Rational. Rhapsody

CORBA Development Guide

Rational Rhapsody
CORBA 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 IIBM ®Rational® Rhapsody® 7.5 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

CORBA Setup, Libraries, and Files

IBM ®Rational® Rhapsody® facilitates development of distributed applications using the Common Object Request Broker Architecture (CORBA). Rational Rhapsody CORBA helps you to develop your applications to run in a client-server environment, and it supplies tools to support the following tasks:

- Define, use, and manipulate CORBA modules
- Manage CORBA interfaces, exceptions, and types
- Use CORBA as an integral part of the implementation model
- Link CORBA-domain constructs to the C++ domain
- Specify an IDL-to-C++ mapping and manage the mapping details (IDL is the Interface Definition Language.)

CORBA Development Requirements

Before using the Rational Rhapsody CORBA features, the developer should be familiar with Rational Rhapsody in C++. To use the CORBA features, the following system components must be installed and setup on the developer's computer:

- An ORB (TAO)
- An IDL compiler (supplied with the ORB)
- CORBA header files
- ORB Libraries for CORBA

Note

CORBA is supported in Windows C++ only; therefore, you must have a supported Windows C++ compiler running on the development machine. When you install Rational Rhapsody, you must select the installed C++ compiler so that the CORBA features can be used.

Object Request Broker (ORB)

An Object Request Broker (ORB) running on the network acts as the “glue” for distributed applications running on one or more processors. The ORB provides these services:

- Connects requests from a client to the server component that is capable of responding to the request, regardless of where on the network the server is running.
- Frees the client from being forced to know the location of a server on the network in order to use one of its services.

The supported Rational Rhapsody CORBA ORB is TAO (from ACE). ACE is an open source framework that provides many components and patterns for developing high-performance, distributed real-time and embedded systems. The TAO ORB supports efficient abstractions for sockets, demultiplexing loops, threads, synchronization primitives.

Installing and Building Your TAO Libraries

Before creating a CORBA project, you must first create the ORB libraries. Since TAO is the supported ORB, you must install ACE and build the TAO libraries before using the Rational Rhapsody CORBA.

Accessing ACE and TAO Downloads

Rational Rhapsody 7.5 requires the ACE-5.5 or higher and TAO-1.5 or higher. These are packaged as the ACE and TAO zip file on the TAO Web site at **http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/TAO-INSTALL.html**.

Scroll down from the starting point to locate your system requirements and library building instructions.

ACE and TAO Setup Process

You install Rational Rhapsody after TAO and the CORBA libraries are built and your C++ compiler is installed. The sequence of events is as follows:

1. Install a C++ compiler (such as Microsoft Visual C++.net Standard)
2. Install ACE
3. Install TAO and the CORBA header files
4. Build CORBA libraries
5. Install Rational Rhapsody and specify the C++ environment

During the Rational Rhapsody installation, the same C++ compiler you used to build the TAO CORBA libraries must be selected as the C++ compiler to be used with Rational Rhapsody.

Note

If you are using a different ORB, follow that vendor’s instructions to build the necessary libraries. You can also refer to the **Using Other ORBs** section in this guide for additional information.

TAO Properties

The Rational Rhapsody properties in the `CORBA::TAO` metaclass define how the TAO ORB interacts with Rational Rhapsody. These properties can be accessed for a CORBA project as follows:

1. Open the C++ project that you want to use to create a CORBA model.
2. Select **File** > **Project Properties**. This displays the Properties tab in the Features dialog box.
3. Select the **All** view and navigate to the `CORBA::TAO` group of properties.

The following sections describe each group of the TAO properties according to their functions. For additional information about CORBA properties, refer to the **Creating the Server Component** section.

Makefile Settings

The following TAO properties specify makefile-related information:

- `CORBAIncludePath` specifies the path to the additional include files for your CORBA ORB. The specified path is appended to the "include" path in the Rational Rhapsody-generated makefile. The default for this property is `$(ACE_ROOT)\TAO\ $(ACE_ROOT)\ $(ACE_ROOT)\TAO\orbsvcs`.
- `CORBALibs` specifies the path to the TAO libraries for your CORBA ORB. The specified path is appended to the object/library search path in the Rational Rhapsody-generated makefile. The three default library names, listed in this property's selection area, are `TAO_PortableServd.ib`, `TAO_Valuetyped.lib`, and `aced.lib`. For more information about TAO libraries, refer to the **Installing and Building Your TAO Libraries** section.
- `CPP_CompileSwitches` provides a string that allows you to specify additional C++ compiler switches.
- `CPP_LinkSwitches` provides a default empty string that allows you to specify additional link switches.
- `CPP_StandardInclude` provides a string that allows you to specify additional header files to be included in the generated sources, that are required when your component is compiled with the TAO include files. The default value is `tao/CORBA.h;tao/PortableServer/POA.h`.

Note

This information is only used in the makefile if it is needed.

Implementing a Class with the ORB

Rational Rhapsody gives you control over how a C++ class is bound to the ORB. Using the `CORBA::Class::DefaultImplementationMethod` property, you can specify whether a given C++ class should implement a CORBA interface using the inheritance approach or TIE approach. Once you have set this property, Rational Rhapsody uses the following properties (under `CORBA::TAO`) to implement the interface:

- `DefTIEString` - If `DefaultImplementationMethod` is set to `TIE`, the `DefTIEString` property specifies a template for the string generated into every IDL file that contains a CORBA interface.
- `Skeleton` - If `DefaultImplementationMethod` is set to `Inheritance`, the `Skeleton` property defines a format string that the implementing class inherits. The default is `POA_$interface`. The CORBA interface name replaces "`$interface`" in the generated code.

IDL Compiler Settings

The following properties (under `CORBA::TAO`) contain the settings for the IDL compiler:

- `IDLCompileCommand` specifies the compile command for your IDL compiler.
- `IDLCompileSwitches` specifies your IDL compiler's compilation switches. There are two ways to attach a CORBA implementation class to the ORB: BOA or TIE. The TAO "-B" flag compiles the IDL so that BOA objects are used. This is also the default for the property.

Mainline Code

The `ServerMainLineTemplate` property (under `CORBA::TAO`) specifies a code segment that is generated in the executable main file if the Rational Rhapsody configuration is defined as a CORBA server. You define a configuration to be a server by setting the `CORBAEnable` property (under `CORBA::Configuration`) to `CORBAServer`. This code segment should perform any initialization and additional setup steps before diving into the CORBA loop.

If you modify the `ServerMainLineTemplate` property, remember that every double-quote character must preceded with a backslash (`\"`).

Similarly, the `ClientMainLineTemplate` property (under `CORBA::TAO`) enables you to add code to the main function of a CORBA client.

Environment Parameter

If the `AddCORBAEnvParam` property (under `CORBA::TAO`) is set to `Checked`, an “environment” parameter (of the type `CORBA_env`) is added as the last argument to CORBA operations. The default value is `Cleared`.

Initialization Properties

The following properties (under `CORBA::TAO`) enable you to control the initialization of the ORB:

- `InitialInstance` specifies any additional initial instance routines required by the ORB. This code template will be generated for each instance of a specific class (implementing one or more CORBA interfaces).
- `InitializeORB` specifies the ORB initialization routines. In most cases, this is the first executable command in the `main` function of the CORBA server.

Makefiles for Building CORBA Applications

The Rational Rhapsody dual-phase code generation process produces IDL files for items tagged with CORBA stereotypes and C++ files for the remaining items. CORBA setup code is generated in the second phase of code generation. The `make` process links the IDL and C++ files with the CORBA skeleton.

The `make` first calls the IDL compiler to translate the IDL code to C++. Next, it calls the C++ compiler to compile the C++ output of the IDL compiler, along with the Rational Rhapsody native C++ output, into an assembly language image of the component.

Note

Rational Rhapsody can animate only one executable at a time. Therefore, animation can be enabled for either the client or the server component, but not both.

IDL Compiler-Generated Files

This section describes the files generated by the Interface Definition Language (IDL) compiler, supplied with the ORB.

File Naming Conventions

The CORBA specification does not force ORB vendors to use a unified naming convention for IDL compiler products. Moreover, it does not define what these products should contain. Therefore, a set of properties was created in Rational Rhapsody to address this issue.

All IDL compilers generate specification files (with function headers and signatures only) and implementation files (with function definitions and bodies). For a given CORBA interface, the IDL compiler can conceivably create the following code:

- **Skeleton code -** Server-side code. Can be built of two files, one for specification and one for implementation.
- **Stub code -** Client-side code. Can be built of two files, one for specification and one for implementation.

This means that the IDL compiler can potentially create four files, whose names are derived from the IDL file name.

For example, compiling an IDL file named `X.idl` with TAO leads to the following three files:

- `X.hh` (specification file)
- `XS.cpp` (skeleton implementation file)
- `XC.cpp` (stub implementation file)

Compiling the same `X.idl` file with another IDL compiler, for example `Visibroker`, leads to the following four files:

- `X_s.hh` (skeleton specification file)
- `X_c.hh` (stub specification file)
- `X_s.c` (skeleton implementation file)
- `X_c.c` (stub implementation file)

The following properties were created in Rational Rhapsody to define the IDL compiler file-naming behavior:

- `ImplementationExtension` specifies the extension of implementation files. The default is `.cpp`.
- `SkeletonImplementationName` is a string that defines the naming behavior for skeleton implementation files. The default is `$interfaceS`.

- `SkeletonSpecificationName` is a string that defines the naming behavior for skeleton specification files. The default is `$interfaceS`.
- `SpecificationExtension` is a string that specifies the extension for specification files. The default is `.h`.
- `StubImplementationName` is a string that defines the naming behavior for stub implementation files. The default is `$interfaceC`.
- `StubSpecificationName` is a string that defines the naming behavior for stub specification files. The default is `$interfaceC`.

File Usage

When you want to create a server, you need the skeleton code generated by the IDL compiler; when you want to create a client, you need the stub code. However, with different ORBs and IDL compilers, the skeleton and stub code is mapped to different files.

For example, server developers using TAO need only to compile and link with the generated skeleton file (for example, `XS.cpp` and `X.hh`). However, server developers using `Visibroker` need to compile and link with both the skeleton file and the stub file (for example, `X_s.hh`, `X_s.c`, `X_c.hh`, and `X_c.c`).

The following properties address this issue by specifying which files should be used to create a client, server, or process that is both a client and a server:

- `NeededObjForClient` is an enumerated type that specifies the file needed to create an object. The default is `Stub`.
- `NeededObjForServer` is an enumerated type that specifies the file needed to create a server. The possible values are as follows:
  - `Stub`
  - `Skeleton`
  - `Both`
- `NeededObjForClientServer` is an enumerated type that specifies the file needed to create a client server. The possible values are as follows:
  - `Stub`
  - `Skeleton`
  - `Both`

Stereotypes and Types

This section describes the CORBA stereotypes and types supported by Rational Rhapsody and provides the information necessary to understand the development steps in the **Creating a CORBA Model** section.

CORBA Stereotypes

Rational Rhapsody provides three stereotypes to indicate model elements that adhere to the CORBA standard:

- «`CORBAModule`»
- «`CORBAInterface`»
- «`CORBAException`»

«CORBAModule» Stereotype

The «`CORBAModule`» stereotype is applied to packages. It indicates that a package contains only CORBA-stereotyped model elements.

CORBA modules can contain:

- Other «`CORBAModule`» stereotyped packages
- «`CORBAInterface`» stereotyped classes
- «`CORBAException`» stereotyped classes
- CORBA types

Rational Rhapsody does not generate C++ code for «`CORBAModule`» stereotyped packages - it generates them into CORBA modules.

The «`CORBAModule`» stereotype is optional for packages unless the package contains CORBA types, which can be defined only in CORBA modules. If you want to define CORBA types, you can do so in a CORBA module.

If the `CPP_CG::Package::DefineNameSpace` property for the package is set to `True`, Rational Rhapsody generates the CORBA interfaces in the package - all encapsulated within the scope of a CORBA module. The scope name is the same as the package name.

«CORBAInterface» Stereotype

The «`CORBAInterface`» stereotype is applied to classes. It indicates that a class should be mapped to an IDL interface during code generation. Rational Rhapsody generates only IDL code for CORBA interfaces; it does not generate C++ code for them.

A class that inherits from a «`CORBAInterface`» stereotyped class exposes the interface. The CORBA interface itself exposes nothing.

Attributes and Operations of «CORBAInterface»

A «`CORBAInterface`» class can have both attributes and operations. These are generated into attributes and operations with the same names in the IDL interface. Data types used for attributes and operations are generated "as-is" in the IDL files. Therefore, you must use CORBA data types, defined in either the predefined CORBA types package or in your own «`CORBAModule`» or «`CORBAInterface`».

Create subclasses in the model to realize IDL interfaces as follows:

- Every CORBA operation must have a corresponding C++ operation in the realizing class.
- Every CORBA attribute must have a corresponding C++ attribute in the realizing class. You must provide `get` and `set` operations in the realizing class.
- Every CORBA type (for example, `long`) must have a corresponding type (for example, `CORBA::long`) in the realizing class. You can import these types from the CORBA predefined types package.

To simplify the process, you can drag-and-drop CORBA operations and attributes from a CORBA interface to a regular class. Rational Rhapsody automatically converts the types.

The following constraints apply to CORBA interfaces concerning code generation:

- Both attributes and operations of CORBA interfaces cannot be classified as public, private, or protected. Therefore, generated IDL files refer only to public attributes and operations. Protected and private attributes are ignored.
- «`CORBAInterface`» stereotyped classes cannot be instantiated. Therefore, operation bodies, if they exist, are ignored.
- The `virtual`, `static`, and `const` keywords have no meaning for «`CORBAInterface`» classes. Therefore, the `virtual/static` keyword is ignored during IDL attribute generation.

You can make an attribute of a CORBA interface `readonly` (a CORBA keyword) by setting the attribute's `CORBA::Attribute:IsReadOnly` property to `True`. To make an operation of a CORBA interface `oneway`, set the operation's `CORBA::Operation::IsOneWay` property to `True`.

In addition, the following standard UML options are available for operations:

- Operation arguments can have a direction of in, out, or inout. Specify these values in the Argument dialog box.
- The `CORBA::Operation::ThrowExceptions` property enables you to specify the exceptions that an operation throws. For example, if an operation throws the exceptions `exc1` and `exc2`, set `"exc1, exc2"` for the `ThrowExceptions` property.

Relations with «CORBAInterface» Classes

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

Associations and Aggregations

An outgoing or symmetric relation arrow leaving a «`CORBAInterface`» class can target only another «`CORBAInterface`» class. An incoming relation arrow coming into a «`CORBAInterface`» class can originate in either a regular class or another «`CORBAInterface`» class.

Outgoing or symmetric relations from «`CORBAInterface`» classes are mapped to accessor and mutator methods (such as `get()`, `set()`, `add()`, and `clear()`) in the generated IDL as follows:

- If the multiplicity of the target role is one, the accessor's return type and the type of the mutator's parameter are the same as the type of the target «`CORBAInterface`». In addition, the mutator's parameter has a direction of `in`.

  For example, the following IDL is generated for the interface `A`, which has a directed relation to an interface `B` with a multiplicity of one:

  ```
    interface A {
    ////   User-implicit entries  ////
        B getItsB();
        void setItsB(in B p_B);
  };
  ```

- If the multiplicity of the target role is greater than one, a type definition for an IDL sequence is generated for the source interface.

  For example, the following IDL sequence definition is generated for interface `C`, which has a symmetric relation to interface `D` with a multiplicity of two:

  ```
  typedef sequence<C> CSeq;
  ```

The `CORBA::Class::IDLSequence` property enables you to specify the implementation of the IDL sequence name, as follows:

- The default value, `$interfaceSeq`, expands to the name of the interface with the “Seq” suffix. For example, for an interface `C`, the generated sequence name is `CSeq`.
- You can turn off generation of the type definition by setting the property to an empty string.

Generalizations

A «`CORBAInterface`» can inherit only from another «`CORBAInterface`».

Inheritance between two «`CORBAInterface`» classes is generated into an inheritance between the corresponding IDL interfaces. For example, if a «`CORBAInterface`» `H` inherits from a «`CORBAInterface`» `G`, the following IDL code is generated for `H`:

```
interface H : G {};
```

An inheritance arrow between a regular class and a «`CORBAInterface`» is interpreted as a realization (implementation) of the interface. This is the typical architecture used to implement a CORBA server.

Note

Generally, for configurations with such a construct, the `CORBA::Configuration::CORBAEnable` property must be set to `CORBAServer` to avoid code generation errors.

There are two ways to realize object adapters in CORBA:

- `Inheritance`
- `TIE`

The `CORBA::Class::DefaultImplementationMethod` specifies the implementation method (`Inheritance` or `TIE`) for the project. In other words, if `DefaultImplementationMethod` is set to `Inheritance`, all realizations of CORBA interfaces are implemented using inheritance by default.

For example, when using TAO and the inheritance implementation method, the following code is generated for class `J`, which inherits from «`CORBAInterface`» `I`:

```
class J : virtual public IBOAImpl {
public :
    // Constructors and destructors
    J(const char* instanceName = "");
    -J();
};
```

In the generated code, the realizing class `J` inherits from the `IBOAImpl` class, which is generated by the TAO IDL compiler.

You can override the default implementation method by setting the `CORBA::Class::TIERealizes` or `InheritanceRealizes` property for a specific class to the name of the «`CORBAInterface`» classes that it realizes. In other words, even if you are using inheritance as the default implementation method for the project, you can still use TIE as the implementation method for a particular class by setting its `TIERealizes` property to the name of the «`CORBAInterface`» that it realizes. You can have the same class realize different «`CORBAInterface`» classes using different methods by setting the `TIERealizes` and `InheritanceRealizes` properties for the same class to the names of the «`CORBAInterface`» class that it should realize using either method.

Compositions

«`CORBAInterface`» classes cannot be contained in any element. «`CORBAInterface`» classes themselves contain only «`CORBAException`» classes.

«CORBAException» Stereotype

The «`CORBAException`» stereotype is applied to classes. It indicates that the class should be mapped during code generation to an IDL exception. CORBA IDL exceptions can be defined within the scope of a «`CORBAInterface`» or a CORBA module. CORBA exceptions cannot have a global scope.

CORBA IDL exceptions can have attributes, but not operations. Any operations found in a «`CORBAException`» class are ignored during code generation.

CORBA IDL exceptions cannot inherit from other CORBA IDL exceptions.

Mapping «CORBAExceptions» to Code

A «`CORBAException`» stereotyped class is not generated in its own file. Instead, it is generated into the file of the encapsulating entity - the class or package in which it is defined. Therefore, a «`CORBAException`» defined in a class can be “thrown” by any of its operations. An exception defined in a CORBA module can be thrown by any operation in any class within that module.

To make your design clear, you should draw a «`Usage`» arrow from a class to an exception that it throws.

CORBA Types

Rational Rhapsody includes a package of *predefined CORBA types*. This package contains the basic CORBA IDL types, which you can assign to any attribute, operation return type, or argument or an operation that belongs to a «`CORBAInterface`».

The predefined types are as follows:

any	boolean	char
double	fixed	float
long	longdouble	longlong
octet	short	string
unsignedlong	unsignedlonglong	unsignedshort
wchar	wstring	

CORBA Predefined Reference Package

To import the CORBA predefined types package into your model, do the following:

1. Select **File** > **Add to Model**.
2. Navigate to `<Rational Rhapsody installation path>\Share\Properties`.
3. In the **Add To Model** dialog box, change the file type filter to **Package** (*.sbs).
4. Select the `CORBA.sbs` package, the **As Reference** radio button, and click **Open**.

The CORBA package is added as a read-only (RO) reference package to the model. Note that because the CORBA types package is added as a reference package, no code is generated for it.

Once the CORBA package is imported, the CORBA IDL data types are displayed in the **Type** drop-down list (for example, when you select **Type is Typedef’ed** in the Operation dialog box).

Defining Native CORBA Types

You can create CORBA structures, enumerations, and typedefs using the same steps you would for creating such items in C++.

To define these CORBA types:

1. Create a new type inside a `<<CORBA Module>>` package or `<<CORBA Interface>>` class.
2. On the General tab of the Features dialog box for the type, select **Language** from the **Kind** drop-down list.
3. On the Declaration tab of the Features dialog box, declare the IDL.
4. Click **OK**.

Note

To define a CORBA union, use the method described in the section, **Using Stereotypes to Define a CORBA Union**

Using Stereotypes to Define a CORBA Union

To define a CORBA union, you use the stereotypes:

- `<<CORBAFixedUnion>>`
- `<<CORBAVariableUnion>>`

To define a CORBA union, follow these steps:

1. Create a new type inside a `<<CORBA Module>>` package or `<<CORBA Interface>>` class.
2. On the General tab of the Features dialog box for the type, select **Language** from the **Kind** drop-down list.
3. On the Declaration tab of the Features dialog box, enter the code for the union.
4. Click **OK**.
5. Set the property `CORBA::Type::CORBAStereotype` to `CORBAFixedUnion` or `CORBAVariableUnion`.

CORBA Types and Code Generation

CORBA types are ignored during C++ code generation; they are relevant only for CORBA IDL generation. When generating IDL, Rational Rhapsody maps any CORBA types to the corresponding C++ types based on the UML mapping scheme. Most IDL compilers use the same scheme.

You can override the default C++ mapping scheme as follows:

- To change the mapping scheme, modify the `in`, `inout`, `out`, and `ReturnValue` properties under `CORBA::C++Mapping_CORBA<implementation><type>`.
  For events and triggered operations, modify the property `TriggerArgument`.
- <implementation> is either `Fixed` or `Variable` (according to the value of the property `CORBA::Type::C++Implementation`).
  <type> is one of `Structure`, `Union`, `Enumeration`, `Array`, `Sequence`, or `Basic`.

Mapping CORBA Types to Code

Because the mapping of CORBA data types to C++ code is determined by both the type and usage (for example, whether the item is assigned to its `in`, `inout`, `out`, or `return`), each type is mapped to a certain C++ construct according to its usage either during code generation or when you copy it from the CORBA domain to the C++ domain (such as when you drag an attribute or operation from a CORBA interface to a regular class).

There are two properties that affect mapping of CORBA types to C++ code, in the case of types and interfaces:

- `CORBA::Type::C++Implementation` - possible values are `Fixed` and `Variable`.
- `CORBA::Class::C++Implementation` - possible values are `Reference` and `Variable`.

The table below indicates the mapping settings used for the different property values.

	Property Value	**C++ Implementation**	**Metaclass Used when Declaring Argument in Operation Signature**
Structure	Fixed	<Structure name>	`CORBA::C++Mapping_CORBA FixedStruct`
	Variable	<Structure name>_var	`CORBA::C++Mapping_CORBA VariableStruct`
Array Typedef	Fixed	<Typedef name>	`CORBA::C++Mapping_CORBA FixedArray`
	Variable	<Typedef name>_var	`CORBA::C++Mapping_CORBA VariableArray`
Sequence Typedef	Fixed	<Typedef name>_var	`CORBA::C++Mapping_CORBA Sequence`
	Variable	<Typedef name>_var	`CORBA::C++Mapping_CORBA Sequence`
Language Type	Fixed	<Type name>	`User can choose; default is CORBA::C++Mapping_CORBABas ic`
	Variable		
Simple Typedef	Simple Typedef recurses to the last real type that the typedef redefines, and uses the value of that type's `CORBA::Type::C++Implementation` if it is a type, or the value of `CORBA::Class:C++Implementation` if it is an interface, and the type's corresponding `CORBA::C++Mapping_CORBA`<implementation><type> metaclass.		

Creating a CORBA Model

This section illustrates some of the basic Rational Rhapsody operations required to create a CORBA model.

Building CORBA Applications

The following steps define the general process for creating CORBA components using Rational Rhapsody:

1. Create a C++ project in Rational Rhapsody. (See the **Creating a New CORBA Project** for more details.)
2. Define CORBA interfaces by assigning CORBA stereotypes to model elements. (See the **Creating the Required CORBA Stereotype** and **Creating the CORBA IObserver Interface** section for more details.)
3. Create client and server relations between interfaces and classes in the structural model. (See the **Creating the System's Class and Inheritance** sections for more details.)
4. Set up CORBA properties for the appropriate model elements. (See the **Creating the Server Component** for more details.)
5. Generate code. Rational Rhapsody automatically generates Interface Definition Language (IDL) code for the items tagged as CORBA stereotypes and C++ code for the remaining items
6. Build client and server components. Rational Rhapsody generates the IDL code needed for the client and server components. This IDL code is then compiled by the IDL compiler to create the necessary CORBA stubs and skeletons.
7. Start the server and client by running the executables.

You determine what classes you want to have communicate with each other and specify that they have the <<CORBAInterface>> stereotype.

Rational Rhapsody Sample Models

The instructions used in this example are based on the CORBA project in the C++ samples in your Rational Rhapsody installation. The CORBA samples directory contains three projects for the SDM (Security Door Management) model. You might find it useful to review these projects before following the practice instructions in this section.

To examine the projects, follow these steps:

1. Open Rational Rhapsody in C++: From the Windows Start menu, select **Programs > IBM Rational > IBM Rational Rhapsody *version#* > Rhapsody Development Edition > Rhapsody in C++**.
2. Select **File > Open** to open the Open dialog box.
3. Navigate to **Samples > CPPSamples > CORBA**.

SDM_Observers Sample Model

In the Rational Rhapsody C++ samples, the SDM_Observers model contains three different components:

- SDM (Security Door Management)
- policeObserver
- alarmObserver

Each component performs its own task. A client is developed in the Client_sdm_observers model.

This system is designed to detect unauthorized entry (“break-in”) to protected buildings and to notify the proper authorities to respond to a break-in. The general system requirements are as follows:

- The Security Door Manager (SDM) software monitors a Door in order to detect a door access violation (break-in).
- If there is a break-in, the software signifies a break-in and a list of observers is notified. These observers are registered in the SDM.
- Observers, such as police at the police station and security officers located remotely, are notified and the alarm is activated.
- The police observers and alarm observers are registered with the Security Door Manager, and they request to be notified if a break-in (event) occurs.
- When the break-in event occurs, the Act operation starts and the notification is sent to the observers.

- The alarm is activated and the police are called.

Creating a New CORBA Project

When you create practice files, save your files in a different directory from the sample directory to allow you to compare your practice work with the original Rational Rhapsody samples.

Create a new CORBA project with these steps:

1. Open Rational Rhapsody in C++: From the Windows Start menu, select **Programs > IBM Rational > IBM Rational Rhapsody *version#* > Rhapsody Development Edition > Rhapsody in C++**. You must create this project using the C++ development edition.
2. Select **File** > **New** from the main menu bar to display the New Project dialog box.
3. Type `Sdm_Observers` as the **Project name** and select a folder to use as your practice project directory.
4. Use the “Default” **Type** since it contains all of the items needed for a CORBA project.
5. Click **OK**. The system opens a new project and creates an Object Model Diagram named Model1 in the Drawing area.

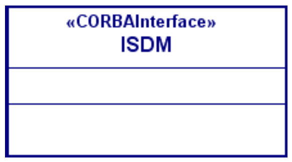
   Note: In the browser, select **Object Model Diagrams** > **Model1**, and right-click `Model1`.
6. Select **Diagram Properties** from the menu, and in the **General** tab, type `Main_Model` in the **Name** field.
7. Click **OK** to save the name change and close the dialog.

Creating the Required CORBA Stereotype

In order to create a CORBA component, you must then create a class and define it as a CORBA interface stereotype. To create the interface class and make it a CORBA stereotype, follow these steps:

1. In the object model `Main_Model`, select the **Class** icon ▤.
2. Drag the pointer in the diagram drawing area to create a class.
3. Type the name of the class, `ISDM`, over the default name that the system supplied. This is the interface for SDM.
4. Right-click the `ISDM` class diagram and select **Features** from the menu.
5. Select the **General** tab and in the **Stereotype** field, select `CORBAInterface` from the pull-down menu.
6. Click **OK** to save the changes and close the dialog.

The `ISDM` class is now a CORBA interface. During code generation Rational Rhapsody generates IDL code for this class. Your diagram should resemble this example.

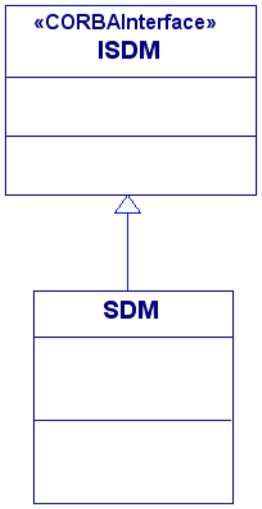


Creating the System's Class and Inheritance

In order to represent the system and create and inheritance between the system and its interface, you must now create the SDM class with these steps:

1. In the object model diagram, draw another class named `SDM` below the `ISDM` class.

2. Select the **Inheritance** icon and click on the top of the SDM class and then on the bottom of the ISDM class to create the inheritance relationship between the two classes. This means `SDM` implements the CORBA interface `ISDM`. At this point, the diagram should resemble this example.



3. In the browser, the open the **Packages** and right-click Default. Select **Add New** > **Event**.

4. Type `evNotify` into the open area in the browser to create the Notify event.

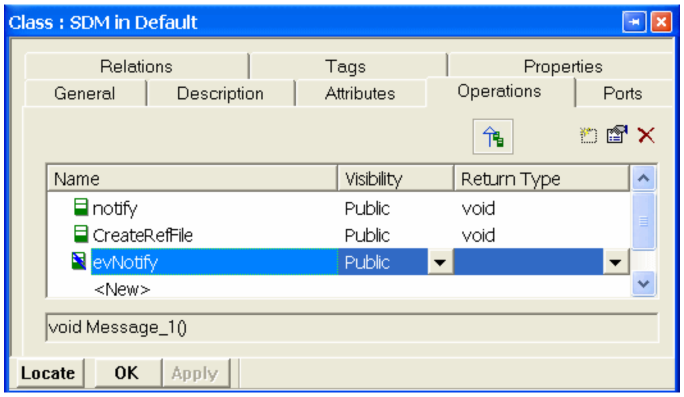
5. In the browser, the open the **Packages** > **Default** > **Classes** > **SDM**.

6. Right-click and select the **Features** option to display the Features dialog box.

7. Select the **Operations** tab and from the <New> pull-down menu select the specified type, listed below. Then enter the text to name the new operations and assign all three Public visibility:

   - PrimitiveOperation - `notify`
   - PrimitiveOperation - `CreateRefFile`
   - Reception - `evNotify`

At this point the Operations in the Features dialog box should resemble this example.

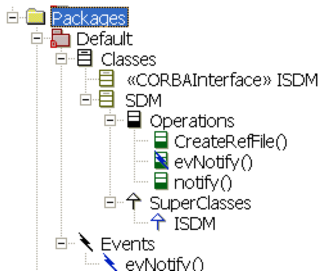


Class : SDM in Default

Name	Visibility	Return Type
notify	Public	void
CreateRefFile	Public	void
evNotify	Public	
<New>		

void Message_1()

8. Click **OK** to save the operations and close the dialog box. At this point, the browser should resemble this example.



```
Packages
  Default
    Classes
      «CORBAInterface» ISDM
      SDM
        Operations
          CreateRefFile()
          evNotify()
          notify()
        SuperClasses
          ISDM
    Events
      evNotify()
```

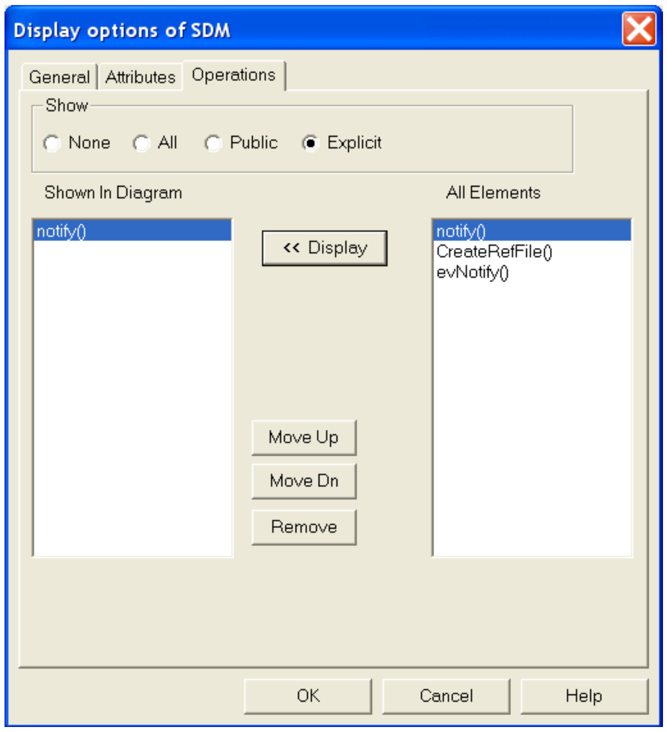
Implementing the SDM Operations

At this point, the basic structure of the SDM system is drawn in the Object Model Diagram. Now the operations in the system need to be implemented.

When the evNotify event is generated, the SDM needs to receive the event and call the Notify operation, created in the previous section. To accomplish this, add the implementation code to the Notify operation with these steps:

1. Right-click `Notify` in the browser to display a menu.
2. Select the **Features** option and then the **Implementation** tab in the Features dialog box. Note that the item selected in the browser forms the title bar of the dialog box.
3. Type the code in the Implementation area in this example:
4. Click **OK** to save the code.
5. To display the Notify operation in the class, right-click the `SDM` class in the diagram and select **Display Options**.
6. Select the **Operations** tab.

7. In the **All Elements** list, highlight the notify() item and click **Display** to put it into the **Shown in Diagram** column.


Display options of SDM

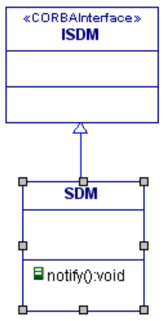
General | Attributes | Operations

Show: ( ) None ( ) All ( ) Public (● Explicit)

Shown In Diagram: notify()

All Elements: notify(), CreateRefFile(), evNotify()

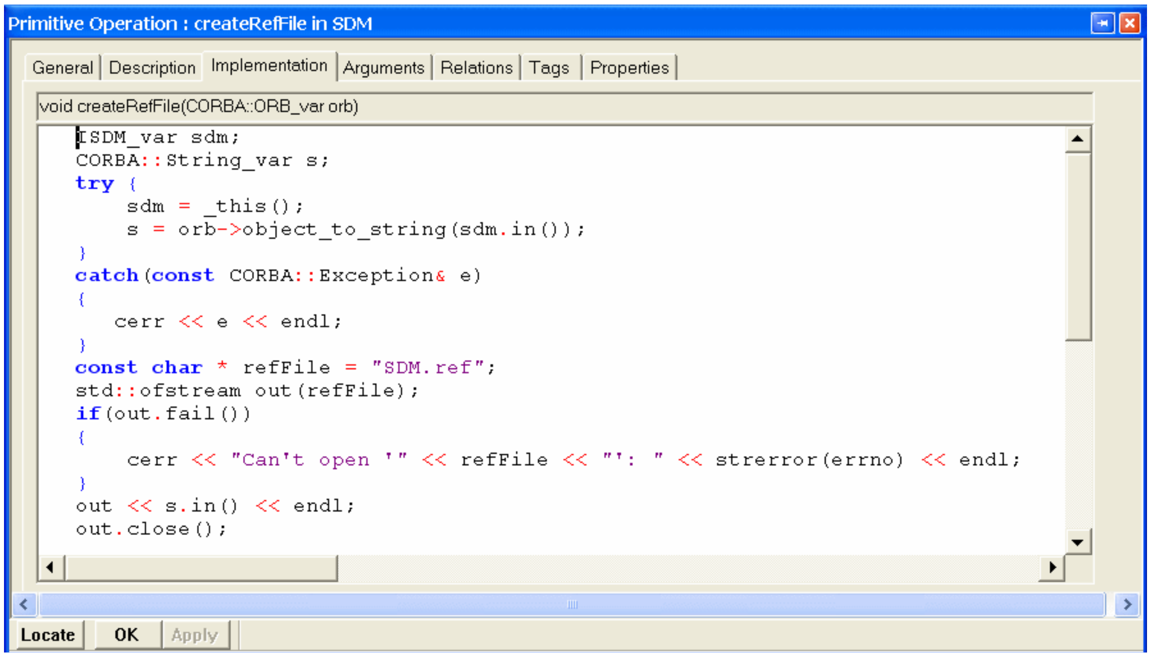
8. Click OK to save this display change. At this point the diagram should resemble this example.



Next you need to add the implementation code for `CreateRefFile` with these steps:

1. Right-click `CreateRefFile` in the browser.
2. Select the **Features** option and then the **Implementation** tab in the Features dialog box.

3. Type the code in the **Implementation** area. To speed up this step, you might want to copy this code from the SDM_Observers project in the Samples directory.



void createRefFile(CORBA::ORB_var orb)

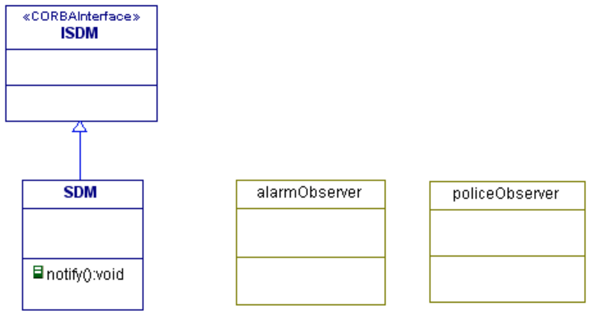
```
ISDM_var sdm;
CORBA::String_var s;
try {
    sdm = _this();
    s = orb->object_to_string(sdm.in());
}
catch(const CORBA::Exception& e)
{
   cerr << e << endl;
}
const char * refFile = "SDM.ref";
std::ofstream out(refFile);
if(out.fail())
{
    cerr << "Can't open '" << refFile << "': " << strerror(errno) << endl;
}
out << s.in() << endl;
out.close();
```

4. Click **OK** to save the code.

Defining the Notify Operation

Now the Notify operation needs a list of observers to notify. Make the following changes in the diagram:

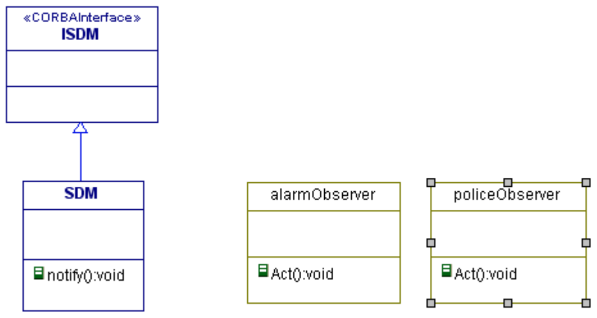
1. To the right of the SDM class, draw another class and change the generated name to be `alarmObserver`.
2. To the right of the new alarmObserver class, draw a `policeObserver` class. At this point your diagram should resemble this example.



The Notify operation also needs to call an `Act` operation for the observers. Follow these steps to add it:

1. In the browser, right-click the `alarmObserver` to display the menu and select Features.
2. Select the **Operations** tab to add the new operation.
3. Select **PrimitiveOperation** and name it `Act`. Assign it “Public” **Visibility** and a “Void” **Return Type**.
4. Click **OK** to save.
5. Repeat steps 1 to 4 to create the `policeObserver` class.

6. Then make the Act operation display in the diagram. At this point, your diagram should resemble this example.



The `Act` operation needs to be implemented with “activating alarm” or “calling the police.” Follow these steps to add the necessary implementations:

1. In the browser, right-click `Act()` under the `alarmObserver`.
2. Select the **Features** option and then the **Implementation** tab.
3. Type the following code in the Implementation area:

```
below.omcout << "Activating alarm..." << omendl;
```

4. Click OK to save.
5. In the browser, right-click `Act()` under the `policeObserver`.
6. Select the **Features** option and then the **Implementation** tab.
7. Type the following code in the Implementation area:

```
omcout << "Calling the police..." << omendl;
```

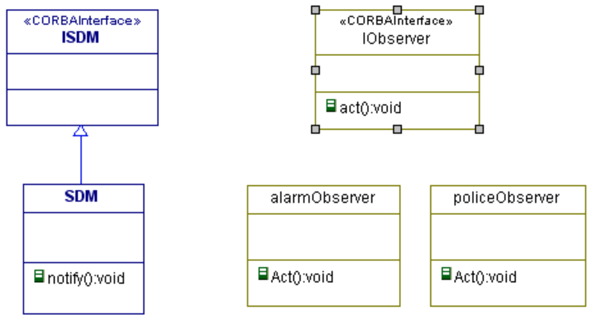
8. Click **OK** to save.

Creating the CORBA IObserver Interface

In order to make all of these elements work together, the `Act` operation is inherited by the observers from the CORBA `IObserver` interface. When the `Act` operation is called for Alarm Observers, the alarm is activated.

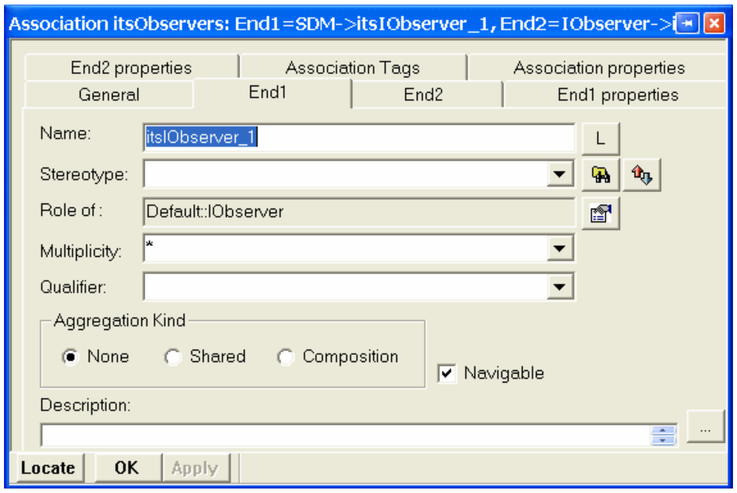
Follow these steps to incorporate these concepts into the model:

1. Above the two observer classes, draw another class and change the generated name to be `IObserver`.
2. Right-click to display the **Features** dialog box, on the **General** tab select `CORBAInterface` in the **Stereotype** field and `Sequential` in the **Concurrency** field.
3. Click **Apply** to save the changes and keep the dialog box open.
4. Select the **Operations** tab, and add the PrimitiveOperation `act` in the same manner as used previously. Click **OK**.
5. Right-click the new IObserver class and select the **Display Options** > **Operations**.
6. Select the `act` operation to be displayed and click OK. At this point the diagram should resemble this example.



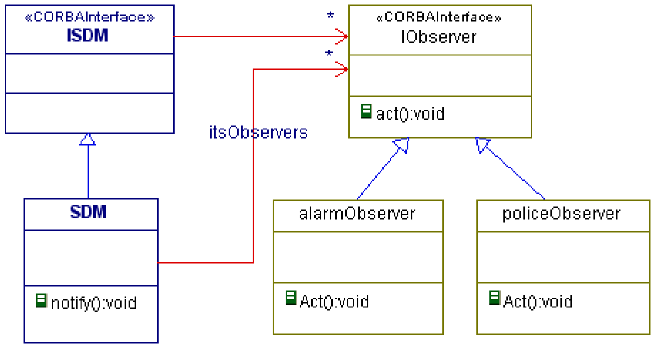
7. Select the **Inheritance** icon on the Drawing toolbar. Click on the `AlarmObserver` and draw an inheritance line to the `IObserver` and click to end the line.
8. Draw another inheritance line from `policeObserver` to the `IObserver`.
9. Select the **Directed Association** icon on the Drawing toolbar. Click the `SDM` class and then the `IObserver` class. Type the name `itsObservers` in the area highlighted on the line.

10. Right-click the directed association line and select "*" for the **Multiplicity** in the dialog box. Click **OK**.



Name: itsIObserver_1
Stereotype:
Role of : Default::IObserver
Multiplicity: *
Qualifier:
Aggregation Kind: [✓ None] Shared Composition
[✓ Navigable]
Description:

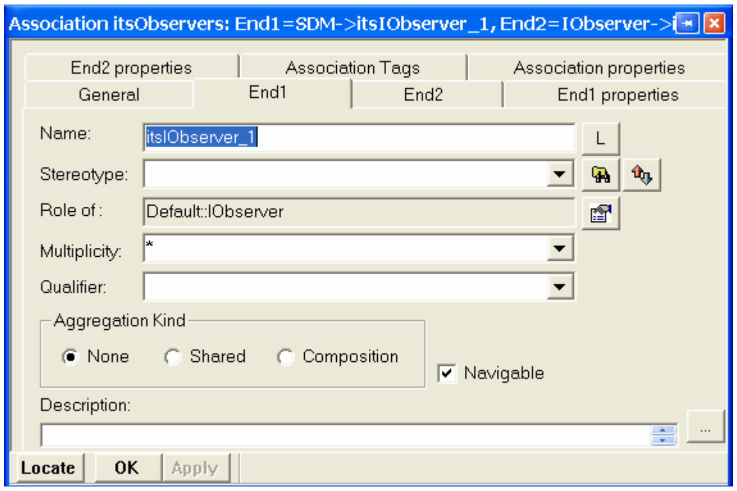
11. Draw another directed association from ISDM to IObserver without giving the line a name, but select "*" as the **Multiplicty**, as for the previous association. At this point your diagram should resemble this example.



Using the ISDM Interface to RegisterObservers

The ISDM interface is the CORBA interface used to register the observers with the Security Door Manager. IObserver is the CORBA interface the Security Door Manager uses to communicate with the observers. The class SDM has a relation to the IObserver holding its list of attached observers. The “attach” is implemented by calling the Rational Rhapsody-generated `addItsObservers` operation. Follow these steps to put these changes into the model:

1. Select the Directed Association icon on the Drawing toolbar. Click the `SDM` class and then the `IObserver` class. Type the name `itsObservers` in the area highlighted on the line.
2. Right-click the directed association line and select “*” for the **Multiplicity** in the dialog box. Click **OK**.



Name: itsIObserver_1

Stereotype:

Role of : Default::IObserver

Multiplicity: *

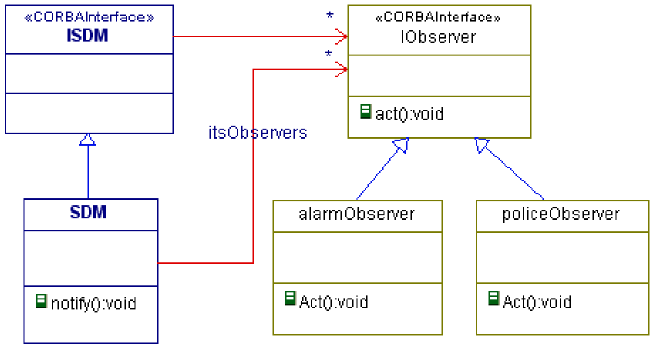
Qualifier:

Aggregation Kind: [● None] Shared Composition

[✓] Navigable

Description:

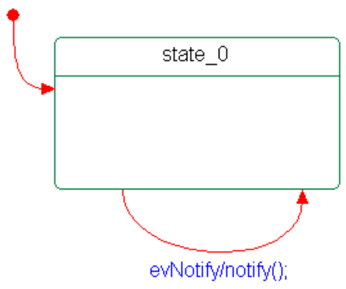
3. Draw another directed association from ISDM to IObserver without giving the line a name, but select “*” as the **Multiplicty**, as for the previous association. At this point your diagram should resemble this example.



Creating a Statechart for the evNotify Event

You must add a statechart to the SDM that shows the evNotify actions. When the evNotify event is generated (representing a criminal’s break-in to the protected building), the event is received by SDM, and the Notify operation is called. Follow these steps to add the statechart:

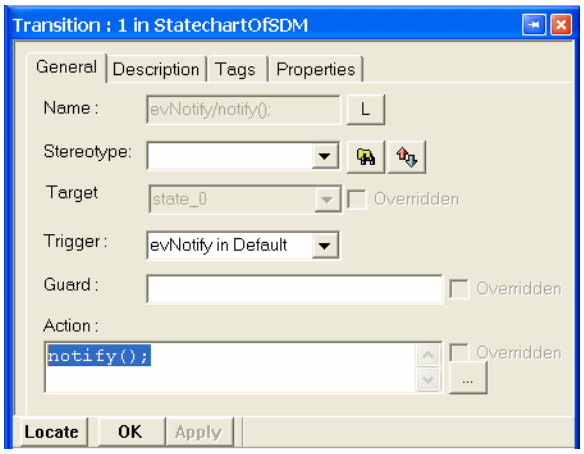
1. Right-click the SDM class in the diagram.
2. Select **New Statechart**.
3. Select the **State** icon from the Drawing toolbar and draw a state in the empty diagram area.
4. Select the **Default Connector** icon and draw the connector from outside the state to the edge of the state.



5. To complete the statechart, select the **Transition** icon and draw the line from one edge of the bottom of the state to the other edge.

6. Type `evNotify` as the name for this transition.

7. To complete the statechart, double-click the transition line to add the `notify();` **Action**.



Transition : 1 in StatechartOfSDM

Name : evNotify/notify();

Stereotype:

Target state_0

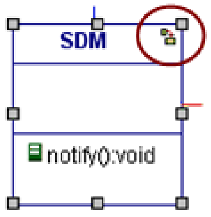
Trigger : evNotify in Default

Guard :

Action : notify();

8. Click **OK**.

9. Click the tab for the object model diagram and note that a statechart icon is now displayed in the SDM class. The icon is circled in this example.



Creating the Server Component

To create the server component for this simple CORBA model, follow these steps:

1. In the browser, right-click the **Components** item.
2. Select **Add New Component**.
3. Type `SDM_Server` as the component name.

Setting the CORBA Server Properties

1. In the browser, right-click the SDM_Server component and select **Features and the Properties** tab.
2. Select the **All** view.
3. Navigate to the `CORBA::Configuration::CORBAEnable` property and set it to `CORBAServer`.
4. Click **OK** to apply your changes and close the dialog box.

Defining the Server Initialization

1. In the browser, expand the `SDM_Server` > **Configurations**.
2. Double-click the `DefaultConfig` configuration for the `SDM_Server` component to display the Features dialog box.
3. Select the **Initialization** tab.
4. In the **Initialization code** field, type the code to instantiate SDM on startup.

   The `InstanceNameInConstructor` property (under `CORBA::Class`) specifies whether to generate a constructor that can accept an instance name for a class that implements a CORBA interface. If this property is `Checked` and the default implementation method is `Inheritance`, you can instantiate a realizing class with a specific name.

   As an alternative to writing initialization code manually, you can simply select SDM as an "initial instance." In this case, Rational Rhapsody automatically creates a single initial instance ofSDM with a string name indicator.
5. Make sure that animation is disabled for the `DefaultConfig` configuration (in the **Settings** tab, the **Instrumentation Mode** setting should be **None**).
6. Click **OK** to apply your changes and close the Features dialog box.

Building the Server Component

Now that the Server component is defined, generate code for it with these steps:

1. From the main menu at the top of the interface, select the **Code** > **Generate** > **DefaultConfig** options.
2. Build the `SDM_Server` component, select the **Code** > **Build Server.exe** menu options.

   If the `SDM_Server` component compiles without errors, you are now ready to build the `SDM_Client` component.

Building the Client Component

To generate code for the `SDM_Client` component, follow these steps:

1. In the browser, select the `SDM_Client` component from the Component list.
2. From the main menu at the top of the interface, select the **Code > Generate** > **DefaultConfig** options.
3. Build the `SDM_Client` component, select the **Code > Build Server.exe** menu options.

Troubleshooting the Build

If errors display in the Build window at the bottom of the interface, examine the messages and return to the section of the instructions relating to the feature referenced in the messages. Then check the following:

- Go back through the instructions to be certain that all of the steps were performed.
- Did you create the project using the C++ development edition of Rational Rhapsody?
- Do you have an IDL compiler installed?

Clients and Servers

CORBA interfaces play a significant role in the design of both clients and servers. *Servers* are CORBA-enabled executables that are able to respond to remote invocations. Servers link to CORBA skeletons.

Clients are components that use a server IDL, represented either by Rational Rhapsody model elements, or an external IDL file. Clients link to CORBA stubs. The IDL compiler can generate server (skeleton) code or client (stub) code from a CORBA interface.

Building the Client

There are several steps required to build the client:

- Add a client class to the Object Model Diagram
- Associate that client class to the CORBA Interface class

Running the Application through an ORB

Before you can execute the model, you must start the ORB daemon. With TAO, you must also register the new `Server` component in the implementation repository.

Do the following:

1. Run the ORB daemon (for example, `orbixd`). Once the daemon is running, the following message should be displayed in a new window:

   ```
   [orbixd: Server "IT_daemon" is now available to the network]
   ```

2. Open a command prompt window, change directory to where the server program (`server.exe`) is located, and register the server component with the ORB. In TAO, the command is as follows:

   ```
   > putit Server <path>\server.exe
   ```

   Note that the first argument to the `putit` command is the server logical name, which consists of the component name by default. The second argument is the location of the server executable, which must include the full path - even if it is the current directory.

   If the `putit` command is successful, a message similar to the following is displayed:

   ```
   [<connection#>: New Connection (<hostname>,
      IT_daemon, *, <username>, pid=<program ID>,
      optimized) ]
   ```

   To check which servers are registered, use the following TAO command:

   ```
   > lsit
   ```

3. In Rational Rhapsody, select **Code** > **Run Client.exe**.

4. In the Animation toolbar, click **Go** to start the program. The program creates an instance of `A` on startup.

5. Open the animated statechart for the `A` instance and generate an `evTry()` event using the Event Generator.

Interpreting CORBA Interfaces

Rational Rhapsody interprets the model and automatically decides whether to generate skeleton (server-side) or stub (client-side) code for a CORBA interface. However, you can override the default model interpretation using the following properties (under `CORBA::Configuration`):

- `ExposeCorbaInterfaces` - Generates server IDL code for the CORBA interface
- `UseCorbaInterfaces` - Generates client IDL code for the CORBA interface

To use these properties, assign a comma-separated list of `CORBA interfaces` from the scope to the appropriate property. Rational Rhapsody interprets the interface according to the request.

Servers

A class that realizes a «`CORBAInterface`» essentially needs the server-side code for the interface. Rational Rhapsody interprets an inheritance relationship as a request to generate, compile, and link with the server-side code of the CORBA interface.

A class that directly or indirectly inherits from a CORBA interface must implement *all* the operations in the parent CORBA interfaces. The bodies of the CORBA interface methods must be implemented in the realizing class. In addition, the realizing class must implement the attributes of its parent CORBA interfaces, and provide accessor and mutator operations (with the appropriate types) for each attribute.

You must manually implement all the attributes, operations, and associations of a CORBA interface in the realizing class. The best way to do this is to drag-and-drop attributes, operations, and relations from the CORBA interface into the realizing class. This ensures that CORBA types in the CORBA interface are translated to the corresponding C++ types in the realizing class. It also ensures that the appropriate accessors and mutators are generated for the attributes.

It is important to follow all the guidelines described in the following sections for server realization. If you do not adhere to these guidelines, the compiler might report errors at compile time, or CORBA exceptions might be thrown during run time. Rational Rhapsody performs some checks to detect possible violations before generating code.

Realizing Server Attributes

Each attribute defined in a CORBA interface is mapped to a CORBA attribute of the same name in the IDL file.The standard IDL generator generates accessor (`get`) and mutator (`set`) operations for all attributes. For example, for an attribute named `att`, the following accessor and mutator are generated in the IDL file:

```
att(); // accessor
att(value); // mutator
```

Classes that inherit from CORBA interfaces must implement the accessor and mutator operations for each attribute, except for `readonly` attributes, which do not require a mutator.

Although C++ allows method overloading, it does not allow a data member and a method to have the same name. Therefore, the data member in the realizing class must not have the same name as the attribute in the CORBA interface. You should copy the attributes from the CORBA interface to the realizing class.

Realizing Server Operations

Classes that inherit from CORBA interfaces must implement each operation of the CORBA interface. Each of the realizing operations must have the same name and the same number and order of arguments as the operation in the parent CORBA interface. The argument types in the realizing operation must be derived from the interface according to the IDL-to-C++ mapping scheme specified by the CORBA standard.

For example, if the following CORBA interface operation, the argument has type `long` :

```
op1(long arg1);
```

The type of the argument of the corresponding operation in the realizing class is `CORBA::long` is as follows:

```
op1(CORBA::long arg1);
```

The type conversion is done automatically when you copy (drag-and-drop) the operation from the CORBA interface to the realizing class.

The `EnvParamType` property (under `CORBA::TAO`) specifies whether to generate an additional `CORBA_env&` parameter for operations. This property is normally set at the component level, thus affecting all packages, classes, and operations within the component’s scope.

Realizing Server Relations

For every outgoing relation from a CORBA interface, you must provide the following methods in the realizing class:

- An accessor
- A mutator
- An `add()` method
- A `clear()` method

If the multiplicity of the target role is not one, you must also provide a CORBA sequence declaration (see **Realizing Server Associations**).

CORBA Type Translation

The types used in the realizing class must correspond to the CORBA types defined in a CORBA module and the `CORBAStereotype` property to create the correct argument list for generated operations.

It is important to note the following:

- User-defined CORBA IDL types are generated in the IDL file generated for the package or class in which the type is defined.
- Types defined in a «`CORBAModule`» stereotyped package are constrained to be only CORBA IDL types.

Realizing Server Associations

To realize a directed association from one CORBA interface to another CORBA interface, you must implement the accessor operations in the subclass that implements the source CORBA interface. Do this using one of the following methods:

- Directly define the accessors in the realizing class.
- Draw an association with the same target role from the realizing class to the target CORBA interface. Rational Rhapsody generates accessors with appropriate signatures.

In the latter case, if the relation has a multiplicity of one, the return and argument type of the accessor should be `IB_ptr`. If the multiplicity is many, the return type of the accessor should be `IB_seq`, and the realizing operation should return an instance of the `IB_seq` class that is generated by the IDL compiler and populated with the associated objects.

Clients

A class that has an association or a «Usage» relation to a CORBA interface needs the client-side code for the interface. Rational Rhapsody interprets an association/usage relationship as a request to generate, compile, and link with the client-side code of the CORBA interface.

Mixed and Standalone CORBA Interfaces

A «CORBAInterface» class that has both children and relations leads to the generation, compilation, and linkage of both server-side and client-side code.

Rational Rhapsody generates both server and client IDL code for the CORBA interface `IServer` and, in this case, keeps both. The `Client` class, which has an association relation to `IServer`, links to `IServer`'s client-side (stub) code. The `Server` class, which inherits from `IServer`, links to `IServer`'s server-side (skeleton) code. Therefore, if you allocate the classes to components (as described in the Hello World example), you end up with two executables (`Client.exe` and `Server.exe`) that share the CORBA interface.

Rational Rhapsody cannot interpret a CORBA interface that has neither children nor relations. Therefore, nothing (neither stub nor skeleton code) is generated for it. You can force stub or skeleton generation using the `ExposeCorbaInterfaces` and `UseCorbaInterfaces` properties (under `CORBA::Configuration`). See **Interpreting CORBA Interfaces** for details.

Mapping Clients and Servers to Components

In the CORBA context, a component can be either a client, a server, or both. From the deliverable point of view, a component can be either an executable or a library.

Mapping to Deliverable Components

The options for mapping CORBA components to deliverable components are as follows:

- **CORBA server executable -** You can map a CORBA interface and a class that implements it to the same component. Rational Rhapsody generates the code and makefile to create a server executable from the component. Set the `CORBA::Configuration::CORBAEnable` property to `CORBAServer` to generate a CORBA server `main()` loop.
- **CORBA client executable -** You can map a CORBA interface and a class that has an association to it to the same component. Rational Rhapsody generates the code and makefile to create a client executable from the component. Set the `CORBAEnable` property to `CORBAClient` to generate a client executable.

- **CORBA client/server executable -** You can map a CORBA interface, a class that implements it, and a class that has an association to it to the same component. Rational Rhapsody generates the code and makefile to create an executable capable of running as both a client and a server from the component.
- **CORBA interface library -** You can map your CORBA design into one component and your C++ design into another component. The CORBA component can generate a library that contains either the server library, the client library, or both. The C++ component can then use this library. See **Mapping Clients and Servers to Components** for more information.

Mapping Clients, Servers, and Interfaces to Libraries

Using Rational Rhapsody, you can create components with servers, clients, and interfaces packed in libraries. To build a library, do the following:

1. Allocate only CORBA items to a library component.
2. As you want, set properties for the exposed (or used) classes, attributes, operations, and types.
3. Generate code.

The end result is a library of stubs or skeletons, or whatever you have selected.

Using External IDL Files

To include an external IDL file in a CORBA model, do the following:

1. Add the external CORBA interface class to the model.

   In the password authentication sample, the `IEx` interface is an external IDL file that is supplied with a server.

2. Set the `CG::Class::UseAsExternal` property for the external class `IEx` to `True` so Rational Rhapsody will not generate code for it.

3. Type the name of the external IDL file (`IEx.idl`) in the `CG::Class::FileName` property for the external class `IEx`.

   Alternatively, you can set the `CORBA::Configuration::IncludeIDL` property for the configuration to the name of the external file

4. If the external file references any additional libraries that are not part of the model, add the external libraries in the **Libraries** field of the Settings tab of the Configuration dialog box.

Note: You can tailor the generated IDL code to various ORB vendors by inserting vendor-specific code segments.

Using Other ORBs

Rational Rhapsody is an open tool that you can adapt to ORBs, other than TAO, supplied by different vendors.

Adapting Rational Rhapsody for Other ORBS

To adapt Rational Rhapsody to an ORB other than TAO, you must modify the ORB adapter layer. This layer includes the following:

- The name of the class being inherited from (assuming inheritance rather than delegation).
- A batch file containing the command used to compile the IDL files.
- An optional `CORBA_env` parameter for operations.
- The format of the file name to include (different for client and server).

The format of compiled IDL file names varies with the IDL compiler in use. For example, Iona adds an “S” to the name of a server component in generated files (for example, an IDL file named `y.idl` is compiled into C++ files named `yS.hh` and `yS.cpp`), whereas other ORB vendors use different conventions. CORBA properties ensure that the correct filename formats are generated in the makefile for a particular CORBA environment.

- The name and format for a publishing function.
- The format used to notify the ORB that a server is available.
- The format for an IOR retrieval function.

This section describes the set of properties that support the Rational Rhapsody and CORBA workflow and specifically, the property set used to record the differences between various ORB solutions.

ORB Configuration

The file (in the `Share\Properties` directory under the Rational Rhapsody installation) contains the subject `CORBA` properties. The `ORB` property (under `CORBA::Configuration`) defines which ORB is selected to work with a specific configuration. You can build a component with different configurations, each using a different ORB, by modifying the `ORB` property for each configuration.

By default, the `ORB` property is set to `TAO`. However, a `UserDefinedORB` metaclass is also available, whose default settings correspond to the TAO settings. To add another ORB, you can modify the `UserDefinedORB` settings to hold the values of the new ORB.

In addition, you can add as many new ORB definitions as you want. This enables teams to use different versions or dialects of the same ORB, or to evaluate new ORBs. To add multiple ORBs, copy the `UserDefinedORB` settings to the `site.prp` file, edit them there, then add more ORB entries as needed.

Index