Rational Suite®

Programmer's Guide to Application Development

Rational Suite Extensibility

VERSION: 2002.05.00

PART NUMBER: 800-025143-000

WINDOWS

Rational®
the software development company

support@rational.com
http://www.rational.com

Part Number: 800-025143-000

Version Number: 2002.05.00

Contents

Preface

RSE delivers a comprehensive set of application programming interfaces (APIs) that provide a single platform on which to develop client and server capabilities between integrated products in Rational Suite.

This manual introduces the basic concepts of Rational Suite Extensibility (RSE) and provides the details for developing applications using the COM client interfaces.

Audience

This manual is intended for administrators, project managers, and all members of the software development team, including requirements developers, software architects and developers, and quality engineers.

Other Resources

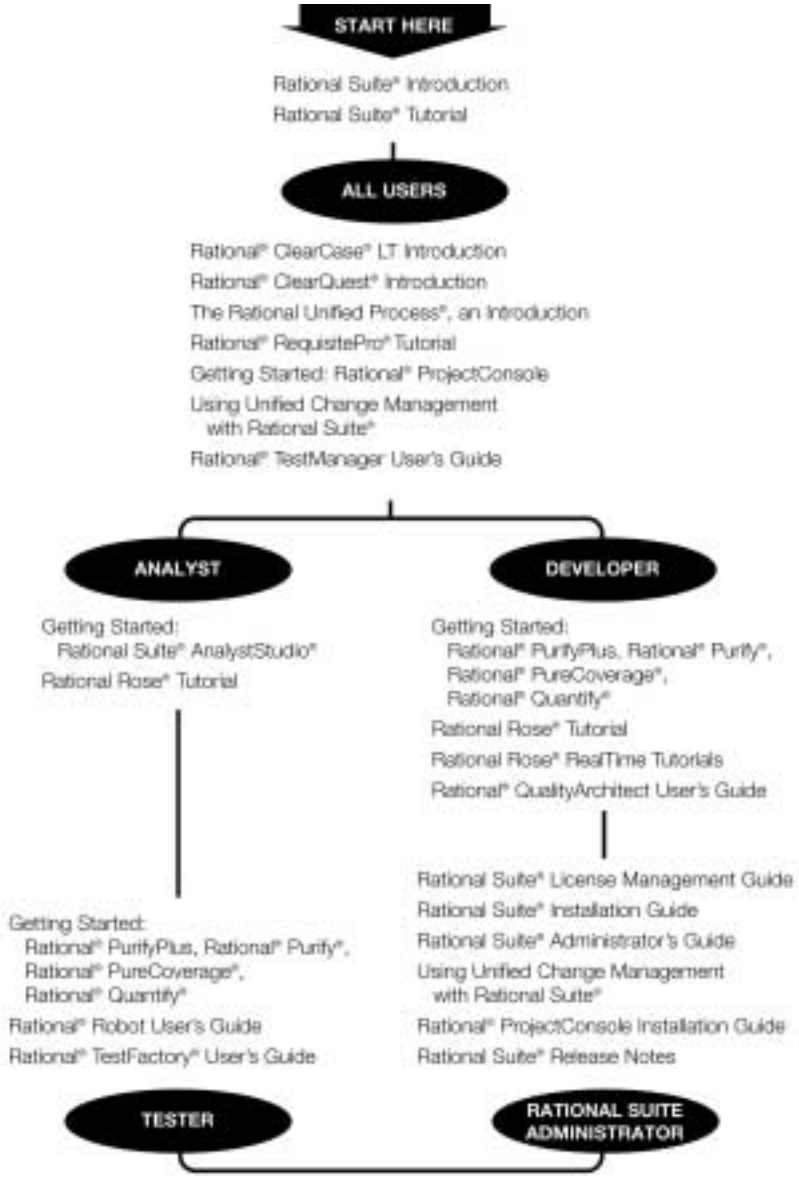
- Other RSE documentation:
  - COM Client API Reference
  - Adapters Reference
  - Programmer's Guide to Adapter Development
- Rational extensibility API references:
  - ClearCase Reference Manual
  - ClearQuest API Reference
  - RequisitePro Extensibility Interface Online Help

    RequisitePro extensibility information is documented in the RequisitePro online help for the RequisitePro Extensibility Interface. It is available from the Help menu on the ReqPro tool palette.
  - Rose Extensibility Reference
  - Team Manager Extensibility Reference
- Online Help is available for Rational Suite.

From a Suite tool, select an option from the **Help** menu.

- All manuals are available online, either in HTML or PDF format. The online manuals are on the Rational Solutions for Windows Online Documentation CD.
- To send feedback about documentation for Rational products, please send e-mail to techpubs@rational.com.
- For more information about Rational Software technical publications, see: http://www.rational.com/documentation.
- For more information on training opportunities, see the Rational University Web site: http://www.rational.com/university.

Rational Suite Documentation Roadmap



Contacting Rational Technical Support

If you have questions about installing, using, or maintaining this product, contact Rational Technical Support as follows:

Your Location	Telephone	Facsimile	E-mail
North America	(800) 433-5444 (toll free) (408) 863-4000 Cupertino, CA	(781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 (0) 20-4546-200 Netherlands	+31 (0) 20-4546-201 Netherlands	support@europe.rational.com
Asia Pacific	+61-2-9419-0111 Australia	+61-2-9419-0123 Australia	support@apac.rational.com

Note: When you contact Rational Technical Support, please be prepared to supply the following information:

- Your name, company name, telephone number, and e-mail address
- Your operating system, version number, and any service packs or patches you have applied
- Product name and release number
- Your case ID number (if you are following up on a previously reported problem)

What Is RSE? 1

Rational Suite delivers a comprehensive set of integrated tools that embody software engineering practices and span the entire software development lifecycle. Each individual application has its own API for retrieving stored information. Until now, you've needed to use a separate API for programming access to each tool in Rational Suite.

Rational Suite Extensibility (RSE) defines a set of interfaces that provides one unified platform for retrieving information in any application within Rational Suite.

The vision of RSE is to provide unified access to Rational Suite. In a sense, RSE makes it possible to view the Suite as a single application, not a collection of separate applications.

The goal of RSE is to support existing integrated product extensibility and enhance current capabilities by providing adaptable, platform-neutral, distributed availability. The RSE interfaces are designed to support equivalent functionality for the platforms that developers need.

Why Create RSE?

RSE supports the accelerating demand for Rational Suite by making it easier to customize the Suite for particular customer situations. RSE satisfies the demand for tighter integration and consistency between individual products in the Suite and customer integrations. Rather than working with individual-product APIs, RSE simplifies the process of writing applications that work with the Suite.

RSE is complementary to the integrated product APIs. This allows RSE code to operate with code written specifically for a given integrated product interface.

Benefits of Using RSE

With RSE:

- You can build client applications that access all integrated product applications, including all Rational Suite products and integrations to the Suite. Access to each integrated product application is through its associated RSE adapter.

- You can build RSE adapters and install them on a system. Each adapter maps an integrated product object model to the RSE object model. These adapters are available to RSE Client Applications.
- New client applications and adapters work with any Suite-enabled technology.
- If a Rational partner writes an application that takes advantage of this technology, it will instantly be capable of using new adapters without modifying code.

Without RSE:

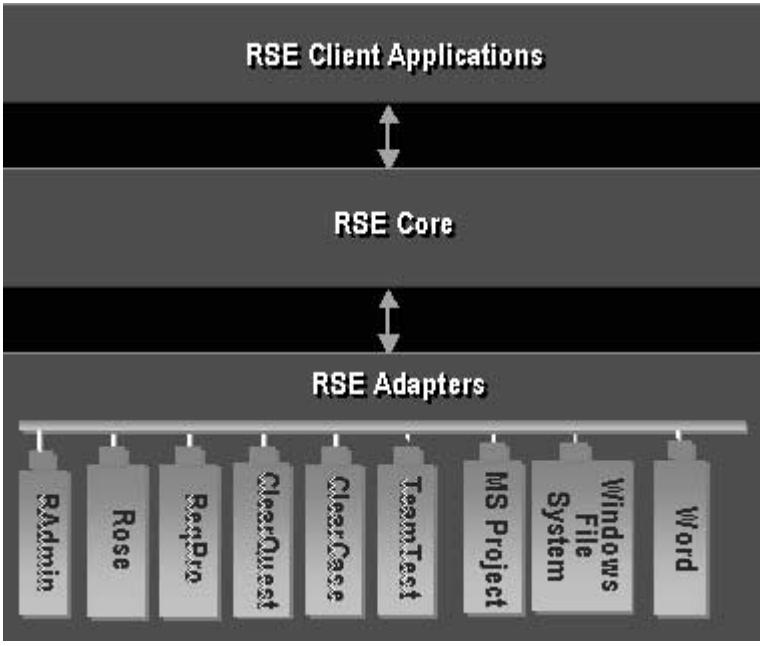
Features must be built using each specific product's extensibility interface. This approach forces you to implement the same features for each product in the Suite. The problem becomes worse as new products are added to the Suite.

RSE Implementation

RSE is implemented by the RSE Core. This core defines a set of interfaces (for example, the RSE COM client interfaces) that provides Rational Suite extensibility.

Figure 1 shows the three-tiered architecture of an RSE client connecting to an integrated product adapter in the Suite.

Figure 1 RSE Implementation



In Figure 1, the lower tier that includes the integrated products refers to the RSE adapters for each integrated product, not the products themselves. For example, ReqPro is the RSE adapter that maps RSE to RequisitePro.

As Figure 1 illustrates:

- RSE client applications provide access to the integrated products in the Suite.
- The RSE core maps the implementation of client interfaces to integrated product RSE adapters. The RSE core provides the interface between client applications and adapters. This implements the RSE client interfaces communicating with the RSE adapters to retrieve information in each of the specific products.
- Product-specific RSE adapters provide data retrieval from the RSE core to each integrated product. Each adapter provides the mapping of an integrated product's data (objects) to an RSE generic object model. Artifacts are the RSE objects that represent integrated product objects.

Using RSE

You can use RSE to create:

- Clients

A client application allows you to retrieve data in the Suite and other integrated products.

- Adapters

  An adapter provides access to the applications that contain the data. Adapters act as servers to RSE clients and allow data to be integrated between individual products in the Suite.

  Individual adapters provide a consistent standard interface between the RSE core and individual products. RSE provides an adapter for each product in the Suite (for example, an adapter named ReqPro for RequisitePro) and also provides adapters for common Microsoft applications. The RSE adapters are:

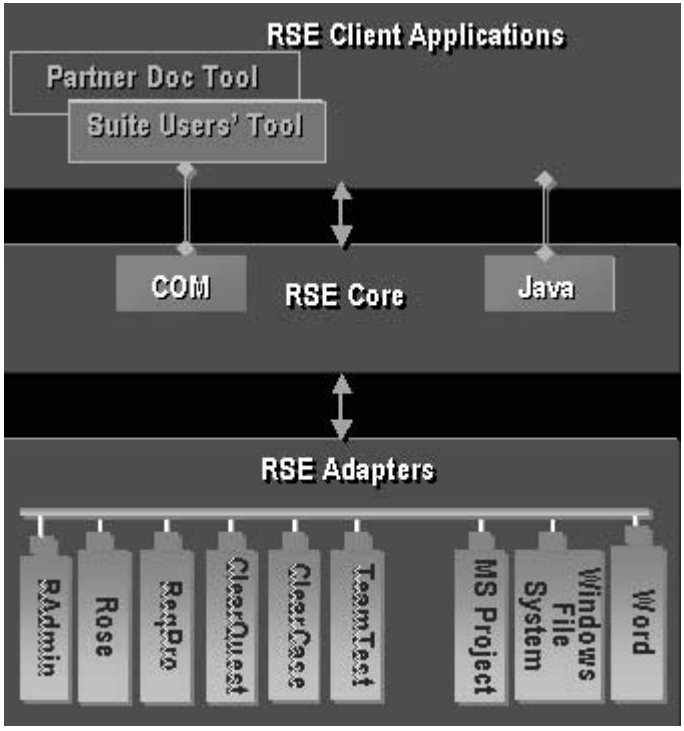
  - Rational Administrator (RAdmin)
  - Rational Rose (Rose)
  - RequisitePro (ReqPro)
  - Rational ClearQuest (ClearQuest)
  - Rational ClearCase (ClearCase)
  - Rational Test Manager (TeamTest)
  - Microsoft Windows File System (FileSys)
  - Microsoft Project (MSProject)
  - Microsoft Word (Word)

RSE Clients

You can create client applications to retrieve data from any product in the Suite. RSE can support multiple client interfaces. COM is currently the supported interface.

Figure 2 shows two client applications to Rational Suite. These applications can retrieve data from any of the Suite products or other integrated products (through the RSE adapters).

Figure 2 RSE Clients



Create client applications to:

- Query Rational Suite for application objects (that map to RSE artifacts), using filtering operators.
- Perform simple artifact create, read, update, and delete operations.
- Provide end-user ease of use for access to Rational Suite and other integrated products.

RSE provides developers of client applications with:

- A single data access API. This means that clients do not have to modify code to access any RSE-enabled application. As more applications become RSE-enabled, RSE clients automatically have access to new application data.

- A consistent mechanism for relating objects within and across applications. Clients can create and manage their own links between objects attaching any semantics to the links that they choose. Clients can also get access to links created by any other client applications, making it easy for clients to share information and implement point-to-point integrations.
- A tight integration with Rational Suite.

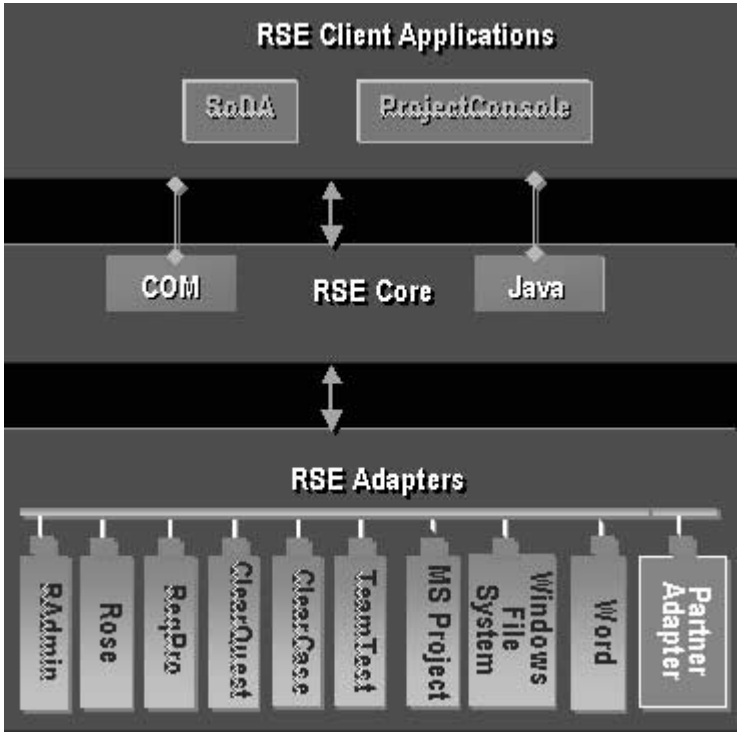
RSE Adapters

You can create adapters that enable applications to integrate with Rational Suite. These applications then act as products in the Suite, supplying data that can be retrieved by client applications.

The adapters connect to the RSE core. Adapters represent defined Rational artifacts stored in each integrated product. Adapters map an integrated product object hierarchy to the RSE artifact hierarchy. An adapter is created for each integrated product. When you create an adapter for an existing application, that application becomes an integration to the Suite, with its data available to all RSE clients.

Figure 3 shows a partner adapter that would allow that partner's application to act as part of the Suite. Data in the partner application would be defined as RSE artifacts in the Partner Adapter and client applications (for example, SoDA) would be able to retrieve this data.

Figure 3 RSE Adapters



Rational partners can create new adapters using RSE, enabling partner applications to act as Suite members.

Adapters can conceptually be seen as server applications to RSE clients. Each adapter can also be seen as a server to the other RSE adapters for each integrated product.

Conclusion

With RSE technology, data in any integrated product in Rational Suite becomes available to an RSE client application through one API. RSE clients can retrieve data from any integrated product in Rational Suite through RSE adapters. The RSE technology provides both client interfaces and adapter interfaces.

- The client interfaces are for creating new RSE client applications.

- The adapter interfaces are for implementing the RSE adapters that are included with Rational Suite and for defining new adapters. RSE adapters are defined for each integrated product in the Suite in order to map individual-product object structures to the RSE common object model.

RSE Object Model 2

Rational Suite Extensibility uses a generic object model that maps the objects of each integrated product to an RSE artifact hierarchy. This common object model enables RSE client applications to retrieve data from any integrated product through one set of interfaces. This mapping is defined in each individual integrated product adapter.

For example, a RequisitePro Project object is mapped to an equivalent RSE Project artifact type in the ReqPro adapter. The Artifact object provides the standard mechanisms to retrieve the properties (for example, Name and Description) of the object and its relationships to other Artifacts (for example, Requirements).

RSE Objects

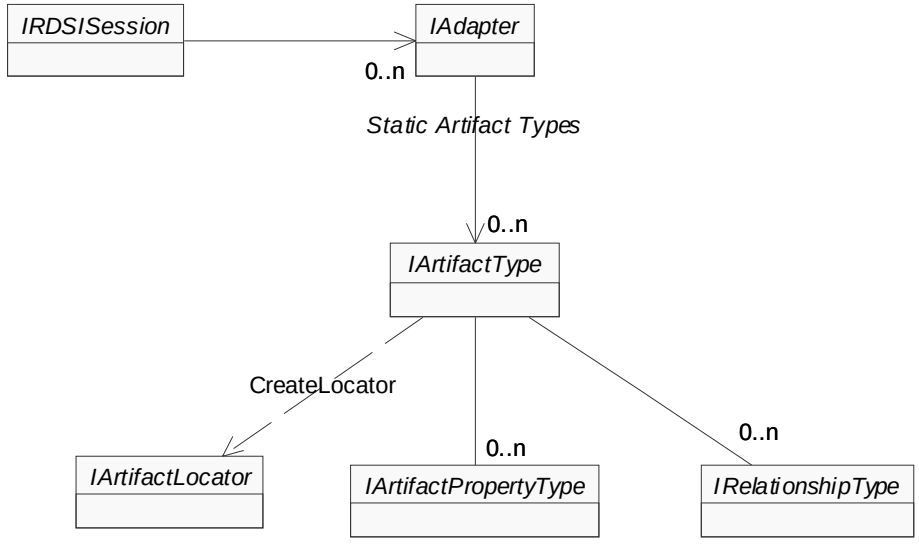
This section provides descriptions and examples of the objects in the generic object model. The primary objects are:

- Session
- Adapter
- Artifact
- Property
- Relationship
- Locator

Object Model Diagram

Figure 4 shows the main objects in the RSE generic object model. As this figure shows, the client point of entry into the RSE is through the Session object.

Figure 4 Main Objects in RSE



Session

A Session object provides access to the installed adapters. A Session object:

- Is created to work with RSE.
- Is the main object that is used to begin locating artifacts.
- Enumerates the adapters that are installed on a system.

Adapter

An adapter provides access to artifact types supported for a given product. Each adapter defines the mapping between a product's objects and the RSE generic object model representation.

An Adapter object:

- Represents a specific product.
- Contains the collection of artifacts supported in a product.
- Allows you to enumerate all of the artifact types that are supported by an adapter.

Each adapter contains the collection of artifact types that map to objects in the integrated product.

Artifacts

Artifacts are used to retrieve specific information from an integrated product.

An Artifact object represents an object in an integrated product. For example, the Rose RSE adapter defines a Class artifact to represent Class objects in a Rose model.

Artifacts:

- Contain properties and other artifacts.
- Provide access to related artifacts.
- Have an artifact type that describes additional information

ArtifactType

An ArtifactType provides detailed information about an artifact type's Locators, Properties, and Relationships.

- Locators are objects that retrieve artifacts
- Properties are attributes of an artifact
- Relationships are the associations between artifacts.

Every instance of an artifact has an artifact type. Examples of artifact types are:

- In RequisitePro:

  Project, Document, and Requirement artifact types.

- In Rose:

  Model, Package, and Class artifact types.

An actual instance of an artifact has a name and an artifact type. For example, in Rose, a class named Order is represented as an artifact with name = Order and artifact type = Class.

Static and Dynamic Artifact Types

The two kinds of RSE artifact types are static and dynamic.

The collection of static artifact types for each adapter includes all predefined artifact types.

Static types are the global artifact definitions (defined in the RSE adapters). Static types include the hierarchy of primary RSE objects that represent the objects in an integrated product. For example, in the RequisitePro RSE adapter (ReqPro), there are Project and Requirement artifact types.

The collection of static artifact types for a given adapter includes all the defined artifact types for that adapter's integrated product. These definitions are global to all top-level objects in an integrated product. The top-level object in an integrated product maps to the root artifact in that product's RSE adapter. In the ReqPro example, a Project is the root artifact in both the product hierarchy and in the ReqPro adapter.

There are also dynamic artifact types that typically represent user-defined artifact types (for example a user-defined Requirement type in RequisitePro). The dynamic types are registered within the artifact that corresponds to the integrated product top-level object (for example, a ReqPro Project artifact). This top-level artifact is the root artifact. The dynamic types may then be accessed through this root artifact.

Dynamic artifact types are registered within the RSE adapters, based on user-defined information in an integrated product. These RSE objects represent instances of user-defined objects in the integrated product (for example, an instance of a user-defined RequisitePro RequirementType).

In RequisitePro, there can be different requirement types defined in the Project properties. In the ReqPro adapter, this translates as dynamic artifact types. These dynamic types become available as additional artifact types when you instantiate RSE objects.

Defined in the RSE ReqPro adapter, there is a Requirement artifact type. This is a static artifact type. One type of requirement is a Use Case requirement type. In the RSE ReqPro adapter, a Use Case requirement is defined as a UCRequirement artifact. This dynamic type is named within the adapter by concatenating the Requirement tag prefix with the text 'Requirement.'

The UCRequirement is a subclass of a Requirement artifact (a subclass is a derived class). The subclass inherits the properties, relationships, and locator information of its superclass (a superclass is a base class). The property types of a UCRequirement artifact in the ReqPro adapter are created dynamically using the attribute types of the Use Case requirement in RequisitePro.

The RSE adapters map the dynamic artifact type hierarchy and register the appropriate artifact types, relationship types, and property types. The way in which this information is retrieved is specific to each integrated product. The dynamic type information is associated with a top-level object in the integrated product, such as a RequisitePro Project object.

The dynamic types for each RSE adapter:

- ReqPro:

Dynamic types are registered by any Project artifact. These types include user-defined Document types, Requirement types, Attributes of those Requirement types, and relationships to user defined Views defined in the Project.

- ClearQuest:

  Dynamic types are registered by the CQDatabase artifact. These include user-defined Record artifact types (typically, artifact types like Defect and ChangeRequest) including their relationships and properties (fields). The dynamic types also include relationships from the CQDatabase artifact to records for each Record type and to the results of all queries defined in the database. Retrieving the related artifacts from a query relationship causes the query to be executed. Similarly, each Query artifact has a Results relationship that also executes the query.

- Rose:

  Dynamic types are registered by the Model artifact. These include properties for each static artifact type that are registered upon locating a model (root artifact).

- RAdmin, ClearCase, TeamTest, FileSys, and MSProject do not have dynamic types.

Properties

Each artifact type has a collection of properties associated with it. Property objects correspond to the individual attributes defined in each integrated product object. Properties are available from the Artifact object.

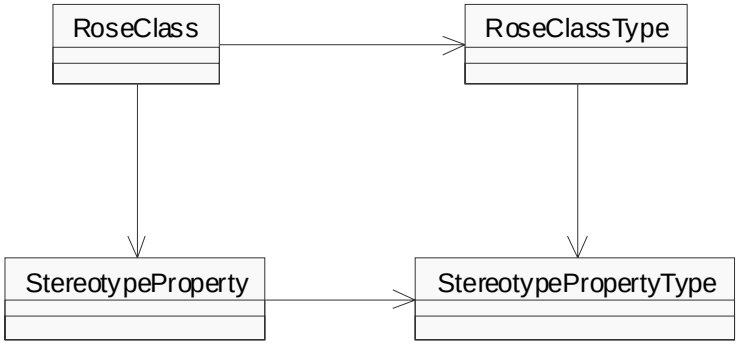
For example, the name and the stereotype of a Rose Class are properties of a Class artifact type. The Name and Stereotype properties are available from the Class artifact.

PropertyType

Every instance of a property has a corresponding property type. Each PropertyType supported by any given Artifact is available from the Artifact's ArtifactType object. This allows the properties supported by an Artifact to be listed without an instance of that Artifact.

As Figure 5 illustrates, the Rose Class has a Property called 'Stereotype', and the ArtifactType for the Rose Class has a PropertyType called 'Stereotype'.

Figure 5 Rose Class Property Example



Examples of property and property types:

- In RequisitePro:

  A Requirement ArtifactType has a Text property. The RequisitePro adapter defines a PropertyType named 'Text' for the Requirement ArtifactType.

- In Rose:

  A Package ArtifactType has a Documentation property. The Rose adapter defines a PropertyType named 'Documentation' for the Package ArtifactType.

Property types are registered with artifact types. The set of adapters maps the individual integrated-product property types to RSE artifacts and properties.

Each property type has a property ID. Property IDs are integer values assigned sequentially as the properties of an artifact type are registered. They are used internally by the RSE core to look up property definitions.

Relationships

Each artifact type defines a set of relationship types.

These relationship types are used to find related artifacts. For example:

- In RequisitePro:

  The Project artifact has a relationship to Requirements. This relationship (named, Requirements) can be used to find the Requirement objects in a Project. A Requirement has a relationship to AttributeValues (named, AttrValues). This relationship enables you to find the AttrbuteValue objects of a Requirement.

- In Rose:

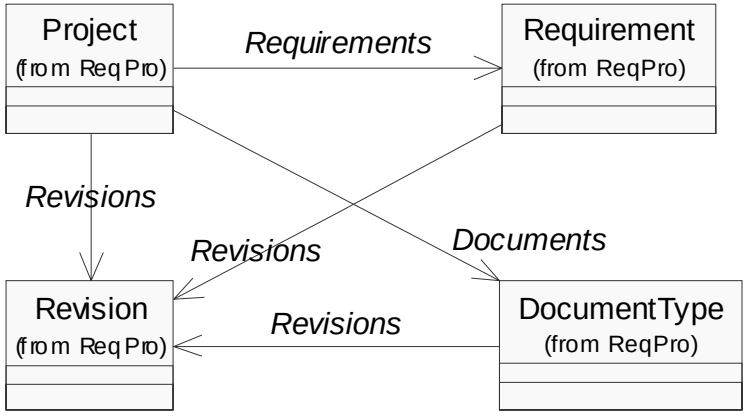
The Package artifact has a relationship to Classes. This relationship (named, Classes) can be used to find the Class objects in a Package. The Class artifact has a relationship to Properties. This relationship (named, Properties) enables you to find Property objects of a Class.

Relationships define the associations between artifacts. An artifact can be associated with any number of relationships. Each relationship links two associated artifacts.

For example, in the ReqPro adapter, Project, Requirement and DocumentType artifacts all have a Revisions relationship to 0–n Revision artifacts. Figure 6 shows these relationships. It also illustrates the Project's Documents relationship to DocumentType and the Requirements relationship to Requirement. You can configure methods for locating artifacts using these relationships. For instance, given a Project, you can retrieve a Revision in the following ways:

- Given the Revision
- Given a DocumentType
- Given a Requirement

Figure 6 ReqPro Relationships Example



Relationships can be of type peer, descendant, or child.

Locators

Locators are RSE objects that are used for finding specific instances of artifacts.

Locators provide a uniform platform for maintaining and resolving references to RSE artifacts. This allows implementing integrations and maintaining references between integrated products.

An artifact locator:

- Finds an artifact, given user-supplied input.
- Can locate and return data from an integrated product.

The IArtifactLocator interface is used to locate artifacts and is capable of representing the locator as a string format that can be persistently saved and resolved at a later time. This artifact reference contains the series of arguments that identifies a specific instance of an artifact. This string can be converted into an artifact locator without loading integrated-product data.

The arguments necessary to locate a specific artifact in an integrated product are defined by that product's RSE adapter. These properties are specific to that type of artifact. The values of these properties are then passed on to the integrated product using the extensibility interface of that product through the adapter. This code is implemented in the adapter and is specific to that integrated product.

Artifact Arguments

Each locator type has a set of arguments for constructing an artifact. These arguments are defined as artifact arguments. Artifact arguments are used to specify values for the information that is needed to locate an artifact. An artifact locator returns an instance of an artifact.

For example, in order to locate a Rose Class, you need the path of the model, the name of the Package and the name of the Class. The artifact arguments for a Class locator type are:

- Model.Path

  The file path to the Model

- Package.Name

  The name of the Package containing the class

- Class.Name

  The name of the Class

You can create an artifact locator to locate a Class by supplying values to these arguments. For example, to locate a Class named Order in a Rose Model, the arguments values are:

- 'C:\Program Files\Rational\Rose\samples\ordersystem\ordersys.mdl'
- 'Business Services'
- 'Order'

Given these arguments, the locator returns an instance of the Order class. The locator string that comprises these arguments is called an *artifact reference*.

Artifact References

An artifact reference is a string containing the arguments used to locate a specific instance of an artifact (for example, a Rose Class named Order).

For example, the following is an artifact reference for the Rose Order class:

```
Rose|Model(Path='C:\Program Files\Rational\Rose\samples\ordersystem')|
Package(Name='Business Services')|Class(Name='Order')
```

In this example, the RSE core locates the model, then the package, and then the class.

Note: Artifact references are sometimes called locator strings or Artifact IDs.

Each artifact reference:

- Serves as a unique identifier for locating a specific instance of an artifact.
- Is a string composed of information about the artifact type to be located and a set of parameters that specify an instance of the given type of artifact.

There are two types of artifact references, Display Name ID and Immutable ID. Each type of artifact reference includes two different formats, one a more readable form (DN) and a one shortened version (ID).

Note: Not all artifact types support all forms of artifact reference. See the *Adapters Reference* manual for information on each RSE artifact type.

In the Rose Ordersystem model, the artifact references to the Order Class artifact are:

- Display Name ID locator

  The human readable Display Name format can be viewed and interpreted by the end user. For example, the Display Name ID for a Rose Class named Order in the Business Services Package in Ordersys.mdl is:

  DN form:

  ```
  Rose|Model(Path='C:\Program Files\Rational\Rose\samples\ordersystem\ordersys.mdl')|
  Package(Name='Business Services')|Class(Name='Order')
  ```

  or

  ID form:

  ```
  Rose|1.1.2|Class('C:\Program Files\Rational\Rose\samples\ordersystem\ordersys.mdl',
  'Business Services','Order')
  ```

- Immutable ID locator

The persistent Identifier form maintains a persistent reference to an artifact. The artifact arguments are Model path and the Class unique ID (UUID). The UUID is a 12 digit serial number. For example, the Immutable ID form of the Artifact ID for the Rose Order class is:

DN form:

```
Rose|Model(Path='C:\Program Files\Rational\Rose\samples\ordersystem\ordersys.mdl')|
Class(UniqueID='3237F8CD03CD')
```

The UniqueID is a 12 digit serial number that identifies the Class specific to the Rose Model.

or

ID form:

```
Rose|1.1|Class('C:\Program Files\Rational\Rose\samples\ordersystem\ordersys.mdl',
'3237F8CD03CD')
```

In most cases, implementing a GUI that allows users to enter arguments for locating artifacts is preferred to presenting the raw display name string. These arguments can then be used to construct the artifact reference. However, there may be some cases when users may encounter the strings, for example, in ascii files. In this case, the more readable format of the Display Name ID is far more appropriate than the ID form of the artifact reference.

For example, in RequisitePro:

- Display Name ID

  The artifact arguments are Project path and Requirement tag. This information is visible in RequisitePro.

- Immutable ID

  The artifact arguments are Project path and Requirement key. The key is the record ID for the Requirement in the RequisitePro database. This information is used internally by RequisitePro but is not displayable.

The primary method of creating a locator is to first enumerate through the list of static artifact types, select a type, and create a locator for this artifact type. Then, you can enumerate through the collection of artifacts for this type and select an artifact. Each of these artifacts has a unique artifact ID.

The IArtiactLocator interface has the ability to enumerate and change the values of the parameters for the locator. It is not necessary to parse the ArtifactID string in order to enumerate the values, nor is it necessary to construct a string to locate an object.

In addition to allowing parameter values to be enumerated and changed, the IArtifactLocator interface supports optional parameters and default values. This allows capabilities for projects to be located using usernames and passwords, but also allowing default access without specifying user information.

Default access does not require login authentication and thus prevents exceptions. The IArtifact interface and the IArtifactType interface allow a Client to get the default Display Name or Immutable ID locator.

For more information on authentication and exception handling, see the "Creating RSE Clients" chapter of this manual.

RelativeID Artifact References

Relative IDs are shortened versions of artifact references that provide a method for locating one artifact, given another artifact. Relative IDs enable you to permanently save artifact references that allow you to reconstruct objects.

Given an artifact type you can create an artifact locator to locate an artifact. You can also use a relative locator to get the artifact. For example, in Rose, you can locate a Class, relative to a Model. The Class relative ID (relative to Model) includes the Package name and Class name. With this relative locator string, you can locate the Class artifact.

In Rose, the absolute locator string (artifact ID) for the Order class in ordersys.mdl is:

```
Rose|Model(Path='c:\Ordersys.mdl')|Package(Name='Business
Services')|Class(Name='Order').
```

The common information stored in this string can be stored once by a client and used by the relative IDs for returning reconstructed artifacts.

For example:

- Given the Package Name (Business Services), the Relative ID for Order is:

  ```
  Class(Name='Order')
  ```

  This relative ID is relative to the Business Services Package. Business Services is the artifact that provides the context for this relative ID to Order.

- Given the model, Ordersys, the relative ID for locating Order is:

  ```
  Package(Name='Business Services')|Class(Name='Order')
  ```

  This relative ID is relative to Model (Model is the context artifact).

To resolve a Relative ID, you need the relative artifact that has the context information (for example, Rose|Model(Name='Ordersys')). This minimizes the amount of information needed to be stored by each object. Root artifact information is supplied by the RSE and can be stored once by the client. This greatly reduces the amount of information needed to be stored for each link (for example, if you were resolving 10,000 links that were all relative to one model object).

For more information on locating artifacts with relative IDs, see the "Creating RSE Clients" chapter of this manual.

Summary

Figure 7 shows the main objects in the RSE object model for retrieving artifacts and their properties.

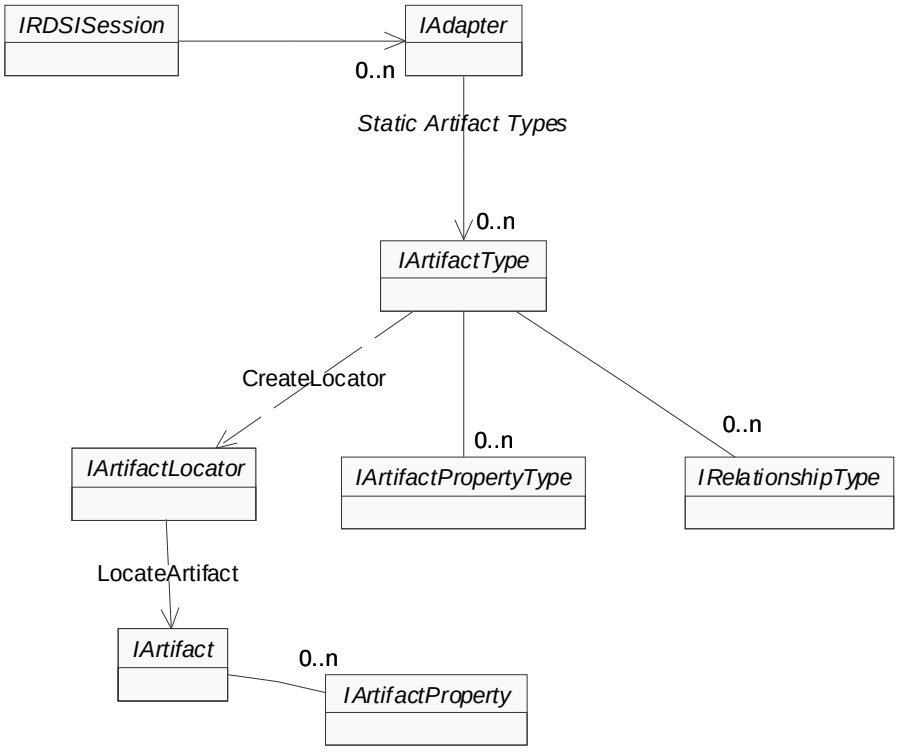
The point of entry into the RSE is through the Session object. A Session object connects to an Adapter object. From this Adapter object, you can get all of the static artifact types supported by that adapter.

Each adapter provides the conversion between internal objects from a specific product and the corresponding RSE artifacts. An adapter includes a defined class for each artifact type. Each of these internal object classes defines properties, relationships, and locators and makes available the associated objects in the integrated products.

When you have the collection of available artifact types in an adapter, you can:

- Create artifact locators to locate artifact types, artifact collections, or specific instances of artifacts.
- Retrieve the available property types for a given artifact type, or retrieve specific instances of artifact properties, for a given artifact.
- Retrieve the relationships for a given artifact type, or use relationships to retrieve artifacts that are related to a given artifact.

Figure 7 RSE Objects



SoDA Application Example

SoDA is a Rational client application that uses RSE to retrieve data from integrated products in Rational Suite. The following code shows how SoDA gets the property collection (all properties) of an object. This example creates a session, locates a ReqPro Requirement artifact (artifact ID is the locator argument), and then retrieves the properties of the Requirement artifact.

In C++:

```
IRDSISessionPtr theSession;
theSession->CreateInstance("RDSICore.Session");
IArtifactPtr theArtifact = theSession->LocateArtifact("ReqPro|
   Project(Path='<YOUR_PROJECT>')|Requirement(FullTag='<YOUR_REQ>')");
IArtifactPropertyCollectionPtr theProperties =
                                        theArtifact->GetProperties();
```

In VB:

```
Dim theSession as RDSISession
Dim theArtifact as Artifact
Dim theProperties as ArtifactPropertyCollection
Set theSession = new RDSISession
Set theArtifact =
theSession.LocateArtifact(“ReqPro|Project(Path=’<YOUR_PROJECT>’)|Requi
rement(FullTag=’<YOUR_REQ>’)”)
Set theProperties = theArtifact.Properties
```

Referencing the RDSICore Type Library

You must reference the RDSICore library into your project. The RDSICore type library is located in Rational\common\RDSICore.dll

To reference the type library in Visual Basic:

1. Click **Project > References**
2. Check RDSICore 1.0 Type Library.

To reference the type library into a C++ project:

1. Click **Tools > OLE/COM Object Viewer**
2. In the OLE/COM Object Viewer dialog, click **File > Bind To File**
3. In the Open dialog, click **Rational/common/RDSICore.dll**

Using RSE 3

This chapter provides a working example of using the client interfaces of RSE. It shows how you locate and retrieve data with an RSE client application from an integrated product in the Suite. The actual examples used in this chapter are:

- TestFramework

  A Visual Basic RSE client application

- Rose adapter

  The RSE Rose adapter that maps Rose objects to RSE artifacts

- Rational Rose (ordersys.mdl)

  Rose is the integrated product used in these examples. Ordersystem is a sample application included in the Rose samples directory. Ordersys.mdl is the model used for retrieving actual instances of Rose objects.

Supported Client Use Cases

RSE provides client interfaces for locating and retrieving information, including:

- Navigating among artifacts

  Navigating through an artifact hierarchy. Given one artifact, finding others in some way, using artifact types and relationships.

- Retrieving static and dynamic metadata

  Static artifact types allow you to understand what types of objects are available and what properties and relationships they contain, without having an integrated product root object. For example, you can receive the collection of static artifact types for the ReqPro adapter without having an actual RequisitePro Project.

  The dynamic metadata is created after you load the root artifact. For example, once you locate a ReqPro Project, the collection of dynamic (user-defined) Requirement types for this Project are added to the Project artifact and created with dynamic properties and relationships.

While static artifact type definitions are global to each integrated product, dynamic types are local to a specific instance of a user-defined object in an integrated product.

- Locating artifacts

  The ability (of the IArtifactLocator interface) to generically find and load artifact information, given a set of arguments.

- Getting and setting properties

  Artifact properties map to the integrated product attributes of the objects.

RSE also enables you to perform more advanced functions including:

- Using Relationships

  This is a building block for the capability of finding artifacts. You can get a collection of related artifacts using relationships.

- Querying

  Querying or filtering is the ability to get related artifacts, given a certain condition. For example, get all related artifacts whose name = 'User', or whose stereotype = 'actors', or only artifacts whose artifact type = 'Class'. These examples return a subset of artifacts of a certain relationship. You can also sort them by name or type.

  Note: Sorting is not currently implemented.

- Supporting persistent references

  Given a locator, you can get a persistent artifact ID. This is the immutable ID string representation of an artifact locator.

- Creating artifacts

  The ability to create an integrated product object from an RSE client application. The client application does this by creating the artifact that maps to integrated product object. For example, creating a RequisitePro Requirement by creating an RSE Requirement artifact through a client application.

- Launching integrated products

  An artifact can open an integrated product and display that artifact's associated object. For example, in Rose, an RSE Class Artifact can open Rose and show its **Class Specification** box.

- Obtaining product-specific interfaces

The ability to get an integrated product's COM object. Getting this internal object (the integrated product object) programmatically is an alternative to launching the integrated product. For example, a client application getting an internal object from RequisitePro through the ReqPro adapter returns the integrated product's object (not the RSE artifact) as if it was from RequisitePro's COM server.

The COM interface to a RequisitePro requirement is no different than the COM interface to an artifact, it just has a different set of properties. Conceptually, this is similar to type casting. For example, given an artifact that is a Requirement, get the actual Requirement.

- Managing objects

  Object (artifact) support is through the integrated product RSE adapters. The RSE adapters manage the artifacts that map to the objects in the integrated products. The RSE core manages which adapters are installed and running.

  RSE provides adapter interfaces to manage the entire artifact creation, construction and deployment process. For information on the adapter interfaces see the *Programmer's Guide to Adapter Development* manual.

This chapter provides descriptions and details for implementing some of these tasks. For additional information see the "Creating RSE Clients" chapter of this manual.

For specific information on each adapter's property types and the underlying property data types in the integrated product, see the *Adapters Reference* manual or consult each product's extensibility user manual and extensibility reference manual.

Client Access

The client application in this chapter retrieves data from the order system model in Rose (ordersys.mdl in the samples directory). It uses instances of artifacts in the RSE Rose adapter that map to Rose (the integrated product) objects. This demonstrates how an RSE client application:

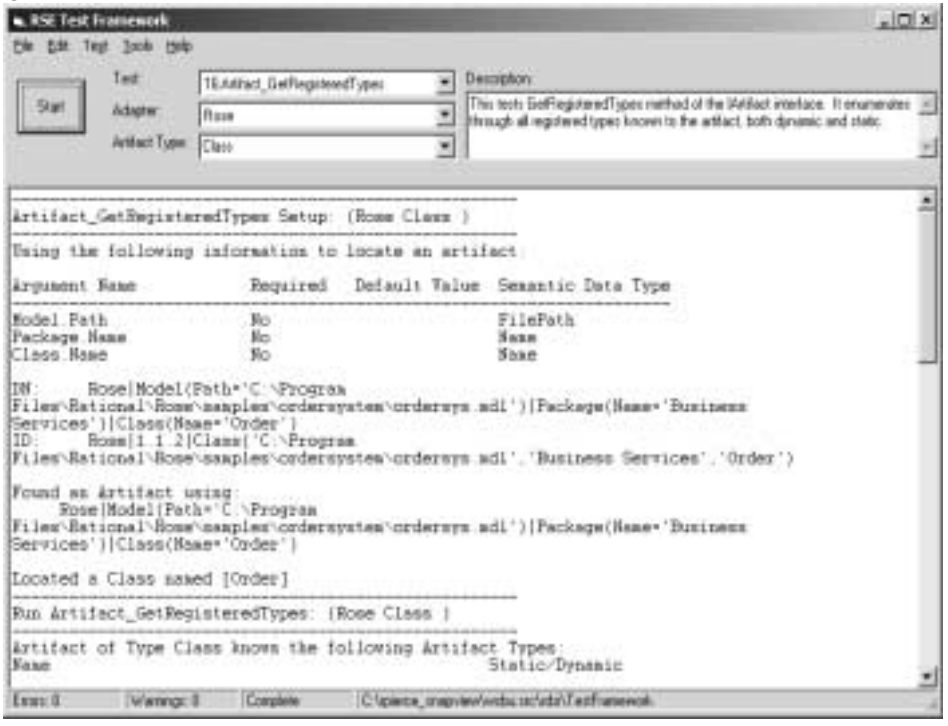
- Finds an artifact type
- Locates an artifact
- Retrieves properties of an artifact
- Gets related artifacts
- Displays artifacts
- Creates an artifact
- Converts artifacts to product-specific interfaces

TestFramework

The TestFramework client application performs common retrieval operations, using methods from the RSE client interfaces. Figure 8 shows the user interface of the TestFramework application. The drop-down boxes allow you to click:

- Test – the test you plan to run
- Adapter – the RSE adapter for the integrated product you want to test
- Artifact Type – the artifact type you want to work with

Figure 8 RSE TestFramework



Rose Ordersystem Model

As the “RSE Object Model” chapter of this manual describes, all integrated product data maps to artifacts, locators, properties, and relationships in RSE adapters. For example, in Rose, there are models (mdl files), packages, and classes. Model, Package, and Class are artifact types defined in the Rose adapter. A Rose class contains information that is converted to generic objects that a client application can retrieve.

For the Order System model:

- The path of the model is Rational/Rose/samples/ordersystem/ordersys.mdl
- The name of the model is ordersys.
- The model contains these packages: User Services, Business Services, and Data Services.

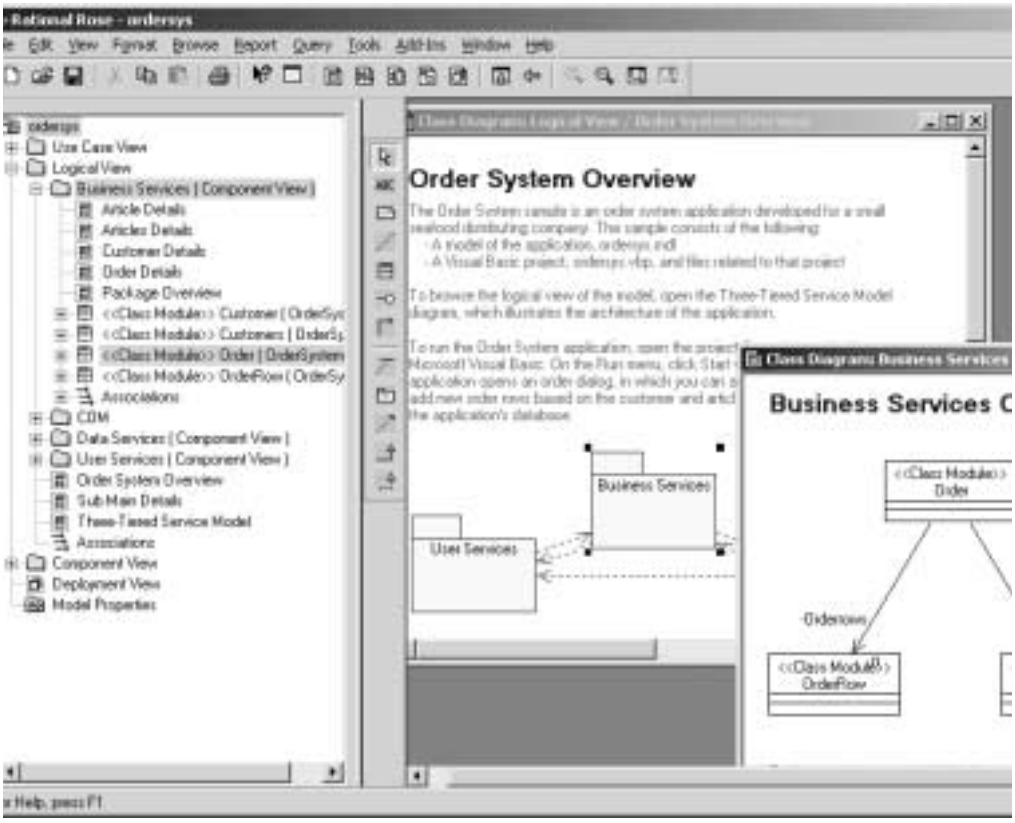
  These packages are represented through the Rose adapter as artifacts. The artifacts have an artifact type of Package.

  The Business Services package contains these classes: Order, OrderRow, Customer, and Customers. These classes are represented as artifacts of type Class.

  Each of these classes contains properties that are represented as class artifact properties in the Rose adapter. For example, in the Rose adapter, Class artifacts include a defined property called Documentation.

Figure 9 shows the Rose Order System model with the Business Services package and the Order class diagrams displayed.

Figure 9 The Rose Ordersystem Model



Model, Package, and Class are Rose objects that are mapped to RSE artifacts. Figure 10 shows the mapping of these Rose objects to the RSE object model. This mapping is implemented in the Rose adapter.

The adapter includes static definitions for Model, Package, and Class artifact types. Ordersys.mdl, Business Services, and Order are actual instances of Rose objects.

As Figure 10 illustrates, the mapping between Rose object and RSE artifact for these artifact types is:

- An .mdl file is an artifact of artifact type Model.

  Ordersys.mdl is an actual instance of a Model.

- A package is an artifact of type Package.

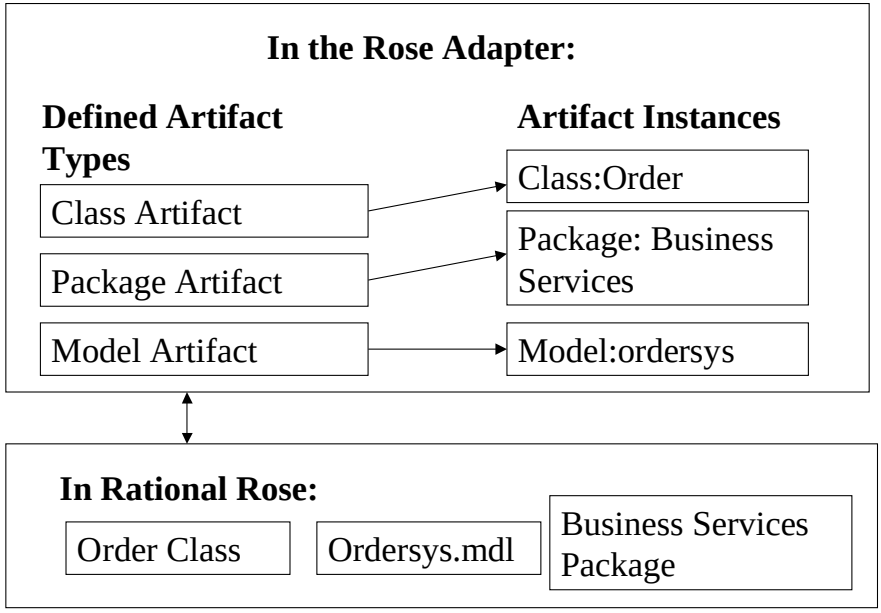
  Business Services is an actual instance of a Package.

- A class is an artifact of type Class.

  Order is an actual instance of a Class.

Each artifact type defines its properties and relationships. It also includes definitions for locators that can locate each artifact or a collection of artifacts of a certain type. The RSE client application uses the RSE core to gain access to this underlying structure, defined in each integrated product RSE adapter.

Figure 10 Mapping Artifacts to Objects



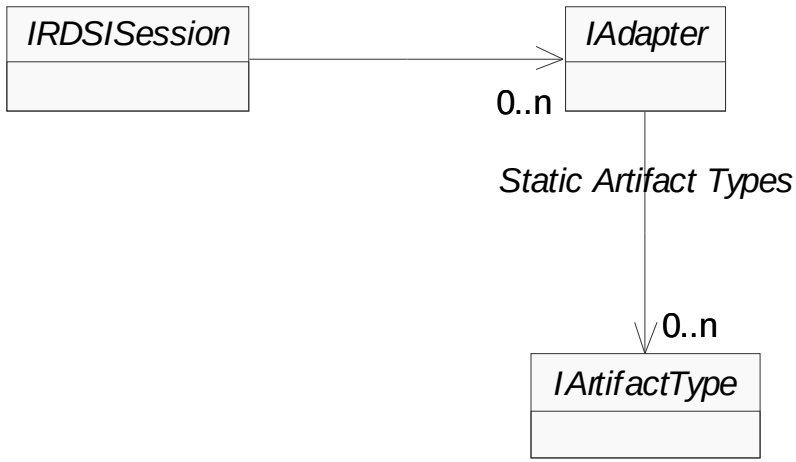
Finding an Artifact Type

You can find an artifact type as follows:

- Specifying an adapter and calling StaticArtifactTypes to retrieve the collection of artifact types available.
- Enumerating through the static artifact type collection and selecting an artifact type.

Figure 11 shows the object model for finding an artifact type.

Figure 11 Finding an Artifact Type



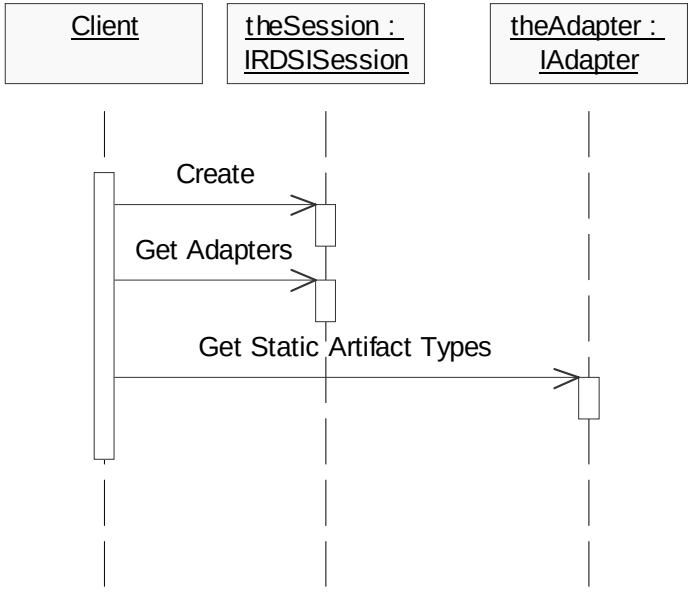
As the sequence diagram for finding an ArtifactType in Figure 12 illustrates:

- The Client creates a session.
- Given a session, you can enumerate the adapters to find a specific adapter. All installed adapters are available to the session.
- Given an adapter, you can retrieve specific artifact types supported by the adapter.

The Rose adapter defines 47 artifact types. When you select the Rose adapter, all artifact types supported by the Rose adapter become available. This set of static artifact types is available in the Rose adapter ArtifactType collection.

When you instantiate objects, some artifacts have information in them that extends what is available. These are dynamic artifact types. For example, in addition to the static artifact types defined in the RequisitePro adapter (ReqPro), when you open a Project (Project artifact type), this Project defines requirement types. These requirement types become available as additional artifact types.

Figure 12 Client Interfaces for Finding an ArtifactType



IAdapter is the object that runs when you select a specific adapter. It is an instance of an RSE adapter.

The list of artifact types that can be returned is constructed dynamically. For example, if you create a new RSE adapter for a Project Planner application, install it from the client end and select this new adapter, all of the adapter's artifact types appear.

Finding an ArtifactType Test

Figure 13 shows the TestFramework setup for finding the artifact types for a Rose Model:

- The Adapter drop-down list is initialized using the IRDSISession interface.
- The ArtifactType drop-down list is initialized using the IAdapter interface.

Figure 13 Finding an ArtifactType

The TestFramework returns all the static artifact types registered by the current adapter. It creates a new session, gets the adapters, and gets the static artifact types from the selected adapter.

Code for Finding an ArtifactType

The Test Framework uses the following code to fill the adapter combo box with the names of each adapter installed on the system:

```
Set m_Session = New RDSISession
Dim theAdapter As Adapter
For AdapterID = 0 To m_Session.Adapters.Count – 1
   Set theAdapter = m_Session.Adapters.Item(AdapterID)
   RDSIAdapterList.AddItem theAdapter.Name
Next AdapterID
```

Given an adapter, you can get the collection of artifact types defined by the adapter. The Test Framework uses the following code to populate the artifact type list.

```
Dim theAdapter As Adapter
Dim TypeID As Integer
Dim theArtifactType As ArtifactType
Dim ArtifactTypeList As ArtifactTypeCollection
Set theAdapter = GetCurrentAdapter()
If Not theAdapter Is Nothing Then
   For TypeID = 0 To theAdapter.StaticArtifactTypes.Count–1
      Set theArtifactType =
             theAdapter.StaticArtifactTypes.Item(TypeID)
      ArtifactTypeList.AddItem theArtifactType.Name
   Next TypeID
End If
```

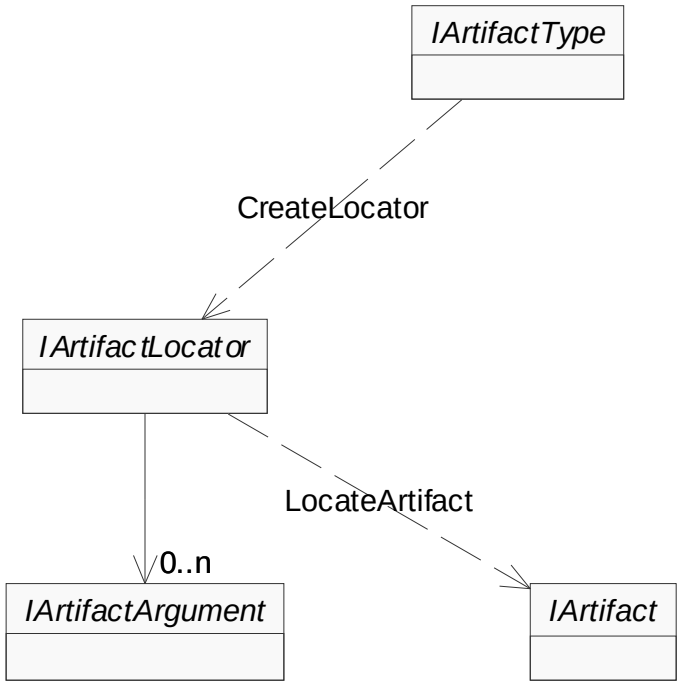
For more information on artifact types, see the “Creating RSE Clients” chapter of this manual.

Locating an Artifact

To locate a specific artifact, you must first specify the artifact type. Given the type, you can then construct a locator with the correct artifact arguments as described by the type. You provide values for the artifact arguments. These arguments represent the artifact reference. Given the artifact reference, the locator returns an instance of an artifact.

The object model for locating an artifact is in Figure 14.

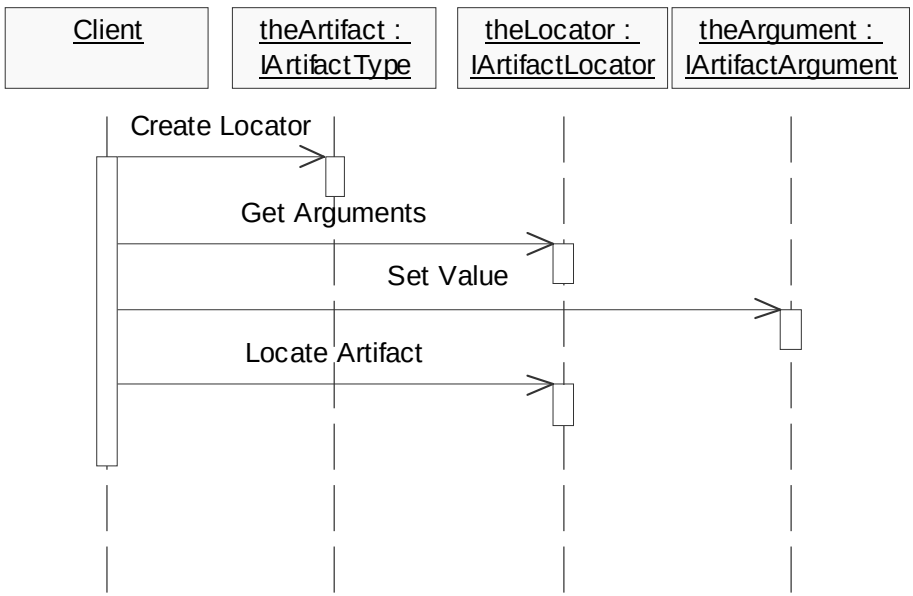
Figure 14 Objects for Locating an Artifact



As the sequence diagram for locating an Artifact in Figure 15 illustrates:

- Given an ArtifactType, use the CreateLocator method to get a locator (IArtifactLocator) for that object. Get the list of arguments for that locator type and provide argument values (IArtifactArgument) to the locator.
- Given an ArtifactLocator, use the LocateArtifact method to locate the artifact. Given the locator, you can provide values for its arguments to find an artifact. Locating an artifact returns an instance of an artifact.

Figure 15 Client Interfaces for Locating an Artifact



The following example shows how the Test Framework locates the Order class in the Rose Order System model using the client interfaces shown in Figure 15.

The Order class is located and returned through the Rose adapter. The Rose class object maps to an instance of a class artifact in the Rose adapter.

Figure 16 shows the Order class specification in Rose.

Figure 16 Order Class

The objects used in this example are:

- ArtifactType

  This example locates the class named Order. Its ArtifactType is named “Class”.

- ArtifactLocator

  The locator object for a Rose Class has three arguments. To locate the Order Class, you need the path of the model, the name of the Package, and the name of the Class.

- ArtifactArgument

The artifact arguments specify the information needed to locate an artifact. For example, in Rose the arguments are specific attributes in Rose and each is an ArtifactArgument.

- The path of the Model

  C:\Program Files\Rational\Rose\samples\ordersystem\ordersys.mdl

- The name of the Package containing the class

  Business Services

- The name of the Class

  Order

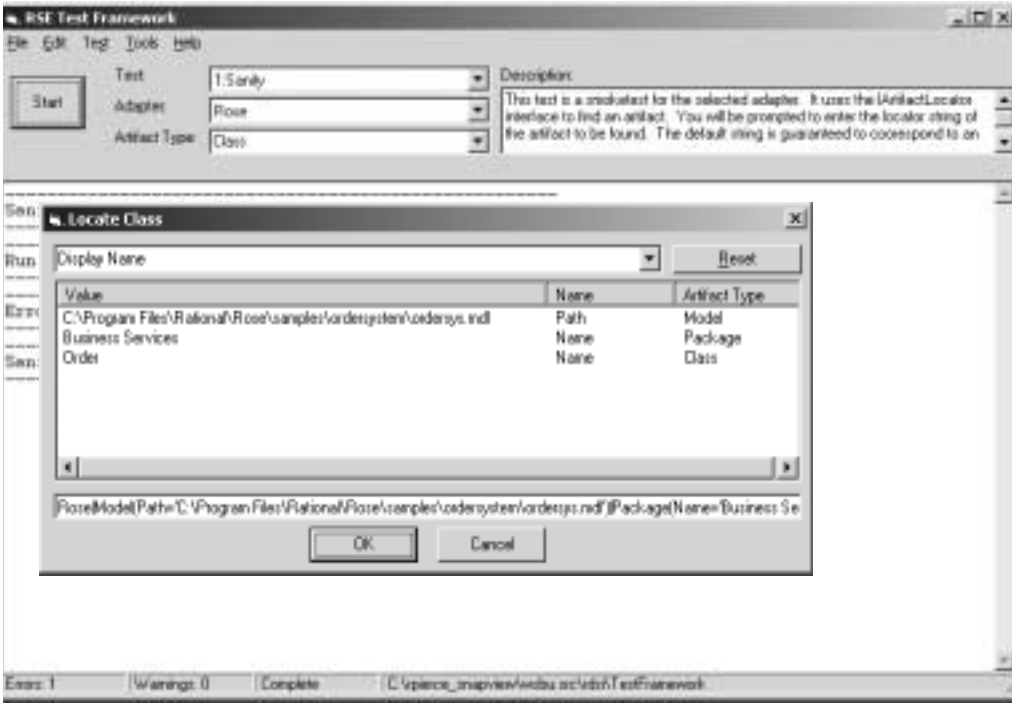
- Artifact

  Represents the Class named Order in Rose.

Locating an Artifact Test

Figure 17 shows the TestFramework Sanity test for locating an artifact. When you click **Start** in TestFramework, a **Locate Class** box appears.

Figure 17 Running the Sanity Test to Locate an Artifact



In this box, you can enter the three ArtifactArgument parameters for path (C:\Program Files\Rational\Rose\samples\ordersystem\ordersys.mdl), package (Business Services) and class (Order). Click OK and the TestFramework returns a message that it found the Order class in Rose.

As another implementation possibility, you can also display a property page or table and a user can enter the values to find a particular artifact in an integrated product.

Code for Locating an Artifact

The TestFramework application uses the following code to initialize the locator dialog box used by the Sanity test. It uses the ArtifactType to create a locator. Given the artifact type, two locators are created for the dialog box, a display name locator and an immutable ID locator.

The display name locator contains information that is understandable to the user. The immutable ID locator contains a persistent reference to an artifact that is understandable to the system.

In this example:

- `theType` is an instance of a Rose Class ArtifactType.
- `m_Locator` is a locator for the Class ArtifactType.

```
Private Sub Initialize(theType As ArtifactType)
   Set m_Locator = theType.CreateLocator()
End Sub
```

The following code is used by the TestFramework to initialize the locator arguments list in the locator dialog box.

```
Private Sub UpdateControls()
  Dim theArg As ArtifactArgument

  For ArgID = 0 To m_Locator.Arguments.Count - 1
    Set theArg = m_Locator.Arguments.Item(ArgID)
    Set theListItem = ListView.ListItems.Add(theArg.Value)
    theListItem.SubItems(1) = theArgument.ArgumentName
    theListItem.SubItems(2) = theArgument.ArtifactTypeName
  Next ArgID
End Sub
```

The following code saves the values in the locator dialog box by setting the ArtifactArgument values belonging to an ArtifactLocator. It iterates through the locator's arguments, getting each argument from the collection of arguments and puts each argument name, type, and argument value into the dialog box. Different objects may contain different information but the same piece of code is initialized in this dialog box for all integrated products.

```
Private Sub OKButton_Click()
    Dim theArguments As ArtifactArgumentCollection
```

```
    Dim theArgument As ArtifactArgument

    For ItemID = 0 To ListView.ListItems.Count - 1
        Set theListItem = ListView.ListItems.Item(ItemID + 1)
        Set theArgument = m_Locator.Arguments.Item(ItemID)
        theArgument.Value = theListItem.Text
    Next ItemID
End Sub
```

TestFramework uses the following code to locate an Artifact. The LocateArtifact method of the IRDSISession interface returns an instance of Order (the Rose Order class).

In this example:

- `theLocator` is a locator for the Class ArtifactType returned by the locator dialog.
- `LocateAnArtifact` is the return value for the function. This is the Rose Class Artifact named Order.

```
Function LocateAnArtifact(bPrintLocator As Boolean) As Artifact
    Dim theLocator As ArtifactLocator

    Set theLocator = GetALocator(True)
    Set LocateAnArtifact = theLocator.LocateArtifact()
End Function
```

LocateArtifact coordinates with the individual products in Rational Suite, each with its own API but with RSE the above piece of code accomplishes this search mechanism for all integrated products.

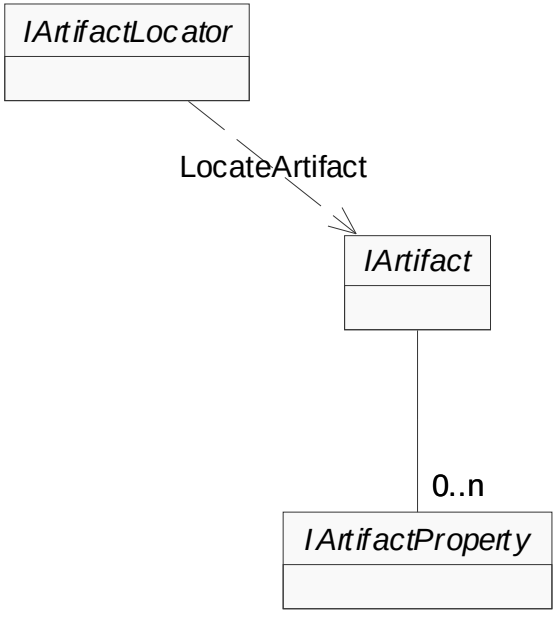
For more information on artifact types, see the “Creating RSE Clients” chapter of this manual.

Retrieving Properties of an Artifact

You can retrieve the properties of an artifact after you have located the artifact by calling the IArtifact Properties method. After you retrieve the collection of properties, you can get or set values of specific properties.

Figure 18 shows the object model for retrieving the properties of an artifact.

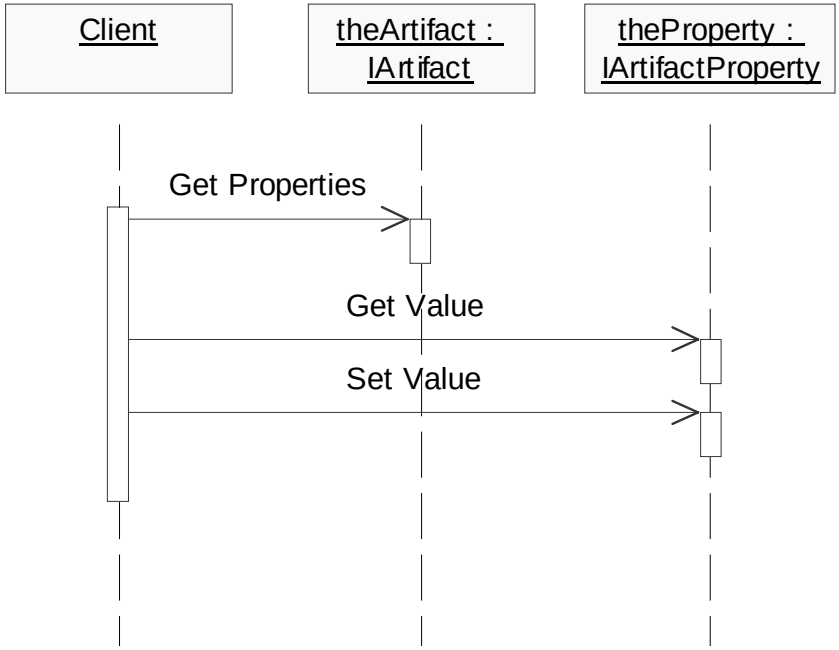
Figure 18 Object Model for Retrieving Properties



As the sequence diagram in Figure 19 shows:

- Given a specific artifact that has been located, you can then access that artifact's properties.
- The artifact has a collection of properties, and each property has a value that can be accessed. (Note: Some properties are not modifiable)

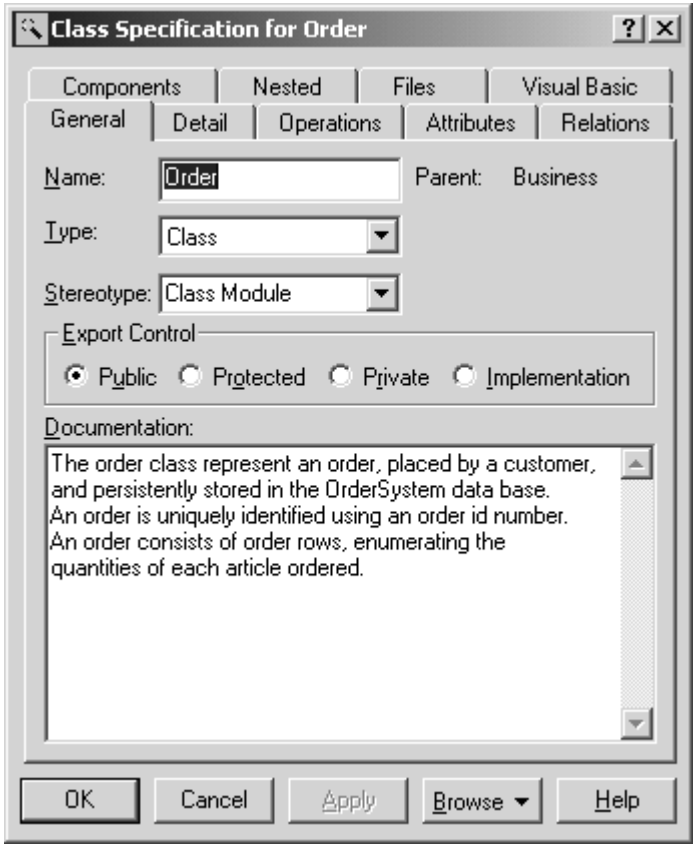
Figure 19 Retrieving Properties



The following example retrieves the properties of the Order class in the Rose Order System model.

Figure 20 shows the Order class specification in Rose.

Figure 20 Order Class Specification



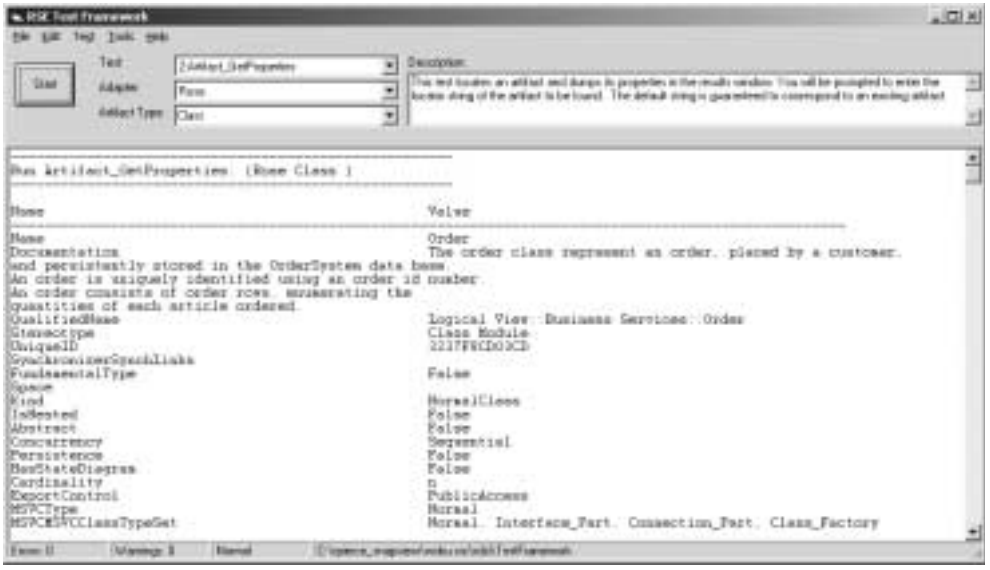
Retrieving Properties of an Artifact Test

Figure 21 shows the TestFramework Get_Properties test for retrieving the properties of the Order class in Rose. When you click **Start** in the TestFramework, a **Locate Class** box appears. The Locator arguments are:

- Path of the Model (C:\Program Files\Rational\Rose\samples\ordersystem\ordersys.mdl)
- Name of the Package (Business Services)
- Name of the Class (Order)

Click OK and the TestFramework returns the properties in the Order class in Rose.

Figure 21 Running the Artifact_GetProperties Test



Retrieving the Value of a Property

The name of the second listed property is Documentation. The value for Documentation is the text string that includes the Documentation text.

In Rose, in the **Class Specification for Order** box, there is a General tab. At the bottom of this box, is the Documentation property. This is defined as a Property in the Rose adapter of data type Text. You can enter additional text in this section. In this example the extra text 'We were here!' is added. Click Apply.

To retrieve this from the TestFramework, select the PropertyGet_Basic test for a Rose Class. Click **Start** and **OK** on the **Locate Class** box. Scroll down to the Documentation property and the Value includes the 'We were here!' at the end of the text string.

Code for Retrieving Properties of an Artifact

This method prints the values of all the properties of a given artifact (theArtifact). `theArtifact` is the Rose Class Artifact named Order.

```
Sub PrintArtifactProperties(theArtifact As Artifact)
    Dim theProperties As ArtifactPropertyCollection
    Dim theProperty As ArtifactProperty

    Set theProperties = theArtifact.Properties
    PrintResults 0, "Artifact Type: " + theArtifact.Type.Name
    For i = 0 To theProperties.Count - 1
        Set theProperty = theProperties.Item(i)
        PrintResults 1, theProperty.Type.Name + " = " + _
                        CStr(theProperty.Value) + _
                        GetSemanticTypeName(theProperty.Type)
    Next i
End Sub
```

For more information on retrieving properties, see the "Creating RSE Clients" chapter of this manual.

Getting Related Artifacts

You can get the related artifacts of an artifact by specifying an artifact type and a relationship type. The object types and the instances used in this section are:

- Artifact

  The Business Services Package.

- ArtifactType

  A Rose Package

- RelationshipType

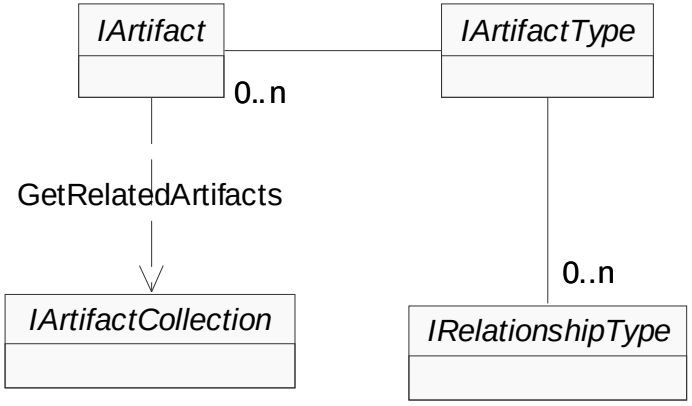
  The Classes relationship type.

- ArtifactCollection

  The collection of Class Artifacts contained in the Business Services Package.

Figure 22 shows the object model for getting related artifacts.

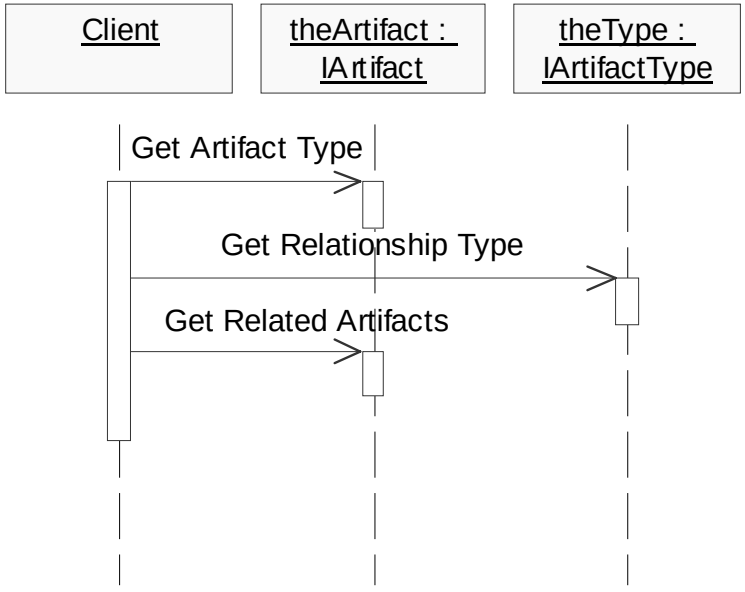
Figure 22 Getting Related Artifacts Object Model



As the sequence diagram in Figure 23 shows:

- Given an artifact, you can get its type.
- Given an ArtifactType, you can ask for it to get a specific relationship. RelationshipType is the object that represents a relationship.
- Given an artifact, you can get the collection of artifacts (IArtifactCollection) that is related to an artifact.

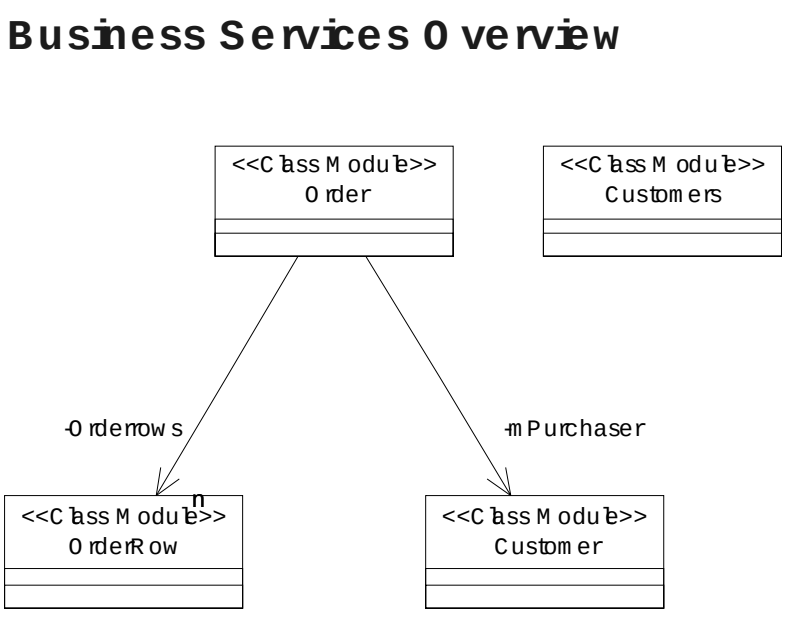
Figure 23 Getting Related Artifacts



This example retrieves the related artifacts in the Business Services package in the Order System model.

Figure 24 shows the classes in the Business Services package in Rose.

Figure 24 Business Services Package Overview

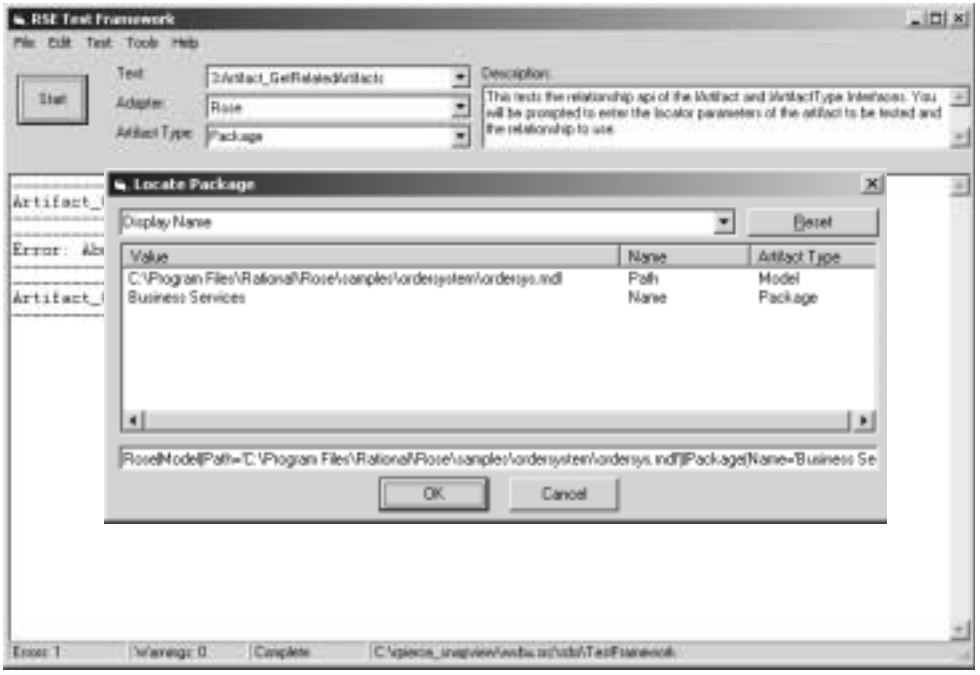


Getting Related Artifacts Test

Figure 25 shows the TestFramework GetRelatedArtifacts test for retrieving the collection of related artifacts. In this example, the test is done for the Business Services package in the Order System model. When you click **Start** in the TestFramework, a **Locate Package** box appears. The Locator dialog arguments are:

- Path of the Model (C:\Program Files\Rational\Rose\samples\ordersystem\ordersys.mdl)
- Name of the Package (Business Services)

Figure 25 Running the GetRelatedArtifacts Test



Click **OK** and a **Choose Relationship** box appears.

Select Classes. Click OK. The TestFramework returns the collection of artifacts for the Classes relationship in the Business Services package. This includes the classes in the Business Services package shown in Figure 24:

- Order
- OrderRow
- Customer
- Customers

Code for Getting Related Artifacts

This is the method that enumerates and prints the name and type of each Artifact related to `theArtifact` by `theRel`.

- `theArtifact` is the Rose Package Artifact named Bank.
- `theRel` is the Classes Relationship.

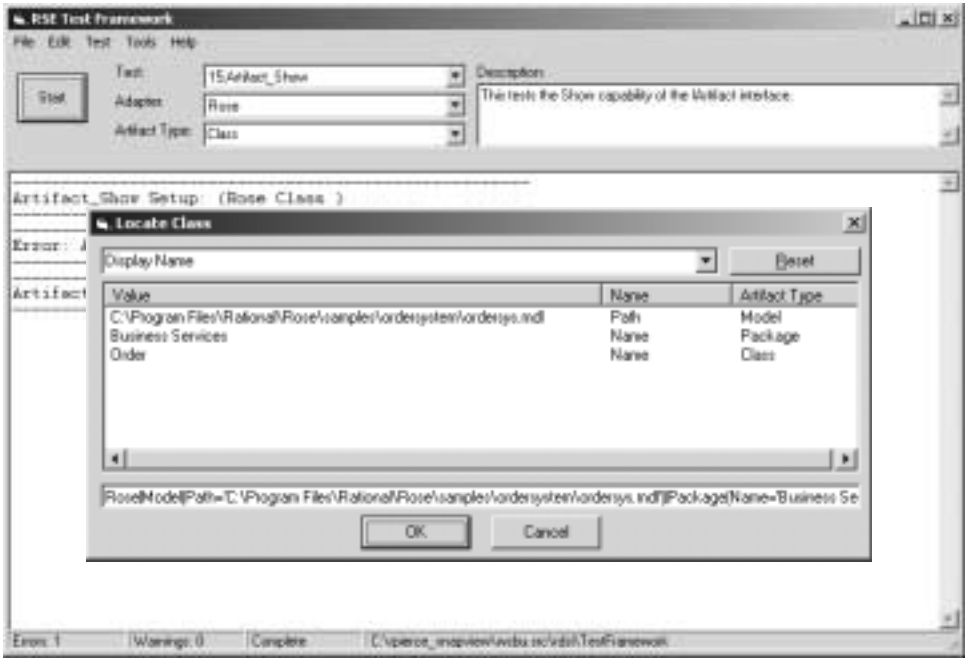
```
Sub PrintRelatedArtifacts(theArtifact As Artifact, _
                          theRel As RelationshipType)
  Dim theRelArtifacts As ArtifactCollection
  Dim theRelArtifact As Artifact

  Set theRelArtifacts = theArtifact.GetRelatedArtifacts(theRel)
  For ArtifactID = 0 To theRelArtifacts.Count - 1
     Set theRelArtifact = theRelArtifacts.Item(ArtifactID)
      PrintResults 2, theRelArtifact.Type.Name + _
                     " named [" + theRelArtifact.Name + "]"
   Next ArtifactID
End Sub
```

Displaying Artifacts

You can invoke an integrated-product GUI by specifying an artifact and using the Show method of the IArtifactGUI interface. This example uses the Order class in the Business Services package of the ordersys model.

Figure 26 shows the TestFramework Artifact_Show test. The selected adapter is Rose and the ArtifactType is Class. Click **Start** and the **Locate Class** box appears. The arguments are the path of the model, the package and the class.

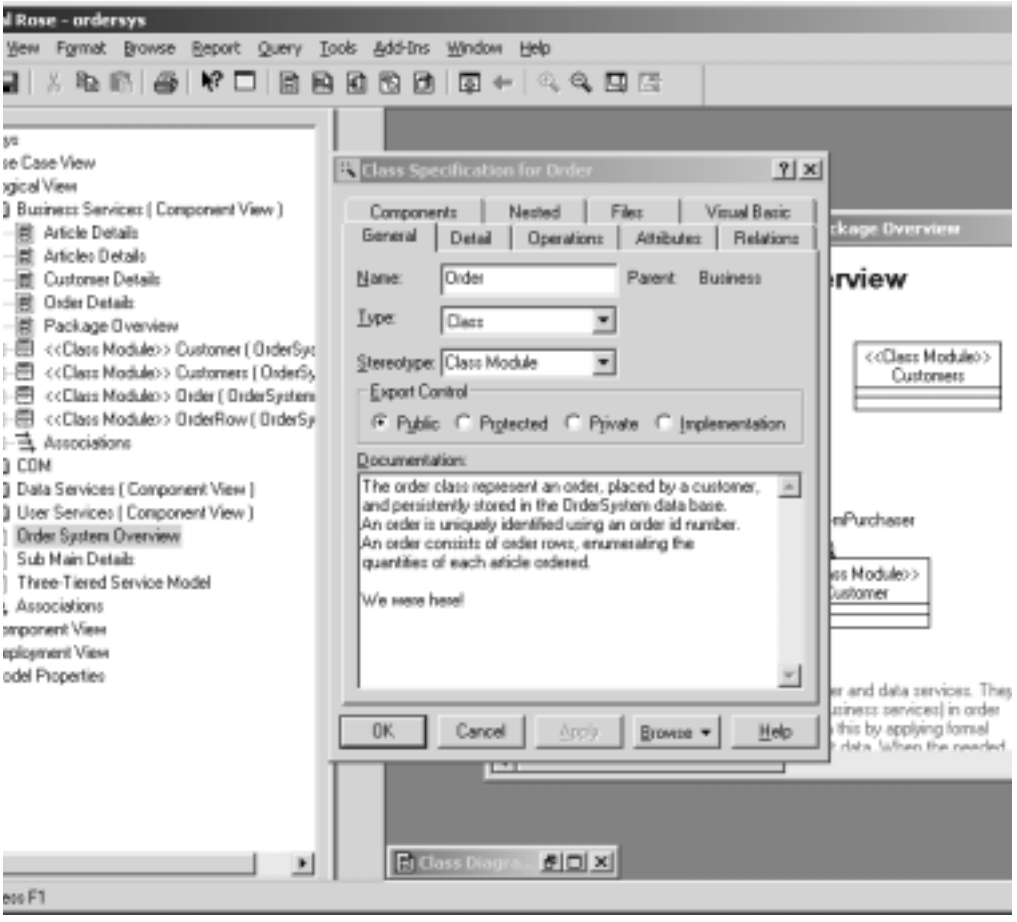
Figure 26 Running the Artifact_Show Test



Click OK and the Show method launches Rose and opens the Class specification for Order, as shown in Figure 27.

IArtifactGUI.Show instructs the given artifact to make its GUI visible.

Figure 27 Displaying the Integrated-Product GUI



Code for Displaying an Artifact

Given an artifact, invoke the GUI by calling the Show method of the IArtifactGUI interface.

m_ContextArtifact is the Rose Package Artifact named Bank.

```
Private Sub Test_RunTest()
```

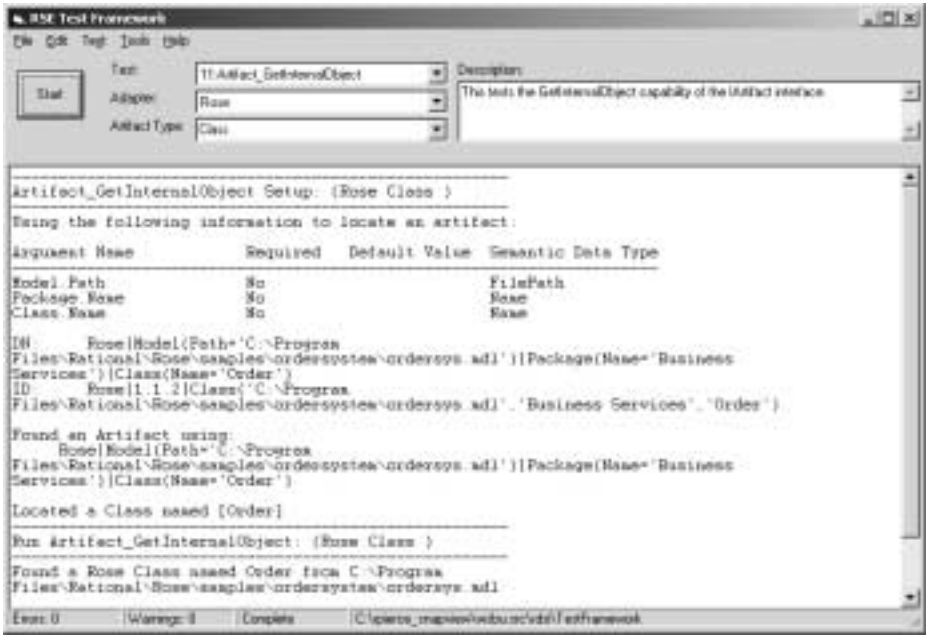
```
   If m_ContextArtifact.GUI.CanShow() Then
      PrintResults 0, "Artifact can show...”
      m_ContextArtifact.GUI.Show
   Else
      PrintResults 0, "Artifact can't show...”
   End If
End Sub
```

Getting the Internal Object

You can convert artifacts to product-specific interfaces using the GetInternalObject method of the IArtifact interface. This is useful for integrating RSE and specific integrated-product extensibility APIs.

Figure 28 shows the GetInternalObject test for the Rose class named Order.

Figure 28 Running the Artifact_GetInternalObject Test



This method converts an artifact into a product-specific interface using the GetInternalObject method. If a RoseItem is returned, Rose Extensibility Interface supplied values are found and the test prints out the REI class name, the name of the item and the name of the model.

m_ContextArtifact is the Rose Package Artifact named Model.

Code that represents a Rose object is in bold print. (This is the code that is different for each integrated product API.)

```
Private sub test_RunTest()
   Dim theObject as object
   Set theObject = m_ContextArtifact.GetInternalObject()
   If not theObject is nothing then
      Dim theItem as RoseItem
      Set theItem = theObject
      If not theItem is nothing then
         PrintResults 0, “Found a Rose " & _
            theItem.IdentifyClass() & " named " & _
            theItem.Name & " from " & theItem.Model.Name
      End if
   End if
End sub
```

Using the Display Name ID for a Locator:

Rose|Model(Path='C:\Program Files\Rational\Rose\samples\ordersystem\ordersys.mdl')|Package(Name='Business Services')|Class(Name='Order')

the test returns:

```
Found a Rose Class named Order from
C:\Program Files\Rational\Rose\samples\ordersystem\ordersys.mdl
```

For information on an integrated-product’s extensibility see that product’s documentation. For example, for Rose, see the *Rose Extensibility User’s Guide*.

Creating RSE Clients

4

This section demonstrates how RSE can be used to perform some typical client application functions. Excerpts of source code are included to show specifically how RSE can be used to implement client applications. For the purposes of simplicity, the only integrated products discussed in this chapter are Rose and RequisitePro. The examples provided are compatible with other products.

- Locating Artifacts

  Describes the methods for locating artifacts and provides information on artifact IDs and artifact locator objects. There are also examples covering authentication and error handling.

- Getting and Setting Properties

  Describes how to get and set an artifact's property values.

- Getting Related Artifacts

  Provides information on Query interfaces for getting a collection of artifacts, given an artifact.

- Displaying Artifacts

  Describes the Artifact Show method for an artifact launching an integrated product and opening its associated object.

- Converting Between Artifact Object and Internal Object

  Covers the compatibility between Rational Suite Extensibility and integrated product extensibility APIs.

- Creating and Deleting Artifacts

Locating Artifacts

The methods for locating artifacts are:

- Using an artifact locator
- Using an artifact reference

  You can locate an artifact directly, given an artifact ID. You can use the LocateArtifact method if you have an artifact ID from a previous session and you want to relocate an artifact.

Both methods require an artifact ID. This Artifact ID is an absolute ID and maps to the root artifact for a given adapter. With this root artifact, you can also create relative IDs using the CreateRelativeArtifactID method to store references to other artifacts. These relative IDs can also be used to locate artifacts.

For locating an artifact:

- Use ArtifactType.CreateLocator when prompting a user for data.
- Use LocateArtifact when using an artifact ID from a previous session.

Using an Artifact Locator

The primary method for locating an artifact (as described in Chapter 3) is to select an adapter and get the collection of static artifact types for that adapter. You then enumerate through the types, select a type, and create an artifact locator for this type.

Given the locator, you can then set values for the locator arguments and locate an artifact.

To locate an artifact using an artifact locator, you:

- Create an artifact locator
- Initialize the locator arguments
- Locate the artifact

Creating an Artifact Locator

The following code creates a locator for the currently selected artifact type. It creates the locator with these argument values.

```
Function GetLocator (AdapterName As String, ArtifactTypeName As
String) As ArtifactLocator
   Dim theRDSISession As New RDSISession
   Dim theAdapter As Adapter
   Dim theType As ArtifactType
   Dim theLocator As ArtifactLocator
Set theAdapter = theRDSISession.Adapters.Item(AdapterName)
Set theType = theAdapter.StaticArtifactTypes.Item (ArtifactTypeName)
Set theLocator = theType.CreateLocator
(LocatorType:=rsLocator_Display_Name)
Set GetLocator = theLocator
End Function
```

Initializing Locator Arguments

The IArtifactLocator interface enumerates the artifact arguments for a given locator type. You can set values for these arguments or provide a GUI that allows a user to set these values.

The following is an example of the locator arguments defined in an artifact locator that locates a Rose Class. Table 1 lists the locator arguments (with their names and values) in this example.

Table 1 Locator Arguments

ArtifactType	Argument Name	Value
Model	Path	C:\Ordersys.mdl
Package	Name	Business Services
Class	Name	Order

The Locator.Arguments method returns the collection of arguments needed for a given locator. The following code example sets the values for the arguments to create a locator.

Using the Rose Class locator arguments above, you could call SetLocatorArg as follows:

```
Sub InitializeRoseClassLocator (theLocator As ArtifactLocator)
   SetLocatorArg(theLocator, “Model”, “Path”, “C:\Ordersys.mdl”)
   SetLocatorArg(theLocator, “Package”, “Name”, “Business Services”)
   SetLocatorArg(theLocator, “Class”, “Name”, “Order”)
End Sub
Sub SetLocatorArg (theLocator As ArtifactLocator,
                   ArtifactTypeName As String,
                   ArgumentName As String,
                   Value As Variant)
   Dim theArguments As ArgumentCollection
   Set theArguments = theLocator.Arguments
   For i = 1 to theArguments.Count
      If theArguments[i].ArtifactTypeName = ArtifactTypeName And
      theArguments[i].ArgumentName = ArgumentName Then
      theArguments[i].Value = Value
      End If
      Next i
End Sub
```

In this example the:

- ArtifactTypeName arguments are Model, Package, and Class.
- ArgumentName arguments are Path, Name (the Package name), and Name (the Class name).
- Value arguments are C:\Ordersys.mdl, Business Services, and Order.

In addition to allowing argument values to be enumerated and changed, the artifact locator interface supports optional arguments and default values. This allows capabilities for projects to be located using usernames and passwords, but also allowing default access without specifying user information. For more information, see *Authentication and Exception Handling* on page 75.

Locating an Artifact

Given the locator arguments, you call the ArtifactLocator.LocateArtifact method to get the artifact. For example:

```
Dim theArtifact As Artifact
Dim theLocator As ArtifactLocator
Set theLocator = GetLocator (“Rose”,”Class”)
InitializeRoseClassLocator theLocator
Set theArtifact = theLocator.LocateArtifact()
```

Using an ArtifactID

An artifact ID is a string representation of an artifact locator. You can store these strings and use them later to create locators and relocate artifacts.

The following ArtifactID is the string representation of the Rose Class locator object:

```
Rose|Model(Path=<c:\Ordersys.mdl>)|
Package(Name='Business Services')|Class(Name='Order')
```

To locate an artifact using an ArtifactID, you first get an ArtifactID and then use it as an argument in the RDSISession.LocateArtifact method.

Getting an ArtifactID

You can get an ArtifactID:

- from an artifact locator, using ArtifactLocator.GetArtifactID
- from an artifact, using Artifact.GetDefaultArtifactID

With either of these methods, you can get the DisplayName or ImmutableID form of the ArtifactID.

- The display name form is the most descriptive and readable form of the information needed to locate an artifact.
- The ImmutableID is defined to be the best form for storing persistent references to an artifact.

Getting an ArtifactID from an Artifact Locator

Given an artifact locator, you can get an Artifact ID using the ArtifactLocator.GetArtifactID method. For example:

```
ArtifactID = theLocator.GetArtifactID (rsLocator_Display_Name)
```

You can also get the ImmutableID form:

```
ArtifactID = theLocator.GetArtifactID(rsLocator_Immutable_ID)
```

Getting an ArtifactID from an Artifact

Given an artifact, you can get an Artifact ID using the Artifact.GetDefaultArtifactID method. For example:

```
ArtifactID =
theArtifact.GetDefaultArtifactID(rsLocator_Display_Name)
```

You can also get the ImmutableID form:

```
ArtifactID =
theArtifact.GetDefaultArtifactID(rsLocator_Immutable_ID)
```

Locating an Artifact with an ArtifactID

Given an ArtifactID, you can locate the artifact with the RDSISession.LocateArtifact method.

In the following example, for locating a Rose Model, the ArtifactID is:

Rose|Model(Path='C:\Visual Models\myModel.mdl')

- The adapter name is `Rose`.
- The artifact type is `Model.`
- The path of the model is `C:\Visual Models\myModel.mdl`.

Clients use the RDSISession to establish a connection to the RSE core without any RSE object as its context.

```
Dim theArtifact As Artifact
Dim ArtifactID As String
Dim theRDSISession As New RDSISession
ArtifactID = "Rose|Model(Path='C:\Visual Models\myModel.mdl')"
Set theArtifact = theRDSISession.LocateArtifact (ArtifactID)
```

Using Relative Artifact IDs

A relative artifact ID is a reference to an artifact given another artifact. This enables simplified versions of the information needed to locate artifacts.

For example:

- The absolute ArtifactID to the Rose Order Class is:

  Rose|Model(Path=<C:\Ordersys.mdl>)|
  Package(Name='Business Services')|Class(Name='Order')

The absolute artifact ID includes all information for an artifact, including its location (path) and extra information such as username and flags.

- The RelativeArtifactID for this object, relative to Package is:

  ```
  Class(Name='Order')
  ```

  You can use this shorter reference, given the ArtifactID for the Package. In this example, Package is the context artifact.

Both forms of ID provide a method for storing references to artifacts that can be used to relocate the artifacts later. RelativeArtifactIDs allow less information to be stored.

Getting a RelativeArtifactID

You get a RelativeArtifactID by:

- Getting a context locator
- Getting the RelativeArtifactID from the context locator

Getting a Context Locator

To get a RelativeArtifactID, you first need a context locator.

For example, to get the RelativeArtifactID for a Rose Class, relative to Package, you first need the Package context locator. A context artifact provides the context to the lower level artifacts.

- Given an artifact, you can get an artifact locator. For example, given a Rose Class, you can get a Class ArtifactLocator object using the Artifact.CreateLocator method.
- Given an artifact locator, you can get a context locator. The locator object calls ArtifactLocator.ContextLocator to get a context locator object (for example a Rose Model ArtifactLocator object).
- From the context locator, you can get the context artifact ID by calling the Locator.GetArtifactID method.

This context artifact information is then used for getting a relative artifact ID.

Getting a RelativeArtifactID from a Context Locator

From the context locator, can get the relative artifact ID with the Locator.GetRelativeArtifactID(ContextLocator) method.

The GetRelativeArtifactID method returns a relative artifact ID that contains the arguments and property values relative to the context artifact. This ID is stored and can be used later.

The following code example shows the relative locator capabilities of the IArtifactLocator interface. It shows how you get a relative artifact ID from an object.

In this example, the context locator is the object one level up (for example, the context locator for Class is Package; the context locator for Package is Model).

```
Function GetRelativeArtifactID (theLocator As ArtifactLocator) As
                                                                String

   Dim theContextLocator As ArtifactLocator
   Set theContextLocator = theLocator.ContextLocator()
   GetRelativeArtifactID =
           theLocator.GetRelativeArtifactID(theContextLocator,
                                            rsLocator_Display_Name)

End Function
```

Given a relative artifact ID, you can create a relative locator to locate an artifact. You can also save both the ArtifactID and RelativeArtifactIDs in a file, close a client application, then open it later, and use the IDs by creating locators with these IDs (using the RDSISession.CreateArtifactLocator method) and locating the artifacts.

Locating an Artifact with a RelativeArtifactID

You can locate an artifact, given a context locator and a RelativeArtifactID. Given the relative ID, you can create a locator using the ArtifactLocator.CreateRelativeLocator method. You can use this relative locator to locate the artifact.

For example:

- The ArtifactID of the Package context locator is:

  Rose|Model(Path=<C:\Ordersys.mdl>)|
  Package(Name='Business Services')

- The RelativeArtifactID for the Class is:

  Class(Name='Order')

To locate a Rose Class, given a Package:

- Create a context locator, using the RDSISession.CreateArtifactLocator method. This is a locator to the Package, since Package is the context artifact.

  ```
  theContextLocator = theSession.CreateArtifactLocator(ContextID)
  ```

- Given the RelativeArtifactID, create a relative locator to the Class.

  The type of relative locator can be DisplayName or UniqueID.

  ```
  Set theLocator =
  theContextLocator.CreateRelativeLocator(RelativeArtifactID)
  ```

- Use the relative locator to locate the artifact

  ```
  Set theArtifact = theLocator.LocateArtifact()
  ```

You can also get the ArtifactID for the context locator and save both IDs, convert them back to locators, and use the locators to locate the artifacts.

In the ReqPro adapter, given a Requirement artifact, with the following locator arguments:

Name	Value
Project.Path	C:\ReqPro\Project 1\Project 1.rqs
Project.UserName	Admin
Project.Password	
Project.Flags	0
Requirement.FullTag	REQA1

you can locate the Requirement, with a RelativeArtifactID, relative to Project.

- The absolute ArtifactID for the Requirement is:

  ReqPro|Project(Path='C:\ReqPro\Project 1\Project 1.rqs',
  UserName='Admin',Password='',Flags='0')|Requirement(FullTag='REQA1')

- The RelativeArtifactID (Relative to the Project artifact type) is:

  Requirement(FullTag='REQA1')

Using the RelativeArtifactID, you create a Relative Locator from the Project to locate a REQARequirement.

The ArtifactID for Project is:

ReqPro|Project(Path='C:\ReqPro\Project 1\Project 1.rqs',
UserName='Admin',Password='',Flags='0')

You can use this ID and the Requirement RelativeArtifactID to locate the Project and the Requirement.

Authentication and Exception Handling

When locating artifacts, the following exceptions can be thrown:

- E_ACCESSDENIED Authentication failed

  This means that either an incorrect username or password was entered.

- E_PENDING Authentication required

  This means that a username and password are required.

If these exceptions are thrown when locating an artifact, you must resolve the locator by providing user login information. This includes a valid username and password.

The following exception handler code resolves access denied and authentication required errors. If this were a real GUI, then in either exception case, you would need to display a login dialog, find the artifact argument that has a data type of UserName or Password (get the UserName or Password from the locator), and prompt user for the information. Then, enter this information back into the locator.

```
Function LocatorUtils_DoLocateArtifact(theLocator As ArtifactLocator)
As Artifact
    On Error GoTo HandleError
    Dim theArtifact As Artifact


TryAgain:
    Set theArtifact = theLocator.LocateArtifact()


    If theArtifact Is Nothing Then
       PrintError 0, "LocateArtifact did not find an artifact using: "
+ vbCrLf + theLocator.GetArtifactID(rsLocator_Display_Name)
        Exit Function
    End If
    Set LocatorUtils_DoLocateArtifact = theArtifact
    Exit Function


HandleError:
    ' Authentication Failed
    If Err.Number = &H80070005 Then
        If DisplayLoginDialog (theLocator) = False Then
           Exit Function
        End If
        GoTo TryAgain
    End If
```

```
    ' Authentication Required
    If Err.Number = &H8000000A Then
        If DisplayLoginDialog (theLocator) = False Then
           Exit Function
        End If
        GoTo TryAgain
    End If

    PrintRuntimeError 0, "LocateArtifact did not find an artifact."
End Function

Function DisplayLoginDialog (theLocator As ArtifactLocator) As Boolean
        Dim theLoginDialog As New LoginDialog

        ' Enumerates the arguments of the locator, looking for
        ' the one with SemanticDataType of UserName
        LoginDialog.UserName = GetUserName (theLocator)
        ' Enumerates the arguments of the locator, looking for
        ' the one with SemanticDataType of Password
        LoginDialog.Password = GetPassword (theLocator)
        LoginDialog.Show 1, Me
        If LoginDialog.Canceled = True Then
               DisplayLoginDialog = False
        End If

        ' Sets the UserName of the locator
        SetUserName theLocator, LoginDialog.UserName
        SetPassword theLocator, LoginDialog.Password
        DisplayLoginDialog = True
End Function
```

Getting and Setting Properties

Given an artifact, you can retrieve actual property types, values, and names using methods of the IArtifact interface. Before you set values, you must first check the IArtifactPropertyType.SetAllowed method to verify that a property is allowed to be set.

Getting the Values of Properties

In this example, the code displays a property sheet containing the values of all of the properties of the Artifact.

The following example loads a property sheet. When the property sheet is initialized, the properties of the specified artifact are enumerated and added to the list control. This uses the Artifact's Properties member to enumerate through all of the properties for the Artifact. For each property, the name is added to column 0 and the value to column 1 of the list. The properties that are not allowed to be set are disabled.

```
Sub LoadProperties (theArtifact As Artifact)
   Dim theProperty As ArtifactProperty
   Dim PropID As Long
   ‘Enumerate all of the properties for the Artifact:
   For PropID = 0 To theArtifact.Properties.Count - 1
      Set theProperty = theArtifact.Properties (PropID)
      ‘Add a new row
      RowID = thePropertyList.AddRow ()
      ‘ Initialize with property info
      thePropertyList.Add RowID, 0, theProperty.Name
      thePropertyList.Add RowID, 1, theProperty.Value
      ‘Disable the list elements that cannot be modified:
      If theProperty.Type.SetAllowed () = False Then
         ThePropertyList.EnableRow RoseID, False
      End If
   Next PropID
End Sub
```

For applications that require a set of properties to be used on a large number of artifacts, there is an alternate approach that allows the code to execute with minimal performance overhead. Imagine a report generator that lists the name and documentation of each artifact in a collection. For large numbers of artifacts, the time

spent looking up the ArtifactProperty object in each artifact would add to the time taken to execute the operation. By caching the IArtifactPropertyType for each property to be set, this lookup only needs to happen once.

```
Sub PrintNameAndDocumentation (theArtifacts As ArtifactCollection,
theType As ArtifactType)
   Dim theNamePropertyType As ArtifactPropertyType
   Dim theDocumentationPropertyType As ArtifactPropertyType
   Dim theArtifact As Artifact


   Set theNamePropertyType = theType.PropertyTypes("Name")
   Set theDocumentationPropertyType =
theType.PropertyTypes("Documentation")


For i = 0 to theArtifacts.Count - 1
   Set theArtifact = theArtifacts (i)
   Print theArtifact.GetPropertyValue (theNamePropertyType)
   Print theArtifact.GetPropertyValue (theDocumentationPropertyType)
Next I
```

This approach can be used in order to optimize client tools performance.

Setting the Values of Properties

When the **OK** button is pressed, the SaveProperties method of the property sheet is called. SaveProperties works by enumerating all of the rows of the list control and looking up the property value for that name in the Artifact. If the values don't match, then the IArtifact.SetPropertyValue method is called to update the Artifact.

```
Sub SaveProperties (theArtifact As Artifact)
   Dim PropertyName As String
   Dim theProperty As ArtifactProperty
   Dim NewValue As Variant


   For RowID = 0 to thePropertyList.Rows.Count - 1
      ‘ Get the name of the property stored in the first column
      PropertyName = thePropertyList.GetValue (RowID, 0)
      ‘ Get the value of the property stored in the list
      NewValue = thePropertyList.GetValue (RowID, 1)
      ‘ Get the property from the Artifact
      Set theProperty = theArtifact.Properties (PropertyName)
```

```
      ‘ Only set the property if the Domain is able to do so.
      If theProperty.Type.SetAllowed () Then
      ‘ If the values don't match, update the Artifact
         If theProperty.Value <> NewValue Then
            theProperty.Value = NewValue
         End If
      End If
   Next ListID
End Sub
```

This algorithm could be optimized. You could store a reference to each ArtifactProperty object for each row of the list. The process of setting the property values would become nothing more than iterating through all of the rows of the list and checking values.

```
Sub OptimizedSaveProperties (theArtifact As Artifact)
   Dim PropertyName As String
   Dim NewValue As String
   Dim theProperty As Property

   For RowID = 0 to thePropertyList.Rows.Count - 1
      ‘ Get the value of the property stored in the list
      NewValue = thePropertyList.GetValue (RowID, 1)
      ‘ Get the property object for this Row
      Set theProperty = thePropertyList.GetObjectValue (RowID)
      If Not theProperty Is Nothing Then
         ‘ If the values don't match, update the Artifact
         If theProperty.Value <> NewValue Then
            theProperty.Value = NewValue
         End If
      Else
         ‘ Deal with the error
      End If
   Next RowID
End Sub
```

Getting Related Artifacts

You can use relationships to get related artifacts for a given artifact or artifact type. For a given:

- ArtifactType, you can get its relationship types and the related artifact types.
- Artifact, you can get the collection of related artifacts.
- Artifact collection, you can iterate through the collection and find a specific item.

Getting Relationship Types

You can get a category of relationship types for a given artifact type using the ArtifactType.GetRelationshipTypes method. You can get different related artifact types by specifying a category of relationship type. The categories are:

- rsDescendant
- rsPeer
- rsChild
- rsAll

This allows you to filter for a specific category of relationship type.

The ArtifactType.GetRelationshipTypes method returns the collection of relationship types associated with an artifact type. For example, the following code returns all (`rsAll`) relationship types for `theArtifact`:

```
Set theRelationships =
                theArtifact.Type.GetRelationshipTypes (rsAll)
```

For a given relationship type, you can get the related artifact type, using the RelationshipType.GetRelatedArtifactType method.

Getting Related Artifacts

To retrieve a collection of related artifacts you use the Artifact.GetRelatedArtifacts method and specify the relationship type.

The following code example is a function that returns a collection of all the artifacts related to the `theParentArtifact` by the "ArtifactLinks" relationship.

```
Function GetLinks (theParentArtifact As Artifact) As
ArtifactCollection
```

```
   Dim theRelationships as RelationshipTypeCollection
   Dim theLinksRelationship As RelationshipType
   Set theRelationships =
             theParentArtifact.Type.GetRelationshipTypes (rsAll)
   Set theLinksRelationship = theRelationships.Item ("ArtifactLinks")
   Set GetLinks =
             theArtifact.GetRelatedArtifacts (theLinksRelationship)
End Function
```

Using an Artifact Collection

Given a collection of artifacts, you can search through the collection to find a specific artifact:

- Using an iterator
- Using a For loop

For better performance with large artifact collections, using an iterator is more efficient than a For loop.

Iterating Through an Artifact Collection

You can use the Iterator interface to iterate through a given artifact collection.

The IArtifactCollection GetIterator method returns a new instance of an IArtifactIterator. Each iterator begins iteration from the beginning of the collection.

The iterator interface includes the following methods:

- HasNext(BOOL *bHasNext);
- Next(IArtifact **ppArtifact);

This allows the following code to be written to iterate the artifacts in a collection:

```
   IterateArtifacts (theArtifacts As ArtifactCollection)
   Dim Iterator As ArtifactIterator
   Set Iterator = theArtifacts.GetIterator
   While Iterator.HasNext ()
      Set theArtifact = Iterator.Next ()
   WEnd
```

The Iterator method allows ClearQuest artifacts to be enumerated efficiently. When ClearQuest executes a query, it returns a record set. The record set can not return the number of records in the set. It is only able to enumerate each of the records sequentially, from the beginning of the set to the end.

It is guaranteed that every artifact collection is able to return an iterator. This supports the principal that clients are able to write code once that works with all adapters. You can also use the count based collection interface on ClearQuest, but it runs slower than the iterator. Rational recommends that clients convert code as needed to use the iterators, especially where performance is critical.

Looping Through an Artifact Collection

The following code example loops through an artifact collection and gets the name and default ArtifactID for each artifact in the collection:

```
   For ArtifactID = 0 To theArtifactCollection.Count - 1
      Set theArtifact = theArtifactCollection.Item(ArtifactID)
      ArtifactName = theArtifact.Name
      ArtifactIDText =
              theArtifact.GetDefaultArtifactID(rsLocator_Display_Name)
Next ArtifactID
```

Filtering and Sorting

Note: Sorting is not currently implemented.

The RSE filtering and sorting mechanism is complementary to the methods that you have already seen for getting related artifacts from a context artifact. Filtering is accomplished by passing an ArtifactFilter object to GetRelatedArtifacts. Sorting is accomplished by passing an ArtifactSort object to either method. Both of these are optional parameters in both methods.

This is the basic sequence:

1. Create an ArtifactFilter object
2. Initialize it with a filter string (for example, "Name = Bob and Status = Open")
3. Create an ArtifactSort object
4. Initialize it with a sort string (for example, "Ascending Name, Descending Status")
5. Pass these objects when calling GetRelatedArtifacts.

The following code example shows how to define and execute a search for related artifacts of a given type by filtering:

```
Function QueryRelatedArtifacts (theArtifact As Artifact, RelName As
String, FilterString As String) As ArtifactCollection
  Dim theRelationships As RelationshipTypeCollection
  Dim theRelationship As RelationshipType
  Dim theFilter As ArtifactFilter


  Set theRelationships = theArtifact.Type.GetRelationshipTypes(rsAll)
  Set theRelationship = theRelationships.Item(RelName)
  Set theFilter = theRelationship.CreateArtifactFilter
  theFilter.SetFilterString (FilterString)
  Set QueryRelatedArtifacts =
theArtifact.GetRelatedArtifacts(theRelationship, theFilter)
End Sub
```

Initializing the Filter String

You can filter for artifacts using the artifact Name or Key. When filtering, you must surround the property name with single quotes for the parser to handle property names with spaces. For example:

```
'Data ModelerIsTable' = 'False'
```

In the above example, DataModelerIsTable could be a Name or Key, comparing its value to the string ‘False.’

The following rules apply for specifying a filter string:

- IS legal:

  'Data ModelerIsTable' = 'False'

- IS NOT legal:

  Data ModelerIsTable = 'False'

- IS legal:

  Name = 'Purchase Order'

- IS legal:

  'Name' = 'Purchase Order'

Filter String Example

You can use a filter string to filter for the collection of objects in a Rose model that have a specific property value in common.

For example, to retrieve the collection of Class artifacts that all have the Boolean DataModelerIsTable property set to False, you can filter for a collection of artifacts using the Model.AllClasses relationship with the following filter string:

```
'Data ModelerIsTable' = 'False'.
```

Another example of a filter string for the AllClasses relationship is:

```
Stereotype = 'Interfaces'
```

Using the AllClasses relationship, this filter string returns the Class artifacts with the Stereotype property set to Interfaces.

Most queries start from the name of a property, followed by an operator and a value. There can be any number of property names, values, and operators in a filter string.

Filtering Operators

Table 2 lists the relational operators that the IFilter interface supports:

Table 2 Filtering Operators

Operator	Description	Example
=	Equal	Position = 'Manager'
<>	Not equal	Position <> 'Manager'
!=	Not equal	Position =! 'Manager'
<	Less than	Salary < 5000
>	Greater than	ParentObject.UniqueID > NULL
<=	Less than or equal to	Salary <= 5000
>=	Greater than or equal to	Salary >= 5000
AND	Logical AND	Salary > 5000 AND Benefits > 10000
OR	Logical OR	Salary > 5000 OR Benefits > 10000
LIKE		Position LIKE ('Manager' OR 'Staff')
NOT LIKE		Position NOT LIKE 'Staff'

Operator	Description	Example
BETWEEN		**Salary BETWEEN 30000 AND 50000**
NOT BETWEEN		**Salary NOT BETWEEN 30000 AND 50000;**
IN		**Position IN ('Manager', 'Staff')**
NOT IN		**Position NOT IN ('Manager', 'Staff')**
IsKindOf		**ParentObject IsKindOf 'Item'**

The syntax rules for filter strings are:

- If the value of the property is a string, include it in single quotes. For example, Position = 'Manager'
- If the name of the unary relationship has spaces, enclose it in single quotes. For example, ParentObject.'Other Object' = 'Object Name'
- If a logical statement includes a number of comparisons, use parentheses. For example, Position = 'Manager' AND (Salary > 5000 OR Benefits > 10000)

It is possible to specify a chain of related objects to be extracted before the evaluation. Each object is extracted from the previous object. The relationships must be unary and the relationship names are separated by periods. For example, ParentObject.ParentObjectOfParentObject.Status = 'open'

For all of the operators, the last identifier in the chain of objects is a property name, except for IsKindOf (for example, ParentObject.ParentObjectOfParentObject IsKindOf 'document').

Rose Adapter Filter String Examples

Some valid filter strings for the Package artifact type and the NestedSubPackages relationship type:

```
Name IN ('Package1', 'Package3')
Name NOT IN ('Package3', 'Package5')
Name BETWEEN 'Package1' AND 'Package4'
Name NOT BETWEEN 'Package3' AND 'Package5'
Name != 'Package1'
Name <> 'Package1'
```

```
Name > ‘Package1’  AND Name =< ‘Package10’
Name >= ‘Package1’ OR UniqueID > ‘0’
```

For the Model artifact type and AllRelationships Relationship Type:

```
IsKindOf 'Role'
```

For the Model artifact type and Packages Relationship Type:

```
ParentPackage.Name = 'Bank'
Name LIKE 'Package'
Name NOT LIKE 'Package'
```

The LIKE Operator

The RSE LIKE operator searches the expression from its first character for the pattern provided in LIKE. After this character is found, LIKE calls it a match, no matter what follows after the match.

For example, Name LIKE 'b' returns anything starting with b.

Note: These operations are case sensitive. For example, getting the Rose Models from the Rational Administrator adapter (RAdmin) with:

- Path LIKE 'c:' returns nothing
- Path LIKE 'C:' returns all models on C:\

You can use single quotes or double quotes.

- Path LIKE "C:" returns nothing
- Path LIKE "C:" returns all models on C:\

If your filter string contains quotes within it, then you must surround the entire string with quotes.

LIKE Syntax

Literals

All characters are literals except those in the following list. These characters are literals when preceeded by a "\".

- \
- .
- *
- ?

- +
- (
-)
- {
- }
- [
-]
- ^
- $

If you are specifying a path that includes backslashes ("\"), you must change them as follows:

- Path LIKE 'C:\' returns nothing
- Path LIKE 'C:\\' returns all models on C:\

Wildcard

The dot character "." matches any single character.

Repeats

A repeat is an expression that is repeated an arbitrary number of times.

- An expression followed by "*" can be repeated any number of times including zero (0–n times). For example:

  “Name LIKE ba*” matches anything that starts with be, ba, baa and other similar examples.

  “Name LIKE ba*t” returns baaatklklk, since it has the specified pattern at the beginning.

- An expression followed by "+" can be repeated any number of times, but at least once (1–n times).
- An expression followed by "?" may be repeated zero or one times only (0 or 1 time).

When it is necessary to specify the minimum and maximum number of repeats explicitly, the bounds operator "{}" may be used, thus "a{2}" is the letter "a" repeated exactly twice, "a{2,4}" represents the letter "a" repeated between 2 and 4 times, and "a{2,}" represents the letter "a" repeated at least twice with no upper limit. Note that

there must be no white space inside the {}, and there is no upper limit on the values of the lower and upper bounds. All repeat expressions refer to the shortest possible previous sub-expression: a single character; a character set, or a sub-expression grouped with "()" for example.

For example:

"ba*" matches all of "b", "ba", "baaa" etc.

"ba+" matches "ba" or "baaaa" for example but not "b".

"ba?" matches "b" or "ba".

"ba{2,4}" matches "baa", "baaa" and "baaaa".

Parenthesis

Parentheses serve two purposes, to group items together into a sub-expression, and to mark what generated the match.

For example:

- Name LIKE 'b{3,5}' returns baaa and also baaaaas since it has at least 3 a's.

"(ab)*" matches all of the string "ababab". It matches 0–n ab's.

"a(ab)+" matches "aabab". It matches at least one ab.

Alternatives

Alternatives occur when the expression can match either one sub-expression or another, each alternative is separated by a "|". Each alternative is the largest possible previous sub-expression; this is the opposite behavior from repetition operators.

For example:

"a(b|c)" matches "ab" or "ac".

"abc|def" matches "abc" or "def".

Sets

A set is a set of characters that can match any single character that is a member of the set. Sets are delimited by "[" and "]" and can contain literals, character ranges, character classes, collating elements and equivalence classes. Set declarations that start with "^" contain the complement of the elements that follow. These matches are case sensitive.

- Name LIKE '[a-d]'

Returns Packages that start from 'a' to 'd'

For example:

- Character literals:

  "[abc]" matches either of "a", "b", or "c".

  Name LIKE '[a-d]' returns Packages that start from 'a' to 'd'

  "[^abc] matches any character other than "a", "b", or "c".

- Character ranges:

  "[a-z]" matches any character in the range "a" to "z".

  "[^A-Z]" matches any character other than those in the range "A" to "Z".

Character classes are denoted using the syntax "[:classname:]" within a set declaration, for example "[[:space:]]" is the set of all white-space characters.

For example, “Package[[:digit:]]” returns Package1, Package2, and Package3.’

For the Rose adapter, using the Package artifact type and NestedSubPackages relationship type:

Name LIKE 'Package[[:digit:]]' returns Package1, Package3

Table 3 lists the available character classes.

Table 3 Character Classes

Name	Description
Alnum	Any alphanumeric character.
alpha	Any alphabetical character a–z and A–Z. Other characters may also be included depending upon the locale.
blank	Any blank character, either a space or a tab.
cntrl	Any control character.
digit	Any digit 0-9.
graph	Any graphics character.
lower	Any lower case character a-z. Other characters may also be included depending upon the locale.
print	Any printable character.
punct	Any punctuation character.

Name	Description
space	Any white-space character.
upper	Any uppercase character A–Z. Other characters may also be included depending upon the locale.
xdigit	Any hexadecimal digit character, 0–9, a–f and A–F.
word	Any word character –all alphanumeric characters plus the underscore.

Table 4 lists some shortcuts that you can use in place of the character classes.

Table 4 Character Class Shortcuts

Shortcut	Meaning
\w	Equivalent to [[:word:]]
\W	Equivalent to [^[:word:]]
\d	Equivalent to [[:digit:]]
\D	Equivalent to [^[:digit:]]
\s	Equivalent to [[:space:]]
\S	Equivalent to [^[:space:]]
\l	Equivalent to [[:lower:]]
\L	Equivalent to [^[:lower:]]
\u	Equivalent to [[:upper:]]
\U	Equivalent to [^[:upper:]]

For example, in the Rose adapter, using the Package artifact type and NestedSubPackages relationship type:

- Name LIKE 'Package\d'

  Returns Package1, Package3

- Name LIKE 'Package\s'

  Returns ‘Package 2.’ Returns nothing, if there are no Package names with a space in the name.

- Name LIKE 'Package\S'

  Returns Package1, Package3

Single Character Escape Sequences

Table 5 lists the escape sequences that are aliases for character codes.

Table 5 Character Code Aliases

Escape Sequence	Character Code	Meaning
\a	0x07	Bell character
\f	0x08	Form feed
\n	0x0A	Newline character
\r	0x0D	Carriage return
\t	0x09	Tab character
\v	0x0B	Vertical tab
\e	0x1B	ASCII Escape character
\0dd	0dd	An octal character code, where *dd* is one or more octal digits
\xXX	0xXX	A hexadecimal character code, where XX is one or more hexadecimal digits
\x{XX}	0xXX	A hexadecimal character code, where XX is one or more hexadecimal digits, optionally a unicode character
\cZ	z-@	An ASCII escape sequence control+Z, where Z is any ASCII character greater than or equal to the character code for '@'

Using Collections

A collection of artifacts contains lists of related artifacts. There are client interfaces for the following types of collections:

- Artifact (IArtifactCollection)
- Artifact type (IArtifactTypeCollection)
- Property type (IArtifactPropertyTypeCollection)

- Property (IArtifactPropertyCollection)
- Relationship type (IRelationshipTypeCollection)
- Adapter (IAdapterCollection)
- Artifact argument (IArtifactArgumentCollection)
- Artifact locator (IArtifactLocatorCollection)
- Graphics format type (IArtifactGraphicsFormatTypeCollection)

These interfaces define various collections that are used throughout the RSE COM APIs. They all implement a common set of methods which include the following:

- Count

  Returns the count of the items in the collection.

- Item

  Returns a reference to the object at the given index using its default interface.

- Find

  Returns a reference to the object with matching 'Name' using its default interface.

- IsModifiable

  Returns TRUE if the client is able to modify the contents of the collection by adding and removing items, FALSE otherwise. In general, only user-created collections can be modified. Collections returned by RDSI objects are typically read-only to the client.

- Remove

  Call this to remove the object at the specified index from the collection.

- Add

  Appends the specified object to the end of the collection.

- AddCollection

  Appends the contents of the specified collection to this collection.

Finding an Item in a Collection

You can search for an artifact in a collection by the artifact's Name or Key, using the Item method. You can also use the Name or Key property to find a property type or relationship type. A Key is the parsed version of Name with spaces and punctuation removed.

Item is a method in each collection interface. You use Item by passing an integer or an artifact Name or Key (as a variant data type).

- If the argument in Item contains a string then it searches for an item by the given Name or Key.

  Item takes the Name or Key and returns the corresponding object in the collection. Name or Key corresponds to a string.

- If the argument is an integer, then it returns the object corresponding to that ID.

  For example, Item(0) returns the object in the first location in the collection, Item(1) returns the object in the second location in the collection.

For example, in the Rose model Order System, there is a package named Business Services. To find this Package artifact, you can:

- Get the item by number:

```
Set theArtifact = theArtifactCollection.Item(3)
```

- Get the item by Name:

```
Set theArtifact = theArtifactCollection.Item('Business Services')
```

- Get the item by Key:

```
Set theArtifact = theArtifactCollection.Item('BusinessServices')
```

Maintaining a List of Artifacts

Maintaining your own list of artifacts is easily solved using RSE. Because all collections can be created, RSE clients can use the ArtifactCollection to store references to artifacts. For example:

```
Private RootArtifacts As ArtifactCollection

Sub Construct
Set RootArtifacts = New ArtifactCollection
End Sub

Sub AddRootArtifact (theRootArtifact As Artifact)
RootArtifacts.Add theRootArtifact
End Sub
```

The Artifact instance passed to AddRootArtifact is now a registered Artifact in the ArtifactCollection.

Displaying Artifacts

The following code displays the Artifact in its native application. This capability is not always available, so client code must check before invoking the Show method.

Determining if an Artifact Can be Shown

Determine if an artifact can be shown by calling the CanShow method of the IArtifactGUI interface. For example:

```
Function CanDisplayArtifact (theArtifact As Artifact) As Boolean
CanDisplayArtifact = theArtifact.GUI.CanShow ()
End Function
```

Showing an Artifact in its Application

If an artifact can be shown, you show the Artifact by calling the Show method of the IArtifactGUI interface. If the Artifact is not able to be shown, this method will do nothing.

```
Sub DisplayArtifact (theArtifact As Artifact)
theArtifact.GUI.Show
End Sub
```

Converting Between the Artifact Object and the Internal Object

The RSE is intended to be complementary to each point product server. Therefore, the RSE interfaces are defined to allow this conversion to be done with minimal effort and overhead.

Getting the Internal Object from an Artifact

This conversion is very straightforward. The GetInternalObject method returns a pointer to the internal object. Then, given this pointer, you can call any integrated product interface functions.

The following example is a function that returns the RoseItem corresponding to an Artifact. If the artifact is a Rose artifact, then the internal object will be a RoseItem. The assignment between the IUnknown returned by GetInternalObject and the RoseItem return value causes QueryInterface to be called on the IUnknown. If the internal object is a RequisitePro requirement, for example, then the result will be a return value of Nothing, the Visual Basic version of NULL.

```
Function GetInternalRoseItem (theArtifact As Artifact) As RoseItem
Set GetInternalRoseItem = theArtifact.GetInternalObject ()
End Function
```

This provides the ability to plug RSE code into an integrated-product-specific code. This allows developers to work at the appropriate level of abstraction for the problem at hand, and allows RSE code to integrate with legacy code written to the specific integrated-product COM interface.

Creating and Deleting Artifacts

RSE provides interfaces for creating new artifacts and deleting artifacts. In both instances, you must first check if the action is permitted using the Boolean IsCreateDeleteAllowed method.

Creating Artifacts

Create an artifact using the CreateArtifact method to set values before the object is actually created. CreateArtifact has an optional parameter of type ArtifactArguments list to specify an initial set of property values for the new artifact. The following example initializes a GUI with the list of required creation properties.

```
Function LoadParameters (ParentArtifact As Artifact,
                         theRelationshipType As RelationshipType,
                         Name As String) As ArtifactArgumentCollection
Dim theArtifactArguments As ArtifactArgumentCollection
Dim theChildType As ArtifactType
Dim theArgument As ArtifactArgument
```

Get the ArtifactArguments from the relationship type. If any of the arguments are mandatory, they will be in the objects Arguments property. This collection can be iterated if necessary. The legal arguments are in the Arguments.Type.PropertyTypes collection, which is the same as theChildType.PropertyTypes.

```
Set theArtifactArguments =
                         theRelationshipType.CreateArtifactArguments()

RowID = 0
For PropID = 0 To ArtifactArguments.Count - 1
   Set theArgument = theArtifactArguments (PropID)
```

```
   PropertyName = thePropertyList.GetValue (RowID, 0)
   thePropertyList.Add RowID, 0, theArgument.ArgumentName
   thePropertyList.Add RowID, 1, theArgument.Value
   RowID = RowID + 1
Next PropID
Set LoadParameters = theArtifactArguments
End Sub
```

This function uses the property values from the GUI to initialize values in the ArtifactArguments list that will be used to create the new Artifact object.

```
Function CreateChildArtifact (ParentArtifact As Artifact,
theRelationshipType As RelationshipType) As Artifact
Dim ArtifactArguments As ArtifactArgumentCollection
Dim theRelationshipType As RelationshipType
Dim theArgument As ArtifactArgument

' Check the metadata to see if the type can be created before trying
If theRelationshipType.IsCreateDeleteAllowed () Then
   Exit Function
End If

Set CreateArguments = theRelationshipType.CreateArtifactArguments ()
For RowID = 0 to thePropertyList.Rows.Count - 1
   ArgumentName = thePropertyList.GetValue (RowID, 0)
   ArgumentValue = thePropertyList.GetValue (RowID, 1)
   Set theArgument = CreateArguments.Arguments (ArgumentName)
   theArgument.Value = PropertyValue
Next RowID
```

The properties in the Arguments object are used to initialize the artifact:

```
Set CreateChildArtifact =
ParentArtifact.CreateArtifact(theRelationshipType, ArtifactArguments)
End Function
```

Deleting Artifacts

Artifacts are deleted using an ArtifactRelationshipType and the context artifact corresponding to that relationship. The following code will delete an artifact if it is allowed.

```
Sub DeleteArtifact (ContextArtifact As Artifact, RelationshipName As
String, theArtifact As Artifact)
   Dim AllRelationships As RelationshipCollection
   Set AllRelationships = ParentArtifact.Type.GetRelationships (All)
   Set theRelationshipType = AllRelationships.Item (RelationshipName)
   If theRelationshipType.CreateAndDeleteAllowed () Then
      ContextArtifact.DeleteArtifact(theRelationshipType, theArtifact)
   End If
End Sub
```

Index

A

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V

W