended until the first transaction is completed. If the transaction is to be suspended, the business object will request that the

transaction coordinator determine suspending the transaction would cause a deadlock, and, if so, the second transaction will be rolled back.

2.3.2.7 Commit protocol

The Transaction Service and any BOF must support an extended two-phase commit protocol. There are two extensions.

First, a validate phase will precede the prepare phase. In the validate phase, the transaction coordinator sends a validate message to each business object to validate its state in order to insure integrity of the system on completion of the transaction.

Second, within the prepare and commit phases, priority will be given to business objects so that all business objects will receive the request before any database management resources. This allows business object processing to be completed before associated persistence processing proceeds. It is necessary for staging of database updates.

2.3.1.8 Life cycle manager

Within an address space, each class of object that can be instantiated remotely will have an instance of an associated life cycle manager in that address space. A life cycle manager is a specialization of a CORBA*factory* object. In addition to creating local instances of the associated business object class, if the business objects are persistent, the life cycle manager will manage activation and deactivation of those business objects in the address space.

2.3.1.9 Unique external identifier

Each business object must be given an external identifier that is unique within a specified naming context. This external identifier is used as the persistent identity of the object and must be registered with the Naming Service. The Naming Service will always have the last known object reference for the object.

2.3.2 Run-Time Sharing of Objects

Business objects, as noted above, should be developed for use by multiple enterprises. However, the sharing of business objects should not only be at the level of shared software. Business objects should be shared at run time so that everyone in the enterprise who examines information on a particular part or invoice or work order will see the same information--the same business object. This is important for several reasons.

2.3.2.1 Consistent business information

First of all, the run-time sharing of business objects means that information used by all participating applications is consistent. There are no delays in transfer of updates to different locations. There are no transformations to introduce inconsistencies. Changes once made are immediately available everywhere. This will result in better-informed and consistent business decisions and control of business operations.

2.3.2.2 Collaborative computing

These shared business objects can be made available to multiple users simultaneously to support collaborative computing. Different users may use different applications to examine the information from different perspectives. When changes are applied, all users will see the effects and be able to immediately provide feedback or contribute to further development of a solution.

2.3.2.3 Propagation of change

The changes applied to shared business objects can also cause other dependent actions to be initiated. A user may need to know when a price changes or a task is completed. Shared business objects can be asked to communicate events of interest for a variety of purposes either for specific situations or general cases. Workflow management can be driven by events from the relevant business objects so that users can immediately be made aware of the need for action based on the new state of an object.

2.3.2.4 Legacy integration

A BOF may provide for integration of legacy applications as run-time shared objects. There are two fundamental configurations which were described earlier. First adaptor business objects may be used to encapsulate the legacy application as one or more objects with appropriate interfaces. Second, business objects may share a database with the legacy system so that the legacy database becomes the persistent object store for business objects.

2.3.3 Application Development Productivity

A BOF should significantly improve the productivity of business system developers by providing a simplified abstraction of the distributed objects infrastructure. The following paragraphs describe aspects of this abstraction.

2.3.3.1 Life Cycle

A BOF provides implementations of life cycle services--create, remove, copy and move. The implementation of these services assures that objects are instantiated in appropriate address spaces and the actions are properly coordinated with the database if the objects are persistent.

2.3.3.2 Relationships

A relationship, discussed further below, is a reference in one business object to one or more other business objects. Relationships between business objects may span address spaces. In addition, BOF functionality must maintain bi-directional references between the participants to assure the integrity of relationships and minimize the effort required of the application developer.

2.3.3.3 Concurrency control

Accesses to shared objects must be controlled to prevent concurrency conflicts. The BOF implements concurrency control mechanisms so that application developers send messages to

objects just as they would in a single process environment. The concurrency control is implicit, based on the transaction context.

2.3.3.4 Transactions

A transaction is defined as a group of operations applied to a system to take it from one consistent state to another. To insure the integrity of operations performed by different users or business processes, transactions must be executed in a manner so that the result is the same as if they were executed serially, one after another. This serialization of transactions is accomplished by the Transaction Service in conjunction with concurrency control provided by the BOF.

All business object operations are performed in the context of a transaction. The transaction context is passed implicitly. The application developer is only required to declare the beginning of a transaction and the end of the transaction, either commit or rollback. A BOF implements the mechanisms for coordination with concurrency control and to perform a commit or rollback. The BOF will automatically perform database updates when a transaction is committed if persistent objects were changed. A two-phase commit is used so that transactions may span multiple databases.

2.3.3.5 Persistence

The BOF makes persistence transparent to the application programmer so that persistent objects are activated when they are referenced and are deactivated when they are no longer in use. The BOF uses an encapsulated persistence management facility so that the storage and retrieval of persistent objects requires little or no programming and specifications provide for transparently interfacing diverse databases.

2.3.3.6 Query

Queries to retrieve business objects based on arbitrary selection criteria are independent of the use of database storage. Queries can include both persistent and non-persistent objects, and the scope of a query is logically bounded by membership in a collection or relationship rather than being determined by the bounds of a particular database.

2.3.3.7 Workload distribution

The BOF provides for assignment of objects to address spaces when they are created or moved to control workload balancing transparent to applications. Address space configuration specifications will determine which address spaces can accept an object of a particular class. Allocation logic and system management controls can be used to control where objects are instantiated or moved at run time.

2.3.4 Plug-and-Play Applications

A distributed objects architecture using BOF's will enable the development of systems that share objects on an enterprise scale. These objects will be the primary source of information about the business and will implement many of the enterprise processes. They will necessarily be carefully controlled and protected.

At the same time, there will be a need to integrate new and ad hoc solutions to business problems. These may range from specialized interfaces for data imaging to complex simulations that support management decisions. These solutions should be driven by the shared objects, but should not require modification of the shared objects except to the extent that new information is to be made available for sharing.

The BOF provides the ability to adapt new applications to the shared model without modification to the shared model, while preserving integrity protections of the shared model. Some of the possible configurations have been described above. In general, plug-and-play applications are expected to be executable components which become an address space (or possibly multiple, interoperating address spaces) at run time.

Shared objects might also be marketed in executable components addressing particular aspects of the business. However, since the enterprise objects will likely evolve over time and will incorporate unique aspects of the business, it is more likely that they will be marketed as abstract classes that can be specialized for the particular enterprise.

The following paragraphs describe the key BOF facilities supporting plug-and-play applications.

2.3.4.1 Adaptation

The BOF provides an adaptor facility to link applications to shared objects. Each application object that represents a concept in the shared model is implemented in the application as an adaptor. Essentially, the adaptor forwards messages to the corresponding shared object. The adaptor provides two basic capabilities: (1) it allows the signatures and return values of messages to be translated to comply with the interface of the shared object, and (2) it allows additional instance variables and methods of only local application interest to be attached to the adaptor as local extensions.

Essentially an adaptor functions within its application like any other business object. The difference is that some or all of its attributes, relationships and operations are incorporated from the remote object that it adapts. The application containing the adapter is not limited to the attributes, relationships and operations of the remote object, but it can add its own to the adaptor.

For example, project planning might be a local application. It would keep track of activities and tasks, and it would track assignments of employees to those tasks. The local employee objects might be adaptors for enterprise employee objects. The names and job classifications of the employees might be drawn from the enterprise objects, while other project planning information and operations on the employee adaptors would be local to the adaptors. The project planning application would work with the adaptors as if they were self-contained employee objects.

This allows applications to be developed which can be mapped to different business object models, or, conversely, it allows users to purchase independently developed applications and adapt them to interface to their own business model.

2.3.4.2 Security

The shared objects will contain proprietary information as well as information critical to the operation of the enterprise. Access to this information must be restricted so that end users can only access authorized information regardless of the application they may implement on their local machine.

Views provide this control. Each business object can have a number of views authorized for different users. The view defines a particular subset of messages that will be acceptable from a user. Security of access to the view is managed by the OMG Security Service. The BOF provides supporting mechanisms for providing the appropriate view in response to a user request.

2.3.4.3 Validate method

While views and the security facility will restrict access of a particular user, they will not protect the shared objects from application errors. Widespread sharing of objects can greatly increase both the risk and consequential impact of erroneous updates. A means of protection from such erroneous updates is provided by the commit validation protocol.

The commit validation protocol causes a validate method to be executed on each changed business object before it is committed at the end of a transaction. The validate method is expected to verify that the new state of the object meets integrity criteria. If the new state is not acceptable, the transaction will not be allowed to commit.

This provides important protection against application errors.

The BOF specification does not restrict the form of a validate method. It may be written in an implementation-specific rule syntax, or simply implemented as conditional logic in the BOF programming language. The only requirement is that a collection of exceptions is returned if the state is not acceptable. This collection allows the application to report all detected exceptions to the user for possible corrective action.

2.3.4.4 Change notification

Change notification provides a mechanism by which plug-in applications can become active participants in enterprise activity. Change notification allows any application to monitor authorized aspects of shared objects so that application processing can be triggered by actions occurring elsewhere in the enterprise.

A user might plug in a data visualization application to monitor activity in some area of the business. As information changes the display will be updated. A user might employ another plug-in application to monitor certain shared objects for conditions of particular interest and even initiate corrective action. Another user might plug in an application to view and update certain objects shared with another user so that they can collaborate on a solution.

2.3.4.5 Deployment of executables

These plug-in applications need not be distributed as source programs, but can be executable components. The CORBA infrastructure supports the activation of a process on an already active distributed objects network. The application can be designed to accept adaptor mappings from a file or user input to specify the linkage of the application to the shared objects.

Thus independently developed applications can be deployed and integrated into various environments without compromising the proprietary interests of the developer. Since the capabilities, implementations, programming languages and, possibly, programming paradigms of BOF's will vary, the plug-and-play components would generally incorporate a particular BOF.

The portability of BOF's to different ORBs will be restricted if the BOF uses ORB features that are not available on other ORBs.

2.3.5 Heterogeneity

The BOF enables the integration of a diversity of technology and solutions. The key to this integration is the CORBA infrastructure and the implementation of standard protocols for business object interoperation. Examples of the dimensions of diversity are described briefly below.

2.3.5.1 Languages

OMG has defined CORBA language mappings for a number of languages. These and other languages may be used with a BOF in one of four modes:

1. A BOF may be implemented in the particular language so that full function business objects may be implemented using the BOF.
2. Applications or components can be implemented to interface with the BOF protocol without implementing business objects. Protocol implementation might be limited if the component does not require all of the capabilities of business objects (e.g., not multi-threaded, no change notification). Large-grained objects encapsulating legacy applications might be implemented in this mode.
3. The application or component might be implemented as a client (sending but not receiving messages for business objects) using one or more adaptors to interface to the BOF protocol. This could be done with a minimal amount of functionality particularly if the application objects are not persistent and don't allow concurrent processing.
4. A simple, non-persistent, non-shared client application might be implemented using only message-sending to shared objects to obtain information and display it.

2.3.5.2 Paradigms

Different solution paradigms may be integrated into the distributed environment. For example, solutions might be implemented with rules, neural networks, logic programming, genetic algorithms, and so on. These may be implemented with specialized languages. Since integration requires an interface to CORBA facilities, the solution would require at least an interface using a language that has a CORBA mapping, most likely C++. However, an application environment might otherwise be in an entirely unique language designed to support the paradigm.

Integration might be through any one of the modes described under languages, above. Where there is sufficient demand, it would be appropriate for BOF's to be developed to support particular, specialized paradigms.

2.3.5.3 Computer and communication networks

CORBA products enable the interoperability of objects on a variety of computers and operating systems using diverse communication facilities. Differences between operating systems and object request brokers (ORBs) may prevent the same BOF from running in diverse computing

environments. In some cases, BOF implementations may be designed to run on specialized computers or operating systems, such as a BOF to support an engineering analysis application running on a parallel processing super computer. Interoperability must also be achieved over a variety of communications facilities including LAN's, the internet and dial-up connections. The implementation of different BOF's compliant with the standard protocol and supported by a CORBA-compliant infrastructure will enable the interoperation of business objects running in a diversity of environments.

2.3.5.4 User interfaces

A variety of user interface implementations may be employed. The user interface may be integrated in an application environment with business objects or it may interoperate as an independent component interfacing through one of the four modes described above, most likely mode 3 or 4. Such components may be large and complex such as multimedia or virtual reality interfaces. The BOF does not limit the types or technology except that it must be able to function as a client to business objects using the BOF protocols.

2.3.5.5 Desktop applications

There is considerable demand for interfacing to desktop applications, such as spreadsheets and business graphics, to shared information. Such applications may support interfaces such as Microsoft's COM. The desktop application protocols could be implemented on business objects so that these applications could interoperate directly with the business objects over a COM/CORBA bridge, or an intermediate application could be developed to provide the COM interface to the desktop and use a BOF adaptor interface to shared objects.

2.3.5.6 Development tools and techniques

The BOF specification does not define development tools or techniques. The BOF should minimize the amount of effort required to implement a business object model and program its methods. A BOF may provide an interface to accept input from a modeling tool or repository to facilitate the development process. In some cases, a BOF may be more effective if it provides its own development tool designed to support a particular language, paradigm or interfacing capability.

A BOF might accept IDL for interface specifications and an object model for class specifications. The IDL and object model could be developed with an automated analysis and design tool. However, the BOF would still require specifications for design options, the database interface and address space configurations along with business logic for the business object methods.

2.4 Business Object Interoperability Interfaces

This section describes in detail the interfaces and related structures that must be supported for business object interoperability. These interfaces are the basis for interactions between business objects implemented in the same or different BOF's as well as interactions of other business system components that are to interoperate with business objects. This includes user interface facilities, workflow management, intelligent agents and analytical tools implemented in specialized languages.

The business object interoperability interfaces and related structures will be described under the topics listed below.

- Base business object interface
- Specialized business objects interfaces
- Adaptors
- Business object Identification
- Life cycle services interface
- Change notification interfaces
- Transactional resource interface
- View request interface
- View interface
- Workload management service interfaces
- Transaction coordinator interface
- Life cycle manager interface
- Name service interface
- Query evaluator interface
- Exceptions
- Business object state data
- Business objects iterator interface

A number of these interfaces occur together on business objects but are defined independently since they may be used in other contexts or for partial implementations of BOF facilities. A normal, shared business object will include the base business object, life cycle services, change notification, transactional business object, and view request interface. Views and adaptors are specializations of the base business object. There are a number of interfaces for supporting services, many of which are specializations of CORBA services. The BOF also observes consistent protocols for exceptions, passing of object state data, and iterators. These are discussed at the end of this section.

2.4.1 Base Business Object interface

The base business object interface reflects the basic business object model. The base business object has methods to support BOF operations and it inherits several other interfaces which also include BOF functionality. The base business object interface specification is shown below.

```
interface BofBusinessObject : BofLifeCycle::BofLifeCycleObject ,
    BofTransactions::BofTransactionalResource,
    BofNotification::BofNotificationSupplier,
    BofNotification::BofNotificationConsumer{
    /*
    ** Business object attributes.
    */

    readonly attribute BofObject Identifier the_identifier;
```

```
readonly attribute string                               the_type;

/*
** Interface to bind and unbind the object to naming service
** The exceptions raised will most likely come from the
** naming service.
*/

void bind(in BofName qualified_name)
        raises(BofException);
};
```

The above interface is inherited by specific business objects where attributes, relationships and operations specific to the business object are added. Components of the above interface are discussed below.

2.4.1.1 The identifier

The *identifier* of a business object provides a unique identity within an appropriate context based on its primary external identifier. This identity can be used to locate the object in its database and is always bound to its current object reference with the Naming Service.

The identifier is a sequence of name-value pairs. This form allows for identifiers that are comprised of several concatenated keys. The name-value pairs are the attribute/key name and value of each key component.

2.4.1.2 The type

The type is the implementation type of the business object expressed as a string. The type must be provided when the object is created. It determines the life cycle manager used to create, activate and deactivate the object.

2.4.1.3 Bind

The **bind** operation is provided as the business developer interface. It provides additional instance information for submission of the bind request to the Naming Service. The complementary unbind operation is accessed on the Naming Service directly since the instance may not be available when an unbind is desired, and there is no additional information needed.

2.4.2 Specialized Business Objects Interface

The base business object interface is inherited by all business objects. Specialized business objects then add attributes, relationships and operations to define the specific capabilities of objects that represent business concepts. This section defines the generic protocols for attributes, relationships and operations.

2.4.2.1 Attributes

Attribute methods, **get** and **set**, are of the same form as for other CORBA objects. These methods incorporate additional functionality to support BOF facilities.

2.4.2.2 Relationships

Relationships require a set of methods for each relationship. The IDL extension for declaration of relationships specified later in this document will implicitly define method signatures to operate on the relationships.

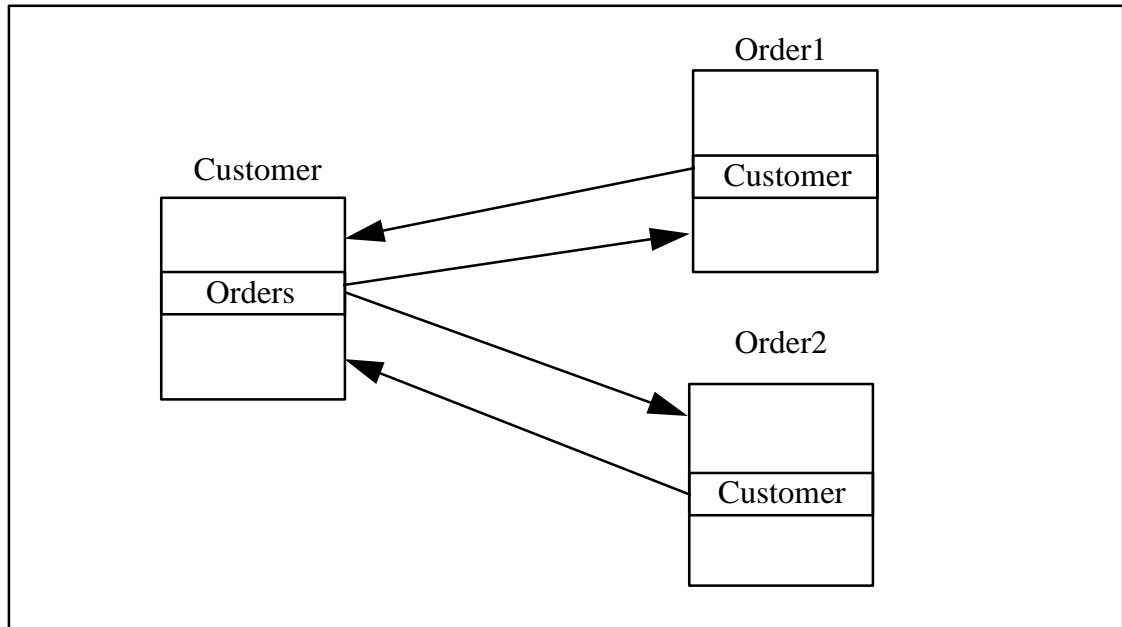


Figure 2.5, Complementary Relationships

All relationships are implemented as binary, one-to-one or one-to-many relationships. Many-to-many relationships are not allowed and can easily be represented as two one-to-many relationships. N-ary relationships can always be implemented as binary relationships. In most cases, binary, one-to-many implementations of relationships will provide a more adaptable generalization of the domain information.

All relationships are complementary. As illustrated in Figure 2.5, Complementary Relationships, the **Customer** and **Order1** objects are related as are the **Customer** and **Order2** objects. Each object has a relationship for any object related to it **Orders** on the **Customer** object and **Customer** on the **Order** objects. If the relationship changes from one object's perspective, it must also be updated from the other object's perspective.

The methods implied for a relationship will depend upon whether it is declared as referencing one or many related objects. The following methods are implied for a one-to-many relationship where <name> designates the name of the relationship. See sample expansions in Appendix B.

```

void <name> (in unsigned long batchSize,
            out sequence members,
            out BusinessObjectsIterator iterator)
    raises (BofException);
void add_<name> (in BusinessObject member)
    raises (BofException);
void delete_<name> (in BusinessObject member)
  
```

```

    raises (BofException);
boolean is_member_<name> (in BusinessObject member)
    raises (BofException);
void private_add_<name> (in BusinessObject member)
    raises (BofException);
void private_delete_<name> (in BusinessObject member)
    raises (BofException);

```

The **<name>** method returns a sequence of members of the relationship up to a maximum number defined by **batchSize**. If there are additional members, then **iterator** is returned, otherwise **iterator** is nil. The **BusinessObjectsIterator** protocol is defined later.

The **add_<name>** and **delete_<name>** methods add or delete from the relationship, respectively, the specified member business object.

The **is_member_<name>** method is a predicate to determine if the specified BusinessObject is a member of the relationship.

Relationships between business objects are complementary. The name of a complementary relationship is specified in the IDL relationship declaration. For each relationship, the related object must support a complementary relationship which will be updated to correspond to any updates to the relationship on the target object. The **private_add_<name>** and **private_delete_<name>** methods support these complementary updates for relationships to many objects, and **private_get_<name>** and **private_set_<name>** support complementary updates for relationships with a single reference. These methods are only to be performed by the BOF relationship management logic. These methods are necessary to prevent updates to complementary relationships from entering a loop. They are not for use by the business programmer and should never be visible through views.

Similarly, the methods to access a relationship specified as having only one related object (one-to-one or many-to-one) will have the following form:

```

void <name> (out BusinessObject member);
    raises (BofException);
void set_<name> (in BusinessObject member)
    raises (BofException);
void delete_<name> (in BusinessObject member)
    raises (BofException);
boolean is_member_<name> (in BusinessObject member)
    raises (BofException);
void private_set_<name> (in BusinessObject member)
    raises (BofException);
void private_delete_<name> (in BusinessObject member)
    raises (BofException);

```

2.4.2.3 Operations

Operations are declared in the same form as other CORBA object operations except that exceptions should conform to the BOF exception specifications.

2.4.3 Adaptors

Adaptors are specialized business objects that delegate messages to a primary object. The purpose of an adaptor is to provide the interface to another level of abstraction and to support interface translation between an application and a shared object.

```
interface BofAdaptor: BofBusinessObject {
    readonly attribute BofBusinessObject primary_object;
};
```

The logic of adaptor methods is to forward messages to the primary business object, with signature translation if necessary. It is expected that the BOF implementation will facilitate programming of this forwarding, but the approach will be implementation specific.

The primary object is identified by the **primary_object** attribute. This is not implemented as a relationship since it is inappropriate for such dependencies to exist across levels of abstraction. Adaptors are abstractions of the model they represent, but are not part of the model.

The abstract adaptor class should implement the copy and move methods. Copy should propagate to the primary object. The move method should move the adaptor, but should not propagate to the primary object.

2.4.4 Business Object Identifiers

Business objects can be identified in three different ways: (1) with an inter-operable object reference, (2) with a persistent identifier and (3) with an external identifiers. The first of these is the identifier used for directing requests to objects in a CORBA environment and is defined by the associated ORB. The other two are defined by this specification.

2.4.4.1 Persistent identifier

A persistent business object must be associated with a particular database (a persistence context) and have a unique key to differentiate it from other objects of the same type. This key will appear as one or more object attributes. In order to provide an appropriate key for various operations, each business object has a **readonly** attribute called **the_identifier**. This attribute returns a list of name value pairs, specifying the attribute names and their values, which make up the identifier.

```
typedef string          Istring;
typedef Istring        BofName;

struct BofNameValue{
    string    name;
    any value;
};

typedef sequence<BofNameValue> BofNameValues;

typedef BofNameValues BofObjectIdentifier;
typedef BofNameValues BofBusinessObjectState;
```

```
typedef BofNameValues BofInitializerList;  
typedef BofNameValues BofCriteria;
```

The persistence management service associated with a particular database accepts the object's identifier as an input to most of its operations and uses it along with the object type to identify the object's data in the persistent storage. Persistence management is discussed later. All business objects are required to have this identifier.

2.4.4.2 External identifier

In order to send a message to a business object, its object reference must be obtained from the naming service. The user or other external source will give the object a name for future reference. This is an external identifier which is bound to the object reference in the Naming Service. The identifier will be bound within a context appropriate to its usage so that it will uniquely identify the object. For example, the context for part numbers might be part names within a particular corporation and for social security numbers would be United States taxpayers. When the user uses the external name to refer to the object, the application will associate the appropriate context and use the Naming Service to resolve the external name to the appropriate object reference.

An object may have multiple identifiers, and an identifier must be unique only within a naming context. An external identifier, qualified by a naming context, is frequently referenced in this specification as a qualified name. One name binding must always exist for any business object and thus every business object must have a unique qualified name. The BOF will use the value of **the_identifier** qualified by the object type to create this primary identifier. The primary identifier will always be bound to the most recent object reference for the object.

2.4.5 Life Cycle Services Interface

A business object supports the remove, copy and move life-cycle methods. The create method is implemented on the life cycle manager. The **BOFLifeCycleObject** interface is specified below. This is the same in most respects as the interface defined in Section 6 of the CORBA Services specifications.

```
typedef BofNameValues BofObjectList;  
  
interface BofLifeCycleObject{  
    void remove() raises(BofException);  
    void copy(in BofFactoryFinder there,  
             inout BofObjectList the_copied_list)  
             raises(BofException);  
    void move(in BofFactoryFinder there,  
             inout BofObjectList the_moved_list)  
             raises(BofException);  
};
```

The life cycle methods are discussed in the following paragraphs. The **BofFactoryFinder** specified in the **copy** and **move** operations is a specialized factory finder discussed later in the Workload Management interfaces section. The factory finder returns a life cycle manager (which provides a factory service) in an appropriate address space based on the type of the object being copied or moved.

2.4.5.1 Remove

The **remove** method will cause the object to be removed if it can be deleted from all relationships. The immediate effects of **remove** is to make the object logically no longer accessible. Removal from the database and from memory will not occur until the removing transaction commits. When **remove** is committed, any outstanding requests for change notification will be issued **removed** notice and the notification requests will be terminated.

2.4.5.2 Copy

The **copy** method will create a replica of the target object in the same or a different address space. If the **there** parameter is nil, the copy will be in the same address space, otherwise **there** is a BOF factory finder which returns a life cycle manager in an appropriate address space. The default **copy** method will not traverse relationships automatically but will simply obtain an **BofNameValues** of the target object attributes and relationships and pass it to life cycle manager in the target address space to instantiate the object. The object will then add itself to the **copied BofNameValues** in the criteria sequence of name-value pairs; if there is no **copied BofNameValues** then one is created. The **copied BofNameValues** is given the original object reference as a key and the new object reference as its value. Recursive traversal of relationships may be accomplished by specialization of the **copy** method on the business object implementation.

The recursive copy protocol is to first to check the **copied BofNameValues** list to see if the current object is already in the list as copied; if it is, then control is simply returned. Second, obtain an instance of the class of this object from the appropriate object manager passing the identifier to be used for the copy. Put the stringified object reference of the current object in the **copied BofNameValues** list as a key with the stringified object reference of the new object as the associated value. Third, a **BofNameValues** list of attributes and relationships is built for the new object to be created. This list is built by using the current value of each attribute and the current values of each relationship that is not being traversed. If a relationship is to be traversed, then a **copy** message is sent to each related business object, the **copied BofNameValues** list is passed in the **copy** criteria sequence. After the **BofNameValues** list of attributes and relationships for the new object is built, it is sent to the life cycle manager to initialize the new object. Control is returned.

The **copy** method signature is the same form as defined by the CORBA Life Cycle Service except that the criteria **BofNameValues** list is an in-out parameter. The return value is the object reference of the copied object.

2.4.5.3 Move

The **move** method will delete the target object from its current address space and re-instantiate it in a designated address space. The target address space is determined by the **there** parameter which must be a BOF factory finder. Since the object will be given a new object reference, this new object reference will be given to the Naming Service. Any outstanding requests for change notification will be moved with the object. Notices will be sent to each requester providing the new object reference.

The default **move** method will not traverse relationships automatically but will simply obtain an **BofNameValues** list of the target object attributes and relationships and pass it to life cycle

manager in the target address space to instantiate the new object. The object will then add itself to the **moved BofNameValues**list in the criteria sequence of name-value pairs; if there is no **moved BofNameValues**list, then one is created. The **moved BofNameValues**list is given the original object reference as a key and the new object reference as its value. Recursive traversal of relationships may be accomplished by specialization of the **move** method on the business object implementation.

The recursive move protocol is to first to check the **moved BofNameValues**list to see if the current object is already in the list as moved; if it is, then control is simply returned. Second, obtain an instance of the class of this object from the appropriate object manager passing the identifier of this object. Put the stringified object reference of the current object in the **moved BofNameValues**list as a key with the stringified object reference of the new object as the associated value. Third, a **BofNameValues**list of attributes and relationships is built for the new object being created. This list is built by using the current value of each attribute and the current values of each relationship that is not being traversed. If a relationship is to be traversed, then a **move** message is sent to each related business object, the **moved BofNameValues**list is passed in the **move** criteria sequence. After the **BofNameValues**list of attributes and relationships for the new object is built, it is sent to the life cycle manager to initialize the new object. The current object is **removed** and control is returned.

The **move** method signature is the same form as defined by the CORBA Life Cycle Service except that the criteria **BofNameValues**list is an in-out parameter, and the operation returns a value (not void). The return value is the new object reference for the moved object.

2.4.6 Change Notification Interfaces

Change notification is a service provide by individual business objects. Upon request, a business object will send a notice whenever designated aspects of the business object change--(aspects include attributes and relationships). There are two components to the change notification interface: the supplier and the consumer. The supplier provides notification of changes after receiving a request for notification. The consumer receives the notices as changes occur. The consumer need not be the source of the request. The supplier will continue to send notices until the request is terminated.

There are three levels of service for notices:

1. Immediate. Immediate notification occurs as changes are applied and before they are committed. This allows the activity of an application to be observed as it happens. Notification in this mode will be terminated if either the supplier or the consumer become inactive.
2. Persistent and immediate. This provides notices in the same manner as above, except that the request for notification will be preserved if the supplier becomes inactive, but will be terminated if the consumer becomes inactive.
3. Guaranteed delivery. This guarantees that the request will be preserved if the supplier or the consumer become inactive, and when the consumer is inactive notices will be persistent until they can be delivered. In this mode, notices are only issued when the supplier commits at which time the notices are also committed for delivery.

Notice of change will be issued by the supplier under the following circumstances:

1. The value of an aspect of interest changes and an immediate notification request exists for that aspect. The notice will contain a **BofNameValues** list with the name of the aspect and its new value.
2. An object is committed when the value of one or more aspects of interest has changed and a guaranteed notification request exists for those aspects. The notice will contain a **BofNameValues** list with the names of each of the changed aspects and their new values.
3. An object is deactivated with outstanding transient requests for notification (immediate level of service). The notice will contain a **BofNameValues** list with the aspect “deactivated” and a value true.
4. An object is deleted with outstanding requests for notification. The notice will contain a **BofNameValues** list with the aspect **deleted** with a value of true along with names and values of other aspects of interest to the consumer. Since **deleted** will imply that the object no longer exists, the **deleted** notice will only be sent when the associated transaction commits, even for the **Immediate** level of service.
5. An object is a candidate for deactivation but has outstanding requests for notification that are not persistent. A **Polling** change notice may be issued to determine if the consumer is still active. The consumer can ignore a **Polling** message since the supplier will receive an exception if the consumer is no longer available.
6. When an object is committed, it will send a **Committed** change notice (“Committed” name and “true” value). This will be included in 2, above, for guaranteed service.
7. When an object is rolled back, it will send a **Rollback** change notice to all outstanding immediate notification requests (“Rollback” name and “true” value).

Note that, except for the special status events described above, a consumer will only receive notices of change to aspects specifically designated in the request. The notice uses a **BofNameValues** list so that multiple aspect changes can be reported in a single message. This is particularly important for the guaranteed delivery mode where notice of multiple changes will often occur when changes are committed.

2.4.6.1 Supplier interface

The supplier interface is specified below. It has a single method.

```
typedef sequence<string> BofAspects;

enum BofNotificationLevel { Immediate, Persistent, Guaranteed };

interface BofNotificationSupplier{
    BofBusinessObjectState notify(in BofAspects aspects,
        in BofNotificationConsumer consumer,
        in octet consumer_parm,
        in BofNotificationLevel level)
        raises(BofException);
    void stop_notice(in BofNotificationConsumer consumer,
        in octet consumer_parm)
        raises(BofException);
}
```

```
};
```

A request for notification is directed to the supplier business object which is the source of the events. The **notify** request specifies the receiver, the aspects of interest, a consumer notice parameter to be returned with notices and a level of service designation.

The aspects of interest may be changed for an active request by sending a message with the same form and parameters except for an appropriately modified list of aspects of interest.

A request for termination of notification is directed to the supplier business object as a **stop_notice** message. The consumer and consumer notice parameter must be the same as the original request, but the level of service parameter may be nil.

The request returns a BofNameValues list which contains the names and current values of the aspects of interest. For details of the BofNameValues structure, see the discussion of the business objects state data.

2.4.6.2 Consumer interface

The consumer interface also has a single method and is specified below.

```
interface BofNotificationConsumer{
    void notice(in BofNotificationSupplier supplier,
               in octet consumer_parm,
               in BofBusinessObjectState state);
};
```

The interface accepts notices. The **notice** message includes the object reference of the supplier, the consumer notice parameter which was provided with the request, and **BofNameValues** list of the changed aspect(s) and their associated new values. The object reference of the supplier is included because under some circumstances it may have changed since the request was submitted. The notice parameter is provided to allow the consumer to associate the notice with the reason for the request to accommodate situations where the consumer may provide a service that is driven by events from different supplier objects.

2.4.7 Transactional Resource Interface

The transactional object interface is inherited by all business objects and any other objects that require BOF-compatible transaction control. This includes any object that may be required to take some action when the transaction commits.

In general, a transactional resource has state that is affected by the operations of a transaction. The identity of a transaction is passed implicitly in the BOF environment. The change of state of the resource must be preserved if the transaction commits or rolled back if the transaction is rolled back. The resource is also expected to support transaction serialization by detecting when it is being accessed by a second transaction when a first transaction is still active. In such a circumstance, the resource must request the first transaction coordinator to suspend the second transaction pending termination of the first transaction.

The resource must also implement methods for **validate**, **prepare**, **commit** and **rollback**. The **validate** method is discussed below. Prepare is the first phase of a two-phase commit where the resource informs the coordinator if it is prepared to commit its state. When **commit** is received, the resource commits the changed state. The resource must also be able to reverse the effects of a transaction if it receives a rollback message.

The following resource interface specification does not express the prepare, commit and rollback method signatures because they are inherited.

```
interface BofTransactionalResource :
    CosTransactions::Resource,
    CosTransactions::TransactionalObject{
    void validate()
        raises(BofException);
};
```

2.4.7.1 Validate method

The **validate** method is invoked to evaluate conditional expressions which will determine if the state of the object, and related objects if appropriate, meets business policy and object model integrity requirements. The method returns a sequence of exceptions as an exception if any unacceptable condition is encountered. The intention is that the method continues to evaluate its conditions even though an exception is encountered so that all exceptions can be returned to the application for potential resolution by the user. Likewise, the transaction service is expected to invoke all **validate** methods to identify all errors, rather than stopping when the first **validate** encounters one or more exception. On the other hand, for the **prepare** method, the transaction service is expected to stop as soon as **prepare** returns a vote of **Rollback**.

The **validate** method is always invoked prior to the prepare phase of the transaction two-phase commit to determine if the transaction should be allowed to commit. The validate method may be executed at any time by an application to obtain a sequence of exceptions regarding the current state of the object.

2.4.8 View Request Interface

The view request interface is provided for an object which provides alternative interfaces, i.e., views. A view defines the methods the target object will accept from a set of applications and users. The view request interface defines one method, **get_view**. A **get_view** returns an authorized view. The view request interface is defined separately from the business object interface since there may be circumstances where a view should be available from an object that is not a BOF supported business object. The interface is specified below.

```
interface BofViewRequest
    BofView get_view (in string view_context)
        raises (BofException);
};
```

The type of view returned will depend upon the **view_context** parameter. The **view_context** parameter, generally, defines the actions that are authorized for a category of users performing a

category of activities, i.e., it defines a set of views. Access to the view is controlled by the Security Service, so that if the user is not authorized for the view returned, any attempts to use it will be rejected.

2.4.9 View Interface

A view is an interface that supports a particular usage of the object it represents. Since a number of different implementations (i.e., classes) may participate in the same usage, the view must be defined as the polymorphic interface for the classes that participate in that usage. Since a view will, generally, not support certain life cycle methods such as move and copy, it may not be defined by inheritance from more abstract business object interfaces, but rather by explicit expression of the method signatures that are to be enabled. Views inherit from the View interface specified below.

```
interface BofView {
    readonly attribute string view_context;
    BofNameValues notify (in BofNotification :: BofAspects aspects,
                        in BofConsumer consumer,
                        in string consumer_parm,
                        in string level)
        raises (BofException);
};
```

The **view_context** attribute defines the context in which the view is used and is defined as discussed above for the view request interface.

The **notify** method has the same signature as the **notify** method for change notification, but is essentially a pass-through method with one exception. The method logic must screen the attribute and relationship names passed in the **aspects** parameter to assure that the caller is not asking for notification of aspects it is not otherwise authorized to access.

2.4.10 Workload Management Service Interfaces

The Workload Management Service provides for the distribution of workload based on system activity. The **BofWorkloadManager** interface provides for administrative control. The **BofWorkloadManagerBasedFinder** returns a life cycle manager based on current workload, and the **BofWorkloadLogger** tracks activity.

The workload manager maintains information on system activity based on host, component instance, and life cycle manager. A *host* is a computer which may have several active address spaces. A *component* is an executable which may be executed as one or more active address spaces on one or more hosts. Each active address space is a *component instance*. A *life cycle manager* is responsible for the creation, activation and deactivation of objects of a particular class in and address space. The activity at each of these levels will be used to determine appropriate allocation of workload.

2.4.10.1 Workload manager

The Workload Manager interface provides information based on the availability of life cycle managers and the current distribution of workload. The interface is specified below.

```

struct BofManagedTypeWorkload { //structure for workload data
    string managed_type_name;
    string manager_name;
    long active_load;
    long reference_count;
};
typedef sequence<BofManagedTypeWorkload>
    BofManagedTypeWorkloadList;

interface BofWorkloadManager{
    BofLifeCycle::BofLifeCycleManager find_manager(
        in string object_type,
        in BofCriteria the_criteria)
        raises(BofException);

    void get_host_load_list(in unsigned long batchSize,
        out BofWorkloadList workload,
        out BofWorkloadListIterator iterator)
        raises(BofException);

    void new_component(in string component_name,
        in string executable_name,
        in BofManagedTypes managed_types)
        raises(BofException);

    void remove_component(in string component_name)
        raises(BofException);

    BofComponents get_components_list()
        raises(BofException);

    BofComponentInstanceWorkload new_instance(
        in string component_name,
        in string host_name)
        raises(BofException);

    void remove_instance(in string component_name,
        in long instance_number)
        raises(BofException);

    void activate_instance(in string component_name,
        in long instance_number)
        raises(BofException);

    void deactivate_instance(in string component_name,
        in long instance_number)
        raises(BofException);
};

```

The **least_loaded_management** method returns an appropriate life cycle manager based on the **object_type** and the **criteria** specifications.

The **get_host_load_list** returns current workload information for each life cycle manager.

The **new_component** method registers an execution component with the workload manager for monitoring. The **remove_component** method removes it. The **get_components_list** returns the currently registered components.

The **new_instance** method identifies a new address space as an instance of an executable module. The **remove_instance** method drops the instance (presumably it is no longer active). The **activate_instance** and **deactivate_instance** methods define the life cycle manager's availability to create objects.

2.4 10.2 Workload manager based finder

The **BofWorkloadManagerBasedFinder** interface is used by the **copy** and **move** operations or in other cases where there is a need to define a mechanism to obtain members of a category of life cycle managers (e.g., life cycle managers at a particular geographical location). It is based on the **FactoryFinder** interface defined for CORBA Life Cycle Services. However, it only returns a single life cycle manager (i.e., factory). It interfaces to the Workload Management Service to request a life cycle manager for an object of type **factory_key** based on the criteria installed in it by its factory.

```
interface BofFactoryFinder{
    BofLifeCycleManager find_factory(in string factory_key)
        raises (BofException);
};
interface BofWorkloadBasedManagerFinder :
    BofLifeCycle::BofFactoryFinder{
};
```

2.4 10.3 Finder factory

The workload management **BofFactoryFinderFactory** creates an instance of the **BofWorkloadManagerBasedFinder** described above with selection criteria for location of life cycle managers. The interface is specified below.

```
interface BofFactoryFinderFactory{
    FactoryFinder create(in BofCriteria the_criteria);
};
interface BofWorkloadBasedManagerFinder :
    BofLifeCycle::BofFactoryFinder{
};
```

2.4 10.4 Workload logger

The workload logger interface accepts data on current server activity. This information is used to monitor system activity and distribute workload.

```
interface WorkloadLogger{
    oneway void LogObjectData(in string instance_name,
        in BofObjectDataList object_data_list);
};
```

2.4.11 Transaction Coordinator Interface

The BOF transaction coordinator interface is a specialization of the COS Transaction Service coordinator interface with the addition of the `will_deadlock_transaction` and `waiting_for_transaction` methods.

The `will_deadlock_transaction` method accepts the identity of a transaction about to be suspended pending completion of the target coordinator transaction. The coordinator must determine if suspending the `pending_tc` will cause a deadlock, in which case the caller will terminate rather than suspend the waiting transaction.

The `waiting_for` method notifies the Transaction Service that the target transaction is waiting for the parameter transaction. This provides the information necessary for deadlock detection. This method is called after the `will_deadlock_transaction` returns false.

```
interface BofCoordinator : CosTransactions::Coordinator{
    boolean will_deadlock_transaction(
        in CosTransactions::Coordinator pending_tc)
        raises(BofException);
    void waiting_for(in CosTransactions::Coordinator tc)
        raises(BofException);
};
```

2.4.12 Life Cycle Manager Interface

The life cycle manager is an extended factory for business objects. In addition to functioning as a factory, it manages business object activation and deactivation. The interface is specified below.

```
interface BofLifeCycleManager : CosTransactions::TransactionalObject{
    BofLifeCycleObject create(in string persistent_context,
        in BofInitializerList init_values)
        raises(BofException);
    BofLifeCycleObject activate(in BofName qualified_name,
        in string persistent_context)
        raises(BofException);
    BofName get_default_naming_context(
        in BofName qualified_name);
};
```

The create method takes a `persistent_context` parameter (required if the object is persistent) and an optional initial values `BofNameValues` list. The `persistent_context` parameter is discussed in the Persistence Management Interfaces section, below. It is a prefix for the object identifier which will form a persistence ID; the persistence ID uniquely identifies the object in persistent storage.

The initial values `BofNameValues` list is a list of attribute and relationship names and values. These values will be assigned to the attributes and relationships of the new object.

Activation is on request with the object identifier and persistent context as arguments to identify the object in persistent storage.

Deactivation is managed by the BOF implementation.

2.4.13 Name Service Interface

The Name Service is based on the CORBA Name Service, but with additional information attached to the binding. This information is passed to the **bind** method as indicated in the interface specification, below. This additional information is the object type, the persistent context and the life cycle manager object reference.

```
interface BofNameService{

    // bind a new name to the object
    void bind(in BofName qualified_name,in Object object,
             in string object_type,in string persistent_context,
             in Object manager)
        raises(BofException);

    // resolve a name to object reference
    Object resolve(in BofName qualified_name
                 raises(BofException);

    // resolve a moved reference
    Object resolve_ref(in Object old_object)
        raise(BofException);

    // unbind a name
    void unbind(in BofName qualified_name)
        raises(BofException);

    // rebind all the data associated with previous binding
    void rebind(in BofName qualified_name,in Object object,
              in string persistent_context,in Object manager)
        raises(BofException);

    // this rebinds recursively for all other names of the previous
    // object
    void move(in Object old_object,
            in Object new_object,
            in Object new_manager)
        raises(BofException);

    // remove all names bound to the object
    void remove(in Object object)
        raises(BofException);
};
```

The object reference of the life cycle manager is held to prevent duplication of an object which could be activated in alternative address spaces. The default will be to activate objects where they were previously active.

The **resolve** method takes an identifier with a context (qualified name) and returns the associated object reference. The **unbind** method removes the binding identified by the context and

identifier (qualified name). **Therebind** changes the binding associated with a context and identifier. The move is similar to **rebind**, except that it does the equivalent operation on all bindings for the object reference. **The remove** method removes all bindings for the specified object reference. **The move** operation will redirect all bindings for an object to a new object reference with a new object manager (i.e., a new address space). **The resolve_ref** will return the current object reference for an old object reference.

2.4.14 Query Evaluator Interface

Queries are performed against actual or implicit collections of objects. A query evaluator performs a query and returns a query result. For BOF queries, a query evaluator will always return a **BofObjectIterator** which iterates over a sequence of **BOFNameValues**. Each **BofNameValue** represents a tuple from the query where the names identify query attributes. The **BofObjectIterator** extends the **BofIterator** interface to provide methods which will return object references to the objects associated with each tuple. If the query was against a collection of object references, the object reference is returned, whereas if the query was against a database, the associated object will be activated in order to return its object reference.

In this way, a query can retrieve attributes of interest to display a “list” or “table” of items without activating all of the associated objects. When one of the items is selected, that object can then be activated.

All BOF one-to-many relationships return a **BofIterator** which will also function as a query evaluator. A **BofDBS** (database service interface representing a database connection) will provide a query evaluator which also returns a **BofObjectIterator** as a result. The interface specification below reflects the addition of method to the iterator interface to provide access to the associated objects.

```
interface BofObjectIterator : Bof::BofIterator{
    boolean next_object(out Bof::Bof BusinessObject object);
    boolean next_n_objects(in unsigned long how_many,
        out Bof::BusinessObjects objects);
    boolean next_object_at(in unsigned long at,
        out Bof::BofBusinessObject object);
};

interface BofQueryableIterator : BofObjectIterator,
    CosQuery::QueryEvaluator{
};
```

The evaluator protocol is consistent with the COS Query Service query evaluator interface.

2.4.15 Exceptions

BOF facilities return exceptions as a **BofException** a “user-defined” exception form. The **BofException** is a sequence of exceptions having a consistent form. This allows multiple exceptions to be returned so that multiple problems may be addressed, as where a user has a number of data entry errors. It also simplifies the coding for exceptions by allowing more generic logic than standard CORBA exceptions.

```

enum BofExceptionCategory { EnvironmentException,
                             NormalException,
                             DataEntryException,
                             RuleViolationException,
                             IntegrityException,
                             ServiceError };

struct BofError{
    BofExceptionCategory    exception_category;
    string                  exception_source;
    long                    exception_code; // specific to the source
    string                  exception_reason;
};

typedef sequence <BofError> BofErrors;

exception BofException{BofErrors errors;};

```

2.4.16 Business Object State Data

The state of an object must be passed between address spaces under several circumstances: when initial values are passed for creation, when it is saved or restored in a database, when it is moved or copied, and for change notices. These operations all use the same form for passing state data: the **BofNameValues** form.

BofNameValues is an **NVList** as defined in Section 17 of the Core CORBA specifications. This structure allows values to be associated with each attribute or relationship name. Attributes will have a single value, but relationships may have multiple values.

The values of relationships are placed in the **BofNameValues** list as sequences of values. When only changes are being passed, two entries, differentiated by flags, may be passed for a changed relationship: one is added relationships and the other is deleted relationships. The form is specified in pseudo IDL, below.

```

typedef string              Istring;
typedef Istring             BofName;

struct BofNameValue{
    string    name;
    any      value;
};

typedef sequence<BofNameValue> BofNameValues;

typedef BofNameValues BofObjectIdentifier;
typedef BofNameValues BofBusinessObjectState;
typedef BofNameValues BofInitializerList;
typedef BofNameValues BofCriteria;

```

The valid attributes and relationships for a business object will be specified with DDL (Data Definition Language) which is defined by CORBA as similar to IDL but without functional specifications. In addition to the elimination of operations, the DDL for an object may differ from the IDL because some of the attributes and relationships may be computed values and will not be stored in memory or in the database. In addition, objects with different IDL interfaces

may share the same DDL because they are functionally different but have the same state variables. The DDL name will be passed with the BofNameValues for database operations as part of identifying the mapping of object data to database structures.

2.4.17 Business Objects Iterator Interface

A consistent pattern is employed for access to collections within the BOF environment. Except for the addition of **restart** method, this pattern is adopted from the CORBA Naming Service specification. Here it applies, for example, to access of members of a relationship, to access workload lists in the Workload Management Service, and to access lists of bound objects returned from the Naming Service. The following is a typical iterator interface where the caller may request either the next member of a list or the next “n” members in a sequence.

```
interface BofIterator{
    boolean next_value(out BofNameValue value);
    boolean next_n_values(in unsigned long how_many,
        out BofNameValues values);
    boolean get_value_at(in unsigned long at,
        out BofNameValue value);
    unsigned long count();
    void reset();
    void destroy();
};
```

The **next_one** method returns the next member starting with the first. The **next_n** method returns a sequence of length **batch_size** the next group of members. The **reset** method causes the iterator to return to the beginning of the collection.

The following is the pattern for a method signature that returns such an iterator. The **batch_size** parameter defines the size of the initial sequence to be returned. If the number of members is equal to or less than the **batch_size**, then the iterator parameter is **nil** on return, otherwise it contains an appropriate iterator.

```
void get_list (in unsigned long batch_size,
    out BofList list,
    out BofIterator iterator)
    raises (BofException);
```

2.5 Business Object Persistence Management Interfaces

Persistence Management provides a high-level abstraction of persistent storage mechanisms. It allows objects of the same class to be stored in databases of diverse types, and it allows shared persistent objects to be dynamically re-assigned to different BOF's, from the same or different vendors.

Persistence Management is defined separately from the business object interoperability specifications because interoperability between business objects does not depend upon standardization of the Persistence Management interfaces. Specifications for these interfaces will enable BOFs to utilize different databases and Persistence Management services and will allow

business objects to move between different BOFs while the same database continues to provide persistence.

Persistence management requires that the data storage requirements of objects be specified separately from their interface definitions. This is because attributes and relationships in specified in IDL may not be stored in the object, but may be computed values. The data storage requirements of an object should be specified in DDL (Data Definition Language) which has been defined by OMG as similar to IDL but without and specification of functionality--only stored attributes and relationships are defined. When an object is stored or retrieved from a database, its DDL specification is identified for communication with Persistence Management to define what is to be stored and retrieved.

In the communication between a BOF and a Persistence Management service, the state of an object is passed in a **BofNameValues** list. The structure of this **BofNameValues** list is the same as that used for the copy, move and change notification operations. When an object is activated, the **BofNameValues** will contain all of the necessary attribute and relationship data elements to instantiate the object. When changes to the object are committed, only the changed values will be communicated in the **BofNameValues** so that the transfer of data is minimized and the database update operations may be more efficient.

Persistence management must then translate between the object data in the **BofNameValues** to the storage structures of the database. This mapping will be implementation dependent and will be related to the type of database (e.g., object-oriented or relational) and the database schema. This will allow the same object classes to be stored in different databases, such as databases of legacy applications.

The Persistence Management service must also support queries. The Database Service Manager provides a DBS (database service) which is a connection to the database and is also a query evaluator. This query evaluator supports the specialization of the CORBA Query Service described earlier in this specification to provide access to the results of a query through a consistent iterator.

The two Persistence Management interfaces, the Database Service Manager interface (**BofDBSManager**) and the Database Service interface (**BofDBS**) are described below.

2.5.1 Database Service Manager Interface

The **BofDBSManager** allocates connections to databases. Connections are represented by **BofDBS** objects, discussed below. A connect request returns **BofDBS** which is used to perform the database operations associated with a particular transaction. If multiple requests are received for the same transaction, the DBS manager will allocate the same DBS object, i.e., it will assign the same connection. When the transaction commits, the DBS will be released back to the DBS manager with the disconnect method.

The **register** and **un_register** methods are for creating the DBS objects associated with each database. When the database is activated, the appropriate number of DBS objects are registered with the DBS manager. If the database is shut down, the **un_register** method will remove the DBS objects from availability.

The **connect** method returns a **BofDBS** as a connection to the specified database. The **disconnect** method returns the connection to the available pool.

```

interface BofDBSManager {
    BofDBS connect(in string persistent_context,
                  in BofTransactions::BofCoordinator tc)
        raises(BofException);
    oneway void disconnect(in BofDBS dbs);
    void register_DBS(in string datastore_type, in BofDBS dbs);
    oneway void un_register_DBS(in BofDBS dbs);
};

```

2.5.2 Database Service Interface

An instance of **BofDBS** represents a connection to a database. There may be many connections to a particular database depending upon the licensing arrangements. The use of **BofDBS** objects manages compliance with the licensing restrictions. The interface is specified below.

```

interface BofDBS : BofTransactions::BofStorageResource
    CosQuery::QueryEvaluator {
    readonly attribute string datastore_type;

    void initialize ( in string stringified_pid);
    void create(inout BofObjectIdentifier id,
               in BofBusinessObjectState data)
        raises(BofException);
    void update(in BofObjectIdentifier id,
               in BofBusinessObjectState data)
        raises(BofException);
    void restore ( in BofObjectIdentifier id,
                  out BofBusinessObjectState data)
        raises(BofException);
    void delete ( in BofObjectIdentifier id)
        raises(BofException);
};

```

BofDBS provides the interface for database operations. The DBS manager will assure that only one DBS is allocated per database for any transaction. When allocated, the DBS registers as a transaction resource in order to receive **prepare** and **commit** messages.

The DBS provides four database operations. The **create**, **update** and **restore** operations take a **BofBusinessObjectState** parameter. This parameter contains a **BofNameValues** list (see discussion in business objects interoperability section). For **create**, all initial values are passed; for an **update**, only the changed values are passed.

The **restore** method retrieves the state of an object from the database and returns the attribute and relationship values in the out parameter, **BofBusinessObjectData**.

The persistence management service is responsible for translating between the **BofNameValues** list and the data storage structures. A DDL specification for the state data elements for objects of a particular class defines the object data requirement independent of the BOF implementation. The names and types of the data in the **BofNameValues** list must correspond to this specification which is also the basis for the mapping specification.

The **commit** protocol will provide all of the updates associated with a transaction before the DBS receives a **prepare** message. This allows overlapping updates for the same transaction to be consolidated by the DBS to minimize database activity.

2.6 External Interfaces

In general, business objects incorporate other CORBA services through the BOF. Other external components, including other BOFs, should interoperate with the business objects through the business object interfaces defined above. This includes agents and user interfaces. Adaptors can be used to implement these interfaces so that common components, such as alarm agents or graphical displays may be adapted to link to specific business objects and be installed as ad hoc plug-and-play components in the distributed objects environment.

2.7 Extensions to IDL and DDL

Two extensions to are proposed here. The first adds a view declaration to IDL and the second adds relationship declarations to IDL and DDL.

2.7.1 View Declaration

A view declaration could be the same as a normal interface declaration except that it must identify the primary interface from which the view derives. The methods defined on a view must be a subset of the methods defined on the primary interface. The names and signatures of the attributes, relationships or methods that it includes must match the corresponding declarations on the primary interface except that a qualification of “read only” must be allowed to limit access to selected attributes or relationships. The expansion of relationship declarations on the view will never include relationship **private_add** and **private_delete** methods because they are only for use by the BOF within the same level of abstraction.

A view declaration has the following form as a modification to IDL specifications:

```
<interface_dcl>      ::= <interface_header> { “<interface_body>” }
<interface_header> ::= [ “view” ] interface “<identifier>”
```

2.7.2 Relationship Declaration

A relationship declaration is similar to an attribute declaration except that a relationship may have one or multiple values and a relationship has a corresponding relationship on objects it references. This results in the implicit declaration of different accessor methods. Without the relationship declaration in IDL, it would be necessary for these methods to be explicitly declared for each relationship.

The DDL specification is used for the Persistence Management interface. It defines the names and types of data elements to be stored and consequently the names and types of data elements passed between a BOF and a Persistence Management service. It defines the BOF side of the mapping of business objects to data storage. Without a DDL relationship declaration, the requirement for multiple values in the object state information would not be defined.

The relationship declaration must include a specification of the name of the complementary relationship on objects that participate in the declared relationship. This defines a requirement of participating objects and provides information necessary for the BOF to maintain the complementary relationship. The form of the relationship declaration for IDL is specified below.

```

<export> ::= <type_dcl>“;”
           | <const_dcl>“;“
           | <except_dcl>“;”
           | <attr_dcl>“;”
           | <op_dcl>“;”
           | <rel_dcl>“;”
<rel_dcl> ::= [“readonly”]”relationship””one”|”many”
             <param_type_spec><simple_declarator>“inverse””one”|”many”<comp_relationship>

```

The declaration for DDL is the same except that the readonly qualifier is not included.

```

<export> ::= <type_dcl>“;”
           | <const_dcl>“;“
           | <except_dcl>“;”
           | <attr_dcl>“;”
           | <op_dcl>“;”
           | <rel_dcl>“;”
<rel_dcl> ::= ”relationship””one”|”many”
             <param_type_spec><simple_declarator>“inverse””one”|”many”<comp_relationship>

```

Other aspects of the relationship such as the roles of the participants (e.g., container or contained) and requirements for relationship traversal by life cycle operations are considered to be private to the implementation of the business object and inappropriate to define in the interface specification.

Part 3, Compliance with RFP Requirements

3.1 Compatibility with Existing OMG Standards

3.1.1 Object Request Broker

An extension to the ORB specification is needed for BOF support. The goal of the BOF is to provide an abstraction that conceals computational details. To achieve this, it is desirable that the BOF be able to intercept system level exceptions and determine appropriate action from a business systems perspective. Consequently there is a need to be able to intercept exceptions being returned through an ORB. When the business programmer sends a message to another business object or service, the BOF should be able to intercept exceptions being returned. In this way, exceptions related to deactivation of an object, failure of a server, or other system level exception could be best handled by the BOF and would avoid the need for considerable exception code within business system methods.

3.1.2 Transaction Service

The proposal adopts the current Transaction Service with two modifications:

1. The two-phase commit must be extended to issue a validate message to each affected object prior to the prepare phase. The validate message will be issued to all business objects even if some exceptions are encountered.
2. All business objects must receive the prepare message before any database resource, and similarly all business objects must receive the commit message before any database resource. This provides for staging database updates (updates from all business objects) before the database operation is performed.
3. The transaction coordinator must accept two additional message types for deadlock detection. The first message provides information regarding the active transaction that is causing a subject transaction to be suspended, and the second message requests deadlock detection when a transaction is about to be suspended.

3.1.3 Security

The proposal incorporates the CORBA Security Service to control access to views which provide the primary control for access to shared business objects. Views restrict the particular user's access to a shared business object. Access decision functions, which are invoked by the Security Service, will further qualify the user as where an employee is authorized to see his/her salary information but not the salary information of other employees.

3.1.4 Naming Service

The current Naming Service is extended by this specification to include additional binding data: the identity of the database in which a persistent object is stored, and the identifier of the life cycle manager which manages its activation and deactivation. This information is necessary for

providing a consistent mechanism for resolution of external identifiers, re-activation of objects, activation of objects in databases shared with external systems (e.g., legacy systems), and location of objects moved or activated on a different server.

The current CORBA specification provides for location forwarding. With location forwarding, the owner of the server side of a connection (typically an ORB) returns `LOCATION_FORWARD` status to which the client may submit `location_request` message to obtain the new object reference. However, the server is not required to implement location forwarding. Furthermore, location forwarding relies on the continued existence of the server that issued the original object reference. The target object may, in fact, have been moved so that the server could be shut down.

The current CORBA specification also allows for an ORB to activate a persistent object, but there is no specification for this mechanism. In addition, this activation mechanism assumes the continued existence of the server that issued the object reference for the now unavailable object. Furthermore, while the ORB may activate the object, there is no mechanism to assure that the object has not also been activated on a different server.

For these reasons, we see the Naming Service as the master point of reference for locating objects. If the object is active, then the Naming Service provides the object reference. If it is inactive, the Naming Service has the location and identity of its persistent state.

3.1.5 Relationship Service

The existing Relationship Service is not incorporated in this proposal. The current service design is overly complex and is designed to contain objects in a relationship rather than to provide an encapsulated mechanism for maintaining relationships intrinsic to the objects. The current service is more suited to relationships that are externally attached to objects.

This proposal encapsulates relationships in order to reliably implement BOF services and maintain relationship integrity. The proposed approach also reserves control of traversal of relationships (e.g., for copy, move and delete operations) to the object implementation. To the extent relationships should be traversed, the associated method should be specialized by the business object developer.

3.1.6 Collections

The proposal does not expose any collection interfaces. Relationships incorporate collections, but they are fully encapsulated. Access to such collections are only through relationship accessor methods, through an iterator returned by a relationship method or through a query evaluator returned by a relationship method.

3.1.7 Event Service

The proposal does not incorporate the event service for change notification since the requirements differ. On the surface, change notification is a direct communication of events between a supplier and a consumer. However, in this mode, the event service only anticipates a one-to-one relationship--one supplier and one consumer--whereas change notification supports many-to-many supplier to consumer relationships.

The many-to-many model could be implemented with event channels, one per consumer for each supplier. However, each notification request is qualified with the aspects of interest so that only some of the events from a supplier should be forwarded to a particular consumer.

In addition, the authorized events for a consumer must be restricted to those for aspects the consumer has authority to access. The proposal provides for screening of event requests by the view which provides authorized consumer access to the supplier business object. The view will only allow requests for change notification for authorized aspects.

The proposal simplifies the interface and minimizes network traffic by causing all notification requests to be directed to the supplier. Immediate (light weight) notices are communicated directly from the supplier to the consumer. While the **push** method might be used for notices, there is no benefit to using the same signature for a significantly different service. In addition, the structure that would be passed as a single argument would result in more application code and less clarity than a method signature that defines explicit arguments.

3.1.8 Persistence Service

This specification does not incorporate the current CORBA Persistent Object Service (POS). The proposed specification provides the following capabilities not supported by the POS:

1. A mechanism for transfer of object state values using an **NVList** structure. The structure contains only changed values for an update, reducing the amount of data transferred between the object's address space and the persistence service.
2. A representation and framework for management of connections to the database supporting the use of one connection per transaction. The current persistence service does not manage available connections.
3. A more concise, consistent and explicit interface for communication of BOFs with the persistence management facility. This specification adopts DDL as the storage requirement specification language. The consistency and technology independence of this interface supports the runtime mobility of business objects between BOFs.
4. This specification provides a separation of object instantiation from persistence management which provides greater flexibility and mobility of object assignments to address spaces.
5. This specification supports transaction serialization and concurrency control on objects independent of their persistence.

3.1.9 Query Service

The specification adopts the existing Query Service but extends the specifications to assure compatibility across BOFs. For example, the result is defined as an iterator which is also a query evaluator.

3.1.10 Concurrency Service

The proposal encapsulates concurrency control within the business object. This is more efficient than using a shared concurrency service for multiple objects. The current COS Concurrency Service can co-exist in the same environment.

3.1.11 Life Cycle Service

This specification adopts a **LifeCycleObject** interface that is consistent with the COS Life Cycle Service. The BOF life cycle operations use a specialized factory finder to interface to the BOF Workload Management service. The generic factory interface is not used because it does not support specification of initial values which are required for creation of a BOF business object.

3.1.12 Exceptions

The proposal includes a “user defined” exception form that simplifies business programming. This form is a sequence of exceptions rather than a single exception. This allows for multiple exceptions to be returned so that multiple discrepancies can be presented to the user to be considered and addressed together. The BOF exceptions have a consistent form for use by common exception-handling code.

3.2 Compliance with RFP General Requirements

The following comments address requirements specified in Section 5.6 of the RFP.

3.2.1 Interoperability

The proposal provides interoperability of business objects between BOF's of different vendors. The protocol also supports interoperability with user interfaces and other client applications that do not have full BOF support.

3.2.2 Separation of Technology Issues

Business objects are abstractions which completely hide the underlying technology. Business objects may be dynamically moved between BOF's of different vendors if those BOF's have compatible implementations of the same classes--they must support the same interface(s) and have the same instance variables even though they might be implemented in different languages.

3.2.3 Extensibility of Business Objects

Business objects can be specialized (i.e., customized) through inheritance, delegation and adaptors. Conventional inheritance and delegation are supported. In addition, the specification defines adaptors.

An adaptor uses delegation to incorporate the state and functionality of a shared object. The adaptor may provide signature translation if the adaptor context has different semantics or was

developed independently. The adaptor can also have local attributes, relationships and operations not present on the shared object.

Adaptors may also provide active representation using change notification. Change notification provides the mechanism by which changes in the primary object are communicated to the adaptor as they occur. Consequently, these changes may be immediately reflected in related aspects of the adaptor object.

3.2.4 Reusability

Reusability is supported at several levels.

- The business object abstraction supports reuse of the business object model across BOF's.
- The implementation of objects can be shared on multiple projects using the same BOF.
- Executables that incorporate BOF code can be shared in multiple environments. Adaptors improve portability through the adaptor signature translation capability to adapt to an independently developed shared object (e.g., enterprise) model.
- Applications--working sets of objects that address a particular business problem-- can be designed with adaptors to link to shared objects. The applications can be plugged into an enterprise environment by adapting the application representation to the shared object representation.
- Business objects can be used with diverse databases including shared, legacy databases.

3.2.5 Scalability

The use of views and adaptors provides for an adaptable interface between applications and shared enterprise objects while providing restricted views for diverse uses of shared objects. The interoperability of BOF's allow business objects implemented with BOF's from different vendors to interoperate. The consistent business object interface allows diverse implementations of applications and user interfaces to incorporate shared objects through views and adaptors. The diversity of underlying technologies is completely concealed behind the business object abstraction.

As a result, a large system can be configured from independently developed components and changes and addition of components can be localized by using adaptors to resolve differences.

3.2.6 Ease of Development and Deployment

There are a number of factors that improve the ease with which business systems can be developed, deployed and enhanced.

- Business system developers can concentrate on the business solution without being concerned about issues of distributed processing, database design and sharing of objects.

- Independent of the business solution, business object implementations can be configured into executables which individually or in groups comprise applications.
- The shared objects representation of the enterprise can be developed and extended with minimal impact on applications through the use of adaptors.
- Applications may be plugged into active networks and link to shared objects to become immediately integrated with the enterprise system
- Shared objects may be dynamically moved from one implementation to another (with appropriate consideration of upward compatibility) to upgrade or convert their implementation on a server-by-server basis.

3.2.7 Application Integration

Applications are integrated through shared objects representing the domain of sharing--viewed as the enterprise. These shared objects are incorporated into each application through adaptors. The applications then work with a shared representation of the state of the enterprise. Applications may have different local object models but share objects through using the signature translation capability provided by adaptors. In addition, an application may be shared by multiple users where each attaches a user interface that shares the application objects.

3.2.8 Security

The primary security mechanism is provided through views. A view defines access to a shared object through a restricted set of methods authorized for the user of the view. Views and adaptors assure that protected enterprise objects never leave the secure environment and are only visible and modifiable to the extent authorized.

Access to the view is controlled by the security service. The creation of views is provided by the associated business object based on a view context parameter which defines the usage of the business object--in general terms, the business function. The business object may incorporate user identification information from the security service to further qualify the request as where an employee is authorized to view his own salary data but not the salary data of others.

3.3 Compliance with RFP Optional Requirements

The following comments address requirements specified in Section 5.7 of the RFP.

3.3.1 How business objects implement the business model

The specification does not restrict the business object programming/specification language. The specification defines an interoperability interface for business objects that supports a high-level business object abstraction. The abstraction may be implemented through code generation in compiled languages such as C++, it might be programmed interactively in an environment like Smalltalk, or it might be implemented with a specialized or proprietary language. These variations in approach are left to the implementer. The specification defines interfaces which all can support and be interoperable with each other.

The specification also provides adaptors which provide for local representation of shared objects. The local representation may be a subset of the shared object, it may perform signature translation to adapt local semantics to global semantics, or to adapt independently developed implementations. The local representation may also add attributes, relationships and operations not defined on the shared object. This allows independently developed business solutions and business models to be reconciled without major rework or redevelopment.

3.3.2 Legacy applications

Legacy applications may be incorporated two fundamentally different ways: through a functional interface or through a shared database.

The functional interface may be implemented with or without adaptors. In either case, the legacy system must provide an IDL interface. If adaptors are used, then the adaptors can provide concurrency control and transaction serialization if they properly represent mutually exclusive state information within the legacy system (i.e., if concurrency can be properly controlled at the interface level). If adaptors are not used, then the legacy system must provide a business object interface with sufficient functionality to support the intended use of the interface.

The shared database approach provides a more straightforward integration mechanism. This specification anticipates legacy database sharing. The Persistence Management facility incorporates the ability to map objects to relational databases in diverse ways where business objects of one class may be stored in heterogeneous databases with different schema and different database management systems. The use of shared databases is the more common practice in client-server implementations of object technology.

3.3.3 Flexibility and longevity

Flexibility and longevity are achieved in three fundamental ways: (1) through the hiding of implementation and infrastructure behind the business object abstraction, (2) through the use of adaptors and (3) through the ability to move objects to different BOF's.

The business object abstraction allows changes in infrastructure components and configurations to occur without affecting the implementation of business objects. Even if the code may require recompilation or relinking, the business object specifications need not change unless there is a corresponding business change.

Adaptors localize changes. Changes in shared objects may be accommodated by signature translations in adaptors in the client applications. Changes in client applications may be kept localized by signature translations and the addition of local attributes, relationships and operations.

The execution location of an object can be changed to cause the same persistent object to execute in a different executable. The executable might incorporate an upgraded version of the object, or it might be an implementation of the same class in a different BOF.

These mechanisms allow incremental, localized implementation of change which are key the flexibility and longevity of large systems.

3.3.4 Generality and desktop integration

Interoperability of business objects at a peer level requires the full functionality of a CORBA environment and compliance with the BOF interfaces. Interoperability is less restrictive for clients of business objects: implementations that depend on the business objects but which the business objects do not depend upon. These may be user interfaces, ad hoc agents, or independent applications. At a minimum the client must be able to send messages to business objects. Depending upon the functionality required, the clients will have additional requirements. If the client will effect changes in the business objects, then the client must begin and terminate transactions. If the client is to be updated as changes occur in the business objects, then the client must be able to receive change notification messages.

These alternative forms of clients might be implemented in various ways with various languages. The communication with BOF objects may be through an ORB or through a COM/CORBA interworking mechanism. The client may be an internet browser with an ORB link. The client may be interfaced through an intermediate application that uses adaptors to convert message protocols or start and terminate transactions. This might be particularly appropriate for integration with desktop applications.

If the client implements shared objects or other concurrent processing, then it will need additional BOF functionality and may be implemented as a BOF application and will require a CORBA infrastructure.

3.3.5 Proof of commonality

This requirement applies to the specification of common business objects which are not part of this submission.

3.3.6 Specification of business objects and metadata

This requirement appears to be focused on specifications for common business objects which are not part of this submission.

For a BOF, the form of specification of business objects and metadata is left to the BOF implementer to determine. Different languages and different programming paradigms will call for different approaches. The fundamental specifications for all BOFs are IDL for interoperable business object interfaces and DDL for specification of the state storage requirements of objects.

The object model might be specified with a proprietary language, a BOF tool or an independent modeling tool which provides specifications that can be imported by the BOF. This is but one form of specification required, however. The BOF must also accommodate differences between business object interface specifications and their implementations. Attributes and relationships may be stored or computed. Methods must be programmed. Adaptors must incorporate mappings between different models. Object state storage requirements must be specified and mapped to database schema. Executables must be configured for the business objects they will support and then allocated to the network nodes where they will execute. These are all specifications that must be developed as input to the business object implementations. The BOF allows these to be assigned to appropriate specialists so that the business solution developer can concentrate on the business problem. The specific forms of these specifications will depend on the BOF implementation.

3.3.7 Multilingual use

This requirement applies to the specification of common business objects which are not part of this submission.

3.4 Compliance with RFP Technical Criteria

The following comments address specifications in Section 5.8 of the RFP.

3.4.1 Change and event notification

Ad hoc change notification is defined as a special service provided by every business object. Notification of change in one or more aspects of a particular business object can be requested from that business object without prior programming. The request applies only to the specific object and notices will be issued only to the specified consumer or consumers. The implementation of views assures that change notices do not make available information that the consumer would not otherwise be able to access.

3.4.2 Active views

Active views are provided through adaptors and change notification. Control of restricted access is provided separately by views associated with the business object. When an adaptor is activated, it may request notification of change in aspects of interest in the primary object. These notices may be used to maintain a cache or update other related local instance variables as the primary object changes.

3.4.3 Transparent persistence

Persistence is fully transparent at the business process developer level. Separate specifications are entered to define the association of business objects to databases and the mapping of objects to tables for relational databases. Objects are activated when needed, the database is updated implicitly by transaction commit, and the object is deactivated when no longer in use.

3.4.4 Search mechanism

The CORBA query service, with extensions is used for generalized search, independent of whether the objects are transient or persistent, and independent of the type of database used or persistence management implementations. Development of a more sophisticated pattern matching mechanism is deferred until the basic BOF capabilities and interfaces are solidified.

3.4.5 Backout

Backout is supported through the transaction service. Using nested transactions, partial back-out can be performed. Backout independent of the transaction service carries the risk that the backout will not yield a consistent state due to associated changes that were caused by an application but not recognized by the application developer.

3.4.6 Concurrency and Serialization

Concurrency control and transaction serialization are fully supported with minimal visibility to the business developer. The business developer must begin and terminate a transaction. If the object is persistent, the database(s) will be updated when the transaction is committed. Concurrency control is built into every business object.

3.4.7 Nested transactions

Nested transactions will be supported where a transaction service is available that supports nested transactions.

3.4.8 Referential integrity and garbage collection

The referential integrity of relationships between business objects is insured by the encapsulation of relationship management.

Business objects must be explicitly deleted if they are to become unknown to the enterprise. Explicit deletion will cause all relationships to be resolved and the database to be updated at the time of transaction commit. In the event that the business object is being monitored by a display or other dependent process, any such dependent facility is expected to employ the change notification facility to obtain notices of relevant events on the object and all such consumers will receive a notice of deletion of the object.

Business objects may also become inactive and be removed from computer memory with their state preserved in a database. There remains the possibility that an object reference to the inactive object may exist somewhere in the distributed environment. If a message is sent using that object reference, the exception will be intercepted, the object will be re-activated, and the message will be forwarded to the re-activated object.

3.4.9 Encapsulated attributes and relationships

Attributes and relationships are fully encapsulated.

3.4.10 Constraints, rules and policies

Business policies, rules and constraints that define the integrity of the object model are implemented as **validate** methods on objects. The **validate** method is invoked prior to commit of an update transaction. The **validate** method may be implemented in various ways depending upon the BOF implementation.

Development of mechanisms for constraint-based and rule-based reasoning is deferred until the basic BOF capabilities and interfaces are solidified. The change notification and rollback mechanisms are the basic enablers for such facilities.

3.4.11 Relationship management

Relationships are managed by encapsulated logic so that the integrity of complementary references is assured. Access to relationships is provided through accessor methods defined according to a consistent protocol. The submission proposes an IDL and DDL extension for specification of relationships so that the consistent interface can be generated rather than requiring each method signature to be programmed explicitly.

3.4.12 External name management

The Naming Service provides context-based translation of external names to object references. The Naming Service is incorporated into related BOF mechanisms.

3.4.13 Exception/fault resolution

The standard CORBA exception protocol is used with a “user defined” exception that provides for reporting of multiple exceptions in a consistent form for BOF exceptions. This allows multiple errors to be reported to a user for corrective action and the form reduces the amount of business programming required to handle exceptions.

Nested transactions will also provide for limiting the scope of failure. Subordinate transactions can be backed out and restarted without terminating the parent transaction. This capability is anticipated in the design but will require additional mechanisms for coordination of restart with user operations.

3.4.14 Configuration management

This requirement has several aspects related to customizing and installing business objects and applications.

3.4.14.1 Object upgrade installation

With appropriate consideration of upgrade compatibility, a business object can be moved at run time or activated into an executable with a new implementation. The new implementation might be the same or a different BOF. The primary constraint is the implications of changes in the DDL which might add instance variables for which there is no current value or the space is not provided in persistent storage.

3.4.14.2 Configuration of executables

Configuration of executables is a function of the BOF implementation. The specification allows for an executable to include one or many business object classes, and for object classes to be incorporated in different configurations of executables, even different BOFs.

3.4.14.3 Adaptation of application components

Adaptors provide for accommodating differences between applications incorporating adaptors and the shared objects they represent. In most cases changes to shared objects should be upward

compatible by addition of methods and instance variables rather than redefinition. As long as methods and instance variables are only added, existing applications will not be affected. These upgrades can be installed as described above.

3.4.14.4 Local extensions to shared objects

Adaptors also provide for local extensions to shared objects. The adaptor incorporates the shared object state and functionality and may add local state and functionality without affecting the shared object.

3.4.14.5 Implementation and location transparency

Business objects are implementation and location transparent. As noted above, a business object may be moved to a different server, a different BOF and a different language as long as long as the persistent storage is compatible or is independently converted.

3.4.14.6 Dynamic workload balancing

The workload manager facility monitors system activity and determines appropriate selection of execution environments for the activation or move of objects to balance the workload. The mechanisms by which appropriate allocation is determined is BOF implementation specific.

3.4.15 Composite object bounds

The bounds of composite objects are determined by the traversal of relationships for certain recursive operations such as copy and move. These bounds may be different for different purposes and may differ for specialized sub-classes. The specification considers these traversal decisions to be private to the business object implementation. The default methods are expected to not perform traversal, but they provide a recursive protocol by which specialized methods can traverse relationships to propagate the operations while addressing the possibility that the structure may have repeated components or circularities.

3.4.16 External resource representation

The representation of external resources is beyond the scope of this submission but might be addressed by proposals for common business objects.

3.4.17 User attributes and preferences

The representation of external resources is beyond the scope of this submission but might be addressed by proposals for common business objects.

3.4.18 Textual representation

Textual representation of object structures is not addressed by this submission. Given an appropriate language specification, the textual representation could easily be implemented as a limited extension to the business object interface.

3.4.19 Executable object expressions

Development of a language for executable object expressions is deferred until the basic BOF specifications are established.

3.4.20 Loose binding

Assuming that the basic business concepts are compatible, independently developed applications can be dynamically added to a shared objects environment by mapping their adaptors to the interfaces of the appropriate shared objects. This mapping might be done through programming and compilation or it might be done interactively with an appropriate tool. The implementation will depend on the BOF implementation.

In general, there is not magic. The business models must be compatible even though the terminology might differ. The semantics of the operations must be compatible even though the method signatures may differ. To assure that the plug-and-play is appropriate and the results are predictable, the business developer must still examine the models and semantics of the respective implementations and perform explicit mapping.

3.4.21 Instance specialization

Instance specialization is not supported by this submission although it is not incompatible with this submission. The challenge is to provide the protocols and services by which, not only the objects themselves, but the IDL interfaces, the related application components and the persistent storage can be coordinated to also reflect the ad hoc specialization.

Within the scope of this specification, “property objects” may be attached to an object through a relationship. These property objects could incorporate logical extensions to the business object. This, however, would not provide an extended interface (additional methods).

3.4.22 Reflection

Some reflection is implicit in this submission since certain operations require access to meta-data of the business objects. Full reflection is not addressed by this submission because it reflection requires implementation in a reflective language. However, if a BOF is implemented in a reflective language, then business objects implemented in other BOFs could be moved into the reflective environment for reflection on an ad hoc basis.

3.4.23 External interfaces

This submission is not inconsistent with the development of a consistent protocol for interaction of business objects with user interface systems or desktop applications. The ability to interface to business objects through Microsoft COM or other protocols using adaptors has been discussed. The invocation of **validate** methods on business objects assures that changes committed by a transaction comply with defined constraints, rules and policies for those objects.

3.5 Compliance with End User Requirements

The following comments address the requirements specified in Appendix B of the RFP.

3.5.0 Completeness

Each of the defined interfaces is described in Part 2 of this proposal, specifically Sections 2.3 and 2.4. Each of the operations are discussed with reference to key parameters. These descriptions along with the context presented in Sections 2.1 and 2.2 should provide a complete understanding of the intent of the specifications.

3.5.1 Primary requirements

3.5.1.1 Interoperability

In the context of a BOF, interoperability must be considered with respect to three dimensions: (1) interoperability with business objects and associated services provided by another BOF from a different vendor on a peer-to-peer basis (e.g., interoperability of enterprise objects), (2) interoperability with user interfaces or other client components, (3) interoperability between layers of abstraction (as for application objects linked to enterprise objects), and (4) interoperability of different BOFs with different persistence management facilities. Most of these issues are addressed in Sections 2.3.1, 2.3.4 and 2.3.5 along with discussions of the interfaces in Sections 2.4 and 2.5. The contextual framework for the approach is supported by discussions in Part 1, Section 1.3. See also Sections 3.2.1 and 3.4.14.

3.5.1.2 Portability

Portability of business objects can occur at several levels: (1) as business object specifications the conceptual models should be supported by different BOFs and a standard model specification might be used for input of the conceptual model to different BOFs, (2) business objects or working sets of business objects may be plugged into a distributed objects environment and adapted to the local interfaces, (3) instances of business objects may be dynamically moved between different BOFs that implement the same classes (not necessarily in the same language) assuming the interfaces and the data storage specifications are consistent.

These issues are discussed in Section 2.3.4 and in the design of the interfaces in Sections 2.3 and 2.4. See also Section 3.4.14 and 3.2.4.

Implementations of business objects will not be source-code portable since different BOFs will be implemented in different programming languages and may utilize different techniques to optimize different aspects of the design of business objects or provide certain types of functionality. The BOF may not be portable since ORBs do not support application portability and different BOFs may be implemented for particular computing environments. While the BOF provides transparency for the business developer, the BOF may exploit particular environments or component services to provide better performance and achieve a competitive advantage.

3.5.1.3 Substitutability

Processes interfacing to business objects should not be aware of differences between the BOFs that support those business objects. The best demonstration of this will be the ability to dynamically move a business object from one BOF to another (where the same classes are implemented by both) without processes using that business object being aware of the change. These issues are discussed in Sections 2.2.2, 2.3.4 and 2.3.5. Substitutability is achieved through implementation of the interfaces in Sections 2.4 and 2.5.

3.5.1.4 Life cycle

The life cycle interface is supported except for the generic object manager interface which does not provide for specification of initial values. Broader functionality is provided by the Workload Manager which provides access to life cycle managers (i.e., factories) based on workload distribution. See sections 2.4.5, 2.4.12 and 2.4.10. See also 3.1.11.

3.5.2 Administrative requirements

3.5.2.1 Installation/de-installation

The Workload Manager provides interfaces for managing the activation and deactivation of address spaces and life cycle managers (factories). See Section 2.4.10.

3.5.2.2 Upgrade

Through controls provided by the Workload Manager, and new address space can be activated, and activity in an old address space can be dynamically migrated to the new, upgraded address space without system shut-down. See Section 2.2.2, 2.3.4.5 and 2.5.

3.5.2.3 Performance management

Performance management is supported by the Workload Manager which supports monitoring activity and the redistribution of workload. See Section 2.4.10.

3.5.3 Additional requirements

3.5.3.1 Testing and problem determination/resolution

Testing and problem determination interfaces Generalized user interfaces will be supported by adaptors and the change notification facility for examination and monitoring of the state and activity of specific objects of interest as well as traversing relationships to examine related objects. See section 2.3.4.

Exception codes Exception codes are defined in the IDL. See Appendix A.

Object state interface The state of all business objects will be accessible through their interfaces. Security provisions will need to be established in each situation to provide access with appropriate controls.

Invocation history interface This is beyond the scope of this proposal. Interfaces for a complement of testing tools should be defined to support development using BOFs and related facilities. This should be the subject of a future RFP.

3.5.3.2 Version compatibility

Specialized business object interfaces should be substitutable for the business objects they are based upon as long as the implementation of the specialized objects preserves the semantics of the original objects. This will depend upon the business object developer's implementation. It will be possible to install new implementations of upward compatible business objects dynamically by redistribution of work as described above (Section 3.5.2.2). See also Sections 2.2.2 and 2.3.5.

3.6 Conformance with Existing Industry Standards

The RFP calls for a statement of compliance with ISO/IEC 10746 Reference Model of Open Distributed Processing (RM/ODP). This specification is based on existing OMG specifications which are compliant with the RM/ODP model. In addition, these specifications support a computing abstraction which will minimize the effort required to implement a computational model (as defined by RM/ODP) in a distributed objects environment. It also improves the separation of the implementation of the computational model from the implementations of the engineering and technical models. This is accomplished by encapsulating many of the facilities and services associated with distributed objects computations and integrating additional facilities to improve the flexibility and scalability of business systems using this architecture.

Conformance to the RM/ODP model may be further evaluated by considering how the proposal supports the RM/ODP distribution transparencies. In the following paragraphs the transparencies are considered in the context of a BOF implementation that is possible within the specifications defined in this proposal.

1. Access transparency. The Persistence Management facility provides for transparency of access to data storage facilities that is independent of the location and type of database as well as the representation of the data elements.
2. Failure transparency. Business objects operate in a transaction context so that a failure will always restore the business objects to a consistent state. Nested transactions can limit the scope of failures and recovery so that sub-transactions can restart from their beginning consistent state.
3. Location transparency. The BOF provides complete location transparency for business objects. Object references are obtained through the Naming Service using appropriate external identifiers and name contexts. The objects interoperate independent of location and implementation technology.
4. Migration transparency. Business objects may migrate at run time from one environment to another and from one BOF implementation to another as long as the class has been implemented in a consistent manner in the source and destination environments (i.e., the state variables are consistent, the interface specification is the same and the method implementations are consistent). The BOF implementation and the programming language need not be the same.

5. Relocation transparency. Relocation transparency is comprehended by this specification through the use of the Naming Service as the primary point of reference for the current location of any business object. Full relocation transparency depends upon Object Request Broker support to allow the BOF to intercept object reference exceptions and redirect them to the new location of the business object.
6. Replication transparency. While replication is not explicitly addressed by this proposal, it could be implemented by a BOF that is compliant with this specification using the Change Notification facility to maintain consistent state.
7. Persistence transparency. Deactivation and reactivation of objects is completely concealed by the BOF. The business system developer obtains an object reference from the Naming Service and interoperates with that object without any explicit involvement with persistence management. If the object is not active, the BOF will activate it. When the object becomes inactive, the BOF will deactivate it. The identification and schema of the particular database will also be concealed from the business system developer, and business objects of the same class may be transparently stored in relational databases, object-oriented databases or legacy databases. The storage, retrieval and mapping facilities are encapsulated in the Persistence Management facility.
8. Transaction transparency. Except for declaring the beginning and termination of a transaction, the operation of transactions is completely transparent to the business system developer. Every business object incorporates concurrency control and is capable of rollback if the transaction cannot be completed. Deadlocks are also detected and resolved transparently.

Appendix A: Consolidated IDL Specifications

This appendix contains the IDL modules that support this specification as follows:

- Common
- Exceptions
- Business Object
- Life Cycle Manager
- Change Notification
- Naming Service
- Transaction Service
- Persistence Management
- Workload Management

Common

```
#ifndef bof_common_idl
#define bof_common_idl
/*
** This IDL file contains some definitions which are shared by
** many other IDL files.
*/

typedef string          Istring;

struct BofNameValue{
    string    name;
    any      value;
};

typedef sequence<BofNameValue> BofNameValues;

typedef BofNameValues BofObjectIdentifier;
typedef BofNameValues BofBusinessObjectState;
typedef BofNameValues BofInitializerList;
typedef BofNameValues BofCriteria;

typedef Istring          BofName;

#endif
```

Exceptions

```
#ifndef bof_exception_idl
#define bof_exception_idl

/*
```

```

** Bof exception is designed to possibly pass more
** than one reason of failure. This is useful because
** in a lot of situations, a business objects may have
** multiple errors and it is efficient to carry on to
** the end and report all the erros rather than
** report them one-by-one.
*/

enum BofExceptionCategory { EnvironmentException, NormalException,
                             DataEntryException, RuleViolationException,
                             IntegrityException, ServiceError };

struct BofError{
    BofExceptionCategory    exception_category;
    string                   exception_source;
    long                    exception_code; // specific to the source
    string                   exception_reason;
};

typedef sequence <BofError> BofErrors;

exception BofException{
    BofErrors errors;
};
#endif

```

Business Object

```

// IDL for business object

#include "BofCommon.idl"
#include "BofException.idl"
#include "BofLifeCycle.idl"
#include "BofTransaction.idl"
#include "BofNotification.idl"

/*
** In general, access to an object's attributes may be
** denied if the object is already involved in a different
** transaction and the transaction service does not support
** pessimistic control OR if an attempt is made to modify object
** an object outside of a transaction. In this case a CORBA
** system exception is raised.
*/

module Bof{

    interface BofBusinessObject : BofLifeCycle::BofLifeCycleObject ,
                                   BofTransactions::BofTransactionalResource,
                                   BofNotification::BofNotificationSupplier,
                                   BofNotification::BofNotificationConsumer{

        /*
        ** Business object attributes.
        */

        readonly attribute BofObjectIdentifier the_identifier;
        readonly attribute string                the_type;

        /*
        ** Interface to bind the object to naming service and unbind.
        ** The exceptions raised will most likely come from the

```

```

    ** naming service.
    */

    void    bind(in BofName qualified_name) raises(BofException);
};

typedef sequence<BofBusinessObject> BusinessObjects;

/*
** An adaptor is a business object which delegates.
*/

interface BofAdaptor: BofBusinessObject {
    readonly attribute BofBusinessObject primary_object;
};

/*
** The view request interface will be used by business objects
** that support views--usually the enterprise objects.
*/

interface BofViewRequest
    BofView get_view (in string view_context)
        raises (BofException);
};

/*
** The view interface is the abstract interface for views.
** In addition to these methods, specific views will include
** only the method the user is allowed to access.
*/

interface BofView {
    readonly attribute string view_context;
    BofNameValues notify (in BofNotification :: BofAspects aspects,
        in BofConsumer consumer,
        in string consumer_parm,
        in string level)
        raises (BofException);
};

/*
** This is the generic form of an iterator.
*/

interface BofIterator{
    boolean    next_value(out BofNameValue value);
    boolean    next_n_values(in unsigned long how_many,out
        BofNameValues values);
    boolean    get_value_at(in unsigned long at,out BofNameValue
        value);

    unsigned long    count();

    void        reset();
    void        destroy();
};
};

```

Life Cycle Manager

```

#ifndef bof_lifecycle_idl
#define bof_lifecycle_idl

```

```

#include "CosTransactions.idl"
#include "BofCommon.idl"
#include "BofException.idl"

// exception codes for life cycle operations

module BofLifeCycle{

    // typedefs

    typedef BofNameValues BofObjectList;

    // exception codes

    const long NotRemovable                = 1;
    const long NoFactory                    = 2;
    const long NotCopyable                  = 3;
    const long InvalidCriteria              = 4;
    const long CannotMeetCriteria          = 5;
    const long NotMovable                   = 6;

    // forward declarations

    interface BofLifeCycleObject;
    interface BofLifeCycleManager;
    interface BofFactoryFinderManager;
    interface FactoryFinder;

    /*
    ** The object_type which is provided to the find_factory operation
    ** is the type of the object.
    */

    interface BofFactoryFinder{
        BofLifeCycleManager find_factory(in string factory_key);
    };

    /*
    ** the criteria provided to the create operation is used
    ** later on by the FactoryFinder to locate the appropriate
    ** factory. It is in fact passed on to the work load management
    ** service.
    */
    interface BofFactoryFinderFactory{
        FactoryFinder create(in BofCriteria the_criteria);
    };

    interface BofLifeCycleObject{
        void copy(in BofFactoryFinder there,
                 inout BofObjectList the_copied_list)
            raises(BofException);
        void move(in BofFactoryFinder there,
                 inout BofObjectList the_moved_list)
            raises(BofException);
        void remove() raises(BofException);
    };

    interface BofLifeCycleManager :
        CosTransactions::TransactionalObject{
        BofLifeCycleObject create(in string persistent_context,
                                   in BofInitializerList init_values)
            raises(BofException);
    };
}

```

```

        BofLifeCycleObject activate(in BofName qualified_name,
                                   in string persistent_context)
                                   raises(BofException);
        BofName            get_default_naming_context(
                                   in BofName qualified_name);
    };
};
#endif

```

Change Notification

```

#ifndef bof_notification_idl
#define bof_notification_idl

#include "BofCommon.idl"
#include "BofException.idl"

module BofNotification{

    // typedefs

    typedef sequence<string> BofAspects;

    enum BofNotificationLevel { Immediate, Persistent, Guaranteed };

    // exceptions

    // forward declarations

    interface BofNotificationConsumer;

    // interfaces

    interface BofNotificationSupplier{
        BofBusinessObjectState notify(in BofAspects aspects,
                                     BofNotificationConsumer consumer,
                                     consumer_parm,
                                     BofNotificationLevel level)
                                     raises(BofException);
        void stop_notice(in BofNotificationConsumer consumer,
                        consumer_parm)
                        raises(BofException);
    };

    interface BofNotificationConsumer{
        void notice(in BofNotificationSupplier supplier,
                  in octet consumer_parm,
                  in BofBusinessObjectState state);
    };
};

#endif

```

Naming Service

```

#ifndef bof_nameservice_idl
#define bof_nameservice_idl

#include "BofCommon.idl"
#include "BofException.idl"

module BofNaming{

    // exception codes

    const long NotFound                = 1;
    const long CannotProceed          = 2;
    const long InvalidName            = 3;
    const long AlreadyBound           = 4;
    const long NotEmpty                = 5;

    interface BofNameService{

        // bind a new anme to the object
        void bind(in BofName qualified_name,in Object object,
                 in string object_type,in string persistent_context,
                 in Object manager)
            raises(BofException);

        // resolve a name to object reference
        Object resolve(in BofName qualified_name) raises(BofException);

        // resolve a moved reference
        Object resolve_ref(in Object old_object) raise(BofException);

        // unbind a name
        void unbind(in BofName qualified_name) raises(BofException);

        // rebind all the data associated with previous binding
        void rebind(in BofName qualified_name,in Object object,
                  in string persistent_context,in Object manager)
            raises(BofException);

        // this rebinds recursively for all other names of the previous
        // object
        void move(in Object old_object,in Object new_object,in Object
                 new_manager)
            raises(BofException);

        // remove all names bou nd to the object
        void remove(in Object object) raises(BofException);
    };
};

#endif

```

Transaction Service

```

#ifndef bof_transactionalresource_idl
#define bof_transactionalresource_idl

#include "CosTransactions.idl"
#include "BofException.idl"

```

```

module BofTransactions{

    // exception codes

    interface BofCoordinator : CosTransactions::Coordinator{
        boolean will_deadlock_transaction(in CosTransactions::Coordinator
            pending_tc)
            raises(BofException);
        void    waiting_for(in CosTransactions::Coordinator tc)
            raises(BofException);
    };

    interface BofTransactionalResource :
        CosTransactions::Resource,
        CosTransactions::TransactionalObject{
        void    validate() raises(BofException);
    };

    /*
    ** This interface serves to distinguish DBMS resources so that
    ** the transaction service can stage the commit protocol.
    */
    interface BofStorageResource : CosTransactions::Resource{
    };
};
#endif

```

Persistence Management

```

#ifndef bof_persistence_idl
#define bof_persistence_idl

#include "BofCommon.idl"
#include "BofException.idl"
#include "BofTransaction.idl"

module BofPersistence{

    //forward declarations
    interface BofDBSManager;
    interface BofDBS;

    interface BofDBSManager {
        BofDBS    connect(in string persistent_context,
            in BofTransactions::BofCoordinator tc)
            raises(BofException);
        oneway void disconnect(in BofDBS dbs);
        void register_DBMS(in string datastore_type, in BofDBS dbs);
        oneway void un_register_DBMS(in BofDBS dbs);
    };

    interface BofDBS : BofTransactions::BofStorageResource
        CosQuery::QueryEvaluator {
        readonly attribute string datastore_type;

        void    initialize ( in string stringified_pid);
        void    create(inout BofObjectIdentifier id,
            in BofBusinessObjectState data)
            raises(BofException);
        void update(in BofObjectIdentifier id,
            in BofBusinessObjectState data)
            raises(BofException);
        void restore ( in BofObjectIdentifier id,

```

```

        out BofBusinessObjectState data)
        raises(BofException);
    void delete ( in BofObjectIdentifier id)
        raises(BofException);
    };
}
#endif

```

Query Service

```

#ifndef bof_query_idl
#define bof_query_idl

#include "BofBusinessObject.idl"
#include "CosQuery.idl"

module BofQuery{

    /*
    ** This is an iterator which is able to convert values into object
    ** reference
    ** if needed and return them. In case the name value list already
    ** contains
    ** object reference, it is functionally same as BofIterator.
    */

    interface BofObjectIterator : Bof::BofIterator{
        boolean next_object(out Bof::BofBusinessObject object);
        boolean next_n_objects(in unsigned long how_many,out
            Bof::BusinessObjects objects);
        boolean next_object_at(in unsigned long at,out
            Bof::BofBusinessObject object);
    };

    /*
    ** This is an iterator, which is also an evaluator and hence can be
    ** used
    ** to evaluate further queries on the collection that it represents.
    */

    interface BofQueryableIterator : BofObjectIterator,
        CosQuery::QueryEvaluator{
    };
};

#endif

```

Workload Management

```

#ifndef bof_workload_idl
#define bof_workload_idl

#include "BofCommon.idl"
#include "BofException.idl"
#include "BofLifeCycle.idl"

module BofWorkloadManagement{

    /*
    ** This struct describes the stats for the individual operations for
    ** the
    */

```

```

** types of business objects in a process.
*/

struct BofBusinessOperationStats{
    string          operation_name;
    unsigned long   hits;           // total number of access
                                to this operation
    double          total_time;    // total time spent in
                                this operation
};

typedef sequence <BofBusinessOperationStats> BofOpDataList;

struct BofBusinessObjectStats{
    string          managed_type_name;
    string          manager_name;
    long           active_load;
    unsigned long   hits;           // total number of access to
                                instances of this type
    double          totalTime;     // total time spent in
                                those accesses
    BofOpDataList  op_list;
};

typedef sequence <BofBusinessObjectStats> BofObjectDataList;

typedef sequence <string> BofManagedTypes;

/*
** This structure desc ribes a component which.
*/
struct BofComponentData{
    string          component_name;
    string          executable_name;
    BofManagedTypes  managed_types;
};

typedef sequence<BofComponentData>          BofComponents;

typedef BofBusinessObjectStats
        BofManagedTypeWorkload;

typedef sequence<BofManagedTypeWorkload> BofManagedTypeWorkloadList;

struct BofComponentInstanceWorkload{
    string          host_name;
    string          component_name;
    long           instance_number;
    boolean        active;
    BofManagedTypeWorkloadList  workload;
};

typedef sequence<BofComponentInstanceWorkload> BofWorkloadList;

interface BofWorkloadListIterator{
    boolean        next_one(out BofComponentInstanceWorkload b)
                    raises(BofException);
    boolean next_n(in long batchSize,out BofWorkloadList workload)
                    raises(BofException);
    void          destroy();
    void          reset() raises(BofException);
};

/*

```

```

** This interface is a specialization of factory finder which uses
** BofWorkloadManagement service to find the suitable factory
** based on the known work load of the factories in the desired
    location.
*/
interface BofWorkloadBasedManagerFinder :
    BofLifeCycle::BofFactoryFinder{
};

interface BofWorkloadBasedManagerFinderFactory :

    BofLifeCycle::BofFactoryFinderFactory{
};

/*
** This is the main interface for the work load management service.
*/
interface BofWorkloadManager{

    BofLifeCycle::BofLifeCycleManager find_manager(

        in string object_type,

        in BofCriteria the_criteria)

        raises(BofException);

    void  get_host_load_list(in unsigned long batchSize,
        out BofWorkloadList workload,
        out BofWorkloadListIterator iterator)
        raises(BofException);

    void  new_component(in string component_name,
        in string executable_name,
        in BofManagedTypes managed_types)
        raises(BofException);

    void  remove_component(in string component_name)
        raises(BofException);

    BofComponents get_components_list() raises(BofException);

    BofComponentInstanceWorkload new_instance(in string
        component_name,
        in string host_name) raises(BofException);

    void  remove_instance(in string component_name,
        in long instance_number)
        raises(BofException);

    void  activate_instance(in string component_name,
        in long instance_number)
        raises(BofException);

    void  deactivate_instance(in string component_name,
        in long instance_number)
        raises(BofException);
};

/*
** This interface is used by the WL agents to log statistical data
** about the work loads of the business objects with the work load
** management service.

```

```

    */
    interface WorkloadLogger{
        oneway void LogObjectData(in string instance_name,
                                   in BofObjectDataList object_data_list);
    };
};
#endif

```

Relationships

The following IDL is generic for relationships. <name> is replaced with the name of the specific relationship.

```

// Relationship with many members

void <name> (in unsigned long batchSize,
            out sequence members,
            out BusinessObjectsIterator iterator)
    raises (BofException);
void add_<name> (in BusinessObject member)
    raises (BofException);
void delete_<name> (in BusinessObject member)
    raises (BofException);
boolean is_member_<name> (in BusinessObject member)
    raises (BofException);
void private_add_<name> (in BusinessObject member)
    raises (BofException);
void private_delete_<name> (in BusinessObject member)
    raises (BofException);

```

```

// relationship restricted to one member

void <name> (out BusinessObject member);
    raises (BofException);
void set_<name> (in BusinessObject member)
    raises (BofException);
void delete_<name> (in BusinessObject member)
    raises (BofException);
boolean is_member_<name> (in BusinessObject member)
    raises (BofException);
void private_set_<name> (in BusinessObject member)
    raises (BofException);
void private_delete_<name> (in BusinessObject member)
    raises (BofException);

```


Appendix B: Example of Relationship IDL

This appendix illustrates the use of relationship declarations, the equivalent IDL and the resulting code for a C++ binding.

Consider an object model shown in Figure B.1, which contains **Claim** and **Vehicle**. There may be several vehicles involved in a claim however only one vehicle can be involved in a claim at a time.

Figure B.2 shows the IDL for the **Claim** interface. The claim declares a relationship with many vehicles where the vehicles refer back with **claim** relationship.

Figure B.3 shows the IDL for the **Vehicle** interface. The vehicle declares a relationship with one claim where the claim refers back with **vehicles** relationship.

Figure B.4 shows the implied IDL that would be an expansion of the relationship declaration in Figure B.2.

Figure B.5 shows the implied IDL that would be an expansion of the relationship declaration in Figure B.3.

Figure B.6 shows the C++ code that would be generated from the IDL in Figure B.2.

Figure B.7 shows the C++ code that would be generated from the IDL in Figure B.3.

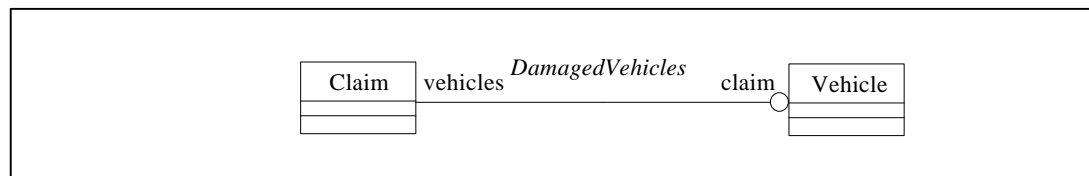


Figure B.1, Relationship between Claim and Vehicles

```
interface Vehicle;
typedef sequence<Vehicle> Vehicles;

interface Claim : Bof::BofBusinessObject{
    relationship many Vehicle vehicles inverse one claim;
};
```

Figure B.2, Claim.idl

```
interface Claim;

interface Vehicle : Bof::BofBusinessObject{
    relationship one Claim claim inverse many vehicles;
```

```
};
```

Figure B.3, Vehicle.idl

```
interface Vehicle;
typedef sequence<Vehicle> Vehicles;

interface Claim : Bof::BofBusinessObject{

    void        vehicles (in unsigned long how_many,
                          out Vehicles members ,
                          out BofQuery::BofQueryableIterator result)
                          raises (BofException);

    void        add_vehicles(in Vehicle member)
                          raises (BofException);
    void        delete_vehicles(in Vehicle member)
                          raises (BofException);
    boolean     is_member_vehicles(in Vehicle member)
                          raises (BofException);
    void        private_add_vehicles(in Vehicle member)
                          raises (BofException);
    void        private_delete_vehicles(in Vehicle member)
                          raises (BofException);
};
```

Figure B.4, Representative IDL for Claim

```
interface Claim;

interface Vehicle : Bof::BofBusinessObject{

    void        claim (out Claim member)
                    raises (BofException);
    void        set_claim(in Claim member)
                    raises (BofException);
    void        delete_claim(in Claim member)
                    raises (BofException);
    boolean     is_member_claim(in Claim member)
                    raises (BofException);
    void        private_set_claim(in Claim member)
                    raises (BofException);
    void        private_delete_claim(in Claim member)
                    raises (BofException);
};
```

Figure B.5, Representative IDL for Vehicle

```
class Claim_i: public virtual Bof::BofBusinessObject_i{
public:
    virtual void vehicles (CORBA::ULong how_many,
                          Vehicles*& members,
                          BofQuery::BofQueryableIterator_ptr& result,
                          CORBA::Environment &);

    virtual void add_vehicles (Vehicle_ptr member,
                              CORBA::Environment &);
};
```

```
virtual void delete_vehicles (Vehicle_ptr member,  
                             CORBA::Environment &);  
  
virtual CORBA::Boolean is_member_vehicles (Vehicle_ptr member,  
                                           CORBA::Environment &);  
  
virtual void private_add_vehicles (Vehicle_ptr member,  
                                  CORBA::Environment &);  
  
virtual void private_delete_vehicles (Vehicle_ptr member,  
                                     CORBA::Environment &);  
};
```

Figure B.6, C++ Bindings for Relationship Shown in Figure B.2

```
class Vehicle_i: public virtual Bof_BofBusinessObject_i{  
public:  
    virtual void claim (Claim_ptr& member,  
                       CORBA_Environment &);  
    virtual void set_claim (Claim_ptr member,  
                           CORBA_Environment &IT_env);  
  
    virtual void delete_claim (Claim_ptr member,  
                              CORBA_Environment &);  
  
    virtual CORBA_Boolean is_member_claim (Claim_ptr member,  
                                           CORBA_Environment &);  
  
    virtual void private_set_claim (Claim_ptr member,  
                                   CORBA_Environment &);  
  
    virtual void private_delete_claim (Claim_ptr member,  
                                      CORBA_Environment &);  
};
```

Figure B.7, C++ Bindings for Relationship Shown in Figure B.3

Glossary

Adaptor: A specialized business object adapts the representation of another business object which exists in another level of abstraction. For example, where an Employee adaptor in an application represents an Employee object that is shared at the enterprise level. An adaptor delegates message processing to the implementation of methods on the primary business object. It may translate the method signatures from the form received to a form acceptable to the primary object in order to translate the semantics of different domains, or to adapt an independently developed application. An adaptor may also have local methods and instance variables which extend its capabilities beyond that of the primary object to meet local requirements.

Base business object An interface specification and, typically, an abstract class for implementation, from which all business objects inherit. It provides the common functionality of business objects. Specializations will contain attributes, relationships and operations that define the unique characteristics of specific types of business objects.

Business systems developer This terminology is used within this specification to refer to system developers who implement business solutions using business objects. This is similar to the common terminology of “application developer” except that developers in the future will be less likely to develop applications instead of business objects and components. In addition, within the Enterprise Computing Model, some developers may work on applications, but others may work on the enterprise layer and still others may work on agents, all of which are working on solutions to business problems.

Change notification A mechanism by which any business object can be asked to provide notices of change in one or more of its attributes and relationships. Each request specifies where the notices are to be directed. The business object has no prior specific programming to provide the requested notices. Change notification is used to keep displays up to date and to initiate processes depending on relevant changes in business objects.

Commit The action of finalizing and making persistent the results of a process--in the BOF context, the action taken to finalize a transaction. It is also the name of the second phase of a two-phase commit where the first phase is “prepare.” The two phase commit is used to coordinate the completion of actions of multiple, distributed persistent resources (e.g., databases). It is essential that either all resources commit or none commit to assure the integrity of the system.

Data Definition Language (DDL) A language specification defined by OMG which is a subset of IDL for specification of object data. In the BOF specification it is adopted as an appropriate form of expression of the data to be exchanged between the BOF and the associated Persistence Management service for managing the persistence of business objects.

External identifier Any code or string of characters provided by the user or an external source that can be used to identify an object. The context of the external identifier is required to determine its meaning and its association to a particular object, as where a room number only has meaning within the context of a particular building. The BOF Naming Service may bind the same object to many external identifiers if this is useful to the user and the application. However, there will be one external identifier which is considered the primary identifier and will be used to

Glossary

identify the object in persistent storage as well. Such identifiers may be a single string or value, or they may be a concatenation of keys.

Factory finder: An object which possesses selection criteria for locating a life cycle manager (i.e., factory) object for any particular class of object. Factory finders are used, for example, in copy and move operations to find life cycle managers needed to create new instances of the copied or moved objects in a new location. The factory finder is passed recursively if the operation is recursive, locating different life cycle managers that meet the criteria to create instances of the various objects being copied or moved.

Iterator: An object which retrieves successive members or groups of members from a collection. An iterator may operate on a real or virtual collection, as where the members being returned are not determined until they are requested. Iterators are used to access the collections implicit in one-to-many relationships, to provide query results and for other situations where access is provided to a controlled and/or shared collection.

Legacy system A legacy system is typically a system being replaced. In general terms it is a system that does not comply with current architectural specifications for the computing environment or is in some way incompatible with systems currently under development. The primary concern with legacy systems is integration and/or migration. Integration involves efforts to build a transparent interface for cooperation. Migration involves replacing components of the legacy system until it is no longer needed.

Life cycle. A set of operations on objects related to their creation and destruction. These are **create, remove, copy and move**. For the BOF, this is extended to include **activate** and **deactivate**, i.e., retrieval from persistent storage and removal from main storage.

Life cycle manager A factory object with additional functionality. A business object is instantiated in an address space by the life cycle manager for its specific type which exists in the target address space. Consequently, each existing factory object in a distributed environment represents an address space where an object of the type it creates can be created. In addition to creation, a life cycle manager controls the activation and deactivation of persistent objects. When an existing object is to be retrieved from a database, the request will be handled by a life cycle manager in the address space where the object is to be instantiated. The life cycle manager will then determine when the object can be deactivated.

Naming Service The Naming Service associates (binds) names, usually identifiers that are meaningful to humans, to object references which are the identifiers used to send messages to objects. In the BOF specification, the Naming Service is the focal point for locating objects and provides mechanisms for resolving object references that have become invalid because the target object has been moved or deactivated.

Object state data The data that represents the current state of an object, usually all of its instance variables. In this proposal, this is a sequence of name-value pairs that is passed between the object and persistence management to instantiate or store a persistent object. A similar mechanism is used to pass the current values of attributes and relationships for change notices. The **move** and **copy** methods also use a similar form to create new instances of objects being copied or moved.

Persistence Management A service defined to support heterogeneous BOFs by interfacing to persistent storage for business objects. Persistence Management will store or retrieve the object state data for a business object, performing translation, if necessary, between the object's assignment of values to attributes and relationships, and the elements of the persistent storage

structure. Persistence Management is incorporated in transaction management to support a two-phase commit and rollback capability, and it manages available (licensed) connections to each database manager.

Prepare. The action of preparing to make persistent the results of a process--in the BOF context, the action taken to prepare for commit of a transaction. It is also the name of the first phase of a two-phase commit where the second phase is "commit." The two phase commit is used to coordinate the completion of actions of multiple, distributed persistent resources (e.g., databases). It is essential that either all resources commit or none commit to assure the integrity of the system.

Primary identifier Sometimes just called the object identifier, the primary identifier is a sequence of characters or concatenation of sequences which can be used to uniquely identify an object. The primary identifier will be used to identify the object in persistent storage. It is also used as the name for the primary binding for the object in the Naming Service.

Query. A process by which information meeting certain criteria is retrieved. In CORBA environments, a query is expected to search collections of objects which may be real or virtual so that objects may be within in-memory collections or they may be in a database and the same query protocol will apply. In the BOF specification, a query evaluator is obtained for a target collection (or database) and it is asked to evaluate a specified query. The query evaluator returns an iterator, which represents the resulting collection, which again may be real or virtual. The iterator will return individual or groups of members from the collection on request. The iterator is also a query evaluator and can be asked to select members from its collection and provide another result iterator.

Relationship A mutual reference between business objects. BOF relationships are binary as opposed to n-ary, meaning that there are two types of objects with two different roles in any relationship. N-ary relationships can always be implemented as binary relationships and the implementations are much simpler and easier to understand. The cardinality of BOF relationships may be one-to-one or one-to-many, but not many-to-many. Many-to-many relationships can always be represented as two one-to-many relationships, and, in most cases, there will be a need at some time for the added associative entity. BOF relationships are always complementary, meaning that if A has a relationship with B, then B has a relationship with A, and the BOF will assure that these relationships remain consistent.

Transaction context Information that is implicitly passed along with messages to objects which provides information about the current transaction. It is used by the BOF to determine when transactions are in conflict and to determine where a business object should register when it becomes a participant in a transaction.

Transaction serialization The process by which the execution of multiple, competing transactions is coordinated such that the resultant impact on the system is the same as if the transactions were executed in sequence, one after another. This assures that each transaction takes the system from one consistent state to another without interference from another transaction.

Validate method The BOF specification defines a validate method to be supported on all business objects. Execution of the validate method will cause the state of the object, and possibly related objects, to be examined for consistency with the business model, business policies and other integrity criteria. This can be used to report discrepancies to a user to assist in their correction. It is always used to evaluate the consistency of affected objects when a transaction commits. This assures that the result that is committed to the database meets appropriate criteria.

Glossary

View. An object which presents a subset interface of a primary object. A primary object may have many views which provide access to different, not necessarily mutually exclusive, subsets of the primary object's methods. Views have no persistent state. The primary purpose of views is for security.

Workload Manager The Workload Manager provides a service for managing the distribution of work by allocation of objects to address spaces. The workload manager will find an appropriate life cycle manager for instantiation of a business object based on the current distribution of workload. It also provides support for a user interface for administrative monitoring and control of distributed resource.