# Task and Session CBOs Specification

**Version 1.0**
**New Edition: April 2000**

# *Contents*

# Contents

# *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.

## *Associated OMG Documents*

Formal documents are OMG's final, published specifications. Currently, formal documentation is available in both PDF and PostScript format from the OMG web site. Use this URL to access the OMG formal documents: http://www.omg.org/library/specindx.html.

The formal documentation is organized as follows:

- *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 Languages*, a collection of language mapping specifications. See the individual language mapping specifications.
- *CORBA Services: Common Object Services Specification* contains specifications for OMG's Object Services.
- *CORBA Facilities: Common Facilities Specification* includes OMG's Common Facility specifications.
- *CORBA Domain Technologies*, a collection of stand-alone specifications that relate to the following domain industries:
  - *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 specification 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, Suite 201
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:

- NIIP Consortium
- OSM

# Task & Session with Resource Objects 1

## Contents

This chapter contains the following sections

| Section Title | Page |
|---|---|
| “Resource and Process Objects” | |
| “Summary of Optional versus Mandatory Interfaces” | |
| “Proposed Compliance Points” | |

## 1.1 Resource and Process Objects

### 1.1.1 Process Objects

Process objects, represented as Tasks in this specification, describe User units of work that bind a User to selected data and process resources. Viewed in terms of model-view-controller, Tasks are the controller with commands and selections. A Task defines *what* instances to process and *how* to process (workflow, tool, or other executable). A Task may represent:

- a simple request such as “edit a file” where *what* is file x and *how* is editor y
- a more abstract request where *what* is collection j of files and *how* is workflow k (which may contain a hierarchy of workflows)

Tasks that describe simple requests are very similar to objects found in all systems - commands. Task extends and generalizes the notion of command objects to include abstract selections which facilitate dynamic binding of data and process resources.

Task objects represent information that is typically not presented at user interfaces.

Simple requests (e.g., “edit a file”) create a Task but what is exposed at the user interface is likely to be the editor, not a presentation of the Task object. Similarly Tasks using workflows may, or may not, be presented as hooks to workflow viewer and worklist handler user interfaces.

Task objects represent the decomposition of work, within projects, organizations, and by people, to atomic units (individual work items) that are independent of, and isolated from, each other. Dependencies between Task objects can be handled, as in the real world, by waiting, polling, and requesting. Task objects may only be executed by their assigned (by resource utilization mechanisms), authenticated User, not by another User. If a process has a Task dependency it must either wait for this dependency to be met or define rules for satisfying the dependency through alternative paths. Task objects provide an abstraction of work consistent with how people and projects define and manage their work. This includes the reality that collaboration between people requires recognition of their independence and separateness in time and space.

A Task is associated with one, and only one, User. Tasks may, however, depend on other Tasks. Dependent and independent Tasks have general differences, as described below, but are represented in the same way.

### *Independent Tasks*

- Typically created by direct User requests such as “edit a file” or “print a document” as unplanned units of work.
- Can be created by other Users.
- Do not depend on other Tasks; however, when workflows are used this may not be known until runtime.

### *Dependent Tasks*

- Typically created by workflow and project management tools or scheduled by event handlers, as planned units of work.
- Contain rules for sequencing, synchronization, and event handling.
- Use concurrency mechanisms, configurations, and versions.

Unplanned Tasks typically bind a tool, rather than a workflow process resource, to a specific data resource. Using the “edit a file” example the Task object created will bind the requester (User), to the edit tool selected (process resource), and to the file selected (data resource). It is also possible that the request to “edit a file” included the selection of an edit workflow, rather than an edit tool, and a collection of files, rather than a single file. This, however, is just another Task with more abstract selections – the process Resource is the selected workflow (which may itself invoke a sequence of tools and other workflows), and the data Resource is the selected collection of files (which will be selected by the User and/or workflow when the Task executes).

Planned Tasks may contain dependencies on other Users, with other skills. They are usually controlled by workflows and operate on data resource contexts with types and versions used by process rules for selection of instances at runtime.

Tasks, planned and unplanned, are scalable atomic units of work that:

- capture user requests and assigned work
- can be executed by workflows or tools
- specify information instances to process and interpret
- can have abstract selections that dynamically bind process and data
- utilize resources selected by resource assignment mechanisms
- contain history for enabling recovery and analysis of cost and performance
- model units of work in terms of people and resources to enable collaboration.

### 1.1.2 Resource Objects

Resource is implemented with adapters that wrap distributed, loosely coupled, concrete resources. Adapted resources are CORBA components which include dynamically wrapped internet resources, workflows, resource managers, and domain objects.

Resources are collected in Workspaces and used to represent:

- process resources such as workflows and applications
- data resources which include files, pages, domain, and other CORBA objects.

Resource implementations are responsible for maintaining the integrity and consistency of the User computing environment. This includes referential integrity between resources, change notification, and recovery mechanisms.

Resources are like “bookmarks” in browsers that provide:

- links to independent resource objects with managed loose coupling
- role based links to units of work (Tasks)
- resource sharing via CORBA security and concurrency mechanisms
- typed resources that use interoperation capabilities provided by CORBA

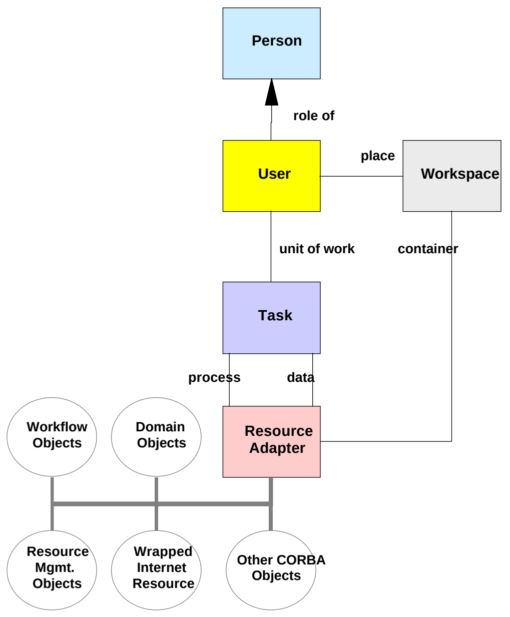

*Figure 1-1* Task and Session with Resource Objects

## 1.2 Summary of Optional versus Mandatory Interfaces

Task and Session objects define a model of systems that people interact with. For this model to be complete and consistent, all interfaces in this specification are mandatory.

## 1.3 Proposed Compliance Points

There is only one proposed compliance point, the IDL specification in this specification.

# *Task and Session Interfaces* 2

## *Contents*

This chapter contains the following sections

This specification defines cooperative components that form a framework representing the basic model for users of distributed systems.

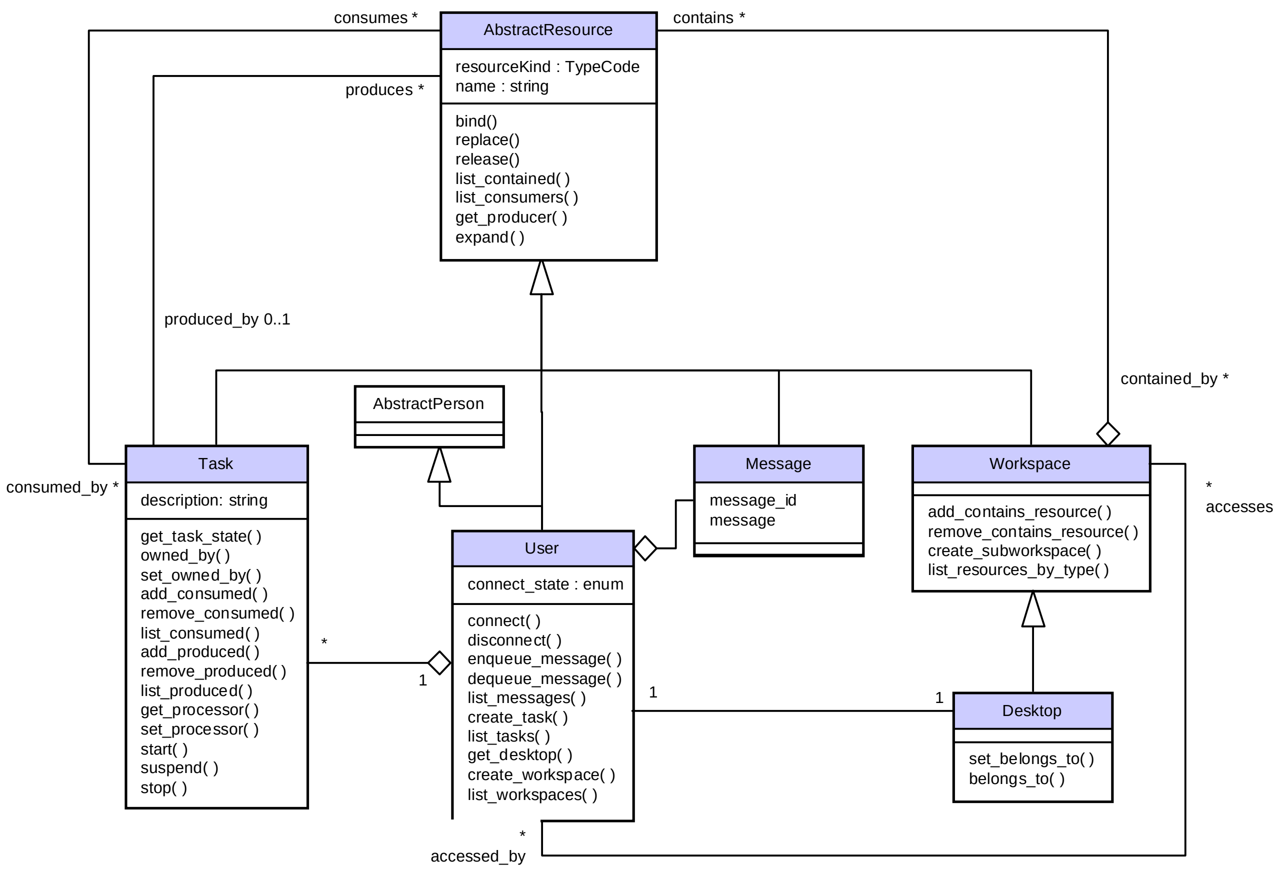

*Figure 2-1* Components

## 2.1 *IdentifiableDomainObject*

**IdentifiableDomainObject** is an abstract base type for **BaseBusinessObject** through which object identity may be managed across independently managed domain. The attribute **domain** qualifies the name space associated with the object identity provided under the **IdentifiableObject** interface. The **AuthorityId** type is a **struct** containing the declaration of a naming authority (ISO, DNS, IDL, OTHER, DCE), and a string defining the naming entity. The **same_domain** operation is a convenience operation to compare two **IdentifiableDomainObject** object instances for domain equivalence.

### 2.1.1 *IDL Specification*

```
interface IdentifiableDomainObject :
    CosObjectIdentity::IdentifiableObject
    {
```

```
    readonly attribute NamingAuthority::AuthorityId domain;
    boolean same_domain(
        in IdentifiableDomainObject other_object
    );
};
```

## 2.2 BaseBusinessObject

**BaseBusinessObject** is the abstract base class for all principal Task and Session objects. It has identity, is transactional, has a lifecycle, and is a notification supplier and consumer.

### 2.2.1 IDL Specification

```
interface BaseBusinessObject :
    Session::IdentifiableDomainObject,
    CosLifeCycle::LifeCycleObject,
    CosNotifyComm::StructuredPushSupplier,
    CosNotifyComm::StructuredPushConsumer
    {
};
```

The **CosNotification** service defines a **StructuredEvent** that provide a framework for the naming of an event and the association of specific properties to that event. All events specified within this facility conform to the **StructuredEvent** interface. This specification requires specific event types to provide the following properties as a part of the **filterable_data** of the structured event header.

Under the **CosNotification** specification all events are associated with a unique domain name space. This specification establishes the domain namespace "**org.omg.session**" for structured events associated with **AbstractResource** and its sub-types.

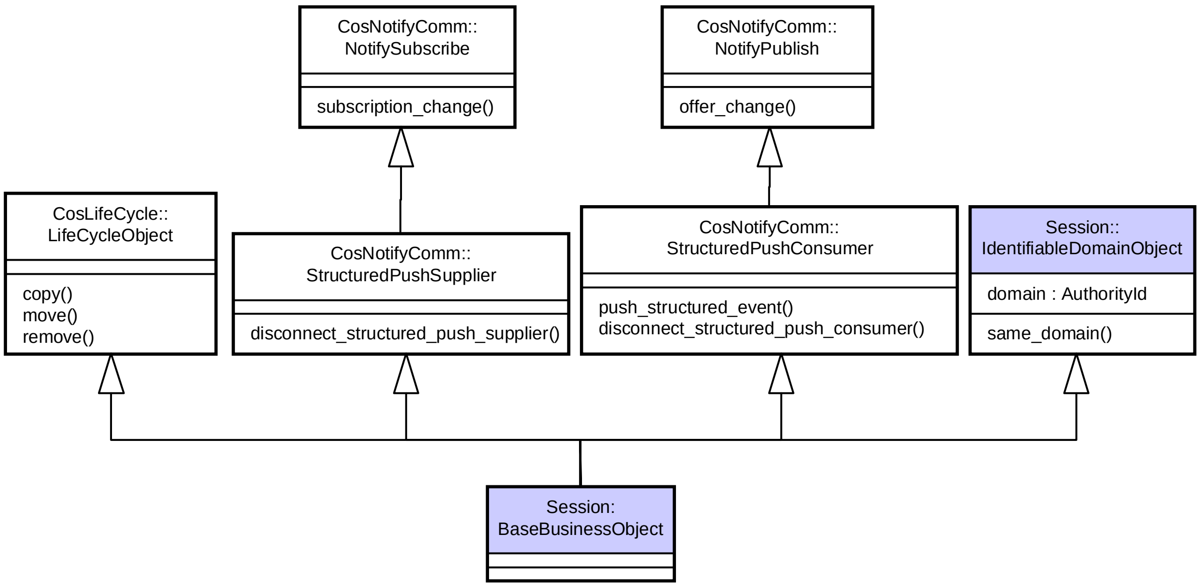
CosNotifyComm:: NotifySubscribe
subscription_change()
CosNotifyComm:: NotifyPublish
offer_change()
CosLifeCycle:: LifeCycleObject
copy()
move()
remove()
CosNotifyComm:: StructuredPushSupplier
disconnect_structured_push_supplier()
CosNotifyComm:: StructuredPushConsumer
push_structured_event()
disconnect_structured_push_consumer()
Session:: IdentifiableDomainObject
domain : AuthorityId
same_domain()
Session: BaseBusinessObject

*Figure 2-2* Base Business Object Diagram

## 2.3 Data Types

These type definitions specify user, task, message, resource, and workspace sequences.

### 2.3.1 IDL Specification

```
typedef sequence<Session::User>Users;
typedef sequence<Session::Workspace>Workspaces;
typedef sequence<Session::Task>Tasks;
typedef sequence<Session::AbstractResource>AbstractResources;
typedef sequence<Session::Message>Messages;
typedef sequence<Session::Link>Links;
```

## 2.4 Iterators

The interfaces defined below specify iterators used for the user, task, workspace, resource, and message sequences.

```
interface UserIterator : CosCollection :: Iterator { };
interface WorkspaceIterator : CosCollection :: Iterator { };
interface TaskIterator : CosCollection :: Iterator { };
interface AbstractResourceIterator : CosCollection :: Iterator { };
interface MessageIterator : CosCollection :: Iterator { };
interface LinkIterator : CosCollection :: Iterator { };
```

The core Task and Session interfaces are:

- **AbstractPerson**, defines information about people. In this model it is a placeholder for party and organization models.
- **User,** defines people as distributed computing users with messages and state as well as workspace, task, and resource associations.
- **Message**, defines basic interface for sending asynchronous messages to Users
- **Desktop**, links Users to Workspaces.
- **Workspace**, defines private and shared places for Resources and Tasks.
- **Task**, defines and manages of User units of work.
- **AbstractResource**, links resource objects to Task and Workspace objects.
- **Link**, defines a resource dependency.

## 2.5 Link

The **Link** type is a struct used within the Task and Session framework as an argument to operations that establish relationship dependencies between resources such as usage and containment. The **Link** type is used as an argument to the **bind**, **replace**, and **release** operations of an **AbstractResource** and as a type exposed under the **expand** operation.

### 2.5.1 IDL Specification

```
typedef long LinkKind;
typedef sequence<LinkKind>LinkKinds;

// reference (abstract)
const LinkKind references = 0;
const LinkKind referenced_by = 1;

// usage (abstract)
const LinkKind uses = 2;
const LinkKind used_by = 3;

// consumption
const LinkKind consumes = 8;
const LinkKind consumed_by = 9;

// production
const LinkKind produces = 10;
const LinkKind produced_by = 11;

// process
const LinkKind processes = 12;
const LinkKind processed_by = 13;
```

```
// containment
const LinkKind contains = 4;
const LinkKind contained_by = 5;

// rights (abstract)
const LinkKind holds = 6;
const LinkKind grants = 7;

// access rights
const LinkKind accesses = 14;
const LinkKind accessed_by = 15;

// adminstration rights
const LinkKind administers = 16;
const LinkKind administered_by = 17;

// ownership rights
const LinkKind owns = 18;
const LinkKind owned_by = 19;

struct Link {
      LinkKind kind;
      AbstractResource resource;
};

struct LinkExtent {
      LinkKind kind;
      AbstractResources seq;
      AbstractResourceIterator iterator;
};
```

### 2.5.2 Link Structural Features

| Name | Type | Purpose |
|---|---|---|
| kind | LinkKind | A value qualifying the kind of relationship the link represents. |
| resource | AbstractResource | The abstract resource that is the subject of the association. |

### 2.5.3 Technical Note

In the absence of a value based mechanism to express the kind of links that can exist between resources, the following hierarchy shall be assumed in evaluation of the correspondence of a link kind during the execution of the expand operation on AbstractResource. Non shaded blocks indicate abstract link kind values; whereas, shaded blocks indicate concrete link kinds.

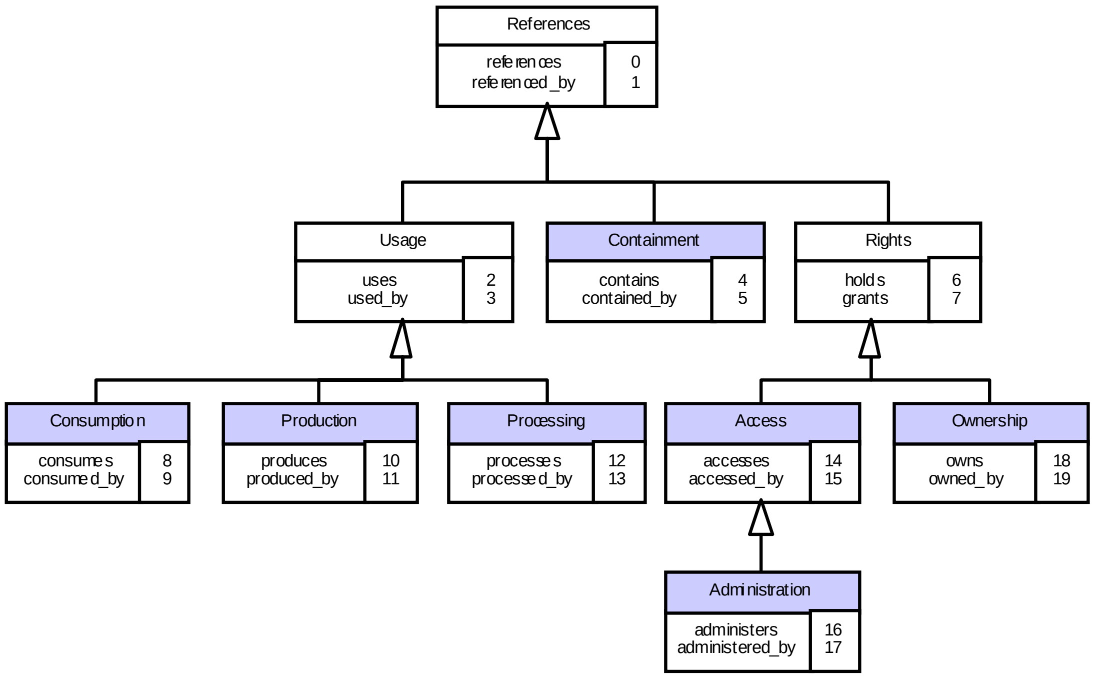

*Figure 2-3* Implicit hierarchy of LinkKind values

For example, invoking the **apply** operation on a User with an *abstract* **link_kind** of **authorizes** will result in the return of a sequence of *concrete* **LinkExtent** instances that expose the extent of the respective relationship applicable to the type of resource. Specifically, the sequence for the example query will return one **LinkExtent** referencing **owns** as the **LinkKind** (because this is the only concrete kind of link authorized by a user). A **LinkKind** of **references** would return all possible links because **references** is the most implicitly abstract kind of link.

## *2.5.4 Usage*

The Link type is a generalized utility that enables an AbstractResource, User, Task or Workspace to declare a dependency which is exposed directly under the expand operation on AbstractResource, and indirectly through related list operations.

The Link type is provided as a means through which the type and subject resource of a dependency may be declared by the resource raising the dependency to the target. Declaration of dependency between resources enables referential integrity between resources irrespective of technology or administrative domain boundaries. Declaration, modification, and retraction of dependencies are achieved through invocation of the **bind**, **release**, and **replace** operations on the AbstractResource type by a client resource.

For example, a Task may wish to register a dependency on a data resource in another domain. The Task invokes the **bind** operation on the target resource using itself as the **resource** argument, and **consumes** as the link **kind**. A subsequent query on the Task using the **expand** operation will expose a Link referencing the resource under the resource under the **consumes** link **kind** (i.e., the Task **consumes** the resource). An expand query on the target resource will return a Link referencing the Task under the reciprocal link **kind** of **consumed_by** (i.e., the resource is **consumed_by** the Task).

The following table details the **LinkKind** constants, a description of the constant value, its cardinality, and its reciprocal link kind.

*Table 2-1* LinkKind exposed by AbstractResource

| LinkKind | Description | Cardinality | Reciprocal |
|---|---|---|---|
| consumed_by | The resource argument references the Task that this resource is consumed by. | * | consumes |
| processes | The resource argument references the Task that this resource is acting as a processor to. | * | processed_by |
| produced_by | The resource argument references the Task that this resource is produced by. | 0,1 | produces |
| contained_by | References the Workspace that this resource is contained by. | * | contains |

*Table 2-2* LinkKind exposed by User

| LinkKind | Description | Cardinality | Reciprocal |
|---|---|---|---|
| owns | The resource argument references a Task owned by this User. | * | owned_by |
| accesses | References a Workspace that this User is authorized to access | * | accessed_by |
| administers | References a Workspace that this User is authorized to administer, granting the user the right to modify access lists. | * | administered_by |

*Table 2-3* LinkKind exposed by Task

| LinkKind | Description | Cardinality | Reciprocal |
|---|---|---|---|
| consumes | The resource argument references an AbstractResource that this Task is consuming. | * | consumed_by |

*Table 2-3* LinkKind exposed by Task *(Continued)*

| | | | |
|---|---|---|---|
| processed_by | The resource argument references an AbstractResource that this Task is processed by. | 0..1 | processes |
| produces | The resource argument references an AbstractResource that this Task is producing. | * | produced_by |
| owned_by | The resource argument references a User that this Task is owned by. | 1 | owns |

*Table 2-4* LinkKind exposed by Workspace

| **LinkKind** | **Description** | **Cardinality** | **Reciprocal** |
|---|---|---|---|
| contains | The resource argument references an AbstractResource that this Workspace contains. | * | contained_by |
| accessed_by | The resource argument references a User included in the access control list of this Workspace. | * | accesses |
| administered_by | The resource argument references a User included in the access control list of this Workspace, that has the right to modify the Workspace ACL. | * | administers |

## *2.6 AbstractResource*

### *2.6.1 Description*

An AbstractResource is a transactional and persistent CORBA objects contained in one or more Workspaces. They may be selected, consumed, and produced by Tasks. AbstractResources are found and selected by tools and facilities that present lists of candidate resources. These lists may be filtered by things like security credentials, by type, and by implementation. CORBAservice Trading can be used to build resource candidate lists. Resources selected from the lists are then wrapped by the tool or facility as AbstractResources. Task and Workspace are dependent on the AbstractResources they use and contain. Implementations are required to notify Task and Workspace of changes and defer deletion requests until all linked Tasks signal their readiness to handle.

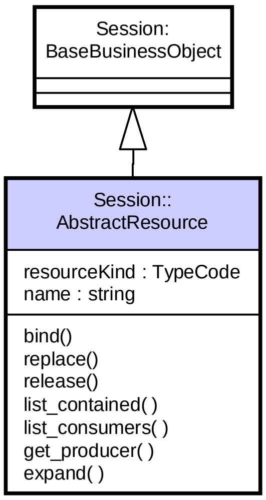

*Figure 2-4* AbstractResource Diagram

## 2.6.2 Structural Associations

*Table 2-5* LinkKind exposed by AbstractResource

| **LinkKind** | **Description** | **Cardinality** | **Reciprocal** |
|---|---|---|---|
| consumed_by | The resource argument references the Task that this resource is consumed by. | * | consumes |
| processes | The resource argument references the Task that this resource is acting as a processor to. | * | processed_by |
| produced_by | The resource argument references the Task that this resource is produced by. | 0,1 | produces |
| contained_by | The resource argument references the Workspace that this resource is contained within. | * | contained |

### *Attributes*

| **Name** | **Type** | **Purpose** |
|---|---|---|
| name | string | Resource name. |
| resourceKind | CORBA::TypeCode | The most derived type that this resource represents. |

*Table 2-6* AbstractResource Filterable Data Properties

| Name | Type | Description |
|---|---|---|
| timestamp | TimeBase::UtcT | Date and time of to which the event is issued. |
| source | AbstractResource | Abstract resource raising the event. |

### 2.6.3 Structured Events

*Table 2-7* Life Cycle Structured Event Table

| Event: | Description | | |
|---|---|---|---|
| move | Notification of the transfer (move) of a AbstractResource under which the identity is changed. The source of the event supplies the old instance identity. | | |
| | Supplementary properties: | | |
| | new | AbstractResource | Reference containing the new object identity. |
| remove | Notification of the removal of an AbstractResource | | |

### 2.6.4 Structural Features

*Table 2-8* Feature Event Table

| Event: | Description | | |
|---|---|---|---|
| update | Notification of the change of a value of an attribute from value x to value y, where x represents the old value and y represents the new value. | | |
| | Supplementary properties: | | |
| | feature | string | Attribute name. |
| | old | any | Old value. |
| | new | any | New value. |
| bind | Notification of the addition of a link from a dependant resource to this resource. | | |
| | Supplementary properties: | | |
| | link | Link | The link defining the dependency. |
| replace | Notification of the replacement of a link under this resource. | | |
| | Supplementary properties: | | |
| | old | Link | The link being replaced. |

*Table 2-8* Feature Event Table

| | | | |
|---|---|---|---|
| | new | Link | The replacement Link. |
| release | Notification of the release of a dependency link from this resource. | | |
| | Supplementary properties: | | |
| | link | Link | The link being released. |

## 2.6.5 IDL Specification

```
interface AbstractResource :
    BaseBusinessObject {
    attribute string name;
    // readonly attribute string key; /* deprecated, issue 2698 */
    readonly attribute TypeCode resourceKind;
    exception ResourceUnavailable{ };
    exception ProcessorConflict{ };
    exception SemanticConflict{ };
    void bind(
        in Link link
    ) raises (
        ResourceUnavailable,
        ProcessorConflict,
        SemanticConflict
    );
    void replace(
        in Link old,
        in Link new
    ) raises (
        ResourceUnavailable,
        ProcessorConflict,
        SemanticConflict
    );
    void release(
        in Link link
    );
    void list_contained (
        in long max_number,
        out Session::Workspaces workspaces,
        out WorkspaceIterator wsit
    );
    void list_consumers (
        in long max_number,
        out Tasks tasks,
        out TaskIterator taskit
    );
    Task get_producer(
    ); // get_producer replaces list_producers, issue 2701
    //void list_producers (
    //    in long max_number,
    //    out Tasks tasks,
    //    out TaskIterator taskit);
    void expand ( // levels argument removed
          in LinkKinds link_types, // string replaced by LinkKinds
          in long max_number,
          out LinkExtents seq, // return value updated
          out LinkExtentIterator iterator // return value updated
    );
};
```

### 2.6.6 Declaration of Dependencies

The bind, replace and release operations enable a client to declare a dependency on an AbstractResource. When a Task, User, or Workspace establishes a usage of containment dependency on an AbstractResource, it is required to invoke the **bind** operation. When dependencies are changed, such as the modification of the owner of a Task or the replacement of a resource within a workspace, an implementation is required to invoke the **replace** operation. When a relationship is retracted, as a result of the completion of a task, an implementation is required to invoke the **release** operation on resources to which it has established a dependency.

```
void bind(
      in Link link
) raises (
      ResourceUnavailable,
      ProcessorConflict,
      SemanticConflict
);
void replace(
      in Link old,
      in Link new
) raises (
      ResourceUnavailable,
      ProcessorConflict,
      SemanticConflict
);
void release(
      in Link link
);
```

Exceptions raised under the bind and replace operations include **ResourceUnavailable**, **ProducerConflict**, and **SemanticConflict**. The **ResourceUnavailable** and **ProducerConflict** exception may be raised by an implementation to indicate that the resource that is the target of a **bind** or **replace** operation is unable to fulfill the request. **ResourceUnavailable** may be raised as a result of a concurrency control conflict. The **ProducerConflict** exception may be raised in a situation where the producer resource is unable to support the association (for example, as a result of a processing capacity limit). A **SemanticConflict** exception may be raised if an attempt is made to violate the cardinality or type rules concerning the link kind referenced under the Link argument.

### 2.6.7 Workspaces

This operation returns a list of Workspaces containing this resource.

```
    void list_contained (
        in long max_number,
        out Session::Workspaces workspaces,
        out WorkspaceIterator wsit
);
```

### 2.6.8 Task Consumers

This operation returns a list of Tasks using or consuming this resource.

```
    void list_consumers (
        in long max_number,
        out Tasks tasks,
        out TaskIterator taskit
);
```

### 2.6.9 Task Producer

This operation returns the Task that produced this resource.

```
    Task get_producer( );
```

### 2.6.10 Get Resource Tree by Link Kind

This operation asks an AbstractResource to return a set of resources linked to it by a specific relationship. Objects returned are, or are created as, AbstractResources. This operation may be used by desktop managers to present object relationship graphs.

```
struct LinkExtent {
    LinkKind kind;
    AbstractResources seq;
    AbstractResourceIterator iterator;
};

typedef sequence<LinkExtent>LinkExtents;

// from AbstractResource

    void expand (
        in LinkKinds link_types,
        in long max_number,
        out LinkExtents seq,
        out LinkExtentIterator iterator
    );
```

*Table 2-9 Expand Argument list*

| **Argument** | **Description** |
|---|---|
| link_types | A sequence of LinkKind structures that defines the set of abstract of concrete link kinds that the expand operation should evaluate. |
| max_number | The maximum number of elements to be included in the seq of exposed LinkExtent instances. |
| seq | A sequence of LinkExtent structures. |
| iterator | An iterator of LinkExtent structures. |

## *2.7 AbstractPerson*

### *2.7.1 Description*

The **AbstractPerson** interface is a placeholder for organization and other models that define information about people. When **AbstractPerson** uses an organization model it obtains information about Users (role_of AbstractPerson) is including things like roles and membership within projects and organizations.

**AbstractPerson** inherits from the interface **CosPropertyService::PropertySetDef**, providing mechanisms through which implementations may attribute features to a person such as a name, address information, or history.

### *2.7.2 IDL Specification*

```
interface AbstractPerson :
    CosPropertyService::PropertySetDef {
};
```

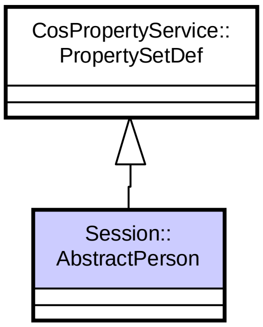

*Figure 2-5* AbstractPerson Diagram

### 2.7.3 Structured Events

*Table 2-10* AbstractPerson Structure Event Table

| **Event** | **Description** | | |
|---|---|---|---|
| property | Notification of the change in the value of a property | | |
| | Supplementary properties: | | |
| | old | Property | The old value of the property, possibly null in the case of a new property addition. |
| | new | Property | The value of the property, possibly null in the case of property deletion. |

## 2.8 User

### 2.8.1 Description

User is a role of a person in a distributed computing environment. Information about the person is inherited by User. In this specification Users have tasks and resources located in workspaces on a desktop, as well as a message queue and a connection state.

A specialization of User can add things like preferences.

### 2.8.2 IDL Specification

```
interface User  :
    AbstractResource,
    AbstractPerson,
CosLifeCycle::FactoryFinder // issue 2689
{
enum connect_state {connected, disconnected};
readonly attribute connect_state connectstate; // issue 2685
exception AlreadyConnected {};
exception NotConnected {};
void connect()
      raises (AlreadyConnected);
void disconnect()
      raises (NotConnected);
void enqueue_message (
      in Message new_message);
void dequeue_message (
      in Message message);
void list_messages(
      in long max_number,
      out Messages messages,
      out MessageIterator messageit);
Task create_task (
      // in string key, /*removed, issue 2698 */
      in string name,
      in AbstractResource process,
      in AbstractResource data);
void list_tasks (
      in long max_number,
      out Tasks tasks,
      out TaskIterator taskit
);
Desktop get_desktop ( );
Workspace create_workspace (
      // in string key, /*removed, issue 2698 */
      in string name,
      in Users accesslist
);
void list_workspaces (
      in long max_number, // issue, 2687
      out Session::Workspaces workspaces,
      out WorkspaceIterator wsit
);

};
```

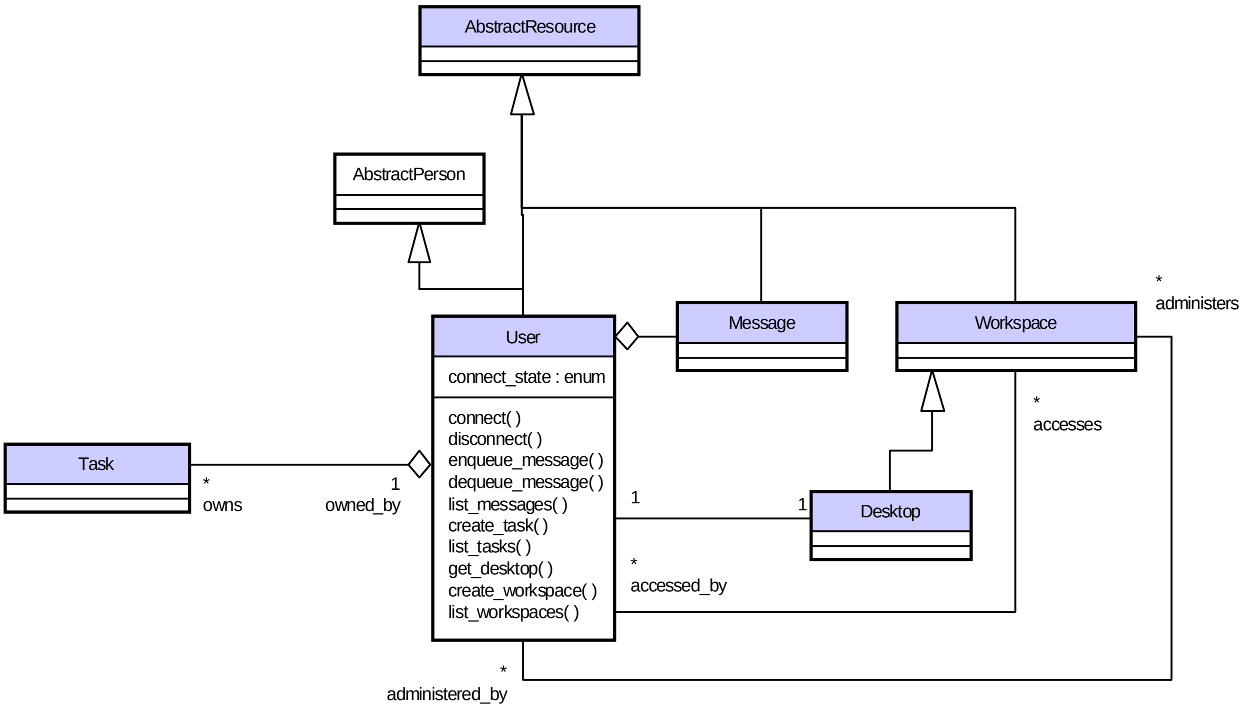

*Figure 2-6* AbstractResource - User Diagram

## 2.8.3 *Structural Features*

*Table 2-11* LinkKind exposed by User

| LinkKind | Description | Cardinality | Reciprocal |
|---|---|---|---|
| owns | The resource argument references a Task owned by this User. | * | owned_by |
| accesses | References a Workspace that this User is authorized to access | * | accessed_by |
| administers | References a Workspace that this User is authorized to administer, granting the user the right to modify access lists. | * | administered_by |

*Table 2-12* Supplementary associations

| Feature | Type | Description |
|---|---|---|
| desktop | Desktop | Link to resources and task in a distributed workspaces. |
| messages | MessageIterator | Receives asynchronous messages for this user. |

***Attributes***

| Name | Type | Purpose |
|---|---|---|
| connectstate | connect_state | Declaration of the connected state of a physical user to the system, may be one of the enumerated values **connected** or **disconnected**. |

## 2.8.4 Structured Events

*Table 2-13 User Structure Event Table*

| Event | Description | | |
|---|---|---|---|
| connected | Notification of the change of the connected state of a User. | | |
| | Supplementary properties: | | |
| | value | boolean | True indicated that the user is connected, false indicates that the user is disconnected. |

## 2.8.5 Connection State

This represents the basic current state of a User connection (logically at, or not at, the desktop). Asynchronous processes and events are managed in the disconnected state within the limitations this state imposes. When the User reconnects informational messages and actions required, if any, are presented.

Information which people expect to be retained between connections is persistent. The currency of this information must be sufficient to provide consistency over synchronous and asynchronous (including abrupt system failure) terminations.

```
enum connect_state {
connected,
disconnected
};

    readonly attribute connect_state connectstate;
```

*Table 2-14* connect_state Enumeration Table

| Value | Purpose |
|---|---|
| connected | The User is connected (logged in) to the system. |
| disconnected | The User is not connected to the system (logged off). |

## 2.8.6 Connect Operations

Connect establishes the User session for clients, such as desktop managers to present Workspaces and the computing environment. Successful completion of this operation will result in the session state being changed to connected. Clients of disconnect interact with the User and the computing environment to close the session. When complete, the session state is set to not connected.

```
exception AlreadyConnected {};
exception NotConnected {};

void connect(
) raises (
AlreadyConnected
);

void disconnect(
) raises (
NotConnected
);
```

## 2.8.7 Message Queue

These operations are used to enqueue, dequeue, and get a list of messages.

```
void enqueue_message (
in Message new_message
);

void dequeue_message (
in Message message
);

void list_messages(
in long max_number,
out Messages messages,
out MessageIterator messageit
);
```

Message factory location is achieved through invocation of the **find_factories** operation (inherited from the **CosLifeCycle::FactoryFinder** interface), and passing the **CosNaming::Name** sequence of “Factory” and “MessageFactory” as the **factory_key** argument. Client applications may choose to create a message within or external to the domain of the User to whom a message is enqueued.

**Note –** It is a recommendation of the Task and Session 1.1 RTF that the Message reviewed in the context of pass-by-value services.

### *2.8.8 User Tasks and Tasklist*

These operations are used to create and list Tasks. Task creation includes the initial specification of “who,” “what,” and “how” for the Task. The User instance of this interface is “who,” the “what” is the AbstractResource data to update or produce, and the “how” is the AbstractResource process (workflow, tool, etc.) used.

```
Task create_task (
in string name,
in AbstractResource process,
in AbstractResource data
);

void list_tasks (
in long max_number,
out Tasks tasks,
out TaskIterator taskit
);
```

### *2.8.9 Desktop Operations*

This operation returns the Desktop that links one User to many Workspaces. Workspaces may have many Users linked via Desktop.

```
Desktop get_desktop ( );
```

### *2.8.10 Workspace Operations*

Users may create and find their Workspaces with these operations. As Workspace may be shared implementations must set access control lists to the Users sequence specified with the **create** operation.

```
Workspace create_workspace (
in string name,
in Users accesslist
);

void list_workspaces (
out Workspaces workspaces,
out WorkspaceIterator wsit
);
```

## 2.9 Message

This interface defines the basic structure for messages that are enqueued to Users and dequeued for presentation by a desktop manager or used, as needed, by other clients. It is expected that Message will be specialized by implementations and user message definition standards. Typical messages include asynchronous completion, notification of Workspace content changes, and communications from other people.

Message factory location is achieved through invocation of the **find_factories** operation (inherited from the **CosLifeCycle::FactoryFinder** interface), and passing the **CosNaming::Name** sequence of “Factory” and “MessageFactory” as the **factory_key** argument. Client applications may choose to create a message within or external to the domain of the User to whom a message is enqueued.

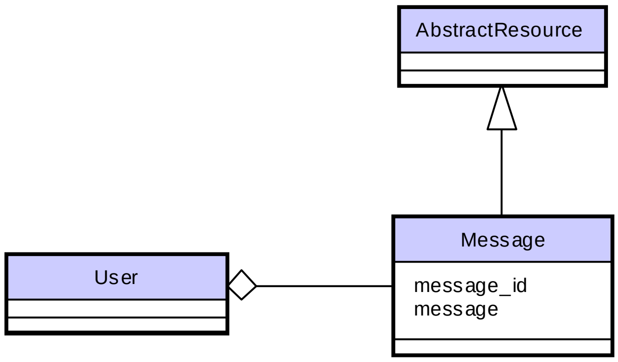

*Figure 2-7* AbstractResource - User Message Diagram

#### *Attributes*

| **Name** | **Type** | **Purpose** |
|---|---|---|
| message_id | any | Message name |
| message | any | Message description |

### 2.9.1 IDL Specification

```
interface Message : AbstractResource {
    attribute any message_id;
    attribute any message;
};
```

```
interface MessageFactory {
      Message create(
            in any message_id,
            in any message
      );
};
```

## 2.10 Desktop

### 2.10.1 Description

The Desktop interface links Users to many Workspaces and Workspaces to many Users. Each User has one Desktop and many Workspaces. Workspaces may be shared so they may have many Users.

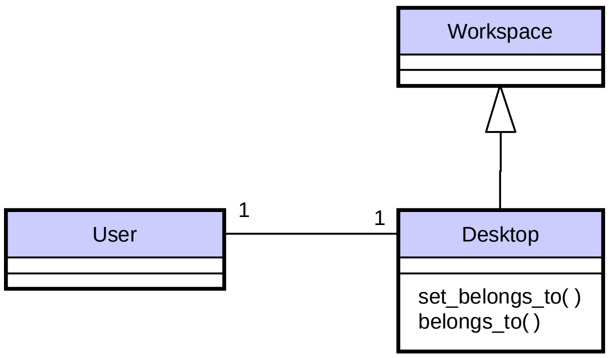

*Figure 2-8* Desktop Diagram

### 2.10.2 Structural Features

| **Keyword** | **Type** | **Description** |
|---|---|---|
| owner | User | Identify desktop owner |

### 2.10.3 IDL Specification

```
interface Desktop:Workspace {
      void set_belongs_to(
in Session::User user
);
      User belongs_to();
};
```

### 2.10.4 Ownership

These operations set and return the User that is the owner of this Desktop.

```
    void set_belongs_to(
in Session::User user
);
    User belongs_to();
```

## 2.11 Workspace

### 2.11.1 Description

Workspace defines private and shared places where resources, including Task and Session objects, may be contained. Workspaces may contain Workspaces. The support for sharing and synchronizing the use of objects available in Workspaces is provided by the objects and their managers. Each Workspace may contain any collection of private and shared objects that the objects and their managers provide access to, and control use of.

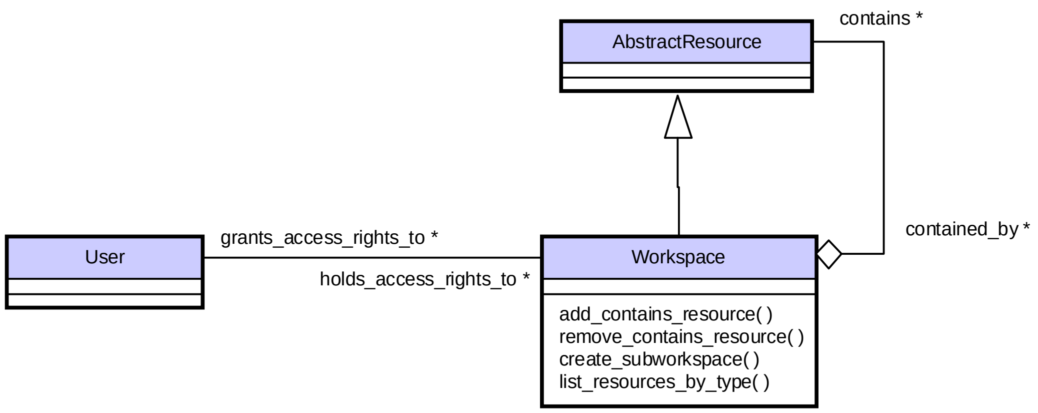

*Figure 2-9* Workspace Diagram

### 2.11.2 Structural Features

*Table 2-15* LinkKind exposed by Workspace

| LinkKind | Description | Cardinality | Reciprocal |
|---|---|---|---|
| contains | The resource argument references an AbstractResource that this Workspace contains. | * | contained_by |
| accessed_by | The resource argument references a User included in the access control list of this Workspace. | * | accesses |
| administered_by | The resource argument references a User included in the access control list of this Workspace, that has the right to modify the Workspace ACL. | * | administers |

### 2.11.3 IDL Specification

```
interface Workspace :
    AbstractResource,
    CosLifeCycle::FactoryFinder
    {
    void add_contains_resource(
        in AbstractResource resource
    );
    void remove_contains_resource(
        in AbstractResource resource
    );
    Workspace create_subworkspace (
        in string name,
        in Users accesslist
    );
    void list_resources_by_type(
        in TypeCode resourcetype,
        in long max_number,
        out AbstractResources resources,
        out AbstractResourceIterator resourceit
    );
};
```

### 2.11.4 Container Operations

These operations will add and remove AbstractResources to/from a Workspace. They will also create a new Workspace contained in this Workspace.

An implementation of **add_contains_resource** must invoke the **bind** operation on the target resource with the link kind of **contains** and the issuing Task as the resource. An implementation of **remove_contains_resource** must invoke the **release** operation on the target resource with the link kind of **contains** and the issuing Task as the resource. An implementation of **create_subworkspace** must invoke the **bind** operation on newly created workspace using the **contains** link kind and the parent workspace as the resource argument.

```
void add_contains_resource(
    in AbstractResource resource
);
void remove_contains_resource(
    in AbstractResource resource
);

Workspace create_subworkspace (
    in string name,
    in Users accesslist
);
```

### 2.11.5 List Resources

The **list resources** operation will return a list of all Workspace resources by type. This facilitates organization of resource types by user interfaces and use by task creation, workflow, and other functions requiring specified types.

```
void list_resources_by_type(
    in TypeCode resourcetype,
    in long max_number,
    out AbstractResources resources,
    out AbstractResourceIterator resourceit
);
```

### 2.11.6 Administration

On creation of a new workspace (through either the **create_workspace** or **create_subworkspace** operations), the principal User creating the new instance is implicitly associated with the workspace as administrator. As administrator, the User holds rights enabling the modification of the access control list through **bind**, **replace**, and **release** operations.

## 2.12 Task

### 2.12.1 Description

A Task represents a unit of work defined by users. It represents the binding (which can be dynamic) between the data to be processed, the method for processing, and the User responsible. The duration of a Task can range from a short, single, non-repeatable process step to long process steps that can be repeated many times. A Task can be completed with the execution of a single tool, or the execution of a flow. Tasks may be suspended to the extent that the tool or flow used to execute them can be suspended. Tasks can serve as the repository of execution history information.

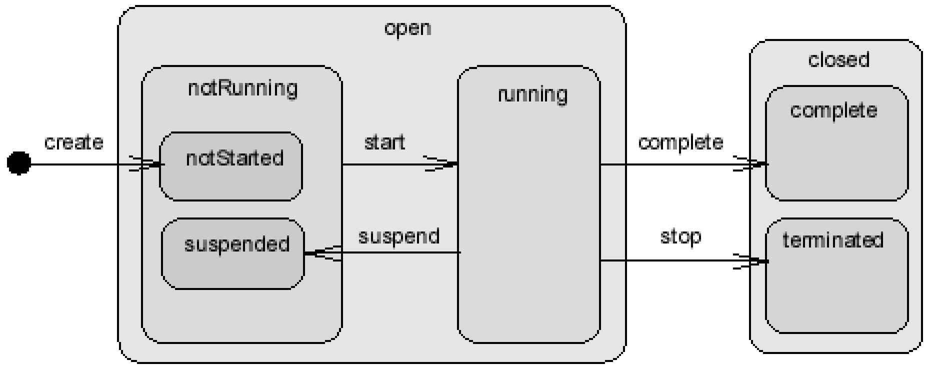

*Figure 2-10* Task State Diagram

The task state is determined by the state of its execution and the state of the data content being processed. The task state and data state are related but independent. The data state contains information about the application or system object. The task state contains information about the task. For example, when a fault simulator completes execution it is not necessarily true that the fault simulation task has completed – the completeness depends on the value of the fault coverage. The value of the fault coverage is based on the data, what has been called the “data state.” The execution of the fault simulator is independent of the results of the execution. Also the fault coverage may be changed, independent of the fault simulator, if the parameters of the design are changed.

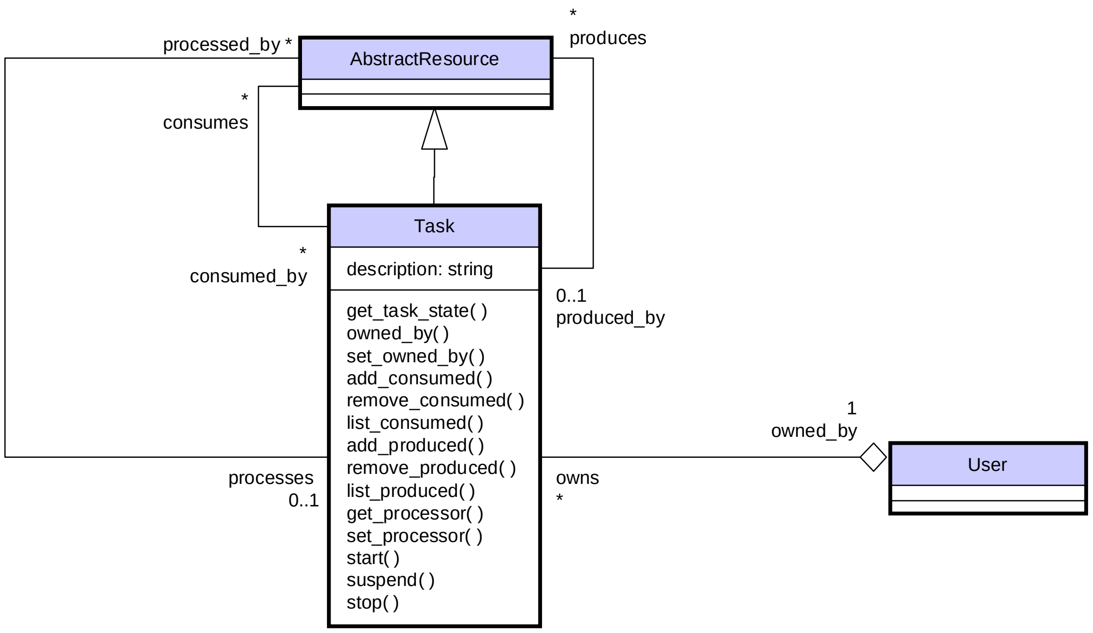

*Figure 2-11* Task Diagram

## 2.12.2 *Structural Features*

*Table 2-16* LinkKind exposed by Task

| LinkKind | Description | Cardinality | Reciprocal |
|---|---|---|---|
| consumes | The resource argument references an AbstractResource that this Task is consuming. | * | consumed_by |
| processed_by | The resource argument references an AbstractResource that this Task is processed by. | 0..1 | processes |
| produces | The resource argument references an AbstractResource that this Task is producing. | * | produced_by |
| owned_by | The resource argument references a User that this Task is owned by. | 1 | owns |

## 2.12.3 *IDL Specification*

```
interface Task : AbstractResource {
    exception CannotStart {};
    exception AlreadyRunning {};
    exception CannotSuspend {};
    exception CurrentlySuspended {};
    exception CannotStop {};
    exception NotRunning {};
    attribute string description;
    enum task_state {
        open, not_running, notstarted, running,
        suspended, terminated, completed, closed
    };
    task_state get_task_state();
    User owned_by();
    void set_owned_by (
        in User new_task_owner
    );
    void add_consumed(
        in AbstractResource resource
    );
    void remove_consumed(
        in AbstractResource resource
    );
    void list_consumed (
        in long max_number,
        out AbstractResources resources,
        out AbstractResourceIterator resourceit
    );
    void add_produced(
        in AbstractResource resource);
    void remove_produced(
        in AbstractResource resource
    );
    void list_produced (
        in long max_number,
        out AbstractResources resources,
        out AbstractResourceIterator resourceit
    );
    void set_processor(
        in AbstractResource processor
    ) raises (
        ProcessorConflict
    );
    AbstractResource get_processor( );
    void start ( ) raises (CannotStart, AlreadyRunning);
    void suspend ( ) raises (CannotSuspend, CurrentlySuspended);
        void stop ( ) raises (CannotStop, NotRunning);
};
```

### *Attributes*

| Name | Type | Purpose |
|---|---|---|
| description | string | Task description. |

## 2.12.4 Structured Events

*Table 2-17 Task Structured Event Table*

| Event | Description | | |
|---|---|---|---|
| process_state | Notification of the change of state of a **Task**. | | |
| | Supplementary properties: | | |
| | value | task_state | Task state enumeration. |
| ownership | Notification of the change of ownership of a **Task**. | | |
| | Supplementary properties: | | |
| | owner | User | **User** assigned as owner of the **Task**. |

## 2.12.5 Task Description

This attribute can describe the Task.

**attribute string description;**

## 2.12.6 Task State

States are defined, as in Figure 2-10 on page 2-27, and the **get_task_state** operation returns the current Task state, as calculated by the implementation.

*Table 2-18 task_state Enumeration Table*

| Value | Description |
|---|---|
| open | Task is not finished and active. |
| closed | Task is finished and inactive. |
| not_running | Task is active and quiescent, but ready to execute. |
| running | Task is active and executing. |
| notstarted | Task is active and ready to be initialized and started. |
| suspended | Task is active, has been started and suspended. |
| completed | Task is finished and completed normally. |
| terminated | Task finished and stopped before normal completion. |

```
enum task_state {
open,
not_running,
notstarted,
running,
suspended,
terminated,
completed,
closed
};

task_state get_task_state(
);
```

### 2.12.7 Task Owner

These operations will return the User that owns the Task and reassign a Task to another User. Task reassignment is a process that requires an agreed protocol for one User to transfer a Task to another User. Authority to reassign belongs to the User or authorized agents and managers, such as workflow and project management systems.

```
User owned_by(
);
void set_owned_by (
in User new_task_owner
);
```

### 2.12.8 Resource Usage

These operations add, remove, and list execution and information resources consumed by the Task.

An implementation of **add_consumed** must invoke the **bind** operation on the target resource with the link kind of **consumes** and the issuing Task as the resource. An implementation of **remove_consumes** must invoke the **release** operation on the target resource with the link kind of **consumes** and the issuing Task as the resource.

```
void add_consumed(
    in AbstractResource resource
);

void remove_consumed(
    in AbstractResource resource
);

void list_consumed (
    in long max_number,
    out AbstractResources resources,
    out AbstractResourceIterator resourceit
};
```

### 2.12.9 Resource Production

These operations create, modify, and list associations to resources produced by the Task.

An implementation of **add_produced** must invoke the **bind** operation on the target resource with the link kind of **produces** and the issuing Task as the resource. An implementation of **remove_produces** must invoke the **release** operation on the target resource with the link kind of **produces** and the issuing Task as the resource.

```
void add_produced(
    in AbstractResource resource);

void remove_produced( in AbstractResource resource );

void list_produced (
    in long max_number,
    out AbstractResources resources,
    out AbstractResourceIterator resourceit
);
```

### 2.12.10 Task Execution

These operations start/resume, stop, or suspend Tasks. Successful completion of start Task will set its state to Running. Completion of stop Task will set the state to Terminated. Successful completion of suspend Task will result in a Suspended state.

```
void start (
) raises (
CannotStart,
AlreadyRunning
);

void suspend (
) raises (
CannotSuspend,
CurrentlySuspended
);

void stop (
) raises (
CannotStop,
NotRunning
);
```

A Task is associated to an AbstractResource that acts as the processor for the Task. The protocol of interaction between a Task and its processing resource is implementation dependent. An example of a processor resource is an application editor, simulation, or workflow engine.

The following operations set the AbstractResource acting in this capability.

```
void set_processor(
    in AbstractResource processor
) raises (
    ProcessorConflict
);
AbstractResource get_processor( );
```

The **set_processor** operation may raise the **ProcessorConflict** exception if the **AbstractResource** passed under the processor argument is unable or unwilling to provide processing services to the task.

An implementation of **set_process** must ensure that appropriate bind and release operations are invoked on the processor resources in order to ensure referential integrity. When a task is initially created, the implementation is responsible for invocation of the **bind** operation on the abstract resource that is serving as the processor, using the **processed_by** link kind. Subsequent invocations of **set_processor** are responsible for the releasing and re-establishing **processed_by** links on the old and new process resource using the **release** and **bind** operation on the respective process resources.

# Complete IDL A

## A.1 Full IDL Listing

```
// Task and Session RTF V2.0 of Session.idl

#ifndef _SESSION_
#define _SESSION_

//#include <CosTransactions.idl> // retracted 2680
#include <CosLifeCycle.idl>
#include <CosObjectIdentity.idl>
#include <CosCollection.idl>
#include <NamingAuthority.idl> // added 2680
#include <CosNotifyComm.idl> // added 2692
#include <CosPropertyService.idl> // added 2682

#pragma prefix "omg.org"
#pragma javaPackage "org.omg"

module Session {

    interface AbstractResource;
    interface Task;
    interface Workspace;
    interface AbstractPerson;
    interface User;
    interface Message;
    interface Desktop;

    // sequence defintions

    typedef sequence<Session::AbstractResource>AbstractResources;
    typedef sequence<Session::Task>Tasks;
    typedef sequence<Session::Message>Messages;
    typedef sequence<Session::User>Users;
    typedef sequence<Session::Workspace>Workspaces;
```

```
// The following two types have been deprecated by inclusion of
// CosNotification under issue 2692

// enum event_type {ProcessEvent, DataEvent};
// struct ResourceEvent {event_type type; any eventdata;};

// the following types have been introduced as a replacement to the
// usage and containment interface defintions and address issues
// 2681, 2683, 2687, 2688, 2693, 2698, 2699 2700 and 2701.

typedef long LinkKind;
typedef sequence<LinkKind>LinkKinds;

// reference (abstract)
const LinkKind references = 0;
const LinkKind referenced_by = 1;

// usage (abstract)
const LinkKind uses = 2;
const LinkKind used_by = 3;

// consumption
const LinkKind consumes = 8;
const LinkKind consumed_by = 9;

// production
const LinkKind produces = 10;
const LinkKind produced_by = 11;

// process
const LinkKind processes = 12;
const LinkKind processed_by = 13;

// containment
const LinkKind contains = 4;
const LinkKind contained_by = 5;

// rights (abstract)
const LinkKind holds = 6;
const LinkKind grants = 7;

// access rights
const LinkKind accesses = 14;
const LinkKind accessed_by = 15;

// adminstration rights
const LinkKind administers = 16;
const LinkKind administered_by = 17;

// ownership rights
const LinkKind owns = 18;
const LinkKind owned_by = 19;

struct Link {
        LinkKind kind;
        AbstractResource resource;
};

typedef sequence<Session::Link>Links;

interface AbstractResourceIterator : CosCollection :: Iterator { };
interface TaskIterator : CosCollection :: Iterator { };
interface MessageIterator : CosCollection :: Iterator { };
interface WorkspaceIterator : CosCollection :: Iterator { };
interface UserIterator : CosCollection :: Iterator { };

struct LinkExtent {
        LinkKind kind;
        AbstractResources seq;
        AbstractResourceIterator iterator;
};

typedef sequence<LinkExtent>LinkExtents;

interface LinkExtentIterator : CosCollection :: Iterator { };

// Defintion of BaseBusinessObject has been enhanced to include explicit
// declaration of the domain under which an object identity is scoped.
// Issue 2680 also calls for the retraction of the TransactionalObject
// from the BaseBusinessObject inheritance graph.

interface IdentifiableDomainObject :
        CosObjectIdentity::IdentifiableObject
        {
        readonly attribute NamingAuthority::AuthorityId domain;
        boolean same_domain(
                in IdentifiableDomainObject other_object
        );
};

// Resolution of BaseBusinessObject event semantics under issue 2692

interface BaseBusinessObject :
        Session::IdentifiableDomainObject,
        CosLifeCycle::LifeCycleObject,
        CosNotifyComm::StructuredPushSupplier,
        CosNotifyComm::StructuredPushConsumer
        {
};

// AbstractResource has been enhanced as a result of a number of
// issues that directly or indirectly effect this base type.  These
// issues include 2681, 2683, 2687, 2688, 2693, 2698, 2699 2700 and 2701.

interface AbstractResource :
        BaseBusinessObject {
        // exception NotRemoved {}; /* already defined under LifeCycleObject */
        attribute string name;
        // readonly attribute string key; /* deprecated, issue 2698 */
        readonly attribute TypeCode resourceKind;
        exception ResourceUnavailable{ };
        exception ProcessorConflict{ };
        exception SemanticConflict{ };
        void bind(
            in Link link
        ) raises (
            ResourceUnavailable,
            ProcessorConflict,
            SemanticConflict
        );
        void replace(
            in Link old,
            in Link new
        ) raises (
            ResourceUnavailable,
            ProcessorConflict,
            SemanticConflict
        );
        void release(
            in Link link
        );
        void list_contained (
            in long max_number,
            out Session::Workspaces workspaces,
            out WorkspaceIterator wsit
        );
        void list_consumers (
            in long max_number,
            out Tasks tasks,
            out TaskIterator taskit
        );
        Task get_producer(
        ); // get_producer replaces list_producers, issue 2701
        //void list_producers (
        //  in long max_number,
        //  out Tasks tasks,
        //  out TaskIterator taskit);
        void expand ( // levels argument removed
            in LinkKinds link_types, // argument type string replaced by LinkKinds
            in long max_number,
            out LinkExtents seq, // return value updated
            out LinkExtentIterator iterator // return value updated
        );
        /* remove operation depricated, issue 2702 */
        //void remove ( ) raises(NotRemoved);
    };

    interface AbstractPerson :
        CosPropertyService::PropertySetDef // issue 2682
        {
    };

    interface User  :
        AbstractResource,
		AbstractPerson,
		CosLifeCycle::FactoryFinder // issue 2689
		{
		enum connect_state {connected, disconnected};
		readonly attribute connect_state connectstate; // issue 2685
		exception AlreadyConnected {};
		exception NotConnected {};
		void connect()
			raises (AlreadyConnected);
		void disconnect()
			raises (NotConnected);
		void enqueue_message (
			in Message new_message);
		void dequeue_message (
			in Message message);
		void list_messages(
			in long max_number,
			out Messages messages,
			out MessageIterator messageit);
		Task create_task (
			// in string key, /*removed, issue 2698 */
			in string name,
			in AbstractResource process,
			in AbstractResource data);
		void list_tasks (
			in long max_number,
			out Tasks tasks,
			out TaskIterator taskit
		);
		Desktop get_desktop ( );
		Workspace create_workspace (
			// in string key, /*removed, issue 2698 */
			in string name,
			in Users accesslist
		);
		void list_workspaces (
			in long max_number, // issue, 2687
			out Session::Workspaces workspaces,
			out WorkspaceIterator wsit
		);

	};

	interface Message : AbstractResource {
		attribute any message_id;
		attribute any message;
	};

	// Issue 2689
	// The following factory interface is defined as the means through which
instances
	// of message may be created within an appropriate administrative domain.

	interface MessageFactory{
		Message create(
                in any message_id,
                in any message
            );
        };

        interface Workspace :
            AbstractResource,
            CosLifeCycle::FactoryFinder // issue 2689 - target for copy and move
            {
            void add_contains_resource(
                in AbstractResource resource
            );
            void remove_contains_resource(
                in AbstractResource resource
            );
            Workspace create_subworkspace (
                // in string key, /*removed, issue 2698 */
                in string name,
                in Users accesslist
            );
            void list_resources_by_type(
                in TypeCode resourcetype, // issue 2691
                in long max_number,
                out AbstractResources resources,
                out AbstractResourceIterator resourceit
            );
            // oneway void resource_event (in ResourceEvent event);
    /* depricated, issue 2692 */
        };

        interface Desktop:Workspace {
            void set_belongs_to(
                in User user
            );
            User belongs_to();
        };

        interface Task : AbstractResource {
            exception CannotStart {};
            exception AlreadyRunning {};
            exception CannotSuspend {};
            exception CurrentlySuspended {};
            exception CannotStop {};
            exception NotRunning {};
            attribute string description;
            enum task_state {
                open, not_running, notstarted, running,
                suspended, terminated, completed, closed
            };
            task_state get_task_state();
            User owned_by();
            void set_owned_by (
                in User new_task_owner
            );
            void add_consumed(  // issue 2695, was add_consumer
                in AbstractResource resource
            );
            void remove_consumed( // issue 2695, was remove_consumer
                in AbstractResource resource
            );
            void list_consumed (
                in long max_number,
                out AbstractResources resources,
                out AbstractResourceIterator resourceit
            );
            void add_produced( // issue 2695, was add_producer
                in AbstractResource resource);
            void remove_produced( // issue 2696
                in AbstractResource resource);
            void list_produced (
                in long max_number, // 2687
                out AbstractResources resources,
                out AbstractResourceIterator resourceit
            );
            AbstractResource get_processor( ); // issue 2688
            void set_processor( // issue 2688
                in AbstractResource processor
            ) raises (
                ProcessorConflict
            );
            /* resource_event depricated, issue 2692 */
            // oneway void resource_event (in ResourceEvent event);
            /* list_history depricated, issue 2692 */
            // CosCollection::Iterator list_history (in string filter);
            void start ( ) raises (CannotStart, AlreadyRunning);
            void suspend ( ) raises (CannotSuspend, CurrentlySuspended);
            void stop ( ) raises (CannotStop, NotRunning);
        };

        // The following two interfaces are deprecated in line with the inclusion of the
        // Link type and supporting operations.
        // interface Containment : BaseBusinessObject {};
        // interface Usage : BaseBusinessObject {};

    };

    #endif /* _SESSION_ */
```

# Conformance B

## B.1 Proposed Compliance Points

There is only one proposed compliance point, the IDL specification in this specification.

## B.2 Summary of Optional versus Mandatory Interfaces

Task and Session objects define a model of systems that people interact with. For this model to be complete and consistent all interfaces in this specification are mandatory.