

13.1 Overview

13.1.1 Service Description

An object supports an interface. An interface consists of operations and attributes. The interface is statically defined in OMG IDL. Two objects are of the same type if they support the same interface.

Properties are typed, named values dynamically associated with an object, outside of the type system. There are many useful cases for properties. For example:

- **Object Classification** -- A particular document may be classified as important; it must be read by the end of the day. Another document is marginally important; it must be read by the end of the month. Yet another document is not marked important. The classification of the document was invented by the user. It is not part of the document's type. However, a user may use a standard utility to find all documents marked important.
- **Object Usage Count** -- An on-line service download utility increments a counter every time an object has been downloaded by a user. The information is associated with the object but it is not part of the object's type.

The property service implements objects supporting the *PropertySet* interface or the *PropertySetDef* interface. The *PropertySet* interface supports a set of properties. A property is two tuple of: <property_name , property_value>. property_name is a string that names the property. property_value is of type *any* and carries the value assigned to the property.

The *PropertySetDef* interface is a specialization (subclass) of the *PropertySet* interface that exposes the characteristics (or metadata) of each property (e.g. readonly or read/write access). In general, this specification will use the term *PropertySet* to refer to the collection of properties and will only use the term *PropertySetDef* when explicitly referring to operations related to property metadata.

The association of properties with an object is considered an implementation detail. This property service specification allows for the creation of *PropertySets* or *PropertySetDefs* via factory interfaces, or an object may inherit the *PropertySet* or *PropertySetDef* interfaces.

Client's Model of Properties

As with CORBA attributes, clients can get and set property values. However, with properties, clients can also dynamically create and delete properties associated with an object. Clients can manipulate properties individually or in batches using a sequence of the *Property* data type called *Properties*.

In addition, when using objects that support the *PropertySetDef* interface, clients can create and manipulate properties and their characteristics, such as the property mode. The *PropertySetDef* interface also provides operations for clients to retrieve *constraint* information about a *PropertySet*, such as allowed property types.

To aid in the client's view of properties associated with an object, the client may request a list of property names (*PropertyNames*) or the number of properties.

Iterators are used by the property service to return lists of properties when the number of properties exceeds that which is expected by the client. Iterators contain operations that allow clients fine-grained control over the enumeration of properties.

Object's Model of Properties

Every object that wishes to provide a property service must support either the *PropertySet* or *PropertySetDef* interface. *PropertySet* is the interface that provides operations for defining, deleting, enumerating and checking for the existence of

properties. The *PropertySetDef* interface is a subclass of *PropertySet* that provides operations to retrieve *PropertySet* constraints, define and modify properties with modes, and to get and set property modes.

Subclasses of *PropertySet* or *PropertySetDef* may impose restrictions on some or all of the properties they store.

Properties are intended to be the dynamic equivalent of CORBA attributes. As such, the *PropertySet* interface provides exceptions to allow implementors to support the concepts of a *readonly* property and a *fixed* property (i.e., a property that cannot be deleted). In addition, the *PropertySetDef* interface provides operations for implementors to declare their *PropertySet* constraints to clients. This mechanism is for those implementations that need the dynamics of properties, yet want the interface control of CORBA attributes.

A *PropertySet* object may support the storage of property data types itself, or there may be a “generic” *PropertySet* implementation that handles the parsing of property data types and the memory management associated with storing properties. This is considered an implementation detail.

When a *PropertySet* object receives a `define_property` request from a client, it must ensure there are no `property_name` conflicts and then retain the property information such that the object can later respond to `get_property`, `delete_property`, and `is_property_defined` requests from clients.

When a *PropertySet* object receives a `define_property` request to an existing property from a client, it must ensure that the *any* TypeCode of the `property_value` of the request matches the existing property’s *any* TypeCode.

Use of property modes within a *PropertySet* is an implementation issue, as clients can neither access nor modify a property mode. For example, an implementation may define some initial readonly properties at create time and raise the `ReadOnlyProperty` exception if a client attempts to define a new property value.

13.1.2 OMG IDL Interface Summary

The property service defines interfaces to support functionality described in the previous sections. The following table gives a high-level description of the property service interfaces.

Table 13-1 Property Service Interfaces

Interface	Purpose
PropertySet	Supports operations for defining, deleting, enumerating and checking for the existence of properties.
PropertySetDef	Supports operations for retrieving PropertySet constraints and getting and setting property modes.
PropertiesIterator	Supports operations to allow clients fine-grained control over the enumeration of properties.
PropertyNamesIterator	Supports operations to allow clients fine-grained control over the enumeration of property names.
PropertySetFactory	Creates PropertySets.
PropertySetDefFactory	Creates PropertySetDefs.

13.1.3 Summary of Key Features

The following are key features of the Property Service:

- Provides the ability to dynamically associate named values with objects outside the static IDL-type system.
- Defines operations to create and manipulate sets of name-value pairs or name-value-mode tuples.

The names are simple OMG IDL strings. The values are OMG IDL *any*s. The use of type *any* is significant in that it allows a property service implementation to deal with any value that can be represented in the OMG IDL-type system. The modes are similar to those defined in the *Interface Repository AttributeDef* interface.

- Designed to be a basic building block, yet robust enough to be applicable for a broad set of applications.
- Provides “batch” operations to deal with sets of properties as a whole.

The use of “batch” operations is significant in that the systems and network management (SNMP, CMIP, ...) communities have proven such a need when dealing with “attribute” manipulation in a distributed environment.

- Provides exceptions such that *PropertySet* implementors may exercise control of (or apply constraints to) the names and types of properties associated with an object, similar in nature to the control one would have with CORBA attributes.
- Allows *PropertySet* implementors to restrict modification, addition and/or deletion of properties (readonly, fixed) similar in nature to the restrictions one would have with CORBA attributes.
- Provides client access and control of constraints and property modes.
- Does not rely on any other object services.

13.2 Service Interfaces

13.2.1 CosPropertyService Module

The CosPropertyService module defines the entire property service, which consists of data types, exceptions and the following interfaces:

- *PropertySet*
- *PropertySetDef*
- *PropertySetFactory*
- *PropertySetDefFactory*
- *PropertiesIterator*
- *PropertyNamesIterator*

Data Types

The CosPropertyService module provides a number of structure and sequence data types to manipulate *PropertySet* and *PropertySetDef* information.

```

/*****
/*  Data Types
*****/

typedef string PropertyName;
struct Property {
    PropertyName property_name;
    any property_value;
};

enum PropertyModeType {
    normal,
    read_only,
    fixed_normal,
    fixed_readonly,
    undefined
};

struct PropertyDef {
    PropertyName property_name;
    any property_value;
    PropertyModeType property_mode;
};

struct PropertyMode {
    PropertyName property_name;
    PropertyModeType property_mode;
};

typedef sequence<PropertyName> PropertyNames;
typedef sequence<Property> Properties;
typedef sequence<PropertyDef> PropertyDefs;
typedef sequence<PropertyMode> PropertyModes;
typedef sequence<TypeCode> PropertyTypes;

```

Figure 13-1 Data types.

A `property` is a two tuple of: `<property_name, property_value>`. `property_name` is a string, which names the property. `property_value` is of type *any* and carries the value assigned to the property. This data type is considered the base type for dealing with property data and is used throughout the *PropertySet* interface.

Clients can manipulate properties individually or in batches using a sequence of the `Property` data type called `Properties` or, when appropriate, a sequence of the `PropertyName` data type called `PropertyNames`.

A `PropertyDef` is a three tuple of: `<property_name, property_value, property_mode_type>`. `property_name` is a string, which names the property. `property_value` is of type *any* and carries the value assigned to the property. `property_mode_type` is an enumeration that defines the characteristics of the property. A property definition combines property characteristics (metadata) and property data information and is used in the *PropertySetDefFactory* and *PropertySetDef* interfaces. The `PropertyDef` data type provides clients access and control of property metadata.

Clients can manipulate property definitions individually or in batches using a sequence of the `PropertyDef` data type called `PropertyDefs`.

A `PropertyMode` is a two tuple of: `<property_name, property_mode_type>`. `property_name` is a string, which names the property. `property_mode_type` is an enumeration that defines the characteristics of the property. The `PropertyMode` data type is used in the *PropertySetDef* interface and provides clients access and control of property metadata.

Clients can manipulate property modes individually or in batches using a sequence of the `PropertyMode` data type called `PropertyModes`.

There are five mutually exclusive property mode types defined:

- **Normal** means there are no restrictions to the property. A client may define new values to an existing property or delete this property.
- **Readonly** means clients can only get the property information. However, a readonly property may be deleted.
- **Fixed_Normal** means the property cannot be deleted. However, clients are free to define new values to an existing property.
- **Fixed_Readonly** means the property cannot be deleted and clients can only get the property information.

- **Undefined** is used to signify `PropertyNotFound` when requesting a multiple get mode request. Using this on an operation that sets the mode of a property (e.g. `set_mode` or `define_property_with_mode`) will raise the `UnsupportedMode` exception.

Restrictions on the `property_mode_type` field is an implementation issue. For example, a *PropertySetDef* implementation may choose to not support a client setting a property to the `fixed_readonly` mode.

Exceptions

The *PropertySet* interface supports the following exceptions.

```
/* *****  
/* Exceptions  
/* *****  
exception ConstraintNotSupported{};  
exception InvalidPropertyName {};  
exception ConflictingProperty {};  
exception PropertyNotFound {};  
exception UnsupportedTypeCode {};  
exception UnsupportedProperty {};  
exception UnsupportedMode {};  
exception FixedProperty {};  
exception ReadOnlyProperty {};  
  
enum ExceptionReason {  
    invalid_property_name,  
    conflicting_property,  
    property_not_found,  
    unsupported_type_code,  
    unsupported_property,  
    unsupported_mode,  
    fixed_property,  
    read_only_property  
};  
  
struct PropertyException {  
    ExceptionReason reason;  
    PropertyName failing_property_name;  
};  
  
typedef sequence<PropertyException> PropertyExceptions;  
  
exception MultipleExceptions {  
    PropertyExceptions exceptions;  
};
```

Figure 13-2 PropertySet interface exceptions.

- **ConstraintNotSupported**

Indicates that either the `allowed_property_types`, `allowed_properties`, or `allowed_property_defs` parameter could not be properly supported by this *PropertySet* or *PropertySetDef*.

- **InvalidPropertyName**

Indicates that the supplied `property_name` is not valid. For example, a `property_name` of length 0 is invalid. Implementations may place other restrictions on `property_name`.

- **ConflictingProperty**

Indicates that the user is trying to modify an existing `property_name` with an *any* `TypeCode` in a `property_value` that is different from the current.

- **PropertyNotFound**

Indicates that the supplied `property_name` is not in the *PropertySet*.

- **UnsupportedTypeCode**

Indicates that a user is trying to define a property having an *any* `TypeCode` that is not supported by this *PropertySet*.

- **UnsupportedProperty**

Indicates that a user is trying to define a property not supported by this *PropertySet*.

- **FixedProperty**

Indicates that a user is trying to delete a property that the *PropertySet* considers undeletable.

- **ReadOnlyProperty**

This indicates that a user is trying to modify a property that the *PropertySet* considers to be *readonly*.

- **MultipleExceptions**

This exception is used to return a sequence of exceptions when dealing with the “batch” operations of `define_properties` and `delete_all_properties` in the *PropertySet* interface, `define_properties_with_modes` and `set_property_modes` in the *PropertySetDef* interface, `create_initial_propertyset` in the *PropertySetFactory* interface, and `create_initial_propertysetdef` in the *PropertySetDefFactory* interface. Each operation defines the valid entries that may occur in the sequence.

A `PropertyException` is a two tuple of: `<reason, failing_property_name>`. `reason` is an enumeration reflecting one of the exceptions defined above. `failing_property_name` is a string, which names the property. The sequence of property exceptions returned as `MultipleExceptions` is the `PropertyExceptions` data type.

13.2.2 *PropertySet Interface*

The *PropertySet* interface provides operations to define and modify properties, list and get properties, and delete properties.

The *PropertySet* interface also provides “batch” operations, such as `define_properties`, to deal with sets of properties as a whole. The execution of the “batch” operations is considered best effort (i.e., not an atomic set) in that not all suboperations need succeed for any suboperation to succeed.

For `define_properties` and `delete_properties`, if any suboperation fails, a `MultipleExceptions` exception is returned to identify which property name had which exception.

For example, a client may invoke `define_properties` using three property structures. The first property could be accepted (added or modified), the second could fail due to an `InvalidPropertyName`, and the third could fail due to a `ConflictingProperty`. In this case a property is either added or modified in the *PropertySet*, and a `MultipleExceptions` is raised with two items in the `PropertyExceptions` sequence.

The `get_properties` and `delete_all_properties` “batch” operations utilize a boolean flag to identify that mixed results occurred and additional processing may be required to fully analyze the exceptions.

Making “batch” operations behave in an atomic manner is considered an implementation issue that could be accomplished via specialization of this property service.

Defining and Modifying Properties

This set of operations is used to define new properties to a *PropertySet* or set new values on existing properties.

```
/* Support for defining and modifying properties */
void define_property(
    in PropertyName property_name,
    in any property_value)
    raises(InvalidPropertyName,
           ConflictingProperty,
           UnsupportedTypeCode,
           UnsupportedProperty,
           ReadOnlyProperty);

void define_properties(
    in Properties nproperties)
    raises(MultipleExceptions);
```

Figure 13-3 Operations used to define new properties or set new values.

define_property

Will modify or add a property to the *PropertySet*. If the property already exists, then the property type is checked before the value is overwritten. If the property does not exist, then the property is added to the *PropertySet*.

To change the *any* TypeCode portion of the *property_value* of a property, a client must first *delete_property*, then invoke the *define_property*.

define_properties

Will modify or add each of the properties in *Properties* parameter to the *PropertySet*. For each property in the list, if the property already exists, then the property type is checked before overwriting the value. If the property does not exist, then the property is added to the *PropertySet*.

This is a batch operation that returns the *MultipleExceptions* exception if any *define* operation failed.

Table 13-2 Exceptions Raised by Define Operations

Exception Raised	Description
InvalidPropertyName	Indicates that the property name is invalid. (A property name of length 0 is invalid; implementations may place other restrictions on property names.)
ConflictingProperty	Indicates that the property indicated created a conflict in the type or value provided.
UnsupportedTypeCode	Indicates that the <i>any</i> TypeCode of the <code>property_value</code> field is not supported in this <i>PropertySet</i> .
UnsupportedProperty	Indicates that the supplied property is not supported in this <i>PropertySet</i> , either due to <code>PropertyName</code> restrictions or specific name-value pair restrictions.
ReadOnlyProperty	Indicates that the property does not support client modification of the <code>property_value</code> field.
MultipleExceptions	The <code>PropertyExceptions</code> sequence may contain any of the exceptions listed above, multiple times and in any order.

Listing and Getting Properties

This set of operations is used to retrieve property names and values from a *PropertySet*.

```

/* Support for Getting Properties and their Names */
unsigned long get_number_of_properties();

void get_all_property_names(
    in unsigned long how_many,
    out PropertyNames property_names,
    out PropertyNamesIterator rest);

any get_property_value(
    in PropertyName property_name)
    raises(PropertyNotFound,
           InvalidPropertyName);

boolean get_properties(
    in PropertyNames property_names,
    out Properties nproperties);

void get_all_properties(
    in unsigned long how_many,
    out Properties nproperties,
    out PropertiesIterator rest);

```

Figure 13-4 Operations used to retrieve property names and values.

get_number_of_properties

Returns the current number of properties associated with this *PropertySet*.

get_all_property_names

Returns all of the property names currently defined in the *PropertySet*. If the *PropertySet* contains more than *how_many* property names, then the remaining property names are put into the *PropertyNamesIterator*.

get_property_value

Returns the value of a property in the *PropertySet*.

get_properties

Returns the values of the properties listed in `property_names`.

When the boolean flag is true, the `Properties` parameter contains valid values for all requested property names. If false, then all properties with a value of type `tk_void` may have failed due to `PropertyNotFound` or `InvalidPropertyName`.

An separate invocation of `get_property` for each such property name is necessary to determine the specific exception or to verify that `tk_void` is the correct *any* `TypeCode` for that property name.

This approach was taken to avoid a complex, hard to program structure to carry mixed results.

get_all_properties

Returns all of the properties defined in the *PropertySet*. If more than `how_many` properties are found, then the remaining properties are returned in *PropertiesIterator*.

Table 13-3 Exceptions Raised by List and Get Properties Operations

Exception Raised	Description
<code>PropertyNotFound</code>	Indicates that the specified property was not defined for this <code>PropertySet</code> .
<code>InvalidPropertyName</code>	Indicates the property name is invalid. (A property name of length 0 is invalid; implementations may place other restrictions on property names.)
<code>MultipleExceptions</code>	The <code>PropertyExceptions</code> sequence may contain any of the exceptions listed above, multiple times and in any order.

Deleting Properties

This set of operations can be used to delete one or more properties from a *PropertySet*.

```

/* Support for Deleting Properties                                */
void delete_property(
    in PropertyName property_name)
    raises(PropertyNotFound,
           InvalidPropertyName,
           FixedProperty);

void delete_properties(
    in PropertyNames property_names)
    raises(MultipleExceptions);

boolean delete_all_properties();

```

Figure 13-5 Operations used to delete properties.

delete_property

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

delete_properties

Deletes the properties defined in the `property_names` parameter. This is a batch operation that returns the `MultipleExceptions` exception if any delete failed.

delete_all_properties

Variation of `delete_properties`. Applies to all properties.

Since some properties may be defined as fixed property types, it may be that not all properties are deleted. The boolean flag is set to false to indicate that not all properties were deleted.

A client could invoke `get_number_of_properties` to determine how many properties remain. Then invoke `get_all_property_names` to extract the property names remaining. A separate invocation of `delete_property` for each such property name is necessary to determine the specific exception.

Note – If the property is in a *PropertySetDef*, then the `set_mode` operation could be invoked to attempt to change the property mode to something other than fixed before using `delete_property`.

This approach was taken to avoid the use of an iterator to return an indeterminate number of exceptions.

Table 13-4 Exceptions Raised by `delete_properties` Operations

Exception Raised	Description
PropertyNotFound	Indicates that the specified property was not defined.
InvalidPropertyName	Indicates that the property name is invalid. (A property name of length 0 is invalid; implementations may place other restrictions on property names.)
FixedProperty	Indicates that the <i>PropertySet</i> does not support the deletion of the specified property.
MultipleExceptions	The <i>PropertyExceptions</i> sequence may contain any of the exceptions listed above, multiple times and in any order.

Determining If a Property Is Already Defined

The `is_property_defined` operation returns true if the property is defined in the *PropertySet*, and returns false otherwise.

```
boolean is_property_defined(  
    in PropertyName property_name)  
    raises(InvalidPropertyName) ;
```

Figure 13-6 `is_property_defined` operation.

13.2.3 *PropertySetDef Interface*

The *PropertySetDef* interface is a specialization (subclass) of the *PropertySet* interface. The *PropertySetDef* interface provides operations to retrieve *PropertySet* constraints, define and modify properties with modes, and to get or set property modes.

It should be noted that a *PropertySetDef* is still considered a *PropertySet*. The specialization operations are simply to provide more client access and control of the characteristics (metadata) of a *PropertySet*.

The *PropertySetDef* interface also provides “batch” operations, such as `define_properties_with_modes`, to deal with sets of property definitions as a whole. The execution of the “batch” operations is considered best effort (i.e., not an atomic set) in that not all suboperations need to succeed for any suboperation to succeed.

For `define_properties_with_modes` and `set_property_modes`, if any suboperation fails, a `MultipleExceptions` exception is returned to identify which property name had which exception.

For example, a client may invoke `define_properties_with_modes` using four property definition structures. The first property could be accepted (added or modified), the second could fail due to an `UnsupportedMode`, the third could fail due to a `ConflictingProperty`, and the fourth could fail due to `ReadOnlyProperty`. In this case a property is either added or modified in the *PropertySetDef* and a `MultipleExceptions` exception is raised with three items in the `PropertyExceptions` sequence.

The `get_property_modes` “batch” operation utilizes a boolean flag to signal that mixed results occurred and additional processing may be required to fully analyze the exceptions.

Making “batch” operations behave in an atomic manner is considered an implementation issue that could be accomplished via specialization of this property service.

Retrieval of PropertySet Constraints

This set of operations is used to retrieve information related to constraints placed on a *PropertySet*.

```
/* Support for retrieval of PropertySet constraints*/  
void get_allowed_property_types(  
    out PropertyTypes property_types);  
  
void get_allowed_properties(  
    out PropertyDefs property_defs);
```

Figure 13-7 Operations used to retrieve information related to constraints.

get_allowed_property_types

Indicates which types of properties are supported by this *PropertySet*. If the output sequence is empty, then there is no restrictions on the *any* TypeCode portion of the *property_value* field of a *Property* in this *PropertySet*, unless the *get_allowed_properties* output sequence is not empty.

For example, a *PropertySet* implementation could decide to only accept properties that had *any* TypeCodes of *tk_string* and *tk_ushort* to simplify storage processing and retrieval.

get_allowed_properties

Indicates which properties are supported by this *PropertySet*. If the output sequence is empty, then there is no restrictions on the properties that can be in this *PropertySet*, unless the *get_allowed_property_types* output sequence is not empty.

Defining and Modifying Properties with Modes

This set of operations is used to define new properties to a *PropertySet* or set new values on existing properties.

```
/* Support for defining and modifying properties */
void define_property_with_mode(
    in PropertyName property_name,
    in any property_value,
    in PropertyModeType property_mode)
    raises(InvalidPropertyName,
           ConflictingProperty,
           UnsupportedTypeCode,
           UnsupportedProperty,
           UnsupportedMode,
           ReadOnlyProperty);

void define_properties_with_modes(
    in PropertyDefs property_defs)
    raises(MultipleExceptions);
```

Figure 13-8 Operations used to define new properties or values.

define_property_with_mode

This operation will modify or add a property to the *PropertySet*. If the property already exists, then the property type is checked before the value is overwritten. The property mode is also checked to be sure a new value may be written. If the property does not exist, then the property is added to the *PropertySet*.

To change the *any* TypeCode portion of the *property_value* of a property, a client must first *delete_property*, then invoke the *define_property_with_mode*.

define_properties_with_modes

This operation will modify or add each of the properties in the *Properties* parameter to the *PropertySet*. For each property in the list, if the property already exists, then the property type is checked before overwriting the value. The property mode is also checked to be sure a new value may be written. If the property does not exist, then the property is added to the *PropertySet*.

This is a batch operation that returns the `MultipleExceptions` exception if any `define` operation failed.

Table 13-5 Exceptions Raised by `define` Operations

Exception Raised	Description
<code>InvalidPropertyName</code>	Indicates that the property name is invalid. (A property name of length 0 is invalid; implementations may place other restrictions on property names.)
<code>ConflictingProperty</code>	Indicates that the property indicated created a conflict in the type or value provided.
<code>UnsupportedTypeCode</code>	Indicates that the <i>any</i> <code>TypeCode</code> of the <code>property_value</code> field is not supported in this <i>PropertySet</i> .
<code>UnsupportedProperty</code>	Indicates that the supplied property is not supported in this <i>PropertySet</i> , either due to <code>PropertyName</code> restrictions or specific name-value pair restrictions.
<code>UnsupportedMode</code>	Indicates that the mode supplied is not supported in this <i>PropertySet</i> .
<code>ReadOnlyProperty</code>	Indicates that the property does not support client modification of the <code>property_value</code> field.
<code>MultipleExceptions</code>	The <code>PropertyExceptions</code> sequence may contain any of the exceptions listed above, multiple times and in any order.

Getting and Setting Property Modes

This set of operations is used to get and set the property mode associated with one or more properties.

```

/* Support for Getting and Setting Property Modes */
PropertyModeType get_property_mode(
    in PropertyName property_name)
    raises(PropertyNotFound,
           InvalidPropertyName);

boolean get_property_modes(
    in PropertyNames property_names,
    out PropertyModes property_modes);

void set_property_mode(
    in PropertyName property_name,
    in PropertyModeType property_mode)
    raises(InvalidPropertyName,
           PropertyNotFound,
           UnsupportedMode);

void set_property_modes(
    in PropertyModes property_modes)
    raises(MultipleExceptions);
};

```

Figure 13-9 Operations used to get and set property mode.

get_property_mode

Returns the mode of the property in the *PropertySet*.

get_property_modes

Returns the modes of the properties listed in *property_names*.

When the boolean flag is true, the `property_modes` parameter contains valid values for all requested property names. If false, then all properties with a `property_mode_type` of undefined failed due to `PropertyNotFound` or `InvalidPropertyName`. A separate invocation of `get_property_mode` for each such property name is necessary to determine the specific exception for that property name.

This approach was taken to avoid a complex, hard to program structure to carry mixed results.

set_property_mode

Sets the mode of a property in the *PropertySet*.

Protection of the mode of a property is considered an implementation issue. For example, an implementation could raise the `UnsupportedMode` when a client attempts to change a `fixed_normal` property to `normal`.

set_property_modes

Sets the mode for each property in the `property_modes` parameter. This is a batch operation that returns the `MultipleExceptions` exception if any set failed.

Table 13-6 Exceptions Raised by Get and Set Mode Operations

Exception Raised	Description
<code>PropertyNotFound</code>	Indicates that the specified property was not defined.
<code>InvalidPropertyName</code>	Indicates that the property name is invalid. (A property name of length 0 is invalid; implementations may place other restrictions on property names.)
<code>UnsupportedMode</code>	Indicates that the mode supplied (set operations only) is not supported in this <i>PropertySet</i> .
<code>MultipleExceptions</code>	The <code>PropertyExceptions</code> sequence may contain any of the exceptions listed above, multiple times and in any order.

13.2.4 *PropertiesIterator Interface*

A *PropertySet* maintains a set of name-value pairs. The `get_all_properties` operation of the *PropertySet* interface returns a sequence of Property structures (Properties). If there are additional properties, the `get_all_properties` operation returns an object supporting the *PropertiesIterator* interface with the additional properties.

The *PropertiesIterator* interface allows a client to iterate through the name-value pairs using the `next_one` or `next_n` operations.

Resetting the Position in an Iterator

The `reset` operation resets the position in an iterator to the first property, if one exists.

```
void reset();
```

Figure 13-10 `reset` operation.

next_one, next_n

The `next_one` operation returns true if an item exists at the current position in the iterator with an output parameter of a property. A return of false signifies no more items in the iterator.

The `next_n` operation returns true if an item exists at the current position in the iterator and the `how_many` parameter was set greater than zero. The output is a properties sequence with at most the `how_many` number of properties. A return of false signifies no more items in the iterator.

```
boolean next_one(out Property aproperty);  
boolean next_n(  
    in unsigned long how_many,  
    out Properties nproperties);
```

Figure 13-11 `next_one` and `next_n` operations (properties)

Destroying the Iterator

The `destroy` operation destroys the iterator.

```
void    destroy();
```

Figure 13-12 `destroy` operation.

13.2.5 PropertyNamesIterator Interface

A *PropertySet* maintains a set of name-value pairs. The `get_all_property_names` operation returns a sequence of names (*PropertyNames*). If there are additional names, the `get_all_property_names` operation returns an object supporting the *PropertyNamesIterator* interface with the additional names.

The *PropertyNamesIterator* interface allows a client to iterate through the names using the `next_one` or `next_n` operations.

Resetting the Position in an Iterator

The `reset` operation resets the position in an iterator to the first property name, if one exists.

```
void reset();
```

Figure 13-13 `reset` operation.

next_one, next_n

The `next_one` operation returns true if an item exists at the current position in the iterator with an output parameter of a property name. A return of false signifies no more items in the iterator.

The `next_n` operation returns true if an item exists at the current position in the iterator and the `how_many` parameter was set greater than zero. The output is a *PropertyNames* sequence with at most the `how_many` number of names. A return of false signifies no more items in the iterator.

```
boolean next_one(out PropertyName property_name);  
boolean next_n(  
    in unsigned long how_many,  
    out PropertyNames property_names);
```

Figure 13-14 `next_one`, `next_n` operations (PropertyNames)

Destroying the Iterator

The *destroy* operation destroys the iterator.

```
void    destroy();
```

Figure 13-15 `destroy` operation.

13.2.6 PropertySetFactory Interface

The `create_propertyset` operation returns a new *PropertySet*. It is considered an implementation issue as to whether the *PropertySet* contains any initial properties or has constraints.

The `create_constrained_propertyset` operation allows a client to create a new *PropertySet* with specific constraints. The modes associated with the allowed properties is considered an implementation issue.

The `create_initial_propertyset` operation allows a client to create a new *PropertySet* with specific initial properties. The modes associated with the initial properties is considered an implementation issue.

```
interface PropertySetFactory
{
    PropertySet create_propertyset();
    PropertySet create_constrained_propertyset(
        in PropertyTypes allowed_property_types,
        in Properties allowed_properties)
        raises(ConstraintNotSupported);
    PropertySet create_initial_propertyset(
        in Properties initial_properties)
        raises(MultipleExceptions);
};
```

Figure 13-16 *PropertySetFactory* interface.

Deletion of any initial properties is an implementation concern. For example, an implementation may choose to initialize the *PropertySet* with a set of *fixed_readonly* properties for *create_propertyset* or choose to initialize all *allowed_properties* to be *fixed_normal* for *create_constrained_propertyset*.

The relationship of a *PropertySet* to a specific object is an implementation issue.

13.2.7 *PropertySetDefFactory* Interface

The *create_propertysetdef* operation returns a new *PropertySetDef*. It is considered an implementation issue as to whether the *PropertySetDef* contains any initial properties or has constraints.

The *create_constrained_propertysetdef* operation allows a client to create a new *PropertySetDef* with specific constraints, including property modes.

The `create_initial_propertysetdef` operation allows a client to create a new *PropertySetDef* with specific initial properties, including property modes.

```
interface PropertySetDefFactory
{
    PropertySetDef create_propertysetdef();
    PropertySetDef create_constrained_propertysetdef(
        in PropertyTypes allowed_property_types,
        in PropertyDefs allowed_property_defs)
        raises(ConstraintNotSupported);
    PropertySetDef create_initial_propertysetdef(
        in PropertyDefs initial_property_defs)
        raises(MultipleExceptions);
};
```

Figure 13-17 *PropertySetDefFactory* interface.

It should be noted that deletion of initial or allowed properties is tied to the property mode setting for that property. In other words, initial or allowed properties are not inherently safe from deletion.