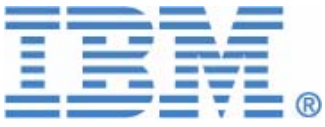
Rational. Rhapsody

IBM

COM Development Guide

Rational Rhapsody
COM Development Guide



Before using the information in this manual, be sure to read the “Notices” section of the Help or the PDF available from **Help > List of Books**.

This edition applies to IBM® Rational® Rhapsody® 7.5 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

COM Development Introduction

Distributed applications are client/server applications in which clients and servers typically run on different processors, which can be located on either the same machine or on different machines. The client and server applications can be written in the same language (for example, C++) or a mixture of several different languages (such as C++, C, and Java) and can run on the same operating system or different operating systems.

There are several mechanisms in use for allowing mixed, distributed applications to find and interact with each other over a network. Rational Rhapsody supports two such mechanisms:

- COM—The Component Object Model.
- CORBA®—The Common Object Request Broker Architecture, endorsed by the OMG®.

Using Rational Rhapsody to Develop COM Applications

If you are a COM domain user, you are developing COM distributed systems using COM interfaces as part of your process. This book describes how you can use Rational Rhapsody to achieve your goal of making COM an integral part of your software model. Using Rational Rhapsody, you can define, use, and manipulate COM interfaces and libraries. You can also use Rational Rhapsody to link the COM domain constructs to a high-level language domain (the C++ domain).

You can use Rational Rhapsody to do any of the following with COM:

- Design a server.
- Design a client.
- Import a TLB file.
- Design an interface (forward engineering).

The following sections describe these tasks in detail.

Design of Clients and Servers

If an interface is already developed (either with Rational Rhapsody or imported using the TLB import utility), you can use it as-is and concentrate on implementing the client or server. Rational Rhapsody enables ATL-based server implementation development.

The artifacts of client/server development with Rational Rhapsody are the following:

- For servers, COM components (objects) and servers (DLLs and EXEs); for clients, COM C++ clients
- Ability to debug COM servers using the Rational Rhapsody built-in animation and debugging facilities
- Visual representation of the relations between the various COM interfaces in the system and their implementing classes

See **COM Servers** and **COM Clients** for more information on designing COM clients and servers with Rational Rhapsody.

Import of Type Libraries

You can import existing type libraries into Rational Rhapsody so Rational Rhapsody elements can reference them. The resulting structure is a Rational Rhapsody package that reflects the type library, its interfaces, and coclasses. A package that is the result of a type library import is read-write, but code generation is disabled for it.

If you implement COM servers, you can use an imported type library package as the basis for deriving new interfaces. If you implement COM clients, you can use the type library to interact with external TLB interfaces. See **TLB Importer** for information on how to use the Rational Rhapsody TLB importer.

Interface Design

Rational Rhapsody enables you to develop interfaces by generating the following artifacts:

- A `ProxyStub.dll` file
- COM IDL structures and files consisting of interface coclasses (both incoming and outgoing)
- TLB files

The benefit to using Rational Rhapsody to generate these files is that it gives you a visual description of the different COM interfaces that comprise the system, the relationships between them, and their packaging. Therefore, the results of designing an interface with Rational Rhapsody are a visual model of the interface, as well as binary components. Implementers of COM servers and clients can use these artifacts in designing their respective systems.

See **COM Interfaces** for more information on designing COM interfaces.

Generation of COM Artifacts from UML Models

The Unified Modeling Language™ (UML™) profile defined in Rational Rhapsody supports generation of the following COM artifacts:

- COM IDL files
- C++/ATL classes
- In-process and executable COM servers

These artifacts are described in subsequent chapters.

Rational Rhapsody Threads and the COM Apartment Model

There is no direct mapping between Rational Rhapsody threads and any other UML modeling construct and the COM apartment/threading model for a class under design. Therefore, you can freely define the apartment model for every generated ATL class using the `ThreadingModel` property (under `ATL::Class`).

Note the following threading behavior:

- **Reactive classes and COM threads**—Reactive classes are generated with guarded destructors. Object destruction that is caused by a zero reference count is done only after all event processing is finished.

- **Concurrency and COM threads**—Rational Rhapsody supports only sequential ATL classes, not active ones. This restriction is enforced with a check.
- **Process termination**—Upon termination of a Rational Rhapsody animation session, the generated code calls `CoUninitialize()` before terminating the thread.

Hello World Example

This section describes how to use Rational Rhapsody to design a client/server application to display “Hello World” using COM.

The general steps are as follows:

1. Set up Rational Rhapsody to use COM.
2. Create a COM executable server.
3. Create a COM client.
4. Run the client to invoke the server.

The following sections describe these steps in detail.

Step 1: Setting Up Rational Rhapsody to Use COM

To use COM with Rational Rhapsody, you must do the following:

1. Perform a custom install of Rational Rhapsody, selecting the run-time and framework sources for C++.
2. Open a DOS window and change directory to the `<Rhapsody>\Share\LangCpp` directory. Rebuild the framework using the command appropriate for your platform. For example, for Windows NT® systems, enter the following command:

   ```
   ..\etc\msmake.bat msbuild.mak
   ```

3. Add the `<Rhapsody>\Share\LangCpp\lib` directory to your `PATH` environment variable.

Step 2: Creating a COM Executable Server

Create an executable server with Rational Rhapsody as follows:

1. Start Rational Rhapsody and create a new project named `COM_Example`.
2. Add a component named `COMServer` to the project.
3. Set the component stereotype to «`COM EXE`».

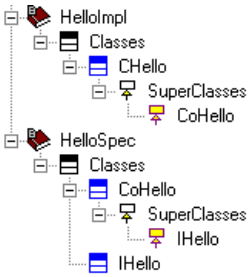
Dividing the Component into Packages

The server component will consist of two packages: one for the interface specification and another for the implementation.

Do the following:

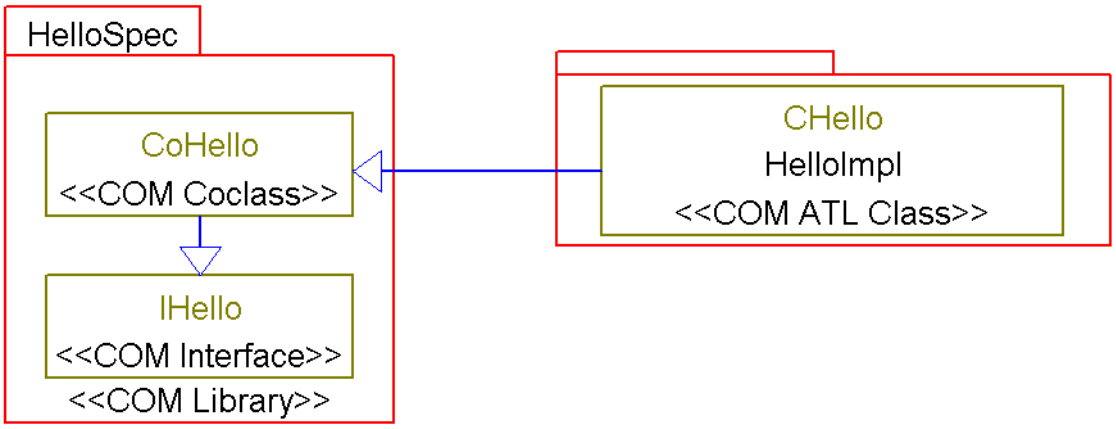
1. Add two packages named `HelloSpec` and `HelloImpl` to the `COMServer` component.
2. Give the `HelloSpec` package a stereotype of «`COM Library`». Do not give the `HelloImpl` package any stereotype.
3. Make the following changes to the `HelloSpec` library:
   a. Add a class named `IHello` and give this class a stereotype of «`COM Interface`».
   b. Add a class named `CoHello` and give this class a stereotype of «`COM Coclass`».
   c. From the `CoHello` class, add a superclass relation to the `IHello` interface.
4. To the `HelloImpl` package, add a class named `CHello` and give this class a stereotype of «`COM ATL Class`».
5. From the `CHello` ATL class, add a superclass relation to the `CoHello` coclass in the `HelloSpec` library.

   The following figure shows the resultant browser view.



6. Create an object model diagram from the new model elements as follows:
   a. Select **Tools > Diagrams > Object Model Diagram**; click **New** in the resultant dialog box.
   b. In the New Diagram dialog box, type a name for the diagram, check the **Populate Diagram** option, then click **OK**.
   c. In the Populate Diagram dialog box, select **Relations Among Selected**, leave all the types of relations selected, select the `HelloImpl` library and the `HelloSpec` package, and use **Orthogonal** layout style.
7. Click **OK**.

Rational Rhapsody creates the OMD. Rearrange the OMD so it looks like the following figure.



Adding an Operation to the COM Object

To add an operation to display the “Hello World” message, follow these steps:

1. To the `CHello` interface, add an operation called `SayHello`.
2. Specify the return type of the `SayHello` operation as `HRESULT` (uncheck the **Use existing type** option and type `HRESULT` in the **C++ Declaration** field).
3. Copy the `SayHello` operation from the interface to her COM ATL class by pressing Ctrl and dragging the operation from `IHello` to `CHello` in the browser.
4. Make sure the `CHello` operation’s `ATL::Operation::STDMETHOD` property is set to `True`.
5. Set the implementation of the `SayHello` operation in the `CHello` class to the following:

```
MessageBox(NULL, “Hello World”, “Hello World!”,
    MB_OK);
return S_OK;
```

Building the COM Server

To build the server component, follow these steps:

1. In the browser, highlight the `COMServer` component, right-click, and select **Features** from the pop-up menu. In the **Selected Elements** field, check the `HelloSpec` library and the `HelloImpl` package
2. For the `DefaultConfig` configuration of the `COMServer` component, verify that the `CG::Configuration::StartFrameworkInMainThread` property is set to `False`.
3. Set the `COMServer` component as the active component.
4. Generate code for the `DefaultConfig` configuration of the `COMServer` component (**Code** > **Generate**).
5. Build the `COMServer.exe` executable (**Code** > **Build COMServer.exe**).
6. When the build is done, open a DOS command prompt window, change directory to the location of the `ComServer.exe` application, and register the server by executing the following command at the command prompt:

```
> COMServer.exe /RegServer
```

The COM server is now built and registered.

The following table lists the commands to register and unregister the different types of COM servers.

COM Server Type	Register Command	Unregister Command
«COM EXE»	`> <server>.exe /RegServer`	`> <server>.exe /UnregServer`
«COM DLL»	`> regsvr32 <server>.dll`	`> regsvr32 /U <server>.dll`

Step 3: Creating a COM Client

To create the COM client that calls the server to print “Hello World”, follow these steps:

1. Create a new component named `Client` and set its type to **Executable**.
2. Set `Client` as the active component.
3. Set the `COM::Configuration::COMEnable` property for this component to `Client`.
4. Add a package named `Client` to the model.
5. Add a class named `HelloClient` to the `Client` package.
6. Add an attribute named `caller` of type `IHelloPtr` to the `HelloClient` class.

7. Add an operation named `Call` to the `HelloClient` class, give it a return type of `void`, and set its implementation to the following:

```
caller->SayHello();
```

8. Add explicit initial instances of the following classes to the `DefaultConfig` configuration of the `Client` component (under the Initialization tab for the configuration):
   - `HelloClient` in the `Client` package
   - `CoHello` in the `HelloSpec` package
9. Add the following initialization code to the `DefaultConfig` configuration of the `Client` component:

```
p_HelloClient->setCaller(m_pUnkCoHello);
p_HelloClient->Call();
```

10. Add an `include` for the `Client` component header to the `HelloClient` class by setting the `CPP_CG::Class::SpecIncludes` property for the `HelloClient` class to `MainClient.h`.
11. Add the `Client` package to the scope of the `Client` component (under Selected Elements).
12. Generate and make the `Client` component.

Step 4: Running the Client to Invoke the Server

Run the `Client` executable. The client invokes the server to display a message box that says "Hello World." Click **OK** to close the message box.

Code Generation

Rational Rhapsody provides several *stereotypes* for model elements (components, packages, and classes) that adhere to COM. These stereotypes provide input for COM IDL code generation. This section describes the relations between them and any restrictions that might apply to model elements that use them.

Another stereotype is applied to ATL classes that implement COM interfaces, which provides input for C++ code generation of ATL classes. See the section **C++ Code Generation Phase** for detailed information.

IDL Code Generation Phase

To create a COM component, you must explicitly declare it to be either a library, an in-process server, or an executable server by applying one of the following stereotypes to the component:

- «`COM DLL`»—Creates an in-process (inproc) server
- «`COM EXE`»—Creates an executable server
- «`COM TLB`»—Creates an interface library

COM IDL code is generated for components with any one of these stereotypes. IDL code generation is blocked for any model containing COM constructs when the component to be built does not have one of these stereotypes. This prevents erroneous COM element definitions in the generated IDL.

The following sections describe each of these stereotypes in detail.

«COM DLL» Stereotype

The «COM DLL» stereotype is applied to components that are to be built into COM inproc servers that take the form of dynamic link libraries (DLLs). When building a DLL server, there are two ways to compile and link:

- By merging the ProxyStub.dll and TLB into the server (DLL)
- By generating the ProxyStub.dll and TLB separately from the server (DLL)

«COM EXE» Stereotype

The «COM EXE» stereotype is applied to components that are to be built into out-of-process servers that take the form of executable programs. To compile and link an executable server, you generate the ProxyStub.dll and TLB separately from the server (EXE).

«COM TLB» Stereotype

The «COM TLB» stereotype is applied to components that are to be built into a TLB (a library of COM interfaces) and a ProxyStub.dll file. Running a make on such components invokes first the Microsoft MIDL compiler and then the C++ compiler.

The first (MIDL) phase of the make yields a TLB file. It also yields the sources for the ProxyStub.dll file, if the COM::Configuration::GenerateProxyStubDll property is set to True. In this case, the second (C++ compiler) phase compiles the sources yielded by the first phase into the ProxyStub.dll file. If the property is left as False (the default value), the ProxyStub.dll file is not built.

A component with a «COM TLB» stereotype can have only the following elements within its scope:

- Packages stereotyped as «COM Library»
- Classes stereotyped as «COM Interface»
- Classes stereotyped as «COM Coclass»

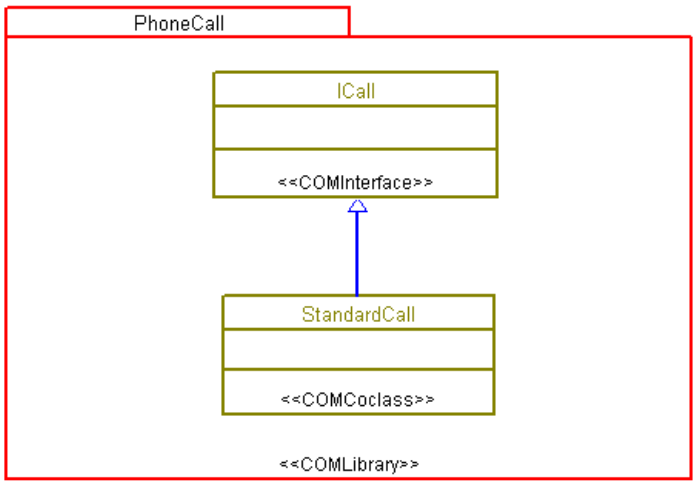
«COM Library» Stereotype

To create a package of COM constructs, you must explicitly declare the package to be a COM library. This is done by applying the «COM Library» stereotype to the package. A package with a «COM Library» stereotype is mapped to a COM IDL library.

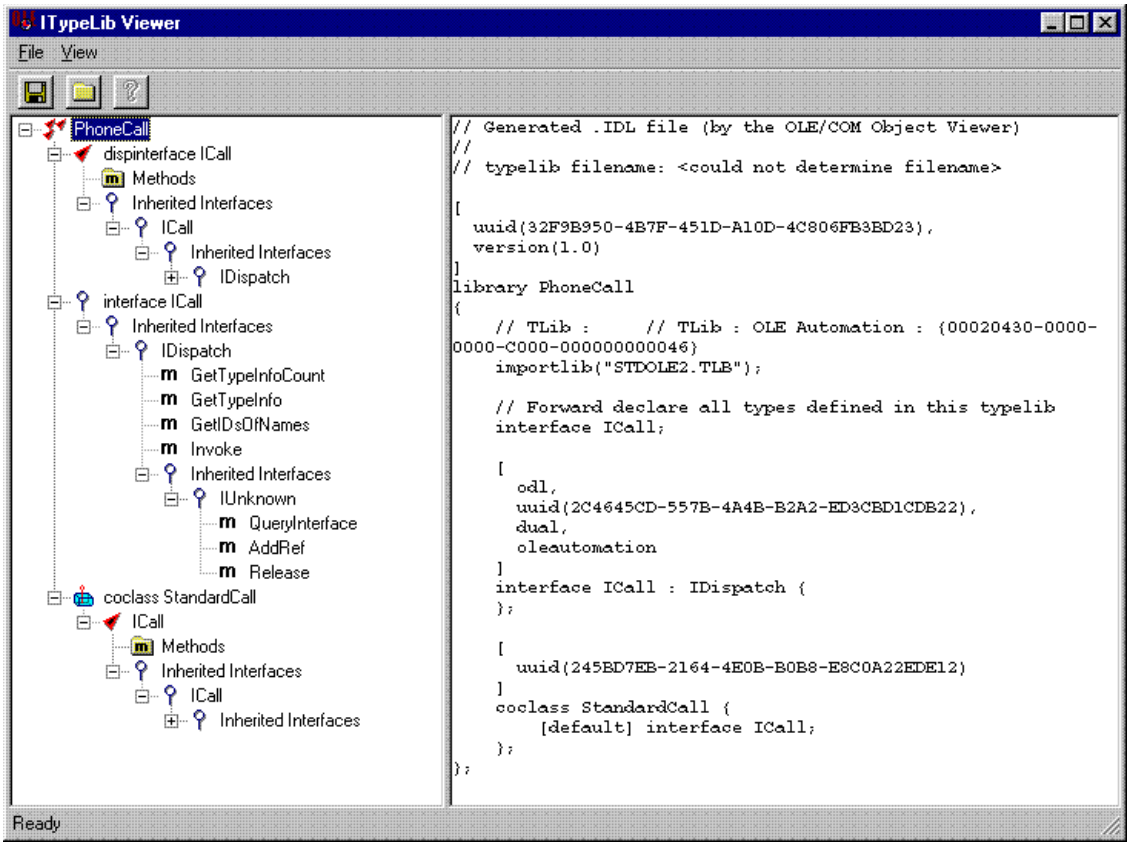
The «COM Library» stereotype is applied to packages that contain COM-stereotyped classes. A «COM Library» package can contain only classes that have one of the following stereotypes:

- «COM Interface»
- «COM Coclass»

The COM IDL code generator generates a COM TLB file containing the «COM Library» (package) and «COM Coclasses» that expose the «COM Interfaces». For example, the following object model diagram would be generated into a COM TLB file containing the PhoneCall library, which contains the StandardCall coclass, which exposes the ICall interface.



The following figure shows the IDL code generated for the TLB (in the Microsoft Visual Studio OLE View interface type library viewer).



«COM Interface» Stereotype

To adhere to COM, you must explicitly declare a class as either a COM interface, a coclass, or an ATL class. To do this, apply one of the following stereotypes to the class:

- «`COM Interface`»
- «`COM Coclass`»
- «`COM ATL Class`»

The «`COM Interface`» stereotype is applied to classes. It indicates that the COM IDL code generator should map the class during code generation to an interface in IDL. Rhapsody does not generate C++ code for «`COM Interface`» classes.

«COM Interface» Attributes

By default, attributes of a «COM Interface» class are mapped in the generated IDL to a pair of accessor/mutator interface operations. For example, the following IDL code is generated for a «COM Interface» IA with an attribute a1:

```
interface IA : IDispatch {
//  Interface Operations
   [propput, id(2)] HRESULT a1([in]int val);
   [propget, id(2)] HRESULT a1([out, retval]int *pval);
};
```

The implementation property (under COM::Attribute) specifies how these operations are generated. The possible values of the implementation property for «COM Interface» attributes are as follows:

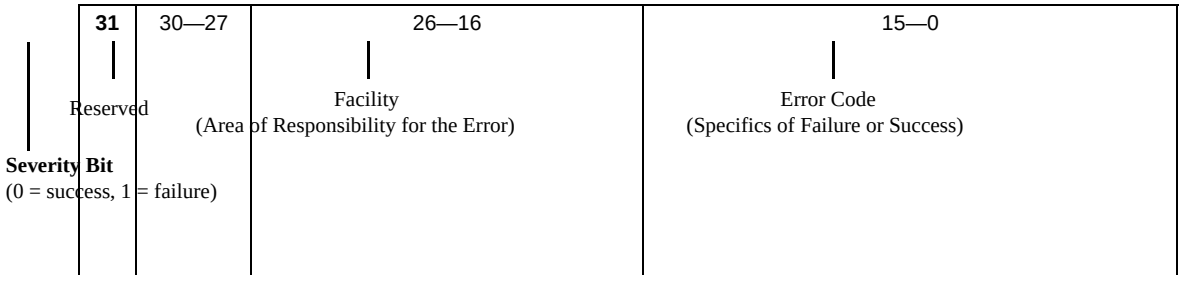
- propget—Generate an accessor only.
- propput—Generate a mutator only.
- propputref—Generate a by-reference mutator only.
- propget&propput—Generate both an accessor and a mutator (the default value).
- propget&propputRef—Generate both an accessor and a by-reference mutator.

«COM Interface» Operations

Operations of a «COM Interface» class are mapped in the generated IDL to interface operations with a return type of HRESULT. For example, the following IDL code is generated for a «COM Interface» IA with an operation op1():

```
interface IA : IDispatch {
//  Interface Operations
   [id(1)] HRESULT op1();
};
```

The return type HRESULT for interface operations is a 32-bit value that indicates success or failure. Bit 31 is the most significant bit and the one that is checked to determine the success or failure of the operation. The following figure shows the purpose of the bits in HRESULT.



You can use the COM macros `SUCCEEDED` or `FAILED` in your code to check for the success or failure of an interface operation. For example:

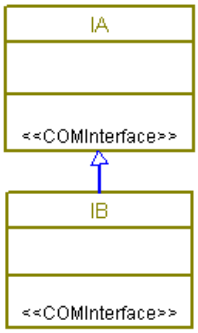
```
HRESULT hr = IA->op1();
if (FAILED(hr))
{
   // do something
}
```

To specify a return type of `HRESULT` for an operation in Rhapsody, clear the **Use existing type** option and type `HRESULT` in the **C++ Declaration** field when you define the operation. Rhapsody checks for deviations from supported return types for «`COM Interface`» operations.

All operation attributes have a property to specify their description clause, which is enclosed in square brackets ([]). See the **COM Description Clause** section for more information.

«COM Interface» Inheritance

A «`COM Interface`» class can inherit from another «`COM Interface`» class.



Note the following:

- A standard (non-COM) class can inherit directly from a «`COM Interface`» class.
- A «`COM Interface`» class cannot inherit from a non-«`COM Interface`» class.
- COM does not allow multiple inheritance for «`COM Interface`» classes.

«COM Interface» Relations

Relations between «COM Interface» classes are mapped to elements in the generated IDL depending on the type and multiplicity of the relation.

The following rules apply to incoming/outgoing relations for «COM Interface» classes:

- An outgoing or symmetric relation arrow leaving a «COM Interface» class can target only another «COM Interface».
- An incoming relation arrow entering a «COM Interface» class can originate in either a regular class or another «COM Interface».
- An outgoing or symmetric relation arrow from a «COM Interface» class is mapped to accessor/mutator methods (such as get, set, add, and clear) in the generated IDL. The accessor's return type and the mutator parameter's type are the same as that of the target «COM Interface». In addition, the mutator's parameter has a direction of in.

Predefined and External Interfaces

All «COM Interface» classes must inherit from IUnknown, either directly or indirectly. Any «COM Interface» class that does not inherit from another interface inherits from IUnknown by default. The Type property (under COM::Interface) specifies the base class for interfaces. The possible values are as follows:

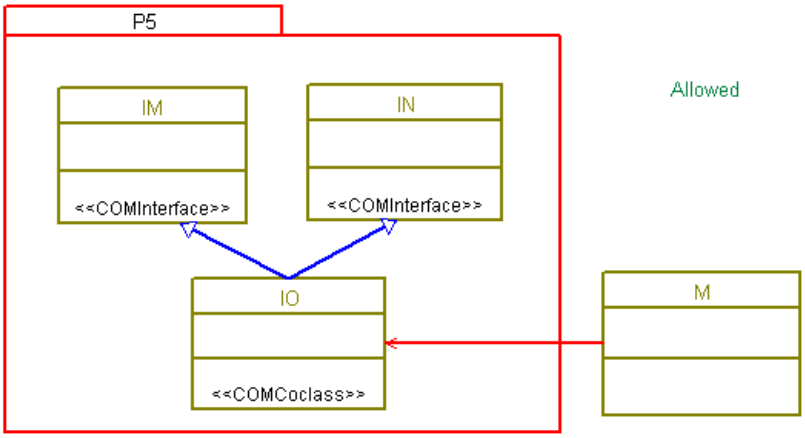
- Dual—IDispatch base class (default). IDispatch inherits from IUnknown.
- Custom—IUnknown base class
- dispinterface—Pure automation interface

«COM Coclass» Stereotype

The «COM Coclass» stereotype is applied to classes that are to be generated into IDL coclasses. A «COM Coclass» class that inherits from a COM interface exposes the COM interface. For example, in the OMD the StandardCall coclass exposes the ICall interface.

Note the following:

- The «COM Coclass» stereotype indicates that the COM IDL code generator should map the class during code generation to a coclass in COM IDL. Rhapsody does not generate C++ code for «COM Coclass» classes.
- A «COM Coclass» class can inherit from one or more «COM Interface» classes, but not from any other type of class.
- A «COM Coclass» class can only have incoming relations/associations, but not outgoing ones.



«COM Coclass» Inheritance

The DefaultInterface property (under COM::coclass) specifies the default interface that a «COM Coclass» class should expose. This property is empty by default. To override the property, assign it the name of the «COM Interface» class that you want a coclass to expose. If you set the DefaultInterface property while a package or component is selected, the property is automatically applied to all new «COM Coclass» classes that you create in that package or component.

«COM Coclass» Associations

A «COM Coclass» can use a «COM Interface» class. In UML, this is expressed in terms of an outgoing dependency relation from the «COM Coclass» to the «COM Interface» class with a

«ConnectionPoint» stereotype on the dependency relation. The DefaultInterface property (under COM::Dependency) specifies the default interface that a «COM Coclass» requires.

See **COM Connection Points** for more information on connection points.

COM Description Clause

In COM IDL syntax, interfaces, coclasses, libraries, operations of an interface, arguments of those operations, and interfaces exposed by a coclass all have a description clause, enclosed in square brackets, in the generated IDL.

For example, the description clause generated for a «`COM Interface`» class `IA` is as follows:

```
//## class IA
[
   dual,
   uuid(6559BEA2-8D6D-11d4-80A5-005056C54916),
   pointer_default(unique),
   object
]
```

Rhapsody has a dedicated set of properties (for packages, classes, operations, and arguments) that specify how the description clause is generated. One of the most important COM properties stored in these properties is the UUID (universal unique identifier) or GUID (global unique identifier).

The description clause properties fall into one of three categories:

- **Automatically generated**—Rhapsody automatically generates a unique ID into the `uuid` property (under `COM::Interface/Library/Operation`) for a COM dispinterface, library, or operation, respectively. The ID is generated on the first code generation for the element if the default property value (empty string) is not overridden.
- **Manually set**—If you override the value of the `uuid` property, Rhapsody uses this value for the uuid in the description clause.
- **Optional, verbatim property**—The `AppendToClause` property (under `COM::Interface/Attribute/Argument/Library/coclass/Operation`) enables you to add free text that is generated into the end of the description clause, before the closing bracket.

C++ Code Generation Phase

The «COM ATL Class» stereotype is applied to classes that *implement* COM interfaces. A «COM ATL Class» class that inherits from a «COM Coclass» stereotyped class implements the interface that the coclass exposes. C++ code, not IDL code, is generated for ATL classes.

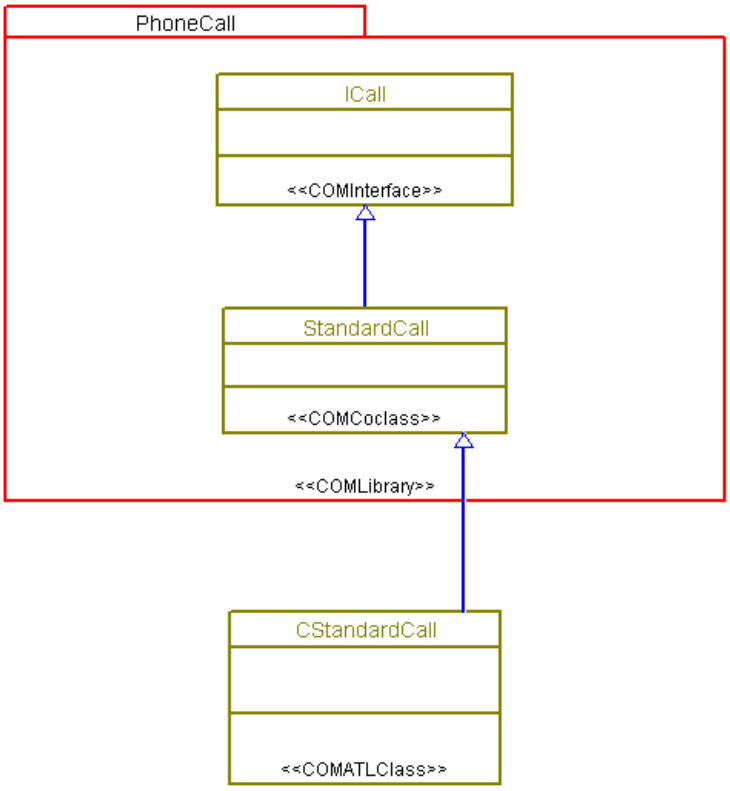
A «COM ATL Class» cannot have initial instances; instantiation of ATL classes must be done using the COM system methods.

Rational Rhapsody code generation properties provide support for the following ATL concepts:

- Threading model
- Dual/custom interface
- Aggregation
- Support error information
- Connection points
- Free-threaded marshaller

«COM ATL Class» Stereotype

In the following figure, the `CStandardCall` ATL class implements the `ICall` interface, which the `StandardCall` coclass exposes.



The «`COM ATL Class`» stereotype indicates that the C++ code generator should map the class to a C++ class with additional ATL instrumentation. COM IDL code is not generated for this kind of class.

«COM ATL Class» Operations and Macros

Rhapsody adds a set of standard operations, macros, and keywords to each «COM ATL Class» stereotyped class. Properties control the generation of these elements. For example, the DeclareClassFactory property specifies a template for the generation of code for ATL classes. The following is an example of the kind of template that could be entered in this property:

```
class $DeclarationModifier $class :
    public CComObjectRootEx<$ThreadModel>,
    public CCOMCoClass<$class, &CLSID_def>,
    public IDispatchImpl<Idef, &IID_Idef,
           &LIBID_ALL_KIND_OF_ATLLib>
{
    ...
}
```

This template references information stored in the following ATL::Macro properties:

- ATLRootClass—Specifies the ATL root class
- ATLClassObject—Specifies the ATL class that implements a COM coclass
- IDispatchImpl—Provides support for animation

ATL Operations

Rhapsody adds several operations to «COM ATL Class» stereotyped classes. These operations are exposed in the browser, so you can view, edit, or delete them as needed. Once generated, these operations are not automatically deleted if you remove the «COM ATL Class» stereotype from the class.

The following operations are automatically added to ATL classes:

- FinalConstruct and FinalRelease—ATL class initialization and cleanup methods. You must provide the implementation.
- InterfaceSupportsErrorInfo—Provides support for IsupportErrorInfo. This operation is generated if the SupportErrorInfo property (under ATL::Class) is set to Yes. Rhapsody provides a default implementation, which you can override.
- CreateFreeThreadedMarshaller and ReleaseFreeThreadedMarshaller—Creates and releases the free-threaded marshaller, respectively. The IMarshal implementation object is generated, along with these two operations, if the FreeThreadedMarshaller property (under ATL::Class) is set to Yes. You must explicitly call the create and release operations in your code (for example, in FinalConstruct and FinalRelease).

ATL Macros

The following macros are generated for «COM ATL Class» stereotyped classes:

```
DECLARE_RHAPSODY_REGISTER(CLSID_$coclass, "$TypeName",
   "$VersionIndepProgID", "$ProgID", "$ThreadingModel",
   COMPAPPID)
DECLARE_PROTECT_FINAL_CONSTRUCT()
BEGIN_COM_MAP($class)
   COM_INTERFACE_ENTRY(Idef)
   COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
```

The ATL::Macro properties that control how these macro templates are generated include the following:

- ClassRegistration—Specifies the ATL class registration macro
- DeclareProtect—Specifies the macro that protects the ATL object from being deleted if, during FinalConstruct, the nested object increments the reference count and then decrements the count to 0
- BeginInterfaceMap—Specifies the start macro for a COM map of an interface class
- InterfaceEntry—Specifies the ATL macro that defines the COM map interface entry point
- EndInterfaceMap—Specifies the end macro for a COM map of an ATL class

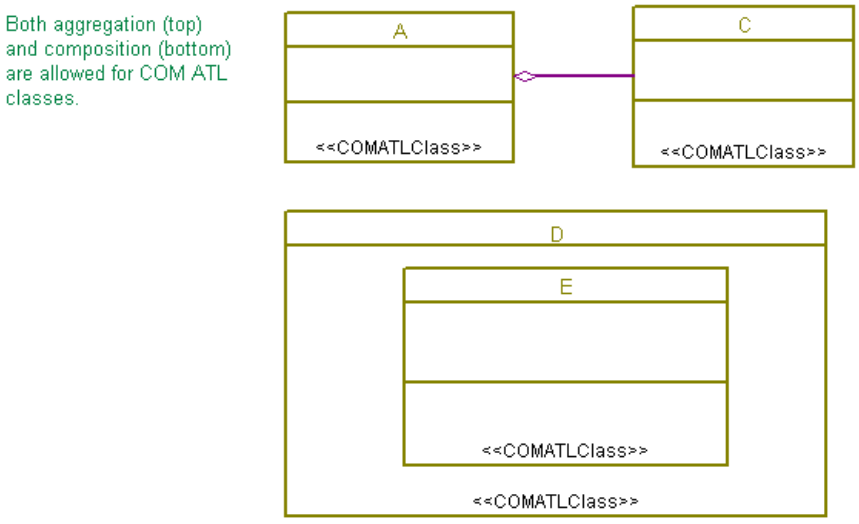
«COM ATL Class» Aggregations

Rhapsody does not support COM aggregation of ATL classes in terms of generating the AGGREGATABLE keyword for «COM ATL Class» stereotyped classes. However, it does support both aggregation and composition for these classes in the UML sense of generating accessors/mutators for them. You can turn off this support using the Aggregation property (under ATL::Class). To enable or disable aggregation for a «COM ATL Class», the Aggregation property must be set for the aggregated (part) class, *not* the aggregate (whole) class.

The possible values of the Aggregation property are as follows:

- Yes—The ATL class can be aggregated by another class. This is the default value.
- No—The ATL class cannot be aggregated by another class.
- Only—The ATL class can exist *only* as an aggregated class.

The following figure shows an OMD with aggregation and composition of COM ATL classes.



COM Components

Three type components contribute to the building of systems with COM:

- **Servers**—COM-enabled components that are able to respond to remote invocations
- **Clients**—Components that use the server IDL code, as represented by either Rhapsody model elements or an external IDL file
- **COM interfaces**—Components that play a significant role in the design of both clients and servers

COM Servers

COM servers contain classes that implement COM interfaces. You can build COM servers in two ways:

- As a single component including both the interface and implementation in the same component.

  A component that contains both interfaces and implementing classes requires both IDL and C++ compilation.

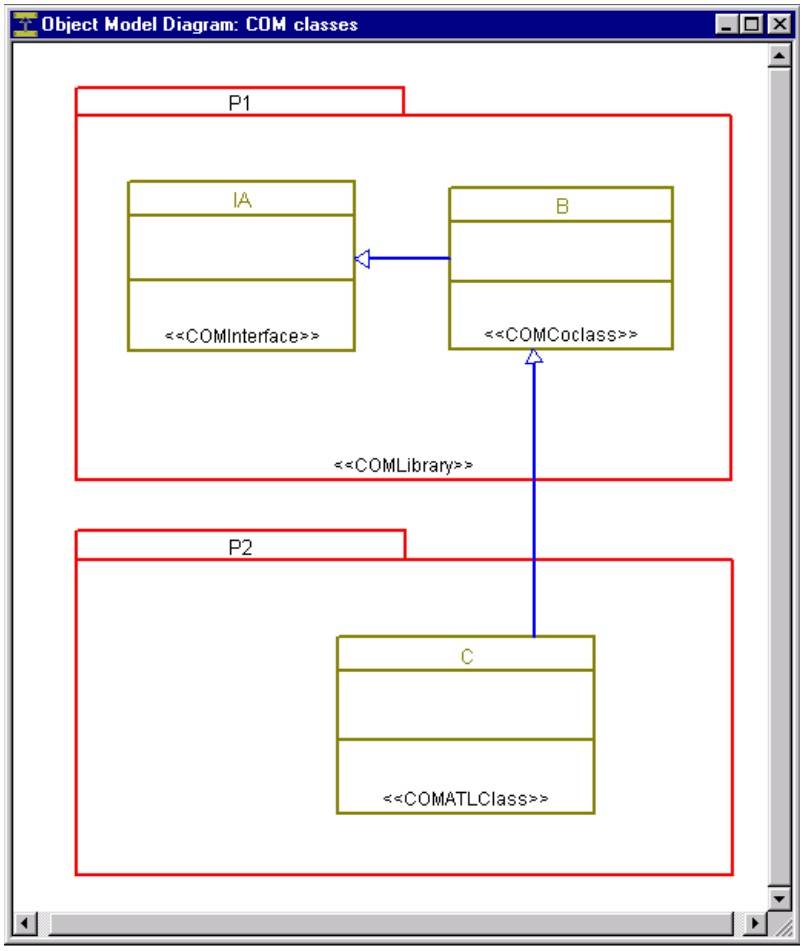
- As separate components providing an interface/implementation separation in the deliverable.

  A component that contains only the implementing classes requires only C++ compilation. In this case, the compilation requires both a `ProxyStub.dll` and a TLB, which must be defined elsewhere (within a separate «`COM TLB`» component).

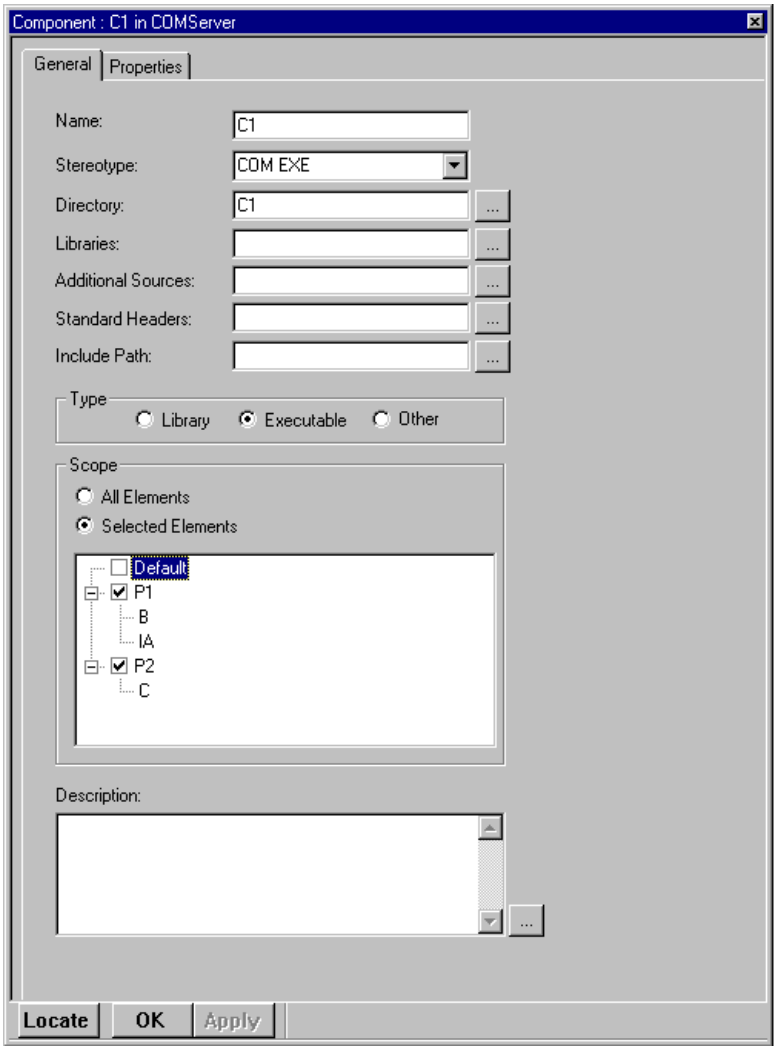
For example, the following figure shows two packages:

- Package `P1` is a «`COM Library`» that contains «`COM Interface`» `IA` and «`COM Coclass`» `B`.
- Package `P2` contains «`COM ATL Class`» `C`, which implements the «`COM Interface`» in package `P1`.

Packages `P1` and `P2` can be placed in either the same component or in different components.

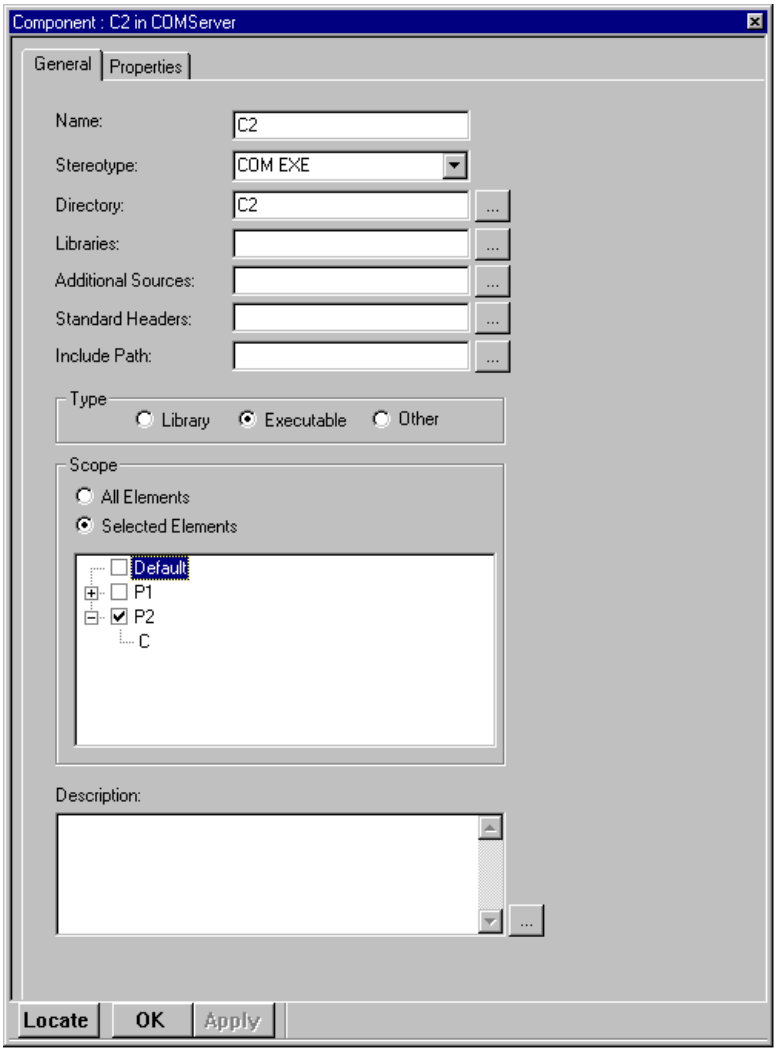


The following figure shows a specification for a component that contains both package `P1` (a «`COM Library`» containing the «`COM Interface`» to be implemented) and package `P2` (which contains the implementing «`COM ATL Class`» `C`). This component requires both IDL and C++ compilation.



Only DLLs can contain both COM interfaces and implementing classes in the same component. In this case, the `ProxyStub.dll` and TLB are merged into the same server (DLL).

The following case shows a specification for a component that contains only an implementing «`COM ATL Class`» named `C`. This component requires only C++ compilation. It expects the `ProxyStub.dll` and TLB files to be provided in a separate component. This is the preferred design for COM server components.



Both DLLs and EXEs can be built from components in which the interfaces and implementing classes are allocated to separate components. In this case, the `ProxyStub.dll` and TLB are generated separately from the server (DLL or EXE).

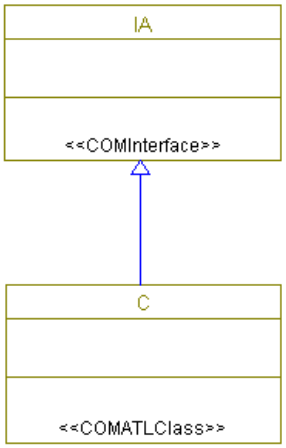
Implementation of a COM Server

A «COM ATL Class» stereotyped class that inherits from a «COM Coclass» stereotyped class functions as a COM server object, which implements the interfaces that the COM coclass exposes. The following is an example of the C++ code that Rhapsody would generate for the ATL class C, which represents this type of construct:

```
class ATL_NO_VTABLE C :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<C, &CLSID_B>,
   public IDispatchImpl<IA, &IID_IA, &LIBID_P1>
{...}
```

Conversely, a «COM ATL Class» stereotyped class that inherits directly from a «COM Interface» class implements the interface directly. This type of construct is useful for implementing connection points (see **COM Connection Points**).

For example, the ATL class C shown in the following figure directly implements the interface IA. Although C is also an ATL class, its code is slightly different from that of the class C. The class C shown here lacks the CComCoClass<> parent and it inherits from CComObjectRoot rather than the CComObjectRootEx<> parameterized class.



Both of these ATL classes would have a COM_MAP section.

The code for this class C would be as follows:

```
class C :
   public IDispatchImpl<IA, &IID_IA, &LIBID_P6>,
   public CComObjectRoot {...}
```

Inproc and Executable COM Servers

COM supports two different types of servers:

- An in-process (or COM DLL) server, as a dynamic link library (DLL) containing COM objects
- An out-of-process (or COM EXE) server, as an executable containing COM objects

The in-process (inproc) approach is the most common.

COM InProc Server Generation

A component that has a stereotype of «`COM DLL`» is built into a DLL or inproc server (see **«COM DLL» Stereotype**). Any COM inproc server must implement a predefined set of COM exported methods to handle issues such as module locking, unlocking, and registration. Rhapsody has several code generation properties to provide templates that specify how these methods should be generated. You can modify all of these properties to customize the generated code as desired.

These properties (under `ATL::Configuration`) are as follows:

- `InProcServerExports`—Provides a template for the `DEF` file used during DLL creation.
- `InProcServerMainModule`—Provides a template for the declaration and definition of each of the COM methods exported in the DLL. The default exported methods are `DllCanUnloadNow`, `DllGetClassObject`, `DllRegisterServer`, and `DllUnregisterServer`.
- `InProcServerMainLineTemplate`—Provides a template for the `Dllmain` function.
- `InProcServerRegistration`—Provides a template for generating the ATL server registration code.

In addition, a `<component name>.def` file is generated for an inproc server from the `ATL::Configuration::InProcServerExports` property, if the file does not already exist.

COM Executable Server Generation

A component that has a stereotype of «COM EXE» is built into an out-of-process, or executable, server. A DEF file is not generated for an executable server.

The main code section for a COM executable server is called Winmain. This main code section:

- Is encapsulated within CoInitializeEx and CoUninitialize function calls. Any other thread opened within the client scope (by an active class) is wrapped with the same initialization/uninitialization code (as specified by both the OutProcServerRegistration and OutProcServerMainLineTemplate properties).
- Registers and unregisters the server in the Windows registry using standard COM command line switches (for example, /register), as specified by the OutProcServerRegistration property.
- Registers the COM coclasses (as specified by the OutProcServerRegistration property).
- Performs the main COM loop and lifetime control (as specified by the OutProcServerMainModule property).
- Assigns a UUID to the server (as specified by the AppId property).
- Can contain any user-generated code (as specified by the OutProcServerMainLineTemplate property) in place of the main code generated by the ATL wizard.

Additional Generated Files

The following additional files are generated for both inproc and executable servers:

- A resource file is always generated for the server if the component contains a «COM Library» stereotyped package. The resource file has the same name as the server. It contains an element named TypeLibrary of type Other, which specifies the inclusion of the type library into the resource file. You can modify this file by adding your own file elements.
- The RhapRegistery.h (and .cpp) files, which contain basic routines for registering ATL classes, are copied from the $OMROOT\MakeTempl directory only if they do not already exist. You can modify these files as desired.
- The stdafx.h file is generated from the ATL::Configuration::OutProcStdAfx property for executable servers and from the InProcStdAfx property for inproc servers, if it does not already exist. You can modify this file as desired.

ProxyStub.dll Generation

The following properties control generation of the `ProxyStub.dll` file:

- `COM::Configuration::GenerateProxyStubDll`—Specifies whether Rhapsody should generate the `ProxyStub.dll`. Note that even if the property is `True`, Rhapsody generates the file only if it does not already exist.
- `COM::Configuration::ProxyStubDefFileName`—Specifies the name of the `.def` file used to generate the `ProxyStub`.

  The default value is `$componentps.def`, where `$component` is replaced with the name of the server.

- `ATL::Configuration::ProxyStubExports`—Specifies the content of the `ProxyStub.dll` file. You can modify this content as desired.

COM Clients

Code generation for COM clients is fully automated. Rhapsody generates COM client code when the `COM::Configuration::COMEnable` property for the component is set to `Client`.

Importing TLB Files

Rhapsody supports the use of existing TLB files by adding `import` statements for the appropriate TLB file to the code generated for any client class or package that has an association or «`Usage`» stereotyped dependency relation to a «`COM Interface`» or «`COM Library`».

The `import` statements can be added to a client by either of the following methods:

- You can add them manually by setting the `ATL::Configuration::TypeLibImportFormat` property.
- They can be automatically generated from a «`Usage`» dependency on a «`COM Coclass`» or «`COM Library`».

Initializing a Client

The client's `main` code section is encapsulated within `CoInitialize` and `CoUninitialize` function calls. Any other thread that is opened (by active classes) within the client's scope is wrapped with the same initialization code.

You can customize how code is generated for the client's `main` code section by modifying the following properties:

- `COMInitialize`—Adds COM initialization code to the client's `main` section. The default code is as follows:

  ```
  CoInitialize(NULL);
  ```

- `COMUninitialize`—Adds COM uninitialization code to the client's `main` section. The default code is as follows:

  ```
  CoUninitialize(NULL);
  ```

Instantiating Coclasses

In the common workflow, a client calls a COM-specific instantiation method (for example, `CoCreateInstance` or one of its variants) to instantiate a coclass. The method takes as arguments the coclass to be instantiated and the interface through which access is required. The output of the instantiation method is a pointer to the created instance.

Rhapsody supports this basic workflow by enabling you to instantiate «`COM Coclass`» stereotyped classes using the **Initial Instances** setting for a configuration. Selecting a «`COM Coclass`» as an initial instance has the following effect:

- An appropriate `import` statement is generated.
- A generic COM smart pointer is defined. The memory to which this pointer points is automatically released when the pointer is destroyed.
- The COM object is instantiated and its pointer is assigned to the smart pointer.

COM Interfaces

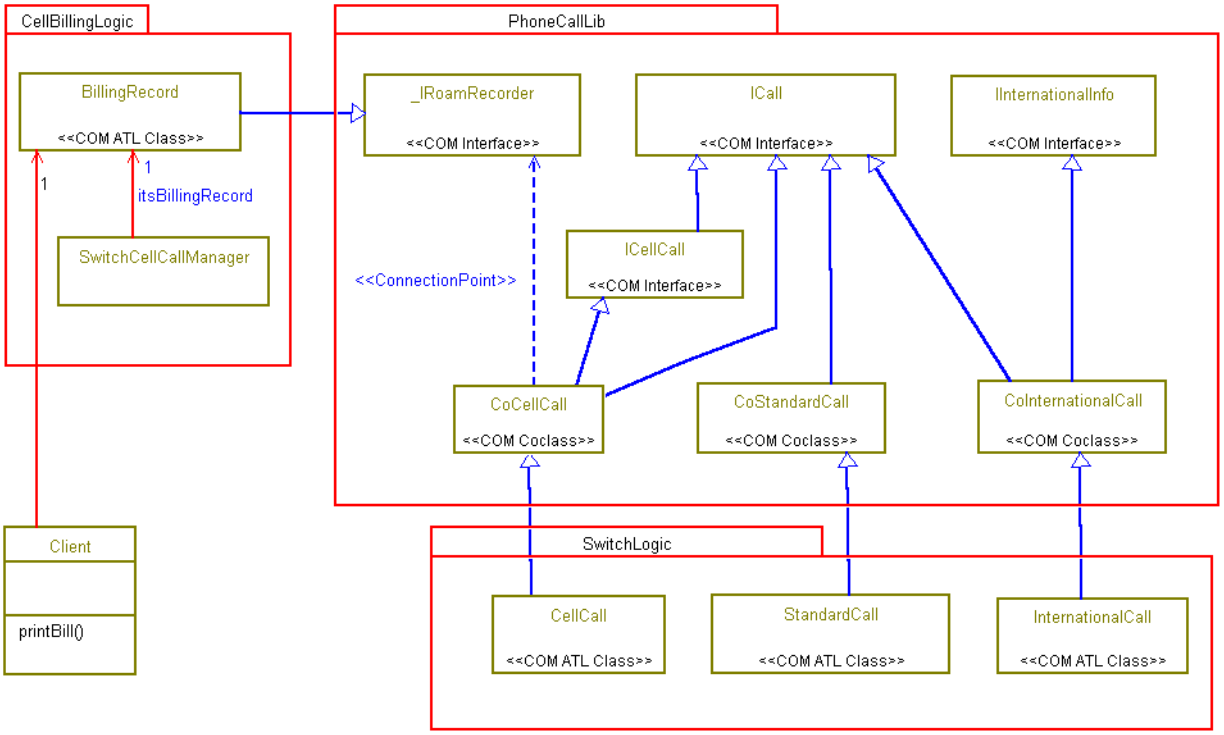
A component defined as a «COM TLB» is allocated a single «COM Library», the «COM Interface» classes in this package, and the «COM Coclass» classes that expose the interfaces. This component can have only one «COM Library» defined within its scope.

Such a component generates IDL files only. The **Build** command invokes the MIDL (Microsoft IDL) compiler, which creates the TLB file and the associated ProxyStub.dll.

Example of a Complete COM System

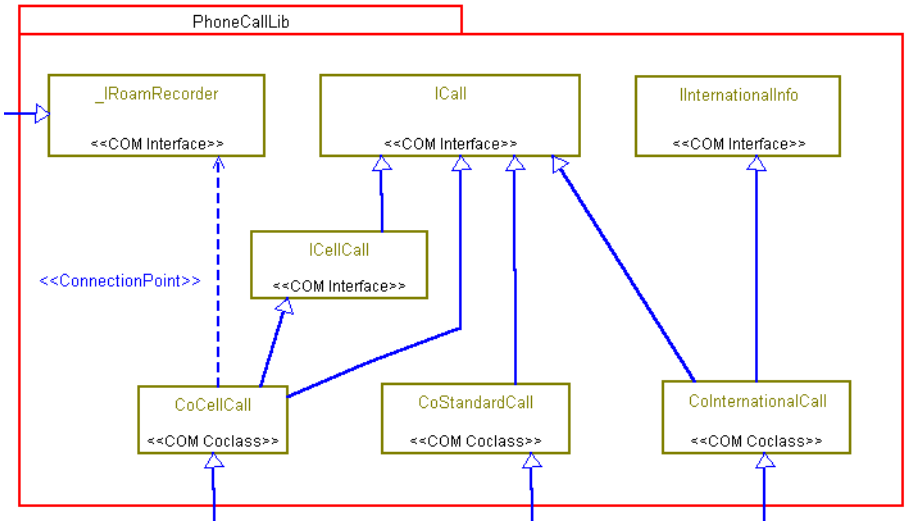
The following example of an imaginary telecom switch describes a complete COM-enabled system consisting of the following components:

- An interface consisting of a «COM TLB»
- A server consisting of a «COM DLL» that uses the interface
- Another server consisting of an executable that acts as a client in relation to the DLL server
- A class that is built into a simple executable, which is the ultimate client in this system



The PhoneCall Interface

In this example, the `PhoneCallLib` package with its aggregated classes is allocated to a Rhapsody component named `PhoneCall`, which is stereotyped «`COM TLB`».

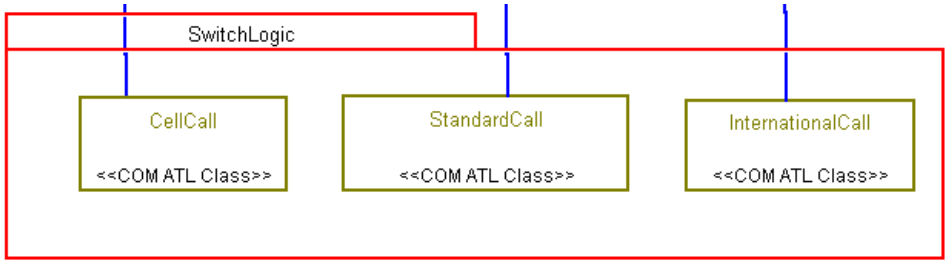


For the `PhoneCall` interface:

- Rhapsody generates a COM IDL file to describe each «`COM Interface`» that it contains.
- The interfaces reflect the visual structure (the interface inheritance tree).
- The `PhoneCall` package is a «`COM Library`» containing all the «`COM Coclasses`». The `_IroamRecorder` interface is a source interface for the `CellCall` coclass.
- The **Rhapsody Build** command invokes the MIDL compiler to create the `PhoneCall.tlb` and `ProxyStub.dll`.

The SwitchLogic Server

The `SwitchLogic` package, with its aggregate classes, is allocated to a Rhapsody component that is stereotyped as «`COM DLL`».

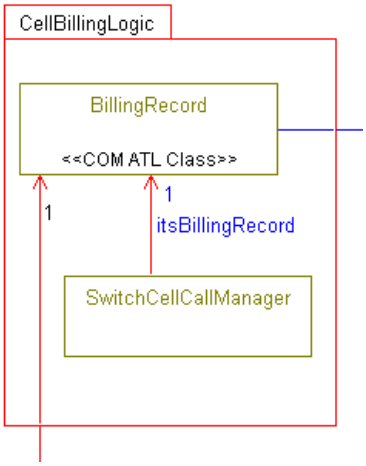


For the `SwitchLogic` DLL server component:

- Rhapsody generates ATL classes (see **C++ Code Generation Phase**).
- Connection point machinery is added to the `CellCall` ATL class (see **COM Connection Points**).
- The Rhapsody **Build** command yields a COM DLL.

The CellBillingLogic Server/Client

The `CellBillingLogic` package, with its aggregated classes, is allocated to a Rhapsody component that is stereotyped as «`Executable`». This component acts as both a server and a client.



For this component:

- Rhapsody creates COM support, as described in **COM Clients**.
- With this COM support, the `SwitchCellCall Manager` class is able to cocreate instances of `CellCall` COM objects.
- Rhapsody generates an ATL class for `BillingRecord`. This class is not a coclass, and therefore cannot be instantiated with `cocreateinstance`.

Import of a TLB

From the modeling point of view, any package using the `PhoneCall` interfaces, such as the client, simply needs to import the appropriate TLB. Rhapsody generates the `import` statements automatically. In this example, the client can access the `PhoneCall` information, as well as a `ClientDatabase` to print client bills. The `ClientDatabase` can reflect a COM TLB file to be imported into the Rhapsody system.

TLB Importer

The TypeLibrary Importer imports a COM type library into a Rhapsody model. Once a type library is imported, you can view its contents—including its interfaces, coclasses, attributes, and operations—in the Rhapsody browser. Thus, you can use the TypeLibrary Importer to analyze any kind of COM component and then create relations to coclasses, instantiate coclasses, inherit from interfaces, and implement the interfaces in your own model.

Starting the TypeLibrary Importer

To import a type library into a Rhapsody model, do the following:

1. Select **Tools** > **TypeLibrary Importer**.
2. In the resulting Open dialog box, select the type library you want to import (for example, `Atl.dll`).

   Type libraries can be encapsulated in `.tlb` (COM type library) files, `.olb` (Visual Basic object library) files, or they can be built into COM servers and contained in `.dll`, `.exe`, or `.ocx` files. Rhapsody can import type libraries from any of these types of files.

3. Click **Open**.

   If the import is successful, Rational Rhapsody displays a message indicating that the import operation was completed successfully.

The result of the import of a type library are new packages added to the model, which contain the COM interfaces, types, coclasses, inheritance relationships, attributes, and operations from the imported type library.

Note

Type libraries are imported as read-write, but code generation is disabled for them.

Assignment of COM Stereotypes

Imported elements are assigned the appropriate COM stereotypes for their respective metatypes in the Rhapsody model, as follows:

- Packages imported from a type library are given the stereotype «`COM Library`».
- Interfaces are given the stereotype «`COM Interface`».
- Coclasses, which expose interfaces, are given the stereotype «`COM Coclass`».

Imported Properties

The TypeLibrary Importer automatically sets some COM properties for imported elements. This guarantees that you can use the imported elements properly in a Rhapsody model (for example, to create COM interfaces that inherit from imported interfaces). One property that is always set for imported interfaces, libraries, and coclasses is the `uuid` property (under `COM::<metatype>`).

The import also sets the properties listed in the following table.

For...	Property
«`COM Interfaces`»	`COM::Interface::Type`
Attributes of a «`COM interface`»	`COM::Attribute::id` and `implementation`
Operations of a «`COM interface`»	`COM::Operation::id`
Arguments of operations defined in a «`COM interface`»	`COM::Argument::readonly` and `retval`

Implicit Import

If an interface or type in an imported type library is found to have been derived from a «`COM Interface`» defined in another type library, the necessary parent interfaces and types from that type library are also imported.

Importer Error Handling

The TypeLibrary Importer handles import errors as follows:

- If the importer encounters a fatal error (for example, if it is unable to add a package to the model), it displays an error message and terminates the import.
- If the importer encounters a non-fatal error, it displays a warning message and continues, after which it compiles a report of all the warning messages that occurred during the import.

Refreshing an Imported Type Library

When the TypeLibrary Importer imports a type library, it creates a clone of it in the current Rhapsody project. There is no connection between this clone and the original type library. The imported library will not reflect any changes made to the original type library, which must be reimported if the Rhapsody model is to be kept current when the type library is updated.

To refresh a previously imported type library, do the following:

1. Select **Tools** > **Type Library Importer**.
2. In the resulting Open dialog box, select the previously imported type library and click **Open**.
   Rational Rhapsody detects that the library already exists in the model.
3. Select one of the following options:
   - **Import As** to import the library with a different name.
     Rational Rhapsody imports the type library, resulting in two packages:
     - The out-of-date type library is contained in the package with the existing name (for example, `ATLLib`).
     - The updated type library is contained in the package with the new name (for example, `TakeTwo`).
   - **Overwrite** to overwrite the existing library.
     In this case, Rational Rhapsody automatically creates a backup of the existing library for you. The result is again two packages, but this time:
     - The out-of-date type library is contained in the package named `<library>_bkupn`, where *n* is a number beginning with 1.
     - The updated type library is contained in the package with the existing name (for example, `ATLLib`).

Synthesizing Diagrams from Imported Type Libraries

Once you have imported a type library, you can create an OMD from it to show its internal structure.

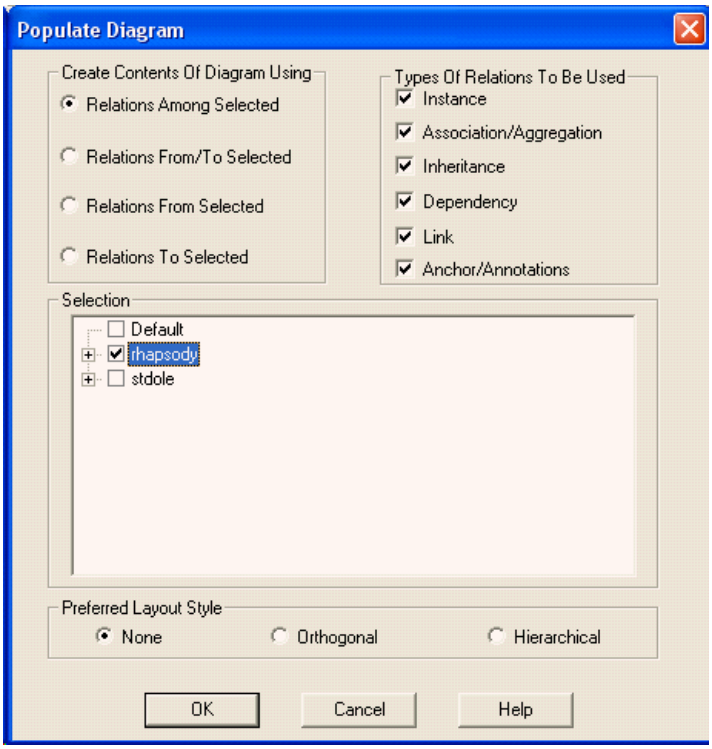
Note

The following example shows the OMD that can be synthesized from the import of the Rhapsody type library (`rhapsody.tlb`), which is shipped with Rhapsody and encapsulates the Rhapsody COM API.

Do the following:

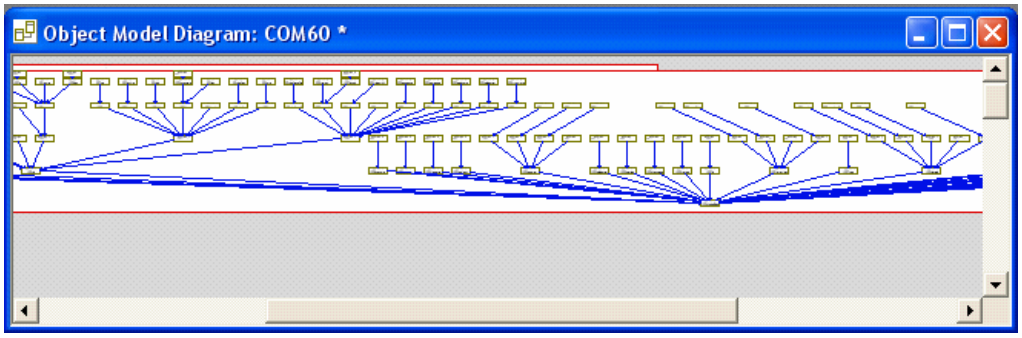
1. Import the `rhapsody.tlb` type library from the root directory of your Rational Rhapsody installation.
2. In the main toolbar, click the **Object Model Diagram** tool.
3. In the resulting Open object model diagram dialog box, click **New**.
4. In the resulting New Diagram dialog box, enter a name for the new diagram and check the **Populate Diagram** option. Click **OK**.

5. In the resulting Populate Diagram dialog box, select the package containing the imported type library (in this case, `rhapsody`) so all the interfaces contained in the library will be included in the diagram.



6. Click **OK** to dismiss the dialog box.

   The following figure shows the result—an OMD showing the structure of the `rhapsody` type library and its interfaces, which are used in the Rhapsody COM API.



7. Zoom in on the diagram. You can see that:

- The Rhapsody type library is represented as a package named `rhapsody` with a stereotype of «`COM Library`».
- Each interface is represented as a class with a stereotype of «`COM Interface`».

Examine the features of some of the imported «`COM Interfaces`» to see that their attributes and operations have been imported. Note their types.

Examine the properties of some of the imported «`COM Interfaces`» to see how the COM properties have been set for imported elements.

COM Connection Points

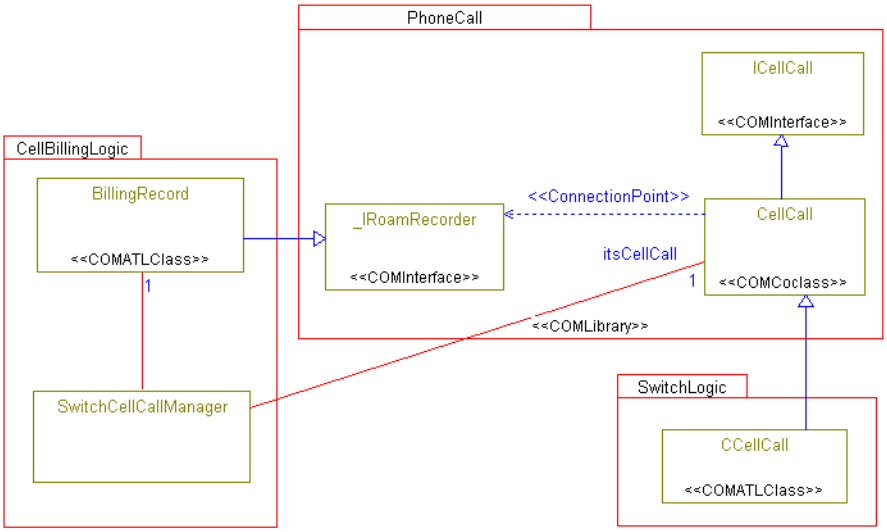
COM connection points provide one way to implement the publish/subscribe design pattern. In this design pattern, the client subscribes to a service provided by the server, then goes to sleep. When the server is ready to publish the service, it sends a callback message to the client, which wakes up and responds to the message.

A design implementing a COM connection point has the following characteristics:

- The «`COM Interface`» class is defined as a connection point in the COM IDL using the `source` keyword.
- The implementing «`COM ATL Class`» class must implement the `IConnectionPointContainer` interface and a `CONNECTION_POINT_MAP`.
- The client of that server must implement the connection point.

In the UML view of a client/server design, the server consists of an interface, a coclass, and an implementing (ATL) object, and the client class uses the implementing object. In the UML view of a connection point structure, the connection point is represented as a dependency, rather than an inheritance, relation between a coclass and an interface. The coclass uses the connection point interface, rather than exposing it. The dependency is stereotyped «`ConnectionPoint`», and the `SOURCE` keyword is generated in the IDL for the interface on which the coclass depends.

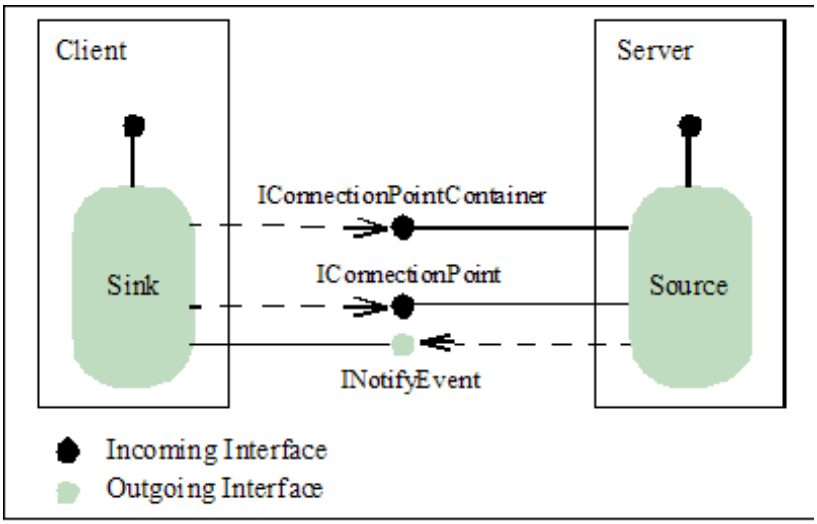
For example, consider the following model of a cellular phone call. The dependency between the `_IRoamRecorder` interface and the `CellCall` coclass is stereotyped «`ConnectionPoint`». The `CellCall` coclass uses the `_IRoamRecorder` interface, while exposing the `ICellCall` interface. The client, the `SwitchCellCallManager`, implements the `_IRoamRecorder` connection point.



The connection point machinery is hidden (for example, the `IConnectionPointContainer` and `IConnectionPoint` interfaces do not appear in the OMD). Rhapsody automatically generates this machinery if one or more connection points are defined for a coclass.

COM View Versus UML View of Connection Points

COM literature typically shows connection points as having the following structure:



The server implements the connection point facilities using `IConnectionPointContainer` and `IConnectionPoint`. The client implements the outgoing interface using `INotifyEvent`. From the client's perspective, the outgoing interface is simply any COM interface that it must implement. In practice, the client implementation using the ATL mechanism is much more complex.

Rhapsody supports the design of COM connection points by enabling you to build any of the following with a UML model:

- A server with a connection point interface.
- A client that implements a specific connection point defined in a Rhapsody model.
- A client that implements a specific connection point defined outside of a Rhapsody model (that is, in an external TLB).
- A complete client/server design that reflects the connection point structure.
- Automatic implementation of connection points.

Code Generation for Connection Points

The templates for Rhapsody code generation are held in properties. This allows you to customize the code generation of connection points for a particular class, for all classes in the model, or any intermediate level, such as all classes in a configuration or package. The following sections describe the properties for generating connection point code for a server with outgoing interfaces and a client that implements the connection point server.

Server with Outgoing Interfaces

The generated IDL code for a server with outgoing interfaces looks similar to the following

```
coclass CellCall
{
    [default] interface ICellCall;
    [default, source] interface _IRoamRecorder;
};
```

This sample code is for the `CellCall` coclass from the `PhoneCall` package in the phone call example.

The Implementing ATL Object

If an implementing ATL object depends on one or more connection point interfaces, it must do all of the following:

- Implement `IConnectionPointContainer`.
- Implement the connection point proxy.
- Have a map of connection points.

These items are implemented as shown in the following sample code for the `CCellCall` object from the phone call example:

```
class ATL_NO_VTABLE CCellCall:
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CCellCall, &CLSID_ CellCall>,
    public IConnectionPointContainerImpl<CCellCall>,
    public IDispatchImpl<ICellCall, &IID_ ICellCall,
           &LIBID_ PhoneCallLib>,
    {
    ...
    BEGIN _CONNECTION_POINT_MAP(CX)
    CONNECTION_POINT_ENTRY(DIID_INotifyEvent)
    END_CONNECTION_POINT_MAP()
    ...
}
```

The following table lists the code generation properties that specify how this code is generated.

Property	Subject and Metaclass	Purpose
DeclarationModifier	ATL::Class	Specifies the ATL class declaration modifier
ATLRootClass	ATL::Macro	Specifies the root class of an implementing ATL class
ATLClassObject	ATL::Macro	Specifies the ATL class that implements a class object
ATLConnectionPointImpl	ATL::Macro	Specifies the ATL class that implements the IConnectionPointContainer interface
IDispatchImpl	ATL::Macro	Specifies the ATL class that implements the IDispatch interface
ConnectionPointProxy Class	ATL::Macro	Specifies the proxy class for the connection point
BeginConnectionPoint Map	ATL::Macro	Specifies the macro to start a connection point map for an ATL class
ConnectionPointMap Entry	ATL::Macro	Specifies the macro for the connection point map entry
EndConnectionPointMap	ATL::Macro	Specifies the macro to end a connection point map for an ATL class

Proxy Class

The proxy class is defined in a new file. For example, the following code would be generated for the proxy class in the phone call model in a specification file called `CcellCall_CP.h`:

```
template <class T>
class CProxy_RoamRecorder: public
    IConnectionPointImpl<T, &DIID_RoamRecorder,
    CComDynamicUnkArray>
{
    public:
};
```

The `ATL::Configuration::ATLProxyClass` property specifies how this code is generated. This property provides the following template for generating the ATL proxy class for a connection point:

```
"\
#ifndef CP$interface_H \
#define CP$interface_H \
\
\
$import \
\
template <class T>\
class CProxy$interface : public
    IConnectionPointImpl<T, &$IDinterface,
           CComDynamicUnkArray>\
    {\
           public:\
           \
                  $operations\
           \
    };\
#endif"
```

The `ATLProxyClass` property uses the following keywords:

- `$interface`—Replaced with the name of the connection point interface.
- `$IDinterface`—Replaced with the ID of the connection point interface.
- `$operations`—Replaced with declarations of any operations that are members of the interface.

Methods of the Outgoing Interface

For each method of the outgoing interface, a firing method is added to the `<interface>_CP.h` file. For example:

```
HRESULT Fire_RecordCallStartTime()
{
    CComVariant varResult;
    T* pT = static_cast<T*>(this);
    int nConnectionIndex;
    int nConnections = m_vec.GetSize();

    for (nConnectionIndex = 0;
           nConnectionIndex < nConnections;
           nConnectionIndex++)
    {
           pT->Lock();
           CComPtr<IUnknown> sp =
                  m_vec.GetAt(nConnectionIndex);
           pT->Unlock();

           IDispatch* pDispatch =
                  reinterpret_cast<IDispatch*>(sp.p);
           if (pDispatch != NULL)
           {
                  VariantClear(&varResult);
                  DISPARAMS disp = {NULL, NULL, 0, 0};
                  pDispatch->Invoke(0x1, IID_NULL,
                  LOCALE_USER_DEFAULT, DISPATCH_METHOD, &disp,
                  &varResult, NULL, NULL);
           }
    }
    return varResult.scode;
}
```

The `ATL::Configuration::ATLDispInterfaceCPFireOperation` property specifies how this code is generated. It provides the following template for generating the connection point firing operation for outgoing interfaces:

```
"\
$opRetType Fire_$opname($arguments)\
{\
    CComVariant varResult;\
    T* pT = static_cast<T*>(this);\
    int nConnectionIndex;\
    CComVariant* pvars = NULL ;\
    if($noOfArgs > 0)\
           pvars = new CComVariant[$noOfArgs];\
    \
    int nConnections = m_vec.GetSize();\
    \
    for (nConnectionIndex = 0; nConnectionIndex <
           nConnections; nConnectionIndex++)\
    {\
           pT->Lock();\
           CComPtr<IUnknown> sp =
                  m_vec.GetAt(nConnectionIndex);\
           pT->Unlock();\
           IDispatch* pDispatch =
                  reinterpret_cast<IDispatch*>(sp.p);\
           if (pDispatch != NULL)\
```

```
            {\
                  VariantClear(&varResult);\
                  \
                  /*initialize pvars[..]*/\
                  \
                  DISPPARAMS disp = {pvars, NULL, $noOfArgs, 0};\
                  pDispatch->Invoke($id, IID_NULL,
                  LOCALE_USER_DEFAULT, DISPATCH_METHOD, &disp,
                  $&varResult, NULL, NULL);\
            }\
      }\
      delete[] pvars;\
      return varResult.scode;\
   }"
```

The `ATLDispInterfaceCPFireOperation` property uses the following keywords:

- `$opRetType`—Replaced with the operation return type.
- `$opname`—Replaced with the name of the operation to which a `Fire` operation is being added.

Client of a Connection Point Server

A client of a connection point server is modeled as an ATL class that directly inherits from a «COM Interface» class. For example, the BillingRecord ATL class in the phone call example is the client of the _IRoamRecorder connection point server. Code for the BillingRecord would be generated as follows:

```
class BillingRecord :
    public IDispatchImpl<_RoamRecorder,
           &IID_RoamRecorder, &LIBID_PhneCallLib>,
    public CComObjectRoot
{
    public:
    BillingRecord() {}

    BEGIN_COM_MAP(BillingRecord)
           COM_INTERFACE_ENTRY(IDispatch)
           COM_INTERFACE_ENTRY(_RoamRecorder)
    END_COM_MAP()
};
```

The code generation properties (under ATL::Macro) that specify how this code is generated include the following:

- IDispatchImpl—Specifies the ATL class that implements the IDispatch interface
- BeginInterfaceMap—Specifies the start macro for a COM map of an ATL class
- InterfaceEntry—Specifies the ATL macro that defines the COM map interface entry
- EndInterfaceMap—Specifies the end macro for a COM map of an ATL class

Index

Symbols

A

B

C

D

E

M

O

P

R

S

T

U

W