erwin® Data Modeler

API Reference Guide

Release 9.8

erwin

Documentation Changes

The following documentation updates have been made since the last release of this documentation:

- Following topics have been updated to document the latest Target Database Servers.
  - Metadata Tags
  - Property Bag Contents for Persistence Unit and Persistence Unit Collection

Contact erwin

Understanding your Support

Review support maintenance programs and offerings.

Registering for Support

Access the erwin support site and click Sign in to register for product support.

Accessing Technical Support

For your convenience, erwin provides easy access to "One Stop" support for all editions of erwin Data Modeler, and includes the following:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- erwin Support policies and guidelines
- Other helpful resources appropriate for your product

For information about other erwin products, visit http://erwin.com/products.

Provide Feedback

If you have comments or questions, or feedback about erwin product documentation, you can send a message to techpubs@erwin.com.

erwin Data Modeler News and Events

Visit www.erwin.com to get up-to-date news, announcements, and events. View video demos and read up on customer success stories and articles by industry experts.

Contents

Appendix B: erwin DM Metamodel 185

Chapter 1: Introduction to API

The Script Client API that is part of erwin DM provides advanced customization capabilities that enable you to access and manipulate modeling data in memory at runtime, as well as models persisted in files and in a mart. The API interfaces are automation-compatible and provide extensive design and runtime facilities for third-party integrators as well as users of script-based environments.

The API complements the original modeling tool with custom components when you use scripts, add-ins, and COM-based API technologies. The API is flexible and promotes a seamless integration of the modeling tool in a client development cycle.

This section contains the following topics

Major Features

The API is a group of interfaces that includes the following features:

Active Model Data Objects (AMDO)

Lets a third-party client to access model data through a COM automation-compatible API. This feature is the major component in the API functionality. All interfaces that comprise the API are automation-based, and are therefore *dual*. These dual interfaces allow you faster access to methods and properties. Using dual interfaces, you can directly call the functions without using an *Invoke()* function.

Collections and enumerators

Facilitates programming constructions in script languages that target the AMDO automation features.

Connection points

Delivers a collection of connection points interfaces and support for the *ITypeInfo2* interface to support the sync event facilities of languages.

Automation-rich error handling

Supports automation-rich error handling through *IErrorInfo* interfaces exposed by the API components.

Active Model Directory

Lets you navigate available model storage, including marts. Delivers the ability for a client to open or to create a model in a file as well as from a mart.

Active Scripting

Lets you host a scripting environment and provide an invocation mechanism for script and add-in components. A mechanism is provided to register add-ins and scriplets with the Active Scripting environment.

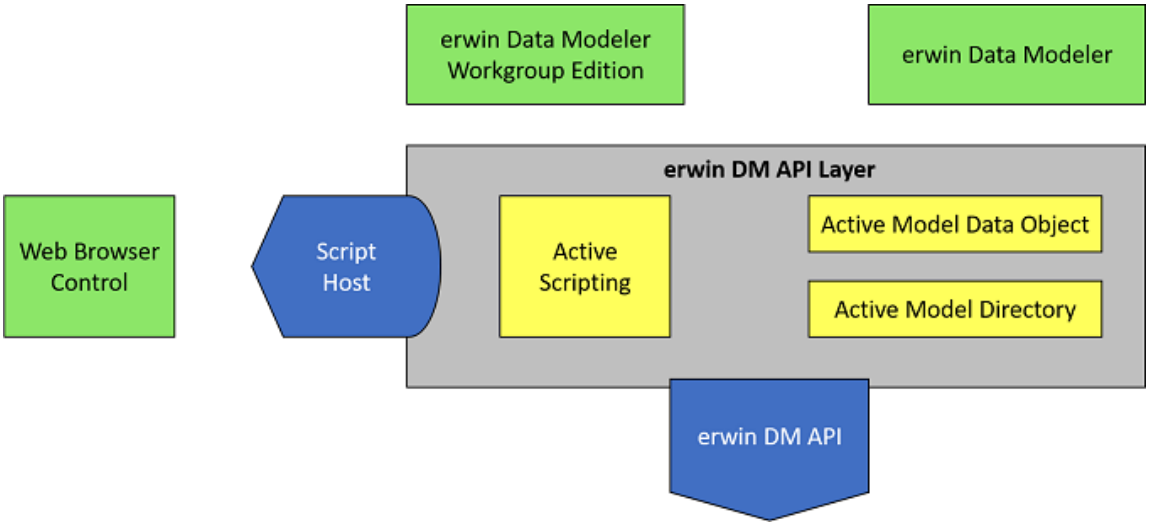
Typical Use Cases

The typical use cases of the API are automation and scripts to support specific interface design requirements imposed by COM automation standards. For example, you can be limited to a single incoming and outgoing interface exposed by any particular COM object. This limitation is due to the fact that the only recognizable interface type for pure automation is *IDispatch* and it renders the use of *QueryInterface* functionality unfit. The common technique to address the problem includes Alternate Identities and read-only properties that expose secondary interfaces.

Another example of a targeted domain customer is one using alternative (not C++) languages to implement a client. The list includes Visual Basic, VB Script, Java Script, and so on. The list includes specially tailored language idioms to encapsulate language-COM binding, such as collections of objects, connection points, rich error handling, and so on.

The API combines number of components and presents them as a set of interfaces accessible using COM.

The list of integrated components includes erwin Data Modeler and Microsoft Internet Explorer.



Standalone Client

One of the ways the API is used is as a standalone client. A third-party client activates the API as an in-process server. The API component does not have visual representation, that is, it does not expose a user interface. The API provides Active Model Directory facilities to specify a target model from a list of available models. Active Model Data Objects provide session-based access to model data.

There are times when API clients can compete with other parties over access to model data. Using erwin® Data Modeler Workgroup Edition provides advanced model sharing facilities to prevent other parties from accessing the model during your session.

Add-in Component or Script

Another way the API is used is as an add-in component or script. erwin DM hosts third-party add-in modules and scripts. The Active Scripting component in the API provides a mechanism for registering modules with a host tool, arranging representation in the host user interface, creating add-in menus, and invoking them on the host menu selection or event.

The add-in module is a client DLL, activated in-process.

The script is a VBScript or JScript procedure embedded in a DHTML document, activated using a menu or a model event. This Active Scripting provides hosting for web browser control and makes the API objects available through the *window.external* property of the DHTML object model.

You can observe changes in a model on the screen and can activate a pause to investigate the state of a model by accessing the modeling tool user interface.

Chapter 2: API Components

This section contains the following topics

Overview

The API is a collection of interfaces that represent erwin DM functionality. The application exports the top-level interface, from which the client obtains lower-level interfaces as needed. Interfaces are logically grouped into tiers, where each tier includes interfaces that represent the functionality of the application. Each tier is represented in the following sections, with a table describing the interfaces grouped into that tier.

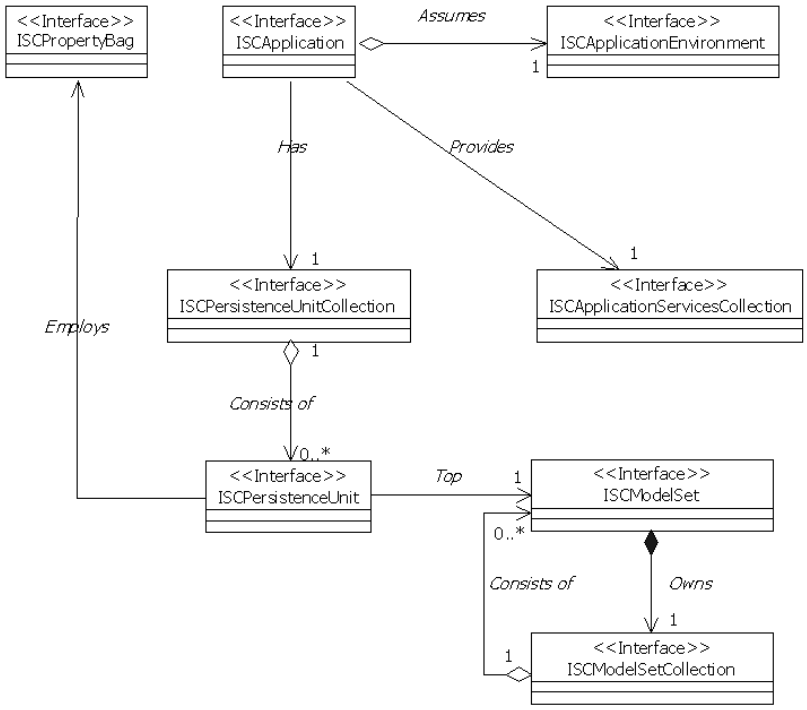
Application Tier

The Application Tier represents erwin DM functionality, establishes access to models in persistent storage, and controls the exchange between models in memory and models in persistent storage. The following table describes the interfaces of the Application Tier:

Interface	Role
ISCApplication	Represents application-wide functionality, and serves as the entry point for the interface hierarchy of the API. Holds a list of available persistence units and connections between the client and persistence units.
ISCApplicationEnvironment	Provides information about the runtime environment.
ISCPersistenceUnitCollection	Collects all active persistence units known to the application.
ISCPersistenceUnit	Represents an active persistence unit (such as a erwin DM model) within the application. A persistence unit groups data in the form of model sets. Clients can connect to persistence units to manipulate them and the data they contain.
ISCModelSetCollection	Represents model sets associated with a persistence unit.

Interface	Role
ISCModelSet	Represents a model set (such as EMX or EM2 classes of model data) within a single persistence unit.
ISCPropertyBag	Represents an array of properties for application tier interface calls.

This is a graphical representation of the relationships of the Application Tier:

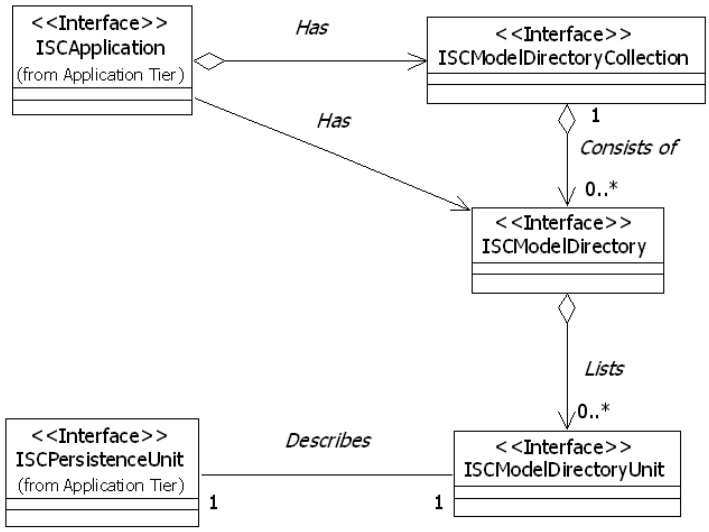


Model Directory Tier

The Model Directory Tier accesses and manipulates the persistence storage directories, such as a file system directory or a mart directory. The following table describes the interfaces of the Model Directory Tier:

Interface	Role
ISCModelDirectoryCollection	Enumerates all top-level model directories available for the API client.
ISCModelDirectory	Encapsulates information on a single model directory entry.
ISCModelDirectoryUnit	Encapsulates information on a single directory unit.

This is a graphical representation of the relationships of the Model Directory Tier:

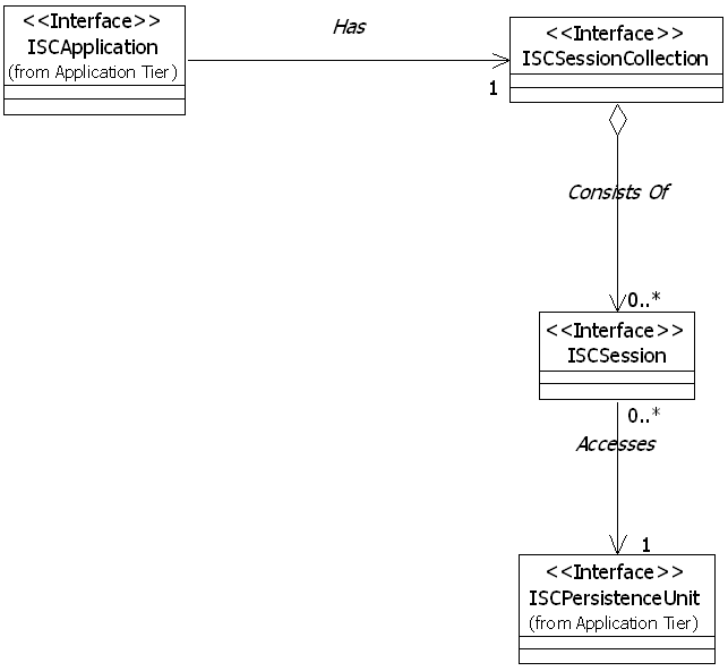


Sessions Tier

The Sessions Tier establishes access to model data in memory. The following table describes the interfaces of the Sessions Tier:

Interface	Role
ISCSessionCollection	Collects all active sessions between the API client and the persistence units.
ISCSession	Represents an active connection between the client and a model. Clients create sessions, and then open them against model sets of persistence units. An open session exposes a single level (such as data, metadata, and so on) of a model set.

This is a graphical representation of the relationships of the Sessions Tier:

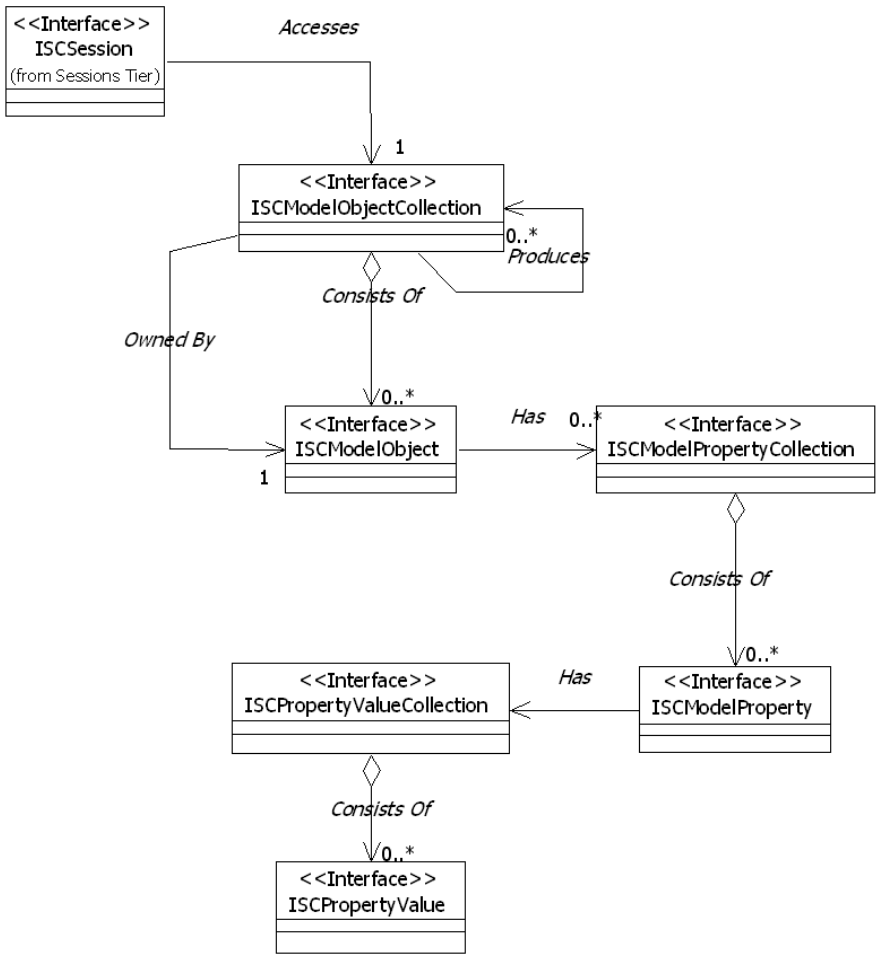


Model Data Tier

The Model Data Tier accesses and manipulates model data. The following table describes the interfaces of the Model Data Tier:

Interface	Role
ISCModelObjectCollection	Represents objects available for manipulation. Membership in this collection can be limited by establishing filter criteria.
ISCModelObject	Accesses and manipulates a single object within a model.
ISCModelPropertyCollection	Represents a list of properties owned by a single object. The list can be limited by using filters.
ISCModelProperty	Accesses and manipulates a single property. Properties may contain multiple values. Values within a multi-valued property are accessed by keys. The current multi-valued property implementation treats the value list as an array, and the key is the array index.
ISCPropertyValueCollection	Represents a list of single property values.
ISCPropertyValue	Data and a key are contained within a single value.

This is a graphical representation of the relationships of the Model Data Tier:



Access to Model Data

The API allows API clients to manipulate models. An API client locates models in persistence storage by using the *Model Directory Collection*, *Model Directory*, and the *Model Directory Unit* components. By using its properties, the *Model Directory Unit* provides the information necessary to register the unit with the pool of available persistence units by using the *Persistence Units* collection. The API client can then specify access attributes such as read-only or ignore locks. A new model can be created and registered with a persistence unit collection. erwin DM can add or remove models from the pool as a response to user interface actions.

A persistence unit maintains a set of properties to control visibility in the application user interface, access attributes, and so on. A persistence unit organizes data as a group of linked model sets. The model sets are arranged in a tree-like hierarchy with a single model set at the top. The top model set in the persistence unit contains the bulk of the modeling data. The API uses the abbreviation *EMX* to identify the top model set. The EMX model set owns a secondary model set abbreviated as *EM2*, that contains user options and user interface settings.

API clients access the model data by constructing a session and connecting it to a model set using the *Session* component. A model set contains several levels of data. It contains the data the application manipulates, such as entity instances, attribute instances, or relationship instances.

The model set also contains metadata, which is a description of the objects and properties that *may* occur within the application's data. In erwin DM, metadata includes object and property classes, object aggregations, and property associations. The metadata defines each object class that may occur within a model, for example, an entity class, an attribute class, or a relationship class. Object aggregations identify an ownership relationship between classes of objects. For example, a model owns entities, entities own attributes, and so on. The property associations define property usage by object classes. For instance, the metadata includes property associations for every object class that has the *Name* property.

Clients specify the necessary level of model data at the same time as connecting a session to a model set. When a new model is created it acquires a set of default objects, such as model object, main subject area, and stored display. The initial API implementation supports the following levels:

Name	Description	Supported Actions
SCD_SL_ M0	Model Level	Access model data, create and delete objects (including the entire model), and set property values.
SCD_SL_ M1	Metamodel Level	Access object and property definitions, along with other metadata. Create and delete user-defined properties and user-defined object definitions.

Levels are identified by long integer values. Values have symbolic definitions.

Objects and Properties

The API presents data in object/property form. In a erwin DM model, for example, an attribute is represented by an instance of an *Attribute* object. The name of the attribute is contained in the *Name* property of the *Attribute* object.

Object Identifiers

Each object must bear an identifier, which is a value that uniquely identifies the object instance. Internally, object identifiers are 20 bytes long. They contain two components: a GUID (also known as a UUID) in the first 16 bytes, and a 32-bit unsigned integer suffix in the last 4 bytes.

A GUID contains the following components:

- One 32-bit unsigned integer
- Two 16-bit unsigned integers
- Eight 8-bit unsigned integers (represented as unsigned characters)

These components total of 128 bits, or 16 bytes. Therefore, an object identifier contains an extra 32-bit unsigned integer (the 4 byte suffix) at the end for a total of 160 bits, or 20 bytes.

To simplify working with object identifiers and due to COM automation limitations on datatypes, the API uses a string to represent object identifiers.

The following table lists aliases used in this guide and in the interface definitions:

Type Name	Format	Use
SC_OBJID	{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}+suffix	Object identifier
SC_CLSID	{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}+suffix	Class (object, property type, and so on) identifier
SC_MODELTYPEID	{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}+suffix	Model type identifier
SC_CREATORID	{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}	Creator identifier

The identifiers whose GUID component contains zero is one set of object identifiers that is predefined. If the final 4 bytes of the identifier also contain zero, the identifier represents a null identifier. Other values of the offset are reserved for future use.

Object Identifiers and Type Codes

Consider the relationship between object instances in the *SCD_SL_ M0* layer and object instances in the *SCD_SL_ M1* layer. An instance in the *SCD_SL_ M0* layer is described by an instance in the *SCD_SL_ M1* layer. For instance, a single object in the *SCD_SL_ M1* layer describes every entity instance in the *SCD_SL_ M0* layer.

Since all type codes are also object identifiers, they must have the same format.

Properties, Property Flags, and Value Facets

Properties present data in the form of values and additional flags.

Property values are either scalar with a single value, or non-scalar with multiple values. More information about scalar and non-scalar property values is located in the Scalar and Non-Scalar Property Values (see page 23) section.

Property values are defined by a property type, such as a string or an integer. More information about property types is located in the Enumerations (see page 159) section.

Two types of additional property flags exist:

Property level flags

Provide information about the property and are read-only. Property level flags can provide the following information about a property instance:

Metadata information

Shows whether a property in the metadata is user-defined or contains a scalar value.

Property state information

Shows whether or not a property is read-only.

Data source information

Shows whether or not a data source is calculated.

Property value level flags

Convey information about property value and can be updated.

An individual property level flag is represented by a bit field in the property flag's value. The flags are provided for information only and cannot be changed. More information about specific property flags is located in the Enumerations (see page 159) section.

The value level flags, or facets, convey additional data associated with property value such as if a property value was 'hardened' and cannot be changed due to inheritance.

An individual facet is identified by a numeric ID or a name and has one of three possible states: non-set, set to TRUE, or set to FALSE.

The facets are treated as part of the property value. Assigning a new value to a property places all facets in the non-set state. Similarly, a value update or removal renders all facets into the non-set state. There is only one combination of facets per property, either scalar or non-scalar. Changes in individual values of non-scalar properties do not affect the property facets. More information about specific value facets is located in the Property Bag for Application Environment (see page 163) section.

Scalar and Non-Scalar Property Values

A scalar property is a property that can be represented as a single value. The properties that contain multiple values (either homogeneous or heterogeneous) are non-scalar properties.

The type of a property can be recognized by reviewing the property flags. Scalar properties have a *SCD_MPF_SCALAR* flag.

More information about specific property flags is located in the Enumerations (see page 159) section.

The value of a scalar property or a single member of a non-scalar property is accessed through the *Value* property of the *ISCModelProperty* interface.

Note: Heterogeneous non-scalar properties are not supported by this product. Members in a non-scalar property always have the same datatype.

A property, either scalar or non-scalar, can have a special NULL value. The properties with a NULL value have a *SCD_MPF_NULL* flag set.

Collections and Automation

Automation defines the *IEnumVARIANT* interface to provide a standard way for the API clients to iterate over collections. Every collection interface in the API exposes a read-only property named *_NewEnum* to let the API clients know that the collection supports iteration. The *_NewEnum* property returns a pointer on the *IEnumVARIANT* interface.

The *IEnumVARIANT* interface provides a way to iterate through the items contained by a collection. This interface is supported by an enumerator interface that is returned by the *_NewEnum* property of the collection.

The *IEnumVARIANT* interface defines the following member functions:

Next

Retrieves one or more elements in a collection starting with the current element.

Skip

Skips over one or more elements in a collection.

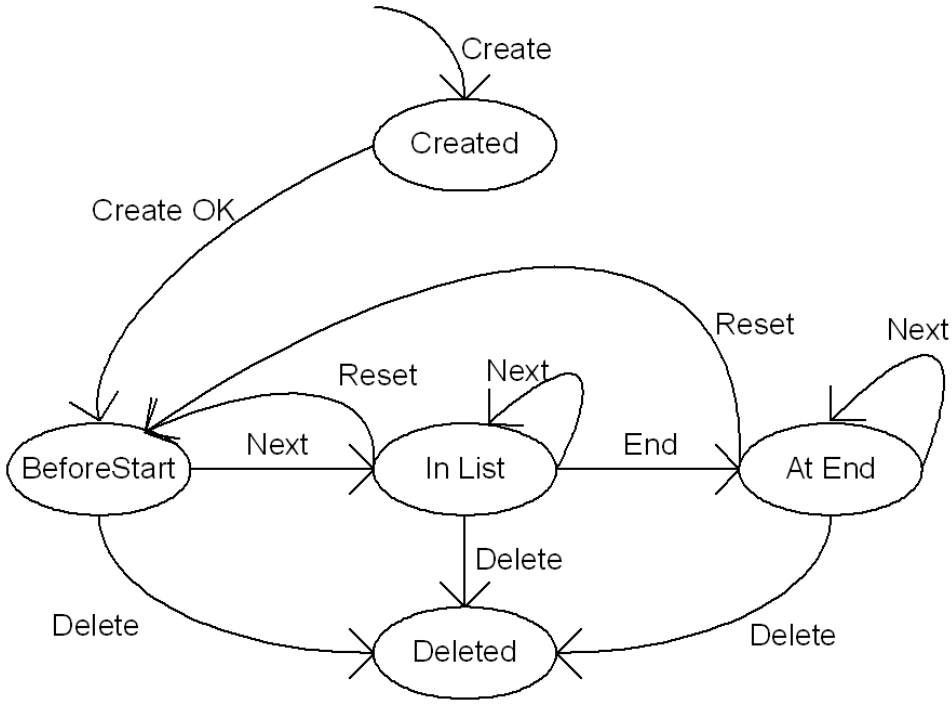
Reset

Resets the current element to the first element in the collection.

Clone

Copies the current state of the enumeration so you can return to the current element after using *Skip* or *Reset*.

The *IEnumVARIANT* collection implements a Rogue Wave Software, Inc. style *advance and return* iteration. For this reason, they have the following life cycle:



IPtUCModelPropIter Life Cycle

When the iterator is created, it enters the *Created* state, and then forces itself into the *BeforeStart* state. A successful advance drives the iterator into the *InList* state, while an unsuccessful advance drives it into the *AtEnd* state. A *Reset* drives the iterator back to the *BeforeStart* state, and deletion drives it into the *Deleted* state.

Note: The iterator is positioned over a member of the collection (that is, associated with a current member) *only* if it is in the *InList* state.

_NewEnum Property of a Collection Object

The *_NewEnum* property identifies support for iteration through the *IEnumVARIANT* interface. The *_NewEnum* property has the following requirements:

- The name is *_NewEnum*.
- It returns a pointer to the enumerator *IUnknown* interface.
- The Dispatch identification for the property is *DISPID = DISPID_NEWENUM (-4)*.

Default Properties

A default property for automation is the property that is accessed when the object is referred to without any explicit property or method call. The property dispatch identifier is *DISPID_VALUE*.

Optional Parameter

To support automation client requirements, all optional parameters are represented as *VARIANT*. For that reason, a parameter type in an interface description is only to document an expected type in the *VARIANT* structure.

The API Sample Client

Two Visual Basic .NET sample projects are provided with the API, *erwinSpy.NET.x64.exe* and *erwinSpy.NET.x86.exe*.

If you run the Custom Setup type of installation, select the erwin API Sample Client when prompted to select the program features that you want to install. After installation, you can access the two sample Visual Basic .NET projects from the erwinSpy.NET subdirectory in the erwin® Data Modeler installation folder.

Using the API Sample Client

This section describes how to utilize the API sample client as a standalone version and as an add-in component.

The standalone version of the sample program is either *erwinSpy.NET.x86.exe* or *erwinSpy.NET.x64.exe*. You can build erwinSpy.NET to create *erwinSpy.NET.x86.exe* or *erwinSpy.NET.x64.exe*. This program is a erwin DM model data browser that you can use to research data internals, such as the metamodel, model data, and model objects and their properties.

Using *erwinSpy.NET.x86.exe* or *erwinSpy.NET.x64.exe*, you can open an *.erwin file by clicking Open on the File menu. When a model is opened or selected from File menu, model objects from the model are displayed in the left pane. You can view a model object's hierarchy (parents and children) and properties by double-clicking on the object.

You can access the model data and metamodel information from the Models menu. Use the Models submenu to access the model data and the MetaModels, EM2 ModelSets, EM2 ModelSets Meta submenus to access the metamodel data.

The add-in version of the sample program is *erwinSpy_Addin.NET* project. You can use the *erwinSpy_Addin.NET* to create a 32-bit (erwinSpy_AddIn.NET.x86.dll) or 64-bit (erwinSpy_AddIn.NET.x64.dll) add-in component. The add-in component runs when you select it from the Tools, Add-Ins menu. After you build the add-in component with the *erwinSpy_Addin.NET* project, you must register it.

Register the Add-in Component

After you build the add-in component with the e*rwinSpy_AddIn.NET* project, you must register it.

To register the add-in component

1. Navigate to the erwinSpy.NET\bin folder in the installation directory.
2. Copy the add-in component to the erwinSpy.NET\bin folder.
3. Rename the add-in depending on your operating environment.
   - For a 32-bit application, rename the add-in to *erwinSpy_AddIn.NET.x86.dll*
   - For a 64-bit application, rename the add-in to *erwinSpy_AddIn.NET.x64.dll*
4. Enter one of the following commands in the command prompt depending on your operating environment.
   - For a 32-bit application, enter **register.bat 32**
   - For a 64-bit application, enter **register.bat 64**

   The add-in component is registered.

Make a VB.NET Library COM Callable

The VB.NET library is not loaded automatically to erwin Data Modeler. You have to make the VB.NET library COM callable.

Follow these steps:

1. Create a VB.NET library project in Visual Studio 2013.
2. Add a COM template class.
3. Right-click on Project and select Add, Component, COM class.
4. Copy the RegisterFunction, UnregisterFunction, and GetSubKeyNmae function from erwinSpy.vb in erwinSpy_AddIn.NET project to the COM template class.
5. Add your public function which can be shared with other users.

   The VB.NET library is now COM callable.

erwin Spy

The erwin Spy application visualizes metadata information and provides intrinsic and model-specific metadata. It demonstrates the API functionality and provides a set of useful features to study how model data is stored. erwin Spy reads the erwin DM metamodel and simplifies the task of comprehending the intricate details of any erwin DM model, which can be a complicated net of model objects, properties, and cross-references. When you install erwin DM, you can choose to install the optional erwin Spy utility.

There are two versions of the utility available in the erwin Spy.NET\bin folder, the standalone version, *erwinSpy.NET.exe*, and the add-in version, *erwinSpy_AddIn.NET.dll*.

These versions are identical in functionality and vary only in how you want to launch the application. The standalone version runs without erwin DM present and can access models stored in .erwin files, while the add-in version launches within erwin DM from the Tools menu and can access models stored in either erwin DM memory or in .erwin files.

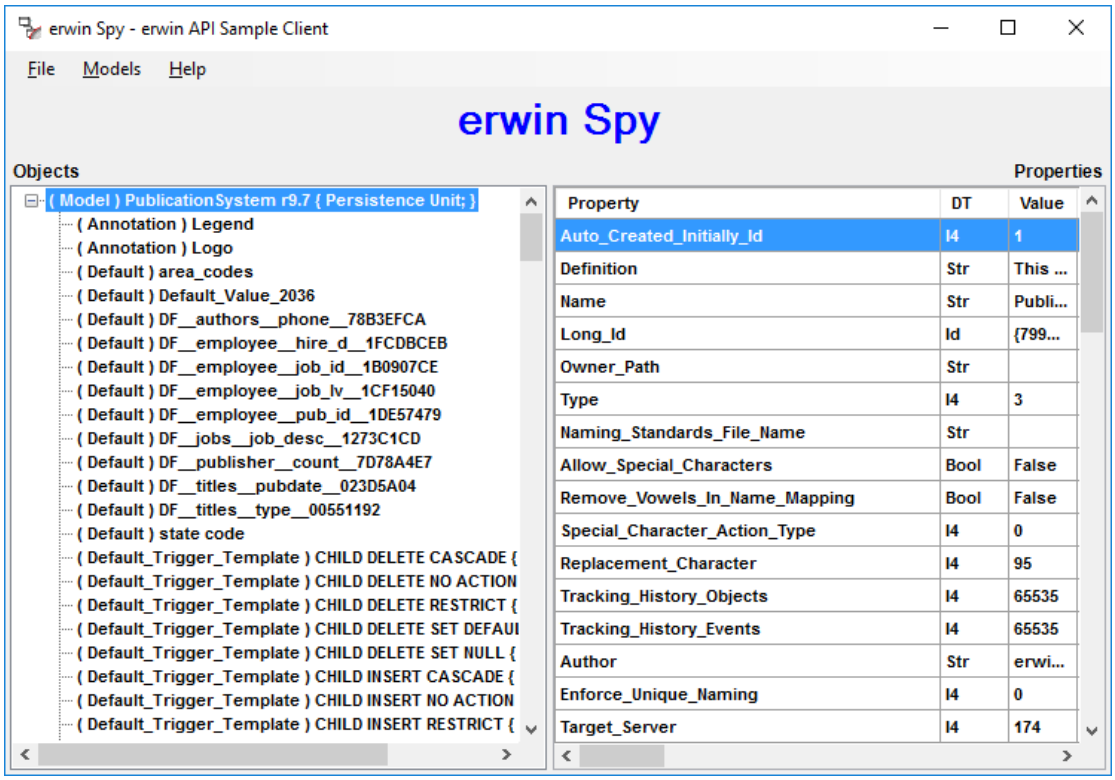
Note: See the Add-In Manager online help for more information about defining an add-in software application for the Tools, Add-Ins menu.

How the erwin Spy Application Works

To see how erwin Spy can help you visualize metadata information, do the following:

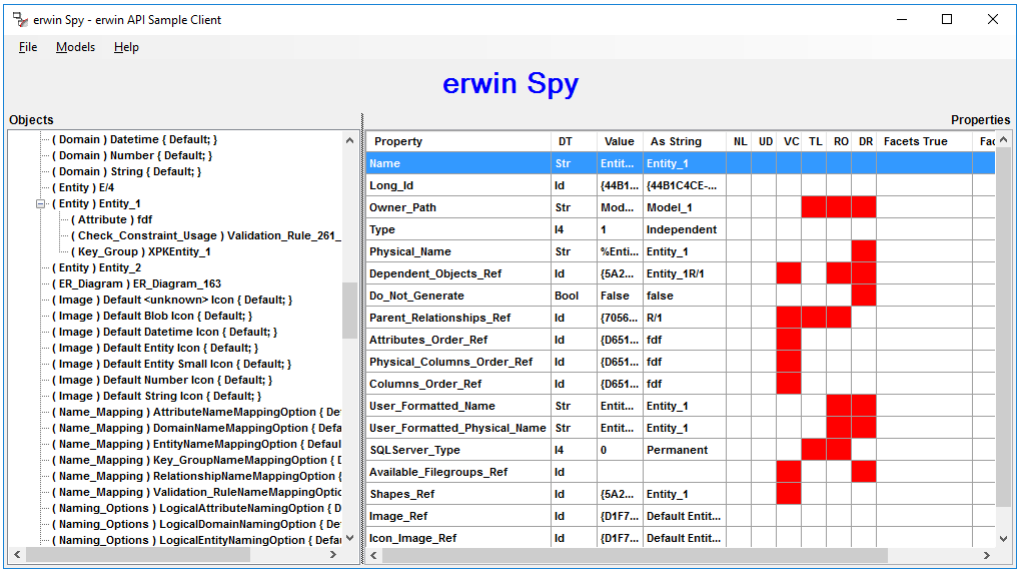
- Start with an empty logical-physical model.
- Click erwin Spy on the Tools, Add-Ins menu to launch erwin Spy.

  Note: Ensure that you have added the erwin Spy application as a erwin DM add-in application on the Tools, Add-Ins menu. See the Add-In Manager online help for more information on defining an add-in software application.
- Select the top item on the Models menu in erwin Spy, which should be your empty model.
- Double-click the Model object in the left pane to expand it. You should see a picture similar to the following illustration:



There are many objects listed by erwin Spy. Even though the model is empty, you still see objects there that represent erwin DM defaults, such as *Domains*, *Main Subject Area*, *Trigger Templates*, and so on. All default objects are marked with a { Default; } flag to the right of the type of the model object.

The right pane of erwin Spy displays object properties. To see a specific object's properties, select the object, click the button located in the center of the screen, and the selected object's properties display in the right panel. The following illustration shows the properties of a specific entity that was added to this model:



The first column shows property names, such as *Name*, *Long ID*, *Type*, *Physical Name*, and so on.

The second column, DT, shows property datatypes, such as *Str* for a string, *I4* for a number, *Bool* for Boolean, *Id* for a reference to another object, and so on.

The third column, Value, displays the property value in native format.

The fourth column, As String, displays the property value reinterpreted as a string. To understand this better, look at Physical Name in the left column. Its value in the Value column is *%EntityName()*, which is a macro, while As String holds the macro expansion, *Entity_1*.

The rest of the columns in the right pane represent property flags. The following list describes the meaning of these columns:

NL

Displays properties with NULL/no value.

Note: The flag is never on for erwin Spy.

UD

Displays user-defined properties.

VC

Displays vector properties.

TL

Displays properties that are maintained by erwin DM and that cannot be changed directly using the API.

RO

Displays read-only properties.

DR

Displays derived properties whose value was inherited (from a parent domain, for example).

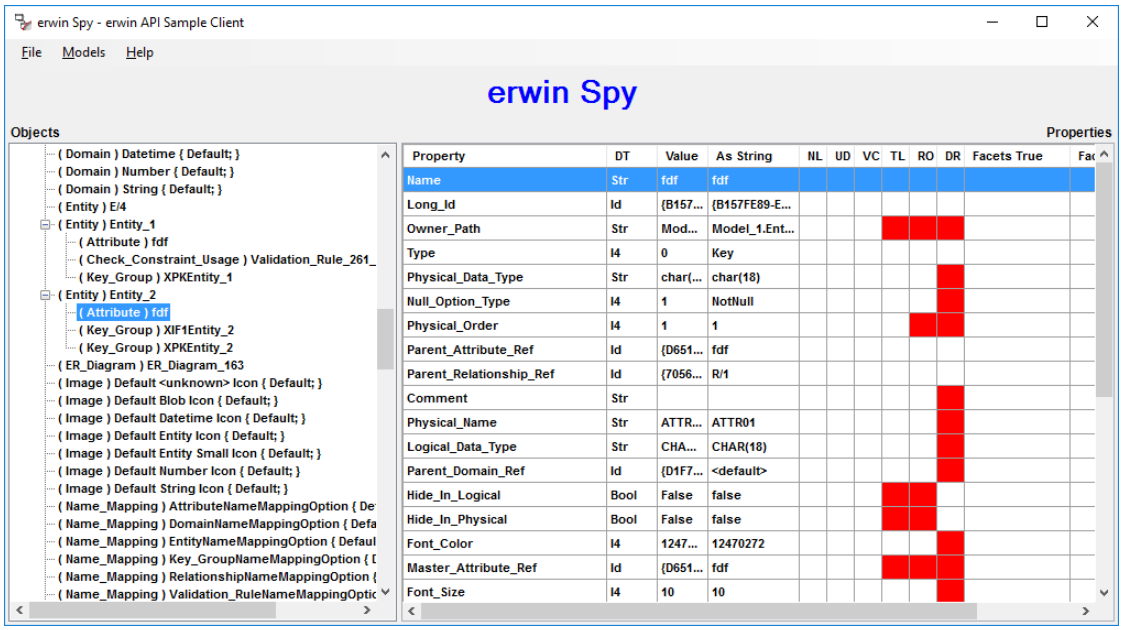
Facets True

Displays the facet value of a property that is set to True.

Facets False

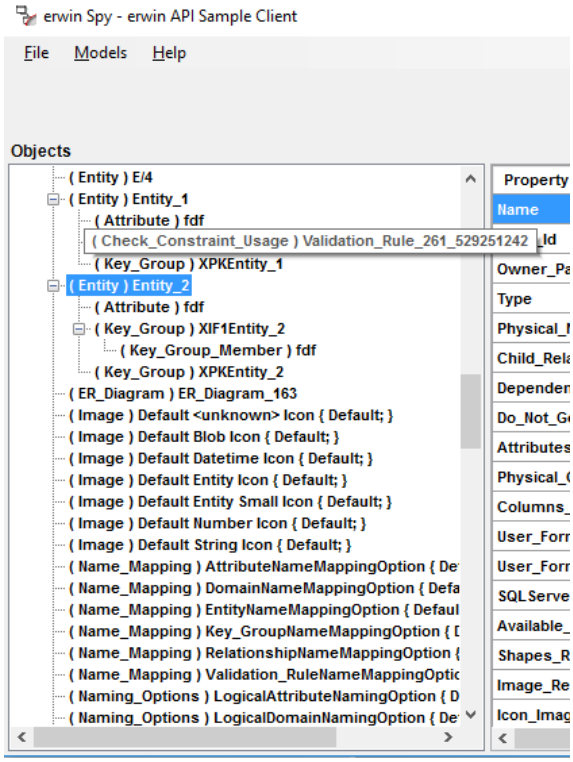
Displays the facet value of a property that is set to False.

In the previous illustration, a primary key attribute named *ATTR01* was added to *Entity_1*. It was migrated to *Entity_2* by creating an identifying relationship. When you double-click *Entity_2*, and then select *ATTR01*, you can see how erwin Spy displays the information. You can click the button in the center of the screen to view its properties on the right.



Since the attribute for the *Parent_Relationship_Ref* property is a product of foreign key migration, this property shows which relationship object is used to store data about it. The value Id in the DT column shows that the property is a reference, which means that the value is a unique ID of the involved relationship object.

Look at the name in the As String column or locate an object by its unique ID to traverse back to the relationship object. To see object IDs, click Show Ids on the File, Options menu. With this option enabled, when the cursor is positioned over an object in the left panel, that object's unique ID is displayed in a popup window, as shown in the following illustration:



Now compare the *Parent_Relationship_Ref* property with the *Parent_Attribute_Ref* and the *Master_Attribute_Ref* properties. The *Master_Attribute_Ref* property is read-only. This means that it is displayed for informational purposes only and cannot be changed using the API. As you build your model, you can expand objects in the model to see how erwin DM uses their properties to represent different relationships in the model.

Use the erwin Spy utility to see and understand the details of the data in a erwin DM model that is available through the API. If you need to learn how particular data is represented in a erwin DM model, you can use the scenarios that were just described. Start with an empty model, create the minimum model that is necessary to represent the feature in question, and then use erwin Spy to look at the details of the data representation.

Chapter 3: API Tasks

This chapter describes how to perform basic tasks using the API. Each task is documented with a table that lists the interfaces and methods needed for that task. In most cases, the table shows a subset of all the methods for that interface. A complete list of API interfaces and their methods is located in the appendix API Interfaces Reference (see page 105).

This section contains the following topics

API Environment

The API is packaged as a set of COM Dynamic Link Libraries (DLL) and works as a part of a customer process. *EAL.dll* is responsible for launching the API environment. When erwin DM is installed, *EAL.dll* and the rest of the API components are copied to the erwin Data Modeler directory, and the installer registers the API with the System Registry.

To use the API in a development environment, use the API Type Library embedded as a resource in the *EAL.dll* file. This operation is language specific. Consult your development environment documentation for details.

The API works in two different modes, standalone mode and add-in mode.

The API is activated and controlled by a client application that hosts its own process in the standalone mode.

In the add-in mode, the API is also activated and controlled by a client application, but the client application is implemented as a *COM DLL*. The erwin DM executable owns a process and all the client application DLLs run inside of that process. COM DLLs must be registered with the System Registry and with the erwin DM Add-In Manager so that it can be available for add-in mode activation.

Behavior of the API components in both modes is the same with a few exceptions that are discussed further in this section.

The API is implemented as a tree of COM interfaces. The application exports the top-level interface, from which the client fetches lower-level interfaces as needed.

Creating the ISCApplication Object

The entry point into the interface hierarchy of the API is through the *ISCApplication* interface. The *ISCApplication* interface provides access to the persistence units and sessions. You must create an instance of *ISCApplication* prior to using any of the other interfaces in the API.

Example 1

The following example illustrates how to use C++ to create the *ISCApplication* object:

```
#import "EAL.dll" using namespace SCAPI;
ISCApplicationPtr scAppPtr;
HRESULT hr;
hr = scAppPtr.CreateInstance(__uuidof(SCAPI::Application));
```

The following example illustrates how to use Visual Basic .NET to create the *ISCApplication* object:

```
Dim scApp As SCAPI.Application
scApp = New SCAPI.Application

// Or the alternative with the ProgId
Dim oApp As Object
oApp = CType(CreateObject("ewin9.SCAPI"), SCAPI.Application)
```

Application Properties

You can get information about the erwin DM application by using the following tables.

ISCApplication Interface

The following table contains information on the *ISCApplication* interface:

Signature	Description	Valid Arguments
BSTR Name()	Modeling Application Title	None
BSTR Version()	Modeling Application Version	None
BSTR ApiVersion()	API version	None
ISCApplicationEnvironment ApplicationEnvironment()	Reports attributes of runtime environment and available features such as add-in mode, user interface visibility, and so on	None
ISCPersistenceUnitCollection * PersistenceUnits()	Returns a collection of all persistence units loaded in the application.	None
ISCSessionCollection * Sessions()	Returns a collection of sessions created within the application	None

ISCApplicationEnvironment

The following table contains information on the *ISCApplicationEnvironment* interface:

Signature	Description	Valid Arguments
ISCPropertyBag PropertyBag(VARIANT Category[optional], VARIANT Name[optional], VARIANT AsString[optional])	Populates a property bag with one or more property values as indicated by Category and Name	Category: ■ Empty – Complete set of features from all categories returned ■ VT_BSTR – Features returned from the given category Name: ■ Empty – All properties from the selected category are returned ■ VT_BSTR – The property with the given name and category returned AsString: ■ Empty – All values in the property bag are presented in their native type ■ VT_BOOL – If set to TRUE, all values in the property bag are presented as strings

Feature categories in the *Category* parameter of the *PropertyBag* property are hierarchical and use a dot (.) to define feature subsets. For example, the *Application* category populates a property bag with a complete set of erwin DM features, while *Application.API* provides a subset related to the API.

If the *Category* parameter is not set, then the *PropertyBag* property returns the complete set of all the features from all the available categories.

Example 2

The following example illustrates how to use the API to retrieve the Application Features using C++. It uses the *Application* object created in Example 1.

```
void IteratePersistenceUnits(ISCApplicationPtr & scAppPtr)
{
  ISCPropertyBagPtr scBag;

  // Retrieve all of application environment properties in one call
  scBag = scAppPtr ->GetApplicationEnvironment()->GetPropertyBag();
  // Get an array with categories by using empty string as a category name
  scBag = scAppPtr ->GetApplicationEnvironment()->GetPropertyBag("",   "Categories")

  // Get Api Version value Application Api category
  scBag = scAppPtr ->GetApplicationEnvironment()->GetPropertyBag  ("Application.Api","Api Version")
}
```

The following example illustrates how to use the API to retrieve the Application Features using Visual Basic .NET. It uses the *Application* object created in Example 1.

```
Public Sub GetApplicationFeatures(ByRef scApp As SCAPI.Application)
  Dim scBag As SCAPI.PropertyBag
  ' Retrieve all of application environment properties in one call
  scBag = scApp.ApplicationEnvironment.PropertyBag
  ' Retrieve values
  PrintPropertyBag(scBag)
  ' Get an array with categories by using empty string as a category name
  scBag = scApp.ApplicationEnvironment.PropertyBag("", "Categories")
  ' Retrieve a list of categories from the bag
  Dim aCategories() As String
  Dim CategoryName As Object
  If IsArray(scBag.Value("Categories")) Then
    ' Retrieve an array
    aCategories = scBag.Value("Categories")
    If aCategories.Length > 0 Then
      ' Retrieve values on category basis
      For Each CategoryName In aCategories
        ' Get a property bag with values for the category
        scBag = scApp.ApplicationEnvironment.PropertyBag(CategoryName)
        Console.WriteLine("   Values for the " + CategoryName + " category:")
        ' Retrieve values
        PrintPropertyBag(scBag)
      Next CategoryName
    End If
  End If
  ' Get Api Version value Application Api category
  scBag = scApp.ApplicationEnvironment.PropertyBag("Application.Api", "Api   Version")
  ' Retrieve values
  PrintPropertyBag(oBag)
End Sub
' Retrieves and prints values from a property bag
Public Sub PrintPropertyBag(ByRef oBag As SCAPI.PropertyBag)
  Dim Idx As Short
  Dim nIdx1 As Short
  If Not (oBag Is Nothing) Then
    For Idx = 0 To oBag.Count - 1
      If IsArray(oBag.Value(Idx)) Then
        ' Retrieve an array
        If oBag.Value(Idx).Length > 0 Then
          Console.WriteLine(Str(Idx) + ") " + oBag.Name(Idx) + " is an array: ")
          For nIdx1 = 0 To UBound(oBag.Value(Idx))
            Console.WriteLine("    " + oBag.Value(Idx)(nIdx1).ToString)
          Next nIdx1
        End If
        Else
          ' A single value
          Console.WriteLine(Str(Idx) + ") " + oBag.Name(Idx) + " = " +  oBag.Value(Idx).ToString)
```

```
        End If
    Next Idx
End If
End Sub
```

Accessing a Model

An API client accesses model data by working with a pool of available persistence units. A persistence unit is the API concept that describes all data related to a single model. A persistence unit can be accessed and saved to persistence storage, such as a file or a model in a mart. A client manipulates persistence units by using the Persistence Units collection.

The existence of some persistence units in the application is dictated by a context in which an instance of the application was created. For example, in standalone mode, none of the units exist at launch time. Methods from the unit collection interface must be used to accumulate units in the collection. In add-in component mode, the collection contains all the units known to the erwin DM user interface at the time when the client component is activated.

When the client program is terminated, the arrangement for the persistence units in memory for standalone mode is that all units are closed. In add-in component mode, after the client program has ended, the units are still open and available in the erwin DM user interface with the exception of those that were explicitly closed and removed from the persistence unit collection before exiting the program.

Note: For erwin DM, the collection is a snapshot. The collection includes only those units that exist at the moment of collection construction (such as at the moment when the *PersistenceUnits* method of the *ISCApplication* interface was called). An exception to this is units added or deleted from the collection-these changes are reflected. All new collections reflect the changes as well.

Using the API as an Add-in Tool

When the API client is a DLL that is invoked by clicking Add-Ins from the Tools menu, the client runs within the environment of erwin DM. As a result, all the models that are currently open within erwin DM are populated in the *PersistenceUnits* property of the *ISCApplication* interface, when an instance of the interface is created.

To iterate through the models that are currently open in erwin DM, you can use the *ISCApplication* interface, *ISCPersistenceUnitCollection* interface, and the *ISCPersistenceUnit* interface, which are described in the sections that follow.

ISCApplication Interface

The following table contains information on the *ISCApplication* interface:

Signature	Description	Valid Arguments
ISCPersistenceUnitCollection PersistenceUnits()	Returns a collection of all persistence units loaded in the application	None

ISCPersistenceUnitCollection Interface

The following table contains information on the *ISCPersistenceUnitCollection* interface:

Signature	Description	Valid Arguments
ISCPersistenceUnit Item(VARIANT nIndex)	Passes back a pointer for the PersistenceUnit component identified by its ordered position	Index: ■ VT_UNKNOWN – A pointer to a session. Retrieves the persistence unit associated with the session. ■ VT_I4 – Index within the collection. Collection index is from 0 to size-1. Retrieves the persistence unit in the collection with the given index. ■ VT_BSTR – Application-wide unique persistence unit identifier.
long Count()	Number of persistence units in the collection	None

ISCPersistenceUnit Interface

The following table contains information on the *ISCPersistenceUnit* interface:

Signature	Description	Valid Arguments
BSTR Name()	Returns the name of the persistence unit	None
SC_MODELTYPEID ObjectId()	Returns an identifier for the persistence unit	None

Signature	Description	Valid Arguments
ISCPropertyBag PropertyBag(VARIANT List[optional], VARIANT AsString[optional])	Returns a property bag with the properties of the persistence unit	List: ■ VT_BSTR – Semicolon-separated list of property names. Returns a property bag with the unit properties in the given list. AsString: ■ VT_BOOL – Returns a property bag with all values presented as strings if set to TRUE. Otherwise, the values are presented in its native format.
VARIANT_BOOL HasSession()	Returns TRUE if a unit has one or more sessions connected	None
VARIANT_BOOL IsValid()	Returns TRUE is self is valid	None

Property Bag Members for a Persistence Unit

The following table shows some property names and descriptions for property bag members of an existing persistence unit.

Note: A complete set of available properties is located in the appendix API Interfaces Reference (see page 105).

Property Name	Type	Description
Locator	BSTR	Returns the location of the persistence unit, such as file name. Not available for models without a persistence location, such as new models that were never saved.
Disposition	BSTR	Returns the disposition of the persistence unit, such as read-only.
Persistence_Unit_Id	SC_MODELTYPEID	Retrieves an identifier for the persistence unit.
Model_Type	Long	Retrieves the type of the persistence unit, such as logical, logical-physical, and physical models.
Target_Server Target_Server_Version Target_Server_Minor_Version	Long	Retrieves the target database properties for physical and logical-physical models.
Active_Model	Boolean	TRUE if the persistence unit represents the current model and is active in the erwin DM user interface. Not available for the API in standalone mode.

Property Name	Type	Description
Hidden_Model	Boolean	TRUE if a model window with the persistence unit data is not visible in the erwin DM user interface. Not available for the API in standalone mode.
Active_Subject_Area_and_Stored_Display	SAFEARRAY(BSTR)	Reports names of active Subject Area and Stored Display model objects. This indicates the Subject Area and Stored Display that erwin DM shows on the screen. The returned value is a safe-array with two elements. The first element is a name for the active Subject Area and the second element is for the Stored Display.

ISCPropertyBag Interface

The following table contains information on the *ISCPropertyBag* interface:

Signature	Description	Valid Arguments
long Count()	Returns the number of properties	None
VARIANT Value(VARIANT Property)	Retrieves the indicated property in the bag	Property: ■ VT_BSTR – Name of property. Value of the property with the given name in the property bag. ■ VT_I4 – Zero-based property index. Value of the property with the given index in the property bag.
BSTR Name(long PropertyIdx)	Retrieves the indicated property name with the given index. Range of indices is from 0 to size-1.	None

Example 3

The following example illustrates how to use the API as an add-in tool to iterate through the open models using C++. The example uses the *Application* object created in Example 1:

```
void IteratePersistenceUnits(ISCApplicationPtr & scAppPtr)
{
  ISCPersistenceUnitCollectionPtr scPUnitColPtr;
  scPUnitColPtr = scAppPtr->GetPersistenceUnits();

  ISCPersistenceUnitPtr scPUnit = 0;
  long lCnt = scPUnitColPtr->GetCount();

  for(long i = 0; i < lCnt; i++)
  {
    scPUnit = scPUnitColPtr->GetItem(i);
    CString csName = scPUnit->GetName();  // name of model
    ISCPropertyBagPtr scPropBag = scPUnit->GetPropertyBag("Locator;Active  Model");
    long index = 0;
    CComVariant vPathName = scPropBag->GetValue(ColeVariant(index));  // full
    //path of model
    index = 1;
    CComVariant cActiveModel = scPropBag->GetValue(COleVariant(index)); // true if active model
    // ...
  }
}
```

The following example illustrates how to use the API as an add-in tool to iterate through the open models using Visual Basic .NET. The example uses the *Application* object created in Example 1:

```
Public Sub IteratePersistenceUnits(ByRef scApp As SCAPI.Application)

    Dim scPersistenceUnitCol as SCAPI.PersistenceUnits

    Dim numUnits As Integer
    Dim scPUnit As SCAPI.PersistenceUnit

    scPersistenceUnitCol = scApp.PersistenceUnits

    ' Count open units
    numUnits = scPersistenceUnitCol.Count
    If (numUnits > 0) Then
                For Each scPUnit In scPersistenceUnitCol
                    Dim propBag As SCAPI.PropertyBag

                    propBag = scPUnit.PropertyBag("Locator")
                    Console.WriteLine( persUnit.Name )   ' name of model
                    Console.WriteLine( propBag.Value(0)) ' full path of model
                     ' ...
                Next
    End If
End Sub
```

Using the API as a Standalone Executable

When the API client is a standalone executable, the client runs outside the erwin DM environment. As a result, when the *ISCApplication* interface is created, the *PersistenceUnits* property is an empty collection. Even if erwin DM is running and there are open models, the *PersistenceUnits* property is still empty because the API environment is independent of the erwin DM environment. To get a valid persistence unit, the API client needs to either create a new model or open an existing model.

Creating a Model

To create a new model using the API, you first need to create a new instance of *ISCPropertyBag*. The *ISCPropertyBag* interface is a property bag that is used to hold the properties of the new model. The following properties are used in creating a new model.

Note: A complete set of properties is located in the appendix API Interfaces Reference (see page 105).

Property Name	Type	Description
Model_Type	Long	Sets the type of the persistence unit as follows: ■ 1 – Logical (for logical models; this is the default if no type is provided) ■ 2 – Physical (for physical models) ■ 3 – Combined (for logical/physical models)
Target_Server Target_Server_Version Target_Server_Minor_Version	Long	Sets the target database properties for physical and logical/physical models.

Once the property bag is created and populated, a new persistence unit must be created within the persistence unit collection.

ISCPersistenceUnitCollection Interface

The following table contains information on the *ISCPersistenceUnitCollection* interface:

Signature	Description	Valid Arguments
ISCPersistenceUnit * Create(ISCPropertyBag * PropertyBag, VARIANT ObjectId [optional])	Creates a new unit, and registers the unit with the collection	ObjectId: ■ Empty – The API assigns an ID to the new persistence unit. ■ VT_BSTR – The API assigns the given ID to the new persistence unit.

ISCPropertyBag Interface

The following table contains information on the *ISCPropertyBag* interface:

Signature	Description	Valid Arguments
VARIANT_BOOL Add(BSTR Name, VARIANT Value)	Adds a new property to the bag	Value: All *VARIANT*s are valid. The function returns TRUE if the property was added to the bag, otherwise, it is FALSE.

Example 4

The following example illustrates how to create a new persistence unit using C++. The example uses the *Application* object created in Example 1:

```
ISCPersistenceUnitPtr CreateNewModel(ISCApplicationPtr & scAppPtr)
{
    ISCPersistenceUnitCollectionPtr scPUnitColPtr;
    scPUnitColPtr = scAppPtr->GetPersistenceUnits();

    ISCPropertyBagPtr propBag;
    HRESULT hr =propBag.CreateInstance(__uuidof(SCAPI::PropertyBag));
    if (FAILED(hr))
      return;
    propBag->Add("Name", “Test Model”);
    propBag->Add("ModelType", “Logical”);
    ISCPersistenceUnitPtr scPUnitPtr = scPUnitColPtr->Create(propBag,vtMissing);
    return scPUnitPtr;
}
```

The following example illustrates how to create a new persistence unit using Visual Basic .NET. The example uses the *Application* object created in Example 1:

```
Public Function CreateNewModel(ByRef scApp As SCAPI.Application) As SCAPI.PersistenceUnit
   Dim scPersistenceUnitCol as SCAPI.PersistenceUnits
   scPersistenceUnitCol = scApp.PersistenceUnits

   Dim propBag As New SCAPI.PropertyBag

   propBag.Add("Name", "Test Model")
   propBag.Add("ModelType", 0)
   CreateNewModel = scPersistenceUnitCol.Create(propBag)
End Function
```

Opening an Existing Model

An existing erwin DM model is opened by adding a persistence unit to the persistence unit collection (*ISCPersistenceUnitCollection*). When the API client is an add-in tool, opening a model through the API also opens the model in the erwin DM user interface.

ISCPersistenceUnitCollection Interface

The following table contains information on the *ISCPersistenceUnitCollection* interface:

Signature	Description	Valid Arguments
ISCPersistenceUnit * Add(VARIANT Locator, VARIANT Disposition [optional])	Adds a new persistence unit to the unit collection	Locator: ■ VT_BSTR – Full path to the erwin DM model. This is the model that is loaded into the persistence unit. Disposition: ■ VT_BSTR – Arranges access attributes, such as read-only.

Note: Detailed descriptions of the location and format of the *Disposition* parameters is located in the appendix API Interfaces Reference (see page 105).

Example 5

The following example illustrates how to open an existing model using C++. The example uses the *Application* object created in Example 1:

```
ISCPersistenceUnitPtr OpenModel(ISCApplicationPtr & scAppPtr, CString & csFullPath)
{
    ISCPersistenceUnitCollectionPtr scPUnitColPtr;
    scPUnitColPtr = scAppPtr->GetPersistenceUnits();
    ISCPersistenceUnitPtr scPUnitPtr = scPUnitColPtr- >Add(COleVariant(csFullPath));
    return scPUnitPtr;
}
```

The following example illustrates how to open an existing model using Visual Basic .NET. The example uses the *Application* object created in Example 1:

```
Public Function OpenModel(ByRef scApp As SCAPI.Application, _
                fullModelPath As String) As SCAPI.PersistenceUnit
  Dim scPersistenceUnitCol as SCAPI.PersistenceUnits
  scPersistenceUnitCol = scApp.PersistenceUnits

  OpenModel = scPersistenceUnitCol.Add(fullModelPath)
End Sub
```

Opening a Session

Before the objects of a model can be accessed using the API, an *ISCSession* instance must first be established for the *ISCPersistenceUnit* of the model. To open a session for a persistence unit, add a new *ISCSession* to the *ISCSessionCollection*, and then open the *ISCPersistenceUnit* in the new session.

ISCSessionCollection Interface

The following table contains information on the *ISCSessionCollection* interface:

Signature	Description	Valid Arguments
ISCSession * Add()	Constructs a new, closed Session object, and adds it to the collection	None

ISCSession Interface

The following table contains information on the *ISCSession* interface:

Signature	Description	Valid Arguments
VARIANT_BOOL Open(IUnknown * Unit, VARIANT Level [optional], VARIANT Flags [optional])	Binds self to the persistence unit identified by the *Unit* parameter	Unit: ■ Pointer to a persistence unit that was loaded. Attaches the persistence unit to the session. Level: ■ Empty – Defaults to data level access (*SCD_SL_M0*). ■ *SCD_SL_M0* – Data level access. Flags: ■ Empty – Defaults to *SCD_SF_NONE*. ■ *SCD_SF_NONE* – Specifies that other sessions can have access to the attached persistence unit. ■ *SCD_SF_EXCLUSIVE* – Specifies that other sessions cannot have access to the attached persistence unit.

Example 6

The following example illustrates how to open a session using C++. The example uses the *Application* object created in Example 1 and the *CreateNewModel* function from Example 4:

```
ISCSessionPtr OpenSession(ISCApplicationPtr & scAppPtr)
{
    ISCSessionCollectionPtr scSessionColPtr = scAppPtr->GetSessions();
    ISCSessionPtr scSessionPtr = scSessionColPtr->Add();  // add a new session
    ISCPersistenceUnitPtr scPUnitPtr = CreateNewModel(scAppPtr); // From Example 4

    CComVariant varResult = scSessionPtr->Open(scPUnitPtr, (long) SCD_SL_M0); // open unit
    if (varResult.vt == VT_BOOL && varResult.boolVal == FALSE)
      return NULL;
    return scSessionPtr;
}
```

The following example illustrates how to open a session using Visual Basic .NET. The example uses the *Application* object created in Example 1 and the *CreateNewModel* function from Example 4:

```
Public Function OpenSession(ByRef scApp As SCAPI.Application) As SCAPI.Session
    Dim scSessionCol As SCAPI.Sessions
    Dim scPUnit As SCAPI.PersistenceUnit
    scSessionCol = scApp.Sessions
    OpenSession = scSessionCol.Add  'new session

    scPUnit = CreateNewModel(scApp)  ' From Example 4
    scSession.Open(scPUnit, SCD_SL_M0)  ' open the persistence unit
End Sub
```

Accessing a Model Set

A persistence unit contains data as a group of linked model sets. The model sets are arranged in a tree-like hierarchy with a single model set at the top.

The top model set in a persistence unit contains the bulk of modeling data. The erwin DM API uses the abbreviation *EMX* to identify the top model set.

The *EMX* model set owns a secondary model set, abbreviated as *EM2*, that contains user options and user interface settings.

The *ISCSession* interface allows you to open the top model set by simply providing a pointer to the *ISCPersistenceUnit* interface in *ISCSession::Open* call.

It is possible to iterate over all model sets constituting a persistence unit. While iterating, a pointer to the *ISCModelSet* interface can be used to open a session with the particular model set. This is done by submitting the pointer to *ISCSession::Open* call as the first parameter, instead of a persistence unit.

The *ModelSet* property of the *ISCPersistenceUnit* interface provides the starting point for iteration over a persistence unit's model sets. The use of the *OwnedModelSets* property of *ISCModelSet* allows you to iterate over the next level of model sets in the persistence unit.

ISCPersistenceUnit Interface

The following table contains information on the *ISCPersistenceUnit* interface:

Signature	Description	Valid Arguments
ISCModelSet * ModelSet()	Passes back a pointer on the top model set in the Persistence Unit.	None

ISCModelSet Interface

The following table contains information on the *ISCModelSet* interface:

Signature	Description	Valid Arguments
ISCModelSetCollection * OwnedModelSets()	Provides a collection with directly owned model sets.	None

ISCModelSetCollection Interface

The following table contains information on the *ISCModelSetCollection* interface:

Signature	Description	Valid Arguments
ISCModelSet * Item(VARIANT nIndex)	Passes back a pointer for a *ModelSet* component.	nIndex: ■ VT_I4 – Index of a model set in the model set collection. The index is zero-based. ■ VT_BSTR – Model set identifier. ■ VT_BSTR – Class identifier for metadata associated with a model set. ■ VT_BSTR – Class name for metadata associated with a model set.

Note: For information about metadata class identifiers and names, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

ISCSession Interface

The following table contains information on the *ISCSession* interface:

Signature	Description	Valid Arguments
VARIANT_BOOL Open(IUnknown * ModelSet, VARIANT Level [optional], VARIANT Flags [optional])	Binds self to the model set identified by the *ModelSet* parameter	ModelSet: ■ Pointer to a model set from a persistence unit that was loaded. Attaches the model set to the session. Level: ■ Empty – Defaults to data level access (*SCD_SL_M0*). ■ *SCD_SL_M0* – Data level access. Flags: ■ Empty – Defaults to *SCD_SF_NONE*. ■ *SCD_SF_NONE* – Other sessions can have access to the attached persistence unit. ■ *SCD_SF_EXCLUSIVE* – Other sessions cannot have access to the attached persistence unit.

Example 7

The following example illustrates how to open a session with the EM2 model of a persistence unit using C++. The example uses the *Application* object created in Example 1 and the *CreateNewModel* function from Example 4:

```
void OpenEM2(ISCApplicationPtr & scAppPtr)
{
  ISCSessionCollectionPtr scSessionColPtr = scAppPtr->GetSessions();
  ISCPersistenceUnitPtr scPUnitPtr = CreateNewModel(scAppPtr); // From Example 4

  ISCModelSetPtr scEMXModelSetPtr = scPUnitPtr->ModelSet(); // Collect the   top model set
  ISCModelSetPtr scEM2ModelSetPtr = scEMXModelSetPtr- >GetOwnedModelSets()->GetItem(COleVariant("EM2"));
  if (scEM2ModelPtr != NULL)
  {
    ISCSessionPtr scSessionPtr = scSessionColPtr->Add();  // add a new session
    CComVariant varResult = scSessionPtr->Open(scEM2ModelSetPtr);
    if (varResult.vt == VT_BOOL && varResult.boolVal == FALSE)
      return;

  // ...
}
```

The following example illustrates how to open a session with the EM2 model of a persistence unit using Visual Basic .NET. The example uses the *Application* object created in Example 1 and the *CreateNewModel* function from Example 4:

```
Public Sub OpenEM2(ByRef scApp As SCAPI.Application )

  Dim scSession As SCAPI.Session
  Dim scEMXModelSet As SCAPI.ModelSet
  Dim scEM2ModelSet As SCAPI.ModelSet
  Dim scPUnit As SCAPI.PersistenceUnit

  scSessionCol = scApp.Sessions
  scPUnit = CreateNewModel(scApp)  ' From Example 4

  ' Access the top model set - of EMX type
  scEMXModelSet = persUnit.ModelSet
  ' Access an owned EM2 model set by class name
  scEM2ModelSet = scEMXModelSet.OwnedModelSets("EM2")
  Console.WriteLine(vbTab + " Access EM2 Model Set by class name" + scEM2ModelSet.Name + _" Id " +
scEM2ModelSet.ModelSetId)
  Console.WriteLine(vbTab + vbTab + " Class Name " + scEM2ModelSet.ClassName + _" Class id " + scEM2ModelSet.ClassId)
  scSession = scSessionCol.Add  ' new session
  scSession.Open(scEM2ModelSet, SCD_SL_M0)  ' connect EM2 to a session
  '...
End Sub
```

Accessing Objects in a Model

You can access model objects through the *ModelObjects* property in an active *ISCSession* instance. The *ModelObjects* property is a collection of all model objects associated with the persistence unit of the session. The *ModelObjects* property is an instance of the *ISCModelObjectCollection*. Iteration through an instance of *ISCModelObjectCollection* is done in a depth-first fashion, and returns instances of *ISCModelObject*.

The following sections describe the interfaces used to access model objects.

ISCSession Interface

The following table contains information on the *ISCSession* interface:

Signature	Description	Valid Arguments
ISCModelObjectCollection * ModelObjects()	Creates a *ModelObject* collection for the session	None

ISCModelObjectCollection Interface

The following table contains information on the *ISCModelObjectCollection* interface:

Signature	Description	Valid Arguments
long Count()	Number of objects in the collection	None
IUnknown _NewEnum()	Constructs an instance of the collection enumerator object	None

ISCModelObject Interface

The following table contains information on the *ISCModelObject* interface:

Signature	Description	Valid Arguments
BSTR ClassName()	Returns the class name of the current object	None
SC_OBJID ObjectId()	Uniquely identifies the current object	None

Signature	Description	Valid Arguments
BSTR Name()	Returns the name or a string identifier of the current object	None
SC_CLSID ClassId()	Returns the class identifier of the current object	None
ISCModelObject * Context()	Passes back the context (parent) of the object	None

Example 8

The following example illustrates how to access model objects using C++. The example uses the *Application* object created in Example 1 and the *OpenSession* function from Example 6:

```
void IterateObjects(ISCApplicationPtr & scAppPtr)
{
  ISCSessionPtr scSessionPtr = OpenSession( scAppPtr );   // From Example 6
  //Make sure the Session Ptr is Open
  if(!scSessionPtr->IsOpen())
  {
    AfxMessageBox("Session Not Opened");
    return;
  }
  ISCModelObjectCollectionPtr scModelObjColPtr = scSessionPtr >GetModelObjects();
  IUnknownPtr _NewEnum = NULL;
  IEnumVARIANT* ObjCollection;

  _NewEnum = scModelObjColPtr ->Get_NewEnum();
  if (_NewEnum != NULL)
  {
    HRESULT hr = _NewEnum->QueryInterface(IID_IEnumVARIANT, (LPVOID*) &ObjCollection);
    if (!FAILED(hr))
    {
      while (S_OK == ObjCollection->Next(1,&xObject,NULL))
      {
        ISCModelObjectPtr pxItem = (V_DISPATCH (&xObject));
        // ISCModelObject in xObject was AddRefed already. All we need is to
        //attach it to a smart pointer
        xObject.Clear();
        // Process the Item
        CString csName = (LPSTR) pxItem->GetName();
        CString csID = (LPSTR) pxItem->GetObjectId();
        CString csType = (LPSTR) pxItem->GetClassName();
        // ...
      }
```

```
    if (ObjCollection)
        ObjCollection->Release();
  }
}
```

The following example illustrates how to access model objects using Visual Basic .NET. The example uses the *Application* object created in Example 1 and the *OpenSession* function from Example 6:

```
Public Sub IterateObjects(ByRef scApp As SCAPI.Application)
  Dim scSession As SCAPI.scSession
  Dim scModelObjects As SCAPI.ModelObjects
  Dim scObj As SCAPI.ModelObject

  scSession = OpenSession( scApp )      ' From Example 6
  ' Make sure that the session is open
  If scSession.IsOpen() Them
    scModelObjects = scSession.ModelObjects

    For Each scObj In scModelObjects
      Console.WriteLine( scObj.Name )
      Console.WriteLine( scObj.ObjectId )
      Dubug.WriteLine( scObj.ClassName )
    Next
  End If
End Sub
```

Accessing a Specific Object

You can directly access model objects in an *ISCModelObjectCollection* instance by using the *Item* method of the interface.

ISCModelObjectCollection Interface

The following table contains information on the *ISCModelObjectCollection* interface:

Signature	Description	Valid Arguments
ISCModelObject * Item(VARIANT nIndex, VARIANT Class [optional])	Returns an *IUnknown* pointer for a *Model Object* component identified by the *nIndex* parameter	nIndex: ■ VT_UNKNOWN – Pointer to the *ISCModelObject* interface. Given object is returned from the collection. ■ VT_BSTR – ID of an object. The object with the given identifier is returned from the collection. ■ VT_BSTR – Name of an object. If the name of an object is used, the *Class* parameter must also be used. The object with the given name and given *Class* type is returned from the collection. Class: ■ Empty – The object specified by *nIndex* is returned from the collection. ■ VT_BSTR – Name of a class. Must be used if the *nIndex* parameter is the name of an object. Returns the object with the given name and given *Class*. ■ VT_BSTR – Class ID of object type. Must be used if the *nIndex* parameter is the name of an object. Returns the object with the given name and given *Class* identifier.

Note: For information about valid object class names and identifiers, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin Data Modeler installation folder.

Example 9

The following example illustrates how to access a specific object using C++. The example uses a *Session* object from Example 6:

```
void GetObject(ISCSessionPtr & scSessionPtr, CString & csID)
{
  ISCModelObjectCollectionPtr scModelObjColPtr = scSessionPtr->GetModelObjects();
  ISCModelObjectPtr scObjPtr = scModelObjColPtr->GetItem(COleVariant(csID));
  // ...
}
```

The following example illustrates how to access a specific object using Visual Basic .NET. The example uses a *Session* object from Example 6:

```
Public Sub GetObject(ByRef scSession As SCAPI.Session, ByRef objID As String)
  Dim scObjCol as SCAPI.ModelObjects
  Dim scObj as SCAPI.ModelObject

  scObjCol = scSession.ModelObjects
  scObj = scObjCol.Item(objID)  ' retrieves object with given object ID
End Sub
```

Filtering Object Collections

You can create subsets of a collection by using *ISCModelObjectCollection::Collect* method. The *Collect* method creates a new instance of the *Model Objects* collection component based on the filtering criteria specified in the parameters of the method. The filtering criteria is optional, and any number of combinations of criteria can be used.

ISCModelObjectCollection Interface

The following table contains information on the *ISCModelObjectCollection* interface:

Signature	Description	Valid Arguments
ISCModelObjectCollection * Collect(VARIANT Root, VARIANT ClassId [optional], VARIANT Depth [optional], VARIANT MustBeOn [optional], VARIANT MustBeOff [optional])	Creates a Model Objects collection, which represents a subcollection of itself. The method creates a valid collection even though the collection may be empty.	Root: ■ VT_UNKNOWN – ISCModelObject pointer of the root object. Returns the descendants of the given object. ■ VT_BSTR – The Object ID of the root object. Returns the descendants of the object with the given object identifier. ClassId: ■ VT_ARRAY|VT_BSTR – SAFEARRAY of class IDs. Returns the descendants of the root with the given object class identifiers. ■ VT_ARRAY|VT_BSTR – SAFEARRAY of class names. Returns the descendants of the root with the given object class name. ■ VT_BSTR – Class ID. Returns the descendants of the root with the given object class identifier. ■ VT_BSTR – Semicolon delimited list of class IDs. Returns the descendants of the root with the given class identifiers. ■ VT_BSTR – Class name. Returns the descendants of the root with the given class name. ■ VT_BSTR – Semicolon delimited list of class names. Returns the descendants of the root with the given class names. ■ Empty – Returns all descendants regardless of class type.

Signature	Description	Valid Arguments
		Depth: ■ VT_I4 – Maximum depth. Returns the descendants of the root at a depth no more than the given depth. A depth of -1 represents unlimited depth. ■ Empty – Returns all descendants of the root (unlimited depth). MustBeOn: ■ VT_I4 – Returns the descendants of the root with the given object flags set. ■ Empty – Defaults to SCD_MOF_DONT_CARE. MustBeOff: ■ VT_I4 – Returns the descendants of the root that do not have the given object flags set. ■ Empty – Defaults to SCD_MOF_DONT_CARE.

Note: For information about valid object class names and identifiers, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin Data Modeler installation folder. More information about *SC_ModelObjectFlags* is located in the appendix API Interfaces Reference (see page 105).

The following sections show the code examples for the different filters.

Example 10

The following example illustrates the *Object Type* filter using C++. The example uses the *Session* object from Example 6 and creates a collection of objects of *csType* type, owned by the *rootObj* object:

```
void FilterObjects(ISCSessionPtr scSessionPtr, ISCModelObjectPtr & rootObj,
CString & csType)
{
  ISCModelObjectCollectionPtr scModelObjectsPtr;
scModelObjectsPtr = scSessionPtr->GetModelObjects()->Collect(rootObj->GetObjectId(), COleVariant(csType));
  // ...
}
```

The following example illustrates the *Object Type* filter using Visual Basic .NET. The example uses the *Session* object from Example 6 and creates a collection of objects of *csType* type, owned by the *rootObj* object:

```
Public Sub FilterObjects(ByRef scSession As SCAPI.Session, _
            ByRef rootObj As SCAPI.ModelObject, ByRef objType as String)

  Dim scModelObjects As SCAPI.ModelObjects
  scModelObjects = scSession.ModelObject.Collect(rootObj, objType)
  ' scModelObjects will contain only objects of type objType

End Sub
```

Example 11

The following example illustrates the *Depth* filter using C++:

```
void FilterObjects(ISCSessionPtr scSessionPtr, ISCModelObjectPtr & rootObj,
CString & csType, long depth)
{
  ISCModelObjectCollectionPtr scModelObjectsPtr;
  scModelObjectsPtr = scSessionPtr->GetModelObject()->
      Collect(rootObj->GetObjectId(), COleVariant(csType),depth);
  // ...
}
```

The following example illustrates the *Depth* filter using Visual Basic .NET:

```
Public Sub FilterObjects(ByRef scSession As SCAPI.Session, _
      ByRef rootObj As SCAPI.ModelObject, ByRef classID As String, depth As Integer)

  Dim scModelObjects As SCAPI.ModelObjects
  scModelObjects = scSession.ModelObjects.Collect(rootObj, classID, depth)

End Sub
```

Example 12

The following example illustrates the *MustBeOn/MustBeOff* filter using C++. The example uses the *Session* object from Example 6:

```
void FilterObjects(ISCSessionPtr scSessionPtr, ISCModelObjectPtr & rootObj, long depth)
{
   ISCModelObjectCollectionPtr scModelObjectsPtr;
   scModelObjectsPtr = scSessionPtr->GetModelObjects()->
      Collect(rootObj->GetObjectId(), vtMissing, depth, SCD_MOF_USER_DEFINED);
   // ...
}
```

The following example illustrates the *MustBeOn/MustBeOff* filter using Visual Basic .NET. The example uses the *Session* object from Example 6:

```
Public Sub FilterObjects(ByRef scSession As SCAPI.Session, _
        ByRef rootObj As SCAPI.ModelObject, depth As Integer)

   Dim scModelObjects As SCAPI.ModelObjects
   scModelObjects = scSession.ModelObjects.Collect(rootObj, , depth, SCD_MOF_USER_DEFINED)

End Sub
```

The following example illustrates how to create a note through API:

```
Sub updateAttribute()


 ' This Creates an Instance of SCApplication

 Set SCApp = CreateObject("erwin9.SCAPI")


  'Declare a variable as a FileDialog object.

 Dim fd As FileDialog

 'Create a FileDialog object as a File Picker dialog box.

 Set fd = Application.FileDialog(msoFileDialogFilePicker)


 fd.AllowMultiSelect = False

 fd.Filters.Clear

 fd.Filters.Add "erwin File", "*.erwin", 1

 If (fd.Show = -1) Then

    strFileName = fd.SelectedItems.Item(1)

 Else

    Exit Sub

 End If
```

```
'Set the object variable to Nothing.
Set fd = Nothing

'strFileName = "C:\models\test03.erwin"

 '  This is the name of the .erwin Model that needs to be updated
Set SCPUnit = SCApp.PersistenceUnits.Add("erwin://" & strFileName)
Set SCSession = SCApp.Sessions.Add
SCSession.Open (SCPUnit)
Set SCRootObj = SCSession.ModelObjects.Root
Set SCEntObjCol = SCSession.ModelObjects.Collect(SCRootObj, "Entity")
Dim nTransId
nTransId = SCSession.BeginNamedTransaction("Test")

  For Each oEntObject In SCEntObjCol
   On Error Resume Next
   Set oEntCol = SCSession.ModelObjects.Collect(oEntObject, "Attribute")
    For Each oAttObject In oEntCol
       Set oUserNote = SCSession.ModelObjects.Collect(oAttObject).Add("Extended_Notes")
       oUserNote.Properties("Comment").Value = "Test note1"
       oUserNote.Properties("Note_Importance").Value = "0"    'enum {0|1|2|3|4|5}
       oUserNote.Properties("Status").Value = "1"          'enum {1|2|3}
    Next oAttObject

  Next oEntObject
```

```
SCSession.CommitTransaction (nTransId)

SCSession.Close


' Save the model

Call SCPUnit.Save("erwin://" & strFileName)

MsgBox "Incremental-Save successfully"

SCApp.Sessions.Remove (SCSession)


SCApp.PersistenceUnits.Clear


SCPUnit = Null


SCSession = Null


End Sub
```

Accessing Object Properties

You can access the properties of an object through the Properties property of *ISCModelObject*. The *Properties* property is an instance of *ISCModelPropertyCollection*. The *ISCModelPropertyCollection* contains instances of *ISCModelProperty*.

Iteration of Properties

This section describes the interfaces involved with the iteration of properties.

ISCModelObject Interface

The following table contains information on the *ISCModelObject* interface:

Signature	Description	Valid Arguments
ISCModelPropertyCollection * Properties()	Returns a property collection of all available properties	None

ISCModelPropertyCollection Interface

The following table contains information on the *ISCModelPropertyCollection* interface:

Signature	Description	Valid Arguments
Long Count()	Number of properties in the collection	None
IUnknown _NewEnum()	Constructs an instance of the collection enumerator object	None

ISCModelProperty Interface

The following table contains information on the ISCModelProperty interface:

Signature	Description	Valid Arguments
BSTR ClassName()	Returns the class name of the property	None
SC_CLSID ClassId()	Returns the class identifier of the property	None
Long Count()	Contains the number of values in the property	None
BSTR FormatAsString()	Formats the property value as a string	None

Example 13

The following example illustrates the iteration of properties using C++. The example uses a *Model Object* object from Example 9:

```
void IterateObjectProperties(ISCModelObjectPtr & scObjPtr)
{
  ISCModelPropertyCollectionPtr propColPtr = scObjPtr->GetProperties();

  // Iterate over the Collection
  IUnknownPtr _NewEnum = NULL;
  IEnumVARIANT* propCollection;

  _NewEnum = propColPtr->Get_NewEnum();
  if (_NewEnum != NULL)
  {
    HRESULT hr = _NewEnum->QueryInterface(IID_IEnumVARIANT, (LPVOID*) &propCollection);
    if (!FAILED(hr))
    {
      COleVariant xObject;
      while (S_OK == propCollection->Next(1,&xObject,NULL))
      {
        ISCModelPropertyPtr scObjPropPtr = (V_DISPATCH (&xObject));
        xObject.Clear();
        if (scObjPropPtr.GetInterfacePtr())
        {
          CString csPropName = (LPSTR) scObjPropPtr->GetClassName();
          CString csPropVal= (LPSTR) scObjPropPtr->FormatAsString();
          // ...
        }
      } // property iteration
    }
    if (propCollection)
      propCollection->Release();
  }
}
```

The following example illustrates the iteration of properties using Visual Basic .NET. The example uses a *Model Object* object from Example 9:

```
Public Sub IterateObjectProperties(ByRef scObj As SCAPI.ModelObject)

  Dim scObjProperties As SCAPI.ModelProperties
  Dim scObjProp As SCAPI.ModelProperty
  scObjProperties = scObj.Properties
  For Each scObjProp In scObjProperties
    Debug.WriteLine( scObjProp.ClassName )
    Console.WriteLine( scObjProp.Name )
  Next

End Sub
```

ISCModelProperty Interface

The following table contains information on the *ISCModelProperty* interface:

Signature	Description	Valid Arguments
long Count()	Contains the number of values in the property. For scalar properties, the number of values in the property is always one. It is possible to have a non-scalar property with no elements. In this case, the number of values in the property will be zero.	None
SC_ModelPropertyFlags Flags()	Returns the flags of the property.	None
VARIANT Value(VARIANT ValueId [optional], VARIANT ValueType [optional])	Retrieves the indicated property value in the requested format.	ValueId: ■ Empty – Valid for a scalar property only. ■ VT_I4 – Zero-based index within a homogeneous array. The value of the member indicated by this index is returned. ValueType: ■ Empty – Indicates a native datatype for a return value. ■ SCVT_DEFAULT – Indicates a native datatype for a returned value. ■ SCVT_BSTR – Requests conversion to a string for a returned value.

Note: More information about SC_ModelPropertyFlags is located in the Enumerations (see page 159) section. More information about property datatypes is located in the SC_ValueTypes (see page 162) section.

Example 14

The following example illustrates how to access scalar property values using C++. The example uses a *Model Property* object from Example 13:

```
void GetScalarProperty(ISCModelPropertyPtr & scObjPropPtr)
{
  if (scObjPropPtr->GetCount() <= 1)
  {
    _bstr_t bstrPropVal= scObjPropPtr->FormatAsString();
    // ...
  }
}
```

The following example illustrates how to access scalar property values using Visual Basic .NET. The example uses a *Model Property* object from Example 13:

```
Public Sub GetPropertyElement(ByRef scObjProp As SCAPI.ModelProperty)

   If (scObjProp.Flags And SCAPI.SC_ModelPropertyFlags.SCD_MPF_NULL) Then
     Console.WriteLine( "The value is Null" )
   Else
    If (scObjProp.Flags And SCAPI.SC_ModelPropertyFlags.SCD_MPF_SCALAR) Then
       Console.WriteLine( scObjProp.Value.ToString() )
    Else
        For j = 0 To scObjProp.Count-1
              Console.WriteLine( scObjProp.Value(j).ToString() )
        Next
    End If
   End If

End Sub
```

Iterating Over Non-Scalar Property Values

The properties that contain multiple values (either homogeneous or heterogeneous) are non-scalar properties. To access the individual values of a non-scalar property, the *PropertyValues* member of the *ISCModelProperty* interface is used. The *PropertyValues* member is an instance of *ISCPropertyValueCollection*. Each member of *ISCPropertyValueCollection* is an instance of *ISCPropertyValue*. The *ValueId* member of the *ISCPropertyValue* interface identifies the individual property values in a non-scalar property. *ValueId* can either be a zero-based index or the name of the non-scalar property value member if the property type is a structure.

ISCModelProperty Interface

The following table contains information on the *ISCModelProperty* interface:

Signature	Description	Valid Arguments
ISCPropertyValueCollection * PropertyValues()	Returns the values for the property	None

ISCPropertyValueCollection Interface

The following table contains information on the *ISCPropertyValueCollection* interface:

Signature	Description	Valid Arguments
long Count()	Number of values in the collection	None
IUnknown _NewEnum()	Constructs an instance of the collection enumerator object	None

ISCPropertyValue Interface

The following table contains information on the *ISCPropertyValue* interface:

Signature	Description	Valid Arguments
VARIANT ValueId(VARIANT ValueType [optional])	Uniquely identifies the value in a non-scalar property.	ValueType: ■ SCVT_I2 – If the property is non-scalar, the value of the property index is returned. ■ SCVT_I4 – If the property is non-scalar, the value of the property index is returned. ■ SCVT_BSTR – The name of the non-scalar property member if it is available, or else the index of the member is returned. ■ SCVT_DEFAULT – If the property is non-scalar, the value of the property index is returned. ■ Empty – Defaults to SCVT_DEFAULT.

Signature	Description	Valid Arguments
SC_CLSID PropertyClassId()	Returns the class identifier of the current property	None
BSTR PropertyClassName()	Returns the class name of the current property	None
VARIANT Value(VARIANT ValueType [optional])	Converts the current value to the passed value type.	ValueType: ■ SCVT_DEFAULT – Indicates a value in the native format. ■ SCVT_BSTR – String representation of the property value. ■ Target Type – Identifies a target for a type conversion. ■ Empty – Defaults to SCVT_DEFAULT.
SC_ValueTypes ValueType()	Passes back the identifier of the value default type	None
SC_ValueTypes ValueIdType()	Passes back the identifier of the value identifier default type	None
SC_ValueTypes * GetSupportedValueTypes()	Groups a list of supported value types and returns it as a *SAFEARRAY*	None
SC_ValueTypes * GetSupportedValueIdTypes()	Groups a list of supported value types for the current value identifier and returns it as a *SAFEARRAY*	None

Note: More information about value datatypes is located in the SC_ValueTypes (see page 162) section.

Example 15

The following example illustrates how to access non-scalar property values using C++. The example uses a *Model Property* object from Example 13:

```
void IterateNonScalarProperties(ISCModelPropertyPtr & scObjPropPtr)
{
  if (scObjPropPtr->GetCount() > 1)
  {
    ISCPropertyValueCollectionPtr propVals = scObjPropPtr->GetPropertyValues();
    long numVals = propVals->GetCount();
    for (long i = 0; i < numVals; i++)
    {
      ISCPropertyValuePtr propValPtr = propVals->GetItem(COleVariant(i));
      VARIANT valType;
      V_VT(&valType) = VT_I4;
      V_I4(&valType) = SCVT_BSTR;
      bstr_t bstrPropVal = propValPtr->GetValue(valType);

      // ...
    }
  }
}
```

The following example illustrates how to access non-scalar property values using Visual Basic .NET. The example uses a *Model Property* object from Example 13:

```
Public Sub IterateNonScalarProperties(ByRef scObjProp As SCAPI.ModelProperty)
  Dim scPropValue as SCAPI.PropertyValue

  If (scObjProp.Count > 1) Then
    For Each scPropValue In scObjProp.PropertyValues
            If (scPropValue.ValueIdType = SCVT_BSTR) Then
              Console.WriteLine( scPropValue.ValueId(SCVT_BSTR),": ", _
        scPropValue.Value.ToString())
            Else
         Console.WriteLine (scPropValue.Value.ToString())
            End If
    Next
  End If
End Sub
```

Accessing a Specific Property

For non-scalar properties, you can directly access individual values by using the *Item* method of *ISCPropertyValueCollection*.

ISCPropertyValueCollection Interface

The following table contains information on the *ISCPropertyValueCollection* interface:

Signature	Description	Valid Arguments
ISCPropertyValue * Item(VARIANT ValueId)	Returns a single value from the property value collection	ValueId: ■ VT_I4 – Index of the member in a non-scalar property. ■ VT_BSTR – Name of a member in a non-scalar property.

Note: For r7.3, erwin DM does not support naming of non-scalar property members.

Example 16

The following example illustrates how to access a specific property using C++. The example uses a *Model Object* object from Example 9:

```
// This function retrieves a specific value with the given index from the property with the
// given name.
ISCPropertyValuePtr GetPropValue(ISCModelObjectPtr & scObjPtr, CString & csName, int index)
{
   ISCModelPropertyCollectionPtr propColPtr = scObjPtr->GetProperties();
   ISCModelPropertyPtr scObjPropPtr = propColPtr->GetItem(COleVariant(csName));
   ISCPropertyValueCollectionPtr propVals = scObjPropPtr->GetPropertyValues();
   return propVals->GetItem(COleVariant(index));
}
```

The following example illustrates how to access a specific property using Visual Basic .NET. The example uses a *Model Object* object from Example 9:

```
' This function retrieves a specific value with the given index from the property with the
' given name.
Public Function GetPropValue(ByRef scObj As SCAPI.ModelObject, ByRef propName As String, _index As Integer) As
SCAPI.PropertyValue
   Dim scProp as SCAPI.ModelProperty
   Set scProp = scObj.Properties.Item(propName)
   Set GetPropValue = scProp.PropertyValues.Item(index)

End Function
```

Filtering Properties

Subsets of an instance of *ISCModelPropertyCollection* can be created by using its *CollectProperties* method of *ISCModelObject*. The *CollectProperties* method creates a new instance of *ISCModelPropertyCollection* based on the filtering criteria specified in the parameters of the method. By filtering the property collection, you can retrieve properties of a certain class, properties with specified flags set, or properties that do not have specified flags set. The filtering criteria is optional, and any number of combinations of criteria can be used. More information about specific property flags is located in the Enumerations (see page 159) section.

Note: For more information about identifiers used in property classes, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

ISCModelObject Interface

The following table contains information on the *ISCModelObject* interface:

Signature	Description	Valid Arguments
ISCModelPropertyCollection * CollectProperties(VARIANT ClassIds [optional], VARIANT MustBeOn [optional], VARIANT MustBeOff [optional])	Returns a property collection of the type that you require	ClassIds: ■ Empty – All properties of the object are returned. ■ VT_ARRAY|VT_BSTR – SAFEARRAY of property class IDs. Returns the properties with the given property class identifiers. ■ VT_ARRAY|VT_BSTR – SAFEARRAY of property names. Returns the properties with the given class names. ■ VT_BSTR – ID of a property class. Returns the property with the given property class identifier. ■ VT_BSTR – Name of a property. Returns the property with the given class name. ■ VT_BSTR – List of property class IDs delimited by semicolons. Returns the properties with the given property class identifiers. ■ VT_BSTR – List of property names delimited by semicolons. Returns the properties with the given class names. MustBeOn: ■ Empty – Defaults to SCD_MPF_DONT_CARE and returns all properties. ■ VT_I4 – SC_ModelObjectFlags flags that must be on. Returns the properties with the specified flags set. MustBeOff: ■ Empty – Defaults to SCD_MPF_NULL and returns all properties. ■ VT_I4 – SC_ModelObjectFlags flags that must be off. Returns the properties that do not have the specified flags set.

Note: Setting certain filter criteria can influence the effectiveness of data retrieving. For example, setting the *MustBeOn* filter to *SCD_MPF_DERIVED* builds a collection with only the calculated and derived properties. Requests to evaluate the calculated and derived properties will reduce performance while iterating over the collection. However, setting the *MustBeOff* filter to the same value, *SCD_MPF_DERIVED*, which excludes the calculated and derived properties, improves performance.

Example 17

The following example illustrates how to filter properties using C++. The example uses a *Model Object* object from Example 9:

```
void GetProperties(ISCModelObjectPtr & scObjPtr)
{

  ISCModelPropertyCollectionPtr propColPtr;

  propColPtr = scObjPtr->GetProperties();   // no filtering

  VARIANT valFlags;
  V_VT(&valFlags) = VT_I4;
  V_I4(&valFlags) = SCD_MPF_SCALAR;

  propColPtr = scObjPtr->CollectProperties(vtMissing, valFlags, vtMissing);  // scalar properties only
  propColPtr = scObjPtr->CollectProperties(vtMissing, vtMissing, valType);  // non-scalar properties only
}
```

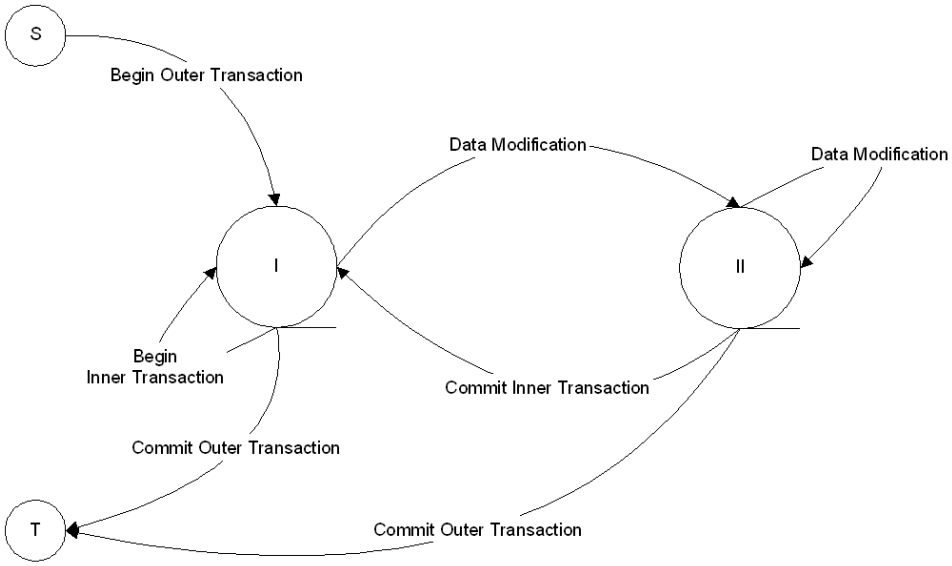
The following example illustrates how to filter properties using Visual Basic .NET. The example uses a *Model Object* object from Example 9:

```
Public Sub( ByRef scObj As SCAPI.ModelObject )
  Dim scObjProperties As SCAPI.ModelProperties

  scObjProperties = scObj.Properties   ' no filtering

  scObjProperties = scObj.CollectProperties(, SCD_MPF_SCALAR)  ' scalar properties only

  scObjProperties = scObj.CollectProperties(, , SCD_MPF_SCALAR)  ' non-scalar properties only
End Sub
```

Modifying the Model Using Session Transactions

In order to make modifications to a model, session transactions must be used. Prior to making a modification, either *BeginTransaction()* or *BeginNamedTransaction()* must be called. Once all the modifications are completed, *CommitTransaction()* must be called.

Note: Nested transactions and rollbacks are supported with certain limitations. The limitation is illustrated in the following state diagram:



After the beginning of an outer transaction, the API is in State I of the diagram. A new nested transaction can be opened or the outer transaction can be closed. Any operation other than the open or close of a transaction, such as creating, modifying objects, properties, and so on, will transfer the API to State II. In that state further modifications can continue, but no new nested transactions are allowed. The API continues to be in that state until the current transaction is committed or rolled back.

Use of nested transactions allows better control over modification flow. The following examples describe the uses:

Commit Transaction

Carries out enlisted modifications immediately. Therefore, without closing the outer transaction, the small nested transactions can reflect separate steps of the complex changes with the results of the committed transaction instantly available for the consumption by the next step.

Rollback

Cancels out the results of all nested transactions. This includes transactions that were committed before the outer transaction rollback.

Begin Transaction

To indicate that a modification to the model is about to occur, either the *BeginTransaction()* or the *BeginNamedTransaction()* must be called.

ISCSession Interface

The following table contains information on the *ISCSession* interface:

Signature	Description	Valid Arguments
VARIANT BeginTransaction()	Opens a transaction on the session. Returns an identifier of the transaction.	None
VARIANT BeginNamedTransaction(BSTR Name, VARIANT PropertyBag [optional])	Opens a transaction on the session with the given name. Returns an identifier of the transaction.	Name – Provides a name for a new transaction. PropertyBag – Collection of optional parameters for the transaction.

Example 18

The following example illustrates modifying the model using the *Begin Transaction* in C++. The example uses a *Session* object from Example 6:

```
void OpenSession(ISCSessionPtr & scSessionPtr )
{
  variant_t transactionId;  // transaction ID for the session

  VariantInit(&transactionId);
  transactionId = scSessionPtr->BeginTransaction();

  // ...
}
```

The following example illustrates modifying the model using the *Begin Transaction* in Visual Basic .NET. The example uses a *Session* object from Example 6:

```
Public Sub OpenSession( ByRef scSession As SCAPI.Session )
  Dim m_scTransactionId As Variant

  scTransactionId = scSession.BeginNamedTransaction("My Transaction")
End Sub
```

Commit Transaction

The *CommitTransaction()* is used to commit the modifications to the in-memory model.

Note: The Commit only applies to the in-memory model while the API is running. To persist the modifications, the model must be explicitly saved using the *ISCPersistenceUnit::Save()* function.

ISCSession Interface

The following table contains information on the *ISCSession* interface:

Signature	Description	Valid Arguments
VARIANT_BOOL CommitTransaction(VARIANT TransactionId)	Commits the specified transaction	None

Example 19

The following example illustrates modifying the model using the *Commit Transaction* in C++. The example uses a *Session* object from Example 6:

```
void Transaction(ISCSessionPtr & scSessionPtr )
{
  variant_t transactionId;  // transaction ID for the session

  VariantInit(&transactionId);
   transactionId = scSessionPtr->BeginTransaction();

   // Make modifications to the model here ....

   scSessionPtr->CommitTransaction(transactionId);
}
```

The following example illustrates modifying the model using the *Commit Transaction* in Visual Basic .NET. The example uses a *Session* object from Example 6:

```
Public Sub Transaction(ByRef scSession As SCAPI.Session )
  Dim scTransactionId As Variant

  scTransactionId = scSession.BeginTransaction

  ' make modifications here ...

  scSession.CommitTransaction( scTransactionId )
End Sub
```

Creating Objects

The first step in creating a new object is to retrieve the *ISCModelObject* instance of the parent of the new object. From the parent of the new object, retrieve its child objects in an instance of *ISCModelObjectCollection*. Then, add the new object to the child objects collection.

Note: For information about valid object class names and identifiers, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

ISCModelObjectCollection Interface

The following table contains information on the *ISCModelObjectCollection* interface, which is used when you create a new model object:

Signature	Description	Valid Arguments
ISCModelObjectCollection * Collect(VARIANT Root, VARIANT ClassId [optional], VARIANT Depth [optional], VARIANT MustBeOn [optional], VARIANT MustBeOff[optional])	Creates a *Model Objects* collection, which represents a subcollection of itself	Root: ■ VT_UNKNOWN – The *ISCModelObject* pointer of the root object. Returns the descendants of the given object. ■ VT_BSTR – The ID of the root object. Returns the descendants of the object with the given object identifier. ClassId: ■ Empty – Not needed when obtaining the children of an object. Depth: ■ VT_I4 – Set depth to 1 when obtaining the immediate children of an object. MustBeOn: ■ Empty – Not needed when obtaining the children of an object. MustBeOff: ■ Empty – Not needed when obtaining the children of an object.
ISCModelObject * Add(VARIANT Class, VARIANT ObjectId [optional])	Adds an object of type *Class* to the model	Class: ■ VT_BSTR – Name of a class. Creates an object of the given class name. ■ VT_BSTR – Class ID of an object type. Creates an object of the class with the given identifier. ObjectId: ■ Empty – The API assigns an object identifier for a new object. ■ VT_BSTR – ID for a new object. The API assigns the given object identifier to the new object.

Example 20

The following example illustrates how to create objects using C++. The example uses a *Session* object from Example 6:

```
// NOTE: ISCSession::BeginTransaction() must be called prior to calling this
// function
// ISCSession::CommitTransaction() must be called upon returning from this
// function
void CreateObject(ISCSessionPtr & scSessionPtr, CString & csType,
                                ISCModelObjectPtr & parentObj)
{
  variant_t transactionId;  // transaction ID for the session
  VariantInit(&transactionId);
  transactionId = scSessionPtr->BeginTransaction();
  ISCModelObjectCollectionPtr childObjColPtr =
scSessionPtr->GetModelObject()->Collect(parentObj->GetObjectId(),vtMissing,(long)1);  // get
  // child objects
  // Add child object to collection
  ISCModelObjectPtr childObjPtr = childObjColPtr->Add(COleVariant(csType));
  // ...
  scSessionPtr->CommitTransaction(transactionId);
}
```

The following example illustrates how to create objects using Visual Basic .NET. The example uses a *Session* object from Example 6:

```
Public Sub AddNewObject(ByRef scSession As SCAPI.Session, _
        ByRef parentObj As SCAPI.ModelObject, type As String)
  Dim scObj as SCAPI.ModelObject
  Dim scChildObjCol As SCAPI.ModelObjects
  Dim transactionID as Variant

  transactionID = scSession.BeginTransaction
  scChildObjCol = scSession.ModelObjects.Collect(parentObj, , 1)  ' child objects collection
  scObj = scChildObjCol.Add(type)   ' add new object to the child object collection

  scSession.CommitTransaction( transactionID )
End Sub
```

Setting Property Values

To set a property value of a model object, use the *Value* member of an instance of the *ISCModelProperty* interface.

Setting Scalar Property Values

The valid *VARIANT* types that can be used to set a scalar property value is dependent on the type of the property.

Note: For more information, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

ISCModelProperty Interface

The following table contains information on the *ISCModelProperty* interface:

Signature	Description	Valid Arguments
void Value(VARIANT ValueId [optional], VARIANT ValueType [optional], VARIANT Val)	Sets the indicated property value with the given value	ValueId: ■ Empty – Not used when setting scalar properties. ValueType: ■ Empty – Not used. Val: ■ Dependent upon the property type.

Note: For information about valid property values, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

Example 21

The following example illustrates how to set scalar property values using C++. The example uses a *Model Object* object from Example 9 and assumes that a transaction has opened:

```
// NOTE: ISCSession::BeginTransaction() must be called prior to calling this
// function
// ISCSession::CommitTransaction() must be called upon returning from this
// function
void SetNameProperty(ISCModelObjectPtr & scObjPtr, CString & csName)
{
  ISCModelPropertyCollectionPtr propColPtr = scObjPtr->GetProperties();
  CString csPropName = "Name";
  ISCModelPropertyPtr nameProp = propColPtr > GetItem(COleVariant(csPropName));
  if (nameProp != NULL)
    nameProp->PutValue(vtMissing, (long) SCVT_BSTR, csName);
}
```

The following example illustrates how to set scalar property values using Visual Basic .NET. The example uses a *Model Object* object from Example 9 and assumes that a transaction has opened:

```
' NOTE: ISCSession::BeginTransaction() must be called prior to calling this function
' ISCSession::CommitTransaction() must be called upon returning from this function
Public Sub SetScalarPropValue(ByRef scObj As SCAPI.ModelObject, ByRef val As Variant)
   Dim modelProp As SCAPI.ModelProperty
   modelProp = scObj.Properties("Name")
   modelProp.Value = val
End Sub
```

Setting Non-Scalar Property Values

To set a non-scalar property value, you must identify the specific value that you want to set. This is done using the *ValueId* parameter. The *ValueId* can either be the zero-based index of the property value collection or the name of the member if the property is a structure.

Note: For r7.3, erwin DM does not support naming non-scalar property members.

ISCModelProperty Interface

The following table contains information on the *ISCModelProperty* interface:

Signature	Description	Valid Arguments
void Value(VARIANT ValueId [optional], VARIANT ValueType [optional], VARIANT Val)	Sets the indicated property value with the given value	ValueId: ■ VT_I4 – Index for a non-scalar property of which the given value is set. ■ VT_BSTR – Name of a member in a non-scalar property of which the given value is set. ValueType: ■ Empty – Not used. Val: ■ Dependent upon the property type.

Note: For information about valid property values, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

Example 22

The following example illustrates how to set non-scalar property values using C++. The example uses a *Model Object* object from Example 9 and assumes that a transaction has opened:

```
// NOTE: ISCSession::BeginTransaction() must be called prior to calling this
// function
// ISCSession::CommitTransaction() must be called upon returning from this
// function
void SetNameProperty(ISCModelObjectPtr & scObjPtr, CString & csValue)
{
  ISCModelPropertyCollectionPtr propColPtr = scObjPtr->GetProperties();
  CString csPropName = "Non-Scalar";
ISCModelPropertyPtr nameProp = propColPtr > GetItem(COleVariant(csPropName));
  if (nameProp != NULL)
  // Setting the first element
  nameProp->PutValue(COleVariant(0L), (long) SCVT_BSTR, csValue);
}
```

The following example illustrates how to set non-scalar property values using Visual Basic .NET. The example uses a *Model Object* object from Example 9 and assumes that a transaction has opened:

```
' NOTE: ISCSession::BeginTransaction() must be called prior to calling this function
' ISCSession::CommitTransaction() must be called upon returning from this function
Public Sub SetScalarPropValue(ByRef scObj As SCAPI.ModelObject, ByRef val As Variant)
  Dim modelProp As SCAPI.ModelProperty
  modelProp = scObj.Properties("Name")
  Dim index As Long
  Index = 0   ' Setting index to zero
  modelProp.Value(index) = val  ' index is used to access non-scalar property
End Sub
```

Deleting Objects

You can delete an object by removing the *ISCModelObject* interface instance of the object from the instance of *ISCModelObjectCollection*. You identify the object that you want to delete either by its pointer to the interface or by its object identifier.

ISCModelObjectCollection Interface

The following table contains information on the *ISCModelObjectCollection* interface, which is used to delete model objects:

Signature	Description	Valid Arguments
VARIANT_BOOL Remove(VARIANT Object)	Removes the specified model object from a model	Object: ■ VT_UNKNOWN – ISCModelObject * pointer to the object that you want to delete. Removes the given object. ■ VT_BSTR – ID of the object. Removes the object with the given object identifier.

Example 23

The following example illustrates how to delete objects in C++ if there is a model objects collection and that a transaction has opened:

```
CString csID;  // ID of object to be removed
// ...
CComVariant bRetVal = scObjColPtr->Remove(COleVariant(csID));
```

The following example illustrates how to delete objects in Visual Basic .NET if there is a model objects collection and that a transaction has opened:

```
bRetVal = scObjCol.Remove(objID)
```

Deleting Properties and Property Values

Properties are deleted by removing the property from the instance of the *ISCModelPropertyCollection* interface. If the property is non-scalar, the individual property value can be removed by using the *RemoveValue* method of the *ISCModelProperty* interface.

Note: For more information about valid property names and property identifiers, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

The following sections describe the interfaces used to delete model properties and model property values.

ISCModelPropertyCollection Interface

The following table contains information on the *ISCModelPropertyCollection* interface:

Signature	Description	Valid Arguments
VARIANT_BOOL Remove(VARIANT ClassId)	Removes the indicated property from the bound object	ClassId: ■ VT_UNKNOWN – *ISCModelProperty* pointer to the object that you want to remove. Removes the given property. ■ VT_BSTR – Name of the property. Removes the property with the given class name. ■ VT_BSTR – ID of the property. Removes the property with the given class identifier.

ISCModelProperty Interface

The following table contains information on the *ISCModelProperty* interface:

Signature	Description	Valid Arguments
VARIANT_BOOL RemoveValue(VARIANT ValueId [optional])	Removes the specified value from the property	ValueId: ■ Empty – For scalar properties only. ■ VT_I4 – Index of a non-scalar property. Removes the value with the given index in a non-scalar property. ■ VT_BSTR – Name of the property member in a non-scalar property. Removes the value of the non-scalar property member with the given name.
VARIANT_BOOL RemoveAllValues()	Remove all values from the property	None

Example 24

The following example illustrates how to delete scalar properties using C++ if there is a model object and a transaction is open:

```
CString propName("Some Property Name");
// ...
CComVariant bRetVal = scObjPtr->GetProperties()->Remove(COleVariant(propName));
```

The following example illustrates how to delete scalar properties using Visual Basic .NET if there is a model object and a transaction is open:

```
Dim propName As String
propName = "Some Property Name"

bRetVal = scObj.Properties.Remove(propName)
```

Deleting Non-Scalar Property Values

To remove all the values from a non-scalar property, remove the property itself from the *ISCModelPropertyCollection* using the *Remove* method. To remove a specific value from a non-scalar property, use the *RemoveValue* method of the *ISCModelProperty* interface. As with accessing the non-scalar property values, the property value is identified using the *ValueId* parameter. *ValueId* can either be the zero-based index of the value, or the name of the member if the property type is a structure.

Note: For r7.3, erwin DM does not support naming non-scalar property members.

Example 25

The following example illustrates how to delete non-scalar property values using C++ if there is a model object and a transaction is open:

```
ISCModelPropertyCollectionPtr propColPtr = scObjPtr->GetProperties();
CString csPropName = "Some Property Name";
ISCModelPropertyPtr scPropPtr = propColPtr->GetItem(COleVariant(csPropName));
long index;  // index of a member in a non-scalar property
index = 0;   // Set to the first element
// ...
bRetVal = scPropPtr->RemoveValue(index);  // remove single value from the property
```

The following example illustrates how to delete non-scalar property values using Visual Basic .NET if there is a model object and a transaction is open:

```
Dim scProp As SCAPI.ModelProperty
scProp = scObj.Properties("Some Property Name")
bRetVal = scProp.RemoveValue(index)  ' Remove single value from the property
```

Saving the Model

If modifications were made to the erwin DM model, the persistence unit must be saved in order to persist the changes.

ISCPersistenceUnit Interface

The following table contains information on the *ISCPersistenceUnit* interface:

Signature	Description	Valid Arguments
VARIANT_BOOL Save(VARIANT Locator [optional], VARIANT Disposition [optional]	Persists model data to external storage	Locator: ■ VT_BSTR – Full path of the location to store the model. Provides a new location for the persistence unit data source as a string with a file or mart item location, along with the attributes required for successful access to storage. ■ Empty – Indicates the use of the original persistence unit location. Disposition: Specifies changes in access attributes, such as read only.

Example 26

The following example illustrates how to save a model using C++. The example uses a *Persistence Unit* object from Example 5:

```
void Save( ISCPersistenceUnitPtr & scPUnitPtr )
{
  ISCPropertyBagPtr propBag = scPUnitPtr->GetPropertyBag ("Locator");
  long index = 0;
  _bstr_t bstrFileName = propBag->GetValue(COleVariant(index));

  // Change bstrFileName to a new location
  scPUnitPtr->Save(bstrFileName);
}
```

The following example illustrates how to save a model using Visual Basic .NET. The example uses a *Persistence Unit* object from Example 5:

```
Public Sum Save( scPUnit As SCAPI.PersistenceUnit )
  Dim propBag as SCAPI.PropertyBag
  propBag = scUnit.PropertyBag("Locator")
  Dim sFileName As String
  sFileName = propBag.Value("Locator")
  sFileName = sFileName + ".bak"
  scPUnit.Save(sFileName )
End Sub
```

Accessing Metamodel Information

You can obtain the metamodel of erwin DM by using the API. The metamodel can be accessed in the same manner as an erwin DM model. As in the case with model data, the *ISCPersistenceUnit* or *ISCModelSet* pointer in an *ISCSession::Open* call indicates the model set with which you are working.

There is a special case for the intrinsic metamodel. To obtain the intrinsic metamodel for a specific class of metadata, you can use the Property Bag component created with the *PropertyBag* method of the *ISCApplicationEnvironment* interface. A Property Bag instance populated with *EMX_Metadata_Class* or *EM2_Metadata_Class* properties from the *Application* category indicates the type of the intrinsic metamodel to access. The instance must be submitted as the first parameter in an *ISCSession::Open* call, instead of *ISCPersistenceUnit* or *ISCModelSet* pointers. If the first parameter in an *ISCSession::Open* call is NULL, then the intrinsic metamodel for the top model set in a persistence unit, the EMX class metadata, will be accessed.

To indicate that a session will access metamodel information, you set the *Level* parameter of the *Open* method to *SCD_SL_M1*.

ISCApplicationEnvironment Interface

The following table contains information on the *ISCApplicationEnvironment* interface:

Signature	Description	Valid Arguments
ISCPropertyBag PropertyBag(VARIANT Category[optional], VARIANT Name[optional], VARIANT AsString[optional])	Populates a property bag with one or more property values as indicated by Category and Name	Category: ■ VT_BSTR – Features returned from the given category. Must be Application. Name: ■ VT_BSTR – The property with the given name and category is returned. Must be EMX Metadata Class for EMX metadata and EM2 Metadata Class for EM2 metadata. AsString: ■ Empty – All values in the property bag are presented in their native type.

ISCSession Interface

The following table contains information on the *ISCSession* interface:

Signature	Description	Valid Arguments
VARIANT_BOOL Open(IUnknown * Unit, VARIANT Level [optional], VARIANT Flags [optional])	Binds self to the intrinsic metamodel, persistence unit, or model set identified by the *Unit* parameter	Unit: ■ NULL – The intrinsic metamodel for the top model set in a persistence unit. For the current version this is EMX class metadata. ■ ISCPropertyBag – The intrinsic metamodel defined by the metadata class in the first property of the bag. ■ ISCPersistenceUnit – The metamodel for the top model set in the persistence unit. ■ ISCModelSet – The metamodel for the model set. Level: ■ *SCD_SL_M1* – Metadata access. Flags: ■ Empty – Defaults to *SCD_SF_NONE*.

Example 27

The following example illustrates how to access an intrinsic metamodel using C++. The example uses an *Application* object from Example 1:

```
void AccessMetaModel( ISCApplicationPtr & scAppPtr )
{
  ISCSessionPtr scSessionPtr = scAppPtr->GetSessions()->Add();  // add a new
  // session
  // Open EMX intrinsic metamodel
  CComVariant varResult = scSessionPtr->Open(NULL, (long) SCD_SL_M1);  //  meta-model level
  if (varResult.vt == VT_BOOL && varResult.boolVal == FALSE)
    return;
  // ...
}
```

The following example illustrates how to access an intrinsic metamodel using Visual Basic .NET. The example uses an *Application* object from Example 1:

```
Public Sub AccessMetaModel( ByRef scApp As SCAPI.Application )
  Dim scBag As SCAPI.PropertyBag
  Dim scSession As SCAPI.Session

  ' Get a property bag with the EM2 metadata class
  scBag = scApp.ApplicationEnvironment.PropertyBag("Application ", "EM2 Metadata Class")

  ' Open EM2 intrinsic metamodel
  scSession = scApp.Sessions.Add
  scSession.Open( scBag, SCD_SL_M1 )
End Sub
```

Closing the API

When the client of the API has finished accessing the model, the sessions that were open must be closed, and the persistence unit collection must be cleared.

ISCSession Interface

The following table contains information on the *ISCSession* interface:

Signature	Description	Valid Arguments
VARIANT_BOOL Close()	Disconnects self from its associated persistence unit	None

ISCSessionCollection Interface

The following table contains information on the *ISCSessionCollection* interface:

Signature	Description	Valid Arguments
VARIANT_BOOL Remove(VARIANT SessionId)	Removes a Session object from the collection	SessionId: ■ VT_UNKNOWN – Pointer to the ISCSession interface. Removes the given session from the collection. ■ VT_I4 – Zero-based index in the session collection. Removes the session with the given index from the collection.

Example 28

The following example illustrates how to close a session using C++. It assumes that there is a *Session* object and the session is open. The examples use an *Application* object from Example 1:

```
void CloseSessions( ISCApplicationPtr & scAppPtr )
{
  ISCSessionCollectionPtr scSessionColPtr = scAppPtr->GetSessions();
  ISCSessionPtr scSessionPtr = scSessionColPtr->GetItem(COleVariant(0L))
  // close the sessions
  scSessionPtr->Close();   // close a single session
  scSessionColPtr->Clear();  // clear the collection of sessions
}
```

The following example illustrates how to close a session using Visual Basic .NET. It assumes that there is a *Session* object and the session is open. The examples use an *Application* object from Example 1:

```
Public Sub CloseSessions( scApp As SCAPI.Application )
  Dim scSessionCol As SCAPI.Sessions
  scSessionCol = scApp.Sessions
  Dim scSession As SCAPI.Session

  For Each scSession In scSessionCol
    scSession.Close
  Next
  While (scSessionCol.Count > 0)
    scSessionCol.Remove (0)
  End
End Sub
```

Clearing Persistence Units

This section describes how to clear persistence units.

The effect of leaving persistence units in the *Persistence Units* collection is dictated by a context in which an instance of the application is created. If a client is using the API in the standalone mode, all units are closed. If a client is using the API as an add-in component, then after the client program is over, units are still open and available in the application user interface with the exception of those that were explicitly closed and removed from the persistence unit collection before exiting the program.

ISCPersistenceUnitCollection Interface

The following table contains information on the *ISCPersistenceUnitCollection* interface:

Signature	Description	Valid Arguments
VARIANT_BOOL Clear()	Purges all units from the collection	None

Example 29

The following example illustrates how to clear persistence units using C++. It assumes that there is an *Application* object from Example 1:

```
// remove the persistence units
scAppPtr->GetPersistenceUnits()->Clear();
```

The following example illustrates how to clear persistence units using Visual Basic .NET. It assumes that there is an *Application* object from Example 1:

```
scApp.PersistenceUnits.Clear
```

Error Handling

The API uses a generic COM error object to handle errors. Depending on the programming environment, languages have their own protocols to retrieve errors from the generic error object. For example, C++ and Visual Basic .NET use exception handling to handle errors. To ensure a stable application, it is recommended that API clients use error handling to trap potential errors such as attempting to access an object that was deleted, or attempting to access an empty collection.

Example 30

The following example illustrates error handling using C++. It assumes that there is a *Model Object* object from Example 9:

```
long GetObjectProperties(ISCModelObjectPtr & scObjPtr)
{
   // Get the collection of Properties
   ISCModelPropertyCollectionPtr scPropColPtr;
   try
   {
      scPropColPtr = scObjPtr->GetProperties();
      if (!scPropColPtr.GetInterfacePtr())
      {
         AfxMessageBox("Unable to Get Properties Collection");
         return FALSE;
      }
      // ...
   }
   catch(_com_error &error)
   {
      AfxMessageBox(error.Description());
   }
}
```

The following example illustrates error handling using Visual Basic .NET. It assumes that there is a *Model Object* object from Example 9:

```
Public Sub GetObject(ByRef scSession As SCAPI.Session, ByRef objID As String)
   Dim scObjCol as SCAPI.ModelObjects
   Dim scObj as SCAPI.ModelObject

   Try
      scObjCol = scSession.ModelObjects
      scObj = scObjCol.Item(objID)  ' retrieves object with given object ID
   Catch ex As Exception
      ' Failed
      Console.WriteLine(" API Failed With Error message :" + ex.Message())
   End Try
End Sub
```

In addition to the generic error object, the API provides an extended error handling mechanism with the Application Environment Message log. The message log can handle a sequence of messages that is useful in a context of complex operations like transactions.

More information about the Application Environment Message log organization is located in the Property Bag for Application Environment (see page 163) section.

ISCApplicationEnvironment

The following table contains information on the *ISCApplicationEnvironment* interface:

Signature	Description	Valid Arguments
ISCPropertyBag PropertyBag(VARIANT Category[optional], VARIANT Name[optional], VARIANT AsString[optional])	Populates a property bag with one or more property values as indicated by Category and Name	Category: VT_BSTR – Must be Application.API. Name: ■ VT_BSTR – The property with the given name and category is returned. Must be *Is Empty* to determine if the message log has messages. To retrieve the message log content, it must be *Log*. AsString: ■ Empty – All values in the property bag are presented in their native type. ■ VT_BOOL – If set to TRUE, all values in the property bag are presented as strings.

Example 32

The following example illustrates how to use the API to check messages from the API extended message log using C++. It assumes that there is an *Application* object from Example 1:

```
CString GetExtendedErrorInfo(ISCApplicationPtr & scAppPtr)
{
  CString csExtendedErrors = "";
  long index = 0;

  // Do we have messages in the log?
  variant_t val = scAppPtr->GetApplicationEnvironment()-> GetPropertyBag("Application.Api.MessageLog","Is Empty")->
GetValue(COleVariant(index));

  if (val.vt == VT_BOOL && val.boolVal == false)
  {
    // Retrieve the log
    val = m_scAppPtr->GetApplicationEnvironment()-> GetPropertyBag("Application.Api.MessageLog","Log")->
GetValue(COleVariant(index));
    if (val.vt & VT_ARRAY)
    {
        // this is a SAFEARRAY

        VARIANT HUGEP *pArray;
        HRESULT hr;

        // Get a pointer to the elements of the array.
        hr = SafeArrayAccessData(val.parray, (void HUGEP**)&pArray);
        if (FAILED(hr))
          return csExtendedErrors;

        long numErrors = 0;
        VARIANT vValue = pArray[0];  // number of errors
        if (vValue.vt == VT_I4)
          numErrors = vValue.lVal;

      // ...
      SafeArrayUnaccessData(val.parray);
    }
  }
}
```

The following example illustrates how to use the API to check messages from the API extended message log using Visual Basic .NET. It assumes that there is an *Application* object from Example 1:

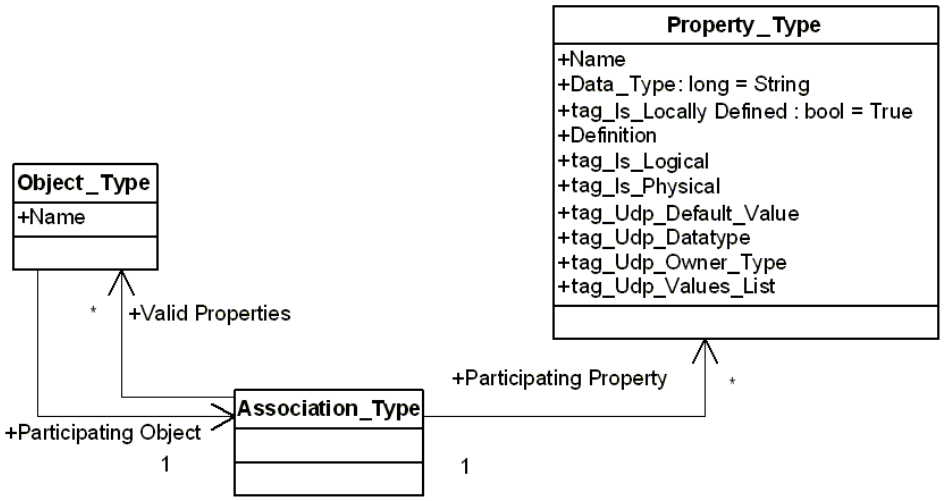
```
Public Sub GetExtendedErrorInfo( ByRef scApp As SCAPI.Application )
  Dim nSize As Integer
  Dim nWarnings As Integer
  Dim nErrors As Integer
  Dim nIdx As Integer
  Dim nMsgNumber As Integer
  Dim aErrors() As Object
  ' Do we have messages in the log?
  If scApp.ApplicationEnvironment.PropertyBag("Application.Api.MessageLog", _ "Is Empty").Value(0) = False Then
    ' Retrieve a log
    aErrors = _
    scApp.ApplicationEnvironment.PropertyBag("Application.Api.MessageLog", _ "Log").Value(0)
     nSize = Int(aErrors(0))
     nIdx = 1
     nMsgNumber = 0
     Do While nMsgNumber < nSize
       Console.WriteLine("Error " & aErrors(nIdx) & "  " + aErrors(nIdx + 2))
       Select Case aErrors(nIdx + 1)
         Case SCAPI.SC_MessageLogSeverityLevels.SCD_ESL_WARNING
       nWarnings = nWarnings + 1
         Case SCAPI.SC_MessageLogSeverityLevels.SCD_ESL_ERROR
       nErrors = nErrors + 1
       End Select
       nIdx = nIdx + 8
       nMsgNumber = nMsgNumber + 1
     Loop

     Console.WriteLine("Total number of errors in the transaction " & Str(nSize) & " with: " _& Str(nWarnings) & " warnings, "
& Str(nErrors) & " errors.")

  End If
End Sub
```

Advanced Tasks

The material in this section provides examples of some advanced tasks and how they can be executed.

Creating User-Defined Properties

A User-Defined Property (UDP) is an example of a client expanding the erwin DM metadata and involves creating and modifying objects on the metadata level. The structure of the UDP definition is similar to the definition of all native properties. The following diagram shows the metamodel objects involved when you define a UDP:



In this diagram an instance of the *Property_Type* object defines a UDP class, the *Object_Type* object defines an object class with which the UDP is associated, and the *Association_Type* object defines the association between object and property classes.

You are only required to create an instance of the *Property_Type* object to define a UDP. erwin DM populates the rest of the necessary data. The following table describes the properties and tags of the *Property_Type* object:

Property or Tag Name	Description	Valid Arguments
Name	Property, UDP name	erwin DM upholds the following convention in naming UDPs to ensure their uniqueness. The convention is a three part name separated with dot (.) symbols: <ObjectClassName>.<Logical/Physical>.<Name> An example of this naming convention is: *Model.Logical.My UDP* The erwin DM editors display only the last component.
Data_Type	Property, SCVT_BSTR	The property is read-only and set by erwin DM. All UDP values have a string datatype.
tag_Is_Locally_Defined	Property, TRUE	The property is read-only and set to TRUE for all user-defined metadata.
Definition	Property, Optional	Optional – Text that displays the UDP description.

Property or Tag Name	Description	Valid Arguments
tag_Is_Logical	Tag, TRUE or FALSE	Optional – The tag has a TRUE value for UDPs used in logical modeling.
tag_Is_Physical	Tag, TRUE or FALSE	Optional – The tag has a TRUE value for UDPs used in physical modeling.
tag_Udp_Default_Value	Tag	Optional – A string with the UDP default value.
tag_Udp_Data_Type	Tag	Defines the interpretation for the UDP value in the erwin DM editors. The valid values are: ■ 1 (Integer) ■ 2 (Text) ■ 3 (Date) ■ 4 (Command) ■ 5 (Real) ■ 6 (List) The property value can be: ■ VT_I4 – Uses the numeric values listed above. ■ VT_BSTR – Uses the string values listed above. Assumes the Text type if it is not specified.
tag_Udp_Owner_Type	Tag	Required. Defines an object class to host instances of the UDPs. ■ VT_BSTR – Name of an object class. Indicates the host class by the given class name. ■ VT_BSTR – Class ID of an object class. Indicates the host class by the given identifier.
tag_Udp_Values_List	Tag	String with comma-separated values. Only values from the list are valid values for a UDP. Valid only if the *tag_Udp_Data_Type* tag is set to List.

//the following example was changed in r9.6, because the

Example 33

The following example illustrates how to use the API to define a UDP using Visual Basic Script:

```
Dim oAPI

Set oAPI = CreateObject("erwin9.SCAPI.9.0")

Dim oPU

Set oPU = oAPI.PersistenceUnits.Create(Nothing)

Dim oSession

Set oSession = oAPI.Sessions.Add

SCD_SL_M1 = 1

call oSession.Open(oPU, SCD_SL_M1)

Dim TransId

TransId = oSession.BeginNamedTransaction("Create UDP")

Dim oUDP

Set oUDP = oSession.ModelObjects.Add("Property_Type")

' Populate properties

        ' Add udp with Text type

Set oUDP = oSession.ModelObjects.Add("Property_Type")

oUDP.Properties("Name").Value = "Entity.Logical.My UDP1"

oUDP.Properties("tag_Udp_Owner_Type").Value = "Entity"

oUDP.Properties("tag_Is_Logical").Value = True

oUDP.Properties("tag_Udp_Data_Type").Value = 2

oUDP.Properties("tag_Udp_Default_Value").Value = "Text"

 oUDP.Properties("tag_Order").Value = "1"

        'Add udp with list type

Set oUDP = oSession.ModelObjects.Add("Property_Type")
```

```
oUDP.Properties("Name").Value = "Entity.Logical.My UDP5"

oUDP.Properties("tag_Udp_Owner_Type").Value = "Entity"

oUDP.Properties("tag_Is_Logical").Value = True

oUDP.Properties("tag_Udp_Data_Type").Value = 6

oUDP.Properties("tag_Udp_Values_List").Value = "1,2,3"
oUDP.Properties("tag_Udp_Default_Value").Value = "1"

oUDP.Properties("tag_Order").Value = "1"

' Commit changes

oSession.CommitTransaction (TransId)

' Release the session

oSession.Close

Set oSession = Nothing

oAPI.Sessions.Clear

' Save to the file

   Call oPU.Save("C:\Temp1\UDP.erwin", "OVF=Yes")
```

History Tracking

Historical information can be saved for your model, entities, attributes, tables, and columns. erwin DM uses *History* objects to store the information in the model.

The API provides functionality that allows you to customize the process of history tracking without having to work with the *History* objects directly. The *BeginNamedTransaction* function of the *ISCSession* interface accepts a Property Bag instance populated with the history tracking properties. The properties are in effect at the initiation of an outer transaction and are confined to the scope of the transaction.

ISCSession Interface

The following table contains information on the *ISCSession* interface:

Signature	Description	Valid Arguments
VARIANT BeginNamedTransaction(BSTR Name, VARIANT PropertyBag [optional])	Opens a transaction on the session with the given name. Returns an identifier of the transaction.	Name – Provides a name for a new transaction. PropertyBag – Collection of parameters for history tracking in the transaction.

The following table describes the properties used in creating a new model:

Property Name	Type	Description
History_Tracking	Boolean	TRUE – Indicates that all historical information generated during the transaction will be marked as the API event. The TRUE value is assumed if the property is not provided. FALSE – Uses the standard erwin DM mechanism of history tracking.
History_Description	BSTR	When the *History_Tracking* property is TRUE, it provides the content for the Description field of the history event.

Note: A complete set of available properties is located in the appendix API Interfaces Reference (see page 105).

Example 34

The following example illustrates how to mark history records for entities and attributes as API events, and how to mark history records with the API History Tracking description using Visual Basic .NET:

```
Public Sub Main()
  Sub Main()
    Dim oApi As New SCAPI.Application
    Dim oBag As New SCAPI.PropertyBag
    Dim oPU As SCAPI.PersistenceUnit

    ' Construct a new logical-physical model. Accept the rest as defaults
    oBag.Add("Model_Type", "Combined")
    oPU = oApi.PersistenceUnits.Create(oBag)

    ' Clear the bag for the future reuse
    oBag.ClearAll()

    ' Start a session
    Dim oSession As SCAPI.Session

    oSession = oApi.Sessions.Add
    oSession.Open(oPU)

    ' Prepare a property bag with the transaction properties
    oBag.Add("History_Description", "API History Tracking")

    ' Start a transaction
    Dim nTransId As Object

    nTransId = oSession.BeginNamedTransaction("Create Entity and Attribute", oBag)

    ' Create an entity and an attribute
    Dim oEntity As SCAPI.ModelObject
    Dim oAttribute As SCAPI.ModelObject

    oEntity = oSession.ModelObjects.Add("Entity")
    oAttribute = oSession.ModelObjects.Collect(oEntity).Add("Attribute")
    oAttribute.Properties("Name").Value = "Attr A"

    ' Commit
    oSession.CommitTransaction(nTransId)

  End Sub
```

You can select the history options for the model objects for which you want to preserve history, as well as to control the type of events to track. This is done within the History Options tab in the Model Properties dialog.

If the check box for API events is cleared (unchecked), then no historic events from the API category are recorded. It is possible to control the status of that check box, as well as the check boxes for model object types from the API, by controlling the value of properties in the model where the status of these check boxes is stored.

Appendix A: API Interfaces Reference

This appendix lists the interfaces contained in the API, together with the methods and arguments associated with these interfaces. There is also a section that contains information regarding enumerations and describes various Property Bag components.

This section contains the following topics:

ISCApplication

The *ISCApplication* interface is the entry point for the API client. Only one instance of the component can be externally instantiated to activate the API. The client navigates the interface hierarchy by using interface properties and methods to gain access to the rest of the API functionality.

The following table contains the methods for the *ISCApplication* interface:

Method	Description
BSTR ApiVersion()	The API version.
ISCApplicationEnvironment * ApplicationEnvironment()	Reports attributes of runtime environment and available features, such as add-in mode, user interface visibility, and so on.
ISCModelDirectoryCollection * ModelDirectories()	Collects model directories accessible from the current machine.
BSTR Name()	Modeling tool application name.
ISCPersistenceUnitCollection * PersistenceUnits()	Returns a collection of all persistence units loaded in the application.
ISCSessionCollection * Sessions()	Returns a collection of sessions created within the application.
BSTR Version()	Modeling tool application version.
BSTR ResolveMartModelPath(BSTR modelLongId)	Returns the path of the given model. Returns empty if no model exists with the given details.

API Interfaces

This section describes each API interface, and the methods associated with them. Where applicable, signatures and valid arguments are also described.

Note: Some parameters contain an [optional] designation. This means that this particular part of the parameter is optional and not required.

ISCApplicationEnvironment

The *ISCApplicationEnvironment* interface contains the information about the runtime environment.

The following table contains the methods for the *ISCApplicationEnvironment* interface:

Method	Description
ISCPropertyBag * PropertyBag(VARIANT Category [optional], VARIANT Name [optional], VARIANT AsString [optional])	Populates a property bag with one or more property values as indicated by Category and Name.

Note: More information about *ISCApplicationEnvironment* is located in the Property Bag for Application Environment (see page 163) section.

ISCApplicationEnvironment::PropertyBag Arguments

Here is the signature for the *PropertyBag* function:

```
ISCPropertyBag *PropertyBag(VARIANT Category, VARIANT Name, VARIANT AsString)
```

The following table contains the valid arguments for the *PropertyBag* function:

Parameter	Valid Type/Value	Description
Category [optional]	Empty	Complete set of features from all categories are returned.
Category [optional]	VT_BSTR – Name of category	Features from the given category are returned.
Name [optional]	Empty	All properties from the selected category are returned.
Name [optional]	VT_BSTR – Property name	The property with the given name and category is returned.

Parameter	Valid Type/Value	Description
AsString [optional]	Empty	All values in the property bag are presented in native type.
AsString [optional]	VT_BOOL – TRUE or FALSE	If set to TRUE, all values in the property bag are presented as strings.

Note: More information about category and property names relating to VT_BSTR is located in the Property Bag for Application Environment (see page 163) section.

ISCModelDirectory

The Model Directory encapsulates information on a single model directory entry. Examples of the Model Directory are a file system directory or a mart library.

The following table contains the methods for the *ISCModelDirectory* interface:

Method	Description
VARIANT_BOOL DirectoryExists(BSTR Locator)	Returns TRUE if a specified directory exists.
VARIANT_BOOL DirectoryUnitExists(BSTR Locator)	Returns TRUE if a specified directory unit exists.
SC_ModelDirectoryFlags Flags()	Model Directory flags. A 32-bit property flag word.
VARIANT_BOOL IsOfType(ISCModelDirectory * Directory)	Returns TRUE if Directory has the same type of connection as self. For example, directory entries from the same mart and with the same login attributes, such as user, password, and so on, are considered of the same type.
ISCModelDirectory * LocateDirectory (BSTR Locator, VARIANT Filter [optional])	Starts enumeration over the directory sub-entries.
ISCModelDirectory * LocateDirectoryNext()	Locates the next sub-entry in the directory enumeration. Returns a NULL pointer if no more model directory entries can be found.
ISCModelDirectoryUnit * LocateDirectoryUnit (BSTR Locator, VARIANT Filter [optional])	Starts enumeration over the directory units.
ISCModelDirectoryUnit * LocateDirectoryUnitNext()	Locates the next unit in the directory enumeration.
BSTR Locator()	Location of the directory including the absolute path and parameters. Does not include password information.

Method	Description
BSTR Name()	Model Directory name. For example, the file system directory name without path information.
ISCPropertyBag* PropertyBag(VARIANT List [optional], VARIANT AsString [optional])	Returns a pointer on a property bag with the directory properties. **Note:** A directory property is present in the resulting bag only if it has a value. If the property does not have any value set, the property bag will not have the property listed.
void PropertyBag(VARIANT List [optional], VARIANT AsString [optional], ISCPropertyBag* Property Bag)	Accepts a pointer on a property bag with the directory properties.
SC_ModelDirectoryType Type()	Type of a directory.

ISCModelDirectory::DirectoryExists Arguments

Here is the signature for the *DirectoryExists* function:

```
VARIANT_BOOL DirectoryExists( BSTR Locator)
```

The following table contains the valid arguments for the *DirectoryExists* function:

Parameter	Valid Type/Value	Description
Locator	BSTR – String with a directory name	Identifies a directory path. For an absolute path, the mart database information and access parameters are ignored.

ISCModelDirectory::DirectoryUnitExists Arguments

Here is the signature for the *DirectoryUnitExists* function:

```
VARIANT_BOOL DirectoryUnitExists( BSTR Locator)
```

The following table contains the valid arguments for the *DirectoryUnitExists* function:

Parameter	Valid Type/Value	Description
Locator	BSTR – String with a directory name	Identifies a directory unit path. For an absolute path, the mart database information and access parameters are ignored.

ISCModelDirectory::IsOfType Arguments

Here is the signature for the *IsOfType* function:

```
VARIANT_BOOL IsOfType(ISCModelDirectory * Directory)
```

The following table contains the valid arguments for the *IsOfType* function:

Parameter	Valid Type/Value	Description
Directory	ISCModelDirectory *. Model Directory component pointer	Identifies a directory

ISCModelDirectory::LocateDirectory Arguments

Here is the signature for the *LocateDirectory* function:

```
ISCModelDirectory * LocateDirectory (BSTR Locator, VARIANT Filter)
```

The following table contains the valid arguments for the *LocateDirectory* function:

Parameter	Valid Type/Value	Description
Locator	BSTR – String with a directory location	Identifies a directory path that can contain wildcard characters in the last path component in order to search for sub-entries. If the path provides an exact location, it can also be used to return to a single model directory entry. For an absolute path, the mart database information and access parameters are ignored.
Filter [optional]	VT_BSTR – Options	Specifies a set of options to narrow a search.

ISCModelDirectory::LocateDirectoryUnit Arguments

Here is the signature for the *LocateDirectoryUnit* function:

```
ISCModelDirectoryUnit * LocateDirectoryUnit (BSTR Locator, VARIANT Filter)
```

The following table contains the valid arguments for the *LocateDirectoryUnit* function:

Parameter	Valid Type/Value	Description
Locator	BSTR – String with a directory or unit location	Identifies a directory path that can contain wildcard characters in the last path component in order to search for units. If the path provides an exact location, it can also be used to return to a single model directory unit. For an absolute path, the mart database information and access parameters are ignored.
Filter [optional]	VT_BSTR – Options	Specifies a set of options to narrow a search.

ISCModelDirectory::PropertyBag Arguments (Get Function)

Here is the signature for the *PropertyBag (Get)* function:

```
ISCPropertyBag * PropertyBag(VARIANT List, VARIANT AsString)
```

The following table contains the valid arguments for the *PropertyBag (Get)* function:

Parameter	Valid Type/Value	Description
List [optional]	VT_BSTR – Semicolon separated list of property names	Provides a list of the model directory properties. If the list is provided, only listed properties are placed in the returned property bag.
List [optional]	Empty	Requests a complete set of properties.

Parameter	Valid Type/Value	Description
AsString [optional]	VT_BOOL – TRUE or FALSE	If set to TRUE, requests that all values in the bag to be presented as strings. The default is FALSE with all values in their native format.
AsString [optional]	Empty	All values in the property bag are presented in native type.

Note: Information about valid property names for VT_BSTR is located in the Property Bag for Model Directory and Model Directory Unit (see page 170) section.

ISCModelDirectory::PropertyBag Arguments (Set Function)

Here is the signature for the *PropertyBag (Set)* function:

void PropertyBag(VARIANT List, VARIANT AsString, ISCPropertyBag * propBag)

The following table contains the valid arguments for the *PropertyBag (Set)* function:

Parameter	Valid Type/Value	Description
List [optional]		Not used
AsString [optional]		Not used
propBag	ISCPropertyBag *	A pointer on a property bag with the directory properties to process.

Note: Information about valid property names and format for ISCPropertyBag * is located in the Property Bag for Model Directory and Model Directory Unit (see page 170) section.

ISCModelDirectoryCollection

The Model Directory Collection lists all top-level Model Directories available including the one made available with the application user interface. A client can register new Model Directories with this collection.

Method	Description
IUnknown _NewEnum()	Constructs an instance of the collection enumerator object.
ISCModelDirectory * Add(BSTR Locator, VARIANT Disposition [optional])	Adds a new top-level directory on the list of available directories.

Method	Description
VARIANT_BOOL Clear()	Removes all the top-level directories from a collection and disconnects the directories from associated marts.
long Count()	The number of ModelDirectory components in the collection.
ISCModelObject * Item(long nIndex)	Returns an *IUnknown* interface pointer identified by its ordered position.
VARIANT_BOOL Remove(VARIANT Selector, VARIANT_BOOL Disconnect [optional])	Removes a top-level directory from the list of available directories.

ISCModelDirectoryCollection::Add Arguments

Here is the signature for the *Add* function:

```
ISCModelDirectory * Add(BSTR Locator, VARIANT Disposition)
```

The following table contains the valid arguments for the *Add* function:

Parameter	Valid Type/Value	Description
Locator	BSTR – A model directory location	Identifies a model directory location along with the attributes required for successful access to storage.
Disposition [optional]	VT_BSTR – List of keywords parameters	Arranges access attributes, such as resume session.

ISCModelDirectoryCollection::Item Arguments

Here is the signature for the *Item* function:

```
ISCModelDirectory * Item(long nIndex)
```

The following table contains the valid arguments for the *Item* function:

Parameter	Valid Type/Value	Description
nIndex	A long number	Identifies an ordered position of a Model Directory item. The index is zero-based.
Class [optional]	Empty	Returns the object specified by *nIndex*.

ISCModelDirectoryCollection::Remove Arguments

Here is the signature for the *Remove* function:

VARIANT_BOOL Remove(VARIANT Selector, VARIANT_BOOL Disconnect [optional])

The following table contains the valid arguments for the *Remove* function:

Parameter	Valid Type/Value	Description
Selector	VT_UNKNOWN – ISCModelDirectory pointer	An object pointer for the Model Directory to remove.
Selector	VT_I4 – Numeric index	Identifying a model directory for removal with a zero-based index.

ISCModelDirectoryUnit

The Model Directory Unit encapsulates information on a single directory unit. A file system file and a model in a mart are examples of the Model Directory Unit.

The following table contains the methods for the *ISCModelDirectoryUnit* interface:

Method	Description
SC_ModelDirectoryFlags Flags()	Model directory unit flags. A 32-bit property flag word.
VARIANT_BOOL IsOfType(ISCModelDirectory * Directory)	Returns TRUE if directory has the same type of connection as self. For example, directory entries from the same mart and with the same login attributes, such as user, password, and so on, are considered of the same type.
BSTR Locator()	Location of the directory unit including the absolute path and parameters. Does not include password information.
BSTR Name()	Model directory unit name. For example, the file system file name without path information.

Method	Description
ISCPropertyBag* PropertyBag(VARIANT List [optional], VARIANT AsString [optional])	Returns a pointer on a property bag with the directory unit properties. **Note:** A directory unit property is present in the resulting bag only if it has a value. If the property does not have any value set, the property bag will not have the property listed.
void PropertyBag(VARIANT List [optional], VARIANT AsString [optional], ISCPropertyBag* Property Bag)	Accepts a pointer on a property bag with the directory unit properties.
SC_ModelDirectoryType Type()	Type of a directory.

Note: More information about Model Directory flags is located in the Enumerations (see page 159) section.

ISCModelDirectoryUnit::IsOfType Arguments

Here is the signature for the *IsOfType* function:

```
VARIANT_BOOL IsOfType(ISCModelDirectory * Directory)
```

The following table contains the valid arguments for the *IsOfType* function:

Parameter	Valid Type/Value	Description
Directory	ISCModelDirectory * – Model Directory component pointer	Identifies a directory

ISCModelDirectoryUnit::PropertyBag Arguments (Get Function)

Here is the signature for the *PropertyBag (Get)* function:

```
ISCPropertyBag * PropertyBag(VARIANT List, VARIANT AsString)
```

The following table contains the valid arguments for the *PropertyBag (Get)* function:

Parameter	Valid Type/Value	Description
List [optional]	VT_BSTR – Semicolon separated list of property names	Provides a list of the model directory unit properties. If the list is provided, only listed properties are placed in the returned property bag.
List [optional]	Empty	Requests a complete set of properties.

Parameter	Valid Type/Value	Description
AsString [optional]	VT_BOOL – TRUE or FALSE	If set to TRUE, requests that all values in the bag to be presented as strings. The default is FALSE with all values in their native format.
AsString [optional]	Empty	All values in the property bag are presented in native type.

Note: Information about valid property names for VT_BSTR is located in the Property Bag for Model Directory and Model Directory Unit (see page 170) section.

ISCModelDirectoryUnit::PropertyBag Arguments (Set Function)

Here is the signature for the *PropertyBag (Set)* function:

void PropertyBag(VARIANT List, VARIANT AsString, ISCPropertyBag * propBag)

The following table contains the valid arguments for the *PropertyBag (Set)* function:

Parameter	Valid Type/Value	Description
List [optional]		Not used
AsString [optional]		Not used
propBag	ISCPropertyBag *	A pointer on a property bag with the unit properties to process.

Note: Information about valid property names and format for ISCPropertyBag * is located in the Property Bag for Model Directory and Model Directory Unit (see page 170) section.

ISCModelObject

The *ISCModelObject* interface represents an object in a model.

The following table contains the methods for the *ISCModelObject* interface:

Method	Description
SC_ModelObjectFlags Flags()	Returns the flags of the object.
SC_CLSID ClassId()	Returns the class identifier of the current object.
BSTR ClassName()	Returns the class name of the current object.

Method	Description
ISCModelPropertyCollection * CollectProperties(VARIANT ClassIds [optional], VARIANT MustBeOn [optional], VARIANT MustBeOff [optional])	Returns a property collection of the type that you want. This method always returns a valid collection even if the collection is empty.
ISCModelObject * Context()	Passes back the context (parent) of the object in the model's object tree. Passes back NULL if the current object is the tree root.
VARIANT_BOOL IsInstanceOf(VARIANT ClassId)	Returns TRUE if self is an instance of the passed class. This method respects inheritance. If ClassId contains an ancestor class, the method returns TRUE.
VARIANT_BOOL IsValid()	Returns TRUE if self is valid. This method is used to detect if the referenced object is deleted.
BSTR Name()	Returns the name or a string identifier of the current object.
SC_OBJID ObjectId()	Uniquely identifies the current object.
ISCModelPropertyCollection * Properties()	Returns a property collection of all available properties.

Note: More information about SC_ModelObjectFlags is located in the Enumerations (see page 159) section.

ISCModelObject::CollectProperties Arguments

Here is the signature for the *CollectProperties* function:

```
ISCModelPropertyCollection * CollectProperties(VARIANT ClassIds, VARIANT MustBeOn, VARIANT MustBeOff)
```

The following table contains the valid arguments for the *CollectProperties* function:

Parameter	Valid Type/Value	Description
ClassIds [optional]	Empty	All properties of the object are returned.
ClassIds [optional]	VT_ARRAY|VT_BSTR – SAFEARRAY of property IDs	Provides a list of property class identifiers.
ClassIds [optional]	VT_ARRAY|VT_BSTR – SAFEARRAY of property names	Provides a list of property class names.
ClassIds [optional]	VT_BSTR – ID of a property	Identifies a property class.
ClassIds [optional]	VT_BSTR – Name of a property	Identifies a property class.
ClassIds [optional]	VT_BSTR – List of IDs delimited by semicolons	Provides a list of property class identifiers.

Parameter	Valid Type/Value	Description
ClassIds [optional]	VT_BSTR – List of property names delimited by semicolons	Provides a list of property class names.
MustBeOn [optional]	Empty	Defaults to *SCD_MPF_DONT_CARE* which indicates no filtering.
MustBeOn [optional]	VT_I4 – SC_ModelObjectFlags flags that must be on	Identifies the properties with the specified flags set.
MustBeOff [optional]	Empty	Defaults to *SCD_MPF_DONT_CARE* which indicates no filtering
MustBeOff [optional]	VT_I4 – SC_ModelObjectFlags flags that must be off	Identifies the properties that do not have the specified flags.

Note: For information about valid property class identifiers and valid property class names, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder. More information about *SC_ModelObjectFlags* is located in the Enumerations (see page 159) section.

ISCModelObject::IsInstanceOf Arguments

Here is the signature for the *IsInstanceOf* function:

```
VARIANT_BOOL IsInstanceOf(VARIANT ClassId)
```

The following table contains the valid arguments for the *IsInstanceOf* function:

Parameter	Valid Type/Value	Description
ClassId	VT_BSTR – ID of an object class	Identifies a target object class by the given identifier.
ClassId	VT_BSTR – Name of an object class	Identifies an object class by the given name.

Note: For information about valid object class names and identifiers, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin Data Modeler installation folder.

ISCModelObjectCollection

The *ISCModelObjectCollection* interface is a collection of objects in the model that is connected to the active session. Membership in this collection can be limited by establishing filter criteria.

The following table contains the methods for the *ISCModelObjectCollection* interface:

Method	Description
IUnknown _NewEnum()	Constructs an instance of the collection enumerator object.
ISCModelObject * Add(VARIANT Class, VARIANT ObjectId)	Adds an object of type *Class* to the model.
SC_CLSID * ClassIds()	Returns a *SAFEARRAY* of class identifiers (such as object type IDs). Represents a value of the Model Object collection attribute that limited the membership in the collection at the time when this collection was created and can be used for reference purposes. *ClassIds* contains a list of acceptable class identifiers (such as object types). If this list is non-empty, the collection includes only those objects whose class identifier appears in the list. If the list is empty or returns a NULL pointer, then all objects are included.
BSTR * ClassNames()	Similar to *ClassIds* except that it returns a *SAFEARRAY* of class names (such as object type names).
ISCModelObjectCollection * Collect(VARIANT Root, VARIANT ClassId [optional], VARIANT Depth [optional], VARIANT MustBeOn [optional], VARIANT MustBeOff [optional])	Creates a collection of Model Objects, which represents a subcollection of itself. All filtering criteria specified in the Collect call is applied toward membership in the collection. The method creates a valid collection even though the collection may be empty. All enumerations are depth-first.
long Count()	Number of objects in the collection. The number does not include the root object.
long Depth()	Depth limit on iteration in the collection. -1 represents unlimited depth.
ISCModelObject * Item(VARIANT nIndex, VARIANT Class [optional])	Returns an *IUnknown* pointer for a Model Object component identified by the *Index* parameter.
SC_ModelObjectFlags MustBeOff()	Filter on model object flags in the collection.
SC_ModelObjectFlags MustBeOn()	Filter on model object flags in the collection.

Method	Description
VARIANT_BOOL Remove(VARIANT Object)	Removes the specified model object from a model.
ISCModelObject * Root()	Returns a pointer to the root object in a collection.

Note: For information about valid object class names and identifiers, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin Data Modeler installation folder.

ISCModelObjectCollection::Add Arguments

Here is the signature for the *Add* function:

```
ISCModelObject * Add(VARIANT Class,  VARIANT ObjectId)
```

The following table contains the valid arguments for the *Add* function:

Parameter	Valid Type/Value	Description
Class	VT_BSTR – Name of a class	Identifies an object class by the given class name.
Class	VT_BSTR – Class ID of an object type	Identifies an object class by the given identifier.
ObjectId [optional]	Empty	The API assigns an object identifier for a new object.
ObjectId [optional]	VT_BSTR – Object ID for a new object	The API assigns the given object identifier to the new object.

Note: For information about valid object class names and identifiers, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin Data Modeler installation folder.

ISCModelObjectCollection::Collect Arguments

Here is the signature for the *Collect* function:

ISCModelObjectCollection * Collect(VARIANT Root, VARIANT ClassId, VARIANT Depth, VARIANT MustBeOn, VARIANT MustBeOff)

The following table contains the valid arguments for the *Collect* function:

Parameter	Valid Type/Value	Description
Root	VT_UNKNOWN – ISCModelObject pointer of the root object	Provides a context (parent) object for the collection.
Root	VT_BSTR – ID of the root object	Provides a context (parent) object for the collection.
ClassId [optional]	VT_ARRAY|VT_BSTR – SAFEARRAY of class IDs	Contains a list of acceptable class identifiers.
ClassId [optional]	VT_ARRAY|VT_BSTR – SAFEARRAY of class names	Contains a list of acceptable class names.
ClassId [optional]	VT_BSTR – Class ID	Provides a class identifier for a monotype collection.
ClassId [optional]	VT_BSTR – Semicolon delimited list of class IDs	Contains a list of acceptable class identifiers.
ClassId [optional]	VT_BSTR – Class name	Provides a type name for a monotype collection.
ClassId [optional]	VT_BSTR – Semicolon delimited list of class names	Contains a list of acceptable class names.
ClassId [optional]	Empty	Returns all descendents regardless of class type.
Depth [optional]	VT_I4 – Maximum depth for descendents. Depth of 1 returns the immediate children of the root. A depth of -1 (which is the default value) represents unlimited depth.	Returns the descendents of the root at a depth no more than the given depth.
Depth [optional]	Empty	Returns all descendents of the root (unlimited depth).
MustBeOn [optional]	VT_I4 – SC_ModelObjectFlags that must be set	Provides a set of required flags.
MustBeOn [optional]	Empty	Defaults to *SCD_MOF_DONT_CARE*.

Parameter	Valid Type/Value	Description
MustBeOff [optional]	VT_I4 – SC_ModelObjectFlags that must not be set	Provides a set of flags that must not be set.
MustBeOff [optional]	Empty	Defaults to *SCD_MOF_DONT_CARE*.

Note: For information about valid object class names and identifiers, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin Data Modeler installation folder. More information about *SC_ModelObjectFlags* is located in the Enumerations (see page 159) section.

ISCModelObjectCollection::Item Arguments

Here is the signature for the *Item* function:

```
ISCModelObject * Item(VARIANT nIndex, VARIANT Class)
```

The following table contains the valid arguments for the *Item* function:

Parameter	Valid Type/Value	Description
nIndex	VT_UNKNOWN – Pointer to ISCModelObject interface	Identifies an object with the Model Object pointer.
nIndex	VT_BSTR – ID of an object	Identifies an object with the given object identifier.
nIndex	VT_BSTR – Name of an object	If the name of an object is used, the *Class* parameter must also be used. Identifies an object with the given name and given object class.
Class [optional]	Empty	Only if *nIndex* is not an object name.
Class [optional]	VT_BSTR – Name of a class	Must be used if the *nIndex* parameter is the name of an object. Identifies an object class name.
Class [optional]	VT_BSTR – Class ID of object type	Must be used if the *nIndex* parameter is the name of an object. Identifies an object class identifier.

Note: For information about valid object class names and identifiers, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin Data Modeler installation folder.

ISCModelObjectCollection::Remove Arguments

Here is the signature for the *Remove* function:

VARIANT_BOOL Remove(VARIANT Object)

The following table contains the valid arguments for the *Remove* function:

Parameter	Valid Type/Value	Description
Object	VT_UNKNOWN. ISCModelObject – pointer to an object	Identifies the removed object by the Model Object pointer.
Object	VT_BSTR – ID of the object	Identifies the removed object by the object's identifier.

ISCModelProperty

The *ISCModelProperty* interface represents a property of a given object.

The following table contains the methods for the *ISCModelProperty* interface:

Method	Description
BSTR ClassName()	Returns the class name of the property.
BSTR FormatAsString()	Formats the property value as a string.
ISCPropertyValueCollection * PropertyValues()	Returns the collection of values for the model property
long Count()	Contains the number of values in the property.
SC_CLSID ClassId()	Returns the class identifier of the property.
SC_ModelPropertyFlags Flags()	Returns the flags of the property.
SC_ValueTypes DataType(VARIANT ValueId [optional])	Passes back the identifier of the native value type for the indicated property value.
VARIANT_BOOL GetValueFacetIds(Long* FacetsTrueBasket, Long* FacetsFalseBasket)	Retrieves available property facet IDs. *FacetsTrueBasket* is a SAFEARRAY of facet ID numbers. The listed facets have TRUE as a value. *FacetsFalseBasket* is a SAFEARRAY of facet ID numbers. The listed facets have FALSE as a value. The method returns FALSE if the property does not have a value.

Method	Description
VARIANT_BOOL GetValueFacetNames(BSTR* FacetsTrueBasket,BSTR* FacetsFalseBasket)	Retrieves available property facet names. *FacetsTrueBasket* is a SAFEARRAY of facet name strings. The listed facets have TRUE as a value. *FacetsFalseBasket* is a SAFEARRAY of facet name strings. The listed facets have FALSE as a value. The method returns FALSE if the property does not have a value.
VARIANT_BOOL IsValid()	Returns TRUE if self is valid.
VARIANT_BOOL RemoveAllValues()	Removes all values from the property.
VARIANT_BOOL RemoveValue(VARIANT ValueId [optional])	Removes the specified value from the property. If no values remain after the removal, the property has a NULL value. Returns TRUE if the value was removed.
VARIANT Value(VARIANT ValueId [optional], VARIANT ValueType [optional])	Retrieves the indicated property value in the requested format.
Void SetValueFacets(VARIANT* FacetsTrueBasket, VARIANT* FacetsFalseBasket)	Assigns new values to the property facets. *FacetsTrueBasket* is a list of facets to be set to TRUE. It is either a SAFEARRAY of facet ID numbers, a SAFEARRAY of facet name strings, or a string with semicolon-separated facet names. *FacetsFalseBasket* is a list of facets to be set to FALSE. It is either a SAFEARRAY of facet ID numbers, a SAFEARRAY of facet name strings, or a string with semicolon-separated facet names. The method returns FALSE if the property does not have a value
void Value(VARIANT ValueId [optional], VARIANT ValueType [optional], VARIANT Val)	Sets the indicated property value with the given value.

Note: For information about valid property class identifiers and valid property class names, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder. More information about *SC_ModelPropertyFlags* is located in the Enumerations (see page 159) section. More information about property datatypes is located in the SC_ValueTypes (see page 162) section.

ISCModelProperty::DataType Arguments

Here is the signature for the *DataType* function:

```
SC_ValueTypes DataType(VARIANT ValueId)
```

The following table contains the valid arguments for the *DataType* function:

Parameter	Valid Type/Value	Description
ValueId	Empty	Valid if a property is scalar or if all elements of a multi-valued property have the same datatype.
ValueId	VT_I4 – Index	Ignored if the property is scalar. Identifies an element in a multi-valued property with a zero-based index.
ValueId	VT_BSTR – Name of a non-scalar element	Ignored if the property is scalar. If the property is multi-valued, indicates an element by name.

ISCModelProperty::RemoveValue Arguments

Here is the signature for the *RemoveValue* function:

```
VARIANT_BOOL RemoveValue(VARIANT ValueId)
```

The following table contains the valid arguments for the *RemoveValue* function:

Parameter	Valid Type/Value	Description
ValueId	Empty	Valid for a scalar property only.
ValueId	VT_I4 – Index	Ignored if the property is scalar. Identifies an element in a multi-valued property with a zero-based index.
ValueId	VT_BSTR – Name of a non-scalar element	Ignored if the property is scalar. If the property is multi-valued, indicates an element by name.

ISCModelProperty::Value Arguments (Get Function)

Here is the signature for the *Value (Get)* function:

```
VARIANT Value(VARIANT ValueId, VARIANT ValueType)
```

The following table contains the valid arguments for the *Value (Get)* function:

Parameter	Valid Type/Value	Description
ValueId [optional]	Empty	Valid for a scalar property only.
ValueId [optional]	VT_BSTR – Name of a non-scalar element	Ignored if the property is scalar. If the property is multi-valued, indicates an element by name.
ValueId [optional]	VT_I4 – Index of a non-scalar element	Ignored if the property is scalar. If the property is multi-valued, indicates an element by a zero-based index.
ValueType [optional]	Empty	Indicates a native datatype for return values.
ValueType [optional]	VT_I4 – SCVT_DEFAULT	Indicates a native datatype for return values.
ValueType [optional]	VT_I4 – SCVT_BSTR	Indicates a conversion to a string for return values.

ISCModelProperty::Value Arguments (Set Function)

Here is the signature for the *Value (Set)* function:

```
void Value(VARIANT ValueId, VARIANT ValueType, VARIANT Val)
```

The following table contains the valid arguments for the *Value (Set)* function:

Parameter	Valid Type/Value	Description
ValueId [optional]	Empty	Valid for a scalar property only.
ValueId [optional]	VT_I4 – Index of a non-scalar property	Indicates a value position with a zero-based index in a non-scalar property. A value of -1 causes a new value to be added at the end of the vector.
ValueId [optional]	VT_BSTR – Name of the element in a multi-valued property	Indicates a value position with the given name.

Parameter	Valid Type/Value	Description
ValueType [optional]	Empty	Not used
Val	Dependent upon the property type	

ISCModelProperty::GetValueFacetIds Arguments

Here is the signature for the *GetValueFacetIds* function:

```
VARIANT_BOOL GetValueFacetIds(Long* FacetsTrueBasket, Long* FacetsFalseBasket)
```

The following table contains the valid arguments for the *GetValueFacetIds* function:

Parameter	Valid Type/Value	Description
FacetsTrueBasket	SAFEARRAY(VT_I4) – Array of facet IDs	Lists facets that are set and have TRUE as a value.
FacetsFalseBasket	SAFEARRAY(VT_I4) – Array of facet IDs	Lists facets that are set and have FALSE as a value.

Note: More information about *FacetsTrueBasket* and *FacetsFalse Basket* is located in the Property Bag for Application Environment (see page 163) section.

ISCModelProperty::GetValueFacetNames Arguments

Here is the signature for the *GetValueFacetNames* function:

```
VARIANT_BOOL GetValueFacetNames(BSTR* FacetsTrueBasket, BSTR* FacetsFalseBasket)
```

The following table contains the valid arguments for the *GetValueFacetNames* function:

Parameter	Valid Type/Value	Description
FacetsTrueBasket	SAFEARRAY(VT_BSTR) – Array of facet names	Lists facets that are set and have TRUE as a value.
FacetsFalseBasket	SAFEARRAY(VT_BSTR) – Array of facet names	Lists facets that are set and have FALSE as a value.

Note: More information about *FacetsTrueBasket* and *FacetsFalse Basket* is located in the Property Bag for Application Environment (see page 163) section.

ISCModelProperty::SetValueFacets Arguments

Here is the signature for the *SetValueFacets* function:

void SetValueFacets(VARIANT FacetsTrueBasket, VARIANT FacetsFalseBasket)

The following table contains the valid arguments for the *SetValueFacets* function:

Parameter	Valid Type/Value	Description
FacetsTrueBasket	SAFEARRAY(VT_I4) – array of facet IDs	A list of facets to be set to TRUE.
FacetsTrueBasket	SAFEARRAY(VT_BSTR) – array of facet names	A list of facets to be set to TRUE.
FacetsTrueBasket	VT_BSTR – string with facet names separated by semicolon	A list of facets to be set to TRUE.
FacetsFalseBasket	SAFEARRAY(VT_I4) – array of facet IDs	A list of facets to be set to FALSE.
FacetsFalseBasket	SAFEARRAY(VT_BSTR) – array of facet names	A list of facets to be set to FALSE.
FacetsFalseBasket	VT_BSTR – string with facet names separated by semicolon	A list of facets to be set to FALSE.

Note: More information about *FacetsTrueBasket* and *FacetsFalse Basket* is located in the Property Bag for Application Environment (see page 163) section.

ISCModelPropertyCollection

The *ISCModelPropertyCollection* interface is a collection of properties for a given model object. Membership in this collection can be limited by establishing filter criteria.

The following table contains the methods for the *ISCModelPropertyCollection* interface:

Method	Description
IUnknown _NewEnum()	Constructs an instance of the collection enumerator object.
ISCModelProperty * Add(VARIANT ClassId)	Construct a new property for a bound model object if it does not exist.

Method	Description
SC_CLSID * ClassIds()	Returns a *SAFEARRAY* of property class identifiers in the property collection. Represents a value of the *ModelProperties* collection attribute that limited the membership at the time when this collection was created and can be used for reference purposes. *ClassIds* contain an array of acceptable class identifiers (such as property classes). If this list is non-empty, the property collection includes only those properties whose class identifier appears on the list. If the list is empty or the caller supplies a NULL pointer, the collection includes all the properties owned by the object.
BSTR * ClassNames()	Same as the *ClassIds* property, but returns a *SAFEARRAY* of property type names in the property collection.
long Count()	Number of properties in the collection.
VARIANT_BOOL HasProperty(VARIANT ClassId, VARIANT MustBeOn [optional], VARIANT MustBeOff [optional])	Returns TRUE if the object owns a property of the passed class. Treats properties as absent if they fail to satisfy *ClassIds*, *MustBeOn*, and *MustBeOff* attributes of the collection. Alternative *MustBeOn*, *MustBeOff* can be offered using optional parameters.
VARIANT_BOOL HasPropertyFacets(VARIANT ClassId, VARIANT MustBeOn [optional], VARIANT MustBeOff [optional], VARIANT FacetsMustBeSet [optional])	Returns TRUE if the object owns a property of the passed class. Treats properties as absent if they fail to satisfy *ClassIds*, *MustBeOn*, and *MustBeOff* attributes of the collection. Alternative *FlagsMustBeOn*, *FlagsMustBeOff*, *FacetsMustBeSet* can be offered using optional parameters. *FacetsMustBeSet* indicates that a property must have one or more facets. The parameter can be either a *SAFEARRAY* of the facet's ID numbers, a *SAFEARRAY* of the facet's name strings, or a string with facet names separated by a semicolon.

Method	Description
ISCModelProperty * Item(VARIANT Class)	Returns a model object property. The method checks if the property exists. If it does not, the method creates a property description, returns an *ISCModelProperty* instance, and sets the NULL flag for the property. A new property value can be set by using the *Value* property of the instance. However, it will fail to retrieve a value before it is set. The method allows you to create an instance of *ISCModelProperty* for properties like *ReadOnly*, *Maintained By the Tool*, and so on. The value for these properties cannot be changed or assigned. Yet property flags, datatype, and so on are available even when the collection does not have the property instance. Use *HasProperty* to check on the existence of the property for a model object instance.
SC_ModelPropertyFlags MustBeOff()	Filter on property flags in the collection. The filter is set when the property collection is created through the *ISCModelObject::CollectProperties* method.
SC_ModelPropertyFlags MustBeOn()	Filter on property flags in the collection. The filter is set when the property collection is created through the *ISCModelObject::CollectProperties* method.
VARIANT_BOOL Remove(VARIANT ClassId)	Removes the indicated property from the bound object. Successful execution of the call renders all binds with the removed property invalid. The client should release all *ISCModelProperty* pointers, and all related Value Collection and Value pointers known to represent such an association. Calls to interfaces fail and the *IsValid* method returns FALSE.

Note: For information about valid property class identifiers and valid property class names, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder. More information about *SC_ModelPropertyFlags* is located in the Enumerations (see page 159) section.

ISCModelPropertyCollection::Add Arguments

Here is the signature for the *Add* function:

```
ISCModelProperty * Add(VARIANT ClassId)
```

The following table contains the valid arguments for the *Add* function:

Parameter	Valid Type/Value	Description
ClassId	VT_BSTR – Name of a property class	Provides a new property type name.
ClassId	VT_BSTR – ID of a property	Provides a new property class identifier.

Note: For information about valid property class identifiers and valid property class names, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

ISCModelPropertyCollection::HasProperty Arguments

Here is the signature for the *HasProperty* function:

```
VARIANT_BOOL HasProperty(VARIANT ClassId, VARIANT MustBeOn, VARIANT MustBeOff)
```

The following table contains the valid arguments for the *HasProperty* function:

Parameter	Valid Type/Value	Description
ClassId	VT_BSTR – Name of a property	Identifies a class name for a property.
ClassId	VT_BSTR – ID of a property	Identifies a class identifier for a property.
MustBeOn [optional]	VT_I4 – SC_ModelPropertyFlags that must be set	Provides a set of required flags.
MustBeOn [optional]	Empty	Default is set to the *MustBeOn* filter that was used to create the property collection.
MustBeOff [optional]	VT_I4 – SC_ModelPropertyFlags that must not be set	Provides a set of flags that must not be set.
MustBeOff [optional]	Empty	Default is set to the *MustBeOff* filter that was used to create the property collection.

Note: For information about valid property class identifiers and valid property class names, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder. More information about *SC_ModelPropertyFlags* is located in the Enumerations (see page 159) section.

ISCModelPropertyCollection::HasPropertyFacets Arguments

Here is the signature for the *HasPropertyFacets* function:

VARIANT_BOOL HasPropertyFacets(VARIANT ClassId, VARIANT FlagsMustBeOn, VARIANT FlagsMustBeOff, VARIANT FacetsMustBeSet)

The following table contains the valid arguments for the *HasPropertyFacets* function:

Parameter	Valid Type/Value	Description
ClassId	VT_BSTR – Name of a property	Identifies a class name for a property.
ClassId	VT_BSTR – ID of a property	Identifies a class identifier for a property.
FlagsMustBeOn [optional]	VT_I4 – SC_ModelPropertyFlags that must be set	Provides a set of required flags.
FlagsMustBeOn [optional]	Empty	Default is set to the *MustBeOn* filter that was used to create the property collection.
FlagsMustBeOff [optional]	VT_I4 – SC_ModelPropertyFlags that must not be set	Provides a set of flags that must not be set.
FlagsMustBeOff [optional]	Empty	Default is set to the *MustBeOff* filter that was used to create the property collection.
FacetsMustBeSet [optional]	SAFEARRAY(VT_I4) – array of facet IDs	Indicates one or more facets that a property must have.
FacetsMustBeSet [optional]	SAFEARRAY(VT_BSTR) – array of facet names	Indicates one or more facets that a property must have.
FacetsMustBeSet [optional]	VT_BSTR – string with facet names separated by semicolon	Indicates one or more facets that a property must have.
FacetsMustBeSet [optional]	Empty	No facet requirements

Note: For information about valid property class identifiers and valid property class names, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder. More information about *SC_ModelPropertyFlags* is located in the Enumerations (see page 159) section. More information about *FacetsMustBeSet* is located in the Property Bag for Application Environment (see page 163) section.

ISCModelPropertyCollection::Item Arguments

Here is the signature for the *Item* function:

```
ISCModelProperty * Item(VARIANT Class)
```

The following table contains the valid arguments for the *Item* function:

Parameter	Valid Type/Value	Description
Class	VT_BSTR – ID of a property	Provides the property class identifier.
Class	VT_BSTR – Name of a property	Provides the property class name.

Note: For information about valid property class identifiers and valid property class names, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

ISCModelPropertyCollection::Remove Arguments

Here is the signature for the *Remove* function:

```
VARIANT_BOOL Remove(VARIANT ClassId)
```

The following table contains the valid arguments for the *Remove* function:

Parameter	Valid Type/Value	Description
ClassId	ISCModelProperty *	Identifies a property with a Model Property object.
ClassId	VT_BSTR – Name of the property	Identifies the property with a class name.
ClassId	VT_BSTR – ID of the property	Identifies the property with a class identifier.

Note: For information about valid property class identifiers and valid property class names, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

ISCModelSet

A Model Set component provides access to a member of a hierarchically organized collection of model sets.

The following table contains the methods for the *ISCModelSet* interface:

Method	Description
SC_MODELTYPEID ClassId()	Class identifier for metadata associated with the model set.
BSTR ClassName()	Class name for metadata associated with the model set.
VARIANT_BOOL DirtyBit()	Returns a flag that indicates that the data has changed in the model set.
void DirtyBit(VARIANT_BOOL)	Sets the flag that indicates that the data in the model set has changed.
SC_MODELTYPEID ModelSetId()	Passes back an identifier for the model set.
BSTR Name()	Passes back a persistence unit name.
ISCModelSet * Owner()	A pointer to the owner model set. Returns NULL for the top model set in the persistence unit.
ISCModelSetCollection * OwnedModelSets()	Provides a collection with directly owned model sets.
SC_MODELTYPEID PersistenceUnitId()	The identifier for the persistence unit that contains the model set.
ISCPropertyBag * PropertyBag(VARIANT List [optional], VARIANT AsString [optional])	Returns a property bag with the model set's properties. A model set property is present in the resulting bag only if it has a value. If the property does not have any value set, the property bag will not have the property listed.
void PropertyBag(VARIANT List [optional], VARIANT AsString [optional], ISCPropertyBag * propBag)	Sets a model set with the properties in the given property bag.

Note: For information about metadata class identifiers and names, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

ISCModelSet::PropertyBag Arguments (Get Function)

Here is the signature for the *PropertyBag (Get)* function:

```
ISCPropertyBag * PropertyBag(VARIANT List, VARIANT AsString)
```

The following table contains the valid arguments for the *PropertyBag (Get)* function:

Parameter	Valid Type/Value	Description
List [optional]	VT_BSTR – Semicolon separated list of properties	Provides a list of the model set properties. If the list is provided, only listed properties are placed in the returned property bag.
List [optional]	Empty	Requests a complete set of properties.
AsString [optional]	VT_BOOL – TRUE or FALSE	If set to TRUE, requests that all values in the bag to be presented as strings. The default is FALSE with all values in their native format.
AsString [optional]	Empty	All values in the property bag are presented in native type.

Note: More information about property names is located in the Property Bag for Persistence Units and Persistence Unit Collections (see page 173) section.

ISCModelSet::PropertyBag Arguments (Set Function)

Here is the signature for the *PropertyBag (Set)* function:

```
void PropertyBag(VARIANT List, VARIANT AsString, ISCPropertyBag * propBag)
```

The following table contains the valid arguments for the *PropertyBag (Set)* function:

Parameter	Valid Type/Value	Description
List [optional]		Not used
AsString [optional]		Not used
propBag	ISCPropertyBag *	A pointer on a property bag with the model set properties to process.

ISCModelSetCollection

A Model Set Collection contains all model sets directly owned by an owner model set.

The following table contains the methods for the *ISCModelSetCollection* interface:

Method	Description
IUnknown _NewEnum()	Constructs an instance of a model set enumerator object.
long Count()	Number of model sets in the collection.
ISCPersistenceUnit * Item(VARIANT nIndex)	Passes back a pointer for a ModelSet component.
ISCModelSet * Owner()	Returns a pointer to the owner model set.

ISCModelSetCollection::Item Arguments

Here is the signature for the *Item* function:

```
ISCModelSet * Item(VARIANT nIndex)
```

The following table contains the valid arguments for the *Item* function:

Parameter	Valid Type/Value	Description
nIndex	VT_UNKNOWN – Pointer to ISCPersistenceUnit	Creates a clone for the Model Set object.
nIndex	VT_I4 – Index of a model set in the model set collection	Ordered position in the collection. The index is zero-based.
nIndex	VT_BSTR – Model Set ID	Model set identifier.
nIndex	VT_BSTR – Metadata Class ID	Class identifier for metadata associated with a model set.
nIndex	VT_BSTR – Metadata Class name	Class name for metadata associated with a model set.

Note: For information about metadata class identifiers and names, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

ISCPersistenceUnit

A Persistence Unit encapsulates the information required to connect to an existing, outer level persistence unit within an application.

The following table contains the methods for the *ISCPersistenceUnit* interface:

Method	Description
VARIANT_BOOL DirtyBit()	Returns a flag that indicates that the data has changed in the persistence unit.
void DirtyBit(VARIANT_BOOL)	Sets the flag that indicates that the data in the persistence unit has changed.
VARIANT_BOOL HasSession()	Returns TRUE if a unit has one or more sessions connected.
VARIANT_BOOL IsValid()	Returns TRUE if self is valid.
ISCModelSet * ModelSet()	Passes back a pointer on the top model set in the Persistence Unit.
BSTR Name()	Passes back a persistence unit name.
SC_MODELTYPEID ObjectId()	Passes back an identifier for the persistence unit.
ISCPropertyBag * PropertyBag(VARIANT List [optional], VARIANT AsString [optional])	Returns a property bag with the persistence unit's properties. A unit property is present in the resulting bag only if it has a value. If the property does not have any value set, the property bag will not have the property listed.
void PropertyBag(VARIANT List [optional], VARIANT AsString [optional], ISCPropertyBag * propBag)	Sets a persistence unit with the properties in the given property bag.
VARIANT_BOOL Save(VARIANT Locator [optional], VARIANT Disposition [optional])	Persists model data to external storage. Uncommitted transactions are ignored.

Note: More information about property descriptions is located in the Property Bag for Persistence Units and Persistence Unit Collections (see page 173) section.

ISCPersistenceUnit::PropertyBag Arguments (Get Function)

Here is the signature for the *PropertyBag (Get)* function:

```
ISCPropertyBag * PropertyBag(VARIANT List, VARIANT AsString)
```

The following table contains the valid arguments for the *PropertyBag (Get)* function:

Parameter	Valid Type/Value	Description
List [optional]	VT_BSTR – Semicolon separated list of properties	Provides a list of the unit properties. If the list is provided, only listed properties are placed in the returned property bag.
List [optional]	Empty	Requests a complete set of properties.
AsString [optional]	VT_BOOL – TRUE or FALSE	If set to TRUE, it requests that all values in the bag be presented as strings. The default is FALSE and all values are in their native format.
AsString [optional]	Empty	All values in the property bag are presented in native type.

Note: More information about valid property names is located in the Property Bag for Persistence Units and Persistence Unit Collections (see page 173) section.

ISCPersistenceUnit::PropertyBag Arguments (Set Function)

Here is the signature for the *PropertyBag (Set)* function:

```
void PropertyBag(VARIANT List, VARIANT AsString, ISCPropertyBag * propBag)
```

The following table contains the valid arguments for the *PropertyBag (Set)* function:

Parameter	Valid Type/Value	Description
List [optional]		Not used
AsString [optional]		Not used
propBag	ISCPropertyBag *	A pointer on a property bag with the unit properties to process.

ISCPersistenceUnit::Save Arguments

Here is the signature for the *Save* function:

VARIANT_BOOL Save(VARIANT Locator, VARIANT Disposition)

The following table contains the valid arguments for the *Save* function:

Parameter	Valid Type/Value	Description
Locator [optional]	VT_BSTR – Full path to a storage location	Provides a new location for the persistence unit data source as a string with a file or mart item location, along with the attributes required for successful access to storage.
Locator [optional]	Empty	Indicates the use of the original persistence unit location.
Disposition [optional]	VT_BSTR – List of keywords parameters	Specifies changes in access attributes, such as read-only.

Note: More information about the format of the *Locator* parameter is located in the Locator Property section.

ISCPersistenceUnit::ReverseEngineer

Here is the signature for the ReverseEngineer function:

HRESULT ReverseEngineer ([in]ISCPropertyBag * PropertyBag,[in]VARIANT REoptionpath,[in] VARIANT REConnectionString,[in] VARIANT REPassword);

The following table contains the valid arguments for the ReverseEngineer function:

Parameter	Valid Type/Value	Description
PropertyBag	ISCPropertyBag * – Pointer to a Property Bag object.	Contains options for reverse engineering.
REoptionpath	VT_BSTR – Path.	Specifies the full path to the items storage for reverse engineering.
REConnectionString	VT_BSTR – Database connection string.	Identifies the database connect string.
REPassword	VT_BSTR – Connection password. Null for windows authentication.	Identifies the password used for database connection.

The following table contains the valid arguments for the PropertyBag parameter.

Parameter	Valid Type/Value	Description
System_objects	VT_BOOL -- True or False. Default: False	Retrieves system objects. True: System objects are retrieved. False: System objects are not retrieved.
Oracle_Use_DBA_Views	VT_BOOL – True or False Default: False. Only valid for Oracle.	Use DBA Views for reverse engineering. True: Use DBA Views. False: Do not use DBA Views.
Synch_Table_Filter_By_Name	VT_BSTR Default: Null	Reverse engineers the tables that contain the input filter strings.Multiple filter strings are specified as comma separated values.
Synch_Owned_Only	VT_BOOL – True or False. Default: False	Retrieves tables and views of users. True: Retrieve from current user or owners. False: Retrieve from all.
Synch_Owned_Only_Name	VT_BSTR Default: Null	Reverse engineers tables and views owned by the specified users.
Case_Option	25090:None 25091:lower 25092:Upper Default: None	Specifies the case conversion option for physical names.
Logical_Case_Option	25045: None 25046: UPPER 25047: lower 25048:Mixed Default: None	Specifies the case conversion option for logical names.
Infer_Primary_Keys	VT_BOOL– True or False. Default: None	Infers primary key columns for the tables that are based on defined indexes. True: Primary Keys option is selected. False: Primary Keys option is not selected.

Parameter	Valid Type/Value	Description
Infer_Relations	VT_BOOL– True or False. Default: False	Infers the relationships between tables that are based on either primary key column names or defined indexes. True: Relations Option is selected. False: Relations Option is not selected.
Infer_Relations_Indexes	VT_BOOL– True or False. **Note:** Set the value to Indexes or Names when Infer_Relations is set to Relations. Default: False.	Infers the relationships from the table indexes. True: Indexes option is selected. False: Names option is selected.
Remove_ERwin_Generated_Triggers	VT_BOOL—True or False. Default: True.	Removes erwin generated triggers. True: Remove Include Generated Triggers. False: Do not remove Include Generated Triggers.
Force_Physical_Name_Option	VT_BOOL—True or False. Default: Force	Overrides the physical name property for all objects in logical/physical models automatically during reverse engineering. True: Force physical name option. False: Do not force physical name option.

Connection String

Server=<Target Server type>:<MajorVersion>:<MinorVersion>
|AUTHENTICATION=<AuthenticationType>|USER=<UserName>|<ServerParameter>=<ServerParameterValue>

Example:

SERVER=16:10:0|AUTHENTICATION=4|USER=erwin|1=3|2=r8|3=127.0.0.1\\erwin_mart01

The following table describes the valid values for a connection string.

Parameter	Value	Description
SERVER	<TargetServerType> is an integer value. 1: Access 2: DB2 3: DB2UDB 4: Foxpro 5: Inforrmix 6: Ingres 7: ISeries 8: MySQL 9: ODBC 9: PostgreSQL 10: Oracle 11: Progress 12: Redbrick 13: SAS 14: Sybase 15: SybaseIQ 16: SQLServer 17: Teradata 18: SQLAzure 19. Hive	Specifies the type of the database server.
AUTHENTICATION	4 or 8 4: Database authentication 8: Windows authentication	Specifies the authentication type.
User	User Name	Specifies the user name.

The following table describes the type and value of ServerParameter:

Server Parameter	Server Parameter Value	Description
1	2 or 3	2: Indicates "Use ODBC data source". 3: Indicates "Use Native Connection"
2	String	Identifies the database.
3	String	Identifies the server name.
4	String	Identifies the alternate catalog name.
5	String	Identifies the ODBC data source name.
6	String	Identifies the connection string for the database.
7	String	Identifies the access database path.
8	String	Identifies the system database path.
9	String	Identifies the password for access system database.
10	Boolean 0 or 1	0: ODBC data browse is turned off. 1: ODBC data browse is turned on.
11	Boolean 0 or 1	0: Do not use encrypted connection. 1: Use encrypted connection.
12	Boolean 0 or 1	0: Do not connect to Oracle as SYSDBA. 1: Connect to Oracle as SYSDBA.
13 **Note: Applicable only to Hive**	1 or 2 or 3	1: REDB using Hive 2: REDB using MySQL Metastore 3: REDB using PostgreSQL Metastore

Note: For the target database, Hive, an additional server parameter, 13, is required as shown in the following example:

For REDB-PureHive**:**

Call oPersistenceUnit.ReverseEngineer(oPropertyBag,, "SERVER=19:2:1|AUTHENTICATION=4|USER=<hive-user>|1=2|5=<cloudera dsn>|10=0|**13=1**", "<hive-password>")

For REDB-Metastore MySQL**:**

Call oPersistenceUnit.ReverseEngineer(oPropertyBag,, "SERVER=19:2:1|AUTHENTICATION=4|USER=<mysql-user>|1=2|5=<mysql dsn>|10=0|**13=2**", "<mysql-password>")

For REDB-Metastore PostgreSQL:

Call oPersistenceUnit.ReverseEngineer(oPropertyBag,, "SERVER=19:2:1|AUTHENTICATION=4|USER=<postgresql-user>|1=2|5=<postgresql dsn>|10=0|**13=3**", "<postgresql-password>")

Reverse Engineering Sample Script:

```
Dim oAPI
        Set oAPI = CreateObject("erwin9.SCAPI.9.0")

    Dim oPropertyBag
        Set oPropertyBag = CreateObject("erwin9.SCAPI.PropertyBag.9.0")

    Call oPropertyBag.Add("Model_Type", "Combined")
    Call oPropertyBag.Add("Target_Server", 1075859016)
    Call oPropertyBag.Add("Target_Server_Version", 10)

    Dim oPUnitCol
    Set oPUnitCol = oApi.PersistenceUnits

    Dim oPersistenceUnit
    Set oPersistenceUnit = oPUnitCol.Create(oPropertyBag)

    'oPropertyBag = CreateObject("erwin9.SCAPI.PropertyBag.9.0")
    'oPropertyBag = oApi.ApplicationEnvironment.PropertyBag
    oPropertyBag.ClearAll()
    Call oPropertyBag.Add("System_Objects", True)
    Call oPropertyBag.Add("Oracle_Use_DBA_Views", False)
    Call oPropertyBag.Add("Synch_Owned_Only", False)
    Call oPropertyBag.Add("Synch_Owned_Only_Name", "")
    Call oPropertyBag.Add("Case_Option", 25091)
    Call oPropertyBag.Add("Logical_Case_Option", 25046)
    Call oPropertyBag.Add("Infer_Primary_Keys", False)
    Call oPropertyBag.Add("Infer_Relations", False)
    Call oPropertyBag.Add("Infer_Relations_Indexes", False)
    Call oPropertyBag.Add("Remove_ERwin_Generated_Triggers", False)
```

```
    Call oPropertyBag.Add("Force_Physical_Name_Option", False)
        Call oPropertyBag.Add("Synch_Table_Filter_By_Name", "")

    Call oPersistenceUnit.ReverseEngineer(oPropertyBag, "c:\\re.xml",
"SERVER=16:10:0|AUTHENTICATION=4|USER=erwin|1=3|2=r8|3=127.0.0.1\\erwin_mart01", "ca123456")
Call oPersistenceUnit.Save("c:\\test.erwin", "OVF=Yes")
```

ISCPersistenceUnit::ForwardEngineer

Here is the signature for the *ForwardEngineer_DB* function:

```
HRESULT FEModel_DB([in] VARIANT ConnectionInfo, [in] VARIANT Password, [in] VARIANT OptionXML, [out, retval]
VARIANT_BOOL *ppVal);
```

The following table contains the valid arguments for the *ForwardEngineer* function:

Parameter	Valid Type/Value	Description
ConnectionInfo	VT_BSTR	Specifies the connection string to the database. For more information, see Connection Sting in ISCPersistenceUnit::ReverseEngineer.
Password	VT_BSTR Null if the authentication type is Windows.	Specifies the connection password to the database.
OptionXML	VT_BSTR	Specifies the full path to items storage for forward engineering.

Here is the signature for the *ForwardEngineer_DDL* function:

```
HRESULT FEModel_DDL([in] VARIANT Locator, [in] VARIANT OptionXML, [out, retval] VARIANT_BOOL *ppVal);
```

Parameter	Valid Type/Value	Description
Locator	VT_BSTR	Specifies the full path of the output script file. (.sql/.ddl)
OptionXML	VT_BSTR	Specifies the full path to items storage for forward engineering.
ppVal	VT_BOOL	Specifies a return value.

Forward Engineering Sample Script:

```
Dim oAPI
        Set oAPI = CreateObject("erwin9.SCAPI.9.0")
        Dim oPersistenceUnit
    Set oPersistenceUnit = oApi.PersistenceUnits.Add("c:\\test.erwin", "")
    Call
oPersistenceUnit.FEModel_DB("SERVER=16:10:0|AUTHENTICATION=8|USER=erwin|1=3|2=ModelTest|3=127.0.0.1",
"ca123456", "c:\\fe.xml")
    Call oPersistenceUnit.FEModel_DDL("c:\\test.sql", "c:\\fe.xml")
```

ISCPersistenceUnitCollection

The *ISCPersistenceUnitCollection* contains all outer level persistence units loaded in the application. It contains one entry for each active data model.

The following table contains the methods for the *ISCPersistenceUnitCollection* interface:

Method	Description
IUnknown _NewEnum()	Constructs an instance of unit enumerator object.
ISCPersistenceUnit * Add(VARIANT Locator, VARIANT Disposition [optional])	Adds a new persistence unit to the unit collection.
VARIANT_BOOL Clear()	Purges all units from the collection.
long Count()	Number of persistence units in the collection.
ISCPersistenceUnit * Create(ISCPropertyBag * PropertyBag, VARIANT ObjectId [optional])	Creates a new unit, and registers the unit with the collection.
ISCPersistenceUnit * Item(VARIANT nIndex)	Passes back an *IUnknown* pointer for a PersistenceUnit component.
VARIANT_BOOL Remove(VARIANT Selector, VARIANT Save [optional])	Removes a persistence unit from the collection.

Note: More information about property descriptions is located in the Property Bag for Persistence Units and Persistence Unit Collections (see page 173) section.

ISCPersistenceUnitCollection::Add Arguments

Here is the signature for the *Add* function:

```
ISCPersistenceUnit * Add(VARIANT Locator, VARIANT Disposition)
```

The following table contains the valid arguments for the *Add* function:

Parameter	Valid Type/Value	Description
Locator	VT_BSTR – Persistence unit location	Identifies a location for the persistence unit data source as a string with a file or mart item location, along with the attributes required for successful access to storage.
Disposition [optional]	VT_BSTR – List of keywords parameters	Arranges access attributes, such as read only.

Note: More information about the *Locator* and *Disposition* parameters is located in the Locator Property section.

ISCPersistenceUnitCollection::Create Arguments

Here is the signature for the *Create* function:

```
ISCPersistenceUnit * Create(ISCPropertyBag * Property Bag, VARIANT ObjectId)
```

The following table contains the valid arguments for the *Create* function:

Parameter	Valid Type/Value	Description
Property Bag	ISCPropertyBag * – Pointer to a Property Bag object	Supplies required and optional properties to the creation process, such as type of the model.
ObjectId [optional]	Empty	Generates an ID for the new persistence unit.
ObjectId [optional]	VT_BSTR – Object ID for the new persistence unit	Provides an identifier for the new persistence unit.

Note: More information about property names and format is located in the Property Bag for Persistence Units and Persistence Unit Collections (see page 173) section.

ISCPersistenceUnitCollection::Item Arguments

Here is the signature for the *Item* function:

```
ISCPersistenceUnit * Item(VARIANT nIndex)
```

The following table contains the valid arguments for the *Item* function:

Parameter	Valid Type/Value	Description
nIndex	VT_UNKNOWN – Pointer to ISCPersistenceUnit	Creates a clone for the *Persistence Unit* object.
nIndex	VT_I4 – Index of a persistence unit in the persistence unit collection	Ordered position in the collection. The index is zero-based.
nIndex	VT_BSTR – ID of a persistence unit	Application-wide unique persistence unit identifier.

ISCPersistenceUnitCollection::Remove Arguments

Here is the signature for the *Remove* function:

```
VARIANT_BOOL Remove(VARIANT Selector, VARIANT Save)
```

The following table contains the valid arguments for the *Remove* function:

Parameter	Valid Type/Value	Description
Selector	VT_UNKNOWN – Pointer to ISCPersistenceUnit interface	Identifies the persistence unit.
Selector	VT_BSTR – ID of a persistence unit	Application-wide unique persistence unit identifier.
Selector	VT_I4 – Index of a persistence unit in the persistence unit collection	Ordered position in the collection. The index is zero-based.
Save [optional]	VT_BOOL	If set to TRUE, it saves the persistence unit prior to removing it from the collection. By default, all unsaved data is saved unless the *Save* parameter has a FALSE value, or the unit has a temporary status with an unspecified location property.

Note: Models should be closed prior to exiting the application. Add the following line in your code to provide a call to explicitly close the model prior to exiting your application:

```
...
SaveNewPersistenceUnit(ThePersistenceUnit, DefaultFileName)
TheApplication.PersistenceUnits.Remove(ThePersistenceUnit, False)
...
```

ISCPropertyBag

The *ISCPropertyBag* interface is used to set and access the properties of *ISCApplicationEnvironment*, *ISCPersistenceUnit*, and *ISCModelSet*. The *ISCPropertyBag* is also used to set the properties of a new persistence unit.

The following table contains the methods for the *ISCPropertyBag* interface:

Method	Description
VARIANT_BOOL Add(BSTR Name, VARIANT Value)	Adds a new property to the bag. Does not check for duplicate names. Returns TRUE if the property was added to the bag, otherwise, it is FALSE.
void ClearAll()	Removes all properties from the bag.
long Count()	Returns the number of properties.
BSTR Name(long PropertyIdx)	Retrieves the indicated property name in the bag.
VARIANT Value(VARIANT Property)	Retrieves the indicated property in the bag.
void Value(VARIANT Property, VARIANT Val)	Sets the indicated property in the bag.

ISCPropertyBag::Add Arguments

Here is the signature for the *Add* function:

```
VARIANT_BOOL Add(BSTR Name, VARIANT Value)
```

The following table contains the valid arguments for the *Add* function:

Parameter	Valid Type/Value	Description
Name	BSTR	Name of a new property.
Value	Dependent on the property	Value for a new property.

ISCPropertyBag::Name Arguments

Here is the signature for the *Name* function:

```
BSTR Name(long PropertyIdx)
```

The following table contains the valid arguments for the *Name* function:

Parameter	Valid Type/Value	Description
PropertyIdx	Long	A zero-based index for the requested name.

ISCPropertyBag::Value Arguments (Get Function)

Here is the signature for the *Value (Get)* function:

```
VARIANT Value(VARIANT Property)
```

The following table contains the valid arguments for the *Value (Get)* function:

Parameter	Valid Type/Value	Description
Property	VT_BSTR – Name of the property	Identifies retrieved property.
Property	VT_I4 – Index of the property	Zero-based property index in the Property Bag.

ISCPropertyBag::Value Arguments (Set Function)

Here is the signature for the *Value (Set)* function:

```
void Value(VARIANT Property, VARIANT Val)
```

The following table contains the valid arguments for the *Value (Set)* function:

Parameter	Valid Type/Value	Description
Property	VT_BSTR – Name of the property	Identifies the property to update.
Val	Dependent on the property	Value for the given property.

ISCPropertyValue

The *ISCPropertyValue* interface is a single value of a given property.

The following table contains the methods for the *ISCPropertyValue* interface:

Method	Description
SC_ValueTypes * GetSupportedValueIdTypes()	Groups a list of supported value types for the current value identifier and returns it as a SAFEARRAY. The *GetValue* method must be able to convert the current value into any value type whose code appears in the returned list. If the list is empty, the value is available only in its native (such as default) format. Reference properties must return an empty list.
SC_ValueTypes * GetSupportedValueTypes()	Groups a list of supported value types and returns it as a SAFEARRAY. The *GetValueId* method must be able to convert the current value into any value type whose code appears in the returned list. If the list is empty, then the current identifier is available only in its native (such as default) format.
SC_CLSID PropertyClassId()	Returns the class identifier of the current property.
BSTR PropertyClassName()	Returns the class name of the current property.
VARIANT Value(VARIANT ValueType [optional])	Converts the current value to the passed value type.
VARIANT ValueId(VARIANT ValueType [optional])	Uniquely identifies the value in a non-scalar property.
SC_ValueTypes ValueIdType()	Passes back the default type of the *ValueId* that identifies the value within the non-scalar property.
SC_ValueTypes ValueType()	Passes back the default type of the property value.

Note: More information about value data types is located in the SC_ValueTypes (see page 162) section.

ISCPropertyValue::ValueId Arguments

Here is the signature for the *ValueId* function:

```
VARIANT ValueId(VARIANT ValueType)
```

The following table contains the valid arguments for the *ValueId* function:

Parameter	Valid Type/Value	Description
ValueType [optional]	VT_I4 – SCVT_I2 or SCVT_I4	Returns *VT_EMPTY* if property is scalar. If it is non-scalar, the value of the zero-based index of the property is returned.
ValueType [optional]	VT_I4 – SCVT_BSTR	Returns *VT_EMPTY* if the property is scalar, returns the name of the non-scalar property member if it is available, or else it returns the index of the member.
ValueType [optional]	VT_I4 – SCVT_DEFAULT	Returns *VT_EMPTY* if the property is scalar. If it is non-scalar, the value of the zero-based index of the property is returned.
ValueType [optional]	Empty	Defaults to *SCVT_Default*.

ISCPropertyValue::Value Arguments

Here is the signature for the *Value* function:

```
VARIANT Value(VARIANT ValueType)
```

The following table contains the valid arguments for the *Value* function:

Parameter	Valid Type/Value	Description
ValueType [optional]	VT_I4 – SCVT_DEFAULT	Identifies a request for the property value in native format.
ValueType [optional]	VT_I4 – SCVT_BSTR	Identifies a request for the string conversion for the property value.
ValueType [optional]	VT_I4 – Type of property	Identifies a target for type conversion.
ValueType [optional]	Empty	Defaults to *SCVT_DEFAULT*.

ISCPropertyValueCollection

The *ISCPropertyValueCollection* interface is a collection of values for a non-scalar property.

The following table contains the methods for the *ISCPropertyValueCollection* interface:

Method	Description
IUnknown _NewEnum()	Constructs an instance of the collection enumerator object.
long Count()	Number of values in the collection.
ISCPropertyValue * Item(VARIANT ValueId)	Returns a single value from the property value collection.
VARIANT_BOOL Facet (VARIANT Facet)	Retrieves a facet. It fails if the facet is not set. Facet is either a facet ID or facet name.
void Facet (VARIANT Facet, VARIANT_BOOL Val)	Sets a facet with the given value. Facet is either a facet ID or facet name.
VARIANT_BOOL RemoveFacet (VARIANT Facet)	Removes a facet to non-set state. Facet is either a facet ID or facet name.

ISCPropertyValueCollection::Item Arguments

Here is the signature for the *Item* function:

```
ISCPropertyValue * Item(VARIANT ValueId)
```

The following table contains the valid arguments for the *Item* function:

Parameter	Valid Type/Value	Description
ValueId	VT_I4 – Index of the element in multi-valued property	Identifies an element with a zero-based index.
ValueId	VT_BSTR – Name of an element in a multi-valued property	Identifies an element by name.

ISCPropertyValueCollection::Facet Arguments (Get Function)

Here is the signature for the *Facet (Get)* function:

VARIANT_BOOL Facet (VARIANT Facet)

The following table contains the valid arguments for the *Facet (Get)* function:

Parameter	Valid Type/Value	Description
Facet	VT_I4 – Facet ID	Retrieves a facet value. It fails if the facet is not set.
Facet	VT_BSTR – Facet name	Retrieves a facet value. It fails if the facet is not set.

Note: More information is located in the Property Bag for Application Environment (see page 163) section.

ISCPropertyValueCollection::Facet Arguments (Set Function)

Here is the signature for the *Facet (Set)* function:

Void Facet (VARIANT Facet, VARIANT_BOOL Val)

The following table contains the valid arguments for the *Facet (Set)* function:

Parameter	Valid Type/Value	Description
Facet	VT_I4 – Facet ID	Sets a facet with the given value. It fails if the facet is not set.
Facet	VT_BSTR – Facet name	Sets a facet with the given value. It fails if the facet is not set.

Note: More information is located in the Property Bag for Application Environment (see page 163) section.

ISCPropertyValueCollection::RemoveFacet Arguments

Here is the signature for the *RemoveFacet* function:

VARIANT_BOOL RemoveFacet (VARIANT Facet)

The following table contains the valid arguments for the *RemoveFacet* function:

Parameter	Valid Type/Value	Description
Facet	VT_I4 – Facet ID	Removes a facet to non-set state.
Facet	VT_BSTR – Facet name	Removes a facet to non-set state.

Note: More information is located in the Property Bag for Application Environment (see page 163) section.

ISCSession

The *ISCSession* interface is an active connection between the API client and a model.

The following table contains the methods for the *ISCSession* interface:

Method	Description
VARIANT BeginTransaction()	Opens a transaction on the session. The method passes back a transaction identifier. Implementations use the identifier to scope Commit and Rollback operations. If the application does not support nested transactions, it passes back VT_EMPTY. Transaction nesting is implicit. If an API client invokes *BeginTransaction* and a transaction is already open, the new transaction is nested inside the existing one.
VARIANT BeginNamedTransaction(BSTR Name, VARIANT PropertyBag [optional])	Opens a transaction on the session. Similar to *BeginTransaction* with an option to provide a transaction name and additional properties.
VARIANT_BOOL ChangeAccess(SC_SessionFlags Flags)	Changes the model access to the specified level.
VARIANT_BOOL Close()	Disconnects self from its associated persistence unit or model set.
VARIANT_BOOL CommitTransaction(VARIANT TransactionId)	Commits the specified transaction and all nested transactions contained within it.
SC_SessionFlags Flags()	Returns a set of flags associated with the session.

Method	Description
VARIANT_BOOL IsValid()	Returns TRUE if self is valid.
VARIANT_BOOL IsTransactionEmpty(VARIANT All [optional])	TRUE if there was no data modification applied from the beginning of the outer transaction or for the duration of the current transaction. Returns TRUE with no open transaction present.
SC_SessionLevel Level()	Returns the level at which the persistence unit or model is bound. This value is valid only if the session is open.
VARIANT_BOOL IsOpen()	TRUE only if the session is open.
ISCModelObjectCollection * ModelObjects()	Creates a *ModelObject* collection for the session. The returned collection contains every object associated with the persistence unit or model set.
SC_MODELTYPEID ModelSetId()	Passes back an identifier for the model set associated with the session.
BSTR Name()	Name of the associated persistence unit or model set. Contains a valid name only when self is in the *Opened* state.
VARIANT_BOOL Open(IUnknown * Target, VARIANT Level [optional], VARIANT Flags [optional])	Binds to the persistence unit, model set, or intrinsic metamodel identified by the *Target* parameter.
ISCPersistenceUnit * PersistenceUnit()	Persistence unit associated with the session. Contains a valid pointer only when it is in the *Opened* state.
long TransactionDepth()	Returns the current depth level of the nested transaction. Returns zero if there are no active transactions present.

Note: More property information about the *BeginNamedTransaction* method is located in the Property Bag for Session (see page 178) section. More information about *SC_SessionFlags* and *SC_SessionLevel* is located in the Enumerations (see page 159) section.

ISCSession::BeginNamedTransaction Arguments

Here is the signature for the *BeginNamedTransaction* function:

```
VARIANT_BOOL BeginNamedTransaction(BSTR Name, VARIANT PropertyBag )
```

The following table contains the valid arguments for the *BeginNamedTransaction* function:

Parameter	Valid Type/Value	Description
Name	BSTR	Provides a name for a new transaction.
PropertyBag	Empty	No optional parameters.
PropertyBag	VT_UNKNOWN – Pointer to a Property Bag object	Collection of the transaction properties.

Note: More information about the transaction properties is located in the Property Bag for Session (see page 178) section.

ISCSession::CommitTransaction Arguments

Here is the signature for the *CommitTransaction* function:

```
VARIANT_BOOL CommitTransaction(VARIANT TransactionId)
```

The following table contains the valid arguments for the *CommitTransaction* function:

Parameter	Valid Type/Value	Description
TransactionId	The ID of the session	Provides a transaction identifier.

ISCSession::IsTransactionEmpty Arguments

Here is the signature for the *IsTransactionEmpty* function:

```
VARIANT_BOOL IsTransactionEmpty(VARIANT All)
```

The following table contains the valid arguments for the *IsTransactionEmpty* function:

Parameter	Valid Type/Value	Description
All	Empty	Identifies a request on the status of the current transaction.
All	VT_BOOL, FALSE	Identifies a request on the status of the current transaction.
All	VT_BOOL, TRUE	Identifies a request on the status of all transactions starting with the beginning of the outer transaction.

ISCSession::Open Arguments

Here is the signature for the *Open* function:

```
VARIANT_BOOL Open(ISCPersistenceUnit * Unit, VARIANT Level, VARIANT Flags)
```

The following table contains the valid arguments for the *Open* function:

Parameter	Valid Type/Value	Description
Target	ISCPersistenceUnit * – pointer to a persistence unit	Provides a persistence unit to attach.
Target	ISCModelSet * – pointer to a model set	Provides a model set to attach.
Target	ISCPropertyBag * – pointer to a property bag	Provides a property bag with the description of an intrinsic metamodel to attach.
Level [optional]	Empty	Defaults to *SCD_SL_M0*.
Level [optional]	SCD_SL_M0	Data-level access.
Level [optional]	SCD_SL_M1	Metamodel access.
Flags [optional]	Empty	Defaults to *SCD_SF_NONE*.

Parameter	Valid Type/Value	Description
Flags [optional]	SCD_SF_NONE	Other sessions can have access to the attached persistence unit.
Flags [optional]	SCD_SF_EXCLUSIVE	Other sessions cannot have access to the attached persistence unit.

ISCSession::RollbackTransaction Arguments

Here is the signature for the *RollbackTransaction* function:

```
VARIANT_BOOL RollbackTransaction(VARIANT TransactionId)
```

The following table contains the valid arguments for the *RollbackTransaction* function:

Parameter	Valid Type/Value	Description
TransactionId	The ID of the session	Provides a transaction identifier.

ISCSessionCollection

The Session Collection contains the active connections between the API client and the application.

The following table contains the methods for the *ISCSessionCollection* interface:

Method	Description
IUnknown _NewEnum()	Constructs an instance of a session enumerator object.
ISCSession * Add()	Construct a new, closed *Session* object, and adds it to the collection.
VARIANT_BOOL Clear()	Removes all *Session* objects from the collection
long Count()	The number of sessions in the collection.
ISCSession * Item(long nIndex)	Passes back a session identified by its ordered position.
VARIANT_BOOL Remove(VARIANT SessionId)	Removes a *Session* object from the collection. If the session is opened, it is closed before it is removed. All committed changes are saved in the persistence unit.

ISCSessionCollection::Item Arguments

Here is the signature for the *Item* function:

```
ISCSession * Item(long Index)
```

The following table contains the valid arguments for the *Item* function:

Parameter	Valid Type/Value	Description
Index	long-Index	Provides a zero-based index of a session.

ISCSessionCollection::Remove Arguments

Here is the signature for the *Remove* function:

```
VARIANT_BOOL Remove(VARIANT SessionId)
```

The following table contains the valid arguments for the *Remove* function:

Parameter	Valid Type/Value	Description
SessionId	VT_UNKNOWN – Pointer to the ISCSession interface	Identifies a session with the *Session* object.
SessionId	VT_I4 – Index in the session collection	Provides a zero-based index of a session.

Enumerations

This section contains information regarding the various enumerations for the API. The enumerations define valid values for various properties.

SC_ModelDirectoryFlags

The following table contains the properties and enumerations for SC_ModelDirectoryFlags:

Property	Enumeration	Description
SCD_MDF_DIRECTORY	0	Directory
SCD_MDF_ROOT	1	Root directory

SC_ModelDirectoryType

The following table contains the properties and enumerations for *SC_ModelDirectoryType*:

Property	Enumeration	Description
SCD_MDT_NONE	0	Type is not available
SCD_MDT_FILE	1	File system
SCD_MDT_MART	2	Mart

SC_ModelObjectFlags

The following table contains the properties and enumerations for SC_ModelObjectFlags:

Property	Flag Bit	Enumeration	Description
SCD_MOF_DONT_CARE		0	No flags are set
SCD_MOF_PERSISTENCE_UNIT	0	1	Object is a persistence unit (such as model)
SCD_MOF_USER_DEFINED	1	2	Object is user-defined (such as user-defined properties)
SCD_MOF_ROOT	2	4	Object is the root object (such as model)
SCD_MOF_TOOL	3	8	Object is maintained by the tool
SCD_MOF_DEFAULT	4	16	Object is created by the tool and not removable
SCD_MOF_TRANSACTION	5	32	Object is new or updated in a transaction and the transaction was not committed

SC_ModelPropertyFlags

The following table contains the properties and enumerations for SC_ModelPropertyFlags:

Property	Flag Bit	Enumeration	Description
SCD_MPF_DONT_CARE		0	No flags are set
SCD_MPF_NULL	0	1	Property has NULL value or no value

Property	Flag Bit	Enumeration	Description
SCD_MPF_USER_DEFINED	1	2	Property is user-defined
SCD_MPF_SCALAR	2	4	Property is scalar
SCD_MPF_TOOL	3	8	Property is maintained by the tool
SCD_MPF_READ_ONLY	4	16	Property is read-only (not used in erwin DM)
SCD_MPF_DERIVED	5	32	Property is inherited, calculated, or derived
SCD_MPF_OPTIONAL	6	64	Property is optional and can be removed

SC_SessionFlags

The following table contains the properties and enumerations for SC_SessionFlags:

Property	Enumeration	Description
SCD_SF_NONE	0	Session has non-exclusive access to its connected persistence unit. Other sessions can connect to the same persistence unit.
SCD_SF_EXCLUSIVE	1	Session has exclusive access to its connected persistence unit. No other sessions are allowed to access the persistence unit.

SC_SessionLevel

The following table contains the properties and enumerations for SC_SessionLevel:

Property	Enumeration	Description
SCD_SL_NONE	-1	Not used
SCD_SL_M0	0	Data level access
SCD_SL_M1	1	Metamodel access

SC_ValueTypes

The following table contains the properties and enumerations for SC_ValueTypes:

Property	Enumeration	Description
SCVT_NULL	0	Missing value
SCVT_I2	1	Signed 16-bit integer
SCVT_I4	2	Signed 32-bit integer
SCVT_UI1	3	Unsigned 8-bit integer. Do not use this type to hold character data.
SCVT_R4	4	4 byte floating point real
SCVT_R8	5	8 byte floating point real
SCVT_BOOLEAN	6	Boolean
SCVT_CURRENCY	7	64-bit currency value
SCVT_IUNKNOWN	8	IUnknown interface pointer
SCVT_IDISPATCH	9	IDispatch interface pointer
SCVT_DATE	10	Date value in VARIANT_DATE format
SCVT_BSTR	11	String
SCVT_UI2	12	Unsigned 16-bit integer
SCVT_UI4	13	Unsigned 32-bit integer
SCVT_GUID	14	GUID
SCVT_OBJID	15	A string (VT_BSTR) contains an object identifier with offset
SCVT_BLOB	16	SAFEARRAY of unsigned BYTEs
SCVT_DEFAULT	17	Default value type
SCVT_I1	18	Signed 1 byte integer. Do not use this type to hold character data.
SCVT_INT	19	Machine-dependent signed integer
SCVT_UINT	20	Machine-dependent unsigned integer
SCVT_RECT	21	Rectangle-array of four integers (VT_ARRAY & VT_I2)
SCVT_POINT	22	Point-array of two integers (VT_ARRAY & VT_I2)
SCVT_I8	23	Signed 64-bit integer

Property	Enumeration	Description
SCVT_UI8	24	Unsigned 64-bit integer
SCVT_SIZE	25	Size array of two integers (VT_ARRAY & VT_I4)

Property Bag Reference

This section contains information about the content of the Property Bag container. A property bag is a placeholder for an array of properties. The content of the bag is dictated by a source interface.

Property Bag for Application Environment

This property bag provides access for Application Features sets. The parameters of the *PropertyBag* call determine the context of the bag. The contents of the bag can have one of two available forms, either native format or a string based on the optional parameter of the *PropertyBag* property of the *ISCApplicationEnvironment* interface.

Feature categories in the *Category* parameter of the *PropertyBag* property are hierarchical and use a dot (.) to define feature subsets. For example, the *Application* category populates a property bag with a complete set of erwin DM features, while *Application.API* provides a subset related to the API.

If the *Category* parameter is not set, then the *Property Bag* property returns the complete set of all the features from all the available categories.

The following tables summarize the available feature categories and list the *Property Bag* properties for each category.

ISCApplicationEnvironment::PropertyBag

The *PropertyBag* function from the *ISCApplicationEnvironment* interface populates a property bag with one or more property values as indicated by Category and Name.

Here is the signature for the *ISCApplicationEnvironment PropertyBag* function:

```
ISCPropertyBag * PropertyBag(VARIANT Category, VARIANT Name, VARIANT AsString)
```

The following table contains the valid arguments for the *ISCApplicationEnvironment PropertyBag* function:

Parameter	Valid Type/Value	Description
Category [optional]	Empty	Complete set of features from all categories are returned.
Category [optional]	VT_BSTR – Name of category	Features from the given category are returned.
Name [optional]	Empty	All properties from the selected category are returned.
Name [optional]	VT_BSTR – Name of property	The property with the given name and category is returned.
AsString [optional]	Empty	All values in the property bag are presented in native type.
AsString [optional]	VT_BOOL – TRUE or FALSE	If set to TRUE, all values in the property bag are presented as strings.

Category Parameter Contains an Empty String

The following table describes the *Category* parameter that contains an empty string:

Property Name	Type	Description
Categories	SAFEARRAY(BSTR)	Returns an array of all the available categories.

Application Category

The following table describes the *Application* category, which describes the features associated with the erwin DM tool:

Property Name	Type	Description
Title	BSTR	Provides the erwin DM title.
Version	BSTR	Provides the erwin DM version.
Hosting_Application	Long	■ 0 – Returns 0 if the API is controlled by third-party application, in standalone mode. ■ 1 – Returns 1 if the erwin DM user interface is active and the API is in add-in mode.
Metadata_Version	Long	Metadata value for the current version of erwin DM.

Property Name	Type	Description
EMX_Metadata_Class	SC_MODELTYPEID	Metadata class identifier for EMX model sets.
EM2_Metadata_Class	SC_MODELTYPEID	Metadata class identifier for EM2 model sets.

Application.API Category

The following table describes the *Application.API* category, which describes the features associated with the API:

Property Name	Type	Description
API_Version	BSTR	Provides the version of the API interfaces.
API_Major_Version_Number	Long	The API major version number.
API_Minor_Version_Number	Long	The API minor version number.

Application.API.Features Category

The following table describes the *Application.API.Features* category, which summarizes the level of support the API provides in its main set of operations:

Property Name	Type	Description
Undo	Long	Describes the ability to undo operations. ■ 0 – Undo not supported. ■ Non-zero – Undo is supported.
Redo	Long	Describes the ability to redo undone operations. ■ 0 – Redo not supported. ■ Non-zero – Redo is supported.
Change_Logging	Long	Describes the ability to report all changes since the last synchronization with the client. ■ 0 – Change logging not supported. ■ Non-zero – Change logging is supported.

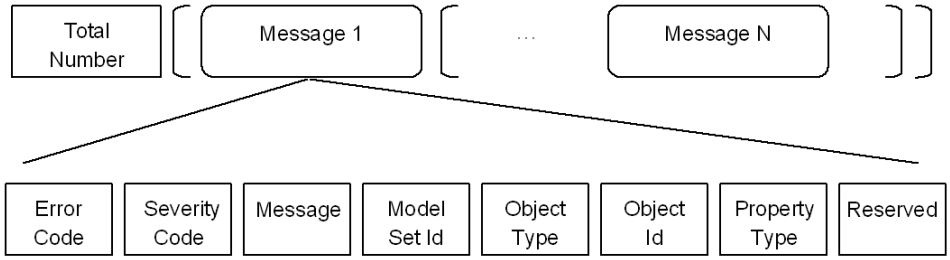
Property Name	Type	Description
Ownership_Support	Long	Queries the support level of the application for object ownership. The following describes the support levels: ■ 0 – The application does not support object ownership. ■ 1 – The application supports ownership and the ownership meta-relation contains no cycles. ■ 2 – The application supports ownership and the ownership meta-relation contains cycles.
Transactions	Long	Describes the level of support for transaction control. The following describes the support levels: ■ 0 – No support for transactions. ■ 1 – Begin and End only. No nesting. ■ 2 – Begin, End, and Rollback. No nesting. ■ 3 – Begin, End, and Rollback, with arbitrary transaction nesting.

Application.API.MessageLog Category

The following table describes the *Application.API.MessageLog* category, which provides access to additional messages registered during API operations:

Property Name	Type	Description
Is_Empty	Boolean	Returns TRUE if the message log is not empty. The log is reset before the beginning of every API operation.
Log	SAFEARRAY(VARIANT)	Returns the content of the log.

The Property Log from the *MessageLog* category is organized as a one-dimensional *SAFEARRAY* with *VARIANT* type as elements. The array has the following structure:



The following table describes the elements of the array:

Message Log Element	Type	Description
Total Number	Long	Total number of messages in the array. The value can be zero if there were no messages when the *Log* property was requested.
Error Code	BSTR	A message string identifier.
Severity Code	Long	The following are the SC_MessageLogSeverityLevels severity codes: ■ *SCD_ESL_NONE* – No severity code was assigned. ■ *SCD_ESL_INFORMATION* – Information message. ■ *SCD_ESL_WARNING* – Warning message. ■ *ESD_ESL_ERROR* – Error message.
Message	BSTR	Message text.
Model Set Id	SC_MODELSETID	An identifier of a model set associated with a message. An element has the *VARIANT* type *VT_EMPTY* if no data was provided.
Object Type	SC_CLSID	Class identifier for a model object associated with a message. An element has the *VARIANT* type *VT_EMPTY* if no data was provided.
Object Id	SC_OBJID	Identifier for a model object associated with a message. The identifier is unique in the scope of the model set. An element has the *VARIANT* type *VT_EMPTY* if no data was provided.
Property Type	SC_CLSID	Class identifier for a property associated with a message. An element has the *VARIANT* type *VT_EMPTY* if no data was provided.
Reserved		Always marked as *VT_EMPTY*.

Note: For information about object class identifiers and property class identifiers, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder. More information about using the *Model Set Identifier* to locate a model set is located in the Accessing a Model (see page 39) and Accessing a Model Set (see page 50) sections. More information about using the *Class Identifier* to learn more about object types and property types is located in the Accessing Metamodel Information (see page 89) section. More information about using the *Object Identifier* to access the associated model object is located in the Accessing Objects in a Model (see page 53) section.

Application.Modeling Category

The *Application.Modeling* category describes the features associated with the erwin DM modeling engine:

Property Name	Type	Description
Model_Property_Value_Facet_Ids	SAFEARRAY(Long)	The data is organized as a one-dimensional SAFEARRAY with the Long type as elements. The elements represent property value facet IDs available in model data. The elements are ordered to match the order in the *Model_Property_Value_Facet_Names*.
Model_Property_Value_Facet_Names	SAFEARRAY(BSTR)	The data is organized as a one-dimensional SAFEARRAY with the BSTR type as elements. The elements represent property value facet names available in model data. The elements are ordered to match the order in the *Model_Property_Value_Facet_Ids*.

The following table lists available property facets:

Property Name	Type	Description
Hardened	5	Indicates that a value will not change due to inheritance. For example, a name for a foreign key attribute in a child entity.
AutoCalculated	3	Indicates that a value is auto-calculated by the tool. For example, cardinality is auto-calculated by default. In this case, the auto-calculated facet is set to true.

Application.Modeling.Physical Category

The following table describes the *Application.Modeling.Physical* category, which describes the features associated with physical modeling in erwin DM:

Property Name	Type	Description
DBMS_Brand_And_Version_List	SAFEARRAY(Long)	The data is organized as a one-dimensional SAFEARRAY with the Long type as elements. The elements are grouped into subsets of three. The first member of the subset contains a DBMS brand identifier, the second member is the major version value, and the last member is the minor version value.

Application.Persistence Category

The *Application.Persistence* category describes the features associated with persistence support in erwin DM. There are no properties in this category.

Application.Persistence.FileSystem Category

The following table describes the *Application.Persistence.FileSystem* category, which describes the features associated with the file system:

Property Name	Type	Description
Current_Directory	BSTR	Absolute path for the current local directory.

Application.Persistence.Mart

The following table describes the *Application.Persistence.Mart* category, which describes the features associated with persistence support in erwin® Data Modeler Workgroup Edition:

Property Name	Type	Description
Mart_Connection_Types	SAFEARRAY(BSTR)	Enumerate mart supported database connection types.

Property Bag for Model Directory and Model Directory Unit

This *Property Bag* provides access to the properties of the *Model Directory* and the *Model Directory Unit* objects. The *PropertyBag* property for both the *ISCModelDirectory* interface and the *ISCModelDirectoryUnit* interface populates the bag with the set of current properties. The same property of these interfaces allows modification of directory (if it is not read-only) or directory unit attributes. The contents of the bag can have one of two available forms, either native format or as a string based on the optional parameter of the *PropertyBag* property of *ISCModelDirectory* and *ISCModelDirectoryUnit*. The client can populate the bag in either of these two forms. Different forms can be mixed in the same instance of the bag.

Not all properties that exist in the directory or directory unit have to be present in the bag when it is submitted. All property data as well as property names are validated by the API, and all are either accepted or rejected. The rejection forces a method call to fail. If the bag includes properties that are read-only at the moment, for example, the *Locator* property, then such properties are ignored and do not affect validation of the bag data.

The following table lists the *Property Bag* properties and data types for the *Model Directory*:

Property Name	Type	Read-only	Description
Directory_Name	BSTR	No	Returns a directory name without the path information. Applying a new value renames a directory. For the mart root directory, this is a repository name. The property does not allow the modification of the repository name.
Locator	BSTR	Yes	Location of a directory including absolute path and parameters. For a mart, parameters do not include password information.
Directory_Path	BSTR	Yes	Directory absolute path.
Created_By	BSTR	Yes	Identification for a user that has created a directory. For erwin® Data Modeler Workgroup Edition only, a mart user ID is used.
Updated_By	BSTR	Yes	Identification for a user that has updated a directory. For erwin® Data Modeler Workgroup Edition only, a mart user ID is used.

Property Name	Type	Read-only	Description
Created	SAFEARRAY(Long)	Yes	Creation date of a directory. The time is an array of numbers in the following order: ■ Seconds after minute (0 - 59) ■ Minutes after hour (0 - 59) ■ Hours since midnight (0 - 23) ■ Day of month (1 - 31) ■ Month (0 - 11; January = 0) ■ Year (current year) ■ Day of week (0 - 6; Sunday = 0) ■ Day of year (0 - 365; January 1 = 0)
Updated	SAFEARRAY(Long)	Yes	Update date of a directory. The time is an array of numbers in the following order: ■ Seconds after minute (0 - 59) ■ Minutes after hour (0 - 59) ■ Hours since midnight (0 - 23) ■ Day of month (1 - 31) ■ Month (0 - 11; January = 0) ■ Year (current year) ■ Day of week (0 - 6; Sunday = 0) ■ Day of year (0 - 365; January 1 = 0)
Description	BSTR	No	A directory description. This is only for erwin® Data Modeler Workgroup Edition.

The following table lists the *Property Bag* properties and datatypes for the *Model Directory Unit*:

Property Name	Type	Read-only	Description
Directory_Unit_Name	BSTR	No	Returns a directory unit name without path information. Applying a new value renames a directory unit.
Locator	BSTR	Yes	Location of a directory unit including absolute path and parameters. For a mart, parameters do not include password information.

Property Name	Type	Read-only	Description
Directory_Unit_Path	BSTR	Yes	Directory unit absolute path.
Created_By	BSTR	Yes	Identification for a user that has created a unit. For erwin® Data Modeler Workgroup Edition only, a mart user ID is used.
Updated_By	BSTR	Yes	Identification for a user that has updated a unit. For erwin® Data Modeler Workgroup Edition only, a mart user ID is used.
Created	SAFEARRAY(Long)	Yes	Creation date of a directory. The time is an array of numbers in the following order: ■ Seconds after minute (0 - 59) ■ Minutes after hour (0 - 59) ■ Hours since midnight (0 - 23) ■ Day of month (1 - 31) ■ Month (0 - 11; January = 0) ■ Year (current year) ■ Day of week (0 - 6; Sunday = 0) ■ Day of year (0 - 365; January 1 = 0)
Updated	SAFEARRAY(Long)	Yes	Update date of a directory. The time is an array of numbers in the following order: ■ Seconds after minute (0 - 59) ■ Minutes after hour (0 - 59) ■ Hours since midnight (0 - 23) ■ Day of month (1 - 31) ■ Month (0 - 11; January = 0) ■ Year (current year) ■ Day of week (0 - 6; Sunday = 0) ■ Day of year (0 - 365; January 1 = 0)
Description	BSTR	Yes	A directory description. This is only for erwin® Data Modeler Workgroup Edition.
Model_Type	Long	Yes	Retrieves the type of a unit model. A model type can be logical, logical/physical, or physical.

Property Name	Type	Read-only	Description
Object_Count	Long	Yes	Reports total number of objects in the unit. This is only for erwin® Data Modeler Workgroup Edition.
Entity_Count	Long	Yes	Reports total number of entity objects in the unit. This is only for erwin® Data Modeler Workgroup Edition.
Is_Template	Boolean	Yes	Returns TRUE if a unit model is a template.

Property Bag for Persistence Units and Persistence Unit Collections

This *Property Bag* provides access to the properties of a persistence unit. An empty *Property Bag* can be obtained through a call to the *CoCreateInstance* of the COM API. The client populates a bag and then submits it as a parameter for the *Create* method of the *ISCPersistenceUnitCollection* interface. Alternatively, the present state of persistence unit properties can be retrieved through the *PropertyBag* property of *ISCPersistenceUnit*. The retrieved value can be reviewed, modified, and submitted back through the *PropertyBag* property of the same interface. The contents of the bag can have one of two available forms: native format or as a string based on the optional parameter of the *PropertyBag* property of the *ISCPersistenceUnit*. The client can populate the bag in either of these two forms. Different forms can be mixed in the same instance of the bag.

Not all properties that exist in the unit have to be present in the bag when it is submitted. All property data as well as property names are validated by the API and either all are accepted or all are rejected. The rejection forces a method call to fail. If the bag includes properties that are read-only at the moment, for instance, the model type for a erwin DM model when the model was created previously, then such properties are ignored and will not affect validation of the bag data.

ISCPersistenceUnit::PropertyBag Arguments (Get Function)

Here is the signature for the *PropertyBag (Get)* function:

```
ISCPropertyBag * PropertyBag(VARIANT List, VARIANT AsString)
```

The following table contains the valid arguments for the *PropertyBag (Get)* function:

Parameter	Valid Type/Value	Description
List [optional]	VT_BSTR – Semicolon separated list of properties	Provides a list of the unit properties. If the list is provided, only listed properties are placed in the returned property bag.
List [optional]	Empty	Requests a complete set of properties.
AsString [optional]	VT_BOOL – TRUE or FALSE	If set to TRUE, it requests that all values in the bag be presented as strings. The default is FALSE and all values are in their native format.
AsString [optional]	Empty	All values in the property bag are presented in native format.

ISCPersistenceUnit::PropertyBag Arguments (Set Function)

Here is the signature for the *PropertyBag (Set)* function:

```
void PropertyBag(VARIANT List, VARIANT AsString, ISCPropertyBag * propBag)
```

The following table contains the valid arguments for the *PropertyBag (Set)* function:

Parameter	Valid Type/Value	Description
List [optional]		Not used
AsString [optional]		Not used
propBag	ISCPropertyBag *	A pointer on a property bag with the unit properties to process

Property Bag Contents for Persistence Unit and Persistence Unit Collection

The following table lists the Property Bag properties and datatypes recognized by erwin DM:

Property Name	Type	Read-only	Description
Locator	BSTR	Yes	Returns the location of the persistence unit, such as file name. Not available for models without a persistence location, such as new models that were never saved.
Disposition	BSTR	Yes	Returns the disposition of the persistence unit, such as read-only.
Persistence_Unit_Id	SC_MODELTYPEID	No	Retrieves and sets an identifier for the persistence unit. A new identifier can be assigned to the existing persistence unit. In this case, the old identifier will be placed in the persistence unit's branch log. **Note:** For more information, see the description of the *Branch Log* property.
Branch_Log	SAFEARRAY (SC_MODELTYPEID)	After create	Retrieves and sets the branch log of the persistence unit identifiers. A persistence unit retains its log of identifiers. erwin DM uses the branch logs of the persistence units for extended identification match. The API uses only the most current identifier for searching in the *Persistence Unit Collection*.

Property Name	Type	Read-only	Description
Model_Type	Long	After create	Retrieves and sets the type of the persistence unit, such as logical, logical/physical, and physical models. Can be set when a persistence unit is created; after that the property becomes read-only. Available values are: ■ 1 – Logical, for logical models. This is the default if no value is provided. ■ 2 – Physical, for physical models. ■ 3 – Combined, for a logical/physical model.
Target_Server Target_Server_Version Target_Server_Minor_Version	Long	After create	Retrieves and sets the target database properties for physical and logical-physical models. Can be set when a persistence unit is created; after that the property becomes read-only. **Note:** For available values for the *Target_Server* property, see the next table.
Storage_Format	Long	After create	Retrieves and sets the storage format, which has a value of *Normal* for a model and a value of *Template* for a model template. Can be set when a persistence unit is created; after that the property becomes read-only. Available values are: ■ 4012 – Normal, for a regular model. This is the default if no value is provided. ■ 4016 – Template, for a template model.
Active_Model	Boolean	No	TRUE if the persistence unit represents the current model and is active in the erwin DM user interface. Not available when using the API in standalone mode.

Property Name	Type	Read-only	Description
Hidden_Model	Boolean	No	TRUE if a model window with the persistence unit data is not visible in the erwin DM user interface. Not available when using the API in standalone mode.
Active_Subject_Area_and_Stored_Display	SAFEARRAY(BSTR)	No	Reports names of active Subject Area and Stored Display model objects. This indicates the Subject Area and Stored Display that erwin DM shows on the screen. The returned value is a safe array with two elements. The first element is a name for the active Subject Area and the second element is for the Stored Display. Providing a new set of Subject Area and Stored Display names can change this selection. The change has an effect immediately if the model is active in the erwin DM user interface or in the next model opened by the erwin DM user interface. Optionally, to change a selection, you need only a *BSTR* with a name for a new Subject Area. From the Subject Area you provide, the API chooses the first Stored Display as active.

The *Target_Server* property is a vector that consists of three members. The first member of the vector contains a DBMS brand identifier, the second member is the major version value, and the last member is the minor version value.

The following table lists DBMS brand identifiers for the *Target_Server* property. The table also lists the brand names that are used when the identifier is presented as a string:

DBMS Brand	DBMS Brand Name	DBMS Brand ID
DB2 for i	DB2	1075859019
DB2 for LUW	DB2 UDB	1075858977
DB2 for z/OS	FoxPro	1075858978
Hive	Hive	1075859187
Informix	Informix	1075859006

DBMS Brand	DBMS Brand Name	DBMS Brand ID
MySQL	Ingres	1075859129
ODBC/Generic	ODBC	1075859009
Oracle	Oracle	1075858979
PostgreSQL	PostgreSQL	1075918977
Progress	Progress	1075859010
SAS	SAS	1075859013
SQL Server	SQL Server	1075859016
SQL Azure	SQL Azure	1075859180
SAP ASE	Sybase	1075859017
SAP IQ	Sybase	1075859130
Teradata	Teradata	1075859018

Property Bag for Session

This Property Bag provides additional information to the *BeginNamedTransaction* function of the *ISCSession* interface and can be submitted as the second optional argument of the function. The contents of the bag can have one of two available forms: native format or as a string. The client can populate the bag in either of these two forms. Different forms can be mixed in the same instance of the bag.

Not all properties have to be present in the bag when it is submitted. All property data as well as property names are validated by the API, and all are either accepted or rejected. The rejection forces a method call to fail.

The transaction properties are in effect at the initiation of an outer transaction and are confined to the scope of the transaction.

The following table lists the Property Bag properties and datatypes for the *BeginNamedTransaction*:

Property Name	Type	Read-only	Description
History_Tracking	Boolean	No	TRUE – Indicates that all historical information generated during the transaction will be marked as the API event. A TRUE value is assumed if the property is not provided. FALSE – Uses the standard erwin DM mechanism of history tracking.
History_Description	BSTR	No	When the *History_Tracking* property is TRUE, it provides the content of the history event Description field.

Location and Disposition in Model Directories and Persistence Units

The API describes the location of Persistence Units and their disposition in persistence storage facilities with the *Locator* and *Disposition* properties. This information is required by some of the API methods and is also accessible using Property Bags. Examples of persistence storage for erwin DM models are file system and mart.

Locator Property

The following table describes the syntax supported by the *Locator* property:

Syntax	Arguments
[provider://]pathinfo[?param=value[;param=value]…n]	provider: This is a type of persistence storage. Use *erwin* to specify file system, and use *mart* for a mart. If this is skipped, *erwin* is the default. pathinfo: This is the path to the storage location, which is either a file path or the mart path. param: This is either a parameter name or a keyword. value: This is a text string.

There are no param keywords defined for the file system persistence storage.

A list of *Locator* param keywords for use with the *mart* type of provider for models stored in a mart is described in the following table.

Note: There is a special arrangement for the erwin® Data Modeler Workgroup Edition *Locator*. Part of the *Locator* string with params can be omitted if an application has connections open with one or more mart repositories. In this case, the params part of the *Locator* string can have only partial information or not be present at all, as long as it is clear to which connection from the available list it refers.

Currently, erwin® Data Modeler Workgroup Edition allows only one open connection to a mart repository at any given time. Therefore, it is possible, after establishing a connection, to omit the params part of the *Locator* string completely and to provide the model path information only.

The following table provides a list of *Locator* param keywords for use with the *mart* type of provider for models stored in a mart:

Complete Name	Abbreviation	Description
Server	SRV	Location where the application server exists.
Trusted Connection	TRC	This is an optional parameter. When set to YES--it instructs to use the Windows authentication model for login validation. When set to No or when the value is not mentioned--it instructs to use username and password to log in, in which case the *UID* and *PSW* keywords must be specified.
Version Number	VNO	Version number of the model.

Complete Name	Abbreviation	Description
User	UID	Login user name. Do not specify *UID* when using Windows Authentication.
Password	PSW	User login password. Do not specify *PSW* if you use Windows Authentication (Trusted Connection set to YES).
Port Number	PRT	Port number to which the application server listens.
Application Name	ASR	Name of the application server.
IIS	IIS	This is an optional parameter. YES--connects to MartServer using IIS. No or not mentioned about this property--instructs to use PortNo to connect MartServer, in which case PortNo must be specified.

The following table describes various scenarios in which you can use the *Locator* param keyword along with the *mart* type of provider for models stored in a mart:

Scenario	Description
erwin® Data Modeler Workgroup Edition	Your Libraries/ Models are stored in the Mart under the catalog named “Mart”. Mart is the default name, you can change it. A library can contain a library. If a library that is specified in path does not exist in the Mart, the library is created at the time of saving the model and the model is stored in that library. If you have a model named *MyModel* located in *MyLib*, which is in an SSL secured *Mart*, you can use the following: mart://<CatlogName>/<Libraryname>/<ModelMName>?VNO=<versionno>;TRC=NO;SRV=<ServerLocation>;PRT=<portno>;ASR=<ApplicationServerName>;SSL=<YES/NO>;UID= <user id>;PSW=<password> For example: mart://Mart/MyLib/MyModel?VNO=1;TRC=NO;IIS=NO;SRV=<ServerLocation>;PRT=<portno>;ASR=<ApplicationServerName>;SSL=<YES>;UID= <user id>;PSW=<password>
Local drive	If you have a model called *mod.erwin* located in the models directory on the C drive, you can use the following: C:\models\mod.erwin

Disposition Property

The *Disposition* parameter provides optional information for the API to access model data specified by the *Locator* parameter.

The following table describes the syntax supported by the *Disposition* property:

Syntax	Arguments
param=value[;param=value]…n	param: This is either a parameter name or a keyword. value: Yes/No/specified values for some params.

The following table lists *Disposition* param keywords for use with the *erwin* type of provider, such as for models stored in the file system:

Complete Name	Abbreviation	Description
Read Only	RDO	Requests read-only access to a file. Available for the *Persistence Unit Collection Add* method. Full access to a persistence unit is possible if the parameter was not specified.
Overwrite File	OVF	Overwrites an existing file upon Save. Available for the *Persistence Unit Save* method. There is no overwrite if the parameter is not specified.
Main Subject Area	MSA	Keep the main Subject Area Value: Yes/No
Diagram	DGM	Keep the diagrams for the main Subject Area Value: Yes/No
Theme	THM	Apply a default theme to each diagram Value: Yes/No

Complete Name	Abbreviation	Description
Transforms	XFM	Transform object view should be converted into a specified type. The param values are: 1. ”XFM=RESOLVE” (default value if the parameter was not specified) [It converts the model transform objects into Target object view] 2. ”XFM=REVERSE” [It converts the model transform objects into Source object view)] 3. ”XFM=CONVERT”[It converts the Model transform objects into current view in which the model is having]

For example: The disposition parameter is as follows:
(“RDO=Yes;MSA=Yes;DGM=NO;THM=Yes;XFM=”REVERSE”)

The following table lists *Disposition* param keywords for use with the *mart* type of provider for models opened from, or stored in a mart:

Complete Name	Abbreviation	Description
Read Only	RDO	Request a read only access to a model while opening it from Mart.
Overwrite Session	OVS	Overwrite an existing session. If the parameter is not specified, it uses the existing session; if not, it creates a session.
Overwrite Model	OVM	Overwrite an existing model in a mart. Available for the *Persistence Unit Save* method. There is no overwrite if the parameter is not specified.

Appendix B: erwin DM Metamodel

This appendix lists information regarding the erwin DM metamodel.

Note: For more information, see the HTML document *erwin Metamodel Reference*, in the Metamodel Reference Bookshelf located in the erwin® Data Modeler installation folder.

This section contains the following topics:

Metadata Element Renaming

Metadata element renaming affects object types, property types, and API-specific property types. In r7.3, much of the metadata in erwin DM was renamed. These name changes fall into two categories:

- Consistent naming and better representation of the model data. For example, the property type *For* was renamed to *For_Character_Type*.
- Replacement of space characters with underscores in all metadata element names. Prior to erwin DM r7.3, both object type and property type names accessed using the API contained spaces, but when saving to XML format, those same names used underscores. To remove this inconsistency, all space characters within such names have been replaced by underscores.

Overall, this change is transparent and will not affect your day-to-day work. Awareness of this change, however, is important if you use the API and the new ODBC interface, and have some familiarity with the pre-r7.3 metadata names. Existing API applications and scripts must be updated to account for any new metadata names before use with erwin DM. To assist you with this updating process, the following CSV files are provided with the erwin DM installation in the <Program Files>\erwin\Data Modeler r9\metadata changes:

Renamed Metadata (SCAPI).csv

Provides a list of the full set of changed metadata names. It is a two column CSV file that contains the old name, new name pairs.

Renamed Metadata (XML).csv

Provides the subset of metadata names that appear as changed in XML files.

Note: Not included in this file are those metadata names where the only change was the replacement of space characters with underscores, since erwin DM's XML format already uses underscores in object type names and property type names.

Renamed SCAPI Properties.csv

Provides a list of the API-only property names that were renamed.

Metadata Organization

The metadata includes object and property classes, object aggregations, and property associations.

Object classes

Define the type of objects that may occur within a model such as an entity class, an attribute class, or a relationship class.

Property classes

Define the type of properties an object may have such as the *Name* property, *Comment* property, or *Parent_Domain_Ref* property.

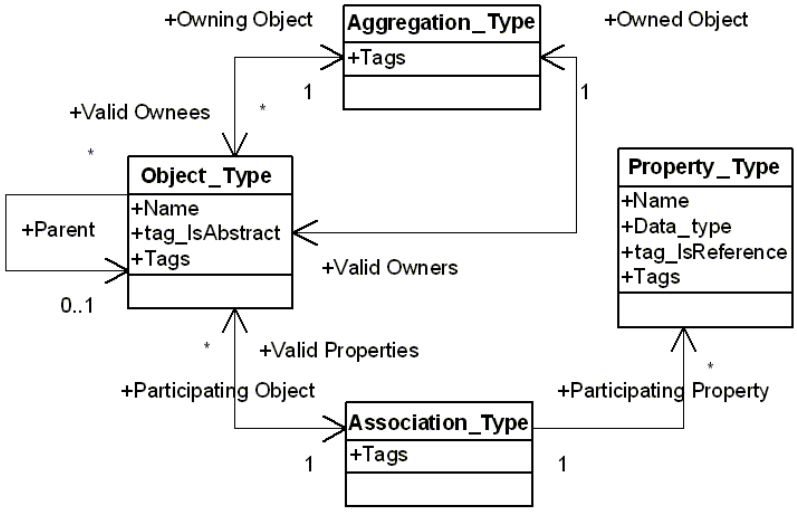
Object aggregations

Identify an ownership relationship between classes of objects, such as a model that owns entities, or entities that own attributes, and so on.

Property associations

Define property usage by object classes. For example, the metadata includes property associations for every object class that has the *Name* property.

The following diagram shows the organization of the metadata:

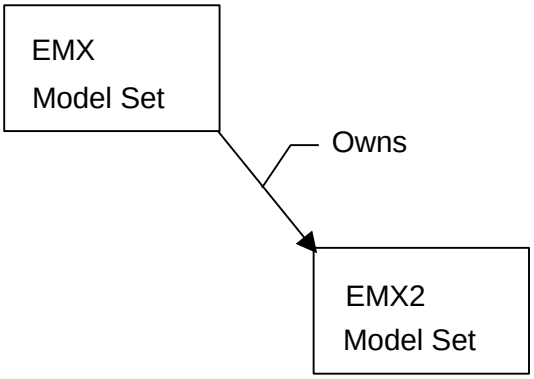


Metamodel Elements

erwin DM organizes data as a group of linked model sets. The model sets are arranged in a tree-like hierarchy with a single model set at the top.

The top model set contains the bulk of the modeling data. The API uses the abbreviation EMX to identify the top model set.

The EMX model set owns a secondary model set, abbreviated as EM2, which contains user interface settings and user options for erwin DM services such as Forward Engineering, Complete Compare, and so on.



The API clients access the model data by constructing a session and connecting it to a model set using the *Session* component.

A model set contains several levels of data. It contains the data the application manipulates, such as entity instances, attribute instances, relationship instances, and so on.

The model set also contains metadata, a description of the objects and properties that may occur within the application's data.

Metadata Tags

Each metadata object may include one or more tags. A tag is a metadata object property that conveys certain descriptive meta information, such as if an object class is logical, physical, valid for a specific target DBMS, and so on.

Note: A tag on an object aggregation overrides the identical tag set on the associated owned object class. A tag on a property association overrides the identical tag set on the associated property class.

The following table lists some of the EMX metadata tags:

Tag Name	Datatype	Description
tag_Bit_Field_Values … tag_Bit_Field_Values_2	String	Describes valid values for a bit field property. A combination of values from the description list can be used as a value for the property. The descriptions are grouped as follows: {<value>|<string equivalent>|<internal>}
DBMS_Brands_And_Versions	Integer, vector	Defines conditions when an object or property class is available for physical modeling with the specific DBMS. Assumes that the *tag_Is_Physical* has a TRUE value. Absence of the tag indicates that the class is available for all DBMS targets, but only if *tag_Is_Physical* has a TRUE value. A NULL value for the tag indicates that the class is not available for any DBMS. DBMS brand IDs are described in the next table.
DBMS_Is_Represented	Integer, vector	Defines conditions when an object or property class represents a concept in the specific DBMS. Assumes that the *DBMS_Brands_And_Versions* tag is valid for the class. Absence of the tag indicates that the class is available for all DBMS targets, but only if the *DBMS_Brands_And_Versions* tag is valid for the class. A NULL value for the tag indicates that the class is not available for any DBMS. DBMS brand IDs are described in the next table.
DBMS_Is_Top_Level_Object	Integer, vector	Defines conditions when an object class is considered top level, such as when it has a *CREATE* or *DROP* statement associated with it for the specific DBMS. Assumes that the *DBMS_Is_Represented* tag is valid for the class. Absence of the tag indicates that the class is available for all DBMS targets, if the *DBMS_Is_Represented* tag is valid for the class. A NULL value for the tag indicates that the class is not a top level object for any DBMS. DBMS brand IDs are described in the next table.

Tag Name	Datatype	Description
tag_Enum_Values … tag_Enum_Values_10	String	Describes valid values for an enumerated property. Only one value from the description list can be used as a value for the property. The descriptions are grouped as follows: {<value>|<string equivalent>|<internal>}
tag_Is_Font_Or_Color	Boolean	TRUE for classes responsible for model data visualization.
tag_Is_For_Data_Movement	Boolean	TRUE for an object or property class that is available for dimensional and data warehouse modeling.
tag_Is_Graphic_Data	Boolean	TRUE for classes responsible for model data visualization.
tag_Is_Logical	Boolean	TRUE for an object or property class that is available for logical modeling.
tag_Is_Physical	Boolean	TRUE for an object or property class that is available for physical modeling.
tag_Holds_User_Settings	Boolean	TRUE for classes responsible for storing options for erwin DM features.

DBMS specific tags, such as *DBMS_Brands_And_Versions*, *DBMS_Is_Represented*, and *DBMS_Is_Top_Level_Object*, are vectors and organize data in groups of triplets as described below:

First element

Specifies the DBMS brand ID.

Second element

Specifies the minimum version level for the DBMS, multiplied by 1000.

Third element

Specifies the maximum version level for the DBMS, multiplied by 1000; 999000 indicates the absence of a maximum level.

For example, consider the property *Oracle_Index_Partition_Type*. It contains a DBMS-specific tag, *DBMS_Brands_And_Versions*. This tag contains three elements specific for this property: 1075858979, 8000, 999000. The first element, the DBMS brand ID, is for Oracle, which is 1075858979. The second element, the minimum version level for this DBMS, multiplied by 1000, is 8000. This means the minimum DBMS version level for this DBMS, which is Oracle, is 8.0. The third element, the maximum version level for this DBMS, is 999000, which means there is no maximum version level for this DBMS.

The following table lists DBMS brand IDs:

DBMS Brand	DBMS Brand ID
DB2 for i	1075859019
DB2 for LUW	1075858977
DB2 for z/OS	1075858978
Hive	1075859187
Informix	1075859006
MySQL	1075859129
ODBC/Generic	1075859009
Oracle	1075858979
PostgreSQL	1075918977
Progress	1075859010
SAS	1075859013
SQL Server	1075859016
SQL Azure	1075859180
SAP ASE	1075859017
SAP IQ	1075859130
Teradata	1075859018

Abstract Metadata Objects

The metadata organization makes use of generalizations with the ability to derive a specialized object class from an abstract object class using generalization association. Specialized classes can then be marked as abstract, and then they can be used as a source for further specializations.

Only instances of the concrete, non-abstract object classes may occur within the application's data. erwin DM uses the generalization mechanism to flatten metadata by replicating aggregations, associations, and tags from the abstract object classes in the concrete object classes.

Metamodel Classes

A unique metadata class identifies what type of metadata a model set contains.

EMX Class Model Set

Contains the bulk of model data such as entities and attributes. The class name is *EMX* and the class identifier is the value defined in the *Application Environment* component, category *Application*, property *EMX_Metadata_Class*.

EM2 Class Model Set

Stores additional data such as user interface settings and user options for erwin DM services such as Forward Engineering and Complete Compare. The class name is *EM2* and the class identifier is the value defined in the *Application Environment* component, category *Application*, property *EM2_Metadata_Class*.

XML Schema

You can use the XML schema provided with this product to view metadata descriptions.

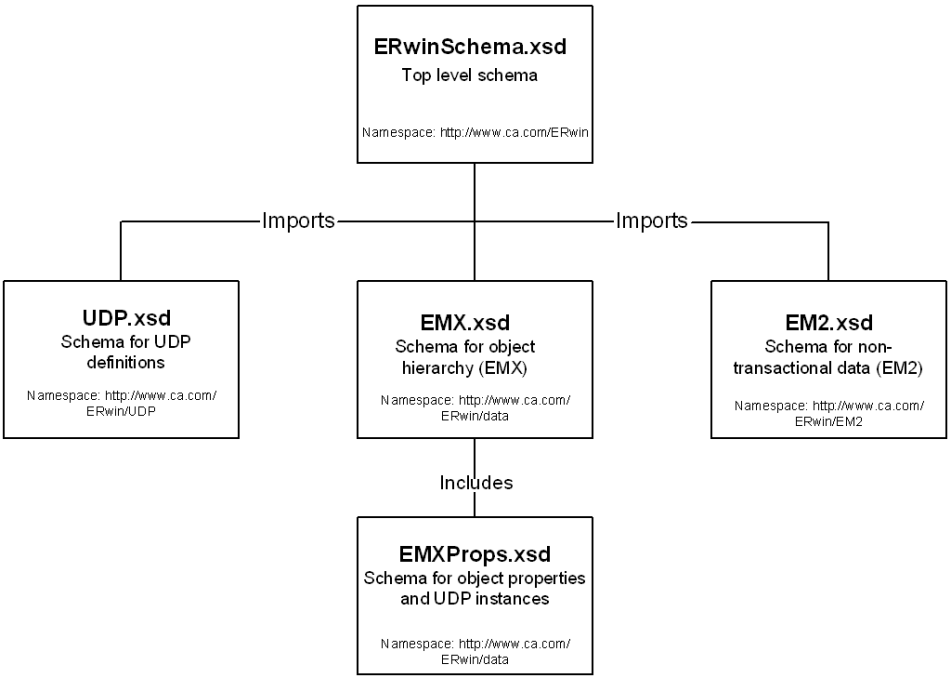
An XML schema is a document or a set of documents that defines the XML file's structure and legal elements. XML schemas can be used to ensure that an XML file is syntactically correct and conforms to the defined schema. erwin DM provides such a schema and uses the schema to validate XML files when they are opened in the tool.

The erwin DM installation places the complete set of XML schema files necessary for an XML file validation into the \Doc directory. The schema files have .xsd extensions and are described in the following list:

- erwinSchema.xsd is the top level schema file.
- UDP.xsd is the schema file for UDP definitions.
- EMX.xsd is the schema file for object hierarchy.
- EM2.xsd is the schema file for non-transactional data.
- EMXProps.xsd is the schema file for object properties and UDP instances.

XML schemas contain descriptions of model object and property classes and define property containment by object classes. Schema definitions for EMX and EM2 classes are provided. XML schemas do not include deprecated classes.

The following diagram illustrates the five erwin DM XML schema files:



The schema files under the \Doc directory are not database-specific and represent the entire erwin DM metamodel. The schema contains all possible objects and properties for all valid database targets. If you need database-specific schema, those files are located in the Doc\DBMS_schemas directory. Within the Doc\DBMS_schemas directory, there is a folder for each supported target database. The database-specific schema files are stored in that folder and only consist of objects and properties that are valid for the given database target.

Note: The XML schema that is in the \Doc directory is always used by erwin DM to validate an XML file; the database-specific schema is not used. The database-specific schemas are provided for documentation purposes and to assist third-party tool integrators to determine the valid objects and properties for a given database target. An external XML validation tool can be used to validate an XML file against a database-specific schema.