

# Rational Suite®

## Programmer's Guide to Adapter Development Rational Suite Extensibility

VERSION: 2002.05.00

PART NUMBER: 800-025144-000

WINDOWS



**IMPORTANT NOTICE**

**COPYRIGHT**

Copyright ©1999-2001, Rational Software Corporation. All rights reserved.

Part Number: 800-025144-000

Version Number: 2002.05.00

**PERMITTED USAGE**

THIS DOCUMENT CONTAINS PROPRIETARY INFORMATION WHICH IS THE PROPERTY OF RATIONAL SOFTWARE CORPORATION (“RATIONAL”) AND IS FURNISHED FOR THE SOLE PURPOSE OF THE OPERATION AND THE MAINTENANCE OF PRODUCTS OF RATIONAL. NO PART OF THIS PUBLICATION IS TO BE USED FOR ANY OTHER PURPOSE, AND IS NOT TO BE REPRODUCED, COPIED, ADAPTED, DISCLOSED, DISTRIBUTED, TRANSMITTED, STORED IN A RETRIEVAL SYSTEM OR TRANSLATED INTO ANY HUMAN OR COMPUTER LANGUAGE, IN ANY FORM, BY ANY MEANS, IN WHOLE OR IN PART, WITHOUT THE PRIOR EXPRESS WRITTEN CONSENT OF RATIONAL.

**TRADEMARKS**

Rational, Rational Software Corporation, Rational the e-development company, ClearCase, ClearCase Attache, ClearCase MultiSite, ClearDDTS, ClearQuest, ClearQuest MultiSite, DDTS, Object Testing, Object-Oriented Recording, ObjecTime, Design, Objectory, PerformanceStudio, ProjectConsole, PureCoverage, PureDDTS, PureLink, Purify, Purify'd, Quantify, Rational, Rational Apex, Rational CRC, Rational Rose, Rational Suite, Rational Summit, Rational Visual Test, Requisite, RequisitePro, RUP, SiteCheck, SoDA, TestFactory, TestFoundation, TestMate, The Rational Watch, AnalystStudio, ClearGuide, ClearTrack, Connexis, e-Development Accelerators, ObjecTime, Rational Dashboard, Rational PerformanceArchitect, Rational Process Workbench, Rational Suite AnalystStudio, Rational Suite ContentStudio, Rational Suite Enterprise, Rational Suite ManagerStudio, Rational Unified Process, SiteLoad, TestStudio, VADS, among others, are either trademarks or registered trademarks of Rational Software Corporation in the United States and/or in other countries. All other names are used for identification purposes only, and are trademarks or registered trademarks of their respective companies.

Microsoft, the Microsoft logo, Active Accessibility, Active Channel, Active Client, Active Desktop, Active Directory, ActiveMovie, Active Platform, ActiveStore, ActiveSync, ActiveX, Ask Maxwell, Authenticode, AutoSum, BackOffice, the BackOffice logo, BizTalk, Bookshelf, Chromeffects, Clearlead, ClearType, CodeView, Computing Central, DataTips, Developer Studio, Direct3D, DirectAnimation, DirectDraw, DirectInput, DirectMusic, DirectPlay, DirectShow, DirectSound, DirectX, DirectXJ, DoubleSpace, DriveSpace, FoxPro, FrontPage, Funstone, IntelliEye, the

IntelliEye logo, IntelliMirror, IntelliSense, J/Direct, JScript, LineShare, Liquid Motion, the Microsoft eMbedded Visual Tools logo, the Microsoft Internet Explorer logo, the Microsoft Office Compatible logo, Microsoft Press, the Microsoft Press logo, Microsoft QuickBasic, MS-DOS, MSDN, Natural, NetMeeting, NetShow, the Office logo, One Thumb, OpenType, Outlook, PhotoDraw, PivotChart, PivotTable, PowerPoint, QuickAssembler, QuickShelf, Realmation, RelayOne, Rushmore, SourceSafe, TipWizard, TrueImage, TutorAssist, V-Chat, VideoFlash, Virtual Basic, the Virtual Basic logo, Visual C++, Visual FoxPro, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, the Visual Studio logo, Vizact, WebBot, WebPIP, Win32, Win32s, Win64, Windows, the Windows CE logo, the Windows logo, Windows NT, the Windows Start logo, and XENIX are trademarks or registered trademarks of Microsoft Corporation in the United States and other countries.

FLEXIm and GLOBEtrotter are trademarks or registered trademarks of GLOBEtrotter Software, Inc. Licensee shall not incorporate any GLOBEtrotter software (FLEXIm libraries and utilities) into any product or application the primary purpose of which is software license management.

Portions Copyright ©1992-20xx, Summit Software Company. All rights reserved.

**PATENT**

U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,835,701. Additional patents pending.

Purify is licensed under Sun Microsystems, Inc., U.S. Patent No. 5,404,499.

**GOVERNMENT RIGHTS LEGEND**

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the applicable Rational Software Corporation license agreement and as provided in DFARS 277.7202-1(a) and 277.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (Oct. 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 227-14, as applicable.

**WARRANTY DISCLAIMER**

This document and its associated software may be used as stated in the underlying license agreement. Rational Software Corporation expressly disclaims all other warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability or fitness for a particular purpose or arising from a course of dealing, usage, or trade practice.

# Contents

<b>Preface</b> .....	<b>ix</b>
Audience .....	ix
Other Resources .....	ix
Rational Suite Documentation Roadmap .....	xi
Contacting Rational Technical Support .....	xii
<b>1 What Is RSE?</b> .....	<b>13</b>
Why Create RSE? .....	13
Benefits of Using RSE .....	13
RSE Implementation .....	14
Using RSE .....	15
RSE Clients .....	16
RSE Adapters .....	18
Conclusion .....	19
<b>2 RSE Object Model</b> .....	<b>21</b>
RSE Objects .....	21
Object Model Diagram .....	21
Session .....	22
Adapter .....	22
Artifacts .....	23
ArtifactType .....	23
Properties .....	25
PropertyType .....	25
Relationships .....	26
Locators .....	27
Artifact Arguments .....	28
Artifact References .....	29
RelativeID Artifact References .....	31
Summary .....	32
SoDA Application Example .....	33
Referencing the RDSICore Type Library .....	34
<b>3 Developing an RSE Adapter</b> .....	<b>35</b>
Architectural Overview .....	35
Developing an Adapter Overview .....	36
Setting Up an Adapter Project .....	36
Opening a Workspace .....	37

Creating a New ATL Project . . . . .	38
Adding Dependency to the CPP Framework . . . . .	40
Modifying Project Settings . . . . .	40
Modifying the Code Generation Settings . . . . .	40
Modifying the Preprocessor Settings . . . . .	42
Defining an Adapter Instance . . . . .	45
Modifying the New IDL File . . . . .	48
Modifying the Registry File . . . . .	53
Modifying the New AdapterInstance.h . . . . .	55
Modifying the New AdapterInstance.cpp . . . . .	59
Modifying the New stdafx.h . . . . .	61
Building the New Adapter dll. . . . .	61
<b>4 Using RSE Adapter Interfaces . . . . .</b>	<b>63</b>
Overview . . . . .	63
RequisitePro Example . . . . .	65
Summary . . . . .	67
Registering Artifact Types . . . . .	68
Adapter Instance . . . . .	68
Declaring and Adding Artifact Types . . . . .	68
ReqPro Adapter Example . . . . .	69
Adding Artifact Types . . . . .	70
Dynamic Artifact Types . . . . .	71
Implementing Artifact Types for an Adapter . . . . .	72
Implementing a Class for each Artifact Type . . . . .	72
Registering a Property . . . . .	73
Registering a Relationship . . . . .	74
Registering a Locator . . . . .	76
Defining a Locator . . . . .	77
Defining Locator Arguments . . . . .	78
Defining a Collection of Artifact Locators . . . . .	80
Registering Creation Arguments . . . . .	83
Using the Maps Mechanism . . . . .	84
Registering Maps . . . . .	85
Declaring Artifact Types . . . . .	85
Defining Artifact Types . . . . .	86
Definition Registration Macros . . . . .	87
Registering Properties . . . . .	88
Registering Relationships . . . . .	89
Registering Graphics Format Types . . . . .	93

Handler Declaration Macros . . . . .	93
Pass-Through Property Definitions . . . . .	94
Internal Object to Integrated-Product Object . . . . .	95
Getting an Application Object . . . . .	96
Adapter Internals . . . . .	97
C++ Framework Classes . . . . .	97
Adapter Operations . . . . .	98
Adapter Interfaces . . . . .	100
<b>5 Adapter Interface Methods . . . . .</b>	<b>101</b>
InternalObjectTypeRegistrar . . . . .	101
FRWInternalObjectTypeRegistrar . . . . .	102
AddArtifactType . . . . .	102
AddCreationArgument . . . . .	103
AddCreationPropertyArgument . . . . .	104
AddDynamicProperty . . . . .	106
AddDynamicProperty_ReadOnly . . . . .	106
AddDynamicRelationshipType . . . . .	107
InternalObjectRegistrar . . . . .	108
FRWInternalObjectRegistrar . . . . .	108
Property Type Registration Methods . . . . .	109
AddOverrideProperty . . . . .	109
AddOverrideProperty_ReadOnly . . . . .	110
AddProperty . . . . .	110
AddProperty_ReadOnly . . . . .	112
FindPropertyTypeID . . . . .	113
RegisterRunningObjectTableKey . . . . .	113
Relationship Type Registration Methods . . . . .	115
AddFilteredRelationshipType . . . . .	115
AddOverrideFilteredRelationshipType . . . . .	117
AddOverrideRelationshipType . . . . .	117
AddRelationshipType . . . . .	118
Locator Registration Methods . . . . .	119
AddAbsoluteLocator . . . . .	120
AddCreationArgument . . . . .	121
AddCreationPropertyArgument . . . . .	122
AddKeyType . . . . .	124
AddLocatorArgument . . . . .	125
AddRelativeLocator . . . . .	127
Graphics Registration Methods . . . . .	130
AddGraphicsFormatType . . . . .	130

Using the Mapping Mechanism .....	131
<b>Index .....</b>	<b>133</b>



# Preface

This guide introduces the basic concepts of Rational Suite Extensibility (RSE) and provides the details for developing adapters using the C++ framework adapter interfaces.

## Audience

---

This guide 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

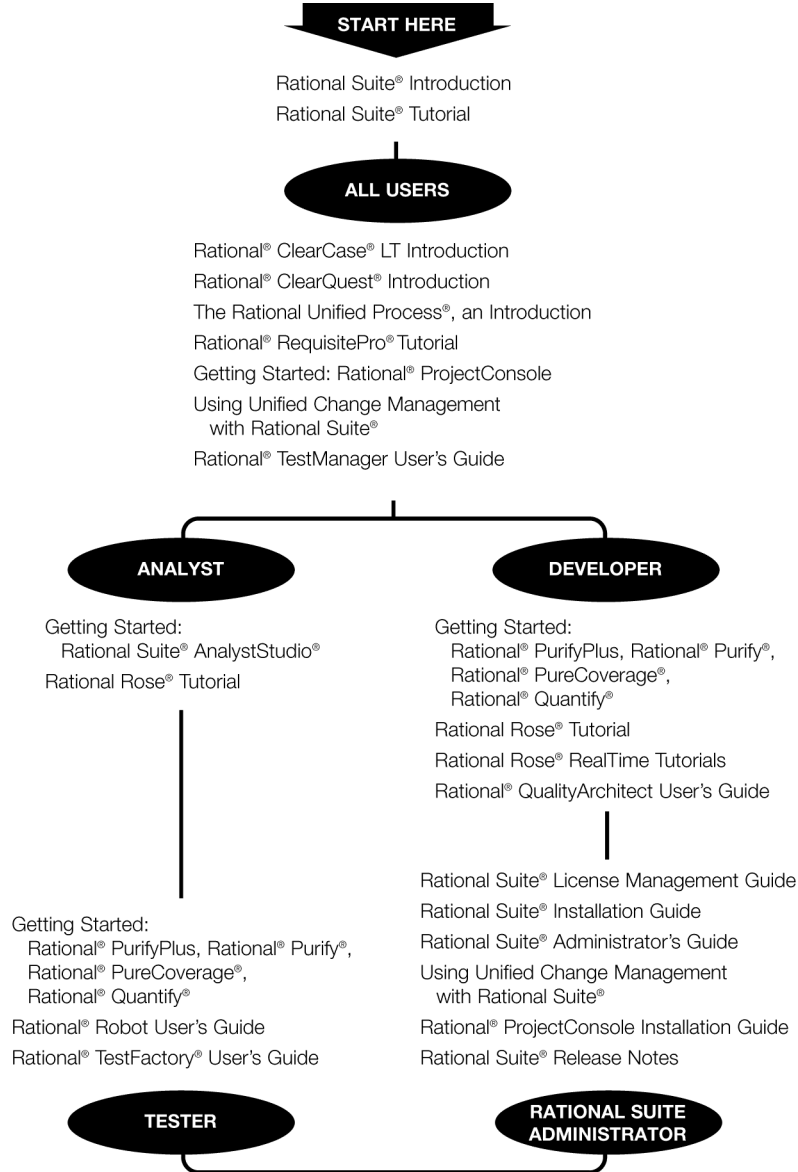
---

- Other RSE documentation:
  - Programmer’s Guide to Application Development
  - Adapters Reference
  - COM Client API Reference
- 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](mailto: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.

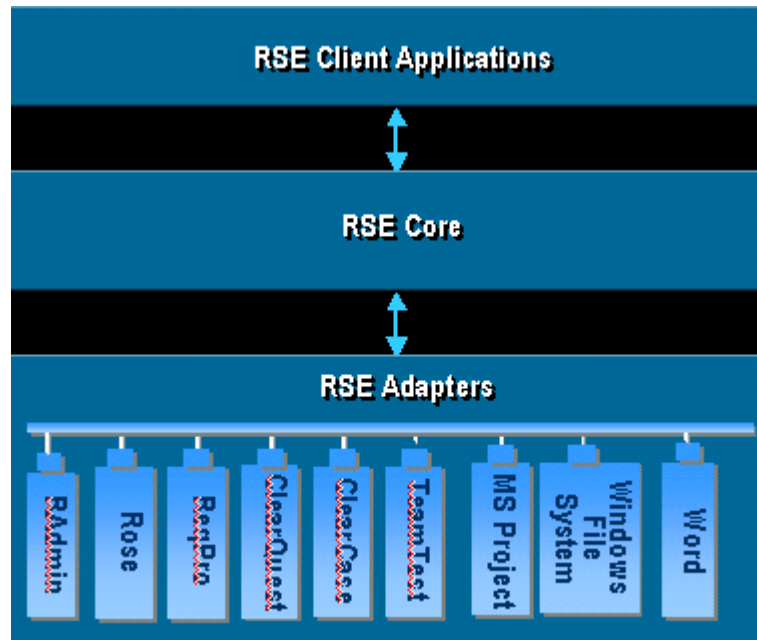
## RSE Implementation

---

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)

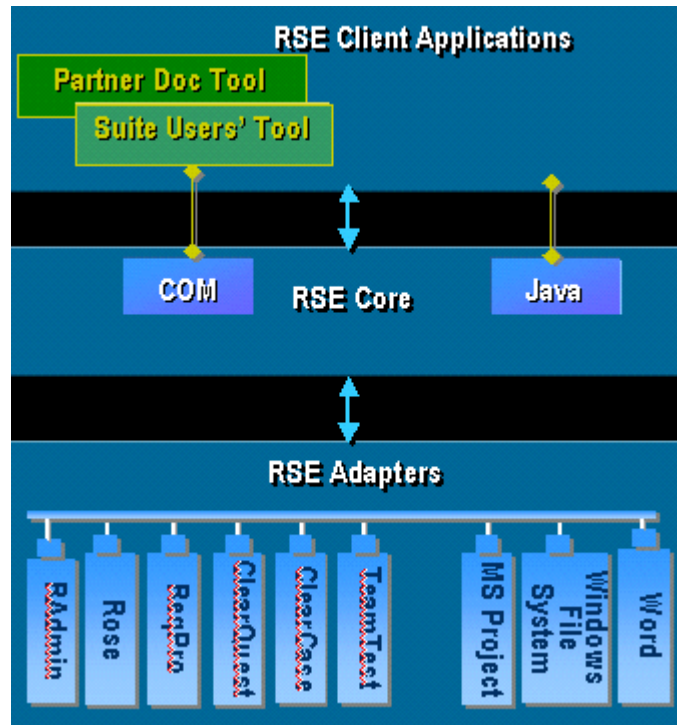
## **RSE Clients**

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.

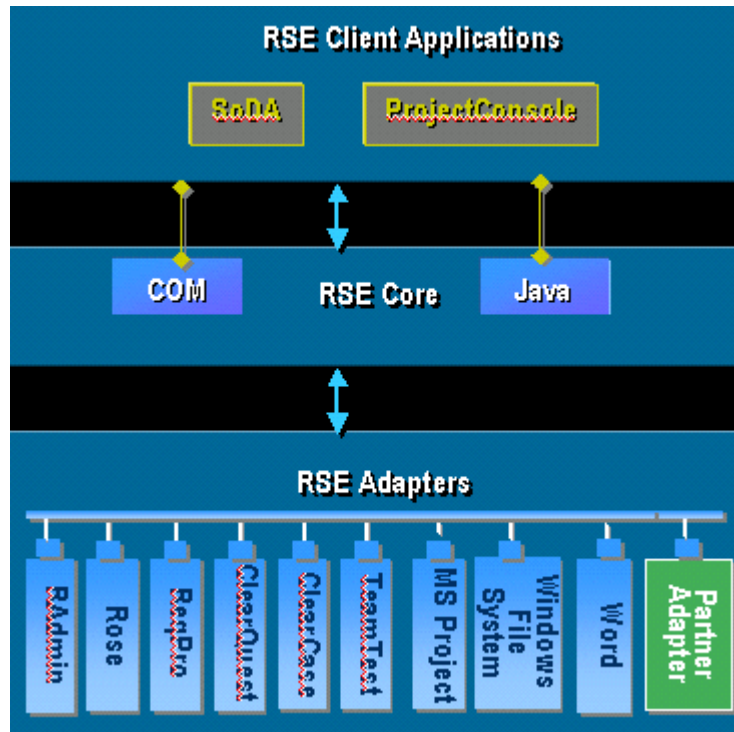
## **RSE Adapters**

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.

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

---

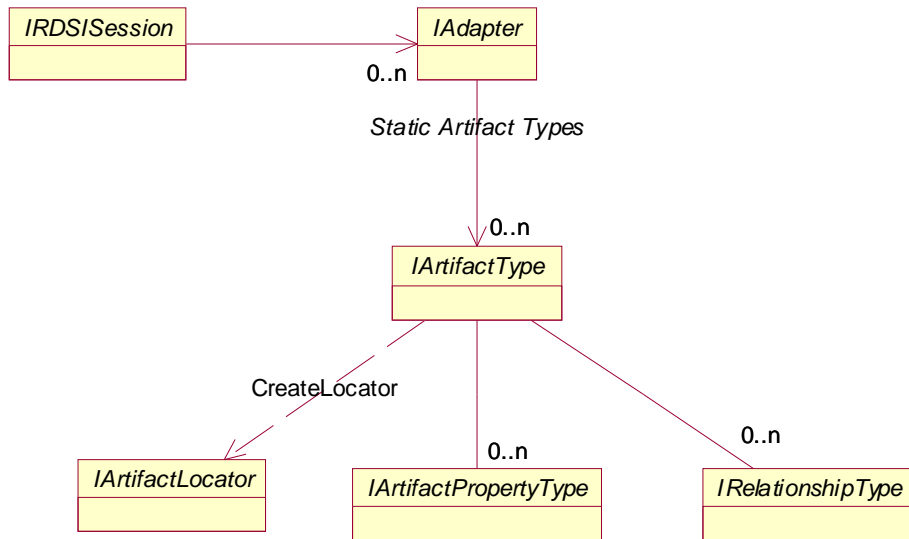
This section provides descriptions and examples of the objects in the generic object model. The primary objects are:

- 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.

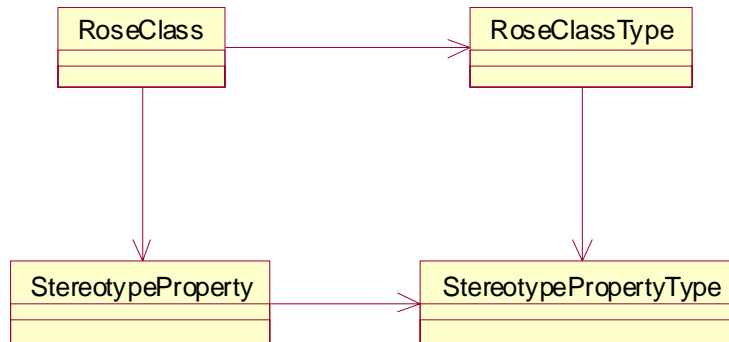
For example, the name and the stereotype of a Rose Class are properties of a Class artifact type. The Name and Stereotype properties are available from the Class artifact.

## 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 AttributeValue objects of a Requirement.
- In Rose:

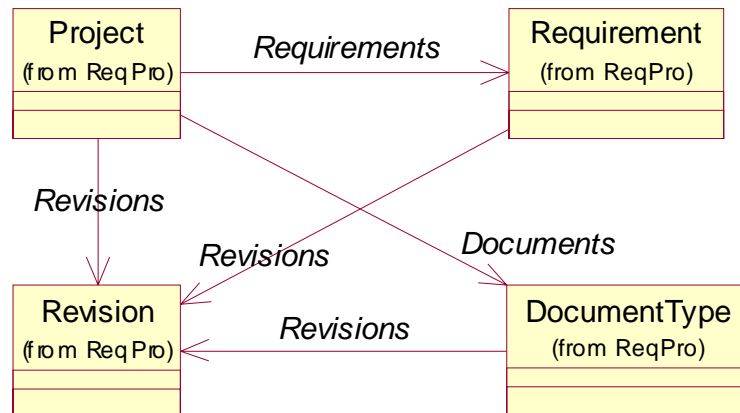
The Package artifact has a relationship to Classes. This relationship (named, Classes) can be used to find the Class objects in a Package. The Class artifact has a relationship to Properties. This relationship (named, Properties) enables you to find Property objects of a Class.

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 IArtifactLocator 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.

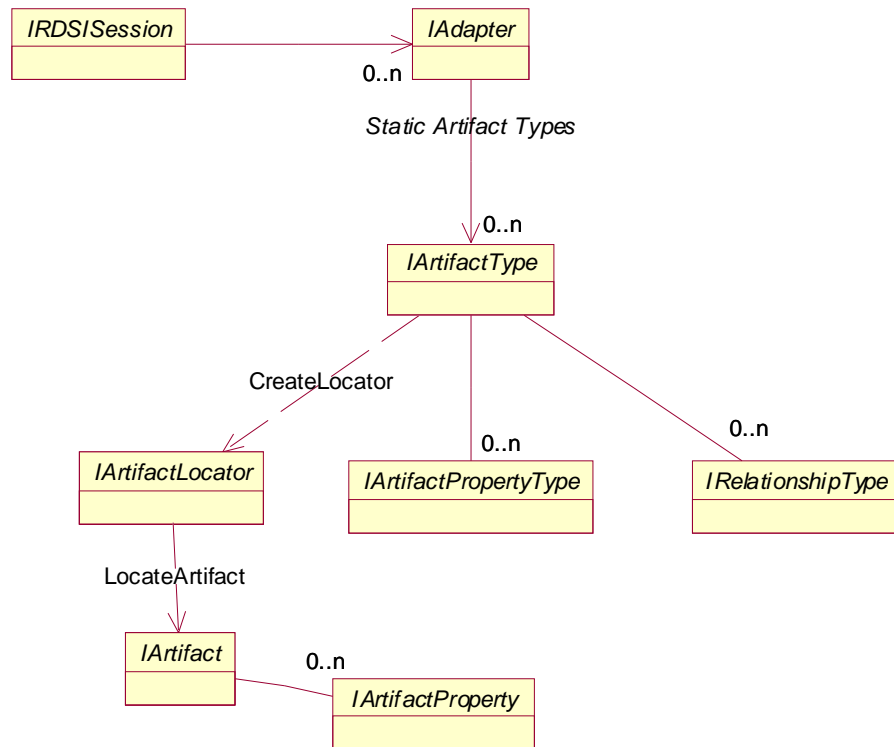
Each adapter provides the conversion between internal objects from a specific product and the corresponding RSE artifacts. An adapter includes a defined class for each artifact type. Each of these internal object classes defines properties, relationships, and locators and makes available the associated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

```
IRDSSessionPtr 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>')|Requirement(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**

This chapter describes how to develop an RSE adapter. For example, you can create an adapter to integrate a third-party tool with information in Rational Suite.

An Adapter can map a single source of information (for example, RequisitePro, Rose or other application) or be a cross-product adapter that maps more than one product.

RSE Adapters are components that map an integrated product object hierarchy to the RSE Artifact hierarchy. They define the RSE Artifact hierarchy and implement the integration between the RSE and the associated integrated product. RSE client applications retrieve information from integrated products through the RSE adapters.

## Architectural Overview

---

There are two major tiers that allow adapters to be implemented with minimal complexity:

- Adapter interfaces

The set of interfaces that the RSE Core uses to communicate with each adapter. The Adapter interfaces map from the conceptual organization of the RSE client interfaces to the conceptual organization of the adapter implementer. This mapping dramatically simplifies the complexity of the artifacts implemented in each adapter. Conceptually, it allows adapters to be implemented in a variety of development languages, including C++, Visual Basic, Visual J++, and pure Java.

**Note:** The current release only supports development of C++ adapters.

- C++ Framework

The C++ Framework wraps the adapter interface implementation and provides a default artifact behavior, further simplifying the task of implementing an adapter.

The C++ Framework simplifies the implementation of the artifact class. Modified versions of the framework can be built to support specific types of internal objects, such as IDispatch, generic COM, or CORBA.

From the adapter developer's perspective, the C++ Framework serves as a template from which you derive classes that map to your application's objects.

## Developing an Adapter Overview

---

The C++ Framework simplifies the development of new adapters by providing a template of base classes in `TemplateAdapter`. You create a new adapter by deriving new classes from the `TemplateAdapter` base classes. When you develop a new adapter, follow these steps:

- 1 Unpack the files in `RSEAdapter.zip` (this chapter uses `D:\Adapters` as the target directory). This zip file includes the `ReqPro`, `FileSys`, and `Word` RSE adapters.
- 2 Open a workspace and create a new adapter project (See the “Setting Up an Adapter Project”). You must create your new adapter project in the same directory as the included RSE adapters.

The following sections of this chapter describe these steps in more detail.

Once you have an adapter project, you can define and implement artifact types. The following steps are described in the “Using RSE Adapter Interfaces” chapter of this manual.

- 3 Define the artifact types.
- 4 Implement each artifact type. For each type, this includes:
  - Register and implement locators
  - Register and implement properties
  - Register and implement relationships

## Setting Up an Adapter Project

---

The first step in developing a new adapter is to create a new project in Microsoft Visual C++ Version 6.0.

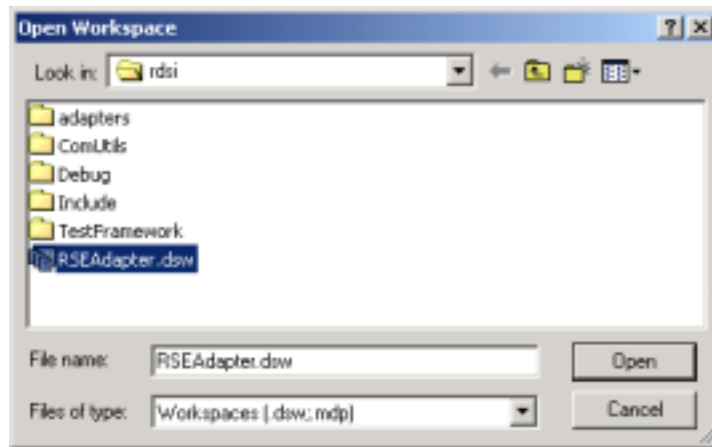
You set up a new adapter project by:

- 1 Opening a Workspace.
- 2 Creating a new ATL Project.
- 3 Creating the necessary project dependencies.
- 4 Modifying project settings.
- 5 Defining an adapter instance.
- 6 Modifying project files. This includes modifying the:
  - IDL file

- Registry file
  - Adapter instance .h and .cpp files
  - Stdafx.h file
- 7 Building the adapter dll.

## Opening a Workspace

- 1 Start Visual C++ and click **File > Open Workspace**.
- 2 Select RSEAdapter.dsw located in D:\Adapters\RSEAdapter\wsbu.src\rdsi.  
This workspace includes the ReqPro, FileSys, and Word adapter projects.  
**Note:** This location is correct if you unpacked the RSEAdapter.zip file to D:\Adapters.



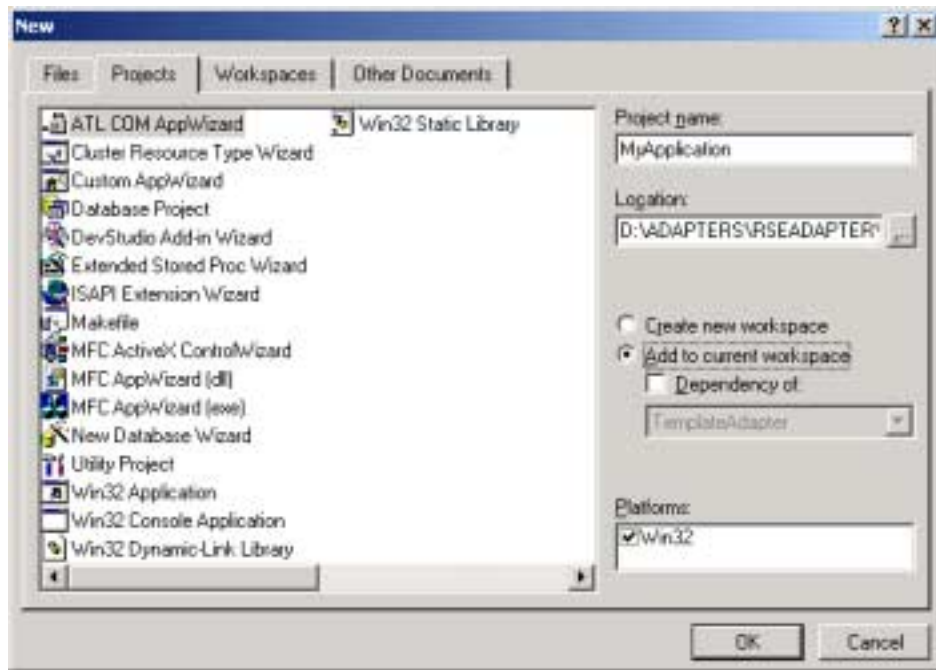
## Creating a New ATL Project

After you open the workspace, set up a new adapter project by creating a new ATL project.

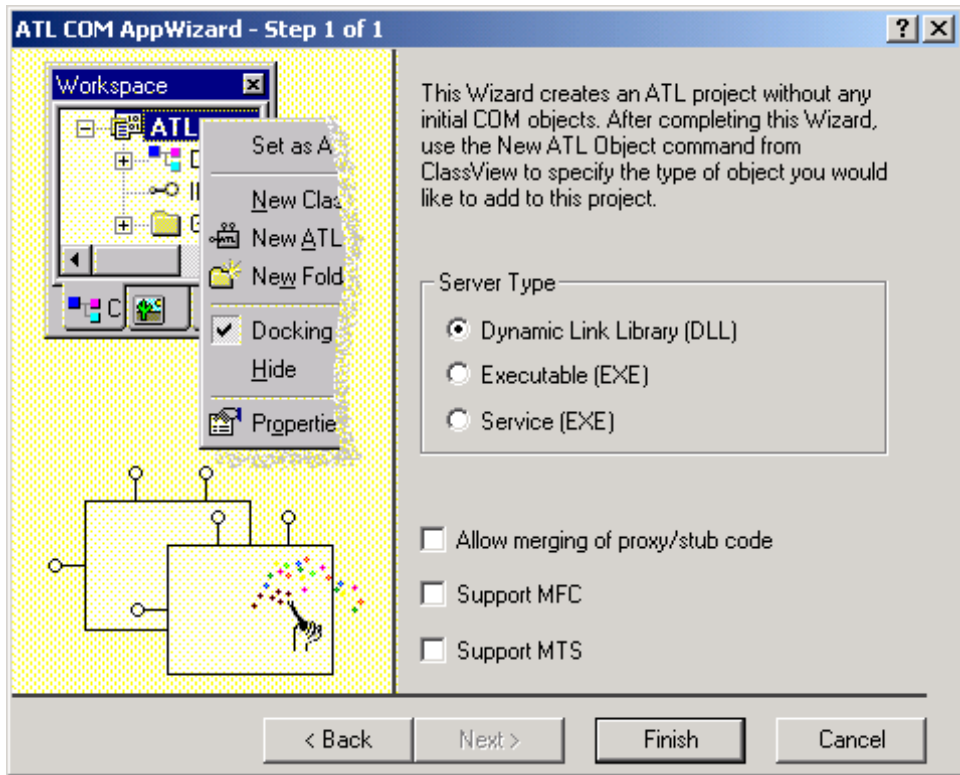
- 1 Click **File > New**.
- 2 On the Projects tab, select **ATL COM AppWizard**. Name the Project (**MyApplication**) and enter the correct location for the project (D:\ADAPTERS\RSEADAPTER\WSBU.SRC\RDS\ADAPTERS\MyApplication).

**Note:** The location of your new adapter project must be in the same directory as the other RSE adapters that are included in RSEAdapter.zip.

- 3 Click **Add to current workspace**.



- 4 Click **OK**.



- 5 On the ATL COM AppWizard, select **Dynamic Link Library (DLL)**. Click **Finish**, then click **OK** in the next dialog box.

The new project is added to the current workspace. The ATL wizard generates the source code. You then make changes to support the adapter namespace.

- 6 From the Visual C++ main menu, click **File > View** to display the new project.

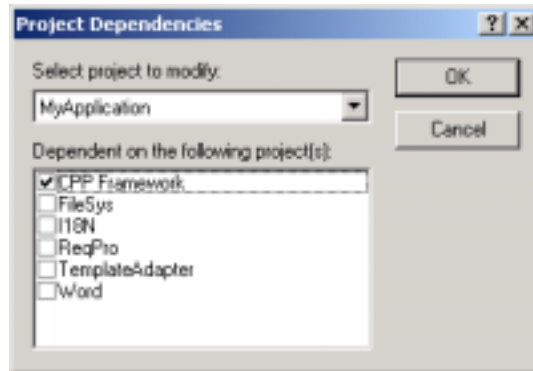
At this point, source files have been added, but there no classes. The name of the adapter is **MyApplication.cpp**.

- 7 Compile the project to create the dll file by clicking **Build**. This is the main entry point into this new adapter. After building this dll, you add a dependency to the CPP Framework and define the new adapter instance.

## Adding Dependency to the CPP Framework

To create the necessary dependency to the C++ Framework, you must add a dependency to the CPP Framework project:  
(D:\Adapters\RSEAdapter\wsbu.src\rds\adapters\CPP Framework\CPP Framework.dsp).

- 1 Click **Project > Dependencies** from the main menu.
- 2 Select the CPP Framework check box and click **OK**.



## Modifying Project Settings

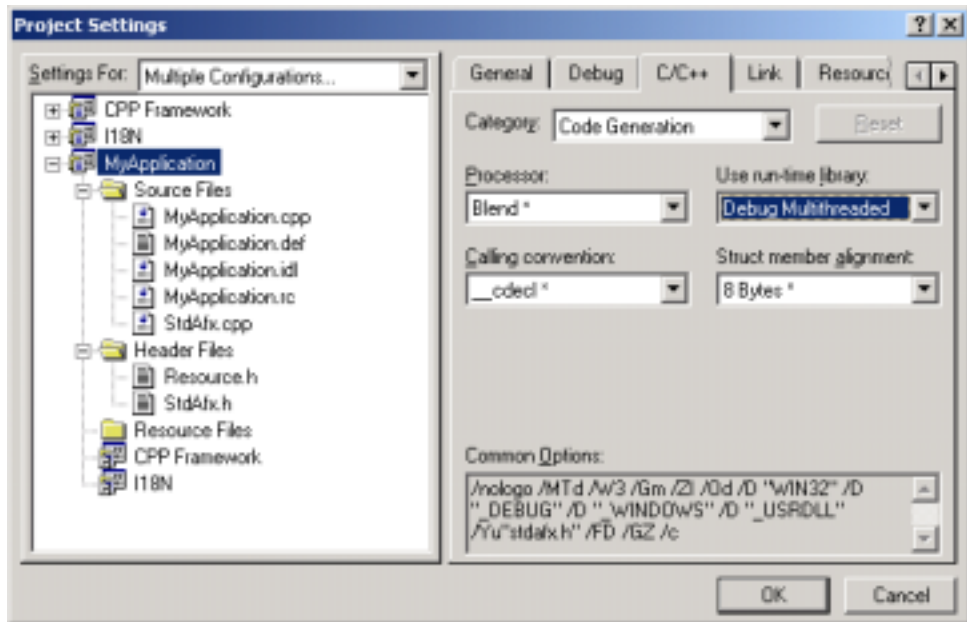
- Modifying the Code Generation Settings
- Modifying the Preprocessor Settings
- Modifying the C++ Language Settings

### Modifying the Code Generation Settings

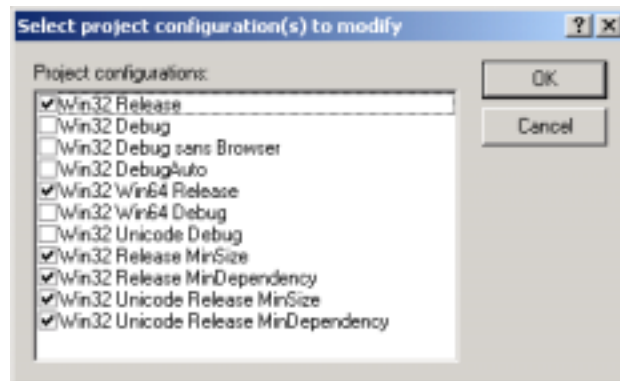
In the current environment, you must modify the project settings for code generation.

- 1 Click **Project > Settings**.
- 2 In the Setting For dialog box, choose **Multiple Configurations**.

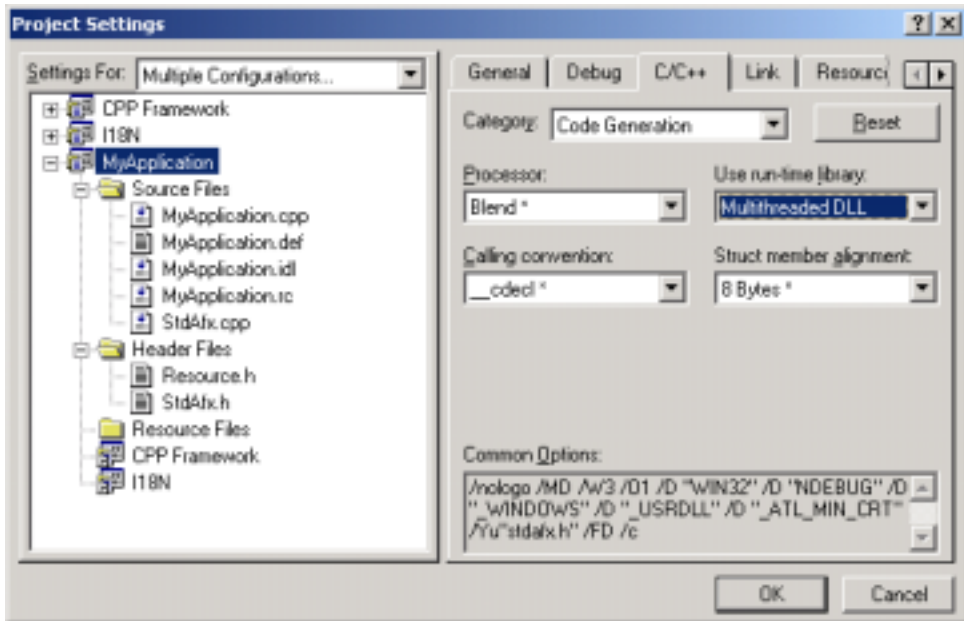




- 3 In the Select project configuration(s) to modify dialog box, check all the Win32 Debug options and click **OK**.



- 4 In the Project Settings dialog box, on the C/C++ tab, in the Category list box, select **Code Generation**.
- 5 Select **Debug Multithreaded DLL** in the Use run-time library list box and click **OK**.



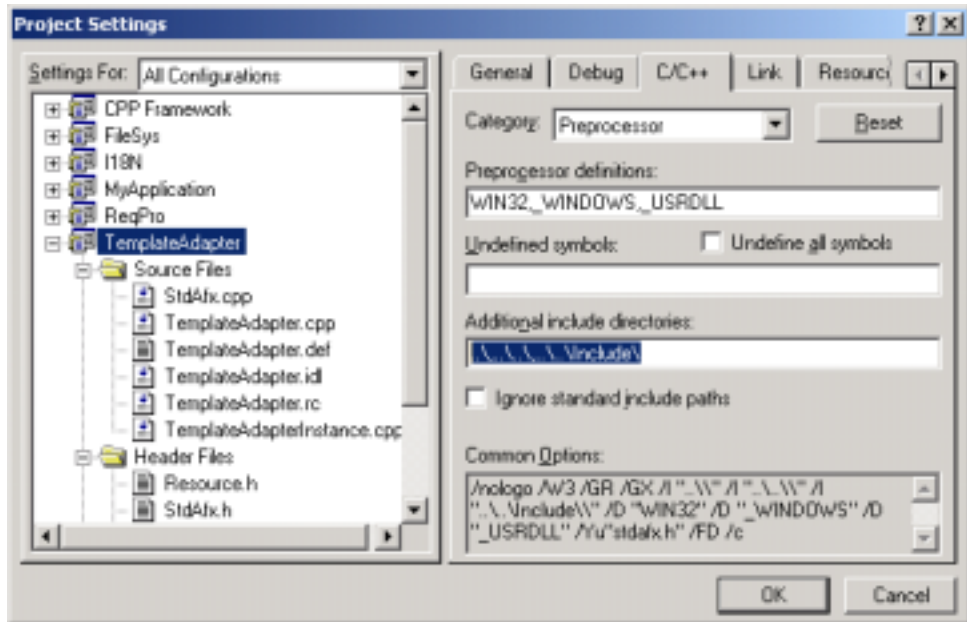
- 6 Click **Project > Settings**.
- 7 In the Setting For dialog box, choose **Multiple Configurations** and check all the Release configurations to modify.
- 8 Click **OK**.
- 9 In the Project Settings dialog box, on the C/C++ tab, in the Category list box, select **Code Generation**. Select **Multithreaded DLL** in the Use run-time library list box and click **OK**.

You must modify two additional project settings before compiling your project. This includes the Preprocessor and C++ Language categories.

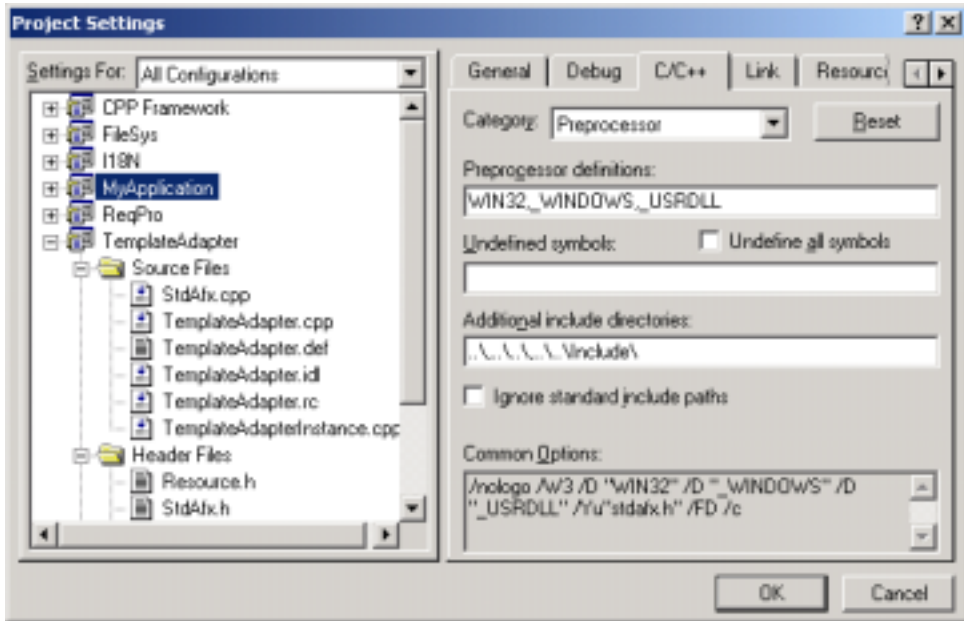
## Modifying the Preprocessor Settings

AdapterProtocol.tlb is imported in stdafx.h. To specify the path of AdapterProtocol.tlb:

- 1 Click **Project > Settings**.
- 2 In the Settings For field, select **All Configurations**.
- 3 Click the C/C++ tab and select **Preprocessor** in the Category field.
- 4 Click the TemplateAdapter project and copy the **Additional include directories** information.



- 5 Click the MyApplication project and paste in the Additional include directories information (..\..\..\Include\).

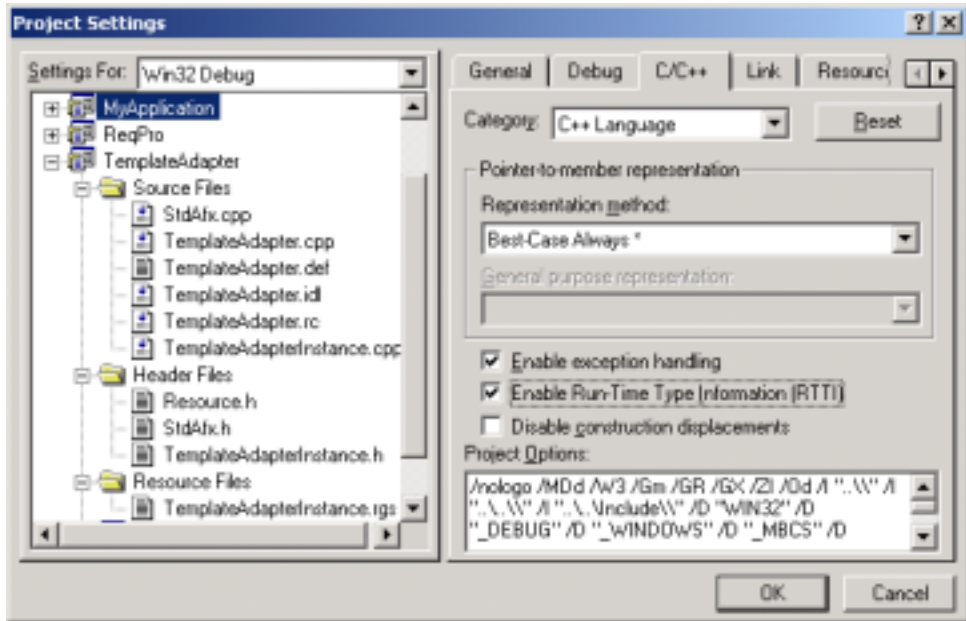


- 6 Click **OK**.

## Modifying the C++ Language Settings

Finally, you must also modify the project settings for the C++ Language Category.

- 1 Click **Project > Settings**.
- 2 In the Settings For list box, select **Win32 Debug**.
- 3 On the C/C++ tab, select **C++ Language** in the Category list box.
- 4 Select the boxes for **Enable exception handling** and **Enable Run-Time Type Information**.



Click **OK**.

Your project now has the necessary project settings.

## Defining an Adapter Instance

Each adapter instance represents an RSE adapter. The adapter instance declares the artifact types defined by this adapter. You define a new adapter instance for each new adapter.

When implementing a new adapter, you derive an AdapterInstance object from the Framework AdapterInstance class.

RSE clients retrieve information about the adapter and the static metadata available from the adapter to the client from the RSE core AdapterInstance object. It can be instantiated without creating an instance of the integrated product server.

An Adapter Instance:

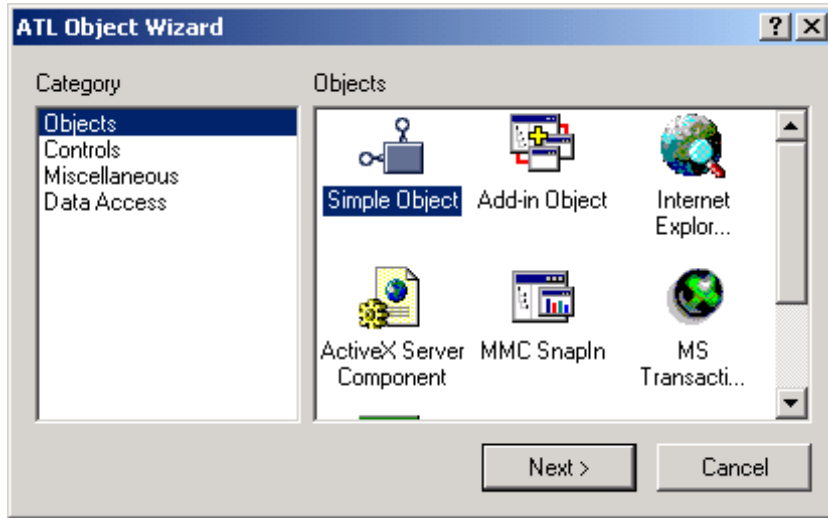
- Is an object that represents an instance of an adapter.

For example, ReqProAdapterInstance represents an instance of the ReqPro adapter. The ReqProAdapterInstance class derives from the Framework AdapterInstance class.

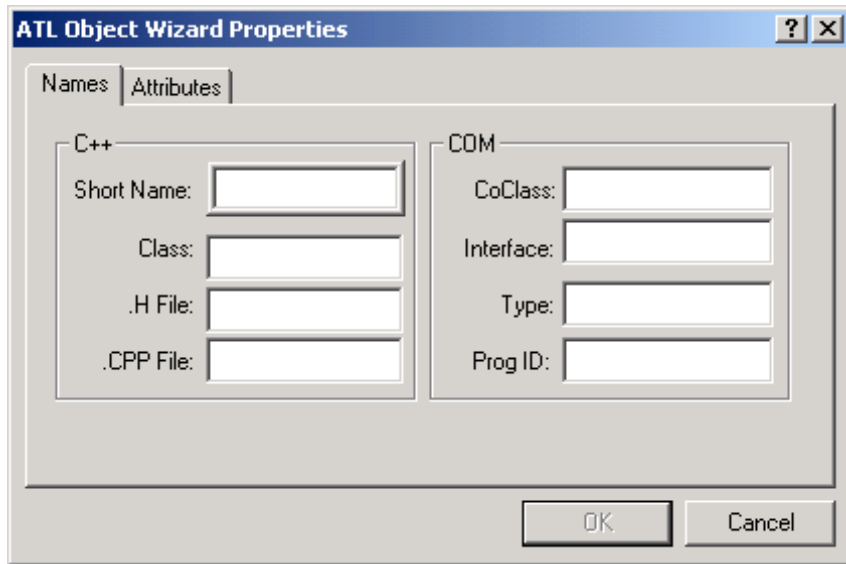
- Registers information with the RSE core and declares all static metadata available from this adapter. This includes the hierarchy of all artifact types, properties, relationships, and locators that are defined in this adapter.
- Is an ATL COM object.

This class derives from the C++ Framework AdapterInstance class.

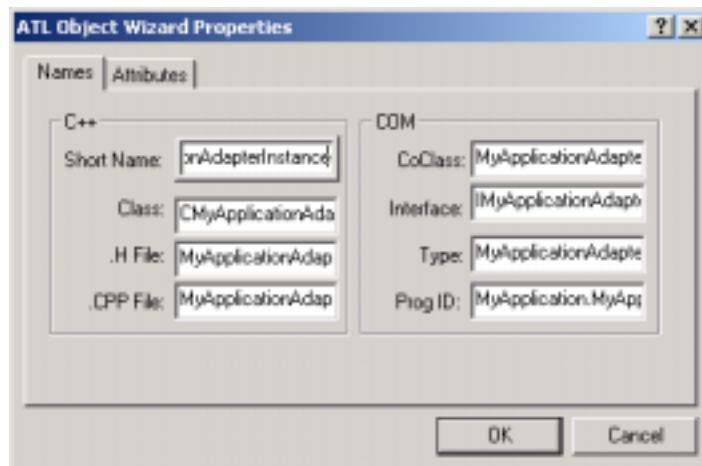
- 1 Click **Insert > New ATL Object**



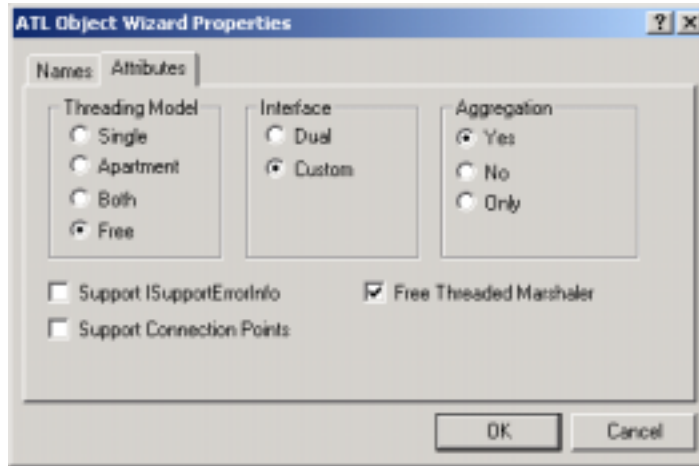
- 2 Click **Simple Object > Next**.



- 3 On the Names tab, in the Short Name field, enter the name of the new adapter instance. For example, **MyApplicationAdapterInstance**. This automatically fills in the other fields.



- 4 On the attributes tab, select **Free** in the Threading Model field and select **Custom** in the Interface field.



Leave the other default values on the attributes tab. Click OK.

This generates the new AdapterInstance cpp file (**MyApplicationAdapterInstance.cpp**) and the idl file that defines the internal stubs for this new adapter.

## Modifying the New IDL File

You must clean up the new idl file with the following code-specific modifications:

- Import AdapterProtocol.idl.
- Delete AdapterInstance object.

### 1 Importing AdapterProtocol.idl

AdapterProtocol.idl defines all of the adapter interfaces. You must add a reference for AdapterProtocol.idl to your new adapter idl file.

Add the following line from TemplateAdapter.idl to your new adapter idl file, below the `import "ocidl.idl";` statement.

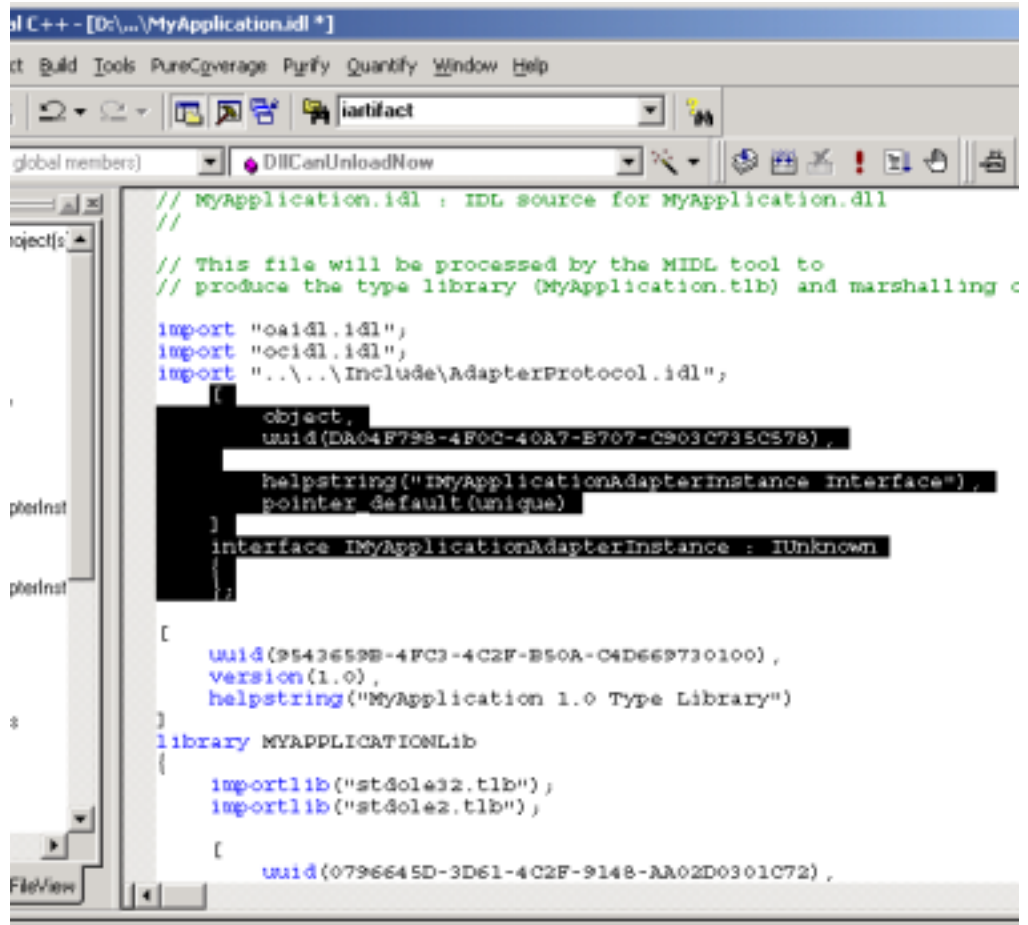
```
import "..\..\Include\AdapterProtocol.idl";
```





## 2 Deleting the AdapterInstance Object

You must delete the new adapter instance object in this idl file. This code is directly below the import line you just added.



```
// MyApplication.idl : IDL source for MyApplication.dll
//
// This file will be processed by the MIDL tool to
// produce the type library (MyApplication.tlb) and marshalling c

import "oaidl.idl";
import "ocidl.idl";
import "..\..\Include\AdapterProtocol.idl";

[
    object,
    uuid(DA04F798-4F0C-40A7-B707-C903C735C578),
    helpstring("IMyApplicationAdapterInstance Interface"),
    pointer_default(unique)
]
interface IMyApplicationAdapterInstance : IUnknown
{
};

[
    uuid(9543659B-4FC3-4C2F-B50A-C4D669730100),
    version(1.0),
    helpstring("MyApplication 1.0 Type Library")
]
library MYAPPLICATIONLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(0796645D-3D61-4C2F-9148-AA02D0301072),
```

## 3 You must also modify the coclass MyApplicationAdapterInstance definition from:

```
{
    [default] interface IMyApplicationAdapterInstance;
};
```

to:

```
{
    [default] interface IAdapterInstance;
    interface ITypeContainer;
};
```

The correct code for this file should now be:

```
// MyApplication.idl : IDL source for MyApplication.dll
//
// This file will be processed by the MIDL tool to produce
// the type library (MyApplication.tlb) and marshalling code.

import "oaidl.idl";
import "ocidl.idl";
import "..\..\Include\AdapterProtocol.idl";

[
    uuid(AA68ABA6-6F68-40E0-99CF-25C26D084978),
    version(1.0),
    helpstring("MyApplication 1.0 Type Library")
]
library MYAPPLICATIONLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(A5FE3DD9-FB73-4800-8094-4969D5163348),
        helpstring("MyApplicationAdapterInstance Class")
    ]
    coclass MyApplicationAdapterInstance
    {
```

```

    [default] interface IAdapterInstance;
interface ITypeContainer;
};
};

```

```

// MyApplication.idl : IDL source for MyApplication.dll
//
// This file will be processed by the MIDL tool to
// produce the type library (MyApplication.tlb) and marshalling c
//
import "oaidl.idl";
import "ocidl.idl";
import "..\..\Include\AdapterProtocol.idl";

[
    uuid(AD68ABA6-6F68-40E0-99CF-25C26D084978),
    version(1.0),
    helpstring("MyApplication 1.0 Type Library")
]
library MYAPPLICATIONLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(A5FE3DD9-FB73-4800-8094-4969D5163348),
        helpstring("MyApplicationAdapterInstance Class")
    ]
    coclass MyApplicationAdapterInstance
    {
        [default] interface IAdapterInstance;
        interface ITypeContainer;
    };
};

```

This code correctly defines the new adapter instance.

## Modifying the Registry File

The registry file (**MyApplicationAdapterInstance.rgs**) determines what goes into the registry when the adapter registers itself.

The version program ID identifies and registers the adapter.

```
VersionIndependentProgID = s
'MyApplicationAdapter.MyApplicationAdapterInstance'
```

You must add additional information to the generated registry file.

- 1 Open the **TemplateAdapterInstance.rgs** file and select the following code.

```
HKLM
{
  NoRemove SOFTWARE
  {
    'Rational Software'
    {
      RDSI
      {
        Adapters
        {
          ForceRemove Template
          {
            val Name = s 'Template'
            val ConnectData = s 'TemplateAdapter.TemplateAdapterInstance'
          }
        }
      }
    }
  }
}
```

- 2 Copy this code and paste it into the new adapter instance registry file at the end of the file.
- 3 You must then rename the adapter information that you pasted into the registry file from **TemplateAdapter** to the name of your adapter (**MyApplication**). The value of **ConnectData** must be equal to **VersionIndependentProgID**.

For example:

```
HKLM
{
  NoRemove SOFTWARE
  {
    'Rational Software'
    {
      RDSI
      {
        Adapters
        {
          ForceRemove MyApplication
          {
            val Name = s 'MyApplication'
            val ConnectData = s 'MyApplication.MyApplicationAdapterInstance'
          }
        }
      }
    }
  }
}
```

## Modifying the New AdapterInstance.h

You must modify code in the new adapter instance header file.

First, you must include the base class AdapterInstance.h from the CPP Framework to your new AdapterInstance.h file.

- 1 Copy the following line from TemplateAdapterInstance.h and add to MyApplicationAdapterInstance.h just below the `#include "resource.h"` statement:

```
#include "CPP Framework/AdapterInstance.h"
```

- 2 Replace the following code:

```
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CMyApplicationAdapterInstance,
&CLSID_MyApplicationAdapterInstance>,
public IMyApplicationAdapterInstance
```

with:

```
public CComCoClass<CMyApplicationAdapterInstance,
&CLSID_MyApplicationAdapterInstance>,
public FRWAdapterInstance
```

As the above code illustrates, you:

- Remove the following lines:

```
public CComObjectRootEx<CComSingleThreadModel>,
public IMyApplicationAdapterInstance
```

- Add:

```
public FRWAdapterInstance
```

- 3 You must also delete the following lines from this file:

```
DECLARE_PROTECT_FINAL_CONSTRUCT()
```

```
BEGIN_COM_MAP(CMyApplicationAdapterInstance)
```

```
COM_INTERFACE_ENTRY(IMyApplicationAdapterInstance)
```

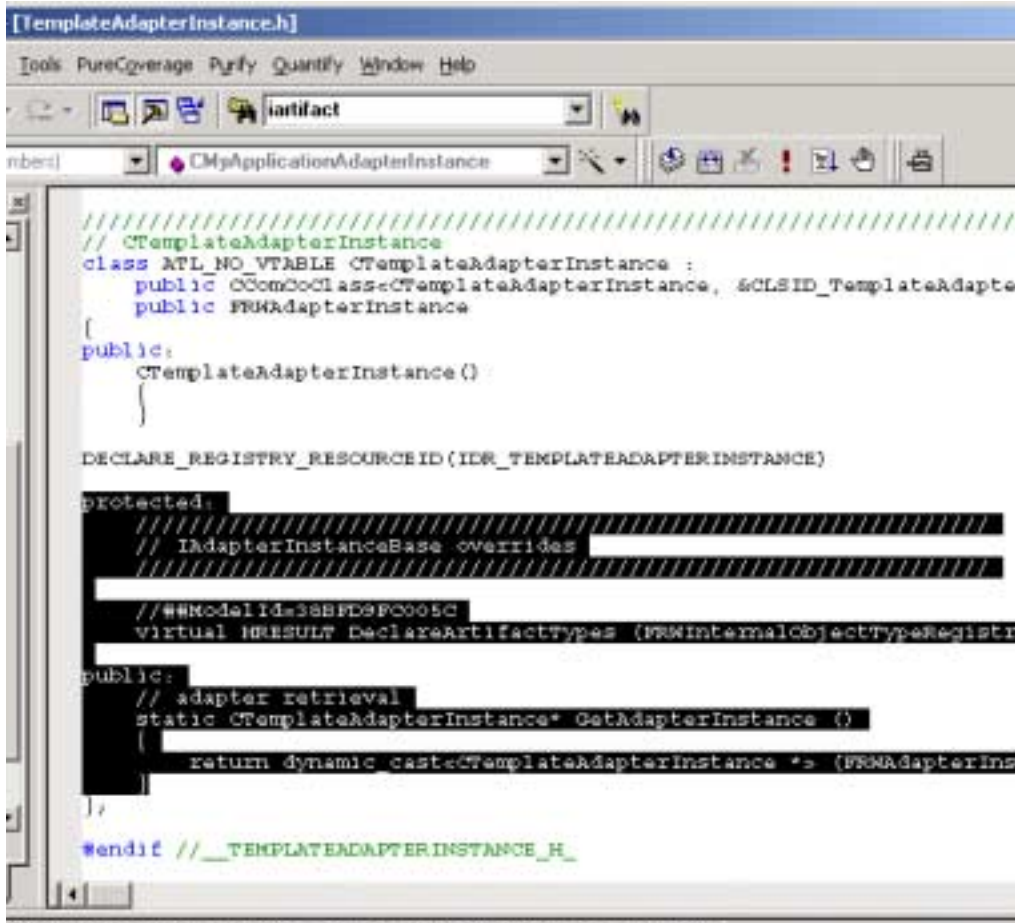
```
END_COM_MAP()
```

```
// IMyApplicationAdapterInstance
```

```
public:
```

You replace this code with code from TemplateAdapterInstance.h.

- 4 Copy the following code from TemplateAdapterInstance.h and paste it into your new adapter instance header file.



```

[TemplateAdapterInstance.h]
Tools PureCoverage Purify Quantify Window Help
artifact
CMApplicationAdapterInstance

////////////////////////////////////
// CTemplateAdapterInstance
class ATL_NO_VTABLE CTemplateAdapterInstance :
public CComCoClass<CTemplateAdapterInstance, &CLSID_TemplateAdapte
public FRWAdapterInstance
{
public:
CTemplateAdapterInstance()
}

DECLARE_REGISTRY_RESOURCEID(IDR_TEMPLATEADAPTERINSTANCE)

protected:
////////////////////////////////////
// IAdapterInstanceBase overrides
////////////////////////////////////
//ModelId=38BFDE9F0005C
virtual HRESULT DeclareArtifactTypes (FRWInternalObjectType*registI
public:
// adapter retrieval
static CTemplateAdapterInstance* GetAdapterInstance ()
{
return dynamic_cast<CTemplateAdapterInstance *> (FRWAdapterIns
}
}

#endif // __TEMPLATEADAPTERINSTANCE_H_

```



The correct code is:

```
protected:
    //////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    // IAdapterInstanceBase overrides
    //////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    virtual HRESULT DeclareArtifactTypes
(FRWInternalObjectTypeRegistrar &ObjectTypeRegistrar);

public:
    // adapter retrieval
    static CTemplateAdapterInstance* GetAdapterInstance ()
    {
        return dynamic_cast<CTemplateAdapterInstance *>
(FRWAdapterInstance::FRWGetAdapterInstance ());
    }
```

This code defines the two required methods for this class:

- DeclareArtifactTypes  
Declares all of the artifact types for this adapter
- GetAdapterInstance  
Creates an instance of this adapter

5 Rename this code to your new adapter name. For example:

```
static CMyApplicationAdapterInstance* GetAdapterInstance ()
{
    return dynamic_cast<CMyApplicationAdapterInstance *>
(FRWAdapterInstance::FRWGetAdapterInstance ());
}
```

Here is the correct code in the new adapter instance.h file:

```
// MyApplicationAdapterInstance.h : Declaration of the
CMyApplicationAdapterInstance
```

```

#ifdef __MYAPPLICATIONADAPTERINSTANCE_H_
#define __MYAPPLICATIONADAPTERINSTANCE_H_

#include "resource.h"          // main symbols
#include "CPP Framework/AdapterInstance.h"

////////////////////////////////////
////////////////////////////////////
// CMyApplicationAdapterInstance
class ATL_NO_VTABLE CMyApplicationAdapterInstance :

    public CComCoClass<CMyApplicationAdapterInstance,
    &CLSID_MyApplicationAdapterInstance>,
    public FRWAdapterInstance
{
public:
    CMyApplicationAdapterInstance()
    {
    }

    DECLARE_REGISTRY_RESOURCEID(IDR_MYAPPLICATIONADAPTERINSTANCE)
protected:
    //////////////////////////////////
    //////////////////////////////////
    //////////////////////////////////

    virtual HRESULT DeclareArtifactTypes
    (FRWInternalObjectTypeRegistrar &ObjectTypeRegistrar);

public:
    // adapter retrieval
    static CMyApplicationAdapterInstance* GetAdapterInstance ()
    {
        return dynamic_cast<CMyApplicationAdapterInstance *>
        (FRWAdapterInstance::FRWGetAdapterInstance ());
    }
};

#endif //__MYAPPLICATIONADAPTERINSTANCE_H_

```

## Modifying the New AdapterInstance.cpp

- 1 Copy the following line from TemplateAdapterInstance.cpp and paste it into your new adapter instance cpp file after the other #include statements.

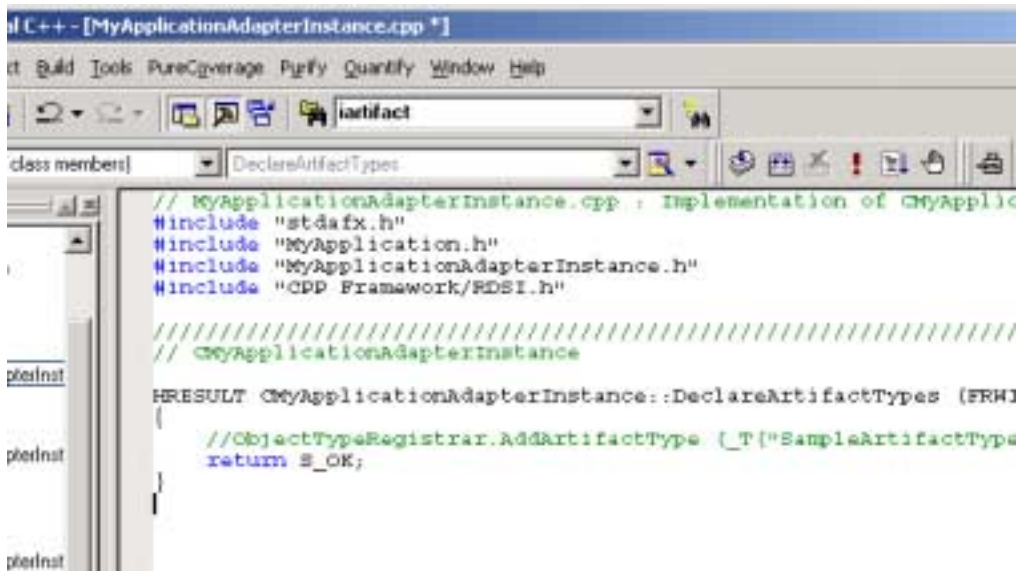
```
#include "CPP Framework/RDSI.h"
```

- 2 Implement the register method in MyApplicationAdapterInstance.cpp by copying the following code from TemplateAdapterInstance.cpp and pasting it into MyApplicationAdapterInstance.cpp.

```
HRESULT CTemplateAdapterInstance::DeclareArtifactTypes  
(FRWInternalObjectTypeRegistrar &ObjectTypeRegistrar)  
{  
    //ObjectTypeRegistrar.AddArtifactType (_T("SampleArtifactType"),  
INTERNAL_OBJECT_FACTORY(CSampleArtifactType),  
INTERNAL_OBJECT_REGISTRAR (CSampleArtifactType));  
    return S_OK;  
}
```

- 3 Rename the instance as follows:

```
HRESULT CMyApplicationAdapterInstance::DeclareArtifactTypes  
(FRWInternalObjectTypeRegistrar &ObjectTypeRegistrar)  
{  
    //ObjectTypeRegistrar.AddArtifactType (_T("SampleArtifactType"),  
INTERNAL_OBJECT_FACTORY(CSampleArtifactType),  
INTERNAL_OBJECT_REGISTRAR (CSampleArtifactType));  
    return S_OK;  
}
```



The correct code for your new adapter instance cpp file is:

```

// MyApplicationAdapterInstance.cpp : Implementation of
CMyApplicationAdapterInstance
#include "stdafx.h"
#include "MyApplication.h"
#include "MyApplicationAdapterInstance.h"
#include "CPP Framework/RDSI.h"

////////////////////////////////////
// CMyApplicationAdapterInstance

HRESULT CMyApplicationAdapterInstance::DeclareArtifactTypes
(FRWInternalObjectTypeRegistrar &ObjectTypeRegistrar)
{
    //ObjectTypeRegistrar.AddArtifactType (_T("SampleArtifactType"),
INTERNAL_OBJECT_FACTORY(CSampleArtifactType),
INTERNAL_OBJECT_REGISTRAR (CSampleArtifactType));
    return S_OK;
}

```

DeclareArtifactTypes is where each artifact type is registered, using the AddArtifactType method for each artifact type. See the "Using RSE Adapter Interfaces" chapter of this manual for more information.

## Modifying the New stdafx.h

In the new adapter project's stdafx.h file:

Add the following lines just below the #include <atlcom.h> statement:

```
#include "CPP Framework/rdsi.h"
#include "resource.h"
#import "include/AdapterProtocol.tlb" no_implementation
rename("GUID", "_GUID") rename_namespace("RDSIAdapterProtocol")
```

**Note:** You import your integrated product library in stdafx.h and you have to include this line in stdafx.h. For example, in the MyApplication adapter:

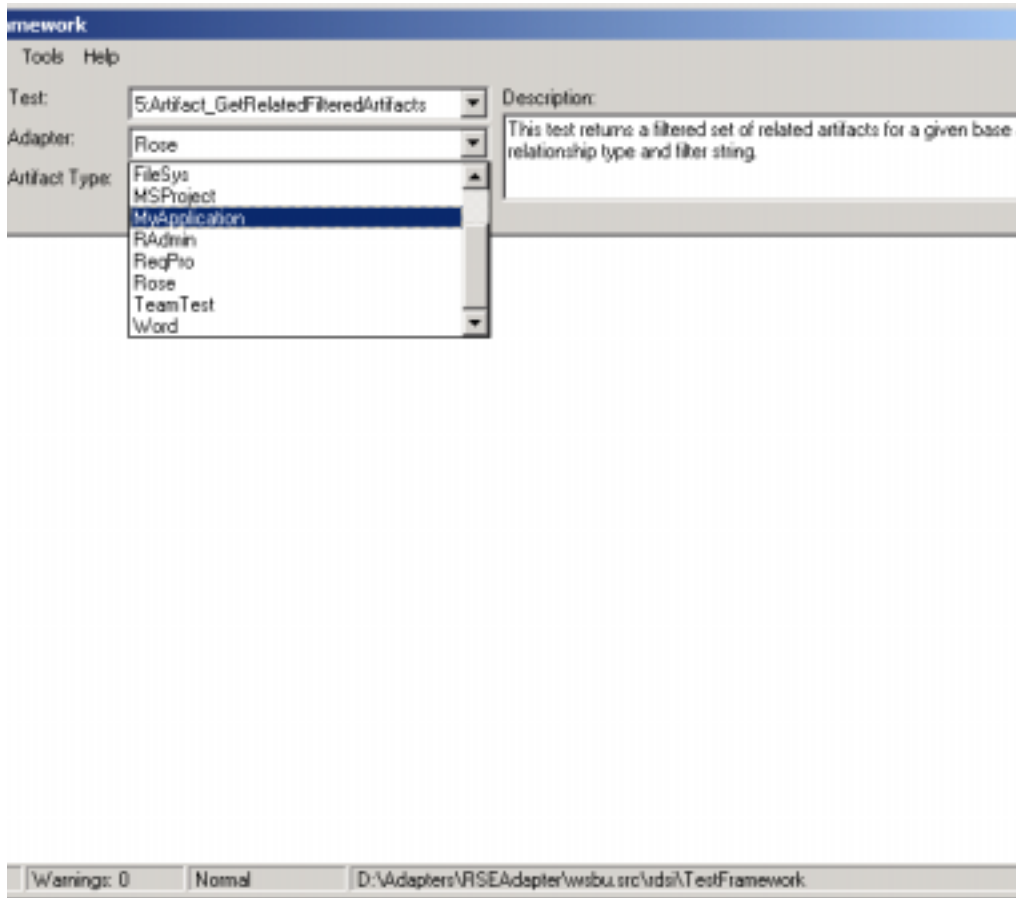
```
#import MyApplication.tlb
```

You are now ready to build the new adapter dll.

## Building the New Adapter dll

Click **Build > Build MyApplication.dll**

Your new adapter is now available from all RSE client applications. For example, run the Test Framework sample application (TestFramework.exe) and the list box now includes the new adapter.



When you define artifact types, they are available from this client application. You can now define and implement artifacts that map to objects in your integrated product.

# Using RSE Adapter Interfaces

# 4

This chapter provides information on implementing the artifact types for a given adapter. This includes:

- Registering artifact types in an AdapterInstance class
- Implementing each artifact type in an InternalObject-derived class

For each artifact type, this includes defining and implementing the properties, relationships, and locators. You also define the locator arguments for constructing the locators. For a full listing of the methods for implementing artifact types, see the “Adapter Interface Methods” chapter of this manual.

Additional topics include:

- Registering creation arguments
- Defining and implementing artifacts using the maps mechanism
- Internal object to Integrated-Product object description
- Adapter internals information

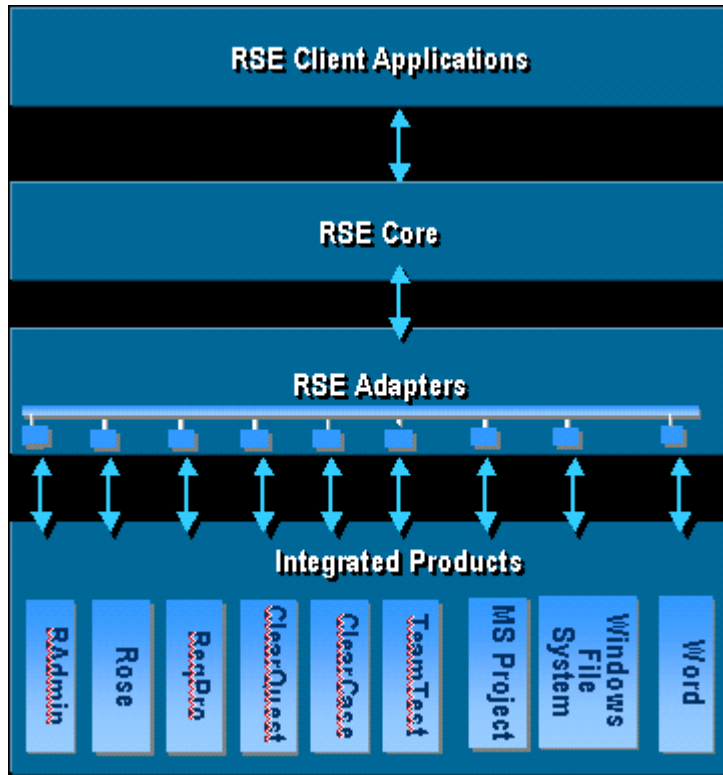
## Overview

---

The RSE object model provides one unified API for access to all Rational Suite-enabled integrated products. This common object model represents integrated product-specific objects as artifact types. RSE enables this common object model through adapters.

Figure 8 illustrates the architecture through which a client retrieves information from an integrated product.

**Figure 8** Retrieving Information from an Integrated Product



Clients request artifacts. Adapters map artifacts to integrated-product objects.

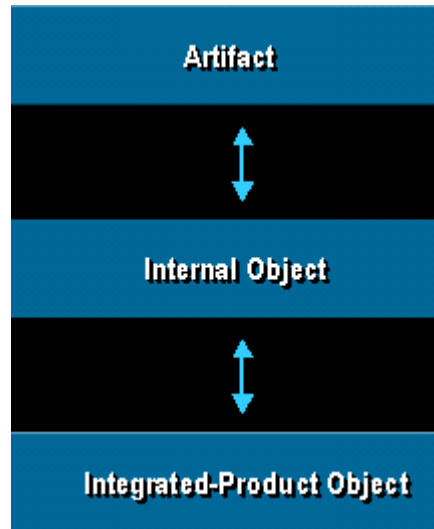
Adapters define artifacts as internal objects. An internal object is the implementation class for a specific artifact type. The RSE core implements this architecture by creating instances of the artifacts and returning them to the client.

Figure 9 illustrates the architecture by which an artifact type represents an integrated-product object.

- Each artifact type is represented by an internal object class that is defined in a given adapter.
- Each adapter links the internal objects to their corresponding integrated product objects.



**Figure 9 Representing an Integrated Product Object**

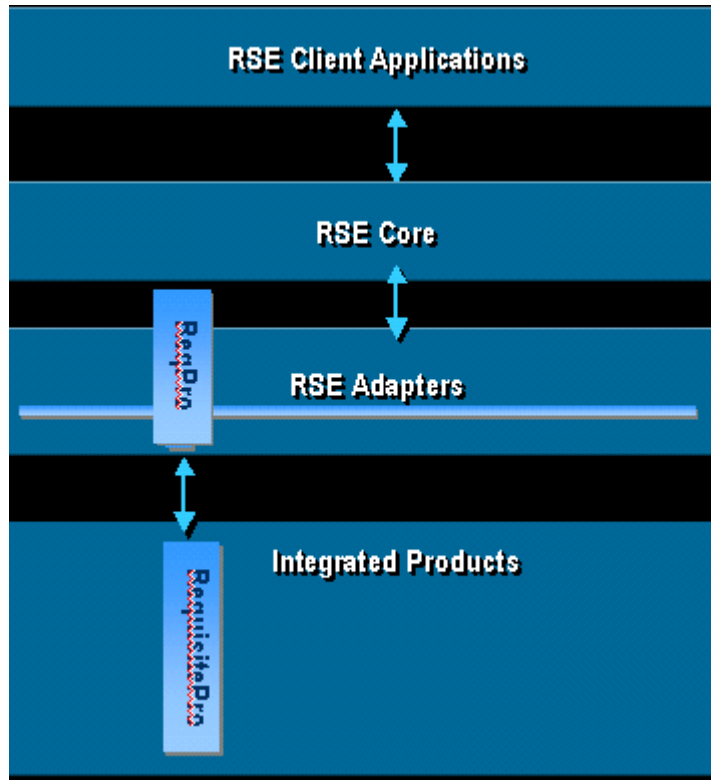


There is a one to one correspondence between artifacts and internal objects. Each artifact is linked to an internal object through the RSE core. The front end of RSE Core works with artifacts, the backend (CPP Framework and the adapters) works with internal objects. Users of the client interfaces only see artifacts. Writers of adapters work with internal objects. This separates functionality and isolates the RSE Core from the adapters.

### **RequisitePro Example**

Figure 10 illustrates a client application retrieving information from RequisitePro through the RSE architecture.

**Figure 10 RequisitePro Example**



In this example, a client application retrieves RequisitePro objects through the ReqPro adapter as follows:

- The RSE core processes the client request and communicates with the ReqPro adapter.
- The ReqPro adapter links the RSE object model to the RequisitePro object model. The adapter acts as a server to RSE clients. The RSE object model maps artifact types to integrated-product objects, using internal object classes.
- The ReqPro adapter gets the RequisitePro object and returns the corresponding internal object. Each internal object class corresponds to an artifact type.
- The RSE core creates the artifact instance and returns it to the client application.

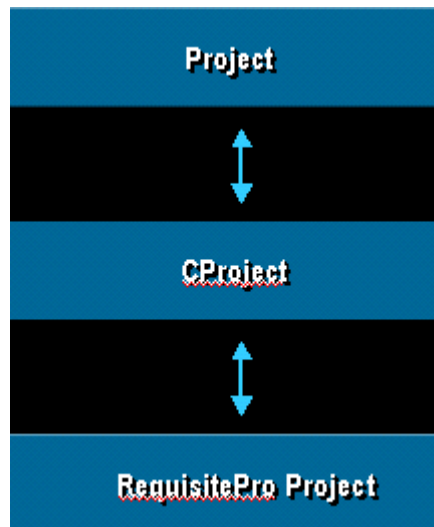
If the client request is for a RequisitePro Project, then the association between objects is as follows:

- The client requests a Project artifact type, passing in the argument for the Project Path in order to locate the Project.

- The ReqPro adapter locates the RequisitePro Project and returns an instance of the associated internal object (CProject).
- The RSE core creates the corresponding artifact instance and returns it to the client.

The ReqPro Project artifact type is defined as the CProject internal object class in the ReqPro adapter. This internal object links to the actual RequisitePro Project object. Figure 11 illustrates the architecture by which a Project artifact type represents a RequisitePro Project object.

**Figure 11 ReqPro Adapter Project Artifact Type**



## Summary

In review:

- Artifacts represent objects in the integrated products. Internal object classes implement this mapping.
- Each artifact type corresponds to an internal object class.
- Each internal object class maps to a specific object type in an integrated product.

For more information see the “Internal Object to Integrated-Product Object” section of this chapter.

## Registering Artifact Types

---

Once you have defined and set up an adapter (as the previous chapter of this manual presents), you register the artifact types for the adapter in its `AdapterInstance` class. The `AdapterInstance` associates each artifact type and its corresponding internal object class.

- The artifact types are registered and located by an adapter's `AdapterInstance` object.
- For each artifact type, you define and implement an internal object class.
- Each internal object class is a `FRWInternalObject`-derived subclass.
- In each subclass, you define an artifact type's properties, relationships, and locators using `InternalObject` methods. You also define the artifact arguments for the artifact references.

### Adapter Instance

The adapter instance object:

- Links an adapter to an integrated-product server.
- Declares the artifact types in an adapter

The adapter instance object locates the internal objects for the integrated product. The adapter instance makes these artifact types available to RSE client application requests. The complete list of subclasses is what the client interface `Adapter.StaticArtifactTypes` method returns.

The adapter instance creates an `InternalObjectTypeRegistrar` object in which it declares its artifact types.

### Declaring and Adding Artifact Types

Each adapter instance registers its artifact types with the `DeclareArtifactTypes` method. Within this overridden method, each artifact type for an adapter is defined with the `AddArtifactType` method of the `InternalObjectTypeRegistrar` class. This registers the name of each artifact type and associates each implementation (internal object) class with its artifact type. The actual registration is done by the static `Register` method of each implementation class.

For an existing adapter, you can add functionality to an adapter in two ways:

- Adding artifact types

Defining and implementing additional internal object subclasses and associating them with artifact type names using the AddArtifactType method.

- Adding functionality to existing artifact types

Implementing additional specifications to existing internal object subclasses.

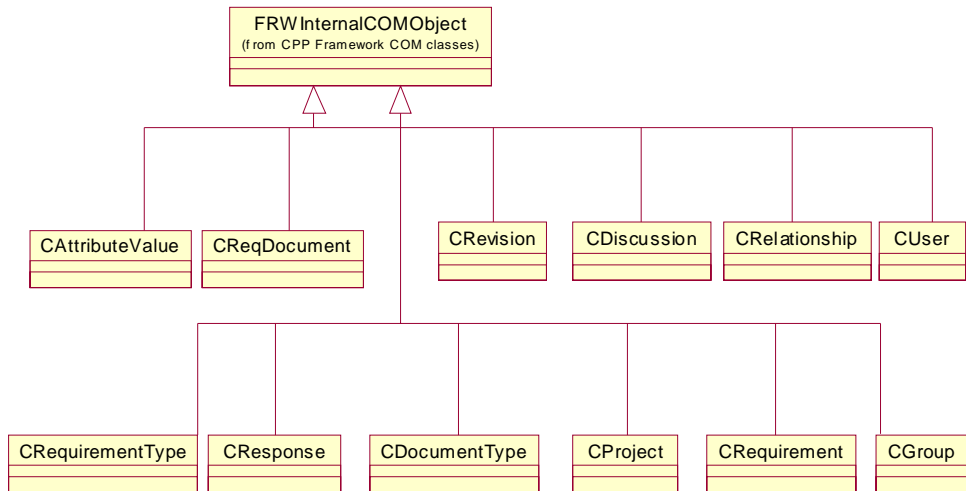
In the InternalObjectRegistrar object, you define method calls that make the calls to an integrated product server. The RSE core implements the method calling mechanism to the C++ Framework.

For descriptions of the available methods in the adapter interfaces, see the “Adapter Interface Methods” chapter of this manual.

## ReqPro Adapter Example

Figure 12 shows the FRWInternalCOMObject subclasses implemented by the ReqPro adapter. (It is not a complete specification of the objects implemented by the ReqPro adapter.) These internal objects map to the ReqPro artifact types that are declared and registered in the ReqProAdapterInstance.

**Figure 12 ReqPro Adapter Internal Objects**



The following implementation classes define the static artifact types in the ReqPro adapter:

- CProject
- CRequirement

- CRelationship
- CRequirementType
- CDiscussion
- CResponse
- CUser
- CGroup
- CAttributeValue
- CReqDocument
- CRevision
- CDocumentType
- CView
- CPermission

## Adding Artifact Types

The following code from the ReqPro adapter defines the Project and Requirement ArtifactTypes. CProject and CRequirement are internal object subclasses. The DeclareArtifactTypes method overrides the method in the AdapterInstance base class of the C++ framework.

```
HRESULT CReqProAdapterInstance::DeclareArtifactTypes
(FRWInternalObjectTypeRegistrar &ObjectTypeRegistrar)
{
    Registrar.AddArtifactType (_T("Project"),
                               NULL,
                               INTERNAL_OBJECT_FACTORY(CProject),
                               INTERNAL_OBJECT_REGISTRAR (CProject)
                               false);
    Registrar.AddArtifactType (_T("Requirement"),
                               NULL,
                               INTERNAL_OBJECT_FACTORY(CRequirement),
                               INTERNAL_OBJECT_REGISTRAR (CRequirement)
                               true);
}
```

The arguments for the AddArtifactType method are:

- Artifact type

The name of artifact type. For example, "Project".

- **Superclass**

The name of the superclass for an artifact type. NULL if there is not a superclass.

- **Implementation class**

The internal object factory. This argument associates an internal object class and the artifact type. The internal object factory instantiates the object. For example, instantiate a CProject internal object.

- **Internal object registration method.**

The Register method is passed the registration object context and constructs the Project registrar object. The registrar object knows how to call methods of the internal object (CProject).

This associates the internal object class and artifact type. For example, in the ReqPro adapter instance, associate the Project artifact type with the CProject internal object class. The implementation of this association is in the Register method in Project.cpp.

- **Abstract**

A Boolean for whether this is an abstract artifact type. For example, Project is not an abstract class and Requirement is an abstract class.

An abstract artifact type is a type that serves as a Superclass only. An abstract artifact type has no instances.

The DeclareArtifactTypes method registers all static artifact types for an adapter. This registers the name and associates the implementation class with the type. The actual registration of properties, relationships, and locators is done by the static Register method of each implementation class.

## Dynamic Artifact Types

Artifact types registered by the adapter namespace are available to all namespaces within the adapter. Dynamic types can be registered using the RegisterDynamicArtifactTypes method of other namespaces implemented by this adapter.

Dynamic artifact types are user-defined artifacts that are instantiated within a container artifact. For example, a ReqPro Project artifact is a dynamic artifact type container for user-defined Requirement artifact types. These dynamic Requirement types, as well as all dynamic Properties and Relationships, are instantiated when the Project is instantiated. This dynamic registration occurs within the Project creation.

Each requirement type has a TagPrefix. The ReqPro adapter represents these types as artifact types named "<TagPrefix>Requirement" (for example UCRequirement or FEATRequirement). The base type for all of the user-defined requirement types is the defined artifact type "Requirement". Any requirement can be viewed through this type.

For more information on InternalObjectTypeRegistrar methods and defining dynamic types, see the "Adapter Interface Methods" chapter of this manual.

## Implementing Artifact Types for an Adapter

---

After defining the artifact types in the AdapterInstance Registrar object with the AddArtifactType method, you implement each artifact type in an InternalObject-derived class.

### Implementing a Class for each Artifact Type

You implement an artifact type by defining the properties, relationships and locators for that type in an InternalObject-derived class Registrar object. You first construct the Registrar object for this class. For example:

```
FRWInternalObjectRegistrar<CRequirement> Registrar (Context);
```

The following code is from the ReqPro adapter. It implements the Requirement artifact type.

```
class CRequirement : public FRWInternalCOMObject
{
public:
...
static void Register (const FRWRegistrationContext &Context);
...
};
```

Register is a static method implemented by FRWInternalObject subclasses. This method registers all properties and relationships for the type.

Dynamic artifact type registration can be contained within a static artifact type class. For example, in RequisitePro, a dynamic requirement type can be registered within the Project registration. The information necessary to register dynamic types is available from the Context argument.



## Registering a Property

Register the properties for this object using the `AddProperty` method of the Registrar object. The Registrar object sets up the framework classes necessary to map requests to get and set property values to the implementation methods.

Each property type registration returns a unique Property ID that you can use later, for registering locator arguments or creation arguments. The property ID return value is an integer.

The following code defines the Text property of a Requisitepro Requirement:

```
/*static*/ void CRequirement::Register (
    const FRWRegistrationContext &Context)
{
    // Construct a Registrar object for this class
    FRWInternalObjectRegistrar<CRequirement> Registrar (Context);

    Registrar.AddProperty ("Text",
        GetText, SetText,
        VT_BSTR,
        frwDescription);
}
```

The arguments for the `AddProperty` method are:

- Property Name
- Get method

The name of the implementation method that retrieves this property from the internal object. The argument for the Get method is a variant value or a null pointer.

- Set method (if not a read-only property)

The name of the implementation method that can modify this property. The argument for the Set method is a variant value or a null pointer.

- Data type

The data type of this property

- Semantic data type

The semantic data type, if any, of this property. A semantic data type is a description of a data type. This provides extra information on a property's or artifact argument's data type. The semantic data types are: rsDataObject, rsDescription, rsDirectory, rsFileMoniker, rsFileOrDirectory, rsFilePath, rsName, rsNone, rsPassword, rsURL, rsUserName.

- Maximum size

The maximum size of this property.

The following code implements the Text property of a RequisitePro Requirement:

```
void CRequirement::GetText(_variant_t &Value)
{
    Value = mInterfacePtr->GetText();
}

void CRequirement::SetText(_variant_t Value)
{
    bstr_t bstrValue = (bstr_t)Value;
    mInterfacePtr->PutText(bstrValue);
    mInterfacePtr->Save ();
}
```

The Get method takes a variant reference argument value. The Set method takes a variant argument value.

To add a property that cannot be modified, you can use the AddProperty\_ReadOnly method. For a complete listing of property type methods, see the “Adapter Interface Methods” chapter of this manual.

## Registering a Relationship

Register the relationships for this object using the AddRelationshipType method of the Registrar object.

The following code defines the AttrValues relationship of a RequisitePro Requirement:

```
/*static*/ void CRequirement::Register (
    const FRWRegistrationContext &Context)
{
    // Construct a Registrar object for this class
    FRWInternalObjectRegistrar<CRequirement> Registrar (Context);
```

...

```
Registrar.AddRelationshipType (Child,  
                               "AttrValues",  
                               "AttributeValue",  
                               GetAttrValues,  
                               ZeroToMany);
```

The arguments for the AddRelationshipType method are:

- Relationship type (RelationshipCategory eCategory)  
For example, Peer, Child, Descendant, or Parent.
- Relationship name
- The artifact type name  
The name of the artifact type that this relationship points to.
- Get method (GetRelatedArtifactMethodPtr pGetMethod)  
This method includes product specific code (for example, RequisitePro extensibility (RPX)) that enumerates the attribute values and returns them in the collection. The argument for the GetMethod is a pointer to the FRWArtifactAdapterCollection.
- Cardinality  
For example, ZeroToMany or ZeroToOne.
- Create method, if any
- Delete method, if any

The arguments in the above example specify that the Requirement artifact type has a Child relationship type named AttrValues to the AttributeValue artifact type. The GetAttr Values method implements this relationship and there can be zero to many attribute values for the Requirement.

The following code implements the AttrValues relationship of a RequisitePro Requirement:

```
void CRequirement::GetAttrValues(FRWArtifactAdapterCollection *  
pObjects)  
{  
    long lCount = 0;  
    long lStartIndex = 1;
```

```

// Get the collection object
_AttrValuesPtr lAttrColl = mInterfacePtr->GetAttrValues();
lCount = lAttrColl->GetCount();
int AttributeValueTypeID = FindArtifactTypeID("AttributeValue");
_AttrValuePtr lItemPtr;

for (long i = lStartIndex; i <= lCount; i++)
{
    //Get the ith item from the collection
    _variant_t lVar(i);
    lItemPtr = lAttrColl->GetItem(lVar,eAttrValueLookup_Index);
    if (lItemPtr == NULL) continue;
    pObjects->Add(AttributeValueTypeID, lItemPtr, this);
}
}

```

A relationship type can return the collection of arguments for creating an artifact. These arguments are defined in the adapters as creation arguments.

For a complete listing of relationship type methods, see the “Adapter Interface Methods” chapter of this manual.

## Registering a Locator

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

The implementation of a locator consists of:

- Locator registration

The locator registration defines the locator for the given artifact type being registered. This registration defines the locator and the properties of the artifact that are to be used as parameters. When developing new adapters, every locator parameter must be registered as a property of the locator's artifact type. For the existing Suite integrated product adapters, every locator parameter is registered as a property of the artifact.

- Locator resolution

Artifact reference resolution is the mechanism that the adapters use to pass references from artifacts to the RSE core. Locators provide consistent support for use case management (UCM) references.

## Defining a Locator

Register the locators and the locator arguments using the `AddRelativeLocator` (or `AddAbsoluteLocator`) and `AddLocatorArgument` methods of the Registrar object.

You use the `Add` locator methods to define each locator for each artifact type. The `AddLocatorArgument` method defines each argument for an artifact locator. These arguments can map to artifact properties, be a defined argument such as `UserName` or `Password`, or map to an internal property ID.

The following code defines the `Display Name` Locator of a `RequisitePro Requirement`. This locator is a relative locator, relative to `Project`. `Project` is the context artifact.

```
/*static*/ void CRequirement::Register (
    const FRWRegistrationContext &Context)
{
    // Construct a Registrar object for this class
    FRWInternalObjectRegistrar<CRequirement> Registrar (Context);
    ...
    Registrar.AddRelativeLocator (LOCATE_WITH_TAG,
        "Project",
        LocateWithTag,
        LOCATE_DEFAULT_DISPLAY_NAME);
    Registrar.AddLocatorArgument (LOCATE_WITH_TAG, 1, "FullTag");
}
```

The arguments for the `AddRelativeLocator` method are:

- Locator ID

The string definition for the locator. In the above example, `LOCATE_WITH_TAG` is the locator ID.

- Relative artifact type name

This is the context artifact type for a client application. `Project` is the relative artifact type. In the above example, this locator is relative to `Project`.

- Locate method

The method that implements the locator. In the above example, `LocateWith Tag` is the method that implements this locator.

- Locator flag

In the above example, `LOCATE_DEFAULT_DISPLAY_NAME` is the locator flag, specifying that this locator is a default display name locator.

The `LOCATE_WITH_TAG` Requirement type locator is a relative locator that has one locator argument. `FullTag` is the artifact property that maps to this locator argument.

You can also define locators that are not relative to another artifact. These are defined as Absolute locators. For more information, see the “Adapter Interface Methods” chapter of this manual.

These are common combinations that may be used when registering locators:

```
#define LOCATE_DISPLAY_NAME_AND_ID
LOCATE_DISPLAY_NAME | LOCATE_IMMUTABLE_ID

#define LOCATE_DEFAULT_DISPLAY_NAME_AND_ID
LOCATE_DEFAULT_IMMUTABLE_ID | LOCATE_DEFAULT_DISPLAY_NAME
```

## Defining Locator Arguments

The `AddLocatorArgument` method specifies the artifact arguments for constructing the locator. There are three formats of this method:

- Mapping an argument to an artifact property ID.
- Mapping an argument to an artifact property name.
- Specifying an argument that is not a property.

In the previous code example, for the display name locator definition of the `RequisitePro Requirement` artifact type,

```
Registrar.AddLocatorArgument (LOCATE_WITH_TAG, 1, "FullTag")
```

the `AddLocatorArgument` defines a locator argument:

- `LOCATE_WITH_TAG` is the locator ID for this relative locator.
- “1” is the param argument. The “1” represents the ordinal of the parameter (that is, first = 1, second = 2, third = 3, and so on).

This argument is mapped to the `FullTag` property. This allows the RSE Core to get the proper argument value when it forms a locator for an existing Requirement object.

- `FullTag` is the name of the artifact property that maps to this argument.

The following code implements the Display Name Locator of the ReqPro Requirement.

```
bool CRequirement::LocateWithTag ( FRWInternalObject *pRelativeObject,
                                   const FRWArguments &Params,
                                   FRWInternalObjectReference &Context)
{
    CProject *pProject = dynamic_cast<CProject*> (pRelativeObject);
    _variant_t vtTag = Params.GetArg (1);
    // Product specific code...
    _ProjectPtr pInternalProject = pProject->GetInternalProjectPtr ();
    int iTypeID;
    _RequirementPtr pRequirement;
    pRequirement = pInternalProject->GetRequirement(vtTag,
                                                    ReqPro40::eReqLookup_Tag,
                                                    ReqPro40::eReqWeight_Medium,
                                                    ReqPro40::eReqFlag_Empty)

    if (pRequirement == NULL)
        return false;
    iTypeID =
        pProject->FindArtifactTypeID(GetArtifactTypeName(pRequirement));

    _ASSERT (iTypeID != 0);

    if (iTypeID == 0)
        return false;

    Context.Attach(iTypeID, NewRequirement(pRequirement, pCProject));
    return true;
}
```

The LocateWithTag method has three arguments:

- pRelativeObject  
This is the relative object. The locator locates an object starting at this context. In this example, Project is the relative artifact.
- Params

The locator arguments. Params contains the parameters to the locator that are used to find an object. In this case there is only one parameter, the tag.

- Context

The context object is the object to be located. Context contains information that is used to attach the located object (represented by an internal object) to an RSE artifact that represents that object.

In the call to the Attach method, the internal object is created by the call to NewRequirement. The iTypeID parameter identifies the type of artifact that is being created.

The Attach method saves the parameter passed to it in a member variable of the object type so that the object can access the corresponding integrated product object when necessary.

## Defining a Collection of Artifact Locators

When developing a new adapter, every locator argument must be registered as a property of the locator's artifact type. The locator arguments are necessary to construct the artifact locator. For example, you need the Path of a Rose model in order to construct a Model artifact locator that locates the Model.

This section shows how a portion of the Rose adapter could be implemented. Table 1 lists artifact types and the artifact arguments for each of these artifact type locators.

**Table 1** Locator Arguments

Artifact Type	Artifact Locator Arguments
Model	Path
Package	Name
	Qualified Name
Class	Name
Item	UniqueID

The adapter defines the Item class as a superclass of the Model, Package and Class. Therefore, any Item locators also apply to the Model, Package and Class types.



## Locator Definitions

This section provides locator definitions for a portion of the Rose adapter.

The Model registers one locator with one parameter, the path. The locator is registered as an absolute locator. The following table describes the locator:

**Table 2 Locate Model by Path**

Model Locator Type	Locator String
Display Name	Rose   Model(Path="")
Immutable ID	Rose   1   Model("")

The Package registers two locators, one by Name and one by Qualified name. Each locator is relative to the model. The following tables describe the locators:

**Table 3 Locate Package by Name**

Package Locator Type	Locator String
Display Name	Rose   Model(Path="")   Package(Name="")
Immutable ID	Rose   1.1   Package ("", "")

**Table 4 Locate Package by Qualified Name**

Package Locator Type	Locator String
Display Name	Rose   Model(Path="")   Package(QualifiedName="")
Immutable ID	Rose   1.2   Package ("", "")

The Class registers one locator relative to the Package. Since the Package has two locators, the Class adds its relative definition to each. The following tables describe the locators:

**Table 5 Locate Class, Relative to Package Name**

Class Locator Type	Locator String
Display Name	Rose   Model(Path="")   Package(Name="")   Class(Name="")
Immutable ID	Rose   1.1.1   Class ("", "", "")

**Table 6 Locate Class, Relative to Package Qualified Name**

Class Locator Type	Locator String
Display Name	Rose   Model(Path="")   Package(QualifiedName="")   Class(Name="")
Immutable ID	Rose   1.2.1   Class("","","")

Lastly, the Item registers one locator relative to the Model. The following table describes the locator:

**Table 7 Locate Item, Relative to Model**

Item Locator Type	Locator String
Display Name	Rose   Model(Path="")   Item("UniqueID="")
Immutable ID	Rose   1.1   Model("","")

As a result, the following display name locators are supported by the following objects:

**Table 8 Supported Display Name Locators**

Artifact Locator Type	Locator String
Item	Rose   Model(Path="")   Item("UniqueID="")
Model	Rose   Model(Path="")
Package	Rose   Model(Path="")   Package(Name="")
	Rose   Model(Path="")   Package(QualifiedName="")
	Rose   Model(Path="")   Item("UniqueID="")

Artifact Locator Type	Locator String
Class	Rose   Model(Path='')   Package(Name='')   Class(Name='')
	Rose   Model(Path='')   Package(QualifiedName='')   Class(Name='')
	Rose   Model(Path='')   Item("UniqueID=")

For a complete listing of locator methods and their arguments, see the “Adapter Interface Methods” chapter of this manual.

## Registering Creation Arguments

Creation arguments are any arguments required to create an artifact. This applies only in cases where creation of new artifacts is supported. The client must specify the values of these arguments when creating an artifact. The adapter uses these arguments to create the artifact.

The situations where a new artifact is created can be:

- Creation of a dynamic type contained within a static artifact type (for example, a ReqPro dynamic Requirement type created within the Project artifact type)
- Creation of an instance of a static artifact type (for example, creating a new Project artifact).

In either of the two situations, the adapter is responsible for specifying what the required arguments are by calling `AddCreationArgument`:

- Creation of a dynamic type requires the arguments to be registered with `InternalObjectTypeRegistrar::AddCreationArgument`.
- Creation of a static type requires the arguments to be registered with `InternalObjectRegistrar::AddCreationArgument`.

A creation method is registered through a call to a `CreateArtifact` method (for example, `CreateRequirement` or `CreateDynamicRelatedArtifact`). The types of artifacts created may be dynamic or static.

In the ReqPro adapter, the Project class provides examples of `CreateArtifact` and `CreateDynamicRelatedArtifact`. There are creation arguments for both static and dynamic artifact types. Note that `CreateRequirement` is specific to the ReqPro adapter whereas `CreateDynamicRelatedArtifact` applies to adapters in general.

The `CreateRequirement` method takes three parameter arguments, `ArtifactTypeName`, `Name`, and `Text`. These arguments are registered creation arguments for creating a new requirement:

```

Registrar.AddCreationArgument("Requirements", 1, "TypeName",
VT_BSTR, frwNone);

Registrar.AddCreationArgument("Requirements", 2, "Name", VT_BSTR,
frwName, -1, "New Requirement");

Registrar.AddCreationArgument("Requirements", 3, "Text", VT_BSTR,
frwDescription, -1, "This is the text.");

```

When you register creation arguments, each argument is assigned an ordinal parameter id. For example, if three creation arguments (A, B, and C) are registered, they are assigned parameter ids 1 (for A), 2 (for B), and 3 (for C).

CreateDynamicRelatedArtifact takes two parameter arguments, Name and Text. These arguments are registered creation arguments for creating dynamic related artifacts:

```

Registrar.AddCreationArgument(iProjectTypeID, bstrRelationshipName,
1, "Name", VT_BSTR, frwName, -1, "New Requirement");

Registrar.AddCreationArgument(iProjectTypeID, bstrRelationshipName,
2, "Text", VT_BSTR, frwDescription, -1, "This is the text.");

```

For more information, see the "Adapter Interface Methods" chapter of this manual.

## Using the Maps Mechanism

---

The CPP Framework provides an alternative Maps mechanism that simplifies artifact type registration and property implementation in adapters. You can register the properties, relationships, and graphics format types for the artifact types in an adapter, using the mapping mechanism that is provided in the C++ Framework. This mechanism simplifies the registration for an adapter.

The Maps mechanism is supported by a collection of macros and declarations in the include files Maps.h and InternalObject.h in the directory, rdsi\adapters\CPP Framework. The Maps mechanism (referred to henceforth as just, Maps) is built on top of the normal CPP Framework interfaces that register types. The primary advantages of using Maps are that:

- The registration of properties and relationships is more compactly achieved
- Properties that simply return point product values "as is" are automatically implemented.
- The declaration of properties and relationship can be made more compact.

The easiest way to use Maps is to emulate an existing example. The FileSys adapter provides a simple example. The MSPProject adapter provides a richer example. Note that the MSPProject adapter also demonstrates the value of defining new integrated product-specific declaration macros. See the Root.h file in MSPProject adapter project as an example.

## Registering Maps

Maps registration occurs in the Register function of an artifact type with the following macro:

```
REGISTER_MAPS (typeName, Context)
```

The arguments are the artifact type name and the registration context.

The FileSys adapter Directory artifact type implementation class, CDirectory, uses maps for its property and relationship registration. In the registration of this artifact type, in Directory.cpp, the following statement registers these maps:

```
REGISTER_MAPS (Directory, Context)
```

The mapping functionality is included in the implementation classes with the include statement:

```
#include "CPP Framework/Maps.h"
```

## Declaring Artifact Types

Maps provides the following macros for declaring artifact types (usually used in the adapter instance .cpp file):

- DECL\_ARTIFACT\_TYPE (typeName)
- DECL\_ARTIFACT\_SUBTYPE (typeName, superclassName)

where:

- typeName – the name of the artifact type declared
- superclassName – the super type for an artifact subtype

For example, in the FileSys adapter, Directory is a subtype of DirectoryObject. The superclass declares itself as follows:

```
DECL_ARTIFACT_TYPE (DirectoryObject)
```

The DECL\_ARTIFACT\_SUBTYPE statement enables subtypes to inherit from the superclass. The following statement enables Directory to inherit all of DirectoryObject's properties and relationships.

```
DECL_ARTIFACT_SUBTYPE (Directory, DirectoryObject)
```

The arguments for this method are:

- subclass
- superclass

## Defining Artifact Types

When using Maps, the implementation of an artifact type has the following structure:

```
DECL_ARTIFACT_TYPE (<artifact type name>)

BEGIN_PROPERTIES (<artifact type name>)
    <property macro>...
END_PROPERTIES

BEGIN_OVERRIDE_PROPERTIES (<artifact type name>)
    <property override macro>...
END_OVERRIDE_PROPERTIES

BEGIN_RELATIONSHIPS (<artifact type name>)
    <relationship macro>...
END_RELATIONSHIPS

BEGIN_OVERRIDE_RELATIONSHIPS (<artifact type name>)
    <relationship override macro>...
END_OVERRIDE_RELATIONSHIPS

BEGIN_GRAPHIC_FORMATS (<artifact type name>)
    <graphic format macro>...
END_GRAPHIC_FORMATS

<remainder of implementation>
```

Note that the implementation is divided into a number of sections delimited by `BEGIN_<section name>` and `END_<section name>`. Each of these sections is required but may be empty. It is recommended that the comment `// none` be

inserted in the event of an empty section. The <artifact type name> is the name of the artifact being defined and is the same for each section. Within each section are none or more calls to appropriate macros defined in the Maps.h file.

For example, in Directory.cpp:

```
DECL_ARTIFACT_TYPE (Directory, DirectoryObject)

BEGIN_PROPERTIES (Directory)
    propertyRS (DirectoryPath, VT_BSTR, frwDirectory)
END_PROPERTIES

BEGIN_OVERRIDE_PROPERTIES (Directory)
    // none
END_OVERRIDE_PROPERTIES

BEGIN_RELATIONSHIPS (Directory)
    naryR (Contents, DirectoryObject, Child)
END_RELATIONSHIPS

BEGIN_OVERRIDE_RELATIONSHIPS (Directory)
    // none
END_OVERRIDE_RELATIONSHIPS

BEGIN_GRAPHIC_FORMATS (Directory)
    // none
END_GRAPHIC_FORMATS
```

## Definition Registration Macros

The mapping mechanism provides macros for registering artifact type properties, relationships and graphics format types. For properties and relationships there are additional macros for overriding inherited properties and relationships.

## Registering Properties

A <property macro> defines a property and can take any of the following forms:

- `property(name, dataType)`  
Register a property type.
- `propertyR(name, dataType)`  
Register a read-only property type.
- `propertyS(name, dataType, semType)`  
Register a semantic property type.
- `propertyRS(name, dataType, semType)`  
Register a read-only semantic property type.

The arguments are :

- `name` – name of the property defined
- `dataType` – VARIANT type of property (for example, `VT_BSTR` or `VT_INT`)
- `semanticType` – the “frw” data type (for example, `frwName`) (`frwNone` when omitted).

Each <property macro> defines a property with a specified name, data type, and semantic type. The R indicates a read-only property is defined. S simply distinguishes those macros that have the `semanticType` argument.

**Note:** The naming conventions for maps property macros are as follows:

- R is Read-only
- S is Semantic type

## Registering Properties Example

The FileSys Directory artifact type registers the one property that is specific to Directory as follows:

```
BEGIN_PROPERTIES (Directory)
    propertyRS (DirectoryPath, VT_BSTR, frwDirectory)
END_PROPERTIES
```

The `propertyRS` method registers a read-only property with a specified semantic type. The arguments for this method are:

- The property name



- The data type of this property
- The semantic type of this property

The following `getMethod` implements the `DirectoryPath` property:

```
void CDirectory::GetDirectoryPath(_variant_t &Value)
{
    CDirectoryObject::GetPath (Value);
}
```

### Registering Override Properties

A property override is the registration of a property that overrides a property of an object's superclass. A `<property override macro>` overrides the implementation of an inherited property and has the forms:

- `override_property(name)`
- `override_propertyR(name)`

where:

- `name` is the property name
- `R` designates a read-only property

### Registering Relationships

An object can have a relationship to one (unary) or more than one (nary) instances of a related artifact type.

- Unary is a 0..1 relationship
- Nary is a 0..n relationship

A `<relationship macro>` defines a relationship and unary or nary.

**Note:** The naming conventions for map relationship macros are as follows:

- `F` is Filtered
- `R` is Read-only
- `S` is Semantic type

### Registering a Unary Relationship

A unary `<relationship macro>` can take any of the following forms:

- `unary(name, relatedType, relCategory)`

Register a unary relationship type.

- unaryF(name, relatedType, relCategory)

Register a filtered unary relationship type.

- unaryR(name, relatedType, relCategory)

Register a read-only unary relationship type.

- unaryS(name, relatedType, relCategory, semType)

Register a semantic unary relationship type.

- unaryFR(name, relatedType, relCategory)

Register a filtered read-only unary relationship type.

- unaryFS(name, relatedType, relCategory, semType)

Register a filtered semantic unary relationship type.

- unaryRS(name, relatedType, relCategory, semType)

Register a read-only semantic unary relationship type.

- unaryFRS(name, relatedType, relCategory, semType)

Register a filtered read-only semantic unary relationship type.

The arguments are:

- name – name of the relationship defined
- relatedType – artifact type of the related artifacts
- card – cardinality (ZeroToOne or ZeroToMany)
- semType – semantic type of relationship (usually frwNone)
- relCategory – relationship category (such as, Child, Peer)

In the FileSys adapter DirectoryObject artifact type, is the following unary relationship:

```
BEGIN_RELATIONSHIPS (DirectoryObject)
    unaryR (ParentDirectory, Directory, Peer)
END_RELATIONSHIPS
```

The unaryR method registers a read-only relationship with Cardinality of zero to one (unary). The arguments for this method are:

- The relationship name

- The related artifact type
- The relationship type (relCategory)

### Registering an nary Relationship

An nary <relationship macro> can take any of the following forms:

- nary(name, relatedType, relCategory)  
Register an nary relationship type.
- naryF(name, relatedType, relCategory)  
Register a filtered nary relationship type.
- naryR(name, relatedType, relCategory)  
Register a read-only nary relationship type.
- naryS(name, relatedType, relCategory, semType)  
Register a semantic nary relationship type.
- naryFR(name, relatedType, relCategory)  
Register a filtered read-only nary relationship type.
- naryFS(name, relatedType, relCategory, semType)  
Register a filtered semantic nary relationship type.
- naryRS(name, relatedType, relCategory, semType)  
Register a read-only semantic nary relationship type.
- naryFRS(name, relatedType, relCategory, semType)  
Register a filtered read-only semantic nary relationship type.

The arguments are:

- name – name of the relationship defined
- relatedType – artifact type of the related artifacts
- card – cardinality (ZeroToOne or ZeroToMany)
- semType – semantic type of relationship (usually frwNone)
- relCategory – relationship category (such as Child, Peer)

For example, the relationships specific to Directory (there is only one called, Contents) are listed as follows:

```
BEGIN_RELATIONSHIPS (Directory)
```

```
naryR (Contents, DirectoryObject, Child)
END_RELATIONSHIPS
```

This defines a Child relationship with cardinality of one to many (0..n), named Contents, pointing to DirectoryObject. The naryR method registers a read-only relationship. The arguments for this method are:

- The relationship name
- The related artifact type
- The relationship type (relCategory)

The following getMethod implements the Contents relationship:

```
void CDirectory::GetContents(FRWArtifactAdapterCollection *
pObjects)
{
    GetSubFolders(pObjects);
    GetFiles(pObjects);
}
```

### Registering an Override Relationship

A relationship override is the registration of a relationship that overrides a relationship of an object's superclass. A <relationship override macro> overrides the implementation of an inherited relationship and takes one of these forms:

- `override_relationship(name)`  
Override a relationship
- `override_relationshipF(name)`  
Override a filtered relationship

## Registering Graphics Format Types

A `<graphic format macro>` defines a graphic format for an artifact that has an image and has the form:

```
format (aspect , formatType , description)
```

The arguments are:

- aspect  
The type of object rendered, generally “File.”
- formatType  
The name of the graphic format. WMF, JPG, PNG, BMP, GIF, or ASIS.
- description  
A textual description of the rendering (for example, “Render file as WMF”)

For information on using the mapping mechanism for graphic formats type registration, see the “Adapter Interface Methods” chapter of this manual.

## Handler Declaration Macros

The Maps mechanism also provides a number of macros used to declare the property and relationship handler methods. These macros, defined in the file `InternalObject.h`, are usually used in the “.h” file for the artifact type:

- `DECL_PROP(name)`  
Declare a read/write property.
- `DECL_PROPr(name)`  
Declare a read-only property.
- `DECL_NARYr(name)`  
Declare a read-only nary relationship.
- `DECL_NARY(name)`  
Declare a read/write nary relationship.
- `DECL_NARYfr(name)`  
Declare a filtered read-only nary relationship.

- DECL\_NARYf(name)  
Declare a filtered read/write nary relationship.
- DECL\_UNARYr(name)  
Declare a read-only unary relationship.
- DECL\_UNARY(name)  
Declare a read/write unary relationship.

## Pass-Through Property Definitions

When the property data returned by a point product is to be used “as is”, the handler function(s) can be generated automatically by using one of the following declaration macros instead of those described in the preceding section.

**Note:** The conventions for pass-through property definition macros are as follows:

- 0 (zero) indicates the type parameter ‘t’ (for type) is included
- r is Read-only
- m indicates that handler methods are to be called to get and set the property
- DECL\_PASS\_THROUGH\_PROP0(name, propName, t)  
Declare a property with a property conversion type.
- DECL\_PASS\_THROUGH\_PROP0r(name, propName, t)  
Declare a read-only property with a property conversion type.
- DECL\_PASS\_THROUGH\_PROP(name)  
Declare a property type.
- DECL\_PASS\_THROUGH\_PROPr(name)  
Declare a read-only property type.
- DECL\_PASS\_THROUGH\_PROP0m(name, getName, putName, t)  
Declare a property using integrated-product handler methods and a property conversion type.
- DECL\_PASS\_THROUGH\_PROP0mr(name, getName, t)  
Declare a read-only property using integrated-product handler method.
- DECL\_PASS\_THROUGH\_PROPM(name)  
Declare a property using integrated-product handler methods.

- DECL\_PASS\_THROUGH\_PROPmr(name)

Declare a read-only property type using integrated-product handler method.

The arguments are:

- name – the name of the property seen by the adapter user
- propName – the point product name of the property (when different)
- getName – the interface to call to get the data when not a point product property
- putName – the interfact to call to put the data when not a point product property
- t - the type to which the property is to be converted, if necessary (for example, bool, int)

## Internal Object to Integrated-Product Object

---

The internal objects represent the integrated-product objects that have their own extensibility interface for access to them. Each instance of an internal object class represents an actual integrated-product object. For example the CProject ReqPro internal object links RSE to a RequisitePro Project through the RPX API.

**Note:** The adapter objects that represent integrated-product objects are referred to as internal objects; product-specific objects are referred to as COM IUnknowns.

In most cases, there is a one to one mapping between artifacts in RSE and COM objects in the point product's API. In the adapters, when you create an internal object to represent an artifact, you also store a pointer in that internal object to the corresponding COM object in the integrated product.

When an adapter receives a request for a particular property or relationship of an artifact, the adapter accesses this pointer in the internal object and uses it to invoke the COM object in the integrated product to retrieve the required property or relationship data. When retrieving a:

- Property - the data returned by the integrated product is a value, typically a string.
- Relationship - the data returned by the point product is a collection of COM pointers (or one COM pointer for a relationship with a cardinality of 0..1).

For each COM pointer in a collection, the adapter creates a new internal object to represent the COM object and stores the COM pointer in it.

The linking between an internal object and the corresponding integrated-product object (represented as an IUnknown for COM) occurs in the Attach method. This method sets up the connection between the adapter and the integrated-product server.

In the following example, from the ReqPro Project internal object class:

- `m_pProject` is an RPX (a RequisitePro-specific API) pointer to a RequisitePro Project (after it is set by the following routine). This is the pointer to the corresponding COM object stored in the internal object and it is used to get data from the point product.
- `pInternalObject` is the value used to set `m_pProject` in the Attach method.

```
HRESULT CProject::Attach(IUnknown *pInternalObject)
{
    m_pProject = pInternalObject
    m_FullName = (char*) m_pProject->GetRQSFilePath();
    LockProject();
    return FRWInternalCOMObject::Attach(pInternalObject);
}
```

The Project internal object class is created and is mapped to the object in the integrated-product, referenced by `m_pProject`.

## Getting an Application Object

An RSE client application can retrieve properties and relationships of an artifact type through the IArtifact interface. It can also get the internal object that an artifact type represents, using the `Artifact.GetInternalObject` method. With this internal object, a client can call integrated-product API methods directly.

In order to retrieve an object in an integrated product, you must first get the product's application, or server, object.

In the following RequisitePro-specific extensibility (RPX) code, located in the `ReqProAdapterInstance` class, RSE is calling the RPX connector class (getting an instance of a connector object) and asking for the application object (COM server). In this example:

- `mAppPtr` is a pointer, of RequisitePro type `_ApplicationPtr`, to the RequisitePro Application object.
- `connector` is an instance of the connector object.



- ReqPro40 is the namespace for the COM server object.

```

_ApplicationPtr CReqProAdapterInstance::GetReqProApplication()
{
    return GetAdapterInstance()->GetReqProApplicationInternal();
}
_ApplicationPtr CReqProAdapterInstance::GetReqProApplicationInternal()
{
    try {
        if (mAppPtr == NULL) {
            ReqPro40::_ConnectorPtr connector("ReqPro40.Connector");
            if (connector) {
                VARIANT_BOOL bAutoStart = -1;
                mAppPtr = connector->GetApplication(&bAutoStart);
                connector = NULL;
            }
        }
        return mAppPtr;
    }
}

```

## Adapter Internals

---

This section provides a brief description of the communication between the RSE core and the RSE adapters.

### C++ Framework Classes

The C++ implementation of the adapter interfaces handles events by either calling methods of the artifact implementation class, or by calling methods of the internal COM object directly.

In the C++ framework, the adapter interfaces contain objects that optimize and simplify the implementation of the artifact class, unifying all of the code necessary in one class.

- CAdapterInstance

This class uses the AdapterInstance COM object to communicate with the RSE Core. When implementing adapters, developers derive the adapter instance objects from this class.

- CInternalObjectRegistrar  
This class uses the ArtifactAdapter COM object to communicate with the RSE Core. When implementing adapters, developers derive the artifact objects from this class.
- CArtifactRegistrar  
This class uses the IArtifactRegistrar interface to communicate with the RSE Core.
- CPropertyMap  
The CArtifactAdapter uses this class to get and set the value of properties. It is a virtual base class.
- COverriddenPropertyMap  
This class is able to call methods of the CArtifactAdapter subclasses to get and set properties. It is derived from CPropertyMap.
- CRelatedArtifactMap  
The CArtifactAdapter uses this class to create, delete and get instances of artifacts. It is a virtual base class.
- COverriddenRelatedArtifactMap  
This class is able to call methods of the CArtifactAdapter subclasses to create, delete and get instances of artifacts. It is derived from CRelatedArtifactMap.

## Adapter Operations

The RSE Core initializes the adapter by creating an instance of its adapter object. The core then calls RegisterArtifactTypes, passing an instance of the IArtifactRegistrar interface.

The type registration is simplified through the C++ framework interfaces. The implementation of the adapter object calls the RegisterArtifactTypes method of its CAdapterInstance class member. This method walks through a static artifact type map in order to register the statically defined artifact types for the adapter.

Note: Dynamic types are registered when instances of the adapters are created. The RegisterDynamicTypes method is called on the adapter at that time.

Table 9 summarizes the adapter interfaces operational model.

**Table 9 Adapter Operations**

<b>Operation</b>	<b>Description</b>
Initialization	The RSE Core initializes the adapter by creating an instance of its adapter object. The core then calls RegisterArtifactTypes, passing an instance of the IArtifactRegistrar interface.
Static Artifact Type Registration	The adapter calls the RegisterType method of the IArtifactRegistrar interface when an adapter initializes.
Dynamic Artifact Type Registration	The adapter calls the RegisterType method of the IArtifactRegistrar interface when an adapter creates an ArtifactAdapter object. When the RDSI core creates the ArtifactAdapter, the core calls the RegisterDynamicTypes method of the ArtifactAdapter.
Static Property Registration	The adapter calls the RegisterProperty method of the IArtifactRegistrar interface when initializing the adapter.
Dynamic Property Registration	The adapter calls the RegisterProperty method of the IArtifactRegistrar interface when an adapter creates an ArtifactAdapter object.
Getting the value of a static or a dynamic property	The Artifact Object calls the GetPropertyValue method of the IArtifactAdapter interface. The implementation of this method returns the value of the property.
Setting the value of a static or dynamic property	The Artifact Object calls the SetPropertyValue method of the IArtifactAdapter interface.
Getting child or related artifacts	The Artifact Object calls the GetArtifacts method of the IArtifactAdapter interface.
Creating child or related artifacts	The Artifact Object calls the CreateArtifact method of the IArtifactAdapter interface.
Deleting child or related artifacts	The Artifact Object calls the DeleteArtifact method of the IArtifactAdapter interface.

## Adapter Interfaces

The adapter interfaces provide the mechanism for communication between each adapter and the RSE core. The following adapter objects are COM objects used by the RSE Core to communicate with the adapter. This bridges the component boundary between the RSE Core and the adapter.

- AdapterInstance

This object returns information about the adapter and the static metadata available from the adapter. It can be instantiated without creating an instance of the integrated product server.

- ArtifactAdapter

This object allows the IArtifact implementation in the RSE Core to communicate with the integrated product. It is a translator between the RSE Client interfaces and the integrated product interfaces.

- ArtifactCollectionAdapter

This object interfaces with a integrated product collection to allow artifacts to be created at the time the collection is iterated instead of during the GetRelatedArtifacts or GetChildArtifacts call.

- ArtifactRegistrar

This object allows the adapters to register artifact properties, related types and child types in the RSE Core.

# Adapter Interface Methods

# 5

The methods for defining and implementing artifact types, and each artifact type's properties, relationships, and locator strings are in the following objects:

- `InternalObjectTypeRegistrar`  
Defines an artifact type and any contained dynamic types. This class also includes the methods for defining dynamic properties and relationships for these dynamic types.
- `InternalObjectRegistrar`  
Defines a static artifact type's properties, relationships, and locator arguments.

## InternalObjectTypeRegistrar

---

The `InternalObjectTypeRegistrar` class contains the methods for creating a registrar object and defining the artifact types for an adapter.

**Table 10 InternalObjectTypeRegistrar Methods**

Method	Description
<code>AddArtifactType</code>	Defines an artifact type for an adapter.
<code>AddCreationArgument</code>	Defines arguments that are needed for creating artifact types by a dynamic relationship type. Maps a creation argument to an Argument name.
<code>AddCreationPropertyArgument</code>	Defines arguments that are needed for creating artifact types by a dynamic relationship type. Maps a creation argument to an artifact Property Name or Property ID.
<code>AddDynamicProperty</code>	Registers a property for a dynamic artifact type.
<code>AddDynamicProperty_ReadOnly</code>	Registers a readonly property for a dynamic artifact type.
<code>AddDynamicRelationshipType</code>	Registers a relationship of a dynamic artifact type.

**Table 10 InternalObjectTypeRegistrar Methods**

Method	Description
FRWInternalObjectTypeRegistrar	Creates a registrar object.

## FRWInternalObjectTypeRegistrar

Constructs a registrar object to declare the artifact types for an adapter.

```
FRWInternalObjectTypeRegistrar(IFRWTypeContainerBase  
*pTypeContainerBase)
```

For example:

```
HRESULT CReqProAdapterInstance::DeclareArtifactTypes  
(FRWInternalObjectTypeRegistrar &ObjectTypeRegistrar)
```

## AddArtifactType

Defines an artifact type for an adapter.

```
int AddArtifactType(const wchar_t *pTypeName,  
                  const wchar_t *pSuperclassName,  
                  FRWInternalObjectFactory *pFactory,  
                  FRWInternalObjectRegistrationMethod *pRegistrar  
                  bool bAbstract = false)
```

Registers an artifact type. Associates an artifact type with a class. For example in the ReqPro, this method associates a Project artifact type with a RequisitePro CProject object. The object factory instantiates the object.

The arguments for the AddArtifactType method are:

- Artifact type  
The name of artifact type.
- Superclass  
The artifact type's superclass artifact type name. NULL if there is not a superclass.
- Implementation class
- Internal object registration method.  
The Register method is passed the registration object context and constructs the Project registrar object. The registrar object knows how to call methods of the internal object (CProject).

- **bAbstract**

Specifies whether the artifact type is an abstract type or not. The default value is `false`.

For example:

```
ObjectTypeRegistrar.AddArtifactType (_T("Project"),NULL,NULL,  
INTERNAL_OBJECT_REGISTRAR (CProject));
```

## AddCreationArgument

Defines arguments that are needed for creating artifact types by a dynamic relationship type. Maps a creation argument to a registered artifact argument name. This method is for defining arguments that are needed for creating artifacts.

```
bool AddCreationArgument (int iArtifactTypeID,  
                          const wchar_t * pRelationshipTypeName,  
                          int ParamID,  
                          const wchar_t * ArgumentName,  
                          int DataType,  
                          SemanticDataType iSemanticDataType = frwNone,  
                          int MaxSize = -1,  
                          _variant_t vtDefaultValue = _variant_t(),  
                          bool bRequired = true);
```

The arguments are:

- **Artifact type ID**
- **Relationship type name**
- **ParamID**

The parameter id. This numeric index corresponds to the argument number (in a list of registered locator arguments).

- **Property ID**
- **Variant reference**
- **bRequired**

A boolean specifying whether this is a required argument for constructing the dynamic type. The default value is `TRUE`.

For example, in the ReqPro adapter, the Project artifact uses creation arguments to create user-defined dynamic artifact types. The following argument determines what requirement type to create. It is the name of the requirement type.

```
Registrar.AddCreationArgument (iProjectTypeID,  
                                bstrRelationshipName,  
                                1,  
                                "Name",  
                                VT_BSTR,  
                                frwName,  
                                -1,  
                                "New Requirement");
```

## AddCreationPropertyArgument

This method is for defining arguments that are needed for creating artifact types by a dynamic relationship type. Maps a creation argument to a registered artifact property. You can use this method with a property name or a property ID.

### Using Property ID

An integer value Property ID is returned when you register a property.

```
bool AddCreationPropertyArgument (int iArtifactTypeID,  
                                  const wchar_t * pRelationshipTypeName,  
                                  int iParamID,  
                                  int PropertyID,  
                                  _variant_t vtDefaultValue = _variant_t(),  
                                  bool bRequired = true);
```

The arguments are:

- Artifact type ID
- Relationship type name
- ParamID

The parameter id. This numeric index corresponds to the argument number (in a list of registered locator arguments).

- Property ID
- Variant reference



- `bRequired`

A boolean specifying whether this is a required argument for constructing the dynamic type. The default value is `TRUE`.

### Using Property Name

Maps a creation argument to an artifact Property name. This method is for defining arguments that are needed for creating artifact types from a dynamic relationship type. For example, in the ReqPro adapter, the Project artifact uses creation arguments to create user-defined dynamic artifact types.

```
bool AddCreationPropertyArgument (int iArtifactTypeID,  
                                const wchar_t * pRelationshipTypeName,  
                                int ParamID,  
                                const wchar_t * PropertyName,  
                                _variant_t vtDefaultValue = _variant_t(),  
                                bool bRequired = true);
```

The arguments are:

- Artifact type ID
- Relationship type name
- ParamID

The parameter id. This numeric index corresponds to the argument number (in a list of registered locator arguments).

- Property name
- Variant reference
- A boolean specifying that this is a required argument for constructing the dynamic type. The default is `true`

## AddDynamicProperty

Registers a property for a dynamic artifact type.

```
int AddDynamicProperty(int iArtifactTypeID,
                      const wchar_t * PropertyName,
                      int DataType,
                      SemanticDataType iSemanticDataType = frwNone,
                      long MaxSize = -1);
```

The arguments for the AddDynamicProperty method are:

- Artifact type ID
- Property name
- Data type
- Semantic data type

The semantic data types are: rsDataObject, rsDescription, rsDirectory, rsFileMoniker, rsFileOrDirectory, rsFilePath, rsName, rsNone, rsPassword, rsURL, rsUserName. The default is frwNone.

- Maximum size

The maximum size of this property. The default is -1 (no maximum).

For example:

```
Registrar.AddDynamicProperty (iArtifactTypeID,
                              (char*)sAttributeName.c_str(), VT_BSTR);
```

## AddDynamicProperty\_ReadOnly

Registers a readonly property for a dynamic artifact.

```
int AddDynamicProperty_ReadOnly(int iArtifactTypeID,
                                const wchar_t * PropertyName,
                                int DataType,
                                SemanticDataType iSemanticDataType = frwNone,
                                long MaxSize = -1);
```

The arguments for the AddDynamicProperty\_ReadOnly method are:

- Artifact type ID

- Property name.
- Data type
- Semantic data type

The semantic data types are: `rsDataObject`, `rsDescription`, `rsDirectory`, `rsFileMoniker`, `rsFileOrDirectory`, `rsFilePath`, `rsName`, `rsNone`, `rsPassword`, `rsURL`, `rsUserName`. The default is `frwNone`.

- Maximum size

The maximum size of this property. The default is -1 (no maximum).

For example:

```
Registrar.AddDynamicProperty_ReadOnly (iArtifactTypeID,
(char*)sAttributeName.c_str(), VT_BSTR);
```

## AddDynamicRelationshipType

Registers a relationship of a dynamic artifact type.

```
int AddDynamicRelationshipType(int iArtifactTypeID,
RelationshipCategory eCategory,
const wchar_t * pRelationshipTypeName,
const wchar_t * pChildTypeName,
RelationshipCardinality eCardinality = ZeroToMany,
bool bCreateDeleteSupported = false,
bool bMinimizeSpace = false);
```

The arguments for the `AddDynamicRelationshipType` method are:

- Artifact type ID
- Relationship category
- Child artifact type name
- Cardinality  
ZeroToMany or ZeroToOne
- `bCreateDeleteSupported`  
The default is false
- `bMinimizeSpace`  
The default is false

For example, the following code adds a dynamic relationship type from a ReqPro Project to its user-defined Requirement dynamic artifacts. The relationship name is the requirement Tag Prefix followed by the word Requirement (for example, a user-defined use case requirement is UCRequirement). In this example, UCRequirement is a dynamic relationship type name and a dynamic artifact type name.

```
Registrar.AddDynamicRelationshipType(  
    iProjectTypeID,           // base artifact type  
    Child,                   // relationship category  
    bstrRelationshipName, // name of relationship type  
    bstrArtifactTypeName, // name of target artifact type  
    ZeroToMany,             // Cardinality  
    true);                  // Create/Delete supported
```

## InternalObjectRegistrar

---

This class contains the methods for creating a registrar object and defining the properties, relationships, and locators for an artifact type. Each artifact in an adapter is defined in an internal object-derived class. If you can create an artifact type, then you also define creation arguments for that type.

The types of methods in the InternalObjectRegistrar class includes:

- Property type registration
- Relationship type registration
- Locator registration
- Graphics format type registration

### FRWInternalObjectRegistrar

Constructs a registrar object to implement properties, relationships, locators, and locator arguments for each artifact. For example:

```
FRWInternalObjectRegistrar (const FRWRegistrationContext &Context)  
:m_Context (Context)  
{  
}
```

For example:

```
FRWInternalObjectRegistrar<CProject> Registrar (Context);
```

## Property Type Registration Methods

Table 11 lists the property registration methods.

**Table 11 InternalObjectRegistrar Property Methods**

Method	Description
AddOverrideProperty	Adds a property that is an override from a base or parent object property.
AddOverrideProperty_ReadOnly	Adds a readonly property that is an override from a base or parent object property.
AddProperty	Registers a property for an artifact type.
AddProperty_ReadOnly	Registers a readonly property for an artifact type.
FindPropertyTypeID	Returns the property ID for an artifact property.
RegisterRunningObjectTableKey	Registers key types that correspond to locator arguments in the running object table.

### AddOverrideProperty

Adds a property that is an override from a base or parent object property. Override an object if a subclass needs to register a different function for retrieving the property declared in a superclass. You also need to implement a new Get method for this override.

```
int AddOverrideProperty(wchar_t * PropertyName,  
                       PropertyGetMethodPtr pGetMethod,  
                       PropertySetMethodPtr pSetMethod);
```

For example, a Rose Item has a Name property. Model is a subclass of Item. If Model also had a Name property that is preferred over the Item Name property, then the Model's name property overrides Item's Name property. In defining this property for the Model artifact type, you could use AddOverrideProperty to register this property as an overridden property. For example:

```
Registrar.AddOverrideProperty("Name", GetName);
```

The GetName method implements this property. For example:

```
void cModel::GetName(_variant_t &Value) // override virtual  
{  
    Value = mInterfacePtr->GetName();}
```

## AddOverrideProperty\_ReadOnly

Adds a readonly property that is an override from a base or parent object property. Override an object if a subclass needs to register a different function for retrieving the property declared in a superclass. You also need to implement a new Get method for this override.

```
int AddOverrideProperty_ReadOnly(wchar_t * PropertyName,
                                PropertyGetMethodPtr pGetMethod);
```

For example, in the ClearCase adapter, the Activity artifact type has a Name property that is a readonly override from its virtual base type:

```
Registrar.AddOverrideProperty_ReadOnly ("Name", GetName);
```

The GetName method implements this property. This method takes a variant reference argument.

```
void CActivity::GetName(_variant_t &Value) // override virtual
{
    Value = mInterfacePtr->GetName();
}
```

## AddProperty

Registers a property for an artifact type. An integer value Property ID is returned when you register a property. You can use this method of finding a PropertyID for optimizing registration.

```
int AddProperty(wchar_t * PropertyName,
                PropertyGetMethodPtr pGetMethod,
                PropertySetMethodPtr pSetMethod,
                int DataType,
                SemanticDataType iSemanticDataType = frwNone,
                long MaxSize = -1);
```

The arguments for the AddProperty method are:

- Property Name
- Get method

The name of the implementation method that retrieves this property from the internal object. The argument for the Get method is a variant value or a null pointer.

- Set method

The name of the implementation method that can retrieve this property from the internal object and modify it. The argument for the Set method is a variant value or a null pointer.

- Data type

The data type of this property

- Semantic data type

A semantic data type is a description of a data type. This provides extra information on a property's or artifact argument's data type. The semantic data types are: rsDataObject, rsDescription, rsDirectory, rsFileMoniker, rsFileOrDirectory, rsFilePath, rsName, rsNone, rsPassword, rsURL, rsUserName. The default is frwNone.

- Maximum size

The maximum size of this property. The default value is -1 (no maximum).

For example, in the ReqPro Requirement artifact, add the Name property:

```
Registrar.AddProperty("Name", GetName, SetName, VT_BSTR, frwName);
```

To implement this, use the get and set methods. These methods take a variant reference argument. The GetName and SetName methods implement the Name property:

```
void CRequirement::GetName(_variant_t &Value)
{
    Value = mInterfacePtr->GetName();
}

void CRequirement::SetName(_variant_t Value)
{
    bstr_t bstrValue = (bstr_t)Value;
    mInterfacePtr->PutName(bstrValue);
    mInterfacePtr->Save ();
}
```

## AddProperty\_ReadOnly

Registers a readonly property for an artifact type. An integer value Property ID is returned when you register a property. You can use this method of finding a PropertyID for optimizing registration.

```
int AddProperty_ReadOnly(wchar_t * PropertyName,
                        PropertyGetMethodPtr pGetMethod,
                        int DataType,
                        SemanticDataType iSemanticDataType = frwNone,
                        long MaxSize = -1);
```

The arguments for the AddProperty method are:

- Property Name
- Get method

The name of the implementation method that retrieves this property from the internal object. The argument for the Get method is a variant value or a null pointer.

- Data type

The data type of this property

- Semantic data type

The semantic data type, if any, of this property. The semantic data types are: rsDataObject, rsDescription, rsDirectory, rsFileMoniker, rsFileOrDirectory, rsFilePath, rsName, rsNone, rsPassword, rsURL, rsUserName. The default is frwNone.

- Maximum size

The maximum size of this property. The default value is -1 (no maximum).

For example, a ReqPro example requirement has a HasParent property:

```
Registrar.AddProperty_ReadOnly ("HasParent",GetHasParent,VT_BOOL);
```

The GetHasParent method implements this property. This method takes a variant reference argument.

```
void CRequirement::GetHasParent(_variant_t &Value)
{
    long lCount;
    Value = mInterfacePtr->GetHasParent(&lCount) ? true : false;}

```



## FindPropertyTypeID

Returns the property ID for an artifact property, given an artifact type ID and a property name.

```
int FindPropertyTypeID (wchar_t *pPropertyName);
```

For example, from the ClearQuest adapter:

```
bool CQDatabase::PropertyExists(FRWInternalObjectTypeRegistrar  
Registrar,  
  
    int ArtifactTypeID,  
    _bstr_t PropertyName)  
{  
    return Registrar.GetCoreTypeContainer()->  
        FindPropertyTypeID (ArtifactTypeID, PropertyName)  
        != 0;  
}
```

## RegisterRunningObjectTableKey

Registers the method for returning instances of a specific artifact type (internal object type).

```
void RegisterRunningObjectTableKey (GetKeyMethodPtr pGetMethod);
```

The running object table maps artifact pointers to artifact keys. You register all artifact keys with the AddKeyType method if you want to register it the running object table.

Before calling this method, you must first register all the key types. The names of these key types are the same as the registered locator arguments. Each locator should have its locator types registered as key types in order to be registered in the running object table.

If you register a locator argument as a key type, that argument should be used in all locators for that artifact type. These arguments registered as key types should uniquely identify an artifact type in the RunningObjectTable.

For example, in the ReqPro adapter Project artifact class there is a LocateWithPath locator:

```
Registrar.AddAbsoluteLocator (LOCATE_WITH_PATH, LocateWithPath,  
    LOCATE_DEFAULT_IMMUTABLE_ID|LOCATE_DEFAULT_DISPLAY_NAME);
```

The arguments for this locator are:

```
Registrar.AddLocatorArgument (LOCATE_WITH_PATH, 1, "Path");
```

```

Registrar.AddLocatorArgument (LOCATE_WITH_PATH, 2, "UserName",
VT_BSTR, frwUserName, -1, "", false);

Registrar.AddLocatorArgument (LOCATE_WITH_PATH, 3, "Password",
VT_BSTR, frwPassword, -1, "", false);

Registrar.AddLocatorArgument (LOCATE_WITH_PATH, 4, "Flags",
VT_I2, frwNone, -1, _variant_t ((short)4, VT_I2), false);

```

The flags parameter contains information allowing the project to be opened readonly or exclusive. The legal values are:

- 0 - Normal
- 1 - ReadOnly
- 2 - Exclusive
- 4 - FallbackToReadOnly

The default value is 4.

Each of these locator arguments is also registered as a key type for the running object table. The parameter IDs and argument names correspond to the arguments in the AddLocatorArgument registration above. The first key type (Path) is the only argument that maps to a property type and thus is the only key type with the third argument not set to NULL.

```

Registrar.AddKeyType(1, "Path", "Path");
Registrar.AddKeyType(2, "UserName", NULL);
Registrar.AddKeyType(3, "Password", NULL);
Registrar.AddKeyType(4, "Flags", NULL);

```

You first register key types for an artifact type using the AddKeyType method. You can then call the RegisterRunningObjectTableKey method to enable running object table support for this type.

```
Registrar.RegisterRunningObjectTableKey (GetRunningObjectTableKey);
```

The GetRunningObjectTableKey method implements the running object table key. It returns a unique key that identifies each running instance of this artifact type.

```

void CProject::GetRunningObjectTableKey(FRWInternalObjectReference
&Context, FRWLocatorSinkPtr &pLocatorSink)
{
    CProject* pCProject = (CProject*)Context.GetInternalObject();
    _ASSERT(pCProject != NULL);
}

```

## Relationship Type Registration Methods

Table 12 lists the relationship type registration methods.

**Table 12 InternalObjectRegistrar Relationship Methods**

Method	Description
AddFilteredRelationshipType	Registers a filtered relationship for an artifact type.
AddOverrideFilteredRelationshipType	Adds a filtered relationship that is an override from a base or parent object relationship.
AddOverrideRelationshipType	Adds a relationship that is an override from a base or parent object relationship.
AddRelationshipType	Registers a relationship for an artifact type.

### AddFilteredRelationshipType

Registers a filtered relationship for an artifact type.

```
int AddFilteredRelationshipType (RelationshipCategory eCategory,  
    wchar_t * pRelationshipTypeName,  
    wchar_t * pChildTypeName,  
    GetRelatedFilteredArtifactMethodPtr pGetFilteredMethod,  
    RelationshipCardinality eCardinality = ZeroToMany,  
    CreateRelatedArtifactMethodPtr pCreateMethod = NULL,  
    DeleteRelatedArtifactMethodPtr pDeleteMethod = NULL,  
    bool bMinimizeSpace = false);
```

For example, a ReqPro Project creates a filtered relationship from the Project to its Requirements:

```
Registrar.AddFilteredRelationshipType(Child,  
    "Requirements",  
    "Requirement",  
    GetRequirements,  
    ZeroToMany,  
    CreateRequirement,  
    DeleteRequirement);
```

The GetRequirements method implement this relationship:

```

void CProject::GetRequirements (FRWArtifactAdapterCollection *
pObjects, RDSIAdapterProtocol::IArtifactFilterSink * pFilter,
FWFilteringStatus* pFilteringStatus)
{
    // Get the specific Requirement prefix (if any) from the filter.
    string lReqTypePrefix = GetFilterReqTypePrefix(pFilter);

    GetRequirements(lReqTypePrefix, pObjects, pFilter,
pFilteringStatus);
}

Create Method:

void CProject::CreateRequirement (FRWInternalObjectReference &
Context,

                                const FRWArguments &Params)
// Params is the collection of artifact locator argument types.
{
    _bstr_t bstrArtifactTypeName = Params.GetArg(1);
    _bstr_t bstrName = Params.GetArg(2);
    _bstr_t bstrText = Params.GetArg(3);

    CreateRequirement(Context, bstrArtifactTypeName, bstrName,
bstrText);
}

void CProject::CreateRequirement(FRWInternalObjectReference & Context,
    _bstr_t bstrArtifactTypeName,
    _bstr_t bstrName,
    _bstr_t bstrText)

...

// Delete Method:

void CProject::DeleteRequirement(FRWInternalObject *
pInternalObjectToDelete, short iTypeID)

...

```

## AddOverrideFilteredRelationshipType

Adds a filtered relationship that is an override from a base or parent object relationship. Override an object if a subclass needs to register a different function for retrieving the related artifacts of the relationship type declared in a superclass. You also need to implement a new Get method for this override.

```
int AddOverrideFilteredRelationshipType(  
    wchar_t * pRelationshipTypeName,  
    GetRelatedFilteredArtifactMethodPtr pGetFilteredMethod,  
    CreateRelatedArtifactMethodPtr pCreateMethod = NULL,  
    DeleteRelatedArtifactMethodPtr pDeleteMethod = NULL);
```

The arguments are:

- The relationship type name
- The get method for getting the related artifacts for this relationship type
- Create method (NULL if there is no Create method)
- Delete method (NULL if there is no Delete method)

For example, override the Reqro Requirements filtered relationship type:

```
Registrar.AddOverrideFilteredRelationshipType("Requirements",  
                                             GetRequirements, NULL, NULL)
```

## AddOverrideRelationshipType

Adds a relationship that is an override from a base or parent object relationship. Override an object if a subclass needs to register a different function for retrieving the related artifacts of the relationship type declared in a superclass. You also need to implement a new Get method for this override.

```
int AddOverrideRelationshipType (  
    wchar_t * pRelationshipTypeName,  
    GetRelatedArtifactMethodPtr pGetMethod,  
    CreateRelatedArtifactMethodPtr pCreateMethod = NULL,  
    DeleteRelatedArtifactMethodPtr pDeleteMethod = NULL);
```

The arguments are:

- The relationship type name
- The get method for getting the related artifacts for this relationship type

- Create method (NULL if there is no Create method)
- Delete method (NULL if there is no Delete method)

For example, override the Rose AllClasses relationship type:

```
Registrar.AddOverrideRelationshipType("AllClasses",
                                      GetAllClasses, NULL, NULL)
```

## AddRelationshipType

Registers a relationship for an artifact type.

```
int AddRelationshipType (RelationshipCategory eCategory,
                        wchar_t * pRelationshipTypeName,
                        wchar_t * pChildTypeName,
                        GetRelatedArtifactMethodPtr pGetMethod,
                        RelationshipCardinality eCardinality = ZeroToMany,
                        CreateRelatedArtifactMethodPtr pCreateMethod = NULL,
                        DeleteRelatedArtifactMethodPtr pDeleteMethod = NULL,
                        bool bMinimizeSpace = false);
```

The arguments for the AddRelationshipType method are:

- Relationship type (RelationshipCategory eCategory)  
For example, Peer, Child, Descendant, or Parent.
- Relationship name
- The artifact type name  
The name of the artifact type that this relationship points to.
- Get method (GetRelatedArtifactMethodPtr pGetMethod)  
This method includes product specific code (for example, RequisitePro extensibility (RPX)) that enumerates the attribute values and returns them in the collection. The argument for the GetMethod is a pointer to the FRWArtifactAdapterCollection.
- Cardinality  
For example, ZeroToMany or ZeroToOne.
- Create method, if any
- Delete method, if any

For example, the ReqPro adapter defines a Requirement artifact ParentProject relationship type to its parent project:

```
Registrar.AddRelationshipType (Peer, "ParentProject", "Project",
    GetParentProject, ZeroToOne);
```

The GetParentProject method implement this relationship.

```
void CRequirement::GetParentProject(FRWArtifactAdapterCollection *
    pObjects)
{
    pObjects->Add ("Project", mpParent);
}
```

## Locator Registration Methods

Table 13 lists the locator registration methods.

**Table 13 InternalObjectRegistrar Locator Methods**

Method	Description
AddAbsoluteLocator	Registers an artifact locator for an artifact type.
AddCreationArgument	Registers an argument that is needed for calling a create artifact method. Maps a creation argument to an argument name (for example, a username or password).
AddCreationPropertyArgument	Registers an argument that is needed for calling a create artifact method. Maps a creation argument to an artifact Property Name or Property ID.
AddKeyType	Registers a key type that matches a locator argument for an artifact type.
AddLocatorArgument	Registers a locator argument for an artifact locator.
AddRelativeLocator	Registers a relative locator for an artifact type.

These are common combinations that may be used when registering locators:

```
#define LOCATE_DISPLAY_NAME_AND_ID LOCATE_DISPLAY_NAME |
LOCATE_IMMUTABLE_ID

#define LOCATE_DEFAULT_DISPLAY_NAME_AND_ID
LOCATE_DEFAULT_IMMUTABLE_ID | LOCATE_DEFAULT_DISPLAY_NAME
```

## AddAbsoluteLocator

Registers an artifact locator for an artifact type.

```
bool AddAbsoluteLocator (int iLocatorID,  
                        AbsoluteLocatorMethodPtr pLocateMethod,  
                        unsigned long lFlags);
```

The arguments for the AddAbsoluteLocator method are:

- Locator ID

All locators must use a distinct ID. Once these locators are in use, the IDs can not be changed. For example, the IDs for the locators for the Item class in the ReqPro adapter are:

```
#define LOCATE_WITH_KEY 1  
#define LOCATE_WITH_TAG 2
```

- Locate method

The method that implements this locator.

- Locator flag

Specifies the format of this locator. The locator format types are:

- LOCATE\_IMMUTABLE\_ID
- LOCATE\_DISPLAY\_NAME
- LOCATE\_DEFAULT\_IMMUTABLE\_ID | LOCATE\_IMMUTABLE\_ID
- LOCATE\_DEFAULT\_DISPLAY\_NAME | LOCATE\_DISPLAY\_NAME

For example, the ReqPro Project defines an absolute locator:

```
Registrar.AddAbsoluteLocator (LOCATE_WITH_PATH, LocateWithPath,  
LOCATE_DEFAULT_IMMUTABLE_ID |  
  
LOCATE_DEFAULT_DISPLAY_NAME);  
Registrar.AddLocatorArgument (LOCATE_WITH_PATH, 1, "Path");
```

The LocateWithPath method implements this locator definition as follows:

```
bool CProject::LocateWithPath (const FRWArguments &Params,  
                              FRWInternalObjectReference &Context)  
{  
    _bstr_t bstrPath = Params.GetArg(1);
```



```

    _bstr_t bstrUserName = Params.GetArg(2);
    _bstr_t bstrPassword = Params.GetArg(3);
    int     iFlags       = (short)Params.GetArg(4);

    ReqPro40::_ApplicationPtr pReqProApp;
    ReqPro40::_ProjectPtr pProject;
    CProject* pCProject = NULL;
    Context.Attach("Project", pCProject);
return true;
}

```

The arguments for this Locate method include:

- The parameters to the locator. Params is the collection of artifact locator argument types for a given artifact type.
- The context used to attach the located object

## AddCreationArgument

Defines arguments that are needed for creating artifact types by a relationship type. This registers the arguments that are needed for calling a CreateArtifact method. It maps a creation argument to a registered argument name (for example, a username or password). If there is a Create method (for example in the ReqPro adapter CreateRequirement method, the adapter defines the creation arguments that are used to create the artifact.

The adapter registers the Create method and the types of its arguments.

```

bool AddCreationArgument (wchar_t * pRelationshipTypeName,
                        int iParamID,
                        const wchar_t * ArgumentName,
                        int DataType,
                        SemanticDataType iSemanticDataType = frwNone,
                        int MaxSize = -1,
                        _variant_t vtDefaultValue = _variant_t(),
                        bool bRequired = true);

```

For example, in ReqPro adapter Project artifact type class, the following argument determines what requirement type to create. TypeName is the name of the requirement type:

```
Registrar.AddCreationArgument("Requirements", 1, "TypeName",
                               VT_BSTR, frwNone);
```

Two additional creation arguments are defined:

```
Registrar.AddCreationArgument("Requirements", 2, "Name",
                               VT_BSTR, frwName, -1, "New Requirement");
Registrar.AddCreationArgument("Requirements", 3, "Text",
                               VT_BSTR, frwDescription, -1, "This is the text.");
```

The CreateRequirement method takes these creation arguments to create a requirement:

```
void CProject::CreateRequirement (FRWInternalObjectReference &
Context,
                                const FRWArguments &Params)
{
    _bstr_t bstrArtifactTypeName = Params.GetArg(1);
    _bstr_t bstrName = Params.GetArg(2);
    _bstr_t bstrText = Params.GetArg(3);
    // Params is the collection of artifact locator argument types.
    CreateRequirement(Context, bstrArtifactTypeName, bstrName,
bstrText);
}
```

## AddCreationPropertyArgument

Defines arguments that are needed for creating artifact types by a relationship type. This registers arguments that are needed for calling a CreateArtifact method. It maps a creation argument to a registered artifact property. If there is a Create method (for example in the ReqPro adapter CreateRequirement method, the adapter defines the creation arguments that are used to create the artifact.

The adapter registers the create method and the types of its arguments. There are two forms for this method

- Using a property ID
- Using a property name

### Using Property ID

You can get a Property ID by saving the return value of the AddProperty method.

```

bool AddCreationPropertyArgument (wchar_t * pRelationshipTypeName,
                                  int iParamID,
                                  int PropertyID,
                                  _variant_t vtDefaultValue = _variant_t(),
                                  bool bRequired = true);

```

## Using Property Name

```

bool AddCreationPropertyArgument (wchar_t * pRelationshipTypeName,
                                  int iParamID,
                                  wchar_t * PropertyName,
                                  int DataType,
                                  SemanticDataType iSemanticDataType = frwNone,
                                  int MaxSize = -1,
                                  _variant_t vtDefaultValue = _variant_t(),
                                  bool bRequired = true);

```

For example:

```
Registrar.AddCreationPropertyArgument("Packages", 1, "Name", VT_BSTR,
frwNone);
```

```
Registrar.AddLocatorArgument (PACKAGE_ID_LOCATE_WITH_NAME, 1, "Name");
```

When a client application calls a create artifact function that takes creation arguments, it gets these arguments from a relationship type. The relationship returns the creation arguments for a user to enter values and create an artifact. For example:

```

Dim theNewArtifact As Artifact
Dim theRelationship As RelationshipType
Dim theArgCollection As ArtifactArgumentCollection
Dim theArg As ArtifactArgument
Dim ArgumentID As Integer

Set theArgCollection = theRelationship.CreationArguments
For ArgumentID = 0 To theArgCollection.Count - 1
    Set theArg = theArgCollection.Item(ArgumentID)
    theArg.Value = InputBox("Please enter a value for argument " &
theArg.ArgumentName & ".", CStr(theArg.DefaultValue))

```

```
Next ArgumentID
```

```
Set theNewArtifact =  
m_ContextArtifact.CreateArtifact(theRelationship)  
  
Set CreateNewArtifact = theNewArtifact
```

## AddKeyType

Registers a key type that matches a locator argument for an artifact type. Each locator should have its locator types registered as a key type in order to be registered in the running object table.

```
bool AddKeyType(int iParamID,  
                wchar_t * ArgumentName,  
                wchar_t * PropertyName);
```

The arguments are:

- Parameter ID

An integer that uniquely identifies a key. The parameter ID matches the ID in the AddLocatorArgument method.

- Argument name

The name of the argument.

- Property name

NULL if the argument does not correspond to an artifact property.

Before you register the method that implements the running object table key, you register all the key types. The names of these key types are the same as the registered locator arguments.

The following example registers UserName and Password as key types. UserName and Password are also registered as locator arguments for this artifact type.

```
Registrar.AddLocatorArgument (LOCATE_WITH_PATH, 2, "UserName",  
VT_BSTR, frwUserName, -1, "", false);  
  
Registrar.AddLocatorArgument (LOCATE_WITH_PATH, 3, "Password",  
VT_BSTR, frwPassword, -1, "", false);  
  
Registrar.AddKeyType(2, "UserName", NULL);  
  
Registrar.AddKeyType(3, "Password", NULL);
```

See the *RegisterRunningObjectTableKey* method section of this chapter for more information.

## AddLocatorArgument

Registers a locator argument for constructing an artifact locator. There are three forms of the AddLocatorArgument method:

- Using a property ID
- Using a property name
- Defining an argument that does not map to an artifact property

### Using Property ID

Registers a locator argument for constructing an artifact locator using an artifact property ID. Each property type has a Property ID. You can get a Property ID as a return value when you register a property (for example, PropertyID = Registrar.AddProperty\_ReadOnly).

```
bool AddLocatorArgument (int iLocatorID,  
                        int iParamID,  
                        int PropertyID,  
                        _variant_t vtDefaultValue = _variant_t(),  
                        bool bRequired = true);
```

### Using Property Name

Registers a locator argument for constructing an artifact locator using an artifact property name.

```
bool AddLocatorArgument (int iLocatorID,  
                        int iParamID,  
                        const wchar_t * PropertyName,  
                        _variant_t vtDefaultValue = _variant_t(),  
                        bool bRequired = true);
```

For example, in ReqPro register a Project artifact type locator argument, that maps to the Path property:

```
Registrar.AddLocatorArgument (LOCATE_WITH_PATH, 1, "Path");
```

In this example:

- LOCATE\_WITH\_PATH is the locator id

- 1 is the parameter id for this (Path) locator argument, since it's the first locator argument for this artifact type.
- Path is the name of the Property this locator argument maps to.

### Defining a New Argument

Registers a locator argument for constructing an artifact locator using an argument that does not map to a property.

Use this method to add arguments that do not correspond to properties. These arguments will not be initialized when locators are created for existing artifacts.

```
bool AddLocatorArgument(int iLocatorID,
                       int iParamID,
                       const wchar_t * ArgumentName,
                       int DataType,
                       SemanticDataType iSemanticDataType = frwNone,
                       long MaxSize = -1,
                       _variant_t vtDefaultValue = _variant_t(),
                       bool bRequired = true);
```

For example, in ReqPro, add UserName as a locator argument for constructing a locator to a Project:

```
Registrar.AddLocatorArgument (LOCATE_WITH_PATH,
                              2,
                              "UserName",
                              VT_BSTR,
                              frwUserName,
                              -1,
                              "",
                              false);
```

In this example:

- LOCATE\_WITH\_PATH is the locator id

- The parameter id for UserName is 2. This integer value represents the ordinal argument number. UserName is the second artifact argument for constructing the locator for this artifact type. (Path is the first argument and thus the parameter ID for Path is 1. Password is the third argument and thus the parameter ID for Password is 3.)
- UserName is the argument name
- Data type is VT\_BSTR
- frwUserName is the semantic data type. A semantic data type is a description of a data type. This provides extra information on a property's or artifact argument's data type. The semantic data types are: rsDataObject, rsDescription, rsDirectory, rsFileMoniker, rsFileOrDirectory, rsFilePath, rsName, rsNone, rsPassword, rsURL, rsUserName. The default is frwNone.
- Maxsize is -1 (this is the default value)
- vtDefaultValue is set to "".
- bRequired is set to false making these arguments not required. The default value is TRUE.

## AddRelativeLocator

Registers a relative locator for an artifact type. There are two forms of the AddRelativeLocator method. One uses a RelativeArtifactTypeID and the other uses RelativeArtifactTypeName.

The arguments for the AddRelativeLocator method are:

- Locator ID
- Relative artifact type name or relative artifact type ID

This is the context artifact type

- Locate method
- Locator flag

Specifies the format of this locator. The locator format types are:

- LOCATE\_IMMUTABLE\_ID
- LOCATE\_DISPLAY\_NAME
- LOCATE\_DEFAULT\_IMMUTABLE\_ID | LOCATE\_IMMUTABLE\_ID
- LOCATE\_DEFAULT\_DISPLAY\_NAME | LOCATE\_DISPLAY\_NAME

## Using RelativeArtifactTypeID

```
bool AddRelativeLocator (int iLocatorID,
                        int RelativeArtifactTypeID,
                        RelativeLocatorMethodPtr pLocateMethod,
                        unsigned long lFlags);
```

The following ReqPro example registers a relative locator for a ReqDocument artifact type using a relative artifact type ID. In this example the ReqDocument locator is relative to Project, using document ID:

```
Registrar.AddRelativeLocator (DOCUMENT_ID_LOCATE_WITH_KEY,
                              "Project",
                              LocateWithKey,
                              LOCATE_DEFAULT_IMMUTABLE_ID);

Registrar.AddLocatorArgument (DOCUMENT_ID_LOCATE_WITH_KEY, 1,
                              "DocumentID");
```

In this example:

- DOCUMENT\_ID\_LOCATE\_WITH\_KEY is the locator id
- Project is the relative artifact
- LocateWithKey is theLocate method for this relative locator
- LOCATE\_DEFAULT\_IMMUTABLE\_ID is the locator type

The LocateWithKey method implements this locator:

```
bool CReqDocument::LocateWithKey(FRWInternalObject
*pRelativeObject,
    const FRWArguments &Params,
    FRWInternalObjectReference &Context)
{
    CProject* pCProject = dynamic_cast<CProject*> (pRelativeObject);
    _ASSERT (pCProject != NULL);

    _bstr_t bDocumentID = Params.GetArg (1);
    string sDocumentID = (const char*)bDocumentID;
    long iDocumentID = atoi(sDocumentID.c_str());
    _variant_t spvtIndex(iDocumentID);
```



```

    _variant_t dummy;

    ReqPro40::_DocumentsPtr pDocuments =
    pCProject->GetInternalProjectPtr()->GetDocuments();

    if (pDocuments == NULL) return false;

    ReqPro40::_DocumentPtr pDocument = pDocuments->GetItem(spvtIndex,
    ReqPro40::eDocLookup_Key);

    if (pDocument == NULL) return false;

    Context.Attach("ReqDocument", NewReqDocument(pDocument,
    pCProject));

    return true;
}

```

The arguments for this Locate method:

- pRelativeObject  
The relative object is Project.
- Params  
The one locator argument is DocumentID. Params is the collection of artifact locator argument types for a given artifact type.
- Context  
The context is the information used to attach the located RequisitePro ReqDocument IUnkown (referenced by pDocument) to the RSE ReqDocument internal object.

### Using RelativeArtifactTypeName

```

bool AddRelativeLocator (int iLocatorID,
                        const wchar_t * pRelativeArtifactTypeName,
                        RelativeLocatorMethodPtr pLocateMethod,
                        unsigned long lFlags);

```

For example, in ReqPro, register a ReqDocument relative locator, relative to Project. Locate with name as the argument.

```

Registrar.AddRelativeLocator (LOCATE_WITH_NAME,
                              "Project",

```

```

        LocateWithName,
        LOCATE_DEFAULT_DISPLAY_NAME);
Registrar.AddLocatorArgument (LOCATE_WITH_NAME, 1, "Name");

```

The `LocateWithName` method implements this locator:

```

bool CReqDocument::LocateWithName(FRWInternalObject
*pRelativeObject,

                                const FRWArguments &Params,
                                FRWInternalObjectReference &Context)
{
    CProject* pCProject = dynamic_cast<CProject*> (pRelativeObject);
    _ASSERT (pCProject != NULL);
    ...
    Context.Attach("ReqDocument", NewReqDocument(pDocument,
pCProject));
    return true;
}

```

## Graphics Registration Methods

The graphics registration method is `AddGraphicsFormatType`.

### AddGraphicsFormatType

Registers graphics format types.

```

int AddGraphicsFormatType(wchar_t * pGraphicsTypeName,
                           wchar_t * pDescription,
                           wchar_t * pGraphicsFormatName,
                           RenderToFileMethodPtr pRenderMethod);

```

The arguments are:

- `pGraphicsTypeName`  
The name used by clients to access the graphic file.
- `pDescription`  
A human readable description of the graphic file format.
- `pGraphicsFormatName`

The name of the graphic format.

- `pRenderMethod`

The method called to retrieve the graphic in this particular format.

In general, a graphic may be rendered in any of several formats, so there are often multiple calls to `AddGraphicsFormatType` with the same `Name` argument but different `Type` arguments. For example, from the `FileSys` adapter, the `File` artifact type registers the following graphics format types:

```
Registrar.AddGraphicsFormatType (L"Graphic",L"Graphic - wmf",
L"wmf",RenderGraphic_wmf);

Registrar.AddGraphicsFormatType (L"Graphic",L"Graphic - jpg",
L"jpg",RenderGraphic_jpg);

Registrar.AddGraphicsFormatType (L"Graphic",L"Graphic - png",
L"png",RenderGraphic_png);

Registrar.AddGraphicsFormatType (L"Graphic",L"Graphic - bmp",
L"bmp",RenderGraphic_bmp);

Registrar.AddGraphicsFormatType (L"Graphic",L"Graphic - gif",
L"gif",RenderGraphic_gif);

Registrar.AddGraphicsFormatType (L"Graphic",L"Graphic - asis",
L"asis",RenderGraphic_asis);
```

The `RenderGraphic` methods implement these format type definitions. For example, the `RenderGraphic` method for the `wmf` format type:

```
_bstr_t cFile::RenderGraphic_wmf(_bstr_t bstrFileName, _bstr_t
bstrParameters)
{
    if (! mpGraphic) {
        mpGraphic = new FRWGraphic();
        mpGraphic->LoadFromFile(mPathName);
    }
    return mpGraphic->RenderWMFToFile(bstrFileName, bstrParameters);
}
```

## Using the Mapping Mechanism

You can use the C++ Framework mapping mechanism to streamline the registration of graphics formats types. For example, register the graphics formats types for the `FileSys` adapter `File` artifact type as follows:

```

BEGIN_GRAPHIC_FORMATS (File)
    format (File, WMF , Render file as WMF)
    format (File, JPG , Render file as JPG)
    format (File, PNG , Render file as PNG)
    format (File, BMP , Render file as BMP)
    format (File, GIF , Render file as GIF)
    format (File, ASIS, Pass file as is)
END_GRAPHIC_FORMATS

```

The arguments for each of the format method calls are:

- Aspect  
The name used by clients to access the graphic file.
- Format type  
The name of the graphic format.
- Description  
A human readable description of the graphic file format.

As in the standard registration for adding graphics format types, each format type has a render method. For example, the render method for wmf format is as follows:

```

_bstr_t CFile::RenderFileInWMF(_bstr_t bstrFileName, _bstr_t
bstrParameters)
{
    if (! mpGraphic) {
        mpGraphic = new FRWGraphic();
        mpGraphic->LoadFromFile(mPathName);
    }
    return mpGraphic->RenderWMFToFile(bstrFileName, bstrParameters);
}

```

For more information on using the Maps mechanism, see the “Using RSE Adapter Interfaces” chapter of this manual.

# Index

## A

- Absolute locator 120
- Adapter
  - architecture 35
  - framework classes 35
  - instance 45
  - interfaces 35
  - operations 98
  - overview 35
  - project 36
  - ReqPro 69
- AdapterInstance 45, 100
  - creating 68
  - modifying code 55
  - modifying cpp file 59
  - object 50
  - registry file 53
- AdapterProtocol.idl 48
- Adapters 16, 18, 22
- AddAbsoluteLocator 119, 120
- AddArtifactType 68, 101, 102
- AddCreationArgument 101, 103, 119, 121
- AddCreationPropertyArgument 101, 119, 122
- AddDynamicProperty 101, 106
- AddDynamicProperty\_ReadOnly 101, 106
- AddDynamicRelationshipType 101, 107
- AddFilteredRelationshipType 115
- AddGraphicsFormatType 130
- AddKeyType 119, 124
- AddLocatorArgument 77, 78, 119, 125
  - Locator
    - arguments 77
- AddOverrideFilteredRelationshipType 115, 117
- AddOverrideProperty 109
- AddOverrideProperty\_ReadOnly 109
- AddOverrideRelationshipType 115, 117
- AddProperty 109, 110
- AddProperty\_ReadOnly 109, 112
- AddRelationshipType 74, 115, 118
- AddRelativeLocator 77, 119, 127
- Application object 96
- Applications 16
- Architecture 14
- Arguments
  - artifact 28
  - locator 28
- Artifact 23
  - arguments 28
  - internal object 68
  - references 29
  - relative id 31
- Artifact locator
  - collections 80
- Artifact type 23
  - add 101, 102
  - creation arguments 83, 121
  - creation property arguments 122
  - defining locator arguments 78
  - dynamic 23, 71
  - implementing a class 72
  - locator registration 119
  - property 110
  - registering a locator 76
  - registering a property 73
  - registering a relationship 74
  - registering a type 102
  - relative locator 127
  - static 23
- ArtifactAdapter 100
- ArtifactCollectionAdapter 100
- ATL Project 38
- Attach 80

## C

- C++ framework 35
  - creating a dependency 40
- C++ language settings 40, 44
- Client applications 15, 16
- Code generation
  - settings 40
- Collections
  - artifact locators 80

- COM server 96
- Creating
  - adapter instance 68
  - ATL project 38
- Creation argument 83, 121
  - add 101, 103
  - parameter id 84
- Creation property argument
  - add 101

## D

- DeclareArtifactTypes 68
- Defining
  - a new locator argument 126
  - adapter instance 45
  - locator 77
  - locator arguments 78
  - locators 80
  - relationship types 115
- Dependencies
  - C++ Framework 40
- Developing an adapter 36
- Dynamic
  - add property 101
  - add relationship type 101
  - artifact types 23, 71
  - property 106
  - relationship type 107

## F

- Filtered relationship
  - add 115
- FindPropertyTypeID 109, 113
- Framework classes 35
- FRWInternalObject subclasses 72
- FRWInternalObjectRegistrar 108
- FRWInternalObjectTypeRegistrar 102

## G

- Graphics

- registration methods 130
- using maps registration 93

## H

- Handler 93
- Handler declaration macros
  - maps 93

## I

- ID
  - property 26
  - property id 113
- IDL file 48
- Implementing
  - artifact type 72
- Importing
  - AdapterProtocol.idl 48
- Integrated-product
  - server object 96
- Internal object 68, 95, 96
  - factory 71
  - registration 71
- InternalObjectRegistrar 101, 108
- InternalObjectTypeRegistrar 101

## K

- Key type 124

## L

- Locating
  - internal object 68
- Locator
  - arguments 28, 125
  - defining 77
  - defining a new argument 126
  - defining arguments 78
  - registering 76
  - registering a 120
  - registering a relative locator 127

- registration methods 119
- Locators 27
  - relative id 31

## M

- Maps
  - handler declaration macros 93
  - registering 85
  - registering artifact types 85
  - registering graphics format types 93
  - registering properties 88
  - registering relationships 89
- Maps mechanism 84
- Modifying
  - code generation settings 40
  - code in new AdapterInstance.h file 55
  - IDL file 48
  - new AdapterInstance.cpp file 59
  - registry file 53
  - stdafx.h file 61

## O

- Object
  - AdapterInstance 50
- Object table 109
- Objects 21
- Override
  - property 109
  - relationship type 117
- Overview 35

## P

- Parameter ID 84
- Preprocessor settings 40, 42
- Product server 96
- Project
  - setting up 36
  - settings 40
- Property 25
  - add 109

- add override 109
- dynamic 101
- id 26, 109, 113
- registering 73
- registering a 110
- registration methods 109
- type 25

## R

- RDSICore
  - type library 34
- Readonly
  - property 112
- References
  - relative id 31
  - type library 34
- Register 72
  - method 71
- Registering
  - a property 110
  - an artifact type 102
  - creation arguments 83
  - dynamic types 71
  - graphics 130
  - internal object 108
  - locator 76
  - locator arguments 78, 125
  - locators 119
  - maps 85
  - property 73, 109
  - relationship 74
  - relationship types 115
  - relative locator 127
  - using maps 84
- RegisterRunningObjectTableKey 109, 113
- Registrar
  - internal object 108
  - internal object type 102
- Registry file
  - modifying 53
- Relationship 26
  - add filtered 115
  - add override filtered 117

- registering 74
- Relationship type
  - add 101, 118
  - dynamic types 107
  - override 117
  - registration methods 115
- Relative id 31
- Relative locator
  - registering a 127
- RelativeArtifactTypeID 128
- ReqPro adapter 69
  - internal objects 69
- RPX 75
- RSE
  - adapters 16
  - objects 21
- Running object table 113
  - AddKeyType 124

## **S**

- Server object 96
- Session 22

- Setting up
  - adapter project 36
- Settings
  - code generation 40
- SoDA 33
- Static
  - artifact types 23
- Stdafx.h
  - modifying 61
- subclass 24
- superclass 24

## **T**

- Type library
  - referencing 34

## **U**

- Using
  - RSE 15