Party Management Facility Specification

Version 1.0
February 2001

Copyright 1999, Concept Five Technologies, Inc.
Copyright 1999, Cyborg Systems, Inc.
Copyright 1999, Electronic Data Systems (EDS)
Copyright 1999, Hitachi, Ltd.

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013 OMG®and Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB, CORBA, CORBAfacilities, CORBAservices, and COSS are trademarks of the Object Management Group, Inc. X/Open is a trademark of X/Open Company Ltd.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form at *http://www.omg.org/library/issuerpt.htm*.

Contents

Preface **1**

1. Domain Model and Design Objectives **1-1**

1.1 Service Overview 1-2
1.2 Common Object Model 1-3
1.3 Composition Model 1-4
1.4 Definition of Terms and Assumptions 1-4
1.5 Role Aware Composition Model 1-5
1.6 Party and Contact Information 1-7
1.7 Party Relationships as First Class Objects 1-8
1.8 High Level Comparison with CosRelationships 1-8
1.9 Manager and Object Factory Model 1-9
1.10 Locating Existing Party Information 1-12

2. Party Management Facility Interfaces **2-1**

2.1 Overview 2-2
2.2 CosFinance Module Declaration 2-2
2.3 General Type Information 2-4
2.4 Manager 2-6
2.5 Date and Time Sensitive Objects 2-7
2.6 Common Object 2-8
  2.6.1 CommonObject (Inherited Interfaces) 2-8
  2.6.2 CommonObject (Local Attributes and Methods) 2-11
2.7 Common Container 2-12
  2.7.1 CommonContainer (Inherited Interfaces) 2-13

Contents

2.7.2 CommonContainer (Local behavior) 2-13

2.8 Template Manager 2-14

2.9 Locator 2-15

2.10 Iterator Support 2-17

2.11 PMF Module Declaration 2-22

2.12 General Type Information 2-23

2.13 Role 2-24

2.14 Node 2-26

2.15 Party 2-27

2.16 PartyRole 2-27

2.17 Party Relationship 2-29

2.18 Person 2-29

2.19 Organization 2-29

2.20 Node Manager 2-29

2.21 Party Manager 2-30

2.22 Role Manager 2-30

2.23 PartyRoleManager 2-31

2.24 Relationship Manager 2-31

2.25 PartyRelationship Manager 2-33

2.26 Group Manager 2-33

2.27 ContactInformationFactory 2-33

2.28 Summary 2-33

3. Compliance, Conformance, and Known Issues 3-1

3.1 Compliance with Existing Specifications 3-1

3.1.1 Transaction Service (OTS) 3-1

3.1.2 Relationship Service 3-1

3.1.3 Security Service 3-2

3.1.4 Persistent Object Service (POS) 3-2

3.1.5 Query Service 3-2

3.1.6 Name Service 3-2

3.1.7 Trader Service 3-2

3.1.8 Event Service 3-2

3.1.9 Externalization Service 3-2

3.2 Levels of Conformance 3-3

3.3 Known Issues 3-3

3.3.1 Notification Support 3-3

Contents

4. Security and Party Management 4-1

4.1 Security Issues 4-1

Appendix A - Complete OMG IDL A-1

Appendix B - Collaboration Diagrams B-1

Appendix C - Wrapping Cos Relationships C-1

Appendix D - References D-1

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

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

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

Party Management Overview

A key requirement for financial service organizations is to effectively manage the parties, people, and organizations that relate to their business. They must have efficient and consistent access to party related information including relationship and contact information. The vast majority of organizations have many different computer systems operating their daily business processes that all need access to “name and address” information. Unfortunately, this information is generally embedded within each system in a proprietary manner resulting in an environment that is very expensive and difficult to maintain. As a result, redundant, inconsistent information is often proliferated throughout the organization. The Party Management Facility (PMF) defines a standard set of interfaces that will enable a consistent integration strategy for consumers whether they are software vendors, other systems, or end users.

This document provides a high level view of where the Party Management Facility (PMF) fits into the OMA and into the Financial Industry vertical domain (“CORBAFinancials”). It then illustrates how the proposed interface definitions satisfy the mandatory and optional requirements of the RFP while providing an extensible foundation for commercial products to adhere to and for end users to customize for their specific needs. The interfaces have been supplemented with textual descriptions and scenario diagrams to further illustrate their use in practice. The remainder of the document outlines how the PMF complies with existing OMG standards and closes with topics of discussion as outlined in the RFP.

The Object Management Architecture (OMA) is the basic framework for organizing OMG efforts. The OMA Reference Model is illustrated below.

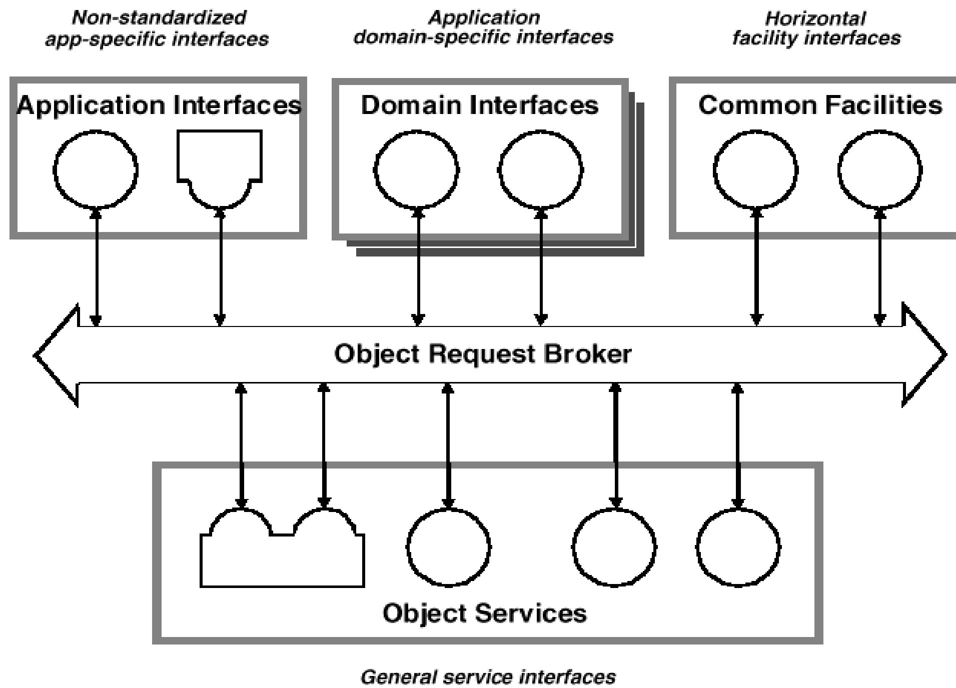


Figure 1. OMG Object Management Architecture (OMA) Reference Model

Within the context of the OMA, the Party Management Facility is clearly a Domain Interface.

The following figure, although not an official OMG diagram, depicts a view of the OMA in more detail with a specific focus on the domain interfaces.

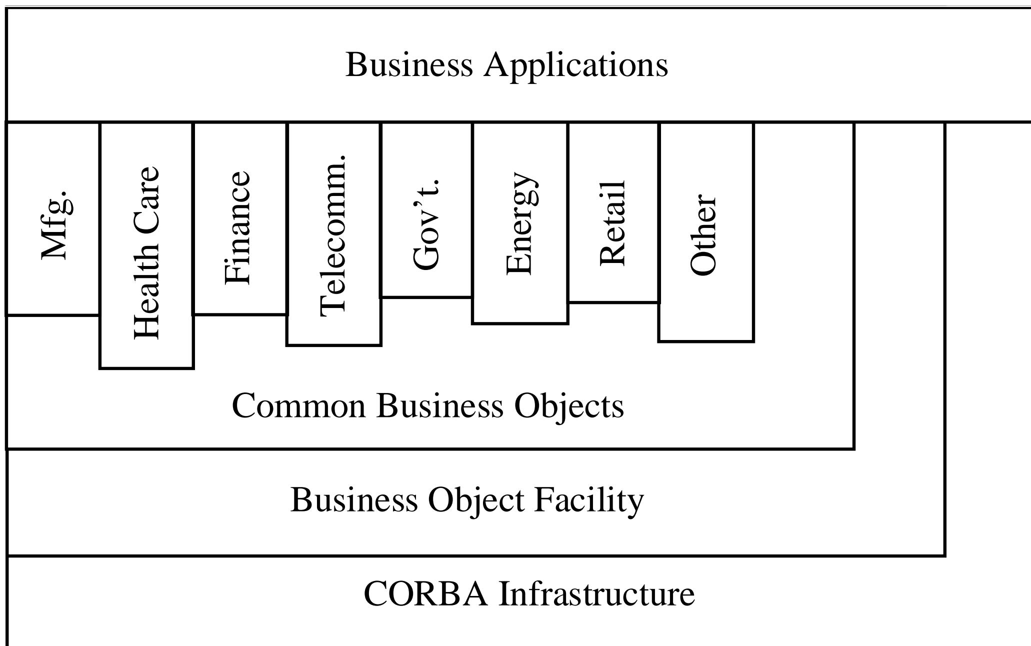


Figure 2. Business Object Domain Task Force (BODTF) view of OMA

The architectural boundaries can be further exploded to depict the focus of the Finance Domain Task Force (FDTF). In the FDTF architectural view the Common Business Objects layer is further specialized with Common Financial Services (objects) and the vertical financial markets can be illustrated as: insurance, banking, brokerage, and securities. Although this response has leveraged work that was put into the Person Identification System (PIDS) created within the healthcare arena, its primary focus is to satisfy the requirements of the FDTF and the finance industry. The Party Management Facility is intended to serve as a Common Financial Service.

"Party" is a general concept that can be used in an endless array of roles depending on the context of its surroundings. The intent of this specification is to provide a core definition of Party that assumes many of the technology characteristics necessary to live in a distributed system. Further, it is intended that vendors will specialize the interface(s) into the specific roles, along with specific attribution and behavior, for their respective lines of business. Figure 3 shows that the bulk of this specification lies in the component category.

Process	Highest level business processes (e.g., Process New Business).
System	Higher level collection of components (e.g., Policy
Component	The Party Management Facility Interface Definitions reside
Object	Specific entity level interfaces are defined here that often provide

Figure 3. Component Category Interfaces

This specification illustrates how the Party Management Facility serves as a focal point for all party related information with the understanding that the PMF is one small component in a much larger picture. The other facilities that have been identified for future standardization include: Product, Agreement, and Investment Management. Many higher level services will be built upon these basic concepts.

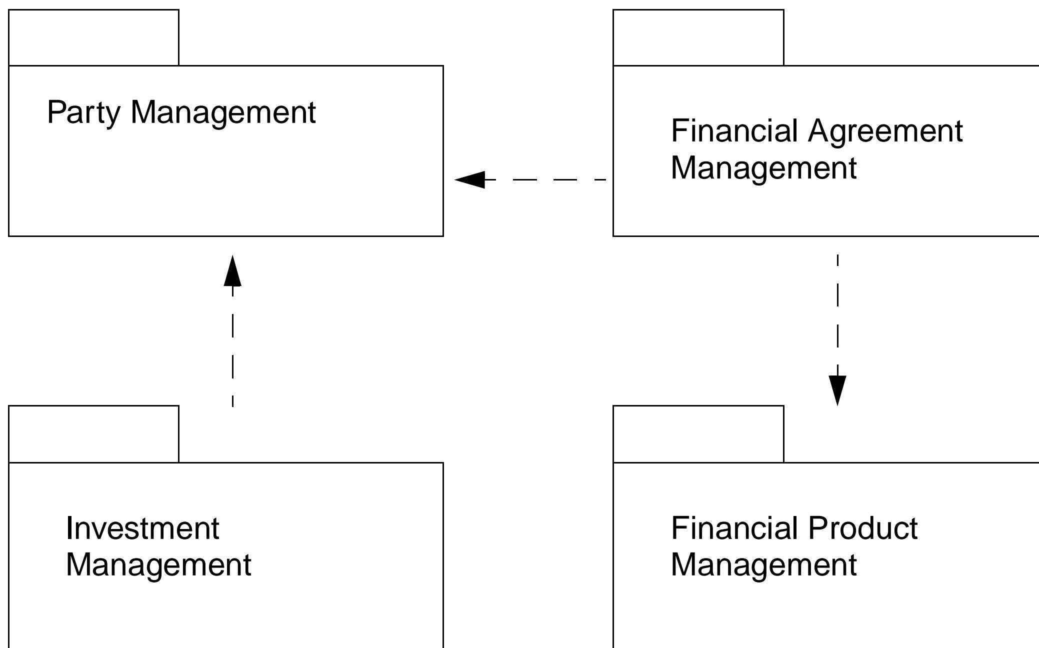


Figure 4. Facilities identified for future standardization

Associated OMG Documents

The CORBA documentation set includes the following:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBA Language Mappings*, a collection of language mapping specifications. See the individual language mapping specifications.
- *CORBAservices: Common Object Services Specification* contains specifications for OMG's Object Services.
- *CORBAfacilities: Common Facilities Specification* is a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Med*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- 2AB, Inc.
- Concept Five Technologies, Inc.
- Cyborg Systems, Inc.
- Data Access Technologies
- Electronic Data Systems (EDS)
- Hitachi, Ltd.
- International Business Machines Corporation
- Open Engineering, Inc.
- System Software Associated, Inc.

Domain Model and Design Objectives 1

Contents

This chapter contains the following topics.

Topic	Page
"Service Overview"	1-2
"Common Object Model"	1-3
"Composition Model"	1-4
"Definition of Terms and Assumptions"	1-4
"Role Aware Composition Model"	1-5
"Party and Contact Information"	1-7
"Party Relationships as First Class Objects"	1-8
"High Level Comparison with CosRelationships"	1-8
"Manager and Object Factory Model"	1-9
"Locating Existing Party Information"	1-12

This specification defines specific interfaces for Party Management that each of the other (future) finance related components will request information from. The goal of this specification and the Party Management Facility (PMF) in general is to define these interfaces at a level where, for example, an insurance company can easily replace their current OMG compliant PMF component with a new one in a seamless manner. That is, since both products conform to the OMG standard, then interoperability at an interface level is assured. And, since OMG IDL promotes a distinct separation between interface and implementation, the remainder of the insurance application remains completely intact, unaware of the new PMF implementation.

The interfaces presented in this specification represent a solution to a well-understood problem, the management of involved party information. This specification does not attempt to address *how* a Party Management Facility is to be implemented (e.g., in terms of access to persistent storage or collection management). Rather, it provides a higher level interface that allows external users (people, application programmers, or other systems) to access and manipulate party related information in a consistent and well-defined manner.

1.1 Service Overview

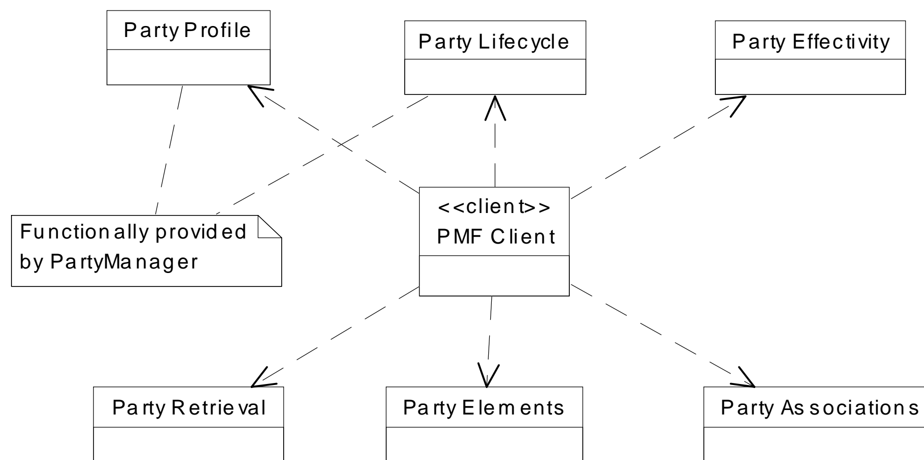


Figure 1-1 Services that comprise the Party Management Facility

Figure 1-1 illustrates the different types of services that comprise the Party Management Facility. The Party Management Facility defines similar types of services as the General Ledger Facility. There are interfaces that support data extraction, party lifecycle management, location services, effective dating, along with services specific to managing and creating parties and their relationships.

This section describes the core aspects of the Party Management Facility by illustrating the interface hierarchy in UML. The models have been broken into small, manageable packages in an effort to succinctly communicate the design, intent, and justification of each logical entity. This section does not explain individual methods or attributes but rather focuses on the overall design. Each IDL interface is described in detail in the next section.

The facility must allow for dynamic attribution and provide the ability to internalize and externalize party information in a well-defined format. This specification has chosen to capture this behavior, and other general services, by making extensive use of pre-existing Common Object Services. Further, it consolidates this behavior into a single interface from which most other Party Management interfaces derive from. Figure 1-2 illustrates this abstract interface entitled **CommonObject**.

1.2 Common Object Model

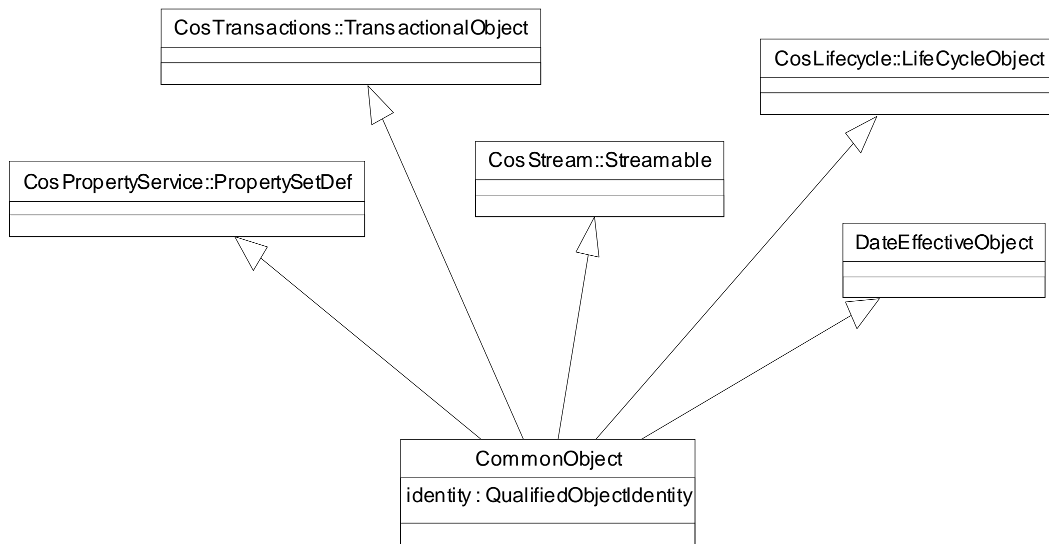


Figure 1-2 Common Object Interface Model

This abstract **CommonObject** interface represents the core behavior that most Party Management interfaces will inherit. To provide for dynamic, although constrained, attribution **CommonObject** derives from the **CosPropertyService**'s **PropertySetDef** interface. In order for the **CommonObject** to potentially be involved in distributed transactions it derives from **CosTransactions::TransactionalObject**. Note that this interface may not be necessary in the future, as transactional semantics become more quality of service oriented. By inheriting from Streamable, each object inherits the ability to internalize and externalize its state into user supplied data streams (e.g., into pre-existing EDI standard formats). **CommonObject** also derives from **LifeCycleObject** so clients may remove a particular instance from the facility. Finally, **CommonObject** derives from **DateEffectiveObject** so that every aspect of the facility has the ability to be date and time stamped. This ability is important to provide point-in-time representation. Each of the specific methods and their applicability to party management is outlined in the IDL listing contained in Section 2.6, "Common Object," on page 2-8.

1.3 Composition Model

The first extension this specification makes to **CommonObject** is fundamental to the manner in which object aggregation is managed throughout the facility. The PMF has been positioned as an extensible service that effectively manages not only party information but also party relationships to other entities. Parties may be related to other parties, financial agreements, assets, or in general to most anything. To satisfy the

requirement of managing these associations, this specification has incorporated the use of the Composite design pattern as documented by Gamma, Helm, Johnson, and Vlissides [1]. The following diagram illustrates this very important extension to the base **CommonObject** into a **CommonContainer**.

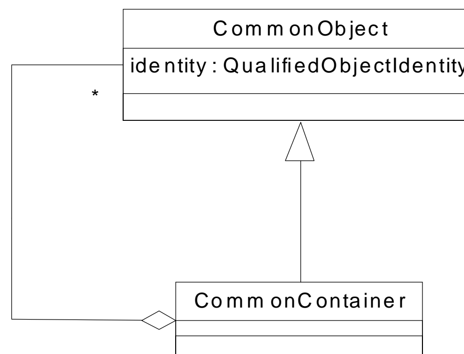


Figure 1-3 Composite Design Pattern

This design pattern is very powerful in terms of creating and managing aggregations of objects that can be referenced through a simple and consistent interface. Generically these interfaces and their unique relationship to each other define the ability to group objects together. In reference to the Composite design pattern, the **CommonObject** is a *component* and the **CommonContainer** is a *composite*. Quite simply, this construct provides the foundation for grouping objects together into compositions that may in fact be recursive. The next few paragraphs describe how these core capabilities have been extended to understand roles, relationships, and specific domain types that will be introduced as mandatory extensions as well as a few illustrative examples.

1.4 Definition of Terms and Assumptions

The basic behavior that is made possible with **CommonObject** and **CommonContainer** lacks role information. For example, assume that Person was a kind of **CommonObject** and a user wished to relate an instance of person as a ‘husband’ with another person who happened to be his ‘wife.’ The general interfaces to create an association between the person and his/her family exist within these base interfaces; however, if role information is necessary, then these interfaces must be extended to become role aware. This concept and the design approach becomes clear by first providing a few basic assumptions.

1. Basic person and organizational information is maintained and implemented as specific derivations of Party (they are *Person* and *Organization* respectively). That is, a person can exist independently in a system without being associated to an insurance policy, annuity, or other customer relationship information. By definition then, a *Person* or *Organization* has a lifecycle of its own, has state that can be externalized, has date aware behavior, and can participate in distributed transactions.

2. *Party* is defined as a base interface over Person and Organization. The term Party implies that the Person or Organization can, and often does, play many Roles relative to the business for which it is being maintained. As a result, this base interface, through Node, has methods that allow for the traversal from Person and Organization to those Roles. The Roles themselves use a form of aggregation to tie a Person or Organization to the related entity they are associated with. A specific Role, PartyRole, has been introduced for those Roles that only Parties can play. For example, an Insured entity would inherit from PartyRole allowing a Person or Organization to participate in a relationship with an Insurance Agreement.

3. This specification takes the position that an instance of a *relationship* between two or more entities does not always dictate the existence of a first class relationship (link) object. To expand upon the previous husband and wife example, depending upon the nature of the system the user may not be interested in specific *marriage* behavior or state. That is, the simple association between two person objects with limited role information attached to each may be sufficient for some systems without creating a first class *marriage* object that represents the association between the two parties. On the other hand, a system may be keenly interested in the *marriage* itself as a first class entity to provide specific behavior, such as divorce(). In particular, first class relationships are indicated by derivations of PartyRelationship that can maintain role constraints per type of relationship. For example, *marriage* could be a specialization of PartyRelationship that constrains the roles to being ‘husband’ and ‘wife.’ This specification recognizes the usefulness of both scenarios and supports both.

1.5 Role Aware Composition Model

Having an understanding of these fundamental concepts described above, Figure 1-4 introduces two new extensions to the model: **Role** and **Node**.

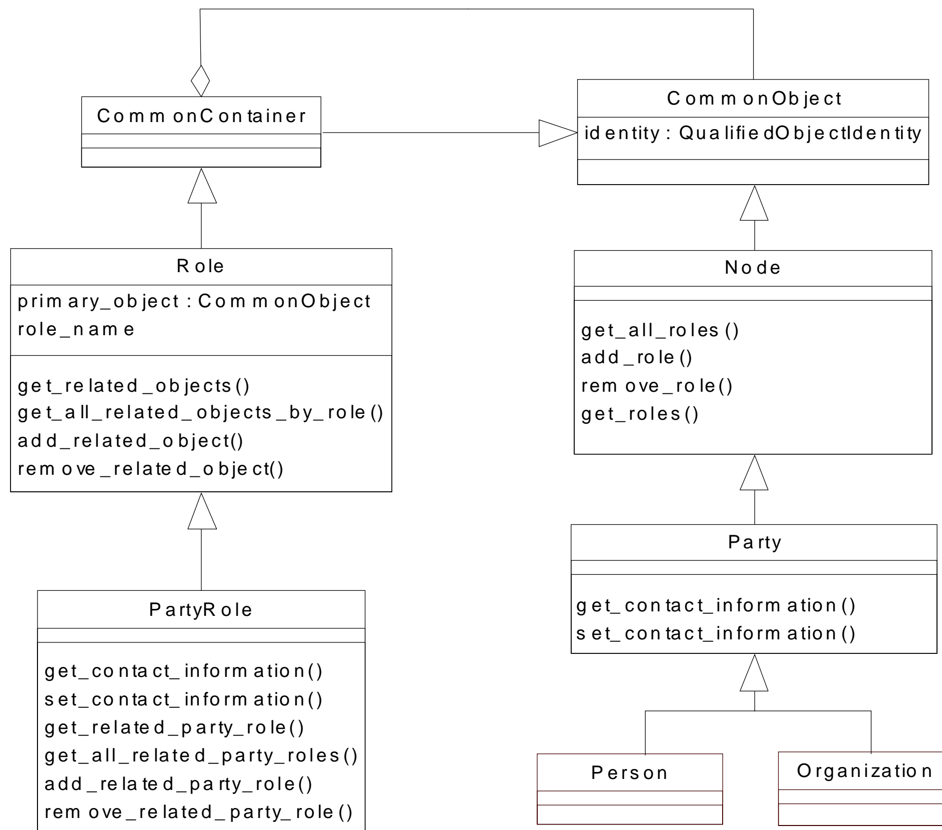


Figure 1-4 Role and Node Interfaces

To reflect back upon the three fundamental concepts provided above, this model shows specifically how Person, Party, and Organization are introduced as unique extensions of **Node** and therefore **CommonObject**. The **Node** and **Role** interfaces are introduced here to provide a richer level of support for aggregations and relationship traversal.

The **Node** interface specializes the core behavior of **CommonObject** providing the user with the ability to obtain all of the roles that a particular object is playing. Parties can also play many roles. For example, if a person were the father in one relationship, possibly with his son, and the insured in another relationship with an insurer, then the **get_all_roles** method would return references to a 'Father' and an 'Insured', where both are likely derivations of **PartyRole**.

The **Role** interface has three purposes:

1. Adds role knowledge to otherwise generic object aggregations.

2. Supports and constrains the unique aggregation between the primary object (i.e., Person) and the role (e.g., Insured) object that adds specific state and behavior to the Person in the context of a relationship.

3. Acts as a base interface for **PartyRole** and subsequently all roles Parties might play, (e.g., Producer or Agent).

PartyRole derives from **Role**. **PartyRole** provides a mechanism from which the fundamental notion of a **Party** can be augmented, through aggregation, with role specific behavior. The specification supplies a user-friendly interface on **PartyRole** such that its related parties or other related objects can be accessed directly without having to direct messages to the more abstract base interface of **Role**. Figure 1-5 shows a more detailed view of **Party** along with its **ContactInformation**.

1.6 Party and Contact Information

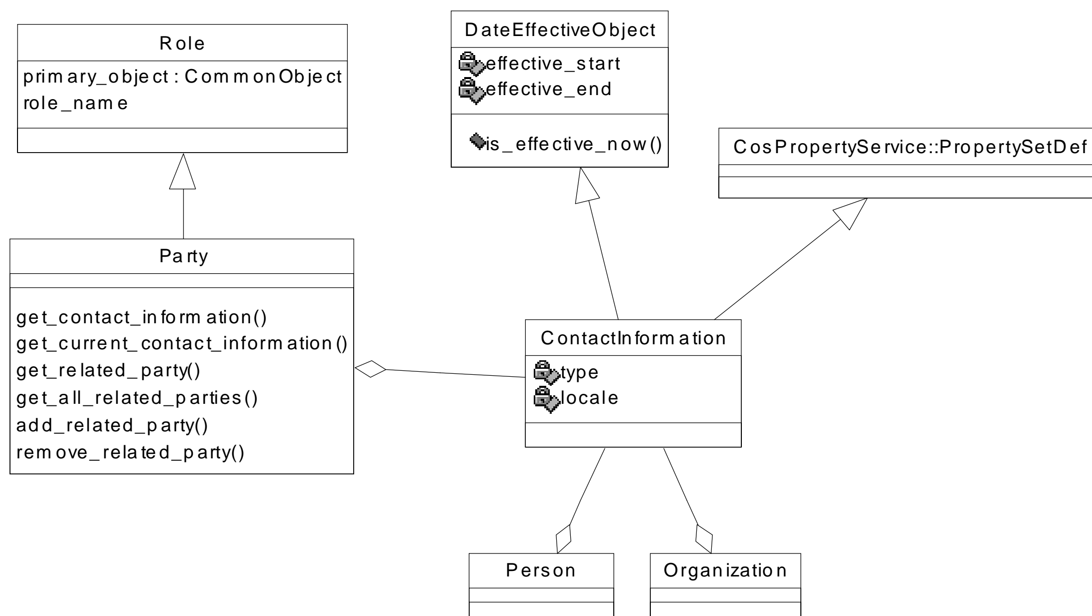


Figure 1-5 Party and Contact Information

A **Party** can have a business address, a home address, a fax number, and other types of contact information. While avoiding the general notion of *location* the specification attempts to encapsulate general forms of people related contact information into a single point of reference. The **ContactInformation** interface is date sensitive enabling the client to request all contact information as-of a specific point-in-time. A **Party** can have any number of different types of contact information (for example, multiple phone numbers representing various ways of contacting the party). **ContactInformation** also inherits from the **PropertySetDef** allowing for dynamic attribution. Dynamic

attribution on **ContactInformation** can help with issues such as Internationalization where the fields may differ by locale. It can also allow for customization (for example, a business address that not only needs street information but also a suite, room, or building number). Note that contact information is available from both **PartyRole** and the **Party** object that it is representing. This provides the **PartyRole** with the option to either delegate a request for contact information to the primary object (Party) or to potentially handle the request itself. Under some circumstances it may be desirable to specialize contact information per relationship.

1.7 Party Relationships as First Class Objects

In an instance where a first class object that represents the relationship itself is required, as in our marriage example above, the specification introduces the **PartyRelationship** interface. In general, the **PartyRelationship** interface provides constrained behavior to a **Role**. That is, it explicitly specifies the roles that may exist in the relationship and does not allow non-supported roles to participate. For example, a user could not add an 'insured' object to a *marriage* relationship.

The interface is illustrated in Figure 1-6. These constraints can be further enforced by the manager interface associated with the relationship type.

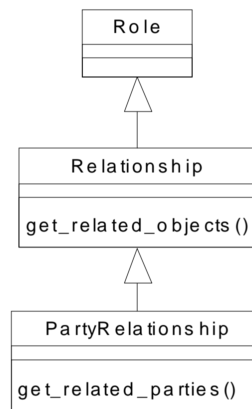


Figure 1-6 PartyRelationship Interface

1.8 High Level Comparison with CosRelationships

Many of these concepts are functionally in synch with the current **CosRelationship** Service specification. There are two (2) fundamental reasons why this specification has chosen not to explicitly extend or otherwise use **CosRelationships**.

1. This specification recognizes that every association between two objects should not result in the creation of a first class relationship (link) object. The overhead of this requirement in **CosRelationships** could prove to be unmanageable. For example, if a very simple system chose to group people into user-defined groups and there wasn't any specific state or behavior introduced as a result of that association, then

it should not be required to create this third object. If this group contained 1000 people the use of **CosRelationships** would result in 1000 people objects, 2000 role objects, and 1000 relationship objects. Each of these has an identity, is most likely persistent, may have to be managed as part of a distributed transaction, and in general consumes unnecessary resources. This characteristic of **CosRelationships** also introduces additional overhead in traversal. For example, in a 1:m relationship if the user wished to obtain individual object references for the many (from the 1) they must first traverse each of the relationship objects (**get_relationships**) to get to the role object on the other side. The role object would then need to be queried (**get_related_object**) to obtain the primary object.

2. The interface exposed to the user of a system explicitly based upon **CosRelationships** is somewhat low-level and complex. For example, to relate two objects together the client must invoke operations on a role factory, create sequences of named roles, and invoke a third set of operations on a relationship factory that results in the creation of a relationship object that has a unique identity and must be managed.

As an alternative, this specification exposes a somewhat higher level interface and positions the use of **CosRelationship**s as an implementation decision that the PMF vendor must make. Appendix B provides a set of scenario diagrams that illustrate how these PMF interfaces could be used to wrap an implementation of **CosRelationships** specifically for Party Management. For more information on the position taken in regard to **CosRelationships** refer to the “Compliance, Conformance, and Known Issues” chapter.

1.9 Manager and Object Factory Model

The following sections define a series of management level interfaces that in general are used to create parties, primary objects, and first class relationship objects along with their respective state and behavior. Figure 1-7 shows the hierarchy for these management level interfaces.

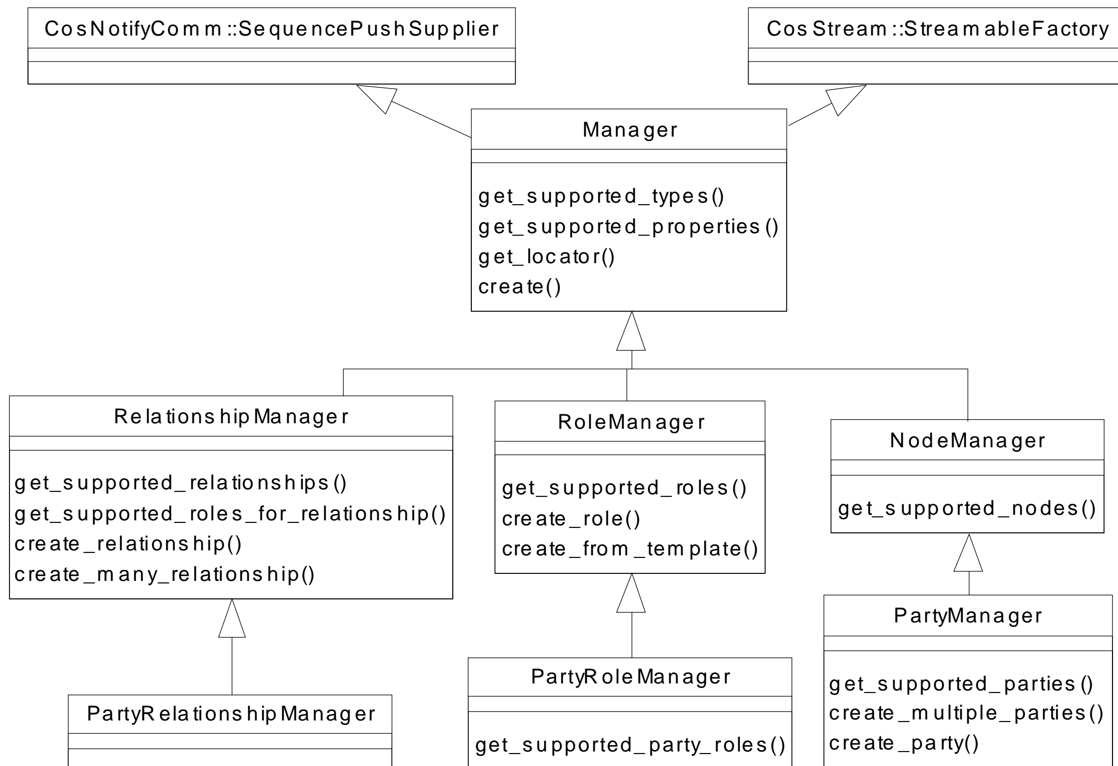


Figure 1-7 Management Level Interfaces Hierarchy

For the most part these interfaces provide the factory behavior and a level of meta information that is often associated with any CORBA based system. The term ‘Manager’ has been used in the specification to indicate a higher level of functionality than generally appears in a traditional *factory* type interface. This additional knowledge is geared toward exposing the kinds (types) of information that the PMF has been configured to support.

The **Manager** interface is responsible to act as a base abstraction for communicating supported type and attribution. The methods **get_supported_types** and **get_supported_properties** offer this level of support respectively. This allows a graphical client to easily display the facility’s options for object creation. It also provides the ability to dynamically generate user interface logic that reflects the properties associated with a specific type. The **Manager** also contains the **create** method to create a **CommonObject**.

Note – The **Manager** derives from the Notification Service's **SequencePushSupplier** interface providing the **Manager** the ability to broadcast type level creation notices. Also, **Manager** derives from **StreamableFactory** so that it can work in conjunction with the **CommonObject**'s inherited **internalize_from_stream** method.

The **PartyRelationshipManager** interface provides the ability to create **PartyRelationship** objects. As discussed previously these relationship objects are not a mandatory aspect of associating objects to one another but do provide a mechanism for additional state and behavior specific to the association of two objects. The **PartyRelationshipManager** also exposes methods to communicate the types of relationships it has the ability to create as well as the valid roles that can participate in each of those relationships.

The **RoleManager** inherits from **Manager** and therefore inherits the **create** and **create_from_template** methods. At the **RoleManager** level, the client will need to narrow the returned **CommonObject** into a **Role** object. **RoleManager** creates **Role** objects by accepting the primary object and the requested role in the form of a string. It has the ability to create specific derivations of **Role**s and can communicate the types of derivations that it can create.

The **PartyRoleManager** inherits from **RoleManager** and therefore inherits the **create** and **create_from_template** methods. At the **PartyRoleManager** level, the client will need to narrow the returned **Role** into a **PartyRole** object. **PartyRoleManager** creates **PartyRole** objects by accepting the primary object (Party) and the requested role in the form of a string. It has the ability to create specific derivations of **PartyRole** and can communicate the types of derivations that it can create.

The **NodeManager** and **PartyManager** interfaces are capable of creating types of objects derived from **Node** and **Party** such as **Person** and **Organization**. These have been segregated, as other aspects of the interface hierarchy, to allow for future expansion beyond Party Management.

This next set of management level interfaces do not derive from **Manager** but simply provide base level factory behavior for their respective types.

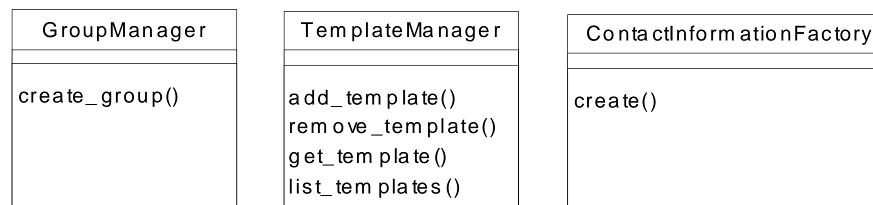


Figure 1-8 Template and Group Management Interfaces

Templates and **Groups** are the final two management interfaces that must be discussed. A **Template** defines a container that contains specific types. For example, in conjunction with the **Role**'s generic ability to group objects the template not only specifies the exact types that will comprise the container but initializes the container with those types. **Template** support is optional.

A **Group** is simply a user-defined, named container of parties.

1.10 Locating Existing Party Information

The final three interfaces presented in the specification reflect the ability to locate parties using a variety of mechanisms. The primary interface is the **Locator**. A reference to the **Locator** can be obtained from the **Manager** interfaces described above or could be directly resolved from a Naming or Trader Service. Figure 1-9 shows the relationship between the **Locator**, **Iterator**, and **Table** interfaces. In general, an **Iterator** is the result of a query issued in one of two ways on the **Locator** (evaluate or query). The optional **Table** interface provides high level access to the data returned by the **Iterator** through its methods that ultimately provide access to a multi-dimensional list of values.

Locator
naming_context trader_components
resolve() evaluate() query()

Iterator
next_object() next_n_objects() next_values() next_n_values() reset() count() object_at() values_at()

Table
number_of_rows number_of_columns column_property_types column_names
get_row() set_row() get_cell() set_cell()

Figure 1-9 Locator, Iterator, and Table Interfaces

Note that the resolve method on the **Locator** provides the ability to specify an as-of-date that enables the **Locator** to return a specific **Party** as it existed on a specific date. Also, if a client has a reference to a **Locator** and still cannot locate the **Party** they are searching for, they have a direct link to the local naming and or trader service through read-only attributes. For additional locator support the Iterator interface provides a mechanism to traverse **Party** objects or data associated with Parties, or both. Using this approach, the PMF vendor has the ability to support the lazy activation of **Party** objects.

Party Management Facility Interfaces 2

Contents

This chapter contains the following topics.

Topic	Page
“Overview”	2-2
“CosFinance Module Declaration”	2-2
“General Type Information”	2-4
“Manager”	2-6
“Date and Time Sensitive Objects”	2-7
“Common Object”	2-8
“Common Container”	2-12
“Template Manager”	2-14
“Locator”	2-15
“Iterator Support”	2-17
“PMF Module Declaration”	2-22
“General Type Information”	2-23
“Role”	2-24
“Node”	2-26
“Party”	2-27
“PartyRole”	2-27
“Party Relationship”	2-29
“Person”	2-29

Topic	Page
“Organization”	2-29
“Node Manager”	2-29
“Party Manager”	2-30
“Role Manager”	2-30
“PartyRoleManager”	2-31
“Relationship Manager”	2-31
“PartyRelationship Manager”	2-33
“Group Manager”	2-33
“ContactInformationFactory”	2-33
“Summary”	2-33

2.1 Overview

The interface hierarchy is scoped within two modules. The **CosFinance** module provides base level interfaces from which the PMF module inherits from or otherwise makes use of. The modules have been segregated in an effort to isolate some of the core behavior that is anticipated to be used in subsequent finance related specifications. Since much of the technology related characteristics such as externalization, lifecycle, and transactional behavior will likely be required for most finance-related initiatives, this specification has taken an extra step to make them easily available and extensible. Collectively, the interfaces can be broken down into the following functional areas: aggregation (composition), attribution, location (query), lifecycle management, and iterator support.

A complete OMG IDL is included in Appendix A. The next portion of this document breaks out each individual interface definition, per module, and provides a description of its use and role.

2.2 CosFinance Module Declaration

```
#ifndef CosFinance_idl
#define CosFinance_idl

#include “CosProperties.idl”
#include “CosLifeCycle.idl”
#include “CosExternalization.idl”
#include “CosTransactions.idl”
#include “CosTime.idl”
#include “CosNotifyComm.idl”
#include “NamingAuthority.idl”
#include “CosNaming.idl”
#include “CosTrader.idl”
```

```
#pragma prefix "omg.org"

module CosFinance
{
    // ...
};

#endif
```

#include "CosProperties.idl"

This file contains all of the type declarations for the **CosProperty** service.

#include "CosLifeCycle.idl"

This file contains all of the type declarations for the **CosLifeCycle** service.

#include "CosExternalization.idl"

This file contains all of the type declarations for the **CosExternalization** service.

#include "CosTransactions.idl"

This file contains all of the type declarations for the **CosTransactions** service. Note, only the **TransactionalObject** interface is used indicating that if an OTS is present, then the base financial interfaces are transactional in nature.

#include "CosTime.idl"

This file contains all of the type declarations for the **CosTime** service.

#include "CosNotifyComm.idl"

This file contains all of the type declarations for the Supplier capability described in the Notification Service.

#include "NamingAuthority.idl"

Since this specification makes use of the qualified identity defined in the Person Identification Service we include **NamingAuthority.idl** for domain information.

#include "CosNaming.idl"

Since this specification references the **NamingContext** from **CosNaming CosNaming.idl** must be included.

#include "CosTrader.idl"

Since this specification references **TraderComponents** from **CosTrader CosTrader.idl** must be included.

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

In order to prevent name pollution and name clashing of IDL types this module uses the pragma prefix that is the reverse of the OMG's DNS name.

Each of the following type declarations and interfaces exists in the module defined above.

2.3 General Type Information

```
//forward declarations

//management
interface Table;
interface Manager;
interface TemplateManager;
interface Iterator;
interface Locator;

//core
interface CommonObject;
interface CommonContainer;
interface DateEffectiveObject;

//typedef
typedef string                       Type;
typedef sequence<string>             Types;


struct Template
{
    string name;
    Types types;
};

typedef string                           QueryExpression;
typedef CosTime::UTO                     Date;
typedef any                              PropertyValue;
typedef sequence<PropertyValue>          PropertyValues;
typedef sequence<Template>               Templates;
typedef sequence<CommonObject>           CommonObjects;
typedef sequence<CommonContainer>        CommonContainers;


//enumerators
enum ModificationState { Update, Correction };
```

```
struct QualifiedObjectIdentity
{
    NamingAuthority::AuthorityId domain;
    Type type;
    NamingAuthority::LocalName id;
};
```

Type, Types

Type is simply an alias for string that is used by the **Manager** interface to determine what specific type of object to create (e.g., 'IndividualPerson' or 'Employer'). It is also used to publish all the types that this **Manager** can create.

Template

A structure representing the name of the template (pre-defined collection) and the many types it contains.

QueryExpression

QueryExpression is a string that represents how to locate a set of objects or data within the local domain. The vendor must supply the exact syntax of the query. It is suggested that the syntax be a derivation of ODMG OQL or SQL.

Date

Date is an alias for **CosTime::UTO** (Universal Time Object). The vendor may wish to extend this low-level abstraction to a more user-friendly date and time structure that allows locale formatting.

PropertyValue, PropertyValues

PropertyValues is a sequence of type **PropteryValue**, which is a CORBA type any. This typedef extends **CosPropertyService**. It has been introduced as an efficiency mechanism to pass all names at once then all data associated with those names. Instead of always passing both name and value as **CosPropertyService** suggests. See Table interface declaration below.

Templates

A sequence of many templates.

CommonObjects

This typedef is an alias for a sequence of **CommonObjects**.

CommonContainers

This typedef is an alias for a sequence of **CommonContainers**.

ModificationState

Any date and time sensitive object needs to be aware of its update state while accessor methods are being invoked. This provides the vendor to automatically monitor and keep track of effective and expiration dates. It also provides the user of the PMF to not imply a real state changed by switching the flag to 'Correction.' See the **DateEffectiveObject** interface for more detail.

QualifiedObjectIdentity

This structure has been, for the most part, borrowed from the Person Identification Service (PIDS) defined by CORBAmed. PIDS defines a **NamingAuthority** that realizes that an identity is only valid within the context of a domain. An **AuthorityId** is the combination of a **RegistrationAuthority**, such as ISO, DNS, IDL, and a **NamingEntity** that is a string. The **LocalName** is a string that contains the value of the domain dependent identification, such as a social security number. This specification has explicitly added the **Type** attribute to differentiate between entity types that may exist on the backend, such as the roles 'Employee' or 'Claimant.'

2.4 Manager

```
interface Manager : CosStream::StreamableFactory, CosNotifyComm::SequencePushSupplier
{
    exception   TypeNotSupported {};
    exception   DuplicateObject {};
    exception   InvalidInitializationType {};
    exception   InvalidInitializationValue {};

    struct InitCommonObject {
                QualifiedObjectIdentity identity;
                CosPropertyService::Properties data;
    };

    typedef sequence<InitCommonObject> InitCommonObjects;

    Types       get_supported_types();

    CommonObject create( in InitCommonObject data )
                        raises (
                                TypeNotSupported,
                                InvalidInitializationType,
                                InvalidInitializationValue,
                                DuplicateObject);

    void        get_supported_properties(
                        in Type type,
                        out CosPropertyService::PropertyDefs property_defs)
                raises (
                                TypeNotSupported );

    Locator         get_locator();
};
```

InitCommonObject struct

A structure to maintain combinations of types and properties. This structure is using as a parameter to the create methods allowing initial values to be streamed into a new object. Note, **QualifiedObjectIdentity** is described above.

create ()

This method performs the creation of a new **CommonObject** or **CommonContainer**, or derivation thereof, that represents the type requested (within the **QualifiedObjectIdentity** structure). If the facility is asked to create a type of object that it does not support, it throws the TypeNotSupported exception. Recognize that identity information is not always available at object creation time. As a result, the value of the identity passed in may be null. Quite often, legacy systems have an internal mechanism to generate uniqueness per type. In this situation, the complete identity would not be available until after completion.

2.5 Date and Time Sensitive Objects

```
interface DateEffectiveObject
{
    attribute ModificationState update_state;

    attribute Date effective_start;
    attribute Date effective_end;

    boolean is_effective_now();
};
```

This interface is used as a base for **CommonObjects** that often require effectivity constraints. For example, roles that parties play in the context of relationships come and go and the system must be able to portray a valid picture of the state at a specific point in time. Also, Agreements that Parties participate in also tend to be relatively dynamic and must be date and time stamped to effectively manage history and audit requirements.

update_state

Any object that inherits this interface will need to be aware of its **update_state** while accessory methods are being invoked. This provides the vendor to automatically monitor and keep track of effective and expiration dates. Note the user of the PMF can set the value to ‘Correction’ to indicate the difference between a valid change in state vs. a data entry error.

effective_start()

The date that the state of the derived type (agreement or role) was committed.

effective_end()

The date that the state of the derived type was modified, in a sense marking the beginning of a new start-to-end duration.

is_effective_now()

Simply returns true or false if the current system date falls between **effective_start()** and **effective_end()** inclusive.

2.6 Common Object

```
interface CommonObject : CosLifecycle::LifecycleObject,
          CosStream::Streamable,
          CosPropertyService::PropertySetDef,
          CosTransactions::TransactionalObject,
          DateEffectiveObject
{
     exception ContainerNotFound {};
     exception CannotRemove {};

     readonly attribute ObjectIdentity identity;

     boolean is_dependent_object();

     boolean is_date_sensitive();

     CommonContainers get_containers();

     void add_container(in CommonContainer container);

     void remove_from_container(in CommonContainer container)
                    raises (ContainerNotFound,
                              CannotRemove);
};
```

2.6.1 CommonObject (Inherited Interfaces)

CosPropertyService::PropertySetDef

The ability to obtain, define (potentially) and set state on a specific object implementation is through the **PropertySetDef** interface. Valid attribution is also exposed to the client through the **PropertySetDef** interface defined within **CosPropertyService**. According to the PMF vendor's implementation technique, they may then allow the consumer to further customize the attribution set. The PMF vendor may supply a set of supported attribution templates. For example, if the PMF is being installed at an insurance company, then they may choose to attribute their Person type with the standard ACORD characteristics. The names and types of this attribution set may appear to the consumer as packaged constraints.

CosLifeCycle::LifeCycleObject

CommonObject inherits from **LifeCycleObject** to allow instance level administration on the target object.

CosLifeCycle::LifeCycleObject::copy (optional)

This method allows the data associated with this **CommonObject** to be copied from one address space to another, possibly on a different host. The factory finder parameter to this method should in fact point to a **PartyManagementFacility::Manager** reference. If the vendor chooses not to implement this method they can simply throw the predefined exception NotCopyable.

CosLifeCycle::LifeCycleObject::move (optional)

The PMF vendor may implement this method in the same manner as copy (see above).

CosLifeCycle::LifeCycleObject::remove

This method effectively removes the object from the system, including its persistent state. This may include removing a reference from a Name service and/or other repositories. The details on what actually gets removed are implementation specific.

CosStream::Streamable

The **CommonObject** interface inherits from **CosStream::Streamable** primarily to implement the **externalize_to_stream** method. Using this approach one of many specializations of **StreamIO** could be passed to a **CommonObject** allowing its state to be externalized to a specific format - such as existing EDI standard formats. This interface inherits from **CosObjectIdentity::IdentifiableObject**. It is assumed that the Party Management Facility will *not* implement **IdentifiableObject** - as a simple unsigned long is not enough information to uniquely identify objects across multiple domains. Rather, **CommonObject** will use the **QualifiedObjectIdentity** described above.

CosStream::Streamable::externalize_to_stream()

It is expected that the vendor will supply this generic streaming capability to stream Party data out to external sources and/or legacy environments. For example, the stream object could format the data encapsulated within the **CommonObject** (Person) into the ANSI X12 standard 275 for patient information.

CosStream::Streamable::internalize_from_stream()

This method can be used to stream data into an object, (e.g., from an EDI input stream).

CosPropertyService::PropertySetDef

The **PropertySetDef** interface, a constrained specialization of **PropertySet**, is inherited primarily to provide generic attribution on all **CommonObject**s. This interface provides for dynamic customization. The inherited behavior listed here provides a quick reference to the requirements. For a more complete explanation refer to the **CosPropertyService** specification.

CosPropertyService::PropertySet::define_property()

This method adds or changes an existing property on the **Party** object.

CosPropertyService::PropertySet::define_properties()

This method will add or change all of the properties in the list to the **Party** object.

CosPropertyService::PropertySet::get_number_of_properties()

Returns the total number of properties currently defined on this **Party** object.

CosPropertyService::PropertySet::get_all_property_names()

Returns all the properties, by name, currently associated with this **Party** object.

CosPropertyService::PropertySet::get_property_value()

Gets the value of an attribute from its name.

CosPropertyService::PropertySet::get_properties()

Gets multiple values of attributes from a list of names.

CosPropertyService::PropertySet::get_all_properties()

Returns all of the properties defined (name and value). If more than **how_many** properties are present the remainder are returned in an iterator.

CosPropertyService::PropertySet::delete_property()

Deletes the property from the **PropertySet** if it exists.

CosPropertyService::PropertySet::delete_properties()

Deletes all of the properties listed in the **property_names** paramater.

CosPropertyService::PropertySet::delete_all_properties()

Blindly deletes all properties.

CosPropertyService::PropertySet::is_property_defined()

Returns true if the property name passed in exists in this **PropertySet**.

CosPropertyService::PropertySetDef::get_allowed_property_types ()

Provides a mechanism for the PMF vendor to communicate to a client explicitly which property types are valid for this type of object. For example, the vendor may not limit the attribution by name only by type and may state that only **tk_string** is allowed.

CosPropertyService::PropertySetDef::get_allowed_properties ()

Allows the vendor to communicate exactly which properties are supported. Note, this method returns a sequence of **PropertyDef**'s which contain name, value, and mode.

CosPropertyService::PropertySetDef::define_property_with_mode ()

Allows the client to provide, or customize, the attribution associated with this object. The vendor could choose to disallow this feature by throwing one of the unsupported exceptions.

CosPropertyService::PropertySetDef::define_properties_with_modes()

Allows the client to provide, or customize, the attribution associated with this object. The vendor could choose to disallow this feature by throwing one of the unsupported exceptions.

CosPropertyService::PropertySetDef::get_property_mode()

Returns the mode of the specified property. Note, the valid modes include **read_only**, **normal**, **fixed_normal**, **fixed_readonly**, and **undefined**.

CosPropertyService::PropertySetDef::get_property_modes ()

Returns a list of modes respective to the names passed in.

CosPropertyService::PropertySetDef::set_property_mode()

Sets the mode on a specific property, provided the property name and mode are valid; otherwise, an exception is thrown.

CosPropertyService::PropertySetDef::set_property_modes ()

Sets the modes on a set of properties, provided the property names and modes are valid; otherwise, an exception is thrown.

CosTransactions::TransactionalObject

This interface does not necessarily require any additional behavior from the PMF vendor. It simply implies that the object may be transactional and that the thread's transaction context should be initialized. If an OTS is being used, it is likely the PMF vendor will wish to register a synchronization interface for the **CommonObject**s so they receive the **before_completion** message prior to transaction preparation.

DateEffectiveObject

As defined above, allows **CommonObject**'s to be date and time aware. This provision allows for the point-in-time representation.

2.6.2 CommonObject (Local Attributes and Methods)

identity

The domain that it lives in (as suggested in PIDS) qualifies the unique identity of the **CommonObject**. See **QualifiedObjectIdentity** description above.

is_dependent_object()

In some instances an object may be fully contained (by value) within another object. Often this implies that the containing object controls the lifecycle of the embedded object as well as its identity. For example, individual Diaries (comments) associated with a **Party** may not require a fully scoped, self-sufficient identity. In this case this method would return TRUE and the identity method would return the identity of the containing object.

is_date_sensitive()

Not all **CommonObject**s, or their potential derivations, will require data sensitivity. The client can query whether or not the implementation supports effectivity for this specific type.

get_containers()

This method allows for bi-directional communication between a **CommonObject** and the container that may have contained it. For example, if a client holds a reference to a **CommonObject** that represents a specific Person, then it could invoke this method to determine which **CommonContainers** (typically roles) have referenced it.

add_container()

This method allows for aggregation to be initiated by the contained object. It is also intended to be called implicitly as a result of invoking **add_contained_object** on a container.

2.7 Common Container

```
interface CommonContainer : CommonObject
{
    exception ObjectNotFound {};
    exception IsDuplicate {};
    exception InvalidAggregation {};
    exception MaximumCardinalityExceeded {};

    void add_contained_object(in CommonObject object,
            in Date as_of_date);

    void add_contained_objects(in CommonObjects objects,
            in Date as_of_date);

CommonObject get_contained_object_by_id(in QualifiedObjectIdentity id,
            in Date as_of_date)
            raises (ObjectNotFound);

    void remove_contained_object(in CommonObject object
            in Date effective_date)
            raises (ObjectNotFound);
```

```
boolean has_contained_object(in CommonObject object,I
                in Date as_of_date);

void get_all_contained_objects(in Date as_of_date,
                out CommonObjects sequence);

void add_from_template(in Template template);

Templates list_templates();
};
```

2.7.1 CommonContainer (Inherited Interfaces)

A **CommonContainer** is a specialization of **CommonObject** and represents a composition of many **CommonObject**s or in fact other **CommonContainer**s. This unique relationship offers generic aggregation capabilities.

2.7.2 CommonContainer (Local behavior)

add_contained_object()

Adds another **CommonObject**, or due to inheritance another **CommonContainer**, to its collection as of the date specified.

add_contained_objects()

Adds many **CommonObject**s, or due to inheritance other **CommonContainer**s, to its collection as of the date specified.

get_contained_object_by_id()

Returns the embedded object that matches the characteristics of **QualifiedObjectIdentity**. Otherwise, throws an exception stating that the object represented by the identity is not embedded in this container. For example, assume that "Diary" is a valid type supported by the facility and its identity is comprised of the date, user id of the creator, and a sequence number. The client could retrieve the full object state of the diary entry by supplying this instance level information through this method.

remove_contained_objects()

The client can remove a contained **CommonObject** from the **CommonContainer** (aggregation) by passing it to this method. It is likely the vendor will use the inherited **is_identical()** method to locate the object in the container.

has_contained_objects()

This method will likely use the inherited **is_identical()** method to determine whether or not the passed in **CommonObject** has been contained within this container. For example, a client could query an Employer to ask whether or not a specific Employee worked for them on the date specified.

get_all_contained_objects()

This method returns all contained object as of a specific date regardless of role.

add_from_template()

This method allows other types to be contained by this container as specified in the template. Since templates in general are optional this method is also optional.

list_templates()

This method returns all the templates that were used to construct the object. Since templates in general are optional this method is also optional.

2.8 Template Manager

```
Struct Template
{
    string name;
    Types types;
};

interface TemplateManager
{
    exception TemplateNotFound {};

    void     add_template(in Template template);

    void     remove_template(in Template template);

    Template get_template(in string template_name)
        Raises(TemplateNotFound);

    Templates list_templates();
};
```

[OPTIONAL INTERFACES]

The **TemplateManager** interface and those lifecycle methods that reference **Templates**, (e.g., **create_from_template**) are specified as optional interfaces. That is, they provide an ease of use quality that although desirable is not required to satisfy the basic PMF behavior.

add_template()

Adds a new template to the repository. Note, since **Templates** are simple structures they can be fully created on the client and passed in to this method.

remove_template()

Removes a template from the repository.

get_template()

Obtains a pre-constructed template for the repository for use in object creation.

list_templates()

Provides a list of all defined and available template definitions.

2.9 Locator

```
interface Locator
{
    typedef sequence<string> CriteriaBasis;
    typedef sequence<string> SearchType;

    exception InvalidQuerySyntax {};
    exception NotImplemented {};
    exception SearchTypeNotSupported {};
    exception CriteriaBasisNotSupported {};
    exception NotFound {};
    exception InvalidIdentifier {};
    exception InvalidAsOfDate {};
    exception TypeNotSupported {};

    readonly attribute NamingAuthority::AuthorityId domain_name;
    readonly attribute CosNaming::NamingContext naming_context;
    readonly attribute CosTrading::TraderComponents trader_components;

    CommonObject resolve(
        in QualifiedObjectIdentity identifier,
        in Date as_of_date )
    raises (
        NotFound,
        InvalidIdentifier,
        InvalidAsOfDate );

    Iterator evaluate(
        in QueryExpression query)
    raises (
        InvalidQuerySyntax,
        NotImplemented );
```

```
    Iterator query(
        in Type object_type,
        in string criteria,
        in CriteriaBasis criteria_basis,
        in SearchType type_of_search)
    raises (
        TypeNotSupported,
        NotImplemented,
        SearchTypeNotSupported,
        CriteriaBasisNotSupported );
};
```

This interface provides a variety of mechanisms to find party related information based on a specified search criteria. A reference to the **Locator** can be obtained from the Manager, a Naming or Trader Service, or ultimately another **PartyLocator**. This location capability becomes federated by providing references to other location services such as Naming or Trader.

resolve()

The resolve method returns a reference to a **Party** object as of a specific point-in-time. The method accepts a named identity, as defined in PIDS, as well as a date parameter indicating the effective date they would like to use from a state perspective. For example, if the **Party** had recently changed their last name a request could be made to view the **Party** as it existed last week or last year. This method is a take-off of the resolve method described in **CosNaming** but adds an intuitive aspect of time.

evaluate()

The evaluate method performs a **CosQuery** like evaluation over a domain centric **QueryableCollection**. That is, this method itself when implemented is directed at a specific domain and is not intended to cross architectural boundaries. The query could in fact be issued over a series of CORBA **Party** objects or redirected to a persistent storage device such as an RDBMS.

query()

This method is a simple, most likely highly used, intuitive mechanism to quickly locate a set of Party related objects that match the criteria provided. The **CriteriaBasis** and **SearchType** values are vendor supplied and can be retrieved by calling their respective accessory methods. Examples include:

- **CriteriaBasis** values may include; lastName, firstName, and SSN.
- **SearchType** values may include; soundsLike, spelledLike, and spelledExactlyLike. For example, this method can be used to quickly locate all the **Party** objects whose last name begins with “Swi” by invoking query (“Party,” “Swi,” “lastName,” “spelledLike”). The Iterator that is returned is described below.

2.10 Iterator Support

```
interface Iterator
{
    exception OutOfBounds {};

    boolean next_object(
        out CommonObject object);

    boolean previous_object(
        out CommonObject object);

    boolean next_n_objects(
        in unsigned long how_many,
        out CommonObjects objects);

    boolean object_at(
        in unsigned long at,
        out CommonObject object)
    raises (
        OutOfBounds );

    void destroy();

    unsigned long count();

    void reset();

    boolean next_values(
        out PropertyValues data);

    boolean previous_values(
        out PropertyValues data);

    boolean next_n_values(
        in unsigned long how_many,
        out Table data);

    boolean values_at(
        in unsigned long at,
        out PropertyValues data)
    raises (
        OutOfBounds );
};

interface Table {

    struct CellId {
        unsigned long row;
        unsigned long column;
```

```
    };

    typedef sequence<CellId> CellIds;

    struct TableCell {
        CellId cell;
        any value;
};

    typedef sequence<TableCell> TableCells;

    enum ExceptionType {
        read_only,
        type_mismatch,
        constraint_mismatch,
        invalid_row_column
    };

    struct TableException {
        CellId cell;
        ExceptionType type;
    };

    typedef sequence<TableException> TableExceptions;

    exception InvalidRow;
    exception InvalidColumn;
    exception IncompleteRow;
    exception TypeMismatch;
    exception ReadOnly;
    exception MultipleExceptions { TableExceptions exceptions; };

    readonly attribute unsigned long number_of_rows;
    readonly attribute unsigned long number_of_columns;
    readonly attribute unsigned long max_number_of_rows;
    readonly attribute CosPropertyService::PropertyTypes
        column_property_types;
    readonly attribute CosPropertyService::PropertyNames column_names;

    void describe_table(out unsigned long number_of_rows,
        out CosPropertyService::PropertyNames
            column_property_names;
        out CosPropertyService::PropertyTypes column_types);

    void get_row(in unsigned long row_number, out PropertyValues values)
            raises (InvalidRow);

    void set_row(in unsigned long row_number, in PropertyValues values)
            raises (MultipleExceptions, IncompleteRow, InvalidRow);

    any get_cell(in unsigned long row, in unsigned long column)
```

```
            raises (InvalidRow, InvalidColumn);

    void set_cell(in unsigned long row, in unsigned long column, in any
    value)
            raises (InvalidRow, InvalidColumn, TypeMismatch,
                ReadOnly);

    void get_cells(in CellIds list, out TableCells cells)
            raises (MultipleExceptions);

    void set_cells(in TableCells cells)
            raises (MultipleExceptions);
};
```

2.10.0.1 Iterator Description

The iterator described above is the primary result of a query (from **Locator** interface). This smart iterator can then be used to traverse the results of the query. In most cases it is desirable to not have a query actually create instances of objects when executed. Quite often a query is executed in an effort to locate a specific instance. For this reason, this interface supports the notion of lazy activation. For example, an application may invoke a query to obtain all Customers whose last name starts with "Sm." The programmer may then display this result in a GUI using the **next_n_values** method, (i.e., data only). Once the user has successfully found John Smith the application may invoke **object_at** method to get an actual reference to the John Smith object, represented through the **CommonObject** interface.

next_object()

Returns the next **CommonObject** reference in the sequence (result set). If there are no more objects to traverse, the method returns false, otherwise it returns true.

previous_object()

Returns the previous **ServiceLevelObject** reference in the sequence (result set). If there are no more objects to traverse, the method returns false, otherwise it returns true.

next_n_objects()

Returns the next '**how_many**' **CommonObject** references in the sequence (result set). If there are no more objects to traverse or '**how_many**' exceeds the number available, the method returns false, otherwise it returns true.

object_at()

Returns the **CommonObject** object reference at a specific location in the sequence (result set). If the index passed in exceeds the boundary of the sequence, then an exception is thrown.

destroy()

The user of this iterator must call the destroy method when finished for the server to effectively manage the memory associated with each result set.

reset()

This method resets the implicit cursor on the iterator back to zero. Note, the use of the iterator is exclusive to the client whom requested it; therefore, concurrency is not an issue.

count()

This method reruns the number of elements in the result set.

next_values()

Returns the next set of data in the sequence (result set). If there is no more data to traverse, the method returns false, otherwise it returns true.

previous_values()

Returns the previous set of data in the sequence (result set). If there is no more data to traverse, the method returns false, otherwise it returns true.

next_n_values()

Returns the next '**how_many**' sets of data in the sequence (result set) in the form of a Table (described below). If there is no more data to traverse or '**how_many**' exceeds the number available, the method returns false, otherwise it returns true.

values_at()

Returns the set of data at a specific location in the sequence (result set). If the index passed in exceeds the boundary of the sequence, then an exception is thrown.

2.10.0.2 *Generic Table Description*

CellId

CellId is a reference to a particular element of a table identified by the row number and column number. The use of the column number rather than column name enables more rapid access than access by row number and column name. Accessing by column names of the table support ad hoc table interactions is supported using the **CosPropertyService::PropertyNames column_names** attribute followed by accessing by **CellId**.

CellIds

CellIds is a sequence of **CellId**.

TableCell

TableCell is a structure containing the **CellId** of an element of a table and its value in a CORBA any.

TableCells

TableCells is a sequence of **TableCell**.

number_of_rows

Attribute **number_of_rows** represents the total number of rows in the table.

Number_of_columns

Attribute **number_of_columns** represents the total number of columns in the table.

max_number_of_rows

Attribute **max_number_of_rows** represents the highest number of rows that a particular table instance is permitted to support.

column_property_types

Attribute **column_property_types** is a sequence of **PropertyTypes** ordered by column number representing the CORBA type of the any contained in the cells of that column. This value borrows from Cos Property Service **CosPropertyService::PropertyTypes**.

column_names

Attribute **column_names** supports identification of the contents of a particular table column by name. A client side library function can be constructed to access and update by cells name around the column number interfaces supported by the table interaction calls described below. The type is borrowed from Cos Property Service **PropertyNames**.

describe_table()

Attribute **describe_table()** supports table description including the number of rows in the table and the types and names of each column.

get_row()

Supports accessing the values of a specified row. The row number is specified as **row_number**. The returned values are type **PropertyValues**. The **PropertyValues** are ordered per the **PropertyNames** specified by the **describe_table()** operation.

set_row()

Supports setting the values of a specified row. The row number is specified as **row_number**. The input values are **PropertyValues**. The types of the property values must correspond element by element to the types as reported by the **describe_table()** operation.

get_cell()

Supports getting the value of a particular element of a table. The element is specified by the row and column. The return value contains the contents of the element.

set_cell()

Supports the setting of a single element of a table. The element is specified by the row and column and the new contents are specified by the value.

get_cells()

Supports the getting of multiple element contents. The elements are specified by the list of **CellIds** and the contents of the cells are returned in a list of cells.

set_cells()

Supports setting elements of a table. The elements and the corresponding new values for the specified elements are contained in the cells input.

2.11 PMF Module Declaration

```
#ifndef PartyManagementFacility_idl
#define PartyManagementFacility_idl

#include "CosFinance.idl"

#pragma prefix "omg.org"

module PMF
{
...
};
```

#include "CosFinance.idl"

This file contains all of the type declarations defined above from the **CosFinance** module.

#pragma prefix “omg.org”

To prevent name pollution and name clashing of IDL types this module uses the pragma prefix that is the reverse of the OMG’s DNS name.

2.12 General Type Information

```
//forward declarations

//management
interface GroupManager;
interface RoleManager;
interface PartyRoleManager;
interface NodeManager;
interface PartyManager;
interface RelationshipManager;
interface PartyRelationshipManager;
interface ContactInformationFactory;

//aggregation
interface Role;
interface Node;

//core
interface Party;
interface Relationship;
interface PartyRelationship;
interface PartyRole;
interface Person;
interface Organization;

//contact information
interface ContactInformation;

//typedefs
typedef string                              RoleName;
typedef sequence<Role>                      Roles;
typedef sequence<RoleName>                  RoleNames;
typedef string                              ContactType;
typedef sequence<string>                    ContactTypes;
typedef sequence<ContactInformation>        ContactInformationSeq;
```

Forward Declarations

These are all the types represented in this module.

Role, Roles

Role is an alias for Role. It represents a primary object (person or organization) in the context of a relationship. For example, 'husband,' 'spouse,' 'programmer,' 'dad,' and 'hacker' could all be Roles of a single primary *Person* object. Roles have the ability to extend the attribution and behavior of the primary object they represent.

RoleName, RoleNames

RoleName is a string representation generally used to describe a **Role** object.

ContactType, ContactTypes

ContactType is an alias for string that is used by the **ContactInformation** interface to specify the type of contact information. Examples of different types of contact information are: "**home_address**," "**business_address**," "**email_address**," and "**home_phone**."

ContactInformationSeq

A sequence of contact information used to set various contact information at once.

2.13 Role

```
interface Role : CommonContainer
{
    exception MoreThanOneContained {};
    exception InvalidContainedRole {};
    exception InvalidRole {};
    exception InvalidAggregation {};
    exception MaximumCardinalityExceeded {};
    exception ObjectNotFound {};

    readonly attribute RoleName role_name;

    attribute CommonObject primary_object;

    CommonObject get_related_object(in RoleName contained_role,
                    in Date as_of_date)
        raises (MoreThanOneContained, InvalidRole);

    void get_all_related_objects_by_role(in RoleName contained_role,
                    in unsigned long how_many,
                    in Date as_of_date,
                    out CommonObjects sequence,
                    out Iterator)
        raises (InvalidRole);

    void add_related_object(in Role object,
                    in Date effective_date)
        raises (IsDuplicate,
            InvalidRole,
            InvalidAggregation,
            MaximumCardinalityExceeded);

    void add_related_objects(in Roles objects,
                    in Date effective_date)
        raises (IsDuplicate,
            InvalidRole,
            InvalidAggregation,
            MaximumCardinalityExceeded);
```

```
    Void remove_related_object(in Role object,
                               in Date effective_date)
        raises (ObjectNotFound);
};
```

role_name

Since the specification dictates that first class role objects must be present in the context of a relationship, the role name and the role object are synonymous. For the most part this attribute simply provides quick stringified access to the type name.

primary_object

The specification specifies an aggregation approach to role behavior. This interface represents the base interface for all roles and therefore provides reference back to the primary object that it is representing in the relationship. The role object may in fact expose methods that are implemented on the primary object. In the context of this specification, the primary object must always be of type Person or Organization.

get_related_object()

This method provides a mechanism to obtain a contained object based on the role of the contained object and the date that it was actually contained. For example, a client could invoke **get_contained_object_by_role()** passing "wife" and "12/12/97" to obtain a reference to a **CommonObject** that represents the *Person* he was married to at that time. Note the signature of this method implies 1:1 types of aggregation. As the example indicates, a party generally does not have more than one active wife at a time. If for example, the **CommonObject** was representing an Employer and this method was invoked to obtain all of the Employees that worked there as of 1/1/82 then, assuming there was more than one, this method would throw the MoreThanOneContained exception. Likewise, if this method was invoked to obtain the "wife" for an individual and they did not have a wife, then the InvalidContainedRole exception would be thrown.

get_all_related_objects_by_role()

This method returns all contained **CommonObjects** based on the role and as of date passed in. If the number of contained objects exceeds the **how_many** parameter, then the remainders are returned in the form of an Iterator. An example use of this method is where the **CommonObject** represents an Employer and the client requests all contained "Employee" objects as of 1/1/82. The signature of this method implies support for 1:m aggregations. However, it may be invoked for 1:1 associations where the result would likely be a sequence of one (1), assuming that **how_many** was not specified as zero (0).

add_related_object()

This method allows for the containment of one **CommonObject** into another **CommonObject**, or **Container**.

add_related_objects()

This method allows for the containment of many **CommonObjects** into another **CommonObject**, or **Container**.

remove_related_object()

This method allows for removing an object from its container.

2.14 Node

```
interface Node : CommonObject
{
    exception UnknownRole {};
    exception RoleNotFound {};
    exception NotSupported {};

    Roles get_all_roles()
        raises (NotSupported);

    RoleNames get_all_role_names()
        raises (NotSupported);

    void add_role(in Role role)
        raises (NotSupported);

    void remove_role(in Role role)
        raises (RoleNotFound,
                NotSupported);

    Roles get_roles(in RoleName role_name,
        Raises(UnknownRoleName,
            NotSupported);
};
```

get_all_roles()

This method returns a list of all the roles a specific **CommonObject** plays. This is a reflection of the relationships created by using the aggregation methods described in **CommonContainer**, a subtype of **CommonObject**. For example, if the **CommonObject** represents a Person, then this method may return *Husband*, *Claimant*, *Lienholder*, and/or *Attorney*. Note, some roles introduce role specific attribution and/or behavior that may only be accessible by issuing a subsequent **resolve()** invocation on the **Manager** interface for the specific role type, (e.g., *Claimaint).*

get_all_role_names()

This method returns a sequence of strings that represents all of the roles this object currently plays. This information is indirectly a result of an aggregation.

add_role()

This method is called implicitly by the **CommonContaine**r to inform this primary object of its new relationship.

remove_role()

This method is called implicitly by the **CommonContainer** to inform this primary object that a previously established relationship is being broken.

get_roles()

This method returns the roles associated with the string role name passed in. For example, **get_roles**("Employee") would return a single or Employee reference or potentially many Employee references if the person worked for multiple companies.

2.15 Party

```
interface Party : Node
{
    ContactInformation get_contact_information(in ContactType type, in Date as_of_date);
    void set_contact_information(in ContactInformation, in Date as_of_date);
};
```

get_contact_information()

Returns a reference to the **ContactInformation** for the specified type (i.e., "home" or "business") as it existed on the date provided.

set_contact_information()

Sets or adds a new set of contact information.

2.16 PartyRole

```
typedef sequence<Party> Parties;
typedef sequence<PartyRole> PartyRoles;

interface PartyRole : Role
{
        ContactInformation get_contact_information(in ContactType type, in Date as_of_date);
        void set_contact_information(in ContactInformation, in Date as_of_date);

        PartyRole get_related_party_role(in RoleName other_role,
                                    in Date as_of_date)
                    Raises(MoreThanOneContained,
                                    InvalidRole);

        void    get_all_related_party_roles(in RoleName contained_role,
                                    in unsigned long how_many,
                                    in Date as_of_date,
                                    out PartyRoles related_parties,
                                    out Iterator iter)
```

```
                    Raises(InvalidRole);

void        add_related_party(in PartyRole other_party,
                            in Date as_of_date)
                    Raises(IsDuplicate,
                            InvalidRole,
                            InvalidAggregation,
                            MaximumCardinalityExceeded);

void        add_related_party_roles(in PartyRoles other_parties,
                            in Date as_of_date)
                    Raises(IsDuplicate,
                            InvalidRole,
                            InvalidAggregation,
                            MaximumCardinalityExceeded);

void        remove_related_party(in Party object,
                            in Date as_of_date)
                    Raises(ObjectNotFound);

};
```

A **PartyRole** represents a **Party** (person or an organization) in a relationship.This interface simplifies the **Role** interface by providing some higher level wrappers to its base functionality.

get_contact_information()

Returns a reference to the **ContactInformation** for the specified type (i.e., “home” or “business”) as it existed on the date provided.

set_contact_information()

Sets or adds a new set of contact information.

get_related_party_role()

Convenience method to gain access to related party role object given its role relative to this object.

get_all_related_party_roles()

Returns all party role objects that play a particular role as of a particular date. For example, a user could ask of an employer reference - provide me with all of the employee objects as of 1/1/98.

add_related_party_role()

Associates another party role with this party role.

add_related_party_roles()

Associates many other party roles with this party role and each assumes their respective roles.

remove_related_party_role()

Tears down a relationship between two party roles.

2.17 Party Relationship

```
Interface PartyRelationship : Relationship
{
    PartyRoles get_related_party_roles();
};
```

get_related_party_roles()

Given a relationship (e.g., Marriage) provides the specific **Party** objects involved in the relationship.

2.18 Person

```
interface Person : Party
{
};
```

Person is a specific derivation of **Party**.

2.19 Organization

```
interface Organization : Party
{
};
```

Organization is a specific derivation of **Party**.

2.20 Node Manager

```
interface NodeManager : CosFinance::Manager
{
        CosFinance::Types     get_supported_nodes();
};
```

Note object creation of base types has been moved to the CosFinance::Manager. Since Node does not add information to the creation process a Node can simply be created by using the inherited create method.

get_supported_nodes()

This method returns the types of the objects this manager is capable of creating.

2.21 Party Manager

```
interface PartyManager : NodeManager
{

interface PartyManager : NodeManager
{
    CosFinance::Types         get_supported_parties();

    Party create_party ( in CosFinance::Manager::InitCommonObject data,
                          in ContactInformationSeq contact_information)
            raises (
                          TypeNotSupported,
                          InvalidInitializationType,
                          InvalidInitializationValue,
                          DuplicateObject);

    Parties create_multiple_parties(
                          in CosFinance::Manager::InitCommonObjects data)
            raises (
                          TypeNotSupported,
                          InvalidInitializationType,
                          InvalidInitializationValue);
};
```

get_supported_parties()

This method returns the types of Parties that this manager is capable of creating.

create_party()

This method simply adds the ability to initialize **ContactInformation** on the Party since the generic create method on the **CosFinance::Manager** interface does not allow for this.

create_multiple_parties()

This method creates many parties in batch mode. If any creation fails, then an exception is thrown and it can be assumed that the entire batch has been rolled back.

2.22 Role Manager

```
interface RoleManager : CosFinance::Manager
{

    RoleNames         get_supported_roles();

    Role create_role(
                in CosFinance::Type role_type,
                in Cosfinance::CommonObject primary_object)
```

```
        raises (
                DuplicateObject,
                TypeNotSupported);

    Role create_from_template(
                in CosFinance::Templates templates)
        raises (
                TypeNotSupported);
};
```

get_supported_roles()

Returns all role types in the form of a string that this manager is capable of creating.

create_role()

Given a primary object and the requested type creates a **CommonObject** object and associates it with the primary. If the client is creating a specialized type of **CommonObject** (i.e., a Party or Role), then they will need to perform a narrow to the appropriate type.

create_from_template()

Creates a **CommonObject** object from the specified template(s). Templates can trigger other elements to be added to the object automatically at creation time. Clients will need to perform a narrow if creating a specialized type of **CommonObject**. Note since templates in general are optional this method is also optional.

2.23 PartyRoleManager

```
interface PartyRoleManager : RoleManager
{
    RoleNames get_supported_party_roles();
};
```

get_supported_party_roles()

Returns all types in the form of a string that this manager is capable of creating.

2.24 Relationship Manager

```
interface RelationshipManager : CosFinance::Manager
{
    exception RoleTypeError {};
    exception UnknownRole {};

    CosFinance::Types      get_supported_relationships();

    RoleNames              get_supported_roles_for_relationship(
                               in CosFinance::Type relationship_type);
```

```
    Relationship create_relationship (
                in CosFinance::Type relationship_type,
                in Role role_a,
                in Role role_b,
                in Cosfinance::Date as_of_date)
        raises (
                RoleTypeError,
                UnknownRole);

    Relationship create_many_relationship(
                in CosFinance::Type relationship_type,
                in Role role,
                in Roles roles,
                in CosFinance::Date as_of_date)
        raises (
            RoleTypeError,
            UnknownRole);

    Relationship create_many_relationship(
                in CosFinance::Type relationship_type,
                in Role role,
                in Roles roles,
                in CosFinance::Date as_of_date)
        raises (
            RoleTypeError,
            UnknownRole);
};
```

get_supported_relationships()

This method returns all of the types of relationships that this manager is capable of creating.

get_supported_roles_for_relationship()

Given a specific relationship provides the roles that are allowed on that relationship. Note this factory constrains construction of a new relationship to comply with the role rules for that relationship.

create_relationship()

Given two role objects a relationship type and an effective date creates a new first class relationship object.

create_many_relationship()

Given a role object and many others along with a relationship type and an effective date creates a new first class relationship object.

2.25 PartyRelationship Manager

```
Interface PartyRelationshipManager : RelationshipManager
{
};
```

This specific derivation of **RelationshipManager** is currently an empty interface. It is assumed that specific constraints could be enforced at this level that are unique for 'Party' relationships.

2.26 Group Manager

```
Interface Group : CommonContainer
{
    attribute string group_name;
};

interface GroupManager
{
    Group create(in string group_name);
};
```

create()

Groups are fairly simple collections without any notion of role – or relational context. For example, a group may be the "Fortune 100 Companies."

2.27 ContactInformationFactory

```
interface ContactInformationFactory
{
    ContactInformation create(in ContactType type);
};
```

create()

Creates a **ContactInformation** object of the appropriate type.

2.28 Summary

The high level interfaces are relatively generic and offer flexibility in terms of implementation techniques. The interfaces offer type, behavior, and attribution extensibility without invalidating the model. They are also readily capable of wrapping a legacy party management system or could be used as the public interface to a new party management system that implements the recommended domain model.

Compliance, Conformance, and Known Issues 3

Contents

This chapter contains the following topics.

Topic	**Page**
"Compliance with Existing Specifications"	3-1
"Levels of Conformance"	3-3
"Known Issues"	3-3

3.1 Compliance with Existing Specifications

3.1.1 Transaction Service (OTS)

This specification addresses the integration of an OTS and does not require any modifications to the current OTS specification. The **CommonObject** interface derives from **TransactionalObject** simply to allow for the implicit propagation of context information in a transactional system. As an implementation choice the PMF vendor may choose to use the Coordinator reference to register the Party reference as a synchronization point and thus make the Party persistent during **before_completion**.

3.1.2 Relationship Service

This specification positions the use of the relationship service as an implementation decision. The fundamental reason for this direction is based on the overhead that **CosRelationships** introduces. This and other justification has been documented in the "Domain Model and Design Objectives" chapter. In addition, direction has been given toward using the PMF interfaces as a wrapper on **CosRelationships** in Appendix C- "Wrapping CosRelationships."

3.1.3 Security Service

The PMF does not expose any security specific interfaces, and instead relies on the underlying CORBA infrastructure and services to provide the security mechanisms needed.

CORBA sec will be used as the underlying mechanism for distributed security and handling access to the IDL-based interfaces of the facility.

For more information see the "Security and Party Management" chapter.

3.1.4 Persistent Object Service (POS)

It is assumed that this specification can make use of either the existing OMG Persistent Object Service or the new PSS that is being defined. This specification places persistence as an implementation issue that is masked behind an OTS transaction. For example, the business objects presented in this document support the **Resource** interface and will make themselves persistent as they receive the 'prepare' message. They do not explicitly publish any other interfaces specifically for persistence needs.

3.1.5 Query Service

This specification does not directly use any of the interfaces suggested in the Query Service. The **PartyLocator** interface does offer an **evaluate()** method similar to Query Service. But includes other value-added methods to support the federation of finding Party objects that adhere to a certain set of criteria.

3.1.6 Name Service

This specification references use of the Name Service and does not require any modifications to the existing service.

3.1.7 Trader Service

This specification references use of the Trader Service and does not require any modifications to the existing service.

3.1.8 Event Service

This specification does not require any modification to the existing Event Service. But this is currently a known issue that must be resolved (i.e., use of Event, Notification, Publish-Subscribe, BOF).

3.1.9 Externalization Service

Specific interfaces from the **CosExternalizationService** have been leveraged. This specification does not require any modifications to the existing service.

3.2 Levels of Conformance

There currently is only one level of conformance.

3.3 Known Issues

3.3.1 Notification Support

The new Notification Service interfaces have been integrated into the **Manager** interface to represent type level notification. There is an outstanding question regarding the consumer registration according to the Notification Service and the requirement regarding support for instance level notification.

Contact Information

The specification currently does not provide a mechanism to locate and retrieve existing references to contact information. This is an issue because many parties may share the same contact information and there needs to be a way to obtain an existing reference and reuse it.

Security and Party Management 4

4.1 Security Issues

At a minimum, the PMF party and relationship interfaces are security sensitive. These objects will have access control requirements to constrain who may view the data (confidentiality) as well as who may modify the data (integrity). Depending on the environment, transmitted requests may also need to be protected from attacks over the network (both passive monitoring and active intrusions).

CORBA sec will be used as the underlying mechanism for distributed security and handling access to the IDL-based interfaces of the facility.

Since the facility allows parties to be externalized, there is an issue relating to security and externalization.

There is a need for storage of the security information and also for encryption of the externalized data. Because of these issues this specification recommends that any application handling externalized objects must be security-aware and trusted to protect the object contents in a way that is consistent with the CORBA security policy for the object. If the application does not adequately protect the data in memory, in persistent store, and on the wire, then CORBA security policy for the object could be subverted. Additionally when the externalized object is internalized, it is the responsibility of a security-aware object factory to assign the appropriate CORBA security policy domain to the newly generated object reference.

There is a need for party management components to register and administer domain information. Examples of this could be payroll/HR department and access to salary information vs. other admin staff and employees. Also, considering a customer service (call center) financial example, it is often the case when calling a credit card company, that for security reasons, the representative on the other side knows only part of the account information. However, when PIN numbers are involved (like changing, verifying, and creating) the customer is transferred to a supervisor so that a single person there does not have all the information.

To address these needs, the party management components will use the CORBA Security Level 2 security policy domain manager and associated policy interfaces to define and administer security policies for party management objects. Security policy domains will be used to maintain separation of sensitive data.

Party management makes heavy use of roles (insurer, insured, husband, and wife). Depending on the roles, there may be a need to limit who can create certain relationships. Also, walking down the tree of relationships and discovering that certain ones exist, and being allowed to play a particular role are additional restrictions that may require security above standard CORBA security. Party management can support design-time and dynamic run-time relationships and role definition. In the case of "slow-changing," or design-time relationships, a party management application could extract party management role information and use that information to set the CORBASec user attribute information. For example, a CORBAsec security attribute called "wife-of" to tie party management and CORBAsec together may be defined. Let's assume that the party management container for Bob is related to another container for Mary by the party management role "wife." (Mary is Bob's wife.) When Mary authenticates (logs in) using CORBASec PrincipalAuthenticator, it could call a party management interface that determines that Mary is Bob's wife, and so creates a CORBASec user credential for Mary that includes the "wife-of" user attribute with the value "Bob." An access policy could be created that only permitted access to clients who have the "wife-of" user attribute set to Bob (i.e., Bob's wife) is allowed to access this information. This capability is utilizing CORBASec Level 2 interfaces. If, however, there is a need to dynamically discover a user's attributes at invocation time (not just at log-in time), when there may be too many attributes used for access control to pass around in the credentials (say, more than 50), if the attributes are changing frequently (say, every few minutes), or if they are data dependent (role or wife changing often), there is a need for application level security and possibly parameter filtering.

Complete OMG IDL — A

```
#ifndef CosFinance_idl
#define CosFinance_idl

#include "CosProperties.idl"
#include "CosLifeCycle.idl"
#include "CosExternalization.idl"
#include "CosTransactions.idl"
#include "CosTime.idl"
#include "CosNotifyComm.idl"
#include "NamingAuthority.idl"
#include "CosNaming.idl"
#include "CosTrader.idl"

#pragma prefix "omg.org"

module CosFinance
{
    //forward declarations

    //management
    interface Table;
    interface Manager;
    interface TemplateManager;
    interface Iterator;
    interface Locator;

    //core
    interface CommonObject;
    interface CommonContainer;
    interface DateEffectiveObject;

    //typedef
    typedef string                    Type;
    typedef sequence<string>          Types;

    //defined here for forward declaration purposes
```

```
struct Template
{
        string name;
        Types  types;
};

typedef string                          QueryExpression;
typedef CosTime::UTO                    Date;
typedef any                             PropertyValue;
typedef sequence<PropertyValue>         PropertyValues;
typedef sequence<Template>              Templates;

//enumerators
enum ModificationState { Update, Correction };

//interfaces
interface Table {

    struct CellId {
            unsigned long row;
            unsigned long column;
    };

    typedef sequence<CellId> CellIds;

    struct TableCell {
            CellId cell;
            any value;
    };

    typedef sequence<TableCell> TableCells;

    enum ExceptionType {
            read_only,
            type_mismatch,
            constraint_mismatch,
            invalid_row_column
    };

    struct TableException {
            CellId cell;
            ExceptionType type;
    };

    typedef sequence<TableException> TableExceptions;

    exception InvalidRow {};
    exception InvalidColumn {};
    exception IncompleteRow {};
    exception TypeMismatch {};
    exception ReadOnly {};
    exception MultipleExceptions { TableExceptions exceptions; };

    readonly attribute unsigned long number_of_rows;
    readonly attribute unsigned long number_of_columns;
```

```
	readonly attribute unsigned long max_number_of_rows;
	readonly attribute CosPropertyService::PropertyTypes column_property_types;
	readonly attribute CosPropertyService::PropertyNames column_names;

	void describe_table(out unsigned long number_of_rows,
				out CosPropertyService::PropertyNames column_property_names,
				out CosPropertyService::PropertyTypes column_types);

	void get_row(in unsigned long row_number, out PropertyValues values)
				raises (InvalidRow);

	void set_row(in unsigned long row_number, in PropertyValues values)
				raises (MultipleExceptions, IncompleteRow, InvalidRow);

	any get_cell(in unsigned long row, in unsigned long column)
				raises (InvalidRow, InvalidColumn);

	void set_cell(in unsigned long row, in unsigned long column, in any value)
				raises (InvalidRow, InvalidColumn, TypeMismatch, ReadOnly);

	void get_cells(in CellIds list, out TableCells cells)
				raises (MultipleExceptions);

	void set_cells(in TableCells cells)
				raises (MultipleExceptions);

};

interface DateEffectiveObject
{
	attribute ModificationState update_state;

	attribute Date effective_start;
	attribute Date effective_end;

	boolean is_effective_now();
};

struct QualifiedObjectIdentity
{
	NamingAuthority::AuthorityId domain;
	Type type;
	NamingAuthority::LocalName id;
};

typedef sequence<CommonObject> CommonObjects;
typedef sequence<CommonContainer> CommonContainers;

interface CommonObject : CosLifecycle::LifecycleObject,
				CosStream::Streamable,
				CosPropertyService::PropertySetDef,
				CosTransactions::TransactionalObject,
				DateEffectiveObject
{
	exception ContainerNotFound {};
```

```
        exception CannotRemove {};

        readonly attribute QualifiedObjectIdentity identity;

        boolean is_dependent_object();

        boolean is_date_sensitive();

        CommonContainers get_containers();

        void add_container(in CommonContainer container);

        void remove_from_container(in CommonContainer container)
                                raises (ContainerNotFound,
                                        CannotRemove);
};

interface CommonContainer : CommonObject
{
        exception ObjectNotFound {};
        exception IsDuplicate {};
        exception InvalidAggregation {};
        exception MaximumCardinalityExceeded {};

        void add_contained_object(in CommonObject object,
                                  in Date as_of_date);

        void add_contained_objects(in CommonObjects objects,
                                   in Date as_of_date);

        CommonObject get_contained_object_by_id(in QualifiedObjectIdentity id,
                                                in Date as_of_date)
                    raises (ObjectNotFound);

        void remove_contained_object(in CommonObject object,
                                     in Date effective_date)
                    raises (ObjectNotFound);

        boolean has_contained_object(in CommonObject object,
                                     in Date as_of_date);

        void get_all_contained_objects(in Date as_of_date,
                                       out CommonObjects objects);

        void add_from_template(in Template template);

        Templates list_templates();
};

interface Manager : CosStream::StreamableFactory, CosNotifyComm::SequencePushSupplier
{
        exception   TypeNotSupported {};
        exception   DuplicateObject {};
        exception   InvalidInitializationType {};
        exception   InvalidInitializationValue {};
```

```
        struct InitCommonObject {
                    QualifiedObjectIdentity identity;
                    CosPropertyService::Properties data;
        };

        typedef sequence<InitCommonObject> InitCommonObjects;

        Types   get_supported_types();

        CommonObject create( in InitCommonObject data )
                        raises (
                                TypeNotSupported,
                                InvalidInitializationType,
                                InvalidInitializationValue,
                                DuplicateObject);

        void    get_supported_properties(
                            in Type type,
                            out CosPropertyService::PropertyDefs property_defs)
                        raises (
                                TypeNotSupported );

        Locator get_locator();
    };

    interface TemplateManager
    {

        exception TemplateNotFound {};

        void add_template(
                in Template template);

        void remove_template(
                in string name)
            raises (
                    TemplateNotFound);

        Template get_template(
                in string name)
            raises (
                    TemplateNotFound);

        Templates list_templates();
    };

    interface Iterator
    {

        exception OutOfBounds {};

        boolean next_object(
                    out CommonObject object)
                raises (
```

```
                    OutOfBounds );

        boolean previous_object(
                    out CommonObject object)
                raises (
                    OutOfBounds );

        boolean next_n_objects(
                    in unsigned long how_many,
                    out CommonObjects objects);

        boolean object_at(
                    in unsigned long at,
                    out CommonObject object)
                raises (
                    OutOfBounds );

        void destroy();

        void reset();

        unsigned long count();

        boolean next_values(
                    out PropertyValues data)
                raises (
                    OutOfBounds );

        boolean previous_values(
                    out PropertyValues data)
                raises (
                    OutOfBounds );

        boolean next_n_values(
                    in unsigned long how_many,
                    out Table data);

        boolean values_at(
                    in unsigned long at,
                    out PropertyValues data)
                raises (
                    OutOfBounds );

    };

    interface Locator
    {
        typedef sequence<string> CriteriaBasis;
        typedef sequence<string> SearchType;

        exception InvalidQuerySyntax {};
        exception NotImplemented {};
        exception SearchTypeNotSupported {};
        exception CriteriaBasisNotSupported {};
        exception NotFound {};
```

```
        exception InvalidIdentifier {};
        exception InvalidAsOfDate {};
        exception TypeNotSupported {};

        readonly attribute NamingAuthority::AuthorityId domain_name;
        readonly attribute CosNaming::NamingContext naming_context;
        readonly attribute CosTrading::TraderComponents trader_components;

        CommonObject resolve(
                        in QualifiedObjectIdentity identifier,
                        in Date as_of_date )
                    raises (
                        NotFound,
                        InvalidIdentifier,
                        InvalidAsOfDate );

        Iterator  evaluate(
                        in QueryExpression query)
                    raises (
                        InvalidQuerySyntax,
                        NotImplemented );

        CriteriaBasis get_supported_search_criteria();

        SearchType      get_supported_search_types();

        Iterator  query(
                        in Type object_type,
                        in string criteria,
                        in CriteriaBasis criteria_basis,
                        in SearchType type_of_search)
                    raises (
                        TypeNotSupported,
                        NotImplemented,
                        SearchTypeNotSupported,
                        CriteriaBasisNotSupported );

    };

};

#endif




#ifndef PartyManagementFacility_idl
#define PartyManagementFacility_idl

#include "CosFinance.idl"

#pragma prefix "omg.org"

module PMF
```

```
{
    //forward declarations

    //management
    interface GroupManager;
    interface RoleManager;
    interface PartyRoleManager;
    interface NodeManager;
    interface PartyManager;
    interface RelationshipManager;
    interface PartyRelationshipManager;
    interface ContactInformationFactory;

    //aggregation
    interface Role;
    interface Node;

    //core
    interface Party;
    interface Relationship;
    interface PartyRelationship;
    interface PartyRole;
    interface Person;
    interface Organization;

    //contact information
    interface ContactInformation;

    //typedefs
    typedef string                              RoleName;
    typedef sequence<Role>                      Roles;
    typedef sequence<RoleName>                  RoleNames;
    typedef string                              ContactType;
    typedef sequence<string>                    ContactTypes;
    typedef sequence<ContactInformation>        ContactInformationSeq;

    //interfaces
    interface Role : CosFinance::CommonContainer
    {

        exception MoreThanOneContained {};
        exception InvalidContainedRole {};
        exception InvalidRole {};
        exception InvalidAggregation {};
        exception MaximumCardinalityExceeded {};
        exception ObjectNotFound {};

        readonly attribute RoleName role_name;
        attribute CosFinance::CommonObject primary_object;

        CosFinance::CommonObject get_related_object(in RoleName contained_role,
                                                    in CosFinance::Date as_of_date)
                    raises (MoreThanOneContained,
                        InvalidRole);
```

```
    void get_all_related_objects_by_role(in RoleName contained_role,
                                         in unsigned long how_many,
                                         in CosFinance::Date as_of_date,
                                         out CosFinance::CommonObjects objects,
                                         out CosFinance::Iterator iter)
            raises (InvalidRole);

    void add_related_object(in Role object,
                            in CosFinance::Date effective_date)
            raises (IsDuplicate,
                    InvalidRole,
                    InvalidAggregation,
                    MaximumCardinalityExceeded);

    void add_related_objects(in Roles objects,
                             in CosFinance::Date effective_date)
            raises (IsDuplicate,
                    InvalidRole,
                    InvalidAggregation,
                    MaximumCardinalityExceeded);

    void remove_related_object(in Role object,
                               in CosFinance::Date effective_date)
            raises (ObjectNotFound);
};

typedef sequence<Party> Parties;
typedef sequence<PartyRole> PartyRoles;

interface PartyRole : Role
{
    ContactInformation get_contact_information(in ContactType type,
                                               in CosFinance::Date as_of_d   ate);
    void set_contact_information(in ContactInformation info,
                                               in CosFinance::Date as_of_d   ate);

    PartyRole get_related_party_role(in RoleName other_role,
                                     in CosFinance::Date as_of_date)
            raises (MoreThanOneContained,
                    InvalidRole);

    void get_all_related_party_roles(in RoleName contained_role,
                                     in unsigned long how_many,
                                     in CosFinance::Date as_of_date,
                                     out PartyRoles related_parties,
                                     out CosFinance::Iterator iter)
            raises (InvalidRole);

    void add_related_party_role(in PartyRole other_party,
                                in CosFinance::Date effective_date)
            raises (IsDuplicate,
                    InvalidRole,
                    InvalidAggregation,
                    MaximumCardinalityExceeded);
```

```
    void add_related_party_roles(in PartyRoles other_parties,
                                 in CosFinance::Date effective_date)
            raises (IsDuplicate,
                    InvalidRole,
                    InvalidAggregation,
                    MaximumCardinalityExceeded);

    void remove_related_party_role(in PartyRole other_party,
                                   in CosFinance::Date effective_date)
            raises (ObjectNotFound);

};

interface Relationship : Role
{
    CosFinance::CommonObjects get_related_objects();
};
interface PartyRelationship : Relationship
{
    PartyRoles get_related_party_roles();
};

interface Node : CosFinance::CommonObject
{
    exception UnknownRoleName {};
    exception RoleNotFound {};
    exception NotSupported {};

    Roles get_all_roles()
            raises (NotSupported);

    RoleNames get_all_role_names()
            raises (NotSupported);

    void add_role(in Role role)
            raises (NotSupported);

    void remove_role(in Role role)
            raises (RoleNotFound,
                    NotSupported);

    Roles get_roles(in RoleName role_name)
            raises (UnknownRoleName,
                    NotSupported);
};

interface Party : Node
{
    ContactInformation get_contact_information(in ContactType type,
                                               in CosFinance::Date as_of_date);
    void set_contact_information(in ContactInformation info,
                                 in CosFinance::Date as_of_date);
};

interface Organization : Party
```

```
    {
    };

    interface Person : Party
    {
    };

interface ContactInformation : CosFinance::DateEffectiveObject,CosPropertyService::PropertySetDef
    {
        attribute ContactType type;
        attribute string locale;
    };

    interface NodeManager : CosFinance::Manager
    {
        CosFinance::Types               get_supported_nodes();
    };

    interface PartyManager : NodeManager
    {
        CosFinance::Types               get_supported_parties();

        Party create_party ( in CosFinance::Manager::InitCommonObject data,
                             in ContactInformationSeq contact_information)
                raises (
                        TypeNotSupported,
                        InvalidInitializationType,
                        InvalidInitializationValue,
                        DuplicateObject);

        Parties create_multiple_parties(
                            in CosFinance::Manager::InitCommonObjects data)
                raises (
                        TypeNotSupported,
                        InvalidInitializationType,
                        InvalidInitializationValue);
    };

    interface RoleManager : CosFinance::Manager
    {

        RoleNames get_supported_roles();

        Role create_role (
                in CosFinance::Type role_type,
                in CosFinance::CommonObject primary_object)
            raises (
                    DuplicateObject,
                    TypeNotSupported);

        Role create_from_template(
                in CosFinance::Templates templates)
            raises (
                    TypeNotSupported);
    };
```

```
interface PartyRoleManager : RoleManager
{

    RoleNames get_supported_party_roles();

};

interface RelationshipManager : CosFinance::Manager
{
exception RoleTypeError {};
    exception UnknownRole {};

    CosFinance::Types       get_supported_relationships();

    RoleNames       get_supported_roles_for_relationship(
                            in CosFinance::Type relationship_type);

    Relationship create_relationship (
                            in CosFinance::Type relationship_type,
                            in Role role_a,
                            in Role role_b,
                            in CosFinance::Date as_of_date)
                raises (
                            RoleTypeError,
                            UnknownRole);

    Relationship create_many_relationship(
                            in CosFinance::Type relationship_type,
                            in Role role,
                            in Roles roles,
                            in CosFinance::Date as_of_date)
                raises (
                            RoleTypeError,
                            UnknownRole);

};

interface PartyRelationshipManager : RelationshipManager
{
};

interface ContactInformationFactory
{
    ContactInformation create();
};

interface Group : CosFinance::CommonContainer
{
    attribute string name;
};

interface GroupManager
{
```

```
        Group create_group(in string group_name);

    };
};

#endif
```

Collaboration Diagrams B

B.1 Usage Models

This appendix illustrates a few usage models to illustrate the anticipated interaction between the interfaces presented.

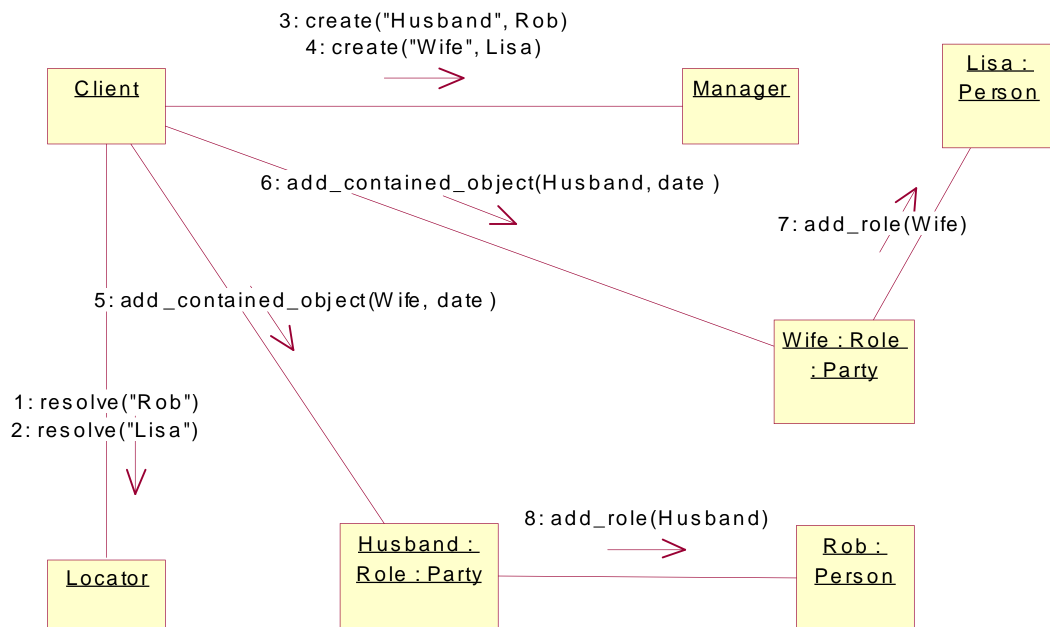
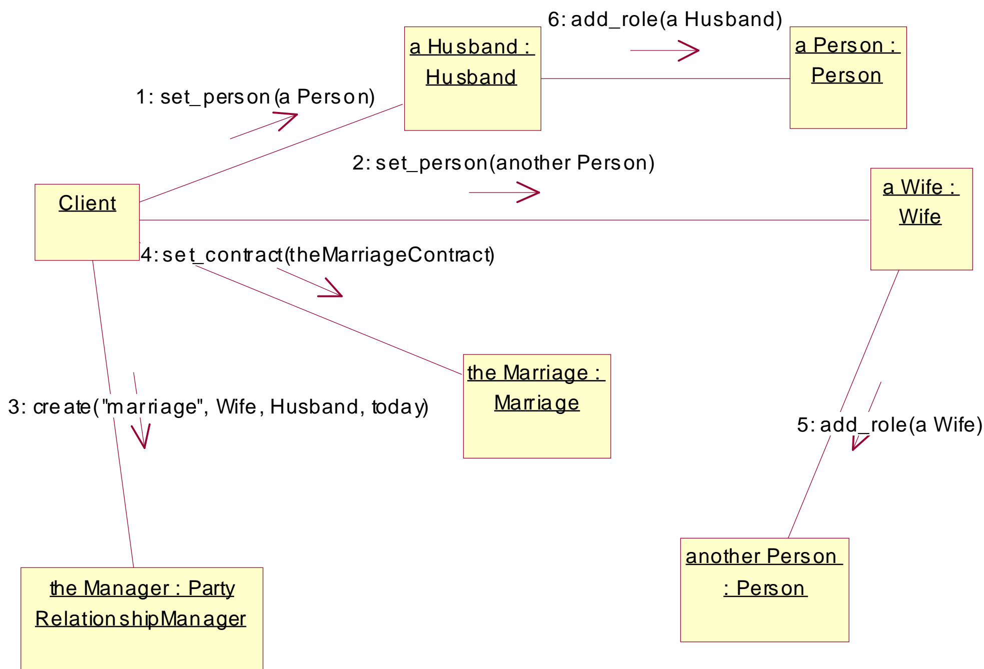


Figure B-1 Aggregate two objects while not creating a first class relationship object

Figure B-1 illustrates the ability to aggregate two, otherwise unaware, objects with each other while not creating a first class relationship object. This model allows the PartyManager interface to create the Husband and Wife role (as first class objects) that use aggregation to extend their primary objects, Person objects Lisa and Rob. As the roles are related through the **add_contained_object** method, actually inherited from Role, the **add_role** method is called implicitly on the primary objects so they can be kept up-to-date in regard to the collective roles they are playing.

The following collaboration diagram illustrates how the PMF is also capable of creating a first class relationship object (e.g., Marriage) if need be.



Note: This collaboration diagram assumes that Marriage is a specialized type of PartyRelationship and therefore has a set_contract method. If not using specialization, add_contained_object could be called with the marriage contract passed in as a CommonObject. Husband and Wife are both specialized party types in this diagram and at the generic level could use set_primary_object() instead of set_person.

Figure B-2 Creating a first class relationship object

Figure B-2 illustrates how the PMF is also capable of creating a first class relationship object (e.g., Marriage) if need be.

Wrapping Cos Relationships C

This specification has introduced a high level interface based on the Composite design pattern as a means for managing object aggregation. From an implementation perspective the PMF vendor could choose to manage these collections in process or could use an implementation of CosRelationships for that purpose. This approach has been taken for reasons outlined in the "Domain Model and Design Objectives" chapter and summarized in the "Compliance, Conformance, and Known Issues" chapter.

This appendix offers a few scenarios to illustrate explicitly how this integration may be implemented. This static model shows how the PMF Role becomes the primary wrapper for **CosRelationships**.

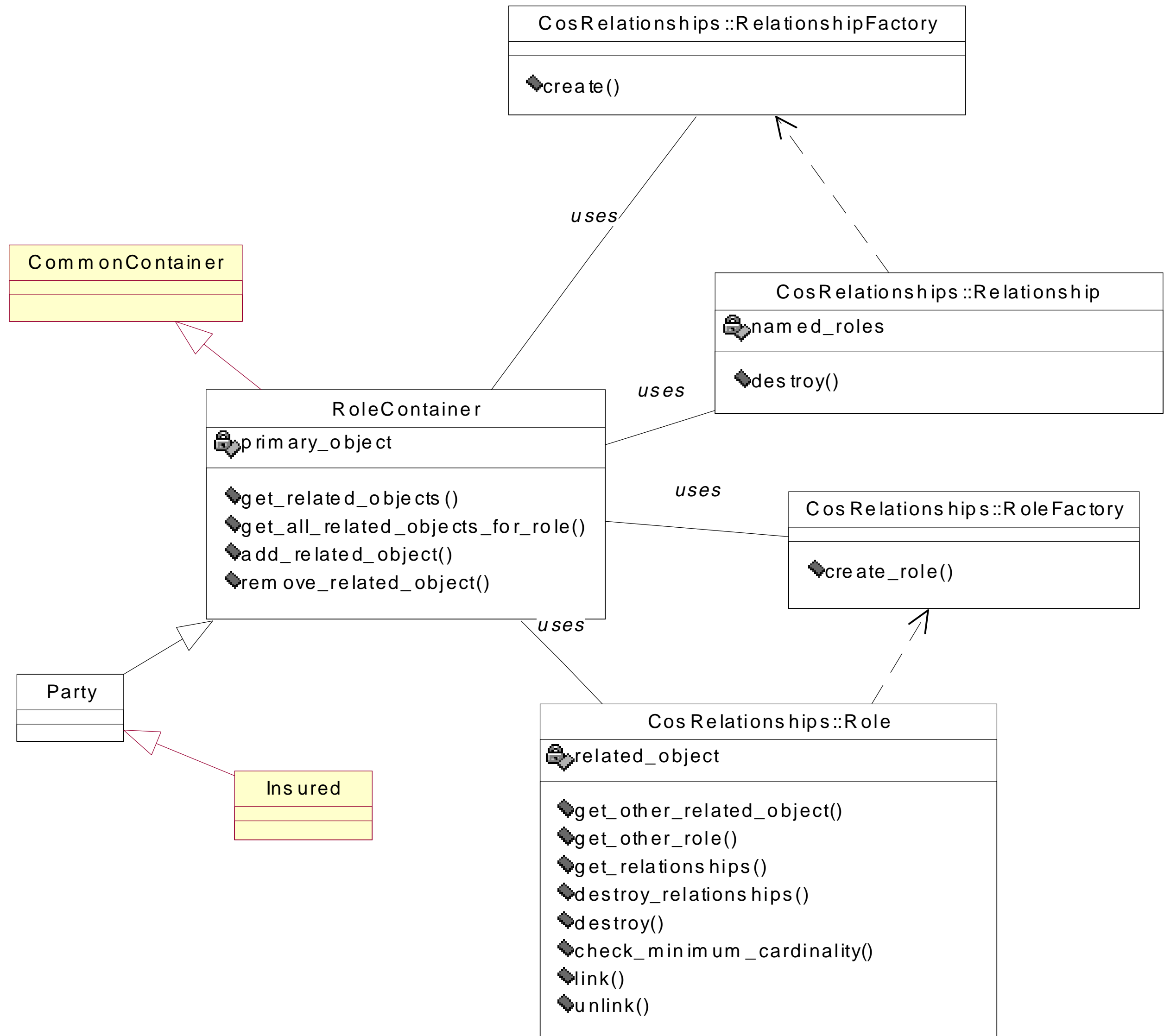


Figure C-1 Wrapping CosRelationships

The following interaction diagrams illustrate the creation and subsequent traversal of a 1:m relationship.

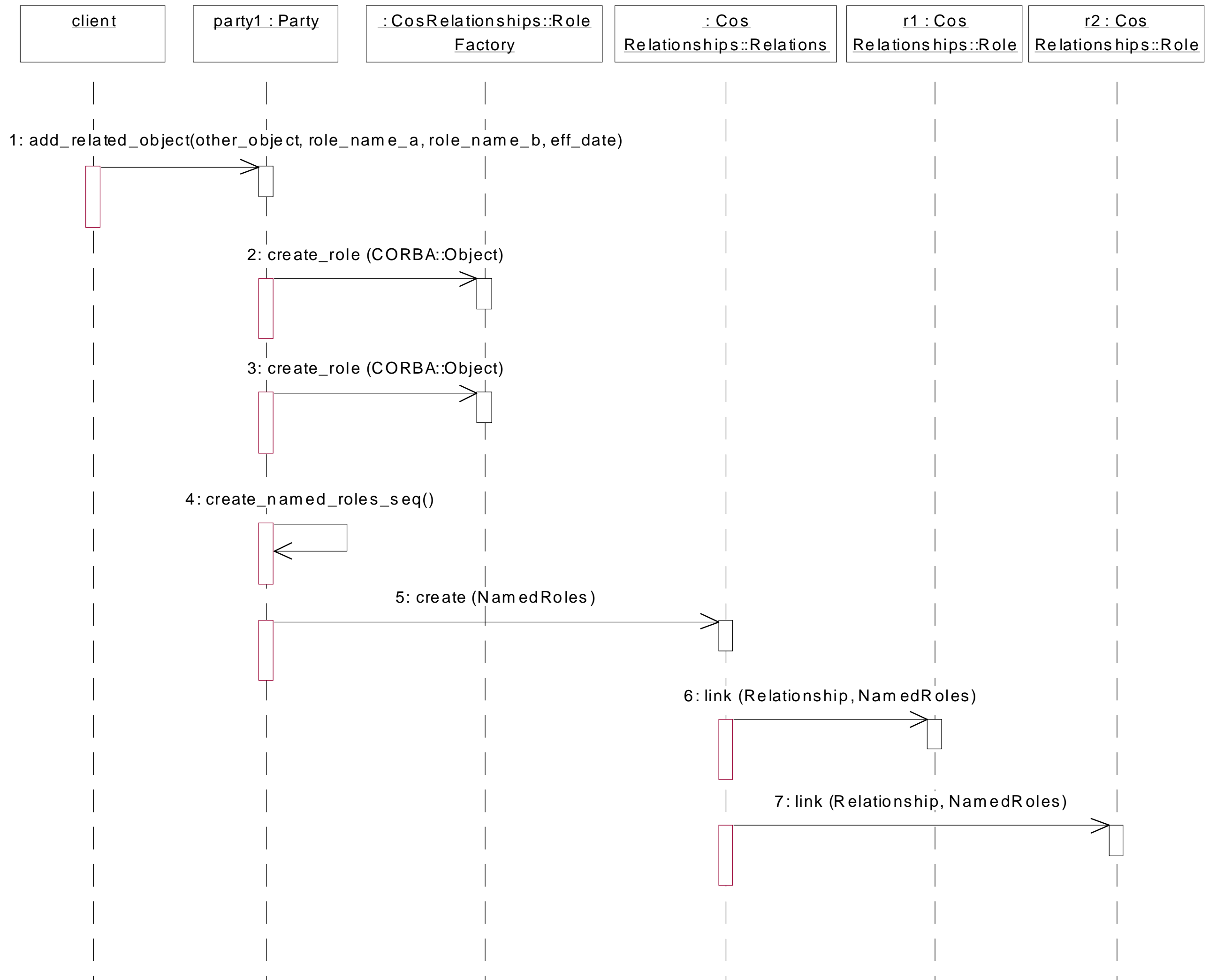


Figure C-2 Relationship Creation Example

Figure C-2 illustrates how the simple interface exposed to the client can wrap the somewhat complex task of creating a relationship on **CosRelationships**. It also illustrates the addition of date and time information that can be used to reconstruct past and future aggregations.

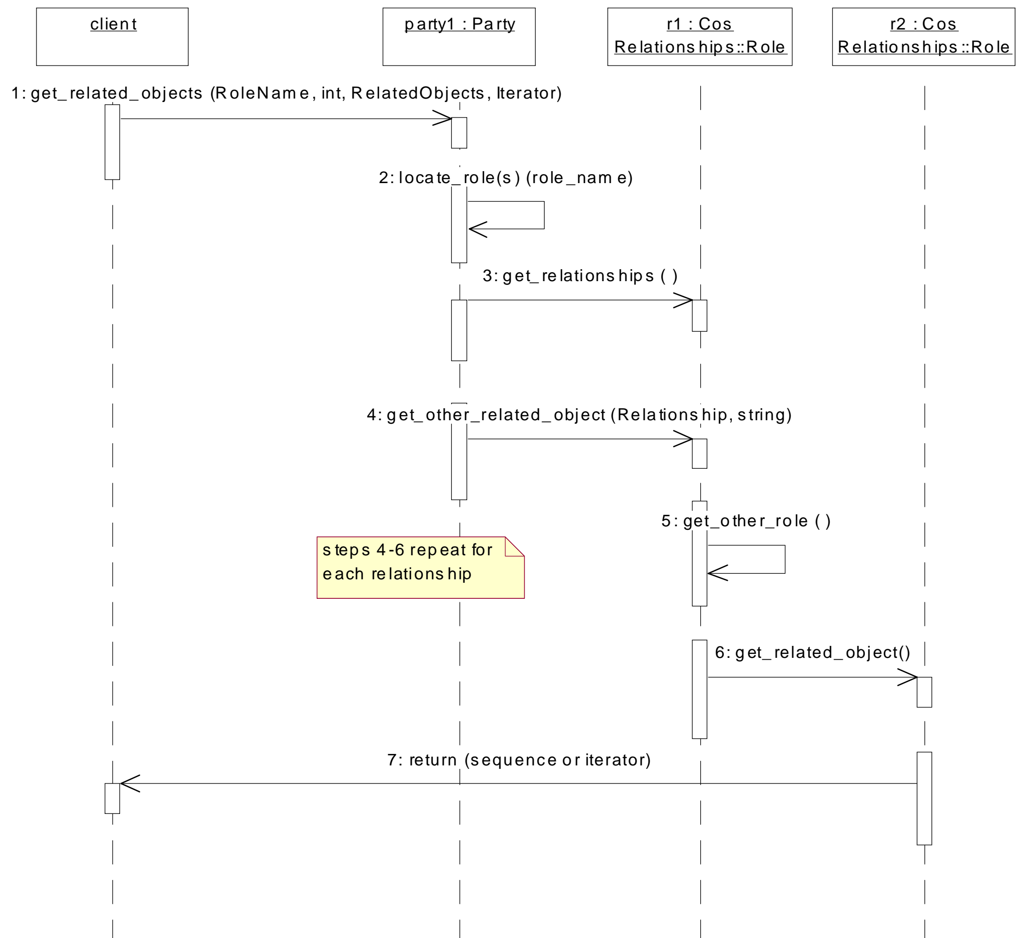
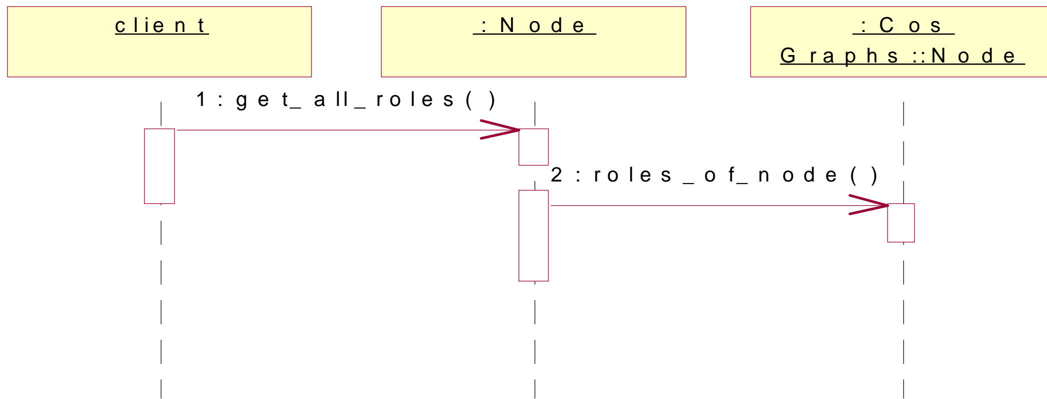


Figure C-3 Single client interface used to trigger retrieval of all related objects

Figure C-3 illustrates how a single client interface could be used to trigger the retrieval of all related objects. That is, in **CosRelationships** the link with each related object is hidden behind a *relationship* instance. In summary, the complexity introduced by encapsulating the primary object behind both a role and a relationship interface can be masked from the user of the PMF. Similarly the **Node** interface could interpose the **CosGraphs::Node** interface assuming **CosGraph::Node**'s had been creating using the factory.


client
: Node
: Cos
Graphs::Node
1 : get_all_roles( )
2 : roles_of_node( )

References D

[1] Gamma, E., Helm, R., Johnson, R., Vlissides, J. “Design Patterns Elements of Reusable Object-Oriented Software.” Addison-Wesley Publishing Company, 1995.

Index

Symbols
#include "CosFinance.idl 2-22

A
add_contained_object() 2-13
add_contained_objects() 2-13
add_container() 2-12
add_from_template() 2-14
add_related_object() 2-25
add_related_objects() 2-26
add_related_party_role() 2-28
add_related_party_roles() 2-28
add_role() 2-27
add_template() 2-15

C
CommonContainer (Inherited Interfaces) 2-13
CommonContainer (Local behavior) 2-13
CommonObject (Inherited Interfaces) 2-8
CommonObject (Local Attributes and Methods) 2-11
Composition Model 1-4
ContactType, ContactTypes 2-24
CosFinanceModule Declaration 2-2
CosLifeCycle
  LifeCycleObject 2-8
    copy (optional) 2-9
    move (optional) 2-9
    remove 2-9
CosPropertyService
  PropertySet
    define_properties() 2-10
    define_property() 2-9
    delete_all_properties() 2-10
    delete_properties() 2-10
    delete_property() 2-10
    get_all_properties() 2-10
    get_all_property_names() 2-10
    get_number_of_properties() 2-10
    get_properties() 2-10
    get_property_value() 2-10
    is_property_defined() 2-10
  PropertySetDef 2-8, 2-9
    define_properties_with_modes() 2-11
    define_property_with_mode () 2-11
    get_allowed_properties () 2-10
    get_allowed_property_types () 2-10
    get_property_mode() 2-11
    get_property_modes () 2-11
    set_property_mode() 2-11
    set_property_modes () 2-11
CosStream
  Streamable 2-9
    externalize_to_stream() 2-9
    internalize_from_stream() 2-9
CosTransactions
  TransactionalObject 2-11
count() 2-20
create() 2-33
create_from_template() 2-31
create_many_relationship() 2-32
create_relationship() 2-32
create_role() 2-31

D
DateEffectiveObject 2-11
Definition of Terms and Assumptions 1-4
destroy() 2-20

E
effective_end() 2-7
effective_start() 2-7
evaluate() 2-16

F
Forward Declarations 2-23

G
General Type Information 2-4
Generic Table Description 2-20
get_all_contained_objects() 2-14
get_all_related_objects_by_role() 2-25
get_all_related_party_roles() 2-28
get_all_role_names() 2-26
get_all_roles() 2-26
get_contact_information() 2-27, 2-28
get_contained_object_by_id() 2-13
get_containers() 2-12
get_related_object() 2-25
get_related_party_role() 2-28
get_roles() 2-27
get_supported_relationships() 2-32
get_supported_roles() 2-31
get_supported_roles_for_relationship() 2-32
get_template() 2-15

H
has_contained_objects() 2-14
High Level Comparison with CosRelationships 1-8

I
identity 2-11
is_date_sensitive() 2-12
is_dependent_object() 2-12
is_effective_now() 2-8
Iterator Description 2-19

L
list_templates() 2-14, 2-15
Locating Existing Party Information 1-12

M
Manager and Object Factory Model 1-9

N
next_n_objects() 2-19
next_n_values() 2-20
next_object() 2-19
next_values() 2-20
Node 2-26

O
object_at() 2-19

Index

P
Party 2-27
Party and Contact Information 1-7
Party Management Facility Interfaces 2-2
Party Manager 2-30
Party Relationships as First Class Objects 1-8
PartyRole 2-27
PMF Module Declaration 2-22
previous_object() 2-19
previous_values() 2-20
primary_object 2-25

Q
query() 2-16

R
remove_contained_objects() 2-13
remove_related_object() 2-26
remove_related_party_role() 2-29
remove_role() 2-27
remove_template() 2-15
reset() 2-20
resolve() 2-16
Role 2-24
Role Aware Composition Model 1-5
Role, Roles 2-23
role_name 2-25
RoleName, RoleNames 2-23

S
set_contact_information() 2-27, 2-28

T
TableCell 2-21

U
update_state 2-7

V
values_at() 2-20