Meta Data Services Programming

Programming Meta Data Services Applications

Microsoft® SQL Server™ 2000 Meta Data Services is an object-oriented repository technology that stores and manages meta data for SQL Server and its components.

Meta Data Services is intended to store meta data, and it is designed to be integrated with other tools and applications. It provides a solution for storing and managing data warehousing definitions, OLAP definitions, design data used in development tools, and any other type of meta data used in a programming environment.

For tool and application developers, Meta Data Services provides an application programming interface (API) that exposes the repository engine and meta model definitions that the engine can manipulate.

With the repository API, you can create tools and applications that use or manipulate data already stored in your repository. You can also add new meta data to accomplish new programming objectives that you define.

Meta Data Services relies on information models to provide meta data definitions. For more information about information models, see Information Model Fundamentals and Information Models.

The following topics provide more information about how to deploy Meta Data Services in a programming environment.

Topic	Description
Repository Object Architecture	Describes repository engine objects and repository type information objects used to define and manage meta data.
Getting Started with Meta Data Services	Describes the programming environment requirements and provides basic information you should know before you start.
Connecting to and Configuring a Repository	Explains how to create and open a repository database.

Defining Information Models	Describes how to define an information model.
Installing Information Models	Explains how to install an information model in a repository database.
Programming Information Models	Describes how to program against an information model in a repository database.
Storage Strategy in a Repository Database	Explains how Meta Data Services stores data in a repository database.
Using OLE DB Scanner	Describes how to use the OLE DB Scanner utility that imports relational data into a repository database.
Using XML Encoding	Describes how to use the Meta Data Coalition (MDC) Extensible Markup Language (XML) Encoding feature for interchanging meta data in XML.

See Also

Meta Data Services Architecture

Meta Data Services Overview

Repository API Reference

Model Installer Reference

Meta Data Services Programming

Repository Object Architecture

The repository object architecture shows how the repository application programming interfaces relate and intersect. The object model is organized into two parts: one that shows the repository engine objects, and another that shows the Repository Type Information Model (RTIM).

Because the repository engine can accommodate data for any tool, its object model reflects a simple, fundamental view of data. This section describes the fundamental object model of a repository and introduces the classes and interfaces that you use to implement the object model in your code.

The following topics provide more information about the repository object architecture.

Topic	Description
Repository Engine Model	Describes the classes and interfaces that drive the repository engine.
Repository Type Information Model	Describes the classes and interfaces that define information models.
Understanding the RTIM Through Examples	Describes the components of an information model using examples.

See Also

Designing Information Models

Repository API

Repository API Reference

Meta Data Services Programming

Repository Engine Model

The repository engine model represents the classes and interfaces that drive the repository engine. Together with the Repository Type Information Model (RTIM), the repository engine model makes up the complete repository object architecture.

The repository engine model includes the following objects.

Object	Description
Repository Objects and Object Versions	An object that is known by a Microsoft® SQL Server™ 2000 Meta Data Services repository and managed by the repository engine
Repository Session Objects	An object that represents a repository instance
Repository Transaction Objects	An object that provides transaction services to a repository database
Repository Root Objects	An object that provides a starting point for information model navigation
Repository Relationship Objects	An object that defines characteristics of a repository relationship
Repository Collections	A collection that contains objects of a similar type
Repository Property Objects	An object that defines characteristics of a repository property
Repository Workspace Objects	An object that represents a workspace in a repository

See Also

Repository Object Architecture

Repository Type Information Model

Repository Objects and Object Versions

A repository object and a repository object version are either COM or Automation objects known to a Microsoft® SQL Server™ 2000 Meta Data Services repository and managed by the repository engine. When you instantiate any object, whether it is a repository engine object or an object from your information model, the repository engine instantiates it as a repository object or repository object version.

You can manipulate a repository object or object version instance from Automation or COM programs using **RepositoryObject** and **RepositoryObjectVersion** classes, objects, and interfaces. You can also use the **ObjectCol** or **VersionCol** collections.

Working with RepositoryObject Objects

Repository Type Information Model (RTIM) objects and repository engine objects are instantiated as **RepositoryObject** objects.

Working with RepositoryObjectVersion Objects

All object instances that are defined by your information model can be instantiated as **RepositoryObjectVersion** objects. Doing so enables you to create and manipulate historical or alternate versions of an object instance. In previous releases of the repository engine, both versioned and nonversioned objects were supported. The nonversioned repository object is maintained for backward compatibility purposes. In SQL Server 2000 Meta Data Services, object instances that you instantiate as either repository objects or repository object versions are functionally equivalent.

By default, most repository interfaces work with the latest version of an object. A few interfaces, such as **IRepositoryObjectVersion**, work with specific versions that you specify.

See Also

Repository Session Objects

The repository session object represents an instance of a single repository. Within a single repository, you can have multiple information models. Each repository instance is associated with one repository database.

The repository session object supports a database connection, transactions, error handling, workspaces, and object instantiation. A repository session object is created and managed by the repository engine. It is part of the repository engine model.

You can manipulate a repository instance from Automation or COM programs using the **Repository** object, **IRepository** interface, or the **Repository** class.

See Also

IRepository Interface

Repository Class

Repository Object

Repository Object Architecture

Repository Transaction Objects

A repository transaction object handles all transactions between a Microsoft® SQL Server™ 2000 Meta Data Services repository instance and a repository database. Whenever you insert, delete, or update data in your repository database, you do so by way of a transaction object. The repository transaction object also tracks the status of a transaction, and it supports options that allow you to instruct repository engine operations.

A transaction object is created and managed by the repository engine. It is part of the repository engine model.

You can manipulate a repository transaction from Automation or COM programs using the **RepositoryTransaction** object, the **Repository** class, or the **IRepositoryTransaction** and **IRepositoryTransaction2** interfaces.

See Also

IRepositoryTransaction Interface

IRepositoryTransaction2 Interface

Repository Class

Repository Object Architecture

RepositoryTransaction Object

Repository Root Objects

The root object is the top-level object in a repository. There is one root object for each repository instance. It is the object from which all navigation begins. All information models and workspaces in a repository are associated with the root object.

As with any repository object, the root object can have any number of relationships with other objects. Each relationship connecting the root object to other objects must conform to a relationship type. The relationship type to which these relationships conform is created by the information model creator. The following figure shows seven such relationships.

The root object occupies a special role that spans both parts of the repository object architecture. In the Repository Type Information Model (RTIM), it is the starting point for navigating to your information models. However, it also belongs to the Repository Engine Object model because it services the repository engine. In addition, it does not describe type information to the same extent that other RTIM objects do. Although you are not prohibited from doing so, it is better to avoid setting properties on the repository root object.

You can access a repository root object from Automation or COM programs using the **ReposRoot** object, the **ReposRoot** class, or the **IReposRoot** interface.

See Also

Repository Object Architecture

ReposRoot Class

ReposRoot Object

Repository Relationship Objects

A relationship is an association between two objects. Relationships bind objects together and give structure to a repository and an information model.

In a repository and in all subsequent information models, objects are connected to each other through a network of relationships. For example, in a model that depicts a database application, the association between a schema and its table is a relationship. Furthermore, the association between a table and its columns, and a column and its data type, are also relationships. In a repository, the connection between one information model and another is also a relationship.

All relationships are accessed by way of a collection. You can only access a relationship through its collections. Understanding how collections and relationships correspond is an important prerequisite to programming an information model. For more information about collections, see Repository Collections.

The following topics provide more detail about the roles that a relationship assumes.

Topic	Description
Relationship Structure: Origin and Destination	Explains how origin and destination objects provide the structure of a relationship.
Relationship Navigation: Source and Target	Explains how source and target objects provide the navigation of a relationship.

You can manipulate repository relationship objects from Automation or COM programs using the **Relationship** object, the **Relationship** class, or the **IRelationship** and **IRelationshipCol** interfaces.

See Also

Example: Associating Data with RTIM

IRelationship Interface

IRelationshipCol Interface

Relationship Class

Relationship Object

Repository Object Architecture

Relationship Structure: Origin and Destination

In every relationship, one object participates as the origin and one object participates as the destination. The terms origin and destination refer to the relative roles of the two objects. Together, they define the primary direction of the relationship. For any given relationship, the assignment of one particular role as the origin and the other role as the destination is arbitrary to the repository engine. In practice, however, the developer typically assigns the origin role to the object that acts or operates on the other object.

For example, in a relationship of the type *schema has tables*, **Schema** is the origin and **Tables** is the destination. In the relationship *table has columns*, **Tables** is origin and **Columns** is the destination. Notice that **Tables** can be both destination and origin, depending on its role in each relationship.

Origin and destination assignments create the structure of an information model. Within a single origin-destination pairing, the origin and destination assignments of the two objects are fixed after the assignments are made.

See Also

Relationship Navigation: Source and Target

Repository Object Architecture

Repository Relationship Objects

Relationship Navigation: Source and Target

You use relationships to navigate through repository contents. From within a relationship, you can retrieve either of the two repository objects that form the relationship.

In a relationship, navigation always moves from a source object towards a target object. Unlike origin and destination, source and target assignments are dynamic; the assignments vary depending on where you want to go. Because you can navigate back and forth across a network of objects, the source object is simply where navigation starts, and the target is where navigation concludes.

Source and target assignments apply to instantiated objects for the duration of a navigation step. Where origin and destination tend to reflect an enduring, real-world relationship that is represented in a model, source and target assignments exist only to provide navigation direction from one object to the next.

See Also

Navigation Overview

Relationship Structure: Origin and Destination

Repository Object Architecture

Repository Relationship Objects

Repository Collections

A repository collection is a set of one or more objects that implement the same interface. Repository collections are instantiated by the repository engine. State information about a collection is stored in a repository so that you can call the object in the same state in which you last left it.

Collections are used to define a relationship between two or more objects, to support navigation, and to manipulate a set of similar objects as a unit.

Collections always reflect information about some kind of relationship. An object typically has multiple collections, reflecting its association with many kinds of objects. Furthermore, because an information model is a network of objects, navigation follows a series of relationships by traversing collections.

All collections are fundamentally the same. However, the repository API provides support for creating a variety of general-purpose and special-purpose collections. The kind of collection that you create is determined by the COM interfaces and Automation objects you use to materialize the collection. Each collection exposes a set of methods and properties designed to support the purpose of the collection type.

For more information about collections, see Defining Relationships and Collections and Understanding Collections.

See Also

ITargetObjectCol Interface

ObjectCol Class

ObjectCol Object

RelationshipCol Class

RelationshipCol Object

Repository Object Architecture

TransientObjectCol Class

Repository Property Objects

A repository property object stores the persistent state of a repository object or a repository object version.

You can use a repository property object to access or manipulate any repository object in a generic way. For example, if you are creating a browsing tool, you can use repository property objects to populate the browser. The data that is returned to you is not tied to specific object instances. However, by using the information that is returned, you can retrieve more specific data about an object.

You can access a repository property object from Automation or COM programs using the **ReposProperty** object, the **ReposProperty** class, or the **IReposProperty** or **IReposProperty2** interfaces.

To associate or access multiple properties of a repository object or repository object version, use the **ReposProperties** collection.

To work with large text or image files, use **IReposPropertyLarge**.

See Also

IReposProperty Interface

Repository Object Architecture

ReposProperty Class

ReposProperty Object

ReposProperties Class

ReposProperties Object

Repository Workspace Objects

A repository workspace is a subset of a shared, central repository. You can define workspaces to materialize an information model as it existed at a specific point in time, or to create a new space for furthering application development without impacting the current code base.

A workspace object exposes methods that allow you to allocate, populate, and manage a workspace. You can only have one version of each object assigned to a workspace at a time.

A workspace object is created and managed by the repository engine. It is part of the repository engine model.

You can access a workspace object from Automation or COM programs using the **Workspace** object, the **Workspaces** collection of the **ReposRoot** object, the **Workspace** class, or the **IWorkspace** or **IWorkspaceItem** interfaces.

See Also

IWorkspace Interface

IWorkspaceItem Interface

Managing Workspaces

Repository Object Architecture

Workspace Class

Workspace Object

Meta Data Services Programming

Repository Type Information Model

The Repository Type Information Model (RTIM) is the object model that defines how information models are stored in a repository.

RTIM objects define the object classes of an information model. RTIM objects are instantiated by the repository engine as repository objects or repository object versions. RTIM objects can also be instantiated as members of a repository collection.

When you model a tool or application in an information model, the definitions must conform to the RTIM objects described in this section. Together with the repository engine model, the RTIM makes up the complete repository object architecture.

The following topics describe the parts of the RTIM model.

Object	Description
Repository Type Library Objects	An object that defines the scope of a single information model
Class Definition Objects	An object that defines a class
Interface Definition Objects	An object that defines an interface
Property Definition Objects	An object that defines a property
Method Definition Objects	An object that defines a method
Parameter Definition Objects	An object that defines a parameter of a method
Relationship Definition Objects	An object that defines a relationship type
Collection Definition Objects	An object that defines a collection type
Alias Objects	An object that defines an alias for any named object
Enumeration Definition Objects	An object that defines an enumeration
Script Definition Objects	An object that defines a script

See Also

Repository Object Architecture

Repository Type Library Objects

A repository type library object defines the scope of an information model. If you are working with a predefined information model or a modeling tool, repository type library objects are created for you when you install the information model. If you are creating type information programmatically, you must create a repository type library object to contain your type definitions.

You can access a repository type library object from Automation or COM programs using the **ReposTypeLib** object, the **ReposTypeLib** class, or the **IReposTypeLib** or **IReposTypeLib2** interfaces.

See Also

IReposTypeLib Interface

Repository Object Architecture

ReposTypeLib Class

ReposTypeLib Object

Class Definition Objects

A class definition object defines a class. In a Microsoft® SQL Server™ 2000 Meta Data Services repository, a class definition object exposes properties, a collection of interfaces, and a collection of scripts.

The following figure shows some classes and the interfaces they implement. In the figure, the **Chapter** class implements two interfaces, **ISpellingChecker** and **IPagination**. Both the **Paragraph** class and the **Chapter** class implement the **ISpellingChecker** interface.

You can access a class definition object from Automation or COM programs using the **ClassDef** object, the **ClassDef** class, or the **IClassDef** or **IClassDef2** interfaces.

See Also

ClassDef Class

ClassDef Object

IClassDef Interface

Repository Object Architecture

Interface Definition Objects

In Automation programs, each object exposes its properties, collections, and behaviors through interfaces. To have the instances of a class exhibit certain behaviors or have certain properties or collections, you implement the appropriate interface for that class.

The Repository Type Information Model (RTIM) accommodates such data by letting you describe interfaces. Each interface can have a set of classes that implements it, and each class can have a set of interfaces that it implements.

In a Microsoft® SQL Server™ 2000 Meta Data Services repository, an interface definition object exposes properties, an ancestors collection, a descendants collection, and a members collection. It also provides for interface implication and script support.

You can access an interface definition object from Automation or COM programs using the **InterfaceDef** object, the **InterfaceDef** class, or the **IInterfaceDef** or **IInterfaceDef2** interfaces.

See Also

IInterfaceDef Interface

InterfaceDef Class

InterfaceDef Object

Repository Object Architecture

Alias Objects

An alias object is a derived member of an interface. This object provides support for delegating members of an interface to other interfaces.

You can access an alias object from Automation or COM programs using the **Alias** object, the **Alias** class, or the **IInterfaceMember2** interface.

See Also

Alias Class

Alias Object

IInterfaceMember2 Interface

Repository Object Architecture

Relationship Definition Objects

A relationship definition object defines a relationship type. You can define a relationship type for relationship characteristics that repeat. For example, *table has columns* represents a type of relationship that repeats for every table that has columns. This relationship can be used to describe how **LoanTable** relates to **LoanID**, how **CustomerTable** relates to **CustomerName**, and how **OrderTable** relates to **OrderDate**.

If you are creating an information model programmatically, you should create a relationship definition object for every relationship that you implement. For more information, see Defining a Relationship.

If you have relationship definition objects that conform to the same template, you can define a relationship collection to represent the set. For more information, see Collection Definition Objects.

You can access a relationship definition object from Automation or COM programs using the **RelationshipDef** object, the **RelationshipDef** class, or the **IReposTypeInfo** interface.

See Also

Example: Associating Data with RTIM

IReposTypeInfo Interface

RelationshipDef Class

RelationshipDef Object

Repository Object Architecture

Collection Definition Objects

A collection definition is meta data about specific kinds of collections. The collection definition object defines the characteristics of a collectionand provides a template to which a collection conforms.

Typically, a collection contains a set of identically structured objects. You can use a collection definition object to create object and relationship collections that provide your tool or application with a way to manipulate sets of objects and relationships as a single unit. An object collection is a set of similar objects. A relationship collection is a set of similar relationships.

In the following example, the right column (**Data**) lists some collections by name, while the kinds of collections are in the left column under **Kind of Data**. The **Kind of Data** column indicates the templates to which the items in the **Data** column must conform. Because the items in the **Data** column are collections, the items in the **Kind of Data** column are called collection types and they conform to a collection definition.

The most important way that a collection can conform to a collection definition is in its size. That is, a collection definition describes the size limitations on any collection conforming to it. In the following table, each instance of the collection definition *publisher-of-book* describes the collection of publishers of a particular book. A typical instance of this collection definition is *publisher-of-Inside-OLE*. In the table, each book has only one publisher.

The collection definition can define this restriction. That is, the *publisher-of-book* collection definition can impose a maximum size of one on each collection conforming to it. Similarly, the collection type can define a minimum size restriction.

The following list contains some other examples:

- Publisher-of-book (zero, one).

The minimum size is zero because not every book has a publisher. The maximum size is one because no book can have two or more publishers.

- Books-of-publisher (zero, many).

  The minimum size is zero because a publisher can exist before it actually publishes any books. The maximum size is many because some publishers can publish more than one book.

- Books-of-person (zero, many).

  The minimum size is zero because not every person is an author. The maximum size is many because some people can write more than one book.

- Authors-of-book (zero, many).

  The minimum size is zero because the authors of some books are anonymous. The maximum size is many because more than one person can coauthor a book.

You can access a collection definition object from Automation or COM programs using the **CollectionDef** object, the **CollectionDef** class, or the **ICollectionDef** interface.

See Also

CollectionDef Class

CollectionDef Object

Defining a Collection

ICollectionDef Interface

Repository Object Architecture

Understanding Collections

Property Definition Objects

A property definition object defines a property. Each property has an interface that exposes it, and each interface can expose many properties.

In a Microsoft® SQL Server™ 2000 Meta Data Services repository, a property definition object exposes properties, a collection of enumeration objects, a collection of scripts, and a collection of aliases.

You can define properties that provide enumerated values or that use script to validate a property value. You can also reuse a property in a new context by assigning it an alias.

In the following example, the **IParagraph** interface exposes two properties, **Left Margin** and **Right Margin**. Both **Left Margin** and **Right Margin** are represented in a repository as property definition objects.

You can access a property definition object from Automation or COM programs using the **PropertyDef** object, the **PropertyDef** class, or the **IPropertyDef** and **IPropertyDef2** interfaces.

See Also

IPropertyDef Interface

PropertyDef Class

PropertyDef Object

Repository Object Architecture

Enumeration Definition Objects

An enumeration definition object exposes a fixed set of constant values. You can use an enumeration definition object to create a property that supports a predefined set of values to select from, or a selection list that provides data values to a user (for example, a selection of countries to choose from).

To define a value list, you use the **EnumerationValueDef** object.

You can access an enumeration definition object from Automation or COM programs using the **EnumerationDef** object, the **EnumerationDef** class, or the **IEnumerationDef** interface.

See Also

EnumerationDef Class

EnumerationDef Object

EnumerationValueDef Object

IEnumerationDef Interface

Repository Object Architecture

Method Definition Objects

An interface can expose one or more methods. A method definition object defines a method that you can attach to an interface. You can enumerate the methods for each interface of an information model. Each method can have one interface that exposes it, and each interface can expose many methods. After you define a method, you can define parameters and scripts to associate with the method.

The following figure shows that the **IParagraph** interface exposes the **Reformat** and **ConvertIndentation** methods.

You can access a method definition object from Automation or COM programs using the **MethodDef** object, the **MethodDef** class, or the **IMethodDef** interface.

See Also

Defining Methods

IMethodDef Interface

MethodDef Class

MethodDef Object

Repository Object Architecture

Parameter Definition Objects

A parameter definition object defines a parameter of a method. You can associate multiple parameters with a single method. You can also reuse a parameter on multiple methods.

You can access a parameter definition object from Automation or COM programs using the **ParameterDef** object, the **ParameterDef** class, or the **IParameterDef** interface.

See Also

Defining a Parameter

IParameterDef Interface

ParameterDef Class

ParameterDef Object

Repository Object Architecture

Script Definition Objects

A script definition object defines an implementation of a script in an information model.

You can access a script definition object from Automation or COM programs using the **ScriptDef** object, the **ScriptDef** class, or the **IScriptDef** interface.

See Also

Defining Script Objects

IScriptDef Interface

Repository Object Architecture

ScriptDef Class

ScriptDef Object

Meta Data Services Programming

Understanding the RTIM Through Examples

This section uses examples to illustrate the objects of the Repository Type Information Model (RTIM).

In addition to the examples provided here, you can review additional topics to further your understanding of information model design. For more information, see Designing Information Models.

Topic	Description
Example: Associating Data with RTIM	Describes how real-world data corresponds to RTIM objects.
Example: A Finished Information Model	Provides a description of a finished information model.

See Also

Repository Object Architecture

Example: Associating Data with RTIM

A Microsoft® SQL Server™ 2000 Meta Data Services repository contains data expressed as objects and relationships, along with their respective property values. The following figure shows some typical data for employees, projects, and subprojects. More details about this figure are provided later in this topic.

Mapping Real-World Data to RTIM objects

The preceding figure includes all the typical kinds of data you will find in a Meta Data Services repository. You can use the figure to understand the classes in the repository object architecture. In the figure, you can see instances of the following classes.

Class	Description
Repository	Describes a repository session. The figure as a whole represents an instance of the **Repository** class.
RepositoryObject	Describes a repository object. The figure shows 12 objects, one of which is the root object; each additional instance is a dot.
ReposRoot	Describes the root object. The root object is the top-level object in a repository from which navigation begins. The root object can have any number of relationships with other objects. The figure shows seven such relationships.
Relationship	Describes an association between two objects. The figure shows 15 relationships; each relationship is an arrow.
RelationshipCol	Describes a set of similar relationships. The items in a relationship collection must have the same source, and the relationships must be the same type For example, consider the relationships between

	Projects and Subprojects. The Genome project is related to Research Design and to Splicing Algorithms. Both relationships have the same source (Genome) and the same type (*includes*), thereby meeting the criteria for a relationship collection. Consider a second relationship collection: the set of Mike's assignments to subprojects. In the figure, this relationship collection appears as a pair of arrows emerging from the dot representing Mike.
TargetObjectCol	Describes a set of objects. For example, one set of objects is the set of subprojects on which Mike works; the set contains two items.

Drilling Down into Relationship Roles

Understanding roles in a relationship is one of the more difficult aspects to information modeling. The following section draws out some of the complexity of relationships by expanding on the example.

Same Object in Same Role

In the relationship collection shown in the following figure, every relationship uses the object describing Mike as the performer of the work on a subproject. The object describing Mike is the origin object in this relationship.

In contrast, the set of relationships shown in the following figure does not constitute a collection because there is no object that all the relationships use in the same role. In fact, the relationships have no object in common, regardless of role.

Common Object in Different Roles

The following figure shows employees and their managers.

The set of relationships shown in the following figure does not constitute a valid relationship collection.

Every relationship in the preceding figure is of the same relationship type, the *manages* type. All the relationships have an object in common: the object describing Frank. One relationship, however, has Frank in the role of person being managed, whereas the other relationships have Frank in the role of person who is managing someone else. Because the relationships do not all use the same object in the same role, they do not constitute a valid relationship collection.

The three relationships in the following figure do constitute a valid collection because Frank is in the manager role for all three relationships.

See Also

Repository Object Architecture

Understanding the RTIM Through Examples

Example: A Finished Information Model

The following figure shows a complete information model that illustrates the various parts of the Repository Type Information Model (RTIM). Details about this figure are provided later in this topic.

The information model in the preceding figure maintains data about files and directories. Thus, there are two classes, **File** and **Directory**.

There are three interfaces:

- **IFile** exposes behavior unique to files. Thus, only the **File** class implements the **IFile** interface.

- **IDirectory** exposes behavior unique to directories. Thus, only the **Directory** class implements the **IDirectory** interface.

- **IDirectoryItem** exposes behavior appropriate to any object that can appear as an item within a directory. Since files can be contained in directories, the **File** class implements **IDirectoryItem**. Similarly, because directories can be contained within directories, the **Directory** class implements **IDirectoryItem**.

There is one relationship type: the **Containment** relationship type.

There are two collection types associated with the **Containment** relationship:

- Collections that conform to the *items-of-directory* collection type are origin collections for **Containment** relationships. The **IDirectory** interface exposes this collection.

- Collections that conform to the *directory-of-item* collection type are destination collections for **Containment** relationships. The **IDirectoryItem** interface exposes this collection.

The **IFile** interface exposes one property: the **Size** property.

The **IDirectoryItem** interface exposes one property: the **ModificationDate** property.

The **IDirectory** interface exposes one property: the **ChildCount** property.

In this example, the information model exposes no methods through any of its interfaces.

See Also

Repository Object Architecture

Understanding the RTIM Through Examples

Meta Data Services Programming

Designing Information Models

When you design a software tool, you must articulate the kinds of data that the tool will manipulate. You can store the definitions of these kinds of data, called types, in the repository by creating an information model. Each information model is, in effect, an object model represented in the repository as data.

This section uses the example of a bookseller's database to introduce information models and it describes how the repository engine can represent them as data. You can use this example as a way to understand how to design an information model.

Topic	Description
Understanding Application Data	Describes how to formulate application structures based on application data
Visualizing Data and Meta Data	Describes techniques you can use to understand application structures
Depicting Relationships Between Objects	Describes how to identify relationships
How Relationships Conform to Relationship Types	Describes how relationships conform to relationship types
Understanding Collections	Discusses collection types and how they relate
Understanding Relationship Roles	Discusses the distinctions in relationship roles and how those distinctions determine relationship collections

See Also

Repository Object Architecture

Understanding the RTIM Through Examples

Understanding Application Data

You can begin planning your information model by answering these questions:

- What kinds of objects will the tool store? That is, what are the classes to which the tool's objects must conform?

- What kinds of relationships will the tool store? That is, what are the relationship types that describe how objects can be related?

- What properties apply to the objects of each class or the relationships of each relationship type?

You can think of any application structure as objects, properties, and relationships. When you store data about your tool or application in a repository, you can create objects, indicate how those objects are related to each other, and define properties for each of those objects or relationships. To create the hypothetical bookseller's tool, you can do the following:

- Create objects such as:
    - **Book**, to store instance data like *Moby Dick* (a book) and *Inside OLE* (a book)

    - **Publisher**, to store instance data like Microsoft Press® (a publisher)

    - **Person**, to store instance data like Kraig Brockschmidt (a person) or Herman Melville (a person)
- Indicate how those objects are related:
    - Herman Melville (a person) wrote *Moby Dick* (a book). Kraig Brockschmidt (a person) wrote *Inside OLE* (a book). These relationships are the same and can be described as **Authorship**.

        - Microsoft Press (a publisher) published *Inside OLE* (a book). This relationship can be described as **Publication**.

- Decide which properties you need to capture additional information for each object:
    - **Birthday** is a property that can describe a person. (The birthday of Herman Melville is November 12, 1819.)
    - **Address** is a property that can describe a publisher. (The address of Microsoft Press is One Microsoft Way.)
- You can also decide which properties you need for relationships:
    - **Year of Publication** is a property that can describe the Publication relationship. (The year of publication for *Inside OLE* is 1995.)

The following figure summarizes this data. The figure shows typical data about specific books, authors, and publishers. Because the data is typical, it helps you visualize the kinds of data that your model must accommodate.

See Also

Designing Information Models

Visualizing Data and Meta Data

This section presents tabular and graphic techniques for visualizing data.

After you identify the objects, property values, and relationships in your tool or application, you can use tabular and graphic techniques to visualize your data. These powerful techniques can help you understand the types of data you need. The following topics describe how to visualize data and meta data.

Topic	Description
Ways to List Data	Describes visualization techniques for understanding tool and application data
Ways to List Meta Data	Describes visualization techniques for transforming your understanding of tool and application data into an information model design

See Also

Designing Information Models

Ways to List Data

The following table lists data about books, people, and publishers. The first column (**Kind of Data**) provides labels for groups of data: books, people, and publishers. The actual data appears in the table's second column (**Data**).

Expressed graphically, the data in the table is shown in the following figure.

The following table expands the preceding table to include relationships. Again, the table uses a convenient grouping of the data. The first column labels each group.

The labels in the left column (**Kind of Data**) are one example of how the object model can store the bookseller's data. The labels identify three classes (**Book**, **Person**, and **Publisher**) and two relationship types (**Authorship** and **Publication**). Because it is a list, you can think of the entries in the **Kind of Data** column as data. Because it is data, you can create another table in which this information appears in the **Data** column.

In the **Data** column, each entry describes exactly one thing, either an object or a relationship. Each entry in the **Data** column describes a particular book, author, publisher, authorship, or publication.

The following figure contains arrows to show the relationships in the preceding table.

See Also

Designing Information Models

Ways to List Meta Data

The repository engine stores meta data as data. This section uses data visualization techniques to demonstrate how to model meta data.

The following table lists and organizes the types of the bookseller's object model. The left column contains convenient groupings of like information, and the right column (**Data**) contains the information itself. The **Data** column describes particular classes (such as **Book** and **Person**) and particular relationship types (such as **Authorship** and **Publication**). The **Kind of Data** column thus reveals a portion of the object model for storing classes and relationship types.

The following table enlarges the preceding table to include properties.

In the preceding table, the information in the **Data** column is equivalent to the information in the following figure. The following figure shows typical data about a typical object model, the bookseller's object model.

See Also

Designing Information Models

Depicting Relationships Between Objects

To depict relationships between objects, use arrows as shown in the following figure.

Diagrams of meta data use these standard conventions:

- Show objects as dots.
- Show relationships as arrows.
- Show kinds of objects as labeled rectangles.

The tabular equivalent of this graphical presentation of data is shown in the following table.

The labels in the **Kind of Data** column constitute a portion of an object model for storing object models. The object model for storing object models is called the Repository Type Information Model (RTIM).

Note Each entry in the **Data** column describes only one thing: either an object or a relationship. The 23 entries in the **Data** column correspond to the 23 dots and arrows in the preceding figure.

See Also

Designing Information Models

How Relationships Conform to Relationship Types

When you store a relationship, the meaning of what you store answers three questions:

- Which two objects are related to each other?

  For example, when you store the relationship indicating that Herman Melville wrote *Moby Dick*, you relate the object describing Herman Melville and the object describing *Moby Dick*.

- How are the two objects related?

  For example, when you store the relationship indicating that Herman Melville wrote *Moby Dick*, you indicate that Melville wrote the book, not that he reads it or criticizes it. You indicate that Melville wrote the book by creating a relationship that conforms to the **Authorship** relationship type.

- What role does each object play in the relationship?

  For example, when you store the relationship indicating that Herman Melville wrote *Moby Dick*, you indicate that Melville wrote *Moby Dick*, not that *Moby Dick* wrote Melville. The object representing Melville plays the role of the writer and the object representing *Moby Dick* plays the role of the thing that was written.

The following figures evaluate whether potential relationships conform to the two relationship types: **Authorship** (of book by person) and **Publication** (of book by publisher).

Potential relationship

The following diagram shows the potential relationship based on relationship type.

Does the relationship conform?

Microsoft Press® publishes *Inside OLE*: Yes, the relationship conforms to the **Publication** relationship type.

Potential relationship

The following diagram shows a potential relationship that does not conform to relationship type.

Does the relationship conform?

Kraig Brockschmidt publishes *Inside OLE*: No, the relationship does not conform to either relationship type. The **Publication** relationship type allows you to save a relationship indicating that a publisher publishes a book. This data indicates that a person publishes a book.

Potential relationship

The following diagram shows the potential relationship based on relationship type.

Does the relationship conform?

Kraig Brockschmidt wrote *Inside OLE*: Yes, the relationship conforms to the **Authorship** relationship type.

Potential relationship

The following diagram shows a potential relationship that does not conform to relationship type.

Does the relationship conform?

Inside OLE publishes Microsoft Press: No, the relationship does not conform to

either relationship type. Although this relationship uses two objects of the correct type, it does not conform because it places those objects in the wrong roles.

Potential relationship

The following diagram shows the potential relationship based on relationship type.

Does the relationship conform?

Microsoft Press publishes *Moby Dick*: Yes, the relationship conforms to the **Publication** relationship type. The relationship conforms, even though the data is inaccurate. (Microsoft Press does not publish *Moby Dick*.)

See Also

Designing Information Models

Understanding Collections

You can read any relationship in two directions. For example, you can say Herman Melville wrote *Moby Dick* or *Moby Dick* was written by Herman Melville. You can paraphrase each of these two statements as follows:

- Herman Melville is in the set of persons who wrote *Moby Dick*.

- *Moby Dick* is in the set of books written by Herman Melville.

Although awkward, this way of articulating relationships highlights the existence of collections. The following two figures show various collections.

- The collection of books written by Herman Melville:

- The collection of persons who wrote *Moby Dick*:

You can think of collections as collections of objects or as collections of relationships, each with a source and a target object. The following figures show the ways to think of collections.

- The collection of books written by Herman Melville:

The figure to the left shows the collection of books written by Herman Melville as an object collection, while the figure to the right shows the same collection as a relationship collection.

- The collection of authors of *Moby Dick*:

The figure on the left shows the collection of authors of *Moby Dick* as an object collection, while the figure to the right shows the same collection as a relationship collection.

The preceding figures make clear that object collections and relationship collections are fundamentally equivalent. They both accommodate the same data. However, when you manipulate a relationship collection from a COM program, you can manipulate it either with an interface called **ITargetObjectCol** or with an interface called **IRelationshipCol**. The first interface lets you manipulate a collection as if it contains objects. The second interface lets you manipulate a collection as if it contains relationships. In Automation, if you do not specify an interface, you implicitly manipulate relationships as object collections because the **RelationshipCol** class implements **ITargetObjectCol** as its default interface.

See Also

Defining a Target Object Collection

Designing Information Models

Understanding Relationship Roles

Each relationship belongs to two relationship collections, one that describes the relationship from the perspective of the origin, and another that describes it from the perspective of the destination.

For example, the relationship *Herman Melville wrote Billy Budd* is a member of two different collections:

- The set of books written by Herman Melville; or, expressed in terms of a relationship collection, the set of authorships for which Herman Melville is the writer

- The set of authors of *Billy Budd*; or, expressed in terms of a relationship collection, the set of authorships for which *Billy Budd* is the written thing

There is a relationship between collection type and relationship type. The following figure shows some relationship types and their attendant collection types.

In the figure, each relationship type has exactly two collection types. The following are true for every relationship:

- Each relationship is a member of two relationship collections.

- Each relationship relates two objects, an origin object and a destination object.

- You can read each relationship in two directions.

- In any relationship, the related objects participate in two separate roles. For example, the roles in the relationship *Kraig Brockschmidt wrote*

Inside OLE, are:

- The role of writing thing, filled by the object describing Kraig Brockschmidt.

- The role of written thing, filled by the object describing *Inside OLE*.

The two roles correspond to the two collection types.

See Also

Designing Information Models

Meta Data Services Programming

Getting Started with Meta Data Services

This section provides information that prepares you for programming Microsoft® SQL Server™ 2000 Meta Data Services applications. You can learn about programming environment requirements, and how to get started with information model definition and programming. For more information about upgrading repository components from previous releases, see Upgrading from Earlier Versions.

The following topics can help you get started.

Topic	Description
Programming Environment	Describes the requirements of your programming environment.
Accessing Automation Object Members	Explains how to access a nondefault member on an Automation object.
Visual C++ Wrappers with Meta Data Services	Explains how to generate and use wrappers on a COM interface.
Using Meta Data Services to Define Information Models	Explains information model definition in Meta Data Services. It also explains how information models enable subsequent application development.
Using Meta Data Services to Program Information Models	Provides basic information for programmers, providing a big picture overview of what programming an information model entails.

See Also

Repository API Reference

Repository Object Architecture

What's New in Meta Data Services

Meta Data Services Programming

Programming Environment

Programming a Microsoft® SQL Server™ 2000 Meta Data Services application requires software and operating systems. Required software works together in an integrated manner. For this reason, the software that you use to build a Meta Data Services application must be installed on the same PC.

The Automation server distributed with Meta Data Services is Repodbc.dll. If you require more server functionality than Repodbc.dll provides, you can create your own Automation server. For more information, see Choosing an Automation Server for a Class.

Additional programming resources are provided through the Meta Data Services Software Development Kit (SDK). The Meta Data Services SDK provides tools that complete your repository environment. Whether you are using COM or Automation interfaces to define or manipulate an information model, be sure to download the Meta Data Services SDK so that you can take advantage of the additional utilities and documentation that it provides.

The following software details the required and optional software you need.

Software	Description
Microsoft Windows® operating system	You can use Windows 98, Windows NT® 4.0, or Windows 2000.
SQL Server or Microsoft Jet, and ODBC	You can use SQL Server 6.5, SQL Server 7.0, and SQL Server 2000, or Microsoft Jet 3.5 and later. You also need ODBC 2.0 or later. A DBMS is required to manage the repository database. For more information, see Repository Databases. The DBMS you use can affect the performance of a repository database and the availability of some features. For more information, see Using Repository Engine Features with Older Databases.
Meta Data Services	Meta Data Services installs with SQL Server 2000.

	Meta Data Services provides the repository engine. You can also obtain Meta Data Services from the Microsoft Repository web site. To install from the Web, a licensed copy of SQL Server 6.5, SQL Server 7.0, SQL Server 2000, or Microsoft Visual Studio® 6.0 must already be installed on your PC.
Modeling tool	(Optional.) A modeling tool is strongly recommended. Rational Rose is the preferred modeling tool for use with this release of Meta Data Services.
The Meta Data Services SDK	(Optional.) The Meta Data Services SDK contains programming and modeling resources. You can obtain the Meta Data Services SDK from the Meta Data Services web site. For more information, see Meta Data Services SDK.
Development tool	(Optional.) COM support is a programming requirement. You can use Microsoft Visual Studio or another development tool that supports COM Automation development.

See Also

Accessing Automation Object Members

Automation Reference

COM Reference

Meta Data Services SDK

Specifications and Limits

Visual C++ Wrappers with Meta Data Services

Accessing Automation Object Members

The repository API exposes a number of Automation objects that support multiple interfaces. For each Automation object, one interface is defined to be the default interface, and the members (the properties, methods, and collections) that are attached to that interface are accessible through the standard Microsoft® Visual Basic® mechanisms.

When accessing members that are attached to an interface that is not the default interface for an Automation object, a different access technique must be used. An additional reference to the object must be declared that explicitly calls for the nondefault interface. The nondefault interface members can then be accessed through the new object reference.

The following example illustrates how to access a property that is attached to an interface that is not the default interface for an Automation object. In this example, the connection string that is used to connect to the repository database is retrieved. Repository objects implement the **IRepositoryODBC** interface; this interface is not the default interface. The **ConnectionString** property is attached to the **IRepositoryODBC** interface. The **ConnectionString** property is the ODBC connection string that Microsoft SQL Server™ 2000 Meta Data Services uses when connecting to a database server.

```
Dim myRepos As Repository
Dim nonDefIfc As IRepositoryODBC
'  Initialize myRepos by opening a connection to a repository database.
Set nonDefIfc = myRepos
connect$ = nonDefIfc.ConnectionString
```

In this example, the **nonDefIfc** object does not use additional resources; rather, it is an alternate view of the **myRepos** object.

See Also

Automation Reference

Repository ConnectionString Property

Repository Object

Visual C++ Wrappers with Meta Data Services

The repository API is based on dispatch interfaces. This means that all properties are manipulated through the **Invoke** method that the **IDdispatch** interface exposes. Using dispatch interfaces from programming languages that are v-table based, such as Microsoft® Visual C++®, can be cumbersome.

Visual C++ version 6.0 provides support for using dispatch interfaces in an easier way than before. It does this through the **#import** directive. The **#import** directive instructs the Visual C++ compiler to read the type library given as a parameter to the directive, and to create v-table based wrappers for the type library. The compiler does this on the fly, and it also updates the wrappers if the type library is updated.

The compiler generates the following two header files with the same name as the type library:

- A .tlh header file that contains definitions of all interfaces and identifiers.

- A .tli header file that contains inline wrapper functions, which convert properties from their respective data types to the variant data type that the **Invoke** method expects. The .tli file is automatically included inside the .tlh file.

Generating the Wrappers

In order to make use of the dispatch support in Visual C++, add the following statement at the top of one of the .cpp files:

```
#include <atlbase.h>
// Required for smart pointer support
#import "rtim.tlb" named_guids
// The following using-directive allows other type libraries to
// reference repository engine objects:
```

```
using namespace RepositoryTypeLib;
#import "uml.tlb"  named_guids
using namespace UML;
```

The Atlbase.h header file is required to support smart pointers. The next two lines instruct the compiler to generate wrapper classes for the main interfaces defined by the repository engine. The compiler automatically wraps type libraries into namespaces that have the same name as the type library. This is done to limit the possibility of name clashes between type libraries. Unfortunately, the wrapper generator does not support references between type libraries. Therefore, the **using namespace** directive is required to automatically map the repository engine interfaces into the default namespace.

After the compiler generates wrappers for the repository engine interfaces, you can use the mechanism mentioned previously to import any required type library. Make sure that the type libraries are imported in a correct dependency order.

When the wrapper is generated, the compiler creates the following two functions for each interface member (such as property or collection):

- **Get***memberName*
- **Put***memberName*

where *memberName* is replaced by the member name.

For example, the **Visibility** method on the **IUMLModelElement** interface (**IUMLModelElement.Visibility**) will be wrapped into the following methods:

- **GetVisibility()**
- **PutVisibility()**

Using the Wrappers

After the compiler generates wrappers for dispatch-based interfaces, smart pointer templates can be used to manipulate these objects. To define a smart pointer for an interface, use a declaration similar to the following:

```
CComPtr<IRepository> pRep;
```

This defines a smart pointer for the **IRepository** interface. To instantiate a repository and assign it to the smart pointer, use the **CoCreateInstance** method of the smart pointer, as shown here:

```
hr = pRep.CoCreateInstance(CLSID_Repository,NULL);
```

After instantiating the repository, it is possible to use methods defined on the **IRepository** interface to open a repository database as follows:

```
CComPtr<IRepositoryObject> pRootRO;
pRootRO = pRep->Open("C:\\test.mdb","","",0);
```

The methods defined on the dispatch interface are accessed using the **->** operator, while helper functions such as **CoCreateInstance** are accessed using the dot (.) operator.

After opening a repository database, it is possible to use the wrappers and the smart pointers to access any object in the repository. For example:

```
CComPtr<IUmlPackage> pPackage;
CComPtr<IRepositoryObject> pRO;
hr = pRootRO.QueryInteface(&pPackage);
for (long n=1;n<pPackage->GetElements()->GetCount();n++)
{

pRO = pPackage->GetElements()->GetItem(n);   // Get the element # n
// Use the element pRO
```

See Also

COM Reference

Meta Data Services Programming

Using Meta Data Services to Define Information Models

Information models define the meta data types that you can store and subsequently manipulate in and from another tool or application. The information model that you create and install determines the physical storage in a repository database.

The information model is a meta model, and it defines the meta data types that programmers can use and otherwise manipulate. The information model that is recommended for use with Microsoft® SQL Server™ 2000 Meta Data Services is the Open Information Model (OIM). This model is recommended because it contains generic meta data that is supported by a variety of third-party vendors, providing instant integration with tools and platforms that you may already be using in your development environment. Although this model is predefined, it can be extended to accommodate meta data that you require.

Typically, you define an information model using a modeling tool. However, you can also create an information model programmatically using the repository API and the COM or Automation interfaces it exposes.

Information Model as a Framework

You can think of an information model as a framework or structure for storing meta data definitions. For example, suppose you want to create design data that programmers can subsequently use to create Microsoft Visual Basic®, Microsoft Visual C++®, and Microsoft Visual J++® applications. In your information model, you define the basic elements of your application once by specifying the objects, defining relationships that associate the objects, and setting properties. Programmers can then use your model definitions in each development environment to program the implementation strategy that each language requires. Using a single information model provides a way to use the same design for multiple implementations.

The following topics provide model designers with information needed to build and deploy an information model.

Topic	Description
Repository Object Architecture	Explains the object architecture that exposes repository engine functionality and the information model objects that the engine can manipulate. This topic includes examples that can help you understand information model definition.
Defining Information Models	Provides detailed information about alternate ways of creating an information model and defining elements of an information model.
Installing Information Models	Explains how to install an information model into a repository database. Installing an information model makes the information model available for programming.

See Also

Repository API Reference

Using Meta Data Services to Program Information Models

Meta Data Services Programming

Using Meta Data Services to Program Information Models

You can program against an information model that is installed in a repository database. Programming against an information model adds, updates, removes, and retrieves data from a repository database.

Typically, the data that you add and otherwise manipulate is design data about a tool or application that you create. Furthermore, the data that you can add and manipulate is defined by the information model. You can think of the information model as a template to which the data you add must conform. For example, to create an application that manipulates a schema, tables, and columns, you need an information model that defines what a schema is, what a table is, and what a column is.

As a programmer who is coding such an application, you populate the schema, table, and column types with meta data instances to be used by the tools and applications you create. The following example provides a simplistic look at how you can program elements of a database application using the Open Information Model (OIM).

What OIM defines	What you create
Schema	A schema for a Microsoft® SQL Server™ database, a Microsoft Jet database, or a new version of each database. In this case, four instances of **Schema** are stored in your information model.
Tables	Tables for **Customers**, **Orders**, and **Products**. For example, you can vary the table definitions based on the schema types, or you can reuse the tables for each schema. Creating separate tables for each schema results in twelve instances of tables in your information model.
Columns	Columns for **Customer**, **Order**, and **Product** tables. Assuming no reuse strategy, you can have a separate column instance for each table and for each schema.

Notice that the instance data you store is all about definitions. Instead of storing "Joe Smith" customer name, you store data about the **CustomerName** column.

Meta data is, by definition, unbiased. The following suggestions describe different ways to reuse meta data.

- Use the meta data objects in two development environments (Microsoft Visual C++® for a desktop application and Microsoft Visual J++® for a Web application), using the syntax of each language to call the same object. For more information about declaring objects, see Programming Fundamentals: Declaring Objects.

- Use the meta data objects in development projects in the same environment (one project for an application you are maintaining for an existing customer, one project for new development). You can use versioned objects and workspaces to isolate changes.

- Export the meta data as Extensible Markup Language (XML) to a different repository.

See Also

Information Models

OIM in Meta Data Services

Repository API Reference

Programming Fundamentals: Declaring Objects

When you program, you instantiate repository objects. Repository objects are COM objects that the repository engine creates on the fly using the type information and object instance data provided in your information model.

The repository object architecture divides objects into engine objects and information model objects. Programming against an information model typically requires that you invoke repository objects that are described by the repository engine object model. In contrast, when a model designer creates the type information, he or she typically uses Repository Type Information Model (RTIM) objects.

Declaring SpellChecker as RTIM Objects

For example, before you can use the following **SpellChecker** structure in your application code, the following declarations for **SpellChecker** must be predefined in your information model in some way that is compatible with the repository API. The following code example shows a hypothetical information model, **MyTypeLib**, and shows some additional definitions for **SpellChecker** that you can work with:

```
DIM oTypeLib as ReposTypeLib
DIM oCSpellChecker as ClassDef
DIM oISpellChecker as InterfaceDef
DIM oPLanguage as PropertyDef
Set oCSpellChecker = oTypeLib.CreateClassDef(CSC_objid, CSpellC
Set oISpellChecker = oCSpellChecker.CreateInterfaceDef(ISC_objid,
Set oPLanguage = oISpellChecker.CreatePropertyDef(PLang_objid, P
```

Declaring SpellChecker in Application Code

At a minimum, to retrieve meta data in your application code, you invoke a repository object that represents a repository session, another repository object that represents the repository type library containing your information model

definitions, and additional repository objects that represent specific meta data instances.

Typically, to support versioning, you should use **RepositoryObjectVersion**. However, you can also use **RepositoryObject** as an alternative.

```
DIM oTypeLib as RepositoryObjectVersion
DIM oCSpellChecker as RepositoryObjectVersion
Dim oISpellChecker As ISpellChecker
oSpellChecker(oISpellChecker).Properties("Language")=French
```

See Also

Programming Fundamentals: Populating a Collection

Using Meta Data Services to Program Information Models

Programming Fundamentals: Populating a Collection

Collections provide navigation and a way to handle a set of objects as a unit. When programming against an information model, you write code that materializes a collection so that you can access and otherwise manipulate its objects at run time.

The following example provides a simple illustration for adding objects to a collection. Suppose your information model contains a **Schema** object that has a collection of **Tables** attached to it. You can populate the **Tables** collection by writing code that adds specific instances (such as a **Customer** table and an **Order** table) to the collection.

You can populate the **Tables** collection with specific table instances using code like the following. Note that the relationships you can create are possible because the information model already contains definitions for collections.

```
Dim oSchema As RepositoryObject
Dim oCTable As RepositoryObject
Dim oISchema As ISchema
Set oSchema=oRepos.GetObject(ObjID_oSchema)
Set oTable=oRepos.GetObject(ObjID_oTable)
Set oISchema=oSchema
oISchema.Tables.Add(table)
```

See Also

[Programming Fundamentals: Declaring Objects]

[Using Meta Data Services to Program Information Models]

Meta Data Services Programming

Connecting to and Configuring a Repository

The repository engine can access repository databases that are managed by either Microsoft® Jet, Microsoft SQL Server™, or SQL Server Runtime Engine.

The repository engine accesses a database through an ODBC driver (version 2.0 or later). You must have ODBC installed on the server hosting the database and on the client from which you are accessing the repository engine.

The ODBC connection string that is used to specify the location of the repository database varies, depending upon which database server is managing the repository database. The ODBC connection string contains *keyword=keyValue* pairs, separated by semicolons. If you do not specify a connection string, the repository engine creates a default repository database.

Before you can connect to a database, you must first instantiate a repository session. After you create a repository instance, you can open an existing database or create a new database. Note that how a database is created varies depending on the DBMS you use.

The following table lists topics that tell you more about database connections and configuration.

Topic	Description
Connecting to a SQL Server Repository Database	Describes how to open or create a SQL Server database connection
Connecting to a Jet Repository Database	Describes how to open or create a Jet database connection
Connecting Through a DSN	Describes how to connect to a repository database through a data source name (DSN)
Default Repository Databases	Explains how the repository engine resolves an unspecified connection string by creating a default database
Replicating Repository Databases	Describes replication behavior for SQL Server repository databases

See Also

IRepository::Create

Repository Create Method

Repository Databases

Storage Strategy in a Repository Database

Upgrading and Migrating a Repository Database

Using Repository Engine Features with Older Databases

Meta Data Services Programming

Connecting to a SQL Server Repository Database

Microsoft® SQL Server™ 2000 is the DBMS recommended for repository databases. Using a SQL Server database yields maximum performance from the repository engine and provides a layer of security that is otherwise unavailable. If you do not own a licensed copy of SQL Server, you can use the SQL Server Runtime Engine that is freely distributed by Microsoft. The SQL Server Runtime Engine can be used to create or open a SQL Server repository database.

When you use a SQL Server repository database, you must either use (that is, open) an existing repository database, or create an empty database. The repository engine cannot automatically create a SQL Server database for you. To the repository engine, creating a new SQL Server database means populating an empty database with the repository SQL tables it needs to store and manage repository data. If you already have a repository database (that is, a database that contains repository SQL tables), you can connect to it through an open statement.

When you create a new, empty SQL Server database, be sure to specify which users can access the database. You must also create the necessary login and user accounts for people who will be accessing the database, and you must assign the appropriate permissions to these accounts. If you want to grant full permissions to everyone, you can use this SQL command to set database access permissions:

```
GRANT ALL TO PUBLIC
```

Creating a New Database

To create a new repository database, use the following syntax. Notice that the first statement creates a repository session. In Microsoft Visual Basic®, be sure to reference Repodbc.dll so that it is available to your program. By default, Repodbc.dll is located in C:\Program Files\Common Files\Microsoft Shared\Repostry.

Use the following code to create a new database in Microsoft Visual C++®:

```
CoCreateInstance(CLSID_Repository, NULL, CLSCTX_INPROC_SE
m_pIRepos->Create(CCOMVariant(SERVER="MyServer";DATABAS
```

Use the following code to create a new database using Visual Basic:

```
DIM oRepos as New Repository
oRepos.Create "SERVER=MyServer;DATABASE=MyDatabase;UID=
```

Note Invoking the **Create** method on an existing repository database simply opens it.

Opening an Existing Database

To connect to an existing SQL Server repository database such as **msdb**, use the SERVER keyword to specify the SQL Server name and the database name. If the database name is not specified, the default database for the user who is opening the database is used. You can also use a data source name (DSN) to connect to a database.

```
CoCreateInstance(CLSID_Repository, NULL, CLSCTX_INPROC_SE
m_pIRepos->Open(CCOMVariant(SERVER="MyServer";DATABASE
```

Administering a SQL Server Database

You can use the utilities and tools that come with SQL Server to administer the repository database (at the database level). For example, if your repository database is damaged due to a power outage or system failure, you should use the recovery tools that are provided with SQL Server to repair the damage. Similarly, if your repository database requires periodic defragmentation, you should use the defragmentation tools that are provided with SQL Server.

CAUTION SQL Server and its components store private meta data in the **msdb** database. While you are encouraged to use and add to existing data, be aware that modifying or deleting it can cause unexpected results. If you introduce a modification that breaks the functionality of SQL Server or its components, you must reinstall the software.

See Also

Connecting Through a DSN

Default Repository Databases

IRepository::Create

IRepository::Open

Repository Create Method

Repository Open Method

Repository SQL Tables

Storage Strategy in a Repository Database

Meta Data Services Programming

Connecting to a Jet Repository Database

If you choose to use a Microsoft® Jet database, you can create it programmatically using the **IRepository Create** method. If you do not specify a complete path, the repository engine uses the default path. For more information, see Default Repository Databases.

You can create a new database using the syntax provided in the following example. Notice that the first statement creates a repository session.

Use the following code to connect to a Jet database in Microsoft Visual C++®:

```
CoCreateInstance(CLSID_Repository, NULL, CLSCTX_INPROC_SE
m_pIRepos->Create(CCOMVariant(DBQ="MyDB.mdb"), CCOMVari
```

Use the following code to connect to a Jet database in Microsoft Visual Basic®:

```
DIM m_pIRepos as New Repository
m_pIRepos.Create(DBQ="MyDB.mdb")
```

To connect to a Jet repository database, use the DBQ keyword to specify the path to the database file. The DBQ keyword must be the first keyword in the connection string, if it is present. If the DBQ keyword is not present, the connection string is assumed to contain only a database path specification. In this case, the repository will add the DBQ keyword to the front of the ODBC connection string before passing it on to the database server. If the Jet database file specified by the DBQ keyword does not exist, the repository engine will create it.

```
CoCreateInstance(CLSID_Repository, NULL, CLSCTX_INPROC_SE
m_pIRepos->Open(CCOMVariant(DBQ="MyDB.mdb"), CCOMVaria
```

See Also

Connecting to a SQL Server Repository Database

Default Repository Databases

Meta Data Services Programming

Connecting Through a DSN

You can use the DSN keyword to specify a data source name (DSN) to connect to an existing Microsoft® Jet or Microsoft SQL Server™ repository database. The DSN keyword specifies a data source name that has been configured using the ODBC Data Source Administrator.

If you are connecting to a SQL Server database, you must explicitly specify the user ID and password in the connection string, even if the values are part of the ODBC registration.

You can connect to a database using the syntax provided in the following example. Notice that the first statement creates a repository session.

```
CoCreateInstance(CLSID_Repository, NULL, CLSCTX_INPROC_SE
m_pIRepos->Open(CCOMVariant(DSN="MyDataSourceName";UID=
```

See Also

Connecting to a Jet Repository Database

Connecting to a SQL Server Repository Database

Default Repository Databases

Meta Data Services Programming

Default Repository Databases

If you do not specify the repository database explicitly, a connection will be established to the default repository database. This database is managed by Microsoft® Jet. Its location is determined by the default value of the **Current Location** registry key.

If you are using the **Create** method and an unspecified connection string, and if the default database does not exist, the repository engine creates the database. If you are using the **Connection** method (or the **Create** method on an existing database) and an unspecified connection string, the repository engine looks for the database at the default location.

The location of the default repository database is stored in the system registry in this registry key:

HKEY_LOCAL_MACHINE\Software\Microsoft\Repository\Current

This registry key must contain a DBQ keyword-value pair, SERVER keyword-value pair, DSN keyword-value pair, or just the path to a Jet repository database. The default value for this registry key is:

windowsDirectory\MsApps\Repostry\Repostry.mdb

Replace windowsDirectory with the path specification for the directory that contains the Microsoft Windows installation. Unless you change this registry key value after installing Microsoft SQL Server™ 2000 Meta Data Services, your default database server is Jet.

See Also

Connecting Through a DSN

Connecting to a Jet Repository Database

Connecting to a SQL Server Repository Database

Meta Data Services Programming

Replicating Repository Databases

Microsoft® SQL Server™ 2000 repository databases can take advantage of the replication features provided by SQL Server to publish a repository to other subscriber repositories.

You can use either transactional or snapshot replication to replicate a repository database. If you implement transactional replication, you can choose whether to support synchronization. Synchronization automatically updates your subscriber databases so that they contain the same content as the publisher. For more information, see Replication Overview.

Replication Requirements for Repository Databases

For repository databases, additional steps beyond those required by SQL Server should be followed to ensure successful replication.

Publishing a Repository Database

- Install information models into a repository database. Before you begin replication, you must install information models into the repository database that you want to publish, and then allow replication to propagate the content across all subscriber databases.

  Installing new or revised information models after replication is enabled can produce unexpected results. In this case, new tables that are associated with new or revised information models are not automatically enabled for replication. If you are updating an existing publisher with newer models, you must manually select the additional tables as articles so that updates to those tables will propagate to subscriber databases.

  Note that you cannot publish **msdb**, the default repository database in SQL Server. You must create an alternate repository database to enable replication.

- Verify that all repository SQL tables and information model tables are selected as articles in the publication. Repository SQL table names have

an **rtbl** prefix. Information model table names are typically prefixed with the name of the model (for example, UML, UMX, GEN, and so on).

You cannot publish a subset of the tables in a repository database. A repository database stores type information in multiple tables. If you omit some tables from the publication, you may not get a complete definition for each repository object.

- Verify that repository stored procedures are not included in your publication. Repository stored procedures are part of the publisher database, but cannot be part of a subscriber database. Repository stored procedure names have an **r_iRtbl** prefix.

  Stored procedures are used by the repository engine to install and update information models in a SQL Server repository database. Replicating a stored procedure can result in an attempt to reinstall an information model that is already installed on a subscriber database.

- Avoid creating data filters or enabling autonomous subscriptions.

Defining Subscriber Databases

After you create a publication, you can create one or more push subscriptions that propagate repository data from the publisher.

Avoid updating subscriber objects from any nonpublisher source. Only the publisher should be allowed to update subscriber objects.

Repository subscriber databases must be read-only. Furthermore, each subscriber can receive content from only one publisher. Repository databases use internal identifiers to store and manipulate meta data. While internal identifiers are unique within a specific repository, they may not be unique across multiple repositories. To avoid duplicate internal identifiers, you must require that each subscriber is read-only and receives all of its updates from a single publisher. To do this, specify that a publication for the publisher database has all of the repository tables as articles, then add read-only repository databases as subscribers.

See Also

Meta Data Services Programming

Defining Information Models

The part of a Microsoft® SQL Server™ 2000 Meta Data Services repository that stores type information is defined by the information models you create and install.

The following topics explain how to create and specify the parts of an information model. For more information about creating and populating a repository database, see Connecting to and Configuring a Repository.

Topic	Description
Repository Identifiers	Describes identifiers that are used to retrieve and manage repository objects
Naming Objects, Collections, and Relationships	Describes naming conventions, name reuse, aliasing, and ways names are created by the repository engine
Creating and Extending Type Information	Describes alternate approaches for creating and extending information models
Defining Inheritance	Explains how inheritance works and how you can implement it for your interfaces
Defining Relationships and Collections	Explains how to define general-purpose and special-purpose relationships and collections
Defining Properties	Explains how to define property definition objects
Defining Methods	Explains how to define methods, parameters, and scripted objects
Generating Views	Describes how to generate SQL views that correspond to your information model

See Also

Information Models

Installing Information Models

Meta Data Services Programming

Repository Identifiers

The repository engine uses identifiers to distinguish objects and object versions from each other.

There is an object identifier for every object in a repository database. When you add an object to a repository (programmatically or by installing a model), the object identifier is created as part of the object definition. This identifier remains with the object until you delete the object from a repository. When you program a repository object, you can use the object identifier to retrieve the object you want.

The repository engine maintains two sets of identifiers: object identifiers (ObjID) and internal identifiers (IntID). One set, the object identifiers, is public. The second set, the internal identifiers, is used by the repository engine. A repository SQL table maps the two sets, and the repository engine maintains the correspondence.

Functionally, object identifiers and internal identifiers are similar. However, the values of internal identifiers are smaller and more efficient for the engine to handle and the database to store. When the repository engine receives a call for an object identifier, it converts the internal identifier into an object identifier that your program can use.

In some cases it is desirable to use the internal identifiers. For example, if you want to query the database directly, you can use the smaller internal identifier that the repository engine uses to store object data. However, when you program with repository objects, you should always use the longer object identifier.

The following table lists topics that provide more information about repository identifiers.

Topic	Description
Object Identifiers and Internal Identifiers	Compares object identifiers and internal identifiers, and provides details about their composition.
Object-Version Identifiers and Internal Object-Version Identifiers	Describes the portion of a repository identifier that stores version

	information, and compares how version identifiers are represented in object identifiers and internal identifiers.
How Repository Identifiers and Stored and Instantiated	Details how internal identifiers are stored and how object identifiers are created from internal identifiers.
Repository Identifier Data Structures	Explains the data structure of repository identifiers. Knowing about internal identifier data structures can help you build a query.
Assigning Object Identifiers	Explains how to assign object identifiers.

See Also

Navigating a Repository

Object Identifiers and Internal Identifiers

Each **RepositoryObject** instance has two identifiers: an object identifier and an internal identifier.

An object identifier is global in scope. It uniquely distinguishes a repository object from all other repository objects represented in all other repository databases.

Internal identifiers correspond to object identifiers, except that internal identifiers are used by the engine.

Both object identifiers and internal identifiers are explained in this topic. Another kind of repository identifier is used for a **RepositoryObjectVersion** instance. For more information, see Object-Version Identifiers and Internal Object-Version Identifiers.

About Object Identifiers

Object identifiers have the following format.

The first 16 bytes of each object identifier constitute a globally unique identifier (or GUID). The next 4 bytes constitute a local identifier.

RepositoryObjects do not include version information. When you are working with **RepositoryObject** instances, the repository engine follows a resolution strategy to select a specific version of a **RepositoryObject** instance. The resolution strategy, not the version indicator, determines which object is selected.

About Internal Identifiers

Each **RepositoryObject** instance also has an internal identifier that distinguishes it from every other object within the same repository database. The internal identifier is used by the repository engine to manipulate the object specified by the object identifier. The internal identifier is an 8-byte quantity of the following form.

The first 4 bytes constitute a site identifier (site ID). For more information about site IDs, see How Repository Identifiers are Stored and Instantiated.

The last 4 bytes constitute the local identifier (local ID). For a **RepositoryObject** instance, the local identifier portion of the internal identifier and the object identifier is the same. That is, each repository object has a single 4-byte local identifier, regardless of whether you are using an object identifier or an internal identifier.

See Also

Assigning Object Identifiers

How Repository Identifiers are Stored and Instantiated

Repository Identifier Data Structures

Repository Identifiers

RepositoryObject Object

RTblSites SQL Table

Object-Version Identifiers and Internal Object-Version Identifiers

Each **RepositoryObjectVersion** instance has two identifiers: an object-version identifier and an internal object-version identifier.

An object-version identifier uniquely distinguishes a repository object from all other repository object versions represented in all other repository databases.

Internal object-version identifiers correspond to object-version identifiers, except that internal object-version identifiers are used by the repository engine. Object-version identifiers and internal object-version identifiers are described in this topic. Another kind of repository identifier identifies a **RepositoryObject** instance. To use identifiers, you need to know about both kinds. For more information about other repository identifiers, see Object Identifiers and Internal Identifiers.

About Object-Version Identifiers

The object-version identifier has the following format.

The first 16 bytes of each object-version identifier constitute a globally unique identifier (or GUID).

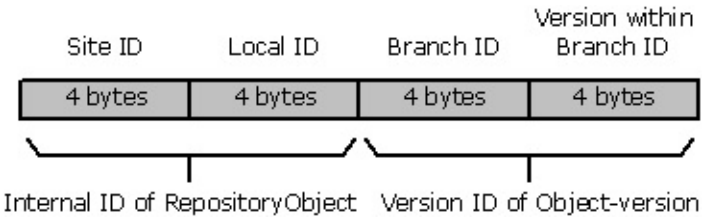
The next 4 bytes constitute a local identifier. Note that the local identifier of a repository object version does not equal the local identifier of a **RepositoryObject** instance.

The next 8 bytes constitute a version indicator. Each object version gets a unique value that identifies a specific version of a particular object (for example, version 3 of an **Employee** object). To get a specific version of an object, you have to traverse the version tree of an object.

About Internal Object-Version Identifiers

Each **RepositoryObjectVersion** instance also has an internal object-version

identifier that distinguishes it from every other object version within the same repository database. The internal object-version identifier is a 16-byte quantity of the following form.



The first 4 bytes constitute a site identifier (site ID). For more information about site IDs, see How Repository Identifiers are Stored and Instantiated.

The first 4 bytes constitute the local identifier (local ID) of the repository object. The second 4 bytes constitute a branch identifier (branch ID); a branch is a portion of a version graph. The third 4 bytes constitute a version-within-branch identifier.

The first 8 bytes make up the internal identifier of the repository object version. The next 8 bytes make up a version identifier.

See Also

Branches in the Version Graph

Repository Identifier Data Structures

Repository Identifiers

RepositoryObjectVersion Object

Version Graph

How Repository Identifiers are Stored and Instantiated

A site identifier (site ID) is a portion of the internal identifier (or internal object-version identifier) of a repository object. A globally unique identifier (GUID) is a portion of an object identifier (or object-version identifier).

There is a one-to-one correspondence between a site ID and its GUID, and each repository database includes a table (**RTblSites**) that maintains this correspondence. Each row of the table associates one GUID with one site ID.

The repository engine uses the one-to-one correspondence between the site identifiers and GUIDs to conserve space in the repository database. When the repository engine stores a repository object, it stores the internal identifier with the object. The engine does not store the GUID or the object identifier with the repository object. When you need to retrieve the object identifier of an object, the repository engine constructs the object identifier by reading the internal identifier stored with the object, matching the site identifier to the appropriate row of the **RTblSites** table, and reading the GUID from that row.

See Also

Object Identifiers and Internal Identifiers

Object-Version Identifiers and Internal Object-Version Identifiers

Repository Identifiers

RTblSites SQL Table

Repository Identifier Data Structures

The following data structures describe object identifiers, object-version identifiers, internal identifiers, and internal object-version identifiers.

If you are querying the database by building a query against the repository tables, you need to know about internal identifier data structures to form the query. Information in this topic about object identifier and object-version identifier data structures is provided for completeness. Only internal identifiers and internal object-version identifiers are used to build queries.

Internal Identifier

```
struct INTID {
ULONG  iSiteID;
ULONG  iLocalID;
};
typedef const INTID &REFINTID;
```

An **INTID** or a **REFINTID** variable is an internal identifier for a specific repository object that uniquely identifies the object within a particular repository database. An internal identifier is not unique across all repositories. Note that an internal identifier is not the same thing as the interface identifier for an interface, or the class identifier that is used to create an instance of a class.

The internal identifier is composed of an internal site identifier (iSiteID) and an internal local identifier (iLocalID).

Internal Object-Version Identifier

```
struct VERSIONID {
INTID  sIntID;
BRANCHID  iBranchID;
VERSIONNUM  iVersionStart;
};
```

typedef const VERSIONID &REFVERSIONID;

A **VERSIONID** or a **REFVERSIONID** variable is an internal identifier for a specific repository object version that uniquely identifies the object version within a particular repository database. It is not unique across all repositories.

The internal identifier is composed of an internal identifier (sIntID), a branch identifier (iBranchID), and a version-within-branch identifier (iVersionStart).

Object Identifiers and Object-Version Identifiers

typedef const OBJECTID OBJID;
typedef const OBJID &REFOBJID;

An **OBJID** or a **REFOBJID** variable can be used in either of two ways:

- It can be an object identifier for a specific repository object in a particular repository database. An object identifier is unique across all repositories.

- It can be an object-version identifier for a specific version of a repository object in a particular repository database. An object-version identifier is unique across all repositories.

An **OBJID** or a **REFOBJID** variable is composed of a global unique identifier (GUID) and a 4-byte local identifier appended to the GUID. The GUID portion of the variable specifies where the identifier was created, and the local identifier has a value that is unique within the repository database. When you use an **OBJID** or a **REFOBJID** variable to contain an object-version identifier, the 4-byte local identifier is not the branch identifier or the version-within-branch identifier of the object version.

See Also

Branches in the Version Graph

Object Identifiers and Internal Identifiers

Assigning Object Identifiers

When you install an information model in a repository, the repository engine creates a number of objects to represent the classes, interfaces, collection types, properties, and relationship types of that model. The assignment of an object identifier to an object occurs when the object definition is inserted into a repository database. If you are installing an information model using the model installer, the repository engine assigns the identifier.

If you are inserting an information model programmatically, you can still let the repository engine assign an identifier automatically, or you can provide an object identifier manually. To let the repository engine assign an identifier for a programmatically inserted object, set the input parameter for the object identifier to **OBJID_NULL**.

In most cases, you should let the repository engine assign object identifiers. The exception is when you are inserting a replica of an object represented in one Microsoft® SQL Server™ 2000 Meta Data Services repository database into another Meta Data Services repository. For each type definition object that you copy to the new repository, you should use the object identifier that was assigned to that object in the existing repository. This will ensure that the type definition has the same identity in both repositories.

If you explicitly assign object identifiers for your definition objects, you must ensure that the object identifiers are unique across all repositories. The following steps are recommended to guarantee such uniqueness:

1. Generate a single unused GUID and use it for the GUID portion (the first 16 bytes) of all definition object identifiers for the information model.

2. Using the **CreateObject** method, manually assign unique local identifiers for each definition object in the information model.

Using Guidgen

When creating object identifiers for your information model, you can use the Guidgen.exe program to create an unused GUID, and use the **DEFINE_OBJID** macro to create the object identifiers. Given a GUID and a unique number for an object, the macro will equate the symbolic name to the value for the object identifier. Use the **DEFINE_OBJID** macro (which is provided for both Microsoft Visual C++® and Microsoft Visual Basic® programmers) to avoid incompatibility problems later.

See Also

Installing Information Models

IRepository::CreateObject

Object-Version Identifiers and Internal Object-Version Identifiers

Repository CreateObject Method

Repository Identifiers

Meta Data Services Programming

Naming Objects, Collections, and Relationships

This section provides guidelines for identifying objects, collections, and relationships by name. Different naming guidelines apply depending on whether you are naming objects of an information model, or naming object instances in a repository.

Naming Information Model Elements

When you create **ClassDef**, **RelationshipDef**, and **CollectionDef** objects in an information model, you specify a name that you can use later to reference that meta data type. You can provide this name by specifying the *Name* parameter in a creation method (for example, **CreateClassDef**, **CreateInterfaceDef**, **CreateRelationColDef**, and so on).

Depending on how you define a relationship collection, you can determine how objects of that collection are subsequently named (this naming occurs when you populate an information model). Specifically, you can specify that object names are explicitly named through the **INamedObject** interface. If you are accustomed to assigning object names yourself, or if your information model is structured in such a way that the destination of a naming relationship collection is not obvious, you can use this interface to attach a **Name** property to an information model object. You can then provide a name when creating an instance of that object.

The following example shows an incomplete code sample that gives you a basic idea about how to implement **INamedObject** for a repository object. When you use this interface, be sure to set the COLLECTION_OBJECTNAMING flag on the collection.

```
Dim oRepository as Repository
Dim oCObject as ClassDef
Dim oINamedObject as InterfaceDef
Dim oIObject as InterfaceDef
Dim oRContains as RelationshipDef
Dim oColObjectContains as CollectionDef
```

...
Set oINamedObject = oRepository.object(OBJID_INamedObject)
oCObject.AddInterface oINamedObject
...
Set oColObjectContains = oIObject.CreateRealtionshipColDef(objid_r

Naming Object Instances

When you populate an information model with meta data instances, you can allow the repository engine to name the object for you, or you can provide a name.

How the Repository Engine Names Object Instances

The repository engine uses relationship collections to create names for objects. Specifically, the relationship collection that determines an object instance name is the target object collection. When the target object collection contains uniquely named objects, and when it is the sole target of the source object, the identity of the target object is unambiguous. However, if more than one target is possible, you should assign an explicit name to avoid having the repository engine select one for you.

You can choose to let objects assume different names when accessed through a relationship, as opposed to the single name that it assumes when it is accessed through the object. Naming an object through a relationship has the benefit of referring to the same object through different names depending on the context in which it is used. In this case, the relationship collection provides the context.

For more information, see Naming and Unique-naming Collections.

How to Explicitly Name an Object Instance

If an object supports the **INamedObject** interface in the information model, you can call an object by its **INamedObject::Name** property. You can also use **IRepositoryItem::Name** to supply a name.

See Also

Type Information Aliasing

The information model elements that you create support type information aliasing. This form of aliasing enables you to define an alternate name for the meta data type so that you can reuse an existing definition in a new context. You can also use type information aliases to preserve existing work when information model names change. For example, Open Information Model (OIM) or Unified Modeling Language (UML) name changes that result from new versions of a model can be accommodated by applying aliases to a changed name.

To define a type information alias, use the following interfaces:

- **IReposTypeInfo2** defines aliases for **Classdef**, **Interfacedef**, **Relationshipdef**, and **Enumerationdef** objects.

- **IInterfaceMember2** defines aliases for **Propertydef**, **Methoddef**, **Alias**, and **Collectiondef** objects.

To use the alias, specify it just as you would the meta data type name. The repository engine keeps track of type information aliases. When you invoke a type information alias, the repository engine returns the appropriate class, interface, or property to which the alias is mapped.

Note Aliasing provides additional functionality when it is applied to interface members. For more information, see Derived Members.

See Also

IInterfaceMember2 Interface

IReposTypeInfo2 Interface

Member Delegation

Naming Conventions

Naming Objects, Collections, and Relationships

Naming Conventions

Names must always be unique within a scope. The scope varies depending on the object. Within a repository, information model names (that is, repository type library names) cannot repeat. Within an information model, class, interface, and relationship names cannot repeat. Similarly, within an interface, property, collection, and method names cannot repeat. Also, within a collection that supports unique naming, object names cannot repeat.

When you create a new information model, choose your names carefully. Otherwise, you may encounter name duplication problems later on if you decide to share information models. One way to avoid name confusion is by using a distinctive prefix on all of your names. An information model name provides an obvious solution. For example, if you are using the Open Information Model (OIM), you can use the subject area names such as Database Schema (or DBSchema) as a prefix.

In addition to unique constraints, the following naming conventions apply to Repository Type Information (RTIM) objects and relationships:

- The name cannot be a reserved SQL or MIDL keyword. Generally, you should avoid any word that is reserved by a DBMS.

- Names can be a maximum of 249 characters in length.

- Any alphanumeric character can be used in the name.

- For object instance names, you can define a name that contains leading or trailing spaces. It can also be an empty string. If the name is all spaces, it is treated as an empty string.

  Spaces within a name are allowed because COM supports it. However, if you include spaces in an interface definition name, you will get an error when you subsequently define properties on that interface.

See Also

Naming and Unique-Naming Collections

Naming Conventions for Generated Views

Naming Objects, Collections, and Relationships

Naming and Unique-Naming Collections

Certain relationships can provide a name by which the origin object refers to the destination object. Such a relationship is called a naming relationship. A collection of naming relationships is a naming collection.

Certain naming collections require that all destination objects in the collection have unique names. Such a collection is referred to as a unique-naming collection.

User requirements may require objects to support multiple names. For example, consider a system in which a single program can have two different file names, because there are two different file systems that allow and disallow long names, respectively. The following figure illustrates this case.

The figure shows four relationships. Each relationship specifies a name by which one of the objects (the origin object) refers to the other object (the destination object). In particular, notice that the object representing the error-handling program file has two different names, ErrHndl and ErrorHandler.

In order to support this kind of capability, the Repository Type Information Model (RTIM) attaches the **Name** property to the relationship type, not to the object class. This enables an object to have as many names as it has relationships (that is, relationships for which it is the destination).

Object Naming Collections

If the COLLECTION_OBJECTNAMING flag is set, there is no relation-specific naming of this object. The object has the same name in the relationship as specified by the **INamedObject::Name** property on the object. Specifying a name during the collection's **Add** operation or attempting to set the **IRepositoryItem:Name** property on the relationship object will return the error EREP_COL_OBJECTNAMING. If you attempt to add an object that does not support **INamedObject** to the collection, the error EREP_COL_OBJECTNOTNAMED is returned.

See Also

CollectionDefFlags Enumeration

INamedObject Interface

Naming Objects, Collections, and Relationships

Repository Errors (alphabetical order)

Retrieving an Object Version's Name

When you try to retrieve the name of an object version, the repository engine can search in several places for the name:

- If the object version implements the **INamedObject** interface, the repository engine retrieves the **Name** property exposed by that interface.

- If the object version does not implement the **INamedObject** interface, the repository engine seeks a destination naming relationship for the object version. With that destination naming relationship, the repository engine performs object-version resolution, yielding a particular origin object version from the relationship's **TargetVersions** collection. The repository engine retrieves the name by which that origin object version refers to the destination object.

See Also

Changing a Destination Relationship's Name

Changing an Object Version's Name

INamedObject Interface

Naming Objects, Collections, and Relationships

Resolution Strategy for Objects and Object Versions

Changing an Object Version's Name

When you change the name of an object version, the repository engine might try to change several names as follows:

- If the object version implements the **INamedObject** interface, the repository engine changes the **Name** property exposed by **INamedObject** unless the object version is unchangeable.

- If the object version has one or more destination naming relationships, the repository engine tries to change a name for each of those relationships. For more information, see Changing a Destination Relationship's Name.

See Also

IRepositoryItem Interface

Repository Object

Repository ConnectionString Property

Retrieving an Object Version's Name

Changing a Destination Relationship's Name

A name associated with a naming relationship is the origin object version's name for the destination object. When you change the name of a destination naming relationship, you simultaneously change an origin version's name for the destination object. If the destination relationship has multiple items in its **TargetVersions** collection, each of those origin versions could have a different name for the destination object. The repository engine follows a resolution strategy to choose a particular origin object version from the destination relationship's **TargetVersions** collection. Next, the repository engine changes the origin object version's name for the destination object, unless the origin object version is unchangeable.

See Also

Changing an Object Version's Name

Resolution Strategy for Objects and Object Versions

Retrieving an Object Version's Name

Naming Stored Procedures

When you use a Microsoft® SQL Server™ database for your repository, the repository engine creates stored procedures for the insertion of rows into the repository SQL tables. This topic describes how these stored procedures are named.

The stored procedure name for a table is generated by prefixing the table name with the string "R_i". Because table names are unique, this naming convention will generate unique stored procedure names. If the length of the table name is greater than **MaxIdentifierLength**-3, however, the table name generation algorithm fails. For this reason, a user may not supply a table name longer than **MaxIdentifierLength**-3. Supplying a longer name causes the error EREP_BADNAME.

When the user does not provide a table name for an interface, the engine automatically generates the table name from the interface name. If the interface name, with the leading "I" stripped off, is less than **MaxIdentifierLength**-4, the interface name will be used as the table name. Otherwise, the interface name is truncated to **MaxIdentifierLength**-7, and a 4-character number is appended to the name to make it unique, before prefixing "R_i.

The engine uses named arguments to call the stored procedures. A named argument starts with the at sign (@) character and is no longer than **MaxIdentifierLength**. Therefore, the property names, which are also column names, must be no longer than **MaxIdentifierLength**-1.

MaxIdentifierLength values are 30 characters for SQL Server version 6.5 and 128 characters for SQL Server version 7.0 and SQL Server 2000.

See Also

Naming Objects, Collections, and Relationships

Repository Errors (Alphabetical Order)

Repository SQL Schema

Meta Data Services Programming

Creating and Extending Type Information

Information models contain type information about the tools and applications you develop. Creating an information model is the first step in developing tools and applications with the Microsoft® SQL Server™ 2000 Meta Data Services repository.

You can build custom information models, or use the predefined information model distributed with SQL Server 2000. SQL Server distributes the Open Information Model (OIM). You can obtain a more recent version of the OIM from the Meta Data Coalition (MDC) or the Meta Data Services Software Development Kit (SDK).

If you are using a predefined information model, the information model is created for you. However, you can extend a predefined information model if you want to add elements that further describe the tool or application you want to develop. Extending an information model is equivalent to creating a new model.

The following topics detail different strategies for creating an information model.

Topic	Description
Creating Type Information Using Modeling Tools	Describes the advantages of creating an information model with modeling tools
Information Model Creation Issues	Identifies choices you can make about a model you create, and identifies some basic requirements for creating a navigable information model
Creating Type Information Programmatically	Details the steps to follow when creating an information model through code

See Also

Getting Started with Meta Data Services

Information Models

OIM in Meta Data Services

Meta Data Services SDK

Repository API Reference

Creating Type Information Using Modeling Tools

Using a third-party tool to create an information model in a visual modeling environment is strongly recommended. Information models are complex to design and difficult to get right the first time. Unless you are creating the simplest of models, or making small changes to an existing model, you should invest in a tool to develop your design.

In addition to providing a visual modeling environment, modeling tools provide support for multiple users, version control, report generation, and integration with application programming environments.

The Microsoft® SQL Server™ 2000 Meta Data Services Software Development Kit (SDK) includes utilities for creating and extending an information model. You can access these utilities from within third-party modeling tools by way of extensions.

See Also

Creating and Extending Type Information

Information Models

Meta Data Services SDK

Information Model Creation Issues

When you insert an information model into a repository, you have the following choices and decisions to consider:

- How much of the tool information will you store in the repository database and how much will you store in other files?
- For each class you define in your information model, what Automation server will create instances of that class?
- Can you tune the database schema to improve performance?
- Should you create a new information model or extend an existing one?
- Should you assign object identifiers, or let the repository engine do this for you?
- How will your information model accommodate navigation from one object to another?

The following topics discuss each of these questions.

Topic	Description
Extending vs. Creating Information Models	Provides guidelines to help you decide whether to create or extend an information model
Choosing Which Information Belongs in the Repository	Provides guidelines to help you decide where a Microsoft® SQL Server™ 2000 Meta Data Services repository fits into your development environment
Choosing an Automation Server	Explains how to instantiate an

for a Class	Automation server in your application code
Tuning the Database Schema of an Information Model	Provides tips that can help you determine how the repository database is created
Accommodating Navigation Within an Information Model	Explains the core requirements for building a navigable information model
Assigning Object Identifiers	Explains the ways in which a repository identifier can be assigned to an object

See Also

Creating Type Information Programmatically

Extending vs. Creating Information Models

You can create a new information model or extend an existing one. In general, each information model should accommodate the data about a particular area of your application environment.

When faced with the decision of whether to extend an existing information model or build a new one, you can follow these guidelines to determine a course of action.

- To accommodate data or structures that are unrelated or only minimally related to existing type information, create a new information model.

- To accommodate additional kinds of data that are closely related to existing type information model, extend an existing information model.

After you decide that extending an information model is the right choice, you must decide whether to extend it through a modeling tool or through the repository API. If you do not own a modeling tool or if the change is small, you can use the repository API to create type information programmatically.

See Also

Creating Type Information Programmatically

Information Model Creation Issues

Choosing Which Information Belongs in the Repository

You do not need to store all of the information for your tool in a Microsoft® SQL Server™ 2000 Meta Data Services repository. For example, suppose your tool helps application developers and systems engineers keep track of the bugs on their software systems. Your tool maintains modules, bug reports, and test suites. Each module has a name, an author, source code, and one or more bugs reported on it. Each bug can have a description, a module on which it is reported, and a test suite used to reproduce the bug. Each test suite can have one or more bugs that it can reproduce. Because your tool maintains each test suite in a file format, you decide not to explicitly insert each test suite into a repository. Instead, you store in the repository only the name of a file containing the test suite.

To decide which information belongs in the repository, consider the following questions:

- Do you want to perform impact analysis on the data?

  The more information you store in the repository, the more impact analysis questions you can answer. Consider the example described previously. Because the information model includes a class describing test suites, you can learn which test suite generates the most bugs.

  Conversely, because the information model does not include a class accommodating individual tests or the persons responsible for them, you cannot use the repository to learn which person discovers the most bugs.

- Is there another file format that is more appropriate for the fine details of the definitions that describe your tool?

  There are two aspects to consider:

  - If a tool manipulates objects whose fundamental units of storage and manipulation are large, a file format can be more efficient than the repository. In this case, it is probably more

effective to store the data objects in their native file format, and to store in the repository a description of each data object.

- If an existing tool already stores its data in a file format, switching to Meta Data Services would require rewriting the tool. To save time, you can choose to retain the existing file format and replicate some subset of the tool data in the repository.

See Also

Creating Type Information Programmatically

Information Model Creation Issues

Choosing an Automation Server for a Class

After you add your information model to a repository, you can run your tool. Periodically, your tool will invoke the **CreateObject** method to create an instance of one of the classes of your information model. **CreateObject** must create a run-time object (that is, an Automation object). To create the run-time object, the repository engine calls **CoCreateInstance**, using as a parameter the **ClassID** you provided as a property of the class.

When the engine calls **CoCreateInstance** with the **ClassID**, the system registry is checked to determine which Automation server contains the required class factory. For most classes in your information model, a generic Automation server for repository objects, Repodbc.dll, suffices. To use the generic server as the Automation server for a class, you can either do nothing, or you can specify Repodbc.dll in the entry for that class in the registry.

Although Repodbc.dll suffices as the Automation server for most repository classes, you will occasionally create a class whose instances require special treatment. For example:

- A class of your information model requires input validation.

  You can validate the property values or collections of each instance of a class by writing a special Automation server for that class.

- A class of your information model replicates some properties retained in another file format outside the repository.

  Suppose your information model includes a class whose instances describe Microsoft® Word documents. Each instance describes a Word document, indicating specifically its title, subject, and author. Your class-specific Automation server must include special code to ensure that the values of the repository properties match the values of title, subject, and author stored in the Word file.

- A class of your information model requires some class-specific behavior that you implement in a method.

Suppose your information model includes a class whose instances describe modem pools. Each instance describes a particular modem pool, including its phone number. Your class-specific Automation server can include a method to automatically dial the number and establish a connection.

Note At this time, the repository engine supports in-process Automation servers only (that is, dynamic-link libraries).

See Also

Creating Type Information Programmatically

Information Model Creation Issues

Tuning the Database Schema of an Information Model

The repository engine stores data in a relational database. When you add an information model, the repository engine enlarges this database by creating new tables and columns to accommodate your tool information. Generally, each interface corresponds to a table, and each property corresponds to a column. When you populate your information model, the repository engine inserts rows into these tables.

You have some control over the database schema that accommodates your tool information. For example, you can:

- Use a single table to contain the interface-specific properties of more than one interface.

  To do this, set the **TableName** property for each interface definition object to the same name before you commit the transaction that is used to create your information model.

- Create an additional index for a table.

  To do this, open the database directly and use the SQL CREATE INDEX command after you commit the transaction that is used to create your information model.

  Note You cannot completely control the database schema. In particular, each table must include the columns **IntID**, **Z_BrID_Z**, and **Z_VS_Z**, and must define the primary key on those columns. Furthermore, you cannot drop columns that your information model uses to store properties.

See Also

Creating Type Information Programmatically

Information Model Creation Issues

Accommodating Navigation within an Information Model

Because the objects in your information model are associated through a network of relationships, you can navigate to each part of an information model through the relationships you define.

To support programming against the information model, you must build in navigation support by way of relationships.

The first relationship must be between the repository root object and an object in your information model. To enable this first navigation step, include in your information model a relationship type whose instances will associate the root object with objects stored in your information model.

Create a relationship type with these characteristics:

- The origin collection type of the relationship type is a member of an interface implemented by the **ReposRoot** class.

- The destination collection type of the relationship type is a member of an interface implemented by a class of the information model.

To create this relationship type

1. Create a new interface and add it to the set of interfaces implemented by the **ReposRoot** class.

2. Create a relationship type associating the newly created interface with some interface implemented by a class of your information model. Choose an interface implemented by a fundamental class, a class whose instances are good objects from which to begin moving to other objects of the information model.

For more information about moving through a repository, see Navigating a Repository.

See Also

Creating Type Information Programmatically

Information Model Creation Issues

Creating Type Information Programmatically

If you do not have a modeling tool, you can create a new information model or extend an existing one programmatically. This section explains the steps you need to follow when creating an information model, and discusses some issues to consider in designing and inserting information models into a Microsoft® SQL Server™ 2000 Meta Data Services repository.

When you insert an information model into a repository, you populate the Repository Type Information Model (RTIM). That is, you create instances of the classes, interfaces, properties, methods, and relationship types of the RTIM.

The following topics describe the steps in creating type information programmatically.

Topic	Description
Begin a Transaction	Explains how to begin a transaction that brackets the information model definitions
Create a Repository Type Library	Describes how to create an empty information model to store subsequent definition
Define Dependencies Between Type Libraries	Explains how to define dependencies between multiple information models
Add Classes to the Repository Type Library	Details how to add class definitions to an information model
Add Interfaces to Each Class	Details how to add interface definitions to an information model
Add Properties to Each Interface	Details how to add property definitions to an interface
Add Methods to Each Interface	Details how to add method definitions to an interface
Add Relationship Types and Pairs of Collection Types	Details how to add relationship and collection definitions to an interface
Commit the Transaction	Describes how to commit the transaction

	that inserts your definitions into a Meta Data Services repository

See Also

Creating and Extending Type Information

Information Model Creation Issues

Information Models

Repository API Reference

Repository Object Architecture

Begin a Transaction

To write data to a repository, bracket your interactions within the scope of a transaction.

To begin a transaction

1. Open or create the repository into which you want to insert the information model. To open an existing repository, use the **Open** method of the **IRepository** interface or the **Repository** object.

   -or-

   To create a new repository, use the **Create** method of the **IRepository** interface.

   Both of these methods return the root object for the open repository.

2. Invoke the **Begin** method of the **IRepositoryTransaction** interface or **RepositoryTransaction** object.

   The **IRepositoryTransaction** interface is accessible through the **Transaction** property of the object that represents your connection to the repository.

See Also

Creating Type Information Programmatically

Create a Repository Type Library

Commit the Transaction

Connecting to and Configuring a Repository

IRepository Interface

IRepositoryTransaction Interface

Create a Repository Type Library

The Repository Type Information Model (RTIM) includes a class named **ReposTypeLib**; each instance of this class corresponds to a repository type library. Each repository type library describes an information model.

To create an instance of the ReposTypeLib class

- Use the **CreateTypeLib** method of the root object's **IManageReposTypeLib** interface or the **ReposRoot** object.

  Note Each instance of **ReposTypeLib** has a collection of types, where each type is either a class, an interface, or a relationship type. The collection is called **ReposTypeInfos**, and is used to ensure that unique names are used for all classes, interfaces, and relationship types in your information model.

See Also

Creating Type Information Programmatically

Define Dependencies Between Type Libraries

IManageReposTypeLib Interface

ReposRoot Object

ReposTypeLib Class

Define Dependencies Between Type Libraries

The Repository Type Information Model (RTIM) allows model developers to define dependencies between type libraries. You can define dependencies if you want to share information models, or leverage an existing information model within a new context.

To define a dependency

- Use the **DependsOn** collection on the **IReposTypeLib2** interface to define a dependency between two type libraries. For example, in order to define a dependency between file allocation table (FAT) and FileSys type libraries:

Use the following code to define a dependency in Automation:

```
FAT.DependsOn("IReposTypeLib2").Add FileSys
```

Use the following code to define a dependency in COM:

```
pFATCol -> Add(pFileSys, RelShipName, &pRelShipName);
```

pFATCol

A pointer to the FAT type library **DependsOn** collection on the **IReposTypeLib2** Interface.

pFileSys

A pointer to the FileSys type library.

RelShipName

The name of the relationship between the FAT and the FileSys type libraries.

pRelShipName

A pointer to the relationship between the FAT and the FileSys type libraries.

See Also

Add Classes to the Repository Type Library

Creating Type Information Programmatically

IReposTypeLib2 DependsOn Collection

Add Classes to the Repository Type Library

According to the Repository Type Information Model (RTIM), you define a new object class by creating an instance of the **ClassDef** class.

To create a new class definition

- Use the **CreateClassDef** method of the **IReposTypeLib** interface that is exposed by your **ReposTypeLib** object.

  Be sure the class identifier that you supply as an input parameter to this method matches the globally unique identification (GUID) of that class in the system registry.

  Note Within the system registry, you can indicate which Automation server the repository engine uses to create instances of your new class. You can use the Automation server that the repository engine provides for all repository objects, or you can use your own server. For more information about deciding which kind of Automation server to use, see Information Model Creation Issues.

See Also

Add Interfaces to Each Class

Creating Type Information Programmatically

ReposTypeLib Object

Add Interfaces to Each Class

Each of the classes in your information model exposes one or more interfaces. Add a new interface to a class by creating an instance of the **InterfaceDef** class.

When you create a custom interface, you must avoid assigning a dispatch ID of 1000 to the interface. **IRepositoryDispatch::get_Properties** reserves this value for itself.

To create a new interface definition

- Use the **CreateInterfaceDef** method of the **IClassDef** interface that is exposed by your **ClassDef** object.

  Be sure the interface identifier that you supply as an input parameter to this method matches the globally unique identification (GUID) that has been assigned to the interface.

  Note Among the interfaces you create for your information model, you must include an interface that the **ReposRoot** class implements. You need this interface and an attendant relationship type to enable navigation to the objects that will populate your information model. For more information about why you need this interface, see Information Model Creation Issues.

See Also

Add Methods to Each Interface

Add Properties to Each Interface

Add Relationship Types and Pairs of Collection Types

ClassDef Object

Creating Type Information Programmatically

IClassDef Interface

Add Properties to Each Interface

Each interface in your information model can expose properties. Attach a new property to an interface by creating an instance of the **PropertyDef** class.

To create a new property definition

- Use the **CreatePropertyDef** method of the **IInterfaceDef** interface that is exposed by your **InterfaceDef** object.

See Also

Add Methods to Each Interface

Add Relationship Types and Pairs of Collection Types

Creating Type Information Programmatically

IInterfaceDef Interface

Add Methods to Each Interface

Each of the interfaces in your information model can expose methods. Attach a new method to an interface by creating an instance of the **MethodDef** class.

To create a new method definition

- Use the **CreateMethodDef** method of the **IInterfaceDef** interface that is exposed by your **InterfaceDef** object.

  Note If your interface has methods, you must provide your own Automation server for classes that implement this interface. For more information about deciding which kind of Automation server to use, see Information Model Creation Issues.

See Also

Add Properties to Each Interface

Add Relationship Types and Pairs of Collection Types

Creating Type Information Programmatically

IInterfaceDef Interface

Add Relationship Types and Pairs of Collection Types

Relationships connect objects to each other in a Microsoft® SQL Server™ 2000 Meta Data Services repository. When you define a new relationship type, you also define an origin collection type and a destination collection type. The origin collection type connects the relationship type to one interface; the destination collection type connects the relationship type to a second interface. The classes that implement those interfaces are now related in your information model.

To create a new relationship type (and its corresponding pair of collection types) and attach it to two interfaces

- Use the **CreateRelationshipDef** method of the **IReposTypeLib** interface that is exposed by your **ReposTypeLib** object. Then, use the CreateRelationshipCol method to create the collection. For more information, see Defining a Collection.

  Note One of the relationship types that you create for your information model must enable navigation from the root object of the repository to some objects of your information model. For more information about why you need this interface, see Information Model Creation Issues.

See Also

Add Properties to Each Interface

Add Methods to Each Interface

Creating Type Information Programmatically

IReposTypeLib Interface

Commit the Transaction

When you have added all of the type definition objects to your information model, use the **Commit** method of the **IRepositoryTransaction** interface to commit your additions to the repository database.

See Also

Creating Type Information Programmatically

Information Model Creation Issues

IRepositoryTransaction Interface

Meta Data Services Programming

Defining Relationships and Collections

Relationships and collections provide the structure and navigation of your information model. After you define the objects you require, you need to associate the objects by defining relationships. The relationships that you create must be defined as collections. You can also create collections that do not contain relationships.

The following collection types are possible.

Collection	Description
Object collection	Contains multiple instances of the same type of object (for example, a set of **StoredProcedure** objects). Object collections are only used in an **ObjectInstances** collection on **ClassDef** and **InterfaceDef** objects. For more information, see Defining a Collection and ObjectCol Object.
Version collection	Contains versioned objects. There are seven kinds of version collections. For more information, see Kinds of Version Collections.
Relationship collection	Contains relationship objects. Each relationship object pairs one source object to one target object. Relationship collections can be used for navigation. For more information, see Defining a Relationship and Defining a Relationship Collection.
Target object collection	Contains all of the target objects of a specific source object. For example, the target object collection of a **Table** source object can be a collection of **Column** objects. A target object collection is represented as a property that returns a **TargetObjectCol** object. For more information, see Defining a Target Object Collection and ITargetObjectCol Interface.

Transient object collection	Contains objects that are populated by code. A transient object collection is a special case of collection type. Where the other collection types are formed from persistent object data, a transient object collection is instantiated from your code. This collection is populated dynamically and does not rely on persistent data to determine its contents. For more information about defining transient object collections, see Programming Transient Object Collections.

See Also

Navigating a Repository

Understanding Collections

Understanding Relationship Roles

Defining a Relationship

Relationships are used to navigate. At a minimum, you need to know how to traverse relationships to get from one object to another. Furthermore, the repository engine supports complex behavior that varies based on relationships: namely, delete propagation, version propagation, view generation, version resolution, and naming relationships. To understand how the repository engine responds to these cases, you need to know about relationships and how they are structured.

Origin and Destination Collections

All relationships are accessed by way of a collection. For this reason, you must always associate objects through a relationship definition. When an origin object accesses a destination object, the origin object accesses a relationship collection that contains destination objects. When a destination object accesses an origin object, the destination object accesses a relationship collection that contains the origin object.

To support access in both directions, you must always provide two collections. From the perspective of the origin object, the relationship collection to which it is attached is a destination collection. Similarly, from the perspective of the destination object, the relationship collection to which it is attached is an origin collection.

When you define a relationship type and its attendant collection types, you must declare one collection type as the origin. To do this, you must set the **IsOrigin** property on the collection.

In addition to providing access, the distinction between origin and destination collections is important because naming collections, unique-naming collections, and sequenced collections can only be defined on origin collections.

The following example illustrates how to create an origin collection that supports unique-naming and sequencing. In this example, **objid_null** is the object identifier, **name_** is the string that defines a name, and **dispid_** indicates a dispatch identifier (a constant not shown in this example).

```
Rem ** Declare interface, collection, and relationship
Dim oTypeLib as RepositoryTypeLib
Dim oRContains as RelationshipDef
Dim oCTableContains as CollectionDef

Rem ** Create the oRContains relationship on oTypeLib
Set oRContains = oTypeLib.CreateRelationshipDef(objid_null, name_

Rem ** Create the relationship collection for oRContains
Rem ** IsOrigin is set to True
Rem ** 10 is the combined bits for CollectionDef flags (2 for uniquen
Set oCTableContains = oTypeLib.CreateRelationshipColDef(objid_nul
```

See Also

CollectionDef IsOrigin Property

Defining Relationships and Collections

Defining a Collection

ICollectionDef IsOrigin Property

IRelationship Interface

Relationship Class

Relationship Object

Defining a Collection

Collections are a kind of property that provide a way to relate and group objects. Each object can support multiple collections.

Collections are materialized at run time, using interfaces that you call. The kind of collection that is materialized depends on the interface you use. Because the repository stores data, the collections that you materialize assume the state that they had the last time the collection was instantiated. For example, a collection that contains three objects at the end of one repository session will still contain those three objects the next time you run a repository session.

The rule for an object-collection association is object to collection to object. In an information model, objects are never related to each other directly. Objects are always associated through a collection. For example, if the relationship between two objects is strictly one-to-one, the collections that associate the objects each contain one object.

To define a collection, use the **CollectionDef** class or **ICollectionDef** interface for COM programs, or **CollectionDef** object for Automation programs.

The following example illustrates how to define two collections for a single relationship. The pattern of two collections for each relationship holds for all relationships that you create. In this example, **objid_null** is the object identifier, **name_** is the string that defines a name, and **dispid_** indicates a dispatch identifier (a constant not shown in this example).

```
Rem ** Declare interfaces, relationship, and collections
Dim oTypeLib as RepositoryTypeLib
Dim oRContains as RelationshipDef
Dim oCTableContains as CollectionDef
Dim oCTableContainedBy as CollectionDef

Rem ** Create the relationship oContains on oTypeLib
Set oRContains = oTypeLib.CreateRelationshipDef(objid_null, name_
```

```
Rem ** Create the Contains and ContainedBy collections
Set oCTableContains = oITypeLib.CreateRelationshipColDef(objid_nu
Set oCTableContainedBy = oTypeLib.CreateRelationshipColDef(objid
```

See Also

CollectionDef Class

CollectionDef Object

Defining Relationships and Collections

ICollectionDef Interface

Understanding Collections

Understanding Relationship Roles

Sequenced Collections

Some information models require the tool or application be able to set the sequence of items in a collection. This requirement occurs when the sequence itself is significant in some way.

A collection that supports the sequencing of its items is referred to as a sequenced collection. Relationships contained within such a collection are sequenced relationships. The Repository Type Information Model (RTIM) supports the definition of collection types for sequenced collections.

For example, consider a report generator tool that displays tables of data where the data is displayed in rows and columns. Each table is represented by an object that conforms to the **Table** class. The columns of the table are represented by objects that conform to the **Column** class. Each table has a collection of columns that are included in it (the relationship that relates a **Table** object to a **Column** object is the *includes* relationship). The following figure illustrates this example.

To determine the order in which the columns will appear when the table is displayed or printed, the report generator tool relies on the sequence of the column items. For the **Student** table, the report is displayed with the **Student ID** column leftmost, the **Last Name** column next, and the **First Name** column on the right-hand side.

To define a sequenced collection, set the COLLECTION_SEQUENCED flag on a collection definition object.

See Also

CollectionDefFlags Enumeration

Defining Relationships and Collections

Understanding Collections

Understanding Relationship Roles

Heterogeneous Collections of Objects

Every collection of relationships is homogeneous. In any relationship collection, each item is a relationship of the same relationship type. Collections of objects, however, can be either homogenous or heterogeneous (that is, the items can have different classes).

A collection of objects can be heterogeneous for the following reasons:

- The Repository Type Information Model (RTIM) allows each interface to be implemented by many classes.

- The RTIM expresses each relationship type as an association between two interfaces rather than as an association between two classes.

Each relationship type describes how the objects of classes implementing particular interfaces can be related. Thus, if several classes implement a particular interface, some relationship types involving that interface can yield collections whose target objects span several classes. As you prepare programs that manipulate such collections, do not assume that the collections will contain homogeneous sets of objects.

Note Plan for change; do not assume that your information model will remain unchanged. Although a particular relationship type of your information model might associate two interfaces that are implemented by exactly one class each, you might someday create other classes that implement those same interfaces. Any user who enlarges the number of classes implementing either of those interfaces introduces the possibility for heterogeneous collections of objects. If your programs that use those collections are dependent upon homogeneous collections, you must rewrite them as soon as you implement the interfaces with several classes. To protect your programs from this cause of obsolescence, write them assuming that any collection of objects can be heterogeneous.

See Also

Defining Relationships and Collections

Defining a Relationship Collection

Tools and applications can sometimes manipulate sets of relationships as a single unit. To represent this functionality in your information model, you can use relationship collections. The Repository Type Information Model (RTIM) lets you describe relationship collection types as templates to which relationship collections must conform.

A relationship collection is a set of similar relationships. To be part of a relationship collection, the relationships must be similar in these two ways:

- They must be of the same relationship type. All the relationships in a collection must conform to the same relationship type.

- They must have the same object in the same role. One object must be common to all relationships in the collection. That object must play the same role (either origin or destination) for all relationships in the collection.

To define a relationship collection, use the **RelationshipCol** class or **IRelationshipCol** interface for COM programs, or **Relationship** object for Automation programs.

See Also

Defining Relationships and Collections

Naming and Unique-Naming Collections

Heterogeneous Collections of Objects

IRelationshipCol Interface

RelationshipCol Class

RelationshipCol Object

Sequenced Collections

Defining a Target Object Collection

A target object collection is a kind of special-purpose collection that is designed for navigation. Since target object collections reduce navigation to a one-step operation, you should use target object collections for most of your navigation.

About Target Object Collections

A target object collection is the set of target objects that are associated with the relationships in a particular relationship collection. The relationship collection in the following figure is one example.

This relationship corresponds to this target object collection.

These associations are valid because the underlying data looks like this.

The object that is common to all of the relationships in the corresponding relationship collection is referred to as the source object. In the preceding figure, the object describing Frank is the source object. The objects describing Kim, Iola, and Fenton are target objects.

Implementing a Target Object Collection

A target object collection is represented as a property, which returns an **ITargetObjectCol** object. On this object, you can invoke **QueryInterface** to access the **IRelationshipCol** and **IVersionCol** interfaces.

To define a target object collection, use the **ITargetObjectCol** interface.

See Also

Defining Relationships and Collections

ITargetObjectCol Interface

Defining a Version Collection

Version collections provide a way to manipulate sets of versioned objects.

To define a versioned collection, use the **VersionCol** class or **IVersionCol** interface for COM programs, or **VersionCol** object for Automation programs.

You can also define a versioned relationship. To define a versioned relationship, use the **VersionRelationship** class or **IVersionRelationship** interface for COM programs, or use the **VersionRelationship** object for Automation programs.

See Also

Defining Relationships and Collections

IVersionCol Interface

IVersionedRelationship Interface

Kinds of Version Collections

Understanding Collections

Understanding Relationship Roles

VersionCol Class

VersionCol Object

VersionedRelationship Class

VersionedRelationship Object

Meta Data Services Programming

Defining Properties

You can create a property definition object to represent properties in an information model. Properties can include enumerated values. You can also specify a property as a virtual member if you want to provide a property value from a source other than the repository.

To define a property definition object, you can use the **PropertyDef** class or the **IPropertyDef** interface for COM programs, or the **PropertyDef** object for Automation programs.

The following topics provide more information about defining properties.

Topic	Description
Virtual Members	Describes how to implement a virtual member in your information model
Repository Enumeration Definition	Explains how to provide an enumerated list of values for a property

See Also

Defining Relationships and Collections

IPropertyDef Interface

PropertyDef Class

PropertyDef Object

Virtual Members

Virtual members are properties and collections that are not allocated storage in a repository database. Creating a virtual member is useful if you want to store property values or collection items somewhere other than the repository.

Virtual members are defined on interfaces. You can define nonpersistent members by setting the flag VIRTUAL_MEMBER in the **InterfaceMemberFlags** enumeration. If this flag is set, the repository engine returns an error if an attempt is made to access this member.

A COM aggregation object must be used to implement the transient storage.

See Also

IInterfaceMember Flags Property

InterfaceMemberFlags Enumeration

Repository Enumeration Definition

The repository enumeration definition is used to specify a fixed set of constant strings or integer values that correspond to real-world concepts as an enumeration. With the following interfaces you can specify an **EnumerationDef** object and associated **EnumerationValue** objects, and associate these objects with **PropertyDef** objects.

- **IEnumerationDef** interface

  The **IEnumerationDef** interface is the default interface for enumeration objects. Use this interface to define new enumeration values.

- **IEnumerationValueDef** interface

  The **IEnumerationValueDef** interface contains a value that can be stored in the **Property** value of an object.

- **IPropertyDef2** interface

  The **IPropertyDef2** interface has a relationship collection called **EnumerationDef**. It contains an optional relationship to a single **EnumerationDef** object.

The following table identifies enumeration objects that support interfaces.

Objects	**Interfaces**
All **Enumeration** objects	**IenumerationDef**
	IrepositoryObject
	IRepositoryObjectStorage
	IreposTypeInfo
	IVersionAdminInfo2
	InamedObject
	ISummaryInformation
EnumerationValue objects	**IEnumerationValue**
	IrepositoryObject

	IRepositoryObjectStorage
	InamedObject
	ISummaryInformation
	IVersionAdminInfo2

See Also

IEnumerationDef Interface

IEnumerationValueDef Interface

IPropertyDef2 Interface

Meta Data Services Programming

Defining Methods

You can define a method definition object to represent methods in an information model. In addition to specifying methods, you can define parameters on a method and on script objects.

To define a method definition object, you can use the **MethodDef** class or the **IMethodDef** interface for COM programs, or the **MethodDef** object for Automation programs.

The following topics provide more information about defining methods and scripts.

Topic	Description
Defining a Parameter	Explains how to define a parameter
Defining Script Objects	Explains how to define a script object that provides the implementation code for a method

See Also

IMethodDef Interface

MethodDef Class

MethodDef Object

Defining a Parameter

Parameter definitions specify a parameter that is attached to a method. With parameter definitions, you can support an ordered collection of parameters that a method uses.

To define a parameter, use the **IMethodDef** and **IParameterDef** interfaces for COM programs, or use the **MethodDef** or **ParameterDef** objects for Automation programs.

The **IMethodDef** interface provides a way to define an ordered list of parameters for that method. **IMethodDef** is the default interface of the **CMethodDef** object that the **IInterfaceDef::CreateMethodDef** method returns.

The **IParameterDef** interface enables you to define in detail each parameter of a method.

These two interfaces, along with the relationships to other classes and interfaces, are shown in the following figure.

Parameter definitions are stored in a table in the repository database called **RTblParameterDef**.

For more information about model graphs and conventions, see the Microsoft® SQL Server™ 2000 Meta Data Services Software Development Kit (SDK).

See Also

Defining Methods

IMethodDef Interface

IParameterDef Interface

RtblParameterDef SQL Table

Meta Data Services SDK

Defining Script Objects

You can assign Microsoft® ActiveX® scripts to method and property definitions in an information model. The repository engine exposes these methods and properties, and invokes the associated script at run time. You can also use scripts to program transient object collections.

You can create script using Microsoft JScript® and Microsoft Visual Basic® Scripting Edition (VBScript). To define scripts in your information model, use the **IScriptDef** interface for COM programs or the **ScriptDef** object for Automation programs. Only one script definition can be associated for each method or property.

Attaching scripts to properties that are defined as BLOBs (that is, **PropertyDef** objects that have **SQLType** set to SQL_LONGVARBINARY or SQL_LONGVARCHAR) is not supported. Attaching scripts to large property objects does not result in an error or warning; the script is not invoked.

IScriptDef properties do not reside on the **IMethodDef** or **IPropertyDef** interfaces by design. Associating a script at the interface or class level allows you to implement the same method in a variety of contexts.

The following topics provide more information about script implementation.

Topic	Description
Binding Scripts	Explains the binding algorithm that links scripts to specific methods and properties
Accessing a Script	Describes how to access a script
Predefined Script Variables	Describes variables that you can use when creating a script
Method Invocation for Scripted Methods	Describes requirements and considerations for invoking a scripted method
Get Method for Scripted Properties	Describes requirements and considerations for creating the **Get**

	function of a scripted property
Put Method for Scripted Properties	Describes requirements and considerations for creating the **Put** function of a scripted property

See Also

Defining Methods

IMethodDef Interface

IScriptDef Interface

Programming Transient Object Collections

ScriptDef Object

Binding Scripts

ScriptDef objects are bound to method and property definitions through relationships. The repository engine uses an algorithm to support the binding.

To support scripting for both method and property interface members, a **ScriptDef** object is associated at the interface member level. Because method and property definitions inherit from interface member objects, an interface member object provides the common ground where an association between script and interface members can be made.

Because interfaces can be aliased, derived, or otherwise reused, script definitions are linked through association to support the levels of indirection that are customary in COM programming. Associations are established through collections of classes, interfaces, and members that you define for each **ScriptDef** object.

During script invocation, the repository engine reads the collections to select a script definition most closely related to the interface. When the repository engine selects the closest script definition, it determines which method calls the script, on which interface, and on what class. The selection process enables support for two conditions that are common to C++ programming: inheriting a method or property implementation, and overriding a default implementation.

A method or property can be associated with the class and interface being executed, the interface being executed, or the closest ancestor that has the script. If a script cannot be selected, then the repository engine returns an error in the case of methods.

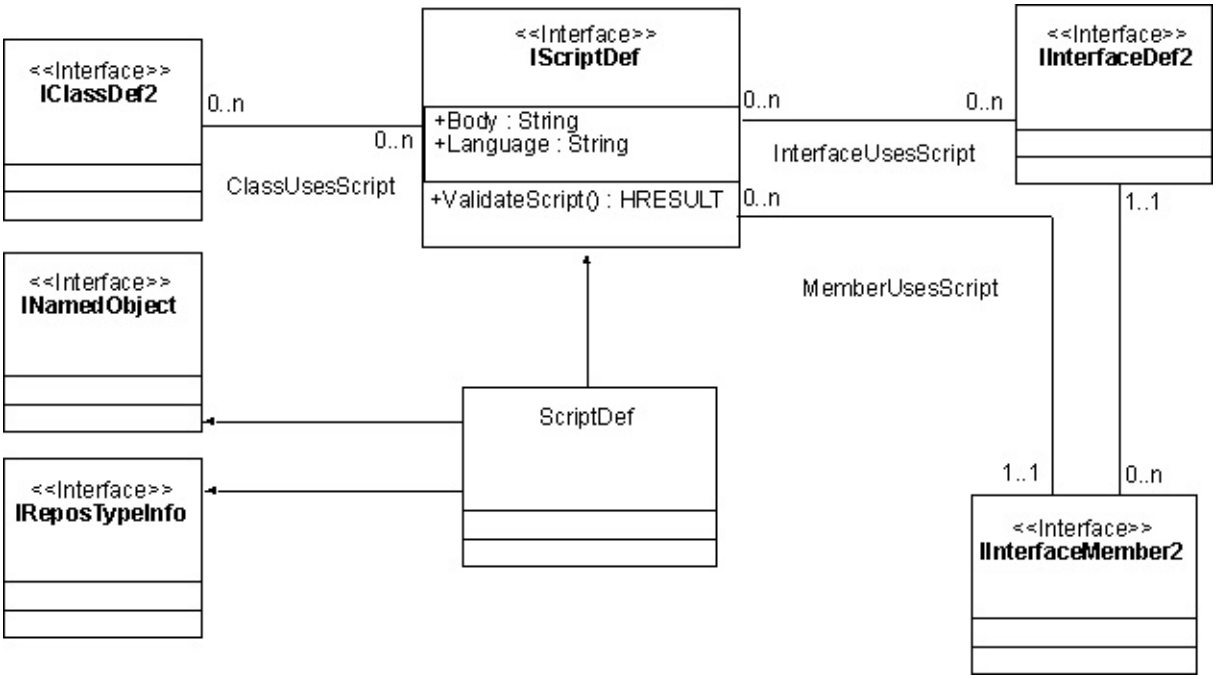
You can implement script for methods and property validation rules that apply to:

- All classes that implement the interface.
- A specific class that implements the interface.

- A derived interface that can override the implementation of a base interface method or property validation rule.

Each method or property can be associated with only one script definition. However, the same script definition can be associated with multiple methods and properties.

The **IScriptDef** interface, along with its relationships to other classes and interfaces, is shown in the following figure.



See Also

Accessing a Script

Defining Script Objects

IScriptDef Interface

ScriptDef Object

Accessing a Script

A script must run within the transaction of the calling program. When scripts encounter an unhandled error or exception, the repository engine reads the error information and populates the repository error queue appropriately. To minimize syntax errors in the script, you can use the **ValidateScript** method to perform a syntax check prior to script invocation.

To execute the script, the repository engine uses the Microsoft® ActiveX® Scripting Engine (VBScript) by default. If you require a more powerful scripting engine, you must instantiate that service from within your script.

There are three different ways to access a script. The first invokes a method; the other two get or put a property.

Invoking a Method

When a script provides the implementation code for a method definition object, you must access the script through method invocation.

When you invoke the method, the repository engine automatically executes the associated script. If there is no associated script, the repository engine returns the error message **E_NOTIMPL**.

Getting and Putting Properties

You can create a script that validates a property before inserting the value of that property into the repository database. In this case, your script (rather than the repository engine) validates the value, gets the value, and puts the value. If you are accessing script to validate a property before storing it in a repository database, you must create **Get** and **Put** methods that are associated with a property.

When you create scripts to retrieve and assign properties, you must always define both **Get** and **Put** operations in the same script.

- When you access any property definition object, the repository engine calls the **Get** portion of the script. You can use a **Get** method with a

property to present properties in your application differently from the way they are stored in the repository engine. If there is no associated script, the property is returned as it is in the repository database.

- When you assign a value to any property definition object, the repository engine executes the **Put** portion of the script. The **Put** portion of the script is used to validate the value. If there is no associated script, the repository engine stores the unvalidated value.

See Also

Defining Script Objects

Get Method for Scripted Properties

Handling Errors

Method Invocation for Scripted Methods

Put Method for Scripted Properties

Predefined Script Variables

The following table lists variables that are predefined for use in scripts. Some variables are initialized as part of the repository session.

Variable	Description
ReposErr	Represents an object that contains two properties: • **ReposErr.Result**, which is an **HRESULT** value that is returned as a result of the **IDispatch::Invoke** call. • **ReposErr.Description**, which is a string that describes the error. This value is guaranteed to exist only for errors generated by the repository or the script engine itself.
CurRepos	Represents the current repository session as an **IDispatch** object.
CurReposODBC	Represents the **IReposODBC** interface on the current repository session.
CurReposObj	Represents a pointer to the **IRepositoryObject2** interface. Use this interface to represent the repository object instance on which the method or validation is being executed.
NestedScripts	Represents a Boolean variable that is stored as a thread-level object. This Boolean variable determines whether nested scripts are called for in the current script. If the user-set Boolean variable does not accept nested scripts, this variable is set to FALSE. After the operation is complete, the system sets it back to TRUE (the default value).

See Also

Accessing a Script

Defining Script Objects

Handling Errors

IRepositoryObject2 Interface

Method Invocation for Scripted Methods

When you provide a script-based implementation for a method, the repository engine selects the script object for the method using a binding algorithm, then invokes the script using the default script engine.

For method invocation to succeed, you must make sure that references in the script correspond to references in the method definition. Specifically, the method name, signature, and returned values that are used to implement the script must be the same as the name and signature of the associated method definition.

When you execute the method, it returns an **HRESULT** value that is copied into the error object. The method invocation returns this value to the caller.

The method can invoke other methods, including itself. You should exercise caution when invoking a method on itself. Doing so may create a recursive condition that can cause a failure in your application.

See Also

Accessing a Script

Defining Script Objects

Get Method for Scripted Properties

Handling Errors

Predefined Script Variables

Put Method for Scripted Properties

Get Method for Scripted Properties

A script-based implementation for a property requires the creation of a **Get** method to retrieve a property value from the repository database.

A **Get** method that you provide substitutes for the get functionality that is typically provided by the repository engine. When your script (rather than the repository engine) provides the implementation, you must handle the retrieval of a property value from the repository database.

For a **Get** method to succeed, the script body must contain a function with the same name as the property, and it must be prefixed with **Get**. For example, if the property name is **ExtendedPrice**, your script must include a function named **GetExtendedPrice**.

When executing the function, the repository engine first performs a lookup to find the property associated with the script. If the property cannot be found, a repository error is returned. Otherwise, the function returns S_OK.

In addition to a **Get** method, you must also define a corresponding **Put** method within the same script. For more information, see Put Method for Scripted Properties.

See Also

Accessing a Script

Defining Script Objects

Handling Errors

IScriptDef Interface

Method Invocation for Scripted Methods

Put Method for Scripted Properties

Put Method for Scripted Properties

A script-based implementation for a property requires the creation of a **Put** method to validate and set a property value in the repository database.

A **Put** method that you provide substitutes for the set functionality that is typically provided by the repository engine. A **Put** method is also the only way to validate a property value prior to saving it in the database. When your script (rather than the repository engine) provides the implementation, you must handle setting and validation of a property value from the repository database.

For a **Put** method to succeed, the script body must contain a function with the same name as the property, and it must be prefixed with **Put**. For example, if the property name is **ExtendedPrice**, your script must include a function named **PutExtendedPrice**.

When the **Put** function is executed, the repository engine returns an error if the new value is invalid. If an error is returned, the repository engine does not store the new value in the repository database. If the function returns S_OK, the value passed to the function is stored in the repository database.

In addition to a **Put** method, you must also define a corresponding **Get** method within the same script. For more information, see Get Method for Scripted Properties.

Validating Multiple Properties Simultaneously

Sometimes two properties are so intertwined that it does not make sense to validate them separately. Instead, you can validate both properties at the same time by following these steps:

1. Set the **Put** method for both properties to return a descriptive error that tells the user the property cannot be set. This step makes the property effectively read-only.

2. Create a method that accepts the values of both properties as parameters. This method validates the property combination. You can

then set each property individually.

See Also

Accessing a Script

Defining Script Objects

Handling Errors

IScriptDef Interface

Method Invocation for Scripted Methods

Predefined Script Variables

Meta Data Services Programming

Defining Inheritance

Inheritance enables you to share and reuse an interface or its members in new ways. The following topics discuss the inheritance techniques that are available.

Topic	Description
Interface Implication	Describes the support of inheritance for interfaces
Member Delegation	Describes the support of inheritance for interface members, specifically relationships
Type Information Aliasing	Describes how you can reuse an object by creating a type information alias

See Also

Defining Information Models

Defining Relationships and Collections

Interface Definition Objects

Interface Implication

Interface implication enables a client application to define a correspondence between two interfaces in an information model such that all of the members on one interface are available to members of another interface. Interface implication offers some of the functionality of multiple inheritance, which is not allowed in COM.

Interface implication supports information model definitions of the form **Interface-I1-implies Interface I2**, which means that any class that implements **I1** also implements **I2**. Consequently, if **I1** is added to the list of implemented interfaces on a class, **I2** will be added to the list automatically. The engine supports implication for such classes, whether the interfaces exist at the time of the implication definition, or are installed into the repository at a later time.

Extending an Information Model Using Interface Implication

Interface implication facilitates information model extension. By using interface implication, you can define a new interface and require that all new and existing classes support it. Interface implication eliminates the need to write a custom procedure that updates existing classes so that they support the new interface.

For example, consider the two interfaces **IA** and **IB** shown in the following figure. Suppose that all classes implementing **IA** now need to implement **IB** as well. By using interface implication, you can define **IA-implies-IB**, as shown in the following figure. This ensures that any class that implements **IA**, such as **C**, will also implement **IB**, even if class **C** is installed after the implication has been defined.

Note In previous versions of the repository engine, interface implication was accomplished only by using the Model Development Kit (MDK). With this release, this restriction no longer applies.

For more information about creating information models by using the MDK, see the Microsoft® SQL Server™ 2000 Meta Data Services Software Development Kit (SDK).

See Also

Adding an Interface Implication

Defining Inheritance

Simulating Multiple Inheritance

Adding an Interface Implication

Interface implication is defined through **IInterfaceDef2**. This interface supports two collections that determine implication: **Implies** and **ImpliedBy**. These collections allow you to define both directions of an implication.

During a commit of any transaction that includes an interface that has an **Implies** collection attached to it, the repository engine adds all implied interfaces to an existing class and then recalculates the class. The result of the recalculation is the same as if the class implemented the implied interface directly.

See Also

Defining Inheritance

Interface Implication

IInterfaceDef2 Interface

Member Delegation

Member delegation supports the assignment of members on one interface to base members on another interface. Delegation can be used to support relationship inheritance.

The following topics provide more information.

Topics	Description
Derived Members	Describes derived members and strategies for using derived members in your information model
Derived Member Requirements	Explains the conditions and requirements that support member derivation
Creating a Derived Member	Describes how to create a derived member and how to add a derived member to an existing class
Derivation Behavior	Explains how the repository engine stores, retrieves, and updates derived members
Example: Basic Member Delegation	Provides sample code that illustrates member delegation
Example: Member Delegation with Filtering	Provides sample code that illustrates filtering on derived collections

See Also

Defining Inheritance

Interface Implication

IInterfaceDef2 Interface

Derived Members

Derived members is a capability that can be used to delegate the implementation of members of one interface to members of another interface, where both interfaces are implemented by the same class.

Aliasing is a simplified form of member delegation, where a member of one interface is derived from a member of another without modifying its underlying semantics. Through aliasing, you can overlap functionality for multiple interfaces.

For example, when interfaces evolve, you can rename properties and methods, place them on different interfaces, and still maintain the naming scheme of the original interface. Similarly, aliasing provides a way to flatten multiple interfaces into a single interface that contains members from all of them. The advantage to flattening a set of interfaces is that it simplifies navigation. Also, aliasing simulates multiple inheritance.

Note You can define aliases for type information elements other than members. For more information about type information aliasing, see Type Information Aliasing.

The following topics discuss how member derivation aliasing enables these scenarios.

Topic	Description
Supporting Multiple Interfaces With Overlapping Functionality	Describes how you can reuse interface definitions through derived members.
Flattening Interfaces	Describes how you can combine interface members into one interface to simplify navigation.
Simulating Multiple Inheritance	Explains how you can simulate multiple inheritance using derived members.

A semantically richer variant of derived members allows a collection on one

interface to be derived from a collection on another interface while, at the same time, filtering out some of the base collection members.

The following topics discuss how member derivation enables these scenarios.

Topic	Description
Specializing Relationship Collections	Describes how you can create special-purpose collections that are based on a general-purpose collection.
Filtering Derived Collections	Describes how you can apply filtering techniques to a derived collection.

See Also

Creating a Derived Member

Defining Inheritance

Derived Member Requirements

Supporting Multiple Interfaces With Overlapping Functionality

As an information model changes to accommodate new functionality, it is common to create a new interface by evolving an existing interface. In this situation, the two interfaces (the old and new versions) have overlapping functionality. The two interfaces can exist together when the old version of the interface must still be supported. In this case, properties can be renamed and placed on different interfaces while keeping the underlying semantics synchronized between the interfaces.

The following graph shows an object exposing the two interfaces:

I1: An interface with a base member **M1**.

I2: An interface with a derived member **M2**.

By using member delegation from **I2** to **I1**, the user can either call the base member (that is, **I1::M1**), or call the derived member (that is, **I2::M2**), which will be delegated to **I1::M1**.

For more information about other ways of combining interface members, see Flattening Interfaces and Simulating Multiple Inheritance.

See Also

Creating a Derived Member

Derived Members

Interface Implication

Flattening Interfaces

You can use derived members to flatten a set of interfaces of a class into a single interface. In this case, the new interface contains all of the combined members of the flattened interfaces. This simplifies the use of the class, because your application does not need to navigate between interfaces of the class.

In the following figure, an object exposes two interfaces, **I1** and **I2**, whose members are **M1** and **M2**. By delegation, the two interfaces could be flattened into one interface **I3** that contains the derived members **Md1** and **Md2**. In this case the call **I3::Md1** will be mapped to **I1::M1**, and the call **I3:Md2** will be mapped to **I2::M2**.

For more information about other ways of combining interface members, see Supporting Multiple Interfaces With Overlapping Functionality and Simulating Multiple Inheritance.

See Also

Creating a Derived Member

Derived Members

Interface Implication

Simulating Multiple Inheritance

In COM, multiple inheritance between interfaces is not supported. However, by using the derived members capability, multiple inheritance can be simulated.

For example, the following figure shows an interface **IA** that inherits from **IB**, and implies **IC** (meaning that any class that supports **IA** must also support **IC**). According to COM, **IA** cannot inherit from **IC** because it already inherits from **IB**. However, with delegation, the members of **IC** could be made available on **IA**. This is not inheritance, because **IC** members are not explicitly mapped into **IA**. Nevertheless, the result is the same because **IA** now includes members of both **IB** and **IC**.

For more information about other ways of combining interface members, see Flattening Interfaces and Supporting Multiple Interfaces With Overlapping Functionality.

See Also

Creating a Derived Member

Derived Members

Interface Implication

Specializing Relationship Collections

Using derived members makes it possible to create a subset or specialize a relationship collection.

For example, in the following figure, **Vehicle** is related to **Engine** through the relationship v*ehicle has engine*. Because the **Motor Vehicle** uses an **Internal Combustion Engine**, it requires specializing the general relationship *vehicle has engine* to *motorVehicle has internalCombustionEngine*. The last relationship is specialized in **Truck** to *truck has dieselEngine*.

For more information about other ways of specializing a collection, see Filtering Derived Collections.

See Also

Creating a Derived Member

Derived Members

Filtering Derived Collections

By using filtering, it is possible to derive specific collections from a general collection. Filtered collections apply to inherited interfaces.

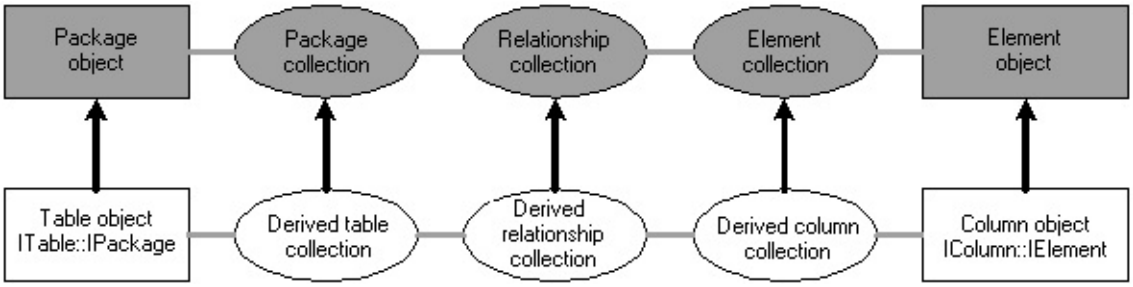
The following example illustrates the basic concept of filtering. In the figure, **IDoc1** interface has the **Elements** collection definition, which contains figures and text. You may find it useful to access only the text or only the figures. With the derivation mechanism that uses filtering, the **IDoc2** interface can have two collection definitions, **Figures** and **Text**. The first contains only figures, and the second contains only text.

Architecture of Filtered Derived Collections

To create a filtered derived collection, you must set up parallel collections that correspond to the base collections. You must define a derived origin collection for the base origin collection, a derived destination collection for the base destination collection, and a derived relationship collection for the base relationship collection. The following example provides an illustration.

The following figure shows two base objects and the collections that relate them.

In the next figure, **Table** object inherits from **Package** object and **Column** object inherits from **Element** object. Derived collections include the **Table** collection, the **Column** collection, and the **Relationship** collection that joins them.



In this example and in all cases where filtering applies, the derived collection is a subset of the base collection. The **Table** collection is made up of a subset of the items in the **Package** collection. The derived relationship matches the items in the **Table** subset with the items in the **Column** subset.

For more information about other ways of specializing a collection, see Specializing Relationship Collections.

See Also

Derived Members

Example: Member Delegation with Filtering

Filtering Collections

Derived Member Requirements

Before you define a derived member, verify that conditions supporting the implementation are in place.

General Requirements

The following requirements apply to all derived members.

- A class that supports an interface with a derived member must also support the interface on which the corresponding base member is defined. In other words, the interface with the derived member must either inherit or imply the interface with the base member.

- A derived member can be on the same or a different interface as its base member.

- A derived member can be derived from another derived member. An interface member must ultimately derive from a member that is not derived. In other words, cyclic derivations are not allowed.

- Derived members can be defined for any interface that is an instance of **InterfaceDef**, including built-in repository engine interfaces, such as **IRepositoryObject**.

Derived Property Definition Requirements

For property definitions, storage data types and lengths of derived properties must be the same as those of the base property.

Derived Collection Definition Requirements

Because a derived collection must map to a base collection, the derived collection and base collection must have correspondent characteristics.

The following requirements apply to derived collections.

Collection Type

A derived origin collection must map to a base origin collection, and a derived destination collection must map to a base destination collection.

Relationship Type

A derived collection can be connected by way of a relationship only to another derived collection. However, a derived collection cannot be connected to a base (stored) collection.

A derived collection definition must be defined in the same transaction as the collection from which it was derived, on the same relationship.

Two derived relationships can specialize the same base relationship and have their collections on the same pair of interfaces. However, because only the generalized relationship is stored in the relationship table (**RTblRelships**), instances of the two specialized relationships are indistinguishable.

Naming

A derived collection must be identical to the base collection with regard to naming characteristics. If the base collection specifies unique naming, the derived collection must also contain uniquely named items. Furthermore, if you add items to a base collection by way of the derived collection, you must verify that the items you add do not break the unique naming constraints of the base collection. For more information, see Naming and Unique-Naming Collections.

Sequencing

A derived collection must be identical to the base collection with regard to sequencing characteristics. Inserting an item into a derived sequenced collection inserts the new relationship into the base collection immediately after its predecessor in the derived collection. Also, moving an item in the derived collection moves the item in the base collection immediately after its predecessor in the derived collection. For more information, see Sequenced Collections.

Delete Propagation

The delete propagation semantics of a derived collection must be the same as the base collection. For more information, see Propagating Deletes.

Version Propagation

The version propagation semantics of a derived collection must be the same as the base collection. For more information, see Propagating Versions.

See Also

Creating a Derived Member

Defining Inheritance

Example: Basic Member Delegation

Example: Member Delegation with Filtering

IInterfaceMember2 interface

InterfaceMemberFlags Enumeration

Creating a Derived Member

Interface members are either base members or derived members. A derived member is mapped to another interface base member through a relationship. Member derivation supports mapping of the form **MemberB Is-Derived-From MemberA**, which means that **MemberA** provides implementation for **MemberB**.

How to Define a Derived Member

Before you can define a derived member, you must verify that the interface that includes the derived member and the interface that contributes the base member are implemented by the same class. For more information about conditions and constraints that apply to derived member definition, see Derived Member Requirements.

To define a derived member, use the **IInterfaceDef2::CreateAlias** method to create an alias that represents the derived member. Aliases are created from the interface on which you add the derived member.

If you use **CreateAlias**, the derived member is automatically mapped to the base member providing implementation details. Mapping is achieved by adding the derived member and the base member to collections. The base member is added to the **ServicedByBaseMember** (the origin) collection, and the alias to the **ServicesDerivedMembers** (the destination) collection. The two collections are the two sides of the **BaseMemberServicesDerivedMembers** relationship class. **IInterfaceMember2** provides these collections.

After you create a derived member, you can add a property definition object to the derived member to enhance its definition.

Adding a Derived Member to an Existing Class

You can add an interface containing derived members to an existing class. No modification of the instances is required as long as both the derived members and the existing instances of the class have valid data for the properties and collections.

However, in one case some existing instances of the class may be undesirable, although they're technically valid. In this case, an interface with a derived collection, for example, may be added to a class that already has instances, and the base member may be read-only. This means that new relationships on instances of this class can be added to the derived collection but not (directly) to the base collection. Thus, all new relationships in the base collection will conform to the definition of the derived collection. However, at the time the derived collection definition was added, there may have been existing instances of the class with relationships on the base collection that do not conform to the derived collection.

Updating a Derived Member

You can define whether a derived member can be updated. Update capability is enabled by default. To prohibit updating, set the INTERFACEMEMBER_READONLY flag to TRUE. For more information, see Derivation Behavior and InterfaceMemberFlags Enumeration.

See Also

Defining Inheritance

Example: Basic Member Delegation

Example: Member Delegation with Filtering

IInterfaceDef2::CreateAlias

IInterfaceMember2 interface

Derivation Behavior

The following are detailed rules for derivations that apply to storage, retrievals, updating collections, and adding derived members to an existing class.

Storage

A property or relationship is always stored by the repository engine on the base interface. That is, there are no instances of derived relationships in **RTblRelships** and there are no columns allocated for derived properties in the repository SQL table of their interface.

Retrievals

When a derived collection is referenced, the repository engine materializes the derived collection by applying a filter to the base collection. For each instance in the base collection, the engine determines whether the target object supports the target interface of the derived collection. The effect for a relationship collection is that all instances are visible at the general level in the base collection, and subsets of the generalized relationship instance collection are visible at the more specialized levels in the derived collections.

Updates to Collections

Use **IInterfaceMember::Flags** to determine whether a derived or base member is updateable.

Add, remove, insert, and move methods on the derived collection are delegated to the corresponding operation on the corresponding item in the base collection. An insert or move method on a sequenced collection places the item relative to the derived collection. For more information about sequencing, see Derived Member Requirements.

The count, enumeration, and type methods on a derived collection are specific to that collection.

See Also

Creating a Derived Member

Defining Inheritance

Example: Basic Member Delegation

Example: Member Delegation with Filtering

IInterfaceMember2 interface

InterfaceMemberFlags Enumeration

Example: Basic Member Delegation

This example includes sample code for creating a derived property and a derived collection. This example illustrates how to create a new interface and define the derived property. Creating a relationship with a predefined base property declares that this property is derived. Similarly, this example also illustrates how to define a derived collection. The same procedure is used for method definitions. There is no change in the programming logic of setting and getting properties or manipulating collections.

The following table identifies the Repository Type Information (RTIM) objects and the corresponding pointers that appear in the sample code.

RTIM object	Pointer
IinterfaceDef	***pINewIface**;
IclassDef	***pIClassDef**;
IpropertyDef	***pIBaseProp**, ***pIDerivedProp**;
IrelationshipDef	***pINewRelshipDef**;
IrelshipColDef	***pIBaseCol**, ***pIDerivedCol**;
IreposTypeLib	***pITypeLib**;

In order to run this sample, you must create a type library and a class definition for a new interface. Also, the collection **pIBaseCol** (a collection that is the same type as the one being delegated) and the property **pIBaseProp** (a property that is the same type as the one being delegated) must have been defined earlier. The pointers **pIBaseCol** and **pIBaseProp** are assumed to have been already set before running this example.

```
// Create a new interface:
pIClassDef->CreateInterfaceDef(CRepVariant(OBJID_INewOrgIface)
CVariant("INewIface"), CRepVariant(IID_INewIface), pIIReposDispat
CVariant("Default"), &pINewIface);

// Create an alias property:
```

```
pINewIface->CreateAlias(CRepVariant(OBJID_ALongDerived),
CVariant("ALongDerived"), DISPID_ALongDerived, pIBaseProp,
&pIDerivedProp);

// Create an alias collection:
pINewIface->CreateAlias(CRepVariant(OBJID_CollectionDerived),
CVariant("CollectionDerived"), DISPID_CollectionDerived,
pIBaseCol, &pIDerivedCol);
```

See Also

Creating a Derived Member

Defining Inheritance

Example: Member Delegation with Filtering

Example: Member Delegation with Filtering

The repository stores information that determines whether a derived collection can be filtered for objects that support a certain target interface.

To filter a derived collection, you must create two derived collection definitions and a new relationship to connect them. One derived collection contains the target objects of interest (that is, the set of target objects, minus those that do not match your filter criteria). The second derived collection contains the origin object. You need a derived origin collection whenever you want to create a relationship that includes a derived destination collection. The new relationship type is used to match the collections.

Note If the derived collections were connected by the relationship type as the base collections, there would be two collections on each side of the relationship type, and the matching of origin and destination collections would not be well defined.

The following table identifies the Repository Type Information (RTIM) objects that are used to create derived collections and shows the corresponding pointers that appear in the example code.

RTIM object	Pointer
IInterfaceDef	***pINewOrgIface**, ***pIDestIface**, ***pIIReposDispatch**;
IClassDef	***pIClassDef**;
IPropertyDef	***pIBaseProp**, ***pIDerivedProp**;
IRelationshipDef	***pINewRelshipDef**
IRelshipColDef	***pIBaseOrgCol**, ***pIDerivedOrgCol**;
IRelshipColDef	***pIBaseDstCol**, ***pIDerivedDstCol**;
IReposTypeLib	***pITypeLib**;

In order to run this sample, you must create a type library and a class definition for a new interface. Also, the interface **pIDestIface** must exist, as well as a relationship from this interface with two collections, **pIBaseOrgCol** and **pIBaseDstCol**. The pointers **pIDestIface**, **pIBaseOrgCol**, and **pIBaseDstCol**

are assumed to have been already set before running this example.

```
// Create interfaces for a given class:
pIClassDef->CreateInterfaceDef(CRepVariant(OBJID_INewOrgIface)
CVariant("INewOrgIface"), CRepVariant(IID_INewOrgIface),
IIReposDispatch, CVariant("Default"), &pINewOrgIface);

// Create a new relationship type:
/* Notice that CVariant is a wrapper of the VARIANT class defined
   in the header file "oleutil.h" */
pITypeLib->CreateRelationshipDef(CRepVariant(OBJID_NULL),
CVariant("A_Relationship"), &pINewRelshipDef);

// Create an origin collection definition:
pINewOrgIface->CreateRelationshipColDef(CRepVariant(OBJID_Me
CVariant("Members"), DISPID_Members, TRUE, COLLECTION_NA
pINewRelshipDef, &pIDerivedOrgCol);

// Get the ServicedBy collection and add the base origin collection:
pIDerivedOrgCol->Interface("IInterfaceMember2")
                    .ServicedBy.Add(pIBaseOrgCol);

// Create the destination collection:
pIDestIface->CreateRelationshipColDef(CRepVariant(OBJID_NULL)
CVariant("Parent"), DISPID_Parent, FALSE, NULL, pINewRelshipDe
pIDerivedDstCol);
// Get the ServicedBy collection and add the base destination collectio
pIDerivedDstCol->Interface("IInterfaceMember2")
                    .ServicedBy.Add(pIBaseDstCol);
```

In this example code, the derived origin collection will filter the objects that support the **IDestIface** interface. Note that a new relationship type is defined. The relationship instances, however, will not use this relationship type. All of the collections will continue to use the base relationship type instead. The new

relationship type will be used to identify the matching collections in **RTblRelColDefs**.

See Also

Creating a Derived Member

Defining Inheritance

Example: Basic Member Delegation

Filtering Derived Collections

Meta Data Services Programming

Generating Views

A repository database stores classes, properties, and relationships in a table structure that does not reflect the composition of an information model. While this arrangement is optimal for the repository engine, it can be difficult to work with if you want to query the database.

To simplify querying, you can generate SQL views of your repository database that correspond to an information model. An SQL view provides a mechanism for gathering elements from the repository tables and assembling them into a virtual table that resembles a specific class, interface, or relationship in your information model. Generated views simplify database queries by eliminating the need to understand the underlying structure of a repository database. In addition, views allow you to represent any relationship, including a many-to-many relationship, as a junction table view, which is something you cannot specify in an information model. Views also provide a way to represent each many-to-one relationship as a foreign key.

View Types

You can define three kinds of SQL views for each class, interface, and relationship. For more information about each view type, see Defining Views in an Information Model and Kinds of SQL Views.

Performance varies for each kind of view. For more information about how to improve view performance, see View Hints.

How to Generate Views

View generation requires Microsoft® SQL Server™ 2000 and repository engine 3.0. The 3.0 repository database format provides storage for the view definitions you add to an information model. For more information about upgrading to a 3.0 database, see Upgrading and Migrating a Repository Database.

View generation is performed by the repository engine in response to flags that you set in the information model. If, while creating a model, the repository engine finds one of these flags set to True, it generates an SQL view from your view definitions.

By default, view generation flags are set to False. To generate a SQL view after an information model is installed, you can write code that sets the flags to True. You can also set the flags in an information model, then reinstall it.

The repository engine synchronizes your generated views with subsequent changes you make to an information model (for example, adding an interface to a class or adding an interface implication). As long as view generation flags remain set to True, synchronization occurs automatically.

Storing SQL Views

View definitions are stored in the repository SQL tables. Class view definitions are stored in the **RTblClassDefs** table. Interface view definitions are stored in the **RTblIfaceDefs** table. Junction table view definitions are stored in the **RTblRelshipDefs** table.

Generated views are stored by your DBMS in the same catalog and schema that contains your information model.

See Also

Repository Databases

RTblClassDefs SQL Tables

RTblIfaceDefs SQL Tables

RTblRelshipDefs SQL Tables

Defining Views in an Information Model

Before you can generate a view, you must add view definitions to your information model using **IViewClassDef**, **IViewInterfaceDef**, or **IViewRelationshipDef**.

You must set at least one of the following flags to cause view generation: GENERATE_RESOLVED_VIEW, GENERATE_NORESOLUTION_VIEW, or GENERATE_WORKSPACE_VIEW. Each one of these flags generates a different kind of view.

You can add view definitions using the Model Development Kit (MDK) or the repository API. The following topics describe the kinds of SQL views you can generate.

Topic	Description
Kinds of SQL Views	Describes the scope and attributes of generated views.
Defining a Class View	Provides information about SQL views that are based on a class.
Defining an Interface View	Provides information about SQL views that are based on an interface.
Defining a Junction Table View	Provides information about SQL views that are based on a relationship class.
Defining View Columns	Provides information about customizing a view column.

MDC OIM SQL Views

The Meta Data Coalition (MDC) Open Information Model (OIM) is distributed with predefined SQL views that are ready to use. These views are added to the views folder of your Microsoft® SQL Server™ 2000 repository database when the MDC OIM is installed.

See Also

Kinds of SQL Views

View definitions can be defined for a shared repository and for workspaces. Flags that you specify on each interface determine the number and type of views that are generated and whether implied interfaces are included in the view.

For each class, interface, and relationship class, you can define three kinds of views.

Kind of view	Description
Workspace	A view that is scoped by a workspace, so it includes only objects that are contained by the workspace. This view is defined when you set GENERATE_WORKSPACE_VIEW.
Version Resolved	A view that supports version resolution, so it includes only the latest version of each object in the shared repository. This view is defined when you set the GENERATE_RESOLVED_VIEW. For more information, see Version Resolution for Generated Views.
Unresolved	A view that does not support resolution, to be used only when the repository is not versioned or when you know that version information does not exist or is unimportant. This view is defined when you set the GENERATE_NORESOLUTION_VIEW. Running an unresolved view against a versioned repository returns multiple entries (that is, all versions of all meta data instances are returned).

Note These views are not mutually exclusive; you can create all three, depending on your requirements.

Choosing one kind of view over another can have an effect on performance. For more information, see View Hints.

See Also

Defining a Class View

Defining an Interface View

Defining a Junction Table View

Generating Views

Versioning Objects

Workspace Management Overview

Defining a Class View

You can direct the repository engine to create a class-oriented view for a class definition object. The generated view includes all the properties (one column for each property) of every interface that is implemented by the class, including those that are implied and inherited. It also includes all many-to-one relationships on those interfaces, representing each one as a foreign key.

When defining a class view, you should verify that the combination of interfaces does not produce a duplicate column name (for example, a **Name** property on two separate interfaces). To ensure that column names are unique, you can create a view column name. For more information, see IViewPropertyDef Interface, Defining View Columns, and Naming Conventions for Generated Views.

To create a class view, use **IViewClassDef** in a way that is similar to the following example:

```
Dim oTypeLib as ReposTypeLib
Dim oTable as ClassDef
Dim oViewTable as IViewClassDef
set oTable = oTypeLib.CreateClassDef(objid_null, oTypeLib_name, oT
set oViewTable=oTable
// Generate a workspace-scoped view by specifying bit=4
oViewTable.flags=4
```

For more information about properties and flags that you can specify, see Defining Views in an Information Model and IViewClassDef Interface.

See Also

Defining a Junction Table View

Defining an Interface View

RTblClassDefs SQL Tables

Defining an Interface View

You can direct the repository engine to create an interface-oriented view for an interface definition object. The generated view includes all properties, including those that are available by way of inheritance, implication, or derivation. It also includes all many-to-one relationships on those interfaces, representing each one as a foreign key.

Interface views are useful for abstract interfaces that are further specialized by other interfaces or that are implemented by different classes. For example, suppose that **IStudent** and **IEmployee** both inherit from **IPerson**. Assume that **IStudent** and **IEmployee** are implemented by classes, but that no class exists for **IPerson**. By generating a view for **IPerson**, you can include instances of both **IStudent** and **IEmployee** using one view.

To create an interface view, use **IViewInterfaceDef**. For more information about properties and flags that you can specify, see Defining Views in an Information Model and IViewInterfaceDef Interface.

See Also

Defining a Class View

Defining a Junction Table View

RTblIfaceDefs SQL Tables

Defining a Junction Table View

You can direct the repository engine to create a junction-table view for a relationship class. This is the only way to represent a many-to-many relationship. You can also create a junction-table view for a many-to-one relationship, although you can express this kind of relationship using a foreign key instead.

For more information, see IViewRelationshipDef Interface.

See Also

Defining a Class View

Defining an Interface View

Defining Views in an Information Model

IReposQuery Interface

RTblRelshipDefs SQL Tables

Defining View Columns

Column names are generated from property interface members. There is one column for each property.

You can create custom names for view columns. Creating custom names can eliminate duplicate names in cases where columns from multiple interfaces are defined in the same view. For more information about defining column names, see IViewPropertyDef Interface.

Using Prefixes to Distinguish Between Names

When necessary, the repository engine adds prefixes to disambiguate identical names. Duplicate names are most likely to occur when you generate a class view on a class that implements interfaces that contain members of the same name.

To resolve duplicate column names, the repository engine attaches the interface name as a prefix to each occurrence of the column name.

For example, consider the following two interface members, **ISpellchecker::Name** and **IThesaurus::name**. When creating a view that combines both members, one item will be called **ISpellchecker.Name** and the item will be called **IThesaurus.name**.

The combined name of base, interface prefix, and view type prefix cannot exceed the maximum length of 118 characters for view columns. For more information, see Naming Conventions for Generated Views.

See Also

Defining a Class View

Defining a Junction-Table View

Defining an Interface View

Version Resolution for Generated Views

Version resolution determines which version of the repository object is included in a view when there are multiple versions to choose from. Version resolution does not apply to Workspace views. A Workspace view contains whatever version of the object is included in the workspace.

Version Resolution Strategy

When generating a view, the repository engine selects the last version of every object. Depending on the combination of versions and relationships that exist, this strategy can exclude some versions from the view, even if they are related to a version that is in the view.

For example, suppose you define a view that contains version one of ObjectA (**VersionA1**), which is related to version one of ObjectB (**VersionB1**). If **VersionB1** has a successor, **VersionB2**, the repository engine selects **VersionA1** and **VersionB2** for the view. Because there is no relationship between **VersionA1** and **VersionB2**, the generated view does not reflect the relationship.

Because view definitions resolve to a single version, version identifiers are not usually included in a view. However, views that use **IRepositoryODBC::ExecuteQuery** to run directly against the underlying database system may need the **VersionID** column for a subsequent **GetObject** operation. To handle this case, you can include the **VersionID** column in the view. This column indicates which version of an object is selected by version resolution. You can include the **VersionID** column by setting the USE_VERSIONID_COLUMN flag on the view definition interface.

See Also

Defining Views in an Information Model

Filtering Collections

IRepositoryODBC Interface

IReposQuery Interface

Naming Conventions for Generated Views

When you generate views, you can either specify a name through your application code or you can allow the repository engine to specify a default view name. View names must conform to certain guidelines.

View names can have a maximum length of 128 characters. View column names can have a maximum length of 118 characters.

SQL keywords are allowed for view names and view column names.

Note If a view name exceeds 128 characters and it is not prefixed, the view name will be truncated to 128 characters.

Name Composition

View names are composite names, formed from a base view name and a prefix.

The base view name can be provided by your application code and stored in a repository database. If no view name is provided, a default base name is created from the name of the class, interface, or relationship on which the view is based. Default names are not stored in a repository database.

If you generate two or more kinds of view, the repository engine adds a prefix to distinguish the kind of view (class, interface, or junction-table) and whether it is resolved.

Prefix Composition

Prefixes are six characters in length, composed of view type and view resolution indicators. Prefixes are added only when it is necessary to distinguish between kinds of views. If you generate one view for each class, interface, or relationship, no prefix is created. If you generate two views for each class, interface, or relationship, one of the views will be prefixed. If you generate three views, two of the views will be prefixed.

Prefixes are assigned based on priority. The following table indicates how priority rotates depending on the kinds of views that are generated. A Workspace

view is always priority 1 if it is generated, and it is never prefixed. If a Workspace view is not generated, priority 1 shifts to Version Resolved. If neither a Workspace view nor a Version Resolved view is generated, priority 1 shifts to Version Unresolved.

Workspace	Version Resolved	Version Unresolved
Priority 1 (not prefixed)	Priority 2 (prefixed)	Priority 3 (prefixed)
Priority 1 (not prefixed)		Priority 2 (prefixed)
	Priority 1 (not prefixed)	Priority 2 (prefixed)
		Priority 1 (not prefixed)

The following table describes each component of the prefix.

Prefix component	Description
RVw	Indicates a repository view.
C, I, or J	Indicates a class view, interface view, or junction-table view.
R or N	Indicates whether a view is resolved (R) for multiple versions, or not resolved (N).
_	Separates the prefix from the base name.

View Name Example

The following table shows the combined base name and prefix for each view type, for each kind of view you can generate.

View type	Workspace	Version Resolved	Version Unresol
Class	MyClassViewName	RVwCR_MyClassViewName	RVwCN_MyCla
Interface	MyIFaceViewName	RVwIR_MyIFaceViewName	RVwIN_MyIFac
Junction-Table	MyRelshipViewName	RVwJR_MyRelshipViewName	RVwJN_MyRels

See Also

Generating Views

Defining Views in an Information Model

RTblIfaceDefs SQL Tables

RTblRelshipDefs SQL Tables

Querying a Repository Database Using SQL Views

To query a database using a view, you can use the **IRepositoryODBC::ExecuteQuery** method or you can run queries using SQL commands. The following example illustrates one approach. Other query syntax is required for querying Workspace views.

```
dim oRepos as Repository
dim ODBC as IRepositoryODBC
set ODBC=oRepos
ODBC.ExcecuteQuery("select intID from cTable where TableName="
```

The addition of a SQL view makes it easier to build select statements that correspond to your information model.

To perform a query, the object that you specify must implement **IRepositoryODBC**, which provides the **ExecuteQuery** method.

Instead of referencing the generated view in your SELECT statement, you specify the class, interface, or relationship object for which you defined a view definition and generated the corresponding view. In this case, **cTable** is a class that has a corresponding **IViewClassDef** defined for it, and a generated view that is available to it. Microsoft® SQL Server™ 2000 uses the generated SQL view transparently to process the query. You do not need to specify the name of the view when creating the query.

Note Collection filters can also be used to facilitate querying. For more information, see Filtering Collections.

Querying Workspace Views

Workspace views are created as user-defined functions. The name of the user-defined function is the view name that you specify. When querying a workspace, you must pass in the workspace **IntID**.

The following example illustrates one way to query a Workspace view. In this example, MyUDF is the view name.

```
dim oRepos as Repository
dim ODBC as IRepositoryODBC
set ODBC=oRepos
ODBC.ExcecuteQuery("select * from MyUDF(IntID)")
```

See Also

Generating Views

IReposQuery Interface

Meta Data Services Programming

Installing Information Models

Installing a model is the process of defining an information model in a repository database. When you install an information model, the repository engine adds entries to the repository SQL schema. After a model is installed, it is available for tools and applications. You can program against an installed information model using the repository API.

Before you can install a model, you must compile it using the Modeling Development Kit (MDK) in the Microsoft® SQL Server™ 2000 Meta Data Services Software Development Kit (SDK). Compiled models have an .rdm file extension.

Note If you are using Open Information Model (OIM) models, Meta Data Services distributes ready-to-install .rdm files for each model.

The following table lists four approaches to installing a model.

Installation approach	Description
Meta Data Browser	Provides a dialog box so that you can browse to the file you want. For more information, see Using Meta Data Browser.
Command line utility	Enables you to run a model installation from a command line. For more information, see Using the Model Installer from the Command Line.
Microsoft® ActiveX® component	Enables you to install an information model from your application code using a program that Meta Data Services provides. For more information, see Using the Model Installer ActiveX Component.
Repository API	Enables you to install an information model programmatically using methods. For more information, see IManageReposTypeLib::CreateTypeLib and ReposRoot CreateTypeLib Method.

See Also

Information Models

Information Model Fundamentals

Meta Data Services Programming

Using the Model Installer from the Command Line

The model installer can be executed through the command-line utility Insrepim.exe. By default, this utility is located in the folder C:\Program Files\Common Files\Microsoft Shared\Repository. It reads an .rdm file produced by or distributed with the Microsoft® SQL Server™ 2000 Meta Data Services Software Development Kit (SDK). From the .rdm file, the utility extracts meta data from the information model and stores it in a repository database. You use command-line arguments to specify the repository database.

Syntax

InsRepIM.exe /f[Model File] /r[Repository Connection String] /u[User

Parameters

Value	Description
[Model File]	The compiled information model (.rdm) file
[Repository Connection String]	The repository database file, either a Data Source Name (DSN) or database and authentication information
[User]	The user name
[Password]	The user password

See Also

Connecting to and Configuring a Repository

Meta Data Services SDK

Meta Data Services Programming

Using the Model Installer ActiveX Component

The file Insrepim.dll is a Microsoft® ActiveX® DLL located in the folder C:\Program Files\Common Files\Microsoft Shared\Repository. It can be used from either a Microsoft Visual Basic® application or a Microsoft Visual C++® application to programmatically install a model file into a repository database.

The component supports the following method:

HRESULT InstallRDM(

BSTR *Connect*,

BSTR *RdmFile*,

BSTR *UserName*,

BSTR *Password*

);

Parameters

Connect
[in]
The repository connection string used to access the database server that hosts the repository database.

RdmFile
[in]
The compiled information model (.rdm) file.

UserName
[in]
The user's name.

Password
[in]
The user's password.

Return Value

S_OK
The method completed successfully.

Error Code
The method failed to complete successfully.

See Also

Connecting to and Configuring a Repository

Repository Errors (Alphabetical Order)

Meta Data Services Programming

Programming Information Models

After you define and install an information model in a Microsoft® SQL Server™ 2000 Meta Data Services repository, you can use the object definitions in the repository in your application code.

The following topics provide information about accessing and manipulating objects in a repository.

Topic	Description
Navigating a Repository	Describes how to access objects through collection navigation, and identifies the repository API objects that perform object manipulation
Versioning Objects	Explains how to version objects, manipulate object versions, and merge versions
Programming BLOBs and Large Text Fields	Describes programming support for binary large objects (BLOBs) and large text fields
Programming Transient Object Collections	Explains how to program a transient object collection
Managing Transactions and Threads	Describes how to set up a transaction, and how the repository engine processes a transaction in single and multiple threads
Managing Workspaces	Explains how to set up and manage a workspace, and how to manipulate workspace contents
Handling Errors	Describes how to handle errors
Optimizing Repository Performance	Describes optimization techniques that you can use to improve repository performance

See Also

Meta Data Services Programming

Navigating a Repository

Because the objects represented in a Microsoft® SQL Server™ 2000 Meta Data Services repository are connected through relationships, you can navigate from one object to any related object. Relationships are implemented through collections. To navigate, you must traverse a collection.

This section describes the process of retrieving objects related to a given object. This section requires an understanding of the information presented in the Repository Type Information Model (RTIM).

The following topics describe how to navigate a repository.

Topic	Description
Navigation Overview	Provides basic information about navigation elements and strategies
Accessing a Repository	Identifies the methods you can use to create or open a repository database, handle errors, and set up a transaction
Accessing Repository Objects	Identifies the methods you can use to manipulate a repository object
Accessing Properties	Identifies interfaces you can use to acquire information about an object
Accessing Relationships	Identifies the methods you can use to manipulate a relationship
Accessing Relationship Collections	Identifies the methods you can use to manipulate a relationship collection
Accessing Target Object Collections	Identifies the methods you can use to manipulate a target object collection
Selecting Items in a Collection	Explains how to select items in a collection and how to work with enumerated items in a collection
Propagating Deletes	Describes how deleting one object can cause the automatic deletion of subsequent objects

See Also

Repository Object Architecture

Repository Type Information Model

Navigation Overview

The information in a Microsoft® SQL Server™ 2000 Meta Data Services repository is a network of objects and relationships. You navigate through this network using collections. Depending on your objective, you can retrieve an object collection, a relationship collection, or some other special-purpose collection.

Typically, you navigate to an object because you want to manipulate it. Manipulations include retrieving or setting a property, deleting an object, deleting a relationship between two objects, or adding a relationship between two objects.

The following figure represents a relationship between a **Table** origin object and a **Column** destination object. The relationship type is *table has columns*. In this relationship, one table can have many columns.

Subsequent topics explore the navigable relationships and strategies that you can implement based on this single origin-destination pairing of **Table** to **Column**.

The following topics provide this information.

Topic	Description
Navigating a Relationship from Two Directions	Describes how you can navigate a relationship from either object in the relationship
Navigating a Relationship Using Two Approaches	Describes alternate approaches for navigating

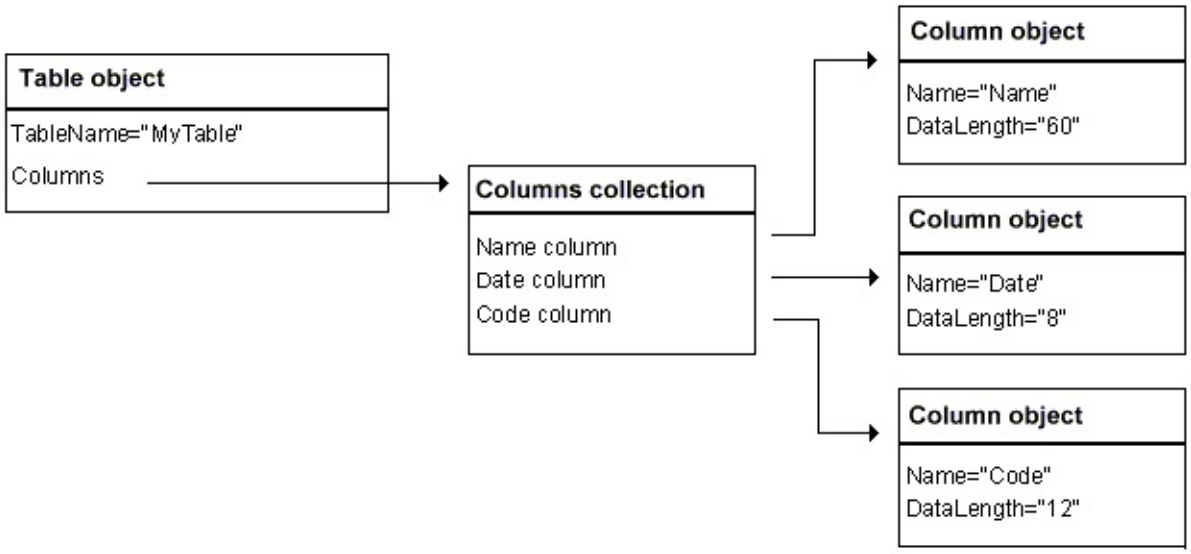
See Also

Navigating a Repository

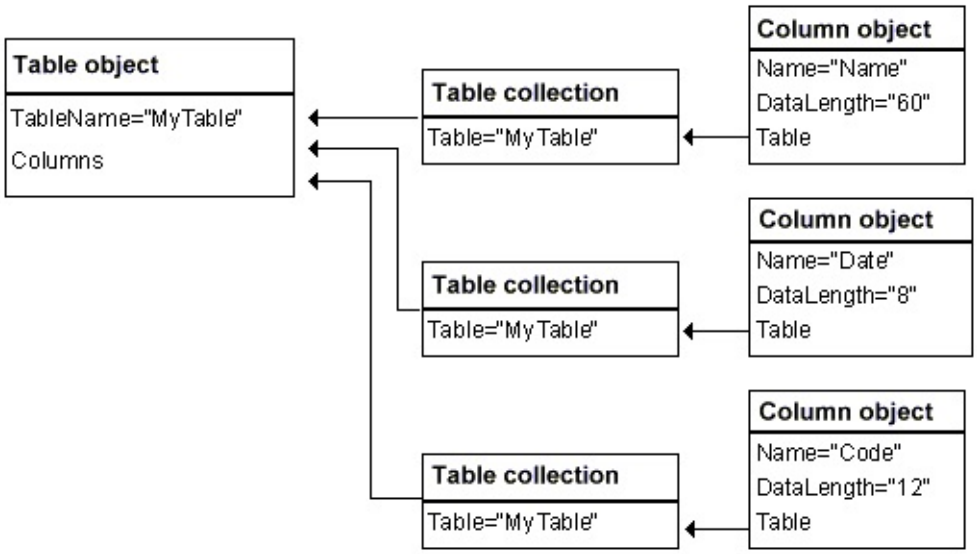
Navigating a Relationship from Two Directions

You can navigate a relationship from the direction of either the origin or destination object. As such, there are always two ways to navigate every relationship.

In the following example, you can navigate from **Table** to **Column**, or you can navigate from **Column** to **Table**. This figure shows the direction of navigation from the **Table** object to a **Column** object, using a **Columns** collection. When navigating from a **Table** object to a **Column** object, you use a **Columns** collection. The navigation and subsequent manipulation is always through the collection. For example, from the **Columns** collection, you can select a column, retrieve or set a column property, add or delete a new column, and so on.



In the next figure, the direction of the navigation is from a **Column** object to the **Table** object. Each **Column** object accesses a separate **Table** collection to navigate back to the **Table**. The same origin-destination relationship between the **Table** origin object and the **Column** destination object supports this alternate approach to navigation. Typically, a relationship from a destination object to an origin object is called a reverse relationship. Furthermore, you can treat this relationship as a separate entity (for example, as a *column containedBy table* relationship).



See Also

Navigating a Relationship Using Two Approaches

Navigating a Repository

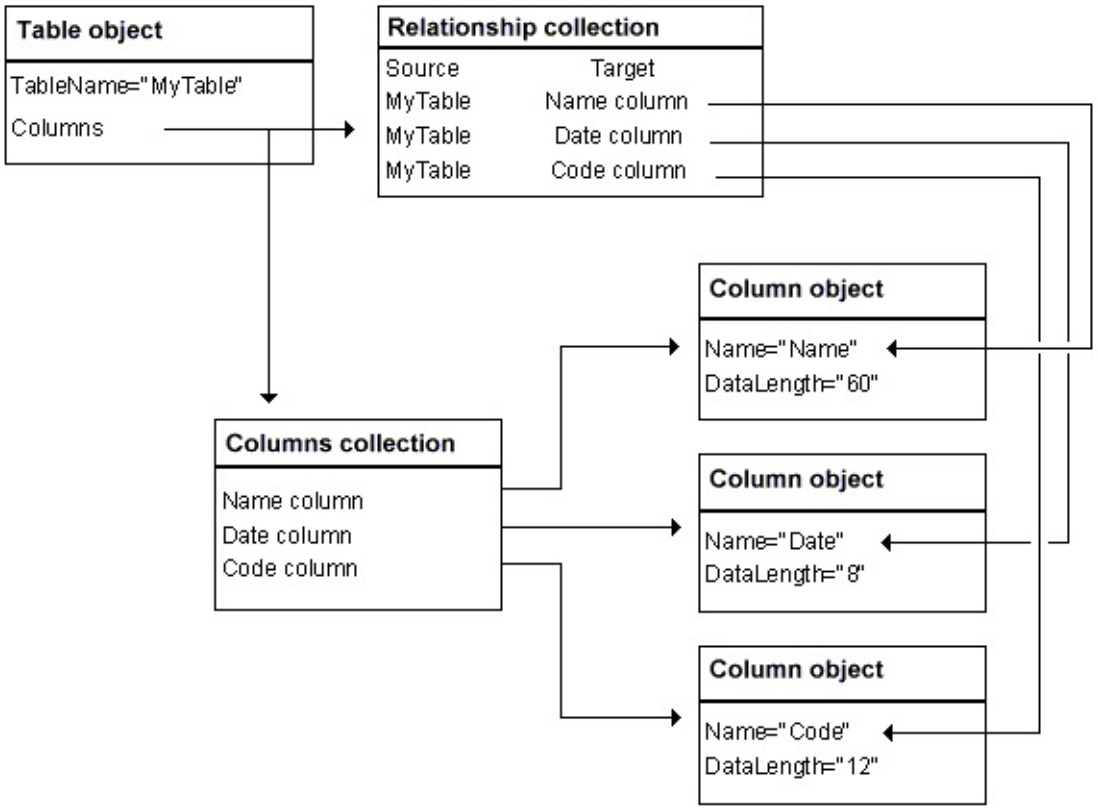
Navigation Overview

Navigating a Relationship Using Two Approaches

After you know the direction of navigation that you want to follow, you can choose between two different approaches to implement the navigation.

You can navigate directly through a relationship collection using the **IRelationshipCol** interface, or you can use a specialized collection that is designed to simplify navigation. This specialized collection is a target object collection. You instantiate an object collection using the **ITargetObjCol** interface.

The following figure shows two different approaches for navigating the same path. In this example, the navigation moves from the **Table** object to a **Column** object.



In this figure, you can navigate through the **Relationship** collection (a two-step approach) or through the **Columns** collection (a one-step approach).

Instantiating the **Columns** collection through **ITargetObjectCol** makes this collection a target object collection.

See Also

Navigating a Relationship from Two Directions

Navigating a Repository

Navigation Overview

Source Objects and Target Objects

A relationship collection associates a source object with one or more target objects. The objects are attached to the relationship collection through interfaces.

The words *source* and *target* convey the potential for browsing. Typically, the source object of a relationship is the object for which you retrieve a collection. The target object of a relationship is the object that can become the target of a step that proceeds from the source object, through the relationship, to the target object.

Source-and-target terminology differs from origin-and-destination terminology. The origin and destination do not change based upon which collection is being used to access a relationship. For example, suppose **Table** is an origin object and **Column** is a destination object. When navigation proceeds from **Table** to **Column**, **Table** is the source object. When navigation proceeds in the reverse direction, **Column** is the source object.

See Also

Navigating a Repository

Accessing a Repository

Access to repository methods and properties is supported at both the COM and Automation level.

For more information about creating and opening a repository database, see Connecting to and Configuring a Repository.

To	Use
Create a new repository database and open it	The **Create** method of the **Repository** object, or use **IRepository::Create**.
Open an existing repository database	The **Open** method of the **Repository** object, or use **IRepository::Open**.
Retrieve repository error information	The standard **Err** Automation object, or use the methods of **IRepositoryErrorQueueHandler**, **IRepositoryErrorQueue**, and **IEnumRepositoryErrors**.
Manage transactions in the repository	The methods of the **RepositoryTransaction** object or the **IRepositoryTransaction** interface. To retrieve an interface pointer to the **IRepositoryTransaction** interface, use the **Transaction** property of the **IRepository** interface.

See Also

Handling Errors

IRepository Interface

IRepositoryTransaction Interface

Repository Class

Repository Databases

Accessing Repository Objects

Access to repository objects is supported at both the COM and Automation level, as shown in the following table.

To	Use
Create an object	The **CreateObject** method of the **IRepository** interface.
Retrieve an object	The **Object** property of the **IRepository** interface.
Delete an object	The **Delete** method of the **IRepositoryItem** interface.
Obtain the value of an early-bound object property	(Automation) The syntax *variable = object.property*, as for any Automation object property. (COM) The **get_PropertyName** method of the COM interface to which the property is attached, where **PropertyName** is the name of the property to be retrieved. You can also use the late-bound property access method.
Obtain the value of a late-bound object property (that is, the object class is not in an available type library)	(Automation) The syntax *variable = object.property*, as for any Automation object property. (COM) The standard Automation method-invocation technique to invoke the **get_PropertyName** method, where **PropertyName** is the name of the property to be retrieved. Because all Microsoft® SQL Server™ 2000 Meta Data Services interfaces that expose late-bound properties are indirectly derived from the **IDispatch** interface, the **GetIDsOfNames** and **Invoke** methods are available for use.
Set the value of an	(Automation) The syntax *object.property = value*,

early-bound object property	as for any Automation object property. (COM) The **put_PropertyName** method of the COM interface to which the property is attached, where **PropertyName** is the name of the property to be set. You can also use the late-bound property access method.
Set the value of a late-bound object property (that is, the object class is not in an available type library)	(Automation) The syntax *object.property = value*, as for any Automation object property. (COM) The standard Automation method-invocation technique to invoke the **put_PropertyName** method, where **PropertyName** is the name of the property to be set. Because all Meta Data Services interfaces that expose late-bound properties are indirectly derived from the **IDispatch** interface, the **GetIDsOfNames** and **Invoke** methods are available for use.
Retrieve the object identifier of an object	The **ObjectID** property of the **IRepositoryObject** interface.
Retrieve the version identifier of an object	The **VersionID** property of the **IRepositoryVersion** interface.
Set the object identifier of an object	The object identifier, which is passed as a parameter when creating the object.
Retrieve the internal identifier of an object	The **InternalID** property of the **IRepositoryObject** interface.

See Also

Accessing Properties

Navigating a Repository

Versioning Objects

Accessing Properties

Repository properties store state information about an object. You can access a repository property to acquire information about an unknown object, and then use the property data that is returned to you to acquire more specific information.

The following interfaces are available for manipulating properties: **IReposProperty**, **IReposProperty2**, **IReposPropertyLarge**, and **IReposProperties**.

To	Use
Retrieve generic data about an unknown object	The **IReposProperty** property, which exposes methods that enable you to retrieve generic information about an object. You can determine an object type (for example, whether it is a class or interface), retrieve an object identifier, or set and retrieve property values.
Retrieve additional data about a specific object	The **IReposProperty2** property, which exposes additional data about an object, such as whether it is a base member or a derived member, whether it is read-only or an origin collection, and so on. You can use this interface to retrieve meta data about the interface without incurring an additional round trip to the database. You can also use this interface to get the **PropertyDef** object that represents the property.
Retrieve all properties of an interface	The **IReposProperties** property, which exposes all properties of a particular interface as a single collection.
Retrieve all properties of a class	The **IRepositoryObject2** property, which exposes a **Properties** collection that you can use to access all properties of all interfaces that are implemented by a class.
Retrieve, set, and navigate a property	The **IReposPropertyLarge** property, which provides methods for manipulating binary large

object that exceeds 64 kilobytes (KB)	objects (BLOBs) and large text fields that exceed 64 KB. For more information, see Programming BLOBs and Large Text Fields.

See Also

Accessing Repository Objects

IReposProperty Interface

IReposProperty2 Interface

IReposPropertyLarge Interface

IReposProperties Interface

Accessing Relationships

Access to relationships is supported at both the COM and Automation level.

To	Use
Create a relationship	The **Add** method of the **IRelationshipCol** interface.
Delete a relationship	The **Remove** method of the **IRelationshipCol** interface.
Retrieve a relationship	(Automation) The syntax **relationshipCollection***(index)*, as for retrieving an item from any Automation collection. (COM) The **get_Item** method of the **IRelationshipCol** interface, specifying the relationship to be retrieved.
Retrieve an object that participates in a given relationship	The **Origin** or **Destination** property of the **IRelationship** interface.

See Also

Navigating a Repository

Accessing Relationship Collections

Access to relationship collections is supported at both the COM and Automation level.

You cannot add or delete a relationship collection after it is created. Sometimes a relationship collection contains no relationships, but the collection still exists. If you retrieve such an empty relationship collection, the repository engine returns an interface pointer to a relationship collection, just as it would for any other relationship collection.

Loading object instance collections can be asynchronous. The calling thread should check to determine whether the load is complete. If the calling thread tries to read data, refresh the collection, or construct an enumerator while loading is in progress, it will be blocked until the load is complete.

Note You can use **IRepositoryObject2** to access specific collections, even if the collections share the same name through an inherited interface. For more information, see IRepositoryObject2 Interface.

To	Use
Add a relationship to a relationship collection	The **Add** method of the **IRelationshipCol** interface.
Remove a relationship from a relationship collection	The **Remove** method of the **IRelationshipCol** interface.
Enumerate the relationships within a relationship collection	(Automation) The syntax **relationshipCollection***(index)*, as for retrieving an item from any Automation collection. (COM) The **get_Count** and **get_Item** methods of the **IRelationshipCol** interface. You can also use the **_NewEnum** method of the **IRelationshipCol** interface to obtain a standard enumerator interface for the collection.

See Also

Navigating a Repository

Retrieving Relationship Collections

Relationship collections can be retrieved at both the COM and Automation level.

Retrieving Relationship Collections Using Automation Interfaces

To retrieve a relationship, first declare a variable as a **RelationshipCol** object (that is, an object that implements the **IRelationshipCol** interface). Next, set the variable equal to the object member that is the collection you want to retrieve.

For example, the following Microsoft® Visual Basic® code retrieves a collection of bug discoveries (a relationship collection) belonging to a particular person into the variable called **DiscoveryCollection**:

```
DIM DiscoveryCollection As RelationshipCol
DIM Person As RepositoryObjectVersion
REM Retrieve the source object into the Person variable
Set DiscoveryCollection = Person.bugs
```

In this example, **DiscoveryCollection**(1) refers to the first relationship in the collection.

Retrieving Relationship Collections Using COM Interfaces

To retrieve a relationship, you can perform the following steps:

- Call the **Invoke** method of the interface to which the collection is attached.

- Pass in DISPATCH_PROPERTYGET to specify that this is a property-get operation.

- Pass in the dispatch identifier for the collection to be retrieved.

  The **Invoke** method will pass an **IDispatch** interface pointer for the

collection back to you.

- Invoke the **QueryInterface** method of the **IDispatch** interface to retrieve an **IRelationshipCol** interface pointer for the relationship collection.

See Also

Navigating a Repository

Accessing Target Object Collections

Access to target object collections is supported at both the COM and Automation level.

Note You can use **IRepositoryObject2** to access specific collections, even if the collections share the same name through an inherited interface. For more information, see IRepositoryObject2 Interface.

To	Use
Include an object in a collection	The **Add** method of the **ITargetObjectCol** interface.
Exclude an object from a collection	The **Remove** method of the **ITargetObjectCol** interface.
Enumerate the objects within a target object collection	(Automation) The syntax **targetObjectCollection***(index)*, as for retrieving an item from any Automation collection. (COM) The **get_Count** and **get_Item** methods of the **ITargetObjectCol** interface, or use the **_NewEnum** method of the **ITargetObjectCol** interface to obtain a standard enumerator interface for the collection.

See Also

Navigating a Repository

Using TargetObjectCol with Relationship Collections

The **TargetObjectCol** class provides a convenient way to manipulate relationship collections. When you manipulate a target object collection, you actually manipulate the corresponding relationship collection.

For example, suppose you are manipulating the target object collection describing the employees managed by Frank, shown in the following figure.

Now suppose you want to add Louise, an existing employee, to the collection. To do so, use the **Add** method, extending the collection to look like the collection in the following figure.

When you call the **Add** method, the repository engine actually adds a relationship to the corresponding relationship collection. Before adding the new relationship, the relationship collection appeared as follows.

Using the **Add** method changed the relationship collection to the following.

See Also

Navigating a Repository

Retrieving Target Object Collections

Object collections can be retrieved at both the COM and Automation level.

Retrieving Object Collections Using Automation Interfaces

To retrieve a relationship, first declare a variable as an object that implements the **ITargetObjectCol** interface. Next, set the variable equal to the object member that is the collection you want to retrieve.

Note All collections in a Microsoft® SQL Server™ 2000 Meta Data Services repository are attached to repository objects as members.

For example, the following Microsoft Visual Basic® code retrieves a collection of bugs (a target object collection) discovered by a particular person into the variable called BugCollection:

```
DIM BugCollection As ITargetObjectCol
DIM Person As RepositoryObject
REM Retrieve the source object into the Person variable
Set BugCollection = Person.bugs
```

In this example, **BugCollection**(1) can refer to the first object in the collection.

Retrieving Relationship Collections Using COM Interfaces

To retrieve a relationship, you can perform the following steps:

1. Call the **Invoke** method of the interface to which the collection is attached.

2. Pass in DISPATCH_PROPERTYGET to specify that this is a property-get operation.

3. Pass in the dispatch identifier for the collection to be retrieved.

4. The **Invoke** method will pass an **IDispatch** interface pointer for the collection back to you.

5. Invoke the **QueryInterface** method of the **IDispatch** interface to retrieve an **ITargetObjectCol** interface pointer for the collection of target objects.

See Also

Navigating a Repository

Selecting Items in a Collection

You can select the items of a collection by index, by sequence, by enumerator, or by name. If you get a collection for an origin object, or plan to move from an origin object to a destination object, all of these selection options are available. If you get a collection for a destination object, or plan to move from a destination object to an origin object, you have fewer selection options. The availability of a selection option is described in each approach.

You can select collection items in the following ways:

- By index. Choose the *n*th item in the collection, where *n* is a number between one and the size of the collection.

  COM: Use the **get_Item** method of the **IRelationshipCol** interface or the **ITargetObjectCol** interface.

  Automation: Use the syntax collection(index), as for any Automation collection.

- By sequence. Use the same programming statements to select by sequence as you use to select by index. The distinction between the two depends only on how the repository orders the items in the collection. In most cases, the repository uses an arbitrary order. But if the collection is a sequenced collection, the repository orders the collection items according to the defined sequence.

  This approach is valid when the origin object and the source object are the same. You cannot select by sequence from a collection belonging to a destination object, because the repository does not sequence collections belonging to destination objects.

- With an enumerator. At the COM level you can get an enumerator object for a collection, and then use the standard enumerator functions (**Next**, **Skip**, **Reset**, and **Clone**). Use the **_NewEnum** method of the **IRelationshipCol** interface or the **ITargetObjectCol** interface to obtain an interface pointer to an enumerator object.

For more information, see Using Enumerators to Work with Items in a Collection.

- By name. If the collection is a collection of names, you can select the item whose name matches the name you supply.

  COM: Use the **get_Item** method of the **IRelationshipCol** interface or the **ITargetObjectCol** interface.

  Automation: Use the syntax myCollection("name").

  Note You cannot select by name from a collection belonging to a destination object, because a Microsoft® SQL Server™ 2000 Meta Data Services repository does not support the naming of origin objects.

  If a name within the collection of names is not unique, the repository will return the first item that it finds with the specified name.

See Also

Navigating a Repository

Using Enumerators to Work with Items in a Collection

You can use enumerators to select items from a collection. When the repository engine establishes an enumerator for you, it reads the repository database to determine which items should appear in the list, and the order in which those items should appear. The enumerator does not, however, contain repository items. Rather, each element of an enumerator identifies a repository item. To retrieve a particular item identified by the next element of an enumerator, use the **Next** method of the enumerator interface.

The elements in an enumerator identify the items in a collection as described by the repository database when the enumerator was instantiated. After you instantiate the enumerator, the collection in the database can change. Specifically, you change a collection in these ways:

- Add an item.

  Your enumerator will not refer to the newly added item. To see the new item, you must instantiate the enumerator again.

- Remove an item.

  Your enumerator will continue to refer to the deleted item. That is, your enumerator will retain an element that refers to the deleted item by its internal identifier. When you call the **Next** method to retrieve the item, the method returns an error.

- Reorder the collection.

  Your enumerator will reflect the old order. To see the new order, you must reinstantiate the enumerator.

See Also

Navigating a Repository

Selecting Items in a Collection

Filtering Collections

You can filter collections to determine which items to include in a collection based on criteria you provide. Filters can be used to define queries, or to work with a subset of all available items that match criteria you define.

Filters that you create can be applied to target object collections, relationship collections, object instance collections, workspaces, and to the repository as a whole. You can also create filters on derived collections.

To create a filter, use the **IReposQuery** interface. **IReposQuery** is implemented by the **Repository** class, the **Workspace** class, and the **RelationshipCol** class.

To create your criteria, you must create a filter in the form of a SQL WHERE clause. You can then attach this filter as a parameter to the **IReposQuery::GetCollection** method.

Filtering only occurs at run time. Filter definitions are not stored in a repository database.

See Also

Filtering Derived Collections

IReposQuery::GetCollection

Navigating a Repository

RelationshipCol Class

Repository Class

Workspace Class

Propagating Deletes

When you delete an object version or relationship, the repository engine can sometimes automatically delete other object versions and their attendant relationships. The automatic removal of an object version is called a propagated deletion. The process by which the repository engine first determines which propagated deletions are necessary and then performs those propagated deletions is called delete propagation.

Delete propagation is very useful for removing orphan objects that are no longer associated with other objects or collections in your information model. Delete propagation does not occur by default. The repository engine performs propagated deletions only when you remove a relationship whose corresponding origin collection type has the COLLECTION_PROPAGATEDELETE flag set. Such relationships are called delete-propagating relationships. This flag must be set in the information model on the collection.

A single delete propagation can result in the removal of many object versions. There are several reasons for this:

- A delete-propagating relationship can have a **TargetVersions** collection containing many items. As a result, deleting the relationship causes the deletion of all objects in the **TargetVersions** collection.
- An object version that you delete can have many delete-propagating origin relationships.
- An object version to be removed automatically (that is, by propagated deletion) can itself have delete-propagating origin relationships.

The following table provides specific topics for each of the actions that trigger delete propagation.

Topic	Description
Delete Propagation After	Removing an origin relationship that

Removing an Origin Relationship	has the COLLECTION_PROPAGATEDELETE set causes delete propagation.
Delete Propagation After Removing a Destination Relationship	Removing a destination relationship that has the COLLECTION_PROPAGATEDELETE set causes delete propagation.
Delete Propagation After Removing a Destination Target Version	Removing an item from the **TargetVersions** collection of an origin relationship that has the COLLECTION_PROPAGATEDELETE set causes delete propagation.
Delete Propagation After Removing an Origin Target Version	Removing an item from the **TargetVersions** collection of a destination relationship that has the COLLECTION_PROPAGATEDELETE set causes delete propagation.
Delete Propagation After Removing an Object Version	Removing an object version that has origin relationships that have the COLLECTION_PROPAGATEDELETE set causes delete propagation.

See Also

CollectionDefFlags Enumeration

Navigating a Repository

Requirements for Changing an Object Version

Requirements for Object-Version Deletion

Requirements for Object-Version Deletion

The following restrictions apply to object-version deletion:

- If the object version has any successor, it cannot be deleted.

- If the object version is a member of a **TargetVersions** collection of an origin relationship, and that relationship's source object version is unchangeable, it cannot be deleted. For more information, see Requirements for Changing an Object Version.

If a to-be-deleted object version does not satisfy these requirements, the repository engine does not necessarily return an error. If you are explicitly deleting the object version with the **Delete** method, the method fails and returns an error. However, if the repository engine is automatically attempting to delete the object version during delete propagation, it does not return an error. Instead, the engine continues to evaluate other object versions as candidates for propagated deletions.

See Also

Navigating a Repository

Propagating Deletes

Requirements for Changing an Object Version

An object version is unchangeable if it is frozen or if it is checked out to a workspace and the attempt to change it does not occur within the context of that workspace.

Note This restriction applies when the repository engine automatically attempts to change an object version for you. The repository engine can automatically change an object during delete propagation. This occurs when a propagated deletion of a destination object version reduces the **TargetVersions** collection of a corresponding origin object version's origin relationship. In effect, the origin object version has been modified automatically by the repository engine.

This restriction also applies when you attempt explicitly to modify an object, for example, by setting one of its properties.

See Also

Navigating a Repository

Propagating Deletes

Requirements for Object-Version Deletion

Delete Propagation After Removing an Origin Relationship

If you delete a delete-propagating origin relationship, or if the repository engine automatically removes one after deleting its attendant origin object version, delete propagation can occur. The repository engine considers performing a propagated deletion on each destination version of the relationship (that is, the repository engine considers performing a propagated deletion on each object version from the **TargetVersions** collection of the deleted origin relationship).

The repository engine considers deleting the target versions in reverse order of their creation (not in the reverse order of their inclusion in the **TargetVersions** collection). In effect, the repository engine works backward through the version graph, attempting to delete leaf nodes before attempting to delete their predecessors.

The repository engine performs a propagated deletion on an object version only if the object version satisfies the requirements for object-version deletion. If the object version does not satisfy the requirements, the repository engine does not perform the propagated deletion on that object version.

Even if the repository engine encounters a candidate for propagated deletion that does not satisfy the requirements for object-version deletion, it continues to evaluate the other candidates. Thus, the entire delete propagation operation can result in the deletion of some of the **TargetVersions**, but not others.

See Also

Propagating Deletes

Requirements for Object-Version Deletion

Version Graph

Delete Propagation After Removing a Destination Relationship

Deleting a destination relationship is similar to removing an item from the **TargetVersions** collection of a destination relationship. Thus, the delete propagation that occurs after such a deletion is equal to the delete propagation occurring after such a removal from a **TargetVersions** collection. For more information, see Delete Propagation After Removing an Origin Target Version.

See Also

Propagating Deletes

Delete Propagation After Removing a Destination Target Version

If you remove an object version from the **TargetVersions** collection of a delete-propagating origin relationship, the repository engine considers performing a propagated deletion on that object version. The repository engine performs a propagated deletion on the destination object version if both of the following conditions hold:

- The destination object version has no other destination relationship of the same type as the deleted relationship.

- The source object satisfies the basic requirements for object-version deletion. For more information, see Requirements for Object-Version Deletion.

See Also

Propagating Deletes

Delete Propagation After Removing an Origin Target Version

If you remove an object version from the **TargetVersions** collection of a delete-propagating destination relationship, the repository engine considers performing a propagated deletion.

Delete propagation always occurs from the origin object toward a destination object. Thus, in this situation, the repository engine considers performing a propagated deletion on the object version that was the source of the relationship whose **TargetVersions** collection you modified. The repository engine performs a propagated deletion on the source object version if all of the following conditions hold:

- The item you removed was the last item in its **TargetVersions** collection.
- The source object version has no other destination relationship of the same type as the destination relationship whose **TargetVersions** collection you modified.
- The source object satisfies the basic requirements for object-version deletion. For more information, see Requirements for Object-Version Deletion.

See Also

Propagating Deletes

Delete Propagation After Removing an Object Version

You can explicitly delete an object version using the **Delete** method. Similarly, the repository engine can automatically delete an object version by performing a propagated deletion operation on it. In either case, the object version is deleted only if it satisfies the basic requirements for object-version deletion.

If an object version you are trying explicitly to delete does not satisfy these requirements, the **Delete** method returns an error. If an object version that the repository engine is trying to delete through propagation does not satisfy these requirements, the repository engine does not return an error. Instead, it continues with the delete propagation operation. That is, the repository engine continues to consider performing propagated deletion operations on other object versions.

Whether an explicit deletion or a propagated deletion is attempted, the repository engine deletes the object version and any of its relationships if the object version satisfies the requirements for object-version deletion.

Note Some of these deleted relationships can be delete-propagating origin relationships. The repository engine considers performing one or more propagated deletions for each. For more information, see Delete Propagation After Removing an Origin Relationship.

See Also

Propagating Deletes

Requirements for Object-Version Deletion

Meta Data Services Programming

Versioning Objects

Information models that you create for use with Microsoft® SQL Server™ 2000 Meta Data Services contain instance data relevant to the tools and applications you build and support. As you continue to develop and maintain these software tools, this instance data is accessed and modified. The ability to view past versions of this instance data can be useful. For example, you can use this information to:

- Reproduce old versions of a software component.
- Analyze differences between two versions of a software component.
- Determine how the relationships between various software components have changed from one release of a software tool to the next.

Meta Data Services maintains past versions of your instance data. These past versions are accessible through version management and workspace management interfaces.

The following topics describe the version management capabilities of Meta Data Services.

Topic	Description
Versioning Overview	Explains basic concepts of object and collection versioning.
Manipulating Object Versions	Explains how you can manipulate an object version programmatically, including how to create, propagate, and freeze object versions.
Manipulating Versioned Relationships	Explains how you can manipulate versioned relationships programmatically.
Resolution Strategy for Objects and Object Versions	Explains how to select an object. You can select a specific version, or allow the repository engine to select an object for

	you.
Version Graph	Describes the version graph and explains how to navigate a network of versioned objects.
Merging Object Versions	Explains how to merge multiple object versions together.

See Also

IRepositoryObjectVersion Interface

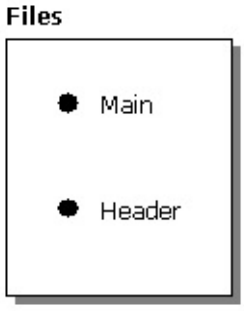
RepositoryObjectVersion Object

Versioning Overview

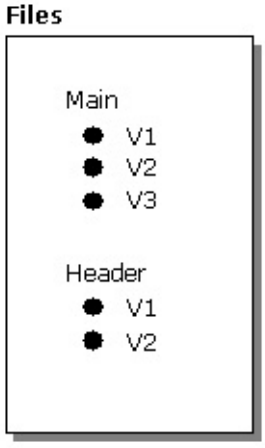
Versioning provides a way to define and redefine objects at specific points in time. This topic uses an example to explain versioning behavior in Microsoft® SQL Server™ 2000 Meta Data Services.

How Versioning Works in Meta Data Services

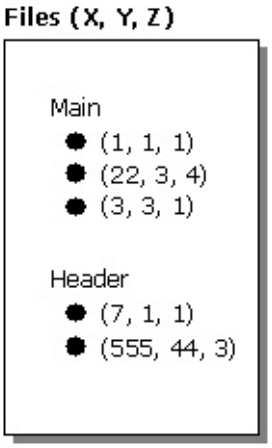
Objects in a Meta Data Services repository conform to classes. The following figure shows two objects that conform to the **File** class.



An individual object can change whenever any of its property values or collections change. Sometimes the new property values or collections simply replace the preceding ones. Other times, you may want to retain both the old values and the new values. The repository engine can retain the old property values and collections with an old version of the object. The following figure shows three versions of the Main file and two versions of the Header file.

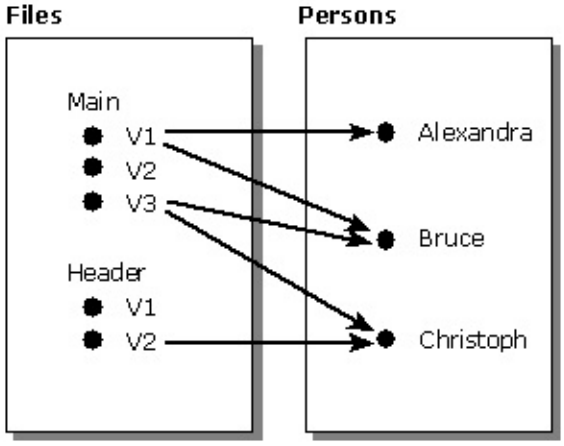


The different versions of an object can have different property values. The following figure includes property values for the properties X, Y, and Z, properties that the **File** class exposes through its various interfaces. In the figure, individual versions have different values for the properties. The picture shows the property values as ordered X-Y-Z triplets. "(22,3,4)" means X=22, Y=3, and Z=4.



Note how this works: the repository engine does not store multiple values for a particular property of an object. Instead, it stores multiple versions of an object such that each individual version can contain its own individual property values.

The different versions of an object can also have different collections. The following figure shows one collection (of the *Persons-of-File* collection type) for each of the five object versions of the **File** class. (To save space, the X-Y-Z triplets are not shown.)

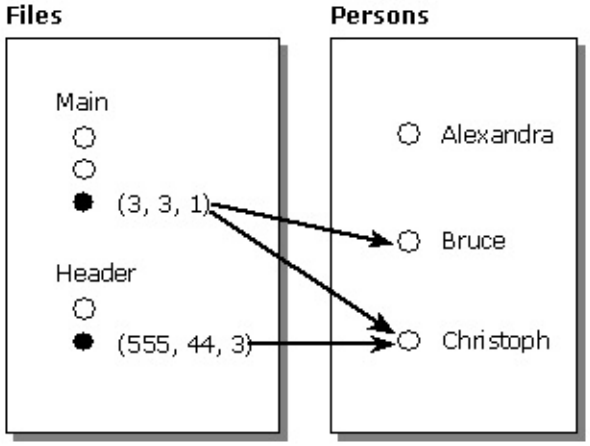


Here are the collections of the five object versions of the **File** class:

- *Persons-of-Version 1-of-Main*: {Alexandra, Bruce}
- *Persons-of-Version 2-of-Main*: { } (The empty set)
- *Persons-of-Version 3-of-Main*: {Bruce, Christoph}
- *Persons-of-Version 1-of-Header*: { } (The empty set)
- *Persons-of-Version 2-of-Header*: {Christoph}

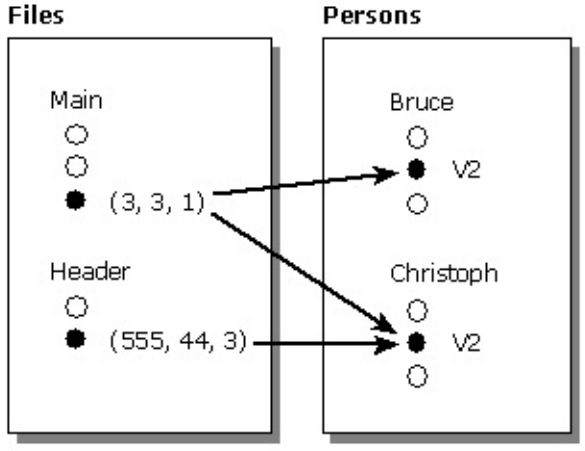
Although the preceding figure shows that the three different versions of Main have three different values for the collection type *Persons-of-File*, things are much simpler at run time. At run time, when your program manipulates an object, it manipulates a particular version of that object. In other words, whenever you secure a reference to an object, the repository engine actually gives you a reference to a specific version of an object. No matter how your program obtains the reference, through **IRepository::get_Object**, through navigation, or through any other technique, the repository engine provides you a reference to one version of that object.

For example, suppose your program has a reference to **Version 3-of-Main** and **Version 2-of-Header**. The following figure distinguishes between object versions to which your program has current references (filled-in circles) and the other object versions (blank circles).



The preceding figure indicates that your program does not currently have a reference to any person. Your program merely has references to some collections that include persons. To get a reference to a specific person (for example, **Bruce**), your program can navigate to it.

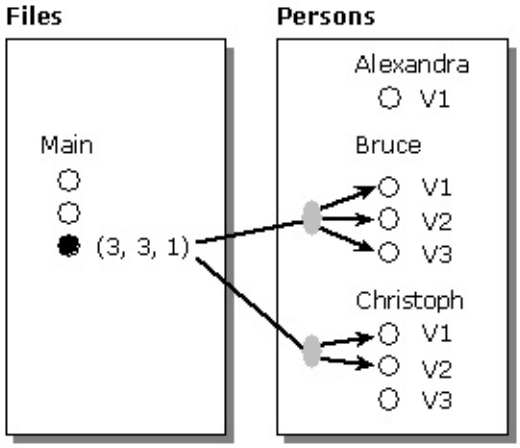
The preceding figure is simplified; it shows only one version of each person. The following figure is more realistic.



The preceding figure shows two collections. One collection is *Persons-of-Version 3-of-Main*, which contains **Version 2-of-Bruce** and **Version 2-of-Christoph**. The other collection is **Persons-of-Version 2-of-Header**; it contains **Version 2-of-Christoph**. The figure also shows that your program already has references to **Version 2-of-Bruce** and **Version 2-of-Christoph** (because the corresponding dots are filled in).

The preceding figure reflects that when you navigate along a relationship to a target object, you navigate to a specific version of that object. The figure reflects this by showing each arrow pointing to a specific version of an object to which you have already navigated (a filled-in circle in the set of versions of persons).

In most situations, this view is adequate. For example, you can think of a target object collection as containing a particular version of each target object. A more detailed view, shown in the following figure, is sometimes useful.



The preceding figure shows a single collection, *Persons-of-Version 3-Of-Main*. It contains two items: **Bruce** and **Christoph**. The figure does not indicate which particular version of **Bruce** is in the collection, because your program has not yet navigated from **Version 3-of-Main** to **Bruce**. But it does indicate that when you do navigate to **Bruce**, the repository engine can return a reference to any of the three versions. Similarly, the picture does not indicate which version of **Christoph** is in the collection, but it does indicate that when you navigate to **Christoph**, the repository engine returns a reference to **Version 1** or to **Version 2**, but not to **Version 3**.

See Also

Repository Object Architecture

Versioning Objects

Kinds of Version Collections

Version collections are used to access repository object versions. There are seven kinds of version collections. All version collections inherit from the **IVersionCol** interface.

ObjectVersions Collection

RepositoryObjectVersion implements the **ObjectVersions** collection. The **ObjectVersions** collection contains all the versions of a particular repository object. For example, if you have multiple versions of the **Column** object, you can access all of them through the **ObjectVersions** collection. To establish this collection, use the **get_ObjectVersions** method of the **IRepositoryObjectVersion** interface.

Predecessor Collection

RepositoryObjectVersion implements the **Predecessor** collection. The **Predecessor** collection contains all the immediate predecessors of an object version. Although only one predecessor is the creation version, multiple predecessors can exist. For example, when you merge an object version into another, existing object version, the object version that you merge becomes a new, noncreation predecessor. To establish this collection, use the **get_PredecessorVersions** method of the **IRepositoryObjectVersion** interface.

Successor Collection

RepositoryObjectVersion implements the **Successor** collection. The **Successor** collection contains all the immediate successors of an object version. An immediate successor is an object that is one step away in the version graph. For example, if **LoanTable_1** is versioned into two more loan tables (**LoanTable_2** and **LoanTable_3**), both **LoanTable_2** and **LoanTable_3** are immediate successors. Subsequent versioning of **LoanTable_2** and **LoanTable_3** results in successors that are not part of the **Successor** collection of **LoanTable_1**. To establish this collection, use the **get_SuccessorVersions** method of the

IRepositoryObjectVersion interface.

TargetVersions Collection

VersionedRelationship implements the **TargetVersions** collection. The **TargetVersions** collection contains the specific versions of a target object that are related to a particular version of a source object. For example, if a **Table** object is related to two versions of the same **Column** object, you can access both versions of the **Column** object through a **TargetVersions** collection. To establish this collection, use the **get_TargetVersions** method of the **IVersionedRelationship** interface.

Contents Collection

Workspace implements the **Contents** collection. The **Contents** collection contains all the object versions present in a workspace. Remember that, at most, one version of each object can appear in a workspace. So, at most, you will have only one instance of each object in a **Contents** collection. For example, if a workspace contains a **Schema**, **Table**, and a **Tables** collection, the **Contents** collection includes a **Schema** object, a **Table** object, and the **Tables** collection object. To establish this collection, use the **get_Contents** method of the **IWorkspace** interface.

Workspaces Collection

RepositoryObjectVersion implements the **Workspaces** collection. The **Workspaces** collection contains all the workspaces in which a particular object version is present. A repository object version can exist in multiple workspaces. For example, if you have one workspace for testing purposes and another workspace for production, both **Workspaces** can contain the same version of the same repository object. In this case, the **Workspaces** collection contains references to both workspaces. To establish this collection, use the **get_Workspaces** method of the **IWorkspaceItem** interface.

Checkouts Collection

Workspace implements the **Checkouts** collection. The **Checkouts** collection contains all the object versions checked out to a particular workspace (that is, all

object versions that can be modified or removed within the context of a workspace). For more information, see Objects Within Workspaces. To establish this collection, use the **get_Checkouts** method of the **IWorkspace** interface.

See Also

IRepositoryObjectVersion Interface

IVersionCol Interface

IVersionedRelationship Interface

IWorkspace Interface

Navigating a Repository

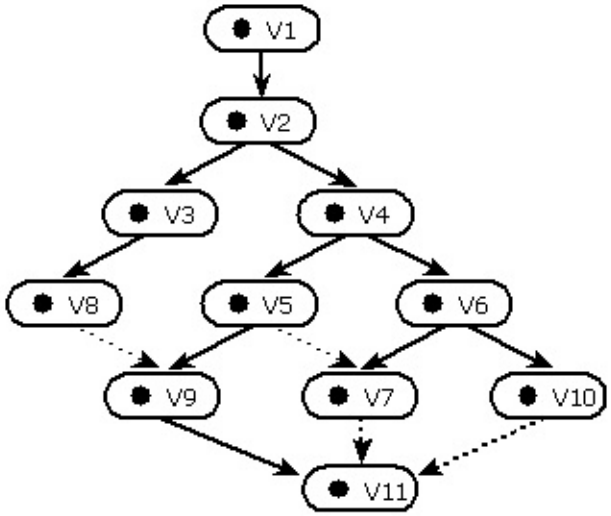
Navigating the Version Graph

Retaining Workspace Context

Version Graph

Version Graph

Each repository object has a version graph, which indicates how the various versions relate to each other. An object's version graph consists of nodes and arrows. Each node represents a version of the object and each arrow points from one object version to a successor of that object version. The following figure shows a typical version graph for an object with 11 versions.



There are two kinds of arrows. A solid arrow indicates the creation of one object version based on another. For example, the solid arrow from **Version 6** to **Version 7** indicates that **Version 7** was created based on **Version 6**. That is, **Version 7** was created when a program invoked the **CreateVersion** method on an **IRepositoryObjectVersion** interface pointer to **Version 6**.

A dashed arrow indicates the merging of property values and collections from one object into another. For example, the dashed arrow from **Version 10** to **Version 11** indicates that property values and collections from **Version 10** were merged into **Version 11**. That is, the dashed arrow was created when a program invoked the **MergeVersion** method with an **IRepositoryObjectVersion** interface pointer to **Version 11** (and the invoking program provided an interface pointer to **Version 10** as an input parameter *i*).

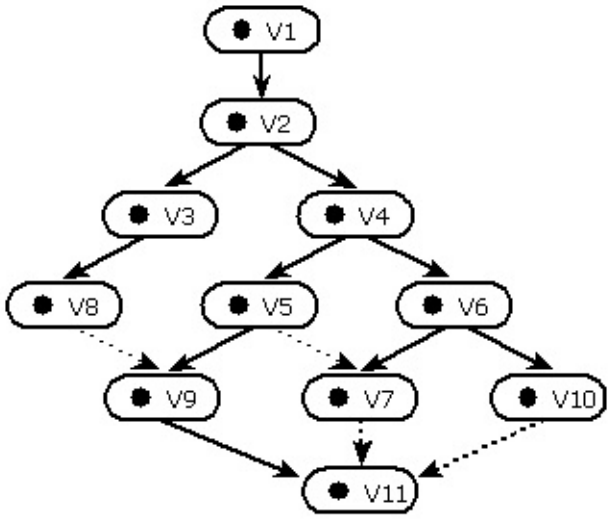
See Also

Navigating the Version Graph

You can navigate a version graph by traversing collections. For more information about the kinds of collections that contain versioned objects, see Kinds of Version Collections.

The repository engine supports navigation of the version graph in the following ways:

- Every object version has a collection of successor versions, the other versions of the same object that immediately follow in the version graph. In the following figure, **Version 5** has two successor versions, **Version 9** and **Version 7**. An object version's set of successor versions can be null. For example, **Version 11** has no successors.



- Every object version has a collection of predecessor versions, the other versions of the same object that immediately precede it in the version graph. In the preceding figure, for example, **Version 11** has three predecessor versions: **Version 9**, **Version 7**, and **Version 10**.

- Every object version (except **Version 1**) has a predecessor creation version, the predecessor version from which the current object version was created. For example, of **Version 11**'s three predecessor versions,

only **Version 9** is its predecessor creation version.

- Every object version has a collection of object versions, the entire set of versions of the object.

See Also

Branches in the Version Graph

Navigating a Repository

Version Graph

Versioning Objects

Manipulating Versioned Relationships

Access to relationships is supported at both the COM level and the Automation level. Given a versioned relationship that connects two repository object versions, you can perform the operations listed in the following table. These operations are performed relative to a specific version of the source object.

To	Use
Pin the destination object version	The **Pin** method of the **IVersionedRelationship** interface.
Unpin the destination object version	The **Unpin** method of the **IVersionedRelationship** interface.
Retrieve (a version of) the target object	The **Target** property of the **IRelationship** interface.
Retrieve the source object version	The **Source** property of the **IRelationship** interface.
Create a new relationship to relate a new target object to a source object	The **Add** method of the **IRelationshipCol** or **ITargetObjectCol** interface.
Relate a subsequent target object version to a source object	The **Add** method for the **TargetVersions** collection. This collection is accessible through the **IVersionedRelationship** interface.

For more information, see the Microsoft® SQL Server™ 2000 Meta Data Services Software Development Kit (SDK).

See Also

Changing a Destination Relationship's Name

Repository API Reference

Versioning Objects

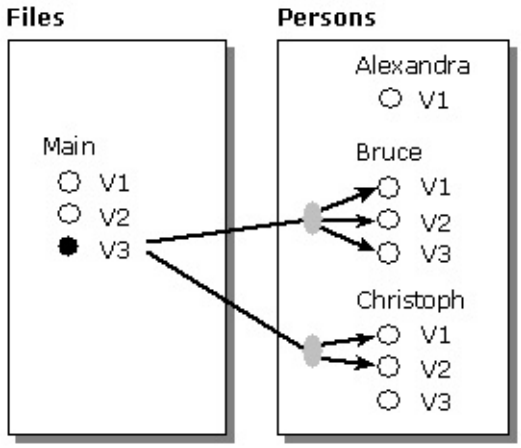
Version-to-Version Relationships

Version-to-Version Relationships

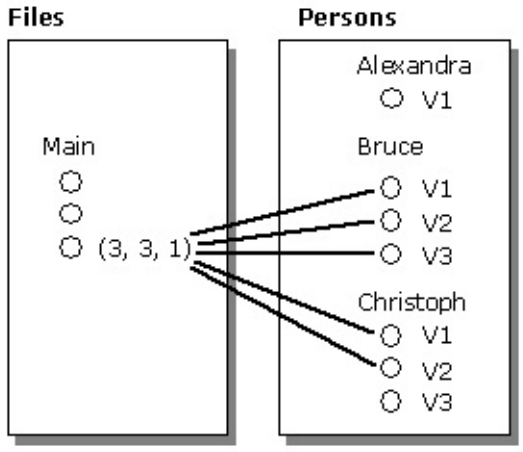
A version-to-version relationship is a relationship that associates a particular version of the origin object with a particular version of the destination object. At run time, the repository object model never presents an individual version-to-version relationship to you. That is, from within a COM or Automation program, you cannot materialize an object corresponding to a version-to-version relationship. Instead, you can materialize an object corresponding to a versioned relationship using the **IVersionedRelationship** interface or **VersionedRelationship** object.

After you materialize the versioned relationship, you can select a specific version-to-version relationship by allowing the repository engine to follow a resolution strategy that picks one for you, or by selecting a specific version from a **TargetVersions** collection.

The following figure shows versioned relationships.

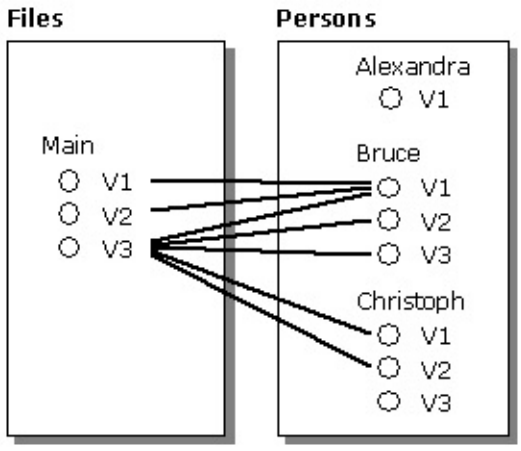


The preceding figure shows two versioned relationships; however, five version-to-version relationships are evident, as shown in the following figure.



In the preceding figure, each line is a version-to-version relationship. The top line indicates that Version 1-of-Bruce is in the **TargetVersions** collection of the versioned relationship owned by Version 3-of-Main. It also indicates that Version 3-of-Main is in the **TargetVersions** collection of the versioned relationship owned by Version 1-of-Bruce. The complete **TargetVersions** collection of the versioned relationship owned by Version 1-of-Bruce might include other items. That is, there might be other version-to-version relationships between Version 1-of-Bruce and individual versions of Main.

The following figure shows that Version 1-of-Bruce has version-to-version relationships to Version 1-of-Main, Version 2-of-Main, and Version 3-of-Main.



The preceding figure shows a total of seven version-to-version relationships.

See Also

IVersionedRelationship Interface

Manipulating Object Versions

Access to repository object versions is supported at both the COM level and the Automation level. Given a specific version of a repository object, you can perform the operations listed in the following table. For more information about retrieving or changing object version names, see Retrieving an Object Version's Name and Changing an Object Version's Name.

To	Use
Create the first object version	The **CreateObject** method of the **IRepository** interface.
Create subsequent object versions	The **CreateVersion** method of the **IRepositoryObjectVersion** interface.
Determine which predecessor version was the creation version	The **PredecessorCreationVersion** method of the **IRepositoryObjectVersion** interface.
Determine how this version of the current object was resolved	The **ResolutionType** method of the **IRepositoryObjectVersion** interface.
Freeze an object version	The **FreezeVersion** method of the **IRepositoryObjectVersion** interface.
Retrieve the object-version identifier of an object version	The **VersionID** property of the **IRepositoryObjectVersion** interface.
Retrieve the state of an object version	The **IsFrozen** method of the **IRepositoryObjectVersion** interface and the **IsCheckedOut** method of the **IWorkspaceItem** interface.
Merge the contents of another object version into the current object version	The **MergeVersion** method of the **IRepositoryObjectVersion** interface.

For more information, see the Microsoft® SQL Server™ 2000 Meta Data Services Software Development Kit (SDK).

See Also

IRepository Interface

IRepositoryObjectVersion Interface

IWorkspaceItem Interface

Repository API Reference

Repository Object

RepositoryObjectVersion Object

Versioning Objects

Workspace Object

Creating Object Versions

Whenever you want to continue modifying an object without overwriting the nonannotational property values and origin collections of the existing object versions, you create a new version of the object. When you create a new object version, you must use an existing, frozen version of the object as the creation version of the to-be-created version. To create the new version, you invoke the **CreateVersion** method with an **IRepositoryObjectVersion** interface pointer to the creation version. The repository engine creates a new, unfrozen version of the object. The new version has property values identical to those of the creation version. The collections of the new version are based on the creation version's collections, as follows:

- The repository engine copies each origin collection whose type has the COLLECTION_NEWORGVERSIONSPARTICIPATE flag set. If this flag is not set, the origin collection is not copied.

- By default, the repository engine does not copy the creation version's destination collections into the newly created version. Your application might, however, include custom behavior for the **CreateVersion** method that does copy some or all destination collections.

When you create a new version of an object, the repository engine modifies the version graph accordingly.

See Also

Branches in the Version Graph

Freezing an Object Version

IRepositoryObjectVersion Interface

Merging Object Versions

Propagating Versions

Propagating Versions

The repository engine can sometimes create a new versioned object automatically, in response to the versioning of another, related object. More specifically, you can create a new version of an origin object automatically when you purposely create a new version of a destination object. The automatic creation of an object version is called a propagated version. Version propagation is the process by which the repository determines which propagated versions are necessary and then performs those propagated version.

You can implement version propagation for collections that contain versioned objects and versioned relationships. The occurrence of version propagation depends on flags you set for the collection that contains the versioned items.

To implement version propagation, you must set the COLLECTION_NEWDESTVERSIONPROPAGATE flag on the collection. When this flag is set, invoking the **CreateVersion** method on a destination object propagates versioning to origin objects related through collections of this type.

After version propagation is in progress, it can continue to propagate origin-destination pairs. This occurs when a newly versioned origin object is simultaneously a destination object of another relationship. In this case, its origin object is also versioned. The versioning of paired objects continues up the version graph until a frozen origin object is encountered. This behavior occurs only while the origin object is unfrozen, and it occurs only for relationships that are created within the same transaction.

Version propagation creates a new, unfrozen version that has property values that are identical to the property values of the creation version. You can set additional **CollectionDefFlags** to further determine how object versions are propagated.

The following **CollectionDefFlags** can be set to determine how and whether version propagation occurs.

Flag	Result
COLLECTION_NEWDESTVERSIONPROPAGATE	Version

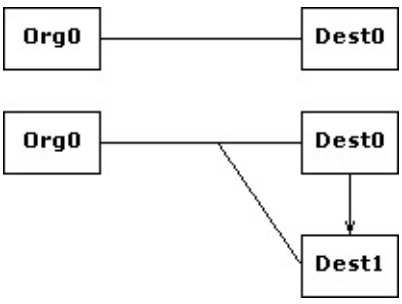
	propagation occurs when the CreateVersion method is invoked on a destination object that is related to an unfrozen origin object.
COLLECTION_NEWDESTVERSIONADD	The origin object always links to the latest version of a destination object, eliminating manual versioning of an origin object in response to a newly versioned destination object.
COLLECTION_NEWORGVERSIONSDONOTPARTICIPATE	The origin collection is not copied from the creation origin object to the newly versioned origin object,

	even if other flags support version propagation.
COLLECTION_NEWDESTVERSIONSDONOTPARTICIPATE	The destination collection is not copied from the creation destination object to the newly versioned destination object, even if other flags support version propagation.

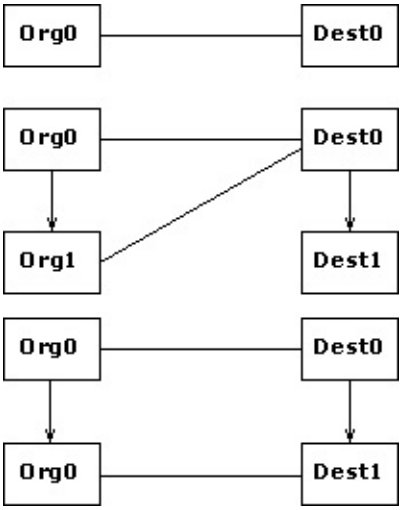
Usage Scenarios

The version propagation functionality supports the following scenarios:

The first scenario, shown in the following figure, demonstrates the case when the origin object should be linked to the latest version of the destination object. In this case, the new version of **Dest**, **Dest1**, is added to the **TargetVersions** collection of the relationship. In this scenario, the COLLECTION_NEWDESTVERSIONADD flag is set, and the origin object is not frozen.



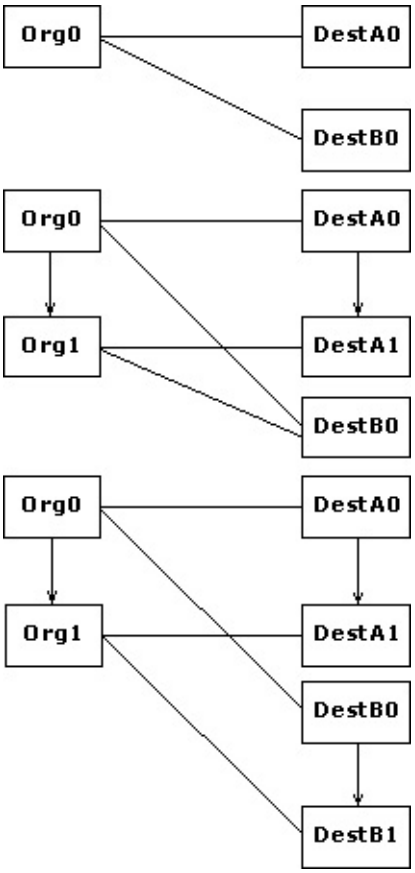
The second scenario, as shown in the following figure, demonstrates the case when the origin object needs to be versioned when the destination object is versioned. In this scenario, the COLLECTION_NEWDESTVERSIONPROPAGATE flag must be set, and the origin object must be frozen.



In this example, a new version of **Dest0**, **Dest1** is created. A new version of **Org0**, **Org1** is then created. Because, by default, the COLLECTION_NEWORGVERSIONSDONOTPARTICIPATE flag is not set, the new version of **Org1** includes a relationship to **Dest0**. This relationship is deleted, and a new relationship with **Dest1** is created. Note that this behavior happens only if the origin object is frozen. If it is not frozen, the origin object is not versioned.

This behavior can propagate. That is, any object for which **Org** is a destination will also be versioned. The behavior will propagate until the engine reaches an object that is not frozen, or is not the destination of any relationships or any relationships for which the propagation flag is not set.

The third scenario, as shown in the following figure, demonstrates the case when an origin object has multiple relationships with destination objects that must be versioned. In this scenario, the COLLECTION_NEWDESTVERSIONPROPAGATE flag is set.



In this example, **Org0** is the origin of relationships with both **DestA0** and **DestB0**. A new version of **DestA0** is created, **DestA1**, and the version is propagated to **Org0**, as described in the previous example. Both the **Org0** and **Org1** have relationships to the existing **DestB0**. When a new version of **DestB0**, **DestB1**, is created, a new relationship to the already versioned **Org1** is added, and the relationship between **Org1** and **DestB0** is deleted.

To summarize, propagating relationships during a transaction creates only a single version of an origin object.

See Also

CollectionDefFlags Enumeration

Propagating Deletes

IRepositoryObjectVersion::CreateVersion

RepositoryObjectVersion CreateVersion Method

Versioning Objects

Freezing an Object Version

In the version graph, each object version with an emerging (solid or dashed) arrow must be frozen. The other object versions can be frozen or unfrozen. One purpose of the repository engine's versioning capability is to let you maintain multiple versions of an object so that you can remember what the object was like at different times. After you decide that a particular version of an object is worth remembering, you must protect that version of the object from further modification. You do this by freezing the object version. To freeze the object version, invoke the **FreezeVersion** method with an **IRepositoryObjectVersion** interface pointer to the version you want to preserve.

When you freeze a version of an object, you prevent any program from modifying any of its origin collections or any of its nonannotational property values. A program can, however, modify a frozen object version in the following ways:

- Modify a frozen object version's destination collections. By allowing such modifications, the repository engine lets you protect an object (such as a text formatting template) from further modification, yet allows other, newly created objects (such as text files) to include the frozen object version in their origin collections.

- Modify an object's annotational properties. If a class exposes (through one of its interfaces) an annotational property, the repository engine stores one value of that property for each object (not one property value for each object version). Thus, if you change an annotational property value on an unfrozen version of an object, the change affects all versions of that object, including the frozen versions.

The repository engine provides two methods for you to manage the frozen status of an object version:

- **FreezeVersion** freezes a version of an object.

- **IsFrozen** (exposed by **IRepositoryObjectVersion**) determines whether an object version is frozen or unfrozen.

See Also

Branches in the Version Graph

Creating Object Versions

IRepositoryObjectVersion Interface

Merging Object Versions

Version Graph

Versioning Objects

Resolution Strategy for Objects and Object Versions

When you retrieve an object or navigate to an object, the repository engine returns an interface pointer to a specific version of that object. You can explicitly ask for a particular version, or you can rely on the repository engine to choose a version of the object for you. For example, you may have repository objects that do not explicitly provide version information (instances of **RepositoryObject** do not provide version information). When objects lack specific version information, the repository engine can choose an instance for you.

If the repository engine chooses for you, it can choose any of the following:

- The most recently created object version.
- The object version present in the workspace in which you are operating.
- The pinned target object version of the relationship that you are navigating along.

You can predict how the repository engine selects an object version to return to you:

- If you explicitly request a specific version of an object, the repository engine retrieves that version; if for any reason it cannot retrieve that version, it returns an error.
- If you are operating within a workspace, the repository engine retrieves the version that is in the workspace; if for any reason it cannot return the in-workspace version of the object, it returns an error.
- If you do not request a specific version and you are not operating within a workspace, the repository engine returns either the most recently created version or (if applicable) the pinned version.

The following topics discuss how the repository engine chooses among versions of an item.

Topic	Description
Requesting a Specific Version	Explains how to select a specific version
Resolution While Operating Within a Workspace	Explains how the repository engine selects an object from a workspace
Resolution While Operating Outside a Workspace	Explains how the repository engine selects an object from a centralized, shared repository

See Also

IRepositoryObject Interface

IRepositoryObjectVersion Interface

Navigating a Repository

RepositoryObject Object

RepositoryObjectVersion Object

Versioning Objects

Requesting a Specific Version

In some situations, you can request a specific version of an object. For example, the **get_Version** method of the **IRepository2** interface retrieves the specific object version whose object-version identifier you supply. If the repository engine cannot return this particular object version to you, it returns an error. For example, it returns an error if the specific version you requested does not exist, or if the specific version you requested is not present in the workspace in which you are operating.

See Also

IRepository2 Interface

Object-Version Identifiers and Internal Object-Version Identifiers

Resolution Strategy for Objects and Object Versions

Versioning Objects

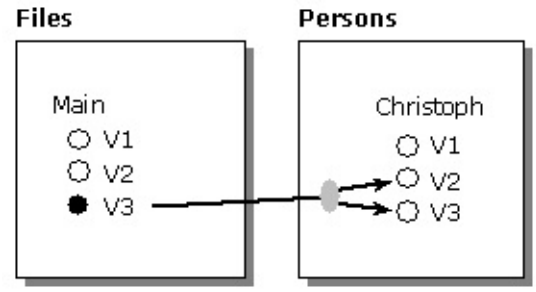
Resolution While Operating Within a Workspace

If you are operating within a workspace, the repository engine returns the version of the object that is present in the workspace. If the repository engine cannot return the in-workspace object to you, it returns an error. The repository engine may fail to return an in-workspace version of the object you requested in the following situations:

- If the workspace contains no version of the requested object, the repository engine returns an error.
- If the workspace contains a version other than the specific version you explicitly requested, the repository engine returns an error.
- If you navigate to a target object along a relationship that is a member of a relationship collection, but the specific version of the target object in your workspace is not among the specific versions of the target object that participate in the relationship, the repository engine returns an error.

For example, the following figure shows one item in the collection **Persons-of-V3-of-Main**. The target object of the item is **Christoph**. Because you have not yet navigated along the relationship, the figure does not show which particular version of **Christoph** will be returned to you; it shows only that it will be **Version 2-of-Christoph** or **Version 3-of-Christoph**.

If you are operating in a workspace that contains **Version 1-of-Christoph**, the repository engine returns an error. When you invoke the **get_Target** method of the **IRelationship** interface, the repository engine cannot find a suitable version of **Christoph** to return to you. It cannot return **Version 1** because **Version 1-of-Christoph** is not related to the source of the navigation. It cannot return **Version 2** or **Version 3** because neither **Version 2-of-Christoph** nor **Version 3-of-Christoph** is in the workspace in which you are operating. (The workspace contains object versions and a workspace can contain only one version of each object.)



In your programs, you can avoid this error by manipulating target object collections rather than relationship collections. This error occurs only when a collection includes an item that the repository engine cannot resolve to an in-workspace object. The only situation in which this occurs is described in the preceding example: the collection is a relationship collection, and none of its items refers to the specific version of the target object that is in the workspace. When you establish a target object collection, however, each item in the collection is a repository object (rather than a relationship to a repository object). If you establish the collection while operating within a workspace, each item in the collection is a version of the target object that is present in the workspace.

See Also

Resolution Strategy for Objects and Object Versions

Versioning Objects

Resolution While Operating Outside a Workspace

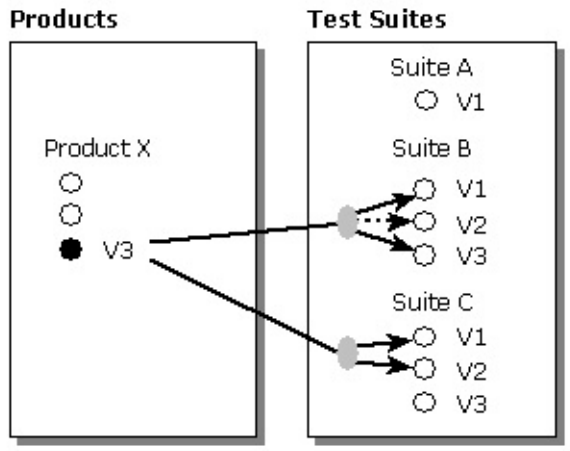
If you do not request a specific version and you are not operating within a workspace, the repository engine generally returns the most recently created version of an object. In one situation, however, the repository engine first tries to find another, preferable version of the requested object. If you are navigating from an origin object to a destination object, and there is a pinned version of the target object, the repository engine returns an interface pointer to the pinned version. If there is no pinned version, the repository engine simply returns an interface pointer to the most recently created version of the target object that participates in the relationship.

Following are some basic facts about pinning:

- The repository engine can return a pinned version (see Example One).
- If there is not a pinned version, the repository engine can return the most recent version (see Example Two).
- A destination object version can be pinned to several origin object versions (see Example Three).
- If a target object version is pinned for one versioned relationship, it is not necessarily pinned for others (see Example Four).
- You can pin at most one version of the destination object for each relationship.
- You can pin a version of the destination object only; you cannot pin an item within the target versions collection of a destination relationship.

Pinning Example One

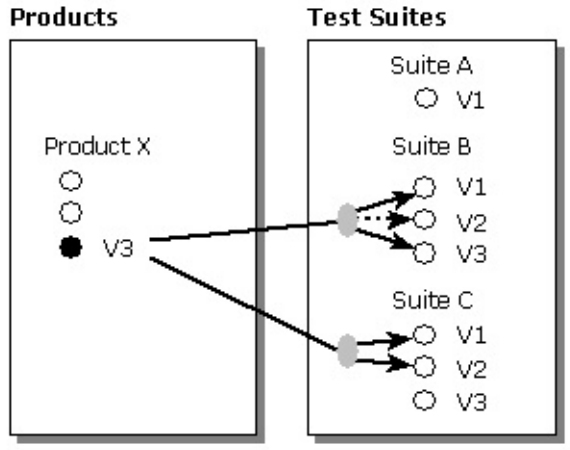
The following figure shows a two-item collection: **TestSuites-of-Version 3-of-Main**. The two items are **SuiteB** and **SuiteC**. If you are not operating in a workspace and you navigate to **SuiteB**, the repository engine discovers a pinned version of the target object. (The figure shows the pinned version with a dashed arrow.) Thus, the repository engine returns **Version 2-of-SuiteB** to your program, even though **Version 3-of-SuiteB** was created more recently and is related to the source object version.



Pinning Example Two

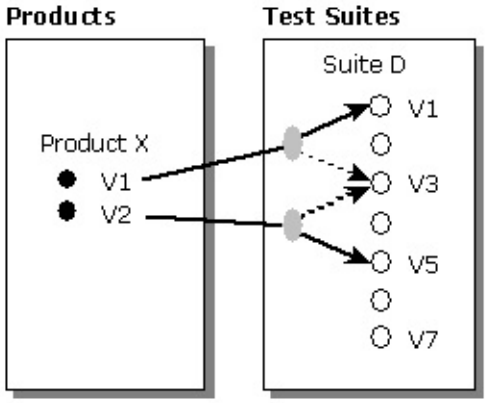
The following figure shows a two-item collection: **TestSuites-of-Version 3-of-Main**. The two items are **SuiteB** and **SuiteC**. If you are not operating in a workspace and you navigate to **SuiteC**, the repository engine finds no pinned version, so it returns **Version 2-of-SuiteC**.

Note Although **Version 3-of-SuiteC** was created more recently, the repository engine does not return it because there is no relationship between it and the source object version (**Version 3-of-ProductX**). The repository engine returns **Version 2-of-SuiteC** because, among the versions related to the source object version, **Version 2-of-SuiteC** is the most recently created version.



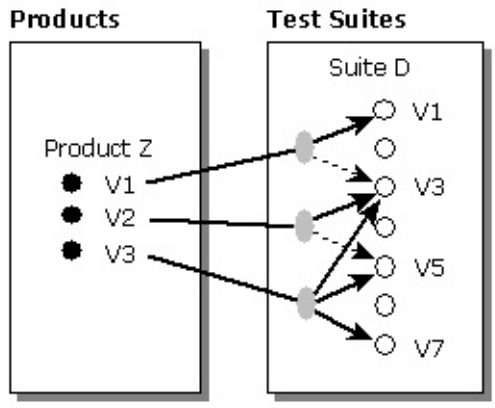
Pinning Example Three

A destination object version can be pinned to any number of origin object versions. For example, the following figure shows that **Version 3-of-SuiteD** is the pinned destination object version of two different items.



Pinning Example Four

If a target object version is pinned for one versioned relationship, it is not necessarily pinned for others. For example, the following figure simultaneously shows three versions of **Product Z**, each of which has a collection containing **Suite D**. All three versions of **Product Z** use **SuiteD** as the target object. The top item uses **Version 3-of-SuiteD** as the pinned version of the target object. The middle item, even though it includes the same version (**Version 3**) of the target object, does not have it pinned; it uses **Version 5-of-SuiteD** as the pinned version of the target object. The bottom item includes both **Version 3** and **Version 5** of **SuiteD**, but it includes no pinned version at all.



See Also

Resolution Strategy for Objects and Object Versions

Versioning Objects

Merging Object Versions

The version management feature of the repository engine supports branching. A branch results when you create a new object version whose predecessor version already has one or more successor versions. Common branching scenarios are:

- When two concurrent development efforts must change the same object.
- When a maintenance change is required on an older version of an already released object.

In scenarios like these, it is sometimes necessary to merge branched lines of development back together. You can merge one object version into another with the **MergeVersion** method of the **IRepositoryObjectVersion** interface. You can merge several branches together by successively merging two branches at a time until all branches have been merged.

The following topics describe the merging process in more detail.

Topic	Description
Merge Overview	Provides basic information about merge behavior
Invoking MergeVersion	Explains prerequisite steps for invoking the **MergeVersion** method
Resolving Merge Conflicts for Properties	Describes how conflicts between property values are resolved
Resolving Merge Conflicts for Collections	Describes how conflicts between collections are resolved
Examples of Merging Versions	Provides before and after examples of merged objects

See Also

Merge Overview

To perform a merge operation, you use the **MergeVersion** method of the **RepositoryObjectVersion** object.

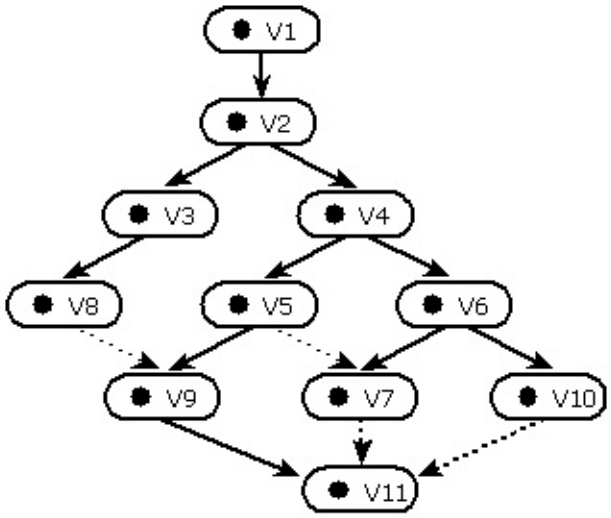
You can predict how the property values and collections in a successor version will change. The **MergeVersion** method modifies one object version, the successor, by combining its property values and collections with those of another version, the predecessor. **MergeVersion** compares the property values and collections of the predecessor version and the successor version to a third version, called the basis version.

MergeVersion does not combine two versions into a third, newly created version. Rather, it merges the property values and collections of one version into another version. After the operation is complete, the modified version becomes a successor of the other version. **MergeVersion** modifies the version graph accordingly.

Calculating the Basis Version

When you invoke the **MergeVersion** method, the repository engine uses the version graph to compare version data. The **MergeVersion** method compares each object version to be merged to a basis version of the same object. The basis version of the two to-be-merged object versions is the most recently created object version that is on the creation path of both the primary object version and the secondary object version of the merge. The creation path of an object version is a path through the version graph leading from the object version directly to **Version 1** of the object. Each step of the path leads from an object version to its predecessor creation version.

You can easily follow an object version's creation path backward from it to **Version 1** by following the solid arrows in reverse. For example, the version graph in the following figure shows that the creation path of **Version 11** goes through these other versions: 9, 5, 4, 2, and 1.



Comparing Collections

As it works, the **MergeVersion** method must compare collections to each other. It compares each collection in the basis version to its corresponding collection in the primary version and in the secondary version. **MergeVersion** considers two collections to be different if either of the following is true:

- One collection contains different objects from the other collection.
- A corresponding pair of items from the two collections differs from each other.

Comparing Versioned Relationships

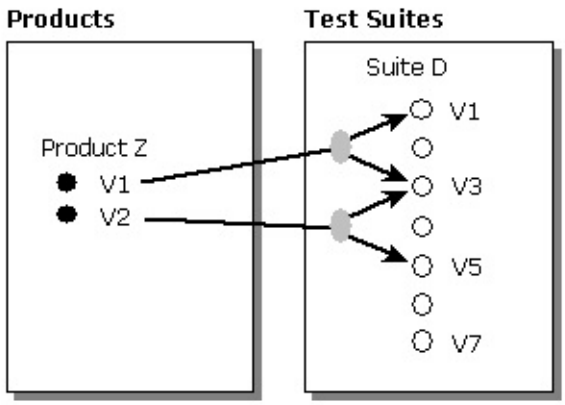
The **MergeVersion** method compares each collection of the basis version of an object to the corresponding collections of the primary version and of the secondary version. As part of these comparisons, the method must compare the versioned relationships of these collections. An item from the basis object version's collection corresponds to an item in the primary or secondary object version's collection if the two items use the same target object. Even if two items correspond, however, they can still differ in important ways. The repository engine considers two versioned relationships to differ if any of the following is true:

- The collection type is a sequenced collection and the two items have

different sequence numbers.

- The collection type is a naming collection and the two items have different names.
- The two items refer to different versions of the target object.
- The two items use a different version of the target object as pinned version.
- One item has a pinned target object version and the other does not.

For example, the following figure shows two items that differ in one respect only; the top item refers to **Versions 1** and **Version 3**. The corresponding item of the second collection refers to **Versions 3** and **Version 5**.



Invoking MergeVersion

When you invoke **MergeVersion**, the object version you will use as predecessor of the merge must already be frozen. The object version to be modified, the successor of the merge, cannot be frozen.

Briefly, the merge operation has these results:

- The object's version graph is updated to indicate that the merge operation occurred.
- The successor object can have different property values or collections.

You invoke **MergeVersion** on the successor version of the merge; you pass a reference to the predecessor as an input parameter. You also pass an indication of which version is to be considered the primary version. The primary version is the version whose member values are given priority when there are merge conflicts between the two versions.

See Also

IRepositoryObjectVersion::MergeVersion

Merge Overview

Merging Object Versions

Resolving Merge Conflicts for Collections

Resolving Merge Conflicts for Properties

Versioning Objects

Resolving Merge Conflicts for Properties

For each property, **MergeVersion** uses this rule to resolve merge conflicts:

- If the primary version differs from the basis version, the repository engine uses the property value from the primary version.
- If only the secondary version differs from the basis version, the repository engine uses the property value from the secondary version.
- If neither version differs from the basis version, the repository engine leaves the property value in the current version unchanged.

See Also

IRepositoryObjectVersion::MergeVersion

Merging Object Versions

Resolving Merge Conflicts for Collections

Version Graph

Versioning Objects

Resolving Merge Conflicts for Collections

For each collection, **MergeVersion** uses flags and rules to resolve merge conflicts. **CollectionDefFlags** that you set for a collection can determine how that collection is merged.

Setting the COLLECTION_MERGEWHOLE Flag

For each origin collection type whose COLLECTION_MERGEWHOLE flag is set, **MergeVersion** uses this rule:

- If the primary version's collection differs from the basis version's collection, the repository engine uses the collection from the primary version. For more information, see the Comparing Collections section of Merge Overview.

- If only the secondary version's collection differs from the basis version's collection, the repository engine uses the collection from the secondary version.

- If neither version differs from the basis version, the repository engine leaves the collection in the current version unchanged.

Not Setting the COLLECTION_MERGEWHOLE Flag

For each origin collection type whose COLLECTION_MERGEWHOLE flag is not set, **MergeVersion** combines the items in the two collections as follows:

- **MergeVersion** includes in the resulting collection each item in the basis version not changed in or deleted from either the primary version or secondary version. For more information, see the Comparing Versioned Relationships section of Merge Overview.

- **MergeVersion** includes in the resulting collection each item in the

primary version's collection that differs from the basis version.

- **MergeVersion** includes in the resulting collection each item in the secondary version's collection that differs from the basis version, provided the corresponding items in the primary version and basis version do not differ from each other.

Note The resulting collection can exclude some items found in the basis object version's collection. For example, if the primary version's collection excludes the item, the resulting collection will exclude the item. Similarly, if the primary version's collection includes an item that is identical to an item in the basis version's collection, but the secondary object version excludes the item, the resulting collection will exclude the item.

For more information about merge behavior, see Examples of Merging Versions.

See Also

CollectionDefFlags Enumeration

IRepositoryObjectVersion::MergeVersion

Merge Overview

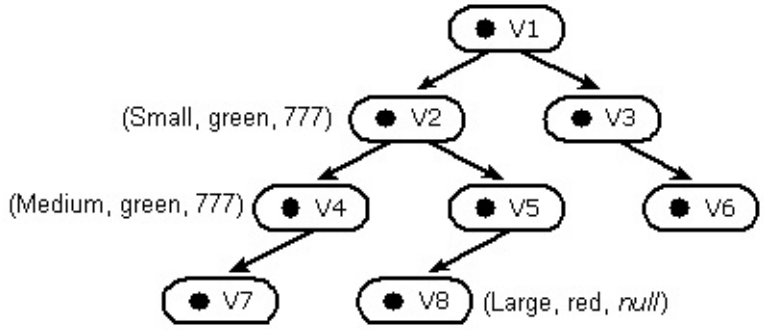
Merging Object Versions

Resolving Merge Conflicts for Properties

Versioning Objects

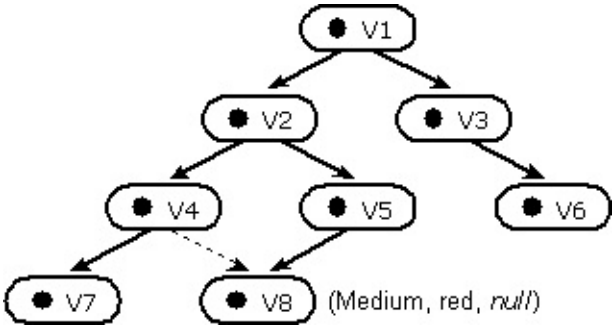
Examples of Merging Versions

A typical version graph for an object is shown in the following figure. The object has three properties: **Size**, **Color**, and **Quantity**. For selected versions of the object, the figure shows the values of these properties as ordered triplets.



If you merge **Version 4** into **Version 8** (using **Version 4** as the primary version), the repository engine uses **Version 2** as the basis version.

The resulting version graph looks like the following.



In the resulting version graph, notice the following:

- **Version 4** is now a noncreation predecessor of Version 8.

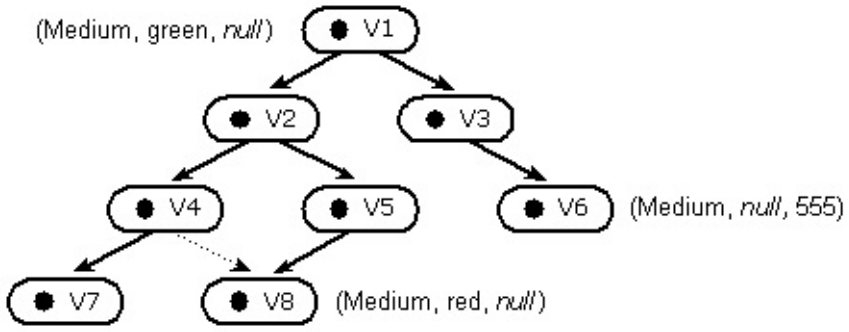
- In **Version 8**, the value of the **Size** property is medium. The change in the primary version, from small to medium, prevails over the change in the secondary version, from small to large.

- In **Version 8**, the value of the **Color** property is red. The change in the

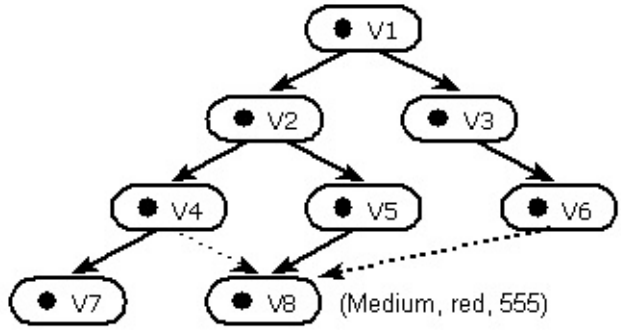
secondary version's value prevails because the corresponding value in the primary version did not change.

- In **Version 8**, the value of the **Quantity** property is null. The change in the secondary version's value prevails because the corresponding value in the primary version did not change.

Later, you merge **Version 6** into **Version 8**, using **Version 8** as the primary version. The repository engine uses **Version 1** as the basis version. Before the merge, the version graph (with important property values shown) looks like the following.



The resulting version graph is shown in the following figure.



In the resulting version graph in the preceding figure, notice the following:

- **Version 6** is now a noncreation predecessor of **Version 8**.

- In **Version 8**, the value of the **Size** property is medium. Neither the primary nor secondary version's value had changed from the basis version's value.

- In **Version 8**, the value of the **Color** property is red. The change in the primary version's value, from green to red, prevails over the change in the secondary version's value, from green to null.

- In **Version 8**, the value of the **Quantity** property is 555. The change in the secondary version's value, from null to 555, prevails because the corresponding value in the primary version did not change.

See Also

Merge Overview

Version Graph

Versioning Objects

Meta Data Services Programming

Programming Objects

This section provides information about programming special-purpose objects and collections.

Topic	Description
Programming BLOBs and Large Text Fields	Describes the binary large object (BLOB) and large text field support that is available through the repository API
Programming Transient Object Collections	Describes how to instantiate a transient object collection

See Also

Accessing Repository Objects

Accessing Target Object Collections

Programming Information Models

Programming BLOBs and Large Text Fields

Repository engine provides interfaces to handle properties that are binary large objects (BLOBs) and large text fields. BLOBs are properties that have values containing text or image data that can be in excess of 64 kilobytes (KB). You can use BLOBs to perform database operations that require you to work with large segments of data at a time.

To define a BLOB, create a **PropertyDef** object, and then do the following:

1. Set the **SQLType** property to SQL_LONGVARBINARY or SQL_LONGVARCHAR.

2. Set the **SQLBlobSize** property to a value greater than 64 KB.

When **SQLType** is set to either SQL_LONGVARBINARY or SQL_LONGVARCHAR, **SQLBlobSize** (rather than **SQLSize**) determines the maximum size.

To work with a BLOB, use the **IReposPropertyLarge** interface. This interface provides methods that support BLOB manipulation. Specifically, it can be used to read, write, move, and seek information about a BLOB.

The locking behavior for the methods on this interface varies from the locking behavior used by other repository interfaces. Specifically, as soon as you invoke the **Write** and **WriteToFile** methods, the repository engine locks the database row until you commit the transaction. In contrast, locking for other rows only occurs for the duration of the commit.

When you use **IReposPropertyLarge** to manipulate an object, avoid using other repository property interfaces (such as **IReposProperty** or **IReposProperty2**) on the same property. These interfaces only work with properties that contain up to 64 KB of data. If your property exceeds 64 KB, you will only get the first 64 KB of it.

When you version a BLOB or large text field property, you can use **CreateVersion** to create the version and **MergeVersion** to combine versions.

MergeVersion always selects the primary version of the BLOB or large text field. The new values are inserted directly into the database (rather than cached). For this reason, atomic operations are not supported for versioning these kinds of properties. For more information about atomicity of operations, see Transaction Management Overview.

For more information about other repository property interfaces, see Accessing Properties.

Note **IReposPropertyLarge** is basically a dispatch-based version of the **IStream** OLE 2.0 interface.

For more information about handling BLOBs, search on "Stream Objects" and "**IStream** Interface" in the MSDN® Library at the Microsoft Web site.

See Also

Accessing Repository Objects

IPropertyDef2 Interface

IRepositoryObjectVersion::CreateVersion

IRepositoryObjectVersion::MergeVersion

IReposProperty Interface

IReposProperty2 Interface

IReposPropertyLarge Interface

PropertyDef Object

RepositoryObjectVersion CreateVersion Method

RepositoryObjectVersion MergeVersion Method

Programming Transient Object Collections

Transient object collections are not stored in a repository database. This means that you can populate a transient object collection dynamically, using application code to determine the criteria for including objects.

Transient object collections extend your ability to create collections by providing a way to create collections at run time. Typically, the way you do this is through script. In this case script, instead of stored repository data, is used to dynamically populate a transient object collection. After a transient object collection is instantiated, you can use the collection in your application code just as you would any other repository collection.

Because it is based on script, a transient object collection can contain any combination of objects that you want. It is not subject to the conformance constraints that apply to other kinds of repository object collections.

Transient object collections are exposed through the **ITransientObjCol** interface.

The following example illustrates how you can use script to populate a transient object collection. Suppose you have an information model that defines a catalog, schemas, and tables. Script that creates a collection of all tables for all schemas in the catalog can be something like the following:

```
Function GetTables()
Set ObjCol = CreateObject("Repository.TransientObjectCol")
Set reposCatalog = CurReposObject
For each reposSchema in reposCatalog.Interface("Catalog").Schemas
    For each reposTable in reposSchema.Interface("Schema").Tables
        ObjCol.Add reposTable
    Next
Next
Set GetTables = ObjCol
End Function
```

See Also

Accessing Repository Objects

Accessing Target Object Collections

Defining Script Objects

ITransientObjectCol Interface

Understanding Collections

Understanding Relationship Roles

Meta Data Services Programming

Managing Transactions and Threads

When you make changes to a repository database, the updates are done within the scope of a transaction. You use repository transaction management methods to ensure that changes to a repository database always leave the database in a consistent state. This section discusses the transaction management support provided by the repository engine, and the threading model that is supported by repository objects.

Knowing how to perform a transaction is necessary if you are creating or extending information models programmatically, or writing programs that populate or update an information model.

Transactions are not required for programming to retrieve data through the repository engine.

The following topics provide more information about transactions.

Topic	Description
Transaction Management Overview	Explains when you should use a transaction and how transactions are implemented by the repository engine.
Design Issues and Transaction Management	Describes design issues that you should consider when implementing transaction behavior in an application.
Repository Objects and Multithreading	Explains transaction behavior and issues that apply when running multiple instances of a repository.

See Also

IRepositoryTransaction Interface

IRepositoryTransaction2 Interface

Repository Class

Transaction Management Overview

Transactions can be used to bracket multiple interactions with the repository engine. Changes to a repository that are a result of such interactions are either all committed or all undone, depending on the way that the transaction is completed. Repository methods that write data to a repository can be executed only within a transaction. Methods that read data from a repository can be executed without a transaction (although such reads can be done against partially updated data).

A repository database can have multiple repository instances connected to it simultaneously, from the same or from different processes. Each repository instance can have at most one transaction active at a time.

As a rule, the repository engine never implicitly cancels transactions. If a failure occurs, you must explicitly terminate the transaction. There is one exception to this rule: If you start a transaction, and then release the repository instance to which the transaction belongs, your transaction will be canceled.

Atomicity of Operations

OPT_ATOMICOP_MODE is an **IReposOption** option that you can set to enable or disable atomicity of operations. If this option is enabled, all operations, except commit, are executed atomically. If this option is disabled, the entire transaction is terminated in cases where an operation fails.

Atomicity of operations creates a backup copy of each row in cache just before an update and restores it in case the operation fails. If the operation is successful, the backup copy is discarded.

The following topics provide more information about defining a transaction.

Topic	Description
Managing Transactions	Explains the steps used to set up a transaction.
Nesting Transactions	Describes the scope and implementation of nesting

	transactions.
Transactions and Caching	Describes caching behavior as it relates to transactions.
Integration with Distributed Transaction Coordinator	Explains how to use Microsoft® Distributed Transaction Coordinator (MS DTC), a component of Microsoft SQL Server™ 2000, to coordinate multiple transactions.

See Also

Design Issues and Transaction Management

Managing Transactions and Threads

Repository Objects and Multithreading

IRepositoryTransaction::abort

Managing Transactions

Each instance of the **Repository** class implements the **IRepositoryTransaction** interface, which supports these methods and properties:

- The **Begin** method, which marks the beginning of a transaction
- The **Commit** method, which marks the end of a transaction
- The **Abort** method, which cancels a transaction and undoes all updates performed by the transaction
- The **GetOption** and **SetOption** methods, which get and set transaction options that control:
    - Whether other transactions are permitted to open the repository database from within the same process.
    - Whether updates are cached until the **Commit** is performed.
    - How long to wait for a transaction to start.
    - How long to wait for a query to complete.
    - How long to wait for a lock on a repository object.
- The **Status** property, which indicates whether or not a transaction is currently active

For each open repository instance, the pointer to the **IRepositoryTransaction** interface is available through the **Transaction** property. For more information about the transaction options you can set, see TransactionFlags Enumeration.

See Also

IRepositoryTransaction Interface

IRepositoryTransaction2 Interface

Managing Transactions and Threads

Nesting Transactions

The repository engine permits nesting of **Begin** and **Commit** method invocations, but no actual transaction nesting occurs. A nested transaction count is maintained for each open repository instance.

Invoking the **Begin** method during an active transaction increments the nested transaction count by one, but has no other effect.

Invoking the **Commit** method during an active transaction decrements the nested transaction count by one. If, and only if, this decrement reduces the nested transaction count to zero, all updates are committed to the repository database.

Invoking the **Abort** method during an active transaction undoes all changes made during that transaction. Changes made during any nested transactions are also undone, even if the **Commit** method has already been invoked for those transactions. The nested transaction count is set to zero.

See Also

IRepositoryTransaction::abort

IRepositoryTransaction::begin

IRepositoryTransaction::commit

Managing Transactions and Threads

Transactions and Caching

The repository engine changes are cached to improve performance. Guaranteed updates to a repository are written to persistent storage only when the active transaction is committed.

By default, multiple repository instances within the same process share a repository cache. Within the same process, when a transaction for one repository instance commits, its updates are immediately visible to transactions executing for other repository instances. These updates are not visible to open repository instances in other processes if those processes already have the preupdate data cached. Explicit refreshes are required to view updates from transactions that have completed in other processes.

You can override the default sharing behavior by setting a flag that allocates a new cache for each repository instance. For more information about REPOS_CONN_NEWCACHE, see ConnectionFlags Enumeration.

You can customize cache behavior by defining different age out strategies for different kinds of rows. For more information about age out strategies and caching behavior, see Optimizing Repository Performance.

To refresh a cache

- Refresh an individual repository item by invoking the **Refresh** method for the repository item.

  This method invalidates unchanged cache data for the repository item. Subsequent requests for that data will be fulfilled by retrieving the data from the repository database.

  -or-

- Refresh all repository items currently cached for an open repository instance by invoking the **Refresh** method associated with that repository instance.

  This method invalidates unchanged cache data for all repository items. Subsequent requests for that data will be fulfilled by retrieving the data

from the repository database.

In addition to explicit refreshes, repository items may be refreshed implicitly at any time by the repository engine, due to execution of internal repository caching algorithms.

See Also

Managing Transactions and Threads

Integration with Distributed Transaction Coordinator

You can design an application that runs a distributed transaction on Microsoft® SQL Server™ 2000 running on Microsoft Windows® 2000.

Before you use Microsoft Distributed Transaction Coordinator (MS DTC) with a SQL Server 2000 Meta Data Services repository, you must install the Windows 2000 Service Pack 1. This service pack fixes an intermittent bug that causes MS DTC to stop responding when committing changes to a repository database.

The protocol for coordinating transaction atomicity across multiple resource managers is a two-phase commit. The Microsoft facility for a two-phase commit is MS DTC. You can enable distributed transactions to support the following scenarios:

- Create an application that updates data in two repositories within the same transaction.
- Create an application that updates data in a repository and in another database within the same transaction.
- Create an application that runs a Microsoft Transaction Server (MTS) to update a repository, while running the application within the transaction that called it.
- Create an information model that aggregates a repository object class and updates another database within the aggregation wrapper.

The distributed transaction must be atomic; that is, it must either commit at all resource managers or terminate at all of them. For more information about supporting atomic operations, see Transaction Management Overview.

The Transaction Protocol

To support MS DTC in your application, you must set the TXN_USE_DTC

transaction flag. **IRepositoryTransaction** supports the TXN_USE_DTC flag on the **GetOption** and **SetOption** methods. The bit value for TXN_USE_DTC is 10. The default value of this option is FALSE. If the value is set to TRUE, each call to **IRepositoryTransaction::Begin** will create an MS DTC transaction.

IRepositoryTransaction::SetOption(10, 1)
IRepositoryTransaction::Begin

MS DTC requires that the participant who started the transaction be the only party who can call **Commit.**

Programming in Visual C++

If you are a Microsoft Visual C++® programmer, you can use the **ITransactionJoin::JoinTransaction** method. You can use this method to cause a repository instance that is not currently running a transaction to become part of an existing MS DTC transaction. The active MS DTC transaction object is passed in as an input argument. For more information about the **ITransactionJoin** interface, see SQL Server Books Online.

Programming in Microsoft Visual Basic

Microsoft Visual Basic® applications must use the following API to enlist an MS DTC transaction:

HRESULT IRepositoryTransaction2::JoinTransaction ([in]VARIANT sVArTxn);

where *sVArTxn* is an **IUnknown** pointer to the distributed transaction coordinator.

See Also

IRepositoryTransaction::begin

IRepositoryTransaction::commit

IRepositoryTransaction::getOption

IRepositoryTransaction::setOption

Managing Transactions and Threads

Design Issues and Transaction Management

When you design an application that writes or updates data in a repository database, you must consider these transaction management issues:

- What are the implications of reading repository data outside the scope of a transaction?
- Is a locking protocol necessary for the application?
- Are repository cache overflows likely to occur, and what can be done to avoid cache overflows?

The following topics provide more information about these issues.

Topic	Description
Reading Repository Data Outside of a Transaction	Explains strategies for successfully reading data outside a transaction.
Using a Lock Protocol	Describes how and why you should use lock protocols.
Avoiding Repository Cache Overflows	Explains how to avoid and correct for cache overflows.

See Also

Managing Transactions and Threads

Repository Objects and Multithreading

Reading Repository Data Outside of a Transaction

A repository engine method that reads data can execute outside a transaction. However, repository data that is read in this way can include partial updates from an active transaction.

To ensure that the data read from a repository does not include partial updates from active transactions, put read requests into a transaction. Otherwise, if you bracket your reads within a transaction and your repository database is a Microsoft® Jet database, you risk overloading the cache.

Microsoft Jet uses an in-memory cache to speed up query processing. Cached data is not released until the transaction is committed. If your repository application is reading a large amount of data, and you are performing the reads within the scope of a transaction to isolate them from the uncommitted changes of other applications, the Jet cache can grow so large that it causes the application to fail. To avoid this, commit your transaction periodically (even though it is a read-only transaction).

See Also

Managing Transactions and Threads

Restrictions for Microsoft Jet Repository Databases

Using a Lock Protocol

Executing transactions concurrently can adversely affect the integrity of repository data if a locking protocol is not used. For example, consider two concurrently executing transactions that both increment an integer property of a repository object. This property represents a sequential counter.

1. Transaction A reads the value of the property. The current value is six.

2. Transaction B reads the same current value of the property.

3. Transaction A increments the retrieved property value by one and writes it back to the repository database. The value in the database is now seven.

4. Transaction B increments its copy of the retrieved property value by one and writes it back to the repository database. The value in the database is still seven. It should be eight.

To avoid this problem, use the **Lock** method in concurrently executing transactions to serialize access to a repository item. The **Lock** method sets an exclusive lock on the item, and refreshes any unchanged cache data for the item. The lock is effective across processes and across computers. If the repository item is already locked, the lock request waits until the lock becomes available. The item is unlocked when the transaction is ended, either by the **Abort** method or by the final invocation of the **Commit** method for the transaction.

By invoking the **Lock** method, the caller has exclusive access to a repository item, as long as all other concurrently executing transactions also obtain a lock on that repository item before updating it.

By default, the repository engine will wait up to 20 seconds to get a lock on a repository object. If this lock time-out value is insufficient, you can increase it by setting a transaction flag. For more information, see TransactionFlags Enumeration.

Caution Calling the **Lock** method on a repository item only prevents other transactions from executing the **Lock** method on the item. It does not block other transactions that are not using the locking protocol from changing the item.

See Also

IRepositoryItem::Lock

IRepositoryTransaction::abort

Managing Transactions and Threads

RepositoryObjectVersion Lock Method

Workspace Lock Method

VersionedRelationship Lock Method

Avoiding Repository Cache Overflows

To enhance performance, repository transactions typically run in writeback mode. In writeback mode, the updates for a transaction are held in the repository cache until the transaction is committed. If a single transaction performs a large number of updates, it can cause the repository cache for the process to overflow.

By setting transaction options through the **SetOption** method, a repository instance can operate in exclusive writeback mode, where it allows no more than one active transaction at a time for a given process and repository database. Using exclusive writeback mode will reduce, but not eliminate, the possibility of a cache overflow. For a very large number of updates within a single transaction, or if memory is limited, the repository cache can still overflow.

To guarantee that cache overflows will not cause transactions to fail, set the exclusive writethrough mode transactional option. In exclusive writethrough mode, updates are immediately flushed from the repository cache. Exclusive writethrough mode does not affect your ability to cancel an active transaction by using the **Abort** method.

See Also

IRepositoryTransaction::Abort

IRepositoryTransaction::SetOption

Managing Transactions and Threads

TransactionFlags Enumeration

Repository Objects and Multithreading

A process can create multiple instances of the **Repository** class, with each instance connected to the same or to different repository databases. Instances of the **Repository** class, as well as instances of other repository-supplied classes, can be instantiated as either COM or Automation objects. With regard to multiple threads executing within a single process, these objects:

- Support multithread processing.
- Use the free-threaded model.
- Are thread-safe objects.

The following topics provide more information about each of these issues.

Topic	Description
Restrictions for Microsoft Jet Repository Databases	Explains how multithreading affects cache behavior in Microsoft® Jet databases.
Synchronizing Commit Operations	Identifies which methods require synchronization between application threads.

See Also

Design Issues and Transaction Management

Managing Transactions and Threads

Repository Class

Transaction Management Overview

Restrictions for Microsoft Jet Repository Databases

When Microsoft® Jet manages a repository database, a special restriction applies to the use of multiple threads. Only the Jet-managed thread that created an open repository instance to a repository database can use the instance. In other words, if your repository database is managed by Jet, construct your application as if the repository were using the apartment thread model.

For more information about using Jet repository databases, see Reading Repository Data Outside of a Transaction.

See Also

Managing Transactions and Threads

Synchronizing Commit Operations

Synchronizing Commit Operations

When programming a multithreaded repository application, synchronize repository commit operations between application threads. Specifically, synchronize the use of the following methods:

- The **Count** method of the **IRelationshipCol** interface
- The **ObjectInstances** method of the **IClassDef** interface
- The **ObjectInstances** method of the **IInterfaceDef** interface
- The **ExecuteQuery** method of the **IRepositoryODBC** interface

See Also

IClassDef::ObjectInstances

IInterfaceDef::ObjectInstances

IRelationshipCol::get_Count

IRepositoryODBC::ExecuteQuery

Managing Transactions and Threads

Restrictions for Microsoft Jet Repository Databases

Meta Data Services Programming

Managing Workspaces

A workspace is a restricted view of the contents of a Microsoft® SQL Server™ 2000 Meta Data Services repository. You define which objects are contained in the workspace. You also define which version of each object is used. You can only have one version of any object checked out to a workspace at one time.

The following topics describe workspace management capabilities.

Topic	Description
Workspace Management Overview	Introduces a workspace and explains the reasons for using one.
Objects Within Workspaces	Describes how objects can be used within a workspace.
Workspace Context	Describes the scope of a workspace, and provides basic information about navigation. Workspace Context also details the operational differences between a repository instance and a workspace.
Accessing Objects in a Workspace	Explains the various ways in which an object can be retrieved within a workspace.
Manipulating Workspaces	Describes how to manipulate a workspace object.
Manipulating Objects in a Workspace	Describes how to manipulate an object within a workspace.

See Also

Versioning Objects

Workspace Management Overview

Each repository can contain multiple workspaces. A workspace is a restricted view of the contents of a repository. The view is restricted for two reasons:

- A workspace can contain only those repository object versions that you explicitly include in it. Thus, if you set up a workspace to apply to a functional area, you can include in that workspace only objects that pertain to that area.

- A workspace can contain only one version of any repository object. Thus, the workspace provides a simple view of the repository's data in which only one version of each object exists. In effect, operating within a workspace simplifies your environment because you do not need to choose among several versions of the same object. When you retrieve an object from a workspace, the repository engine returns the specific version of that object that is present in the workspace.

Although the view of repository data in a workspace is restricted, operating in the context of a workspace is liberating for two reasons:

- A workspace reveals only the objects that are important to you, effectively yielding a custom view of the repository.

- Because a workspace contains at most one version of any object, your within-workspace operations can avoid much of the complexity of a multiversion environment.

See Also

IRepositoryObjectVersion Interface

Managing Workspaces

RepositoryObjectVersion Object

Objects Within Workspaces

An object version can participate in a workspace in two ways:

- An object version can be present in the workspace. That is, the workspace can contain the object version.

  Each workspace has a **Versions** collection containing the object versions present in the workspace. Use the **Add** method to include an object version in a workspace; use **Remove** to exclude an object version.

  Note An object version can be present in more than one workspace. Each object version has a **Workspaces** collection containing the workspaces in which the object is present.

- An object version can be checked out to a workspace.

  When an object version is checked out to a workspace, you can modify that object version only while operating in the workspace. Each workspace has a **Checkouts** collection containing the object versions checked out to the workspace. Use the **Checkout** and **Checkin** methods to control the contents of the **Checkouts** collection.

  Note An object version can be checked out to no more than one workspace, and it can be checked out only to a workspace in which it is already present. Even when you check out an object version to a workspace, that object version can remain present in many other workspaces.

Workspaces support various kinds of collections that determine the scope of a workspace. For more information about the collections you can access in a workspace, see Retaining Workspace Context.

See Also

Managing Workspaces

Workspace Context

Workspace Context

When you operate within a workspace, the workspace provides a specific context for your work. That context includes both the **Versions** collection and the **Checkouts** collection for that workspace. If an object version is not part of either of these collections, it is not part of the context for that workspace.

The behavior of a repository method can vary depending on whether you invoked the method from within the context of a workspace or from within the more general context of a repository instance. For example, if you materialize the **ObjectInstances** collection for a class, the collection can consist of two different sets of items:

- If you are operating within the context of a workspace, the collection contains one item for each object in that workspace conforming to that class.

- If you are operating within the more general context of a repository instance, the collection contains one item for each object in the repository conforming to that class.

Similarly, if you invoke **get_Object**, two different things can happen:

- If you are operating within the context of a workspace, the repository engine follows a resolution strategy to yield the version of the object present in the workspace.

- If you are operating within the more general context of a repository instance, the resolution strategy yields the latest version of the object.

See Also

Establishing Workspace Context

Kinds of Version Collections

Establishing Workspace Context

When you create or open a repository instance, the repository engine returns a reference to the repository root object. From there, you can immediately begin to manipulate other repository objects. For example, you can invoke **get_Object** to materialize a reference to a specific repository object, or you can navigate from the root object to other repository objects. In either of these cases, the resulting references refer to run-time objects that exist within the general context of the open repository instance. Thus, if you invoke methods exposed by this object, the methods perform their work within that context.

On the other hand, you can first establish a workspace context before manipulating any repository objects. To establish a workspace context, you must materialize a workspace object in any of these ways:

- From the root object, you can materialize the **Workspaces** collection and then retrieve a particular workspace from the resulting collection.

- You can invoke the **get_Object** method to explicitly retrieve a reference to the workspace object in whose context you want to operate.

- From the workspace definition object you can use the **ObjectInstances** method to establish a collection of all workspaces in the repository. You can then retrieve a particular workspace from that collection.

After you have a reference to the workspace object, you can operate within the context of that workspace.

To retrieve an object directly within the context of the workspace, you can invoke the **get_Object** method as exposed by the workspace object. The **Workspace** class implements **IRepository2**, making methods like **get_Object** and **get_RootObject** equally available to a repository instance and the workspaces it contains.

To navigate to an object within the context of a workspace, start by invoking the **get_RootObject** method exposed by the workspace object, then navigate to

other objects that are related to the root.

Important differences exist between workspaces and repository instances. For more information about how these differences affect programming within a workspace context, see Workspaces and Repository Instances.

See Also

Managing Workspaces

Navigating a Repository

Retaining Workspace Context

Workspace Context

Retaining Workspace Context

As you navigate the objects present in the workspace, the repository engine retains the workspace context. In other words, if you retrieve an item from a relationship collection or a target object collection, the retrieved target item has the same context as the source item of that relationship.

However, only relationship collections and target object collections retain workspace context. If you retrieve an item from any **VersionCol** object, the reference that the repository engine returns to you has the context of the open repository instance in which you are operating. The object reference does not have an in-workspace context.

For example, suppose that within the context of a workspace, you have a reference to the root object, and you perform these steps:

1. From the root object, you navigate to a particular repository object.

   As you navigate to each object along the navigation path, the repository engine returns whichever object version is present in the workspace. At each step, the reference that the repository engine returns preserves the workspace context.

2. At some point along the navigation path, you materialize the **PredecessorVersions** collection of an object version. Then, you retrieve the first item in that collection.

   The repository engine returns a reference to the oldest predecessor of the object version. Because the **PredecessorVersions** collection is a **VersionCol** object rather than a relationship collection or a target object collection, this reference does not preserve the workspace context. All subsequent manipulations of and navigations from this object occur within the general context of the open **Repository** instance.

See Also

Workspaces and Repository Instances

In many respects, operating within a workspace is just like operating within a larger repository instance. Both the **Workspace** class and the **Repository** class implement the **IRepository2** interface. There are, however, some important differences:

- Some methods exposed by the **IRepository2** interface apply only to **Repository** instances, not to workspaces. You cannot call **Open** or **Create** on a workspace object.

- Unlike repository connections, workspaces are named persistent repository objects. Thus, workspaces can be created, used across multiple sessions, and deleted, if necessary.

See Also

Establishing Workspace Context

IRepository2 Interface

Managing Workspaces

Retaining Workspace Context

Workspace Context

Accessing Objects in a Workspace

You can retrieve a repository object in the context of a workspace by doing one of the following:

- Retrieving the object using the **Object** property that the workspace object exposes.
- Retrieving the root object using the **RootObject** property that the workspace object exposes, and then navigating to other objects by traversing relationship collections.
- Retrieving the object from the workspace **Versions** collection.

When you retrieve an object using any of these alternatives, you retrieve the specific object version that is included in the workspace. For more information, see Workspace Context.

See Also

Establishing Workspace Context

Managing Workspaces

Retaining Workspace Context

Manipulating Workspaces

Operations on workspaces are supported at both the COM level and the Automation level. You can perform the following operations on a workspace.

To	Use
Enumerate the workspaces in a repository instance	The **Workspaces** collection of the **IWorkspaceContainer** interface that is exposed by the root repository object.
Create a workspace	The **CreateObject** method of the **IRepository** interface that is exposed by the open repository instance. Use the **Add** method for the **Workspaces** collection to add the workspace to the collection of workspaces.
Delete a workspace	The **Delete** method of the **IRepositoryItem** interface that is exposed by the workspace object. If you attempt to delete a workspace that contains checked out objects, the delete will fail.
Retrieve the root object in a workspace	The **RootObject** property of the **IRepository** interface that is exposed by the workspace object.
Enumerate the repository objects contained in a workspace	The **Contents** collection of the **IWorkspace** interface that is exposed by the workspace object.
Enumerate the checked out objects in a workspace	The **Checkouts** collection of the **IWorkspace** interface that is exposed by the workspace object.

See Also

Managing Workspaces

Manipulating Objects in a Workspace

Manipulating Objects in a Workspace

Repository objects implement the **IWorkSpaceItem** interface in order to support workspace-related capabilities. The **IWorkSpaceItem** interface is available at both the COM level and the Automation level. Given a specific version of a repository object, you can perform the workspace-related operations listed in the following table.

To	Use
Determine whether an object version is checked out to a workspace	The **CheckedOutToWorkspace** property of the **IWorkSpaceItem** interface.
Determine which workspaces contain a particular object version	The **Workspaces** collection of the **IWorkSpaceItem** interface that is exposed by the object version.
Add an object version to a workspace	The **IWorkspace** interface to obtain access to the **Contents** collection. Then use the **Add** method of the **Contents** collection to add an object version to the workspace.
Remove an object version from a workspace	The **IWorkspace** interface to obtain access to the **Contents** collection. Then use the **Remove** method of the **Contents** collection to remove the object version from the workspace.
Check an object version out to a workspace	The **CheckOut** method of the **IWorkSpaceItem** interface that is exposed by the object version.
Check an object version in from a workspace	The **CheckIn** method of the **IWorkSpaceItem** interface that is exposed by the object version.

See Also

Managing Workspaces

Manipulating Workspaces

Meta Data Services Programming

Handling Errors

Error information is available to programs that use repository engine COM or Automation interfaces. This section gives an overview of repository error handling and presents techniques for accessing repository error information.

Topic	Description
Error Handling Overview	Describes how the repository engine implements error queue and error handling
Accessing Error Information at the Automation Level	Explains how to access repository interfaces at the Automation level
Accessing Error Information at the COM Level	Explains how to access repository interfaces at the COM level
Persisting Error Queue Information	Describes how to retain error queue information
Repository Errors	Documents repository engine error codes and messages in alphabetical or numerical format

Error Handling Overview

If you are programming with COM interfaces, you can use interfaces to work with the error queue and handle errors. Equivalent functionality is not available to Automation objects. You cannot manage the error queue or its contents from an Automation object.

Error Handling

Error handling applies to methods on repository engine objects. In the repository API, COM interface members return an **HRESULT** return value that indicates whether a method completed successfully. If a repository interface member fails to complete successfully, an error object that contains details about the failure is created.

Error objects conform to the **REPOSERROR** data structure. For more information about the data structure of repository errors, see REPOSERROR Data Structure.

Error Queues

An error queue is a collection of error objects. Each repository instance maintains a single error queue. When an error is generated by a repository interface method, the error is added to the error queue of the current repository instance. If multiple errors occur as a result of a single member invocation, all of the errors are added to the error queue of the current repository instance.

You can have multiple repository instances and associated error queues active at one time. Multiple repository instances can be connected to the same repository database. Repository instances can originate from the same or from different processes. A single process can create multiple repository instances.

The repository error queue is a transient object; that is, the contents of the queue are valid only within the same operation in which the error occurred. Subsequent interactions with any repository object will automatically clear the error queue.

To work with the queue, use **IRepositoryErrorQueueHandler** to create an error

queue, assign an error queue to a thread of execution, or retrieve an interface pointer to a thread's currently assigned error queue.

To manage errors within a queue, use **IRepositoryErrorQueue** for repository objects and **IEnumRepositoryErrors** enumeration objects.

See Also

Accessing Error Information at the COM Level

IEnumRepositoryErrors Interface

IRepositoryErrorQueue Interface

IRepositoryErrorQueueHandler Interface

Persisting Error Queue Information

Repository Class

Accessing Error Information at the Automation Level

When a repository interface member generates an error, Automation programs can access the repository error object to obtain error information. For more information about the repository error object, see REPOSERROR Data Structure.

Visual Basic

In Microsoft® Visual Basic®, you use the global **Err** object to handle errors. The first error in a repository error queue is the error that is placed into the **Err** object. For each error, you can use the **On Error** statement to invoke an error handler when an error is encountered. In the error handler, access the properties of the global **Err** object to retrieve the error information. Only the first error in the repository error queue is accessible through the **Err** object. For more information about the global **Err** object, see the Visual Basic documentation.

Script Objects

A predefined variable for script objects, **ReposErr**, can be used to report a result and an error description that you provide. **ReposErr** enables you to create an error to return to the calling application.

See Also

Accessing Error Information at the COM Level

Handling Errors

ScriptDef Object

Accessing Error Information at the COM Level

COM programs can access all of the errors in a repository error queue. You can use **IRepositoryErrorQueue** to select, insert, or remove errors in a repository error queue. You can also persist queue information if you want to return to it after working with other error queues or repository objects.

To access the errors in a COM program

- Use the **QueryInterface** method on any repository object interface to obtain an **IReposErrorQueueHandler** interface pointer. There is an **IRepository** interface pointer associated with each instance of the **Repository** class.

  -or-

- Call the **GetErrorQueue** method of the **IReposErrorQueueHandler** interface to obtain an **IRepositoryErrorQueue** interface pointer.

  -or-

- Use the **Count** method of the **IRepositoryErrorQueue** interface to get the number of elements in the error queue, and the **Item** method to retrieve the error information for each error in the queue.

The repository engine also provides an enumeration interface for errors called **IEnumRepositoryErrors.**

See Also

Error Handling Overview

IEnumRepositoryErrors Interface

IRepositoryErrorQueue Interface

IReposErrorQueueHandler Interface

Persisting Error Queue Information

Repository Class

Persisting Error Queue Information

If you are programming with COM interfaces, you can retain error queue information while you switch to other error queues or work with other **repository** objects.

You can access only one repository error queue at a time. When you switch from one error queue to another, several things occur automatically:

- The **IRepositoryErrorQueue** interface reference to the first error queue is automatically released.
- If that reference is the only remaining reference to the interface, the error queue is destroyed.
- An **IRepositoryErrorQueue** interface reference to the second error queue is automatically added.

Consequently, if you switch from one error queue to a second error queue and then back to the first error queue, the first error queue is destroyed and then re-created as an empty queue.

To switch between multiple error queues and retain all error queue information

1. Obtain an **IRepositoryErrorQueue** interface pointer for the error queue.
2. Explicitly increment the interface reference count using the **AddRef** method that is associated with the error queue.

Note You must repeat these steps for each error queue.

The error queues will be retained as long as you hold these explicit interface references.

To switch back and forth between error queues

- Use the **SetErrorQueue** method of the **IReposErrorQueueHandler** interface. When the error queue information is no longer needed, use the **Release** method to remove the explicit interface references.

See Also

Accessing Error Information at the COM Level

Error Handling Overview

IRepositoryErrorQueue Interface

IReposErrorQueueHandler Interface

Meta Data Services Programming

Optimizing Repository Performance

The biggest factor that affects repository engine performance is the number of round trips the repository engine makes to the underlying database system. As a result, reducing the number of round trips is the single best solution to improving repository engine performance. To be able to minimize the number of round trips, you must understand the repository engine data access strategy. Once you understand this strategy, you can use the tips and hints listed here to improve repository performance.

Data Access Strategy

The repository engine maintains a cache of repository objects. When accessing an object by object identifier or by way of a relationship, the engine first looks in its cache. Similarly, the engine maintains a cache of relationship collections. When accessing a collection on a repository object, the engine first looks in its cache.

Because round trips to the database are expensive, the engine fetches and updates data in batches. For example, when you access a relationship collection, the engine fetches all the relationships in the collection. The engine caches the updates that a transaction performs, and (unless the cache overflows) sends them to the database only when the transaction commits. There are many other cases, too numerous to mention here, where the engine performs batching.

Many of the engine's caching and batching strategies are universally beneficial and require no special consideration when writing an application. However, sometimes the application's usage pattern can have a significant performance effect. The benefits of the caching and batching strategies often require a tradeoff of functional flexibility; consequently, none of the hints can be blindly applied without consideration of possible tradeoffs.

Tips and Hints

The following table lists the tips and hints that you can use to improve repository engine performance.

Topic	Description
General Hints to Improve Performance	Provides general hints about using cached data and storing data
Retrieval Hints	Discusses alternate ways of retrieving data from a repository database
Update Hints	Provides information about update behavior that you can use to improve engine performance
Versioning Hints	Offers a versioning tip that improves performance
Run-Time Tuning	Discusses options that you can set to improve run-time performance
Adjusting Cache Aging for Repository Objects	Explains how you can adjust ageout behavior for specific kinds of rows
View Hints	Offers hints that improve the performance of views when querying a database

See Also

Repository Engine

Repository Databases

Storage Strategy in a Repository Database

General Hints to Improve Performance

The following hints cover cache reuse and model storage strategies.

Reusing Cached Rows

An application can have multiple **repository** instances (for example, database sessions) open at the same time. Although objects are not shared between sessions, cached rows are shared. This offers some opportunities for increased parallelism.

The repository engine shares a cache by default, so this optimization tip is already at work. Be aware that if you set the connection flag REPOS_CONN_NEWCACHE you will lose the benefit of this optimization technique.

Reusing an Interface Instance

It is more efficient to cache the result of **IRepositoryItem::Interface** than to call it many times in a row on the same object and interface. This avoids the cost of a COM object creation and type information lookup.

Storing an Information Model

If you are creating or extending an information model programmatically, you can improve performance by minimizing the number of tables that you use to store properties. You can minimize the number of tables by mapping multiple interfaces to the same table. To do this, before you commit the transaction that is used to create your information model, set the **TableName** property for each interface definition object to the same name. Since the engine must issue a separate SQL query for each table it accesses, when you reduce the number of tables, you reduce the number of database round trips. However, this may cost some space for objects that do not support or populate all the interfaces.

See Also

Optimizing Repository Performance

Retrieval Hints

A fast way to access an object is to use its object identifier. This kind of retrieval is only possible for well-known objects that your application expects to find, such as a container (that is, folder or package) object that is the root of a container hierarchy.

Using Relationships to Fetch Objects

If a given set of objects is usually loaded together, it is helpful to have a relationship collection that points to those objects. If you access the objects through that collection, the engine will load them in one round trip.

Think twice before navigating to a collection that contains only one or two objects of interest. Navigating to the collection loads the entire collection. Instead, try to find another way to navigate to those objects. **ExecuteQuery** to fetch objects provides one such alternative.

Collection Loading Hints

When loading or exporting objects, specify the maximum number of objects in each collection. This is an effective way to allow the repository engine to preload all the object collections for each repository object.

You can also set the **IReposOptions** OPT_PRELOAD_COL_MODE and OPT_EXPORT_MODE options to preload objects in a collection. For more information about option values and descriptions, see IReposOptions Options Table.

Using ExecuteQuery to Fetch Objects

When you know the exact set of objects you want, **IRepositoryODBC::ExecuteQuery** is often a faster way to find the objects than navigating to them by way of collections, because it usually requires many fewer round trips. For convenience, consider writing some view definitions to insulate application programmers from the complexity of the relationship table

(**RTblRelationships**) and type definitions (for example, those stored in **RTblClassDefs** and **RTblIfaceDefs**). If the query-update ratio warrants it, consider adding indexes to the repository SQL interface tables (the ones to which the properties associated with that interface are mapped).

ExecuteQuery can be run asynchronously, in which case the call returns immediately. Later, you use **IObjectCol2** to determine whether the collection that is being loaded is ready to read.

You can use **ExecuteQuery** to explicitly tell the repository engine to prefetch certain objects. However, in addition to calling **ExecuteQuery** for at least one object in the **ObjectCollection** that the query returns, you must access a property on each interface you want to access. This tells the engine to prefetch the properties on those interfaces for all the objects in the collection. As an aside, the engine flushes all updates to the database before running **ExecuteQuery**, so the query is reading exactly the current database state.

Using Named Relationships to Fetch Objects

If an object has the same name in all contexts, make sure its class supports **INamedObject**. This makes it more efficient to fetch the name. That is, the engine fetches the name from **INamedObject::Name** instead of a name from any of the incoming relationships. Note that an update of the name causes an update to all naming relationships pointing to the name.

If an object supports **INamedObject**, the most efficient way to set the **Name** property on the object only (and not on any of its incoming relationships) is to explicitly **QueryInterface** for **INamedObject** and set its **Name** property. For example:

```
Dim oReposObj as RepositoryObject
Set oReposObj.Interface("INamedObject").Name = "Any Name"
```

Note that since the names of the relationships to the object are not updated here, you cannot later fetch by name from the collection. Rather, you have to enumerate the collection and check each object's name. Also note that the property **Name** corresponds to the dispatch ID **DISPID_ObjName** (not **DISPID_Name**).

It is more efficient to follow a relationship in the origin-to-destination direction

than vice versa. This is because the physical representation of relationships in the relationship table is biased in this direction. So, traverse relationships in this direction whenever you have a choice.

See Also

IReposOptions Interface

Optimizing Repository Performance

Repository SQL Schema

Repository SQL Tables

Update Hints

Because the repository engine sends updates to the database in a batch at transaction commit time, a single long transaction is preferable to many small ones. This is true for any combination of inserts, deletes, and updates. For moderate-sized objects, you should be able to update 25,000 objects within one transaction without hitting cache size limitations.

Use the automatic delete propagation feature of relationships wherever possible. This allows the engine to delete objects in a batch.

See Also

Propagating Deletes

Optimizing Repository Performance

Versioning Hints

Do not freeze a version until it is necessary. The engine knows that an unfrozen version can have no successors, and it exploits this knowledge in its access strategies.

See Also

Optimizing Repository Performance

Run-Time Tuning

The repository engine provides excellent performance for typical applications. Some of these optimizations, because they are generic in nature, may not be well tuned for certain applications and may even be detrimental to their performance. The repository engine allows run-time performance tuning that is specific to each application.

The **IReposOptions** interface has the following methods.

Method	Description
SetOption	Sets numerous options, all of which impact repository engine performance in some way. For more information about option values and descriptions, see IReposOptions Options Table.
GetOption	Retrieves a current option value.
ResetOptions	Resets all options to their default values.

See Also

IReposOptions Interface

Optimizing Repository Performance

Adjusting Cache Aging for Repository Objects

The repository engine cache aging mechanism ensures that the engine's client cache is automatically refreshed periodically so that clients can see up-to-date values. The mechanism also affects performance, because the next access to an aged-out entry must be fetched again from the database system.

A new mechanism for aging out rows of different types in the repository engine is used in version 3.0. Different strategies are offered for rows that are referenced, recently used, cached, and static. Ageout strategies are specified based on **IReposOptions** options that you set. These options include **OPT_AGEOUT**, **OPT_TIM_AGEOUT**, and **OPT_PRELOAD_AGEOUT**. For more information about option values and descriptions, see IReposOptions Options Table.

See Also

IReposOptions Interface

Optimizing Repository Performance

View Hints

View definitions can affect performance in two ways: during SQL view generation and when Microsoft® SQL Server™ 2000 compiles a SQL view before executing it. The following hints can help you achieve better performance along these two dimensions.

- Choose an unresolved view (a view that does not support version resolution) if version information is unimportant (for example, when you know that all objects are version one). The repository engine does not perform version resolution if the SQL view is flagged as unresolved.

- When you have a choice between using a class-based view or an interface-based view, choose the class-based view. Interface-based views have an extra join that determines which classes implement the interface. Using a class-based view avoids the performance hit of processing the extra join.

- Choose an interface-based view over a class-based view when querying a small set of interfaces where the key (IntID) is specified. This choice is often preferable because the compilation time can be so much smaller for interface-based views.

- When navigating a relationship, performing the query on a junction view often runs faster than when you represent the relationship as a foreign key on a class or interface view. Using a junction view yields faster performance on average.

- If you have a view that includes a text field, and you reference the text field in a SELECT clause, then you are not allowed to use SELECT DISTINCT. As a result, the query optimizer cannot eliminate certain redundant joins. A solution is to use a nested query on two interface-based views. The inner query uses DISTINCT and includes IntID in the

SELECT clause, but does not reference the text field. This causes the inner query to reference a presumably smaller number (specifically, the IntID), which then joins with the interface-based view that contains the text field.

See Also

Generating Views

Optimizing Repository Performance

Repository Identifiers

Meta Data Services Programming

Storage Strategy in a Repository Database

The database storage model in Microsoft® SQL Server™ 2000 Meta Data Services differs from the run-time object model. While the run-time object model is designed to accommodate run-time operations conveniently, the database model is designed to accommodate storage efficiently.

To save space in the database, Meta Data Services can sometimes store a single copy of a property value, even if that property value describes many object versions. Similarly, Meta Data Services can sometimes store a single copy of a relationship, even if many different object versions have that relationship.

Meta Data Services anticipates which object versions are especially likely to share property values and relationships and which object versions are less likely to. The repository engine uses these guidelines:

- Two versions of the same object are likely to share property values and collections, but two versions of different objects are less likely to do so. In other words, if two object versions are not on the same version graph, they are not especially likely to share property values or relationships. For more information, see Version Graph.

- Within a version graph, two versions that are near each other are more likely to share values; two versions that are far apart are less likely to share values. The repository engine arranges each version graph into branches. For more information, see Branches in the Version Graph. Each branch contains versions that are especially likely to share values.

- Although property values and collections can change, they do not change back and forth frequently. More commonly, a value holds for a few consecutive versions of an object, and then that value changes to a new value, which holds for a few more versions of the object. Thus, when Meta Data Services stores a property value, it stores the property value for an entire range of object versions. For more information, see Ranges in the Version Graph.

Similarly, in a single row of the **RTblRelships** table, Meta Data Services can indicate that every object version in a range (of origin object versions) has a relationship to every object version in a range (of destination object versions).

The repository SQL tables store the physical data of a repository. For more information about object and object version storage, see RTblVersions SQL Table and Interface-Specific Tables

See Also

Connecting to and Configuring a Repository

Repository Databases

Repository SQL Schema

Repository SQL Tables

Branches in the Version Graph

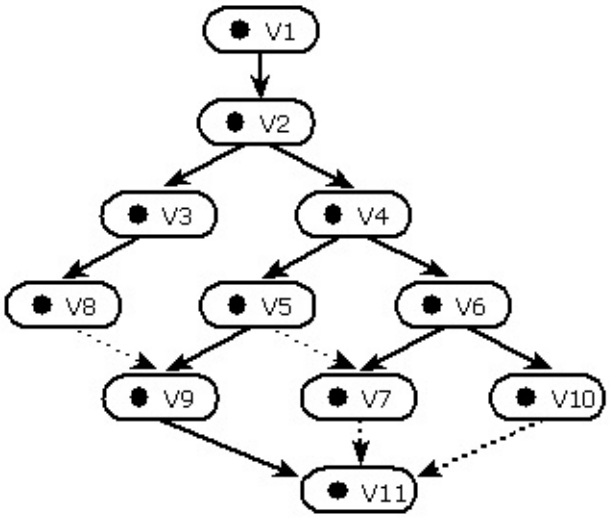
Within the database, the repository engine partitions each version graph into branches. Each branch contains object versions that are especially likely to share property values and target object versions with each other.

A branch of a version graph is a sequence *S* of *n* (*n* > 0) object versions such that:

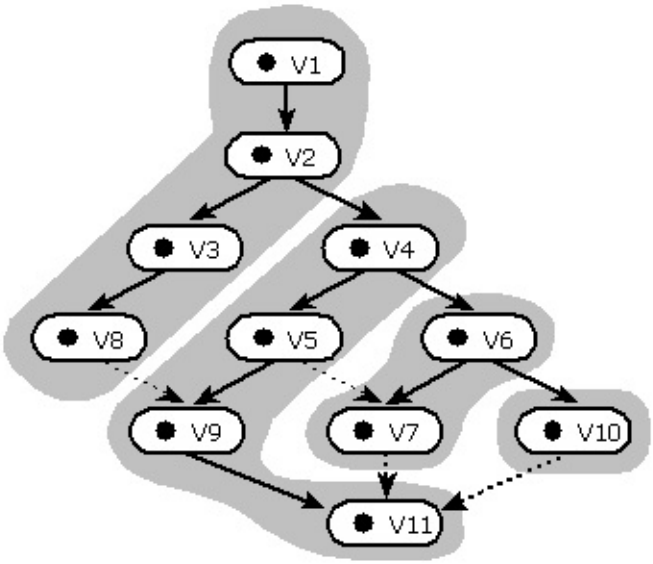
$S = x_1, x_2, x_3, ..., x_n$

and x_1 is the creation predecessor of x_2
and x_k is the creation predecessor of x_{k+1} ($1 < k < n$-2)
and x_{n-1} is the creation predecessor of x_n.

According to this definition, each object version can be in the same branch with at most one of its creation successors. For example, in the following version graph, Version 3 and Version 4 cannot be in the same branch, because they are both creation successors of Version 2.



In fact, this version graph has four branches, as shown in the following figure.



Every version graph has at least one branch, the branch beginning with the first version of the object.

Each object version is a member of a single branch; branches do not overlap.

When you create a new version of an object, the repository engine tries to add the newly created version to the same branch as its predecessor. If that branch already includes a successor to the creation version, the repository engine creates a new branch. The newly created object version is the only element of the new branch.

For example, if you create a new version from Version 8, the repository engine creates the new version and adds it to the branch containing Version 8. But if you create a new version from Version 3, the repository engine creates a new branch for the new version, because Version 3's branch already includes a creation successor of Version 3.

Each branch represents a set of object versions that are especially likely to share property values and have identical relationships. If two object versions exist on separate branches, the repository does not save any space in the database even if those versions share values for all of their properties and have identical collections for all of their collection types. For the repository engine to save space, the similar objects must exist on the same branch. For this reason, the repository engine attempts during **CreateVersion** to assign the new version to an existing branch. The fewer the branches, the higher the likelihood that space can be saved in the database.

The repository engine never moves an object version from one branch to another. After assigning an object version to a branch during the **CreateVersion** method, that object version remains on that branch until the object version is deleted.

See Also

IRepositoryObjectVersion::CreateVersion

RepositoryObjectVersion CreateVersion Method

Storage Strategy in a Repository Database

Version Graph

Ranges in the Version Graph

To save space in the database, the repository engine can associate a property value or relationship with an entire range of object versions. A range of object versions is a set of consecutive elements of a branch. For more information, see Branches in the Version Graph.

To refer to a range, a row of a repository SQL table must include the following four values:

- The Version Graph. That is, the row must refer to the repository object. Use the internal identifier of the object.
- The Branch. This is the portion of the version graph containing the range. Use the branch identifier of the branch.
- The Range Start. This is the element within the branch where the range starts. Use the version-within-branch identifier of the object version.
- The Range End. This is the element within the branch where the range ends. Use the version-within-branch identifier of the object version, or use the special constant VERINFINITY (hex 7fffffff), to indicate an unbounded range.

The repository engine uses unbounded ranges to indicate that properties apply to a set of object versions that can grow as you make new object versions using **CreateVersion**.

For more information about how the repository engine uses unbounded ranges, see Interface-Specific Tables.

See Also

IRepositoryObjectVersion::CreateVersion

Storing Relationships

The run-time object model and the storage schema differ significantly. These differences are most apparent when you query a repository database for information about relationships. In fact, the storage of relationships and the run-time manipulation of relationships differ significantly.

The repository engine uses the **RTblRelships** table to store information about relationships. An individual row of the table can be any of the following:

- A description of an individual version-to-version relationship
- A description of a set of version-to-version relationships
- A description of sequencing and pinning information for a single origin-versioned relationship

See Also

Repository SQL Schema

Repository SQL Tables

RTblRelships SQL Table

Storage Strategy in a Repository Database

Interface-Specific Tables

When you create an information model, the repository engine enlarges the database schema to accommodate the new kinds of data. The additional tables that the repository engine adds are called the extended schema. Generally, the repository engine creates one table for each new interface you create. Several interfaces, however, can share a table. For more information, see Information Model Creation Issues.

Each row of an interface-specific table indicates that a set of property values applies to a particular range of object versions. For more information, see Ranges in the Version Graph.

The primary key of any interface-specific table consists of three columns: **IntID**, **Z_BranchID_Z**, and **Z_VS_Z**, as shown in the following table.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the object
Z_BranchID_Z	**RTBrID**	Indicates the branch of the version graph containing the range to whose items the property values in this row apply
Z_VS_Z	**RTVerID**	A version-within-branch identifier indicating the lower bound of the range to whose items the property values in this row apply
Z_VE_Z	**RTVerID**	A version-within-branch identifier indicating the upper limit of the range to whose items the property values in this row apply
(User-supplied column name for Interface-Specific Property 1)	(User-supplied data type)	A column that corresponds to a property you defined in your information model
(User-supplied	(User-	Other columns that correspond to other

column name for Interface-Specific Property *n*...)	supplied data type)	properties you defined in your information model

Each row indicates a range and a set of property values. Every object version in the range is described by every property value.

See Also

Branches in the Version Graph

Example: Rows of Interface-Specific Tables

Repository SQL Schema

Repository SQL Tables

Storage Strategy in a Repository Database

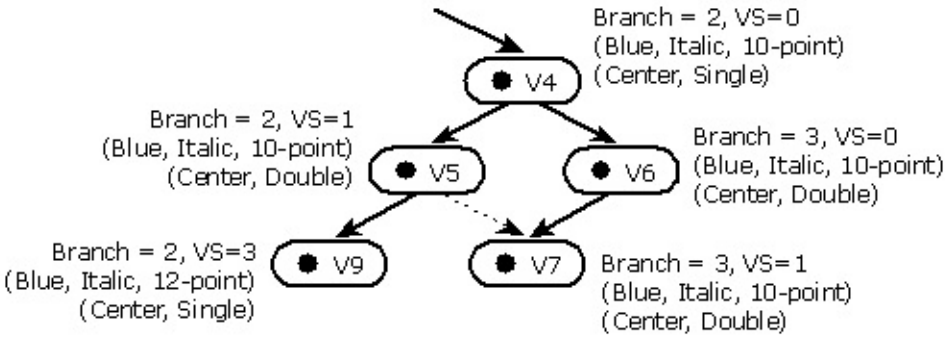
Version Graph

Example: Rows of Interface-Specific Tables

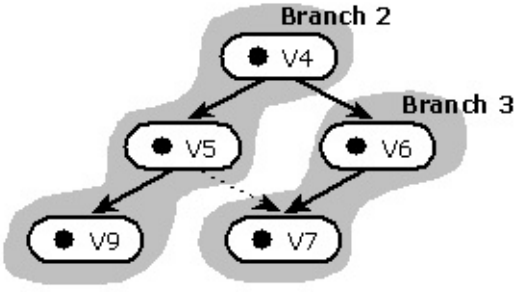
Consider the version graph for a typical object. Suppose the object is part of a user-installed information model; it conforms to the **CParagraph** class. Suppose further that the **CParagraph** class implements two interfaces:

- **IFont** exposes the properties **Color**, **Style**, and **PointSize**.
- **IParagraph** exposes the properties **Alignment** and **Spacing**.

The following figure shows a portion of the version graph, along with properties for each version of the object.



Assume that the object's internal identifier is 7. Assume that Version 7 and Version 9 are leaf nodes; they have no successors. Also assume that there are two branches containing these object versions, as shown in the following figure.



The properties for these object versions are stored in two separate interface-specific tables. The table for the properties of the **IFont** interface includes the following rows.

IntID	Z_BranchID_Z	Z_VS_Z	Z_VE_Z	Color	Style	Point Size
7	2	0	1	Blue	Italic	10
7	2	3	VERINFINITY	Blue	Italic	10
7	2	0	VERINFINITY	Blue	Italic	10

The first row in the preceding table indicates that the properties (Blue, Italic, 10-point) apply to each object version in a range within Branch 2 that begins at Version 4 and ends at Version 5.

The second row indicates that the properties (Blue, Italic, 12-point) apply to each object version in a range within Branch 2 that begins at Version 9 and ends at the end of the branch.

Similarly, the third row indicates that the properties (Blue, Italic, 10-point) apply to each object version in a range within Branch 3 that begins at Version 6 and ends at the end of the branch.

The following table for the properties of the **IParagraph** interface includes the following rows.

IntID	Z_BranchID_Z	Z_VS_Z	Z_VE_Z	Alignment	Spacing
7	2	0	0	Center	Single
7	2	1	VERINFINITY	Center	Double
7	3	0	VERINFINITY	Center	Double

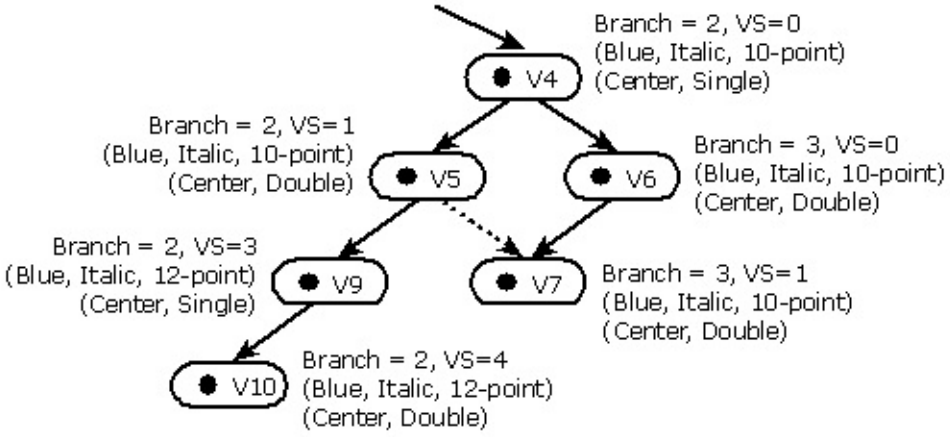
The first row of the preceding table indicates that the properties (Center, Single) apply to each object version in a range within Branch 2 that begins at Version 4 and ends at Version 4, a single-version range.

The second row indicates that the properties (Center, Double) apply to each object version in an unbounded range within Branch 2 that begins at Version 5.

Similarly, the third row indicates that the properties (Center, Double) apply to each object version in an unbounded range within Branch 3 that begins at Version 6.

Within the **Z_VE_Z** column, VERINFINITY indicates that the range has no upper bound. Thus, if you enlarge a branch (by invoking the **CreateVersion** method on the branch's newest object version) the creation predecessor's property values will automatically apply to the newly created version.

For example, suppose you invoke the **CreateVersion** method on Version 9, yielding a version graph, as shown in the following figure.



In the preceding figure, the new object version is on the same branch as its predecessor, and has the same properties as its predecessor. To apply these existing property values to the newly created object, the **CreateVersion** method does not need to modify the **IFont**-specific property table or the **IParagraph**-specific property table, because those tables contained rows that applied those property values to ranges with no upper bound.

See Also

Branches in the Version Graph

Ranges in the Version Graph

Repository SQL Schema

Repository SQL Tables

Storage Strategy in a Repository Database

Version Graph

Meta Data Services Programming

Using OLE DB Scanner

OLE DB Scanner is a feature of Microsoft® SQL Server™ 2000 Meta Data Services that imports relational database schema information from an OLE DB data source and populates instances of the Open Information Model (OIM) Database Schema model in a repository database. This section describes the OLE DB Scanner for Meta Data Services. The following topics provide more detailed information about OLE DB Scanner.

Topic	Description
OLE DB Scanner Overview	Describes OLE DB Scanner and how to apply it.
Supported OLE DB Schema Rowsets in OLE DB Scanner	Lists the OLE DB rowsets and indicates which are supported by OLE DB Scanner.
Navigating the Schema in OLE DB Scanner	Shows a Microsoft® Visual Basic® code example that navigates an OLE DB rowset using the repository API.
Schema Versioning in OLE DB Scanner	Explains versioning behavior for rowsets imported by OLE DB Scanner into a repository database.
Data Type Mappings in OLE DB Scanner	Lists data type equivalents for OLE DB data types and repository data types.
OLE DB Scanner Reference	Provides API reference topics for OLE DB Scanner interfaces.

Meta Data Services Programming

OLE DB Scanner Overview

OLE DB Scanner imports database schema information from an OLE DB data source and populates instances of the Open Information Model (OIM) Database Schema model in a repository database. OLE DB Scanner works with OLE DB providers. When you pass an OLE DB provider to the scanner, it examines the schema and creates a set of corresponding instance objects in the repository database.

OLE DB Scanner is written as a Microsoft® ActiveX® DLL. The scanner provides one dual interface, **IRepOLEDBScanner**. The provided interface is declared as dual so that it can be called from both COM and Automation clients.

IRepOLEDBScanner supports initial scans and rescans. It also supports clients that already have an initialized OLE DB connection. For more information, see OLE DB Scanner Reference.

Meta Data Services Programming

Supported OLE DB Schema Rowsets in OLE DB Scanner

Thirty types of database schema information can be fetched from an OLE DB data source. The Database object name column in the following table shows the name of the database objects that can be fetched. The Supported column indicates whether the item is supported by OLE DB Scanner. If an item is not supported, there is no corresponding item in the Open Information Model (OIM) to capture it. For more information about property descriptions and OLE DB type information, see the OLE DB documentation.

Database object name	Supported
ASSERTIONS	Yes
CATALOGS	Yes
CHARACTER_SETS	No
CHECK_CONSTRAINTS	Yes
COLLATIONS	No
COLUMN_DOMAIN_USAGE	Yes
COLUMN_PRIVILEGES	No
COLUMNS	Yes
CONSTRAINT_COLUMN_USAGE	Yes
CONSTRAINT_TABLE_USAGE	Yes
FOREIGN_KEYS	Yes
INDEXES	Yes
KEY_COLUMN_USAGE	Yes
PRIMARY_KEYS	Yes
PROCEDURE_COLUMNS	Yes
PROCEDURE_PARAMETERS	Yes
PROCEDURES	Yes
PROVIDER_TYPES	Yes
REFERENTIAL_CONSTRAINTS	Yes
SCHEMATA	Yes
SQL_LANGUAGES	No

STATISTICS	No
TABLE_CONSTRAINTS	Yes
TABLE_PRIVILEGES	No
TABLES	Yes
TRANSLATIONS	No
USAGE_PRIVILEGES	No
VIEW_COLUMN_USAGE	No
VIEW_TABLE_USAGE	No
VIEWS	Yes

Meta Data Services Programming

Navigating the Schema in OLE DB Scanner

After the database schema has been scanned into a repository database, the schema can be easily navigated from Microsoft® Visual Basic® or Microsoft Visual C++® using the repository API. For example, the following Visual Basic code navigates in the following order: DataSource, Catalog, Schema, Table, Column, DataType.

```
Set IfD = Repos.object(OBJID_IDbmDataSource)
For Each datasource In IfD.ObjectInstances
   ...
   For Each catalog In datasource("_DataSource").DeployedCatalogs
      ...
      For Each schema In catalog("_Catalog").Schemas
         ...
         For Each table In schema("_Schema").Tables
            If QI(table, "IDbmTable") Then
               ...
               For Each column In table("_Table").Columns
                  ...
                  Set datatype = column("_Column").Attribute.Item(1)
               Next
            End If
         Next
      Next
   Next
Next

Private Function QI(o As RepositoryObject, name As String) As Boole
On Error GoTo Fail
      Dim r As RepositoryObject
      Set r = o.Interface(name)
      QI = True
```

```
        Exit Function
Fail:
    QI = False
End Function
```

Meta Data Services Programming

Schema Versioning in OLE DB Scanner

When scanning a database catalog that is already in a repository database (identified by identical catalog names), the scanner versions the schema at the lowest granularity of change. For example, take the following example data model of an **Authors** table with two columns.

Name	Data type	Size
au_lname	**Varchar**	20
au_fname	**Varchar**	40

Changing the data type of the **au_lname** column results in a new version of the column object and a relationship to the new data type object.

In general, adding or removing an element of a relationship collection requires that you create a new version of the origin object. For example, adding a column to the table results in a new version of the table with the new column object added to the elements collection. Relationships to existing columns are propagated.

Removing a column from the table results in a new version of the table with the column object removed from elements collection. Relationships to existing columns are propagated.

Name	Data type	Size
au_lname	**Varchar**	40
au_mname	**Varchar**	10

The following diagram shows the model for the revised table schema.

Meta Data Services Programming

Data Type Mappings in OLE DB Scanner

Each OLE DB column has an enumerated indicator that must be mapped to a DBMS data type instance. These instances, which are implemented by a class that supports the **IDbmDBMSDataType** interface, are created using the following mapping table and are assigned to columns using the **PROVIDER_TYPES** rowset.

OLE DB type indicator	Repository mapping	Remarks
DBTYPE_EMPTY	**DbmDBMSDataType**	None
DBTYPE_NULL	**DbmDBMSDataType**	None
DBTYPE_RESERVED	**DbmDBMSDataType**	None
DBTYPE_I1	**DbmTinyInt**	None
DBTYPE_I2	**DbmSmallInt**	None
DBTYPE_I4	**DbmInteger**	None
DBTYPE_I8	**DbmQuadInt**	None
DBTYPE_UI1	**DbmTinyInt**	OLE DB Scanner sets **IDtmNumeric.IsSigned** to False
DBTYPE_UI2	**DbmSmallInt**	OLE DB Scanner sets **IDtmNumeric.IsSigned** to False
DBTYPE_UI4	**DbmInteger**	OLE DB Scanner sets **IDtmNumeric.IsSigned** to False
DBTYPE_UI8	**DbmQuadInt**	OLE DB Scanner sets **IDtmNumeric.IsSigned** to False
DBTYPE_R4	**DbmReal**	OLE DB Scanner sets **IDtmNumeric.IsSigned** to True
DBTYPE_R8	**DbmDouble**	OLE DB Scanner sets **IDtmNumeric.IsSigned** to True

DBTYPE_CY	**DbmMoney**	None
DBTYPE_DECIMAL	**DbmDecimal**	OLE DB Scanner sets **IDtmNumeric.IsSigned** to True
DBTYPE_NUMERIC	**DbmNumeric**	OLE DB Scanner sets **IDtmNumeric.IsSigned** to True
DBTYPE_DATE	**DbmDate**	None
DBTYPE_BOOL	**DbmBit**	None
DBTYPE_BYTES	**DbmBinary or DbmVarBinary**	If **IsVariable** is set to True, OLE DB Scanner uses **DbmVarBinary** and sets **IDtmBinary.IsVariable** an **IDtmBinary.Length** to Tru
DBTYPE_BSTR	**DbmDBMSDataType**	OLE DB Scanner sets **IDtmString.Length** and **IDtmString.IsVariable** to
DBTYPE_STR	**DbmChar or DbmVarChar**	If **IsVariable** is set to True, OLE DB Scanner uses **DbmVarChar** and sets **IDtmString.IsVariable** an **IDtmString.Length** to Tru
DBTYPE_WSTR	**DbmNChar or DbmNVarChar**	If **IsVariable** is set to True, OLE DB Scanner uses **DbmVarChar** and sets **CharacterType** equal to **DTM_CHARACTER_TY UNICODE** and **IDtmString.IsVariable** an **IDtmString.Length** to Tru
DBTYPE_VARIANT	**DbmDBMSDataType**	None
DBTYPE_IDISPATCH	**DbmDBMSDataType**	None
DBTYPE_IUNKNOWN	**DbmDBMSDataType**	None
DBTYPE_GUID	**DbmDBMSDataType**	None
DBTYPE_ERROR	**DbmDBMSDataType**	None

DBTYPE_BYREF	**DbmDBMSDataType**	None
DBTYPE_ARRAY	**DbmDBMSDataType**	None
DBTYPE_VECTOR	**DbmDBMSDataType**	None
DBTYPE_UDT	**DbmDBMSDataType**	None
DBTYPE_DBDATE	**DbmDate**	None
DBTYPE_DBTIME	**DbmTime**	None
DBTYPE_DBTIMESTAMP	**DbmTimeStamp**	None

Meta Data Services Programming

Using XML Encoding

Extensible Markup Language (XML) interchange is supported by MSMDCXML.dll, which is installed with Microsoft® SQL Server™ 2000 Meta Data Services.

Meta Data Services supports exporting and importing of meta data through the Meta Data Coalition (MDC) Open Information Model (OIM) XML Encoding format. XML Encoding defines rules for generating XML elements that map to information model elements. The XML Encoding format is published by the MDC and implemented in Meta Data Services to support the exchange of meta data. In Meta Data Services, exporting and importing using XML enables you to exchange meta data with other repositories or tools.

Export and import using XML is supported through a dual interface. You can use the objects separately or together to perform a seamless exchange:

- The export process generates an XML document that contains XML-tagged meta data. The XML document can be expressed in memory or stored in a file.

- The import process converts XML documents to object instance data in a repository database. You can import an XML document from memory or from a file.

The exchange of meta data is directed by your application code. By instantiating export and import objects and calling the methods supported by the objects, you can complete the entire exchange programmatically.

Backward Compatibility

Meta Data Services still recognizes models that are based on earlier versions of the OIM. If you have been using XML Interchange Format (XIF) to import and export repository data, you can still do so. You can also use MDC XML Encoding to achieve the same objective. However, you cannot combine XIF and MDC XML Encoded formats. You must use either XIF or MDC XML Encoding to perform an import and export. You cannot use both in the same exchange.

Although MDC XML Encoding supports an XML format that most closely corresponds to the most recent version of the OIM, the import and export features of MDC XML Encoding can map previous versions of OIM-based models to the newest version of OIM. This mapping occurs automatically during an import or export operation, and it does this without modifying your information model. The advantage of this mapping is that you can exchange data between information models that are based on different versions of the OIM.

The only exception to this mapping correspondence occurs for new Unified Modeling Language (UML) and OIM elements that are not defined by older versions of the OIM. If your objective is to transfer repository data from information models that use new definitions to information models that use older definitions, you will experience some data loss in the conversion. Specifically, the portion of data from the new model that cannot be accommodated by the older model is logged to an error file.

For more information about XML interchange formats and information models, see Upgrading an Information Model.

Additional Topics

The following topics provide more detailed information about XML encoding.

Topic	Description
XML in Meta Data Services	Describes XML Encoding as an extension of OIM and how it is supported in Meta Data Services.
Exporting XML	Explains how to perform an export.
Export Automation Object Example	Shows examples of Microsoft Visual Basic® code that instantiate an **Export** object.
Importing XML	Explains how to perform an import.
Import Automation Object Example	Shows examples of Visual Basic code that instantiate an **Import** object.
XML Encoding Reference	Contains reference topics for the XML Encoding API.

See Also

Meta Data Coalition

OIM in Meta Data Services

Meta Data Services Programming

Exporting XML

In Microsoft® SQL Server™ 2000 Meta Data Services, exporting Extensible Markup Language (XML) is the process of creating an XML document and populating it with meta data tagged as XML elements. You can export data as an XML file or as a string that is stored in memory. After you export the data, you must import it into a repository database or make it available to another tool.

Exporting is implemented by a dual interface. Automation and COM programmers can use the **Export** object and **IExport** interface provided by Meta Data Services.

You can export instance data for specific objects, or recursively through a set of related objects. The XML document is created automatically by the export process. At minimum, you must instantiate the **Export** object and define the repository objects for which you want to export instance data. The methods you invoke determine whether the XML document is stored in memory or as an XML file that you specify.

You can generate an XML Document Type Definition (DTD) that precisely describes the XML that is produced. You can refer to the XML DTD to find out which XML structures you must support. The XML DTD that you generate is based on an information model. The information model can be a version of the OIM or some other information model that you create. XML DTD generation is provided through the Meta Data Services Software Development Kit (SDK).

Exporting Objects

To export object data, use the **Export** object and the **IExport** interface to specify the objects you want. The object can be any repository object. You can specify multiple objects in your application code.

Exporting a Set of Related Objects

You can also export data for related objects within an information model object. The export process will automatically add all target objects recursively as long as you set the COLLECTION_CONTAINING flag for the collection. You must set this flag on the relationship collection in the information model.

To avoid having to manually relate your exported data with the data in the target database, you may need to include the root object as part of your export definition.

See Also

Export Automation Object Example

Importing XML

XML Encoding Errors

XML Encoding Reference

XML IExport Interface Overview

XML IImport Interface Overview

Meta Data Services Programming

Export Automation Object Example

The following examples show how to use the **Export** object in Microsoft® Visual Basic®.

Exporting to a File

The following example shows how to export object instance data for two repository objects. You do not need to bracket an export within a transaction. To release the objects after the export concludes, set the objects to nothing.

```
dim oExp as new Export
dim oMyObj1 as RepositoryObject
dim oMyObj2 as RepositoryObject
dim oRep as new Repository
dim oRoot as RepositoryObject

set oRoot=oRep.Open "SERVER=MyServer;DATABASE=MyDB;UII
set oMyObj1=oRep.Object(objid1)
set oMyObj2=oRep.Object(objid2)
oExp.add oMyObj1
oExp.add oMyObj2
oExp.Export "c:\temp\myXmlFile.xml", INDENTATION
Set oMyObj1=Nothing
Set oMyObj2=Nothing
Set oRoot=Nothing
Set oRep=Nothing
Set oExp=Nothing
```

Exporting Multiple Objects in a Relationship

In the following example, **oMyObj1** is a collection object that relates multiple objects. The COLLECTION_CONTAINING flag, which is set on the collection object, makes exporting a relationship possible. This flag is set in the

information model and does not appear in your export code. Another flag, ADDCONTAINING_BASE (you can also use ADDCONTAINING_MOSTDERIVED) does appear in your export code. This flag supports the selection of objects in a relationship for the export process. This flag depends on the COLLECTION_CONTAINING flag to enable the selection.

```
dim oExp as new Export
dim oMyObj1 as RepositoryObject
dim oRep as new Repository
dim oRoot as RepositoryObject

set oRoot=oRep.Open "SERVER=MyServer;DATABASE=MyDB;UII
set oMyObj1=oRep.Object(objid1)
oExp.add oMyObj1, ADDCONTAINING_BASE
oExp.Export "c:\temp\myXmlFile.xml", INDENTATION
Set oMyObj1=Nothing
Set oRoot=Nothing
Set oRep=Nothing
Set oExp=Nothing
```

Exporting to a String

The following example shows how to export the same object instance data to a string stored in memory:

```
dim oExp as new Export
dim oMyObj1 as RepositoryObject
dim oMyObj2 as RepositoryObject
dim oRep as new Repository
dim oRoot as RepositoryObject

set oRoot=oRep.Open "SERVER=MyServer;DATABASE=MyDB;UII
set oMyObj1=oRep.Object(objid1)
set oMyObj2=oRep.Object(objid2)
```

```
oExp.add oMyObj1
oExp.add oMyObj2
set sXMLStr=Export.GetXML, INDENTATION
Set oMyObj1=Nothing
Set oMyObj2=Nothing
Set oRoot=Nothing
Set oRep=Nothing
Set oExp=Nothing
```

See Also

Exporting XML

Import Automation Object Example

XML IExport Interface Overview

Meta Data Services Programming

Importing XML

Importing an XML document is the process of adding meta data to a target repository. You can import meta data that was previously exported through the **IExport** interface, the **Export** object, or some other mechanism that you define.

To import object data, you can use the **IImport** interface and the **Import** object. You can handle the XML document as a string to import it from memory. More likely, however, you will want to import an XML document from a file.

Import requires the following conditions:

- The XML documents you import must be structured in the format described by Meta Data Coalition (MDC) Open Information Model (OIM) XML Encoding.

- The type information of the target repository or tool must be identical to the type information of the source objects or model. For example, if you export objects from a Unified Modeling Language (UML) information model, you must install an identical UML information model in the target repository database prior to importing. If the information models do not correspond exactly, you will lose data during the import. Rows that fail to import are logged to an error file. This file is named MSMDCXML.log and it is created in your Temp directory.

You can set flags on an import object to determine import behavior. For more information, see IImport::ImportXMLString Method and IImport::ImportXML Method.

Import returns a collection of top-level objects that your application can manipulate. To view and manipulate imported data in a repository database, the imported data must be related to the repository root object. When the information model in the target database corresponds to the information model in the source database, a relationship to the root object may be established automatically. However, whether this linkage occurs depends on the structure and content of the imported data. If your imported data is not related to the root object, you must programmatically add an object from the imported data to a

collection of the root object. This step is necessary to support navigation and to define relationships with objects in other information models.

See Also

Exporting XML

Import Automation Object Example

Installing Information Models

Using the Model Installer ActiveX Component

XML Encoding Errors

Using XML Encoding

XML Encoding Reference

XML IImport Interface Overview

Meta Data Services Programming

Import Automation Object Example

The following examples show how to use the **Import** object in Microsoft® Visual Basic®.

Importing from a File

The following example shows how to import object instance data from a file that contains exported data. The **ImportXML** method returns a collection. After you get the collection, you can enumerate the objects. To release the objects after the import concludes, set the objects to nothing.

```
dim oImp as new Import
dim oRep as new Repository
dim oRoot as RepositoryObject
dim ObjCol as TransientObjCol
set oRoot=oRep.Open "SERVER=MyServer;DATABASE=MyDB;UII
set ObjCol = oImp.ImportXML(oRep, "c:\temp\myXmlFile.xml",NEW
for each obj in ObjCol
. . .
next
Set oRoot=Nothing
Set oRep=Nothing
Set oImp=Nothing
```

Importing from a String

The following example shows how to import object instance data from a string stored in memory:

```
dim oImp as new Import
dim oRep as new Repository
dim oRoot as RepositoryObject

set oRoot=oRep.Open "SERVER=MyServer;DATABASE=MyDB;UII
```

```
oImp.ImportXMLString oRep, sXMLStr, NEWVERSION
Set oRoot=Nothing
Set oRep=Nothing
Set oImp=Nothing
```

See Also

Export Automation Object Example

Importing XML

XML IImport Interface Overview

Meta Data Services Programming

Repository API Reference

The application programming interface (API) for information models and the repository engine is the Repository API. The Repository API is composed of interfaces that define information models, and interfaces that expose the functionality of the repository engine. The interfaces that define information models are collectively known as the Repository Type Information Model (RTIM).

The Repository API Reference contains the definitions for all the core engine APIs for Microsoft® SQL Server™ 2000 Meta Data Services. These interfaces are documented at the Automation level for the Microsoft Visual Basic® programmer and at the Component Object Model (COM) level for the Microsoft Visual C++® programmer.

This table describes the sections of the Repository API reference documentation.

Section	Description
Automation Reference	Introduces the reference documentation for COM Automation objects and members.
COM Reference	Introduces the reference documentation for COM classes, interfaces, and members.
Constants and Data Types	Documents the constant and data types that you can use when programming with the repository API.
Enumerations	Documents the enumerated values for a variety of flags.
Repository Errors	Documents the errors generated by the repository engine.
Repository SQL Schema	Documents the schema of the underlying SQL tables. The schema of the underlying SQL tables is documented to facilitate querying repository data directly through SQL.

For more information about programming against information models and the repository engine, see Programming Meta Data Services Applications.

For more information about other programming interfaces that you can use in Meta Data Services, see XML Encoding Reference and OLE DB Scanner Reference.

See Also

Getting Started with Meta Data Services

Repository Object Architecture

Meta Data Services Programming

Automation Reference

The Automation Reference documents the Automation objects of the repository API. An equivalent reference is available for COM classes and interfaces.

In this documentation, Automation objects are organized into two sections.

Section	Description
Repository Engine Automation Objects	Describes the Automation objects that expose the functionality of the repository engine.
RTIM Automation Objects	Describes the Repository Type Information Model (RTIM) Automation objects. These objects define the abstract classes to which an information model must conform.

See Also

Accessing Automation Object Members

COM Reference

Information Models

Repository API Reference

Repository Engine

Repository Object Architecture

Meta Data Services Programming

Repository Engine Automation Objects

This topic introduces the repository engine objects, which are used to add, retrieve, and change information model data in a repository database.

These objects complement the Repository Type Information Model (RTIM) automation objects that define an information model. The RTIM objects are listed separately. For more information, see RTIM Automation Objects.

The following table lists the repository engine Automation objects in alphabetical order.

Object	Description
ObjectCol Object	Defines a set of repository objects that can be enumerated
Relationship Object	Connects two objects in a repository database
RelationshipCol Object	Defines a set of relationships that are attached to a particular source object
Repository Object	Defines an instance of a single repository session
RepositoryObject Object	Defines an object that is stored in a repository database and managed by the repository engine
RepositoryObjectVersion Object	Defines a versioned object that is stored in the repository database and managed by the repository engine
RepositoryTransaction Object	Defines a transaction
ReposProperties Object	Defines a set of persistent properties and collections that are attached to a repository object or relationship
ReposProperty Object	Defines a persistent property or collection that is attached to an object instance
TransientObjectCol Object	Defines an object collection that you

	can create and dynamically populate at run time using script and object methods rather than persisted data in a repository database
VersionCol Object	Defines a versioned collection of object versions
VersionedRelationship Object	Defines a connection between two repository objects in a repository database
Workspace Object	Defines a subset of a larger, shared repository

See Also

Automation Reference

Information Models

Repository API Reference

Repository Engine

Repository Object Architecture

ObjectCol Object

An object collection is a set of repository objects that can be enumerated. Two kinds of object collections are supported by the repository engine:

- The collection of destination objects that correspond to the relationships in a relationship collection. Use the **RelationshipCol** object to manage this kind of collection.

- The collection of all objects in a repository that conform to a particular class or expose a particular interface. You can instantiate a collection by using the **ObjectInstances** method.

When to Use

Use the **ObjectCol** object to enumerate the collection of repository objects that conform to a particular class or expose a particular interface. With this object, you can:

- Get a count of the number of objects in the collection.

- Retrieve one of the objects in the collection.

- Refresh the cached image of the object collection.

Properties

Property	Description
Count	The count of the number of items in the collection
Item	Retrieves the specified object from the collection

Methods

Method	Description
Cancel	Cancels an in-progress load operation
LoadStatus	Obtains the load status of the collection
Refresh	Refreshes the cached image of the collection

See Also

ClassDef ObjectInstances Method

InterfaceDef ObjectInstances Method

RelationshipCol Object

ObjectCol Count Property

A long integer that contains the count of the number of items in the collection. This is a read-only property.

Syntax

object**.Count**

The **Count** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to an **ObjectCol** object

See Also

ObjectCol Object

ObjectCol Item Property

This property retrieves an object from the collection. This is a read-only property. There are two variations of this property.

Syntax

Set variable = object.**Item**(*index*)
Set variable = object.**Item**(*objId*)

The **Item** property syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObjectVersion**. It receives the specified repository object.
object	An object expression that evaluates to an **ObjectCol** object.
index	The index of the repository object to be retrieved from the collection.
objId	The object identifier of the repository object to be retrieved from the collection.

Remarks

This property yields the latest version of a repository object. The repository engine uses a version resolution strategy to select a specific version to include in the collection. For more information, see Resolution Strategy for Objects and Object Versions.

See Also

ObjectCol Object

RepositoryObjectVersion Object

ObjectCol Cancel Method

The **Cancel** method requests the cancellation of the ongoing load operation. This method only works when the **ExecuteQuery** method is used and you specify whether you want the resulting object collection to be loaded asynchronously. For other object collections, this method has no effect.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IObjectCol2** interface, which inherits from **IObjectCol**. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**Cancel**

The **Cancel** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to an **ObjectCol** object

See Also

IObjectCol2 Interface

ObjectCol Object

ObjectCol LoadStatus Method

Repository ExecuteQuery Method

ObjectCol LoadStatus Method

The **LoadStatus** method is used to obtain the load status of the collection. This method only works when the **ExecuteQuery** method is used and you specify whether you want the resulting object collection to be loaded asynchronously. For other object collections, this method has no effect.

This method is not attached to the default interface for an **ObjectCol**; it is attached to the **IObjectCol2** interface, which inherits from **IObjectCol**. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object**.LoadStatus**

The **LoadStatus** method syntax has the following parts.

Part	Description
variable	A variable declared as **long**. It receives the load status value.
object	An object expression that evaluates to an **ObjectCol** object.

See Also

ObjectCol Object

ObjectCol Cancel Method

Repository ExecuteQuery Method

ObjectCol Refresh Method

This method refreshes the cached image of the object collection. Only cached data that has not been changed by the current process is refreshed.

Syntax

Call object.**Refresh**(*milliSecs*)

The **Refresh** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to an **ObjectCol** object.
milliSecs	This value is ignored. It is kept for backward compatibility.

See Also

ObjectCol Object

Relationship Object

A relationship connects two objects in a repository database.

All repository relationships are versioned. You can version a relationship explicitly by using **VersionedRelationship**, or you can allow the repository engine to version a relationship automatically. The repository engine automatically versions a relationship in cases where version information is unspecified or where legacy relationship objects that were created prior to version support exist.

A versioned relationship can connect a particular version of a repository object to one or more specific versions of the target object. Because every relationship is a **VersionedRelationship** object, you can declare any relationship with the following line, where **myVersionedRship** is the object you are defining:

```
Dim myVersionedRship As VersionedRelationship
```

In earlier releases of the repository engine, the object model included the **Relationship** object, but not the **VersionedRelationship** object. If you have Microsoft® Visual Basic® programs written against earlier releases of the repository engine, those programs might include declarations like the following, where *oldRship* is the object you are defining:

```
Dim oldRship As Relationship
```

These programs will continue to work with Microsoft SQL Server™ 2000 Meta Data Services because the repository API still includes the **Relationship** object. For this reason, the preceding declaration remains valid in Visual Basic. However, because every relationship is a versioned relationship, the object **oldRship** has the same members as any versioned relationship. In effect, the following two lines of code are equivalent:

```
Dim oldRship As Relationship
Dim myVersionedRship As VersionedRelationship
```

Even though all relationships are now versioned relationships, the repository

API includes the **Relationship** object so that you do not need to rewrite your Visual Basic programs that declare objects as **Relationship** objects.

See Also

Repository API

RepositoryObjectVersion Object

VersionedRelationship Object

RelationshipCol Object

A relationship collection is the set of relationships that are attached to a particular source repository object. All of the relationships in the collection must conform to the same relationship type.

When to Use

Use this object to manage the relationships that belong to a particular relationship collection. With this object, you can:

- Get a count of the number of relationships in the collection.
- Add and remove relationships to and from the collection.
- If the collection is sequenced, place a relationship in a specific place in the collection sequence.
- Retrieve a specific relationship or target object from the collection.
- Refresh the cached image of the collection.
- Obtain the type of the collection.

Properties

Property	Description
Count	The count of the number of items in the collection
Item	Retrieves the specified relationship or target object from the collection
Source	The source object for the relationship collection
Type	The object identifier for the definition object of the

	collection

Methods

Method	Description
Add	Adds a relationship to the collection
Insert	Inserts a relationship into a specific place in a sequenced collection
Move	Moves a relationship from one place to another in a sequenced collection
Refresh	Refreshes the cached image of the collection
Remove	Removes a relationship from the collection

See Also

RepositoryObject Object

RelationshipCol Count Property

This property is a long integer that contains the count of the number of items in the collection. This is a read-only property.

Syntax

object**.Count**

The **Count** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **RelationshipCol** object

See Also

RelationshipCol Object

RelationshipCol Item Property

This property retrieves a target object or relationship from the collection. This is a read-only property. There are three variations of this property.

Syntax

Set variable = object.**Item**(*index*) **Set** variable = object.**Item**(*objName*)
Set variable = object.**Item**(*objId*)

The **Item** property syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObject** class. It receives the target object of the specified relationship or the **VersionedRelationship** object.
object	An object expression that evaluates to a **RelationshipCol** object.
index	The index of the relationship to be retrieved from the collection.
objName	The name that the relationship uses to refer to its destination object. This variation can be used only when the target object is also the destination object, and when the collection requires names for destination objects.
objId	The object identifier for the target object to be retrieved from the collection.

Remarks

This property is available on two interfaces: the default interface, **ITargetObjectCol**, and a second interface, **IRelationshipCol**. If you choose to access the property that is exposed by the **IRelationshipCol** interface, your variable receives the specified **VersionedRelationship** object instead of the relationship's target object. In this case, you should declare your variable as a

VersionedRelationship, instead of as a **RepositoryObject**. Each item in the collection is a versioned relationship; each item has a **TargetVersions** collection. When you obtain a reference to the target object of a particular item (with the **get_Target** method of the **IRelationship** interface), the repository engine chooses a particular version of the target object from the items in the versioned relationship's **TargetVersions** collection.

For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

IRelationshipCol Interface

ITargetObjectCol Interface

RelationshipCol Object

VersionedRelationship Object

RelationshipCol Source Property

This property retrieves the source object for the relationship collection. This is a read-only property.

Syntax

Set variable = object.**Source**

The **Source** property syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObjectVersion** object. It receives the source object version of the relationship collection.
object	An object expression that evaluates to a **RelationshipCol** object.

See Also

RelationshipCol Object

RelationshipCol Type Property

This property specifies the type of the collection. More specifically, it is the object identifier of the **CollectionDef** object for the collection. The **Type** property is a read-only property. To copy this property to another variable, use a variable that is declared as a Variant.

Syntax

object.**Type**

The **Type** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RelationshipCol** object

See Also

CollectionDef Object

Object Identifiers and Internal Identifiers

RelationshipCol Object

RelationshipCol Add Method

This method adds a new item to a relationship collection, when the sequencing of relationships in the collection is not important. The new relationship connects the **RepositoryObjectVersion** to the source object version of the collection. The new relationship is passed back to the caller.

Syntax

Set variable = object**.Add**(*reposObj*, *objName*)

The **Add** method syntax has the following parts.

Part	Description
variable	A variable declared as a **VersionedRelationship** object. It receives the new relationship that is created for the *reposObj* **RepositoryObjectVersion**.
object	An object expression that evaluates to a **RelationshipCol** object.
reposObj	The **RepositoryObjectVersion** whose relationship is to be added to the collection.
objName	The name that the new relationship is to use for *reposObj*. This parameter is optional.

Remarks

You can add a relationship to a collection only when the collection's source object is also the collection's origin object.

When you call this method, the origin version must be unfrozen.

You can use this method to create a new versioned relationship between the source object version and a version of the target object. You cannot use it to add to a versioned relationship. If the source object version is already related to any version of the target object, this method fails. You can include another version of the target object in the versioned relationship by adding an item to the versioned

relationship's **TargetVersions** collection.

The value of *plReposObj* is the specific version of the target object.

If you are operating within the context of a workspace, the target object version you specify with *plReposObj* must be present in the workspace.

See Also

RelationshipCol Object

RelationshipCol Insert Method

RepositoryObjectVersion Object

VersionedRelationship Object

RelationshipCol Insert Method

This method adds a relationship to the collection at a specified point in the collection sequence. The new relationship connects the repository object to the source object of the collection. The new relationship is passed back to the caller.

Syntax

Set variable = object.**Insert**(*reposObj*, *index*, *objName*)

The **Insert** method syntax has the following parts.

Part	Description
variable	A variable declared as a relationship. It receives the new relationship that is created for the *reposObj* repository object.
object	An object expression that evaluates to a **RelationshipCol** object.
reposObj	The repository object whose relationship is to be added to the collection.
index	The index of the sequence location where the relationship is to be inserted. If another relationship is already present at this sequence location, the new relationship is inserted before the existing relationship.
objName	The name that the new relationship is to use for the *reposObj* object. This parameter is optional.

Remarks

Relationships may be inserted into a collection only if the collection's source object is also the collection's origin object.

This method can be used only for collections that are sequenced.

When you call this method, the origin version must be unfrozen.

You can use this method to insert a new versioned relationship between the source object version and a version of the target object. You cannot use it to enlarge a versioned relationship. If the source object version already has a relationship to any version of the target object, this method fails. You can include another version of the target object in the versioned relationship by adding an item to the versioned relationship's **TargetVersions** collection.

The value of *plReposObj* is the specific version of the target object.

If you are operating within the context of a workspace, the target object version you specify with *plReposObj* must be present in the workspace.

See Also

RelationshipCol Object

RelationshipCol Move Method

This method moves a **VersionedRelationship** object from one point in the collection sequence to another point.

Syntax

Call object.**Move**(*indexFrom*, *indexTo*)

The **Move** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **RelationshipCol** object
indexFrom	The index of the **VersionedRelationship** object to be moved in the collection sequence
indexTo	The index of the sequence location to which the **VersionedRelationship** object is to be moved

Remarks

This method can be used only with sequenced collections. When you call this method, the origin object version must be unfrozen.

See Also

RelationshipCol Object

Selecting Items in a Collection

RelationshipCol Refresh Method

This method refreshes the cached image of the object collection. Only cached data that has not been changed by the current process is refreshed.

Syntax

Call object.**Refresh**(*milliSecs*)

The **Refresh** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **RelationshipCol** object.
milliSecs	This value is ignored.

See Also

RelationshipCol Object

RelationshipCol Remove Method

This method deletes a relationship from its relationship collection. The exact behavior of this method depends on whether the relationship collection is an origin collection or a destination collection.

If the relationship collection is an origin collection, this method deletes the versioned relationship.

If the relationship collection is a destination collection, this method first performs object-version resolution to yield a single target-object version, and then it removes that target-object version from the relationship's **TargetVersions** collection.

Syntax

Call object.**Remove**(*index*) **Call** object.**Remove**(*objID*)
Call object.**Remove**(*objName*)

The **Remove** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **RelationshipCol** object
index	The index of the relationship to be removed from the collection
objID	The object identifier for the relationship object to be removed from the collection
objName	The relationship that uses this name for its destination object is to be removed from the collection

Remarks

A relationship can be removed by name only if it is a unique-naming relationship.

If the source is the origin, the origin version must be unfrozen.

If the relationship is a destination relationship and the resolution strategy yields a target object version that is frozen, this method fails.

Removal from a sequenced collection does not update the collection sequence order.

See Also

Naming and Unique-Naming Collections

RelationshipCol Object

Resolution Strategy for Objects and Object Versions

Repository Object

A **Repository** object is an instance of a single repository session. The scope of a repository object is a repository database. Because you can have multiple repository databases, you use the **Repository** object to connect and interact with a specific database.

When to Use

You can use a repository instance to:

- Create a new repository database or connect to an existing repository database.
- Access the root repository object, **ReposRoot**.
- Retrieve a **RepositoryObject** or **RepositoryObjectVersion**.
- Create the initial version of a **RepositoryObject** or **RepositoryObjectVersion**.
- Refresh cached repository data.
- Manage repository transactions.

Properties

Property	Description
ConnectionString	The ODBC connection string that the repository engine uses to obtain an ODBC connection. This property is not a default interface member.
MajorDBVersion	The major version number of the first repository engine

	version that introduced this database format. This property is not a default interface member.
MinorDBVersion	The minor version number of the first repository engine version that introduced this database format. This property is not a default interface member.
Object	Retrieves the specified **RepositoryObject**.
ReposConnection	The ODBC connection handle that the repository engine uses to access the repository database. This property is not a default interface member.
RootObject	The **ReposRoot** object of the open repository database.
Transaction	The transaction processing interface.
Version	Retrieves the specified **RepositoryObjectVersion**.

Methods

Method	Description
Create	Creates a new repository database.
CreateObject	Creates a new instance of a **RepositoryObject** or **RepositoryObjectVersion** in the open repository database.
CreateObjectEx	Creates the first version of a new repository object instance of the specified type and explicitly assigns the object-version identifier that is passed in as an argument. This is unlike **CreateObject** method, in which the repository engine assigns the version ID.
ExecuteQuery	Executes an SQL query against the repository database. This method is not a default interface member.
FreeConnection	Releases an ODBC connection handle. This method is not a default interface member.
GetCollection	Returns a result set of objects in a collection based on selection criteria.
GetNewConnection	Obtains a new ODBC connection handle using the same connection settings that the repository engine

	is using to access the repository database. This method is not a default interface member.
GetOption	Gets an option that supports performance optimization at run time.
InternalIDToObjectID	Converts an internal identifier into an object identifier.
InternalIDToVersionID	Converts an internal object-version identifier into an object-version identifier.
ObjectIDToInternalID	Converts an object identifier into an internal identifier.
Open	Opens the specified repository database.
Refresh	Refreshes the cached image of all data for the open repository database.
ResetOption	Resets a run-time performance option to its default value.
SetOption	Sets an option that supports performance optimization at run time.
VersionIDToInternalID	Converts an object-version identifier into an internal object-version identifier.

See Also

Connecting to and Configuring a Repository

Repository ConnectionString Property

This property contains the ODBC connection string that the repository engine uses to connect to a repository database. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryODBC** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**ConnectionString**

The **ConnectionString** property syntax has the following part.

Part	Description
object	The object that represents the open repository instance through which application code or a tool interacts with a repository

Remarks

The ODBC connection string can contain user identification and password information. Be sure to protect this information to prevent unauthorized access.

See Also

Connecting to and Configuring a Repository

IRepositoryODBC Interface

Repository Object

Repository MajorDBVersion Property

This property returns the database version. The database version is created from the version number of the repository engine that created the database. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepository2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**MajorDBVersion**

The **MajorDBVersion** property syntax has the following parts.

Part	Description
variable	Declared as **long**. It receives the database major version.
object	The object that represents the open repository instance.

Remarks

Database version information is stored in the **RTblDatabaseVersion** SQL Table. The value will be 2.0 if the database was created or upgraded by repository engine 2.0, or 3.0 if the database was created or upgraded by repository engine 3.0.

Additional version information is available through the **MinorDBVersion** property.

See Also

IRepository2 Interface

Repository MinorDBVersion Property

Repository Object

RTblDatabaseVersion SQL Table

Upgrading and Migrating a Repository Database

Repository MinorDBVersion Property

This property returns the minor version number of the first repository engine version that introduced this database format. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepository2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**MinorDBVersion**

The **MinorDBVersion** property syntax has the following parts.

Part	Description
variable	Declared as **long**. It receives the database minor version.
object	The object that represents the open repository instance.

Remarks

Database version information is stored in the **RTblDatabaseVersion** SQL Table. Major version information can be retrieved using the **MajorDBVersion** property.

See Also

IRepository2 Interface

Repository MajorDBVersion Property

Repository Object

RTblDatabaseVersion SQL Table

Upgrading and Migrating a Repository Database

Repository Object Property

Use this property to retrieve a particular instance of a **RepositoryObject**. This property is read-only.

Syntax

Set variable = object.**Object(***objectId***)**

The **Object** property syntax has the following parts.

Part	Description
variable	Declared as a **RepositoryObject**. It receives the repository object.
object	The object that represents the open repository instance through which application code or a tool interacts with a repository.
objectId	The object identifier for the repository object to be retrieved.

Remarks

The repository returns the latest version of a repository object. For more information about how the repository engine selects a specific version, see Resolution Strategy for Objects and Object Versions.

See Also

Object Identifiers and Internal Identifiers

Repository Object

Repository ReposConnection Property

This property contains the ODBC connection handle that the repository engine is using to access the repository database. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryODBC** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object**.ReposConnection**

The **ReposConnection** property syntax has the following part.

Part	Description
object	The object that represents the open repository instance through which application code or a tool interacts with a repository

Remarks

Using the repository engine ODBC connection handle does not isolate you from changes made by the repository engine. For example, uncommitted changes made by the repository engine will be visible to your application.

When using the repository engine's ODBC connection handle, you must not change the state of the handle in a way that is incompatible with the repository engine. Specifically, do not:

- Change any ODBC connection options.
- Perform any access operations that are concurrent with repository method invocations.

- Directly commit or rollback a database transaction. The **IRepositoryTransaction** interface must always be used to manage transactions.

Be sure to free the handle obtained through this method before releasing your open repository instance. To free the connection handle, use the **FreeConnection** method.

See Also

Connecting to and Configuring a Repository

IRepositoryODBC Interface

IRepositoryTransaction Interface

Repository FreeConnection Method

Repository Object

Repository RootObject Property

This property is the repository root object for the open repository. The root object provides a starting location for all subsequent navigation through the information models you have installed. This is a read-only property.

Syntax

Set variable = object.**RootObject**

The **RootObject** property syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObject**. It receives the root repository object (**ReposRoot**).
object	The object that represents the open repository instance through which application code or a tool interacts with a repository.

See Also

Repository Object

ReposRoot Object

Repository Transaction Property

This property is the **RepositoryTransaction** object for the open repository instance. This is a read-only property.

Syntax

Set variable = object.**Transaction**

The **Transaction** property syntax has the following parts.

Part	Description
variable	A variable declared as an object. It receives the **RepositoryTransaction** object for this repository instance.
object	The object that represents the open repository instance through which application code or a tool interacts with a repository.

Remarks

You can gain access to the **RepositoryTransaction** object by using this syntax. After you access the **RepositoryTransaction** object, you can access the properties and methods of the **RepositoryTransaction** object through standard variable.method and variable.property syntax. You can also access the properties and methods of the **RepositoryTransaction** object directly by using syntax like the following:

Call object.**Transaction**.*method*

-or-

variable = object.**Transaction.***property*

See the **RepositoryTransaction** object for details on the methods and properties that it provides.

See Also

Repository Object

RepositoryTransaction Object

Repository Version Property

This property retrieves a particular instance of a **RepositoryObjectVersion** from the repository. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepository2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Version(***versionId*, *integer***)**

The **Version** property syntax has the following parts.

Part	Description
variable	Declared as a **RepositoryObjectVersion**. It receives the repository object version.
object	The object that represents the open repository instance through which application code or a tool interacts with a repository.
versionId	The object-version identifier for the repository object to be retrieved.
integer	An integer indicates the strategy used by the repository engine to select a specific object. 1=SPECIFIEDVERSION. This value appears when you explicitly select a specific object version. 2=LATESTVERSION. This value appears when the most recently created version is selected. 3=VERSIONINWORKSPACE. This value appears when the object version in the workspace is selected. 4=PINNEDVERSION. This value appears when the pinned target object version of the relationship that you are currently navigating is selected.

See Also

Repository Object

RepositoryObjectVersion Object

Repository Create Method

Use this method to create a new repository database or to populate an empty database with repository SQL schema. Standard repository SQL tables are automatically created. The root repository object of the new repository is passed back to the calling program.

Syntax

Set variable = object.**Create(***connect*, *user*, *password*, *flags***)**

The **Create** method syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObject** that evaluates to a **ReposRoot** object.
object	The instance of the **Repository** class that you are using to create the new repository database.
connect	The ODBC connection string to be used for accessing the database server that will host your new repository.
user	The user name to use for identification to the database server.
password	The password that matches the user input parameter.
flags	Flags that determine database access and caching behavior for the open repository. For more information, see the ConnectionFlags Enumeration.

Remarks

If the connection string indicates a Microsoft® JET database, the repository engine creates the database and populates it with the repository SQL schema. If the connection string indicates a Microsoft SQL Server™ 6.5, SQL Server 7.0, or SQL Server 2000, or the SQL Server Runtime Engine, the repository engine populates an empty database with the standard SQL schema. In this case, you must create an empty database before invoking this method.

See Also

Connecting to and Configuring a Repository

Repository Object

RepositoryObject Object

Repository SQL Schema

ReposRoot Object

Repository CreateObject Method

This method creates a new repository object of a certain type. You can specify this method in application code to create an instance of a class defined in an information model.

Syntax

Set variable = object.**CreateObject(***typeId*, *objectId***)**

The **CreateObject** method syntax has the following parts.

Part	Description
variable	Declared as a **RepositoryObject** or **RepositoryObjectVersion**. It receives the new **RepositoryObject** or **RepositoryObjectVersion**.
object	The object that represents the open repository instance through which application code or a tool interacts with a repository.
typeId	The object identifier of the class to which the new object conforms. For example, to create an instance of a **Storage** class, specify the object identifier of the **Storage** class.
objectId	The object identifier to be assigned to the new object. Either pass in ObjID_NULL, or leave it unspecified to have an object identifier assigned automatically.

Remarks

Use this method to create the first version of a new **RepositoryObject**. To create subsequent versions, use the **CreateVersion** method of the **RepositoryObjectVersion** object.

This method can be called from a shared repository but not from a workspace. The workaround is to create the object through the central repository and include it in the workspace.

See Also

Choosing an Automation Server for a Class

Object Identifiers and Internal Identifiers

Repository Object

RepositoryObject Object

RepositoryObjectVersion CreateVersion Method

Repository CreateObjectEx Method

This method creates the first version of a new repository object instance of the specified type and explicitly assigns the object-version identifier that is passed in as an argument. This is unlike the **CreateObject** method, in which the repository engine assigns the version ID.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepository2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

set variable = object.**CreateObjectEx(***typeID, objID, extVersionID***)**

The **CreateObjectEx** method syntax has the following parts.

Part	Description
variable	Declared as a **RepositoryObjectVersion**. It receives the new **RepositoryObjectVersion**.
object	The object that represents the open repository instance through which application code or a tool interacts with a repository.
typeId	The object identifier of the class to which the new object conforms. For example, to create an instance of a **Storage** class, specify the object identifier of the **Storage** class.
objectId	The object identifier to be assigned to the new object. Either pass in ObjID_NULL, or leave unspecified to have an object identifier assigned automatically.
extVersionID	The external object-version identifier.

Remarks

This method provides an alternate approach for creating an object instance. To allow the repository engine to create an object identifier for you, use the **CreateObject** method.

See Also

Repository CreateObject Method

Repository Object

Assigning Object Identifiers

Repository ExecuteQuery Method

This method executes the specified SQL query against the repository database, and returns a collection of repository object instances. The columns that are returned by the query must be either just the internal identifier (**IntID**) column, or the internal identifier and the type identifier (**IntID** and **TypeID**) columns of the **RTblVersions** table.

The **ExecuteQuery** method returns all objects based on the identifier. To create a query that applies selection criteria to an object collection, use the **GetCollection** method.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryODBC** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**ExecuteQuery(***queryString***)**

The **ExecuteQuery** method syntax has the following parts.

Part	Description
variable	Declared as an **ObjectCol** object. It receives the collection of objects that meet the selection criteria of the SQL query.
object	The object that represents the open repository instance through which application code or a tool interacts with a repository.
queryString	A string that contains the SQL Query or the name of a stored procedure to be executed.

See Also

Object Identifiers and Internal Identifiers

ObjectCol Object

Repository Object

RTblVersions SQL Table

Repository FreeConnection Method

This method frees an ODBC connection handle.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryODBC** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**FreeConnection(***hdbc***)**

The **FreeConnection** method syntax has the following parts.

Part	Description
object	The object that represents the open repository instance through which application code or a tool interacts with a repository
hdbc	The ODBC connection handle to be released

Remarks

Use this method to free the handle obtained via either the **ReposConnection** property or the **GetNewConnection** method before releasing the open repository instance.

See Also

IRepositoryODBC Interface

Repository Object

Repository GetNewConnection Method

Repository ReposConnection Property

Repository GetCollection Method

This method returns a result set based on selection criteria. The result set is a collection of repository objects. When you use this method on the repository session object, the repository engine filters the collection of all objects in the repository database.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposQuery** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

set variable = object.**GetCollection(***filter***)**

The **GetCollection** method syntax has the following parts.

Part	Description
variable	Declared as an object collection. It receives the new **ObjectCol** instance.
object	The object that represents the open repository instance through which application code or a tool interacts with a repository.
filter	The query string that selects a result set. For more information about query syntax and arguments, see IReposQuery Interface.

See Also

ObjectCol Object

Repository Object

Repository GetNewConnection Method

This method obtains a new ODBC connection handle using the same ODBC connection string that the repository engine uses to access the repository database. Using a new ODBC connection handle isolates you from changes made by the repository engine.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryODBC** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**GetNewConnection**

The **GetNewConnection** method syntax has the following parts.

Part	Description
variable	Receives the new connection handle
object	The object that represents the open repository instance through which application code or a tool interacts with a repository

Remarks

Be sure to free the handle obtained via this method before releasing your open repository instance. To free the connection handle, use the **FreeConnection** method.

See Also

IRepositoryODBC Interface

Repository FreeConnection Method

Repository Object

Repository GetOption Method

This method gets an option that supports the implementation of performance optimization at run time.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposOptions** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**GetOption(***optionIdentifier***)**

The **GetOption** method syntax has the following parts.

Part	Description
object	An expression that evaluates to a **Repository** session object.
optionIdentifier	An option expressed as a **Variant**. You can specify an option name or an option value. For more information about option names and values, see IReposOptions Options Table.

See Also

IReposOptions Interface

Optimizing Repository Performance

Repository Object

Repository ResetOption Method

Repository SetOption Method

Repository InternalIDToObjectID Method

This method translates an internal identifier into an object identifier. The repository engine uses internal identifiers to identify instances of **RepositoryObject** or **RepositoryObjectVersion.**

Syntax

variable = object.**InternalIDToObjectID(***internalId***)**

The **InternalIDToObjectID** method syntax has the following parts.

Part	Description
variable	Receives the object identifier
object	The object that represents the open repository instance through which application code or a tool interacts with a repository
internalId	The internal identifier to be converted

Remarks

Object identifiers are globally unique, and are the same across repositories for the same object. Internal identifiers are unique only within the scope of a single repository.

The translation performed by this method is accomplished without loading the object in question. This enables database queries involving an object or relationship type identifier to be constructed without having to load the definition object.

See Also

Assigning Object Identifiers

Object Identifiers and Internal Identifiers

Repository Object

Repository InternalIDToVersionID Method

This method translates an internal object-version identifier into a repository object-version identifier. The repository engine uses internal object-version identifiers to identify **RepositoryObjectVersions**.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepository2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**InternalIDToVersionID(***intVersionId***)**

The **InternalIDToVersionID** method syntax has the following parts.

Part	Description
variable	Receives the object-version identifier
object	The object that represents the open repository instance through which application code or a tool interacts with a repository
intVersionId	The internal object-version identifier to be converted

Remarks

Object-version identifiers are globally unique, and are the same across repositories for the same object. Internal object-version identifiers are unique only within the scope of a single repository.

The translation performed by this method is accomplished without loading the object version in question. This enables database queries involving an object or relationship type identifier to be constructed without having to load the definition object.

See Also

Repository ObjectIDToInternalID Method

This method translates an object identifier into an internal identifier. Internal identifiers are used by the repository engine to identify repository objects.

Syntax

variable = object.**ObjectIDToInternalID(***objectId***)**

The **ObjectIDToInternalID** method syntax has the following parts.

Part	Description
variable	Receives the internal identifier
object	The object that represents the open repository instance through which application code or a tool interacts with a repository
objectId	The object identifier to be converted

Remarks

Object identifiers are globally unique, and are the same across repositories for the same object. Internal identifiers are unique only within the scope of a single repository.

The translation performed by this method is accomplished without loading the object in question. This enables database queries involving an object or relationship type identifier to be constructed without having to load the definition object.

See Also

Assigning Object Identifiers

Object Identifiers and Internal Identifiers

Repository InternalIDToObjectID Method

Repository Object

Repository Open Method

Use this method to open (connect to) a repository. The root repository object is passed back to the caller.

Syntax

Set variable = object.**Open(***connect*, *user*, *password*, *flags***)**

The **Open** method syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObject.** It receives the root repository object (**ReposRoot**) for the repository.
object	The instance of the **Repository** class that you are using to connect to the repository.
connect	The ODBC connection string to be used for accessing the database server that hosts your repository.
user	The user name to use for identification to the database server.
password	The password that matches the user input parameter.
flags	Flags that determine database access and caching behavior for the open repository. For more information, see ConnectionFlags Enumeration.

See Also

Connecting to and Configuring a Repository

Repository Object

Repository Refresh Method

This method refreshes all of the cached data for this open repository instance. Only cached data that has not been changed by the current process is refreshed.

Syntax

Call object.**Refresh(***milliSeconds***)**

The **Refresh** method syntax has the following parts.

Part	Description
object	The object that represents the open repository instance through which application code or a tool interacts with a repository.
milliSecs	This value is ignored. It is kept for backward compatibility.

See Also

Repository Object

Repository ResetOption Method

This method resets an option that supports performance optimization at run time to its default value.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposOptions** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**ResetOption(***optionIdentifier, value***)**

The **ResetOption** method syntax has the following parts.

Part	Description
object	An expression that evaluates to a **Repository** session object.
optionIdentifier	An option expressed as a **Variant**. You can specify an option name or an option value. For more information about option names and values, see IReposOptions Options Table.

See Also

IReposOptions Interface

Optimizing Repository Performance

Repository GetOption Method

Repository Object

Repository SetOption Method

Repository SetOption Method

This method sets an option that supports performance optimization at run time.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposOptions** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**SetOption(***optionIdentifier, value***)**

The **SetOption** method syntax has the following parts.

Part	Description
object	An expression that evaluates to a **Repository** session object.
optionIdentifier	An option expressed as a **Variant**. You can specify an option name or an option value.
value	The value of the option. The value must be paired with the corresponding *optionIdentifier*. For more information about option names and values, see IReposOptions Options Table.

See Also

IReposOptions Interface

Optimizing Repository Performance

Repository GetOption Method

Repository Object

Repository ResetOption Method

Repository VersionIDToInternalID Method

This method translates a repository object-version identifier into an internal object-version identifier. Internal object-version identifiers are used by the repository engine to identify specific instances of a **RepositoryObjectVersion**.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepository2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**VersionIDToInternalID(***versionId***)**

The **VersionIDToInternalID** method syntax has the following parts.

Part	Description
variable	Receives the internal object-version identifier
object	The object that represents the open repository instance through which application code or a tool interacts with a repository
objectId	The object-version identifier to be converted

Remarks

Object-version identifiers are globally unique, and are the same across repositories for the same object version. Internal object-version identifiers are unique only within the scope of a single repository.

The translation performed by this method is accomplished without loading the object version in question. This enables database queries involving an object or relationship type identifier to be constructed without having to load the definition object.

See Also

RepositoryObject Object

A **RepositoryObject** is an object that is stored in a repository database and managed by the repository engine.

All repository objects are versioned. You can create new object versions explicitly using the **RepositoryObjectVersion**. The repository engine can create version information implicitly in cases where version information is unspecified or where legacy objects that were created prior to version support exist.

A **RepositoryObjectVersion** is a particular rendition of a **RepositoryObject**. Each version of an object can differ from other versions of that object in its property values and collections. When you manipulate a repository object within a Microsoft® Visual Basic® program, for example, you are actually manipulating a particular version of that object. That is, you manipulate a **RepositoryObjectVersion.**

You can declare any repository object version with the following line:

```
Dim newVersionedReposObject As RepositoryObjectVersion
```

In earlier releases of the repository engine, the object model included the **RepositoryObject** but not the **RepositoryObjectVersion**. If you have Visual Basic programs written against earlier releases, those programs might include declarations like the following:

```
Dim oldReposObject As RepositoryObject
```

These programs will continue to work with Microsoft SQL Server™ 2000 Meta Data Services because the repository object model still includes the **RepositoryObject**. As a result, the preceding declaration remains valid in Visual Basic. Whenever you manipulate an object, you actually manipulate a specific version of that object. So the object **oldReposObject** has the same members as any repository object version has. In effect, the following two lines of code are equivalent:

```
Dim myVersionedReposObject As RepositoryObjectVersion
```

```
Dim oldReposObject As RepositoryObject
```

Even though repository objects are now versioned, the repository object model includes the **RepositoryObject** so that you do not need to rewrite your Visual Basic programs that declare an object as a **RepositoryObject**.

When to Use

The **RepositoryObjectVersion** object supersedes **RepositoryObject**. However, if you already have application code that includes **RepositoryObject**, you can maintain that code using **RepositoryObject**. You can also use **RepositoryObject** to work with meta data that is not versioned.

Use the **RepositoryObject** object to manipulate the properties of a repository object, to delete a repository object, or to refresh the cached image of a repository object.

To create a new **RepositoryObject**, use the **CreateObject** method of the **Repository** session object.

Properties

Property	Description
ClassName	The name of a class that defines a repository object, as defined in an information model. This property is not a default interface member.
ClassType	The type of a class that defines a repository object, as defined in an information model. This property is not a default interface member.
Interface	The specified object interface.
InternalID	The internal identifier that a repository instance uses to refer to a repository object.
Name	The name of the repository object.
ObjectID	The object identifier for the repository object.
Repository	The open a repository instance through which this

	repository object was instantiated.
Type	The type of the repository object.

Methods

Method	Description
Delete	Deletes a repository object
Lock	Locks the repository object
Refresh	Refreshes the cached image of a repository object

Collections

Collection	Description
Properties	The collection of all persistent properties that are attached to a **RepositoryObject**

See Also

Repository CreateObject Method

RepositoryObjectVersion Object

RepositoryObject ClassName Property

This property specifies the name of a class that defines a repository object. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObject2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

string=object.**ClassName**

The **ClassName** property syntax has the following parts.

Part	Description
string	A variable length string that can be a maximum of 255 characters
object	An object expression that evaluates to a **RepositoryObject**

Remarks

This property can be used to display the name of the class that defines the object. For example, if you create multiple repository object instances of a **ClassDef** named **StoredProcedure**, the class name associated with each of these repository object instances is **StoredProcedure**.

In Meta Data Browser, **ClassName** values are included with object property information to provide additional information about an object. For example, the **ClassName** of **Model** is **Model**.

See Also

IRepositoryObject2 Interface

RepositoryObject ClassType Property

RepositoryObject Object

RepositoryObject ClassType Property

This property specifies the type of a class as defined by its object identifier. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObject2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**ClassType**

The **ClassType** property syntax has the following parts.

Part	Description
variable	An object expression that evaluates to a **ClassDef** object
object	An object expression that evaluates to a **RepositoryObject**

Remarks

This property can be used to retrieve the abstract class in the information model that defines the object instance. For example, if you create multiple repository object instances of a **ClassDef** named **StoredProcedure** that has an object identifier of **StoredProc_objid**, the class type associated with each of these repository object instances is **StoredProc_objid**.

See Also

IRepositoryObject2 Interface

Repository Identifiers

RepositoryObject ClassName Property

RepositoryObject Object

Using Meta Data Browser

RepositoryObject Interface Property

Use this property to obtain a view of the repository object that uses an interface other than the default interface. This is a read-only property. There are three variations of this property.

Syntax

Set variable = object.**Interface(***interfaceId***)**
Set variable = object.**Interface(***objectId***)**
Set variable = object.**Interface(***interfaceName***)**

The **Interface** property syntax has the following parts.

Part	Description
variable	An object variable. It receives the repository object with the specified interface as the default interface.
object	An object expression that evaluates to a **RepositoryObject**.
interfaceId	The interface identifier for the interface to be retrieved.
objectId	The object identifier for the interface definition to which the interface to be retrieved conforms.
interfaceName	A string containing the name of the interface to be retrieved.

See Also

Assigning Object Identifiers

InterfaceDef Object

Object Identifiers and Internal Identifiers

Repository Object

RepositoryObject Object

RepositoryObject InternalID Property

This property is the internal identifier that the repository engine uses to refer to this object. The internal identifier is unique within the repository, but is not unique across repositories. This is a read-only property. To copy this property to another variable, use a variable declared as a **Variant**.

Syntax

object.**InternalID**

The **InternalID** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryObject**

See Also

Assigning Object Identifiers

Object Identifiers and Internal Identifiers

RepositoryObject Object

RepositoryObject ObjectID Property

RepositoryObject Name Property

This property is a character string that contains the name of the repository object.

The **Name** property is normally derived from the relationship for which this repository object is the destination object. When the name is retrieved, the name from the first naming relationship found is returned. If the object is not the destination of any naming relationship, a null name is returned. However, you can set a name property explicitly. When the name is set, the new name is used for all naming relationships for which the object is the destination.

Syntax

string=object.**Name**

The **Name** property syntax has the following parts.

Part	Description
string	A variable length string that can be a maximum of 255 characters
object	An object expression that evaluates to a **RepositoryObject**

Remarks

If the repository object exposes the **INamedObject** interface, the name that is retrieved is always the **Name** property of the **INamedObject** interface. Likewise, when this property is set, the **Name** property of the **INamedObject** interface and the name associated with all naming relationships are set to the new value.

See Also

INamedObject Interface

Repository Object

RepositoryObject Object

RepositoryObject ObjectID Property

This property is the object identifier for the repository object. The object identifier is unique across all repositories. This is a read-only property. To copy this property to another variable, use a variable declared as a **Variant**.

Syntax

object.**ObjectID**

The **ObjectID** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryObject**

See Also

Assigning Object Identifiers

Object Identifiers and Internal Identifiers

Repository Object

RepositoryObject Object

RepositoryObject InternalID Property

RepositoryObject Repository Property

The **Repository** property is the open repository instance through which a repository object is instantiated. This is a read-only property.

Syntax

Set variable = object.**Repository**

The **Repository** property syntax has the following parts.

Part	Description
variable	A variable declared as an instance of the **Repository** class. It receives the object that represents the open repository instance.
object	An object expression that evaluates to a **RepositoryObject** object.

See Also

Repository Object

RepositoryObject Object

RepositoryObject Type Property

This property specifies the type of the repository object. More specifically, it is the object identifier of the class definition object that defines the repository object. This property is read-only. To copy this property to another variable, use a variable declared as a **Variant**.

Syntax

object.**Type**

The **Type** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryObject**

Remarks

The **Type** property is the object identifier of the class to which the new object conforms. For example, to manipulate an object instance of a **Storage** class, specify the object identifier of the **Storage** class definition object upon which the object instance is based.

For example, if you define three object instances of the **Storage** class (**testStorage1**, **testStorage2**, and **finalStorage1**), all three object instances will have the same **Type** property value.

See Also

Object Identifiers and Internal Identifiers

RepositoryObject Object

RepositoryObject Delete Method

This method deletes a repository object from the repository. Any relationships that connect the object to other objects are deleted. If the repository object is an origin object of a relationship collection, and the relationship type indicates that deletes are to be propagated, all of the destination objects are also deleted.

Syntax

Call object.**Delete**

The **Delete** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryObject**

See Also

Propagating Deletes

Repository Object

RepositoryObject Object

RepositoryObject Lock Method

Use this method to lock the repository object. Locking the object prevents other processes from updating the object while you are working with it. The lock is released when you end the current transaction.

Syntax

Call object.**Lock**

The **Lock** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryObject**

See Also

Repository Object

RepositoryObject Object

RepositoryObject Refresh Method

This method refreshes the cached image of the repository object. Only cached data that has not been changed by the current process is refreshed.

Syntax

Call object.**Refresh(***milliSecs***)**

The **Refresh** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **RepositoryObject**.
milliSecs	This value is ignored. It is kept for backward compatibility.

See Also

Repository Object

RepositoryObject Object

RepositoryObject Properties Collection

The **Properties** collection returns a list of all properties defined on every interface that the class supports.

Property names must be unique within the collection. To distinguish between identically named properties, the repository engine first adds the interface name (for example, **myInterfaceName.myPropertyName**). If the name is still a duplicate, the prefix of the information model is assigned (for example, **myTypeLibPrefix:myInterfaceName.myPropertyName**). In this case, the prefix is provided by the **ReposTypeLib** object.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObject2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Properties**(*index*) **Set** variable = object.**Properties**(*objID*)
Set variable = object.**Properties**(*objName*)

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. It receives the specified property.
object	An object expression that evaluates to a **RepositoryObject**.
index	An integer index that identifies which property in the collection to address. For an integer index, the valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by *object*.**Properties**.**Count**. For more information, see Selecting Items in a

	Collection.
objID	An object identifier that identifies which property in the collection to address.
objName	An object name that identifies which property in the collection to address.

See Also

IRepositoryObject Interface

IRepositoryObject2 Interface

Repository Object

RepositoryObject Object

ReposProperty Object

RepositoryObjectVersion Object

A **Repository** object is an object that is stored in the repository database, and is managed by the repository engine. A **RepositoryObjectVersion** object is a specific rendition of a **RepositoryObject**. Microsoft® SQL Server™ 2000 Meta Data Services can retain multiple renditions of a **RepositoryObject** so that you can reestablish historical states of a particular instance.

When to Use

Use the **RepositoryObjectVersion** object to manipulate the properties of a repository object version, to delete a repository object version, or to refresh the cached image of a repository object version.

Properties

Property	Description
CheckOutWorkspace	The workspace to which the object version is checked out. This property is not a default interface member.
ClassName	The name of a class that defines a repository object, as defined in an information model. This property is not a default interface member.
ClassType	The type of a class that defines repository objects, as defined in an information model. This property is not a default interface member.
Interface	The specified object interface. This property is not a default interface

	member.
InternalID	The internal object identifier that the repository engine uses to refer to the repository object. This property is not a default interface member.
IsCheckedOut	A flag that indicates whether the object version is checked out to a workspace. This property is not a default interface member.
IsFrozen	Indicates whether the object version is frozen.
Name	The name of the repository object version. This property is not a default interface member.
ObjectID	The object identifier for the repository object. This property is not a default interface member.
PredecessorCreationVersion	The object version from which the current object version was originally created.
Repository	The open repository instance through which this repository object was instantiated. This property is not a default interface member.
ResolutionType	An enumerated property that identifies which criteria was used to select a repository object version. This property is not a default interface member.
Type	The type of the repository object.

	This property is not a default interface member.
VersionID	The object-version identifier for the repository object version.
VersionInternalID	The internal object-version identifier that the repository uses to refer to the repository object version.

Methods

Method	Description
CreateVersion	Creates a new version of the current object, based on the current object version.
Delete	Deletes a repository object. This method is not a default interface member.
FreezeVersion	Fixes object version property values and origin collections, permanently preventing further modification to a specific version.
Lock	Locks the repository object. This method is not a default interface member.
MergeVersion	Modifies the current object version by combining its property values and collection with those of another version of the same repository object.
Refresh	Refreshes the cached image of a repository object. This method is not a default interface member.

Collections

Collection	Description

ObjectVersions	The collection of all the versions of the current repository object.
PredecessorVersions	The collection of all immediate predecessor versions of the current repository object version.
Properties	The collection of all of the properties that are attached to the repository object. This collection is not a default interface member.
SuccessorVersions	The collection of all immediate successor versions of the current repository object version.
Workspaces	The collection of all workspaces in which the current object version is present. This collection is not a default interface member.

See Also

RepositoryObject Object

Versioning Objects

RepositoryObjectVersion ClassName Property

This property specifies the name of a class that defines a repository object. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObject2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

string=object.**ClassName**

The **ClassName** property syntax has the following parts.

Part	Description
string	A variable length string that can be a maximum of 255 characters
object	An object expression that evaluates to a **RepositoryObjectVersion**

Remarks

This property can be used to display the name of the class that defines the object. For example, if you create multiple repository object instances of a **ClassDef** named **StoredProcedure**, the class name associated with each of these repository objects instance is **StoredProcedure**.

In Meta Data Browser, **ClassName** values are included with object property information to provide additional information about an object. For example, the **ClassName** of **Model** is **Model**.

See Also

IRepositoryObject2 Interface

RepositoryObjectVersion ClassType Property

This property specifies the type of a class as defined by its object identifier. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObject2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**ClassType**

The **ClassType** property syntax has the following parts.

Part	Description
variable	An object expression that evaluates to a **ClassDef** object
object	An object expression that evaluates to a **RepositoryObjectVersion**

Remarks

This property can be used to retrieve the abstract class in the information model that defines the object instance. For example, if you create multiple repository object instances of a **ClassDef** named **StoredProcedure** that has an object identifier of **StoredProc_objid**, the class type associated with each of these repository objects instance is **StoredProc_objid**.

See Also

IRepositoryObject2 Interface

RepositoryObjectVersion ClassName Property

RepositoryObjectVersion Object

Using Meta Data Browser

RepositoryObjectVersion CheckOutWorkspace Property

This property identifies the workspace to which the repository object version is currently checked out. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IWorkspaceItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**CheckOutWorkspace**

The **CheckOutWorkspace** property syntax has the following parts.

Part	Description
variable	A variable declared as an instance of the **Workspace** class. It receives the object that represents the workspace to which the object version is checked out.
object	An object expression that evaluates to a **RepositoryObjectVersion** object.

Remarks

A workspace is a repository object. The **CheckOutWorkspace** property identifies the workspace object by object identifier or object name.

See Also

IWorkspaceItem Interface

RepositoryObjectVersion Object

Workspace Checkin Method

RepositoryObjectVersion Interface Property

Use this property to obtain a view of the repository object version that uses an alternate interface as the default interface. There are three variations of this property. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Interface(***interfaceId***)**
Set variable = object.**Interface(***objectId***)**
Set variable = object.**Interface(***interfaceName***)**

The **Interface** property syntax has the following parts.

Part	Description
variable	An object variable. It receives the repository object with the specified interface as the default interface.
object	An object expression that evaluates to a **RepositoryObjectVersion** instance.
interfaceId	The interface identifier for the interface to be retrieved.
objectId	The object identifier for the interface definition to which the interface to be retrieved conforms.
interfaceName	A string containing the name of the interface to be retrieved.

See Also

InterfaceDef Object

IRepositoryItem Interface

RepositoryObjectVersion Object

RepositoryObjectVersion InternalID Property

This property is the internal identifier that the repository engine uses to refer to this object. The internal identifier is unique within a repository instance, but not unique across all repositories. To copy this property to another variable, use a variable declared as a **Variant**. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObject** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variant=object.**InternalID**

The **InternalID** property syntax has the following parts.

Part	Description
variant	A string, Boolean, or integer that receives the value of an internal identifier
object	An object expression that evaluates to a **RepositoryObjectVersion** object

Remarks

This property yields the internal object identifier, not the internal object-version identifier.

See Also

IRepositoryObject Interface

Object Identifiers and Internal Identifiers

RepositoryObjectVersion Object

RepositoryObjectVersion ObjectID Property

RepositoryObjectVersion IsCheckedOut Property

This property indicates whether the object version is currently checked out to any workspace. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IWorkspaceItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**IsCheckedOut**

The **IsCheckedOut** property syntax has the following parts.

Part	Description
variable	A Boolean variable
object	An object expression that evaluates to a **RepositoryObjectVersion** object

See Also

IWorkspaceItem Interface

RepositoryObjectVersion Object

RepositoryObjectVersion IsFrozen Property

This property indicates whether the object version is frozen. This property is read-only.

Syntax

variable = object.**IsFrozen**

The **IsFrozen** property syntax has the following parts.

Part	Description
variable	A Boolean variable
object	An object expression that evaluates to a **RepositoryObjectVersion** object

See Also

RepositoryObjectVersion Object

RepositoryObjectVersion Name Property

This property is a character string that contains the name of the repository object version.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object**.Name**=*string*

The **Name** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **RepositoryObjectVersion** instance
string	A variable length string that can be a maximum of 255 characters

Remarks

When you retrieve an object version name, there are several places the repository engine can look for a name.

When you change the value of this property, there may be several names the repository engine tries to change.

Note that when you change this property, the repository engine can, in some circumstances, change some names but not change others. For example, if an object version is the destination of three naming relationships and also implements the **INamedObject** interface, this method will try to change four names. The method returns success if any of the four attempts succeeds.

See Also

Changing an Object Version's Name

INamedObject Interface

IRepositoryItem Interface

Naming Objects, Collections, and Relationships

RepositoryObjectVersion Object

Retrieving an Object Version's Name

RepositoryObjectVersion ObjectID Property

This property is the object identifier for the repository object version. The object identifier is unique across all repositories. To copy this property to another variable, use a variable declared as a **Variant**. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObject** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**ObjectID**

The **ObjectID** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryObject** object

Remarks

This property yields the object identifier, not the object-version identifier.

See Also

Assigning Object Identifiers

IRepositoryObject Interface

Object Identifiers and Internal Identifiers

RepositoryObjectVersion Object

RepositoryObject Object

RepositoryObjectVersion InternalID Property

RepositoryObjectVersion PredecessorCreationVersion Property

The **PredecessorCreationVersion** property is the **RepositoryObjectVersion** instance from which this repository object version was created. This property is read-only.

Syntax

Set variable = object.**PredecessorCreationVersion**

The **PredecessorCreationVersion** property syntax has the following parts.

Part	Description
variable	A variable declared as **RepositoryObjectVersion** object. It receives the object version from which the current object version was created.
object	An object expression that evaluates to a **RepositoryObjectVersion** instance.

Remarks

This property applies to object versions that are subsequently created from an existing object version. If you invoke this method for the first version of an object, it returns an error.

The **PredecessorCreationVersion** property identifies the **RepositoryObjectVersion** by object identifier or object name.

See Also

Assigning Object Identifiers

Object Identifiers and Internal Identifiers

RepositoryObjectVersion Object

RepositoryObjectVersion Repository Property

The **Repository** property is the open repository instance or workspace through which this repository object was instantiated. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObject** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Repository**

The **Repository** property syntax has the following parts.

Part	Description
variable	A variable declared as a reference to any object implementing the **IRepository** interface. It receives the object that represents the open **Repository** instance or the workspace.
object	An object expression that evaluates to a **RepositoryObjectVersion** object.

Remarks

The returned reference can refer to either a repository instance or a workspace. If it refers to a workspace, you manipulate the item within the context of that workspace. If it refers to a repository object, you manipulate the item within the context of a shared repository instance.

See Also

IRepositoryObject Interface

Repository Object

RepositoryObjectVersion Object

RepositoryObjectVersion ResolutionType Property

This property indicates the resolution technique by which the repository engine selected a reference to the current version (rather than a reference to some other version of the same object). This is a read-only property.

Syntax

integer = object.**ResolutionType**

The **ResolutionType** property syntax has the following parts.

Part	Description
integer	An integer that indicates the strategy used by the repository engine to select a specific object. 1= SPECIFIEDVERSION. This value appears when you explicitly select a specific object version. 2=LATESTVERSION. This value appears when the most recently created version is selected. 3=VERSIONINWORKSPACE. This value appears when the object version in the workspace is selected. 4=PINNEDVERSION. This value appears when the pinned target object version of the relationship that you are currently navigating is selected.
object	An object expression that evaluates to a **RepositoryObjectVersion** object.

Remarks

The repository engine automatically sets the value of the **ResolutionType** property whenever you retrieve an object version. For more information about how the repository engine selects object versions, see Resolution Strategy for Objects and Object Versions.

See Also

RepositoryObjectVersion Object

RepositoryObjectVersion InternalID Property

RepositoryObjectVersion Type Property

This property specifies the type of the **RepositoryObjectVersion** instance. More specifically, it is the object identifier of the object definition object for the repository object. To copy this property to another variable, use a variable declared as a **Variant**. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**Type**

The **Type** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryObjectVersion** object

Remarks

The **Type** property is the object identifier of the class to which the new object conforms. For example, to manipulate an object instance of a **Storage** class, specify the object identifier of the **Storage** class upon which the object instance is based.

An object in the repository is simultaneously a repository object, an Automation object, and an object of a specific type, as defined by an information model. The **Type** property identifies a specific object in an information model. The model-specific object is identified by its object identifier. The value of an object identifier is used as the value of the **Type** property for all repository objects that conform to that object definition.

See Also

IRepositoryItem Interface

Object Identifiers and Internal Identifiers

RepositoryObjectVersion Object

RepositoryObjectVersion VersionID Property

This property is the object-version identifier for the **RepositoryObjectVersion** instance. The object-version identifier is unique across all repositories. To copy this property to another variable, use a variable declared as a **Variant**. This property is read-only.

Syntax

object.**VersionID**

The **VersionID** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryObjectVersion** object

Remarks

This property yields the object-version identifier, not the object identifier.

See Also

Object Identifiers and Internal Identifiers

RepositoryObjectVersion Object

RepositoryObjectVersion VersionInternalID Property

This property is the internal object-version identifier that the repository engine uses to refer to this object. The internal object-version identifier is unique within the repository instance, but not unique across all repositories. To copy this property to another variable, use a variable declared as a **Variant**. This property is read-only.

Syntax

object.**VersionInternalID**

The **VersionInternalID** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryObjectVersion** object

Remarks

This property yields the internal object-version identifier, not the internal object identifier.

See Also

Object Identifiers and Internal Identifiers

RepositoryObjectVersion Object

RepositoryObjectVersion ObjectID Property

RepositoryObjectVersion CreateVersion Method

This method creates a new version of a repository object, based on the current version.

Syntax

set variable = object.**CreateVersion(***versionID***)**

The **Refresh** method syntax has the following parts.

Part	Description
variable	An object expression that evaluates to a **RepositoryObjectVersion** object. It receives a reference to the newly created object version.
object	An object expression that evaluates to a **RepositoryObjectVersion** object.
versionID	The value you want the repository to use as an object-version identifier for the newly created object version. If you want the repository to choose a value for you, set this parameter to EXTVERSIONID_NULL, or you can leave it blank.

Remarks

The current object version must be frozen.

The repository engine creates the new version as unfrozen. Its property values are identical to the property values of the current object version.

For each of the predecessor version's origin relationship collections, the repository engine takes this action:

- If the corresponding relationship type has the COLLECTION_NEWORGVERSIONSPARTICIPATE flag set, the repository engine copies the collection to the newly created version.

- If the corresponding relationship type does not have the COLLECTION_NEWORGVERSIONSPARTICIPATE flag set, the repository engine does not copy the collection to the new version.

You cannot invoke this method while operating in a workspace.

See Also

Assigning Object Identifiers

Object Identifiers and Internal Identifiers

RepositoryObjectVersion Object

RepositoryObjectVersion Delete Method

This method deletes the repository object version from the repository.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**Delete**

The **Delete** method syntax has the followings part:

Part	Description
object	An object expression that evaluates to a **RepositoryObjectVersion** instance

Remarks

A delete operation succeeds only under certain conditions.

If an object version has a successor, it cannot be deleted. To delete an object version that has successors, you must delete all successors first.

If an object is checked out to a workspace, you must invoke **Delete** from within that workspace.

If the object version satisfies both of these restrictions, the repository engine deletes it and any of its relationships, including any delete-propagating origin relationships. For each relationship, the repository engine considers performing one or more propagated deletions.

See Also

IRepositoryItem Interface

RepositoryObjectVersion FreezeVersion Method

This method freezes the current **RepositoryObjectVersion** object.

Syntax

Call object.**FreezeVersion**

The **FreezeVersion** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryObjectVersion** object

Remarks

To freeze an object version, the object version must be unfrozen. You can only freeze an object version that is contained by a shared repository. You cannot freeze an object version that is checked out to a workspace.

Freezing an object version prevents changes to property values, collection attributes, and versioned relationships. Specifically, you cannot resize or resequence origin collections. Furthermore, you cannot change the versioned relationships of origin collections. That is, you cannot enlarge or shrink a **TargetVersions** collection of an origin versioned relationship; and you cannot pin or unpin a target object version. However, you can change the name by which the origin object version refers to the target object.

Note Annotational properties are an exception. You can modify the annotational properties of a frozen object version.

The **FreezeVersion** method fails in the following conditions:

- If you call this method for an item currently checked out to any workspace (including the workspace in which you are working), it returns an error.

- If the to-be-frozen object version includes any nonnull origin collection whose corresponding collection type has the COLLECTION_REQUIRESFREEZE flag set, and that nonnull collection includes an item whose **TargetVersions** collection contains an unfrozen object version, the method fails.

See Also

RepositoryObjectVersion Object

RepositoryObjectVersion IsFrozen Property

Workspace CheckIn Method

RepositoryObjectVersion Lock Method

Use this method to lock the repository object version. Locking the object version prevents other processes from locking it while you are working with it. The lock is released when you end the current transaction.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**Lock**

The **Lock** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryObjectVersion** instance

See Also

IRepositoryItem Interface

RepositoryObjectVersion Object

RepositoryObjectVersion MergeVersion Method

This method changes the current object version by combining its property values and origin collections with the property values and origin collections of another version of the same object.

Syntax

Call object.**MergeVersion(***otherVersion* , *flags***)**

The **MergeVersion** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **RepositoryObjectVersion** object.
otherVersion	An object expression that evaluates to a **RepositoryObjectVersion** object. It is the object version that has values that you want to merge into the current object version.
flags	A flag indicating which version (the current version or the other version) the repository should use as the primary version of the merge operation. Primary=1 indicates that the predecessor object is primary and the current object is secondary. Secondary=2 indicates that the predecessor object is secondary and the current object is primary.

Remarks

Merging two object versions requires that the current object version is unfrozen, and the other object version is frozen. Furthermore, the two object versions must be versions of the same object.

Depending on how you set flags, one object version is the Primary Version and

the other object version is the Secondary Version. **MergeVersion** compares the property values and collections of each object version to a third version, called the Basis Version. During the merge, the repository engine considers each property and origin collection type in turn. Relationships are inserted at the end of the sequenced collection.

Merging object versions is a complex process. For more information about the merge process and how the repository engine selects values, see Merge Overview.

See Also

Merging Object Versions

RepositoryObjectVersion Object

RepositoryObjectVersion Refresh Method

This method refreshes the cached image of the repository object version. Only cached data that has not been changed by the current process is refreshed.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObject** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**Refresh(***milliSecs***)**

The **Refresh** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **RepositoryObjectVersion** object.
milliSecs	This value is ignored. It is kept for backward compatibility.

See Also

IRepositoryObject Interface

RepositoryObjectVersion Object

RepositoryObjectVersion ObjectVersions Collection

An **ObjectVersions** collection contains all of the **RepositoryObjectVersion** objects that are versions of the same repository object.

Syntax

Set variable = object.**ObjectVersions**(*index*)

The **ObjectVersions** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **VersionCol** object. It receives a reference to the **Versions-of-Object** collection.
object	An object expression that evaluates to a **RepositoryObjectVersion** object.
index	An integer index that identifies which property in the collection to address. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by *object*.**Properties**.**Count**. For more information, see Selecting Items in a Collection.

Remarks

Within the returned collection, the repository engine sequences the items in order of their creation, with the oldest object version first.

You cannot modify the collection. To add a new object version, use the **CreateVersion** method of the **RepositoryObjectVersion** object.

See Also

RepositoryObjectVersion Object

RepositoryObject Object

Relationship Object

RepositoryObjectVersion PredecessorVersions Collection

A **PredecessorVersions** collection contains all of the **RepositoryObjectVersion** objects that are immediate predecessors of the current object version.

Syntax

Set variable = object.**PredecessorVersions**(*index*)

The **PredecessorVersions** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **VersionCol** object. It receives a reference to the **Predecessor-Versions** collection.
object	An object expression that evaluates to a **RepositoryObjectVersion** instance.
index	An integer index that identifies which property in the collection to address. The valid range is from one to the total number of elements in the collection. For more information, see Selecting Items in a Collection.

Remarks

This method returns only the immediate predecessors of the current object version. If you invoke this method for the first version of an object, it returns an empty collection.

Within the returned collection, the repository engine sequences the items in order of their creation, with the oldest object version first. Objects are indicated by object identifier or object name.

You cannot modify the collection directly. To enlarge the set of predecessor versions of an object, use the **MergeVersion** method of the **RepositoryObjectVersion** object.

See Also

Assigning Object Identifiers

Object Identifiers and Internal Identifiers

RepositoryObjectVersion Object

Version Graph

RepositoryObjectVersion Properties Collection

The **Properties** collection contains all of the persistent properties on all of the interfaces that are attached to the repository object version.

This collection is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObject2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Properties(***index***)**

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. It receives the specified property.
object	An object expression that evaluates to a **RepositoryObjectVersion** object.
index	An integer index or a property name that identifies which property in the collection is to be addressed. For an index, the valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by *object*.**Properties**.**Count**. For property names, unique-naming constraints may apply. For more information, see Selecting Items in a Collection.

See Also

IRepositoryObject2 Interface

Naming and Unique-Naming Collections

RepositoryObjectVersion SuccessorVersions Collection

A **SuccessorVersions** collection contains all of the **RepositoryObjectVersion** objects that are immediate successors of the current object version.

Syntax

Set variable = object.**SuccessorVersions**(*index*)

The **SuccessorVersions** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **VersionCol** object. It receives a reference to the **SuccessorVersions** collection.
object	An object expression that evaluates to a **RepositoryObjectVersion** instance.
index	An integer index that identifies which property in the collection to address. The valid range is from one to the total number of elements in the collection. For more information, see Selecting Items in a Collection.

Remarks

This method returns only the immediate successors of the current object version. If the current object version has no successors, this method returns an empty collection.

Within the returned collection, the repository engine sequences the items in order of their creation, with the oldest object version first. Objects are indicated by object identifier or object name.

You cannot modify the collection directly. To enlarge the set of successor versions of an object, use the **CreateVersion** method of the **RepositoryObjectVersion** object.

See Also

RepositoryObjectVersion Object

Version Graph

RepositoryObjectVersion Workspaces Collection

The **Workspaces** collection contains all of the workspaces in which the object version is present.

This collection is not attached to the default interface for the repository Automation object; it is attached to the **IWorkspaceItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Workspaces**(*index*)

The **Workspaces** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **VersionCol** object. It receives a reference to the **WorkspacesOfVersion** collection.
object	An object expression that evaluates to a **RepositoryObjectVersion** object.
index	An integer index that identifies which member in the collection is to be addressed. The valid range is from one to the total number of members in the collection. For more information, see Selecting Items in a Collection.

Remarks

A workspace is a repository object. Within the **Workspaces** collection, workspaces are identified by object identifier or object name.

See Also

IWorkspaceItem Interface

RepositoryObjectVersion Object

Relationship Object

RepositoryTransaction Object

The repository engine supports the bracketing of multiple changes within the scope of a transaction. Changes to a repository that are bracketed within a transaction are either all committed or all undone, depending on the way that the transaction is completed. Repository methods that are reading data from the repository may be executed outside of a transaction, but methods that write data must be bracketed within a transaction.

You cannot directly instantiate a **RepositoryTransaction** object. When you connect to a repository, a **RepositoryTransaction** object is created for you. It is accessible through the **Repository Transaction** property.

When to Use

Use the **RepositoryTransaction** object to manage repository transactions.

Properties

Property	Description
Status	The transaction status of the repository

Methods

Method	Description
Abort	Cancels the current transaction
Begin	Begins a new transaction
Commit	Commits the current transaction
Flush	Flushes uncommitted changes to the repository database
GetOption	Retrieves various transaction options
SetOption	Sets various transaction options

Remarks

Only one transaction can be active at a time for each opened **Repository** instance. Nesting of **Begin** or **Commit** method invocations is permitted, but no actual nesting of transactions occurs.

See Also

Repository Object

Repository Transaction Property

Managing Transactions and Threads

RepositoryTransaction Status Property

This property indicates what the current transaction status is for the **Repository** instance. If the value is zero, no transaction is active. If the value is nonzero, a transaction is active. To copy this property to another variable, use a variable that is declared as a **Variant**. This is a read-only property.

Syntax

object.**Status**

The **Status** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryTransaction** object for the open **Repository** instance

Remarks

A transaction is considered active until the **Commit** method has successfully executed and the nested transaction count has been decremented to zero. Depending upon the data-flushing capabilities of the underlying database server, the data associated with a committed transaction may or may not be written to the physical storage device when the **Commit** method returns control to its caller.

See Also

Repository Transaction Property

RepositoryTransaction Commit Method

RepositoryTransaction Object

RepositoryTransaction Abort Method

This method cancels the currently active transaction for an open repository instance. All updates made during the transaction are rolled back. The nested transaction count is set to zero.

Syntax

Call object.**Abort**

The **Abort** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryTransaction** object for the currently open **Repository** instance

See Also

Repository Transaction Property

RepositoryTransaction Object

RepositoryTransaction Begin Method

This method increments the nested transaction count by one. If there is no active transaction, this method begins a transaction for the open **Repository** instance.

Syntax

Call object.**Begin**

The **Begin** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryTransaction** object for the currently open **Repository** instance

See Also

Repository Transaction Property

RepositoryTransaction Object

RepositoryTransaction Commit Method

This method decrements the nested transaction count for an open **Repository** instance. If the currently active transaction is not nested, all changes made to repository data within the transaction are committed to the repository database. A transaction is not nested if the nested transaction count equals one.

Syntax

Call object.**Commit**

The **Commit** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryTransaction** object for the currently open **Repository** instance

See Also

Repository Transaction Property

RepositoryTransaction Object

RepositoryTransaction Flush Method

This method flushes cached changes to the repository database.

Syntax

Call object.**Flush**

The **Flush** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **RepositoryTransaction** object for the currently open **Repository** instance

Remarks

You can set flags that determine what happens to data during a transaction. Changes are not written to the database until the transaction is committed. If a concurrent SQL query is run against the repository database, the result of the query will not reflect the uncommitted changes. (This is usually the desired behavior.)

If your tool or application must be able to see uncommitted changes in SQL queries, you can use the **Flush** method to write uncommitted changes to the repository database. All changes made within the scope of the current transaction are flushed. Flushing uncommitted changes does not affect your ability to cancel a transaction using the **Abort** method.

To get and set flags, use the **GetOption** and **SetOption** methods.

See Also

Repository Transaction Property

RepositoryTransaction Object

RepositoryTransaction GetOption Method

RepositoryTransaction SetOption Method

TransactionFlags Enumeration

RepositoryTransaction GetOption Method

This method is used to retrieve various transaction options.

Syntax

variable = object.**GetOption(***whichOption***)**

The **GetOption** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **RepositoryTransaction** object for the currently open **Repository** instance.
whichOption	A parameter that specifies which option to retrieve. For more information about flag values and descriptions, see TransactionFlags Enumeration.
variable	A variable declared as a **Variant**. It receives the value of the specified option.

Remarks

You can only get an option that is already set. You can set an option using the **SetOption** method.

See Also

Repository Transaction Property

RepositoryTransaction Object

RepositoryTransaction SetOption Method

RepositoryTransaction SetOption Method

This method is used to set various transaction options.

Syntax

Call object.**SetOption(***whichOption*, *optionValue***)**

The **SetOption** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **RepositoryTransaction** object for the open **Repository** instance.
whichOption	A parameter that specifies which option to set. For more information about flag values and descriptions, see TransactionFlags Enumeration.
optionValue	The new value for the option.

Remarks

After you set an option, you can retrieve it using the **GetOption** method.

See Also

Repository Transaction Property

RepositoryTransaction Object

RepositoryTransaction GetOption Method

ReposProperties Object

A **Properties** collection is the set of persistent properties and collections that are attached to a repository object or relationship through a particular interface.

When to Use

Use the **ReposProperties** object to enumerate the collection of repository properties that are attached to a particular repository object or relationship.

Properties

Property	Description
Count	The count of the number of items in the collection
Item	Retrieves the specified property from the collection
Type	Retrieves the type of the interface to which these properties are attached

See Also

ReposProperty Object

ReposProperties Count Property

This property is a long integer that contains the count of the number of properties in the collection. This is a read-only property.

Syntax

object.**Count**

The **Count** property syntax has the following part.

Part	Description
object	The repository property collection

See Also

ReposProperties Object

ReposProperties Item Property

This property is used to retrieve a specific repository property from a **Properties** collection. This is a read-only property. There are three variations of this property.

Syntax

Set variable = object.**Item(***index***)**
Set variable = object.**Item(***objName***)**
Set variable = object.**Item(***objId***)**

The **Item** property syntax has the following parts.

Part	Description
variable	An object expression that evaluates to a **ReposProperty** object.
object	The repository property collection.
index	The index of the repository property to be retrieved from the collection.
objName	The name associated with the repository property to be retrieved from the collection.
ObjId	The object identifier of the property definition object for this property.

See Also

ReposProperties Object

ReposProperties Type Property

This property retrieves the object identifier for the interface definition of the interface to which these properties are attached. This object identifier is referred to as the type of the interface. This is a read-only property.

Syntax

variable = object.**Type**

The **Type** property syntax has these parts.

Part	Description
variable	A **Variant** that receives the object identifier for the interface definition
object	The repository property collection

See Also

ReposProperties Object

ReposProperty Object

A repository property is a persistent property or collection that is attached to an object instance. It provides generic access to the properties of repository objects.

When to Use

Use the **ReposProperty** object to access generic meta data about a repository property, or to set the value of a repository property. **ReposProperty** retrieves meta data type information from the repository object instance itself. This eliminates the need to access the information model to obtain data from the **ClassDef**, **InterfaceDef**, **PropertyDef**, or **CollectionDef** that defines the object.

If you are providing browsing functionality in a tool or application, you can use **ReposProperty** to retrieve data about an object. Based on the values you obtain through **ReposProperty**, you can access more specific meta data about the object instance.

ReposProperty retrieves meta data about an object by accessing cached data. It also provides properties and methods for handling special case scenarios when accessing binary large objects (BLOBs) or large text fields.

Properties

Property	Description
APIType	The C data type of the property. It returns an API type enumeration value for the property. This property is not a default interface member.
CurrentPosition	A position within a BLOB or large text field. It establishes a starting point anywhere within a BLOB or large text field for performing Read and Write operations. This property is not a default interface member.

Flags	Flags that specify attributes of an interface member, such as whether it is hidden, read-only, virtual, or derived. This property is not a default interface member.
IsBaseMember	A flag that indicates whether the property is a base member. This property is not a default interface member.
IsMostDerived	A flag that indicates whether the property is the most recently derived member of a base member. This property is not a default interface member.
IsOriginCollection	A flag that indicates whether the collection is the origin of the relationship. This property is not a default interface member.
IsReadOnly	A flag that, when set to TRUE, returns a value associated with the current property. This property is not a default interface member.
Name	The name of the property.
PropType	An in-memory pointer to an object instance. This property is not a default interface member.
Size	The size of a BLOB or large text field. This property is not a default interface member.
Type	The type of the property, expressed as an object name or object identifier.
Value	The value of the property.

Methods

Property	Description

Close	Directs the repository engine to stop reading from or writing to a BLOB or large text field. This method is not a default interface member.
Read	Reads a large property value provided through a BLOB or large text field, starting at the current position. This method is not a default interface member.
ReadFromFile	Reads the contents of a BLOB or large text field from a file. This method is not a default interface member.
Write	Writes a large property value to a BLOB or large text field, starting at the current position. This method is not a default interface member.
WriteToFile	Stores the contents of a BLOB or large text field as a file. This method is not a default interface member.

See Also

ReposProperties Object

ReposProperty APIType Property

This property returns the C data type of the property. The value is an API type enumeration value for the property. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposProperty2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**APIType**

The **APIType** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object

See Also

IReposProperty2 Interface

ReposProperty Object

SQL and API Types Used in Property Definitions

ReposProperty CurrentPosition Property

This property stores a position within a binary large object (BLOB) or large text field. It establishes a starting point anywhere within a BLOB or large text field for performing **Read** and **Write** operations without loading the BLOB or large text field in memory. You can set and retrieve this property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposPropertyLarge** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**CurrentPosition**

The **CurrentPosition** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object

See Also

IReposPropertyLarge Interface

Programming BLOBs and Large Text Fields

ReposProperty Object

ReposProperty Flags Property

This property returns enumerated values that specify attributes of an interface member, such as whether it is hidden, read-only, virtual, or derived. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposProperty2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Flags**=(*integer*)

The **Flags** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object.
integer	Flag values are bit flags, and they can be combined to set multiple options. For more information about flag values and descriptions, see the InterfaceMemberFlags Enumeration.

See Also

IReposProperty2 Interface

ReposProperty Object

ReposProperty IsBaseMember Property

This Boolean property returns a flag that indicates whether the property is a base member. For more information about base and derived members, see Member Delegation. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposProperty2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**IsBaseMember**

The **IsBaseMember** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object

See Also

IReposProperty2 Interface

ReposProperty Object

ReposProperty IsMostDerived Property

This Boolean property returns a flag that indicates whether the property is the most derived member of a base member. Member derivations are scoped to branches in the version graph. For example, if there are two branches and each one has derived members, the IsMostDerived property returns True for each branch. For more information about base and derived members, see Member Delegation. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposProperty2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**IsMostDerived**

The **IsMostDerived** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object

See Also

IReposProperty2 Interface

ReposProperty Object

ReposProperty IsOriginCollection Property

This Boolean property returns a flag that indicates whether the collection is the origin of the relationship. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposProperty2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**IsOriginCollection**

The **IsOriginCollection** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object

See Also

IReposProperty2 Interface

ReposProperty Object

ReposProperty IsReadOnly Property

This Boolean property returns True if the current property is read-only. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposProperty2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**IsReadOnly**

The **IsReadOnly** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object

Remarks

A property value is stored as a **Value** property.

See Also

IReposProperty2 Interface

ReposProperty Object

ReposProperty Value Property

ReposProperty Name Property

This property stores the name of the repository property. This property is read-only.

Syntax

string=object.**Name**

The **Name** property syntax has the following parts.

Part	Description
string	A variable length string that can be a maximum of 255 characters
object	An object expression that evaluates to a **ReposProperty** object

See Also

ReposProperty Object

ReposProperty PropType Property

This property returns an in-memory pointer to a **PropertyDef** instance. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposProperty2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**PropType**

The **PropType** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object

Remarks

After you obtain an in-memory pointer to an object instance, you have the **PropertyDef** object upon which the instance is based.

See Also

IReposProperty2 Interface

Object Identifiers and Internal Identifiers

ReposProperty Object

Assigning Object Identifiers

ReposProperty Size Property

This property returns the size of a binary large object (BLOB) or large text field. This property is read-only.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposPropertyLarge** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**Size**

The **Size** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object

See Also

IReposPropertyLarge Interface

ReposProperty Object

ReposProperty Type Property

This property is the type of the repository property; that is, it is the object identifier of the **PropertyDef** or **CollectionDef** object to which this repository property conforms. You use a **Variant** variable to receive the **Type** property. This property is read-only.

Syntax

object.**Type**

The **Type** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object

See Also

Object Identifiers and Internal Identifiers

ReposProperty Object

ReposProperty PropType Property

ReposProperty Value Property

This property is the value of the repository property. You use a **Variant** variable to receive this property value.

Syntax

object.**Value**
object.**Value** = newValue

The **Value** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object
newValue	An expression that evaluates to a value of the appropriate type for the repository property

See Also

ReposProperty Object

ReposProperty PropType Property

ReposProperty Close Method

This method directs the repository engine to stop reading from or writing to a binary large object (BLOB) or large text field.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IReposPropertyLarge** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**Close**

The **Close** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object

Remarks

When you release a property object, the repository engine automatically closes a BLOB or large text field for you. However, if you want to free up memory or terminate a read-write operation before releasing an object, you can use **Close** to do so.

Be aware that if you keep an object in memory and you have not called **Close** before committing a transaction, a **Write** operation will not be committed during the transaction.

See Also

IReposPropertyLarge Interface

ReposProperty Object

ReposProperty Read Method

ReposProperty Write Method

ReposProperty Read Method

This method reads into memory a large property value provided through a binary large object (BLOB) or large text field, starting at the current position.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IReposPropertyLarge** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**Read**(*sizeToRead*, *pSizeRead*, *psBlob*)

The **Read** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object
sizeToRead	The amount of memory you allocate in advance to store the data to be read
pSizeRead	The actual size of the data that is read
psBlob	A Variant pointer to a location that stores the data to be read. The location you specify must be able to accommodate the amount of preallocated memory

Remarks

After you read data, you can use the **Close** method to release memory and resources.

See Also

IReposPropertyLarge Interface

ReposProperty Close Method

ReposProperty CurrentPosition Property

ReposProperty Object

ReposProperty ReadFromFile Method

ReposProperty Write Method

ReposProperty ReadFromFile Method

This method sets a binary large object (BLOB) or large text field property value to the contents of a file. This method does not support the **CurrentPosition** property. If content exists within the BLOB or large text field, the **ReadFromFile** method will overwrite it.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IReposPropertyLarge** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**ReadFromFile**(*BSTR filename*)

The **ReadFromFile** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object
BSTR filename	The fully qualified path and file name that provides content to the BLOB or large text field

Remarks

After you read data, you can use the **Close** method to release memory and resources.

See Also

IReposPropertyLarge Interface

ReposProperty Close Method

ReposProperty Object

ReposProperty Read Method

ReposProperty Write Method

ReposProperty Write Method

This method takes data and writes it to a binary large object (BLOB) or large text field, starting at the current position.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IReposPropertyLarge** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**Write**(*psBlob*)

The **Write** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object
psBlob	A Variant pointer to the location where data is to be written. The location you specify must contain the amount of preallocated memory

Remarks

After you write data, you can use the **Close** method to release memory and resources.

See Also

IReposPropertyLarge Interface

ReposProperty Close Method

ReposProperty CurrentPosition Property

ReposProperty Object

ReposProperty WriteToFile Method

ReposProperty WriteToFile Method

This method stores the contents of a binary large object (BLOB) or large text field to a file. You must specify a fully qualified path and file name.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IReposPropertyLarge** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**WriteToFile**(*BSTR filename*)

The **WriteToFile** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **ReposProperty** object
BSTR filename	The fully-qualified path and file name that stores the value of a BLOB or large text field

Remarks

After you write data, you can use the **Close** method to release memory and resources.

See Also

IReposPropertyLarge Interface

ReposProperty Close Method

ReposProperty Object

ReposProperty Write Method

TransientObjectCol Object

The **TransientObjectCol** object is an object collection that you can create and dynamically populate at run time using script and object methods rather than stored data in a repository database. **TransientObjectCol** simulates a standard, stored repository object collection.

You can have multiple transient object collections at one time. The object collection can contain only repository objects. Although enumeration is supported, sequencing is not. Except for the fact that the object collection is not saved to a repository database, it is identical in functionality to the **ObjectCol** object.

TransientObjectCol is instantiated by application code. Applications that use **TransientObjectCol** can treat the object collection exactly the same way as any other repository object collection.

Objects represented in **TransientObjectCol** are not versioned.

When to Use

Use this object to create an object collection that is instantiated by application code and populated dynamically at run time. With this object, you can:

- Create an object collection that is not stored in a repository database.
- Get a count of the number of objects in the collection.
- Add and remove objects to and from the collection.

Properties

Property	Description
Count	The count of the number of objects in the collection
Item	Retrieves a specific object from the collection

Methods

Method	Description
Add	Adds an object to the collection.
Refresh	Supports backward compatibility of the **Refresh** method. This method is not used.
Remove	Removes an object from the collection.

See Also

ITargetObjectCol Interface

MethodDef Object

ObjectCol Class

ScriptDef Object

TransientObjectCol Class

TransientObjectCol Count Property

This property is a long integer that contains the count of the number of items in the collection. This is a read-only property.

Syntax

Object.**Count**

The **Count** property syntax has the following part.

Part	Description
object	The object collection created by **TransientObjectCol**

See Also

TransientObjectCol Object

TransientObjectCol Add Method

This method is used to add target objects to an object collection.

Syntax

object**.Add**(*reposObj*, *objName*)

The **Add** method syntax has the following parts.

Part	Description
object	The object collection created by **TransientObjectCol**.
reposObj	The repository object to be added to the collection.
objName	The name that the new collection is to use for *reposObj*. This parameter is optional.

Remarks

Populating a **TransientObjectCol** is done using the **Add** method for each object that you want to add to the collection.

See Also

TransientObjectCol Object

TransientObjectCol Remove Method

TransientObjectCol Refresh Method

This method has no effect. It is included for backward compatibility only.

Syntax

Call object.**Refresh**(*milliSecs*)

The **Refresh** method syntax has the following parts.

Part	Description
object	The object collection created by **TransientObjectCol**.
milliSecs	This value is ignored. It is kept for backward compatibility.

See Also

TransientObjectCol Object

TransientObjectCol Remove Method

This method removes a specified object from a transient object collection.

Syntax

Call object.**Remove**(*index*) **Call** object.**Remove**(*objName*)
Call object.**Remove**(*objID*)

The **Remove** method syntax has the following parts.

Part	Description
object	The object collection created by **TransientObjectCol**.
index	The index of the object to be removed from the collection. For more information, see Selecting Items in a Collection.
objName	The object that uses this name for its destination object is to be removed from the collection.
objID	The object identifier of the object to be removed from the collection.

Remarks

This property removes a specific repository object from the collection. You can identify an object by its position in the collection (as indicated by the index) or by identifier.

See Also

Object Identifiers and Internal Identifiers

TransientObjectCol Object

TransientObjectCol Item Collection

Use this property to retrieve an object from the collection. This is a read-only property. There are two variations of this property.

Syntax

Set variable = object.**Item**(*index*)
Set variable = object.**Remove**(*objName*)
Set variable = object.**Item**(*objId*)

The **Item** property syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObject** or **RepositoryObjectVersion**. It receives the specified repository object.
object	The object collection created by **TransientObjectCol**.
index	The index of the repository object to be retrieved from the collection.
objName	The name of the object to be retrieved from the collection.
objId	The object identifier of the repository object to be retrieved from the collection.

Remarks

This property retrieves a specific repository object from the collection by its position in the collection (as indicated by the index) or by identifier.

See Also

Object Identifiers and Internal Identifiers

RepositoryObject Object

RepositoryObjectVersion Object

TransientObjectCol Object

Workspace Object

A workspace is a repository object that can provide a context for your work that is separate from other work occurring in the repository. A workspace is a subset of a larger, shared repository. Within a workspace, you can have only one version of any given repository object at a time.

When to Use

Use the **Workspace** object to control the contents of a workspace.

Properties

Property	Description
CheckedOutToWorkspace	Always null, because a workspace cannot be contained in or checked out to another workspace. This property is not a default interface member.
Interface	The specified object interface. This property is not a default interface member.
InternalID	The internal object identifier of the workspace. This property is not a default interface member.
IsCheckedOut	Always FALSE, because a workspace cannot be present in (or checked out to) another workspace. This property is not a default interface member.

IsFrozen	Always FALSE, because you cannot freeze a workspace. This property is not a default interface member.
Name	The name of the workspace. This property is not a default interface member.
Object	A property used to retrieve a particular repository object. This property is not a default interface member.
ObjectID	The object identifier of the workspace. This property is not a default interface member.
MajorDBVersion	The major version number of the first repository engine version that introduced this database format. This property is not a default interface member.
MinorDBVersion	The minor version number of the first repository engine version that introduced this database format. This property is not a default interface member.
PredecessorCreationVersion	Always null, because each workspace has only one version. This property is not a default interface member.
Repository	The open repository instance through which

	this workspace was instantiated. This property is not a default interface member.
ResolutionType	Always LATEST_VERSION, because each workspace has only one version. This property is not a default interface member.
RootObject	The root repository object of the open repository. This property is not a default interface member.
Transaction	The transaction-processing interface. This property is not a default interface member.
Type	An object identifier of the type to which this workspace conforms. The property is always the object identifier of the workspace definition object of the Repository Type Information Model (RTIM). This property is not a default interface member.
Version	Retrieves the specified object version. This property is not a default interface member.
VersionID	The object-version identifier of the workspace. This property is not a default interface member.
VersionInternalID	The internal object-version identifier of the workspace.

Methods

Method	Description
Checkin	Returns an error, because a workspace cannot be present in (or checked out to) another workspace. This method is not a default interface member.
Checkout	Returns an error, because a workspace cannot be present in (or checked out to) another workspace. This method is not a default interface member.
Create	Returns an error, because you cannot create a new repository database from within a workspace. This method is not a default interface member.
CreateObject	Creates a new repository object in the open repository. This method is not a default interface member.
CreateVersion	Returns an error, because each workspace has only one version. This method is not a default interface member.
Delete	Deletes a workspace. This method is not a default interface member.
FreezeVersion	Returns an error, because you cannot freeze a workspace. This method is not a default interface member.
InternalIDToObjectID	Converts an internal identifier into an object

	identifier.
InternalIDToVersionID	Converts an internal object-version identifier into an object-version identifier. This method is not a default interface member.
Lock	Locks the workspace. This method is not a default interface member.
MergeVersion	Returns an error, because each workspace has only one version. This method is not a default interface member.
ObjectIDToInternalID	Converts an object identifier into an internal identifier. This method is not a default interface member.
Open	Returns an error, because you cannot open a new repository database from within a workspace. This method is not a default interface member.
Refresh	Supports backward compatibility of the **Refresh** method. This method is not used. This method is not a default interface member.
Refresh (from IRepositoryObjectVersion)	Supports backward compatibility of the **Refresh** method. This method is not used. This method is not a default interface member.
VersionIDToInternalID	Converts an object-version identifier into an internal object-version identifier. This method is not a default interface member.

Collections

Collection	Description
Checkouts	The collection of object versions checked out to the workspace.
Containers	A collection that contains this workspace. The collection has only one item, the root object. This collection is not a default interface member.
Contents	The collection of object versions present in the workspace.
ObjectVersions	The collection of all the versions of the repository object representing the workspace. It always contains one item, because you cannot invoke **CreateVersion** on a workspace. This collection is not a default interface member.
PredecessorVersions	Always null, because you cannot invoke **CreateVersion** on a workspace. This collection is not a default interface member.
Properties	The collection of all properties that are attached to the workspace.
SuccessorVersions	Always null, because you cannot invoke **CreateVersion** on a workspace. This collection is not a default interface member.
Workspaces	Always null, because a workspace cannot be contained in other workspaces. This collection is not a default interface member.

Workspace CheckedOutToWorkspace Property

The property is always null, because a workspace cannot be present in (or checked out to) another workspace.

Remarks

This member is exposed by the **IRepositoryObjectVersion** interface, which is a nondefault interface for this object. Because of how interface inheritance works, this member is made available to the **Workspace** object by convention.

See Also

RepositoryObjectVersion Object

Workspace Object

Workspace Interface Property

This property obtains an alternate interface for the default interface of the workspace. This is a read-only property. There are three variations of this property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Interface(***interfaceId***)**
Set variable = object.**Interface(***objectId***)**
Set variable = object.**Interface(***interfaceName***)**

The **Interface** property syntax has the following parts.

Part	Description
variable	An object variable. It receives the workspace with the specified interface as the default interface.
object	An object expression that evaluates to a **Workspace** object.
interfaceId	The interface identifier for the interface to be retrieved.
objectId	The object identifier for the interface definition to which the interface you want to retrieve conforms.
interfaceName	A string that contains the name of the interface to be retrieved.

See Also

Object Identifiers and Internal Identifiers

RepositoryObjectVersion Object

Workspace Object

Workspace InternalID Property

This property is the internal identifier that the repository engine uses to refer to this workspace. Each workspace has an internal identifier that is unique within the repository, but not unique across repositories. To copy this property to another variable, use a variable declared as a **Variant**. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**InternalID**

The **InternalID** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **Workspace** object

See Also

Object Identifiers and Internal Identifiers

RepositoryObject ObjectID Property

RepositoryObjectVersion Object

Workspace Object

Workspace IsCheckedOut Property

The property is always FALSE because a workspace cannot be present in (or checked out to) another workspace.

Remarks

This member is exposed by the **IRepositoryObjectVersion** interface, which is a nondefault interface for this object. Because of how interface inheritance works, this member is made available to the **Workspace** object by convention.

See Also

RepositoryObjectVersion Object

Workspace Object

Workspace IsFrozen Property

The property is always FALSE, because you cannot freeze a workspace.

Remarks

This member is exposed by the **IRepositoryObjectVersion** interface, which is a nondefault interface for this object. Because of how interface inheritance works, this member is made available to the **Workspace** object by convention.

See Also

RepositoryObjectVersion Object

Workspace Object

Workspace MajorDBVersion Property

This property retrieves the major version number of the first repository engine version that introduced this database format. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepository2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**MajorDBVersion**

The **MajorDBVersion** property syntax has the following parts.

Part	Description
variable	Declared as **long**. It receives the database major version.
object	The object that represents the open repository instance.

See Also

IRepository2 Interface

RepositoryObjectVersion Object

Workspace MinorDBVersion Property

Workspace Object

Workspace MinorDBVersion Property

This property retrieves the minor version number of the first repository engine version that introduced this database format. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepository2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**MinorDBVersion**

The **MinorDBVersion** property syntax has the following parts.

Part	Description
variable	Declared as **long**. It receives the database minor version.
object	The object that represents the open repository instance.

See Also

IRepository2 Interface

Workspace MajorDBVersion Property

Workspace Object

Workspace Name Property

This property is a character string that contains the name of the workspace.

The **Name** property is normally a property of the relationship for which this repository object is the destination object. However, because the **Workspace** object exposes the **INamedObject** interface, the name retrieved is the value of the **Name** property exposed by this interface. When you set this property, the repository engine sets two things: the name property of the **INamedObject** interface, and the name associated with every naming relationship in which this workspace participates as the destination object.

This property is not attached to the default interface for the repository Automation object; it is attached to the **INamedObject** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**Name**

The **Name** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **Workspace** object

Remarks

In addition to the **Name** property exposed by the **INamedObject** interface, each workspace can have other names, because each workspace has a destination naming relationship to the root object. When you retrieve the name of a workspace, the repository engine retrieves the value of the name as exposed by the **INamedObject** interface. When you set the name, the engine attempts to change some or all of the names of the workspace.

See Also

Changing an Object Version's Name

INamedObject Interface

Workspace Object

Workspace Object Property

This property retrieves a particular repository object. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Object(***objectId***)**

The **Object** property syntax has the following parts.

Part	Description
variable	Declared as a **RepositoryObject**. It receives the repository object.
object	The object that represents the workspace through which this program is interacting with the repository.
objectId	The object identifier for the repository object to be retrieved.

Remarks

The repository engine returns the specific version of the repository object that is present in the workspace. If no version of the object is present in the workspace, this property returns an error.

See Also

RepositoryObject Object

Workspace Object

Workspace ObjectID Property

This property is the object identifier for the workspace. Each workspace has an object identifier that is unique across all repositories. This is a read-only property. To copy this property to another variable, use a variable declared as a **Variant**.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**ObjectID**

The **ObjectID** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **Workspace** object

See Also

Object Identifiers and Internal Identifiers

RepositoryObjectVersion Object

Repository ObjectIDToInternalID Method

Workspace Object

Workspace PredecessorCreationVersion Property

The property is always null, because each workspace has only one version.

Remarks

This member is exposed by the **IRepositoryObjectVersion** interface, which is a nondefault interface for this object. Because of how interface inheritance works, this member is made available to the **Workspace** object by convention.

See Also

RepositoryObjectVersion Object

Workspace Object

Workspace Repository Property

This property is the open repository instance through which this workspace was instantiated. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Repository**

The **Repository** property syntax has the following parts.

Part	Description
variable	A variable declared as an instance of the **Repository** class. It receives the object that represents the open repository instance.
object	An object expression that evaluates to a **Workspace** object.

See Also

Repository Object

RepositoryObjectVersion Object

ReposProperty Object

Workspace Object

Workspace ResolutionType Property

The property is always LATEST_VERSION, because each workspace has only one version.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

RepositoryObjectVersion Object

Workspace Object

Workspace RootObject Property

This property is the root repository object for the open repository. This is a read-only property. The returned reference to the root object has the context of the current workspace.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepository** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**RootObject**

The **RootObject** property syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObject**. It receives the root repository object.
object	The object that represents the workspace through which this program is interacting with the repository.

See Also

ReposRoot Object

RepositoryObject Object

Workspace Object

Workspace Transaction Property

This property is the **RepositoryTransaction** object for the open repository instance. This is a read-only property.

Syntax

Set variable = object.**Transaction**

The **Transaction** property syntax has the following parts.

Part	Description
variable	A variable declared as an object. It receives the **RepositoryTransaction** object for this repository instance.
object	The object that represents the workspace through which this program is interacting with the repository.

Remarks

You can gain access to the **RepositoryTransaction** object by using this syntax. Then you can access the properties and methods of the **RepositoryTransaction** object by using the **variable**.*method* and **variable**.*property* syntax. You can also access the properties and methods of the **RepositoryTransaction** object directly, using syntax similar to that shown here:

Call object.**Transaction**.method

-or-

variable = object.**Transaction**.property

For more information about the **RepositoryTransaction** object and the methods and properties that it provides, see RepositoryTransaction Object.

See Also

Repository Transaction Property

Workspace Object

Workspace Type Property

This property specifies the type of the workspace. More specifically, it is the object identifier of the workspace definition object of the type information model. **Type** is a read-only property. To copy this property to another variable, use a variable declared as a **Variant**.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**Type**

The **Type** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **Workspace** object

Remarks

An object in an information model is identified by its object identifier. The value of an **ObjectID** property is used as the value of the **Type** property for all repository objects that conform to that object definition.

See Also

Object Identifiers and Internal Identifiers

RepositoryObject Object

RepositoryObjectVersion Object

Workspace Object

Workspace Version Property

This property retrieves a particular repository object version from the workspace. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepository2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Version(***versionId***)**

The **Version** property syntax has the following parts.

Part	Description
variable	Declared as a **RepositoryObjectVersion**. It receives the repository object version.
object	The object that represents the workspace through which this program is interacting with the repository.
versionId	The object-version identifier for the repository object to be retrieved.

Remarks

This method returns an error if the requested version is not present in the workspace.

See Also

IRepository2 Interface

RepositoryObjectVersion Object

Workspace Object

Workspace VersionID Property

This property is the object-version identifier for the workspace. Each workspace has an object-version identifier that is unique across all repositories. This is a read-only property. To copy this property to another variable, use a variable declared as a **Variant**.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**VersionID**

The **VersionID** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **Workspace** object

See Also

RepositoryObject InternalID Property

RepositoryObjectVersion Object

Workspace Object

Workspace VersionInternalID Property

This property is the internal object-version identifier that the repository engine uses to refer to this workspace. Each workspace has an internal object-version identifier that is unique within the repository, but not unique across repositories. This is a read-only property. To copy this property to another variable, use a variable declared as a **Variant**.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**VersionInternalID**

The **VersionInternalID** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **Workspace** object

See Also

Object Identifiers and Internal Identifiers

RepositoryObject ObjectID Property

RepositoryObjectVersion Object

Workspace Object

Workspace Checkin Method

This method always returns an error, because a workspace cannot be present in (or checked out to) another workspace.

Remarks

This member is exposed by the **IRepositoryObjectVersion** interface, which is a nondefault interface for this object. Because of how interface inheritance works, this member is made available to the **Workspace** object by convention.

See Also

RepositoryObjectVersion Object

Workspace Object

Workspace Checkout Method

This method always returns an error, because a workspace cannot be present in (or checked out to) another workspace.

Remarks

This member is exposed by the **IRepositoryObjectVersion** interface, which is a nondefault interface for this object. Because of how interface inheritance works, this member is made available to the **Workspace** object by convention.

See Also

RepositoryObjectVersion Object

Workspace Object

Workspace Create Method

This method creates a new repository instance. When operating on a workspace, this method always fails. To create a new repository database, use the **Create** method of the **Repository** object.

See Also

Repository Object

Repository Create Method

Workspace Object

Workspace CreateObject Method

This method creates a new **RepositoryObject** of the specified type. After you create a new object, you must include it in the workspace.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**CreateObject(***typeId***,** *objectId***)**

The **CreateObject** method syntax has the following parts.

Part	Description
variable	Declared as a **RepositoryObject**. It receives the new repository object.
object	The object that represents the workspace through which this program is interacting with the repository.
typeId	The type of the new object; that is, the object identifier of the object definition to which the new object conforms.
objectId	The object identifier to be assigned to the new object. To have the repository engine assign an object identifier for you, pass in ObjID_NULL or do not supply this optional parameter.

Remarks

This method creates a new object in the repository, but it does not insert the newly created object into the workspace in whose context you are operating.

This method can only be called from the shared repository but not from a workspace. The workaround is to create the object through the central repository and include it in the workspace.

See Also

RepositoryObject Object

Workspace Object

Workspace CreateVersion Method

This method always returns an error, because each workspace has only one version.

Remarks

This member is exposed by the **IRepositoryObjectVersion** interface, which is a nondefault interface for this object. Because of how interface inheritance works, this member is made available to the **Workspace** object by convention.

See Also

RepositoryObjectVersion Object

Workspace Object

Workspace Delete Method

This method deletes the workspace from the repository. Any relationships that connect the workspace to other objects are deleted. If the workspace is an origin object of a relationship collection, and the relationship type indicates that deletes are to be propagated, all destination objects are also deleted.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**Delete**

The **Delete** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **Workspace** object

See Also

Propagating Deletes

RepositoryObjectVersion Object

Workspace Object

Workspace FreezeVersion Method

This method always returns an error, because you cannot freeze a workspace.

Remarks

This member is exposed by the **IRepositoryObjectVersion** interface, which is a nondefault interface for this object. Because of how interface inheritance works, this member is made available to the **Workspace** object by convention.

See Also

RepositoryObjectVersion Object

Workspace Object

Workspace InternalIDToObjectID Method

This method translates an internal identifier into an object identifier. Internal identifiers are used by the repository engine to identify repository objects.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepository** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**InternalIDToObjectID(***internalId***)**

The **InternalIDToObjectID** method syntax has the following parts.

Part	Description
variable	Receives the object identifier
object	The object that represents the workspace through which this program is interacting with the repository
internalId	The internal identifier to be converted

Remarks

Repository object identifiers are globally unique, and are the same across repositories for the same object. Internal identifiers are unique only within the scope of a single repository.

The translation performed by this method is performed without loading the object in question. This enables database queries involving an object or relationship type identifier to be constructed without having to load the definition object.

See Also

Object Identifiers and Internal Identifiers

Repository ObjectIDToInternalID Method

Workspace Object

Workspace InternalIDToVersionID Method

This method translates an internal object-version identifier into an object-version identifier.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepository** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**InternalIDToVersionID(***internalVersionId***)**

The **InternalIDToVersionID** method syntax has the following parts.

Part	Description
variable	Receives the object-version identifier
object	The object that represents the current workspace through which this program is interacting with the repository
internalVersionId	The internal object-version identifier to be converted

Remarks

Repository object-version identifiers are globally unique, and they are the same across repositories for the same object version. Internal object-version identifiers are unique only within the scope of a single repository.

The translation performed by this method is performed without loading the object-version in question. This enables database queries involving an object or relationship type identifier to be constructed without having to load the definition object.

See Also

Workspace Lock Method

This method locks the workspace. Locking the workspace prevents other processes from locking the object describing the workspace while you are working with it. The lock is released when you end the current transaction.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**Lock**

The **Lock** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **Workspace** object

See Also

RepositoryObjectVersion Object

Workspace Object

Workspace MergeVersion Method

This method always returns an error, because there is only one version of each workspace.

Remarks

This member is exposed by the **IRepositoryObjectVersion** interface, which is a nondefault interface for this object. Because of how interface inheritance works, this member is made available to the **Workspace** object by convention.

See Also

RepositoryObjectVersion Object

Workspace Object

Workspace ObjectIDToInternalID Method

This method translates an object identifier into an internal identifier. Internal identifiers are used by the repository engine to identify repository objects.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**ObjectIDToInternalID(***objectId***)**

The **ObjectIDToInternalID** method syntax has the following parts.

Part	Description
variable	Receives the internal identifier
object	The object that represents the workspace through which this program is interacting with the repository
objectId	The object identifier to be converted

Remarks

Repository object identifiers are globally unique, and are the same across repositories for the same object. Internal identifiers are unique only within the scope of a single repository.

The translation performed by this method is performed without loading the object in question. This enables database queries involving an object or relationship type identifier to be constructed without having to load the definition object.

See Also

Object Identifiers and Internal Identifiers

Repository InternalIDToObjectID Method

Workspace Object

Workspace Open Method

This method connects to a repository database. When operating on a workspace, this method always fails.

To open (connect to) a repository, use the **Open** method of the **Repository** object.

Remarks

This member is exposed by the **IRepositoryObjectVersion** interface, which is a nondefault interface for this object. Because of how interface inheritance works, this member is made available to the **Workspace** object by convention.

See Also

Repository Create Method

Workspace Object

Workspace Refresh Method

This method refreshes all cached data for the open repository instance. Only cached data that has not been changed by the current process is refreshed. Even though you are operating within the context of a workspace, this method refreshes all cached data for the open repository instance, including cached data not present in the workspace.

Syntax

Call object.**Refresh(***milliseconds***)**

The **Refresh** method syntax has the following parts.

Part	Description
object	The object that represents the workspace through which this program is interacting with the repository
milliseconds	This value is ignored

See Also

Workspace Object

Workspace Refresh (from IRepositoryObjectVersion) Method

This method refreshes the cached image of the repository object that describes the workspace. Only cached data that has not been changed by the current process is refreshed.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryObjectVersion** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**Refresh(***milliSecs***)**

The **Refresh** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **Workspace** object
milliSecs	This value is ignored

See Also

Workspace Object

Workspace VersionIDToInternalID Method

This method translates an object-version identifier into an internal object-version identifier. Internal object-version identifiers are used by the repository engine to identify repository object versions.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepository2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

variable = object.**VersionIDToInternalID(***versionId***)**

The **VersionIDToInternalID** method syntax has the following parts.

Part	Description
variable	Receives the internal identifier
object	The object that represents the workspace through which this program is interacting with the repository
versionId	The object-version identifier to be converted

Remarks

Repository object-version identifiers are globally unique, and they are the same across repositories for the same object version. Internal object-version identifiers are unique only within the scope of a single repository.

The translation performed by this method is performed without loading the object version in question. This enables database queries involving an object or relationship type identifier to be constructed without having to load the definition object.

See Also

IRepository2 Interface

Object Identifiers and Internal Identifiers

Repository InternalIDToObjectID Method

RepositoryObjectVersion Object

Workspace Object

Workspace Checkouts Collection

This collection contains all repository object versions checked out to the workspace.

Syntax

Set variable = object.**Checkouts(***index***)**
Set variable = object.**Checkouts(***objectID***)**

The **Checkouts** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObjectVersion** object. It receives the specified object version.
object	An object expression that evaluates to a **Workspace** object.
index	An integer index that identifies which item in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**Checkouts**.**Count**. For more information, see Selecting Items in a Collection.
objectID	An object identifier of the object version checked out to the workspace.

See Also

Object Identifiers and Internal Identifiers

RepositoryObjectVersion Object

Workspace Object

Workspace Containers Collection

This collection specifies the root object that contains this workspace. Although the maximum size of the collection is defined as unlimited, the collection always contains one object. This is because there is only one root object, and only the root object can contain workspaces.

Syntax

Set variable = object.**Containers(***index***)**

The **Containers** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposRoot** object or as any object that supports the **IWorkspaceContainer** interface. It receives the object containing the interface.
object	The object that represents a **Workspace** object.
index	An integer index that identifies which element in the collection is to be addressed. This value must be one, because in this release, the root object is the only object that implements the **IWorkspaceContainer** interface. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**WsContainer_Contains_Workspace**	This is the type of relationship by which all items of the collection are connected to a

		common source object.
Source Is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-sensitive Names	Not applicable	Case-sensitive naming is not

		applicable for this collection.
Unique Names	Not applicable	Unique naming is not applicable for this collection.

See Also

ClassDef Object

InterfaceDef Object

ReposRoot Object

Workspace Object

Workspace Contents Collection

This collection contains all repository object versions present in the workspace.

Syntax

Set variable = object.**Contents(***index***)**
Set variable = object.**Contents(***objectID***)**

The **Contents** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObjectVersion** object. It receives the specified object version.
object	An object expression that evaluates to a **Workspace** object.
index	An integer index that identifies which item in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**Contents**.**Count**. For more information, see Selecting Items in a Collection.
objectID	An object identifier of the object version present in the workspace.

See Also

Object Identifiers and Internal Identifiers

RepositoryObjectVersion

Workspace Object

Workspace ObjectVersions Collection

This collection contains all **RepositoryObjectVersion** objects that are versions of the same repository object.

Syntax

Set variable = object.**ObjectVersions**

The **ObjectVersions** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **VersionCol** object. It receives a reference to the **Versions-of-Object** collection.
object	An object expression that evaluates to a **Workspace** object.

Remarks

Each **Workspace** object has only one version (because you cannot invoke the **CreateVersion** method on a workspace). Thus, this collection always contains only one item.

See Also

Relationship Object

RepositoryObjectVersion Object

Workspace Object

Workspace PredecessorVersions Collection

This collection contains all **RepositoryObjectVersion** objects that are immediate predecessors of the current object version.

Syntax

Set variable = object.**PredecessorVersions**

The **PredecessorVersions** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **VersionCol** object. It receives a reference to the **Predecessor-Versions** collection.
object	An object expression that evaluates to a **Workspace** object.

Remarks

Each **Workspace** object has only one version (because you cannot invoke the **CreateVersion** method on a workspace). Thus, this collection is always null.

See Also

RepositoryObjectVersion Object

Version Graph

Workspace Object

Workspace Properties Collection

This collection contains all persistent properties and collections that are attached to an object through a particular interface. The **Workspace** object exposes three separate **Properties** collections. These collections are exposed by:

- The **IWorkspace** interface (the default).
- The **IRepositoryObjectVersion** interface.
- The **INamedObject** interface.

Syntax

Set variable = object.**Properties(index)**

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. It receives the specified property.
object	An object expression; it evaluates to an object that exposes **IWorkspace**, **IRepositoryObjectVersion**, or **INamedObject** as the default interface.
index	An integer index that identifies which property in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**Properties**.**Count**.

Remarks

For more information about accessing a member of an interface that is not the

default interface, see Accessing Automation Object Members.

See Also

INamedObject Interface

RepositoryObjectVersion Object

ReposProperty Object

Workspace Object

Workspace SuccessorVersions Collection

This collection contains all **RepositoryObjectVersion** objects that are immediate predecessors of the current object version.

Syntax

Set variable = object.**SuccessorVersions**

The **SuccessorVersions** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **VersionCol** object. It receives a reference to the **Predecessor-Versions** collection.
object	An object expression that evaluates to a **Workspace** object.

Remarks

Each **Workspace** object has only one version (because you cannot invoke the **CreateVersion** method on a workspace). Thus, this collection is always null.

See Also

Relationship Object

RepositoryObjectVersion Object

Version Graph

VersionCol Object

Workspace Object

Workspace Workspaces Collection

This collection is always empty, because workspaces are not contained by other workspaces.

Remarks

This member is exposed by the **IRepositoryObjectVersion** interface, which is a nondefault interface for this object. Because of how interface inheritance works, this member is made available to the **Workspace** object by convention.

See Also

RepositoryObjectVersion Object

Workspace Object

VersionCol Object

A version collection is a collection of object versions. You can establish several different kinds of version collections.

When to Use

Use the **VersionCol** object to manage the contents of a workspace, to manage the target object versions of a versioned relationship, to navigate an object's version graph, or to manipulate all the versions of a particular object.

Properties

Property	Description
Count	The count of the number of object versions in the collection
Item	Retrieves the specified object version from the collection

Methods

Method	Description
Add	Adds an object version to the collection
Refresh	Refreshes the cached image of the collection
Remove	Removes an object version from the collection

See Also

Kinds of Version Collections

RelationshipCol Insert Method

RelationshipCol Move Method

VersionCol Count Property

This property is a long integer that contains the count of the number of items in the collection. This is a read-only property.

Syntax

object.**Count**

The **Count** property syntax has the following part.

Part	Description
object	The version collection

See Also

VersionCol Object

VersionCol Item Property

This property retrieves an item from the collection. This is a read-only property. There are three variations of this property.

Syntax

Set variable = object.**Item(***index***)**
Set variable = object.**Item(***objectId***)**
Set variable = object.**Item(***objectVersionId***)**

The **Item** property syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObjectVersion**, or as any object that implements the **IrepositoryObjectVersion** interface. It receives the item.
object	The version collection.
index	The index of the item to be retrieved from the collection.
objectId	The object identifier for the object version or workspace to be retrieved from the collection. You can supply an *objectID* only for the **Versions-of-Workspace** collection, the **Workspaces-of-Version** collection, or the **Checkouts-of-Workspace** collection.
objectVersionId	The object-version identifier for the item to be retrieved from the collection. You can supply an *objectVersionID* for any version collection.

See Also

Accessing Automation Object Members

Kinds of Version Collections

VersionCol Object

VersionCol Add Method

This method adds a new item to a relationship collection, when the sequencing of relationships in the collection is not important. The new relationship connects the *reposObj* object version to the source object version of the collection. The new relationship is passed back to the caller.

Syntax

Set variable = object.**Add(***reposVersion***)**

The **Add** method syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObjectVersion** object. It receives the object version that is added to the collection.
object	The version collection.
reposVersion	The **Repository** object version to be added to the collection.

Remarks

There are many different kinds of object-version collections. You can apply the **Add** method to some of them, but not to others. This method succeeds for:

- **TargetVersions** collections. You can use this method to enlarge the set of versions of a particular target object that are related to a particular source object.

- **Contents** collections. You can use this method to add an object version to the set of items contained in the workspace.

This method fails for:

- **Predecessors** collection. To enlarge an object version's set of

predecessors, use the **MergeVersion** method of the **RepositoryObjectVersion** object.

- **Successors** collection. To enlarge an object version's set of successors, use the **CreateVersion** method of the **RepositoryObjectVersion** object.

- **ObjectVersions** collection. To enlarge an object's set of versions, use the **CreateVersion** method of the **RepositoryObjectVersion** object.

- **Workspaces** collection. To enlarge the set of workspaces to which an item belongs, you add the object version to a workspace, rather than add a workspace to an object version. In other words, you use the **Add** method of the **VersionCol** object, but the version collection you are manipulating is the **Versions-of-Workspace** collection, not the **Workspaces-of-Version** collection.

- **Checkouts** collection. To check out another item to a workspace, use the **Checkout** method of the **Workspace** object.

See Also

Kinds of Version Collections

VersionCol Refresh Method

This method refreshes the cached image of the version collection. Only cached data that has not been changed by the current process is refreshed.

Syntax

Call object.**Refresh(***milliSecs***)**

The **Refresh** method syntax has the following parts.

Part	Description
object	The version collection.
milliSecs	This value is ignored.

See Also

VersionCol Object

VersionCol Remove Method

This method deletes an object version from a version collection.

Syntax

Call object.**Remove(***index***)**
Call object.**Remove(***objectId***)**
Call object.**Remove(***objectVersionId***)**

The **Remove** method syntax has the following parts.

Part	Description
object	The version collection.
index	The index of the item to be removed from the collection.
objectID	The object identifier for the object version or workspace to be removed from the collection. You can supply an *objectID* only for the **Versions-of-Workspace** collection, the **Workspaces-of-Version** collection, or the **Checkouts-of-Workspace** collection.
objectVersionID	The object-version identifier for the item to be removed from the collection. You can supply an *objectVersionID* for any version collection.

Remarks

There are many different kinds of object-version collections. You can apply this method to some of them, but not to others. The **Remove** method works for:

- **Target-Versions** collections. You can use this method to reduce the set of versions of a particular target object that are related to a particular source object.

- **Versions-of-Workspace** collections. You can use this method to

remove an object version to the set of items contained in the workspace.

This method fails for:

- **Predecessor-Versions** collections. To enlarge an object version's set of predecessors, use the **MergeVersion** method of the **RepositoryObjectVersion** object.

- **Successor-Versions** collections. To enlarge an object version's set of successors, use the **CreateVersion** method of the **RepositoryObjectVersion** object.

- **Versions-of-Object** collections. To enlarge an object's set of versions, use the **CreateVersion** method of the **RepositoryObjectVersion** object.

- **Workspaces-of-Version** collections. To remove a workspace from the set of workspaces in which an object version is present, you must explicitly remove the object version from that workspace's **Versions-of-Workspace** collection.

- **Checkouts-of-Workspace** collections. To reduce the number of items checked out to a workspace, use the **Checkin** method of the **Workspace** object.

See Also

Kinds of Version Collections

VersionCol Object

VersionedRelationship Object

A relationship connects two repository objects in a repository database. A relationship has an origin object, a destination object, and a set of properties. Each relationship conforms to a particular relationship type. You can version a relationship using this object.

When to Use

Use the **VersionedRelationship** object to manipulate the properties of a versioned relationship, to delete a versioned relationship, or to refresh the cached image of a versioned relationship.

Properties

Property	Description
Destination	The destination object of the relationship
Interface	The specified object interface
Name	The name of the relationship's destination object
Origin	The origin object of the relationship
Repository	The open repository instance through which this relationship was instantiated
Source	The source object of the relationship
Target	The target object of the relationship
Type	The type of the relationship

Methods

Method	Description
Delete	Deletes a relationship
Lock	Locks the relationship
Pin	Establishes a particular item in the **TargetVersions** collection as the pinned target version of the

	relationship
Unpin	Ensures that no item in the **TargetVersions** collection is pinned

Collections

Collection	Description
Properties	The collection of all of the properties that are attached to the relationship
TargetVersions	The collection of all versions of the target object that are related to the source object version of the relationship

See Also

Relationship Object

Versioning Objects

VersionedRelationship Destination Property

This property is the destination object of the current version of the relationship. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRelationship** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Destination**

The **Destination** property syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObjectVersion**. It receives the destination object for the relationship.
object	An object expression that evaluates to a **VersionedRelationship** object.

Remarks

If the object is a destination versioned relationship, this property is equivalent to the **Source** property. If the object is an origin versioned relationship, this property is equivalent to the **Target** property.

See Also

IRelationship Interface

Relationship Object

VersionedRelationship Object

VersionedRelationship Source Property

VersionedRelationship Target Property

VersionedRelationship Interface Property

This property obtains a view of the **VersionedRelationship** object that uses an alternate interface as the default interface. This is a read-only property. There are three variations of this property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Interface(***interfaceId***)**
Set variable = object.**Interface(***objectId***)**
Set variable = object.**Interface(***interfaceName***)**

The **Interface** property syntax has the following parts.

Part	Description
variable	An object variable. It receives the relationship object with the specified interface as the default interface.
object	An object expression that evaluates to a **VersionedRelationship** object.
interfaceId	The interface identifier for the interface to be retrieved.
objected	The object identifier for the interface definition to which the interface to be retrieved conforms.
interfaceName	A string containing the name of the interface to be retrieved.

Remarks

Because the **VersionedRelationship** class implements a limited set of interfaces, the input parameter you supply must specify one of the following interfaces:

IVersionedRelationship, **IRelationship**, **IRepositoryItem**, **IRepositoryDispatch**, or **IAnnotationalProperties**.

See Also

IAnnotationalProps Interface

IRelationship Interface

IRepositoryDispatch Interface

IRepositoryItem Interface

IVersionedRelationship Interface

Relationship Object

VersionedRelationship Object

VersionedRelationship Name Property

This property is a character string that contains the name that the relationship assigns to the destination object.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**Name**

The **Name** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **VersionedRelationship** object

Remarks

A relationship's name is the name by which the origin object version refers to every destination object version in its **TargetVersions** collection. When you access or set the **Name** property for an origin versioned relationship, it is this name that the repository engine retrieves or sets for you.

When you access or set the **Name** property for a destination versioned relationship, the repository engine takes a different action. For more information, see Changing a Destination Relationship's Name.

See Also

INamedObject Interface

IRepositoryItem Interface

VersionedRelationship Origin Property

This property is the origin object of the current version of the relationship. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRelationship** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Origin**

The **Origin** property syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObjectVersion** object. It receives the origin object version for the relationship.
object	An object expression that evaluates to a **VersionedRelationship** object.

Remarks

If the object is an origin versioned relationship, this property is equivalent to the **VersionedRelationship Source** property. If the object is a destination versioned relationship, this property is equivalent to the **VersionedRelationship Target** property.

See Also

IRelationship Interface

Relationship Object

VersionedRelationship Object

VersionedRelationship Repository Property

This property is the open repository instance or workspace through which this relationship was instantiated. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Repository**

The **Repository** property syntax has the following parts.

Part	Description
variable	A variable declared as a reference to any object implementing the **IRepository** interface. It receives the object that represents the open repository instance or the workspace.
object	An object expression that evaluates to a **VersionedRelationship** object.

Remarks

This method returns a reference to either a **Repository** object or a **Workspace** object. If it returns a **Workspace** object, you are manipulating the item within the context of that workspace. If it returns a **Repository** object, you are manipulating the item not within the context of a workspace, but within the context of a shared **Repository** instance.

See Also

IRepositoryItem Interface

VersionedRelationship Source Property

This property is the source object of the current version of the relationship. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRelationship** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Source**

The **Source** property syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObjectVersion**. It receives the source object version for the relationship.
object	An object expression that evaluates to a **VersionedRelationship** object.

See Also

IRelationship Interface

Relationship Object

VersionedRelationship Object

VersionedRelationship Target Property

This property is the target object of the current version of the relationship. This is a read-only property.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRelationship** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Target**

The **Target** property syntax has the following parts.

Part	Description
variable	A variable declared as a **RepositoryObjectVersion**. It receives the target object version for the relationship.
object	An object expression that evaluates to a **VersionedRelationship** object.

Remarks

A versioned relationship can have a **TargetVersions** collection containing the set of object versions related (through the versioned relationship) to the source object version. The repository engine follows a resolution strategy to select a specific object version to return.

See Also

IRelationship Interface

Relationship Object

Resolution Strategy for Objects and Object Versions

VersionedRelationship Object

VersionedRelationship Type Property

This property specifies the type of the versioned relationship. More specifically, it is the object identifier of the relationship definition object for the versioned relationship. **Type** is a read-only property. To copy this property to another variable, use a variable that is declared as a **Variant**.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**Type**

The **Type** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **VersionedRelationship** object

See Also

IRepositoryItem Interface

Relationship Object

VersionedRelationship Object

VersionedRelationship Delete Method

This method deletes a relationship from its relationship collection.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**Delete**

The **Delete** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **VersionedRelationship** object

Remarks

If the item to be deleted is an origin versioned relationship, this method fails unless the source object version satisfies the basic requirements for changing an object version. Assuming the source object version can be changed, the repository engine deletes the entire relationship (rather than merely removing one item from the **TargetVersions** collection of the relationship). In other words, after this method finishes, no version of the destination object remains related to the origin object version. Furthermore, if the relationship is a delete-propagating relationship, the repository engine considers performing one or more propagated deletions.

If the item to be deleted is a destination versioned relationship, the repository engine follows a resolution strategy to yield a single origin object version from the **TargetVersions** collection of the relationship. If that origin object version cannot be changed (that is, if it does not satisfy the requirements for changing an object version), this method fails. Assuming that the origin object version can be

changed, the repository engine removes it from the **TargetVersions** collection of the relationship. Furthermore, if the relationship is a delete-propagating relationship, the repository engine considers performing one or more propagated deletions.

For more information, see Propagating Deletes.

See Also

IRepositoryItem Interface

Relationship Object

Requirements for Changing an Object-Version

Resolution Strategy for Objects and Object Versions

VersionedRelationship Object

VersionedRelationship Lock Method

This method locks the versioned relationship. Locking the relationship prevents other processes from locking the relationship while you are working with it. The lock is released when you end the current transaction.

This method is not attached to the default interface for the repository Automation object; it is attached to the **IRepositoryItem** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Call object.**Lock**

The **Lock** method syntax has the following part.

Part	Description
object	An object expression that evaluates to a **VersionedRelationship** object

See Also

IRepositoryItem Interface

Relationship Object

VersionedRelationship Object

VersionedRelationship Pin Method

This method marks a particular version of the target object as the pinned version.

Syntax

Call object.**Pin(***objectVersion***)**

The **Pin** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **VersionedRelationship** object
objectVersion	The **Repository** object version to be pinned

Remarks

The *objectVersion* you supply must be a member of the versioned relationship's **TargetVersions** collection.

Remember that no versioned relationship can have more than one pinned target object version. If the relationship already has a pinned target object version, it becomes unpinned when you call this method, and the *objectVersion* you supply becomes the pinned target object version.

If the relationship is a destination relationship, the **Pin** method fails.

If the origin of the relationship is unchangeable, the **Pin** method fails.

See Also

Relationship Object

VersionedRelationship Object

VersionedRelationship Unpin Method

This method ensures that the versioned relationship has no pinned version.

Syntax

Call object.**Unpin**

The **Unpin** method syntax has the following part.

Part	Description
object	An object expression that evaluates to **a Versioned Relationship** object

Remarks

If the relationship is a destination relationship, the **Unpin** method fails.

If the origin of the relationship is unchangeable, the **Pin** method fails.

See Also

Relationship Object

VersionedRelationship Object

VersionedRelationship Properties Collection

This collection contains all of the stored properties and collections that are attached to an object through a particular interface. The **VersionedRelationship** object exposes two separate properties collections. These collections are exposed by:

- The **IVersionedRelationship** interface (the default).
- The **IAnnotationalProps** interface.

Syntax

Set variable = object.**Properties(***index***)**

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. It receives the specified property.
object	An object expression. It evaluates to an object that exposes **IVersionedRelationship** or **IAnnotationalProps** as the default interface.
index	An integer index that identifies which property in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**Properties**.**Count**.

Remarks

For more information about how to access a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

Relationship Object

ReposProperty Object

VersionedRelationship Object

VersionedRelationship TargetVersions Collection

This collection contains all of the **RepositoryObjectVersion** objects that are related to the source object version through the versioned relationship.

Syntax

Set variable = object.**TargetVersions**

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **VersionCol** object. It receives a reference to the **TargetVersions** collection.
object	An object expression that evaluates to a **VersionedRelationship** object.

See Also

Relationship Object

ReposProperty Object

VersionedRelationship Object

Meta Data Services Programming

RTIM Automation Objects

This section introduces the Repository Type Information Model (RTIM) objects, which are used to create or extend information models.

These objects work with the repository engine automation objects that are used to drive the repository engine. The repository engine objects are listed separately. For more information, see Repository Engine Automation Objects.

The following table lists RTIM Automation objects in alphabetical order.

Object	Description
Alias Object	Defines a derived property that is based on another property without changing the meaning of underlying property
ClassDef Object	Adds interfaces to a class
CollectionDef Object	Defines how instances of a particular type of collection will behave
EnumerationDef Object	Represents an association of enumerated values
EnumerationValueDef Object	Represents a single member of an enumeration value set
InterfaceDef Object	Defines an interface object, including its properties and members
MethodDef Object	Defines a method object
ParameterDef Object	Defines a parameter object
PropertyDef Object	Defines a property object
RelationshipDef Object	Defines a relationship object
ReposRoot Object	Defines the starting point in a repository for both type information and object instance data navigation
ReposTypeLib Object	Defines an information model in a repository database

ScriptDef Object	Represents a Microsoft® ActiveX® script that you can associate with a method or property definition

See Also

Automation Reference

Information Models

Repository API Reference

Repository Engine

Repository Object Architecture

Alias Object

An **Alias** object supports member delegation of property definitions. You can use the **Alias** object to define a derived property that is based on another property without changing the meaning of underlying property.

An **Alias** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. In addition to the members described here, you can access members that are defined for those objects as well as members of **IReposTypeInfo**. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use the **Alias** object to rename an existing property.

Properties

Property	Description
Name	Stores the name of the alias.
MemberSynonym	Stores a synonym of an alias name. This property is optional.

Collections

Collection	Description
ServicedByMember	Identifies the base property to which an alias name is mapped.

See Also

Member Delegation

RepositoryObject Object

RepositoryObjectVersion Object

Alias Name Property

This property is a string that stores the alias name of a property.

Syntax

Object.**Name**=*string*

The **Name** property syntax has the following parts.

Part	Description
object	An **Alias** object
string	A variable length string that can be a maximum of 200 characters in length

Remarks

The alias name that you provide is an alternate name of a property that is identified in the **ServicedByMember** collection.

See Also

Alias Object

Alias ServicedByMember Collection

Alias MemberSynonym Property

This property is a string used as a synonym for an alias name. It applies to **MethodDef**, **PropertyDef**, and **CollectionDef** objects. The value that you specify must be unique.

Syntax

Object.**MemberSynonym**=*string*

The **MemberSynonym** property syntax has the following parts.

Part	Description
object	An **Alias** object
string	A variable length string that can be a maximum of 255 characters in length

See Also

Alias Object

Alias Name Property

Alias ServicedByMember Collection

This collection contains the base property for which you are creating an alias. Each alias can only have one item in its **ServicedByMember** collection.

Syntax

Set variable=object.**ServicedByMember**(*index*)

The **ServicedByMember** syntax has the following parts.

Part	Description
variable	Variable declared as an object.
object	An **Alias** object.
index	An integer index that identifies the member in the collection to be addressed. The valid range is from one. For more information, see Selecting Items in a Collection.

See Also

Alias Object

Alias Name Property

ClassDef Object

The **ClassDef** object helps you create information models by adding interfaces to a class. To insert a new class definition into an information model, use the **ReposTypeLib** object.

Once you have added all of the interfaces, you complete a class definition by committing the transaction that brackets your class definition modifications.

A **ClassDef** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. You can also access members that are defined for those objects and members of **IViewClassDef** and **IVersionAdminInfo**. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use the **ClassDef** object to:

- Add a new or existing interface to a class definition.
- Retrieve the global identifier for the class.
- Access the collection of interfaces that are part of a class definition.

Properties

Property	Description
ClassID	The global identifier of the class
Name	The name of a **ClassDef** object
Synonym	A synonym of the name of the **ClassDef** object

Methods

Method	Description
AddInterface	Adds an existing interface to the class definition
CreateInterfaceDef	Creates a new interface and adds it to the class definition
ObjectInstances	Materializes an object collection containing all of the objects in the repository that conform to this class

Collections

Collection	Description
Interfaces	The collection of all interfaces that are implemented by the class.
ItemInCollections	This collection is empty for class definitions. It is reserved for future use.
Properties	The collection of all persistent properties that are attached to the **ClassDef** object.
ReposTypeLibScopes	The collection of all repository type libraries that contain this class.
ScriptsUsedByClass	The collection of all **ScriptDef** objects that are implemented by this class.

See Also

RepositoryObject Object

RepositoryObjectVersion Object

ReposTypeLib Object

ScriptDef Object

ClassDef ClassID Property

This property contains the global identifier (ClsID) that is assigned to this class. If you copy this property to a variable, declare the variable as a **Variant**.

Syntax

object.**ClassID**

The **ClassID** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **ClassDef** object

See Also

ClassDef Object

Object Identifiers and Internal Identifiers

ClassDef Name Property

This property stores the name of the **ClassDef** object. The **Name** property is made available through the **INamedObject** interface. To use the **Name** property, the class definition object that you create must implement **INamedObject**.

Syntax

Object.**Name**=*string*

The **Name** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **ClassDef** object
string	A variable length string that can be a maximum of 200 characters in length

See Also

ClassDef Object

INamedObject Interface

ClassDef Synonym Property

This property stores a synonym of the name of the **ClassDef** object. The **Synonym** property is made available through the **IReposTypeInfo2** interface. To use the **Synonym** property, the class definition object that you create must implement **IReposTypeInfo2**.

Syntax

Object.**Synonym**=*string*

The **Synonym** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **ClassDef** object.
string	A variable length string that can be a maximum of 200 characters in length. Synonym values must be unique for **ClassDef** objects.

See Also

ClassDef Name Property

ClassDef Object

IReposTypeInfo2 Interface

ClassDef AddInterface Method

The **AddInterface** method adds an existing interface to the collection of interfaces that are implemented by a particular class.

Syntax

Call object.**AddInterface(***interfaceDef*, *flag***)**

The **AddInterface** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **ClassDef** object.
interfaceDef	The **InterfaceDef** definition object for the interface that is to be added to the collection of interfaces that are implemented by this class.
flag	Determines whether the interface is the default interface. If the interface that you are adding is the default interface, pass in the string "Default". Otherwise, pass in a null string.

Remarks

When you indicate that an interface is the default interface for a class, you are actually setting the value of the **ImplementsOptions** annotational property on the **Class_Implements_Interface** relationship to TRUE.

See Also

Accessing Automation Object Members

ClassDef Object

ClassDef CreateInterfaceDef Method

IAnnotationalProps Interface

InterfaceDef Object

ClassDef CreateInterfaceDef Method

The **CreateInterfaceDef** method creates a new interface definition and adds the interface to the collection of interfaces that are implemented by the class.

Syntax

Set variable = object.**CreateInterfaceDef(***sObjId, name, interfaceId, [ancestor], [flag]***)**

The **CreateInterfaceDef** method syntax has the following parts.

Part	Description
variable	A variable declared as an **InterfaceDef** object. It receives the new interface definition.
object	An object expression that evaluates to a **ClassDef** object
sObjId	The object identifier to be assigned to the new interface definition object. If this parameter is set to OBJID_NULL, the repository engine assigns an object identifier for you.
name	The name of the interface that is to be created.
interfaceId	The interface identifier for this interface. If there is none, set this parameter to zero.
ancestor	The **InterfaceDef** definition object for the interface that is the base interface from which the interface being created is derived. This parameter is optional.
flag	Determines whether the interface is the default interface. If the interface that you are adding is the default interface, pass in the string "Default". Otherwise, pass in a null string. This parameter is optional.

Remarks

When you indicate that an interface is the default interface for a class, you are actually setting the value of the **ImplementsOptions** annotational property on

the *Class Implements Interface* relationship to TRUE.

See Also

Accessing Automation Object Members

ClassDef Object

IAnnotationalProps Interface

InterfaceDef Object

ClassDef ObjectInstances Method

This method materializes an object collection containing all of the objects in a repository that conform to this class.

Syntax

Set variable = object.**ObjectInstances**

The **ObjectInstances** method syntax has the following parts.

Part	Description
variable	A variable declared as an **ObjectCol** object. It receives the collection of objects that conform to this class.
object	An object expression that evaluates to a **ClassDef** object.

Remarks

The collection contains one version of each object that conforms to the class. For each such object, the repository engine uses criteria to select which version to include in the collection. For more information, see Resolution Strategy for Objects and Object Versions.

ObjectInstances is not scoped to a workspace. All information models in a repository are included in the scope.

See Also

ClassDef Object

ObjectCol Object

ClassDef Interfaces Collection

The **Interfaces** collection contains all interfaces that are implemented by this class.

Syntax

Set variable = object.**Interfaces(***index***)**

The **Interfaces** collection syntax has the following parts.

Part	Description
variable	A variable declared as an **InterfaceDef** object. It receives the specified interface.
object	An object expression that evaluates to a **ClassDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by the object **Interfaces.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Class_Implements_Interface**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	Yes	The source object for the

		collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-sensitive Names	Not Applicable	Case-sensitive naming is not applicable for this collection.
Unique Names	Not Applicable	Unique naming is not applicable for this collection.

See Also

ClassDef Object

InterfaceDef Object

ClassDef Properties Collection

A **Properties** collection contains all of the persistent properties and collections that are attached to a **ClassDef** object through a particular interface. The **ClassDef** object exposes four separate **Properties** collections. These collections are exposed by:

- The **IClassDef2** interface (the default) or **IClassDef** interface.
- The **IReposTypeInfo** or **IReposTypeInfo2** interface.
- The **IRepositoryObject** or **IRepositoryObject2** interface.
- The **IAnnotationalProps** interface.

Syntax

Set variable = object.**Properties(***index***)**

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. It receives the specified property.
object	An object expression; it evaluates to an object that exposes **IClassDef** or **IClassDef2**, **IReposTypeInfo** or **IReposTypeInfo2**, **IRepositoryObject** or **IRepositoryObject2**, or **IAnnotationalProps** as the default interface.
index	An integer index that identifies which property in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by

	the object **Properties.Count**. For more information, see Selecting Items in a Collection.

Remarks

Additional steps are required for accessing members that are not part of the default interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

ClassDef Object

IAnnotationalProps Interface

ReposProperty Object

ClassDef ReposTypeLibScopes Collection

This is the collection of repository type libraries that contain this definition.

Syntax

Set variable = object.**ReposTypeLibScopes(***index***)**

The **ReposTypeLibScopes** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposTypeLib** object. It receives the specified repository type library object.
object	An object expression that evaluates to a **ClassDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by the object **TypeLibScopes.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**ReposTypeLib_ScopeFor_ReposTypeInfo**	This is the type of relationship by which all items of the collection are connected to a common

		source object.
Source is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	The deletion of an origin object or relationship in the collection causes the

		deletion of the corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-sensitive Names	No	The relationship type for the collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects. This applies to collections whose

		relationship type permits destination objects to be named.

See Also

ClassDef Object

Naming and Unique-Naming Collections

ReposTypeLib Object

ClassDef ScriptsUsedByClass Collection

This is the collection of **ScriptDef** objects that are implemented by this class.

Syntax

Set variable = object.**ScriptsUsedByClass(***index***)**

The **ScriptsUsedByClass** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ScriptDef** object. Receives the specified script definition.
object	An object expression that evaluates to a **ClassDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by the object **ScriptsUsedByClass.Count**. For more information, see Selecting Items in a Collection.

See Also

ClassDef Object

ReposTypeLib Object

ScriptDef Object

CollectionDef Object

A collection type (or collection definition) defines how instances of a particular type of collection will behave. The properties of the collection type determine:

- The minimum and maximum number of items in a collection.
- Whether the collection type is an origin collection type.
- Whether the collection type permits the naming of destination objects, and if so, whether those names are case-sensitive, and required to be unique.
- Whether the collection type permits the explicit sequencing of items in the collection.
- What happens to related objects when objects or relationships in the collection are deleted.
- Whether origin collections of this type are automatically copied to new object versions by the **CreateVersion** method.
- Whether the **MergeVersion** method combines origin collections of this type as a whole, or item-by-item.
- Whether the **FreezeVersion** method requires that destination object versions of relationships of this type be frozen before the attendant origin object versions can be frozen.

The kind of relationship that a particular collection type uses to relate objects to each other is determined by its **CollectionItem** collection. The **CollectionItem**

collection associates a single relationship type to the collection type. To add a new collection type, use the **InterfaceDef** object.

A **CollectionDef** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. In addition to the members described here, you can access members that are defined for those objects as well as members of **IInterfaceMember2** and **IVersionAdminInfo**. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use the **CollectionDef** object to retrieve or modify the properties of a collection type, to determine the kind of relationship that the collection implements, or to determine the interface to which the collection is attached.

Properties

Property	Description
DispatchID	The dispatch identifier to use when accessing an instance of this type of collection
Flags	Flags that specify details about this collection definition
IsOrigin	Indicates whether collections of this type are origin collections
MemberSynonym	Stores a synonym of the collection name
MaxCount	The maximum number of target objects that can be contained in a collection of this type
MinCount	The minimum number of target objects that must be contained in a collection of this type
Name	Stores the name of a collection

Collections

Collection	Description

CollectionItem	The collection of one relationship type that defines the relationship between target objects of this type of collection and a single source object
Interface	The interface to which this collection definition is attached
Properties	The collection of all persistent properties that are attached to the **CollectionDef** object

See Also

IInterfaceMember2 Interface

InterfaceDef Object

RepositoryObject Object

RepositoryObjectVersion Object

CollectionDef DispatchID Property

This property contains the dispatch identifier to use when accessing a collection of this type.

This property is not attached to the default interface for the **CollectionDef** Automation object; it is attached to the **IInterfaceMember** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**DispatchID**

The **DispatchID** property syntax has the following part.

Part	Description
object	An object expression; evaluates to an object that exposes **IInterfaceMember** as the default interface

See Also

CollectionDef Object

IInterfaceMember Interface

CollectionDef Flags Property

The **CollectionDef** object exposes two separate **Flags** properties. One of these properties is exposed by the default interface **ICollectionDef**, and the other is exposed by the **IInterfaceMember** interface. The **Flags** property of both interfaces is described here.

The default **ICollectionDef Flags** property determines:

- Whether the collection type permits the naming of destination objects, and if so, whether those names are case-sensitive, and required to be unique.
- Whether the collection type permits the explicit sequencing of items in the collection.
- What happens to related objects when objects or relationships in the collection are deleted.
- Whether origin collections of this type are automatically copied to new object versions by the **CreateVersion** method.
- Whether the **MergeVersion** method combines origin collections of this type as a whole, or item by item.
- Whether the **FreezeVersion** method requires that destination object versions of relationships of this type be frozen before the attendant origin object versions can be frozen.

The **IInterfaceDef Flags** property is a flag that specifies whether the interface member should be visible to Automation queries. For more information about flag values and their specific purposes, see InterfaceMemberFlags Enumeration.

Syntax

object.**Flags**=(*integer*)

The **Flags** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **CollectionDef** object, for the default **Flags** property. -or- An object expression that evaluates to an object that exposes **IInterfaceMember** as the default interface, for the alternate **Flags** property.
integer	Flag values are bit flags, and may be combined to set multiple options. For more information about flag values and their specific purposes, see CollectionDefFlags Enumeration.

Remarks

For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

CollectionDef Object

IInterfaceMember Interface

CollectionDef IsOrigin Property

This property indicates whether collections of this type are origin collections. If you copy this property to a variable, declare the variable as a Boolean.

Syntax

object.**IsOrigin**

The **IsOrigin** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **CollectionDef** object

See Also

CollectionDef Object

CollectionDef MaxCount Property

This property specifies the maximum number of target objects that a collection of this type can contain. This property is maintained for informational purposes, and is not enforced by the repository engine.

Syntax

object.**MaxCount**

The **MaxCount** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **CollectionDef** object

See Also

CollectionDef Object

CollectionDef MemberSynonym Property

This property is a string used as a synonym for a collection name. The value that you specify must be unique.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IInterfaceMember2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**MemberSynonym**=(*string*)

The **MemberSynonym** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **CollectionDef** object
string	A variable length string that can be a maximum of 255 characters in length

See Also

CollectionDef Object

IInterfaceMember2 Interface

CollectionDef MinCount Property

This property specifies the minimum number of target objects that a collection of this type can contain. This property is maintained for informational purposes, and is not enforced by the repository engine.

Syntax

object.**MinCount**

The **MinCount** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **CollectionDef** object

See Also

CollectionDef Object

CollectionDef Name Property

This property is a string that stores the name of a collection.

Syntax

object.**Name**=(*string*)

The **Name** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **CollectionDef** object
string	A variable length string that can be a maximum of 200 characters in length

See Also

CollectionDef Object

CollectionDef CollectionItem Collection

Every **RelationshipDef** object has two **CollectionDef** objects. You can navigate a relationship definition instance from either of two directions. That is, from a **RelationshipDef** object, you can navigate to its collection of **CollectionDef** objects. Conversely, you can navigate from a **CollectionDef** object to the associated **RelationshipDef** object. To do this, use the **CollectionItem** collection on the **ICollectionDef** interface. For more information about collections and relationships, see Repository Object Architecture.

Syntax

Set variable = object.**CollectionItem(***index***)**

The **CollectionItem** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **RelationshipDef** object. It receives the specified relationship definition object.
object	An object expression that evaluates to a **CollectionDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**CollectionItem.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Collection_Contains_Items**	This is the type of

		relationship by which all items of the collection are connected to a common source object.
Source Is Origin	Yes	The source object for the collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	One	The maximum number of items that can be contained in the collection is one.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-sensitive Names	Not applicable	Case-sensitive naming is not applicable for this collection.
Unique Names	Not applicable	Unique naming is not applicable for this

		collection.

See Also

CollectionDef Object

RelationshipDef ItemInCollections Collection

RelationshipDef Object

CollectionDef Interface Collection

For a particular collection definition, the interface collection specifies which interface exposes a member of the collection type.

This collection is not attached to the default interface for the **CollectionDef** Automation object; it is attached to the **IInterfaceMember** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Interface(***index***)**

The **Interface** collection syntax has the following parts.

Part	Description
variable	A variable declared as an **InterfaceDef** object. Receives the specified interface definition.
object	An object expression; evaluates to an object that implements **IInterfaceMember** as the default interface.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**Interface.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Interface_Has_Members**	This is the type of

		relationship by which all items of the collection are connected to a common source object.
Source Is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum Collection Size	One	The maximum number of items that can be contained in the collection is one.
Sequenced Collection	Yes	As a destination collection, this collection permits an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	Deleting an origin object or a relationship in the collection causes the deletion of a corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-sensitive Names	No	The relationship type for the collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for

		the collection requires that the name of a destination object be unique within the collection of destination objects. This applies to collections whose relationship type permits destination objects to be named.

See Also

CollectionDef Object

InterfaceDef Object

CollectionDef Properties Collection

A **Properties** collection contains all of the persistent properties and collections that are attached to an object through a particular interface. The **CollectionDef** object exposes four separate **Properties** collections. These collections are exposed by:

- The **ICollectionDef** interface (the default).
- The **IInterfaceMember** or **IInterfaceMember2** interface.
- The **IRepositoryObject** or **IRepositoryObject2** interface.
- The **IAnnotationalProps** interface.

Syntax

Set variable = object.**Properties(***index***)**

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. Receives the specified property.
object	An object expression; evaluates to an object that exposes **ICollectionDef**, **IInterfaceMember** or **IInterfaceMember2**, **IRepositoryObject** or **IRepositoryObject2**, or **IAnnotationalProps** as the default interface.
index	An integer index that identifies which property in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by

	object.**Properties.Count.** For more information, see Selecting Items in a Collection.

Remarks

For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

CollectionDef Object

ReposProperty Object

EnumerationDef Object

An enumeration definition object represents an association of enumerated values. The enumerated values that you provide to an enumeration definition are defined through a series of **EnumerationValueDef** objects. The enumeration definition itself can be associated with a **PropertyDef** object.

You can combine enumeration definition objects in collections. Collections allow you to limit the number or filter the range of values that appear to the end user. You can also allow a property definition object to reference an enumeration definition object that is defined in another information model.

Note The repository engine does not restrict objects to the enumeration values associated with a property. Specifying a value that is not in the enumeration value set does not produce an error.

An **EnumerationDef** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. You can also access members that are defined for these objects and members of **IRepositoryObjectStorage**, **IReposTypeInfo2**, and **IVersionAdminInfo2**. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use an **EnumerationDef** object to define a property that uses enumerated values comprised of a fixed set of constant string or integer values.

Properties

Property	Description
Name	Contains the name of the **EnumerationDef** object. The name must be unique within the information model.
Description	Contains a description of the enumeration.
IsFlag	Indicates that the enumeration defines a logical flag. The selected enumeration values should be combined

	logically using OR. This only applies to numeric enumeration values.

Collections

Collection	Description
Values	Collection of **EnumerationValueDef** objects

See Also

EnumerationValueDef Object

Filtering Collections

IRepositoryObjectStorage Interface

IVersionAdminInfo2 Interface

PropertyDef Object

RepositoryObject Object

RepositoryObjectVersion Object

EnumerationDef Name Property

This property stores the name of the **EnumerationDef** object.

Syntax

Object.**Name**=*string*

The **Name** property syntax has the following parts.

Part	Description
object	The **EnumerationDef** object.
string	A variable length string that can be a maximum of 255 characters in length. The name must be unique within the information model.

See Also

EnumerationDef Object

EnumerationValueDef Object

EnumerationDef Description Property

This property stores a description of the **EnumerationDef object** for documentation purposes. This property is not processed by the repository engine.

Syntax

Object.**Description**=*string*

The **Description** property syntax has the following parts.

Part	Description
object	The **EnumerationDef** object
string	A variable length string that can be a maximum of 255 characters in length

See Also

EnumerationDef Object

EnumerationValueDef Object

EnumerationDef IsFlag Property

This Boolean property indicates whether the enumeration definition object defines a logical flag. The selected enumeration values should be combined logically using OR. This only applies to numeric enumeration values.

Syntax

Object.**IsFlag**

The **IsFlag** property syntax has the following part.

Part	Description
object	The **EnumerationDef** object

Remarks

If you need an object to represent a flag structure, and you want that flag to support a series of bit flags that can be combined to set multiple options, you can create an **EnumerationDef** object and set the **IsFlag** property to **True**.

See Also

EnumerationDef Object

EnumerationValueDef Object

EnumerationDef Values Collection

This collection contains **EnumerationValuesDef** objects.

Syntax

Set variable=object.**Values**(*index*)

The **Values** syntax has the following parts.

Part	Description
variable	Variable declared as an object.
object	An **EnumerationDef** object.
index	An integer index that identifies which member in the collection is to be addressed. The valid range is from one to the total number of members in the collection. For more information, see Selecting Items in a Collection.

See Also

Filtering Collections

EnumerationDef Object

EnumerationValueDef Object

EnumerationValueDef Object

An enumeration value definition object defines a single member of an enumeration value set. An **EnumeratedValueDef** object is owned by an **EnumerationDef** object. You can define multiple **EnumerationValueDef** objects to create an array of values for a property definition that uses enumerated values.

An **EnumerationValueDef** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. You can also access members that are defined for these objects and members of **IRepositoryObjectStorage** and **IVersionAdminInfo2**. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use an **EnumerationValueDef** object to associate real-world, constant data values with a property definition.

Properties

Property	Description
EnumerationValueDef EnumValue Property	A string value included in an enumerated set of values for a specified property definition object

See Also

EnumerationDef Object

IRepositoryObjectStorage Interface

IVersionAdminInfo2 Interface

RepositoryObject Object

RepositoryObjectVersion Object

EnumerationValueDef EnumValue Property

This property is a string containing a value that may be stored as the property value of an object.

Syntax

Object.**EnumValue**=*string*

The **EnumValue** property syntax has the following parts.

Part	Description
object	The **EnumerationValueDef** object.
string	A variable length string that can be a maximum of 255 characters in length. This value can be numeric. If you are using the **IsFlag** property of an **EnumerationDef** object to create a series of bit flags, this value must be numeric.

See Also

EnumerationValueDef Object

InterfaceDef Object

The properties, methods, and collections that a class implements are organized into functionally related groups. Each group is implemented as a repository interface. The properties, methods, and collections of each interface are members of the interface. An interface definition is the template to which an interface conforms.

To add a new interface to a repository, use the **ClassDef** object or the **ReposTypeLib** object.

An **InterfaceDef** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. In addition to the members described here, you can access members that are defined for those objects. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use the **InterfaceDef** class to:

- Retrieve or modify properties of an interface definition.
- Determine which members are attached to an interface definition.
- Determine which classes implement an interface.
- Determine the base interface from which an interface derives.
- Determine which interfaces derive from a particular interface.
- Determine which repository objects expose a particular interface.

- Add a new property, method, or collection type to an interface definition.

Properties

Property	Description
Flags	Flags that specify whether the interface is extensible, and whether the interface should be visible to Automation interface queries
InterfaceID	The global interface identifier for the interface
Synonym	Stores a synonym of the interface name
TableName	The name of the SQL table that is used to store instance information for the properties of the interface

Methods

Method	Description
CreateAlias	Creates a new alias definition, and attaches it to the interface definition.
CreateMethodDef	Creates a new method definition, and attaches it to the interface definition.
CreatePropertyDef	Creates a new property definition, and attaches it to the interface definition.
CreateRelationshipColDef	Creates a relationship collection type. The collection type is attached to the interface definition.
ObjectInstances	Materializes an **ObjectCol** collection of all objects in the repository that expose this interface.

Collections

Collection	Description
Ancestor	The collection of one base interface from which this interface derives
Classes	The collection of classes that implement the interface
Descendants	The collection of other interfaces that derive from this interface
Members	The collection of members that are exposed by the interface
Properties	The collection of all persistent properties that are attached to the **InterfaceDef** object
ReposTypeLibScopes	The collection of all repository type libraries that contain this definition
Implies	The collection of **InterfaceDef** objects that are also implemented by this interface
ImpliedBy	The collection of **InterfaceDef** objects that also implement this interface
ScriptsUsedByInterface	The collection of script definition object used by this interface

See Also

ClassDef Object

InterfaceDef Object

PropertyDef Object

RepositoryObject Object

RepositoryObjectVersion Object

ReposTypeLib Object

InterfaceDef Flags Property

This property contains flags that specify whether the interface is extensible, and whether the interface should be visible to Automation interface queries.

Syntax

object.**Flags**=*(integer)*

The **Flags** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to an **InterfaceDef** object.
Integer	A byte that stores a flag value (either 1, 2, or 3). For more information about flag values, see InterfaceDefFlags Enumeration.

See Also

InterfaceDef Object

InterfaceDef InterfaceID Property

This property is the global interface identifier for the interface. If you copy this property to a variable, declare the variable as a **Variant**.

Syntax

object.**InterfaceID**

The **InterfaceID** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to an **InterfaceDef** object

See Also

InterfaceDef Object

Object Identifiers and Internal Identifiers

InterfaceDef Synonym Property

This property is a string used as a synonym for an interface name. The value that you specify must be unique.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IReposTypeInfo2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**Synonym**=*string*

The **Synonym** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to an **InterfaceDef** object
string	A variable length string that can be a maximum of 255 characters in length

See Also

InterfaceDef Object

IReposTypeInfo2 Interface

InterfaceDef TableName Property

This character string property contains the name of the SQL table that is used to store instance information for the properties of the interface.

Syntax

object.**TableName**=*(string)*

The **TableName** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to an **InterfaceDef** object
string	A variable length string that can be a maximum of 30 characters

See Also

InterfaceDef Object

Repository SQL Schema

InterfaceDef CreateAlias Method

This method creates a new alias and attaches it to the interface definition.

Syntax

Set variable = object.**CreateAlias(***sObjId*, *name*, *dispId*, *base***)**

The **CreateAlias** method syntax has the following parts.

Part	Description
variable	A variable declared as an **Alias** object. It receives the new alias definition.
object	An object expression that evaluates to an **InterfaceDef** object.
sObjId	The object identifier to be used for the new alias object. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.
name	A string that stores the name of the new alias. Also, the name of the **InterfaceDef** object containing the base member.
dispId	The dispatch identifier to be used for accessing the new alias.
base	A string that stores the name of the interface member upon which the alias is based.

See Also

Alias Object

InterfaceDef Object

Object Identifiers and Internal Identifiers

InterfaceDef CreateMethodDef Method

This method creates a new method definition and attaches it to the interface definition.

Syntax

Set variable = object.**CreateMethodDef(***sObjId*, *name*, *dispId***)**

The **CreateMethodDef** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to an **InterfaceDef** object.
variable	A variable declared as a **MethodDef** object. It receives the new method definition.
sObjId	The object identifier to be used for the new method definition object. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.
name	The name of the new method.
dispId	The dispatch identifier to be used for accessing the new method.

See Also

InterfaceDef Object

MethodDef Object

Object Identifiers and Internal Identifiers

InterfaceDef CreatePropertyDef Method

This method creates a new property definition and attaches it to the interface definition.

Syntax

Set variable = object.**CreatePropertyDef(***sObjId*, *name*, *dispId*, *CType***)**

The **CreatePropertyDef** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to an **InterfaceDef** object.
variable	A variable declared as a **PropertyDef** object. It receives the new property definition.
sObjId	The object identifier to be used for the new property definition object. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.
name	The name of the new property.
dispId	The dispatch identifier to be used for accessing the new property.
CType	The C data type of the property. For a definition of valid values, see the ODBC documentation.

See Also

InterfaceDef Object

Object Identifiers and Internal Identifiers

PropertyDef Object

InterfaceDef CreateRelationshipColDef Method

This method creates a new collection type, attaches it to this interface, and associates it with the specified relationship type.

Syntax

Set variable = object.**CreateRelationshipColDef(***sObjId*, *name*, *dispId*, *isOrigin*, *flags*, *relshipDef***)**

The **CreateRelationshipColDef** method syntax has the following parts.

Part	Description
object	An object expression that evaluates to an **InterfaceDef** object.
variable	A variable declared as a **CollectionDef** object. It receives the new collection definition.
sObjId	The object identifier for the collection type. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.
name	The name of the new collection type.
dispId	The dispatch identifier to be used for Automation access to collections of this type.
isOrigin	Specifies whether collections of this type are origin collections. This is a Boolean parameter.
flags	Flags that specify naming, sequencing, and delete propagation behavior for the collection type. For more information about flag values, see CollectionDefFlags Enumeration.
RelshipDef	The relationship definition object to which this collection type is connected.

Remarks

By default, the collection definition specifies that zero to many items are

permitted in collections of this type. To specify a different minimum and maximum item count for the new collection type, change the **MinCount** and **MaxCount** properties before committing the transaction that contains this method invocation.

See Also

CollectionDef Object

CollectionDefFlags Enumeration

InterfaceDef Object

Object Identifiers and Internal Identifiers

RelationshipDef Object

InterfaceDef ObjectInstances Method

This method materializes an **ObjectCol** collection of all objects in the repository that expose this interface.

Syntax

Set variable = object.**ObjectInstances**

The **ObjectInstances** method syntax has the following parts.

Part	Description
variable	A variable declared as an **ObjectCol** object. It receives the collection of objects that expose this interface.
object	An object expression that evaluates to an **InterfaceDef** object.

Remarks

The retrieved collection contains one version of each object that conforms to a class exposing this interface. For each such object, the repository engine uses criteria to select which version to include in the collection. For more information, see Resolution Strategy for Objects and Object Versions.

ObjectInstances is not scoped to a workspace. All information models in a repository are included in the scope.

See Also

InterfaceDef Object

ObjectCol Object

InterfaceDef Ancestor Collection

This collection specifies the one base interface from which this interface derives. You use **Ancestor** collections to define inheritance.

Syntax

Set variable = object.**Ancestor(***index***)**

The **Ancestor** collection syntax has the following parts.

Part	Description
variable	A variable declared as an **InterfaceDef** object. It receives the specified base interface definition.
object	An object expression that evaluates to an **InterfaceDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**Ancestor.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Interface_InheritsFrom_Interface**	This is the type of relationship by which all items of the collection are connected to a common source

		object.
Source is Origin	Yes	The source object for the collection is also the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum Collection Size	One	The maximum number of items that can be contained in the collection is one.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-sensitive Names	Not applicable	Case-sensitive naming is not applicable for this collection.
Unique Names	Not applicable	Unique naming is not

		applicable for this collection.

See Also

InterfaceDef Object

InterfaceDef Classes Collection

This collection specifies which classes implement the interface.

Syntax

Set variable = object.**Classes(***index***)**

The **Classes** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ClassDef** object. It receives the specified class definition.
object	An object expression that evaluates to an **InterfaceDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by object.**Classes.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Class_Implements_Interface**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source is Origin	No	The source object for the collection is not the same

		as the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-sensitive Names	Not Applicable	Case-sensitive naming is not applicable for this collection.
Unique Names	Not Applicable	Unique naming is not applicable for this collection.

See Also

InterfaceDef Descendants Collection

This collection specifies other interfaces that derive from this interface.

Syntax

Set variable = object.**Descendants(***index***)**

The **Descendants** collection syntax has the following parts.

Part	Description
variable	A variable declared as an **InterfaceDef** object. It receives the specified interface definition.
object	An object expression that evaluates to an **InterfaceDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by object.**Descendants.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Types	**Interface_InheritsFrom_Interface**	This is the type of relationship by which all items of the collection are connected to a common source object.

Source is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-sensitive Names	Not applicable	Case-sensitive naming is not applicable for this collection.

Unique Names	Not applicable	Unique naming is not applicable for this collection.

See Also

InterfaceDef Object

InterfaceDef Implies Collection

This is the collection of **InterfaceDef** objects that are made available to another interface through implication.

Syntax

Set variable = object.**Implies(***index***)**

The **Implies** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **InterfaceDef** object. It receives the specified interface definition.
object	An object expression that evaluates to an **InterfaceDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. For more information, see Selecting Items in a Collection.

Remarks

You can define an implication between two interface definition objects of the form **Interface1** implies **Interface2**. For each such implication, the repository engine guarantees that every class that implements **Interface1** also implements the members of **Interface2**.

The **Implies** collection contains the interface definition objects that are automatically implemented whenever the current interface definition object is implemented. To define an implication in the opposite direction, use the **ImpliedBy** collection.

For example, if you extend an information model by creating a new version of an interface (**Interface1a**), you can add **Interface2** to the **Implies** collection of

Interface1a to guarantee that **Interface2** members are always available.

See Also

Interface Implication

InterfaceDef Object

ReposTypeLib Object

InterfaceDef ImpliedBy Collection

This is the collection of **InterfaceDef** objects that have been made available to another interface through implication.

Syntax

Set variable = object.**ImpliedBy(***index***)**

The **ImpliedBy** collection syntax has the following parts.

Part	Description
variable	A variable declared as an **InterfaceDef** object. It receives the specified interface definition.
object	An object expression that evaluates to an **InterfaceDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. For more information, see Selecting Items in a Collection.

Remarks

You can define an implication between two interface definition objects of the form **Interface2** is implied by **Interface1**. For example, if **Interface2** is implied by **Interface1**, the **ImpliedBy** collection for **Interface2** can include the **Interface1** object.

The **ImpliedBy** collection provides a way to define which interfaces are part of an implication relationship. This collection reflects the opposite direction of a relationship that is defined by the **Implies** collection.

See Also

Interface Implication

InterfaceDef Members Collection

This collection specifies which members are attached to the interface.

Syntax

Set variable = object.**Members(***index***)**

The **Members** collection syntax has the following parts.

Part	Description
variable	A variable declared as an object. It receives the specified property definition, method definition, or collection definition.
object	An object expression that evaluates to an **InterfaceDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by object.**Members.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Interface_Has_Members**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source is Origin	Yes	The source object for the

		collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	Yes	As a destination collection, this collection permits an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	The deletion of an origin object or relationship in the collection causes the deletion of the corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-sensitive Names	No	The relationship type for the collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination

		objects. This applies to collections whose relationship type permits destination objects to be named.

See Also

CollectionDef Object

InterfaceDef Object

MethodDef Object

PropertyDef Object

InterfaceDef Properties Collection

A **Properties** collection contains all of the persistent properties and collections that are attached to an object through a particular interface. The **InterfaceDef** object exposes four separate **Properties** collections. These collections are exposed by:

- The **IInterfaceDef2** interface (the default) or **IInterfaceDef** interface.
- The **IReposTypeInfo** or **IReposTypeInfo2** interface.
- The **IRepositoryObject** or **IRepositoryObject2** interface.
- The **IAnnotationalProps** interface.

Syntax

Set variable = object.**Properties(***index***)**

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. It receives the specified property.
object	An object expression; it evaluates to an object that exposes **IInterfaceDef** or **IInterfaceDef2**, **IReposTypeInfo** or **IReposTypeInfo2**, **IRepositoryObject** or **IRepositoryObject2,** or **IAnnotationalProps** as the default interface.
index	An integer index that identifies which property in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by

	object.**Properties.Count**.

Remarks

Additional steps are required for accessing members that are not part of the default interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

InterfaceDef Object

ReposProperty Object

InterfaceDef ReposTypeLibScopes Collection

This is the collection of repository type libraries that contain this definition.

Syntax

Set variable = object.**ReposTypeLibScopes(***index***)**

The **ReposTypeLibScopes** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposTypeLib** object. It receives the specified repository type library object.
object	An object expression that evaluates to an **InterfaceDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**TypeLibScopes.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**ReposTypeLib_ScopeFor_ReposTypeInfo**	This is the type of relationship by which all items of the collection are connected to a common

		source object.
Source is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	The deletion of an origin object or relationship in the collection causes the

		deletion of the corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-sensitive Names	No	The relationship type for the collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects. This applies to collections whose

		relationship type permits destination objects to be named.

See Also

InterfaceDef Object

ReposTypeLib Object

InterfaceDef ScriptsUsedByInterface Collection

This is the collection of **ScriptDef** objects that are implemented by this interface.

This collection is not attached to the default interface for this Automation object; it is attached to the **IClassDef2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**ScriptsUsedByInterface(***index***)**

The **ScriptsUsedByInterface** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ScriptDef** object. It receives the specified script definition.
object	An object expression that evaluates to an **InterfaceDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by object.**ScriptDef.Count**. For more information, see Selecting Items in a Collection.

See Also

IClassDef2 Interface

InterfaceDef Object

ReposTypeLib Object

ScriptDef Object

MethodDef Object

When you define a class for an information model, you specify the interfaces that the class implements. For each of those interfaces, you specify the members (properties, methods, and collections) that are attached to the interface. To attach a new method to an interface, use the **CreateMethodDef** method of the **InterfaceDef** object.

The definition of a method as a member of an interface does not result in the storage of method implementation logic in the repository. However, the method name is added to the set of defined member names for that interface. It also reserves the dispatch identifier of the method in the set of defined dispatch identifier values for the interface.

A **MethodDef** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. You can also access members that are defined for those objects and members of **IInterfaceMember2** and **IVersionAdminInfo2**. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use the **MethodDef** object to access or modify the characteristics of a method definition, or to determine the interface definition to which a particular method is attached.

Properties

Property	Description
DispatchID	The dispatch identifier to use when invoking a method that conforms to this method definition
Flags	Flags that specify details about this method definition
MemberSynonym	Stores a synonym of the method name

Methods

Method	Description
CreateParameterDef	Defines a **ParameterDef** object for this method definition

Collections

Collection	Description
Interface	The interface to which this method definition is attached
Properties	The collection of all persistent properties that are attached to the **MethodDef** object

See Also

InterfaceDef CreateMethodDef Method

IInterfaceMember2 Interface

IVersionAdminInfo2 Interface

ParameterDef Object

RepositoryObject Object

RepositoryObjectVersion Object

MethodDef DispatchID Property

This property contains the dispatch identifier that is used to invoke a method that conforms to this method definition.

Syntax

object.**DispatchID**

The **DispatchID** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **MethodDef** object

See Also

MethodDef Object

MethodDef Flags Property

This property is a flag that specifies whether the interface member should be visible to Automation queries.

Syntax

object.**Flags**=(*integer*)

The **Flags** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **MethodDef** object.
integer	Flag values are bit flags, and may be combined to set multiple options. For more information about flag values and descriptions, see the InterfaceMemberFlags Enumeration.

See Also

MethodDef Object

MethodDef MemberSynonym Property

This property is a string used as a synonym for a method name. The value that you specify must be unique.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IInterfaceMember2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**MemberSynonym**=*string*

The **MemberSynonym** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **MethodDef** object
string	A variable length string that can be a maximum of 255 characters in length

See Also

MethodDef Object

IInterfaceMember2 Interface

MethodDef CreateParameterDef Method

This method creates a **ParameterDef** object for a method definition.

Syntax

Set variable = object.**CreateParameterDef(***sObjID, Name, Type, Flags, Description, Default***)**

The **CreateParameterDef** method syntax has the following parts.

Part	Description
variable	A variable declared as a **ParameterDef** object. It receives the new parameter definition.
object	An object expression that evaluates to a **MethodDef** object.
sObjId	The object identifier to be used for the new parameter definition object. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.
name	The name of the new parameter.
type	The data type of the new parameter.
flags	The attributes of the new parameter. For more information about flag values and descriptions, see ParameterDef Flags Property.
description	An alternate description of the parameter that replaces the generic, default string that is generated by the Microsoft® SQL Server™ 2000 Meta Data Services Software Development Kit (SDK). This string is placed into an IDL file.
default	A string denoting the default value for the new parameter.

See Also

MethodDef Interface Collection

For a particular method definition, the **Interface** collection specifies which interface exposes a member of this type.

Syntax

Set variable = object.**Interface(***index***)**

The **Interface** collection syntax has the following parts.

Part	Description
variable	A variable declared as an **InterfaceDef** object. It receives the specified interface definition.
object	An object expression that evaluates to a **MethodDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by object.**Interface.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Interface_Has_Members**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	No	The source object for the

		collection is not the same as the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum Collection Size	One	The maximum number of items that can be contained in the collection is one.
Sequenced Collection	Yes	As a destination collection, this collection permits an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	Deleting an origin object or a relationship in the collection causes the deletion of a corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-sensitive Names	No	The relationship type for the collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects. This applies to collections whose

		relationship type permits destination objects to be named.

See Also

InterfaceDef Object

MethodDef Object

MethodDef Properties Collection

A **Properties** collection contains all of the persistent properties and collections that are attached to an object via a particular interface. The **MethodDef** object exposes three separate **Properties** collections. These collections are exposed by:

- The **IInterfaceMember2** interface (the default) or **IInterfaceMember** interface.
- The **IRepositoryObject** or **IRepositoryObject2** interface.
- The **IAnnotationalProps** interface.

Syntax

Set variable = object.**Properties(***index***)**

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. It receives the specified property.
object	An object expression; evaluates to an object that exposes **IInterfaceMember2** or **IInterfaceMember**, **IRepositoryObject** or **IRepositoryObject2**, or **IAnnotationalProps** as the default interface.
index	An integer index that identifies which property in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by object.**Properties.Count**. For more information, see Selecting Items in a Collection.

Remarks

For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

MethodDef Object

ReposProperty Object

ParameterDef Object

A parameter definition object represents the parameter of a method.

A **ParameterDef** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. You can also access members that are defined for these objects and members of **IReposTypeInfo**. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use the **ParameterDef** object to create parameters for a method definition object that you define.

Properties

Property	Description
Flags	A flag that defines attributes of the parameter value. You can define whether a parameter is the default parameter of the method, is optional, or is passed by reference or by value.
Default	A string denoting the default value for the parameter.
Description	A descriptive string placed into an Interface Definition Language (IDL) file that substitutes for the generic default text for the parameter type.
GUID	A globally unique identifier (GUID) that defines the interface identifier of a COM-based interface parameter.
Type	The data type of the parameter expressed as a constant value.

See Also

ParameterDef Default Property

This property is a string denoting the default value for the parameter.

Syntax

Object.**Default**=*string*

The **Default** property syntax has the following parts.

Part	Description
object	The **ParameterDef** object
string	A variable length string that can be a maximum of 255 characters

See Also

ParameterDef Object

ParameterDef Description Property

This property is a string placed into an IDL file that provides more descriptive information about a parameter, that is, for example, a **DISPATCH** interface. When you generate an Interface Definition Language (IDL) file, the **Description** property can be used instead of the generic default text to identify the parameter. For example, instead of the default text **DISPATCH** *, you can specify something like **IUMLCLass** *.

Syntax

Object.**Description**=*string*

The **Description** property syntax has the following parts.

Part	Description
object	The **ParameterDef** object
string	A variable length string that can be a maximum of 255 characters

See Also

ParameterDef Object

ParameterDef Flags Property

This property is an integer that determines the attributes of a parameter. The sum total of the flag values determines the combination of flags that apply.

Syntax

Object.**Flags**=*integer*

The **Flags** property syntax has the following parts.

Part	Description
object	The **ParameterDef** object
integer	A single flag name value, or an aggregated value that results from combining flag values

Remarks

Parameter definition flags, values, and descriptions are provided in the following table.

Flag Name and Value	Description
PARAMFLAGS_IN = 1	The parameter accepts a value passed to it as input.
PARAMFLAGS_OUT = 2	The parameter passes an output value by reference.
PARAMFLAGS_RETVAL = 4	The parameter passes a return value. Only one parameter for each method can be marked as a return value.
PARAMFLAGS_OPTIONAL = 8	An optional parameter. Once you define a parameter as optional, all subsequent parameters that follow must also be optional.

See Also

ParameterDef Object

ParameterDef GUID Property

This property is a string that stores the globally unique identifier (GUID) that defines the COM-based interface to which the parameter refers.

Syntax

Object.**GUID**

The **GUID** property syntax has the following part.

Part	Description
object	The **ParameterDef** object

Remarks

You cannot set GUID using the **CreateParameterDef** method. Setting the GUID property is useful when you have a dispatch-based interface (for example, **ITransactionObjectCol** object that has a data type of **vt_dispatch**). You can set a GUID as a parameter for a **TransactionObjectCol** object, even though the object is not a method.

See Also

ParameterDef Object

Object Identifiers and Internal Identifiers

ParameterDef Type Property

This property is the data type of the parameter. You can specify most data types that are supported by Automation objects. The value that you specify must be an integer.

Syntax

Set object.**Type**=(*integer*)

The **Type** property syntax has the following parts.

Part	Description
object	The **ParameterDef** object
integer	The integer associated with a Variant data type

Remarks

Automation Variant data types and integers are provided in the following table.

Variant data type	Integer
VT_ARRAY	0x2000
VT_UI1 (BYTE)	18
VT_BOOL	11
VT_BSTR	8
VT_CY (CURRENCY)	6
VT_DATE	7
VT_I2 (SHORT)	2
VT_14 (LONG)	3
VT_R4 (SINGLE)	4
VT_R8 (DOUBLE)	5
VT_DISPATCH	9
VT_UNKNOWN	13
VT_VARIANT	12

See Also

ParameterDef Object

PropertyDef Object

When you define a class for an information model, you specify the interfaces that the class implements. For each of those interfaces, specify the members (properties, methods, and collections) that are attached to the interface.

Before you can attach a property to an interface, a property definition object must exist for the property. The characteristics of the property (its name, dispatch identifier, data type, and various storage details) are stored in the property definition object. These characteristics are defined by the properties of the property definition object.

To create a new property definition

1. Use the **CreatePropertyDef** method of the **InterfaceDef** object.

2. Define any non-default characteristics of your new property definition by manipulating the properties of the property definition object.

3. Commit your changes to a repository database.

A **PropertyDef** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. You can also access members that are defined for these objects, and members of **IInterfaceMember2**, **IViewPropertyDef** and **IVersionAdminInfo2**. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use the **PropertyDef** object to retrieve or modify the characteristics of a property definition, or to determine which interface exposes a particular property.

Properties

Property	Description
APIType	The C data type of the property
ColumnName	The name of the column in the SQL table for this property
DispatchID	The dispatch identifier to use when accessing an instance of this type of property
Flags	Flags that specify details about this property definition
MemberSynonym	Stores a synonym of the property name
SQLBlobSize	The SQL BLOB size of the property
SQLScale	The number of digits to the right of the decimal point for a numeric property
SQLSize	The size in bytes of the property
SQLType	The SQL data type of the property

Collections

Collection	Description
EnumerationDef	The collection of **EnumerationDef** objects to which this property definition is attached
Interface	The interface to which this property definition is attached
Properties	The collection of all persistent properties that are attached to the **PropertyDef** object

See Also

EnumerationDef Object

InterfaceDef CreateProperty Method

RepositoryObject Object

RepositoryObjectVersion Object

PropertyDef APIType Property

The C data type of the property. For a definition of valid values, see the ODBC documentation.

Syntax

object.**APIType**

The **APIType** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **PropertyDef** object

See Also

PropertyDef Object

SQL and API Types Used in Property Definitions

PropertyDef ColumnName Property

An SQL table is used to store instance information for the properties of an interface. By default, there is a column in this table for each property that is defined as a member of the interface. The **ColumnName** string property specifies the name of the column in the SQL table for the property definition.

Syntax

object.**ColumnName**=(*string*)

The **ColumnName** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **PropertyDef** object
string	A variable length string that can be a maximum of 30 bytes

See Also

PropertyDef Object

Repository SQL Schema

PropertyDef DispatchID Property

This property contains the dispatch identifier to use when accessing an instance of this type of member.

This property is not attached to the default interface for the **PropertyDef** Automation object; it is attached to the **IInterfaceMember** interface. For details on how to access a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**DispatchID**

The **DispatchID** property syntax has the following part.

Part	Description
object	An object expression; evaluates to an object that exposes **IInterfaceMember** as the default interface

See Also

PropertyDef Object

PropertyDef Flags Property

The **PropertyDef** object exposes two separate **Flags** properties. Both the default interface, **IPropertyDef**, and a non-default interface, **IInterfaceMember**, expose a **Flags** property that you can set.

- The **IPropertyDef Flags** property is ignored. It is preserved for backward compatibility. Originally, this flag specified whether to create a column for the property. Column creation would occur in the SQL table providing persistent storage for the interface to which the property is attached. Without a column, instances of the property only attached to individual objects when setting the property value for that particular object.

- The **IInterfaceMember Flags** property specifies whether the interface member should be visible to Automation queries. For more information about flag values, see the InterfaceMemberFlags Enumeration.

Syntax

object.**Flags**

The **Flags** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **PropertyDef** object, for the default **Flags** property -or- An object expression that evaluates to an object that exposes **IInterfaceMember** as the default interface, for the alternate **Flags** property

Remarks

For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

IInterfaceMember Interface

PropertyDef Object

PropertyDef MemberSynonym Property

This property is a string used as a synonym for a property name. The value that you specify must be unique.

This property is not attached to the default interface for the repository Automation object; it is attached to the **IInterfaceMember2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**MemberSynonym**=(*string*)

The **MemberSynonym** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **PropertyDef** object
string	A variable length string that can be a maximum of 255 characters in length

See Also

PropertyDef Object

IInterfaceMember2 Interface

PropertyDef SQLBlobSize Property

The SQL Binary Large Object (BLOB) size of the property. For a definition of valid values, see the ODBC documentation.

This property is not attached to the default interface for the **PropertyDef** Automation object; it is attached to the **IPropertyDef2** interface. For details on how to access a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**SQLBlobSize**

The **SQLBlobSize** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **PropertyDef** object

See Also

IPropertyDef2 Interface

Programming BLOBs and Large Text Fields

PropertyDef Object

PropertyDef SQLScale Property

The number of digits to the right of the decimal point for a numeric property. This parameter is ignored unless the **SQLType** property specifies a SQL_NUMERIC, SQL_DECIMAL, or SQL_TIME data type.

Syntax

object.**SQLScale**

The **SQLScale** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **PropertyDef** object

See Also

PropertyDef Object

SQL and API Types Used in Property Definitions

PropertyDef SQLSize Property

The size in bytes of the property. This parameter is ignored when the data type of the property inherently specifies the size of the property.

Syntax

object.**SQLSize**

The **SQLSize** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **PropertyDef** object

Note If a **SQLSize** is set to a value greater than 65535, the engine divides the entered number by 65536 and sets **SQLSize** to the value of the remainder of the division, but no error is returned.

See Also

PropertyDef Object

PropertyDef SQLType Property

The SQL data type of the property. For a definition of valid values, see the ODBC documentation.

Syntax

object.**SQLType**

The **SQLType** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **PropertyDef** object

See Also

PropertyDef Object

SQL and API Types Used in Property Definitions

PropertyDef EnumerationDef Collection

An **EnumerationDef** collection specifies which **EnumerationDef** objects use the property.

Syntax

Set variable = object.**EnumerationDef(***index***)**

The **EnumerationDef** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. It receives the specified property.
object	An object expression that evaluates to **PropertyDef** object.
index	An integer index that identifies which object in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by object.**Count**. For more information, see Selecting Items in a Collection.

See Also

PropertyDef Object

EnumerationDef Object

PropertyDef Interface Collection

For a particular property definition, the **Interface** collection specifies which interface exposes a member of this type.

This collection is not attached to the default interface for the **PropertyDef** Automation object; it is attached to the **IInterfaceMember** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Set variable = object.**Interface(***index***)**

The **Interface** collection syntax has the following parts.

Part	Description
variable	A variable declared as an **InterfaceDef** object. It receives the specified interface definition.
object	An object expression; evaluates to an object that implements **IInterfaceMember** as the default interface.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**Interface.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Interface_Has_Members**	This is the type of

		relationship by which all items of the collection are connected to a common source object.
Source is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum Collection Size	One	The maximum number of items that can be contained in the collection is one.
Sequenced Collection	Yes	As a destination collection, this collection permits an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	The deletion of an origin object or relationship in the collection causes the deletion of the corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-sensitive Names	No	The relationship type for the collection does not permit the use of case-sensitive names for destination objects.

Unique Names	No	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects. This applies to collections whose relationship type permits destination objects to be named.

See Also

PropertyDef Object

PropertyDef Properties Collection

A **Properties** collection contains all of the persistent properties and collections that are attached to an object through a particular interface. The **PropertyDef** object exposes four separate **Properties** collections. These collections are exposed by:

- The **IPropertyDef2** interface (the default) or **IPropertyDef** interface.
- The **IRepositoryObject** or **IRepositoryObject2** interface.
- The **IInterfaceMember** or **IInterfaceMember2** interface.
- The **IAnnotationalProps** interface.

Syntax

Set variable = object.**Properties(***index***)**

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. It receives the specified property.
object	An object expression; evaluates to an object that exposes **IPropertyDef2** or **IPropertyDef**, **IRepositoryObject** or **IRepositoryObjectVersion**, **IInterfaceMember** or **IInterfaceMember2**, or **IAnnotationalProps** as the default interface.
index	An integer index that identifies which property in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by

	object.**Properties.Count**.

Remarks

For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

PropertyDef Object

ReposProperty Object

RelationshipDef Object

When you define an information model according to the repository API, you define classes of objects, types of relationships that can exist between objects, and various properties that are attached to these object classes and relationship types. The relationship types that you define in your information model are represented by instances of the **RelationshipDef** class. To add a new relationship type (also referred to as a relationship definition) to an information model, use the **CreateRelationshipDef** method of the **ReposTypeLib** object.

A **RelationshipDef** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. In addition to the members described here, you can access members that are defined for those objects. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use the **RelationshipDef** object to:

- Access persistent properties that are attached to a relationship definition.
- Determine which collection types are associated with a relationship definition.
- Determine which information models contain a relationship definition.

Properties

Property	Description
Name	The name of a **RelationshipDef** object
Synonym	A synonym of the name of the **RelationshipDef** object

Collections

Collection	Description
ItemInCollections	The collection of two collection types that are associated with this relationship definition
Properties	The collection of all persistent properties that are attached to the **RelationshipDef** object
ReposTypeLibScopes	The collection of all repository type libraries that contain this definition

See Also

RepositoryObject Object

RepositoryObjectVersion Object

ReposTypeLib CreateRelationshipDef Method

ReposTypeLib Object

RelationshipDef Name Property

This property stores the name of the **RelationshipDef** object.

Syntax

Object.**Name**=(*string*)

The **Name** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **RelationshipDef** object
string	A variable-length string that can be a maximum of 200 characters in length

See Also

RelationshipDef Object

INamedObject Interface

RelationshipDef Synonym Property

This property stores a synonym of the name of the **RelationshipDef** object. Synonym values are not unique for relationship definition objects.

This property is not attached to the default interface for the **RelationshipDef** Automation object; it is attached to the **IReposTypeInfo2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

Object.**Synonym**=(*string*)

The **Synonym** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **RelationshipDef** object
string	A variable-length string that can be a maximum of 200 characters in length

See Also

INamedObject Interface

IReposTypeInfo2 Interface

RelationshipDef Name Property

RelationshipDef Object

RelationshipDef ItemInCollections Collection

A relationship type is associated with two collection types. Origin collections conform to one collection type (the origin collection type), and destination collections conform to the other collection type (the destination collection type). The **ItemInCollections** collection contains the two collection definition objects that represent the origin and destination collection types.

If the relationship type has not yet been connected to its origin and destination collection types, this collection can contain less than two collection types.

Syntax

Set variable = object.**ItemInCollections(***index***)**

The **ItemInCollections** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **CollectionDef** object. It receives the specified collection definition.
object	An object expression that evaluates to a **RelationshipDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by object.**ItemInCollections.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Collection_Contains_Items**	This is the type of

		relationship by which all items of the collection are connected to a common source object.
Source Is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Two	The maximum number of items that can be contained in the collection is two.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-sensitive Names	Not applicable	Case-sensitive naming is not applicable for this collection.
Unique Names	Not applicable	Unique naming is not applicable for this collection.

See Also

CollectionDef Object

RelationshipDef Object

RelationshipDef Properties Collection

A **Properties** collection contains all of the persistent properties and collections that are attached to an object through a particular interface. The **RelationshipDef** object exposes three separate **Properties** collections. These collections are exposed by:

- The **IReposTypeInfo2** interface (the default) or **IReposTypeInfo** interface.
- The **IRepositoryObject** or **IRepositoryObject2** interface.
- The **IAnnotationalProps** interface.

Syntax

Set variable = object.**Properties(***index***)**

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. It receives the specified property.
object	An object expression. It evaluates to an object that exposes **IReposTypeInfo2** or **IReposTypeInfo**, **IRepositoryObject** or **IRepositoryObject2**, or **IAnnotationalProps** as the default interface.
index	An integer index that identifies which property in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by object.**Properties.Count**. For more information, see Selecting Items in a Collection.

Remarks

For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

RelationshipDef Object

ReposProperty Object

RelationshipDef ReposTypeLibScopes Collection

This is the collection of repository type libraries that contain the current **RelationshipDef** object. .

Syntax

Set variable = object.**ReposTypeLibScopes(***index***)**

The **ReposTypeLibScopes** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposTypeLib** object. It receives the specified repository type library object.
object	An object expression that evaluates to a **RelationshipDef** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**TypeLibScopes.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**ReposTypeLib_ScopeFor_ReposTypeInfo**	This is the type of relationship by which all items of the collection are connected to a

		common source object.
Source Is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	The deletion of an origin object or relationship in the collection

		causes the deletion of the corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-sensitive Names	No	The relationship type for the collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects. This applies to collections

		whose relationship type permits destination objects to be named.

See Also

RelationshipDef Object

ReposTypeLib Object

ReposRoot Object

There is one root object in each repository. The root object is the starting point for navigating to other objects in the repository. The root object is also the starting point for two kinds of data navigation: type data navigation and instance data navigation.

- **Type data navigation**

  When you create an information model, the corresponding repository type library is attached to the root object through the **ReposTypeLibs** collection. This collection can be used to enumerate all of the information models that are contained in a repository database.

- **Instance data navigation**

  After an information model is installed, a repository database can be populated with object instance data. This instance data consists of objects and relationships that conform to the classes and relationship types of the information model.

  Because the objects are connected through relationships, you can navigate through this data. However, to enable general-purpose repository browsers to navigate this data, the first navigational step must be from the root object of the repository through a root relationship collection to the primary objects of your information model. Primary objects are objects that make a good starting point for navigating to other objects of your information model.

  Because this root relationship collection is different for each information model, the information model must define it. There are two options for attaching this relationship collection to the root object:

  - The **ReposRoot** class implements the **IReposRoot** interface. This interface is provided to information model creators as a connection point. You can add your connecting relationship collection to this interface.

- You can extend the **ReposRoot** class to implement a new interface that is defined in your information model. This interface implements a relationship collection that attaches the root object to the primary objects in your information model.

To facilitate navigation, the root object in all repositories always has the same object identifier. The symbolic name for this object identifier is **OBJID_ReposRootObj**.

A **ReposRoot** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. In addition to the members described here, you can access members that are defined for those objects**.** For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use the **ReposRoot** object to:

- Obtain a starting point for navigating to objects in a repository database.
- Create a new information model container.
- Attach a relationship collection to the root object of the repository that connects to the primary objects of your information model.
- Determine which information models are currently stored in a repository database.

Methods

Method	Description
CreateTypeLib	Creates an empty repository type library that you can use to define a new information model

Collections

Collection	Description
ReposTypeLibs	The collection of repository type libraries that are currently stored in the repository
Properties	The collection of all persistent properties that are attached to the **ReposRoot** object
Workspaces	The collection of all workspaces present in the repository

See Also

IReposRoot Interface

RepositoryObject Object

RepositoryObjectVersion Object

ReposRoot CreateTypeLib Method

This method creates an empty repository type library and attaches it to the root of the repository. Each repository type library represents an information model. After you create an empty information model, you can populate it with classes, interfaces, properties, and so on.

Syntax

Set variable = object.**CreateTypeLib(***sObjId*, *Name*, *TypeLibId***)**

The **CreateTypeLib** method syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposTypeLib** object. It receives the new repository type library.
object	An object expression that evaluates to a **ReposRoot** object.
sObjId	The object identifier to be used for the new repository type library object. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.
Name	The name of the new repository type library.
TypeLibId	The global identifier by which this repository type library is referenced.

Remarks

This method does not create an external type library; it creates a **ReposTypeLib** object in a repository database.

You use this method only when you are creating an information model programmatically. If you are using the model installer to add a predefined information model to a repository, you do not need this method.

See Also

Object Identifiers and Internal Identifiers

ReposRoot Object

ReposTypeLib Object

ReposRoot ReposTypeLibs Collection

This collection contains the repository type libraries currently stored in a repository database. Each repository type library represents an information model.

Syntax

Set variable = object.**ReposTypeLibs(***index***)**

The **ReposTypeLibs** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposTypeLib** object. It receives the specified repository type library.
object	An object expression that evaluates to a **ReposRoot** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by object.**ReposTypeLibs.Count**.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**TlbManager_ContextFor_ReposTypeLibs**	This is the type of relationship by which all items of the collection are connected to a

		common source object.
Source Is Origin	Yes	The source object for the collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	The deletion of an origin object or relationship in the collection causes the

		deletion of the corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-sensitive Names	No	The relationship type for the collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects. This applies to collections whose

		relationship type permits destination objects to be named.

See Also

ReposRoot Object

ReposTypeLib Object

ReposRoot Properties Collection

This collection contains all of the persistent properties and collections that are attached to an object through a particular interface. The **ReposRoot** object exposes four separate **Properties** collections. These collections are exposed by:

- The **IManageReposTypeLib** interface (the default).
- The **IReposRoot** interface.
- The **IRepositoryObject** interface.
- The **IAnnotationalProps** interface.

Syntax

Set variable = object.**Properties(***index***)**

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. It receives the specified property.
object	An object expression. It evaluates to an object that exposes **IManageReposTypeLib**, **IReposRoot**, **IRepositoryObject**, or **IAnnotationalProps** as the default interface.
index	An integer index that identifies which property in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by object.**Properties.Count**.

Remarks

Additional steps are required for accessing members that are not part of the default interface. For more information about how to access a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

ReposRoot Object

ReposRoot Workspaces Collection

This collection is the set of object versions checked out to a workspace.

Syntax

Set variable = object.**Workspaces(***index***)**

The **Workspaces** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **Workspace** object. It receives the specified item in the collection.
object	An object expression that evaluates to a **ReposRoot** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**Workspaces.Count**.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**WsContainer_Contains_Workspaces**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is	Yes	The source object

Origin		for the collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-sensitive	No	The collection

Names		does not permit the use of case-sensitive names for destination objects.
Unique Names	No	The relationship type for the collection does not require that the name of a destination object be unique within the collection of destination objects.

See Also

ReposRoot Object

ReposTypeLib Object

ReposTypeLib Object

Repository type libraries are represented by **ReposTypeLib** objects. There is one repository type library for every information model contained in a repository database. Each information model provides a logical grouping of all of the type definitions.

ReposTypeLib objects are often used to support navigation when traversing an information model. However, you can also use **ReposTypeLib** objects to define or extend information models programmatically. To insert a new information model into the repository database, use the **ReposRoot** object.

A **ReposTypeLib** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. You can also access members that are defined for those objects. For more information about how to access a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use a **ReposTypeLib** object to:

- Define new classes, relationship types, and interfaces for an information model to create or extend an information model programmatically.
- Retrieve or modify the global identifier associated with a repository type library.
- Determine which type definitions are associated with a particular repository type library.

Properties

Property	Description
Name	The name of the **ReposTypeLib** object

Prefix	The prefix of an interface name that distinguishes an interface from other identically named interfaces
TypeLibID	The global identifier for the repository type library

Methods

Method	Description
CreateClassDef	Creates a new class definition object
CreateInterfaceDef	Creates a new interface definition object
CreateRelationshipDef	Creates a new relationship definition object

Collections

Collection	Description
ReposTypeInfos	The collection of all classes, interfaces, and relationship types that are defined in the repository type library
ReposTypeLibContexts	The collection of one repository root object that is the context for the repository type library
Properties	The collection of all persistent properties that are attached to the **ReposTypeLib** object

See Also

RepositoryObject Object

RepositoryObjectVersion Object

ReposRoot Object

ReposTypeLib Name Property

This property stores the name of the **ReposTypeLib** object.

Syntax

object.**Name**=*string*

The **Name** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **ReposTypeLib** object
string	A variable length string that can be a maximum of 255 characters

See Also

ReposTypeLib Object

ReposTypeLib Prefix Property

This property stores a prefix for the information model that distinguishes it from all other information models in a repository.

This property is not attached to the default interface for the **ReposTypeLib** Automation object; it is attached to the **IReposTypeLib2** interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

Syntax

object.**prefix**=(*string*)

The **prefix** property syntax has the following parts.

Part	Description
object	An object expression that evaluates to a **ReposTypeLib** object.
string	A variable length string that can be a maximum of 255 characters. Prefix values are added during model installation. If no prefix is specified, the first three letters of the information model name are applied as a default value.

Remarks

The prefix is also used in XML for identifying namespaces (for example, "Uml" in UmlElement).

Attaching a prefix guarantees that a class that implements interfaces from different information models does not introduce a name conflict when both interfaces share the same name. The prefix is also used in XML for identifying namespaces (for example, "Uml" in UmlElement).

For the Open Information Model (OIM), prefix values are added during model

installation. If no prefix is specified, the first three letters of the information model name are applied as a default value.

For the latest version of the MDC (Meta Data Coalition) OIM, prefix values must be added programmatically. Prefix values are not added during model installation.

See Also

IReposTypeLib2 Interface

ReposTypeLib Object

ReposTypeLib TypeLibID Property

This property is the global identifier for the repository type library. If you copy this property to a variable, declare the variable as a **Variant**.

Syntax

object.**TypeLibID**

The **TypeLibID** property syntax has the following part.

Part	Description
object	An object expression that evaluates to a **ReposTypeLib** object

See Also

Object Identifiers and Internal Identifiers

ReposTypeLib Object

ReposTypeLib CreateClassDef Method

This method creates a new class definition object. No interfaces are attached to the class. After you create a class definition object, you can define it using the **ClassDef** object.

Syntax

Set variable = object.**CreateClassDef(***sObjId*, *Name*, *sClsId***)**

The **CreateClassDef** method syntax has the following parts.

Part	Description
variable	A variable declared as a **ClassDef** object. It receives the new class definition.
object	An object expression that evaluates to a **ReposTypeLib** object.
sObjId	The object identifier to be used for the new class definition object. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.
Name	The name of the new class.
sClsId	The global identifier by which this class is referenced.

See Also

ClassDef Object

Object Identifiers and Internal Identifiers

ReposTypeLib Object

ReposTypeLib CreateInterfaceDef Method

The **CreateInterfaceDef** method creates a new interface definition object. Use the **AddInterface** method of the **ClassDef** object to attach the interface to a class definition object.

Syntax

Set variable = object.**CreateInterfaceDef(***sObjId*, *Name*, *sIId*, *Ancestor***)**

The **CreateInterfaceDef** method syntax has the following parts.

Part	Description
variable	A variable declared as an **InterfaceDef** object. It receives the new interface definition.
object	An object expression that evaluates to a **ReposTypeLib** object.
sObjId	The object identifier to be assigned to the new interface definition object. If this parameter is set to OBJID_NULL, the repository engine assigns an object identifier for you.
Name	The name of the interface that is to be created.
sIId	The interface identifier associated with the signature for this interface. If there is none, set this parameter to zero.
Ancestor	The base interface from which the new interface is derived.

See Also

ClassDef AddInterface Method

InterfaceDef Object

Object Identifiers and Internal Identifiers

ReposTypeLib Object

ReposTypeLib CreateRelationshipDef Method

This method creates a relationship definition object for a new relationship type. Once the relationship definition is created, use the **CreateRelationshipColDef** method of the **InterfaceDef** object to create origin and destination collection definitions for the new relationship type.

Syntax

Set variable = object.**CreateRelationshipDef(***sObjId*, *Name***)**

The **CreateRelationshipDef** method syntax has the following parts.

Part	Description
variable	A variable declared as a **RelationshipDef** object. It receives the new relationship definition.
object	An object expression that evaluates to a **ReposTypeLib** object.
sObjId	The object identifier for the new relationship type. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.
Name	The name of the new relationship type.

See Also

InterfaceDef CreateRelationshipColDef Method

InterfaceDef Object

Object Identifiers and Internal Identifiers

RelationshipDef Object

ReposTypeLib Object

ReposTypeLib ReposTypeInfos Collection

This collection contains all classes, interfaces, and relationship types that are associated with a repository type library. The repository engine uses this collection to enforce the unique naming of all classes, interfaces, and relationship types for a repository type library.

Syntax

Set variable = object.**ReposTypeInfos(***index***)**

The **ReposTypeInfos** collection syntax has the following parts.

Part	Description
variable	A variable declared as an object. It receives the specified class definition, interface definition, or relationship definition.
object	An object expression that evaluates to a **ReposTypeLib** object.
index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the total number of elements in the collection. The number of elements in the collection is specified by object.**ReposTypeInfos.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**ReposTypeLib_ScopeFor_ReposTypeInfo**	This is the type of relationship by which all

		items of the collection are connected to a common source object.
Source Is Origin	Yes	The source object for the collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	Deleting an origin object or a relationship

		in the collection causes the deletion of a corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-sensitive Names	No	This collection does not use case-sensitive names for destination objects.
Unique Names	Yes	The collection requires that the name of a destination object be unique within the collection of destination objects.

See Also

ClassDef Object

InterfaceDef Object

ReposTypeLib ReposTypeLibContexts Collection

This collection contains one repository root object that is the context for a repository type library.

Syntax

Set variable = object.**ReposTypeLibContexts(***index***)**

The **ReposTypeLibContexts** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposRoot** object. It receives the repository root object.
object	An object expression that evaluates to a **ReposTypeLib** object.
Index	An integer index that identifies which element in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**ReposTypeLibContexts.Count**. For more information, see Selecting Items in a Collection.

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**TlbManager_ContextFor_ReposTypeLibs**	This is the type of relationship by which all items of the collection are connected to a

		common source object.
Source Is Origin	No	The source object for the collection is not the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	Deleting an origin object or a relationship in the collection causes the

		deletion of a corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-sensitive Names	No	The collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects.

See Also

ReposTypeLib Properties Collection

A **Properties** collection contains all of the persistent properties and collections that are attached to an object through a particular interface. The **ReposTypeLib** object exposes three separate **Properties** collections. These collections are exposed by:

- The **IReposTypeLib2** interface (the default) and the **IReposTypeLib** interface.
- The **IReposTypeInfo** or **IReposTypeInfo2** interface.
- The **IAnnotationalProps** interface.

Syntax

Set variable = object.**Properties(***index***)**

The **Properties** collection syntax has the following parts.

Part	Description
variable	A variable declared as a **ReposProperty** object. It receives the specified property.
object	An object expression. It evaluates to an object that exposes: • **IReposTypeLib** or **IReposTypeLib2** • **IRepositoryObject** or **IRepositoryObject2** -or- • **IAnnotationalProps** as the default interface.

index	An integer index that identifies which property in the collection is to be addressed. The valid range is from one to the number of elements in the collection. The number of elements in the collection is specified by object.**Properties.Count**. For more information, see Selecting Items in a Collection.

Remarks

Additional steps are required for accessing members that are not part of the default interface. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

See Also

ReposProperty Object

ReposTypeLib Object

ScriptDef Object

A script definition object represents Microsoft® ActiveX® script that you can associate with a method or property definition. A **ScriptDef** object provides a way to store the implementation of a method in an information model. You can also use **ScriptDef** to validate properties before storing them in a repository database.

A **ScriptDef** object is also a **RepositoryObject** and a **RepositoryObjectVersion** object. You can also access members that are defined for those objects and members of **IReposTypeInfo**. For more information about accessing a member of an interface that is not the default interface, see Accessing Automation Object Members.

When to Use

Use the **ScriptDef** object to define a method or a property validation rule.

Properties

Property	Description
Body	Contains the body of a script.
Language	Contains a string that identifies the language in which the script is written. You can provide script in Microsoft Visual Basic® Scripting Edition (VBScript) and Microsoft JScript®.
Name	The name of a **ScriptDef** object.

Methods

Method	Description
ValidateScript	Validates script syntax

Collections

Collection	Description
UsingClasses	Class collections for which the script applies
UsingInterfaces	Interface collections for which the script applies
UsingMembers	Member collections for which the script applies

See Also

Defining Script Objects

MethodDef Object

RepositoryObject Object

RepositoryObjectVersion Object

ScriptDef Body Property

This property stores the body of a script.

Syntax

Object.**Body**=*string*

The **Body** property syntax has the following parts.

Part	Description
object	The **ScriptDef** object
string	A variable length string that can be a maximum of 64 KB in length

Remarks

You can provide validation using the **ValidateScript** method.

See Also

ScriptDef Object

ScriptDef ValidateScript Method

ScriptDef Language Property

This property stores the name of the language in which the script is written.

Syntax

Object.**Language**=*string*

The **Language** property syntax has the following parts.

Part	Description
object	The **ScriptDef** object.
string	A variable length string that can be a maximum of 255 characters in length. Valid values are Microsoft® Visual Basic® Scripting Edition (VBScript) and Microsoft JScript®.

See Also

ScriptDef Object

ScriptDef Name Property

This property stores the name of the **ScriptDef** object.

Syntax

Object.**Name**=*string*

The **Name** property syntax has the following parts.

Part	Description
object	The **ScriptDef** object
string	A variable length string that can be a maximum of 200 characters in length

See Also

ScriptDef Object

ScriptDef ValidateScript Method

This method validates script provided through the **Body** property. Validation is performed by the Microsoft® ActiveX® Scripting Engine for the specified language.

Syntax

Object.**ValidateScript**

The **ValidateScript** method syntax has the following part.

Part	Description
object	The **ScriptDef** object

Remarks

The **ValidateScript** method returns S_OK if the script can be executed; otherwise it returns an error generated by the script engine.

The syntax of the script is checked by instantiating the script. For more information, see Defining Script Objects.

See Also

ScriptDef Object

ScriptDef UsingClasses Collection

This collection contains classes that use the script.

This collection is the origin collection of a relationship that associates a script with a class. The destination collection of this relationship is the **ScriptsUsedByClass** collection.

Syntax

Set variable=object.**UsingClasses**(*index*)

The **UsingClasses** syntax has the following parts.

Part	Description
variable	Variable declared as an object.
object	A **ClassDef** object.
index	An integer index that identifies which class in the collection is to be addressed. The valid range is from one to the total number of classes in the collection. For more information, see Selecting Items in a Collection.

See Also

ClassDef Object

ScriptDef Object

ScriptDef UsingInterfaces Collection

This collection contains interfaces that use the script.

This collection is the origin collection of a relationship that associates a script with an interface. The destination collection of this relationship is the **ScriptsUsedByInterface** collection.

Syntax

Set variable=object.**UsingInterfaces**(*index*)

The **UsingInterfaces** syntax has the following parts.

Part	Description
variable	Variable declared as an object.
object	An **InterfaceDef** object.
index	An integer index that identifies which interface in the collection is to be addressed. The valid range is from one to the total number of interfaces in the collection. For more information, see Selecting Items in a Collection.

See Also

InterfaceDef Object

ScriptDef Object

ScriptDef UsingMembers Collection

This collection contains interface members (methods or properties) that use the script.

This collection is the origin collection of a relationship that associates a script with an interface member. The destination collection of this relationship is the **ScriptsUsedByMember** collection.

Syntax

Set variable=object.**UsingMembers**(*index*)

The **UsingMembers** syntax has the following parts.

Part	Description
variable	Variable declared as an object.
object	A **MethodDef** or **PropertyDef** object.
index	An integer index that identifies which member in the collection is to be addressed. The valid range is from one to the total number of members in the collection. For more information, see Selecting Items in a Collection.

See Also

MethodDef Object

PropertyDef Object

ScriptDef Object

Meta Data Services Programming

COM Reference

The COM Reference documents the COM classes and interfaces of the repository API. An equivalent reference is available for Automation objects.

In this documentation, classes and interfaces are organized into four categories.

Section	Description
Repository Engine Classes	Describes the classes that expose the functionality of the repository engine.
RTIM Classes	Describes the Repository Type Information Model (RTIM) classes. These are the abstract classes to which an information model must conform.
Repository Engine COM Interfaces	Describes the interfaces that expose the functionality of the repository engine.
RTIM COM Interfaces	Describes the interfaces that define the RTIM.

See Also

Automation Reference

Information Models

Repository API Reference

Repository Engine

Repository Object Architecture

Visual C++ Wrappers with Meta Data Services

Meta Data Services Programming

Repository Engine Classes

Repository engine classes are used to add, retrieve, and change information model data in a repository. To create a new information model, or extend an existing one, use the Repository Type Information Model (RTIM) classes. For more information, see RTIM Classes.

All repository engine classes expose the standard **IUnknown** and **IDispatch** interfaces that provide fundamental COM and Automation support.

The following table lists the repository engine classes in alphabetical order.

Class	Description
ObjectCol	Defines a set of repository objects that can be enumerated
Relationship	Connects two repository objects in a repository database
RelationshipCol	Defines a set of relationships that connect a particular source object to a set of one or more target objects
Repository	Defines a connection to a particular repository
RepositoryObject	Defines an object that is stored in a repository database and managed by the repository engine
RepositoryObjectVersion	Defines a versioned object that is stored in a repository database and is managed by the repository engine
ReposProperties	Provides access to the **Properties** collection
ReposProperty	Provides access to a persistent member (a property or collection) of an information model interface
TransientObjectCol	Defines an object collection that you can create and dynamically populate at run time using script and object

	methods rather than persisted data in a repository database
VersionCol	Defines a collection of object versions
VersionedRelationship	Defines a connection between two versioned objects in a repository database
Workspace	Defines a subset of a central, shared repository

See Also

COM Reference

Repository API Reference

Repository Engine

ObjectCol Class

An object collection is a set of repository objects that can be enumerated. Two kinds of object collections are supported by the repository engine:

- The collection of destination objects that correspond to the relationships in a relationship collection. Use the **RelationshipCol** class to manage this kind of collection.

- The collection of all objects in the repository that implement a particular interface. Use the **ObjectCol** class to enumerate objects in this kind of object collection.

Use the **IInterfaceDef::ObjectInstances** method to materialize an instance of this class.

When to Use

Use the **ObjectCol** class to access the collection of repository objects that expose a particular interface.

Interfaces

Interface	Description
IObjectCol	Manages objects in a collection
IObjectCol2	Exposes methods for controlling the load status of an object collection

See Also

IInterfaceDef::ObjectInstances

IRepositoryDispatch Interface

Relationship Class

A relationship connects two repository objects in a repository database. In this release of the repository engine, relationships are versioned. That is, every relationship is a **VersionedRelationship** object. A versioned relationship can connect a particular version of a repository object to one or more specific versions of the target object. Because every relationship is a **VersionedRelationship** object, you can declare any relationship with the following line of Microsoft® Visual Basic®:

```
Dim myVersionedRship As VersionedRelationship
```

In earlier releases of the repository engine, the object model included the **Relationship** class, but not the **VersionedRelationship** class. If you have Visual Basic programs written against earlier releases of the repository engine, those programs might include declarations like the following:

```
Dim oldRship As Relationship
```

These programs will work, because the repository object model still includes the **Relationship** class. Visual Basic recognizes the **Relationship** declaration as valid. But in this release, every relationship is a versioned relationship. So the object **oldRship**, even though it is declared as a **Relationship**, must conform to the **VersionedRelationship** class.

To ensure that objects declared as **Relationship** conform to the **VersionedRelationship** class, the repository engine uses the same Class Factory for both classes. In this way, any object that you declare as a **Relationship** implements the exact same methods as any object that you declare as a **VersionedRelationship**. In effect, the following two lines of Visual Basic code are identical:

```
Dim oldRship As Relationship
Dim myVersionedRship As VersionedRelationship
```

Interfaces

Interface	Description
IAnnotationalProps	Gets and sets annotational properties
IRelationship	Connects two objects in an information model

See Also

IAnnotationalProps Interface

IRelationship Interface

IRepositoryDispatch Interface

IRepositoryItem Interface

RelationshipCol Class

Repository Engine Classes

VersionedRelationship object

RelationshipCol Class

A relationship collection is the set of relationships that connect a particular source repository object to a set of one or more target objects. All of the relationships in the collection must conform to the same relationship type.

When to Use

Use the **RelationshipCol** class to manage a collection of relationships in a repository database.

Interfaces

Interface	Description
IObjectCol	Manages objects in a collection
IRelationshipCol	Manages a collection of relationships
IReposQuery	Provides filters on collections to control how objects appear in an object collection
ITargetObjectCol	Manages objects in a target object collection

See Also

IRepositoryDispatch Interface

Relationship Class

Repository Engine Classes

Repository Class

When you populate an information model, the objects and relationships that conform to the model are stored in a repository. Multiple information models may be stored in the same repository. The **Repository** class represents your connection to a particular repository.

When to Use

You can use the **Repository** class to connect to a repository, retrieve the root object of the repository, create new repository objects, and manage repository transactions and error handling.

Interfaces

Interface	Description
IRepository	Creates and populates a repository
IRepository2	Manages individual versions of repository objects
IReposErrorQueueHandler	Creates and assigns error queues
IRepositoryODBC	Provides access to repository database connection information
IRepositoryODBC2	Exposes methods that enable you to set or get options for retrieving object collections asynchronously
IRepositoryTransaction	Controls repository transactions
IRepositoryTransaction2	Supports distributed, atomic transactions
IReposOptions	Exposes methods for getting, setting, or resetting engine options
IReposQuery	Provides filters on collections to control how objects appear in an object collection

See Also

IRepositoryDispatch

Repository Engine Classes

RepositoryObjectVersion Class

An object version is a specific state of a repository object at a given point in time. Each object version consists of a state that can be permanently fixed (protected from further modification) plus another, always modifiable, aspect. The state of an object version consists of its nonannotational property values and its origin collections. The other, always modifiable, aspect of an object version consists of its annotational properties and its destination collections.

When to Use

Use the **RepositoryObjectVersion** class to manipulate a particular version of a repository object.

Interfaces

Interface	Description
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IRepositoryObject	Retrieves repository object identifiers
IRepositoryObjectStorage	Creates and loads repository objects
IRepositoryObjectVersion	Manages repository object versions
IWorkspaceItem	Manages repository object versions in a workspace

See Also

RepositoryObject Class

Repository Engine Classes

RepositoryObject Class

A repository object is an object that is stored in a repository database and is managed by the repository engine.

In this release of the repository engine, objects can be versioned. A repository object version is a particular edition of a repository object. Each version of an object can differ from other versions of that object in its property values and collections. When you obtain a reference to a repository object, you are actually manipulating a particular version of that object. That is, you manipulate a **RepositoryObjectVersion** object. Because you manipulate particular versions of objects, you can declare any object with the following line of Microsoft® Visual Basic® code:

```
Dim myVersionedReposObject As RepositoryObjectVersion
```

In earlier releases of the repository engine, the object model included the **RepositoryObject** class, but not the **RepositoryObjectVersion** class. If you have Visual Basic programs written against earlier releases, those programs might include declarations like the following:

```
Dim oldReposObject As RepositoryObject
```

These programs will work because the repository object model still includes the **RepositoryObject** object. Visual Basic recognizes the preceding declaration as valid. But whenever you manipulate an object, you actually manipulate a specific version of that object. So the object **oldReposObject**, even though it is declared as a **RepositoryObject**, must conform to the **RepositoryObjectVersion** class.

To ensure that objects declared as **RepositoryObject** conform to the **RepositoryobjectVersion** class, the repository engine uses the same Class Factory for both classes. In this way, any object that you declare as a **Relationship** implements the exact same methods as any object you declare as a **VersionedRelationship**. In effect, the following two lines of Visual Basic code are identical:

```
Dim myVersionedReposObject As RepositoryObjectVersion
Dim oldReposObject As RepositoryObject
```

Interfaces

Interface	Description
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IRepositoryObject	Retrieves repository object identifiers
IRepositoryObject2	Provides binary large object (BLOB) and large text file support, and exposes additional meta data about an object
IRepositoryObjectStorage	Creates and loads repository objects

See Also

IRepositoryDispatch Interface

IRepositoryItem Interface

IRepositoryObject Interface

IRepositoryObjectStorage Interface

Repository Engine Classes

RepositoryObjectVersion

ReposProperties Class

The **ReposProperties** class provides access to the **Properties** collection. The **Properties** collection gives you a convenient mechanism for enumerating through all of the persistent properties and collections of an interface. The **ReposProperty** class can be used to access the individual members in the **Properties** collection.

When to Use

Use the **ReposProperties** class to access the properties and collections of a repository object, when no custom implementation is available, and you do not already know what members are exposed by the object's interface.

Interfaces

Interface	Description
IReposProperties	Provides access to the members that are attached to an interface

See Also

IReposProperty Interface

IRepositoryDispatch Interface

Repository Engine Classes

ReposProperty Class

The **ReposProperty** class provides access to a persistent member (a property or collection) of an information model interface.

When to Use

Use the **ReposProperty** class to access a persistent interface member, when a custom implementation is not available and you do not already know the type or name of the member.

Interfaces

Interface	Description
IReposProperty	Provides access to the members that are attached to an interface

See Also

IReposProperties Interface

IRepositoryDispatch Interface

Repository Engine Classes

TransientObjectCol Class

This class defines an object collection that you can create and dynamically populate at run time using script and object methods rather than persisted data in a repository database. It simulates a standard, persisted object collection.

When to Use

Use this class to create an object collection that is instantiated by application code and populated dynamically at run time. With this object, you can:

- Create an object collection that is not stored in a repository database.
- Get a count of the number of objects in the collection.
- Add and remove objects to and from the collection.

Interfaces

Interface	Description
ITransientObjectCol	Defines a set of repository objects that can be instantiated by an application and populated at run time
IObjectCol	Manages objects in a collection

See Also

Repository Engine Classes

TransientObjectCol Object

VersionCol Class

A version collection is a collection of object versions.

When to Use

Use the **VersionCol** class to manage a collection of object versions.

Interfaces

Interface	Description
IVersionCol	Manages object versions in a collection

See Also

RepositoryObjectVersion Class

RepositoryObjectVersion Object

Repository Engine Classes

VersionedRelationship Class

A relationship connects two repository objects in a repository database. In this release of the repository engine, relationships are versioned. That is, every relationship is a **VersionedRelationship** object. A versioned relationship can connect a particular version of a repository object to one or more specific versions of the target object.

When to Use

Use the **VersionedRelationship** class to manipulate a relationship, or to retrieve the source, target, origin, or destination object for a relationship.

Interfaces

Interface	Description
IAnnotationalProps	Gets and sets annotational properties
IRelationship	Retrieves information about a relationship
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IVersionedRelationship	Manages the **TargetVersions** collection of a versioned relationship

See Also

RelationshipCol Class

Repository Engine Classes

VersionedRelationship object

Workspace Class

A workspace is a subset of the repository within which you can operate on tool data in isolation from other repository activity.

To insert a new workspace into a repository database, use any class that implements the **IWorkspaceContainer** interface. The **ReposRoot** class is one such class.

When to Use

Use the **Workspace** class to perform any operation you would perform within the repository, when you want to perform the operation in isolation from other repository activity.

Interfaces

Interface	Description
INamedObject	Manages object names
IRepository	Creates and populates a repository
IRepository2	Creates and manages subsequent versions of repository objects
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IRepositoryObject	Retrieves repository object identifiers
IRepositoryObjectVersion	Manages individual versions of objects, and the relationships among individual versions of the same object
IRepositoryODBC	Provides access to a repository database through an ODBC connection
IRepositoryODBC2	Exposes methods that enable you to set or get options for retrieving object collections asynchronously
IReposQuery	Provides filters on collections to control how

	objects appear in an object collection
IVersionAdminInfo2	Retains properties inherited from **IVersionAdminInfo** and sets or retrieves version comments
IWorkspace	Manages workspaces
IWorkspaceItem	Manages the participation of object versions in workspaces

See Also

IAnnotationalProps Interface

IRepositoryObjectStorage Interface

IReposTypeLib Interface

Repository Engine Classes

ReposRoot Class

Meta Data Services Programming

Repository Engine COM Interfaces

The repository engine interfaces expose the properties and methods that are used to add, retrieve, and change information model data in a repository database.

These interfaces work with the interfaces that describe an information model. The Repository Type Information Model (RTIM) interfaces are listed separately. For more information, see RTIM COM Interfaces.

All repository engine interfaces inherit from the standard **IUnknown** and **IDispatch** interfaces, which provide fundamental COM and Automation support.

The following repository engine interfaces are listed alphabetically.

Interfaces	Description
IannotationalProps Interface	Accesses the annotational properties of a repository object or relationship
IEnumRepositoryErrors Interface	Provides enumeration capabilities for the set of errors that have been placed on the repository error queue
INamedObject Interface	Accesses the **Name** property of a repository object that exposes this interface
IObjectCol Interface	Enumerates the collection of repository objects that conform to a particular class or expose a particular interface
IObjectCol2 Interface	Controls the load status of an object collection
IRelationship Interface	Connects two repository objects in a repository database
IRelationshipCol Interface	Manages the relationships that belong to a particular relationship collection
IReposErrorQueueHandler Interface	Creates a repository error queue and

	retrieves an interface pointer to an error queue
IRepository Interface	Creates and accesses a repository session
IRepository2 Interface	Manipulates versioned objects within a repository session
IRepositoryDispatch Interface	Accesses the properties and collections of a repository object, when no custom implementation is available
IRepositoryErrorQueue Interface	Manages the errors that belong to a particular repository error queue
IRepositoryItem Interface	Defines general purpose methods that are used to manage repository items
IRepositoryObject Interface	Provides methods to manage repository objects
IRepositoryObject2 Interface	Supports Meta Data Browser by accessing meta data about information models
IRepositoryObjectStorage Interface	Initializes the memory image for a repository object
IRepositoryObjectVersion Interface	Manipulates any version of an object
IRepositoryODBC Interface	Obtains or releases an ODBC connection handle, or retrieves the ODBC connection
IRepositoryODBC2 Interface	Sets or gets options when loading object collections asynchronously
IRepositoryTransaction Interface	Begins, commits, stops, or sets options on a repository transaction, or obtains information about a transaction state
IRepositoryTransaction2 Interface	Begins, commits, or stops a distributed repository transaction
IReposOptions Interface	Gets, sets, or resets engine options

IReposProperties Interface	Accesses a **Properties** collection
IReposProperty Interface	Provides access to a stored member (a property or collection) of an information model interface
IReposProperty2 Interface	Supports Meta Data Browser by retrieving meta data for an interface without having to query the database
IReposPropertyLarge Interface	Handles binary large objects (BLOBs) and large text fields
IReposQuery Interface	Filters on collections for the purpose of controlling how objects appear in an object collection
ISummaryInformation Interface	Maintains **Comments** and **ShortDescription** properties
ITargetObjectCol Interface	Manages the repository objects that belong to a particular relationship collection
ITransientObjectCol Interface	Creates and dynamically populates an object collection at run time using script and object methods rather than stored data in a repository database
IVersionAdminInfo Interface	Retains and manipulates administrative information about object versions
IVersionAdminInfo2 Interface	Retains version string, comment, and description data
IVersionCol Interface	Manages a collection of versioned objects
IVersionedRelationship Interface	Manages a collection of versioned relationship objects
IWorkspace Interface	Manages the object versions present in the workspace and the workspace container
IWorkspaceContainer Interface	Retrieves the collection of workspaces in a repository

IWorkspaceItem Interface	Manages the participation of object versions within workspaces

See Also

COM Reference

Information Models

Repository API Reference

IAnnotationalProps Interface

Annotational properties are repository properties that can be associated with individual repository objects or relationships. Before an annotational property value can be attached to a repository object, two requirements must be met:

- The object must conform to an object class that exposes the **IAnnotationalProps** interface.

- A property definition object must exist for an **IAnnotationalProps** interface property. The name of the property definition object must match the name of your annotational property.

If these two requirements are met, you can attach an annotational property value to an object using the **IReposProperty::put_Value** method to set the value of the annotational property for that particular object.

When to Use

Annotational properties are not recommended. Support for annotational properties will not be included in future releases of the repository engine.

Version 3.0 of the repository engine still supports annotational properties. If you are already using annotational properties, you can use the **IAnnotationalProps** interface to access the annotational properties of a repository object or relationship.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

Remarks

Annotational properties are maintained by the repository engine as string data. The creator and users of the annotational property must get and set the property value using the appropriate data type through the **VARIANT** structure. If a data type other than **string** is used, the repository engine performs the appropriate data conversion.

Because all annotational properties in the repository must be defined as interface members of the **IAnnotationalProps** interface, all annotational property names share the same name space. When you choose a name for an annotational property, make the name as specific and unique as possible.

See Also

ClassDef Class

CollectionDef Class

InterfaceDef Class

IReposProperty::put_Value

MethodDef Class

PropertyDef Class

Relationship Class

RelationshipDef Class

ReposTypeLib Class

ReposRoot Class

IEnumRepositoryErrors Interface

This interface provides enumeration capabilities for the set of errors that have been placed on the repository error queue.

When to Use

Use the **IEnumRepositoryErrors** interface to access the queue of repository errors.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IEnumRepositoryErrors method	Description
Clone	Clones the current enumerator
Next	Returns the next one or more elements
Reset	Resets the enumerator to the beginning
Skip	Skips over the next one or more elements

See Also

Error Handling Overview

IRepositoryErrorQueue::_NewEnum

Repository Errors

IEnumRepositoryErrors::Clone

Use this method to create a clone of the COM enumerator object. After cloning, the two enumerators operate independently of each other.

HRESULT Clone(IEnumRepositoryErrors ***ppIEnum* **)**

Parts

**ppIEnum*

[out]
The interface pointer for the new enumerator object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IEnumRepositoryErrors Interface

IEnumRepositoryErrors::Next

Use this method to retrieve the next one or more elements from the enumeration. There are two variations of this method.

HRESULT Next(**ULONG** *iCount,*
REPOSERR **psErrors,*
ULONG **piFetched*
)

HRESULT Next(IErrorInfo ***ppIErrorInfo*);

Parts

iCount

[in]
The number of elements the caller is requesting.

**psErrors*

[out]
The array of **REPOSERROR** structures for the retrieved items.

**ppIErrorInfo*

[out]
The interface pointer to the error information object for the first element in the error queue.

**piFetched*

[out]
The number of elements actually fetched for the caller.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IEnumRepositoryErrors Interface

REPOSERROR Data Structure

IEnumRepositoryErrors::Reset

Use this method to reset the enumerator to the beginning of the enumeration sequence.

HRESULT Reset(*void***)**

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IEnumRepositoryErrors Interface

IEnumRepositoryErrors::Skip

Use this method to skip over the next one or more elements in the enumeration.

HRESULT Skip(ULONG *iCount***)**

Parts

iCount

[in]
The number of elements to be skipped.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IEnumRepositoryErrors Interface

INamedObject Interface

Typically, a name is associated with a repository object through a naming relationship. The collection for such a relationship provides the scope for the name, and can require that all names in the collection be unique. This is the preferred method for naming objects, when a given object will be the destination of only one naming relationship.

If your information model contains a class that is not the destination of a naming relationship type, or is the destination of multiple relationship types, but no single relationship type is the obvious choice to be the naming relationship type, you can attach the **Name** property to the class. This is accomplished by defining your class to implement the **INamedObject** interface. If your class implements the **INamedObject** interface, the repository engine will use that interface when asked to retrieve or set an object name.

When to Use

Use the **INamedObject** interface to access the **Name** property of a repository object that exposes this interface.

Properties

Property	Description
Name	The name of the object

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the persistent members exposed by the **INamedObject** interface.

Remarks

None of the standard repository engine classes implement the **INamedObject** interface by default. However, the repository engine does use the **INamedObject** interface, if the interface is exposed by a repository object.

When the **IRepositoryItem::get_Name** method is invoked for a repository object, the repository engine will perform these steps to retrieve the name:

1. If the object exposes the **INamedObject** interface, the repository engine returns the value of the **Name** property on the **INamedObject** interface.

2. Otherwise, the repository engine searches for a naming relationship for

which the current object is the destination object, taking the workspace context into consideration.

3. If such a relationship is found, the repository engine returns the name associated with that relationship.

4. If the object is not the destination of a naming relationship, the repository engine returns a null name.

When the **IRepositoryItem::put_Name** method is invoked for a repository object, the repository engine will perform these steps to set the name:

1. The repository engine sets the value of the **Name** property of all naming relationships for which the object is the destination.

2. If the object exposes the **INamedObject** interface, the repository engine also sets the value of the **Name** property attached to that interface.

See Also

IRepositoryItem get_Name

IRepositoryItem put_Name

Naming and Unique-Naming Collections

Naming Objects, Collections, and Relationships

Workspace Context

INamedObject Name Property

This property contains the name of an object that exposes the **INamedObject** interface. The name can be up to 200 bytes in length.

Dispatch Identifier: DISPID_ObjName (68)

Property Data Type: **string**

See Also

INamedObject Interface

IObjectCol Interface

An object collection is a set of repository objects that can be enumerated. Two kinds of object collections are supported by the repository engine:

- The collection of destination objects that correspond to the relationships in a relationship collection. Use the **ITargetObjectCol** interface to manage this kind of collection.

- The collection of all objects in the repository that conform to a particular class or expose a particular interface.

When to Use

Use the **IObjectCol** interface to enumerate the collection of repository objects that conform to a particular class or expose a particular interface. With this interface, you can:

- Get a count of the number of objects in the collection.

- Enumerate the objects in the collection.

- Retrieve an **IRepositoryObject** pointer to one of the objects in the collection.

- Refresh the cached image of the object collection.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers.
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an Automation object.

IobjectCol method	Description
Get_Count	Retrieves a count of the number of objects in the collection.
Get_Item	Retrieves an **IRepositoryObject** interface pointer for the specified collection object.
_NewEnum	Retrieves an enumeration interface pointer for the collection.
Refresh	Refreshes the cached image of the object collection.

See Also

IRepositoryObject Interface

ITargetObjectCol Interface

ObjectCol Class

IObjectCol::get_Count

This method retrieves a count of the number of repository objects that are in the object collection.

HRESULT get_Count(long **piCount***)**

Parameters

**piCount*

[out]
The number of objects in the collection.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IObjectCol Interface

IObjectCol::_NewEnum

This method retrieves an enumeration interface pointer for the object collection. This interface is a standard Automation enumeration interface. It supports the **Clone**, **Next**, **Reset**, and **Skip** methods. You can use the enumeration interface to step through the objects in the collection.

HRESULT _NewEnum(**IUnknown** ***ppIEnumObjects*
)

Parameters

**ppIEnumObjects*

[out]
The enumeration interface pointer.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IObjectCol Interface

IObjectCol::get_Item

This method retrieves the specified object from the collection.

HRESULT get_Item(**VARIANT** *sItem*,
IRepositoryObject ***ppIReposObj*
)

Parameters

sItem

[in]
Identifies the item to be retrieved from the collection. This parameter can be either the index or the object identifier of the item.

**ppIReposObj*

[out]
The **IRepositoryObject** interface pointer for the version of the specified object from the collection.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

When this method is invoked through the **ITargetObjectCol** interface (which inherits this method from the **IObjectCol** interface), an object can also be retrieved by name, but only if it is the destination object of a naming relationship.

This method returns a specific version of the item. To choose which version, the repository engine follows a resolution strategy.

See Also

IObjectCol Interface

IRepositoryObject Interface

Resolution Strategy for Objects and Object Versions

IObjectCol::Refresh

This method refreshes the cached image of the collection. All unchanged data for objects in the collection is flushed from the cache.

HRESULT Refresh(long *iMilliseconds***)**

Parameters

iMilliseconds

[in]
This value is ignored.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Note The **Refresh()** method asynchronously refreshes the object collection (reloads the object collection and refreshes target objects) when the asynchronous mode is in effect. The calling thread should check to determine whether refresh is complete. If the calling thread tries to read data, refresh the collection, or construct an enumerator while refresh is in progress, it will be blocked until refresh is complete.

See Also

IObjectCol Interface

IObjectCol2 Interface

This interface exposes methods that enable you to control the load status of an object collection. The **IObjectCol2** interface also inherits the methods of the **IObjectCol** interface.

When to Use

Use the **IObjectCol2** interface to:

- Obtain the load status of an object collection.
- Cancel the load operation of an object collection.

Methods

IUnknown Method	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch Method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers.
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an Automation object.

IObjectCol Method	Description
get_Count	Retrieves a count of the number of objects in the collection.
get_Item	Retrieves an **IRepositoryObject** interface pointer for the specified collection object.
_NewEnum	Retrieves an enumeration interface pointer for the collection.
Refresh	Refreshes the cached image of the object collection.

IObjectCol2 Method	Description
get_LoadStatus	Obtains the load status of the collection.
Cancel	Requests the cancellation of the ongoing load operation.

See Also

IObjectCol

IRepositoryObject Interface

ITargetObjectCol Interface

ObjectCol Class

IObjectCol2::get_LoadStatus

This method is used to obtain the load status of the object collection.

HRESULT get_LoadStatus(**long** **piStatus*
)

Parts

piStatus

[out, retval]

One of the constant values: READY, INPROGRESS, CANCELLED, or FAILED.

Return Value

S_OK

The method completed successfully.

ErrorValues

The method failed to complete successfully.

See Also

IObjectCol2 Interface

IObjectCol2::Cancel

IObjectCol2::Cancel

This method requests the cancellation of the ongoing load operation of the object collection.

HRESULT Cancel ();

Return Value

S_OK

The method completed successfully.

ErrorValues

The method failed to complete successfully.

See Also

IObjectCol2 Interface

IObjectCol2::get_LoadStatus

IRelationship Interface

A relationship connects two repository objects in the repository database. A relationship has an origin object, a destination object, and a set of properties. Each relationship conforms to a particular relationship type.

When to Use

Use the **IRelationship** interface to manipulate a relationship, or to retrieve the source, target, origin, or destination object for a relationship.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch	

method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IRepositoryItem method	Description
Delete	Deletes a repository item
get_Interface	Retrieves an interface pointer to the specified item interface
get_Name	Retrieves the name associated with an item
get_Repository	Retrieves the **IRepository** interface pointer for an open repository instance of an item
get_Type	Retrieves the type of an item
Lock	Locks the item
put_Name	Sets the name associated with an item

IRelationship method	Description
get_Destination	Retrieves an interface pointer to the destination object
get_Origin	Retrieves an interface pointer to the origin object
get_Source	Retrieves an interface pointer to the source object
get_Target	Retrieves an interface pointer to the target object

See Also

IRepository::Refresh

Relationship Class

IRelationship::get_Destination

This method retrieves an **IRepositoryObject** interface pointer to the destination object version of a relationship.

HRESULT get_Destination(IRepositoryObject ***ppIReposObj*
)

Parameters

**ppIRepObj*

[out]
The **IRepositoryObject** interface pointer for the destination object version.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

If the relationship is a destination versioned relationship, this method is equivalent to the **get_Source** method. If it is an origin versioned relationship, this method is equivalent to the **get_Target** method.

See Also

IRelationship::get_Source

IRelationship::get_Target

IRelationship::get_Origin

IRelationship Interface

IRelationship::get_Origin

This method retrieves an **IRepositoryObject** interface pointer to the origin object version of a relationship.

HRESULT get_Origin(IRepositoryObject ***ppIRepObj**
)

Parameters

**ppIRepObj*

[out]
The **IRepositoryObject** interface pointer for the origin object version.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

If the relationship is an origin versioned relationship, this method is equivalent to the **get_Source** method. If the relationship is a destination versioned relationship, this method is equivalent to the **get_Target** method.

See Also

IRelationship::get_Source

IRelationship::get_Target

IRelationship::get_Destination

IRelationship Interface

IRelationship::get_Source

This method retrieves an **IRepositoryObject** interface pointer to the source object version of a relationship.

HRESULT get_Source(IRepositoryObject ***ppIRepObj**
)

Parameters

**ppIRepObj*

[out]
The **IRepositoryObject** interface pointer for the source object version.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRelationship::get_Target

IRelationship Interface

IRelationship::get_Target

This method retrieves an **IRepositoryObject** interface pointer to a target object version of a relationship.

HRESULT get_Target(IRepositoryObject ***ppIRepObj**
)

Parameters

**ppIRepObj*

[out]
The **IRepositoryObject** interface pointer for the target object version.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

The repository engine uses follows a resolution strategy to choose a specific target object version to return.

There is one situation in which the repository engine can fail to return a version of the target object: If you are operating within a workspace, and the collection of **TargetVersions** of the versioned relationship does not include the object version that is present in the workspace, this method fails.

See Also

IRelationship::get_Source

IRelationship Interface

Resolution Strategy for Objects and Object Versions

IRelationshipCol Interface

A relationship collection is the set of versioned relationships that connect a particular source object version to a set of one or more target objects. All of the relationships in the collection must conform to the same relationship type.

When to Use

Use the **IRelationshipCol** interface to manage the repository relationships that belong to a particular relationship collection. With this interface, you can:

- Get a count of the number of relationships in the collection.
- Enumerate the relationships in the collection.
- Add and remove relationships to and from the collection.
- If the collection is sequenced, place a relationship in a specific spot in the collection sequence.
- Retrieve an **IRelationship** pointer to one of the relationships in the collection.
- Obtain the identifier of the definition object of the collection.
- Retrieve an interface pointer for the source object of the collection.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces

AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRelationshipCol method	Description
Add	Adds a relationship to the collection
get_Count	Retrieves a count of the number of relationships in the collection
_NewEnum	Retrieves an enumeration interface pointer for the collection
get_Source	Retrieves an interface pointer for the collection's source object
get_Type	Retrieves the object identifier for the collection's definition object
Insert	Inserts a relationship into a specific spot in a sequenced collection
get_Item	Retrieves an **IRelationship** interface pointer for the specified relationship
Move	Moves a relationship from one spot to another in a sequenced collection
Refresh	Refreshes the cached image of the relationship collection

Remove	Removes a relationship from the collection

Remarks

The **IRelationshipCol** interface is similar to the **ITargetObjectCol** interface. Use the **IRelationshipCol** interface when you are primarily interested in working with relationships. Use the **ITargetObjectCol** interface when you are primarily interested in working with objects.

See Also

ITargetObjectCol Interface

IRelationshipCol::Add

This method is used to add a new item to a repository relationship collection when the sequencing of relationships in the collection is not important. An interface pointer for the new relationship is passed back to the caller.

HRESULT Add(**IDispatch** **plReposObj*,
BSTR *Name*,
IRelationship ***pplRelship*
);

Parts

**plReposObj*

[in]
The object for which a relationship is to be added to the relationship collection. The value of *plReposObj* is the specific version of the target object. If you are operating within the context of a workspace, the target object version you specify with *plReposObj* must be present in the workspace.

Name

[in]
The name of the new relationship.

**pplRelship*

[out]
The **IRelationship** interface pointer of the newly added relationship.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

You can add a relationship to a collection only when the collection source object is also the collection origin object.

When you call this method, the origin version must be unfrozen.

You can use this method to create a new versioned relationship between the source object version and a version of the target object. You cannot use it to enlarge a versioned relationship. If the source object version already has a relationship to any version of the target object, this method will fail. You can include another version of the target object in the versioned relationship by adding an item to the **TargetVersions** collection of the versioned relationship.

See Also

IRelationship Interface

IRelationshipCol Interface

IRelationshipCol::get_Count

This method retrieves a count of the number of relationships that are in the relationship collection.

HRESULT get_Count(**long** **piCount*
);

Parts

**piCount*

[out]
The number of relationships in the collection.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Each item in the collection is a versioned relationship. Thus, the returned count indicates how many objects are in the collection, not the number of object versions.

See Also

IRelationshipCol Interface

IRelationshipCol::_NewEnum

This method retrieves an enumeration interface pointer for the relationship collection. This interface is a standard Automation enumeration interface. It supports the **Clone**, **Next**, **Reset**, and **Skip** methods. You can use the enumeration interface to step through the relationships in the collection.

HRESULT _NewEnum(**IUnknown** ***ppIEnumRelships*
);

Parts

**ppIEnumRelships*

[out]
The enumeration interface pointer.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRelationshipCol Interface

IRelationshipCol::get_Source

This method retrieves an interface pointer for the source object version of the collection.

HRESULT get_Source(
IRepositoryObject ***ppIInterface*
);

Parts

**ppIInterface*

[out]
The interface pointer of the interface for the source object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRelationshipCol Interface

IRelationshipCol::get_Type

This method retrieves the type of the collection; that is, it returns the object identifier for the definition object of the collection.

HRESULT get_Type(
  VARIANT **pColDefObjId*
);

Parts

**pColDefObjId*

[out]
The object identifier of the definition object of the collection.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRelationshipCol Interface

IRelationshipCol::Insert

This method adds a relationship to the collection at a specified point in the collection sequence. An interface pointer for the new relationship is passed back to the caller.

HRESULT Insert(
   IDispatch **pIReposObj,*
   long *iIndex,*
   BSTR *Name,*
   IRelationship ***ppIRelship*
);

Parts

**pIReposObj*

[in]
The repository object to be inserted into the collection sequence through the new relationship. The value of *plReposObj* is the specific version of the target object. If you are operating within the context of a workspace, the target object version you specify with *plReposObj* must be present in the workspace.

iIndex

[in]
The index of the sequence location where the relationship is to be inserted. If another relationship is already present at this sequence location, the new relationship is inserted in front of the existing relationship.

Name

[in]
The name you supply for the object to which the new relationship connects.

**ppIRelship*

[out]

The **IRelationship** interface pointer for the new relationship.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Relationships can be inserted into a collection only if the collection source object is also the collection origin object.

This method can only be used for collections that are sequenced.

When you call this method, the origin version must be unfrozen.

You can use this method to insert a new versioned relationship between the source object version and a target object version. You cannot use it to enlarge a versioned relationship. If the source object version already has a relationship to any version of the target object, this method will fail. You can include another version of the target object in the versioned relationship by adding an item to the **TargetVersions** collection of the versioned relationship.

See Also

IRelationshipCol Interface

IRelationship Interface

IRelationshipCol::get_Item

This method retrieves the specified relationship from the collection.

HRESULT get_Item(
  VARIANT *sItem*,
  IRelationship ***ppIRelship*
);

Parts

sItem

[in]
Identifies the item to be retrieved from the collection. This parameter can be the index, the name, or the object identifier of the item.

**ppIRelship*

[out]
The **IRelationship** interface pointer for the specified relationship from the collection.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Each item in the collection is a versioned relationship. That is, it has a collection of **TargetVersions**. When you obtain a reference to the target object of a particular item (with the **get_Target** method of the **IRelationship** interface), the repository engine chooses a particular version of the target object from the items

in the **TargetVersions** collection of the versioned relationship.

Using the *sItem* parameter to specify the relationship by destination object name is supported only for naming collections.

See Also

IRelationshipCol Interface

IRelationship Interface

IRelationshipCol::Move

This method moves a relationship from one point in the collection sequence to another point.

HRESULT Move(
  long *iIndexFrom*,
  long *iIndexTo*
);

Parts

iIndexFrom

[in]
The index of the relationship to be moved in the collection sequence.

iIndexTo

[in]
The index of the sequence location to which the relationship is to be moved.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

This method can be used only for collections that are sequenced.

The origin object version must be unfrozen.

See Also

IRelationshipCol Interface

IRelationshipCol::Refresh

This method refreshes the cached image of the collection. All unchanged data for relationships in the collection is flushed from the cache.

HRESULT Refresh(long *iMilliseconds***);**

Parts

iMilliseconds

[in]
This value is ignored.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRelationshipCol Interface

IRelationshipCol::Remove

This method deletes a relationship from its relationship collection. The exact behavior of this method depends on whether the relationship collection is an origin collection or a destination collection.

If the relationship collection is an origin collection, this method deletes the versioned relationship.

If the relationship collection is a destination collection, this method first performs object-version resolution to yield a single target-object version, and then it removes that target-object version from the **TargetVersions** collection of the relationship.

HRESULT Remove(
VARIANT *sItem*
);

Parts

sItem

[in]
Identifies the item to be removed from the collection. This parameter can be either the index or the name associated with the item.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

A relationship can be removed by name only if it is a unique-naming

relationship.

If the source is the origin, the origin version must be unfrozen.

If the relationship is a destination relationship and the resolution strategy yields a target object version that is frozen, this method fails.

Removing an item from a sequenced collection does not update the collection sequence order.

See Also

IRelationshipCol Interface

IReposErrorQueueHandler Interface

Errors that occur while accessing a repository are saved on a repository error queue. A repository error queue is a collection of **REPOSERROR** structures. Each thread of execution with an open repository instance can access one active error queue at a time.

When to Use

Use the **IReposErrorQueueHandler** interface to create a repository error queue, assign an error queue to a thread of execution, or retrieve an interface pointer to a thread's currently assigned error queue.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IReposErrorQueueHandler method	Description
CreateErrorQueue	Creates a new repository error queue
SetErrorQueue	Sets the active error queue for a thread
GetErrorQueue	Retrieves an interface pointer to the currently active error queue for a thread

See Also

Handling Errors

Repository Class

REPOSERROR Data Structure

IReposErrorQueueHandler::CreateErrorQueue

This method creates a repository error queue. After it has been created, the error queue is available to be assigned to a thread context.

HRESULT CreateErrorQueue(IRepositoryErrorQueue
***ppIErrorQueue*
);

Parameters

**ppIErrorQueue*

[out]
The interface pointer for the newly created repository error queue.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposErrorQueue Interface

IReposErrorQueueHandler Interface

IReposErrorQueueHandler::GetErrorQueue

This method retrieves the repository error queue that is assigned to the current thread.

HRESULT GetErrorQueue(
IRepositoryErrorQueue ***ppIErrorQueue*
);

Parameters

**ppIErrorQueue*

[out]
The interface pointer for the error queue that is currently assigned to this thread.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposErrorQueue Interface

IReposErrorQueueHandler Interface

IReposErrorQueueHandler::SetErrorQueue

This method assigns the specified repository error queue to the current thread context.

HRESULT SetErrorQueue(
  IRepositoryErrorQueue **pIErrorQueue*
);

Parameters

pIErrorQueue

[in]
The interface pointer for the error queue to be assigned to this thread.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposErrorQueue Interface

IReposErrorQueueHandler Interface

IRepository Interface

When you define an information model, the classes, relationships, properties, and collections for the model are stored in a repository database. Multiple information models can be stored in the same repository.

When to Use

Use the repository interface to create and access repository databases. You can also use the repository interface to create and access repository objects in a repository database.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers.
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an Automation object.

IRepository method	Description
Create	Creates a repository database.
CreateObject	Creates a new repository object.

get_Object	Retrieves the **IRepositoryObject** interface pointer for a repository object.
get_RootObject	Retrieves the **IRepositoryObject** interface pointer for the root repository object.
get_Transaction	Retrieves the **IRepositoryTransaction** interface pointer for this repository instance.
InternalIDToObjectID	Translates an internal identifier to an object identifier.
ObjectIDToInternalID	Translates an object identifier to an internal identifier.
Open	Opens a repository database.
Refresh	Refreshes unchanged cached repository data.

See Also

Connecting to and Configuring a Repository

IRepository2 Interface

Repository Class

Repository Databases

IRepository::Create

This method creates a new repository. The fundamental repository tables are automatically created in the new repository. An **IRepositoryObject** interface pointer to the repository root object is passed back to the caller.

```
HRESULT Create(    BSTR              Connect,
   BSTR            User,
   BSTR            Password,
   long            fFlags,
   IRepositoryObject **ppIRootObj
);
```

Parts

Connect

[in]
The ODBC connection string to be used for accessing the database server that will host your new repository.

User

[in]
The user name to use for identification to the database server.

Password

[in]
The password that matches the *User* input parameter.

fFlags

[in]
Flags that determine database access and caching behavior for the open repository. For more information, see ConnectionFlags Enumeration.

**ppIRootObj*

[out]

The **IRepositoryObject** interface pointer for the root repository object of the new repository.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

Connecting to and Configuring a Repository

IRepository Interface

IRepositoryObject Interface

IRepository::CreateObject

This method creates the first version of a new repository object. The specified COM interface pointer to the new object is passed back to the caller.

HRESULT CreateObject(**VARIANT** *sTypeId*,
VARIANT *sObjId*,
IRepositoryObject ********ppIReposObj*
);

Parts

sTypeId

[in]
The type of the new object; that is, the object identifier of the class definition to which the new object conforms.

sObjId

[in]
The object identifier to be assigned to the new object. Pass in OBJID_NULL to have the repository engine assign an object identifier for you.

**ppIReposObj*

[out]
The **IRepositoryObject** interface pointer for the new repository object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

The new object will automatically create persistent storage for itself.

You can use this method only to create the first version of a repository object. To create subsequent versions of the object, use **IRepositoryObjectVersion::CreateVersion**.

This method can only be called from the shared repository. It cannot be called from a workspace. The workaround is to create the object through the central repository and include it in the workspace.

See Also

IRepository Interface

IRepository2 Interface

IRepository::get_Object

This method retrieves an **IRepositoryObject** interface pointer to the specified repository object.

HRESULT get_Object(**VARIANT** *sObjectId*,
IRepositoryObject ***ppIReposObj*
);

Parts

sObjectId

[in]
The object identifier of the repository object to be retrieved.

**ppIReposObj*

[out]
The **IRepositoryObject** interface pointer for the repository object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

The returned **IRepositoryObject** interface pointer refers to a specific version of the repository object. The repository engine follows a resolution strategy to choose a specific version.

See Also

IRepository Interface

Resolution Strategy for Objects and Object Versions

IRepository::get_RootObject

This method obtains a pointer to the root object of the repository that is currently open. The root object is the repository object to which all other repository objects are (either directly or indirectly) connected.

**HRESULT get_RootObject(IRepositoryObject ****ppIRootObj*
);

Parts

**ppIRootObj*

[out]
The **IRepositoryObject** interface pointer for the root repository object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepository Interface

IRepositoryObject Interface

IRepository::get_Transaction

This method retrieves the **IRepositoryTransaction** interface pointer for this repository instance. Use the **IRepositoryTransaction** interface to manage repository transactions for this repository instance.

**HRESULT get_Transaction(IRepositoryTransaction ****ppIRepTrans*
);

Parts

**ppIRepTrans*

[out]
The **IRepositoryTransaction** interface pointer.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepository Interface

IRepository::InternalIDToObjectID

This method translates an internal identifier into an object identifier. Internal identifiers are used by the repository engine to identify repository objects.

HRESULT InternalIDToObjectID(**VARIANT** *sInternalID,*
VARIANT *******sObjectId*
);

Parts

sInternalID

[in]
The internal identifier for the repository object.

**sObjectId*

[out]
The object identifier for the repository object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Repository object identifiers are globally unique, and they are the same across repositories for the same object. Repository internal identifiers are unique only within the scope of a single repository.

The translation performed by this method is performed without loading the object in question. This enables database queries involving an object or

relationship type identifier to be constructed without having to load the definition object.

See Also

IRepository Interface

IRepository::ObjectIDToInternalID

IRepository::ObjectIDToInternalID

This method translates an object identifier into an internal identifier. Internal identifiers are used by the repository engine to identify repository objects.

HRESULT ObjectIDToInternalID(**VARIANT** *sObjectID*,
VARIANT ***sInternalId**
);

Parts

sObjectID

[in]
The object identifier for the repository object.

**sInternalId*

[out]
The internal identifier for the repository object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Repository object identifiers are globally unique, and they are the same across repositories for the same object. Repository internal identifiers are unique only within the scope of a single repository.

The translation performed by this method is performed without loading the object in question. This enables database queries involving an object or

relationship type identifier to be constructed without loading the definition object.

See Also

IRepository Interface

IRepository::InternalIDToObjectID

IRepository::Open

This method opens a repository. An **IRepositoryObject** interface pointer to the root object is passed back to the caller.

```
HRESULT Open(    BSTR              Connect,
   BSTR           User,
   BSTR           Password,
   long           fFlags,
   IRepositoryObject  **ppIRootObj
);
```

Parts

Connect

[in]
The ODBC connection string to be used for accessing the database server that hosts your repository.

User

[in]
The user name to use for identification to the database server.

Password

[in]
The password that matches the *User* input parameter.

fFlags

[in]
Flags that determine database access and caching behavior for the open repository. For more information, see ConnectionFlags Enumeration.

**ppIRootObj*

[out]
The **IRepositoryObject** interface pointer for the root **Repository** object of

the open repository.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepository Interface

IRepositoryObject Interface

IRepository::Refresh

This method refreshes all of the cached data for this repository instance. Only cached data that has not been changed by the current process is refreshed.

HRESULT Refresh(long *iMilliseconds***);**

Parts

iMilliseconds

[in]
This value is ignored.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepository Interface

IRepository2 Interface

This interface exposes methods for manipulating object-version identifiers, plus other methods inherited from the **IRepository** interface.

When to Use

Use the **IRepository2** interface to create and access repository databases. You can also use this interface to create and access repository objects in a repository database, and to manipulate repository object versions.

Methods

IUnknown Method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch Method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepository Method	Description
Create	Creates a repository database
CreateObject	Creates a new repository object
get_Object	Retrieves the **IRepositoryObject** interface pointer

	for a repository object
get_RootObject	Retrieves the **IRepositoryObject** interface pointer for the root repository object
get_Transaction	Retrieves the **IRepositoryTransaction** interface pointer for this repository instance
InternalIDToObjectID	Translates an internal identifier to an object identifier
ObjectIDToInternalID	Translates an object identifier to an internal identifier
Open	Opens a repository database
Refresh	Refreshes unchanged cached repository data

IRepository2 Method	Description
InternalIDToVersionID	Translates an internal object-version identifier to an object-version identifier
get_Version	Retrieves the **IRepositoryObjectVersion** interface pointer for a **Repository** object version
VersionIDToInternalID	Translates an object-version identifier to an internal object-version identifier
CreateObjectEx	Creates the first version of a new repository object of the specified type
get_MajorDBVersion	Retrieves the major version number of the first repository engine version that introduced this database format
get_MinorDBVersion	Retrieves the minor version number of the first repository engine version that introduced this database format

See Also

Repository Class

IRepository2::get_Version

Retrieves an **IRepositoryObjectVersion** interface pointer to the specified repository object version.

HRESULT get_Version(**VARIANT** *sVersionId,*
IRepositoryObjectVersion ***ppIReposVersion*
Long ***fFlags*
);

Parts

sVersionId

[in]
The object-version identifier of the repository object version to be retrieved.

ppIReposVersion

[out]
A pointer to the **IRepositoryObjectVersion** interface pointer for the repository object.

fFlag

[out]
Long integer specifying the resolution strategy used to select a specific version of the repository object. *fFlag* can be one of these.

Constant	Value	Description
SPECIFIEDVERSION	1	A specific version explicitly selected
LATESTVERSION	2	The version most recently created
VERSIONINWORKSPACE	3	The version in the workspace
PINNEDVERSION	4	A version that is pinned

Return Value

S_OK

The method completed successfully.

Error Values

The method failed to complete successfully.

See Also

IRepository Interface

Resolution Strategy for Objects and Object Versions

IRepository2::InternalIDToVersionID

This method translates an internal object-version identifier into an object-version identifier. Internal object-version identifiers are used by the repository engine to identify repository object versions.

HRESULT InternalIDToObjectID(**VARIANT** *sIntVersionID*,
VARIANT ***psExtVersionID*
);

Parts

sIntVersionID

[in]
The internal object-version identifier for the repository object.

psExtVersionID

[out]
A pointer to the object-version identifier for the repository object.

Return Value

S_OK

The method completed successfully.

Error Values

The method failed to complete successfully.

Remarks

Repository object-version identifiers are globally unique, and are the same across repositories for the same object. Repository internal object-version identifiers are unique only within the scope of a single repository.

The translation performed by this method is performed without loading the

object version in question. This enables database queries involving an object or relationship type identifier to be constructed without having to load the definition object.

See Also

IRepository::ObjectIDToInternalID

IRepository Interface

IRepository2::VersionIDToInternalID

This method translates an object-version identifier into an internal object-version identifier. Internal object-version identifiers are used by the repository engine to identify repository object versions.

HRESULT ObjectIDToInternalID(**VARIANT** *sExtVersionID*,
VARIANT ***psIntVersionID*
);

Parts

sExtVersionID

[in]
The object-version identifier for the repository object.

psIntVersionID

[out]
A pointer to the internal object-version identifier for the repository object.

Return Value

S_OK

The method completed successfully.

Error Values

The method failed to complete successfully.

Remarks

Repository object identifiers are globally unique, and are the same across repositories for the same object. Repository internal identifiers are unique only within the scope of a single repository.

The translation performed by this method is performed without loading the

object version in question. This enables database queries involving an object or relationship type identifier to be constructed without having to load the definition object.

See Also

IRepository::InternalIDToObjectID

IRepository Interface

IRepository2::CreateObjectEx

This method creates the first version of a new repository object of the specified type. The newly created version is assigned the object-version identifier passed in as an argument. This is unlike **IRepository::CreateObject()**, in which the repository engine assigns the version ID.

HRESULT IRepository2::CreateObjectEx(
VARIANT *sTypeID,*
VARIANT *sObjectID,*
VARIANT *ExtVersionID,*
IRepositoryObjectVersion ***ppRepObjVer*
);

Parts

sTypeID

[in]
The type of the new object; that is, the object identifier of the class definition to which the new object conforms.

sObjectID

[in]
The object identifier to be assigned to the new object. Pass in OBJID_NULL to have the repository engine assign an object identifier for you.

ExtVersionID

[in]
The object-version identifier (20 bytes) to be assigned to the first version of the object.

ppRepObjVer

[out]
The **IRepositoryObjectVersion** pointer to the newly created version.

Return Value

S_OK

The method completed successfully.

Error Values

The method failed to complete successfully.

See Also

IRepository::CreateObject

IRepository2::get_MajorDBVersion

This method retrieves the major version number of the first repository engine version that introduced this database format.

HRESULT get_MajorDBVersion(long **piMajorDBVersion*
);

Parts

piMajorDBVersion

[out, retval]
A pointer to the major version number of the first repository engine version that introduced this database format.

Return Value

S_OK

The method completed successfully.

Error Values

The method failed to complete successfully.

See Also

IRepository2::get_MinorDBVersion

IRepository2::get_MinorDBVersion

This method retrieves the minor version number of the first repository engine version that introduced this database format.

HRESULT get_MinorDBVersion(long **piMinorDBVersion*
);

Parts

piMinorDBVersion

[out, retval]
A pointer to the minor version number of the first repository engine version that introduced this database format.

Return Value

S_OK

The method completed successfully.

Error Values

The method failed to complete successfully.

See Also

IRepository2::get_MajorDBVersion

IRepositoryDispatch Interface

The **IRepositoryDispatch** interface is an enhanced **IDispatch** interface. In addition to all of the standard **IDispatch** methods, **IRepositoryDispatch** also provides access to the **Properties** collection. The **Properties** collection gives you a convenient mechanism to enumerate through all of the persistent properties and collections of an interface.

When you instantiate an Automation object that represents an object from your information model, and that object conforms to a class for which there is no custom implementation (in other words, you have not provided a software implementation of the class), the repository engine will provide an interface implementation for you. This interface implementation uses **IRepositoryDispatch** as its dispatch interface.

When to Use

Use the **IRepositoryDispatch** interface to access the properties and collections of a repository object, when no custom implementation is available.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface

GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	**Description**
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

Remarks

The repository engine will supply an interface implementation only if your interface is defined to inherit from **IDispatch** or **IRepositoryDispatch**.

See Also

IAnnotationalProps Interface

IClassDef Interface

ICollectionDef Interface

IInterfaceDef Interface

IInterfaceMember Interface

IManageReposTypeLib Interface

IPropertyDef Interface

IReposProperties Interface

IReposTypeInfo Interface

IReposTypeLib Interface

IReposRoot Interface

ISummaryInformation Interface

IRepositoryDispatch::get_Properties Method

This method retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides methods to access the **Properties** collection. The **Properties** collection gives you a convenient mechanism to enumerate through all of the persistent properties and collections of an interface.

HRESULT get_Properties(IReposProperties ***ppIReposProps* **);**

Parts

**ppIReposProps*

[out]
The **IReposProperties** interface pointer.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryDispatch Interface

IReposProperties Interface

IRepositoryErrorQueue Interface

Errors that occur while accessing a repository are saved on a repository error queue. A repository error queue is a collection of **REPOSERROR** structures. Individual elements on a repository error queue can be managed in much the same way that elements can be managed in other repository collections. This interface provides those management capabilities.

When to Use

Use the **IRepositoryErrorQueue** interface to manage the errors that belong to a particular repository error queue. With this interface, you can:

- Get a count of the number of error elements in the collection.
- Enumerate the elements in the collection.
- Insert and remove error elements to and from the collection.
- Retrieve one of the error elements in the collection.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IRepositoryErrorQueue method	Description
Count	Returns a count of the number of errors on the queue

Insert	Inserts a new error onto the error queue, in the specified location
Item	Retrieves the specified error from the error queue
Remove	Removes the specified error from the error queue
_NewEnum	Creates an enumerator object for the error queue

See Also

Error Handling Overview

Handling Errors

IReposErrorQueueHandler Interface

REPOSERROR Data Structure

IRepositoryErrorQueue::Count

This method returns the number of errors that are currently on the error queue.

ULONG Count(*void*);

Return Value

The number of error elements on the queue.

See Also

IRepositoryErrorQueue

REPOSERROR Data Structure

IRepositoryErrorQueue::Insert

This method inserts a new element into the error queue. The element can either be inserted at a specific location in the queue, or it can be appended to the end of the queue.

HRESULT Insert(**ULONG** *iIndex*,
REPOSERROR **psError*
);

Parameters

iIndex

[in]
The index of the location in the error queue where this element is to be inserted. To insert an element at the beginning of the error queue, set this parameter to one. To append the element to the end of the error queue, set this parameter to zero.

**psError*

[in]
The error information for the element to be inserted. The information from this structure is copied, and the copy is placed on the error queue.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryErrorQueue

REPOSERROR Data Structure

IRepositoryErrorQueue::Item

This method retrieves the specified element from the error queue. There are two variations of this method.

```
HRESULT Item(    ULONG          iIndex,
   REPOSERROR   *psError
);

HRESULT Item(
   ULONG        iIndex,
   IErrorInfo   **ppIErrorInfo
);
```

Parameters

iIndex

[in]
The index of the location in the error queue of the element to be retrieved.

**ppError*

[out]
The repository error information structure with information from the retrieved element.

**ppIErrInfoObj*

[out]
The interface pointer to an error information object for the retrieved element.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryErrorQueue

REPOSERROR Data Structure

IRepositoryErrorQueue::Remove

This method removes the specified element from the error queue.

HRESULT Remove(**ULONG** *iIndex*
);

Parameters

iIndex

[in]
The index of the location in the error queue of the element to be removed.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryErrorQueue

IRepositoryErrorQueue::_NewEnum

This method creates an enumeration object for the error queue. An interface pointer for the enumeration object is passed back to the caller.

**HRESULT _NewEnum(IEnumRepositoryErrors ***ppIEnum*
);

Parameters

**ppIEnum*

[out]
The interface pointer to the enumeration object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryErrorQueue

IEnumRepositoryErrors Interface

IRepositoryItem Interface

The **IRepositoryItem** interface contains methods that are common to both repository objects and relationships. It contains all of the general-purpose methods that are used to manage repository items.

When to Use

Use the **IRepositoryItem** interface to:

- Retrieve an item type or name.
- Obtain a lock on an item.
- Change the name of an item.
- Delete an item.
- Get a pointer to an alternate interface that the item exposes.
- Get the open repository instance through which the item is accessed.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description

GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	**Description**
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IRepositoryItem method	**Description**
Delete	Deletes a repository item
get_Interface	Retrieves an interface pointer to the specified item interface
get_Name	Retrieves the name associated with an item
get_Repository	Retrieves the **IRepository** interface pointer for an item's open repository instance
get_Type	Retrieves the type of an item
Lock	Locks the item
put_Name	Sets the name associated with an item

See Also

IRepositoryObject Interface

IRelationship Interface

IRepositoryItem::Delete

This method deletes a repository item.

HRESULT Delete(*void***);**

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

If the item to be deleted is a repository object version, this method fails unless the object version satisfies the basic requirements for object-version deletion.

Furthermore, if the object version is checked out to a workspace, the **Delete** method fails unless you invoke it from within the context of that workspace. If the object version satisfies both of these restrictions, the repository engine deletes it and its relationships, including any delete-propagating origin relationships. For each of these relationships, the repository engine considers performing one or more propagated deletions.

If the item to be deleted is an origin versioned relationship, this method fails unless the source object version satisfies the basic requirements for changing an object version.

If the source object version is changeable, the repository engine deletes the entire relationship (rather than merely removing one item from the **TargetVersions** collection of the relationship). That is, after this method finishes, no version of the destination object remains related to the origin object version. Then, if the relationship is a delete-propagating relationship, the repository engine considers performing one or more propagated deletions.

If the item to be deleted is a destination versioned relationship, the repository engine performs object-version resolution strategy to yield a single origin object version from the **TargetVersions** collection of the relationship. If that origin object version does not satisfy the basic requirements for changing an object version, this method fails.

If that origin object version is changeable, the repository engine removes it from the **TargetVersions** collection of the relationship. Then, if the relationship is a delete-propagating relationship, the repository engine considers performing one or more propagated deletions.

See Also

Propagating Deletes

IRepositoryItem Interface

Requirements for Changing an Object-Version

Requirements for Object-Version Deletion

Resolution Strategy for Objects and Object Versions

Workspace Context

IRepositoryItem::get_Interface

This method retrieves the interface pointer for an alternate interface that the item exposes. The specified interface must be an Automation interface; that is, it must support the methods of the **IDispatch** interface.

HRESULT get_Interface(**VARIANT** *whichInterface,*
IDispatch ***ppInterface*
);

Parameters

whichInterface

[in]
Specifies the interface you want to access. This parameter can be the name of the interface, the interface identifier, or the object identifier of the interface definition object in the repository.

**ppInterface*

[out]
The interface pointer for the Automation interface.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Some objects expose multiple interfaces. This method is provided as a mechanism so that the Automation programmer can easily access alternate interfaces for those cases where no type library is available for the class of the

item.

See Also

IRepositoryItem Interface

IRepositoryItem::get_Name

This method retrieves the name associated with a repository item. For repository relationships, this is the name defined by the relationship. For repository objects, this is either:

- The **Name** property of the **INamedObject** interface, if the object exposes that interface.

- The name defined by a relationship for which the object is the destination object.

HRESULT get_Name(BSTR **pName***);**

Parameters

**pName*

[out]
The name associated with the item.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

When you try to retrieve the name of an object version, the repository engine can look in several places for the name.

See Also

INamedObject Interface

IRepositoryItem Interface

IRepositoryItem::put_Name

Retrieving an Object Version's Name

IRepositoryItem::get_Repository

This method retrieves an **IRepository** interface pointer for the open repository instance or workspace through which this repository item was instantiated.

HRESULT get_Repository(IRepository ***ppIRepository*);**

Parameters

**ppIRepository*

[out]
The **IRepository** interface pointer for the open repository instance or workspace.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

The returned **IRepository** interface pointer can refer to either a **Repository** object or a **Workspace** object. If it refers to a **Workspace** object, you are manipulating the item within the context of that workspace. If it refers to a **Repository** object, you are manipulating the item not within the context of a workspace, but within the context of a shared repository instance.

See Also

IRepositoryItem Interface

IRepository Interface

IRepositoryItem::get_Type

This method obtains the object identifier of the repository definition object to which the repository item conforms. This is the type of the repository item.

HRESULT get_Type(VARIANT **psTypeId*);

Parameters

**psTypeId*

[out]
The object identifier of the definition object of the repository item.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryItem Interface

IRepositoryItem::Lock

This method locks a particular repository item. Locking the item prevents other processes from locking the item while you are working with it. The lock is released when you end the current transaction.

HRESULT Lock(*void***);**

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryItem Interface

IRepositoryItem::put_Name

This method changes one or more names of an item.

HRESULT put_Name(BSTR *Name***);**

Parameters

Name

[in]
The name to be associated with the item.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

The behavior of this method depends on whether the to-be-named item is an object version, an origin versioned relationship, or a destination versioned relationship. For more information, see Changing an Object Version's Name and Changing a Destination Relationship's Name.

Note In some circumstances, this method may attempt to change several names. For example, an object version that implements the **INamedObject** interface is the destination of three naming relationships. If you rename the object, this method will attempt to change four names. The method returns S_OK if any of the four attempts succeeds.

See Also

IRepositoryItem Interface

INamedObject Interface

IRepositoryItem::get_Name

IRepositoryObject Interface

The **IRepositoryObject** interface provides methods to manage repository objects.

When to Use

Use the **IRepositoryObject** interface to:

- Retrieve the object identifier or the internal identifier for a repository object.
- Retrieve a repository object type or name.
- Obtain a lock on a repository object.
- Change the name of a repository object.
- Refresh the cached image of a repository object.
- Delete a repository object.
- Get a pointer to an alternate interface that the object exposes.
- Get the open instance of the repository session object through which the object is accessed.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces

AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IRepositoryItem method	Description
Delete	Deletes a repository item
get_Interface	Retrieves an interface pointer to the specified item interface
get_Name	Retrieves the name associated with an item
get_Repository	Retrieves the **IRepository** interface pointer for an open repository instance of an item
get_Type	Retrieves the type of an item
Lock	Locks the item

put_Name	Sets the name associated with an item

IRepositoryObject method	Description
get_InternalID	Retrieves the internal identifier for a repository object
get_ObjectID	Retrieves the object identifier for a repository object
Refresh	Refreshes the cached image of the object

See Also

ClassDef Class

CollectionDef Class

InterfaceDef Class

MethodDef Class

PropertyDef Class

RelationshipDef Class

RepositoryObject Class

ReposRoot Class

ReposTypeLib Class

IRepositoryObject::get_InternalID

This method obtains the internal identifier for a repository object.

HRESULT get_InternalID(VARIANT **psInternalId*);

Parameters

**psInternalId*

[out]
The internal identifier of the current repository object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

This method retrieves the internal object identifier, not the internal object-version identifier.

See Also

IRepositoryObject Interface

IRepositoryObject::get_ObjectID

This method retrieves the object identifier for a repository object.

HRESULT get_ObjectID(VARIANT **psObjectId*);

Parameters

**psObjectId*

[out]
The object identifier of the current repository object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

This method retrieves the object identifier, not the object-version identifier.

See Also

IRepositoryObject Interface

IRepositoryObject::Refresh

This method refreshes the cached image of a particular repository object version. Only unchanged cache data is refreshed.

HRESULT Refresh(long *iMilliseconds***);**

Parameters

iMilliseconds

[in]
This value is ignored.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryItem Interface

IRepositoryObject Interface

IRepositoryObject2 Interface

This interface is implemented by all repository objects. It inherits the properties, methods, and collections of **IRepositoryObject**.

When to Use

Use **IRepositoryObject2** to retrieve information about an object, without having to make a roundtrip to the repository database.

You can also use the **Properties** property on this interface to explicitly define which interfaces and collections to work with. If you are working with inherited interfaces, a collection on the derived interface may assume the same name as the base collection on the base interface. When both collections share the same name, using the **IRepositoryObject** interface can return either collection, even though the collections may be fundamentally different in other ways. Using **IRepositoryObject2** instead of IRepositoryObject allows you to explicitly identify the collection you want, eliminating the possibility that the repository engine will return the wrong collection.

The following example illustrates how to work with same name collections on specific interfaces using **IRepositoryObject2**. In this case, a base interface and an inherited interface each have a collection named **Contains**.

```
Dim MyObject as IRepositoryObject2
MyObject.Properties(IBaseIFace + ".Contains").Value.Add oFile1
MyObject.Properties(IBaseIFace + ".Contains").Value.Add oFile2
MyObject.Properties(IInheritIFace + ".Contains").Value.Add oFile1
```

Properties

Property	Description
ClassName	Contains the name of the class where the property is used
ClassType	Returns the **ClassDef** object that represents this property

Properties	Allows the application to obtain all properties for the object

See Also

ClassDef Object

IRepositoryObject Interface

IRepositoryObject2 ClassName Property

The **ClassName** property identifies the name of the class where the property is used.

Syntax

HRESULT ClassName (

BSTR **pClassName*

);

Dispatch Identifier: DISPID_IRepositoryObject2_ClassName = 31

Parameters

**pClassName*

[out, retval]
A pointer to the string that contains the name of the class.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryObject2 Interface

IRepositoryObject2 ClassType Property

The **ClassType** property returns an in-memory pointer to a **ClassDef** object. An object type is identified by its object identifier.

Syntax

HRESULT ClassDef (

VARIANT *plClassDef

);

Dispatch Identifier: DISPID_IRepositoryObject2_ClassType = 30

Parameters

**plClassDef*

[out]
A pointer to the **ClassDef** object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryObject2 Interface

Repository Identifiers

IRepositoryObject2 Properties Property

The **Properties** property allows the application to obtain all properties for the object, regardless of which interface they are defined on.

Syntax

HRESULT Size (

VARIANT **pIReposProps*

);

Dispatch Identifier: DISPID_IRepositoryObject2_Properties = 1000

Parameters

**pIReposProps*

[out]

Return Value

S_OK

The method completed successfully.

Error Values

None.

See Also

IRepositoryObject2 Interface

IRepositoryObjectStorage Interface

The **IRepositoryObjectStorage** interface initializes the memory image for a repository object. New repository objects are initialized as empty objects. For existing repository objects, the state of the object is retrieved from the repository database.

When to Use

The **IRepositoryObjectStorage** interface is used by the repository engine to materialize repository objects in memory. It is not intended for use by repository applications.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers.
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an Automation object.

IRepositoryObjectStorage method	Description
get_PropertyInterface	Retrieves an **IRepositoryDispatch** interface pointer for accessing the persistent members of one of the supported interfaces of an item.
InitNew	Initializes memory for a new repository object.
Load	Initializes memory for an existing repository object.

See Also

ClassDef Class

CollectionDef Class

InterfaceDef Class

MethodDef Class

PropertyDef Class

RelationshipDef Class

RepositoryObject Class

ReposRoot Class

ReposTypeLib Class

IRepositoryObjectStorage::get_PropertyInterface

This method retrieves an **IRepositoryDispatch** interface pointer for accessing the persistent members of one of the object's supported Automation interfaces.

The **IRepositoryDispatch** interface can be used to get and set member values for the interface specified by the *InterfaceId* input parameter. The interface must be one that is exposed by this object.

HRESULT get_PropertyInterface(**VARIANT** *InterfaceId*,
IRepositoryDispatch ***ppInterface*
);

Parameters

InterfaceId

[in]
The interface identifier of the interface whose properties are to be accessed.

**ppInterface*

[out]
The **IRepositoryDispatch** interface pointer.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

IRepositoryDispatch does not perform any parameter validation, and it cannot be used to access custom methods or nonpersistent properties. It is intended for

use by custom class implementers.

See Also

IRepositoryDispatch Interface

IRepositoryObjectStorage Interface

IRepositoryObjectStorage::InitNew

The repository engine uses this method to initialize a new repository object in memory.

HRESULT InitNew(**IRepository** **pIRepository,*
INTID *sInternalId*
);

Parameters

**pIRepository*

[in]
The repository that contains this object.

sInternalId

[in]
The internal identifier for the new object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepository Interface

IRepositoryObjectStorage Interface

IRepositoryObjectStorage::Load

The repository engine uses this method to load the state information of a repository object into memory from the repository database.

HRESULT Load(**IRepository** **pIRepository,*
INTID *sInternalId*
);

Parameters

**pIRepository*

[in]
The repository that contains this object.

sInternalId

[in]
The internal identifier for the repository object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepository Interface

IRepositoryObjectStorage Interface

IRepositoryObjectVersion Interface

A repository object version is a particular version of a repository object. Each version of an object can differ from other versions of that object in its property values and collections.

When to Use

Use the **IRepositoryObjectVersion** interface to manipulate any version of an object, including the original version established with **IRepository::CreateObject**.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IRepositoryItem method	Description
Delete	Deletes a repository item
Get_Interface	Retrieves an interface pointer to the specified item interface
Get_Name	Retrieves the name associated with an item
Get_Repository	Retrieves the **IRepository** interface pointer for an item's open Repository instance
Get_Type	Retrieves the type of an item
Lock	Locks the item
Put_Name	Sets the name associated with an item

IRepositoryObject method	Description
get_InternalID	Retrieves the internal identifier for a **Repository** object
get_ObjectID	Retrieves the object identifier for a **Repository** object

IRepositoryObjectVersion method	Description
CreateVersion	Creates a new version of an object as a successor to the current object version
FreezeVersion	Disallows further modification of the (nonannotational) property values or origin collections of the current object version

get_IsFrozen	Determines whether the current object version is frozen
get_ObjectVersions	Retrieves an interface pointer to the collection of all versions of the current object
get_PredecessorCreationVersion	Retrieves an interface pointer to the predecessor object version from which the current object version was created
get_PredecessorVersions	Retrieves an interface pointer to the collection of all predecessor versions of the current object version
get_ResolutionType	Returns an indication of which resolution technique the repository engine used in returning the particular version of the current object
get_SuccessorVersions	Retrieves an interface pointer to the collection of all successor versions of the current object version
get_VersionID	Retrieves the object-version identifier of the current object version
get_VersionInternalID	Retrieves the internal object-version identifier of the current object version
MergeVersion	Changes the current object version by combining its property values and collections with another object version

See Also

IRepository::CreateObject

RepositoryObjectVersion Class

Versioning Objects

IRepositoryObjectVersion::CreateVersion

This method creates a new version of an object as a successor to the current object version.

HRESULT CreateVersion(**VARIANT** *sVersionID*
IRepositoryObjectVersion ***ppCreatedVersion*
);

Parameters

sVersionID

[in]
The object-version identifier to be assigned to the new object version. If you want the repository engine to assign an object-version identifier, use a value of EXTVERSIONID_NULL.

***ppCreatedVersion*

[out]
The **IRepositoryObjectVersion** interface pointer for the newly created object version.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

The current object version must be frozen.

The repository engine creates the new version as unfrozen. Its property values

are identical to the current object version's property values.

For each of the predecessor version's origin relationship collections, the repository engine takes this action:

- If the corresponding relationship type has the COLLECTION_NEWORGVERSIONSPARTICIPATE flag set, the repository engine copies the collection to the newly created version.

- If the corresponding relationship type does not have the COLLECTION_NEWORGVERSIONSPARTICIPATE flag set, the repository engine does not copy the collection to the new version.

You cannot invoke this method while operating in a workspace.

See Also

IRepositoryObjectVersion Interface

IRepositoryObjectVersion::FreezeVersion

This method allows further modification of the property values or origin collections of the current object version. You cannot use this method for annotational properties.

HRESULT FreezeVersion(*void***);**

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

After you freeze an object version, you cannot change its nonannotational property values. However, you can change the value of any of its annotational properties.

After you freeze an object, you cannot enlarge or shrink any of its origin collections. You cannot pin, unpin, rename, or resequence any of the items in any of its origin collections. Furthermore, you cannot change any of the individual versioned relationships in any of the origin collections. That is, you cannot enlarge an item's set of target object versions; pin a target object version, and you cannot unpin the pinned target object version.

If you call this method for an item currently checked out to any workspace (including the workspace in which you are working), it returns an error.

If the to-be-frozen object version includes any nonnull origin collection whose corresponding collection type has the COLLECTION_REQUIRESFREEZE flag set, and that nonnull collection includes an item whose **TargetVersions** collection contains a nonfrozen object version, the method fails.

See Also

IRepositoryObjectVersion Interface

IRepositoryObjectVersion::get_IsFrozen

This method determines whether the current object version is frozen.

HRESULT get_IsFrozen(**VARIANT_BOOL** **pbFrozen*
);

Parameters

**pbFrozen*

[out]
TRUE if the current object version is frozen; FALSE if it is not frozen.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryObjectVersion Interface

IRepositoryObjectVersion::get_ObjectVersions

This method retrieves an interface pointer to the collection of all versions of the current object.

HRESULT get_ObjectVersions(**IVersionCol** ***ppObjVersions*
);

Parameters

***ppObjVersions*

[out]
The **IVersionCol** interface pointer for the collection of object versions.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Within the returned collection, the repository engine sequences the items in order of their creation, with the oldest object version first.

You cannot modify the collection. To add a new object version, use **IRepositoryObjectVersion::CreateVersion**.

See Also

IRepositoryObjectVersion Interface

IRepositoryObjectVersion::CreateVersion

IRepositoryObjectVersion::get_PredecessorCreationV

This method retrieves an interface pointer to the predecessor object version from which the current object version was created.

HRESULT get_PredecessorCreationVersion(IRepositoryObjectVersion ***ppPredCreationVersion*
);

Parameters

***ppPredCreationVersion*

[out]
The **IRepositoryObjectVersion** interface pointer for the object version from which the current object version was created.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

If you invoke this method for the first version of an object, it returns an error.

See Also

IRepositoryObjectVersion Interface

IRepositoryObjectVersion::get_PredecessorVersions

This method retrieves an interface pointer to the collection of all predecessor versions of the current object version.

HRESULT get_PredecessorVersions(**IVersionCol** ***ppPredVersions* **);**

Parameters

***ppPredVersions*

[out]
The **IVersionCol** interface pointer for the collection of predecessor object versions.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

This method returns only the immediate predecessors of the current object version.

If you invoke this method for the first version of an object, it returns an empty collection.

Within the returned collection, the repository engine sequences the items in order of their creation, with the oldest object version first.

You cannot modify the collection.

See Also

IRepositoryObjectVersion Interface

IRepositoryObjectVersion::get_ResolutionType

This method returns an indication of the resolution technique by which the repository engine chose to give you a reference to the current version (rather than a reference to any other version of the same object).

HRESULT get_ResolutionType(LONG **pResolutionType*
);

Parameters

**pResolutionType*

[out]
One of four possible values. The following table lists the values.

Constant	Value	Description
SPECIFIEDVERSION	1	Indicates that a specific object version was selected
LATESTVERSION	2	Indicates that the most recently created object version was selected
VERSIONINWORKSPACE	3	Indicates that the version in the workspace was selected
PINNEDVERSION	4	Indicates that a pinned object version was selected

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

The repository engine automatically sets the value of the **ResolutionType** property whenever you retrieve an object version.

See Also

IRepositoryObjectVersion Interface

IRepositoryObjectVersion::get_SuccessorVersions

This method retrieves an interface pointer to the collection of all successor versions of the current object version.

HRESULT get_SuccessorVersions(**IVersionCol** ****ppSuccVersions*
);

Parameters

****ppTargetVersions***

[out]
The **IVersionCol** interface pointer for the collection of successor object versions.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

This method returns only the immediate successors of the current object version.

If the current object version has no successors, this method returns an empty collection.

Within the returned collection, the repository engine sequences the items in order of their creation, with the oldest object version first.

You cannot modify the collection. To add a new successor to the current object version, use **IRepositoryObjectVersion::CreateVersion**.

See Also

IRepositoryObjectVersion Interface

IRepositoryObjectVersion::get_VersionID

This method retrieves the object-version identifier of the current object version.

HRESULT get_VersionID(**VARIANT** **psVersionID*
);

Parameters

**psVersionID*

[out]
The object-version identifier of the current object version.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryObjectVersion Interface

IRepositoryObjectVersion::get_VersionInternalID

This method retrieves the internal object-version identifier of the current object version.

HRESULT get_VersionInternalID(**VARIANT** **psVersionID*
);

Parameters

**psVersionID*

[out]
The internal object-version identifier of the current object version.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryObjectVersion Interface

IRepositoryObjectVersion::MergeVersion

This method changes the current object version by combining its property values and origin collections with the property values and origin collections of another version of the same object.

HRESULT MergeVersion(**IRepositoryObjectVersion** **pOtherVersion*
long *fFlags*
);

Parameters

**pOtherVersion*

[in]
The **IRepositoryObjectVersion** interface pointer for the predecessor of the merge. That is, the object version whose property values and collections should be merged into the current version.

fFlags

[in]
Long integer specifying which object version is the primary and which is secondary in the merge.

Constant	Value	Description
PRIMARY	1	The predecessor object is primary and the current object is secondary.
SECONDARY	2	The predecessor object is secondary and the current object is primary.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Relationships are inserted at the end of the sequenced collection.

The two object versions must be versions of the same object.

The current object version must be unfrozen. The other object version must be frozen.

MergeVersion compares the property values and collections of each object version to a third version, called the Basis Version.

The repository engine considers one of the two to-be-merged object versions as the primary version, and the other to be the secondary version, according to the value of *fFlags* you supply. During the merge, the repository engine considers each property and origin collection type in turn. For each property, **MergeVersion** uses this rule:

- If the primary version differs from the Basis Version, the repository engine uses the property value from the primary version. If only the secondary version differs from the Basis Version, the repository engine uses the property value from the secondary version. If neither version differs from the Basis Version, the repository engine leaves the property value in the current version unchanged.

For each origin collection type whose COLLECTION_MERGEWHOLE flag is set, **MergeVersion** uses this rule:

- If the primary version's collection differs from the Basis Version's collection, the repository engine uses the collection from the primary version. If only the secondary version's collection differs from the Basis Version's, the repository engine uses the collection from the secondary version. If neither version differs from the Basis Version, the repository engine leaves the property value in the current version unchanged.

For each origin collection type whose COLLECTION_MERGEWHOLE flag is not set, **MergeVersion** combines the items in the two collections as follows:

1. **MergeVersion** includes in the resulting collection each item in the Basis Version not changed in or deleted from either the primary version or secondary version.

2. **MergeVersion** includes in the resulting collection each item in the primary version's collection that differs from the Basis Version.

3. **MergeVersion** includes in the resulting collection each item in the secondary version's collection that differs from the Basis Version, provided the corresponding items in the primary version and the Basis Version do not differ from each other.

The resulting collection can exclude some items found in the basis object version's collection. For example, if the primary version's collection excludes the item, the resulting collection excludes the item. Similarly, if the primary version's collection includes an item that is identical to an item in the Basis Version's collection, but the secondary object version excludes the item, the resulting collection excludes the item.

See Also

IRepositoryObjectVersion Interface

IRepositoryODBC Interface

The repository engine stores information in an SQL database. The repository engine connects to the database server through an ODBC connection. The **IRepositoryODBC** interface provides you with access to the database through the same (or a similar) ODBC connection.

Care should be taken when accessing the repository database directly, especially when sharing the repository ODBC connection. Specific restrictions are defined in the detailed information for each interface method. Directly accessing the repository database in a read-only manner is generally considered safe; however, if you tune your repository application to be dependent upon specific features of your database server, you limit the portability of your application.

When to Use

Use the **IRepositoryODBC** interface to obtain or release an ODBC connection handle, or to retrieve the ODBC connection string used by the repository engine.

To obtain a pointer to this interface, use the **IRepository::QueryInterface** method.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers.
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an

	interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an Automation object.

IRepositoryODBC method	**Description**
ExecuteQuery	Executes the specified SQL query against the repository database.
FreeConnection	Releases an ODBC connection handle.
get_ConnectionString	Retrieves the ODBC connection string that the repository engine uses to obtain an ODBC connection.
GetNewConnection	Obtains a new ODBC connection handle using the same connection settings that the repository engine is using to access the repository database.
get_ReposConnection	Retrieves the ODBC connection handle that the repository engine is using to access the repository database.

See Also

IRepositoryODBC2 Interface

Repository Class

IRepositoryODBC::ExecuteQuery

This method executes the specified SQL query against the repository database, and returns a collection of repository objects. The columns that are returned by the query must be either the internal identifier (**IntID**) column, or a combination of the internal identifier and the type identifier (**IntID**, **TypeID**) columns of the **RTblVersions** table.

HRESULT ExecuteQuery(BSTR *queryString*, **IObjectCol** ***ppICol*);**

Parameters

queryString

[in]
The SQL query, or the name of a stored procedure.

***ppICol*

[out]
The collection of objects that meet the selection criteria of the SQL query.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryODBC Interface

RTblVersions SQL Table

IRepositoryODBC::FreeConnection

This method frees an ODBC connection handle.

HRESULT FreeConnection(long *Hdbc***);**

Parameters

Hdbc

[in]
The ODBC connection handle to be released.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Use this method to free the handle obtained through either the **get_ReposConnection** method or the **GetNewConnection** method before releasing an open repository instance.

See Also

IRepositoryODBC Interface

IRepositoryODBC::GetNewConnection

IRepositoryODBC::get_ReposConnection

IRepositoryODBC::get_ConnectionString

This method retrieves the ODBC connection string that the repository engine uses to obtain an ODBC connection to the repository database.

HRESULT get_ConnectionString(BSTR *szString***);**

Parameters

szString

[out]
The ODBC connection string.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

The ODBC connection string can contain user identification and password information. Take care to protect this information from exposure to unauthorized access.

See Also

IRepositoryODBC Interface

IRepositoryODBC::GetNewConnection

This method obtains a new ODBC connection handle using the same ODBC connection string that the repository engine is using to access the repository database. Using a new ODBC connection handle isolates you from changes made by the repository engine.

HRESULT GetNewConnection(long **pHdbc*);

Parameters

**pHdbc*

[out]
A new ODBC connection handle.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Be sure to free the handle obtained through this method before releasing your open repository instance. To free the connection handle, use the **FreeConnection** method.

See Also

IRepositoryODBC Interface

IRepositoryODBC::FreeConnection

IRepositoryODBC::get_ReposConnection

This method retrieves the ODBC connection handle that the repository engine is using to access the repository database.

HRESULT get_ReposConnection(long *pHdbc);

Parameters

**pHdbc*

[out]
A copy of the repository engine ODBC connection handle.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

If you use the repository engine ODBC connection handle, you are not isolated from changes made by the repository engine. For example, uncommitted changes made by the repository engine will be visible to your application.

When using the ODBC connection handle of the repository engine, you must not change the state of the handle in a way that is incompatible with the repository engine. Specifically, do not:

- Change any ODBC connection options.
- Perform any accesses concurrent with repository method invocations.

- Directly commit or roll back a database transaction. The **IRepositoryTransaction** interface must always be used to manage transactions.

Be sure to free the handle obtained through this method before releasing your open repository instance. To free the connection handle, use the **FreeConnection** method.

See Also

IRepositoryODBC Interface

IRepositoryODBC::FreeConnection

IRepositoryODBC2 Interface

This interface exposes methods that enable you to set or get options for retrieving object collections asynchronously, plus other methods inherited from the **IRepositoryODBC** interface.

When to Use

Use the **IRepositoryODBC2** interface to obtain or release an ODBC connection handle, or to retrieve the ODBC connection string used by the repository engine. It is also used to set or get options when loading object collections asynchronously.

To obtain a pointer to this interface, use the **IRepository::QueryInterface** method.

Methods

IUnknown Method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch Method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods

	exposed by an Automation object

IRepositoryODBC Method	Description
ExecuteQuery	Executes the specified SQL query against the repository database
FreeConnection	Releases an ODBC connection handle
get_ConnectionString	Retrieves the ODBC connection string that the repository engine uses to obtain an ODBC connection
GetNewConnection	Obtains a new ODBC connection handle using the same connection settings that the repository engine is using to access the repository database
get_ReposConnection	Retrieves the ODBC connection handle that the repository engine is using to access the repository database

IRepositoryODBC2 Method	Description
GetOption	Obtains the value of the load option
SetOption	Sets the option for loading the collection

See Also

IRepositoryODBC Interface

Repository Class

IRepositoryODBC2::GetOption

This method obtains the value of the load option.

Syntax

HRESULT GetOption(**long** *iOption*,
VARIANT **psValue*);**

Parameters

iOption

[in]
RODBC_ASYNCH.

psValue

[out, retval]
VARIANT_TRUE or VARIANT_FALSE, depending upon whether the RODBC_ASYNCH option has been set.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryODBC Interface

IRepositoryODBC2 Interface

IRepositoryODBC2::SetOption

IRepositoryODBC2::SetOption

This method sets the option for loading the collection. The RODBC_ASYNCH flag can be set only if the underlying database system supports asynchronous operations.

Syntax

HRESULT SetOption(
long *iOption*,
VARIANT *sValue*);

Parameters

iOption

[in]
Specifies the option to set. You can set either RODBC_ASYNCH or RODBC_RESET_OPTIONS.

RODBC_ASYNCH takes an *sValue*.

RODBC_RESET_OPTIONS does not take an *sValue*.

sValue

[in]
TRUE sets the asynchronous mode of load.

FALSE clears the asynchronous mode of load.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryODBC Interface

IRepositoryODBC2 Interface

IRepositoryODBC2::GetOption

IRepositoryTransaction Interface

The repository engine supports transactional processing. Repository engine methods that are reading data from a repository database may be executed outside of a transaction, but methods that write data must be bracketed within a transaction. Only one transaction can be active at a time for each opened repository instance. Nesting of **Begin** or **Commit** method invocations is permitted, but no actual nesting of transactions occurs.

When to Use

Use the **IRepositoryTransaction** interface to begin, commit, or cancel a repository transaction. You can also use this interface to retrieve the information about the transactional state of an open repository instance, and to set transaction options.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers.
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an Automation object.

IRepositoryTransaction method	Description
Abort	Cancels a currently active transaction.
Begin	Begins a new transaction.
Commit	Commits an active transaction.
Flush	Flushes uncommitted changes to the repository database.
GetOption	Retrieves a transaction option.
get_Status	Indicates whether there is a currently active transaction.
SetOption	Sets a transaction option.

See Also

Managing Transactions and Threads

Repository Class

IRepositoryTransaction::Abort

This method cancels the currently active transaction for an open repository. All updates made during the transaction are undone. The nested transaction count is set to zero.

Syntax

HRESULT Abort(*void***);**

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryTransaction Interface

IRepositoryTransaction::Begin

This method increments the nested transaction count by one. If there is no active transaction, this method begins a transaction for the open repository instance.

Syntax

HRESULT Begin(*void***);**

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryTransaction Interface

IRepositoryTransaction::Commit

This method decrements the nested transaction count for an open repository instance. If the currently active transaction is not nested, all changes made to repository data within the transaction are committed to the repository database. A transaction is not nested if the nested transaction count equals one.

Syntax

HRESULT Commit(*void***);**

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryTransaction Interface

IRepositoryTransaction::Flush

This method flushes cached changes to the repository database.

Unless you have set the exclusive-write-through-mode transaction option, changes that you make within the scope of a transaction are cached, and they are not written to the database until the transaction is committed. If a concurrent SQL query is run against the repository database, the result of the query will not reflect the uncommitted changes. Typically, this is the desired behavior.

If your repository application must be able to see uncommitted changes in SQL queries, you can use the **Flush** method to write uncommitted changes to the repository database. All changes made within the scope of the current transaction are flushed. Flushing uncommitted changes does not affect your ability to cancel a transaction through the **Abort** method.

Syntax

HRESULT Flush(void);

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryTransaction Interface

IRepositoryTransaction::Abort

IRepositoryTransaction::GetOption

This method retrieves the value of a transaction option for an open Microsoft® SQL Server™ 2000 Meta Data Services instance.

Syntax

```
HRESULT GetOption(    long        iOption,
   VARIANT *psValue
);
```

Parameters

iOption

[in]
The transaction option to retrieve. For more information about valid values and their meanings, see TransactionFlags Enumeration.

**psValue*

[out]
The value of the specified transaction option.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryTransaction Interface

IRepositoryTransaction::Abort

IRepositoryTransaction::get_Status

This method determines whether there is a currently active transaction.

Syntax

HRESULT get_Status(long **piStatus*);

Parameters

**piStatus*

[out]
The current transaction status. If the value is zero, no transaction is active. If the value is nonzero, a transaction is active.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

A transaction is considered active until the **Commit** method has successfully executed and the nested transaction count has been decremented to zero. Depending on the data-flushing capabilities of the underlying database server, the data associated with a committed transaction may or may not be written to the physical storage device when the **Commit** method returns control to its caller.

See Also

IRepositoryTransaction Interface

IRepositoryTransaction::Commit

IRepositoryTransaction::SetOption

This method sets one of the transaction options for an open Microsoft® SQL Server™ 2000 Meta Data Services instance. You cannot set a transaction option while a transaction is active.

Syntax

HRESULT SetOption(**ULONG** *iOption*,
VARIANT *sValue*
);

Parameters

iOption

[in]
The transaction option to set. For more information about valid values and their meanings, see TransactionFlags Enumeration.

sValue

[in]
The value of the specified transaction option.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRepositoryTransaction Interface

IRepositoryTransaction2 Interface

This interface supports distributed transactions on Microsoft® SQL Server™ 6.5, SQL Server 7.0, and SQL Server 2000. However, the operating system must be Microsoft Windows® 2000. This feature ensures that the distributed transaction is atomic; that is, it either commits at all resource managers or aborts at all of them.

When to Use

Use the **IRepositoryTransaction2** interface to begin, commit, or abort a distributed repository transaction.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryTransaction	

method	Description
Abort	Cancels a currently active transaction
Begin	Begins a new transaction
Commit	Commits an active transaction
Flush	Stores uncommitted changes to the repository database
GetOption	Retrieves a transaction option
get_Status	Indicates whether there is a currently active transaction
SetOption	Sets a transaction option

IRepositoryTransaction2 method	Description
get_DTCTransaction	Retrieves a pointer to the active Microsoft Distributed Transaction Coordinator (MS DTC) transaction

See Also

Integration with Distributed Transaction Coordinator

IRepositoryTransaction Interface

Managing Transactions and Threads

IRepositoryTransaction2::get_DTCTransaction

This method returns a pointer to the active Microsoft® Distributed Transaction Coordinator (MS DTC) transaction. If it is not running inside an MS DTC transaction, this method returns NULL. It allows the caller to enlist other resource managers in the same transaction that is being used to access the repository database. For example, you can use the pointer as a parameter of **IJoinTransaction::JoinTransaction** to instantiate another repository session or an OLE DB provider, which you then can enlist in the MS DTC transaction. Similarly, you can use the pointer in an aggregation wrapper that accesses another resource manager. The wrapped object can get the MS DTC transaction in the transaction that accessed it, so it can enlist the other resource manager in the same transaction.

Syntax

HRESULT get_DTCTransaction (VARIANT **UnKTransaction*
);

Parameters

UnKTransaction

[out]
A pointer to the **IUnknown** interface on the transaction object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

For more information about the **ITransactionJoin** interface, see the MSDN® Library.

See Also

IRepositoryTransaction Interface

IRepositoryTransaction2 Interface

IReposOptions Interface

This interface exposes methods for getting, setting, or resetting engine options. These options are used to change the default engine behavior and to enable the application to override some of the engine optimizations.

When to use

Use the **IReposOptions** interface to:

- Let the application exercise some control over the execution.
- Change the engine's default parameters.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

Method	Description
GetOption	Retrieves the value of an engine's option
SetOption	Sets the value of an engine's option
ResetOptions	Resets the engine's options

See Also

IReposOptions Options Table

IReposOptions::GetOption

Use this method to retrieve the value of an engine's option.

Syntax

HRESULT GetOption(
  BSTR *OptionName*,
  VARIANT **OptionValue*
);

Parameters

OptionName

[in]
One of the string identifiers described in the **IReposOptions** Options Table. For more information about option values and descriptions, see IRepoOptions Options Table.

OptionValue

[out, retval]
A pointer to one of the values shown in the options table.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposOptions::SetOption

IReposOptions::ResetOptions

IReposOptions::SetOption

Use this method to set the value of an engine's option.

Syntax

HRESULT SetOption(
  BSTR *OptionName*,
  VARIANT *OptionValue*
);

Parameters

OptionName

[in]
One of the string identifiers shown in the **IReposOptions** Options Table. For more information about option values and descriptions, see IRepoOptions Options Table.

OptionValue

[out, retval]
One of the values described in the options table.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposOptions::GetOption

IReposOptions::ResetOptions

IReposOptions::ResetOptions

Use this method to reset the values to the repository engine default option values.

Syntax

HRESULT ResetOptions();

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposOptions::GetOption

IReposOptions::SetOption

IReposOptions Options Table

The following table shows the options that can be used as parameters for the SetOption and GetOption methods.

OptionName	OptionValue Default	Type	Description
OPT_RELEASENOREFROW_MODE	FALSE	**boolean**	When set to TRUE, this option causes cache rows to be immediately released when the reference count goes to zero. This is the same as setting the age-out time to zero, except that the value will not be used by the background thread.
OPT_AGEOUT	1000	**ulong**	The number of milliseconds after a repository object pointer gets released until it is outdated. 0xFFFFFFFF (-1) indicates that the object never ages out.
OPT_TIM_AGEOUT	0xFFFFFFFF	**ulong**	The number of milliseconds after a pointer to a type information model object gets released until it is outdated. 0xFFFFFFFF (-1)

			indicates that the object never ages out.
OPT_PRELOAD_AGEOUT	60000	**ulong**	The number of milliseconds after an object is prefetched until it is marked as outdated and ready to be cleaned up by the background thread. 0xFFFFFFFF (-1) indicates that the object never ages out.
OPT_ATOMICOP_MODE	FALSE	**boolean**	Indicates whether atomic operations are enabled. A value of TRUE indicates that atomic operations are enabled, while the value of FALSE indicates that atomic operations are disabled.
OPT_PRELOAD_COL _MODE	0	**long**	The number indicates the maximum number of objects in a collection C such that when a destination object collection D is accessed, for any object in C, then the repository engine will preload D for all objects in C. Zero means this preloading is disabled.
OPT_EXPORT_MODE	FALSE	**boolean**	Loads all origin

			collections on an object at object creation.
OPT_NOPROPERTYPREFETCH_MODE	FALSE	**boolean**	If this option is set to TRUE, it does not prefetch properties on objects in any of the collections.
OPT_LCID	1033 (US English)	**ulong**	The engine allows its clients to change the locale at run-time. This is done by using the **SetOption** method and setting LCID (locale identifier) as value.

See Also

IReposOptions::ResetOptions

IReposProperties Interface

The **IReposProperties** interface provides access to the **Properties** collection. The **Properties** collection gives you a convenient mechanism for enumerating through all of the persistent properties and collections of an interface, when you do not already know the names of all of the interface members.

When you instantiate an Automation object that represents an object from your information model, and that object conforms to a class for which there is no custom implementation (in other words, you have provided no software implementation of the class), the repository engine provides an interface implementation for you. This interface implementation uses **IRepositoryDispatch** as its dispatch interface. This dispatch interface contains one additional method, the **get_Properties** method, which returns an **IReposProperties** interface pointer.

This support enables the Automation programmer to use syntax like the following:

```
Dim firstProperty As ReposProperty
Set firstProperty = repObject.Properties(1)
```

The second statement is resolved in the following way:

- In this example, *repObject* is an Automation instantiation of a repository object where the default implementation has been used.

- The **Properties** term is the Automation level name for the **get_Properties** method that is supplied by the **IRepositoryDispatch** dispatch interface.

- The **get_Properties** method returns the interface pointer to the **IReposProperties** interface.

- The default method of the **IReposProperties** interface is the **get_Item**

method, which returns an **IReposProperty** interface pointer for the specified property object in the **Properties** collection.

At this point, the Automation programmer has access to the first property in the collection through the *firstProperty* object variable.

When to Use

Use the **IReposProperties** interface to access the properties and collections of a repository object when no custom implementation is available, and you do not already know what members are exposed by the object's interface. With the **IReposProperties** interface, you can:

- Get a count of the number of members in the collection.
- Retrieve an **IReposProperty** interface pointer to one of the members in the collection.
- Enumerate the members in the collection.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface

GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IReposProperties method	Description
get_Count	Retrieves the count of the number of members in the collection
get_Item	Retrieves the **IReposProperty** interface pointer for the specified member of the collection
get_Type	Retrieves the type of the interface to which these properties are attached
_NewEnum	Retrieves a standard Automation enumeration interface pointer for the collection

Remarks

Only persistent members (that is, members that are stored in the repository) are represented in the **Properties** collection.

See Also

ReposProperties Class

IRepositoryDispatch Interface

IReposProperty

IReposProperties::get_Count

This method retrieves a count of the number of persistent members (properties and collections) that are in the **Properties** collection.

HRESULT get_Count(long **piCount***);**

Parameters

**piCount*

[out]
The number of members in the collection.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposProperties Interface

IReposProperties::get_Item

This method retrieves the specified member from the **Properties** collection.

HRESULT get_Item(**VARIANT** *sItem*,
IReposProperty ***ppIReposProperty*
);

Parameters

sItem

[in]
Identifies the item to be retrieved from the collection. This parameter can be either the index or the name of the member, or the object identifier of the property definition object for the member.

**ppIReposProperty*

[out]
The **IReposProperty** interface pointer for the specified collection member.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposProperties Interface

IReposProperty Interface

IReposProperties::get_Type

This method retrieves the object identifier for the interface definition of the interface to which these properties are attached. This object identifier is referred to as the type of the interface.

HRESULT get_Type(VARIANT **psTypeId*__);__

Parameters

**psTypeId [out]*

The object identifier for the interface definition object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposProperties Interface

IReposProperty Interface

IReposProperties::_NewEnum

This method retrieves an enumeration interface pointer for the **Properties** collection. This interface is a standard Automation enumeration interface. It supports the **Clone**, **Next**, **Reset**, and **Skip** methods. You can use the enumeration interface to step through the members in the collection.

HRESULT _NewEnum(IUnknown ***ppIEnumProps*);**

Parameters

**ppIEnumProps*

[out]
The enumeration interface pointer.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposProperties Interface

IReposProperty Interface

IReposProperty Interface

The **IReposProperty** interface provides access to a persistent member (a property or collection) of an information model interface.

When you instantiate an Automation object that represents an object from your information model, and that object conforms to a class for which there is no custom implementation (in other words, you have provided no software implementation of the class), the repository engine provides an interface implementation for you. This interface implementation uses **IRepositoryDispatch** as its dispatch interface.

The **IRepositoryDispatch** interface is an enhanced **IDispatch** interface; in addition to all of the standard **IDispatch** methods, **IRepositoryDispatch** also provides access to the **Properties** collection. The **Properties** collection gives you a convenient mechanism to enumerate through all of the persistent properties and collections of an interface. The **IReposProperty** interface can be used to access the individual members in the **Properties** collection.

This support enables the Automation programmer to use syntax like the following:

```
Dim firstPropName As String
Let firstPropName = repObject.Properties(1).Name
```

The second statement resolves in the following way:

- In this example, *repObject* is an Automation instantiation of a repository object where the default implementation has been used.

- The **Properties** term is the Automation level name for the **get_Properties** method that is supplied by the **IRepositoryDispatch** interface.

- The **get_Properties** method returns the interface pointer to the **IReposProperties** interface.

- The default method of the **IReposProperties** interface is the **get_Item** method, which returns an **IReposProperty** interface pointer for the specified property object in the **Properties** collection.

- The **Name** term is the Automation level name for the **get_Name** method that is supplied by the **IReposProperty** interface.

At this point, the Automation programmer has access to the name of the first property in the collection through the *firstPropName* variable.

When to Use

Use the **IReposProperty** interface to access a persistent interface member, when no custom implementation is available, and you do not already know the type or name of the member. With this interface, you can:

- Retrieve the name of a property or collection.

- Retrieve the type of a property or collection.

- Get or set the value of a property.

Methods

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch methods	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers.

GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an Automation object.

IReposProperty methods	**Description**
get_Type	Retrieves the type of a persistent interface member.
get_Name	Retrieves the name of a persistent interface member.
get_Value	Retrieves the value of a persistent interface member.
put_Value	Sets the value of a persistent property.

Remarks

Only persistent members (that is, members that are stored in the repository database) can be accessed by the **IReposProperty** interface.

See Also

ReposProperty Class

IReposProperties Interface

IRepositoryDispatch Interface

IReposProperty::get_Name

This method retrieves the name of a persistent interface member (a property or collection).

HRESULT get_Name(BSTR **pName*);

Parameters

**pName*

[out]
The name of the member.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposProperty Interface

IReposProperty::get_Type

This method retrieves the type of a persistent property or collection; that is, it returns the object identifier of the definition object to which the member conforms.

HRESULT get_Type(VARIANT **psTypeId*);

Parameters

**psTypeId*

[out]
The object identifier of the member's definition object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposProperty Interface

IReposProperty::get_Value

This method retrieves the value of a persistent interface member (a property or collection). If the member is a collection, the retrieved value is a pointer to the interface that supports that type of collection.

HRESULT get_Value(VARIANT ***psValue*****);**

Parameters

**psValue*

[out]
The property value.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposProperty Interface

IReposProperty::put_Value

This method sets the value of a persistent interface property. The type of the input parameter is converted to the storage data type of the property. If the type of the input parameter cannot be successfully converted to the storage data type, this method will return an error.

HRESULT put_Value(VARIANT *sValue***);**

Parameters

sValue

[in]
The property value to be set.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

You cannot set the value of a read-only property or a collection.

See Also

IReposProperty Interface

IReposProperty2 Interface

This interface is used by applications such as the Meta Data Browser to retrieve meta data for an interface without having to query the database.

Properties

Properties	Description
APIType	An API type enumeration constant that identifies the API type of the object.
IsBaseMember	Indicates whether the property is a base member.
IsOriginCollection	Indicates whether the collection is the origin of the relationship.
PropType	Returns the **IIFaceMember** object that represents this property.
IsReadOnly	Returns the meta data information that represents this property.

See Also

IReposProperty Interface

Using Meta Data Browser

IReposProperty2 APIType Property

The **APIType** property contains the data type of the property.

Syntax

HRESULT APIType (

LONG **pAPIType*

);

Dispatch Identifier: DISPID_IReposProperty2_APIType = 42

Parameters

**pAPIType*

[out]
The data type of the property.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposProperty2 Interface

IReposProperty2 IsBaseMember Property

The **IsBaseMember** property indicates whether the property is a base member.

Syntax

HRESULT IsBaseMember (

VARIANT_BOOL **pIsBase*

);

Dispatch Identifier: DISPID_IReposProperty2_IsBaseMember = 41

Parameters

****pIsBase***

[out]
TRUE if the property is a base member.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposProperty2 Interface

IReposProperty2 IsOriginCollection Property

The **IsOriginCollection** property indicates whether the collection is the origin of the relationship.

Syntax

HRESULT IsOriginCollection (

VARIANT_BOOL **pIsOrigin*

);

Dispatch Identifier: DISPID_IReposProperty2_IsOriginCollection = 43

Parameters

**pIsOrigin*

[out]
TRUE if the collection is the origin of the relationship.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposProperty2 Interface

IReposProperty2 PropType Property

The **PropType** property returns the **IIFaceMember** of the **PropertyDef**, **CollectionDef**, or **Alias** object that represents this property.

Syntax

HRESULT PropType (

VARIANT **pIIfaceMember*

);

Dispatch Identifier: DISPID_IReposProperty2_PropType = 40

Parameters

**pIIfaceMember*

[out]
A pointer to the **IIFaceMember** interface of this object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

Alias Object

CollectionDef Object

IReposProperty2 Interface

PropertyDef Object

IReposProperty2 IsReadOnly Property

The **IsReadOnly** property indicates whether a property is read-only.

Syntax

HRESULT IsReadOnly(

VARIANT_BOOL **pIsReadOnly*

);

Dispatch Identifier: DISPID_IReposProperty2_IsReadOnly = 44

Parameters

**pIsReadOnly*

[out]
TRUE if the property is a derived member.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposProperty2 Interface

IReposPropertyLarge Interface

This interface is used to handle binary large objects (BLOBs) and large text fields. The **IReposPropertyLarge** is only available to BLOB and text data. Attempting to use it with other kinds of data will fail.

This interface is intended to be an **IDispatch** version of some **IStream** methods.

When to Use

All text and binary fields can publish this interface, regardless of their size.

Properties

Property	Description
Size	The size of a BLOB in bytes.
CurrentPosition	Used to get and set the current position.

Methods

Methods	Description
Read	Reads a chunk of data from the BLOB or large text field.
ReadFromFile	Sets the value of the BLOB or large text field to be the contents of a file.
Close	Notifies the engine that no additional data is to be read from or written to the BLOB or large text field.
Write	Writes a chunk of data to the BLOB or large text field.
WriteToFile	Stores the contents of a BLOB or large text field in a file.

See Also

IReposProperty Interface

Programming BLOBs and Large Text Fields

IReposPropertyLarge::Size

The **Size** property contains the size (in bytes) of a binary large object (BLOB) or large text field.

Syntax

HRESULT Size (

LONG.....**plSize*

);

Dispatch Identifier: DISPID_IReposPropertyLarge_Size = 32

Parameters

**plSize*

[out]
The size of the BLOB.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposPropertyLarge Interface

IReposPropertyLarge::CurrentPosition

The **CurrentPosition** property is used to get and set a position for a large BLOB.

Syntax

HRESULT CurrentPosition (

LONG.....**plCurrentPosition*

);

Dispatch Identifier: DISPID_IReposPropertyLarge_CurrentPosition = 34

Parameters

**plCurrentPosition*

[out]
Moves the read pointer to a position in the BLOB.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposPropertyLarge Interface

IReposPropertyLarge::Read

This method reads a chunk of data from a BLOB or large text field.

Syntax

HRESULT Read (

LONG *SizeToRead*,

LONG **pSizeRead*,

VARIANT ** psBlob*

);

Dispatch Identifier: DISPID_IReposPropertyLarge_Read = 30

Parameters

SizeToRead

[in]
A request for the amount of data to read.

**plSizeRead*

[out]
The actual amount of data read.

**psBlob*

[out]
A pointer to a location where the retrieved data will be stored. It must be large enough to contain the amount of data requested.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposPropertyLarge Interface

IReposPropertyLarge::ReadFromFile

This method configures the object to use a file for the BLOB value.

Syntax

HRESULT ReadFromFile (

BSTR *FileName*

);

Dispatch Identifier: DISPID_IReposPropertyLarge_ReadFromFile = 35

Parameters

FileName

[in]
The path of the file to read.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Existing data in the BLOB value is overwritten, and the seek position is reset.

See Also

IReposPropertyLarge Interface

IRepositoryPropetyLarge::Close

Moves the read pointer to a position in the BLOB.

Syntax

HRESULT Close (

LONG *IPosition*

);

Dispatch Identifier: DISPID_IReposPropertyLarge_Close = 33

Parameters

IPosition

[in]
A byte offset relative to the start of the BLOB.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

This method tells the engine that no more data is to be read from or written to the BLOB.

See Also

IReposPropertyLarge Interface

IReposPropertyLarge::Write

This method writes a chunk of data to a BLOB or large text field.

Syntax

HRESULT Write (

VARIANT *sBlob*

);

Dispatch Identifier: DISPID_IReposPropertyLarge_Write = 31

Parameters

sBlob

[in]
A pointer to the data to be written.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposPropertyLarge Interface

IReposPropertyLarge::WriteToFile

This method configures the object to write the contents of the BLOB to a file.

Syntax

HRESULT WriteToFile (

BSTR *FileName*

);

Dispatch Identifier: DISPID_IReposPropertyLarge_WriteToFile = 36

Parameters

FileName

[in]
The path of the file where the data is written.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IReposPropertyLarge Interface

IReposQuery Interface

This interface allows you to use filters on collections in order to give you control over the objects that appear in an object collection.

When to use

The **IReposQuery** interface is implemented by the following objects to enable you to apply different queries with given filter conditions:

- The **Repository** object: This allows you to query the whole repository.
- The **Workspace** object: This allows you to query the workspace.
- The relationship collection objects: This allows you to query all instances in the given relationship collection. (Notice that the **ITargetObjectCol** interface implies **IReposQuery**; therefore, all the objects that implement this interface also implement **IReposQuery**.)

Methods

IUnknown Method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch Method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces

	that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IReposQuery Method	**Description**
GetCollection	Filters relationship collections in a workspace or in the whole repository

See Also

Filtering Collections

Repository Object

Workspace Object

IReposQuery::GetCollection

This method is used to filter relationship collections in a workspace or in the whole repository.

Syntax

HRESULT GetCollection(
BSTR *Filter*,
long *Flags*
IObjectCol2 ***ppObjCol*

);

Parameters

Filter

[in]
A text string that limits the objects that appear in the collection. This string can be a maximum of 255 characters in length.

Flags

[in]
An optional parameter to control the synchronicity of the query (default = 0). The following flag values are supported.

Flag enumerator	Value	Description
FILTERCOL_SYNCH	0	The collection is fetched synchronously (default).
FILTERCOL_ASYNCH	2	The collection is fetched asynchronously. The collection pointer returned is an **IObjectCol2** interface, which can be used to query the status of the asynchronous fetch.

ppObjCol

[out, retval]
A pointer to the object collection.

Filter Text String

The text string of the *Filter* parameter is case insensitive. The engine will return an error code if the syntax of the *Filter* parameter is wrong. The *Filter* parameter obeys these rules:

1. The string format is based on the SQL WHERE clause format. For example,
   [PROPA] = 'employee' AND [PROPB]>10.

2. Property names are provided in the format:
   [property identifier]

   where the property identifier can either be the object identifier of the property definition, or in the format:

   typelib.interface.property.

   As a result, any occurrence of the character [has to be escaped if it is not marking the start of a property identifier. The escape sequence is \ (that is, you must use \[whenever you want to specify [ ).

   If the typelib or the interface part is omitted, the following rules apply:

   - If the filtering applies to a relationship collection, the omitted type library is assumed to be the same as the type library in which the relationship collection was defined. In addition, the omitted interface name is assumed to be the same as the target

   - If filtering applies to the repository session, an E_REP_UNKNOWNPROPERTY error is returned.

3. The following operators are supported:

- The Boolean operators AND, OR, and NOT.
- The comparison operators =, <, <=, >, >=, IN, and LIKE.
- Grouping of conditions by parentheses.

4. The following special, case-insensitive clauses are supported:
    - **InstanceOf** (*class definition list*)

      The class definition list is a comma-delimited list of class definition object identifiers. The collection returned contains those objects that are instances of these class definitions. The list elements are considered to be connected together by the OR logical operator.
    - **Implements** (*interface definition list*)

      The interface definition list is a comma-delimited list of interface definition object identifiers. The collection returned contains those objects that support the interfaces given. The list elements are considered to be connected together by the OR logical operator.
5. If the *Filter* parameter is empty, all objects in the collection are returned.

   The *Filter* parameter can include an **Order By** clause that accepts a comma-separated list of property names. The property names follow the same rules as property names in the selection part of the *Filter* parameter.

   The following is an example of a valid *Filter* parameter:

   [FirstName]='Jason' AND [Age]>20 ORDER BY [LastName]

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

Filtering Collections

Filtering Derived Collections

IReposQuery Interface

ISummaryInformation Interface

The **ISummaryInformation** interface maintains **Comments** and **ShortDescription** properties for objects that expose this interface.

When to Use

Use the **ISummaryInformation** interface to access the **Comments** and **ShortDescription** properties of a repository object.

Properties

Property	Description
Comments	General comments about the object
ShortDescription	A brief description of the object

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods

	exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the properties exposed by the **ISummaryInformation** interface.

ISummaryInformation Comments Property

This property contains general comments about an object. Up to 65,536 bytes of information can be stored in this property.

Dispatch Identifier: DISPID_Comments (66)

Property Data Type: **long varchar**

See Also

ISummaryInformation Interface

ISummaryInformation ShortDescription Property

This property contains a short description of an object. Up to 255 bytes of information can be stored in this property.

Dispatch Identifier: DISPID_ShortDesc (67)

Property Data Type: **varchar**

See Also

ISummaryInformation Interface

ITargetObjectCol Interface

A target object collection is a set of repository object versions that are attached to a particular source object version through a relationship collection. At most, one version of each repository object is present in any target object collection.

When to Use

Use the **ITargetObjectCol** interface to manage the repository objects that belong to a particular relationship collection. With this interface, you can:

- Get a count of the number of objects in the collection.
- Enumerate the objects in the collection.
- Add and remove objects to and from the collection.
- If the collection is sequenced, place an object in a specific spot in the collection sequence.
- Retrieve an **IRepositoryObject** pointer to one of the objects in the collection.
- Obtain the type of the collection.
- Retrieve an interface pointer for the collection's source object.
- Refresh the cached image of the target object collection.

Methods

IUnknown methods	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IObjectCol method	Description
get_Count	Retrieves a count of the number of objects in the collection.
_NewEnum	Retrieves an enumeration interface pointer for the collection. This interface is a standard Automation enumeration interface. It supports the **Clone**, **Next**, **Reset**, and **Skip** methods.
get_Item	Retrieves an **IRepositoryObject** interface pointer for the specified collection object.
Refresh	Refreshes the cached image of the target object collection.

ITargetObjectCol method	Description

Add	Adds an object to the collection
get_Source	Retrieves an interface pointer for the collection's source object
get_Type	Retrieves the object identifier for the collection's definition object
Insert	Inserts an object into a specific spot in a sequenced collection
Move	Moves an object from one spot to another in a sequenced collection
Remove	Removes an object from the collection

Remarks

The **ITargetObjectCol** interface is similar to the **IRelationshipCol** interface. Use the **ITargetObjectCol** interface when you are primarily interested in working with objects. Use the **IRelationshipCol** interface when you are primarily interested in working with relationships between objects.

See Also

IRelationshipCol Interface

RelationshipCol Class

ITargetObjectCol::Add

This method is used to add a new item to an object collection, when the sequencing of objects in the collection is not important. An interface pointer for the new relationship is passed back to the caller.

HRESULT Add(**IDispatch** **plReposObj*,
BSTR *Name*,
IRelationship ***pplRelship*
);

Parameters

**plReposObj*

[in]
The repository object version to be added to the collection.

Name

[in]
The name to be assigned to the object that is being added to the collection.

**pplRelship*

[out]
The newly added object's relationship interface pointer.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Objects may only be added to a collection when the collection's source object is also the collection's origin object.

When you call this method, the origin version must be unfrozen.

You can use this method to create a new versioned relationship between the source object version and a version of the target object. You cannot use it to enlarge a versioned relationship. If the source object version already has a relationship to any version of the target object, this method will fail. You can include another version of the target object in the versioned relationship by adding an item to the versioned relationship's **TargetObjects** collection.

The value of *plReposObj* is the specific version of the target object.

See Also

IRelationship Interface

ITargetObjectCol Interface

ITargetObjectCol::get_Source

This method retrieves the **IRepositoryObject** interface pointer for the collection's source object version.

HRESULT get_Source(IRepositoryObject ***ppIInterface*);**

Parameters

**ppIInterface*

[out]
The interface pointer of the **IRepositoryObject** interface for the source object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

ITargetObjectCol Interface

ITargetObjectCol::get_Type

This method retrieves the type of the collection; that is, it returns the object identifier for the collection's definition object.

HRESULT get_Type(VARIANT **pColDefObjId***);**

Parameters

**pColDefObjId*

[out]
The object identifier of the collection's definition object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

ITargetObjectCol Interface

ITargetObjectCol::Insert

This method adds an object to the collection at a specified point in the collection sequence. An interface pointer for the new relationship is passed back to the caller.

HRESULT Insert(**IDispatch** **pIReposObj*,
long *iIndex*,
BSTR *Name*,
IRelationship ***ppIRelship*
);

Parameters

**pIReposObj*

[in]
The repository object to be inserted into the collection sequence.

iIndex

[in]
The index of the sequence location where the object is to be inserted. If another object is already present at this sequence location, the new object is inserted before the existing object.

Name

[in]
The name of the object. Set this parameter to a null string if the object is not referred to by name.

**ppIRelship*

[out]
The **IRelationship** interface pointer for the new object's relationship with the collection's origin object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Objects may only be inserted into a collection when the collection's source object is also the collection's origin object.

This method can only be used for collections that are sequenced.

When you call this method, the origin version must be unfrozen.

You can use this method to insert a new versioned relationship between the source object version and a version of the target object. You cannot use it to enlarge a versioned relationship. If the source object version already has a relationship to any version of the target object, this method will fail. You can include another version of the target object in the versioned relationship by adding an item to the versioned relationship's **TargetObjects** collection.

The value of *plReposObj* is the specific version of the target object.

See Also

IRelationship Interface

ITargetObjectCol Interface

ITargetObjectCol::Move

This method moves an object from one point in the collection sequence to another point.

HRESULT Move(
  long *iIndexFrom*,
  long *iIndexTo*
);

Parameters

iIndexFrom

[in]
The index of the object to be moved in the collection sequence.

iIndexTo

[in]
The index of the sequence location to which the object is to be moved.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

This method can only be used for collections that are sequenced.

The origin object version must be unfrozen.

See Also

ITargetObjectCol Interface

ITargetObjectCol::Remove

This method removes the specified object from the collection. The exact behavior of this method depends on whether the relationship collection is an origin collection or a destination collection.

If the relationship collection is an origin collection, this method deletes the versioned relationship.

If the relationship collection is a destination collection, this method first performs object-version resolution to yield a single target-object version, and then it removes that target-object version from the relationship's **TargetVersions** collection.

HRESULT Remove(VARIANT *sItem*
);

Parameters

sItem

[in]
Identifies the item to be retrieved from the collection. This parameter can be the index, the name, or the object identifier of the item.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

An object can be removed by name only if it is the destination object of a naming relationship.

If the source is the origin, the origin version must be unfrozen.

If the relationship is a destination relationship, and the resolution strategy yields a target object version that is frozen, this method fails.

Removal from a sequenced collection does not update the collection sequence order.

See Also

ITargetObjectCol Interface

Resolution Strategy for Objects and Object Versions

ITransientObjectCol Interface

This interface provides transient object collections that you can create and dynamically populate at run time using script and object methods rather than persisted data in a repository database.

ITransientObjectCol inherits from **IObjectCol**. Except for the fact that a transient object collection is not saved to a repository database, it is identical in functionality to the **ObjectCol** object.

You can have multiple transient object collections at one time. The object collection can contain only repository objects. Although enumeration is supported, sequencing is not. Objects and object collections represented by **TransientObjectCol** are not versioned.

When to Use

Use this interface to create an object collection that is instantiated by application code and populated dynamically at run time. With this interface, you can add and remove objects to and from the collection

Methods

Method	Description
Add	Adds an object to the collection.
Remove	Removes an object from the collection.

See Also

ObjectCol Class

TransientObjectCol Class

ITransientObjectCol::Add

Use this method to add target objects to an object collection.

Syntax

HRESULT Add([in] IDispatch **pIReposObj* **)**

Parts

**pIReposObj*

[in]
The repository object version to be added to the collection

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

Populating a **TransientObjectCol** is done using the **Add** method for each object that you want to add to the collection.

See Also

ITransientObjectCol Interface

ITransientObjectCol::Remove

ITransientObjectCol::Remove

Use this method to remove a specified object from a transient object collection.

Syntax

HRESULT Remove([in] VARIANT *sItem* **)**

Parts

sItem

[in]
Identifies the item to be removed from the collection. This parameter can be the index, the object identifier, or the **Object-Version** identifier of the item.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

This property method a specific repository object from the collection. You can identify an object by its position in the collection (as indicated by the index) or by identifier.

See Also

ITransientObjectCol Interface

Object Identifiers and Internal Identifiers

IVersionAdminInfo Interface

Use this interface to retain and manipulate administrative information about repository object versions.

When to Use

By default, no class implements this interface. But within information models, any class can implement this interface, thereby ensuring that for each object conforming to that class, the repository automatically retains the properties listed here.

Properties

Property	Description
CreateByUser	The user who created the object version.
ModifyByUser	The user who made the most recent modification to the object version.
VersionCreateTime	The date and time the object version was created.
VersionModifyTime	The date and time of the most recent modification to the object version.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers.

GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an Automation object.

See Also

IVersionAdminInfo2 Interface

IVersionCol Interface

IVersionedRelationship Interface

IVersionAdminInfo CreateByUser Property

This property indicates the user who created the object version.

Dispatch Identifier: DISPID_CreateByUser (83)

Property Data Type: **string**

See Also

IVersionAdminInfo Interface

IVersionAdminInfo ModifyByUser Property

This property indicates the user who most recently modified the object version. If the object version is frozen, this property indicates the user who froze the object version.

Dispatch Identifier: DISPID_ModifyByUser (84)

Property Data Type: **string**

Remarks

ModifyByUser does not change when an origin relationship or target object collection is modified.

See Also

IVersionAdminInfo Interface

IVersionAdminInfo VersionCreateTime Property

This property contains the date and time at which the object version was created.

The default value is set to 9999-12-31-00:00:0000 after the object is created but before it is committed. The current date and time are set only after the commit is successful.

Dispatch Identifier: DISPID_VersionCreateTime (81)

Property Data Type: **datetime**

See Also

IVersionAdminInfo Interface

IVersionAdminInfo VersionModifyTime Property

This property contains the date and time at which the object version was most recently modified. If the object version is frozen, this property is the date and time at which the object version was frozen.

The default value is set to 9999-12-31-00:00:0000 after the object is created but before it is committed. The current date and time are set only after the commit is successful.

Dispatch Identifier: DISPID_VersionModifyTime (82)

Property Data Type: **strin**g

Remarks

VersionModifyTime does not change when an origin relationship or target object collection is modified.

See Also

IVersionAdminInfo Interface

IVersionAdminInfo2 Interface

Use this interface to set and retrieve the description of repository object versions. This interface inherits from the **IVersionAdminInfo** interface.

When to Use

By default, no class except for Repository Type Information Model (RTIM) classes implements this interface. But within information models, any class can implement this interface. Use this interface to ensure that the repository automatically retains the properties inherited from **IVersionAdminInfo**. You can also use this interface to set or retrieve the version comments properties on **IVersionAdminInfo2**.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

Properties

IVersionAdminInfo property	Description
CreateByUser	The name of the user who created the object version
ModifyByUser	The name of the user who made the most recent modification to the object version
VersionCreateTime	Date and time the object version was created
VersionModifyTime	Date and time of the most recent modification to the object version

IVersionAdminInfo2 property	Description
VersionLabel	The version string property. It is an application-supplied version label.
VersionComments	The version comment property. It corresponds to the comments added when a file is checked into a version control system.
VersionShortDesc	The short description property. It is a short summary of the version comments.

See Also

Repository Type Information Model

IVersionAdminInfo Interface

IVersionAdminInfo2 VersionLabel Property

This property indicates the version label supplied by the application.

Dispatch Identifier: DISPID_ VersionLabel (90)

Property Data Type: **string**

See Also

IVersionAdminInfo Interface

IVersionAdminInfo2 Interface

IVersionAdminInfo2 VersionComments Property

This property contains user-defined comments about the version.

Dispatch Identifier: DISPID_ VersionComments (92)

Property Data Type: **string**

See Also

IVersionAdminInfo Interface

IVersionAdminInfo2 Interface

IversionAdminInfo2 VersionShortDesc Property

This property is used to add a short description comment.

Dispatch Identifier: DISPID_ VersionShortDesc (91)

Property Data Type: **string**

See Also

IVersionAdminInfo Interface

IVersionAdminInfo2 Interface

IVersionCol Interface

A version collection is a collection of object versions. The repository API supports multiple collection types. For more information about each one, see Kinds of Version Collections.

When to Use

Use the **IVersionCol** interface to manage the contents of a workspace, to manage the target object versions of a versioned relationship, to navigate an object's version graph, or to manipulate all the versions of a particular object.

Methods

IUnknown Method	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch Method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers.
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an Automation object.

IVersionCol Method	Description
Add	Adds an object version to the collection.

get_Count	Returns the number of items in the collection.
get_Item	Returns an interface pointer to an item of the collection.
_NewEnum	Retrieves an enumeration interface pointer for the collection.
Refresh	Refreshes the cached image of the collection.
Remove	Removes an object version from the collection.

See Also

VersionCol Class

IVersionCol::Add

Adds an object version to the collection.

Syntax

HRESULT Add(**IRepositoryObjectVersion** **pIReposVersion*
IRepositoryObjectVersion ***ppIAddedVersion*
);

Parts

**pIReposVersion*

[in]
The **IRepositoryObjectVersion** interface pointer to the object version to be added to the collection.

***ppIAddedVersion*

[in]

a. For Target-Versions, ppIAddedVersion is the same as the first parameter: pIReposVersion.

b. For Versions-of-Workspace, ppIAddedVersion is the workspace-scoped version of pIReposVersion.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

There are many different kinds of object version collections. You can apply this method to some of them, but not to others. This method works for:

- **TargetVersions** collection. You can use this method to enlarge the set of versions of a particular target object that are related to a particular source object.

- **Contents** collection. You can use this method to add an object version to the set of items contained in the workspace.

This method does not work for:

- **Predecessor** collection. To enlarge an object version's set of predecessors, use the **MergeVersion** method of the **IRepositoryObjectVersion** interface.

- **Successor** collection. To enlarge an object version's set of successors, use the **CreateVersion** method of the **IRepositoryObjectVersion** interface.

- **ObjectVersions** collection. To enlarge an object's set of versions, use the **CreateVersion** method of the **IRepositoryObjectVersion** interface.

- **Workspaces** collection. To enlarge the set of workspaces to which an object version belongs, you do not add a workspace to an object version —rather you add the object version to a workspace. In other words, you use the **Add** method of the **IVersionCol** interface. In this case, the version collection you are manipulating is the **Contents** collection, not the **Workspaces** collection.

- **Checkouts** collection. To check out another item to a workspace, use the **Checkout** method of the **IWorkspaceItem** interface.

See Also

IVersionCol Interface

IVersionCol::get_Count

Retrieves count of the number of items in the collection.

Syntax

HRESULT get_Count(**long** **piCount*
);

Parts

**piCount*

[out]
The number of items in the collection.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IVersionCol Interface

IVersionCol::get_Item

Retrieves the specified object version from the collection.

Syntax

HRESULT get_Item(**VARIANT** *sItem*
IRepositoryObjectVersion ***ppIReposVersion*
);

Parts

sItem

[in]
Identifies the item to be retrieved from the collection. This parameter can be the index, the object identifier, or the object-version identifier of the item.

***ppIReposVersion*

[out]
The **IRepositoryObjectVersion** interface pointer for the retrieved object versions.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

There are many different kinds of version collection. The *sItem* parameter can be an object identifier for some version collections, but not for others. It can be an object identifier only for the **ObjectVersions** collection, the **Workspaces**

collection, or the **Checkouts** collection.

See Also

IVersionCol Interface

IVersionCol::_NewEnum

This method retrieves an enumeration interface pointer for the relationship collection. This interface is a standard Automation enumeration interface. It supports the **Clone**, **Next**, **Reset**, and **Skip** methods. You can use the enumeration interface to step through the relationships in the collection.

Syntax

HRESULT _NewEnum(IUnknown ***ppIEnum***
);

Parts

ppIEnum

[out]
The enumeration interface pointer.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IVersionCol Interface

IVersionCol::Refresh

This method refreshes the cached image of the collection. All unchanged data for items in the collection is flushed from the cache.

Syntax

HRESULT Refresh(**long** *iMilliseconds*
);

Parts

iMilliseconds

[in]
This value is ignored.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IVersionCol Interface

IVersionCol::Remove

Removes an object version from the collection.

Syntax

HRESULT Remove(**VARIANT** *sItem*
);

Parts

sItem

[in]
Identifies the item to be removed from the collection. This parameter can be the index, the object identifier, or the **Object-Version** identifier of the item.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

There are many different kinds of **Object-Version** collections. You can apply this method to some of them, but not to others. This method works for:

- **TargetVersions** collections. You can use this method to reduce the set of versions of a particular target object that are related to a particular source object.

- **ObjectVersions** collections. You can use this method to remove an object version from the set of items contained in the workspace.

This method fails for:

- **Predecessor** collections. To enlarge an object version's set of predecessors, use **MergeVersion**.

- **Successor** collections. To enlarge an object version's set of successors, use the **CreateVersion** method of the **IRepositoryObjectVersion** interface.

- **ObjectVersions** collections. To enlarge an object's set of versions, use the **CreateVersion** method of the **IRepositoryObjectVersion** interface.

- **Workspaces** collections. To remove a workspace from the set of workspaces in which an object version is present, you must explicitly remove the object version from that workspace's **ObjectVersions** collection.

- **Checkouts** collections. To reduce the number of items checked out to a workspace, use the **Checkin** method of the **IWorkspaceItem** interface.

The *sItem* parameter can be an object identifier for some version collections, but not for others. It can be an object identifier only for the **ObjectVersions** collection, the **Workspaces** collection, or the **Checkouts** collection.

See Also

IVersionCol Interface

Kinds of Version Collections

IVersionedRelationship Interface

A versioned relationship connects one source object version to any number of versions of a destination object. Versioned relationships are items within relationship collections.

When to Use

Use the **IVersionedRelationship** interface to manipulate a relationship, or to retrieve the **source**, **target**, **origin**, or **destination** object for a relationship.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch	

method	Description
Get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IRepositoryItem methods	Description
Delete	Deletes a repository item
Get_Interface	Retrieves an interface pointer to the specified item interface
Get_Name	Retrieves the name associated with an item
Get_Repository	Retrieves the **IRepository** interface pointer for an item's open repository instance
Get_Type	Retrieves the type of an item
Lock	Locks the item
Put_Name	Sets the name associated with an item

IRelationship method	Description
Get_Destination	Retrieves an interface pointer to the destination object
Get_Origin	Retrieves an interface pointer to the origin object
Get_Source	Retrieves an interface pointer to the source object
Get_Target	Retrieves an interface pointer to the target object

IVersionedRelationship method	Description
Get_TargetVersions	Returns an interface pointer to the set of target versions of the relationship
Pin	Establishes one target version as the pinned

	target version
Unpin	Unpins all target versions

See Also

IRelationship Interface

Versioning Objects

IVersionedRelationship::get_TargetVersions

Retrieves an **IVersionCol** interface pointer to a collection of object versions. Each item in the collection is a particular version of the target object to which a versioned relationship refers.

HRESULT get_TargetVersions(**IVersionCol** ***ppTargetVersions*
);

Parameters

***ppTargetVersions*

[out]
The **IVersionCol** interface pointer for the collection of target object versions.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IRelationship Interface

IVersionedRelationship Interface

IVersionedRelationship::Pin

Identifies which target object version of an origin relationship is the pinned version.

HRESULT Pin(**IRepositoryObjectVersion** **pIReposVersion*
);

Parameters

**pIReposVersion*

[in]
The **IRepositoryObjectVersion** interface pointer for the object version to be pinned.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

You can use this method only for origin relationships. The origin object of the versioned relationship must be unfrozen.

If the origin object of the relationship is checked out to a workspace, the **Pin** method will work only from within that workspace.

When you pin a target object version for versioned relationship, any previously pinned target object version of the relationship becomes unpinned.

The target object version to be pinned must already participate in the relationship.

See Also

IRelationship Interface

IRelationship::Get_Destination

IVersionedRelationship Interface

IVersionedRelationship::Unpin

Declares that no target object version of an origin versioned relationship is pinned.

HRESULT Unpin(void);

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

You can use this method only for origin relationships. The origin object of the versioned relationship must be unfrozen.

If the origin object of the versioned relationship is checked out to a workspace, the **Unpin** method will work only from within that workspace.

See Also

IRelationship Interface

IRelationship::Get_Target

IVersionedRelationship Interface

IWorkspace Interface

A workspace is a subset of the repository within which you can operate on tool data in isolation from other repository activity. The **IWorkspace** interface provides methods for operating on workspaces.

When to Use

Use the **IWorkspace** interface to manage the object versions present in the workspace, the object versions checked out to the workspace, and to manage the workspace containers in which the workspace is present. (In Microsoft® SQL Server™ 2000 Meta Data Services, there is only one workspace container, the root object.)

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers.
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an Automation object.

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IWorkspace method	Description
get_Checkouts	Returns the collection of object versions checked out to the workspace.
get_Contents	Returns the collection of object versions present in the workspace.

Collections

Collection	Description
Containers	The collection of objects containing the current workspace.

See Also

Workspace Class

IWorkspace Containers Collection

This collection specifies the containers of this workspace. The collection contains exactly one item, the root object.

Dispatch Identifier: DISPID_WorkspaceContainers (85)

Remarks

Although the collection's maximum size is defined as many, the collection always contains exactly one object, because **CReposRoot** is the only class that implements **IWorkspaceContainer**, and there is only one object (the root object) conforming to the **CReposRoot** class.

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**WsContainer_Contains_Workspace**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum	Many	The maximum

Collection Size		number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-Sensitive Names	Not applicable	Case-sensitive naming is not applicable for this collection.
Unique Names	Not applicable	Unique naming is not applicable for this collection.

See Also

IWorkspace Interface

IWorkspace::get_Checkouts

This method returns an **IVersionCol** interface pointer to a collection of object versions currently checked out to the workspace.

HRESULT get_Checkouts(**IVersionCol** ***ppWSVersions**
);

Parameters

***ppWSVersions**

[out]
The **IVersionCol** interface pointer to the collection of object versions checked out to the workspace.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

For each repository object, one version at most can be checked out to the current workspace.

See Also

IInterfaceDef Interface

IWorkspace Interface

IWorkspace::get_Contents

IWorkspace::get_Contents

This method returns an **IVersionCol** interface pointer to a collection of object versions currently present in the workspace.

HRESULT get_Contents(**IVersionCol** ***ppWSVersions*
);

Parameters

***ppWSVersions*

[out]
The **IVersionCol** interface pointer to the collection of object versions present in the workspace.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

For each repository object, one version at most can be present in the current workspace.

See Also

IInterfaceDef Interface

IWorkspace Interface

IWorkspace::get_Checkouts

IWorkspaceContainer Interface

The **IWorkspaceContainer** interface contains methods for managing the collection of workspaces within a repository.

When to Use

Use the **IWorkspaceContainer** interface to retrieve the collection of workspaces in a repository.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access

	to the **Properties** collection.

Collections

Collection	Description
Workspaces	The collection of workspaces contained by this repository object

See Also

IWorkspaceItem::get_Workspaces

ReposRoot Class

IWorkspaceContainer Workspaces Collection

This collection specifies the workspaces contained in the workspace container. Only the root object can be a workspace container.

Dispatch Identifier: DISPID_ContainedWorkspaces (84)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**WsContainer_Contains_Workspace**	The type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	Yes	The source object for the collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.

Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	Deleting an origin object or a relationship in the collection causes the deletion of a corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-Sensitive Names	No	The collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects.

See Also

IWorkspaceContainer Interface

IWorkspaceItem::get_Workspaces

IWorkspaceItem Interface

The **IWorkspaceItem** interface contains methods for managing workspace items, that is, object versions that can be present in or checked out to a workspace.

When to Use

Use the **IWorkspaceItem** interface to manage the participation of object versions within workspaces.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer.

	The **IReposProperties** interface provides access to the **Properties** collection.

IWorkspaceItem method	Description
Checkin	Terminates the ability to modify the current object version from within the current workspace
Checkout	Establishes the current workspace as the only workspace within which the current object version can be modified
get_CheckedOutToWorkspace	Returns the workspace to which the current object version is checked out
get_IsCheckedOut	Indicates whether any workspace has the current object version checked out
get_Workspaces	Returns the collection of workspaces in which the current object version is present

See Also

Workspace Object

IWorkspaceItem::Checkin

This method terminates the ability to modify the current object version from within the current workspace.

HRESULT Checkin();

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

When you call this method, you must be operating within the workspace to which the object version is checked out.

See Also

IWorkspaceItem Interface

IWorkspaceItem::Checkout

This method establishes the current workspace as the only workspace within which the current object version can be modified.

HRESULT Checkout();

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

If the object version is already checked out to a workspace, this method returns an error.

See Also

IWorkspaceItem Interface

IWorkspaceItem::get_CheckedOutToWorkspace

This method returns an **IWorkspace** interface pointer of the workspace to which the current object version is checked out.

HRESULT get_CheckedOutToWorkspace(IWorkspace ***ppIWorkspace* **);**

Parameters

***ppIWorkspace*

[out]
The **IWorkspace** interface pointer of the workspace.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

If the object version is not currently checked out to any workspace, this method returns an error.

See Also

IWorkspaceItem Interface

IWorkspaceItem::get_IsCheckedOut

This method determines whether the current workspace item is checked out to a workspace.

HRESULT get_IsCheckedOut(VARIANT_BOOL **pbCheckedOut*
);

Parameters

**pbCheckedOut*

[out]
TRUE if the object version is checked out to a workspace; FALSE if the object version is not checked out to any workspace.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IWorkspaceItem Interface

IWorkspaceItem::get_Workspaces

This method returns the collection of workspaces in which the current object version is present.

HRESULT get_Workspaces(IVersionCol ***ppIWorkspaces*
);

Parameters

***ppIWorkspaces*

[out]
The **IVersionCol** interface pointer to the collection of workspaces in which the object version is present.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

If the current object version is not present in any workspaces, this method returns an empty collection.

See Also

IWorkspaceItem Interface

Meta Data Services Programming

RTIM Classes

The Repository Type Information Model (RTIM) is the object model that the repository engine uses to define and store information models. Use the RTIM classes to programmatically create or extend an information model. These classes build upon the fundamental repository engine classes. For more information, see Repository Engine Classes.

All repository classes expose the standard **IUnknown** and **IDispatch** interfaces that provide fundamental COM and Automation support.

Class	Description
Alias	Defines property classes of a derived property without changing the meaning of the underlying property.
ClassDef	Defines object classes in an information model.
CollectionDef	Defines collection classes of object relationships.
EnumerationDef	Defines object classes of enumeration objects.
EnumerationValueDef	Defines object classes of enumeration value objects.
InterfaceDef	Defines interface classes.
MethodDef	Defines method classes.
ParameterDef	Defines parameter classes.
PropertyDef	Defines property classes.
RelationshipDef	Defines relationship classes.
ReposRoot	Defines an object class of the repository root object. This is the starting point for all repository navigation.
ReposTypeLib	Defines an object class of an information model.
ScriptDef	Defines script definition classes.

See Also

COM Reference

Information Models

Repository API Reference

Alias Class

The **Alias** class supports member delegation. This class defines a derived property that is based on another property without changing the meaning of the underlying property.

When to Use

Use the **Alias** class to rename an existing property.

Interfaces

Interface	Description
IInterfaceMember2	Creates simple, derived members as instances from the **Alias** class.
IRepositoryDispatch	Provides enhanced dispatch support.
IRepositoryItem	Manages repository objects and relationships.

See Also

Alias Object

Member Delegation

RTIM Classes

ClassDef Class

When you define an information model in Microsoft® SQL Server™ 2000 Meta Data Services, you define classes of objects, types of relationships that can exist between objects, and various properties that are attached to these object classes and relationship types. The object classes that you define in your information model are represented by instances of the **ClassDef** class.

To insert a new class definition into an information model, use the **ReposTypeLib** class.

When to Use

Use the **ClassDef** class to complete the definition of a new repository class. You can define new interfaces and attach them to the class definition. You can also attach existing interfaces to the class definition.

Interfaces

Interface	Description
IAnnotationalProps	Gets and sets annotational properties
IClassDef	Manages class definitions
IClassDef2	Manipulates the **ScriptsUsedByClass** collection
INamedObject	Retrieves or sets the class name
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IRepositoryObject	Retrieves repository object identifiers
IRepositoryObjectStorage	Creates and loads repository objects
IReposTypeInfo	Contains the collection of definition objects that are associated with an information model's repository type library
IVersionAdminInfo2	Retains properties inherited from **IVersionAdminInfo** and sets or retrieves version comments

IViewClassDef	Defines database views for a class

See Also

ClassDef Object

ReposTypeLib Class

RTIM Classes

CollectionDef Class

Repository objects are related to each other through relationships. The set of relationships, all of the same type, that relate one object to zero or more other objects, is a relationship collection.

A collection type (also referred to as a collection definition) defines how instances of a particular collection type will behave. The characteristics of the collection type determine:

- The minimum and maximum number of items in a collection.
- Whether the collection type is an origin collection type.
- Whether the collection type permits the naming of destination objects, and if so, whether those names are case sensitive and required to be unique.
- Whether the collection type permits the explicit sequencing of items in the collection.
- What happens to related objects when objects or relationships in the collection are deleted.
- The kind of relationship that a particular collection type uses to relate objects to each other.

A collection is attached to an interface as a member of the interface. To add a new collection type to an interface definition, use the **InterfaceDef** class.

When to Use

Use the **CollectionDef** class to retrieve or modify the properties of a collection

type, or to determine the kind of relationship that the collection implements.

Interfaces

Interface	Description
IAnnotationalProps	Gets and sets annotational properties
ICollectionDef	Manages collection definitions
IInterfaceMember	Relates a member to an interface
IInterfaceMember2	Creates simple, derived members as instances of the **Alias** class, and creates semantically rich derived members as instances of the **CollectionDef** class
INamedObject	Retrieves or sets the class name
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IRepositoryObject	Retrieves repository object identifiers
IRepositoryObjectStorage	Creates and loads repository objects
IVersionAdminInfo2	Retains properties inherited from **IVersionAdminInfo** and sets or retrieves version comments

See Also

CollectionDef Object

InterfaceDef Class

RTIM Classes

EnumerationDef Class

The **EnumerationDef** class defines objects that contain enumerated values.

When to Use

Use the **EnumerationDef** class to create an enumeration object.

Interfaces

Interface	Description
IEnumerationDef	Creates an enumeration object

See Also

EnumerationDef Object

EnumerationValueDef Class

RTIM Classes

EnumerationValueDef Class

The **EnumerationValueDef** class defines objects that represent a specific enumerated value. Each enumerated value is a separate instance of the **EnumerationValueDef** object.

When to Use

Use the **EnumerationValueDef** class to create an enumeration value.

Interfaces

Interface	Description
IEnumerationValueDef	Creates an enumeration value

See Also

EnumerationDef Class

EnumerationValueDef Object

RTIM Classes

InterfaceDef Class

The properties, methods, and collections that a class implements are organized into functionally related groups. Each group is implemented as a COM interface. The properties, methods, and collections of each interface are members of the interface. An interface definition is the template to which an interface conforms. Interface definitions are instances of the **InterfaceDef** class.

To create a new interface definition, use the **ClassDef** class or the **ReposTypeLib** class.

When to Use

Use the **InterfaceDef** class to:

- Retrieve or modify properties of an interface definition.
- Determine which members are attached to an interface definition.
- Determine which classes implement an interface.
- Determine the base interface from which an interface derives.
- Determine what interfaces derive from a particular interface.
- Determine which repository objects expose a particular interface.
- Add a new property, method, or collection type to an interface definition.

Interfaces

Interface	Description
IAnnotationalProps	Gets and sets annotational properties
IInterfaceDef	Manages interface definitions
INamedObject	Retrieves or sets the class name
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IRepositoryObject	Retrieves repository object identifiers
IRepositoryObjectStorage	Creates and loads repository objects
IReposTypeInfo	Contains the collection of definition objects that are associated with an information model's repository type library
IReposTypeInfo2	Allows classes, interfaces and relationships to be referred to by multiple names as aliases
IVersionAdminInfo2	Retains properties inherited from **IVersionAdminInfo** and sets or retrieves version comments
IViewInterfaceDef	Defines a database view for all objects that implement a specific interface

See Also

ClassDef Class

InterfaceDef Object

ReposTypeLib Class

RTIM Classes

MethodDef Class

When you define a class for an information model, you specify the interfaces that the class implements. For each of those interfaces, you specify the members (properties, methods, and collections) that are attached to the interface.

The definition of a method as a member of an interface does not result in the method's implementation logic being stored in the repository. However, it does add the method name to the set of defined member names for that interface. It also reserves the method's dispatch identifier in the set of defined dispatch identifier values for the interface.

Instances of the **MethodDef** class represent method definitions.

To attach a new method to an interface, use the **InterfaceDef** class.

When to Use

Use the **MethodDef** class to access or modify the characteristics of a method definition, or to determine the interface definition to which a particular method is attached.

Interfaces

Interface	Description
IAnnotationalProps	Gets and sets annotational properties
IInterfaceMember	Relates a member to an interface
IInterfaceMember2	Creates simple, derived members as instances of the **Alias** class, and creates semantically rich derived members as instances of the **CollectionDef** class
INamedObject	Retrieves or sets the class name
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IRepositoryObject	Retrieves repository object identifiers

IRepositoryObjectStorage	Creates and loads repository objects
IVersionAdminInfo2	Retains properties inherited from **IVersionAdminInfo** and sets or retrieves version comments

See Also

InterfaceDef Class

MethodDef Object

RTIM Classes

ParameterDef Class

When you define a method for an information model, you can specify parameters that the method implements. Instances of the **ParameterDef** class represent parameters of method definitions.

When to Use

Use the **ParameterDef** class to create parameters for a method definition object that you define.

Interfaces

Interface	Description
IAnnotationalProps	Gets and sets annotational properties
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IRepositoryObject	Retrieves repository object identifiers
IRepositoryObjectStorage	Creates and loads repository objects
IVersionAdminInfo2	Retains properties inherited from **IVersionAdminInfo** and sets or retrieves version comments

See Also

MethodDef Object

ParameterDef Object

RTIM Classes

PropertyDef Class

When you define a class for an information model, you specify the interfaces that the class implements. For each of those interfaces, you specify the members (properties, methods, and collections) that are attached to the interface.

In order to attach a property to an interface, a property definition must exist for the property. The characteristics of the property (its name, dispatch identifier, data type, and various storage details) are stored in the property definition. Property definitions are instances of the **PropertyDef** class.

To attach a new property to an interface, use the **InterfaceDef** class.

When to Use

Use the **PropertyDef** class to access or modify the characteristics of a property definition, or to determine the interface definition to which a particular property is attached.

Interfaces

Interface	Description
IAnnotationalProps	Gets and sets annotational properties
IInterfaceMember	Relates a member to an interface
IInterfaceMember2	Creates simple, derived members as instances of the **Alias** class, and creates semantically rich derived members as instances of the **CollectionDef** class
INamedObject	Retrieves or sets the class name
IPropertyDef	Retains property characteristics
IPropertyDef2	Contains an optional relationship to a single **EnumerationDef** object
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IRepositoryObject	Retrieves repository object identifiers

IRepositoryObjectStorage	Creates and loads repository objects
IVersionAdminInfo2	Retains properties inherited from **IVersionAdminInfo** and sets or retrieves version comments
IViewPropertyDef	Defines the column name of a property in a view

See Also

InterfaceDef Class

PropertyDef Object

RTIM Classes

RelationshipDef Class

When you define an information model in a repository, you define classes of objects, types of relationships that can exist between objects, and various properties that are attached to these object classes and relationship types. The relationship types that you define in your tool information model are represented by instances of the **RelationshipDef** class.

When to Use

Use the **RelationshipDef** class to access the properties of a relationship definition (also referred to as a relationship type).

To insert a new relationship type into an information model, use the **ReposTypeLib** class.

Interfaces

Interface	Description
IAnnotationalProps	Gets and sets annotational properties
IClassDef	Manages class definitions
IClassDef2	Manipulates the **ScriptsUsedByClass** collection
INamedObject	Retrieves or sets the class name
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IRepositoryObject	Retrieves repository object identifiers
IRepositoryObjectStorage	Creates and loads repository objects
IReposTypeInfo	Contains the collection of definition objects that are associated with an information model's repository type library
IVersionAdminInfo2	Retains properties inherited from **IVersionAdminInfo** and sets or retrieves version comments
IViewRelationshipDef	Defines a junction table view of a relationship

	class

See Also

RelationshipDef Object

ReposTypeLib Class

RTIM Classes

ReposRoot Class

There is one root object in each repository. The root object is the starting point for navigating to other objects in the repository. The root object serves as the starting point for both type data navigation and instance data navigation.

- **Type data navigation**

  When you create an information model, the corresponding repository type library is attached to the root object through the **ReposTypeLibs** collection. This collection can be used to enumerate all of the information models (type data) that are contained in a repository.

- **Instance data navigation**

  Once an information model is defined, the repository can be populated with instance data. This instance data consists of objects and relationships that conform to the classes and relationship types of the information model.

  Because the objects are connected via relationships, you can navigate through this data. However, to enable general purpose repository browsers to navigate this data, the first navigational step must be from the root object of the repository through a root relationship collection to the primary objects of your information model. Primary objects are objects that make a good starting point for navigating to other objects of your information model.

  Because this root relationship collection is different for each information model, it must be defined by the information model. There are two options for attaching this relationship collection to the root object:

  a. The **ReposRoot** class implements the **IReposRoot** interface. This interface is provided to information model creators as a connection point. You can add your connecting relationship collection to this interface.

b. You can extend the **ReposRoot** class to implement a new interface that is defined in your information model. This interface implements a relationship collection that attaches the root object to the primary objects in your information model.

To facilitate navigation, the root object in all repositories always has the same object identifier. The symbolic name for this object identifier is OBJID_REPOSROOTOBJ.

When to Use

Use the **ReposRoot** class to:

- Obtain a starting point for navigating to objects in the repository.
- Create a new information model.
- Attach a relationship collection to the root object of the repository that connects to the primary objects of your information model.

Interfaces

Interface	Description
IAnnotationalProps	Gets and sets annotational properties
IManageReposTypeLib	Adds information models (repository type libraries) to a repository
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IRepositoryObject	Retrieves repository object identifiers
IRepositoryObjectStorage	Creates and loads repository objects
IReposRoot	Provides an attachment point for information model instance data
IVersionAdminInfo2	Retains properties inherited from **IVersionAdminInfo** and sets or retrieves version comments

IWorkspaceContainer	Manages the set of workspaces in a repository

See Also

IManageReposTypeLib Interface

ReposRoot Object

RTIM Classes

ReposTypeLib Class

There is one repository type library for every information model contained in a repository database. Each information model provides a logical grouping of all of the type definitions related to a particular application, tool, or tool set that you are developing. Repository type libraries are instances of the **ReposTypeLib** class.

To insert a new information model into a repository database, use the **ReposRoot** class.

When to Use

Use the **ReposTypeLib** class to:

- Define new classes, relationship types, and interfaces for an information model.
- Retrieve or modify the global identifier associated with a repository type library.
- Determine which type definitions are associated with a particular repository type library.

Interfaces

Interface	Description
IAnnotationalProps	Gets and sets annotational properties
INamedObject	Retrieves or sets the class name
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IRepositoryObject	Retrieves repository object identifiers
IRepositoryObjectStorage	Creates and loads repository objects
IReposTypeLib	Creates class, interface, and relationship

	definitions for a repository type library
IReposTypeLib2	Defines dependencies between information models
IVersionAdminInfo2	Retains properties inherited from **IVersionAdminInfo** and sets or retrieves version comments

See Also

ReposRoot Class

ReposTypeLib Object

RTIM Classes

ScriptDef Class

A script definition object represents Microsoft® ActiveX® script that you can associate with a method or property definition.

To support scripting for both method and property interface members, a **ScriptDef** object is associated at the interface member level. Because method and property definitions inherit from interface member objects, an interface member object provides the common ground where an association between script and interface members can be made.

Because interfaces can be aliased, derived, or otherwise reused, script definitions are linked through association to support the levels of indirection that are customary in COM programming. Associations are established through collections of classes, interfaces, and members that you define for each **ScriptDef** object.

During script invocation, the repository engine reads the collections to select a script definition most closely related to the interface. When the repository engine selects the closest script definition, it determines which method calls the script, on which interface, and on what class. The selection process enables support for two conditions that are common to C++ programming: inheriting a method or property implementation, and overriding a default implementation.

A method or property can be associated with the class and interface being executed, the interface being executed, or the closest ancestor that has the script. If a script cannot be selected, then the repository engine returns an error in the case of methods.

When to Use

Use a **ScriptDef** object to store the implementation of a method in an information model. You can also use **ScriptDef** to validate properties before storing them in a repository database.

You can implement methods or property validation rules that apply to:

- All classes that implement the interface.

- A specific class that implements the interface.

- A derived interface for those cases in which you want to override the implementation of a base interface method or property validation rule.

Each method or property can be associated with only one script definition. However, the same script definition can be associated with multiple methods and properties.

Interfaces

Interface	Description
INamedObject	Retrieves or sets the class name
IRepositoryDispatch	Provides enhanced dispatch support
IRepositoryItem	Manages repository objects and relationships
IReposTypeInfo	Relates class, interface, and relationship definition objects to information models
IScriptDef	Associates a Microsoft ActiveX script definition with a method

See Also

RTIM Classes

ScriptDef Object

Meta Data Services Programming

RTIM COM Interfaces

The Repository Type Information Model (RTIM) is the object model the repository engine uses to store information models. The RTIM interfaces expose the properties and methods that are used to programmatically create or extend an information model.

These interfaces build upon the interfaces that drive the repository engine. The repository engine interfaces are listed separately.

All RTIM interfaces expose the standard **IUnknown** and **IDispatch** interfaces, which provide fundamental COM and Automation support.

The following table lists RTIM interfaces alphabetically.

Interfaces	Description
IClassDef Interface	Adds interfaces to a class.
IClassDef2 Interface	Manages the collection of scripts that a class uses.
ICollectionDef Interface	Defines how instances of a particular type of collection will behave.
IEnumerationDef Interface	Defines enumeration objects.
IEnumerationValueDef Interface	Defines enumeration values.
IInterfaceDef Interface	Defines interface objects.
IInterfaceDef2 Interface	Supports interface implication between any two interfaces and aliasing.
IInterfaceMember Interface	Accesses the common properties of an interface member.
IInterfaceMember2 Interface	Allows classes, interfaces and relationships to be referred to by a second name or alias.
IManageReposTypeLib Interface	Creates a type library for storing information models.
IMethodDef Interface	Defines a list of parameters for a method definition.

IParameterDef Interface	Defines in detail each parameter of the method.
IPropertyDef Interface	Defines a property definition object.
IPropertyDef2 Interface	Contains an optional relationship to a single **EnumerationDef** object.
IReposRoot Interface	Provides a starting point to navigate to other objects in a repository.
IReposTypeInfo Interface	Determines which information models contain a particular class, interface, or relationship type.
IReposTypeInfo2 Interface	Allows classes, interfaces and relationships to be referred to by aliases.
IReposTypeLib Interface	Defines new classes, relationship types, and interfaces for an information model, and accesses the global identifier of repository type libraries.
IReposTypeLib2 Interface	Defines dependencies between information models for sharing model information.
IScriptDef Interface	Defines a script definition object.
IViewClassDef Interface	Defines database views for a class.
IViewInterfaceDef Interface	Defines a database view for all objects that implement a specific interface.
IViewPropertyDef Interface	Defines the column name of a property in the view.
IViewRelationshipDef Interface	Defines a junction table view of a relationship class. This is used for views that have many-to-many relationships.

See Also

IClassDef Interface

The **IClassDef** interface helps you create information models by adding interfaces to a class. To insert a new class definition into an information model, use the **IReposTypeLib** interface.

After you add all of the interfaces, you can complete a class definition by committing the transaction that brackets your class definition modifications.

When to Use

Use the **IClassDef** interface to:

- Add a new or existing interface to a class definition.
- Retrieve the global identifier for the class.
- Access the collection of interfaces that are part of a class definition.

Properties

Property	Description
ClassID	The global identifier of the class

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IClassDef method	Description
AddInterface	Adds an existing interface to the class definition
CreateInterfaceDef	Creates a new interface and adds it to the class definition
ObjectInstances	Materializes an **IObjectCol** interface pointer for the collection of all objects in the repository that conform to this class

Collections

Collection	Description

Interfaces	The collection of all interfaces that are implemented by a class

See Also

ClassDef Class

IReposTypeLib Interface

IClassDef::AddInterface

The **AddInterface** method adds an existing interface to the collection of interfaces that are implemented by a particular class.

HRESULT AddInterface(**IInterfaceDef** **plInterfaceDef*,
BSTR *Flags*
);

Parts

plInterfaceDef

[in]
The interface pointer for the interface that is to be added to the collection of interfaces implemented by this class.

Flags

[in]
If the interface that you are adding is the default interface for the class, pass in the string "Default". Otherwise, pass in a null string.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

When you indicate that an interface is the default interface for a class, you are actually setting the value of the **ImplementsOptions** annotational property on the **Class_Implements_Interface** relationship to TRUE.

See Also

IClassDef Interface

IInterfaceDef Interface

IClassDef ClassID Property

The global identifier that is assigned to this class.

Dispatch Identifier: DISPID_ClassID

Property Data Type: GUID

See Also

IClassDef Interface

IClassDef::CreateInterfaceDef

The **CreateInterfaceDef** method creates a new interface definition and adds the interface to the collection of interfaces implemented by the class.

HRESULT CreateInterfaceDef(**VARIANT** *sObjId,*
BSTR *Name,*
VARIANT *sIID,*
IInterfaceDef ***pIAncestor**,
BSTR *Flags,*
IInterfaceDef ***ppIInterfaceDef*
);

Parts

sIObjId

[in]
The object identifier to be assigned to the new interface definition object. If this parameter is set to OBJID_NULL, the repository engine assigns an object identifier for you.

Name

[in]
The name of the interface you are creating.

sIID

[in]
The global identifier associated with the signature for this interface. If there is none, set this parameter to zero.

**pIAncestor*

[in]
The interface pointer to the base interface from which the interface being added is derived.

Flags

[in]
If the interface that you are adding is the default interface for the class, pass in the string "Default". Otherwise, pass in a null string.

**pplInterfaceDef*

[out]
The interface pointer for the new interface.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

When you indicate that an interface is the default interface for a class, you are actually setting the value of the **ImplementsOptions** annotational property on the **Class_Implements_Interface** relationship to TRUE.

See Also

IClassDef Interface

IInterfaceDef Interface

IClassDef Interfaces Collection

The collection of all interfaces that are implemented by this class.

Dispatch Identifier: DISPID_Ifaces (32)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Class_Implements_Interface**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	Yes	The source object for the collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never

		sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-sensitive Names	Not applicable	Case-sensitive naming is not applicable for this collection.
Unique Names	Not applicable	Unique naming is not applicable for this collection.

See Also

IClassDef Interface

IClassDef::ObjectInstances

This method materializes an **IObjectCol** interface pointer for the collection of all objects in the repository that conform to this class.

HRESULT ObjectInstances(
IObjectCol ***ppIObjectCol*
);

Parts

**ppIObjectCol*

[out]
The interface pointer for the object collection.

Return Value

S_OK

The method completed successfully.

ErrorValues

This method failed to complete successfully.

Remarks

The retrieved collection contains one version of each object that conforms to the class. For each such object, the repository engine uses its resolution strategy to choose which version appears in the collection.

ObjectInstances is not workspace-scoped.

See Also

IClassDef Interface

Resolution Strategy for Objects and Object Versions

IClassDef2 Interface

The **IClassDef2** interface is derived from **IClassDef**, **IDepositoryDispatch**, and **IDispatch**.

This interface is used to manage the collection of scripts that a class uses.

When to Use

Use the **IClassDef** derived methods to manipulate the **ScriptsUsedByClass** collection.

Properties

None

Methods

None

Collections

Collection	Description
ScriptsUsedByClass	The collection of scripts that are used by this class

For more information about methods and properties for the functionality this interface provides, see IClassDef Interface.

ScriptsUsedByClass Collection

The collection of scripts being used by this class definition.

Dispatch Identifier: DISPID_IclassDef2_ScriptsUsedByClass (350)

ICollectionDef Interface

A collection type (also referred to as a collection definition) defines how instances of a particular type of collection behave. The properties of the collection type determine:

- The minimum and maximum number of items in a collection.
- Whether the collection type is an origin collection type.
- Whether the collection type permits the naming of destination objects and, if so, whether those names are case-sensitive and required to be unique.
- Whether the collection type permits the explicit sequencing of items in the collection.
- What happens to related objects when objects or relationships in the collection are deleted.
- Whether origin collections of this type are automatically copied to new object versions by the **CreateVersion** method.
- Whether the **MergeVersion** method combines origin collections of this type as a whole, or item by item.
- Whether the **FreezeVersion** method requires destination object versions of relationships of this type to be frozen before their origin object versions can be frozen.

The kind of relationship that a particular collection type uses to relate objects to

each other is determined by its **CollectionItem** collection. The **CollectionItem** collection associates a single relationship type to the collection type.

To add a new collection type, use the **IInterfaceDef** interface.

When to Use

Use the **ICollectionDef** interface to retrieve or modify the properties of a collection type or to determine the kind of relationship that the collection implements.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface

	pointer. The **IReposProperties** interface provides access to the **Properties** collection.

Properties

Property	Description
Flags	Flags that determine the behavior of this type of collection
IsOrigin	The indicator of whether collections of this type are origin collections
MaxCount	The maximum number of target objects that can be contained in a collection of this type
MinCount	The minimum number of target objects that must be contained in a collection of this type

Collections

Collection	Description
CollectionItem	The collection of one relationship type that defines the relationship between target objects of this type of collection and a single source object

See Also

CollectionDef Class

IInterfaceDef Interface

ICollectionDef Flags Property

For a particular type of collection, the **Flags** property determines:

- Whether the collection type permits the naming of destination objects and, if so, whether those names are case-sensitive and required to be unique.
- Whether the collection type permits the explicit sequencing of items in the collection.
- What happens to related objects when objects or relationships in the collection are deleted.
- Whether origin collections of this type are automatically copied to new object versions by the **CreateVersion** method.
- Whether the **MergeVersion** method combines origin collections of this type as a whole, or item by item.
- Whether the **FreezeVersion** method requires that destination object versions of relationships of this type be frozen before the attendant origin object versions can be frozen.

For more information about flag values and descriptions, see CollectionDefFlags Enumeration.

Dispatch Identifier: DISPID_ColFlags (54)

Property Data Type: **long**

See Also

ICollectionDef Interface

ICollectionDef IsOrigin Property

This property indicates whether collections of this type are origin collections.

Dispatch Identifier: DISPID_IsOrigin (57)

Property Data Type: **Boolean**

See Also

ICollectionDef Interface

ICollectionDef MaxCount Property

This property specifies the maximum number of target objects that can be contained in a collection of this type. This property is maintained for informational purposes. It is not enforced by the repository engine.

Dispatch Identifier: DISPID_MaxCount (56)

Property Data Type: **short**

See Also

ICollectionDef Interface

ICollectionDef MinCount Property

This property specifies the minimum number of target objects that must be contained in a collection of this type. This property is maintained for informational purposes. It is not enforced by the repository engine.

Dispatch Identifier: DISPID_MinCount (55)

Property Data Type: **short**

See Also

ICollectionDef Interface

ICollectionDef CollectionItem Collection

Every **RelationshipDef** object has two **CollectionDef** objects. Therefore, every relationship definition instance can be navigated in one of two directions. That is, from a **RelationshipDef** object, you can navigate to its collection of **CollectionDef** objects. Conversely, you can navigate in the opposite direction; that is, from a **CollectionDef** object to the associated **RelationshipDef** object. To navigate in the opposite direction, use the **CollectionItem** collection on the **ICollectionDef** interface. For more information about relationships and collections, see Repository Object Architecture.

Dispatch Identifier: DISPID_CollectionItem (38)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Collection_Contains_items**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	Yes	The source object for the collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	One	The maximum number of items that can be contained in the

		collection is one.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-sensitive Names	Not applicable	Case-sensitive naming is not applicable for this collection.
Unique Names	Not applicable	Unique naming is not applicable for this collection.

See Also

ICollectionDef Interface

RelationshipDef ItemInCollections Collection

IEnumerationDef Interface

The **IEnumerationDef** interface is derived from **IRepositoryDispatch**, which inherits from **IDispatch**. The **IEumerationDef** interface is implemented by the **EnumerationDef** class.

When to Use

IEnumerationDef is the default interface for **Enumeration** objects. Use this interface for defining new enumeration values.

Properties

Property	Description
IsFlag	Indicates that the enumeration defines a logical flag. The selected enumeration values should be combined logically using OR. This property applies only to numeric enumeration values.

There are no methods associated with this interface.

Collections

Property	Description
Values	The collection of **EnumerationValue** objects

See Also

IEnumerationValueDef Interface

IPropertyDef2 Interface

IRepositoryDispatch

Repository Enumeration Definition

IEnumerationDef Values Collection

A collection of **EnumerationValue** objects.

Dispatch Identifier: DISPID_IEnumerationDefIsFlag (365)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**EnumerationDef_Valuesof_EnumerationValueDef**	This is the type of relationship by which all items of the collection ar connected to a common source objec
Source Is Origin	Yes	The source object for th collection is also the orig object.
Minimum Collection Size	One	The minimu number of items that must be contained in the collectio is one.
Maximum Collection	Many	The maximum

Size		number of items that ca be containe in the collection is unlimited.
Sequenced Collection	No	Because it is destination collection, this collecti does not hav an explicitly defined sequence. Collections origin objec are never sequenced.
Deletes Propagated	Yes	Deleting an origin objec or a relationship the collectio causes the deletion of a correspondi destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-sensitive	No	The collecti

Names		does not permit the u of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of destination object be unique with the collectio of destinatio objects.

See Also

IEnumerationValueDef Interface

Repository Enumeration Definition

IEnumerationValueDef Interface

The **IEnumerationValueDef** interface is derived from **IRepositoryDispatch** and **IDispatch**, and is implemented by the **EnumerationValue** class.

Properties

Property	Description
EnumValue	A string containing a value that may be stored in the property value of an object.

There are no methods or collections associated with this interface.

See Also

IEnumerationDef Interface

IPropertyDef2 Interface

IRepositoryDispatch

Repository Enumeration Definition

IEnumerationValueDef::EnumValue

The **EnumValue** property contains a value that may be stored as the property value of an object.

Dispatch Identifier: DISPID_IEnumerationValueDefValue (371)

See Also

IEnumerationValueDef Interface

IInterfaceDef Interface

The properties, methods, and collections that a class implements are organized into functionally related groups. Each group is implemented as a COM interface. Each COM interface that you create can have members consisting of properties, methods, and collections. An interface definition is the template to which that interface conforms.

To add a new interface to the repository, use the **IClassDef** interface or the **IReposTypeLib** interface.

When to Use

Use the **IInterfaceDef** interface to:

- Retrieve or modify properties of an interface definition.
- Determine which members are attached to an interface definition.
- Determine which classes implement an interface.
- Determine the base interface from which an interface derives.
- Determine which interfaces derive from a particular interface.
- Determine which repository objects expose a particular interface.
- Add a new property, method, or collection type to an interface definition.

Properties

Property	Description
Flags	The flags that specify whether the interface is extensible, and whether the interface should be visible to Automation interface queries.
InterfaceID	The global interface identifier for the interface.
TableName	The name of the SQL table that is used to store instance information for the properties of the interface.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers.
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an Automation object.

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IInterfaceDef method	Description
CreateMethodDef	Creates a new method definition, and attaches it to the interface definition.
CreatePropertyDef	Creates a new property definition, and attaches it to the interface definition.
CreateRelationshipColDef	Creates a relationship collection type. The collection type is attached to the interface definition.
ObjectInstances	Materializes an **IObjectCol** interface pointer for the collection of all objects in a repository that expose this interface.

Collections

Collection	Description
Ancestor	The collection of one base interface from which this interface inherits.
Classes	The collection of classes that implement the interface.
Descendants	The collection of other interfaces that derive from this interface.
Members	The collection of members that are attached to the interface definition.

See Also

IClassDef Interface

IInterfaceDef Interface

IReposTypeLib Interface

IInterfaceDef Flags Property

This property contains flags that specify whether the interface is extensible, and whether the interface should be visible to Automation interface queries. For more information about flag values and descriptions, see InterfaceDefFlags Enumeration.

Dispatch Identifier: DISPID_IfaceFlags (50)

Property Data Type: **long**

See Also

IInterfaceDef Interface

IInterfaceDef InterfaceID Property

This property is the global interface identifier for the interface.

Dispatch Identifier: DISPID_InterfaceID (48)

Property Data Type: GUID

See Also

IInterfaceDef Interface

IInterfaceDef TableName Property

This property is the name of the SQL table that is used to store instance information for the properties of the interface. The length of the name must be 30 characters or less.

Dispatch Identifier: DISPID_TableName (49)

Property Data Type: **string**

See Also

IInterfaceDef Interface

IInterfaceDef::CreateMethodDef

This method creates a new method definition and attaches it to the interface definition.

HRESULT CreateMethodDef(**VARIANT** *sObjId***,**
BSTR *Name***,**
long *iDispId***,**
IInterfaceMember ***ppIMethodDef*
);

Parts

sObjId

[in]
The object identifier to be used for the new method definition object. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.

Name

[in]
The name of the new method.

iDispId

[in]
The dispatch identifier to be used for accessing the new method.

**ppIMethodDef* [out]

The interface pointer for the newly created method definition.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IInterfaceDef Interface

IInterfaceMemberInterface

IInterfaceDef::CreatePropertyDef

This method creates a new property definition and attaches it to the interface definition.

HRESULT CreatePropertyDef (**VARIANT** *sObjId***,**
BSTR *Name***,**
long *iDispId***,**
short *iCType***,**
IPropertyDef ***ppIPropertyDef*
);

Parts

sObjId

[in]
The object identifier to be used for the new property definition object. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.

Name

[in]
The name of the new property.

iDispId

[in]
The dispatch identifier to be used for accessing the new property.

iCType [in]

The C data type of the property. For more information, including a definition of valid values, see the ODBC documentation.

**ppIPropertyDef* [out]

The interface pointer for the newly created property definition.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IInterfaceDef Interface

IPropertyDef Interface

IInterfaceDef::CreateRelationshipColDef

This method creates a new collection type, attaches it to this interface, and associates it with the specified relationship type.

HRESULT CreateRelationshipColDef(**VARIANT** *sObjId***,**
BSTR *Name***,**
long *iDispId***,**
boolean *IsOrigin***,**
short *fFlags***,**
IReposTypeInfo ***pIRelshipDef***,**
ICollectionDef ****pICollectionDef***
);

Parts

sObjId [in]
The object identifier for the collection type. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.

Name

[in]
The name of the new collection type.

iDispId [in]

The dispatch identifier to be used for Automation access to collections of this type.

IsOrigin [in]

Specifies whether collections of this type are origin collections.

fFlags

[in]
Flags that specify naming, sequencing, and delete propagation behavior for the collection type. For more information about flag values and descriptions,

see CollectionDefFlags Enumeration.

**pIRelshipDef* [in]

The interface pointer for the relationship definition object to which this collection type is connected.

**ppICollectionDef*

[out]
The interface pointer for the new collection definition object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

By default, the collection definition specifies that zero to many items are permitted in collections of this type. To specify a different minimum and maximum item count for the new collection type, change the **MinCount** and **MaxCount** properties before committing the transaction that contains this method invocation.

See Also

ICollectionDef Interface

IInterfaceDef Interface

RelationshipDef Class

IInterfaceDef::ObjectInstances

This method materializes an **IObjectCol** interface pointer for the collection of all objects in the repository that expose the current interface.

HRESULT ObjectInstances(IObjectCol *****ppIObjectCol*
);

Parts

**ppIObjectCol* [out]

The interface pointer for the object collection.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

Remarks

The retrieved collection contains one version of each object that conforms to a class that exposes the current interface. For each such object, the repository engine uses its resolution strategy to choose which version appears in the collection.

ObjectInstances is not workspace scoped.

See Also

IInterfaceDef Interface

Resolution Strategy for Objects and Object Versions

IInterfaceDef Classes Collection

This collection specifies which classes implement the interface.

Dispatch Identifier: DISPID_Classes (33)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Class_Implements_Interface**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	No	The source object for the collection is not the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.

Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-sensitive Names	Not applicable	Case-sensitive naming is not applicable for this collection.
Unique Names	Not applicable	Unique naming is not applicable for this collection.

See Also

IInterfaceDef Interface

IInterfaceDef Members Collection

This collection specifies which members are attached to the interface.

Dispatch Identifier: DISPID_Members (36)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Interface_Has_Members**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	Yes	The source object for the collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	Yes	As a destination collection, this collection permits an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	The deletion of an origin

		object or relationship in the collection causes the deletion of the corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-Sensitive Names	No	The relationship type for the collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects. This applies to collections whose relationship type permits destination objects to be named.

See Also

IInterfaceDef Interface

IInterfaceDef Ancestor Collection

This collection specifies the one base interface from which this interface derives. You use **Ancestor** collections to define inheritance.

Dispatch Identifier: DISPID_Ancestor (34)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Interface_InheritsFrom_Interface**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	Yes	The source object for the collection is also the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum Collection Size	One	The maximum number of items that can be contained in the collection is one.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence.

		Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-Sensitive Names	Not applicable	Case-sensitive naming is not applicable for this collection.
Unique Names	Not applicable	Unique naming is not applicable for this collection.

See Also

IInterfaceDef Interface

IInterfaceDef Descendants Collection

This collection specifies other interfaces that derive from this interface.

Dispatch Identifier: DISPID_Descendants (35)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Interface_InheritsFrom_Interface**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly

		defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-Sensitive Names	Not applicable	Case-sensitive naming is not applicable for this collection.
Unique Names	Not applicable	Unique naming is not applicable for this collection.

See Also

IInterfaceDef Interface

IInterfaceDef2 Interface

The **IInterfaceDef2** interface inherits from **IInterfaceDef**. It helps you define implication between two interfaces in the form of "Interface I1 implies Interface I2," which means that any class that implements I1 also implements I2. There is a many-to-many relationship named **Interface_Implies_Interface** that relates multiple instances of **IInterfaceDef2** to other instances of itself. Therefore, **IInterfaceDef2** has two collections, **Implies** and **ImpliedBy**, which are the two sides of the relationship.

IInterfaceDef2 also provides the **CreateAlias** method, which adds an alias member to the interface definition.

When to use

Use the **IInterfaceDef2** interface to:

- Define implication between two interfaces.
- Create and add alias members to the interface definition.

Properties

IInterfaceDef property	Description
Flags	Flags that specify whether the interface is extensible, and whether the interface should be visible to Automation interface queries
InterfaceID	The global interface identifier for the interface
TableName	The name of the SQL table that is used to store instance information for the properties of the interface

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IInterfaceDef method	Description
CreateMethodDef	Creates a new method definition, and attaches it to the interface definition.
CreatePropertyDef	Creates a new property definition, and attaches it to the interface definition.
CreateRelationshipColDef	Creates a relationship collection type. The collection type is attached to the interface

	definition.
ObjectInstances	Materializes an **IObjectCol** interface pointer for the collection of all objects in a repository that expose this interface.

IInterfaceDef2 method	Description
CreateAlias	Adds an alias member to the interface definition

Collections

IInterfaceDef collection	Description
Ancestor	The collection of one base interface from which this interface derives
Classes	The collection of classes that implement the interface
Descendants	The collection of other interfaces that derive from this interface
Members	The collection of members that are attached to the interface definition

IInterfaceDef2 collection	Description
Implies	The collection of one interface that implies other interfaces
ImpliedBy	The collection of interfaces implied by another interface

See Also

IClassDef Interface

IInterfaceDef Interface

InterfaceDef Class

IReposTypeLib Interface

IInterfaceDef2 Implies Collection

The collection of one interface that implies other interfaces.

Dispatch Identifier: DISPID_Implies (95)

See Also

IInterfaceDef2 Interface

IInterfaceDef2 ImpliedBy Collection

IInterfaceDef2 ImpliedBy Collection

The collection of interfaces implied by another interface.

Dispatch Identifier: DISPID_ImpliedBy (96)

See Also

IInterfaceDef2 Interface

IInterfaceDef2 Implies Collection

IInterfaceDef2::CreateAlias

The **CreateAlias** method is used in member delegation to add an alias member to the interface definition. The repository engine does not prevent the creation of duplicate alias names. If you want to avoid duplicate aliases, you must verify that the alias name is unique.

This method has the following syntax:

HRESULT CreateAlias(
VARIANT *sObjID***,**
BSTR *Name***,**
long *iDispID***,**
IInterfaceMember **pIIfaceMemBase***,*
IInterfaceMember2 ***ppIIfaceMem2***);**

Parts

sObjID

[in]
The object identifier to be used with the new alias member.

Name

[in]
The name of the new alias member.

iDispID

[in]
The dispatch identifier to be used for accessing the new alias member.

pIIfaceMemBase

[in]
The interface pointer for the base member.

**ppIIfaceMem2*

[out, retval]
The interface pointer for the derived member.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IInterfaceDef2 Interface

Member Delegation

IInterfaceMember Interface

The properties, methods, and collections that a class implements are organized into functionally related groups. Each group is implemented as a COM interface. The properties, methods, and collections of each interface are members of the interface.

The **IInterfaceMember** interface maintains this information for an interface member:

- The member dispatch identifier.
- Information about member visibility.
- The relationship to the interface that exposes a particular interface member.

This information is common to properties, methods, and collection types. The **PropertyDef**, **MethodDef**, and **CollectionDef** classes all implement this interface.

When to Use

Use the **IInterfaceMember** interface to access the common properties of an interface member, or to determine which interface definition has a member of a particular property, method, or collection type.

Properties

Properties	Description
DispatchID	The dispatch identifier to use when accessing an instance of this type of member
Flags	The flags that specify details about this type of member

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

Collections

Collection	Description

Interface	The collection of one interface that exposes this type of member

See Also

CollectionDef Class

IInterfaceMember2 Interface

MethodDef Class

PropertyDef Class

IInterfaceMember DispatchID Property

This property contains the dispatch identifier to use when accessing an instance of this type of member.

Dispatch Identifier: DISPID_DispID (51)

Property Data Type: **long**

See Also

IInterfaceMember Interface

IInterfaceMember Flags Property

This property contains a flag that specifies whether or not the interface member should be visible to Automation queries. For more information about flag values and descriptions, see InterfaceMemberFlags Enumeration.

Dispatch Identifier: DISPID_IfaceMemFlags (52)

Property Data Type: **long**

See Also

IInterfaceMember Interface

IInterfaceMember Interface Collection

For a particular property, method, or collection definition, the **Interface** collection specifies which interface exposes a member of this type.

Dispatch Identifier: DISPID_Iface (37)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Interface_Has_Members**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum Collection Size	One	The maximum number of items that can be contained in the collection is one.
Sequenced Collection	Yes	As a destination collection, this collection permits an explicitly defined sequence. Collections of

		origin objects are never sequenced.
Deletes Propagated	Yes	The deletion of an origin object or relationship in the collection causes the deletion of the corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-Sensitive Names	No	The relationship type for the collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects. This applies to collections whose relationship type permits destination objects to be named.

See Also

IInterfaceMember Interface

IInterfaceMember2 Interface

This interface is used to support aliasing. You can use this interface to allow **PropertyDef**, **MethodDef**, **Alias**, and **CollectionDef** objects to be referred to by a second name or alias.

The **Alias** class implements **IInterfaceMember2** as its default interface. Instances of the **Alias** class are simple, derived members. The **CollectionDef** class also implements **IInterfaceMember2** in order to support the semantically richer kind of derived member.

This interface inherits from **IInterfaceMember**. It also uses methods exposed through **IRepositoryDispatch**. For more information, see IRepositoryDispatch Interface.

When to Use

Use the **IInterfaceMember2** interface to:

- Create simple, derived members as instances from the **Alias** class.
- Create semantically rich derived members as instances from the **CollectionDef** class.

Properties

Property	Description
MemberSynonym	A string used as an alias.

Collections

Collection	Description
ScriptsUsedByMember	A collection that contains a script definition object.

ServicedByBaseMember	The base member that provides implementation to a derived member.
ServicesDerivedMembers	The derived interface member that receives its implementation from a base member on another interface.

See Also

Creating a Derived Member

IInterfaceMember Interface

IReposTypeInfo2 Interface

IInterfaceMember2 MemberSynonym Property

Use this property to create an alias of an interface member. If this property is set, the alias name can reference the **InterfaceDef**, **PropertyDef**, **CollectionDef**, **MethodDef** and **Alias** classes.

The maximum length of this string is 255 characters.

Dispatch Identifier: DISPID_MemberSynonym (394)

Remarks

The repository engine does not allow duplicate synonym values for **InterfaceDef**, **PropertyDef**, **CollectionDef**, **MethodDef**, or **Alias** classes.

See Also

IInterfaceMember2 Interface

Type Information Aliasing

IInterfaceMember2 ScriptsUsedByMember Collection

A **ScriptsUsedByMember** collection contains the interface member (either a method definition or a property definition) that uses a script for its implementation.

This collection is the destination collection of a relationship that associates a script with an interface member. The origin collection of this relationship is the **UsingMembers** collection of the **ScriptDef** object.

Dispatch Identifier: DISPID_IInterfaceMember2ScriptsUsedByMember (356)

See Also

Defining Script Objects

IInterfaceMember2 Interface

IScriptDef Interface

IInterfaceMember2 ServicedByBaseMember Collection

A **ServicedByBaseMember** collection contains the base member that provides the implementation for a member on another interface.

This collection can contain one interface member object. This collection is the origin of a relationship collection that maps the correspondence between the base member and an alias. When you populate this collection, you must also populate the **ServicesDerivedMember** collection to complete the relationship.

Dispatch Identifier: DISPID_IInterfaceMember2ServicedByBaseMember (100)

See Also

Creating a Derived Member

IInterfaceMember2 Interface

IInterfaceMember2 ServicesDerivedMembers Collection

Type Information Aliasing

IInterfaceMember2 ServicesDerivedMembers Collection

A **ServicesDerivedMember** collection contains the derived interface member that receives its implementation from a base member on another interface.

This collection can contain one interface member object. This collection is the destination of a relationship collection that maps the correspondence between the base member and an alias. When you populate this collection, you must also populate the **ServicedByBaseMember** collection to complete the relationship.

Dispatch Identifier: DISPID_IInterfaceMember2ServicesDerivedMembers (99)

See Also

Creating a Derived Member

IInterfaceMember2 Interface

IInterfaceMember2 ServicedByBaseMember Collection

Type Information Aliasing

IManageReposTypeLib Interface

Each information model that is stored in the repository is represented by a repository type library.

When to Use

Use the **IManageReposTypeLib** interface to:

- Create a repository type library for a new information model.
- Determine which information models are currently stored in the repository.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods

	exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IManageReposTypeLib method	Description
CreateTypeLib	Creates a repository type library for a new information model

Collections

Collection	Description
ReposTypeLibs	The collection of repository type libraries that are currently stored in a repository database

See Also

ReposRoot Class

IManageReposTypeLib::CreateTypeLib

This method creates a new repository type library and attaches it to the root of the repository. Each repository type library represents an information model.

HRESULT CreateTypeLib(**VARIANT** *sObjId***,**
BSTR *Name***,**
VARIANT *TypeLibId***,**
IReposTypeLib ***ppIRepTypeLib*
);

Parts

sObjId

[in]
The object identifier to be used for the new repository type library object. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.

Name

[in]
The name of the new repository type library.

TypeLibId

[in]
The global identifier by which this repository type library is referenced.

**ppIRepTypeLib* [out]
The IReposTypeLib interface pointer to the new repository type library object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IManageReposTypeLib Interface

IReposTypeLib Interface

IManageReposTypeLib ReposTypeLibs Collection

The collection of repository type libraries that are currently stored in the repository database. Each repository type library represents an information model.

Dispatch Identifier: DISPID_ReposTypeLibs (40)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**TblManager_ContextFor_ReposTypeLibs**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	Yes	The source object for the collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.

Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	Deleting an origin object or a relationship in the collection causes the deletion of a corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.

Case-Sensitive Names	No	This collection does not use case-sensitive names for destination objects.
Unique Names	Yes	The collection requires that the name of a destination object be unique within the collection of destination objects.

See Also

IManageReposTypeLib Interface

IMethodDef Interface

The **IMethodDef** interface, which inherits from **IInterfaceMember**, allows the model creator to define an ordered list of parameter definitions for a method. The **IMethodDef** interface is the default interface of the **MethodDef** class returned by the **IInterfaceDef::CreateMethodDef** method.

When to use

Use the **IMethodDef** interface to:

- Define a list of parameter definitions for a method.
- Generate fully descriptive Interface Definition Language (IDL) files from the information model.

Properties

IInterfaceMember property	Description
DispatchID	The dispatch identifier to use when accessing a **MethodDef** instance
Flags	Flags that specify details about a **MethodDef** instance

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IMethodDef method	Description
CreateParameterDef	Creates a new parameter definition and attaches it to the end of the parameter list for the particular method

Collections

IInterfaceMember collection	Description
Interface	The collection of one interface that exposes this type of member

IMethodDef collection	Description
Parameters	The collection of parameter definition objects that provide parameters to this method

See Also

CollectionDef Class

IParameterDef Interface

MethodDef Class

PropertyDef Class

IMethodDef::CreateParameterDef

This method creates a new parameter definition and attaches it to the end of the parameter list for the particular method.

HRESULT CreateParameterDef **(**
  VARIANT *sObjID*,
  BSTR *Name*,
  long *Type*,
  long *Flags*,
  BSTR *Description*,
  BSTR *Default*,
  IParameterDef ***pParamDef*
);

Parts

sObjID

[in]
Object ID for the parameter.

Name

[in]
Name of the parameter.

Type

[in]
Type of the parameter. This should be one of the VT_XXXX values defined by OLE Automation.

Flags

[in]
A flag that can take one of the following values. Enumerated values are defined through the **IParameterDef Flags** property.

PARAMFLAGS_IN = 1
PARAMFLAGS_OUT = 2
PARAMFLAGS_RETVAL = 4
PARAMFLAGS_OPTIONAL = 8

Description

[in]
Text inserted into the type library to define the parameter type.

Default

[in]
This is inserted into the type library to define the default value of the parameter.

**ppParamDef*

[out]
A pointer to the **IParameterDef** interface that returns the newly created parameter definition.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IMethodDef Interface

IParameterDef Flags Property

MethodDef Class

IMethodDef Parameters Collection

This collection contains the parameter definition objects that you define for the current method definition.

If you use **CreateParameterDef** method to create the parameter, the parameter is automatically added to this collection for the current method definition.

This collection is sequenced and it must contain only uniquely named items.

See Also

IMethodDef Interface

IMethodDef::CreateParameterDef

MethodDef Class

IParameterDef Interface

The **IParameterDef** interface allows the model creator to define, in detail, each parameter of the method that uses the interface properties listed in this topic. Parameter definitions are stored in a **RTblParameterDef** table in the repository database.

When the engine receives a call to a method defined through these interfaces, it returns **E_NOTIMPL**.

Properties

Property	Description
Type	The data type of the parameter.
Flags	A flag that defines whether the parameter is the default parameter. It also defines whether it is passed by reference or by value.
Description	A string (of 255 characters maximum) to be placed into the IDL file instead of the default text for the parameter type.
Default	A string (of 255 characters maximum) that denotes the default value for the parameter.
GUID	A GUID that defines the interface ID of a **VT_DISPATCH** or **VT_UNKNOWN** object.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch emthod	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

See Also

CollectionDef Class

IInterfaceMember DispatchID Property

IInterfaceMember Flags Property

IInterfaceMember Interface Collection

IMethodDef Interface

IParameterDef Interface

MethodDef Class

PropertyDef Class

Repository SQL Tables

IParameterDef Type Property

This property is the data type of the parameter, which can be any variable type supported by Automation.

Dispatch Identifier: DISPID_ParDefType (104)

Property Data Type: long

See Also

IMethodDef Interface

IParameterDef Interface

MethodDef Class

IParameterDef Flags Property

This property supports flags that define whether the parameter is an optional parameter. It also defines whether it is passed by reference or value, and specifies which of the parameters is a return value.

The trailing parameters are optional. Only one parameter can be marked as a return value.

Dispatch Identifier: DISPID_ParDefFlags (103)

Property Data Type: **long**

The flag can take one of the following values.

Flag value	Description
PARAMFLAGS_IN = 1	Passed by value
PARAMFLAGS_OUT = 2	Passed by reference
PARAMFLAGS_RETVAL = 4	A return value
PARAMFLAGS_OPTIONAL = 8	Optional parameter

See Also

IMethodDef Interface

IParameterDef Interface

MethodDef Class

IParameterDef Description Property

This property is a string (of 255 characters maximum) that can be placed in an IDL file, providing a more meaningful description than that provided through the default text for the parameter type. This allows parameters of the type VT_DISPATCH to be defined, even though the IDL file contains text like "**IRepositoryObject** *".

Dispatch Identifier: DISPID_ParDefDesc (105)

Property Data Type: **string**

See Also

IMethodDef Interface

IParameterDef Interface

MethodDef Class

IParameterDef Default Property

This property is a string (of 255 characters maximum) that denotes the default value for the parameter.

Dispatch Identifier: DISPID_ParDefDefault (106)

Property Data Type: **string**

See Also

IMethodDef Interface

IParameterDef Interface

MethodDef Class

IParameterDef GUID Property

This property is a GUID that defines the interface ID of a VT_DISPATCH or VT_UNKNOWN object. This cannot be set through the **CreateParameterDef** method.

Dispatch Identifier: DISPID_ParDefGUID (107)

Property Data Type: **string**

See Also

IParameterDef Interface

IMethodDef Interface

MethodDef Class

IPropertyDef Interface

A property definition object specifies the characteristics of a particular type of property. These characteristics are defined by the properties of the property definition object.

To create a new property definition

1. Use the **CreatePropertyDef** method of the **IInterfaceDef** interface.

2. Define any non-default characteristics of your new property definition by manipulating the properties of the property definition object.

3. Commit your changes to the repository database.

Use the **IPropertyDef** interface to retrieve or modify the characteristics of a property definition.

Properties

Property	Description
APIType	The C data type of the property
ColumnName	The name of the column in the SQL table for this property
Flags	Specifies details about the property
SQLScale	The number of digits to the right of the decimal point for a numeric property
SQLSize	The size in bytes of the property
SQLType	The SQL data type of the property

Methods

IUnknown method	Description

QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

See Also

IInterfaceDef CreatePropertyDef Method

PropertyDef Class

IPropertyDef APIType Property

This property contains the C data type of the property definition object. For a definition of valid values, see the ODBC documentation.

Dispatch Identifier: DISPID_APIType (59)

Property Data Type: **short**

See Also

IPropertyDef Interface

IPropertyDef ColumnName Property

This property specifies the name of the column in the SQL table for the property definition object. A SQL table is used to store instance information for the properties of an interface. By default, there is a column in this table for each property that is defined as a member of the interface. The length of the column name must be 30 bytes or less.

Dispatch Identifier: DISPID_ColumnName (58)

Property Data Type: **string**

See Also

IPropertyDef Interface

IPropertyDef Flags Property

This property supports a flag that is used to create annotational properties. In this release, the repository engine ignores this flag. By default, a column is created for each property. This flag is preserved for backward compatibility.

In earlier versions, this flag specified whether to create a column for the property. Column creation occurred in the SQL table that provided persistent storage for the interface to which the property was attached. Without a column, instances of the property attached only to individual objects when setting the property value for that particular object.

Dispatch Identifier: DISPID_ColFlags (54)

Property Data Type: **long**

See Also

IPropertyDef Interface

IPropertyDef SQLScale Property

This property sets the number of digits to the right of the decimal point for a numeric property definition object. This parameter is ignored unless the **SQLType** property specifies an SQL_NUMERIC, SQL_DECIMAL, or SQL_TIME data type.

Dispatch Identifier: DISPID_SQLScale (62)

Property Data Type: **short**

See Also

IPropertyDef Interface

IPropertyDef SQLSize Property

This property sets the size in bytes of the property definition object. This property is ignored when the data type of the property inherently specifies the size of the property.

Dispatch Identifier: DISPID_SQL_Size (61)

Property Data Type: **short**

Note If **SQLSize** is set to a value greater than 65535, the repository engine divides the entered number by 65536 and sets **SQLSize** to the value of the remainder of the division, but no error is returned.

See Also

IPropertyDef Interface

IPropertyDef SQLType Property

This property sets the SQL data type of the property definition object. For more information, including a definition of valid values, see the ODBC documentation.

Dispatch Identifier: DISPID_SQLType (60)

Property Data Type: **short**

See Also

IPropertyDef Interface

IPropertyDef2 Interface

The **IPropertyDef2** interface is derived from **IRepositoryDispatch**, which inherits from **IDispatch**, and is implemented by the **PropertyDef** class.

IPropertyDef2 supports the definition of enumerated properties. When you create an enumerated object, you can associate it with a **PropertyDef** object through the **IPropertyDef2** interface.

When to Use

The **IPropertyDef2** interface contains an optional relationship to a single **EnumerationDef** object.

There are no methods or properties associated with this interface. For more information, see IRepositoryDispatch.

Properties

Property	Description
SQLBlobSize	The SQL binary large object (BLOB) size of the property

Collections

IPropertyDef2 collection	Description
EnumerationDef	The collection of enumerated objects that are associated with a property definition object

IPropertyDef2 SQLBlobSize Property

This property contains the SQL Binary Large Object (BLOB) size. When **SQLType** is set to SQL_LONGVARBINARY or SQL_LONGVARCHAR, the **SQLBlobSize** (rather than **SQLSize)** determines the size of a property that is a BLOB. For a definition of valid values, see the ODBC documentation.

Dispatch Identifier: DISPID_SQLBlobSize (87)

Property Data Type: **long**

See Also

IPropertyDef SQLSize Property

IPropertyDef SQLType Property

IPropertyDef2 Interface

Programming BLOBs and Large Text Fields

IPropertyDef2 EnumerationDef Collection

This is a collection of **EnumerationDef** objects.

Dispatch Identifier: DISPID_IPropertyDef2_EnumerationDef = 373

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**PropertyDef_EnumerationFor_EnumerationDef**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	Yes	The source object for the collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection	One	The maximum

Size		number of items that can be contained in the collection is one.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.

Case-sensitive Names	No	The collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects.

See Also

IEnumerationDef Interface

IEnumerationValueDef Interface

IPropertyDef2 Interface

Repository Enumeration Definition

IReposRoot Interface

The **IReposRoot** interface is a placeholder interface; it contains no properties, methods, or collections beyond Automation dispatch methods. It is provided as a convenient connection point to the root object. When you create an information model, you can attach to this interface a relationship collection that provides a navigational connection to the primary objects of your information model.

When to Use

Use the **IReposRoot** interface as a starting point to navigate to other objects in the repository.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

See Also

ReposRoot Class

IReposTypeInfo Interface

This interface relates class, interface, and relationship definition objects to repository type libraries.

When to Use

Use the **IReposTypeInfo** interface to:

- Determine which repository type libraries contain a particular class, interface, or relationship type.

- Determine what collection types are associated with a particular relationship type.

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers.
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface.
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Invoke	Provides access to properties and methods exposed by an Automation object.

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

Collections

Collection	Description
ItemInCollections	The origin and destination collection types that are connected to a relationship definition object.
ReposTypeLibScopes	The collection of repository type libraries that contain a particular class, interface, or relationship type.

See Also

ClassDef Class

InterfaceDef Class

RelationshipDef Class

IReposTypeInfo ItemInCollections Collection

This collection contains the origin and destination collection types that are associated with a particular relationship type. This collection is empty for definition objects that are not relationship definitions.

Dispatch Identifier: DISPID_Collection (39)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**Collection_Contains_items**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum Collection Size	Two.	The maximum number of items that can be contained in the collection is two.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.

Deletes Propagated	No	Deleting an origin object or a relationship in the collection does not cause the deletion of a corresponding destination object.
Destinations Named	No	The relationship type for the collection does not permit the naming of destination objects.
Case-Sensitive Names	Not applicable	Case-sensitive naming is not applicable for this collection.
Unique Names	Not applicable	Unique naming is not applicable for this collection.

See Also

IReposTypeInfo Interface

IReposTypeInfo ReposTypeLibScopes Collection

The collection of repository type libraries that contain a particular class, interface, or relationship type.

Dispatch Identifier: DISPID_ReposTypeLibScopes (43)

Collection characteristic	Value	Description
Relationship Type	**ReposTypeLib_ScopeFor_ReposTypeInfo**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.
Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.

Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	Deleting an origin object or a relationship in the collection causes the deletion of a corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-Sensitive Names	No	The collection does not permit the use of case-sensitive names for destination objects.

Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects.

See Also

IReposTypeInfo Interface

IReposTypeInfo2 Interface

This interface exposes methods that allow you to create synonyms for an existing class, interface, relationship, or enumeration definition object. The **IReposTypeInfo2** interface inherits from the **IReposTypeInfo** interface.

When to Use

Use the **IReposTypeInfo2** interface to allow classes, interfaces, and relationships to be referred to by multiple names as aliases.

Properties

Property	Description
Synonym	A string used as an alias name

See Also

IInterfaceMember2 Interface

Type Information Aliasing

IReposTypeInfo2 Synonym Property

This property contains a string that is used as an alias name. You can use this property to define an alias for **ClassDef**, **RelationshipDef**, **InterfaceDef**, and **EnumerationDef** classes. The maximum length of this string is 255 characters.

Dispatch Identifier: DISPID_Synonym (393)

Remarks

The repository engine does not check for duplicate synonym values. If you want unique synonyms, you must check for duplicate values first.

This interface uses methods exposed through **IRepositoryDispatch**. For more information, see IRepositoryDispatch Interface.

See Also

IReposTypeInfo2 Interface

Type Information Aliasing

IReposTypeLib Interface

There is one repository type library for every information model contained in the repository. Each information model provides a logical grouping of all of the type definitions related to a particular tool (or tool set).

To add a new repository type library to the repository, use the **IManageReposTypeLib** interface.

When to Use

Use the **IReposTypeLib** interface to:

- Define new classes, relationship types, and interfaces for an information model.
- Retrieve or modify the global identifier associated with a repository type library.
- Determine which type definitions are associated with a particular repository type library.

Properties

Property	Description
TypeLibID	The global identifier for the repository type library

Methods

IUnknown method	Description
QueryInterface	Returns pointers to supported interfaces

AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invokes	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IReposTypeLib method	Description
CreateClassDef	Creates a new class definition object
CreateInterfaceDef	Creates a new interface definition object
CreateRelationshipDef	Creates a new relationship definition object

Collections

Collection	Description
ReposTypeInfos	The collection of all classes, interfaces, and

	relationship types that are defined in the repository type library
ReposTypeLibContexts	The collection of one repository root object that is the context for the repository type library

See Also

IManageReposTypeLib Interface

ReposTypeLib Class

IReposTypeLib TypeLibID Property

This property is the global identifier for the repository type library.

Dispatch Identifier: DISPID_TypeLibID (64)

Property Data Type: GUID

See Also

IReposTypeLib Interface

IReposTypeLib::CreateClassDef

This method creates a new class definition object. No interfaces are attached to the class.

HRESULT CreateClassDef(
  VARIANT *sObjId*,
  BSTR *Name*,
  VARIANT *sClsId*,
  IClassDef ** *ppIClassDef*
);

Parameters

sObjId

[in]
The object identifier to be used for the new class definition object. The repository engine will assign an object identifier if you set this parameter to OBJID_NULL.

Name

[in]
The name of the new class.

sClsId

[in]
The global identifier by which this class is referenced.

**ppIClassDef* [out]

The interface pointer to the new class definition object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IClassDef Interface

IReposTypeLib Interface

IReposTypeLib::CreateInterfaceDef

This method creates a new interface definition object. Use the **IClassDef::AddInterface** method to attach the interface to a class definition object.

HRESULT CreateInterfaceDef(**VARIANT** *sObjId,*
BSTR *Name,*
VARIANT *sIId,*
IInterfaceDef **pIAncestor,*
IInterfaceDef ***ppIInterfaceDef*
);

Parameters

sObjId

[in]
The object identifier to be assigned to the new interface definition object. If this parameter is set to OBJID_NULL, the repository engine assigns an object identifier for you.

Name

[in]
The name of the interface that is to be created.

sIId

[in]
The interface identifier associated with the signature for this interface. If there is none, set this parameter to zero.

**pIAncestor*

[in]
The **IInterfaceDef** interface pointer for the base interface from which the new interface is derived.

**pplInterfaceDef*

[out]
The interface pointer for the new interface.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IClassDef::AddInterface

IInterfaceDef Interface

IReposTypeLib Interface

IReposTypeLib::CreateRelationshipDef

This method creates a relationship definition object for a new relationship type. After the relationship definition is created, use the **IInterfaceDef::CreateRelationshipColDef** method to create origin and destination collection definitions for the new relationship type.

HRESULT CreateRelationshipDef(
  VARIANT *sObjId*,
  BSTR *Name*,
  IReposTypeInfo ******** *ppIRelshipDef*
);

Parameters

sObjId [in]
The object identifier for the new relationship type. The repository engine assigns an object identifier if you set this parameter to OBJID_NULL.

Name

[in]
The name of the new relationship type.

**ppIRelshipDef*

[out]
The COM interface pointer to the new relationship definition object.

Return Value

S_OK

The method completed successfully.

Error Values

This method failed to complete successfully.

See Also

IInterfaceDef::CreateRelationshipColDef

IReposTypeLib Interface

RelationshipDef Class

IReposTypeLib ReposTypeInfos Collection

This collection contains all classes, interfaces, and relationship types that are associated with a repository type library. The repository engine uses this collection to enforce the unique naming of all classes, interfaces, and relationship types for a repository type library.

Dispatch Identifier: DISPID_ReposTypeInfos (42)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**ReposTypeLib_ScopeFor_ReposTypeInfo**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	Yes	The source object for the collection is also the origin object.
Minimum Collection Size	Zero	The minimum number of items that must be contained in the collection is zero.
Maximum	Many	The maximum

Collection Size		number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	Deleting an origin object or a relationship in the collection causes the deletion of a corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.
Case-Sensitive Names	No	This collection does not use

		case-sensitive names for destination objects.
Unique Names	Yes	The collection requires that the name of a destination object be unique within the collection of destination objects.

See Also

IReposTypeLib Interface

IReposTypeLib ReposTypeLibContexts Collection

This is the collection of one repository root object that is the context for an information model.

Dispatch Identifier: DISPID_ReposTLBContexts (41)

Remarks

The following characteristics are true for this collection.

Collection characteristic	Value	Description
Relationship Type	**TlbManager_ContextFor_ReposTypeLibs**	This is the type of relationship by which all items of the collection are connected to a common source object.
Source Is Origin	No	The source object for the collection is not the same as the origin object.
Minimum Collection Size	One	The minimum number of items that must be contained in the collection is one.

Maximum Collection Size	Many	The maximum number of items that can be contained in the collection is unlimited.
Sequenced Collection	No	As a destination collection, this does not have an explicitly defined sequence. Collections of origin objects are never sequenced.
Deletes Propagated	Yes	Deleting an origin object or a relationship in the collection causes the deletion of a corresponding destination object.
Destinations Named	Yes	The relationship type for the collection permits the naming of destination objects.

Case-Sensitive Names	No	The collection does not permit the use of case-sensitive names for destination objects.
Unique Names	Yes	The relationship type for the collection requires that the name of a destination object be unique within the collection of destination objects.

See Also

IReposTypeLib Interface

IReposTypeLib2 Interface

The **IReposTypeLib2** interface inherits from the **IReposTypeLib** interface. It allows model creators to define dependencies between information models that are stored in a repository.

The interface **IReposTypeLib2** adds two collections, **DependsOn** and **UsedBy**, which are connected through the **ReposTypeLibDependency** relationship. When an installation script for a repository type library is inserted into a repository database, the repository engine stores this information by using the **DependsOn** collection.

Note The engine does not automatically calculate the dependency information for models created using the repository API.

When to Use

Use the **IReposTypeLib2** interface to define dependencies between type libraries in information models.

Properties

IReposTypeLib Property	Description
TypeLibID	The global identifier for the repository type library

IReposTypeLib2 Property	Description
Prefix	Stores the prefix of an interface name to distinguish an interface from other identically named interfaces

Methods

IUnknown Method	Description
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments the reference count
Release	Decrements the reference count

IDispatch Method	Description
GetIDsOfNames	Maps a single member and a set of argument names to a corresponding set of dispatch identifiers
GetTypeInfo	Retrieves a type information object, which can be used to get the type information for an interface
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
Invoke	Provides access to properties and methods exposed by an Automation object

IRepositoryDispatch Method	Description
get_Properties	Retrieves the **IReposProperties** interface pointer. The **IReposProperties** interface provides access to the **Properties** collection.

IReposTypeLib Methods	Description
CreateClassDef	Creates a new class definition object
CreateInterfaceDef	Creates a new interface definition object
CreateRelationshipDef	Creates a new relationship definition object

Collections

IReposTypeLib Collection	Description
ReposTypeInfos	The collection of all classes, interfaces, and relationship types that are defined in the repository type library
ReposTypeLibContexts	The collection of one Repository root object that is the context for the repository type library

IReposTypeLib2 Collection	Description
DependsOn	The collection that relates type libraries that depend on other type libraries
UsedBy	The collection that relates type libraries used by other type libraries

See Also

IManageReposTypeLib Interface

IReposTypeLib Interface

ReposTypeLib Class

IReposTypeLib2 Prefix Property

This property stores the prefix of an interface name to distinguish an interface from other identically named interfaces. Attaching a prefix guarantees that a class that implements interfaces from different information models does not introduce a name conflict when both interfaces share the same name. The prefix is also used in XML for identifying namespaces (for example, "Uml" in UmlElement).

The maximum length for this prefix is 255 characters.

For the Open Information Model (OIM), prefix values are added during model installation. If no prefix is specified, the first three letters of the information model name are applied as a default value.

For the latest version of the Meta Data Coalition (MDC) OIM, prefix values must be added programmatically. Prefix values are not added during model installation.

Dispatch Identifier: DISPID_IReposTypeLib2Prefix

Property Data Type: string

See Also

IManageReposTypeLib Interface

IReposTypeLib Interface

ReposTypeLib Class

IReposTypeLib2 DependsOn Collection

This is the collection that relates repository type libraries that, in turn, depend on other repository type libraries.

Dispatch Identifier: DISPID_IReposTypeLib2DependsOn (330)

See Also

IManageReposTypeLib Interface

IReposTypeLib Interface

ReposTypeLib Class

IReposTypeLib2 UsedBy Collection

This is the collection that relates repository type libraries used by other repository type libraries.

Dispatch Identifier: DISPID_IReposTypeLib2UsedBy (331)

See Also

IManageReposTypeLib Interface

IReposTypeLib Interface

ReposTypeLib Class

Model Dependency Example

The following example demonstrates the use of both the **DependsOn** and **UsedBy** collections. These examples are written in Microsoft® Visual Basic®.

This example requires the Microsoft SQL Server™ 2000 Meta Data Services Software Development Kit (SDK).

To run this example:

1. Create an information model FileSys.mdl that contains three packages: FileSys, FAT, and NTFS.

2. Create three type libraries: FileSys, FAT, and NTFS.

3. Populate the repository database (FileSys.mdb) with the type libraries by using the SDK component Inrepim.exe.

In this example the type library FileSys is used by FAT and NTFS. Also, both of the type libraries FAT and NTFS depend on the FileSys type library.

The Visual Basic Module

```
'---------------------- Model Dependency Example ----------------------
'Declare OBJIDs assigned to type libraries:
Public Const OBJID_TypeLib_FILESYS = "{{992CF8AC-BD64-11d2
Public Const OBJID_TypeLib_NTFS = "{{992CF8B1-BD64-11d2-AC
Public Const OBJID_TypeLib_FAT = "{{992CF8AF-BD64-11d2-ACE
'---------------------------- Declarations ----------------------------
Public Rep As New Repository
Dim FileSys As RepositoryObject
Dim FAT As RepositoryObject
Dim NTFS As RepositoryObject
Dim Root As RepositoryObject
```

```
Private Sub Main()
'----------Open Repository database and set OBJIDs to objects -----
   Set Root = Rep.Open("FileSys.mdb")
   Set FileSys = Rep.Object(OBJID_TypeLib_FILESYS)
   Set FAT = Rep.Object(OBJID_TypeLib_FAT)
   Set NTFS = Rep.Object(OBJID_TypeLib_NTFS)
'---------------------------- Transaction ----------------------------
   Rep.Transaction.Begin
   FileSys("IReposTypeLib2").UsedBy.Add FAT
   NTFS("IReposTypeLib2").DependsOn.Add FileSys
   Rep.Transaction.Commit
'------------------------------ Cleanup ------------------------------
   Set FileSys = Nothing
   Set FAT = Nothing
   Set NTFS = Nothing
   Set Rep = Nothing
End Sub
'------------------- End of Model Dependency Example ----------------
```

See Also

IManageReposTypeLib Interface

IReposTypeLib Interface

Meta Data Services SDK

ReposTypeLib Class

IScriptDef Interface

IScriptDef is derived from **IUnknown**, **IDispatch**, and **IRepositoryDispatch**.

The **IScriptDef** interface allows the user to associate a Microsoft® ActiveX® script definition with a method without requiring the user to create an aggregation wrapper.

When to Use

Use the **IScriptDef** interface to:

- Define a method on an interface.
- Define a method on a base interface, overriding the base interface method.
- Define a validation rule for a property on an interface.
- Define a validation rule for a property on a base interface, overriding the base interface validation rule.

Properties

Properties	Description
Body	The storage for the body of the script. A variable length string, not to exceed 64 KB in length.
Language	The storage for the language the script is written in. Valid values are Microsoft JScript® and Microsoft Visual Basic® Scripting Edition (VBScript). This string is a maximum of 255 characters.

Methods

Method	Description
ValidateScript	Validates a script's syntax. It returns S_OK if the script can be executed; otherwise it returns a script engine specific error.

Collections

Collection	Description
UsingClasses	Collection of classes for which this script is used
UsingInterfaces	Collection of interfaces for which this script is used
UsingMembers	Collection of members for which this script is used

See Also

Defining Script Objects

MethodDef Object

IScriptDef::ValidateScript

The **ValidateScript** method syntactically validates the script.

HRESULT ValidateScript();

Dispatch Identifier: DISPID_IScriptDef_ValidateScript = 347

Parameters

None.

Return Value

S_OK

The method completed successfully.

Error Values

This method returns a Script Engine specific error if the script engine is unable to instantiate the script.

Remarks

The syntax of the script is checked by instantiating the script.

See Also

IScriptDef Interface

IScriptDef Body Property

Contains the body of a script.

Dispatch Identifier: DISPID_IScriptDef_Body = 346

Property Data Type: **string**

See Also

IScriptDef Interface

IScriptDef Language Property

Contains a string describing the language the script is written in. Valid values are:

- VBScript
- JScript

Dispatch Identifier: DISPID_IScriptDef_Language = 345

Property Data Type: **string**

See Also

IScriptDef Interface

IScriptDef UsingClasses Collection

The collection of classes being used by this script.

Dispatch Identifier: DISPID_IScriptDef_UsingClasses (349)

See Also

IScriptDef Interface

IScriptDef UsingInterfaces Collection

The collection of interfaces being used by this script.

Dispatch Identifier: DISPID_IScriptDef_UsingClasses (352)

See Also

IScriptDef Interface

IScriptDef UsingMembers Collection

The collection of members using this script.

Dispatch Identifier: DISPID_IScriptDef_UsingMembers (355)

See Also

IScriptDef Interface

IViewClassDef Interface

The **IViewClassDef** interface is derived from **IRepositoryDispatch**, which inherits from **IDispatch** and is implemented by the **ClassDef** class.

When to Use

Use this interface to define database views for a class.

Properties

Property	Type	Description
ViewName	**string**	Customized view name for better readability and to help prevent name-space collisions
ViewFlags	**long**	Bit flags that determine the characteristics of the view generated for the class

Dispatch Identifier: DISPID_IViewClassDef (375)

See Also

Defining a Class View

IRepositoryDispatch Interface

IViewInterfaceDef Interface

IViewPropertyDef Interface

IViewRelationshipDef Interface

IViewClassDef ViewName Property

This property contains a custom view name for the class-based view. This property overrides the default name. The maximum length for **ViewName** is 128 characters.

Dispatch Identifier: DISPID_IViewClassDefViewName (377)

Property Data Type: **string**

See Also

IViewInterfaceDef Interface

Naming Conventions for Generated Views

ViewFlags Property

IViewClassDef ViewFlags Property

This property contains flags that determine the characteristics of a generated view that is based on a **ClassDef** object.

Dispatch Identifier: DISPID_IViewClassDefViewFlags (376)

Property Data Type: **long**

The following table describes the bit flags allowed for the **ViewFlags** property.

Name	Bit	Default value	Description
GENERATE_RESOLVED_VIEW	1	0	Specifies whether to genera view that supports version resolution.
GENERATE_NORESOLUTION_VIEW	2	0	Specifies whether to genera view that does no version resolution. This flag should be used on non-versioned repositories.
GENERATE_WORKSPACE_VIEW	4	0	Specifies whether to genera view that is scoped for a workspace.
USE_VERSIONID_COLUMN	8	0	Specifies that the GENERATE_RESOLVED_ view for this class includes **VersionID** column to identi version to which this object resolves.
USE_VERSION_FLAGS_COLUMN	16	0	Specifies whether to includ **Z_VState_Z** of **RTblVersi** the GENERATE_RESOLVED_ view, indicating whether th version is frozen or checke

			a workspace.

See Also

Defining a Class View

IViewInterfaceDef Interface

ViewName Property

IViewInterfaceDef Interface

The **IViewInterfaceDef** interface is derived from **IRepositoryDispatch**, which inherits from **IDispatch** and is implemented by the **InterfaceDef** class.

When to Use

Use this interface to define a database view for all objects that implement a specific interface.

Properties

Property	Type	Description
ViewName	**string**	Customized view name for better readability and to help prevent name-space collisions.
ViewFlags	**long**	Bit flags that determine the characteristics of the view generated for the class

Dispatch Identifier: DISPID_IViewInterfaceDef (378)

See Also

Defining an Interface View

InterfaceDef Object

IRepositoryDispatch Interface

IViewClassDef Interface

IViewPropertyDef Interface

IViewRelationshipDef Interface

IViewInterfaceDef ViewName Property

This is a custom view name that overrides the default view. The **ViewName** can be no longer than 128 characters.

Dispatch Identifier: DISPID_IViewInterfaceDefViewName (380)

Property Data Type: **string**

See Also

IViewInterfaceDef Interface

Naming Conventions for Generated Views

IViewInterfaceDef ViewFlags Property

This property contains flags that determine the characteristics of a generated view that is based on an **InterfaceDef** object.

Dispatch Identifier: DISPID_IViewInterfaceDefFlags (379)

Property Data Type: **long**

The following table describes the bit flags for the **ViewFlag** property.

Flag Name	Position	Default value	Description
GENERATE_RESOLVED_VIEW	1	0	Specifies whether to g view that supports ver resolution.
GENERATE_NORESOLUTION_VIEW	2	0	Specifies whether to g view that does no vers resolution. This flag sl be used on nonversion repositories.
GENERATE_WORKSPACE_VIEW	4	0	Specifies whether to g view that is scoped fo workspace.
USE_VERSIONID_COLUMN	8	0	Specifies that the GENERATE_RESOL view for this interface **VersionID** column to version to which this c resolves.
USE_VERSION_FLAGS_COLUMN	16	0	Specifies whether to i **Z_VState_Z** of **RTbl** the GENERATE_RESOL view. **Z-VState_Z** inc whether the version is

			checked out to a work
EXCLUDE_IMPLIED_INTERFACES	32	0	Specifies whether to i properties of interface directly or indirectly i this interface but are n supertypes of this inte

See Also

Defining an Interface View

IViewInterfaceDef Interface

ViewName Property

IViewPropertyDef Interface

The **IViewPropertyDef** interface defines a custom name that you can use to override the default column name of a view. This interface is derived from **IRepositoryDispatch**, which inherits from **IDispatch** and is implemented by the **PropertyDef** class.

The same view column can appear in multiple views. In each case, the view column name that you define is the same for all occurrences. For example, a view that supports implied interfaces or that is based on an inherited interface includes members from multiple interfaces, creating a case where a single column can appear more than once.

When to Use

Use this interface to define the column name of a property in the view. This prevents name-space collisions and allows for column renaming for better readability in an SQL query statement.

Properties

Property	Type	Description
ViewColumnName	**string**	Customized view name for better readability and to help prevent name-space collisions. A view column name can be a maximum of 128 characters in length. The default value is null.

Dispatch Identifier: DISPID_IViewInterfaceDef (385)

See Also

IViewClassDef Interface

IViewInterfaceDef Interface

IViewRelationshipDef Interface

PropertyDef Object

IViewPropertyDef ViewColumnName Property

Use this property to create a customized column name for a property. The maximum length for this property is 128 characters.

Dispatch Identifier: DISPID_IViewInterfaceDefViewColumnName (386)

Property Data Type: **string**

See Also

IViewPropertyDef Interface

IViewRelationshipDef Interface

The **IViewRelationshipDef** interface is derived from **IRepositoryDispatch**, which inherits from **IDispatch** and is implemented by the **RelationshipDef** class.

When to Use

Use this interface to define a junction table view of a relationship class. This is used for views that have many-to-many relationships.

Properties

Property	Type	Description
ViewFlags	**long**	Bit flags that determine the characteristics of the view generated for the class
ColumnNamePrefix	**string**	This string is prefixed to the column names **NAME, PrevDstID,** and **RelTypeID**. The string is used in all views where the corresponding column appears.
JunctionViewName	**string**	A custom view name that overrides the default view. It applies specifically to a many-to-many relationship or a relationship that has the GENERATE_VIEW flag set.

Dispatch Identifier: DISPID_IViewRelationship (381)

See Also

Defining a Junction Table View

IRepositoryDispatch Interface

IViewInterfaceDef Interface

IViewPropertyDef Interface

IViewClassDef Interface

IViewRelationshipDef ViewFlags Property

This property contains flags that determine the characteristics of a generated view that is based on a **RelationshipDef** object.

Property Data Type: **long**

This table describes the flags property for the **IViewRelationshipDef** interface.

Flag name	Position	Default value	Description
GENERATE_RESOLVED_VIEW	1	0	Specifies whether to g table view that suppor resolution.
GENERATE_NORESOLUTION_VIEW	2	0	Specifies whether to g table view that does n resolution. This flag s on nonversioned repos
GENERATE_WORKSPACE_VIEW	4	0	Specifies whether to g table view that is scop workspace.
INCLUDE_PREVDSTID	64	0	Specifies that the inter the previous element i collection be included view containing the re This flag applies only relationship types.
INCLUDE_RELTYPEID	128	0	Specifies that the view relationship should ha containing the relation joins with **RTblRelsh**
CHOOSE_ORIGIN	256	0	If a relationship is one storage in views conta relationship type's orig

			GENERATE_RESOL GENERATE_NORES and GENERATE_WC have precedence over
INCLUDE_LONGNAMES	512	0	Specifies that long na included in junction ta

Dispatch Identifier: DISPID_IViewRelationshipDefFlags (382)

See Also

Defining Views in an Information Model

IViewRelationshipDef ColumnNamePrefix Property

IViewRelationshipDef Interface

IViewRelationshipDef JunctionViewName Property

RTblRelshipProps SQL Table

IViewRelationshipDef ColumnNamePrefix Property

This string is prefixed to the column names **NAME, PrevDstID,** and **RelTypeID**. The string is used in all views in which the corresponding column appears. The maximum length of this string is 118 characters.

This string is used as the foreign key column (if the relationship is stored as a foreign key), and is attached to all columns that are scoped to the relationship.

Note In a view that involves multiple relationship types, use this property to enhance the readability of column names.

Dispatch Identifier: DISPID_IViewRelationshipDefColumnNamePrefix (383)

Property Data Type: **string**

See Also

IViewRelationshipDef Interface

JunctionViewName Property

Naming Conventions for Generated Views

ViewFlags Property

IViewRelationshipDef JunctionViewName Property

This property is a custom view name that overrides the default view name. It applies only to many-to-many relationships or those that have the GENERATE_RESOLVED_VIEW, GENERATE_NORESOLUTION_VIEW, or GENERATE_WORKSPACE_VIEW flag set. The maximum length of this string is 128 characters.

Dispatch Identifier: DISPID_IViewRelationshipDefJunctionViewName (384)

Property Data Type: **string**

See Also

ColumnNamePrefix Property

Defining a Junction Table View

IViewRelationshipDef Interface

Naming Conventions for Generated Views

ViewFlags Property

Meta Data Services Programming

Constants and Data Types

This section contains information about the constants you can use when programming against the repository API. It also contains reference topics about data types, which provide information that supports conversion, migration, or cross-tool integration. Header files provide additional definitions.

Topic	Description
Repository Constants	Defines the constants used for repository engine classes, interfaces, and objects
SQL and API Types Used in Property Definitions	Maps the API types and SQL types recognized by the repository engine
Repository SQL Data Types	Maps repository data types to SQL data types supported by the underlying database server

Header Files

Various declarations and definitions for the repository API can be found in the following files. The repository API is organized by repository engine classes, interfaces, and objects, and by type information model classes, interfaces, and objects.

- The Repapi.h source file contains Microsoft® Visual C++® definitions specific to the repository engine classes, interfaces, and objects.

- The Reptim.h, Reptim2.h, and Reptim3.h source files contain the constant definitions specific to the type information model classes, interfaces, and objects. Most of the various identifiers (class, interface, object, local, internal, and dispatch) that you may find useful are defined in this file.

- The Repauto.h file contains the definitions of the external enumerations, classes, and interfaces of the repository engine and of the type information model. All of the interfaces found in this file support

Automation-level access.

See Also

Programming Information Models

Repository API

Repository API Reference

Meta Data Services Programming

Repository Constants

These constants are defined for repository engine classes, interfaces, and objects.

Constant	Value	Description
CARD_NOLIMIT	0xFFFF	Specifies that a collection can have an unlimited number of items.
COLUMNNAMESIZE	32 or 255	The maximum length, in bytes, of an SQL column name. Microsoft® SQL Server™ version 6.5 allows 32 bytes. SQL Server 7.0 and SQL Server 2000 allow 255 bytes.
INTID_NULL	0xFFFFFFFF	The null internal identifier.
MEMBERNAMESIZE	64 or 128	The maximum length, in bytes, of a property, method, or collection type name. SQL Server 6.5 allows 64 bytes. SQL Server 7.0 and SQL Server 2000 allow 128 bytes.
OBJID_NULL	See Reptim.h	The null object identifier. Use this value when you want the repository engine to assign an object identifier for you.
PASSWORDSIZE	64	The maximum length, in bytes, of the password string that is used to connect to the repository database.

PROPVALSIZE	220	The maximum length, in bytes, of an annotational property string.
RELSHIPNAMESIZE	249 or 260	The maximum length, in bytes, of a name that a relationship assigns to its destination object. SQL Server 6.5 allows 249 bytes. SQL Server 7.0 and SQL Server 2000 allow 260 bytes.
REPOSERROR_OBJKNOWN	0x00000001L	Returned in the **fFlags** field of the **REPOSERROR** structure. It indicates that the object identifier is known.
REPOSERROR_SQLINFO	0x00000002L	Returned in the **fFlags** field of the **REPOSERROR** structure. It indicates that the SQL error information is valid.
REPOSERROR_HELPAVAIL	0x00000004L	Returned in the **fFlags** field of the **REPOSERROR** structure. It indicates that the **rcHelpFile** and **dwHelpContext** fields are valid.
REPOSERROR_MSG_SIZE	256	The maximum length, in bytes, of the message in the **rcMsg** field of the **REPOSERROR** structure.
TABLENAMESIZE	32 or 255	The maximum length, in bytes, of an SQL table name. SQL Server 6.5 allows 32 bytes. SQL Server 7.0 and SQL Server 2000 allow 255

		bytes.
TIMESTAMP_NULL	{9999, 12, 31, 0, 0, 0, 0}	The null timestamp value.
TYPEINFONAMESIZE	64 or 128	The maximum length, in bytes, of a class, interface, or relationship type name. SQL Server 6.5 allows 64 bytes. SQL Server 7.0 and SQL Server 2000 allow 128 bytes.
TYPELIBNAMESIZE	64 or 128	The maximum length, in bytes, of a repository type library name. SQL Server 6.5 allows 64 bytes. SQL Server 7.0 and SQL Server 2000 allow 128 bytes.
USERSIZE	64 or 128	The maximum length, in bytes, of the user name that is used to connect to the repository database. SQL Server 6.5 allows 64 bytes. SQL Server 7.0 and SQL Server 2000 allow 128 bytes.
VIEWNAMESIZE	128	The maximum length, in bytes, of a user-defined view name.
COLPREFIXSIZE	119	The maximum length, in bytes, of a prefix that identifies a relationship in a generated view.

See Also

CollectionDefFlags Enumeration

ConnectionFlags Enumeration

Generating Views

InterfaceDefFlags Enumeration

InterfaceMemberFlags Enumeration

REPOSERROR Data Structure

TransactionFlags Enumeration

Meta Data Services Programming

SQL and API Types Used in Property Definitions

The following tables show the API types recognized by the repository engine, as well as the SQL types. These values appear in the **APIType** and **SQLType** properties of a **PropertyDef** object. For more information about conversion between SQL and API types, see the Microsoft® ODBC documentation. For more information about API and SQL data type descriptions, see Data Types.

The following table identifies API types that map to Transact-SQL. It is recommended that you not use unlisted API types.

API Types

API type	VALUE	Maps to (T-SQL)
SQL_C_BINARY*	-2	**Binary** or **varbinary**
SQL_C_TINYINT	-6	**tinyint**
SQL_C_BIT	-7	**bit**
SQL_C_CHAR	1	**char** or **varchar**
SQL_C_LONG	4	**int**
SQL_C_SHORT	5	**int**
SQL_C_FLOAT	7	**real**
SQL_C_DOUBLE	8	**float**
SQL_C_DATE	9	**datetime**
SQL_C_TIME	10	**datetime**
SQL_C_TIMESTAMP	11	**datetime**

Note For SQL_C_BINARY use an array of unsigned characters. C++ programmers must use VT-UI1.

SQL Types

SQL type	VALUE	Maps to
SQL_LONGVARCHAR	-1	**text**
SQL_BINARY	-2	**binary**

SQL_VARBINARY	-3	**varbinary**
SQL_LONGVARBINARY	-4	**image**
SQL_TINYINT	-6	**tinyint**
SQL_BIT	-7	**bit**
SQL_CHAR	1	**char**
SQL_NUMERIC	2	**numeric**
SQL_DECIMAL	3	**decimal**
SQL_INTEGER	4	**integer**
SQL_SMALLINT	5	**smallint**
SQL_FLOAT	6	**float**
SQL_REAL	7	**real**
SQL_DOUBLE	8	**real**
SQL_DATE	9	**datetime**
SQL_TIME	10	**datetime**
SQL_TIMESTAMP	11	**datetime**
SQL_VARCHAR	12	**varchar**

The following table identifies API types that are not supported by repository Automation. You can only store and retrieve unsigned integers. It is recommended that you not use these API types.

API Types - Not Supported

API type	VALUE
SQL_C_UTINYINT	-28
SQL_C_STINYINT	-26
SQL_C_ULONG	-18
SQL_C_USHORT	-17
SQL_C_SLONG	-16
SQL_C_SSHORT	-15

See Also

Constants and Data Types

PropertyDef object

Repository SQL Data Types

Meta Data Services Programming

Repository SQL Data Types

Because data types can vary between database management systems, the repository engine maps its own set of repository data types to the SQL data types that are supported by the underlying database server.

This table translates repository data types into SQL data types known by the database server. For data types that vary between different database servers, the data type used for each database server is shown. In these cases, the Microsoft® SQL Server™ data type is shown with (S) appended to it, and the Microsoft Jet server data type is shown with (J) appended to it.

Repository SQL data types appear in repository SQL tables that compose the repository SQL schema.

Repository data type	Database data type	Description
RTBoolean	**bit** (S) **boolean** (J)	A true/false value
RTBrID	4-byte **integer**	A branch identifier
RTCount	2-byte **integer**	The count (that is, cardinality) of a collection
RTDBVersion	40-byte **varchar**	A string that indicates the engine version that created the database
RTDispID	4-byte **integer**	An Automation dispatch identifier
RTFlags	2-byte **integer**	Flag bits that define the behavior of an entity or indicate what kind of row exists in a table
RTGUID	16-byte **binary**	A globally unique identifier
RTIntID	8-byte **binary**	An internal identifier
RTLClock	4-byte **integer**	Logical clock value
RTLocalID	4-byte **binary**	A local identifier; part of an internal identifier
RTLongBinary	**image** (S) **longbinary** (J)	A long binary stream of data

RTLongString	**text** (S) **memo** (J)	A string with a maximum length of approximately 1 gigabyte (GB)
RTNameString	200-byte **varchar**	A special truncated name string used for indexing
RTScale	2-byte **integer**	Scale for numeric data; the number of digits after the decimal point
RTShortString	220-byte **varchar**	A special shortened string value used for indexing
RTSiteID	4-byte **binary**	A site identifier; part of an object identifier
RTSize	2-byte **integer**	The size of a data type, in bytes
RTSQLName	30-byte **varchar**	An SQL identifier; a table or column name
RTSQLType	2-byte **integer**	The ODBC representation of an SQL data type
RTVerID	4-byte **integer**	A version-within-branch identifier

See Also

Constants and Data Types

Repository SQL Schema

Repository SQL Tables

Meta Data Services Programming

Enumerations

Enumerations are a fixed set of values that share the same context. Through the repository API, you can set predefined enumerations on flags to control repository engine behavior for object definitions and some aspects of load operations.

Enumeration values for flags are not the same as **EnumerationDef** objects that you create using the repository API. For more information about enumeration objects, see EnumerationDef Object.

You can set enumeration values for the following flags.

Flags	Description
CollectionDefFlags Enumeration	Defines the behavior of relationship collections
ConnectionFlags Enumeration	Defines the characteristics of a repository database connection
InterfaceDefFlags Enumeration	Defines specific characteristics of an interface definition
InterfaceMemberFlags Enumeration	Defines specific characteristics of an interface member
TransactionFlags Enumeration	Defines which transaction option is to be retrieved or set
LoadStatus Enumeration	Reports on the loading status of an object
RepODBCFlags Enumeration	Enables asynchronous operations for loading object collections

Meta Data Services Programming

CollectionDefFlags Enumeration

This enumeration defines the behavior of a **Relationship** collection. These flags are bit flags, and they can be combined to set multiple options. The absence of a flag indicates that the option is not set.

```
enum {
COLLECTION_NAMING = 1,
COLLECTION_UNIQUENAMING = 2,
COLLECTION_CASESENSITIVE = 4,
COLLECTION_SEQUENCED = 8,
COLLECTION_PROPAGATEDELETE = 16,
COLLECTION_NEWORGVERSIONSPARTICIPATE = 32,
COLLECTION_NEWORGVERSIONSDONOTPARTICIPATE = 64,
COLLECTION_MERGEWHOLE = 128,
COLLECTION_CONTAINING = 256,
COLLECTION_OBJECTNAMING = 512,
COLLECTION_NEWDESTVERSIONADD = 1024,
COLLECTION_NEWDESTVERSIONPROPAGATE = 2048
} CollectionDefFlags;
```

Value	Description
COLLECTION_UNIQUENAMING	If this flag is set, collection requir be unique within This flag applies that permits the r
COLLECTION_CASESENSITIVE	If this flag is set, collection permit destination objec relationship type destination objec
COLLECTION_SEQUENCED	If this flag is set, collection have a

	Collections of or
COLLECTION_NAMING	If this flag is set, collection permit
COLLECTION_PROPAGATEDELETE	If this flag is set, collection requir destination objec deleted if it is the connected to the
COLLECTION_NEWORGVERSIONSPARTICIPATE	If this flag is set, origin collection creation version version.
COLLECTION_NEWORGVERSIONSDONOTPARTICIPATE	If this flag is set, a new object ver COLLECTION_ flag. You cannot However, by def neither flag is se the COLLECTION_ flag were set. Th version of a repo copies new origi creation version
COLLECTION_MERGEWHOLE	Setting this flag **MergeVersion** n causes the succes version. If the su match, the succe secondary versio differences from versions are mer more informatio
COLLECTION_CONTAINING	If this flag is set, column) will be example, a table

	cannot exist outs this flag to deter information, see
COLLECTION_OBJECTNAMING	If this flag is set, name specified b as its relationshi INamedObject In
COLLECTION_NEWDESTVERSIONADD	If this flag is set, destination objec object is created, related origin ob created. The new relationship betw destination objec
COLLECTION_NEWDESTVERSIONPROPAGATE	If this flag is set, destination objec objects related th current collectio This behavior oc destination objec versioned, its ori This behavior ca version graph. If simultaneously a relationship, its c versioning of pai graph until an un This behavior oc origin object is e for a series of rel same transaction

See Also

CollectionDef Flags Property

ICollectionDef Flags Property

IInterfaceDef::CreateRelationshipColDef

InterfaceDef CreateRelationshipColDef Method

Meta Data Services Programming

ConnectionFlags Enumeration

This enumeration defines the characteristics of a connection to a repository database. These flags are bit flags, and may be combined to set multiple options. The absence of a flag indicates that the option is not set.

enum {
REPOS_CONN_NEWCACHE = 2
REPOS_CONN_UPGRADE = 4
REPOS_CONN_RECOMPUTE = 8
} ConnectionFlags;

Value	Description
REPOS_CONN_NEWCACHE	Creates a new cache when you open or create a repository instance. This consumes additional resources.
REPOS_CONN_UPGRADE	Upgrades the repository database tables to the most recent version. Standard repository SQL tables are replaced. Custom repository tables that you create by way of schema extensions are unchanged.
REPOS_CONN_RECOMPUTE	Recomputes all class definitions, and regenerates views and stored procedures.

See Also

IRepository::Create Method

IRepository::Open Method

Repository Create Method

Repository Open Method

Meta Data Services Programming

InterfaceDefFlags Enumeration

This enumeration defines specific characteristics of an interface definition. These flags are bit flags, and you can combine them to set multiple options. The absence of a flag indicates that the option is not set.

```
enum {
INTERFACE_EXTENSIBLE  = 1
INTERFACE_HIDDEN      = 2
} InterfaceDefFlags;
```

Value	Description
INTERFACE_EXTENSIBLE	Specifies that the interface supports extensions
INTERFACE_HIDDEN	Specifies that the interface is not visible to Automation queries

See Also

IInterfaceDef Flags Property

InterfaceDef Flags Property

Meta Data Services Programming

InterfaceMemberFlags Enumeration

This enumeration defines specific characteristics of an interface member. The absence of the flag indicates that the option is not set.

enum {
INTERFACEMEMBER_HIDDEN = 1
INTERFACEMEMBER_READONLY = 2
INTERFACEMEMBER_VIRTUAL = 4
INTERFACEMEMBER_DERIVED = 0x8000
} InterfaceMemberFlags;

Value	Description
INTERFACEMEMBER_HIDDEN	Specifies that the interface member is not visible to Automation queries.
INTERFACEMEMBER_READONLY	Specifies that the interface member cannot be updated by an application.
INTERFACEMEMBER_VIRTUAL	Supports members that are not stored. If this flag is set and the member is a property, the repository engine will not allocate a column for it in the table for the interface. The repository engine will return an error if an attempt is made to access this member. A COM aggregation must be used to implement the member. For more information, see Virtual Members.
INTERFACEMEMBER_DERIVED	Specifies that the interface member is derived from a base member. By default, a member is a base member.

See Also

CollectionDef Flags Property

IInterfaceMember Flags Property

MethodDef Flags Property

PropertyDef Flags Property

Meta Data Services Programming

TransactionFlags Enumeration

This enumeration specifies which transaction option is to be retrieved or set.

```
enum {
TXN_RESET_OPTIONS          = 1
TXN_NORMAL                 = 2
TXN_EXCLUSIVE_WRITEBACK    = 3
TXN_EXCLUSIVE_WRITETHROUGH  = 4
TXN_TIMEOUT_DURATION       = 5
TXN_START_TIMEOUT          = 6
TXN_QUERY_TIMEOUT          = 7
TXN_DBMS_READONLY          = 8
TXN_USE_DTC                = 10
} TransactionFlags;
```

Value	Description
TXN_RESET_OPTIONS	Specifies that all options must be reset their default values. Any associated option value is ignored. It is valid only for setting transaction options.
TXN_NORMAL	Specifies the nonexclusive writeback mode transaction option. Nonexclusive writeback mode allows transactions for other repository instances to execute concurrently. Updates are cached for each session un a transaction is committed. TXN_NORMAL, TXN_EXCLUSIVE_WRITEBACK, a TXN_EXCLUSIVE_WRITETHROUG are write modes. Write modes are mutually exclusive. Only one write mo

	can be specified for each transaction.
TXN_EXCLUSIVE_WRITEBACK	This flag was created for use with version 1 of the repository engine. This flag is no longer valid.
TXN_EXCLUSIVE_WRITETHROUGH	This flag was created for use with version 1 of the repository engine. This flag is no longer valid.
TXN_TIMEOUT_DURATION	Specifies the transaction option that determines the maximum time to wait f a lock. The default value for this option is 20000 milliseconds.
TXN_START_TIMEOUT	Specifies the transaction option that determines the maximum time to wait before starting a transaction. TXN_START_TIMEOUT is the timeou duration if there are any conflicts in starting a transaction (for example, whe two transactions want to use a shared cache in exclusive mode). Setting TXN_START_TIMEOUT to ze means that there is no timeout. As a result, the repository engine will continuously try to start the transaction until it succeeds. The default value for this option is 0 milliseconds.
TXN_QUERY_TIMEOUT	Specifies the transaction option that determines the maximum number of seconds to wait while a database query executing. If multiple applications are performing transactions on the same cache, you ma want to increase this value. Doing so

	gives a transaction from one applicatio more time to complete before a second transaction (from another application) begins. The default value for this option is 10 seconds.
TXN_DBMS_READONLY	Specifies whether you can make chang to the repository database. If the value zero, you can make changes. If the valu is nonzero, the database is read-only. Y can read the value of this option, but yo cannot set it.
TXN_USE_DTC	Specifies whether to use Microsoft® Distributed Transaction Coordinator (M DTC) transactions.

See Also

IRepositoryTransaction::Get Option

IRepositoryTransaction::Set Option

RepositoryTransaction Get Option Method

RepositoryTransaction Set Option Method

Meta Data Services Programming

LoadStatus Enumeration

This enumeration contains the flags for the asynchronous loading status of a collection.

```
enum {
READY = 1,
INPROGRESS = 2,
CANCELLED = 3,
FAILED = 4
} LoadStatus;
```

Value	Description
READY	Loading is complete.
INPROGRESS	Loading in progress.
CANCELLED	Loading has been canceled (by caller).
FAILED	Loading failed (reason unknown).

See Also

RepODBCFlags Enumeration

Meta Data Services Programming

RepODBCFlags Enumeration

This enumeration sets or clears the ASNYCH option of **ExecuteQuery**.

```
enum {
RODBC_RESET_OPTIONS = 1,
RODBC_ASYNCH = 2
} RepODBCFlags;
```

Value	Description
RODBC_RESET_OPTIONS	Reset all options on the ODBC connection.
RODBC_ASYNCH	Execute queries asynchronously.

See Also

IRepositoryODBC::ExecuteQuery

LoadStatus Enumeration

Meta Data Services Programming

Repository Errors

Repository errors are errors returned by the methods of repository interfaces. Repository error objects are described as **REPOSERROR** data structures.

All methods of repository interfaces return an **HRESULT** value that indicates whether the method successfully performed its function. The facility field of these **HRESULT** values is always set to FACILITY_ITF; this indicates that the meaning for any given error code value is specific to the interface from which the error is being reported. All of the standard repository interfaces (that is, interfaces that are automatically supplied with the repository API) use the same set of error codes. These codes are listed in numerical order and in alphabetical order. For more information, see Repository Errors (Numerical Order) and Repository Errors (Alphabetical Order).

See Also

Error Handling Overview

Handling Errors

REPOSERROR Data Structure

Meta Data Services Programming

REPOSERROR Data Structure

Repository engine methods return an **HRESULT** value that indicates whether or not the method completed successfully. If a repository engine method fails to complete successfully, an error object is created that contains details about the failure.

The **REPOSERROR** data structure contains the following details:

```
struct REPOSERROR {
ULONG    iSize;
ULONG    fFlags;
HRESULT  hr;
TCHAR    rcMsg[REPOSERROR_MSG_SIZE];
TCHAR    rcHelpFile[_MAX_PATH];
ULONG    dwHelpContext;
long     iNativeError;
TCHAR    rcSqlState[6];
short    iReserved;
OBJID    sObjID;
GUID     clsid;
GUID     iid;
};
```

iSize

The size in bytes of this data structure.

fFlags

Bit flags that define the validity of certain members of this data structure. Valid values are **REPOSERROR_OBJKNOWN**, **REPOSERROR_SQLINFO**, and **REPOSERROR_HELPAVAIL**. For more information about the meaning of these constants, see Repository Constants.

hr

The **HRESULT** return value that was returned from the method that logged this error.

rcMsg

The text message that is associated with this error. The message can be a maximum of 256 characters.

rcHelpFile

The name of the Help file that contains more information about this error.

dwHelpContext

The Help context identifier that is associated with this error.

iNativeError

The error code that was returned from the database engine. The value of this member is only valid if the *fFlags* member indicates that SQL information is present.

rcSqlState

SQL state information supplied by the database engine. The value of this member is only valid if the *fFlags* member indicates that SQL information is present.

iReserved

This parameter is reserved for use by the repository engine.

sObjID

The object identifier of the object that is associated with this error. The value of this member is only valid if the *fFlags* member indicates that the object is known.

clsid

The class identifier of the object that is associated with this error. The value of this member is only valid if the *fFlags* member indicates that the object is known.

iid

The interface identifier of the interface that is associated with this error. If the interface is not known, or not applicable, the value of this member is set to GUID_NULL.

See Also

IEnumRepositoryErrors::Next

Meta Data Services Programming

Repository Errors (Numerical Order)

The error codes that can be returned as a part of the **HRESULT** return value by repository engine methods are listed here in numerical order. These codes are also listed in alphabetical order. For more information, see Repository Errors (Alphabetical Order).

All error codes are of the form 0x8004*nnnn*. The prefix 8004 is omitted in the following errors to make the code more readable.

(0x1000) EREP_BADPARAMS

(0x1001) EREP_BADNAME

(0x1002) EREP_BADDRIVER

(0x1003) EREP_BADERROR

(0x1004) EREP_BUFFER_OVERFLOW

(0x1005) EREP_NAMETOOLONG

(0x1011) EREP_NOROWSFOUND

(0x1012) EREP_ODBC_CERROR

(0x1013) EREP_ODBC_MDBNOTFOUND

(0x1014) EREP_NEED_DATA

(0x1015) EREP_ODBC_UNKNOWNDRIVER

(0x1016) EREP_ODBC_CREATEFAILED

(0x1017) EREP_ODBC_WARNINGS

(0x1018) EREP_STILL_EXECUTING

(0x1019) EREP_ODBC_NOTCAPABLE

(0x1030) EREP_DB_EXISTS

(0x1031) EREP_DB_NOTCONNECTED

(0x1032) EREP_DB_ALREADYCONNECTED

(0x1033) EREP_DB_DBMSONETHREAD

(0x1034) EREP_DB_CORRUPT

(0x1035) EREP_DB_NOSCHEMA

(0x1036) EREP_DB_DBMSOLD

(0x1037) EREP_DB_READONLY

(0x1038) EREP_DB_INCOMPATIBLEVERSION

(0x1039) EREP_DB_UPGRADE

(0x1041) EREP_TXN_NOTXNACTIVE

(0x1042) EREP_TXN_AUTOABORT

(0x1043) EREP_TXN_TOOMANY

(0x1044) EREP_TXN_TIMEOUT

(0x1045) EREP_TXN_NODATA

(0x1046) EREP_TXN_NOSETINTXN

(0x1047) EREP_TXN_OBJABORTED

(0x1048) EREP_TXN_COLABORTED

(0x1070) EREP_REPOS_CACHEFULL

(0x1071) EREP_REPOS_NONEXTDISPID

(0x1072) EREP_DUPEDISPID

(0x1100) EREP_RELSHIP_EXISTS

(0x1101) EREP_RELSHIP_INVALID_PAIR

(0x1102) EREP_RELSHIP_NOTFOUND

(0x1105) EREP_RELSHIP_ORGONLY

(0x1106) EREP_RELSHIP_OUTOFDATE

(0x1107) EREP_RELSHIP_INVALIDFLAGS

(0x1108) EREP_RELSHIP_NAMEINVALID

(0x1109) EREP_RELSHIP_DUPENAME

(0x1110) EREP_RELSHIP_NONNAMINGCOL

(0x1120) EREP_TYPE_TABLEMISMATCH

(0x1121) EREP_TYPE_COLMISMATCH

(0x1122) EREP_TYPE_NOTNULLABLE

(0x1123) EREP_TYPE_MULTIDEFIFACES

(0x1124) EREP_TYPE_INVERTEDNOTALLOWED

(0x1125) EREP_TYPE_INVALIDSCALE

(0x1126) EREP_TYPE_BADTABLENAME

(0x1127) EREP_TYPE_MULTIPLEANCESTORS

(0x1200) EREP_LOCK_TIMEOUT

(0x1250) EREP_QRY_BADCOLUMNS

(0x1300) EREP_OBJ_NOTINITIALIZED

(0x1301) EREP_OBJ_NOTFOUND

(0x1302) EREP_OBJ_NONAMINGRELSHIP

(0x1303) EREP_OBJ_EXISTS

(0x1304) EREP_VERSION_NOTFOUND

(0x1400) EREP_PROP_MISMATCH

(0x1401) EREP_PROP_SETINVALID

(0x1402) SREP_PROP_TRUNCATION

(0x1403) EREP_PROP_CANTSETREPTIM

(0x1404) EREP_PROP_READONLY

(0x1405) EREP_PROP_NOTEXISTS

(0x1500) EREP_TIM_INVALIDFLAGS

(0x1501) EREP_TIM_FLAGSDEST

(0x1502) EREP_TIM_RELTYPEINVALID

(0x1503) EREP_TIM_CTYPEINVALID

(0x1504) EREP_TIM_TOOMANYCOLS

(0x1505) EREP_TIM_SQLTYPEINVALID

(0x1506) EREP_TIM_SQLSIZEINVALID

(0x1600) EREP_VM_CANTSETFROZEN

(0x1601) EREP_VM_MERGETOFROZEN

(0x1602) EREP_VM_MERGEFROMUNFROZEN

(0x1603) EREP_VM_UNFROZENVERSION

(0x1604) EREP_VM_FROZENVERSION

(0x1605) EREP_VM_CHECKEDOUTVERSION

(0x1606) EREP_VM_DUPBRANCHID

(0x1607) EREP_VM_SUCCESSOREXISTS

(0x1800) EREP_WKS_ITEMEXISTS

(0x1801) EREP_WKS_ITEMNOTEXISTS

(0x1802) EREP_NOTWORKSPACEITEM

(0x1803) EREP_ITEMNOTCHECKEDOUT

(0x1A01) EREP_BLOB_SEEKPASTEND

(0x1A02) EREP_BLOB_TEMPFILE

(0x1A03) EREP_BLOB_USERFILE

(0x1A04) EREP_BLOB_CANNOTSETPOS

(0x1B05) EREP_MEMDEL_DELCOLINVALID

(0x1C00) EREP_COL_OBJECTNAMING

(0x1C01) EREP_COL_OBJECTNOTNAMED

(0x1D00) EREP_UNKNOWNPROPERTY

(0x1D01) EREP_MISSINGLEFTBRACKET

(0x1D02) EREP_MISSINGRIGHTBRACKET

(0x1D03) EREP_MISSINGLEFTPARENTHESIS

(0x1D04) EREP_MISSINGRIGHTPARENTHESIS

(0x1D05) EREP_MISSINGCOMMA

(0x1D06) EREP_PROPERTYNOTFOUND

(0x1D07) EREP_INVALIDFILTER

(0x1D08) EREP_SCRIPT_NESTEDCALL

(0x1D09) EREP_SCRIPT_NOTFOUND

(0x1D0A) EREP_SCRIPT_INVALIDLANGUAGE

(0x1D0B) EREP_VIRTUAL_ALIAS

(0x1D0C) EREP_VIRTUAL_CALL

(0x1E00) EREP_CLASS_TOOCOMPLEX

(0x1E02) EREP_RTIM_CLASS_IS_NOT_CREATEABLE

(0x2000) EREP_VM_DIFFERENTTYPES

(1x1700) EREP_REL_ORGFROZEN

(1x1701) EREP_REL_ORGCLONE

(1x1702) EREP_REL_NONSEQONLY

(1x1703) EREP_REL_ORGPIN

(1x1704) EREP_REL_NOTPINNED

(1x1900) EREP_VCOL_VERSIONNOTMEMBER

(1x1901) EREP_VCOL_INVALIDOP

(1x1950) EREP_COL_NOTSEQUENCED

(1x1B00) EREP_MEMDEL_COLNOTDEFINED

(1x1B01) EREP_MEMDEL_BASEIFACENOTIMPL

(1x1B02) EREP_MEMDEL_BASECOLVIRTUAL

(1x1B03) EREP_MEMDEL_MULTIPLEBASES

(1x1B04) EREP_MEMDEL_CIRCULARCOLS

(1x1E03) EREP_NAME_NOTUNIQUE

See Also

Repository Errors

Meta Data Services Programming

Repository Errors (Alphabetical Order)

The error codes that can be returned as a part of the **HRESULT** return value by repository engine methods are listed here in alphabetical order, according to the symbolic name for each error code. These codes are also listed in numerical order. For more information, see Repository Errors (Numerical Order).

EREP_BADDRIVER (0x1002)

EREP_BADERROR (0x1003)

EREP_BADNAME (0x1001)

EREP_BADPARAMS (0x1000)

EREP_BLOB_CANNOTSETPOS (0x1A04)

EREP_BLOB_SEEKPASTEND (0x1A01)

EREP_BLOB_TEMPFILE (0x1A02)

EREP_BLOB_USERFILE (0x1A03)

EREP_BUFFER_OVERFLOW (0x1004)

EREP_CLASS_TOOCOMPLEX (0x1E00)

EREP_COL_NOTSEQUENCED (1x1950)

EREP_COL_OBJECTNAMING (0x1C00)

EREP_COL_OBJECTNOTNAMED (0x1C01)

EREP_DB_ALREADYCONNECTED (0x1032)

EREP_DB_CORRUPT (0x1034)

EREP_DB_DBMSOLD (0x1036)

EREP_DB_DBMSONETHREAD (0x1033)

EREP_DB_EXISTS (0x1030)

EREP_DB_INCOMPATIBLEVERSION (0x1038)

EREP_DB_NOSCHEMA (0x1035)

EREP_DB_NOTCONNECTED (0x1031)

EREP_DB_READONLY (0x1037)

EREP_DB_UPGRADE (0x1039)

EREP_DUPEDISPID (0x1072)

EREP_INVALIDFILTER(0x1D07)

EREP_ITEMNOTCHECKEDOUT (0x1803)

EREP_LOCK_TIMEOUT (0x1200)

EREP_MEMDEL_BASECOLVIRTUAL (1x1B02)

EREP_MEMDEL_BASEIFACENOTIMPL (1x1B01)

EREP_MEMDEL_CIRCULARCOLS (1x1B04)

EREP_MEMDEL_COLNOTDEFINED (1x1B00)

EREP_MEMDEL_DELCOLINVALID (0x1B05)

EREP_MEMDEL_MULTIPLEBASES (1x1B03)

EREP_MISSINGLEFTBRACKET (0x1D01)

EREP_MISSINGRIGHTBRACKET (0x1D02)

EREP_MISSINGLEFTPARENTHESIS (0x1D03)

EREP_MISSINGRIGHTPARENTHESIS (0x1D04)

EREP_MISSINGCOMMA (0x1D05)

EREP_NAME_NOTUNIQUE (1x1E03)

EREP_NAMETOOLONG (0x1005)

EREP_NEED_DATA (0x1014)

EREP_NOROWSFOUND (0x1011)

EREP_NOTWORKSPACEITEM (0x1802)

EREP_OBJ_EXISTS (0x1303)

EREP_OBJ_NONAMINGRELSHIP (0x1302)

EREP_OBJ_NOTFOUND (0x1301)

EREP_OBJ_NOTINITIALIZED (0x1300)

EREP_ODBC_CERROR (0x1012)

EREP_ODBC_CREATEFAILED (0x1016)

EREP_ODBC_MDBNOTFOUND (0x1013)

EREP_ODBC_NOTCAPABLE (0x1019)

EREP_ODBC_UNKNOWNDRIVER (0x1015)

EREP_ODBC_WARNINGS (0x1017)

EREP_PROP_CANTSETREPTIM (0x1403)

EREP_PROP_MISMATCH (0x1400)

EREP_PROP_NOTEXISTS (0x1405)

EREP_PROP_READONLY (0x1404)

EREP_PROP_SETINVALID (0x1401)

EREP_PROPERTYNOTFOUND(0x1D06)

EREP_QRY_BADCOLUMNS (0x1250)

EREP_REL_NONSEQONLY (1x1702)

EREP_REL_NOTPINNED (1x1704)

EREP_REL_ORGCLONE (1x1701)

EREP_REL_ORGFROZEN (1x1700)

EREP_REL_ORGPIN (1x1703)

EREP_RELSHIP_DUPENAME (0x1109)

EREP_RELSHIP_EXISTS (0x1100)

EREP_RELSHIP_INVALIDFLAGS (0x1107)

EREP_RELSHIP_INVALID_PAIR (0x1101)

EREP_RELSHIP_NAMEINVALID (0x1108)

EREP_RELSHIP_NONNAMINGCOL (0x1110)

EREP_RELSHIP_NOTFOUND (0x1102)

EREP_RELSHIP_ORGONLY (0x1105)

EREP_RELSHIP_OUTOFDATE (0x1106)

EREP_REPOS_CACHEFULL (0x1070)

EREP_REPOS_NONEXTDISPID (0x1071)

EREP_RTIM_CLASS_IS_NOT_CREATEABLE (0x1E02)

EREP_SCRIPT_INVALIDLANGUAGE (0x1D0A)

EREP_SCRIPT_NESTEDCALL (0x1D08)

EREP_SCRIPT_NOTFOUND (0x1D09)

EREP_STILL_EXECUTING (0x1018)

EREP_TIM_CTYPEINVALID (0x1503)

EREP_TIM_FLAGSDEST (0x1501)

EREP_TIM_INVALIDFLAGS (0x1500)

EREP_TIM_RELTYPEINVALID (0x1502)

EREP_TIM_SQLSIZEINVALID (0x1506)

EREP_TIM_SQLTYPEINVALID (0x1505)

EREP_TIM_TOOMANYCOLS (0x1504)

EREP_TXN_AUTOABORT (0x1042)

EREP_TXN_COLABORTED (0x1048)

EREP_TXN_NODATA (0x1045)

EREP_TXN_NOSETINTXN (0x1046)

EREP_TXN_NOTXNACTIVE (0x1041)

EREP_TXN_OBJABORTED (0x1047)

EREP_TXN_TIMEOUT (0x1044)

EREP_TXN_TOOMANY (0x1043)

EREP_TYPE_BADTABLENAME (0x1126)

EREP_TYPE_COLMISMATCH (0x1121)

EREP_TYPE_INVALIDSCALE (0x1125)

EREP_TYPE_INVERTEDNOTALLOWED (0x1124)

EREP_TYPE_MULTIDEFIFACES (0x1123)

EREP_TYPE_MULTIPLEANCESTORS (0x1127)

EREP_TYPE_NOTNULLABLE (0x1122)

EREP_TYPE_TABLEMISMATCH (0x1120)

EREP_UNKNOWNPROPERTY (0x1D00)

EREP_VCOL_INVALIDOP (1x1901)

EREP_VCOL_VERSIONNOTMEMBER (1x1900)

EREP_VERSION_NOTFOUND (0x1304)

EREP_VIRTUAL_ALIAS (0x1D0B)

EREP_VIRTUAL_CALL (0x1D0C)

EREP_VM_CANTSETFROZEN (0x1600)

EREP_VM_CHECKEDOUTVERSION (0x1605)

EREP_VM_DIFFERENTTYPES (0x2000)

EREP_VM_DUPBRANCHID (0x1606)

EREP_VM_FROZENVERSION (0x1604)

EREP_VM_MERGEFROMUNFROZEN (0x1602)

EREP_VM_MERGETOFROZEN (0x1601)

EREP_VM_SUCCESSOREXISTS (0x1607)

EREP_VM_UNFROZENVERSION (0x1603)

EREP_WKS_ITEMEXISTS (0x1800)

EREP_WKS_ITEMNOTEXISTS (0x1801)

SREP_PROP_TRUNCATION (0x1402)

See Also

Repository Errors

EREP_BADDRIVER (0x1002)

The currently installed ODBC driver is too old, and it is incompatible with the repository engine. To continue, update your ODBC driver.

EREP_BADERROR (0x1003)

An internal error has occurred. To continue, stop and then restart the repository engine.

EREP_BADNAME (0x1001)

The name that you have supplied for a table, view, or column name contains characters that are not valid, or it is a reserved word for the database management system (DBMS). To continue, change the name, and then try your request again.

EREP_BADPARAMS (0x1000)

One or more invalid parameters have been passed to a repository engine method. To continue, correct the input parameters, and then try again.

EREP_BLOB_SEEKPASTEND (0x1A01)

You have attempted a seek operation that is defined outside of the range of the data. To continue, verify the location of your data, and then reset the **CurrentPosition** property.

EREP_BLOB_TEMPFILE (0x1A02)

The repository engine cannot create or access a temporary file to read or write the data from a binary large object (BLOB) or large text field.

EREP_BLOB_USERFILE (0x1A03)

The repository engine cannot access the specified file.

EREP_BLOB_CANNOTSETPOS (0x1A04)

The repository engine cannot set the seek pointer to the specified position. As a result, the current position is unchanged.

EREP_BUFFER_OVERFLOW (0x1004)

An overflow error occurred while building an SQL statement. To continue, reduce the number of changed properties or repository objects in the operation, and then try again.

EREP_CLASS_TOOCOMPLEX (0x1E00)

You have specified a class that is too complex.

EREP_COL_NOTSEQUENCED (1x1950)

This operation cannot be performed on a nonsequenced collection.

EREP_COL_OBJECTNAMING (0x1C00)

Although the COLLECTION_OBJECTNAMING flag is set, a name that is specific to the relationship cannot be found for objects within this collection. Most likely, a specific name does not exist.

EREP_COL_OBJECTNOTNAMED (0x1c01)

You have attempted to add an object that does not support the **INamedObject** interface to a collection that requires all objects to support the **INamedObject** interface.

EREP_DB_ALREADYCONNECTED (0x1032)

You have attempted to connect to a repository database that is already open. To continue, skip the redundant **Open** or **Create** method invocation and proceed with the repository interactions that follow that **Open** invocation.

See Also

Repository Create Method

Repository Open Method

EREP_DB_CORRUPT (0x1034)

The repository database has been damaged. For more information about available facilities for restoring or rebuilding the database, see your database server documentation.

EREP_DB_DBMSOLD (0x1036)

This version of Microsoft® SQL Server™ is not supported by the repository engine. To use version 2.0 of the repository engine, you must upgrade to SQL Server version 6.5, SQL Server 7.0, or SQL Server 2000.

EREP_DB_DBMSONETHREAD (0x1033)

The repository database that you have attempted to access is managed by a database server that does not support multithreaded access. The thread attempting the access is not the same as the thread that currently has the open repository instance for the database. To continue, either move your repository database to a database server that supports multithreaded access, or modify the logic of your program to use a single thread for repository database access.

EREP_DB_EXISTS (0x1030)

You have requested that a repository database be created with a name that is already in use for an existing database. If you want to use the existing database, use the **Open** method instead of the **Create** method. If the existing database is no longer needed, delete it. Otherwise, choose a different name, and then try again.

See Also

Repository Create Method

Repository Open Method

EREP_DB_INCOMPATIBLEVERSION (0x1038)

The version of the database that you are using as a repository database is not supported by the repository engine.

EREP_DB_NOSCHEMA (0x1035)

The repository database does not contain the type information model schema. If your repository has not yet been populated with data, install the type information model schema by using the **Create** method to open the repository database. If your repository has been populated with data, restore the database from a backup copy.

See Also

Repository Create Method

EREP_DB_NOTCONNECTED (0x1031)

You have requested an operation that requires a connection to an open repository database, and you do not currently have such a connection. To continue, use the **Open** method on the appropriate repository and try your request again.

See Also

Repository Open Method

EREP_DB_READONLY (0x1037)

You have attempted to change a read-only database management system (DBMS). To continue, contact the system administrator.

EREP_DB_UPGRADE (0x1039)

The repository engine was unable to complete the upgrade operation. The repository SQL schema has not been updated.

EREP_DUPEDISPID (0x1072)

Duplicate dispatch identifiers have been found.

EREP_INVALIDDEPENDENCY (0x1C02)

You have attempted to define a dependency between a model and itself. You can define dependencies only between separate and distinct models.

EREP_INVALIDFILTER (0x1D07)

The filter could not be parsed. To continue, check the filter syntax and try again.

EREP_ITEMNOTCHECKEDOUT (0x1803)

This operation was performed on an item that was not checked out to a workspace.

EREP_LOCK_TIMEOUT (0x1200)

An attempt to obtain a lock on a repository item has timed out. To continue, either increase the lock time-out value and try again, or wait for the item to become available and then try again. For more information about changing the lock time-out value, see RepositoryTransaction SetOption Method.

EREP_MEMDEL_DELCOLINVALID (0x1B05)

The structure of a delegated collection is not valid.

EREP_MEMDEL_COLNOTDEFINED (1x1B00)

You have delegated a member in a different transaction in which the collection was created. To delegate a member, you must do so within the transaction in which the collection was instantiated.

EREP_MEMDEL_BASEIFACENOTIMPL (1x1B01)

This class does not support the interface that is the base interface for a delegated member.

EREP_MEMDEL_BASECOLVIRTUAL (1x1B02)

A base member of the delegated collection is virtual.

EREP_MEMDEL_MULTIPLEBASES (1x1B03)

A delegated member has more than one base member.

EREP_MEMDEL_CIRCULARCOLS (1x1B04)

A circular dependency has been created from delegated collections.

EREP_MISSINGCOMMA (0x1D05)

The INSTANCEOF or IMPLEMENTS clause is missing a comma.

EREP_MISSINGLEFTBRACKET (0x1D01)

The filter string is missing a left bracket.

EREP_MISSINGLEFTPARENTHESIS (0x1D03)

There is no left parenthesis following the INSTANCEOF or IMPLEMENTS clause.

EREP_MISSINGRIGHTBRACKET (0x1D02)

The filter string is missing a right bracket.

EREP_MISSINGRIGHTPARENTHESIS (0x1D04)

There is no right parenthesis following the INSTANCEOF or IMPLEMENTS clause.

EREP_NAME_NOTUNIQUE (1x1E03)

The name you have specified is not unique in the class.

EREP_NAMETOOLONG (0x1005)

The name you have specified exceeds the maximum length allowed for this string.

EREP_NEED_DATA (0x1014)

An ODBC error occurred indicating that a variable-length data item (such as a name) was needed at run time, and was never supplied. To continue, check input parameters.

EREP_NOROWSFOUND (0x1011)

A query operation against the repository database yielded no rows. If you expected data to be returned, verify that your query is correctly constructed.

EREP_NOTWORKSPACEITEM (0x1802)

This item is not a workspace item.

EREP_OBJ_EXISTS (0x1303)

You have attempted to create a repository object that already exists in the repository. This situation can occur if multiple users are attempting to add the same object to the repository concurrently. If this is not the case, eliminate the redundant **Add**, **CreateObject**, or **Insert** method invocation from your program.

See Also

RelationshipCol Add Method

RelationshipCol Insert Method

Repository CreateObject Method

EREP_ODBC_NOTCAPABLE (0x1019)

The ODBC driver does not support the current operation.

EREP_OBJ_NONAMINGRELSHIP (0x1302)

You have attempted to add an object to a collection using the object name, but the collection is not a naming collection.

EREP_OBJ_NOTFOUND (0x1301)

You have attempted to retrieve a repository object that does not exist. If multiple users are accessing the repository database concurrently, this error can occur if one user deletes a repository object while a second user is attempting to retrieve the object. It can also occur if you are using an object identifier that has been saved from prior interactions with the repository, and the object has been deleted between the time that you obtained the object identifier and the time that you attempted to retrieve the repository object. Consider handling this exception with special processing for the case where a repository object no longer exists.

EREP_OBJ_NOTINITIALIZED (0x1300)

An attempt has been made to interact with a repository object that has not been initialized with valid data from the repository database. To continue, ensure that all repository objects in your program are initialized before you attempt to interact with them.

EREP_ODBC_CERROR (0x1012)

A database error has occurred. To continue, check the error queue for more information. You may be able to determine the source of the problem and correct it before trying again.

EREP_ODBC_CREATEFAILED (0x1016)

The creation of an .mdb file has failed. Most likely, you either supplied a wrong path, or you tried to create an .mdb file that already existed. To continue, check the path or delete the .mdb file, and then try again.

EREP_ODBC_MDBNOTFOUND (0x1013)

You have specified a repository database that does not exist or is not accessible. To continue, make sure that the database exists and the name is correct, and then try again.

EREP_ODBC_UNKNOWNDRIVER (0x1015)

The specified ODBC driver is not a valid driver, or is not known to the repository engine. To continue, obtain an ODBC driver (2.0 or later) that is compatible with the repository engine.

EREP_ODBC_WARNINGS (0x1017)

The ODBC driver issued warnings. To continue, check the error queue for the error text.

EREP_PROP_CANTSETREPTIM (0x1403)

You have attempted to modify a property of a definition object that is part of the type information model. Modifying type information model properties is not supported.

EREP_PROP_MISMATCH (0x1400)

An attempt to update a property value in the repository has failed. The data type of the input property cannot be converted to the storage data type. To continue, correct the data type of the input property, and then try the update again.

EREP_PROP_NOTEXISTS (0x1405)

You have attempted to reference a property that does not exist. For repository 2.0 databases, this error is returned if you call **get_VersionID** on any Repository Type Information Model (RTIM) object, including the root object. For repository 3.0 databases, this error is returned if you call **get_VersionID** on any RTIM object, except the root object. To continue, check the property reference (name or dispatch identifier), and then try again.

EREP_PROP_READONLY (0x1404)

Your request to set the value of a property has failed because the property is a read-only property.

EREP_PROP_SETINVALID (0x1401)

You have attempted to modify a collection as if it were a property. The repository engine does not support this type of operation.

EREP_PROPERTYNOTFOUND (0x1D06)

No property was found between two brackets ([]). Check the syntax, and then try again.

EREP_QRY_BADCOLUMNS (0x1250)

An ad-hoc query is missing the **IntID** column or **TypeID** column.

EREP_REL_ORGFROZEN (1x1700)

This operation cannot be performed on a frozen origin object.

EREP_REL_ORGCLONE (1x1701)

A relationship can be cloned only by a version of the origin object.

EREP_REL_NONSEQONLY (1x1702)

This operation cannot be performed on a sequenced relationship.

EREP_REL_ORGPIN (1x1703)

You cannot pin or unpin an origin version.

EREP_REL_NOTPINNED (1x1704)

You cannot unpin a relationship that is not pinned.

EREP_RELSHIP_DUPENAME (0x1109)

You have attempted to add a relationship with a name that is not unique within the collection. The collection requires unique names. To continue, either choose a different name for the relationship or delete the existing relationship with the same name if it is no longer needed.

EREP_RELSHIP_EXISTS (0x1100)

You have attempted to create a relationship that already exists in the repository. To continue, either ignore this error, or eliminate the redundant **Add** or **Insert** method invocation from your program.

See Also

RelationshipCol Add Method

RelationshipCol Insert Method

EREP_RELSHIP_INVALIDFLAGS (0x1107)

Your attempt to add or modify a **Relationship** collection has failed. Either the combinations of flags are invalid, or you are attempting to set flag values on a destination collection. To continue, verify that the origin collection is being used for the operation, and that the flag combinations are valid. For more information about relationship flags, see CollectionDefFlags_Enumeration.

EREP_RELSHIP_INVALID_PAIR (0x1101)

An attempt to add a new relationship between two objects has failed. One or both of the classes to which these objects conform does not support this type of relationship. To continue, verify that the relationship type and the object classes are correct, and then check your information model to verify that it supports the type of relationship that you are trying to create.

EREP_RELSHIP_NAMEINVALID (0x1108)

You have attempted to add a relationship that has an invalid name specified for the destination object. To continue, verify that the name is nonnull and is shorter than the maximum allowed length. For more information about repository text string lengths, see Repository Constants.

EREP_RELSHIP_NONNAMINGCOL (0x1110

The repository engine is unable to perform the current operation on a nonnaming collection.

EREP_RELSHIP_NOTFOUND (0x1102)

You have attempted to retrieve a specific relationship that does not exist, or you have attempted to retrieve a relationship from an empty collection. If multiple users are accessing the repository concurrently, this error can occur if one user deletes a relationship while a second user is attempting to retrieve the relationship. Consider handling this exception with special processing for the case where a collection is empty or a specific relationship no longer exists.

EREP_RELSHIP_ORGONLY (0x1105)

An attempt to move or insert a relationship in a sequenced collection has failed because the **Move** or **Insert** method was invoked through the destination object instead of the origin object. To continue, use the origin object to move or insert a relationship in a sequenced collection.

See Also

RelationshipCol Insert Method

RelationshipCol Move Method

EREP_RELSHIP_OUTOFDATE (0x1106)

Your request has failed because the sequenced **Relationship** collection that you are attempting to update has been changed by another process. To continue, refresh the collection, and then try the update again.

EREP_REPOS_CACHEFULL (0x1070)

The repository engine cache is full. If you are writing new and changed data to the repository, and you cannot reduce the number of steps in the transaction, consider releasing some object references to create additional free space.

See Also

TransactionFlags Enumeration

EREP_REPOS_NONEXTDISPID (0x1071)

You have attempted to add a member to an interface that is defined in the repository engine, but there are no more dispatch identifier values available. To continue, factor the interface into several smaller interfaces.

EREP_RTIM_CLASS_IS_NOT_CREATEABLE (0x1E02)

The repository type class that you defined cannot be created for the information model.

EREP_SCRIPT_INVALIDLANGUAGE (0x1D0A)

The script engine is not installed.

EREP_SCRIPT_NESTEDCALL (0x1D08)

The repository engine detected a nested call in a script. This error occurs when you nest a call within script while the NESTEDSCRIPT flag is set to FALSE.

EREP_SCRIPT_NOTFOUND (0x1D09)

The script object associated with this method or property is either undefined, or it is unrelated. For more information about how the repository engine selects script objects, see ScriptDef Object.

EREP_STILL_EXECUTING (0x1018)

A statement you have executed is still in progress.

EREP_TIM_CTYPEINVALID (0x1503)

You have chosen an invalid C data type for a property. To continue, use a valid C data type.

EREP_TIM_FLAGSDEST (0x1501)

You have attempted to set a collection flag on a destination collection.

EREP_TIM_INVALIDFLAGS (0x1500)

You have specified an invalid combination of **CollectionDef** bit flags.

EREP_TIM_RELTYPEINVALID (0x1502)

The type of a **RelationshipDef** object for a collection is incorrect.

EREP_TIM_SQLTYPEINVALID (0x1505)

You have chosen an invalid SQL data type for a property. To continue, use a valid SQL data type.

EREP_TIM_SQLSIZEINVALID (0x1506)

You have chosen an invalid SQL size for a property data type. To continue, use a valid SQL size.

EREP_TIM_TOOMANYCOLS (0x1504)

The number of collections in use exceeds the maximum allowed for the **RelationshipDef** object. To continue, release the collections that are not in use.

EREP_TXN_AUTOABORT (0x1042)

Resources for an open repository instance were released while a transaction was in progress. The transaction has been canceled; all changes associated with the transaction will be rolled back. To prevent this error in the future, complete an active transaction (through either the **Commit** or the **Abort** method) before releasing an open repository instance.

See Also

RepositoryTransaction Abort Method

RepositoryTransaction Commit Method

EREP_TXN_COLABORTED (0x1048)

The collection has been deleted, or the last transaction that updated the collection has been stopped. In the latter case, release all the pointers to the collection, and then reinstantiate it.

EREP_TXN_NODATA (0x1045)

You have attempted to retrieve the value of a property that is null or does not exist. The action you decide to take depends on the requirements of your task. If the property has a null value, consider handling this exception with special processing.

EREP_TXN_NOSETINTXN (0x1046)

You have attempted to modify the current transaction option settings for an active transaction. To continue, either complete the current transaction and then modify the transaction options or set the transaction options before beginning the transaction.

EREP_TXN_NOTXNACTIVE (0x1041)

You have attempted to update the repository database, but no transaction is active. To continue, bracket your repository updates between **Begin** and **Commit** transaction method invocations.

See Also

RepositoryTransaction Begin Method

RepositoryTransaction Commit Method

EREP_TXN_OBJABORTED (0x1047)

The object was created during a transaction that was stopped. To continue, release all the pointers to the object, and then reinstantiate it.

EREP_TXN_TIMEOUT (0x1044)

This error occurs for query time-outs and transaction time-outs. If you are querying a repository database, the amount of time that the repository engine waits for a query to complete elapsed before the query returned a result. To continue, increase the query time-out value.

If you are attempting to start a transaction, your transaction timed out while waiting to begin. To continue, either increase the start transaction time-out value and retry the transaction or wait for the item to become available and then retry the transaction.

For more information about transaction options, see TransactionFlags Enumeration. For more information about changing the transaction time-out values, see RepositoryTransaction SetOption Method or IRepositoryTransaction::SetOption.

Note DTS users and other tool users who issue queries for large amounts of data can set a registry key to workaround this error. In this case, create a new entry for HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Repository\Engine\ODBC(and set it to large value. Query time-out values are measured in seconds.

EREP_TXN_TOOMANY (0x1043)

A new transaction cannot be started because the maximum number of concurrent transactions has been exceeded. To continue, reduce the number of transactions that are concurrently executing within the same process.

EREP_TYPE_BADTABLENAME (0x1126)

The string specified for the table name is invalid. Most likely, it contains invalid characters.

EREP_TYPE_COLMISMATCH (0x1121)

The conversion of a property value between the stored data type and the data type as specified by the caller has failed. To continue, check the caller-specified data type to verify that it can be converted to the storage data type, as defined by the associated property definition object.

EREP_TYPE_INVALIDSCALE (0x1125)

You have attempted to set the **PropertyDef SQLScale** property of a property definition to an invalid value. To continue, correct the value that you are using, and then try the operation again.

See Also

PropertyDef SQLScale Property

EREP_TYPE_INVERTEDNOTALLOWED (0x1124)

You have attempted to add a property to an interface using the PROPERTY_INVERTED option, and the option is not permitted for the interface. To continue, correct either the property definition or the interface definition.

EREP_TYPE_MULTIDEFIFACES (0x1123)

You have attempted to set more than one interface as the default interface for a class definition. To continue, choose one of the interfaces to be the default interface.

EREP_TYPE_MULTIPLEANCESTORS (0x1127)

There is more than one ancestor specified for the current interface.

EREP_TYPE_NOTNULLABLE (0x1122)

You have attempted to set a property value to the null value, and the property definition does not allow this. To continue, choose one of the permitted property values, and then try the update operation again.

EREP_TYPE_TABLEMISMATCH (0x1120)

An attempt to extend an interface for an information model has failed. The SQL table that is designated as the table to be used for storing property values for the interface does not contain the expected columns. To continue, check the table to determine whether it has been damaged or whether columns have been dropped from the table. You can then restore the table to its prior state and try the request again.

EREP_UNKNOWNPROPERTY (0x1D00)

The property name inside the brackets ([]) could not be resolved. To continue, check the property name, and then try again.

EREP_VCOL_INVALIDOP (1x1901)

This is not a valid operation for collections.

EREP_VCOL_VERSIONNOTMEMBER (1x1900)

This version is not a member of the version collection.

EREP_VERSION_NOTFOUND (0x1304)

The version of the repository object you selected cannot be found.

EREP_VIRTUAL_ALIAS (0x1D0B)

You cannot specify an alias as a virtual property.

EREP_VIRTUAL_CALL (0x1D0C)

The virtual member you specified cannot be called.

EREP_VM_CANTSETFROZEN (0x1600)

You cannot set a property on an object that has been frozen.

EREP_VM_MERGETOFROZEN (0x1601)

You cannot perform a merge operation on an object that has been frozen.

EREP_VM_MERGEFROMUNFROZEN (0x1602)

You cannot perform a merge operation with an unfrozen, secondary version.

EREP_VM_UNFROZENVERSION (0x1603)

This operation cannot be performed on an unfrozen version.

EREP_VM_FROZENVERSION (0x1604)

This operation cannot be performed on a frozen version.

EREP_VM_CHECKEDOUTVERSION (0x1605)

This operation cannot be performed on a checked-out version.

EREP_VM_DUPBRANCHID (0x1606)

A duplicate branch ID was generated for this object.

EREP_VM_SUCCESSOREXISTS (0x1607)

A successor of the version exists. You cannot delete an object version if a successor exists.

EREP_VM_DIFFERENTTYPES (0x2000)

You cannot perform a merge operation on objects of different types.

EREP_WKS_ITEMEXISTS (0x1800)

This item already exists in the workspace. You can have only one version of each object in a workspace.

EREP_WKS_ITEMNOTEXISTS (0x1801)

The item that you selected does not exist in the workspace.

SREP_PROP_TRUNCATION (0x1402)

Your request to set the value of a property has succeeded; however, the value of the property has been truncated because the input property value was too long.

Meta Data Services Programming

Repository SQL Schema

The repository SQL schema is a mapping of information model elements to SQL schema elements. The repository engine uses data in these tables to instantiate and manage COM objects. The repository SQL schema consists of a standard schema and an extended schema.

- The standard schema consists of tables that contain the core information needed to manage all repository objects, relationships, and collections. The standard schema also contains tables that are used by Microsoft® SQL Server™ 2000 Meta Data Services to store the definition information for information models. Standard schema tables are prefixed with **RTbl**.

If you obtained Meta Data Services through SQL Server, repository SQL schema tables are located in the **msdb** system database.

- The extended schema consists of tables that are automatically generated by the repository engine when you create or extend an information model. An interface is mapped to at most one table in a repository database. The table contains the instance data for persistent properties that are attached to the interface. One column in the table is created for each property. If an interface is defined that has no member properties, no table is created.

Adding Data to Repository SQL Schema

You can add data to the repository SQL schema when you install an information model or create an information model programmatically. When you use a SQL Server database for your repository storage, the repository engine creates stored procedures to insert the data. For more information about how these stored procedures are named, see Naming Stored Procedures.

Tuning the Extended Schema

Although the extended schema is automatically generated, experienced model designers can tune the extended schema to optimize performance and data

retrieval. For example, by default, the properties of each interface are stored in a separate SQL table. You can map the properties of multiple interfaces to a single table. You can also specify the column names and data types to be used for property data. You can add indexes to tables, but you must not remove indexes that have been automatically defined by Meta Data Services. For more information, see Tuning the Database Schema of an Information Model.

Querying the Repository

You can construct an SQL query to extract specific information from a repository. Although it is simpler to perform queries through generated views, you can manually build an SQL query against the repository SQL schema if you want a result set that covers more than one information model. To build such a query, you must be familiar with the repository tables. For more information about querying, see Repository SQL Tables and Generating Views.

See Also

Repository Databases

Repository SQL Data Types

Storage Strategy in a Repository Database

Meta Data Services Programming

Repository SQL Tables

The set of SQL tables that make up the standard schema is shown in the following table. For more information about the standard schema, see Repository SQL Schema.

SQL table name	Description
RTblClassDefs	Stores **ClassDef** instance data
RTblDatabaseVersion	Stores the version and the build of the engine that created the repository database
RTblEnumerationDef	Stores **EnumerationDef** instance data
RTblEnumerationValueDef	Stores property values of enumerated properties
RTblIfaceDefs	Stores **InterfaceDef** instance data
RTblIfaceHier	Contains information about interface hierarchies
RTblIfaceMem	Contains information about interface members
RTblNamedObj	Contains values of the **Name** property exposed by the **INamedObject** interface
RTblParameterDef	Stores **ParameterDef** instance data
RTblPropDefs	Stores **PropertyDef** instance data
RTblProps	Stores property values of annotational properties that are attached to repository objects
RTblRelColDefs	Stores **CollectionDef** instance data
RTblRelshipDefs	Stores **RelationshipDef** instance data
RTblRelshipProps	Stores property values of annotational properties that are attached to relationships
RTblRelships	Stores instance data for each version combination present in a two-way versioned relationship
RTblScriptDefs	Stores **ScriptDef** instance data
RTblSites	Stores translations of local site identifiers to global site identifiers
RTblSumInfo	Contains values of the properties exposed by

	the **ISummaryInformation** interface
RTblTypeInfo	Contains information about type information
RTblTypeLibs	Contains information about repository type libraries
RTblVersionAdminInfo	Contains information about the properties exposed by the **IVersionAdminInfo** interface
RTblVersions	Contains information about repository object versions
RTblWorkspaceItems	Contains information about the inclusion of object versions in workspaces

See Also

Repository SQL Data Types

Repository Databases

Storage Strategy in a Repository Database

Meta Data Services Programming

RTblClassDefs SQL Table

RTblClassDefs contains one row for each class that is defined in a repository database.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the class object.
Z_BranchID_Z	**RTBrID**	The branch identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VS_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VE_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always VERINFINITY.
ClassID	**RTGUID**	The global identifier of the class, as recorded in the system registry.
VerPropDescs	**Image**, 16 bytes	Definition information for the class. It is reserved for proprietary use by the repository engine. This field can be NULL. The maximum length for this value is 16 bytes.
PropDescs	**Image**, 16 bytes	This column supports backward compatibility with **RTblClassDefs** in version 1.0. In version 2.0 and later, the value of this column is always NULL. The maximum length for this value is 16 bytes.
ViewName	**Varchar**, 128 bytes	Specifies a user-defined view name for an SQL view based on the class. View generation is supported on Microsoft® SQL Server™ 2000 databases only. The maximum length for this value is 128 bytes. For more information,

		see IViewClassDef Interface.
ViewFlags	**Integer**, 4 bytes	Specifies whether view generation is supported by the class. This value is provided by the **ViewFlags** property. The maximum length for this value is 4 bytes.

Remarks

The **RTblClassDefs** table stores instance data for **ClassDef** objects that you define.

The primary key for this table is formed from the **IntID**, **Z_BranchID_Z**, and **Z_VS_Z** columns. A unique index is defined on the same set of columns.

Z_BranchID_Z, **Z_VS_Z**, and **Z_VE_Z** are included to provide future support for versioning repository API definitions. At this time, there is only one version of **ClassDef**.

See Also

ClassDef Class

ClassDef Object

Defining a Class View

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

RTblVersionAdminInfo SQL Table

RTblVersions SQL Table

Meta Data Services Programming

RTblDatabaseVersion SQL Table

RTblDatabaseVersion contains the version and the build of the repository engine that created the repository database.

Column name	Data type	Description
DatabaseVersion	**RTDBVersion**	The version and the build of the engine in the following format: V1.V2.B1.B2 where: V1: major version number. V2: minor version number. B1: major build number. B2: minor build number. For example, 3.0.6019.0 means that the repository database was created using the engine 3.0 and the build 6019.0.

See Also

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

Meta Data Services Programming

RTblEnumerationDef SQL Table

RTblEnumationDef contains one row for each enumeration definition that is defined in a repository database.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the interface definition object.
Z_BranchID_Z	**RTBrID**	The branch identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VS_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VE_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always VERINFINITY.
IsFlag	**RTBoolean**	A TRUE/FALSE value that indicates whether the enumeration defines a logical flag. This flag applies to numeric enumeration values only.

Remarks

The **RTblEnumationDef** table stores instance data for **EnumerationDef** objects that you define. Enumeration values for an enumeration object are stored in **RTblEnumerationValueDef**.

The primary key for this table is formed from the **IntID**, **Z_BranchID_Z**, and **Z_VS_Z** columns.

Z_BranchID_Z, **Z_VS_Z**, and **Z_VE_Z** are included to provide future support

for versioning repository API definitions. At this time, there is only one version of **EnumerationDef**.

See Also

EnumerationDef Class

EnumerationDef Object

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

RTblEnumerationValueDef SQL Table

RTblVersionAdminInfo SQL Table

RTblVersions SQL Table

Meta Data Services Programming

RTblEnumerationValueDef SQL Table

RTblEnumationValueDef stores enumeration values associated with an enumeration definition object.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the interface definition object.
Z_BranchID_Z	**RTBrID**	The branch identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VS_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VE_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always VERINFINITY.
EnumValue	**Text**, 16 bytes	A string containing a value that can be stored in the property value of an object. The maximum length for this value is 16 bytes.

RTblEnumationValueDef table stores instance data for **EnumerationValueDef** objects that you define.

The primary key for this table is formed from the **IntID**, **Z_BranchID_Z**, and **Z_VS_Z** columns.

Z_BranchID_Z, **Z_VS_Z**, and **Z_VE_Z** are included to provide future support for versioning repository API definitions in case more than one version of this

repository API structure is created.

See Also

EnumerationValueDef Class

EnumerationValueDef Object

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

RTblEnumerationDef SQL Table

RTblVersionAdminInfo SQL Table

RTblVersions SQL Table

Meta Data Services Programming

RTblIfaceDefs SQL Table

RTblIfaceDefs contains one row for each interface that is defined in a repository database.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the interface definition object.
Z_BranchID_Z	**RTBrID**	The branch identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VS_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VE_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always VERINFINITY.
InterfaceID	**RTGUID**	The global identifier of the interface, as recorded in the system registry.
SQLTableName	**Varchar**, 255 bytes	The name of the SQL table used to store property instance data for the interface. This field can be NULL. The maximum length for this value is 255 bytes.
Flags	**RTFlags**	Flags that determine interface behavior. This value is provided by the **InterfaceDef Flags** property. For more information, see InterfaceDefFlags Enumeration.
ViewName	**Varchar**,	Specifies a user-defined view name for

	128 bytes	an SQL view based on the interface. View generation is supported on Microsoft® SQL Server™ 2000 databases only. The maximum length for this value is 128 bytes. For more information, see IViewInterfaceDef Interface.
ViewFlags	**Integer**, 4 bytes	Specifies whether view generation is supported by the interface. This value is provided by the **ViewFlags** property. The maximum length for this value is 4 bytes.

Remarks

The **RTblIfaceDefs** table stores instance data for **InterfaceDef** objects that you define.

The primary key for this table is formed from the **IntID**, **Z_BranchID_Z**, and **Z_VS_Z** columns. A unique index is defined on the same set of columns.

Z_BranchID_Z, **Z_VS_Z**, and **Z_VE_Z** are included to provide future support for versioning repository API definitions. At this time, there is only one version of **InterfaceDef**.

See Also

Defining an Interface View

InterfaceDef Class

InterfaceDef Object

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

RTblVersionAdminInfo SQL Table

RTblVersions SQL Table

Meta Data Services Programming

RTblIfaceHier SQL Table

RTblIfaceHier stores the transitive closure of the interface hierarchy.

Column name	Data type	Description
BaseID	**RTIntID**	The internal identifier for a base **InterfaceDef** object
AncestorID	**RTIntID**	The internal identifier for an **InterfaceDef** object that is an ancestor of the base **InterfaceDef** object

Remarks

The **RTblIfaceHier** table maintains mapping information that supports circular and extended interface relationships. In this table, complex chains of inheritance are broken down into a series of **BaseID** and **AncestorID** pairs until the complete inheritance relationship is expressed as isolated pairs of interfaces.

Interface inheritance represents a many-to-many relationship. An interface identifier can be an **AncestorID** column in one pairing and a **BaseID** column in another pairing. All combinations of interface pairs, whether implicitly or explicitly related, are expressed in the **RTblIfaceHier** table.

The primary key for this table is formed by the **BaseID** and **AncestorID** columns.

See Also

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

Meta Data Services Programming

RTblIfaceMem SQL Table

RTblIfaceMem contains one row for each member of an interface. Interface members include property definitions, method definitions, and collection definitions stored in a repository database.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the member definition object.
Z_BranchID_Z	**RTBrID**	The branch identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VS_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VE_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always VERINFINITY.
DispID	**RTDispID**	The Automation dispatch identifier for the member. This field can be NULL.
Flags	**RTFlags**	Flags that determine member behavior. For more information about flag values, see InterfaceMemberFlags Enumeration.
MemberSynonym	**Varchar**, 255 bytes	A string used as an alias name. The maximum length for this data type is 255 bytes.

Remarks

The **RTblIfaceMem** table stores instance data for members of interfaces. The information contained in this table is used by the repository engine to create an extended schema (or one or more interface-specific SQL tables) when an interface is added.

The primary key for this table is formed from the **IntID**, **Z_BranchID_Z**, and **Z_VS_Z** columns.

Z_BranchID_Z, **Z_VS_Z**, and **Z_VE_Z** are included to provide future support for versioning repository API definitions in case more than one version of this repository API structure is created.

See Also

CollectionDef Class

CollectionDef Object

MethodDef Class

MethodDef Object

PropertyDef Class

PropertyDef Object

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

Meta Data Services Programming

RTblNamedObj SQL Table

RTblNamedObj stores instance data of the **Name** property exposed through the **INamedObject** interface of a repository object.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier of the class
Z_BranchID_Z	**RTBrID**	Indicates the branch of the version graph containing the range to whose items the property values in this row apply
Z_VS_Z	**RTVerID**	A version-within-branch identifier indicating the lower limit of the range to whose items the property values in this row apply
Z_VE_Z	**RTVerID**	A version-within-branch identifier indicating the upper limit of the range to whose items the property values in this row apply
Name	**Text**	The name of the object, as specified by the **Name** property of the **INamedObject** interface

Remarks

The **RTblNamedObj** table is an interface-specific table; its columns correspond to the properties exposed by the **INamedObject** interface. If you create a custom interface, you must implement **INamedObject** if you want to use the **Name** property to refer to an object.

See Also

INamedObject Interface

InterfaceDef Class

Meta Data Services Programming

RTblParameterDef SQL Table

RTblParameterDef stores parameter data associated with method definitions.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the member definition object.
Z_BranchID_Z	**RTBrID**	The branch identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VS_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VE_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always VERINFINITY.
ParamFlags	**RTFlags**	A flag that defines whether the parameter is optional, passed by reference or value, or has a return value. For more information about flag values, see IParameterDef Flags Property.
ParamType	**RTFlags**	The data type of the parameter, which can be any variable type supported by an Automation interface.
ParamDesc	**Varchar**, 255 bytes	A string placed into an Interface Definition Language (IDL) file instead of the default text for the parameter type. The maximum length for this value is 255 bytes.

ParamDefault	**Varchar**, 255 bytes	A string that denotes the default value for the parameter. The maximum length for this value is 255 bytes.
ParamGUID	**RTGUID**	A GUID that defines the interface ID of a **VT_DISPATCH** or **VT_UNKNOWN** object.

Remarks

The **RTblIParameterDef** table stores parameter definitions associated with **MethodDef** objects.

The primary key for this table is formed from the **IntID**, **Z_BranchID_Z**, and **Z_VS_Z** columns.

Z_BranchID_Z, **Z_VS_Z**, and **Z_VE_Z** are included to provide future support for versioning repository API definitions. At this time, there is only one version of **ParameterDef**.

See Also

Defining a Parameter

IParameterDef Interface

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

RTblVersionAdminInfo SQL Table

RTblVersions SQL Table

Meta Data Services Programming

RTblPropDefs SQL Table

RTblPropDefs contains one row for each property definition object that is stored in a repository database.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the property definition object.
Z_BranchID_Z	**RTBrID**	The branch identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VS_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VE_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always VERINFINITY.
SQLColName	**Varchar**, 255 bytes	The name of the column in the SQL table for this property. This field can be NULL. The maximum length for this value is 255 bytes.
APIType	**RTSQLType**	The C language data type for the property. This is the type of the property when it is passed through a repository programming interface.
SQLType	**RTSQLType**	The SQL data type for the property.
SQLSize	**RTSize**	The length in bytes of the property in terms of its SQL data type.
SQLScale	**RTScale**	The scale for a numeric property; the

		number of digits after the decimal point. This field can be NULL.
Flags	**RTFlags**	Flags that determine property behavior. For more information about flag values, see PropertyDef Flags Property.
ViewColumnName	**Varchar**, 128 bytes	A user-defined name applied to a view column. The maximum length for this value is 128 bytes.
SQLBlobSize	**Integer**, 4 bytes	The maximum size of a property definition. The maximum length for this value is 4 bytes.

Remarks

The **RTblPropDefs** table stores instance data for **PropertyDef** objects you create. The repository engine uses information contained in this table to create an extended schema (or interface-specific SQL tables) when an interface is added.

Note Annotational properties are not version-specific. Annotational properties that you create apply to the repository object as a whole.

The primary key for this table is formed from the **IntID**, **Z_BranchID_Z**, and **Z_VS_Z** columns.

Z_BranchID_Z, **Z_VS_Z**, and **Z_VE_Z** are included to provide future support for versioning repository API definitions. At this time, there is only one version of **PropertyDef**.

See Also

IViewPropertyDef Interface

PropertyDef Class

PropertyDef Object

Meta Data Services Programming

RTblProps SQL Table

RTblProps stores one row for each annotational property instance that is attached to a repository object.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the object to which this annotational property is attached.
PropID	**RTIntID**	The internal identifier of a property definition object. A **PropertyDef** object is a prerequisite to using annotational properties.
PropValue	**RTShortString**	The value of the annotational property instance.

Remarks

The **RTblProps** table stores instances of annotational properties that you define for a repository object. An annotational property associates a user-defined text string with a specific repository object. User-defined text strings are stored in this table. A similar table stores data for relationships. For more information, see RTblRelshipProps SQL Table.

The primary key for this table is formed from the **IntID** and **PropID** columns. A nonunique index is defined on concatenated values from the **PropID** and **PropValue** columns.

See Also

PropertyDef Class

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

Meta Data Services Programming

RTblRelColDefs SQL Table

RTblRelColDefs stores one row for each collection type defined in the repository database.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the collection definition object.
Z_BranchID_Z	**RTBrID**	The branch identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VS_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VE_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always VERINFINITY.
RelTypeID	**RTIntID**	Internal identifier of the relationship definition object.
Flags	**RTFlags**	Flags that determine the behavior of collections that conform to this collection type.
MinCount	**RTCount**	The minimum number of repository items that can occur in a collection of this type. This field can be NULL. This property is not enforced by the repository engine.
MaxCount	**RTCount**	The maximum number of repository items that can occur in a collection of this type. This field can be NULL. This

		property is not enforced by the repository engine.
IsOrigin	**RTBoolean**	Determines whether collections that conform to this collection type are origin collections (True), or destination collections (False).

Remarks

The **RTblRelColDefs** associates relationship objects with a collection type. Collection types are distinguished by the **CollectionDefFlag** value. Flag values determine collection characteristics, while **IsOrigin** determines the collection type.

Each relationship in a repository is associated with two relationship collections: an origin collection and a destination collection. Each relationship collection conforms to a collection type. The collection type defines the role that the collection plays in the relationship.

The primary key for this table is formed from the **IntID** and **PropID** columns. A nonunique index is defined on concatenated values of the **PropID** and **PropValue** columns.

Z_BranchID_Z, **Z_VS_Z**, and **Z_VE_Z** are included to provide future support for versioning repository API definitions in case more than one version of this repository API structure is created.

See Also

CollectionDef Class

CollectionDef Object

CollectionDefFlags Enumeration

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

Meta Data Services Programming

RTblRelshipDefs SQL Table

RTblRelshipDefs stores persistent properties associated with relationship definitions defined in the repository database.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the relationship definition object.
Z_BranchID_Z	**RTBrID**	The branch identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VS_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VE_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always VERINFINITY.
ViewFlags	**Integer**, 4 bytes	A set of flags that determine the view generation behavior for relationship definitions. The maximum length for this value is 4 bytes. For more information about flag values, see ViewFlags Property.
ColumnNamePrefix	**Varchar**, 118 bytes	A string prefixed to the column name **NAME**, **PrevDstID**, and **RelTypeID**. The string is used in all views where the corresponding columns appear. The maximum length for this value is 118 bytes.

JunctionViewName	**Varchar**, 128 bytes	Specifies a user-defined view name for an SQL view based on the relationship. View generation is supported on Microsoft® SQL Server™ 2000 databases only. The maximum length for this value is 128 bytes. For more information, see IViewRelationshipDef Interface.

Remarks

The **RTblRelshipDefs** table stores values that direct view generation behavior for relationship definitions and indicate whether to create a junction-table view of a relationship. Another SQL table stores relationship instance data. For more information, see RTblRelships SQL Table.

The primary key for this table is formed from the **IntID**, **Z_BranchID_Z**, and **Z_VS_Z** columns.

Z_BranchID_Z, **Z_VS_Z**, and **Z_VE_Z** are included to provide future support for versioning repository API definitions in case more than one version of this repository API structure is created.

See Also

Defining a Junction Table View

Generating Views

RelationshipDef Class

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

Meta Data Services Programming

RTblRelshipProps SQL Table

RTblRelshipProps stores one row for each annotational property instance that is attached to a relationship definition.

Column name	Data type	Description
OrgID	**RTIntID**	The internal identifier for the origin object of the relationship.
RelTypeID	**RTIntID**	The internal identifier for the relationship type.
DstID	**RTIntID**	The internal identifier for the destination object of the relationship.
PropID	**RTIntID**	The internal identifier of a property definition object. A **PropertyDef** object is a prerequisite to using annotational properties.
PropValue	**RTShortString**	The value of the annotational property instance.

Remarks

The **RTblRelshipProps** table stores instances of annotational properties that you define for a relationship definition. An annotational property associates a user-defined text string with a specific relationship definition. User-defined text strings are stored in this table. A similar table stores data for repository objects. For more information, see RTblProps SQL Table.

Note Annotational properties are not version-specific. Annotational properties that you create apply to the repository object as a whole.

The primary key for this table is formed from the **OrgID**, **RelTypeID**, **DstID**, and **PropID** columns. A single nonunique index is defined on the concatenation of the **PropID** and **PropValue** columns.

See Also

Meta Data Services Programming

RTblRelships SQL Table

RTblRelships stores instance data for each version combination present in a two-way versioned relationship.

Column name	Data type	Description
OrgID	**RTIntID**	The internal identifier for the origin object of the relationship.
Z_OrgBrID_Z	**RTBrID**	Indicates the branch of the version graph containing the origin object version.
Z_OrgVS_Z	**RTVerID**	Indicates the lower limit of the range of origin object versions that have version-to-version relationships described by this row.
Z_OrgVE_Z	**RTVerID**	Indicates the upper limit of the range of origin object versions that have version-to-version relationships described by this row. It can be a special value indicating that the range is unlimited.
Z_OrgLClock_Z	**RTLClock**	For internal use only.
DstID	**RTIntID**	The internal identifier for the destination object of the relationship.
Z_DstBrID_Z	**RTBrID**	For relationship rows, indicates the branch of the version graph containing the destination object version. For auxiliary rows, it can indicate the branch containing the pinned object version of an origin versioned relationship.
Z_DstVS_Z	**RTVerID**	For relationship rows, indicates the lower limit of the range of destination object versions that have version-to-

		version relationships described by this row. For auxiliary rows, it can indicate the pinned object version of an origin versioned relationship.
Z_DstVE_Z	**RTVerID**	Indicates the upper limit of the range of destination object versions having version-to-version relationships described by this row.
Z_DstLClock_Z	**RTLClock**	For internal use only.
OrgTypeID	**RTIntID**	The internal identifier for the class to which the origin object conforms. It is redundantly stored in this table for performance reasons.
RelTypeID	**RTIntID**	The internal identifier for the relationship type.
DstTypeID	**RTIntID**	The internal identifier for the class to which the destination object conforms. It is redundantly stored in this table for performance reasons.
PrevDstID	**RTIntID**	This property is NULL for every relationship row and any auxiliary row of a nonsequenced relationship. For an auxiliary row describing an item in a sequenced relationship collection, this column has a nonNULL value that refers to the previous relationship in the sequenced collection. Specifically, the value is the internal identifier of the previous relationship; that is, the value in the **DstID** column of the relationship row describing that previous relationship.
DstName	**RTNameString**	The name of the destination object. More precisely, the name (as defined

		by this naming relationship) by which each origin version (in the range of origin versions) refers to each destination version (in the range of destination versions). If the relationship is not a naming relationship, then this field is NULL.
DstNameLong	**Text**, 16 bytes	If the name of the destination object is too long for the **DstName** field, this field contains the full name. Otherwise, this field is NULL. The maximum length for this value is 16 bytes.
Z_RelFlags_Z	**RTFlags**	A value of 2 indicates that the row is a relationship row; a value of 1 indicates it is an auxiliary row.

Remarks

The **RTblRelships** table stores information about each object version combination that exists in a two-way versioned relationship. Within a relationship where both objects are versioned, multiple versions can exist for each object pairing. For example, one instance of a relationship may associate Object_X version 3 with Object_Y version 2, a second instance may associate Object_X version 4 with Object_Y version 5, and so on. This table tracks data about each combination of versioned objects.

Another SQL table stores relationship definition properties. For more information, see RTblRelshipDefs SQL Table.

The primary key for this table is formed from the **OrgID**, **Z_OrgBrID_Z**, **Z_OrgVS_Z**, **DstID**, **Z_DstBrID_Z**, **Z_DstVS_Z**, **RelTypeID**, and **Z_RelFlags_Z** columns.

Examples

The **RTblRelships** table stores two kinds of rows: relationship rows and auxiliary rows. Relationship rows store data about specific combinations of

versioned objects. Auxiliary rows contain pinning and sequence information for an origin versioned relationship. Examples illustrate each case and explain how to interpret instance data in the table.

Relationship Row Examples

- In the simplest case, a relationship row describes exactly one version-to-version relationship. For more information, see RTblRelships Example One RTblRelships Example One.

- In the ideal case, a relationship row describes as large a range as possible. For more information, see RTblRelships Example Two.

- A relationship row can describe an unbounded range. For more information, see RTblRelships Example Three.

- In some situations, several relationship rows exist when one would theoretically suffice. For more information, see RTblRelships Example Four.

Auxiliary Row Examples

- An auxiliary row can contain sequencing information. For more information, see RTblRelships Example Five.

- An auxiliary row can contain pinning information. For more information, see RTblRelships Example Six.

- An auxiliary row can contain both pinning and sequencing information. For more information, see RTblRelships Example Seven.

Not every origin versioned relationship has a corresponding auxiliary row. If the origin versioned relationship is not part of a sequencing collection and does not have any member of its **TargetVersions** collection pinned, it does not have an auxiliary row.

See Also

Relationship Class

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

RTblRelships Example One

This example shows exactly one version-to-version relationship. This is the simplest case.

If **Z_OrgVS_Z** = **Z_OrgVE_Z**, the range of origin versions contains exactly one item. Similarly, if **Z_DstVS_Z** = **Z_DstVE_Z**, the range of destination versions contains exactly one item. If both these equalities hold, the row describes exactly one version-to-version relationship.

For example, if a row has the following values, it indicates that there is a version-to-version relationship between a version of the object whose internal identifier is 7 and a version of the object whose internal identifier is 888:

OrgID = 7
Z_OrgBrID_Z = 1
Z_OrgVS_Z = 3
Z_OrgVE_Z = 3
DstID = 888
Z_DstBrID_Z = 4
Z_DstVS_Z = 2
Z_DstVE_Z = 2
RelTypeID = 465
Z_RelFlags_Z = 2

Note that some column values are not shown here.

See Also

RTblRelships SQL Table

RTblRelships Example Two

This example shows how multiple versions of an origin object can be related to one version of a destination object. Multiple versions of an origin object are indicated by the inequality between the two endpoints of a range of origin object versions.

Within a relationship row, if **Z_OrgVS_Z** < **Z_OrgVE_Z**, the row describes more than one version-to-version relationship. For example, suppose that **Z_DstVS_Z** = **Z_DstVE_Z**, but that **Z_OrgVS_Z** < **Z_OrgVE_Z**. The set of destination versions referred to by this row includes exactly one item, but the set of origin versions referred to by this row includes *n* (*n* > 1) items. In this situation, this row of the table indicates the existence of *n* different version-to-version relationships.

For example, if a row contains the following values, it describes three version-to-version relationships, where versions 3, 4, and 5 of an origin object are related to one versioned destination object:

OrgID = 7
Z_OrgBrID_Z = 1
Z_OrgVS_Z = 3
Z_OrgVE_Z = 5
DstID = 888
Z_DstBrID_Z = 4
Z_DstVS_Z = 2
Z_DstVE_Z = 2
RelTypeID = 465
Z_RelFlags_Z = 2

See Also

RTblRelships SQL Table

RTblRelships Example Three

This example shows how the version graph defines an unbounded range for an origin object that is related to one versioned destination object.

Within a relationship row, if **Z_OrgVE_Z** = **VERINFINITY**, the row describes one or more version-to-version relationships, depending on the shape of the version graph.

For example, consider the following two rows and the accompanying version graph of the origin object. Compare the row data to the diagram at the end of this topic.

OrgID = 7
Z_OrgBrID_Z = 2
Z_OrgVS_Z = 5
Z_OrgVE_Z = VERINFINITY
DstID = 888
Z_DstBrID_Z = 4
Z_DstVS_Z = 2
Z_DstVE_Z = 2
RelTypeID = 465
Z_RelFlags_Z = 2

OrgID = 7
Z_OrgBrID_Z = 3
Z_OrgVS_Z = 4
Z_OrgVE_Z = VERINFINITY
DstID = 888
Z_DstBrID_Z = 4
Z_DstVS_Z = 2
Z_DstVE_Z = 2
RelTypeID = 465
Z_RelFlags_Z = 2

The first row describes exactly three version-to-version relationships, because within Branch 2, there are three object versions whose version-within-branch

identifiers are 5 or higher.

The second row describes exactly one version-to-version relationship, because within Branch 3, there is exactly one object version whose version-within-branch identifier is 4 or higher.

See Also

RTblRelships SQL Table

RTblRelships Example Four

This example shows a data set that can be consolidated to remove extraneous values.

In some situations, several relationship rows exist where one would theoretically suffice. For example, consider a row that includes these values:

OrgID = 7
Z_OrgBrID_Z = 4
Z_OrgVS_Z = 3
Z_OrgVE_Z = 8
DstID = 888
Z_DstBrID_Z = 4
Z_DstVS_Z = 2
Z_DstVE_Z = 2
RelTypeID = 465
Z_RelFlags_Z = 2

The row indicates that a version-to-version relationship (to a specific version of the destination object) exists from every Branch 4 version of the origin object whose version-within-branch identifier is between 3 and 8.

Compare the preceding row to the following rows. In particular, notice that the following two rows are effectively equivalent to the preceding row. Taken together, the following two rows indicate exactly what the preceding single row indicates. In other words, given two instances of the same origin-destination object pair, combining the lowest of the **Z_OrgVS_Z** values and the highest of the **Z_OrgVE_Z** values fully represents all version possibilities between this origin-destination object pair.

OrgID = 7
Z_OrgBrID_Z = 4
Z_OrgVS_Z = 3
Z_OrgVE_Z = 5
DstID = 888
Z_DstBrID_Z = 4

Z_DstVS_Z = 2
Z_DstVE_Z = 2
RelTypeID = 465
Z_RelFlags_Z = 2

OrgID = 7
Z_OrgBrID_Z = 4
Z_OrgVS_Z = 6
Z_OrgVE_Z = 8
DstID = 888
Z_DstBrID_Z = 4
Z_DstVS_Z = 2
Z_DstVE_Z = 2
RelTypeID = 465
Z_RelFlags_Z = 2

See Also

RTblRelships SQL Table

RTblRelships Example Five

This example shows sequence data in an auxiliary row. When **Z_RelFlags_Z** is equal to 1, the row is an auxiliary row. When equal to 2, it is a relationship row.

For each item in a sequenced origin versioned relationship, an auxiliary row in **RTblVersions** exists. For example, consider the following sequenced origin versioned relationship.

The figure shows a sequenced origin versioned relationship. Because the versioned relationship has two items, there are two sets of rows in the **RTblVersions** table. One set consists of one relationship row and one auxiliary row, with these values:

OrgID = 008
Z_OrgBrID_Z = 2
Z_OrgVS_Z = 3
Z_OrgVE_Z = 3
DstID = 984
Z_DstBrID_Z = 0
Z_DstVS_Z = 5
Z_DstVE_Z = 5
RelTypeID = 522
PrevDstID = NULL
Z_RelFlags_Z = 2

OrgID = 008
Z_OrgBrID_Z = 2
Z_OrgVS_Z = 3
Z_OrgVE_Z = 3
DstID = 984
Z_DstBrID_Z = 0
Z_DstVS_Z = 5
Z_DstVE_Z = 5
RelTypeID = 522

PrevDstID = SEQUENCE_END
Z_RelFlags_Z = 1

The other item has two version-to-version relationships. The **RTblVersions** table expresses this as a set of three rows with the following values:

OrgID = 008
Z_OrgBrID_Z = 2
Z_OrgVS_Z = 3
Z_OrgVE_Z = 3
DstID = 777
Z_DstBrID_Z = 2
Z_DstVS_Z = 2
Z_DstVE_Z = 2
RelTypeID = 522
PrevDstID = NULL
Z_RelFlags_Z = 2

OrgID = 008
Z_OrgBrID_Z = 2
Z_OrgVS_Z = 3
Z_OrgVE_Z = 3
DstID = 777
Z_DstBrID_Z = 2
Z_DstVS_Z = 3
Z_DstVE_Z = 3
RelTypeID = 522
PrevDstID = NULL
Z_RelFlags_Z = 2

OrgID = 008
Z_OrgBrID_Z = 2
Z_OrgVS_Z = 3
Z_OrgVE_Z = 3
DstID = 777
Z_DstBrID_Z = NULLBRANCH
Z_DstVS_Z = NULLVERSION
Z_DstVE_Z = NULL
RelTypeID = 522

PrevDstID = 984
Z_RelFlags_Z = 1

See Also

RTblRelships SQL Table

RTblRelships Example Six

This example shows pinning information.

If an origin versioned relationship has a pinned target version, the **RTblVersions** table includes an auxiliary row to indicate which target version is pinned. For example, the following figure shows an origin versioned relationship with a pinned target version.

To accommodate this origin versioned relationship, **RTblVersions** includes four rows with the following values, where the four rows correspond to the four arrows (counting the double arrow as two):

OrgID = 008
Z_OrgBrID_Z = 2
Z_OrgVS_Z = 3
Z_OrgVE_Z = 3
DstID = 984
Z_DstBrID_Z = 0
Z_DstVS_Z = 4
Z_DstVE_Z = 4
RelTypeID = 522
Z_RelFlags_Z = 2

OrgID = 008
Z_OrgBrID_Z = 2
Z_OrgVS_Z = 3
Z_OrgVE_Z = 3
DstID = 984
Z_DstBrID_Z = 0
Z_DstVS_Z = 5
Z_DstVE_Z = 5
RelTypeID = 522
Z_RelFlags_Z = 2

OrgID = 008

Z_OrgBrID_Z = 2
Z_OrgVS_Z = 3
Z_OrgVE_Z = 3
DstID = 984
Z_DstBrID_Z = 0
Z_DstVS_Z = 5
Z_DstVE_Z = NULL
RelTypeID = 522
Z_RelFlags_Z = 1

OrgID = 008
Z_OrgBrID_Z = 2
Z_OrgVS_Z = 3
Z_OrgVE_Z = 3
DstID = 984
Z_DstBrID_Z = 0
Z_DstVS_Z = 6
Z_DstVE_Z = 6
RelTypeID = 522
Z_RelFlags_Z = 2

See Also

RTblRelships SQL Table

RTblRelships Example Seven

This example shows a combination of sequencing and pinning information.

An auxiliary row can include both sequencing and pinning information. For example, if a sequenced origin collection includes an origin versioned relationship with a pinned target version, the **RTblVersions** table includes an auxiliary row that indicates which target version is pinned and which item in the sequenced origin collection precedes the current one. For example, the following figure shows a sequences origin collection, one of whose items has a pinned target version.

To accommodate this relationship collection, **RTblVersions** includes:

- Five relationship rows, one for each version-to-version relationship
- Two auxiliary rows, one for each origin versioned relationship.

The auxiliary rows have the following values:

OrgID = 008
Z_OrgBrID_Z = 2
Z_OrgVS_Z = 3
Z_OrgVE_Z = 3
DstID = 984
Z_DstBrID_Z = 0
Z_DstVS_Z = 5
Z_DstVE_Z = NULL
RelTypeID = 522
PrevDstID = SEQUENCE_END
Z_RelFlags_Z = 1

OrgID = 008
Z_OrgBrID_Z = 2
Z_OrgVS_Z = 3

Z_OrgVE_Z = 3
DstID = 777
Z_DstBrID_Z = NULLBRANCH
Z_DstVS_Z = NULLVERSION
Z_DstVE_Z = NULL
RelTypeID = 522
PrevDstID = 984
Z_RelFlags_Z = 1

See Also

RTblRelships SQL Table

Meta Data Services Programming

RTblScriptDefs SQL Table

RTbleScriptDefs stores one row for each script definition object stored in a repository database.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier of the class.
Z_BranchID_Z	**RTBrID**	Indicates the branch of the version graph that contains the range to whose items the property values in this row apply.
Z_VS_Z	**RTVerID**	A version-within-branch identifier that indicates the lower limit of the range to whose items the property values in this row apply.
Z_VE_Z	**RTVerID**	A version-within-branch identifier that indicates the upper limit of the range to whose items the property values in this row apply.
ScriptLanguage	**Varchar**, 255 bytes	The name of the scripting language to be used. The maximum length for this value is 255 bytes.
Body	**Text**, 64 kilobytes (KB)	A string that contains the script body. The maximum length for this value is 64 kilobytes.

Remarks

The **RTblScriptDefs** table stores instances of **ScriptDef** objects associated with **MethodDef** objects. Script instance data includes the script string and the language that supports it.

The primary key for this table is formed from the **IntID**, **Z_BranchID_Z**, and **Z_VS_Z** columns.

Z_BranchID_Z, **Z_VS_Z**, and **Z_VE_Z** are included to provide future support for versioning repository API definitions in case more than one version of this repository API structure is created.

See Also

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

ScriptDef Class

ScriptDef Object

Meta Data Services Programming

RTblSites SQL Table

RTblSites contains translation data that maps a global site identifier to a local site identifier. There is one row for each repository site known to this repository database.

Column name	Data type	Description
SiteID	**RTSiteID**	The site identifier for a repository site that is known to this repository database
SiteGUID	**RTGUID**	The global identifier for the site

Remarks

The **RTblSites** table provides a translation capability between the global site identifier that uniquely identifies a site across all repositories and the local site identifier, which is unique only within the current repository database. The smaller local site identifier is a part of the internal identifier that is used to identify a repository object within the repository database.

The primary key for this table is the **SiteID** column. A unique index is defined on the **SiteGUID** column.

See Also

Object Identifiers and Internal Identifiers

Repository Identifiers

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

Meta Data Services Programming

RTblSumInfo SQL Table

RTblSumInfo stores user-defined descriptive data about **ISummaryInformation** interfaces.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier of the class.
Z_BranchID_Z	**RTBrID**	Indicates the branch of the version graph that contains the range to whose items the property values in this row apply.
Z_VS_Z	**RTVerID**	A version-within-branch identifier that indicates the lower limit of the range to whose items the property values in this row apply.
Z_VE_Z	**RTVerID**	A version-within-branch identifier that indicates the upper limit of the range to whose items the property values in this row apply.
Comments	**RTLongString**	A field used for comments.
ShortDesc	**Varchar**, 255 bytes	The description of the object. The maximum length for this value is 255 bytes.
HelpContext	**Varchar**, 255 bytes	A context-sensitive Help string. The maximum length for this value is 255 bytes.
DescriptionContext	**RTLongString**	A context-sensitive description of the object.
OwnerInformation	**Varchar**, 255 bytes	The name of the current owner. The maximum length for this value is 255 bytes.
Status	**Varchar**, 255 bytes	The current status of the object. The maximum length for this value is 255 bytes.

Author	**Varchar**, 255 bytes	The name of the original author. The maximum length for this value is 255 bytes.
Caption	**Varchar**, 255 bytes	A caption that provides a more descriptive name. The maximum length for this value is 255 bytes.

Remarks

The **RTblSumInfo** table is an interface-specific table; its columns correspond to the properties exposed by the **ISummaryInformation** interface. The repository engine creates and populates this table when you invoke the **ISummaryInformation** interface and insert summary data.

The primary key for this table is formed from the **InitID**, **Z_BranchID_Z**, and **Z_VS_Z** columns.

See Also

InterfaceDef Class

ISummaryInformation Interface

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

RTblVersionAdminInfo SQL Table

Meta Data Services Programming

RTblTypeInfo SQL Table

RTblTypeInfo stores aliases of class, interface and relationship objects.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the member definition object.
Z_BranchID_Z	**RTBrID**	The branch identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VS_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VE_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always VERINFINITY.
Synonym	**Varchar**, 127 bytes	A string used as an alias name. The maximum length for this value is 127 bytes.

Remarks

The **RTblTypeInfo** table extends the repository API to allow classes, interfaces and relationships to be referred to by multiple names as aliases.

The primary key for this table is formed by the **IntID**, **Z_BranchID_Z**, and **Z_VS_Z** columns.

Z_BranchID_Z, **Z_VS_Z**, and **Z_VE_Z** are included to provide future support for versioning repository API definitions in case more than one version of this repository API structure is created.

See Also

CollectionDef Class

CollectionDef Object

MethodDef Class

MethodDef Object

PropertyDef Class

PropertyDef Object

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

Meta Data Services Programming

RTblTypeLibs SQL Table

RTblTypeLibs stores a global identifier for each information model object.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the information model (repository type library).
Z_BranchID_Z	**RTBrID**	The branch identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VS_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VE_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always VERINFINITY.
TypeLibID	**RTGUID**	The global identifier for the repository type library, as recorded in the system registry.
Prefix	**Varchar**, 255 bytes	A string containing the prefix that identifies an object with an information model. The first three characters of the information model provide a default prefix (for example, UML for the generic Unified Modeling Language (UML) model of the Open Information Model (OIM)). The maximum length for this value is 255 bytes.

Remarks

The **RTblTypeLibs** table relates the internal object identifiers of **ReposTypeLib** objects (information models) to their corresponding global identifiers.

The primary key for this table is formed by the **IntID**, **Z_BranchID_Z**, and **Z_VS_Z** columns.

Z_BranchID_Z, **Z_VS_Z**, and **Z_VE_Z** are included to provide future support for versioning repository API definitions in case more than one version of this repository API structure is created.

See Also

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

ReposTypeLib Class

ReposTypeLib Object

Meta Data Services Programming

RTblVersionAdminInfo SQL Table

RTblVersionAdminInfo stores version information for objects created through custom interfaces.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier of the class.
Z_BranchID_Z	**RTBrID**	Indicates the branch of the version graph that contains the range to whose items the property values in this row apply.
Z_VS_Z	**RTVerID**	A version-within-branch identifier that indicates the lower limit of the range to whose items the property values in this row apply.
Z_VE_Z	**RTVerID**	A version-within-branch identifier that indicates the upper limit of the range to whose items the property values in this row apply.
VersionCreateTime	**Datetime**, 8 bytes	The time the version was created. The maximum length for this value is 8 bytes.
VersionModifyTime	**Datetime**, 8 bytes	The time the version was modified. The maximum length for this value is 8 bytes.
CreateByUser	**RTLongString**	The user who created the version.
ModifyByUser	**RTLongString**	The user who modified the version.
VersionLabel	**Varchar**, 255 bytes	A string provided by an application to indicate a version label. The maximum length for this value is 255 bytes.
VersionShortDesc	**Varchar**, 255 bytes	A short summary of the version comments. The maximum length for this value is 255 bytes.

VersionComments	**Text**, 16 bytes	Comments added when a file is checked in a version control system. The maximum length for this value is 16 bytes.

Remarks

The **RTblVersionAdminInfo** table is an interface-specific table; its columns correspond to the properties exposed by the **IVersionAdminInfo** interface. By default, no class of the repository API implements **IVersionAdminInfo**. However, as soon as you insert any class that implements **IVersionAdminInfo**, the engine creates the table.

Each row of this table is either a version row or a merge row. Each version row describes an object version. Each merge row indicates that one or more merge operations occurred between the same pair of object versions. For more information, see RTblVersions SQL Table.

The primary key is formed from the **IntID**, **Z_BranchID_Z**, and **Z_VS_Z** columns.

See Also

InterfaceDef Class

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

RTblNamedObj SQL Table

RTblSumInfo SQL Table

Storage Strategy in a Repository Database

Versioning Objects

Meta Data Services Programming

RTblVersions SQL Table

RTblVersions stores version information about repository objects.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the object.
Z_BranchID_Z	**RTBrID**	A branch identifier. It indicates the branch of the version graph that contains this object version.
Z_VS_Z	**RTVerID**	A version-within-branch identifier. It differentiates the version described by this row from other versions on the same branch.
Z_PredBr_Z	**RTBrID**	A branch identifier. For a version row, this column indicates the branch containing the predecessor creation version of the current object version. For a merge row, this column indicates the branch containing the object version that was the predecessor of the merge operation.
Z_PredVer_Z	**RTVerID**	A version-within-branch identifier. For a version row, this column indicates the predecessor creation version of the current object version. For a merge row, this column indicates the object version that was the predecessor of the merge operation.
TypeID	**RTIntID**	The internal identifier of the class to which the version conforms.
VerIntID	**RTIntID**	The internal identifier of this version.
Z_VState_Z	**RTFlags**	Indicates whether the object version is frozen and whether it is checked out. The object version is frozen only if the last (least significant) bit is set. The

		object version is checked out only if the second-last bit is set.
Z_PredFlags_Z	**RTFlags**	Indicates whether this row is a version row or a merge row. A value of 1 indicates that this is a version row. A value of 2 indicates that this is a merge row.
Z_SuccInc_Z	**RTSuccInc**	For internal use only.
Z_LClock_Z	**RTLClock**	For internal use only.

Remarks

The **RTblVersions** table stores two row types: version rows and merge rows. Each row is either one type or the other. Version rows describe an object version. Merge rows indicate that one or more merge operations occurred between the same pair or object versions.

Because a repository object can have many versions, it can have multiple version rows to store information about each version. For each object, there is exactly one version row that describes the initial version of the object. Within the initial version row, the values of **Z_PredBrID_Z** and **Z_PredVer_Z** are special constants indicating that the first version has no predecessor.

Because an object version can have zero, one, or many non-creation predecessors, each object version can have zero, one, or many merge rows.

The primary key is formed from the **IntID**, **Z_BranchID_Z**, **Z_VS_Z**, **Z_PredBrID_Z**, and **Z_PredVer_Z** columns.

See Also

InterfaceDef Class

InterfaceDef Object

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

Meta Data Services Programming

RTblWorkspaceItems SQL Table

RTblWorkspaceItems stores information about which repository object versions are included within a workspace.

Column name	Data type	Description
IntID	**RTIntID**	The internal identifier for the workspace object.
Z_BranchID_Z	**RTBrID**	The branch identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VS_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always zero.
Z_VE_Z	**RTVerID**	An identifier for repository API versioning. It is reserved for proprietary use by the repository engine. The value of this column is always VERINFINITY.
Z_ItemIntID_Z	**RTIntID**	The internal identifier of the object version present in the workspace.
Z_ItemBranchID_Z	**RTBrID**	The branch identifier of the object version present in the workspace.
Z_ItemVS_Z	**RTVerID**	The version-within-branch identifier of the object version present in the workspace.
Z_ItemFlag_Z	**RTFlags**	A set of flags indicating whether the object version is checked out to the workspace. A value of 0 indicates that the object version is not checked out. A value of 2 indicates that the object

		version is checked out.

Remarks

The **RTblWorkspaceItems** table tracks which object version is part of a workspace. A workspace can have only a single version of any given object.

The primary key is formed from the **IntID**, **Z_BranchID_Z**, **Z_VS_Z**, **Z_ItemIntID_Z**, **Z_ItemBranchID_Z**, and **Z_ItemVS_Z** columns.

See Also

Repository SQL Data Types

Repository SQL Schema

Repository SQL Tables

Meta Data Services Programming

XML Encoding Reference

This section describes the format for exchanging instances of the Open Information Model (OIM) through the use of Extensible Markup Language (XML). The XML encoding format works for any information model that is based on the Meta Data Coalition (MDC) OIM framework. A set of rules governs the encoding of meta data objects by OIM in XML. The XML encoding of OIM types enables the interchange of meta data between heterogeneous repositories. The encoding format defined in this specification is completely driven by the abstract model. The names of the element and attribute tags used in the representation are derived from the model. Documents can be generated and parsed automatically by any implementation of OIM, regardless of technology.

XML DTD

Accompanying this section is a set of XML Document Type Definitions (DTDs) that together form a grammar to express the structure of XML instances. DTDs are currently the only approved mechanism for describing the structure of XML documents. In its current form, DTDs are not expressive enough to cover the semantics of OIM completely. A correct interpretation of an XML document is only possible based on the OIM specification. However, DTDs have been supplied to make understanding the XML documents easier and to help with the development of XML import/export functionality based on this encoding format.

The following topics are discussed in this section.

Topic	Description
XML Encoding Definition	Explains the XML encoding format rules for OIM
OIM-to-XML Mapping	Shows the mapping of the core concepts, which include diagrams and example code
Sample Encoding	Provides an example code of an XML OIM transfer
EBNF Representation	Shows an example of an OIM XML encoding in Extended Backus-Naur Form (EBNF)

Namespaces in OIM	Shows a table of unique namespaces of each OIM information model
DTD for the OIM Namespace	Shows an example of OIM Namespace Definition
XML Import Export	Describes the import and export interfaces for XML in the OIM
XML Encoding Errors	Lists the XML encoding error messages

It is assumed that the reader is familiar with the concepts represented by the OIM. A basic knowledge of XML, COM, and Microsoft® SQL Server™ 2000 Meta Data Services is also assumed throughout this section. This section is based on XML standards as defined by the World Wide Web Consortium (W3C), and XML Namespaces provide a simple method for qualifying names in XML documents. The implementation of namespaces in this section is based on the W3C recommendation Namespaces in XML.

For more information about COM, XML, and OIM, see the MSDN® Library at the Microsoft Web site and the Meta Data Coalition Web site at http://www.mdcinfo.com.

For more information about XML standards, see the W3C Web site http://www.w3.org/.

See Also

Using XML Encoding

XML in Meta Data Services

Meta Data Services Programming

XML Encoding Definition

Extensible Markup Language (XML) encodes information as content enclosed in nested begin/end tag elements and name/value pair attributes on these elements. The XML encoding format defined in this section is based on this encoding rule.

XML provides the following basic concepts to encode information.

Topic	Description
Character Set and Data Types	Describes the character set and data types for encoding used in an XML document
Top-Level Element	Describes the element that encapsulates all transfer information in an XML document
Elements and Attributes	Describes the begin/end tag pairs and the content encapsulated between them
Namespaces	Shows the basic structure of the Open Information Model (OIM) namespace hierarchy to ensure unique elements in an XML document
Nested Lists	Shows, by example, the ordered or unordered sets of elements that can be used to represent hierarchies of elements
Element References	Describes connections between elements to represent network structures of elements
Extensibility	Describes extended vendor-specific meta data types

See Also

Using XML Encoding

XML Encoding Errors

XML Encoding Reference

XML in Meta Data Services

XML Import Export

Meta Data Services Programming

Character Set and Data Types

The Extensible Markup Language (XML) encoding relies on the native XML character set handling based on Unicode.

Values appear as attribute-tagged values. They are represented using the following rules.

Data type	Encoding
String	Any occurrence of & must be replaced by & Any occurrence of < must be replaced by < Any occurrence of > must be replaced by > Any occurrence of " (double quote) must be replaced by "
Date	Must follow the ISO 8601 format.
Numbers	Punctuation must use US English rules (for example, they must use a period as a decimal separator). Numbers can include exponents.
Boolean	False = -1, True = 1.
BLOB	Use MIME Base64 encoding.

See Also

XML Encoding Reference

Meta Data Services Programming

Top-Level Element

The Extensible Markup Language (XML) requires a top-level element (begin/end tag) that encapsulates all information contained in an XML document. Any Document Type Definitions (DTDs) defined or referenced in the document apply to the content of the top-level element.

The Open Information Model (OIM) to XML mapping defines a **transfer** element as the top-level structure. This element encapsulates all structured information that is described in the XML document. Additional features of the transfer element can be nested. The top-level element also maintains administrative information, such as what exporter generated the data and version.

Example

```
<?xml version="1.0"?>
<oim:Transfer version="1.0"
    xmlns:oim="http://www.mdcinfo.com/oim/oim.dtd">
    <oim:TransferHeader
        Exporter="MSMDCXML"
        ExporterVersion="2.0"
        TransferDateTime="19980804T08:15:00"
    >
        . . . user-defined information . . .
    </oim:TransferHeader>
    . . . objects . . .
</oim:Transfer>
```

All structures defined within the remainder of this section are valid only within the begin (<**oim:Transfer**>) and end (</**oim:Transfer**>) tags of the transfer element. The **TransferHeader** element is used to contain information about the component that generated the transfer.

See Also

XML Encoding Reference

Meta Data Services Programming

Elements and Attributes

Elements in Extensible Markup Language (XML) are enclosed within a pair of tags. Each pair includes an opening tag and a closing tag. The content can be either a structure of sub-elements or an unstructured data representation. The following table shows how the meta data element types, which are used to describe the Open Information Model (OIM), are mapped into XML.

Element	XML representation
Information Model	No mapping. For more information, see Namespaces.
Concrete Class	<**class_name**>...</**class_name**>
Attribute	Included as an XML attribute on an XML element, for example, **attribute=value**.
Association	<**association_name**>...</**association_name**>

The tag identifies the type of an element. Additional meta information about the element can be represented by predefined attributes. The following table lists attributes that are currently predefined.

Attribute name	Defined for	Mandatory/optional	Description
OIM:id	Object, association	Optional	Transfer identifier (ID) used to uniquely identify an element in an XML document. The **id** has no meaning outside of a transfer. The **id** is mandatory on objects, but optional on object references.
OIM:objid	Object, association	Optional	Unique identifier of an element in the source or target repository.

OIM:href	Objects	Optional	Hyperlink mechanism to reference objects.
OIM:label	Objects	Optional	The name of an object within the encapsulating association.
OIM:supertype	Object	Optional	Used by extensions to indicate the OIM type that can be used for importing an object.

The OIM-to-XML mapping separates the transfer ID and object ID and treats the object ID as an attribute of the element. This XML encoding is designed to enable the interchange of objects between heterogeneous repositories. There is no common format for object identifiers; furthermore, there is no agreement on how to implement object identity (name based, GUID, disk pointer, and so on).

To provide a generic solution, a uniquely defined ID identifies an object within a transfer; that is, an ID can serve as the target of a reference in the transfer. The structure of the ID is unspecified, but it must be unique in a transfer and it must contain an underscore as the first character. Examples of valid transfer IDs are a running number ("_007") or the name of an object ("_Invoice007").

Note that object identity is necessary to allow a meaningful synchronization of objects between repositories. In a heterogeneous environment, this requires the XML encoding to maintain a cross-reference between the globally unique identifiers (GUIDs) of objects maintained by different repository products. To exchange object IDs as attributes of objects, exchanging repositories must agree on the semantics of the exchange mechanism. To simplify this process, the attribute **objid** is included in the encoding format. If necessary, the first source of a transfer can generate the object ID. Each successive transfer step must maintain the whole object ID and pass it on.

See Also

XML Encoding Reference

Meta Data Services Programming

Namespaces

In the Open Information Model (OIM), classes and associations share the same namespace for a single information model. This means that more than one class and/or association within the same information model cannot share the same name. The following table shows the basic structure of the OIM namespace hierarchy.

Level 1	Level 2	Description
Information Model		Corresponds to an information model
	Class	Class name and the associated attributes
	Association	A collection of nested or linked classes

The OIM-to-Extensible Makeup Language (XML) mapping combines XML Namespaces and a naming convention to provide the following solution.

<	Namespace	:	Name	>
<	Information Model Prefix	:	Class Name	>
<	Information Model Prefix	:	Association Name	>

For example, <x:y> is an element tag for an object of class y in submodel x. Note that because attributes are represented as XML attributes they are scoped as part of the element. Therefore, the attribute names need to be unique only within the class, not the whole subject area of the model.

If attributes in the inheritance chain of the class share the same name, the names of the attributes are expanded to **ClassName.AttributeName**. If the class name is not unique, it is prefixed with the same information model prefix as the namespace.

See Also

XML Encoding Reference

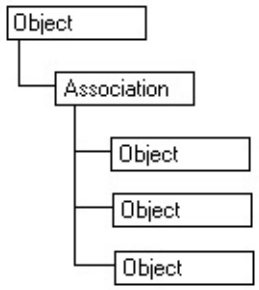
Namespaces in OIM

Meta Data Services Programming

Nested Lists

Extensible Markup Language (XML) represents information as nested lists of elements or references to elements. Lists can be either ordered or unordered; the occurrence of element types is optional or mandatory.

The following diagram shows the representation of the Open Information Model (OIM) "*class has associations*" and "*associations contain objects*" in XML.



Example

```
<object attribute="_">
    <association>
        <object label="C" name="Lisa" seqno="1">
        ...
        </object>
        <object label="A" name="John" seqno="2">
        ...
        </object>
        <object label="B" name="Tom" seqno="3">
        ...
        </object>
    </association>
    ...
</object>
```

Object elements contain lists of association elements, which, in turn, contain lists of object elements.

See Also

XML Encoding Reference

Meta Data Services Programming

Element References

Nested lists of Extensible Markup Language (XML) elements enable the representation of hierarchical structures of objects. References are used to link objects into a general network of associations. The XML hyperlink mechanism is used to reference objects defined internal to a transfer. An internal object is simply referenced by its transfer identifier (ID).

An object reference is represented by the *href* attribute of the element tag:

```
<object_type_name href="#_123"/ >
```

The object reference indicates the type of the object referred to. To learn the object type, you do not need to navigate the object reference to the target object.

See Also

XML Encoding Reference

Meta Data Services Programming

Extensibility

The Open Information Model (OIM) can be extended with vendor-specific meta data types. New classes, attributes, and associations are added using the Universal Modeling Language (UML) representation of the OIM. New elements may be either created from scratch or based on existing OIM types using specialization (inheritance). A vendor may choose to publish the model extensions in order to share the meta data with other vendors, or treat the extension as tool specific (private).

Using the OIM to Extensible Markup Language (XML) mapping rules described in this document, an XML Document Type Definition (DTD) for the model extension can be created from its UML representation. However, the XML DTD does not provide enough information for other vendors to interpret the model. This is a limitation of the current XML standard with DTD as schema description language. DTDs do not capture type inheritance and other sophisticated modeling structures. The World Wide Web Consortium (W3C), as XML standard body, is standardizing a new schema definition language called XML Schema.

Until the XML Schema specification is available, the OIM XML encoding format will support the use of the optional *supertype* attribute. This attribute is used to define which OIM type a new meta data type specializes. In the case where multiple OIM types are specialized, it is the responsibility of the exporting tool to choose one of the types.

The following example shows an instance of a new meta data type that extends the table class in the Database Schema Model.

Example

<Ext:MyTable supertype="DBM:Table" name="xxx" size="yyy" myV

An importer can decode the element structure even if the new subtype is unknown. It simply uses the schema of the known OIM type specified by the *supertype* attribute. Note that the attribute must contain a fully qualified class name (including namespace). It is also necessary to resolve attribute name

conflicts using the rules described in the following sections.

See Also

XML Encoding Reference

Meta Data Services Programming

OIM-to-XML Mapping

This section provides a set of basic diagrams that show the mapping of the core concepts Class, Attribute, and Association, as well as class inheritance from Open Information Model (OIM) into Extensible Markup Language (XML). The Universal Modeling Language (UML) diagram that represents the OIM concepts is provided with the XML encoding.

The following topics include diagrams with examples.

Topic	Description
Classes and Attributes	Shows how the attributes of an OIM class are mapped into XML
Attribute Name Expansion	Shows how attributes that are typed as classes are mapped into XML
Classes and Single Inheritance	Shows how attributes and inherited attributes of a class are mapped into XML
Classes and Multiple Inheritance	Shows a class that inherits attributes from multiple other classes
Associations with XML	Shows how associations are encoded in XML
Object References with XML	Shows an association structure in which the destination object has already been defined
Association Classes (Many-to-Many)	Shows how a many-to-many association class is represented in XML
Association Classes (One-to-Many or One-to-One)	Shows how a one-to-many or one-to-one association class is represented in XML

See Also

XML Encoding Errors

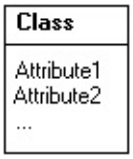
XML Encoding Reference

XML Import Export

Meta Data Services Programming

Classes and Attributes

The following example shows how the attributes of an Open Information Model (OIM) class are mapped into Extensible Markup Language (XML).



Example

```
<Class Attribute1="..." Attribute2="...">
    ...
</Class>
```

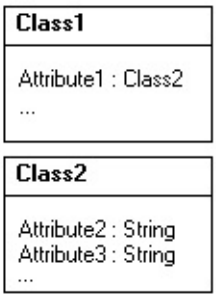
See Also

XML Encoding Reference

Meta Data Services Programming

Attribute Name Expansion

The following example shows how attributes that are typed as classes are mapped into Extensible Markup Language (XML).



Example

```
<Class1 Attribute1Attribute2="..."
    Attribute1Attribute3="...">
    ...
</Class1>
```

In general, given a class *A* with an attribute *B* of type *C*, for each attribute *D(n)* on *C*, create a new attribute on *A* called *BD(n)*. The name of the new attribute on *A* is appended with the name of *D(n)* unless the **ExpandName** tagged value on the attribute definition is set to false.

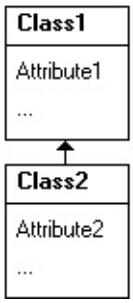
See Also

XML Encoding Reference

Meta Data Services Programming

Classes and Single Inheritance

The following example shows how attributes and inherited attributes of a class are mapped into Extensible Markup Language (XML).



Example

```
<Class2 Attribute1="..." Attribute2="...">
    ...
</Class2>
```

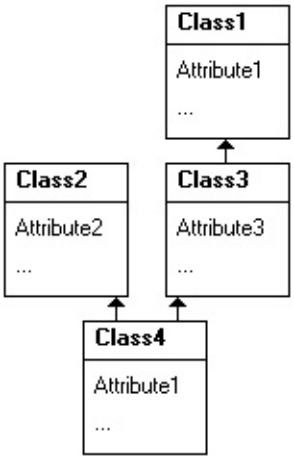
See Also

XML Encoding Reference

Meta Data Services Programming

Classes and Multiple Inheritance

A class can inherit attributes from multiple other classes. The following example shows how such a class is represented.



Example

```
<Class4 oim:id = "_123"
    Class4.Attribute1="..."
    Class1.Attribute1="..."
    Attribute2="..."
    Attribute3="...">
    ...
</Class4>
```

Note Because there is a naming conflict between the two attributes called Attribute1, the name of the class they are defined on is added as a prefix.

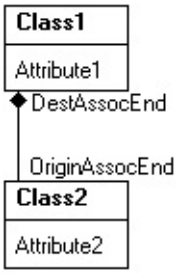
See Also

XML Encoding Reference

Meta Data Services Programming

Associations with XML

The following example shows how Open Information Model (OIM) associations are encoded in Extensible Markup Language (XML).



Example

```
<Class1 oim:id="_1" attrib1="...">
    <Class1OriginAssocEnd> <!-- assoc starts -->
        <Class2 oim:id="_2" oim:seqno="1" label="A"
                name="Alpha" Attribute2="..."/>
        <Class2 oim:id="_3" oim:seqno="2" oim:label="B"
                name="Beta" Attribute2="..."/>
    </Class1OriginAssocEnd> <!-- assoc ends -->
</Class1>
```

If an association name is not specified, the name is generated using the following rule:

OriginClassName + **OriginAssociationEndName**

Given this rule, the association in the preceding example is named <Class1OriginAssocEnd>.

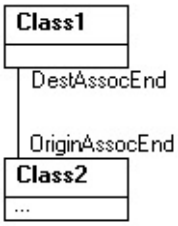
See Also

XML Encoding Reference

Meta Data Services Programming

Object References with XML

The following example shows an association structure in which the destination object has already been defined and therefore needs to be referenced.



Example

```
<Class2 oim:id="_2">
    ...
</Class2>
<Class1 oim:id="_1">
    <Class1OriginAssocEnd>
        <Class2 oim:href="#_2"/>
        ...
    </Class1OriginAssocEnd>
</Class1>
```

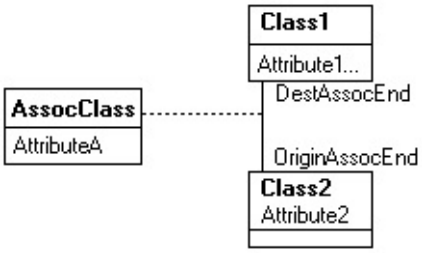
See Also

XML Encoding Reference

Meta Data Services Programming

Association Classes (Many-to-Many)

The following example shows how a many-to-many association class is represented in Extensible Markup Language (XML).



Example

```
<Class1 oim:id="_2">
    <Class1OriginAssocEnd>
        <AssocClass AttributeA="...">
            <AssocClassDestAssocEnd>
                <Class2 oim:id="_3"/>
            </AssocClassDestAssocEnd>
        </AssocClass />
    </Class1OriginAssocEnd>
</Class1>
```

This example encodes the association element into a junction class between the other two classes and uses the association name generation rule to establish the two association names.

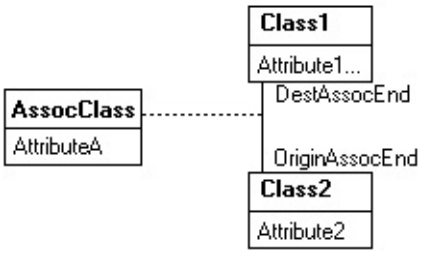
See Also

XML Encoding Reference

Meta Data Services Programming

Association Classes (One-to-Many or One-to-One)

The following example shows how a one-to-many or one-to-one association class is represented in Extensible Markup Language (XML).



Example

```
<Class1 oim:id="_2" attribute1="..."
    <Class1OriginAssocEnd>
        <Class2 attributeA="..."
                Attribute2=oim:id="_3"

    </Class1OriginAssocEnd>
</Class1>
```

This mapping represents all the attributes of the association class as attributes on the destination class.

See Also

XML Encoding Reference

Meta Data Services Programming

Sample Encoding

```
<?xml version="1.0" ?>
    <oim:Transfer xmlns:oim="http://www.mdcinfo.com/oim/oim.dtd"
      <dbm:Catalog oim:id="_1" name="sales" comments="Sample ca
        <dbm:CatalogSchemas>
          <dbm:Schema oim:id="_2" name="dbo">
          <dbm:SchemaTables>
            <dbm:Table oim:id="_3" name="Customer">
              <dbm:ColumnSetColumns>
                <dbm:Column oim:id="_6" name="CustomerID" IsNull
                <dbm:Column oim:id="_7" name="Name" IsNullable="
                <dbm:Column oim:id="_8" name="Address" IsNullable
                <dbm:Column oim:id="_9" name="Phone" IsNullable="
              </dbm:ColumnSetColumns>
            </dbm:Table>
            <dbm:Table oim:id="_4" name="Order" EstimatedRows="1
              <dbm:ColumnSetColumns>
                <dbm:Column oim:id="_10" name="CustomerID" IsNu
                <dbm:Column oim:id="_11" name="OrderID" IsNullab
                <dbm:Column oim:id="_12" name="Date" IsNullable="
              </dbm:ColumnSetColumns>
            </dbm:Table>
            <dbm:Table oim:id="_5" name="OrderItem" EstimatedRow
              <dbm:ColumnSetColumns>
                <dbm:Column oim:id="_13" name="CustomerID" IsNu
                <dbm:Column oim:id="_14" name="OrderID" IsNullab
                <dbm:Column oim:id="_15" name="LineNo" IsNullabl
                <dbm:Column oim:id="_16" name="Description" IsNul
                <dbm:Column oim:id="_17" name="Quantity" IsNullab
                <dbm:Column oim:id="_18" name="UnitPrice" IsNulla
              </dbm:ColumnSetColumns>
```

```
            <dbm:TableUniqueKeys>
              <dbm:UniqueKey oim:id="_19" name="PK_OrderItem'
                <dbm:KeyColumns>
                  <dbm:Column oim:href="#_14" />
                  <dbm:Column oim:href="#_15" />
                </dbm:KeyColumns>
              </dbm:UniqueKey>
            </dbm:TableUniqueKeys>
          </dbm:Table>
        </dbm:SchemaTables>
      </dbm:Schema>
    </dbm:CatalogSchemas>
  </dbm:Catalog>
</oim:Transfer>
```

See Also

Using XML Encoding

XML Encoding Reference

XML in Meta Data Services

Meta Data Services Programming

EBNF Representation

The following defines the grammar of the Open Information Model (OIM) Extensible Markup Language (XML) encoding in Extended Backus-Naur Form (EBNF).

xmlHdr::='<?xml version="1.0">'
oimDoc::=xmlHdr S Transfer
Transfer::='<oim:Transfer' [S 'version="1.0"'] NameSpaceDecl '>' S
  [TransferHeader]
  (Object | Transfer)*
  '</oim:Transfer>'
TransferHeader::='<oim:TransferHeader'
  ['Exporter="' *ExporterName* '"']
  ['ExporterVersion="' *ExporterVersion '"']*
  ['TransferDateTime="' *CurrentDate* '"']
  '/>'
oimNameSpace::='xmlns:oim="http://www.mdcinfo.com/oim/oim.dtd'
oimPrefix::='oim:'
NameSpaceDecl::=oimNameSpace (S ModelSpaceDecl)*
ModelSpaceDecl::= 'xmlns:' modelAbbr '="http://www.mdcinfo.com/o
nsPrefix::=modelAbbr *(for the information model that the class is defi*
objTransID::= '_' uniquifier (*where uniquifier is a running number*)
objID::= unique identifier for the element *(repository dependent)*
seqno::= sequence number within an association
label::= name of object within the association
object::='<' nsPrefix ':' elementName S
'oim:id="' objTransID '"'
  [S 'oim:objid="' objID '"']
  [S 'oim:seqno="' seqno '"']
  [S 'oim:label="' label '"']
  [(S Attribute)*]
  '>'

```
  [(S Association)*]
  '</' nsPrefix ':' elementName '>'
Attribute::=[[nsPrefix ':'] ClassName '.'] AttributeName S? '="' S?
  attributeValue S? '"'
Association   ::=   '<' [nsPrefix ':'] AssociationName '>' S?
  (Object S)*
  </' [nsPrefix ':'] AssociationName '>'
```

See Also

Using XML Encoding

XML Encoding Reference

XML in Meta Data Services

Meta Data Services Programming

Namespaces in OIM

Using Extensible Markup Language (XML) Namespaces, each information model of the Open Information Model (OIM) encoding defines a separate namespace for its XML tags; that is, an individual Document Type Definition (DTD) describes each information model. Information models depend on each other and form a well-defined (acyclic) dependency graph. Meta Data Coalition (MDC) OIM has the following information models.

OIM groupings by subject areas	OIM information models	Namespace identifier
Analysis and Design Model	Unified Modeling Language	**uml**
	UML Extensions	**umx**
	Common Data Types	**dtm**
	Generic Elements	**gen**
Object and Component Description Model	Component Descriptions	**cde**
Database and Data Warehousing	Database Schema Elements	**dbm**
	Data Transformation Elements	**tfm**
	OLAP Schema Elements	**olp**
	Record Oriented Legacy	**rec**
Knowledge Management Model	Semantic Definition Elements	**sim**

The XML namespaces respect the extensibility mechanism of OIM. Users are able to add information models with new elements without causing name conflicts with existing information models or future extensions.

See Also

Using XML Encoding

XML Encoding Reference

XML in Meta Data Services

Meta Data Services Programming

DTD for the OIM Namespace

This is a sample Document Type Definition (DTD) for the Open Information Model (OIM) namespace used by the encoding.

```
<!-- ____________________________________________________________ -->
<!--                    XML Encoding                              -->
<!--              for the Open Information Model                  -->
<!-- ____________________________________________________________ -->

<!-- ____________________________________________________________ -->
<!-- Transfer                                                     -->
<!-- ____________________________________________________________ -->
<!-- A transfer is a unit of exchange in OIM. Transfers might be  -->
<!-- nested.                                                      -->
<!ELEMENT Transfer ( TransferHeader?, ( ANY | Transfer )* ) >
<!ATTLIST Transfer
    version CDATA #FIXED "1.0"
>
<!-- ____________________________________________________________ -->
<!-- TransferHeader                                               -->
<!-- ____________________________________________________________ -->
<!-- A transfer header is used to specify all necessary information  -->
<!-- to define the origin of a transfer in a structured way.      -->
<!-- Exporter         Name of software that generated the transfer -->
<!-- ExporterVersion  Version of software that generated transfer  -->
<!-- TransferDateTime  Date and time that the transfer was created -->
<!ELEMENT TransferHeader (ANY)>
<!ATTLIST TransferHeader
    Exporter CDATA #IMPLIED
    ExporterVersion CDATA #IMPLIED
    TransferDateTime CDATA #IMPLIED
>
```

```
<!-- ____________________________________________________________ -->
<!-- Classes                                                      -->
<!-- ____________________________________________________________ -->
<!-- Classes are output as XML elements. They should all have id,  -->
<!-- objid, href and sequence number as predefined XML attributes. -->
<!-- Unfortunately the DTD grammar does not specify this.         -->
<!-- so these attributes are shown here as an example. The oim:   -->
<!-- namespace qualifier for the predefined attribute is only     -->
<!-- included when one of the predefined attribute has a naming   -->
<!-- conflict with the attributes on the class                    -->
<!-- <!ATTLIST typename                                           -->
<!--                 [oim:]id ID #REQUIRED                        -->
<!--                 [oim:]objid CDATA #IMPLIED                   -->
<!--                 [oim:]href CDATA #IMPLIED                    -->
<!--                 [oim:]seqno CDATA #IMPLIED                   -->
<!--                 [oim:]label CDATA #IMPLIED                   -->
<!--                 [oim:]supertype CDATA #IMPLIED               -->
<!--                                                              -->
<!-- End of DTD ___________________________________________________ -->
```

See Also

Using XML Encoding

XML Encoding Reference

XML in Meta Data Services

Meta Data Services Programming

XML Import Export

This section describes the methods used for importing, exporting, and transferring data from one Microsoft® SQL Server™ 2000 Meta Data Services repository to another.

XML Exporter is a utility that exports objects from a Microsoft repository by using Open Information Model (OIM) XML Encoding. The export is handled by a COM component that has the **MSMDCXML.IExport** program identifier (ID). The component publishes one interface, **IExport**. Through this interface, the client can specify which repository objects to export and initiate the export process. Because it is a dual interface, it can be used by both COM and Automation.

Topic	Description
XML IExport Interface Overview	Describes the **IExport** interface and shows the Interface Definition Language (IDL) definition
IExport::_NewEnum Method	Explains the **NewEnum** method of the **IExport** interface and provides Automation syntax
IExport::Add Method	Explains the **Add** method of the **IExport** interface and provides Automation syntax
IExport::Clear Method	Explains the **Clear** method of the **IExport** interface and provides Automation syntax
IExport::Count Property	Explains the **Count** property of the **IExport** interface and provides Automation syntax
IExport::Export Method	Explains the **Export** method of the **IExport** interface and provides Automation syntax
IExport::GetXML Method	Explains the **GetXML** method of the **IExport** interface and provides Automation syntax
IExport::Item Method	Explains the **Item** method of the **IExport**

	interface and provides Automation syntax
IExport::Remove Method	Explains the **Remove** method of the **IExport** interface and provides Automation syntax

The import process uses an Extensible Markup Language (XML) document to create OIM instances in a Meta Data Services repository. The OIM describes the structure as well as the semantics of the transferred elements. The COM component is used for XML importing **MSMDCXML.IImport** program ID. The component publishes one interface, **IImport**. Through this interface, the client can specify which objects to import and initiate the import process. Because it is a dual interface, it can be used by both COM and Automation.

Topic	Description
XML IImport Interface Overview	Describes the **IExport** interface and IDL definition
IImport::ImportXML Method	Explains the **ImportXML** method of the **IImport** interface and provides Automation syntax
IImport::ImportXMLString Method	Explains the **ImportXMLString** method of the **IImport** interface and provides Automation syntax

See Also

Using XML Encoding

XML Encoding Errors

XML Encoding Reference

Meta Data Services Programming

XML IExport Interface Overview

Using Extensible Markup Language (XML) to export objects from Microsoft® SQL Server™ 2000 Meta Data Services is a two-step process:

1. Mark objects for export.

2. Generate the XML file.

Marking Objects for Export

The client marks objects for export by using the **IExport::Add** method to create an object list. Using this method, the client passes a handle of the repository object to be exported.

The export process creates a collection of all objects that have been added to the export list. The order in which objects are added determines the order in which the objects will appear in the XML document.

After the collection has been created, a client can enumerate through this collection and get information, such as the number of objects, as in a normal collection.

Generating the XML File

The client starts the export by invoking the **IExport::Export** method. The client passes the name of the file into which the XML document should be exported as a parameter of this method. XML Exporter will overwrite the file if it already exists. The client can specify flags that control the way objects are handled in the output. The allowed flags can be combined using a bitwise logical OR operation. For more information about the effect of each flag, see IExport::Export Method.

IDL Definition

The following expandable text is the part of the Interface Definition Language (IDL) file that describes the methods and the properties on the **IExport** Interface. In Automation, properties and methods are attached to the **IExport**

object.

IDL Segment

```
interface IExport : IDispatch
{
   [id(0), helpstring("method Item")] HRESULT Item([in] VARIANT In
   [id(1), helpstring("method Export")] HRESULT Export([in] BSTR X
   [id(2), helpstring("method GetXML")] HRESULT GetXML([in,opti
   [id(3), helpstring("method Add")] HRESULT Add([in] IRepositoryO
   [id(-4), helpstring("method _NewEnum")] HRESULT _NewEnum([
   [propget, id(6), helpstring("property Count")] HRESULT Count([out
   [id(7), helpstring("method Remove")] HRESULT Remove([in] VARI
   [id(8), helpstring("method Clear")] HRESULT Clear();
};
```

See Also

XML Import Export

Meta Data Services Programming

IExport::_NewEnum Method

This method is used to obtain an enumerator property that can be used to enumerate through the list of the exported objects.

COM Syntax

HRESULT _NewEnum(
    IUnknown ***ppVal*
);

Parameters

ppVal [out, retval]

A pointer that points to a location that stores the enumerator of objects in the export list.

Return Value

S_OK indicates successful completion.

An error value indicates that the method failed to complete successfully. For more information, see XML Encoding Errors.

Automation Syntax

The **_NewEnum** property is used by Automation-based programming languages to enumerate through collections. It is never used directly; instead, enumeration constructs use it internally. In Microsoft® Visual Basic®, this enables the following example:

For each item in the collection.

...

Next item

In the example, Collection is an object that contains the **_NewEnum** property.

See Also

XML Import Export

Meta Data Services Programming

IExport::Add Method

This method allows the client to add an object to the list of objects to be exported.

COM Syntax

HRESULT Add(
    IRepositoryObject **pIRO*
    Long *Flags*
);

Parameters

pIRO [in]

A pointer to the repository object to be added to the exported objects list.

Flags [in]

The following table describes the flag.

Enumerator	Value	Description
ADDCONTAINING_BASE	1	Only objects that are contained in base collections of the current object are added.
ADDCONTAINING_MOSTDERIVED	2	Only objects that are in the most derived collections of the current object are added.

Return Value

S_OK indicates successful completion.

An error value indicates that the method failed to complete successfully. For

more information, see XML Encoding Errors.

Automation Syntax

In Automation, this method has the following syntax:

object.**Add** *pIRO [,Flags]*

The **Add** method syntax has the following parts.

Parameter	Description
object	An object declared as **MSMDCXML.Export**
pIRO	An object expression that evaluates to **RepositoryObject**
Flags	ADDCONTAINING

See Also

Member Delegation

XML Import Export

Meta Data Services Programming

IExport::Clear Method

This method removes all objects from the export collection.

COM Syntax

HRESULT Clear(
);

Return Value

S_OK indicates successful completion.

An error value indicates that the method failed to complete successfully. For more information, see XML Encoding Errors.

Automation Syntax

In Automation, this method has the following syntax:

Call Object.Clear

See Also

XML Import Export

Meta Data Services Programming

IExport::Count Property

This method is used to obtain the number of objects that have been added to the export list. In COM, it is called as a method that returns a property. In Automation, it is used as the read-only property of an object.

COM Syntax

HRESULT get_Count(
    long **pVal*
);

Parameters

pVal [out, retval]

A pointer to the location of the number of objects in the export list is stored.

Return Value

S_OK indicates successful completion.

An error value indicates that the method failed to complete successfully. For more information, see XML Encoding Errors.

Automation Syntax

In Automation, this property has the following syntax:

variable = object.Count

The **Count** property syntax has the following parts.

Parameter	Description
object	An object declared as **MSMDCXML.Export**
variable	A **long** variable that contains the value of the **Count** property

See Also

XML Import Export

Meta Data Services Programming

IExport::Export Method

This method exports the marked objects into the file specified by the file name parameter.

COM Syntax

HRESULT Export(
    BSTR *XML*,
    long *Flags*
);

Parameters

XML [in]

The file name of the Extensible Markup Language (XML) document or XML string.

Flags [in]

Flag values that can be combined in a bitwise OR operation to control the way exported objects are handled in the output. The following table describes the flags.

Enumerator	Value	Description
NOOBJID	1	If this bit is set, no object identifiers (OBJID) are returned for the objects being exported.
NOHEADER	2	If this bit is set, the XML file does not include a transfer header.
INDENTATION	4	If this bit is set, the system indents the XML.
UNICODE	8	If this bit is set, the system output is Unicode.
EXPORTBASE	16	If this bit is set, the system exports only base properties and collections.

Return Value

S_OK indicates successful completion.

An error value indicates that the method failed to complete successfully. For more information, see XML Encoding Errors.

Automation Syntax

In Automation, the **Export** method has the following syntax:

object.Export XMLFilename *[,Flags]*

The **Export** method syntax has the following parts.

Parameter	Description
object	An object declared as **MSMDCXML.Export**.
XMLFilename	The XML file name declared as **string**.
Flags	Flag values that can be combined in a bitwise OR operation to control the way exported objects are handled in the output. The values are declared as **long**.

See Also

XML Import Export

Meta Data Services Programming

IExport::GetXML Method

This method exports the marked objects into the Extensible Markup Language (XML) string.

COM Syntax

HRESULT GetXML(
    long *Flags*
    BSTR *XML*,
);

Parameters

XML [out]

The XML output string.

Flags [in]

Flag values that can be combined in a bitwise OR operation to control the way exported objects are handled in the output. The following table describes the flags.

Enumerator	Value	Description
NOOBJID	1	If this bit is set, no object identifiers (OBJID) are for the objects being exported.
NOHEADER	2	If this bit is set, the XML file does not include a transfer header.
INDENTATION	4	If this bit is set, the system indents XML.
UNCLODE	8	If this bit is set, the output is Unicode.
EXPORTBASE	16	If this bit is set, the system exports only base properties and collections.

Return Value

S_OK indicates successful completion.

An error value indicates that the method failed to complete successfully. For more information, see XML Encoding Errors.

Automation Syntax

In Automation, the **GetXML** method has the following syntax:

object.GetXML *[,Flags]*

The **GetXML** method syntax has the following parts.

Parameter	Description
object	An object declared as **MSMDCXML.Export**.
XML	The XML output string.
Flags	Flag values that can be combined in a bitwise OR operation to control the way exported objects are handled in the output. This part is declared as **long**.

See Also

XML Import Export

Meta Data Services Programming

IExport::Item Method

This method allows the client to access elements within the list of objects to be exported.

COM Syntax

HRESULT Item(
    VARIANT *Index*,
    IRepositoryObject ***ppRO*
);

Parameters

Index [in]

A variable that contains the object sequence in the object list. This parameter can be a zero-based numeric index, an object identifier (OBJID), or a string-based OBJID.

ppRO [out, retval]

A pointer to a repository object.

Return Value

S_OK indicates successful completion.

An error value indicates that the method failed to complete successfully. For more information, see XML Encoding Errors.

Automation Syntax

In Automation, the **Item** method is attached to the **IExport** object and has the following syntax:

Set variable = object.**Item***(index)*

The **Item** method syntax has the following parts.

Parameter	Description
object	An object declared as **MSMDCXML.Export**.
variable	An object expression that evaluates to a **RepositoryObject** object.
Index	A variable declared as variant. It contains the object sequence number, string **ObjectId**, or OBJID in the object list.

See Also

XML Import Export

Meta Data Services Programming

IExport::Remove Method

This method removes the selected object from the export collection.

COM Syntax

HRESULT Item(
    VARIANT *Index*,
);

Parameters

Index [out, retval]

A variable that contains the object sequence in the object list. This parameter can be a zero-based numeric index, an object identifier (OBJID), or a string-based OBJID.

Return Value

S_OK indicates successful completion.

An error value indicates that the method failed to complete successfully. For more information, see XML Encoding Errors.

Automation Syntax

The following syntax is used in automation:

```
object.Remove(Index)
```

See Also

XML Import Export

Meta Data Services Programming

XML IImport Interface Overview

Extensible Markup Language (XML) can be used to import XML documents into a Microsoft® SQL Server™ 2000 Meta Data Services repository.

If importing an object that already exists, and this object is marked as versioned, the following rules apply:

- If the version flag is set, the system will freeze the original object and create a new version.

- If the version flag is not set, the system will overwrite the original object to the defined pointer.

IDL Definition

The following expandable text is the part of the Interface Definition Language (IDL) file that describes the methods on the **IImport** interface. In Automation, properties and methods are attached to the import object.

IDL Segment

```
interface IImport : IDispatch
{
  [id(1), helpstring("method ImportXML")] HRESULT ImportXML([i
  [id(2), helpstring("method ImportXMLString")] HRESULT ImportX
};
```

See Also

XML Import Export

Meta Data Services Programming

IImport::ImportXML Method

This method is used to import objects from an Extensible Markup Language (XML) document. The document file name and the repository pointer are passed to the method as parameters. The repository this method uses must be opened in the exclusive mode.

COM Syntax

HRESULT ImportXML(
    IRepository **pRepository*,
    BSTR *XMLFile*,
    ITransientObject ***pp ITOL*
    long *Flags*
);

Parameters

pRepository [in]

A pointer to the **IRepository** interface.

XMLFile [in]

The XML document file name.

***pp ITOL* [out]

A collection of top-level objects to be imported.

Flags [in]

Flag values that control the way XML Importer works.

Enumerator	Bit	Description
NOOVERWRITE	1	If this bit is set, the system generates an error if an object in the file already exists in the target repository.
NEWVERSION	2	If this bit is set, the system

		automatically creates a new version of any object that already exists.
NOOBJECTCHECK	4	If this bit is set, the system does not check for object existence. If the object exists, an error occurs when the object is created or committed.
IGNOREUNKNOWNTAGS	8	If this bit is set, the system ignores unrecognized tags.
LOGUNKNOWNTAGS	16	If this bit is set, the system creates a file called Msmdcxml.log in the Temp directory. The file contains all ignored tags and attributes.
LOGUNMAPPED	32	If this bit is set, the system logs everything that is not mapped during the import from Open Information Model (OIM) 1.0 to the Meta Data Coalition (MDC) OIM.

Return Value

S_OK indicates successful completion.

An error value indicates that the method failed to complete successfully. For more information, see XML Encoding Errors.

Automation Syntax

In Automation, the **ImportXML** method is attached to the **Import** object and has the following syntax:

Set TransientCol = object.**ImportXML** *(pRepository, XMLFile [, Flags])*

See Also

XML Import Export

Meta Data Services Programming

IImport::ImportXMLString Method

This method is used to import objects from an Extensible Markup Language (XML) document provided as a string. The document file name and the repository pointer are passed to the method as parameters. The repository used by this method must be opened in the exclusive mode.

COM Syntax

HRESULT ImportXML(
IUnknown **pRepository*,
BSTR *XML*,
ITransientObject ***pp ITOL*
long *Flags*
);

Parameters

pRepository [in]

A pointer to the **IUnknown** interface used as a repository interface pointer.

XML [in]

The XML string to import from.

***pp ITOL* [out]

A collection of top-level objects.

Flags [in]

Flag values that control the way XML Importer works. These flags, defined in the following table, are mutually exclusive.

Enumerator	Bit	Description
NOOVERWRITE	1	If this bit is set, the system generates an error if an object in the file already exists in the target repository.

NEWVERSION	2	If this bit is set, the system automatically creates a new version of any object that already exists.
NOOBJECTCHECK	4	If this bit is set, the system does not check for object existence. If the object exists, an error occurs when the object is created or committed.
IGNOREUNKNOWNTAGS	8	If this bit is set, the system ignores any tags that are not recognized.
LOGUNKNOWNTAGS	16	If this bit is set, the system creates a file called Msmdcxml.log in the Temp directory. This file contains all ignored tags and attributes.
LOGUNMAPPED	32	If this bit is set, the system logs everything that is not mapped during the import from Open Information Model (OIM) 1.0 to the Meta Data Coalition (MDC) OIM.

Return Value

S_OK indicates successful completion.

An error value indicates that the method failed to complete successfully. For more information, see XML Encoding Errors.

Automation Syntax

In Automation, the **ImportXMLString** method is attached to the **Import** object and has the following syntax:

Set Col = object.**ImportXMLString** *(pRepository, XML, [,Flags])*

See Also

XML Import Export

Meta Data Services Programming

XML Encoding Errors

The following table lists the error codes and messages returned by the XML Interchange Format methods. A workaround or an explanation follows each error.

Error number	Error text	Description
0x80042000	E_REPXML_REPNOTINITIALIZED	Repository is not initia
0x80042001	E_REPXML_INVALIDFILE	XML format for the in
0x80042002	E_REPXML_LIBNOTFOUND	Could not find type lib
0x80042003	E_REPXML_TXNCREATE	Could not create a tran
0x80042004	E_REPXML_OBJNOTFOUND	Object was not found.
0x80042005	E_REPXML_IMPORT_INVALIDFLAG	Invalid flag or combin
0x80042006	E_REPXML_EXPORT_INVALIDFLAG	Invalid flag or combin
0x80042007	E_REPXML_INVALIDFILENAME	File name "%s" is not
0x80042008	E_REPXML_CANTCREATEFILE	Error creating file.
0x80042009	E_REPXML_ITEMEXISTS	Item with transfer ID '
0x8004200a	E_REPXML_ERRORPARSING	Error parsing XML fil
0x8004200b	E_REPXML_ERROREXPORTING	Error occurred while e
0x8004200c	E_REPXML_ERRORADDINGOBJ	

		Error occurred while a
0x8004200d	E_REPXML_ERRORGETITEM	Error occurred while g
0x8004200e	E_REPXML_ERRORREMOVEOBJ	Error occurred when r
0x8004200f	E_REPXML_INVALIDBINARY	Binary property %s of
0x80042010	E_REPXML_COLADDERROR	Error adding to collect
0x80042011	E_REPXML_IMPORTOBJECT	Error creating new obj
0x80042012	E_REPXML_NOTRANSID	Object does not contai
0x80042013	E_REPXML_ERRORSETTINGPROP	Error setting property
0x80042014	E_REPXML_EXPORTOBJEXIST	Object already exists i
0x80042015	E_REPXML_DUPEPREFIX	Prefix %s of model %
0x80042016	E_REPXML_WRITEFILE	Error writing file.
0x80042017	E_REPXML_RETURNCOLERROR	Error occurred adding
0x80042018	E_REPXML_READINGTIM	An error occurred read
0x80042019	E_REPXML_CANTGETCOL	Error getting collectio
0x8004201a	E_REPXML_ERROROPENFILE	Error opening file %s.
0x8004201b	E_REPXML_ERROROPENTEMPFILE	Error opening tempora
0x8004201c	E_REPXML_ERRORWRITETEMPFILE	Error writing to temp

0x8004201d	E_REPXML_UNMAPPEDOBJECT	Name %s is unmappe
0x8004201e	E_REPXML_NOOBJECSTTOEXPORT	No objects to export.

See Also

Using XML Encoding

XML Encoding Reference

XML in Meta Data Services

Meta Data Services Programming

OLE DB Scanner Reference

The scanner provides one dual interface, **IRepOLEDBScanner**. This interface supports two methods, **ScanDB** and **ScanConnection**.

Topic	Description
IRepOLEDBScanner::ScanDB	Copies the schema into the repository.
IRepOLEDBScanner::ScanConnection	Copies the schema from the OLE DB session into the repository.

See Also

Using OLE DB Scanner

Meta Data Services Programming

IRepOLEDBScanner::ScanDB

IRepOLEDBScanner::ScanDB copies the schema from the specified OLE DB data source object into a specified repository. If the catalog is already in the specified repository, the systemwill version and store the original schema.

Syntax

HRESULT ScanDB (**IRepository** * *pRepository*,
IRepositoryObject ***pDbmDataSource*,
BSTR *szProviderName*,
BSTR *szProviderString*,
BSTR *szDataSource*,
BSTR *szCatalog*,
BSTR *szUserName*,
BSTR *szPassword*
);

Parameters

pRepository [in]

A pointer to an **IRepository** interface that represents the repository where the class instances will be stored.

pDbmDataSource [in, out]

A pointer to an interface for a repository data source object. If the object does not support the **IDbmDataSource** interface, the scanner will create the data source object and assign the pointer to the newly created object.

szProviderName [in]

The OLE DB provider name or program identifier.

szProviderString [in, optional]

A provider-specific connection string for the scanner to use during provider initialization.

szDataSource [in, optional]

A provider-specific location of the data source. Typically, this will be a server name or the path of the database file.

szCatalog [in, optional]

A provider-specific database name. If the database name is not specified, the default catalog in the data source will be scanned.

szUserName [in, optional]

The database user name for login. If the user name is specified in the connect string, this parameter is not required.

szPassword [in, optional]

The user password for authentication. If the user name is specified in the connect string, this parameter is not required.

Return Value

S_OK

The method succeeded.

E_FAIL

A provider-specific error occurred.

E_INVALIDARG

Either *pRepository* or *pDbmDataSource* is a null pointer.

Meta Data Services Programming

IRepOLEDBScanner::ScanConnection

IRepOLEDBScanner::ScanConnection copies the schema from the connected OLE DB session object into the specified repository. If the catalog is already in the specified repository, the system will create a versioned copy of the original schema.

Syntax

HRESULT ScanConnection (**IRepository** * *pRepository,*
IRepositoryObject ***pDbmDataSource,*
IUnknown * *pSession,*
BSTR *szCatalog*
);

Parameters

pRepository [in]

A pointer to an **IRepository** interface that represents the repository where the class instances will be stored.

pDbmDataSource [in, out]

A pointer to an interface for a repository data source object. If the object does not support the **IDbmDataSource** interface, the scanner will create the data source object and assign the pointer to the newly created object.

pSession [in]

An interface pointer to an initialized OLE DB session object. The scan is not possible if the session does not support schema information through the **IDBSchemaRowset** interface. Appropriate initialization properties should already be set on the session.

szCatalog [in, optional]

A provider-specific database name. If the database name is not specified, the current or default catalog in the data source will be scanned.

Return Value

S_OK

The method succeeded.

E_FAIL

A provider-specific error occurred.

E_INVALIDARG

Either *pRepository*, *pDbmDataSource*, or *pSession* was a null pointer.

Meta Data Services Programming

Model Installer Reference

The Microsoft® SQL Server™ 2000 Meta Data Services information model installer reads a binary information model RDM file and installs the information model in the specified repository database. The installer can be used to install either prebuilt information model installation files, such as those provided with the Meta Data Coalition (MDC) Open Information Model (OIM), or user information model files generated by the Model Development Kit (MDK). A user model must be successfully compiled by the MDK before it can be installed on a repository database. The compiling process creates the .rdm input string for the model installer.

The model installer has a dependency on the repository engine DLL. It will read the installation script and create the information model in the specified repository.

The installer is compatible with earlier versions of the Meta Data Services repository .rdm files and handles them correctly.

When the model installer recognizes that the model already exists, it will check to see whether any additions have been made (classes, interfaces, properties, collections, relationships, methods, and so on) and reinstall them as required.

Command Line Installer

The command line installer uses the Installer COM server DLL to perform the actual installation or deletion of model files. It outputs any error message to the console window.

The syntax of the two possible command lines are:

InsRepIM.exe /f[*Model File*] /r[*Repository connect string*] /u[*User*] /p[*Password*]

-or-

InsRepIM.exe /d /r[*Repository connect string*] /u[*User*] /p[*Password*]

WARNING Using the flag /d deletes all repository tables and property tables from the repository database that are specified by the connection string using the user

ID and password.

The following table lists the parameters.

Parameter	Description
Model File	The information model data file (with a file extension of .rdm)
Repository connect string	The repository database file data source name (DSN) or a Microsoft Access database file (with a file extension of .mdb)
User	The user's name
Password	The user's password

Example

The following examples show how you can use either the DSN or the .mdb to identify the file name:

```
InsRepIM /f C:\MyRdmFolder\Mar.rdm /r DSN=Mar /u MyName /p

InsRepIM /f C:\MyRdmFolder\Mar.rdm /r C:\MyMdbFolder\Mar.mc
```

Installer COM API

The model installer API is the same as in Microsoft Repository version 2.0. The **IMInstall** COM server publishes the **IMInstall** interface.

The file Insrepim.dll is a Microsoft ActiveX® DLL located in C:\Program Files\Common Files\Microsoft Shared\Repository. It can be used either from a Microsoft Visual Basic® application or a Microsoft Visual C++® application to programmatically install a model file into a repository database.

Topic	Description
IIMInstall::InstallRDM Method	Describes the method that is used to install the model by DSN or connection name
IIMInstall2::InstallRDM Method	Describes the method that is used to

	install the model by repository pointer
Model Installer Errors	Lists the model installer error messages

The model installer uses the following sample Interface Definition Language (IDL) definition to install models into a Meta Data Services repository.

IDL Definition

IDL Segment

```
[
  object,
  uuid(D24FD4A4-BEBC-11D1-8CB9-00C04FC2F51A),
  dual,
  helpstring("IIMInstall Interface"),
  pointer_default(unique)
]
interface IIMInstall : IDispatch
{
  [id(1), helpstring("method InstallRDM")] HRESULT InstallRDM(|
};

[
  object,
  uuid(AF7F843B-FB34-4ff2-BD7D-81DDB284D2A9),
  dual,
  helpstring("IIMInstall2 Interface"),
  pointer_default(unique)
]
interface IIMInstall2 : IDispatch
{
  [id(1), helpstring("method InstallRDM")] HRESULT InstallRDM(|
```

```
};
```

See Also

Installing Information Models

Model Installer Errors

Meta Data Services Programming

IIMInstall::InstallRDM Method

IIMInstall supports the following method:

HRESULT InstallRDM(

BSTR *DSN*,

BSTR *RdmFile*,

BSTR *UserName*,

BSTR *Password*

);

Parameters

DSN [in]

The data source name (DSN) of the repository database.

RdmFile [in]

The information model data file .rdm.

UserName [in]

The user's name.

Password [in]

The user's password.

Return Value

S_OK

The method is successfully completed.

Error Value

The method failed to complete successfully.

See Also

Model Installer Errors

Model Installer Reference

Meta Data Services Programming

IIMInstall2::InstallRDM Method

IIMInstall supports the following method:

HRESULT InstallRDM(

IRepository **pRepos*,

BSTR *RdmFile*,

);

Parameters

**pRepos* [in]

Points to a repository database where the model is to be installed.

RdmFile [in]

The information model data file (with an extension of .rdm).

Return Value

S_OK

The method is successfully completed.

Error Value

The method failed to complete successfully.

See Also

Model Installer Reference

Model Installer Errors

Meta Data Services Programming

Model Installer Errors

The following errors may occur when you install a model into a Microsoft® SQL Server™ 2000 Meta Data Services repository.

Error number	Error text
0x80045001	E_INSREP_BAD_ARGUMENTS
0x80045002	E_INSREP_CANT_OPEN_MODEL_FILE
0x80045003	E_INSREP_CANT_INITIALIZE_COM
0x80045004	E_INSREP_CANT_CREATE_IREPOSITORY
0x80045005	E_INSREP_REPOSITORY_CREATE_FAILS
0x80045006	E_INSREP_PREMATURE_EOF
0x80045007	E_INSREP_WRONG_FILE_TYPE
0x80045008	E_INSREP_UNEXPECTEDERROR
0x80045009	E_INSREP_CANTINSTALLTYPELIB
0x8004500A	E_INSREP_INCOMPATIBLERDMVERSION
0x8004500B	E_INSREP_CANTINSTALLINTERFACEDEF
0x8004500C	E_INSREP_CANTINSTALLPROPERTYDEF
0x8004500D	E_INSREP_CANTINSTALLRELATIONSHIPDEF
0x8004500E	E_INSREP_CANTINSTALLROLEDEF
0x8004500F	E_INSREP_CANTINSTALLCLASSDEF
0x80045010	E_INSREP_TRANSACTIONERROR
0x80045011	E_INSREP_CANTCREATEENUMDEF
0x80045012	E_INSREP_CANTCREATEENUMLITERAL
0x80045013	E_INSREP_CANTCREATEOPERATION
0x80045014	E_INSREP_CANTCREATEALIAS
0x80045015	E_INSREP_IMPLIESFAILED
0x80045016	E_INSREP_ERRORADDINGIFACE
0x80045017	E_INSREP_ERRORGETTINGREPOSROOT
0x80045018	E_INSREP_CANTCREATEPARAMDEF
0x80045019	E_INSREP_INCOMPREPOSVERSION

See Also

Model Installer Reference

Meta Data Services Programming

E_INSREP_BAD_ARGUMENTS

One or more of the arguments passed are not valid.

Meta Data Services Programming

E_INSREP_CANT_CREATE_IREPOSITORY

Microsoft® SQL Server™ 2000 Meta Data Services is not registered on this computer.

Meta Data Services Programming

E_INSREP_CANT_INITIALIZE_COM

The installer failed to initialize COM.

Meta Data Services Programming

E_INSREP_CANT_OPEN_MODEL_FILE

The installer cannot open the model file.

Meta Data Services Programming

E_INSREP_CANTCREATEALIAS

The installer failed to create an alias.

Meta Data Services Programming

E_INSREP_CANTCREATEENUMDEF

The installer failed to create an enumeration.

Meta Data Services Programming

E_INSREP_CANTCREATEENUMLITERAL

The installer failed to create an enumeration literal.

Meta Data Services Programming

E_INSREP_CANTCREATEOPERATION

The installer failed to create a **MethodDef** class.

Meta Data Services Programming

E_INSREP_CANTCREATEPARAMDEF

The installer failed to install a parameter definition.

Meta Data Services Programming

E_INSREP_CANTINSTALLCLASSDEF

The installer failed to install a class definition.

Meta Data Services Programming

E_INSREP_CANTINSTALLINTERFACEDEF

The installer failed to install an interface definition.

Meta Data Services Programming

E_INSREP_CANTINSTALLPROPERTYDEF

The installer failed to install a property definition.

Meta Data Services Programming

E_INSREP_CANTINSTALLRELATIONSHIPDEF

The installer failed to install a relationship definition.

Meta Data Services Programming

E_INSREP_CANTINSTALLROLEDEF

The installer failed to install a relationship collection definition.

Meta Data Services Programming

E_INSREP_CANTINSTALLTYPELIB

The installer failed to install a type library.

Meta Data Services Programming

E_INSREP_ERRORADDINGIFACE

An error occurred while adding an interface.

Meta Data Services Programming

E_INSREP_ERRORGETTINGREPOSROOT

An error occurred while getting the repository root.

Meta Data Services Programming

E_INSREP_IMPLIESFAILED

An error occurred while adding an implication.

Meta Data Services Programming

E_INSREP_INCOMPATIBLERDMVERSION

The RDM file version is incompatible with the installer.

Meta Data Services Programming

E_INSREP_INCOMPREPOSVERSION

The installer requires Microsoft® SQL Server™ 2000 Meta Data Services.

Meta Data Services Programming

E_INSREP_PREMATURE_EOF

The installer ended unexpectedly.

Meta Data Services Programming

E_INSREP_REPOSITORY_CREATE_FAILS

The installer failed while creating a repository.

Meta Data Services Programming

E_INSREP_TRANSACTIONERROR

An error occurred while creating or committing a transaction.

Meta Data Services Programming

E_INSREP_UNEXPECTEDERROR

An unexpected error occurred.

Meta Data Services Programming

E_INSREP_WRONG_FILE_TYPE

File type is unknown.