Rational Rose 2000e
Using Rose CORBA

Contents

Preface

This guide, *Rational Rose 2000e, Using Rose CORBA*, is for anyone who wants to use Rational Rose to model and forward engineer CORBA IDL constructs or reverse engineer CORBA IDL source.

It assumes that you are familiar with CORBA IDL concepts and constructs, and that you are comfortable with basic Rose concepts and procedures.

If you need to learn to use Rose, you should run the Rose tutorial included on your product CD.

How This Manual Is Organized

This manual contains the following seven chapters:

- **Chapter 1—** Introduction to Rational Rose CORBA

  Provides an overview of features, as well as basic forward, reverse and round-trip engineering concepts as they apply to Rose CORBA.
- **Chapter 2** — Mapping CORBA and Rational Rose Model Elements

  Provides general mapping guidelines, information on valid CORBA IDL types, nested and unnested classes, CORBA stereotypes, and custom specifications. It also describes how to define IncludePath settings, directory structures, IDL files, and certain Rose model elements, considering their effects upon one another.
- **Chapter 3** — Mapping CORBA Constants and Enums with Rational Rose Classes

  Provides detailed information and examples for mapping Rational Rose model elements and CORBA Constants and Enums.

- **Chapter 4** — Mapping CORBA Interfaces with Rational Rose Classes

  Provides detailed information and examples for mapping Rose model elements and CORBA Interfaces.
- **Chapter 5** — Mapping CORBA Typedefs with Rational Rose Classes

  Provides detailed information and examples for mapping Rose model elements and CORBA Typedefs.
- **Chapter 6** — Mapping CORBA Exceptions, Structs, and Unions with Rational Rose Classes

  Provides detailed information and examples for mapping Rose model elements and CORBA Exceptions, Structs, and Unions.
- **Chapter 7** — Forward and Reverse Engineering with Rational Rose CORBA

  Provides detailed instructions for forward engineering Rose model elements into CORBA IDL source and reverse engineering CORBA IDL source into a Rose model.

Related Documentation

The information in this guide is also provided in the form of online help. In addition, you will find context-based online help as you complete procedures and work in the various Rational Rose CORBA dialog boxes.

After installation and before you begin using Rational Rose CORBA, please review any **Readme.txt** files and **Release Notes** to ensure that you have the latest information about the product.

For additional resources, refer to the *Using Rational Rose* guide and online help. If you are new to Rational Rose, visual modeling, or the Unified Modeling Language (UML), you may also want to read the book, *Visual Modeling with Rational Rose and UML*, included the Rose product documentation kit.

Online Help and Manuals

Rational Rose CORBA includes comprehensive online help with hypertext links and a two-level search index.

In addition, you can find all of the user manuals online. Please refer to the Readme.txt file (found in the Rational Rose installation directory) for more information.

Chapter 1

Introduction to Rational Rose CORBA

What is Rational Rose CORBA?

Rational Rose CORBA is the Rational Rose add-in that allows you to:

- Forward engineer Rose model elements into CORBA-compliant IDL code
- Reverse engineer CORBA IDL code into appropriate Rose model elements

Rational Rose CORBA provides support for:

- Windows 95, Windows 98, Windows 2000 and Windows NT 4.0 platforms
- Model evolution from analysis to design
- Team development using configuration-management and version-control systems
- Automatic generation of CORBA-compliant IDL source code
- Reverse engineering of CORBA-compliant IDL source code
- Round-trip engineering capability that synchronizes models and generated IDL source across multiple iterations:
    - Changes made to a model are carried through to code during code generation.
    - Changes made to code are carried back to the model during reverse engineering.

- Design, modeling, and visualization of all CORBA IDL types, including:
    - Const
    - Enum
    - Struct
    - Union
    - Typedef
    - Exception
    - Interface
    - Native

Forward Engineering in Rational Rose CORBA

In Rational Rose CORBA, forward engineering is the process of generating CORBA source from one or more classes, packages, or components in a Rational Rose model.

Forward engineering in Rational Rose CORBA is component-centered. This means that the CORBA source generation is based on the component specification rather than on the class specification.

This does not mean, however, that you must work only in component diagrams in order to generate CORBA source. Instead, you can create a class and then assign it to a valid CORBA component in the browser, effectively creating the required component from your class.

Rational Rose CORBA maps the classes in your model to corresponding CORBA constructs based upon the stereotypes you assign to them. Any attributes and relationships you define for Rational Rose CORBA classes map to CORBA IDL attributes. In addition, any operations you define for Rational Rose CORBA Interfaces map to CORBA IDL operations.

Note: *The only CORBA class for which you can define operations is a CORBA Interface class.*

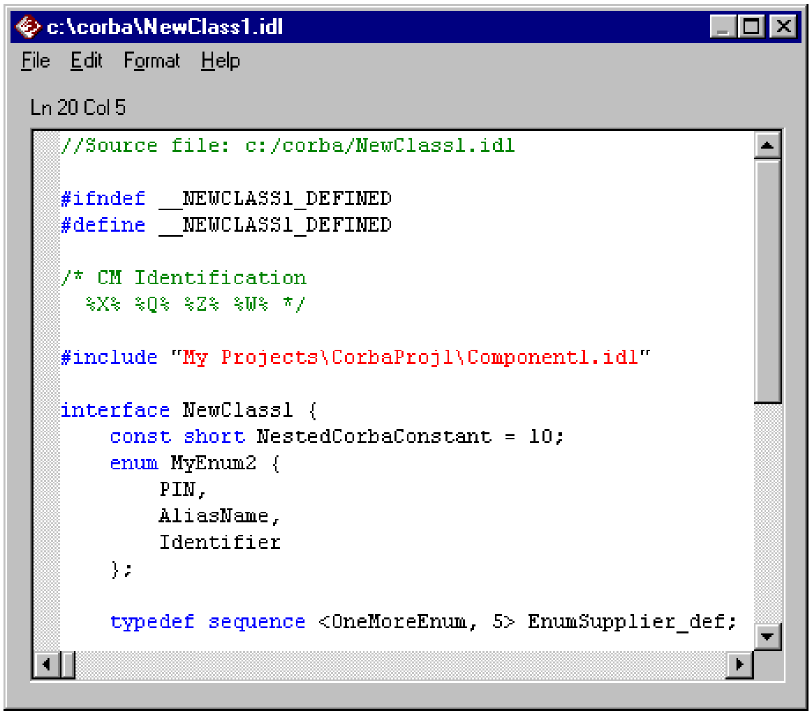
Reverse Engineering CORBA IDL

Reverse engineering is the process of analyzing CORBA IDL source code, mapping it to classes, components and relationships, and storing these elements in a Rational Rose model.

For the most part, when you reverse engineer IDL code that you previously forward-engineered from a Rational Rose model, your original model elements remain the same, except that new information or changes you entered into the IDL code are carried back into the model. The exceptions to this rule are some special cases of Rational Rose constructs that are provided for your convenience and are used for one-time forward engineering only. See Chapter 4, *Mapping CORBA Interfaces with Rational Rose Classes*, and Chapter 6, *Mapping CORBA Exceptions, Structs, and Unions with Rational Rose Classes*, for details on these special cases.

The Built-In CORBA Source Editor

Rational Rose CORBA provides a built-in source editor that allows you to display and edit CORBA IDL code that you generated from a model. The editor window provides easy navigation and uses color to identify keywords, literals, and comments. For details on setting and using the built-in CORBA editor, see Chapter 7, *Forward and Reverse Engineering with Rational Rose CORBA*.



```
//Source file: c:/corba/NewClass1.idl

#ifndef __NEWCLASS1_DEFINED
#define __NEWCLASS1_DEFINED

/* CM Identification
  %X% %Q% %Z% %W% */

#include "My Projects\CorbaProj1\Component1.idl"

interface NewClass1 {
    const short NestedCorbaConstant = 10;
    enum MyEnum2 {
        PIN,
        AliasName,
        Identifier
    };

    typedef sequence <OneMoreEnum, 5> EnumSupplier_def;
```

About CORBA Properties and Specifications

Rational Rose provides two generic mechanisms for controlling how model elements behave and how code is generated for them:

- Model properties, which provide global settings for a project and its classes, attributes, operations, components, and roles
- Standard Specifications, which control individual classes, attributes, operations, components, and roles

In addition to these mechanisms, Rational Rose CORBA provides a set of context-based custom specifications. Custom CORBA specifications extract settings from both the standard Rational Rose specifications and the model property settings, enabling you to display and modify certain CORBA-specific information using a single dialog. Any change you make on a CORBA custom specification updates the corresponding standard specification or model property as well.

Custom CORBA Specifications are available for:

- Project
- Class
- Attribute
- Operation
- Component

For information on updating the CORBA Custom Project Specification, see Chapter 2, *Mapping CORBA and Rational Rose Model Elements*, and Chapter 7, *Forward and Reverse Engineering with Rational Rose CORBA*.

For information on updating CORBA Custom Class, Attribute, Operation, or Component Specifications, see *Using Custom Specifications to Update Model Elements*, in Chapter 2.

RoseTip

You cannot use CORBA Custom Specifications to create classes, attributes or roles, operations, or components. You can only display and modify the settings for existing elements. To create a model element, use the standard Rational Rose toolbars and specifications.

Chapter 2

Mapping CORBA and Rational Rose Model Elements

General Guidelines

The mapping between CORBA IDL and Rational Rose model elements is affected by several factors:

- Include Path definition in the Rational Rose model
- Directory structure of the IDL or the hierarchy of Rational Rose component packages
- CORBA module definitions and Rational Rose logical package definitions
- IDL file contents and Rational Rose component assignments
- IDL `#include` definitions and forward references and Rational Rose relationship definitions
- IDL constructs and Rational Rose classes and stereotypes

CORBA IDL Types

There are three categories of CORBA IDL types: Fundamental Types, Template Types, and User-Defined Types. These types apply to attributes and roles you define for CORBA classes in your model.

In addition, Rational Rose CORBA supports the use of native types, which allow you to define new fundamental types that other CORBA constructs can use as attribute types, or, in the case of operations, as parameter or return types.

Fundamental Types

In a Rational Rose model, CORBA fundamental types are basic data types. The following are valid CORBA fundamental types:

- Integer types
  - short
  - long
  - long long
  - unsigned short
  - unsigned long
  - unsigned long long
- Floating types
  - float
  - double
  - long double
- char
- wchar
- boolean
- octet
- any
- object

Template Types

Template types are specialized data types as defined in the CORBA specification. The following are valid CORBA template types:

- sequence
- string
- wstring
- fixed

User-Defined Types

User-defined types map to CORBA constructs. You define these types as model classes and then use them as attributes of other classes by defining relationships to them. The following are valid CORBA user-defined types:

- const
- enum
- struct
- union
- typedef
- exception
- interface
- native

Using Native Types

The native type maps to a Rational Rose class with the stereotype CORBANative.

The native type, which is similar to a CORBA fundamental type, allows you to specify programming language-dependent types for use by object adapters. In CORBA IDL, the native type declaration allows object adapters to define new fundamental types without requiring changes to the Object Managament Group (OMG) IDL language specifications.

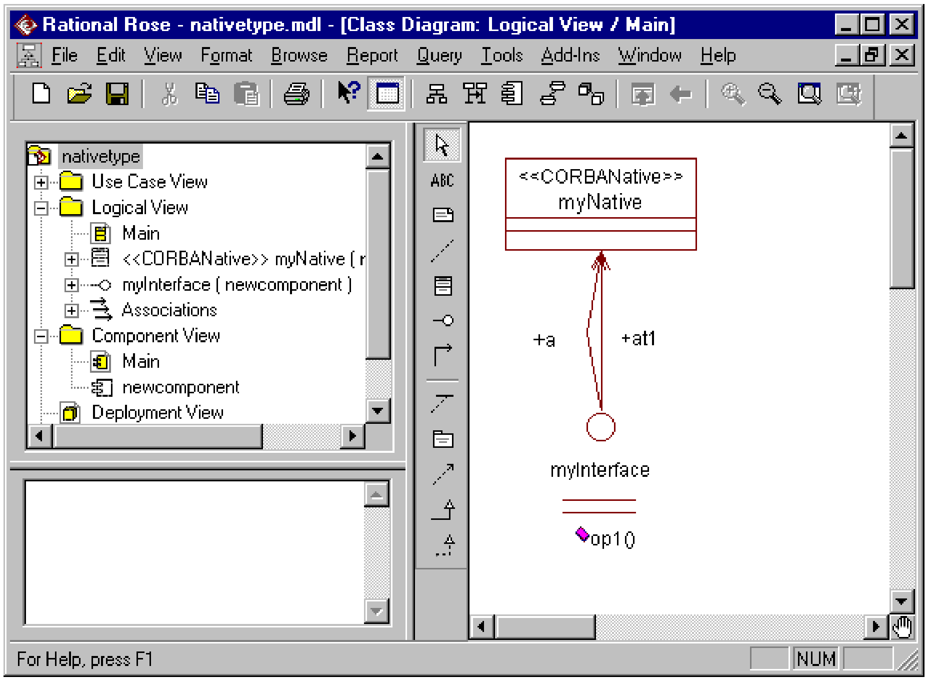
Once you create the type, other CORBA constructs can use it, either as an attribute, or, in the case of operations, as a parameter or return type.

If you forward engineer a native type from Rose, you get a native declaration in your IDL file.

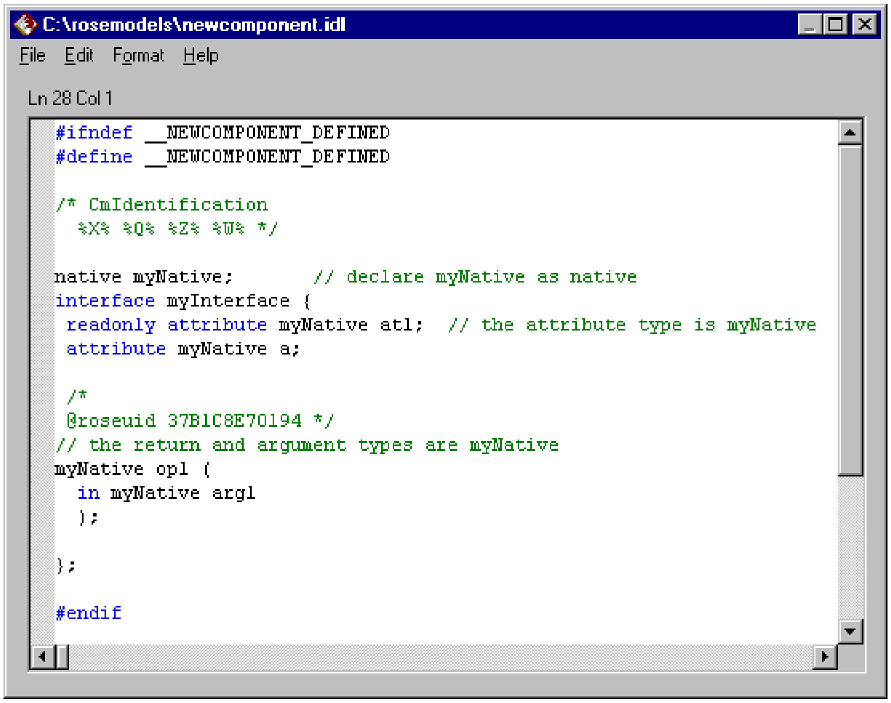
If you reverse engineer a native declaration from CORBA IDL, Rational Rose adds a class with a stereotype of CORBANative to your model.

Creating and Using Native Types

The following example shows a model with a CORBA native type class called MyNative. This class defines a new fundamental type, which can be used as an attribute type by other CORBA types in the model. In this case, because of the association defined between myInterface and MyNative, myInterface has an attribute whose type is the native type defined by myNative.


Rational Rose - nativetype.mdl - [Class Diagram: Logical View / Main]
File Edit View Format Browse Report Query Tools Add-Ins Window Help
nativetype
Use Case View
Logical View
Main
<<CORBANative>> myNative (r
myInterface (newcomponent)
Associations
Component View
Main
newcomponent
Deployment View
<<CORBANative>>
myNative
+a
+at1
myInterface
op1()
For Help, press F1
NUM

The following figure shows the fragment of CORBA IDL that corresponds to these model elements.



```
#ifndef __NEWCOMPONENT_DEFINED
#define __NEWCOMPONENT_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

native myNative;       // declare myNative as native
interface myInterface {
 readonly attribute myNative at1;  // the attribute type is myNative
 attribute myNative a;

 /*
 @roseuid 37B1C8E70194 */
// the return and argument types are myNative
myNative op1 (
  in myNative arg1
  );

};

#endif
```

Nested and Unnested Classes

CORBA IDL constructs (user-defined types) can map to Rational Rose model elements in two ways:

- Within a CORBA Interface definition

  In this context, you define the type as a nested class of an interface class. When you define the type in this way, other entities, such as interfaces, can only reference the type in terms of the interface in which it is nested.
- Outside a CORBA Interface definition

  In this context, you define an independent class to represent the type. When you define the type in this way, other entities, such as interfaces, can reference the type directly.

In either case, you define the class with the stereotype that applies to the particular CORBA type. For example, to create a class representing a CORBA constant you, give it the stereotype **CORBAConstant**.

When you forward engineer your model into CORBA IDL, Rational Rose CORBA creates the IDL code for the class, mapping CORBA attributes to the attributes and relationships defined in the model.

For an example of classes nested in a CORBA Interface, see the Interface examples in Chapter 4.

Stereotypes for CORBA Classes

The valid stereotypes for CORBA classes are:

- CORBAConstant
- CORBAEnum
- CORBAException
- CORBAStruct
- CORBATypedef
- CORBAUnion
- CORBANative
- Interface

Using Custom Specifications to Update Model Elements

This subsection tells how to work with custom specifications for CORBA classes, components, attributes and roles, and operations.

The custom CORBA Project Specification is used for IncludePath Mapping, as well as for Forward and Reverse Engineering. For information on using the custom Project Specification, see *IncludePath Mapping,* later in this chapter, or see *Forward Engineering* and *Reverse Engineering* in Chapter 7.

If the CORBA Add-In is active, there are several ways to access custom specifications for CORBA model elements:

- Double-click a CORBA model element in a diagram or in the browser.
- Select a CORBA model element in a diagram or in the browser and press **F4**.
- Right-click a CORBA model element in a diagram or in the browser and then click **Open Specification** on a context menu.

RoseTip

To open a standard Rational Rose Specification for a CORBA model element when the CORBA Add-In is active, right-click the element in a diagram or in the browser and then click **Open Standard Specification** on the context menu.

Working with the Custom CORBA Class Specification

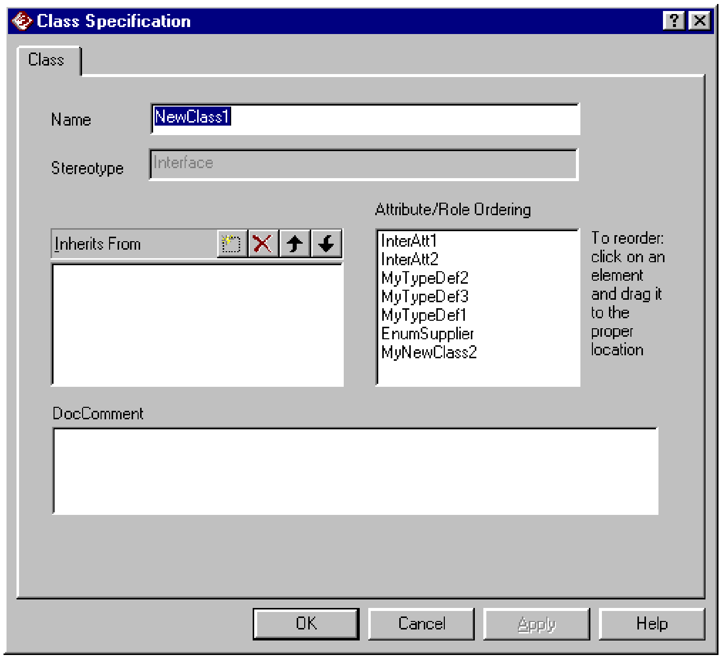
Custom Class Specifications vary according to the stereotype of the class. In general, the custom Class Specification lets you display such information as class name and stereotype, attribute and role order, inheritance relationships and comments. With the exception of the class stereotype, you can also edit this information in the custom specification.

The custom Class Specification gives you access to attribute and role order, but does not allow you to add, delete or edit the attributes or roles themselves.

To access a CORBA custom Class Specification:

1. Right-click the class in a diagram or in the browser.
2. Click **Open Specification**.

Here is a sample custom Class Specification for a CORBA Interface.



Name: NewClass1

Stereotype: Interface

Inherits From

Attribute/Role Ordering

InterAtt1
InterAtt2
MyTypeDef2
MyTypeDef3
MyTypeDef1
EnumSupplier
MyNewClass2

To reorder: click on an element and drag it to the proper location

DocComment

RoseTips

Keep in mind that each CORBA stereotype has a somewhat different custom specification. For example, because Interfaces can have inheritance (generalize) relationships, this Specification includes tools that allow you to specify inheritance. Always use the field-based online help to get full instructions for any of these specifications.

To display a CORBA custom Specification for a class, the class must already be assigned to a component in your Component View and that component must have its language set to CORBA.

Defining Attributes and Operations for CORBA Classes

There are several ways to add attributes and operations to CORBA classes. The easiest and most common are:

- Right-click a class in a diagram or in the browser, and select **New Attribute** or **New Operation**.
- Right-click a class in a diagram or in the browser, and select **Open Standard Specification**. Then insert the attribute or operation on the Attribute or Operation tab of the standard Rose Class Specification.

RoseTips

You can only define operations for a CORBA Interface class. No other CORBA class takes an operation.

When you add an attribute in Rose, its export control is set to **Private**. Since Export Control has no meaning in CORBA IDL, you can either change the setting to **Public**, or ignore the setting. In any case, Rose CORBA ignores the Export Control setting during forward and reverse engineering.

Working with the Custom CORBA Attribute Specification

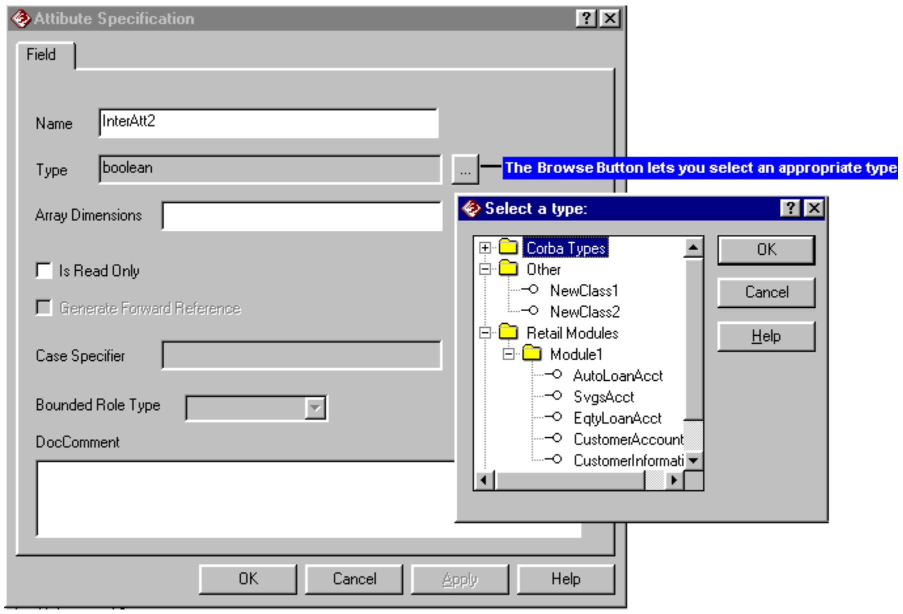
Attribute Specifications let you display and modify CORBA class attributes and roles.

Depending upon the stereotype of the class to which the attribute or role belongs, you can specify data type, array dimensions, implementation type, case specifier, role type, forward references, and read/write accessibility.

To access a custom CORBA Attribute Specification:

1. Right-click the attribute or role in the browser.
2. Click **Open Specification**.

Here is a sample custom Attribute Specification page. Notice that the Type field provides a Browse button to allow you to select the attribute's type from valid CORBA types, as well as from the user-defined types available in your model.



Name: InterAtt2
Type: boolean
Array Dimensions:
Is Read Only
Case Specifier:
Bounded Role Type:
DocComment:

RoseTip

Remember that not all Attribute Specification pages look exactly like this one. Use the field-based online help to get full instructions while using any custom specification.

Working with the Custom CORBA Operation Specification

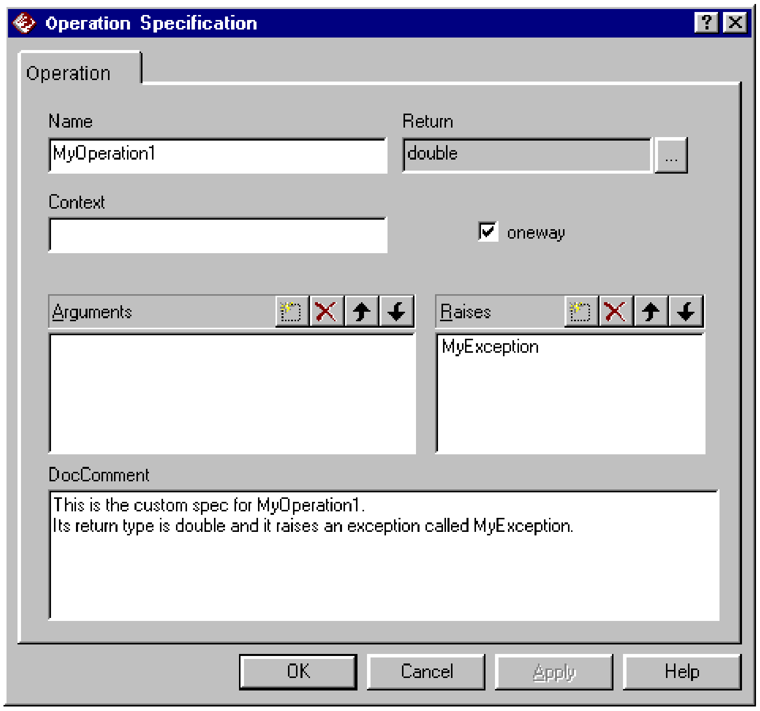
Operation Specifications let you display and modify the return type, context, arguments, and raises relationships for a CORBA operation.

You can also set the operation's **oneway** modifier property. Use the Browse button to select from the list of valid CORBA operation return types.

To access a custom CORBA Operation Specification:

1. Right-click the Operation in the browser.
2. Click **Open Specification**.

As its comment field shows, the following Operation Specification describes a oneway operation called **MyOperation1**. Its return type is double; it has no defined arguments; and it raises an exception called **MyException**.



Name: MyOperation1

Return: double

Context:

[✓] oneway

Raises: MyException

DocComment:
This is the custom spec for MyOperation1.
Its return type is double and it raises an exception called MyException.

Use the field-based online help to get full instructions while using any custom specification.

Only CORBA Interfaces can take operations.

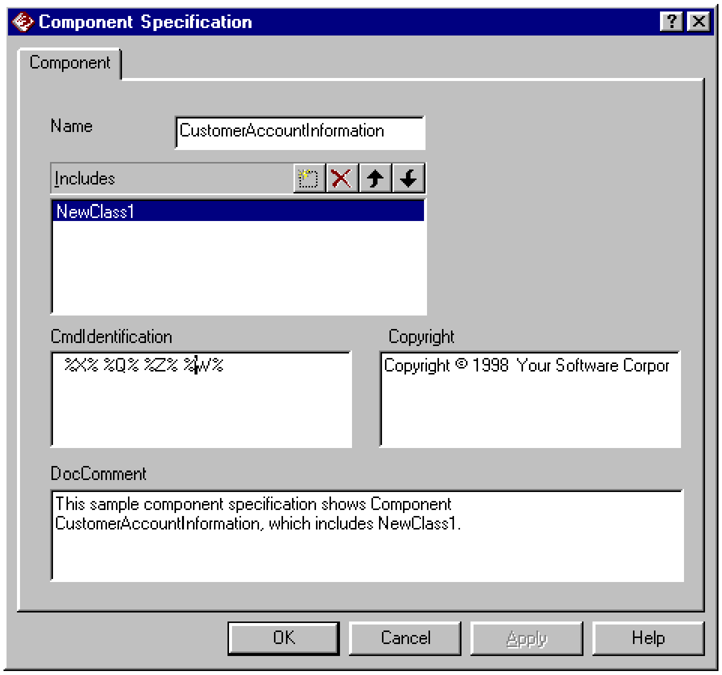
Working with the Custom CORBA Component Specification

Component Specifications let you display and modify information about a component, including other components for which to generate `#include` statements, configuration management identifier information, copyright information and comments.

To access a custom CORBA Component Specification:

1. Right-click the Component in a diagram or in the browser.
2. Click **Open Specification**.

The following sample custom Component Specification defines a component called CustomerAccountInformation. Note that the generated IDL code for this component will have a `#include` statement for **NewClass1**.



Name: CustomerAccountInformation

Includes: NewClass1

CmdIdentification: %X% %Q% %Z% %W%

Copyright: Copyright © 1998 Your Software Corpor

DocComment: This sample component specification shows Component CustomerAccountInformation, which includes NewClass1.

Use the field-based online help to get full instructions while using any custom specification.

Include Path Mapping

Overview

CORBA IncludePaths map to the project level IncludePath property in Rational Rose. Each Include Path, in conjunction with any relative directory paths you define, corresponds to a directory in which the project's IDL files can reside. Any paths you define are appended to the INCLUDE Environment variable.

- During forward engineering, Rational Rose uses the Include Path to create the directory structure for the IDL files it generates.
- During reverse engineering, Rational Rose uses the Include Path to resolve the physical location of the IDL files being reversed into the model.

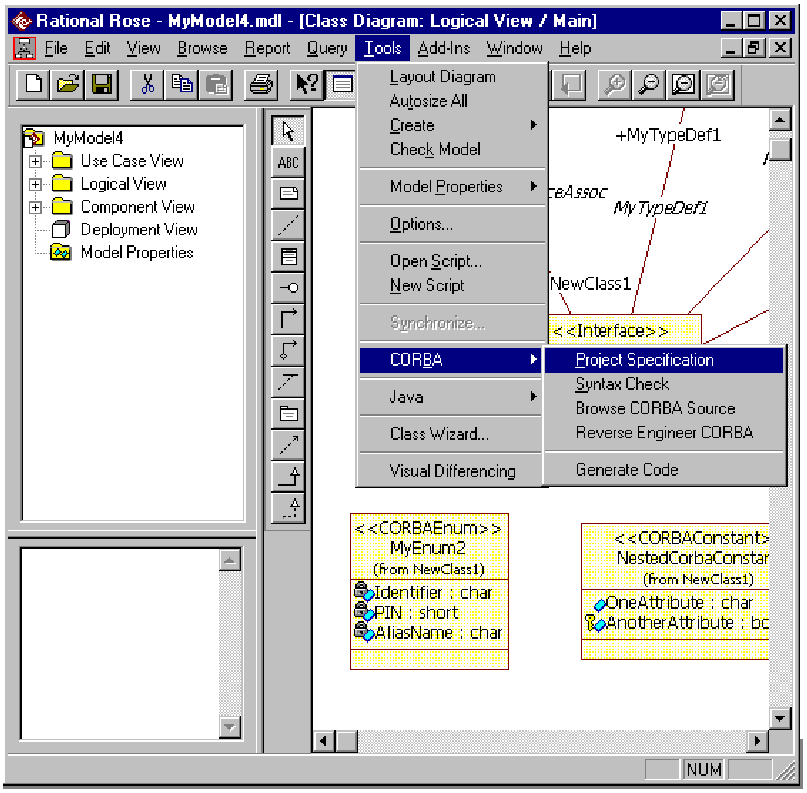
Defining one or more Include Paths allows you to specify relative file locations instead of entering fully qualified file names. This is particularly useful if your project paths use many levels of subdirectories.

You use a CORBA Project Property to define additional paths.

If you have defined any symbols in your model's path map, you can specify the symbol instead of the path. Then, if you want to display the actual path instead of the symbol, you can check **Resolve Path Map Variables**.

Accessing the IncludePath Property

To access the IncludePath property using the CORBA custom Project Specification, click **Tools** > **CORBA** > **Project Specification**.



Extending the Project IncludePath

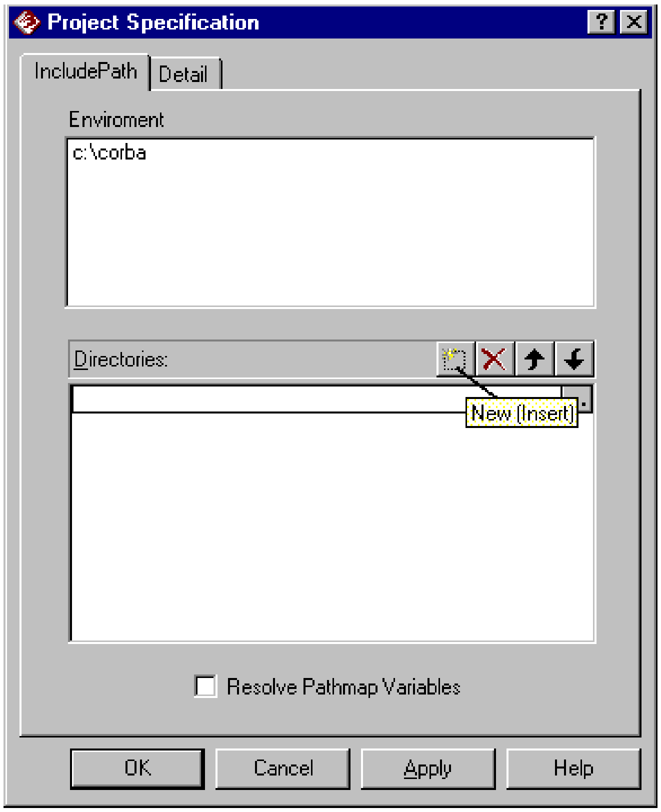
The IncludePath page of the custom Project Specification shows the IncludePath settings that are currently defined in your Environment variable. You can extend the IncludePath setting by adding entries to the Directories section of the dialog.

RoseTip

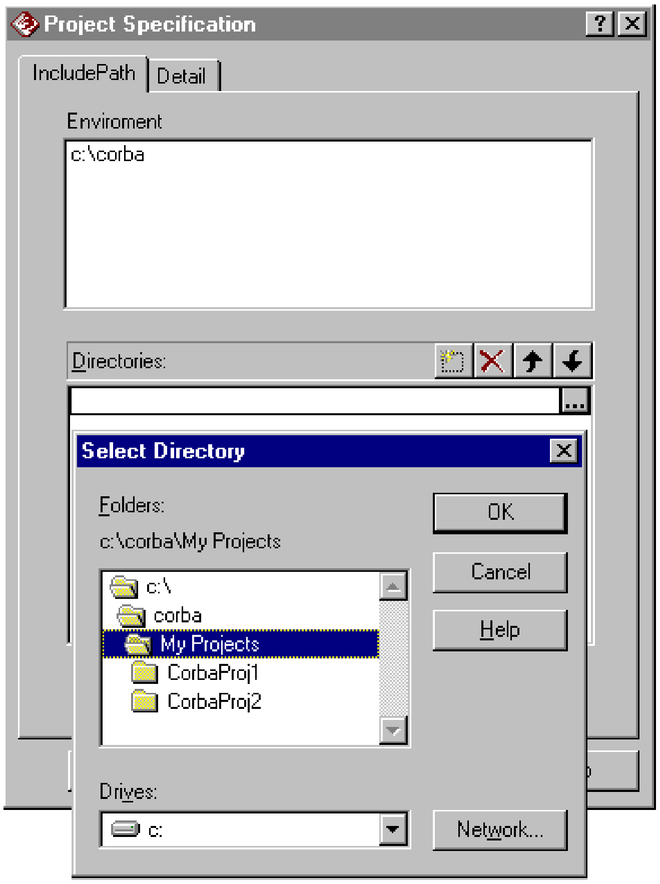
You can also extend the project's IncludePath by using the CORBA tab of the standard Rose Project Properties Specification. However, if you use this specification, you must use the IncludePath property field to add additional directories to the IncludePath. There is no separate **Directories** field on this standard specification.

Follow these steps to add entries to the Directories section of the CORBA custom Project Specification:

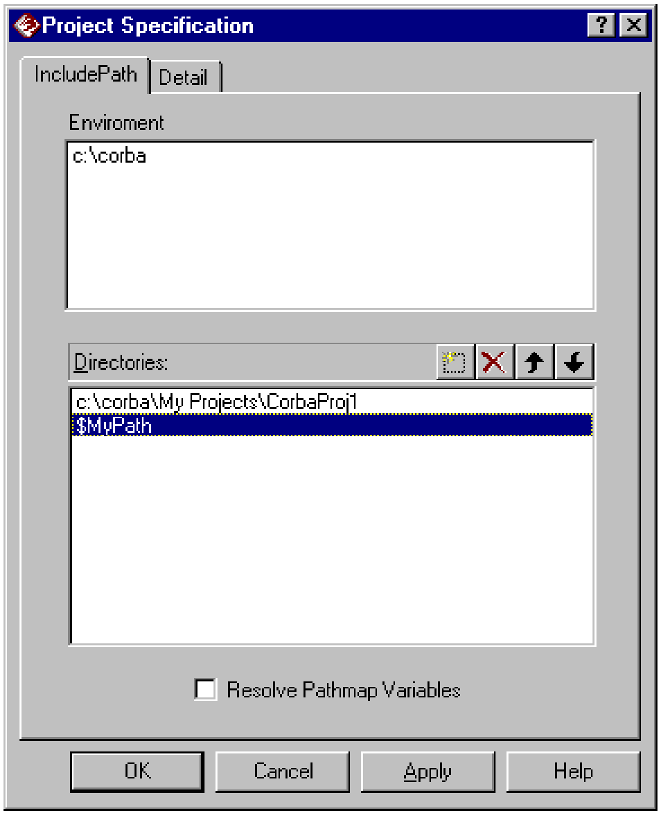
1. Open the CORBA custom Project Specification and click the New icon.



2. Do any or both of the following:
   - Enter a directory path or a path map variable in the field provided.
   - Click the Browse button and select a directory from the **Select Directory** dialog box. Click **OK**.



3. Repeat Step 2 until you have entered or selected all necessary directories. Your specification might look something like this:


Project Specification

IncludePath | Detail

Enviroment

c:\corba

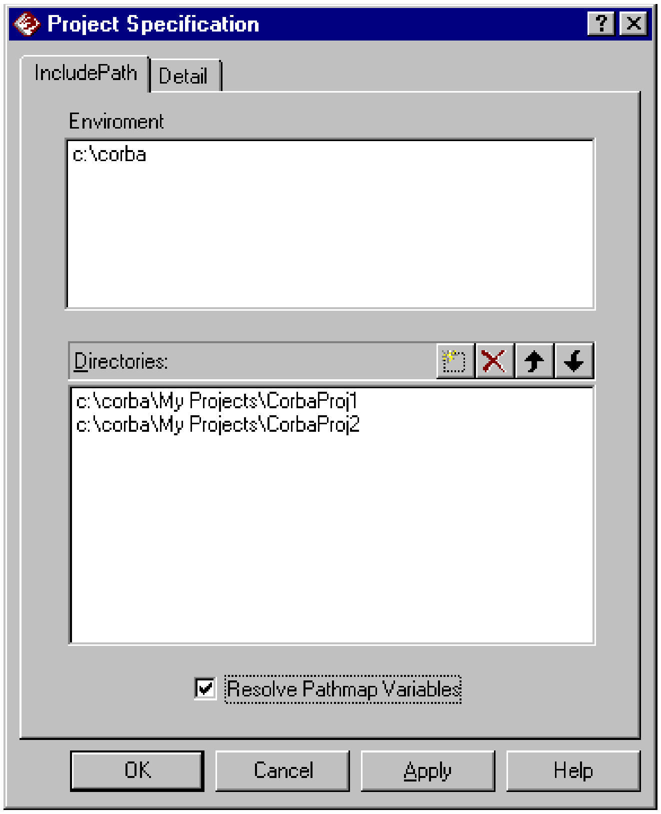
Directories:

c:\corba\My Projects\CorbaProj1
$MyPath

☐ Resolve Pathmap Variables

4. If you entered any path map variables, you can click the **Resolve Pathmap Variables** setting to change the virtual path to the physical path name.

In the following Project Specification, notice that **$MyPath** (the path map variable shown in Step 3) has been converted to the path defined for it in the model's Path Map **(c:\corba\MyProjects\CorbaProj2)**.



Enviroment

c:\corba

Directories:

c:\corba\My Projects\CorbaProj1
c:\corba\My Projects\CorbaProj2

[✓] Resolve Pathmap Variables

RoseTip

In Rational Rose, a virtual path name is created by replacing an actual pathname with a virtual path variable (or symbol) using the Path Map Editor.

To see how this works, display the **Virtual Path Map** dialog box by selecting **File** > **Edit Path Map...** from the menu. For detailed information, click **Help** in this dialog box.

Directory Structure and Component Package Mapping

A directory of CORBA IDL files maps to a component package in a Rational Rose model.

If you forward engineer a component package, Rational Rose CORBA forward engineers each component belonging to the package. The result is a directory corresponding to the component package and, within that directory, an IDL file for each component in the package.

Normally, you set up your model's component view so that the package hierarchy matches your project IncludePath. However, you can create component packages and components outside of the IncludePath hierarchy. If you do so, the Component Mapping dialog will appear during forward engineering and require you to map the component (or its parent package) to a valid IncludePath. See *Mapping Components During Code Generation* in *Chapter 7*, for more information.

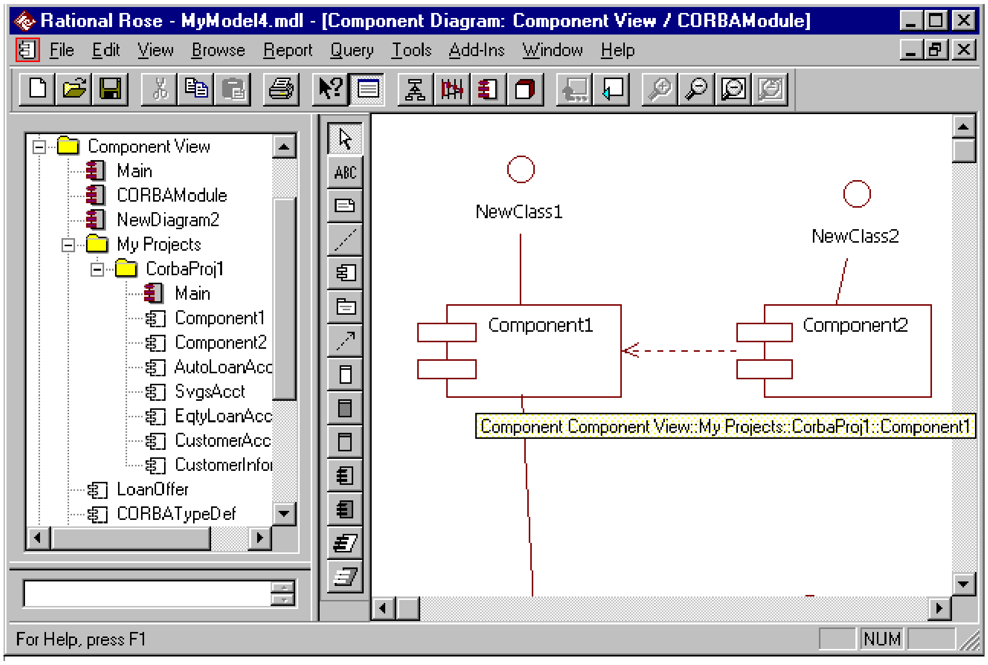
RoseTips

- If you create a hierarchy of packages in your model, this becomes a hierarchy of subdirectories for your generated code. The hierarchy is always relative to one of your model's defined IncludePath entries.
- If you reverse engineer a directory that contains individual IDL files, the resulting model will define a component package that corresponds to the directory, as well as a component for each IDL file contained in that directory. The directory will be relative to one of your model's defined IncludePath entries.
- When you forward engineer a component, Rational Rose CORBA attempts to locate a matching file within the appropriate IncludePath. If it cannot find a matching file, it creates a new one.

Example

This example illustrates the relationship between component packages in a model and the IDL directory structures that are created during forward engineering.

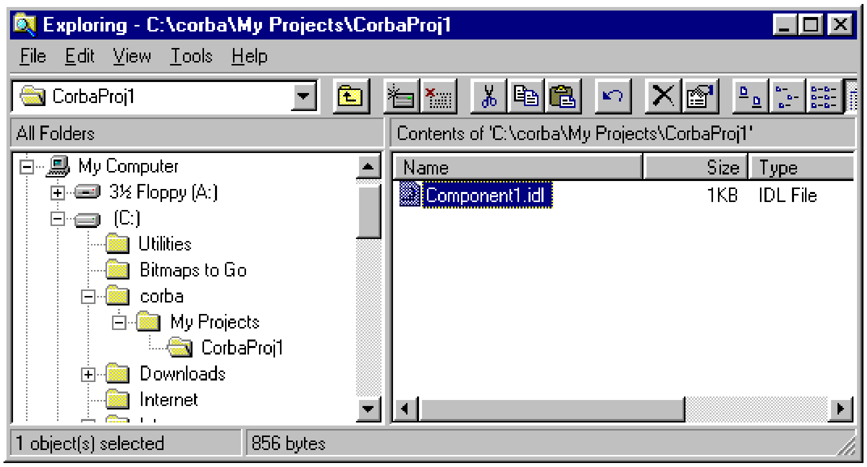
Suppose you have a model with a component package structure like the one shown in **MyModel4.mdl**. Notice that the Component View contains a component package called **My Projects,** which contains a package called **CorbaProj1**. CorbaProj1 contains a component called **Component1** (as well as other components).



Component Component View::My Projects::CorbaProj1::Component1

As you can see, when you generate code from **Component1:**

- The hierarchy of component packages maps to a hierarchy of subdirectories, relative to the IncludePath (**c:\corba** in this case).
- The component maps to an IDL file (**Component1.idl**).



Contents of 'C:\corba\My Projects\CorbaProj1'

Name	Size	Type
Component1.idl	1KB	IDL File

1 object(s) selected   856 bytes

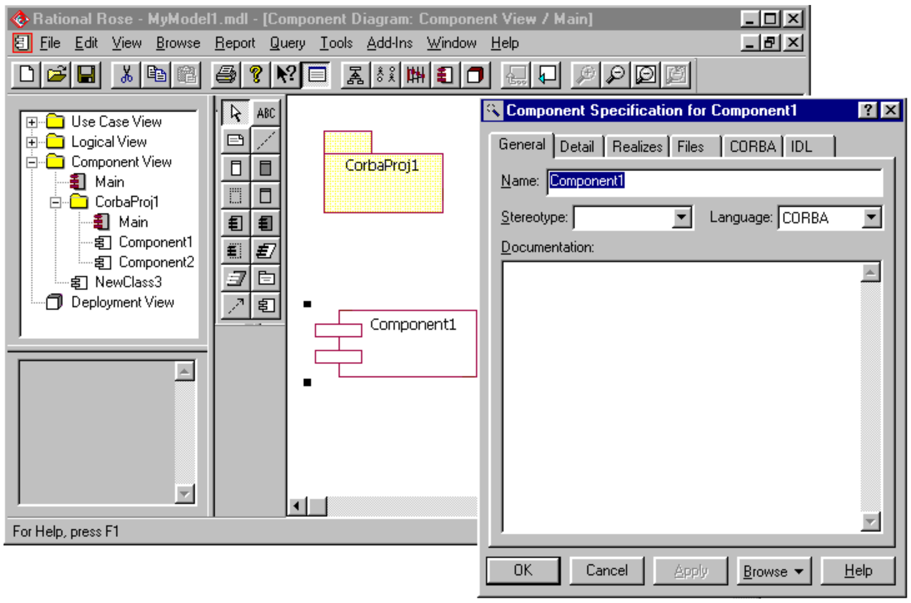
RoseTip

If you set the CreateMissingDirectories project property to True, Rational Rose CORBA will create these subdirectories during forward engineering if they do not already exist. (Use the Detail Page of the Project Properties to set CreateMissingDirectories. See Chapter 7 for instructions.)

You can reverse this example to illustrate the reverse engineering process. However, the dependency relationship shown in the model between Component1 and Component2 would only be added if the Component2.idl file includes a `#include` reference to Component1.idl. See *Relationship Mapping — #Includes and Forward References*, later in this chapter, for more information.

IDL File and Rational Rose Component Mapping

Each CORBA IDL file maps to a Rational Rose component whose assigned language is CORBA. The name of the component is the same as the simple (unqualified) name of the file. So, **Component1** in the model corresponds to a single IDL file called **Component1.idl.**


Rational Rose - MyModel1.mdl - [Component Diagram: Component View / Main]
File Edit View Browse Report Query Tools Add-Ins Window Help
Use Case View
Logical View
Component View
Main
CorbaProj1
Main
Component1
Component2
NewClass3
Deployment View
For Help, press F1
CorbaProj1
Component1
Component Specification for Component1
General Detail Realizes Files CORBA IDL
Documentation:
OK Cancel Apply Browse Help

Name: Component1
Stereotype:
Language: CORBA

When you create logical classes in your model, you can assign one or more of them to the same component. If you assign multiple classes to the same component, a single IDL file will contain the CORBA IDL source that you generate for all of its assigned classes.

If you do not explicitly assign a class to a component in your model, Rational Rose CORBA creates a component with the same name as the class. In this case, the IDL file you generate will contain the source for this one class only.

The location of the IDL files and the placement of components within a package hierarchy are affected by your project's IncludePath entries and mappings. For more information, see *Directory Structure and Component Package Mapping*, earlier in this Chapter.

Relationship Mapping — #Includes and Forward References

When you forward engineer classes or components in a Rose model, you get a single IDL file for each component.

You must assign each class to an appropriate component. (You can assign multiple classes to a single component.) Then, whether you forward engineer from the class or the component, Rose CORBA generates or updates the IDL file for the component.

Component assignments, as well as any relationships you define for classes or components, affect the IDL that Rose CORBA generates.

What Generates a Forward Reference?

For a client class that is in one of the following relationships, Rose CORBA generates a forward reference:

- A dependency relationship between CORBA classes when the supplier class is an interface
- An association relationship between CORBA classes when the supplier class is an interface and the client role's GenerateForwardReference property is set to True

RoseTip

Forward references are only valid when the supplier class is a CORBA interface. To avoid circular #include statements, a forward reference is generated whenever possible.

What Generates a #Include Statement?

For a client class is in one of the following relationships, Rose CORBA generates a #include statement instead of a forward reference:

- A dependency relationship between CORBA classes when the supplier class is not an interface
- An association relationship between CORBA classes when the supplier class is an interface, but the client role's GenerateForwardReference property is set to False
- An association relationship between CORBA classes when the supplier class is not an interface
- A generalization relationship between any CORBA classes, regardless of the supplier class's stereotype and regardless of the value of the client's GenerateForwardReference property.

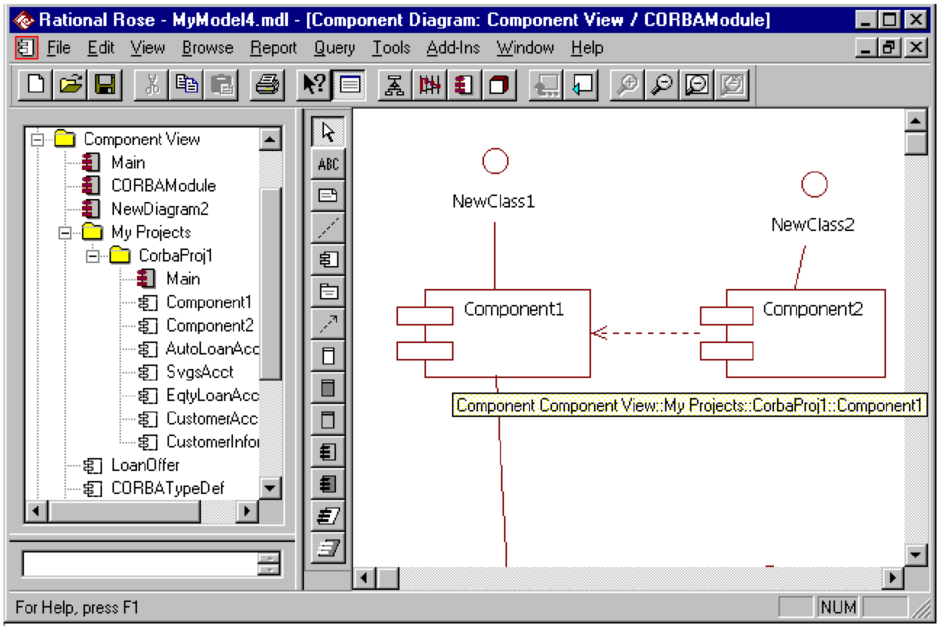
RoseTip

The path name generated in a #include statement is scoped to be relative to the project's IncludePath.

Example 1 - Using a Component Dependency to Map a #Include Statement

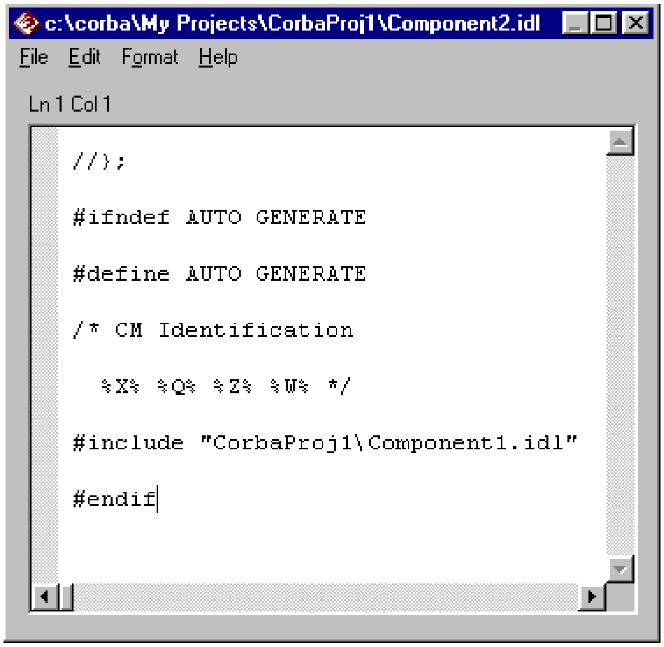
This example shows a `#include` statement generated from components that have a Uses (Dependency) relationship.

In this component diagram, Component1 is the supplier and Component2 is the client in the relationship.



When you forward engineer the components (or the package that contains them), you get one IDL file for each component (Component1.idl and Component2.idl).

The IDL code generated for Component2 is shown below. Because of the relationship between the two components, Component2.idl contains a `#include` statement for Component1.idl.

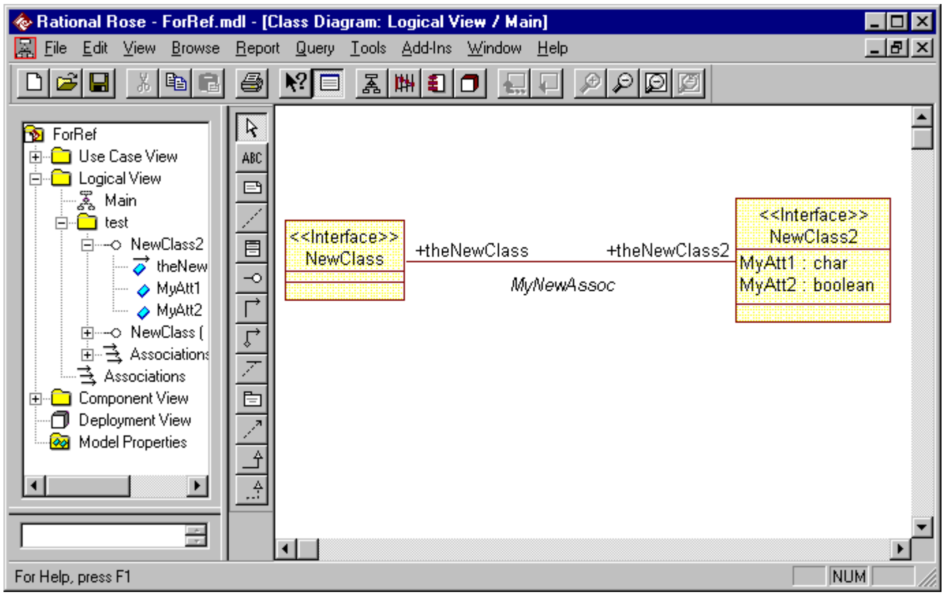


```
//};

#ifndef AUTO GENERATE

#define AUTO GENERATE

/* CM Identification

  %X% %Q% %Z% %W% */

#include "CorbaProj1\Component1.idl"

#endif
```

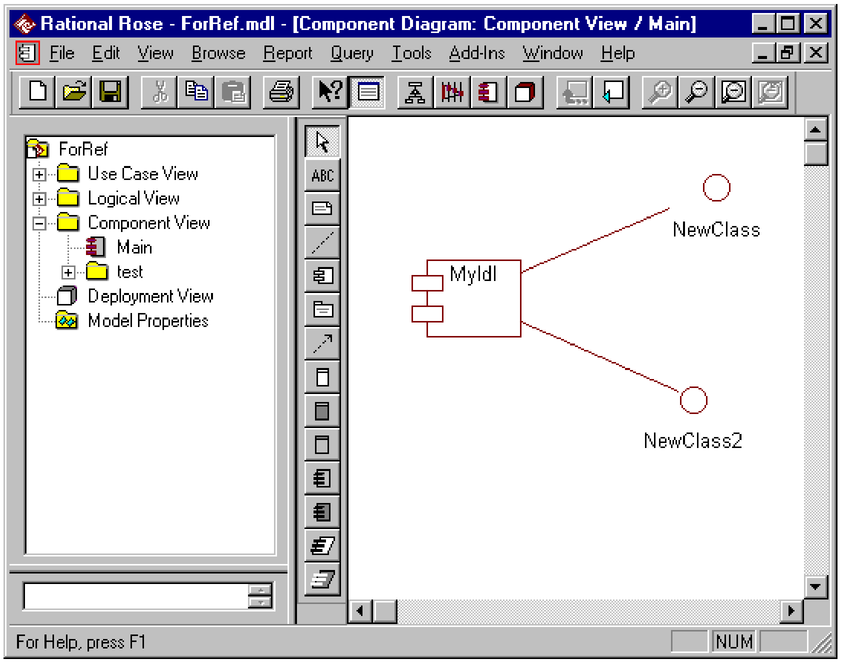
Example 2 - Using a Class Association to Map a Forward Reference

This example shows a forward reference generated from classes that have an association relationship.

This class diagram defines a nondirectional association between two interface classes (**NewClass** and **NewClass2**).



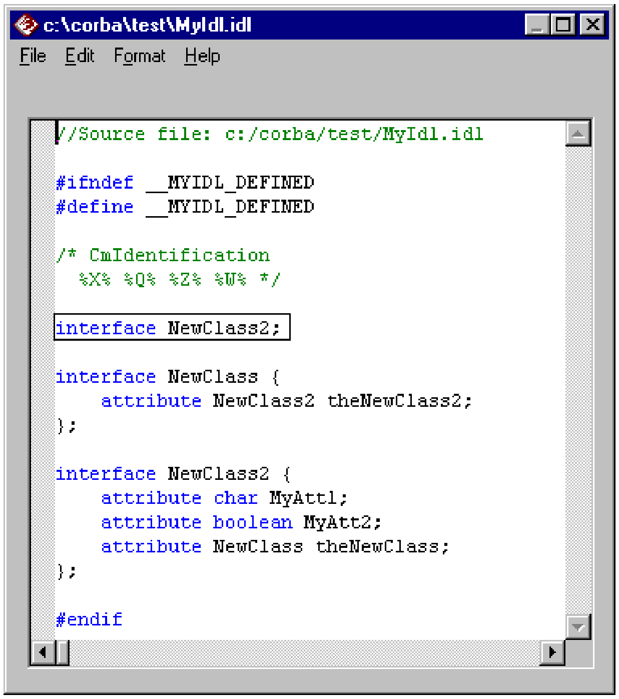
Both classes are assigned to a single component called **MyIDL**, as shown in the following component diagram.


Rational Rose - ForRef.mdl - [Component Diagram: Component View / Main]
File Edit View Browse Report Query Tools Add-Ins Window Help
ForRef
Use Case View
Logical View
Component View
Main
test
Deployment View
Model Properties
ABC
MyIdl
NewClass
NewClass2
For Help, press F1
NUM

When you forward engineer the classes (or the component to which they are assigned), you get an IDL file called **MyIDL.idl**.

Notice the forward reference to **NewClass2** (boxed in the following figure), which allows the code to identify **NewClass2** as an attribute of **NewClass** before **NewClass2** is actually defined in the file.

In the IDL code, the name (declarator) of the attribute is the name of the supplier role of the relationship (**theNewClass2**) and its type is the fully qualified name of the supplier class (**NewClass2**).



```
//Source file: c:/corba/test/MyIdl.idl

#ifndef __MYIDL_DEFINED
#define __MYIDL_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

interface NewClass2;

interface NewClass {
    attribute NewClass2 theNewClass2;
};

interface NewClass2 {
    attribute char MyAtt1;
    attribute boolean MyAtt2;
    attribute NewClass theNewClass;
};

#endif
```

CORBA Module to Rational Rose Logical Package Mapping

A CORBA module maps to a logical package with the stereotype **CORBAModule**. As with any logical package, Rational Rose supports hierarchical module structures by allowing a hierarchy of packages within a model. However, the structure of logical packages in the model does not affect the resulting IDL file structure. Only Component Package structure maps to IDL file structure.

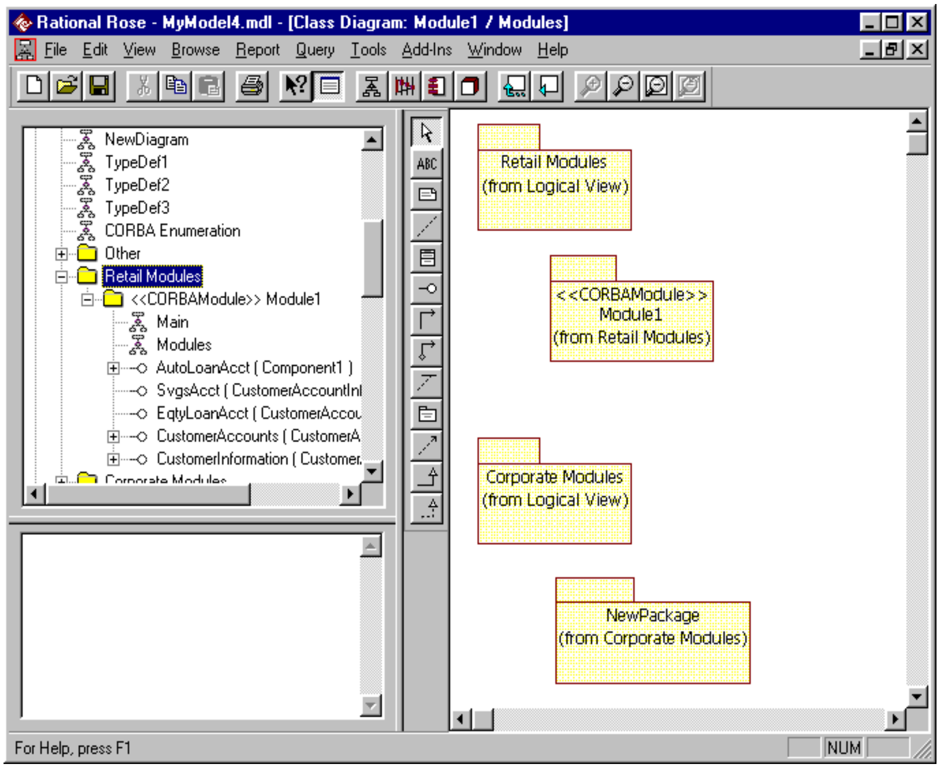
If you forward engineer a logical package stereotyped as CORBAModule, the IDL files for any components assigned to the logical classes belonging to that package will contain the module definition.

Conversely, if you reverse engineer IDL that contains a CORBA module definition, that module will be added to the model as a logical package.

Example

In this example, you can see that the Retail Modules package contains a nested package, whose stereotype is CORBAModule.

Looking at the browser window, notice that the CORBA module contains a set of logical classes. Let's assume that these logical classes have been assigned to a component called **CustomerAccountInformation**.


Rational Rose - MyModel4.mdl - [Class Diagram: Module1 / Modules]
File Edit View Browse Report Query Tools Add-Ins Window Help
NewDiagram
TypeDef1
TypeDef2
TypeDef3
CORBA Enumeration
Other
Retail Modules
<<CORBAModule>> Module1
Main
Modules
AutoLoanAcct (Component1)
SvgsAcct (CustomerAccountInf
EqtyLoanAcct (CustomerAccou
CustomerAccounts (CustomerA
CustomerInformation (Customer
Corporate Modules
Retail Modules (from Logical View)
<<CORBAModule>> Module1 (from Retail Modules)
Corporate Modules (from Logical View)
NewPackage (from Corporate Modules)
For Help, press F1
NUM

When you forward engineer the logical package (**Module1**), Rational Rose CORBA generates the CORBA IDL for its classes and places the code in a file called **CustomerAccountInformation.idl**.

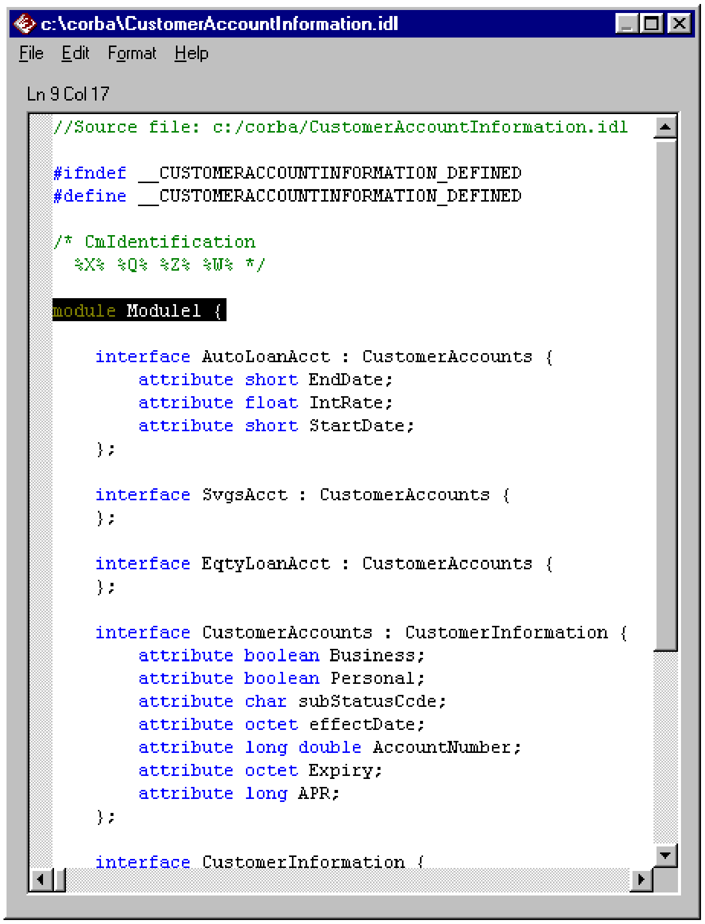
If you open this IDL file, you will see that it contains a CORBA module definition in the format:

```
module moduleidentifier { definition_1; definition_2, … }
```

Note that the *module identifier* corresponds to the package name (**Module1**). In this case, the definitions correspond to interfaces because the classes in **Module1** happened to be defined as CORBA interfaces. However, the definition could be any of the following user-defined types:

- Module
- Struct
- Exception
- Constant
- Typedef
- Union
- Enumeration
- Native

The following CORBA IDL file shows the code for Module1 as generated in CustomerAccountInformation.idl.



```
//Source file: c:/corba/CustomerAccountInformation.idl

#ifndef __CUSTOMERACCOUNTINFORMATION_DEFINED
#define __CUSTOMERACCOUNTINFORMATION_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

module Module1 {

    interface AutoLoanAcct : CustomerAccounts {
        attribute short EndDate;
        attribute float IntRate;
        attribute short StartDate;
    };

    interface SvgsAcct : CustomerAccounts {
    };

    interface EqtyLoanAcct : CustomerAccounts {
    };

    interface CustomerAccounts : CustomerInformation {
        attribute boolean Business;
        attribute boolean Personal;
        attribute char subStatusCode;
        attribute octet effectDate;
        attribute long double AccountNumber;
        attribute octet Expiry;
        attribute long APR;
    };

    interface CustomerInformation {
```

Chapter 3

Mapping CORBA Constants and Enums with Rational Rose Classes

CORBA Constant Mapping

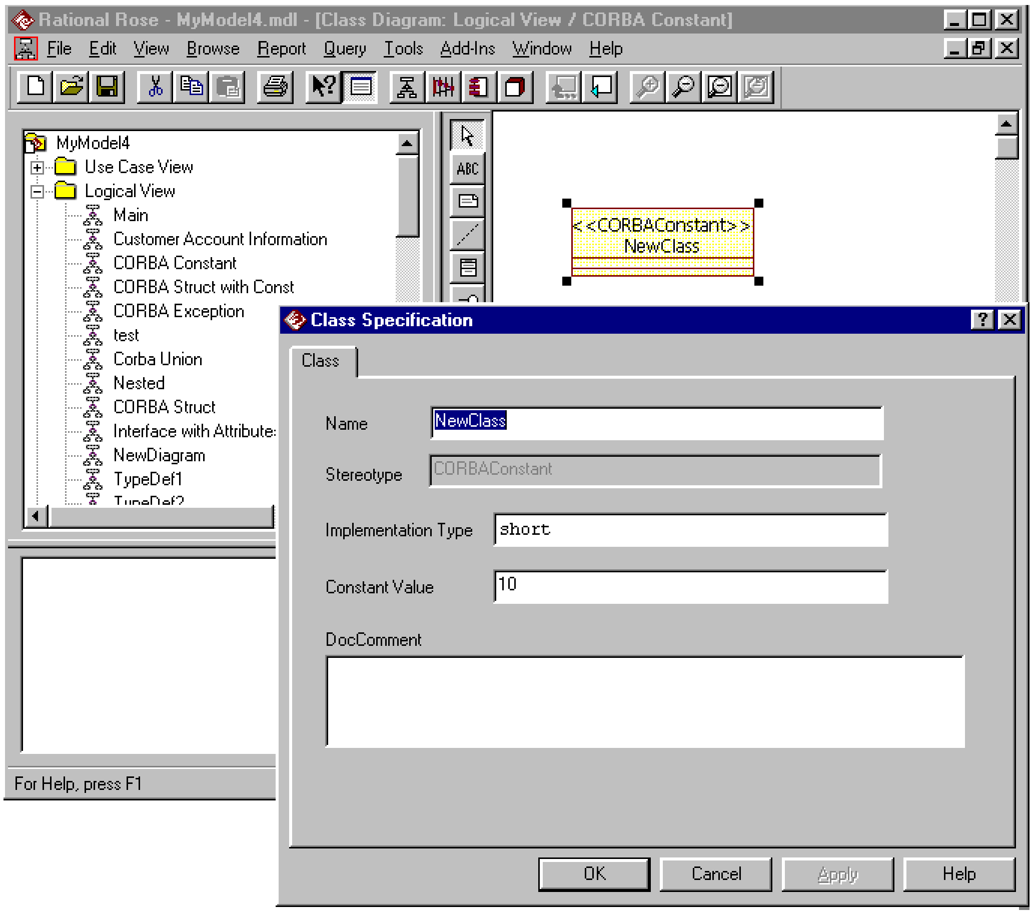
A CORBA Constant maps to a Rational Rose class with the stereotype **CORBAConstant**. You should set the Constant's ImplementationType property to a valid CORBA type before forward engineering. See CORBA IDL Types, in Chapter 2, for more information.

Conversely, if you reverse engineer CORBA IDL code that defines a Constant, Rational Rose CORBA adds the CORBAConstant class and sets its ImplementationType in your model.

Example - Unnested CORBA Constant

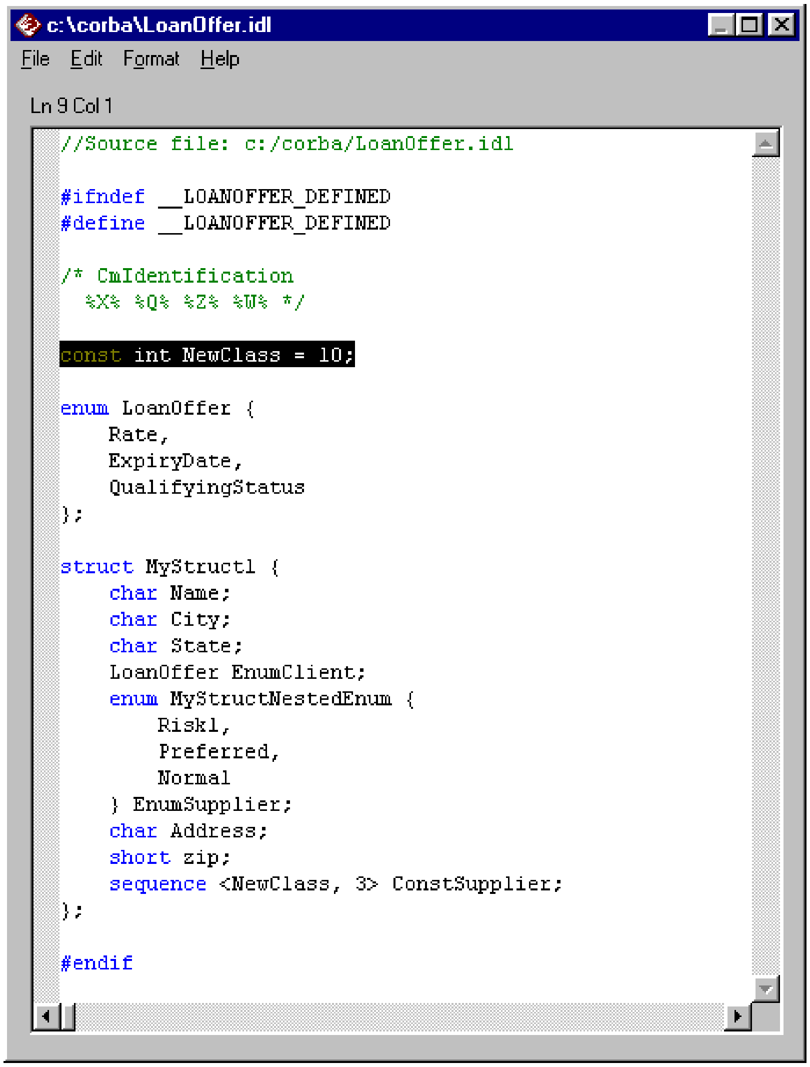
The following example shows a class called **NewClass**, which is a CORBA Constant. Its Implementation Type is **`short`** and its value is `10`.

Note: *For an example of a Constant nested in an Interface, see the Interface examples in Chapter 4.*



Name: NewClass
Stereotype: CORBAConstant
Implementation Type: short
Constant Value: 10
DocComment:

The CORBA IDL code that maps to the Constant shows a simple one-to-one correspondence to the Constant in the model. Because this class is assigned to a component called **LoanOffer**, the IDL code is generated in a file called **LoanOffer.idl**.



```
//Source file: c:/corba/LoanOffer.idl

#ifndef __LOANOFFER_DEFINED
#define __LOANOFFER_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

const int NewClass = 10;

enum LoanOffer {
    Rate,
    ExpiryDate,
    QualifyingStatus
};

struct MyStruct1 {
    char Name;
    char City;
    char State;
    LoanOffer EnumClient;
    enum MyStructNestedEnum {
        Risk1,
        Preferred,
        Normal
    } EnumSupplier;
    char Address;
    short zip;
    sequence <NewClass, 3> ConstSupplier;
};

#endif
```

CORBA Enumeration Mapping

A CORBA Enumeration maps to a Rational Rose class with the stereotype **CORBAEnum**.

In Rational Rose, you define the Enum's values as attributes of the class. You can specify the order of the enumeration by setting the Order property for each of these attributes. If you do not specify attribute order, Rose CORBA assigns the order to the enumeration according to the order in which you list the attributes in the Class specification.

When you forward engineer the Enum, Rational Rose CORBA generates the IDL code that defines an Enum class and uses the Rational Rose class's attributes as the Enum values.

RoseTip

Do not assign data types to CORBA Enum attributes. Data types have no meaning for Enums; if you specify them, Rational Rose CORBA ignores them during forward engineering.

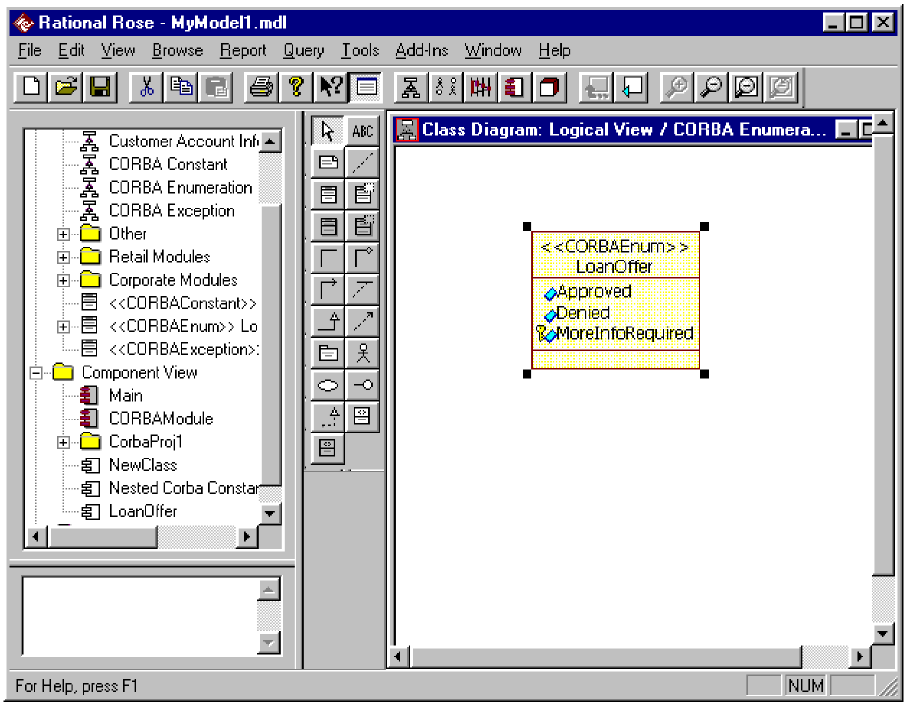
If you reverse engineer an Enum from CORBA IDL, Rational Rose CORBA adds a class with a stereotype of **CORBAEnum** to your model. The attributes of the class correspond to the Enum values in the IDL code.

Use the Enum's custom Class Specification to change the order of its attributes.

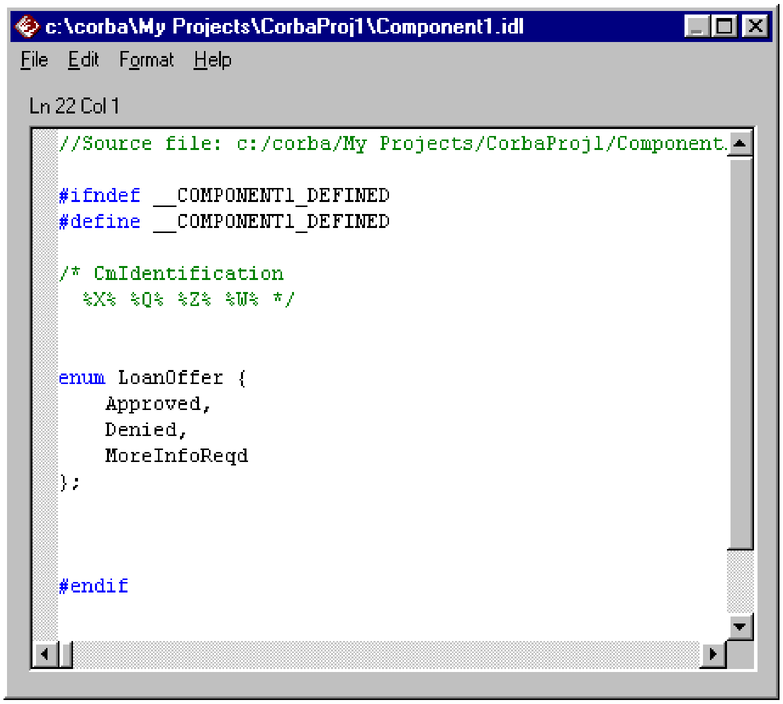
Example - CORBA Enum

The following simple example shows class **LoanOffer** defined as a CORBA Enum with three attributes:

- Approved
- Denied
- MoreInfoRequired



In this example, **LoanOffer** is assigned to **Component1**, so its IDL code is generated into **Component1.idl**.



```
//Source file: c:/corba/My Projects/CorbaProj1/Component.

#ifndef __COMPONENT1_DEFINED
#define __COMPONENT1_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */


enum LoanOffer {
    Approved,
    Denied,
    MoreInfoReqd
};



#endif
```

RoseTip

For an example of an Enum as the supplier in an association with an Interface, see the Interface examples in Chapter 4. For an example of an Enum as the supplier in an association with a Typedef, see the Typedef examples in Chapter 5.

Chapter 4

Mapping CORBA Interfaces with Rational Rose Classes

Overview

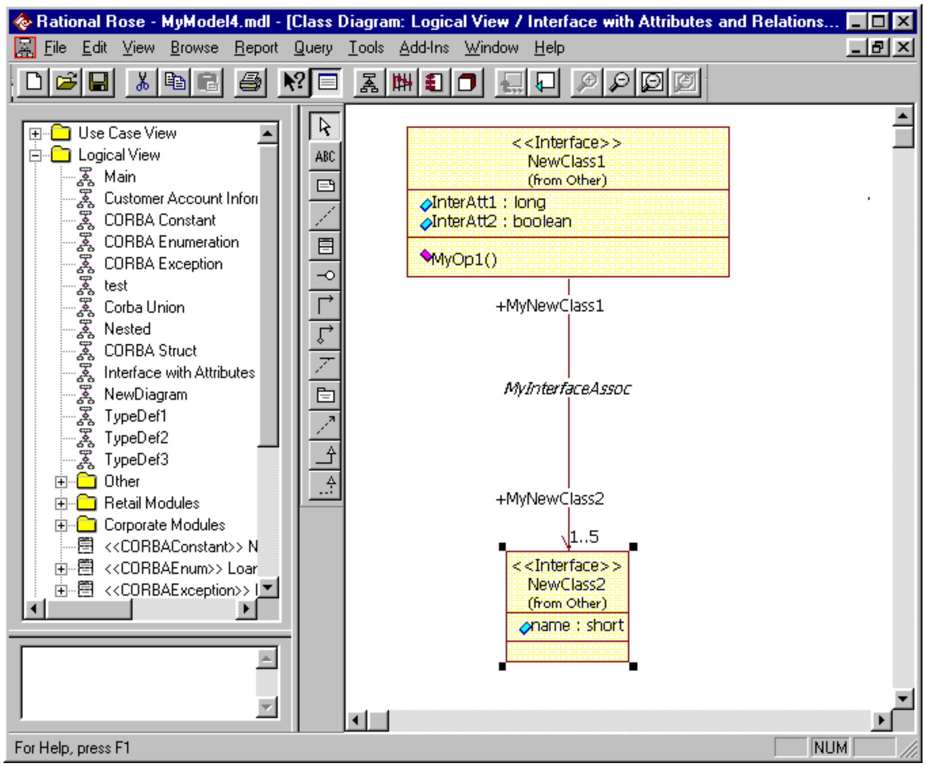
A CORBA Interface maps to a Rational Rose class with the stereotype **Interface**. Interfaces are the only CORBA classes that can have inheritance (generalize) relationships. They are also the only CORBA classes for which you can define operations.

Any CORBA IDL types you define as classes nested in an Interface class map to nested CORBA classes in the generated IDL code. (See Chapter 2 for information on CORBA IDL Types.)

Interface attributes and operations come from several sources:

- Attributes and operations you specify in the class specification using CORBA fundamental or template types map to CORBA attributes and operations, respectively.
- Relationships you define between the interface class and user-defined types in the model map to CORBA attributes. (This includes generalization relationships, which you can use to define inheritance between interfaces.)
- Other, special cases map to CORBA attributes.

In this model, a CORBA Interface called **NewClass1** has attributes that come both from fundamental types and from an association with another CORBA Interface called **NewClass2**.



Interface Attribute and Operation Mapping

Because the CORBA Interface attributes can take several forms, they map to various elements in a Rational Rose model:

- Rational Rose attributes map to CORBA fundamental and template types.
- Rational Rose relationship roles map to references to user-defined types.

In addition, there are multiple ways of creating the following special cases of CORBA Interface attribute definitions. Some methods are applicable for one-time forward engineering only.

- Single-dimension array of user-defined types
- Bounded sequence of user-defined types
- Unbounded sequence of user-defined types

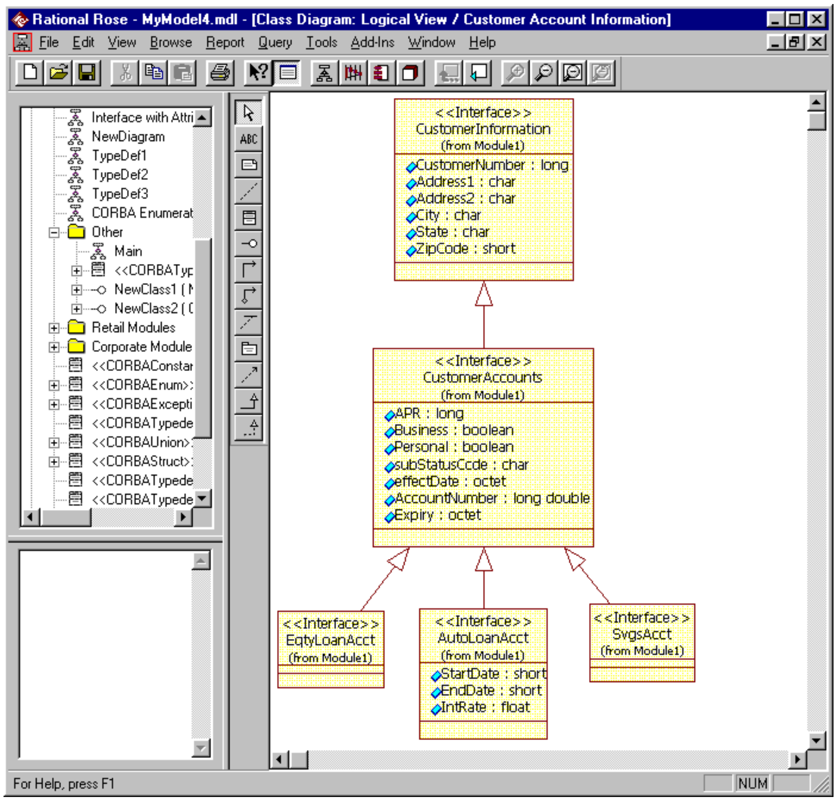
CORBA Interfaces are the only CORBA constructs that can have operations. There is a one to one mapping between each Interface operation in the model and each operation in CORBA IDL code.

See *CORBA Exception Mapping* in Chapter 6 for information on modeling operation exceptions.

The following subsections provide mapping information and examples of Interface inheritance, each form of Interface attribute, as well as Interface operations.

Interface Classes with Inheritance

Interface classes are the only CORBA classes that can inherit from one another. The following figure shows a set of Interfaces in an inheritance (generalizes) relationship.



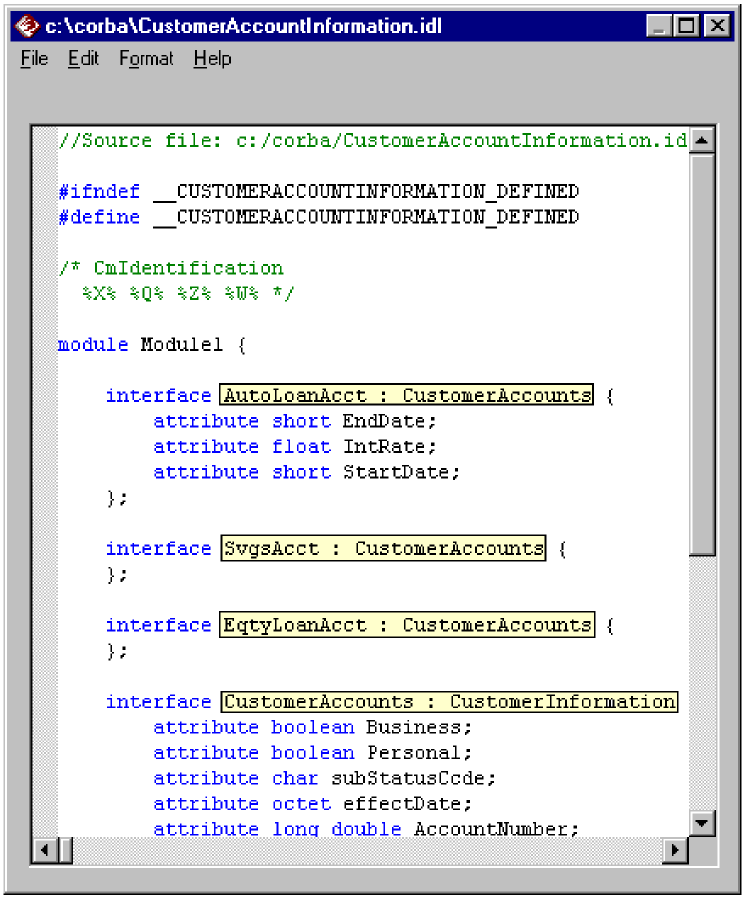
RoseTip

Instead of drawing inheritance relationships yourself, you can use CORBA custom Class Specifications.

1. Open each class's Custom Specification.
2. Use the tools in the **Inherits From** field to select or specify the class's inheritance and click **OK**.
3. In a class diagram, select one of the classes.
4. Select **Expand Selected Elements...** from the **Query** menu.

5. Follow the online help instructions to specify that you want to show inheritance (generalize) relationships in your diagram.

If you examine the IDL code that maps to these Interfaces, you will see that when one Interface inherits from another Interface, its name is followed by a colon (:) and the name of the class from which it inherits.



```
//Source file: c:/corba/CustomerAccountInformation.id

#ifndef __CUSTOMERACCOUNTINFORMATION_DEFINED
#define __CUSTOMERACCOUNTINFORMATION_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

module Module1 {

    interface AutoLoanAcct : CustomerAccounts {
        attribute short EndDate;
        attribute float IntRate;
        attribute short StartDate;
    };

    interface SvgsAcct : CustomerAccounts {
    };

    interface EqtyLoanAcct : CustomerAccounts {
    };

    interface CustomerAccounts : CustomerInformation
        attribute boolean Business;
        attribute boolean Personal;
        attribute char subStatusCode;
        attribute octet effectDate;
        attribute long double AccountNumber;
```

Attributes Defined as Fundamental and Template Types

An Interface attribute that is a CORBA fundamental type or template type is represented in a model as an attribute with the following properties:

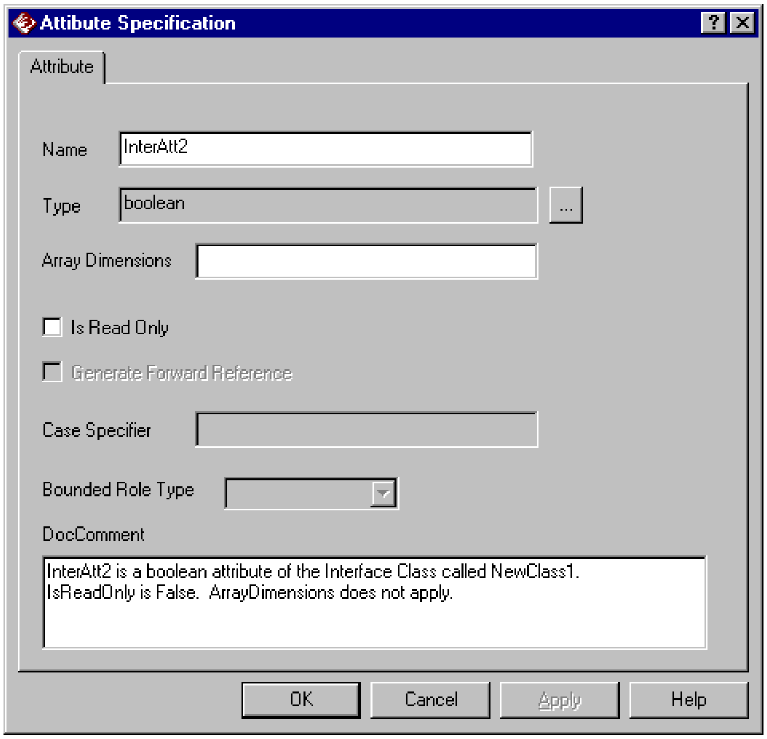
- **IsReadOnly**, a boolean property of the attribute, set to True or False
- **Type**, set to a CORBA fundamental type or template type.

For a complete list of fundamental and template types, refer to *CORBA IDL Types* in Chapter 2.

Example

The following sample custom CORBA Specification defines an attribute of a CORBA Interface as a fundamental type.

In this example the attribute type is **boolean** and the **IsReadOnly** property is set to False (not selected). (**ArrayDimensions** does not apply.)



Name: InterAtt2

Type: boolean

Array Dimensions:

☐ Is Read Only

☐ Generate Forward Reference

Case Specifier:

Bounded Role Type:

DocComment:

InterAtt2 is a boolean attribute of the Interface Class called NewClass1.
IsReadOnly is False. ArrayDimensions does not apply.

Attributes Defined as References to User-defined Types

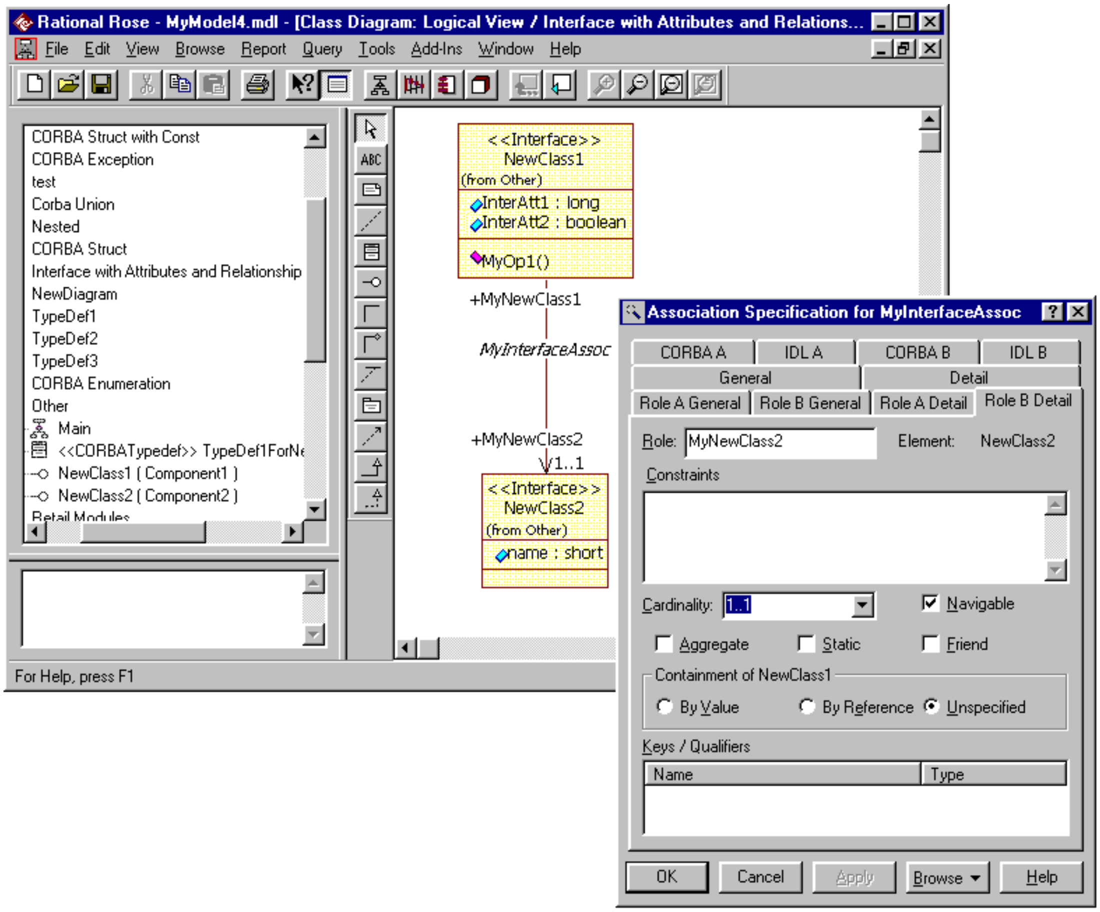
An Interface attribute that is a reference to a user-defined type is represented in a model as a relationship of cardinality 1. In this relationship, the Interface class is the client and the class which represents the user-defined type is the supplier. The classes in this relationship have the following properties:

- IsReadOnly is a boolean attribute of the supplier role.
- Type is the qualified name of the class representing the user-defined CORBA type (a CORBA Enum, for example).

Example

The following example shows an Interface (**NewClass1**) in an association relationship with another Interface (**NewClass2**). You can create this relationship between an Interface and any CORBA user-defined type.

The Association Specification shows that the relationship is of cardinality 1.



Role: MyNewClass2 Element: NewClass2

Cardinality: 1..1 [✓ Navigable]

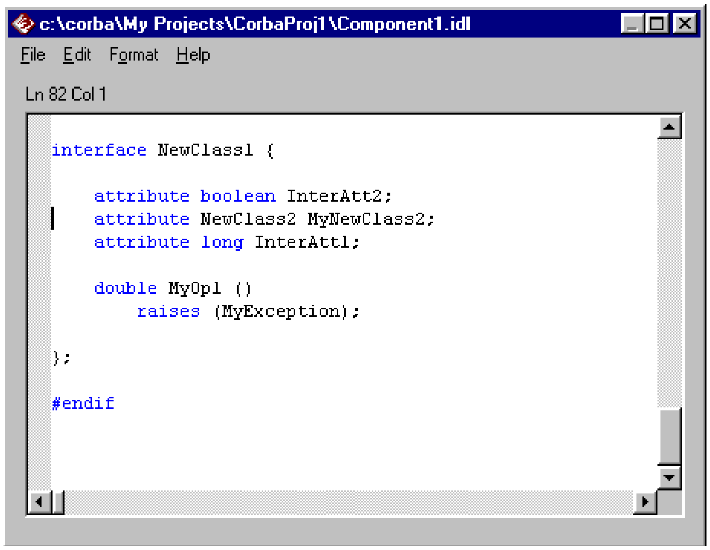
[ ] Aggregate [ ] Static [ ] Friend

Containment of NewClass1: By Value By Reference [✓ Unspecified]

The following figure shows the CORBA IDL that corresponds to the definition of NewClass1, including its relationship to NewClass2.

Notice that:

- NewClass1's attributes in the model (**InterAtt1** and **InterAtt2**) correspond to CORBA attribute definitions in the IDL file.
- NewClass1's operation corresponds to a CORBA Interface operation definition.
- There is another CORBA attribute whose name (called a *declarator* in CORBA) is the name of the supplier role of the relationship (**MyNewClass2**) and whose type is the fully qualified name of the supplier class (**NewClass2**).



```
interface NewClass1 {

    attribute boolean InterAtt2;
    attribute NewClass2 MyNewClass2;
    attribute long InterAtt1;

    double MyOp1 ()
        raises (MyException);

};

#endif
```

Attributes Defined as Single-Dimension Arrays of User-Defined Types

You can represent an Interface attribute that is a single-dimension array of user-defined types by:

- Creating a relationship of bounded cardinality between the Interface and a supplier class that represents the user-defined type
- Setting the supplier class's BoundedRoleType property to Array

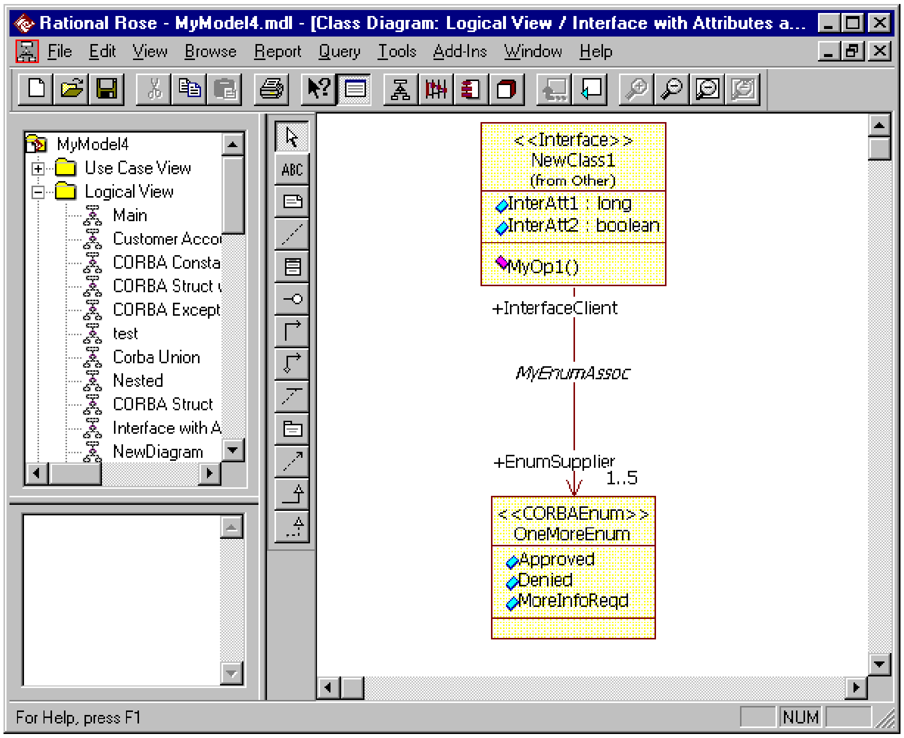
If you generate code from these elements and then reverse engineer the generated IDL code, Rational Rose CORBA creates a nested CORBA Typedef class. This class is the supplier in a relationship with the Interface class.

When you apply cardinality to a relationship, you are indicating the number of links allowed between one instance of a class and the instances of another class.   Bounded cardinality means that the number of instances allowed is greater than zero, but is not infinity.

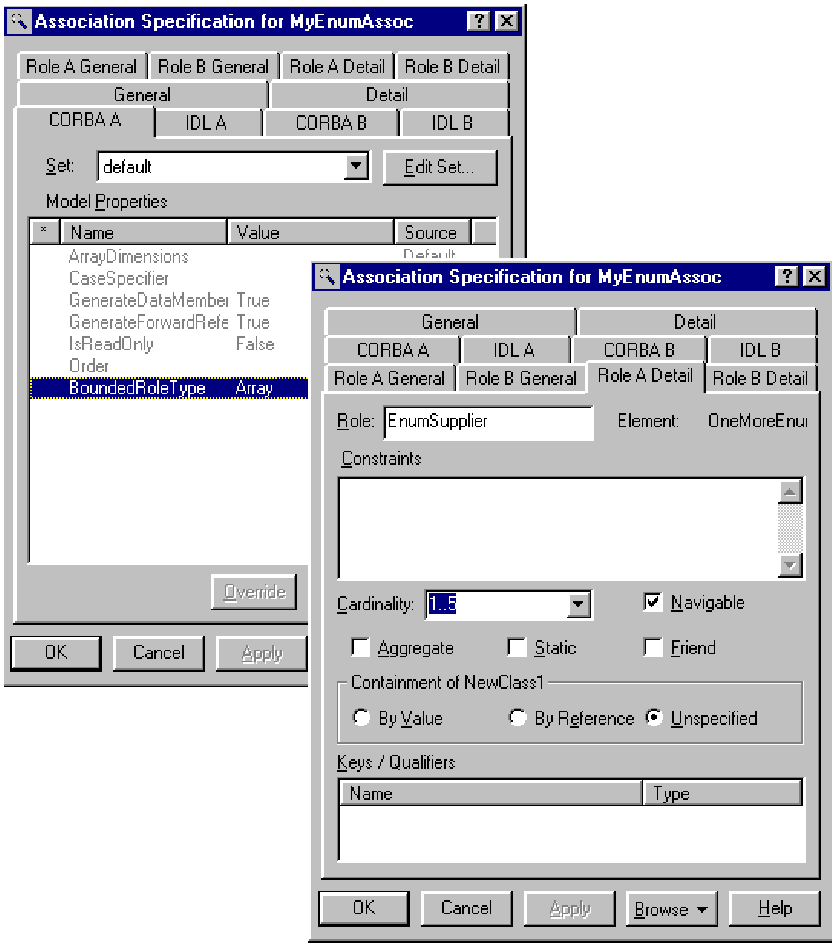
You can define the array as it will eventually be reverse-engineered, if you prefer. However, you may find it easier to define it using bounded cardinality and the BoundedRoleType property, as shown in the following example. Just remember that your reverse-engineered model will be correct, but, depending on your original definition, it may no longer match your original model.

Example

The following example is a special case of Interface attribute definition. It shows an Interface (**NewClass1**) in an association relationship with a single-dimension array of a user-defined type called **OneMoreEnum**. You can create this relationship between an Interface and any user-defined type.



The following figure shows the specifications for the supplier role in the association. Notice that the supplier's BoundedRoleType property is set to `Array` and its Cardinality is bounded.



Set: default

*	Name	Value	Source
	ArrayDimensions		Default
	CaseSpecifier		
	GenerateDataMember	True	
	GenerateForwardRefe	True	
	IsReadOnly	False	
	Order		
	BoundedRoleType	Array	

Role: EnumSupplier Element: OneMoreEnu

Cardinality: 1..5 [✓ Navigable]

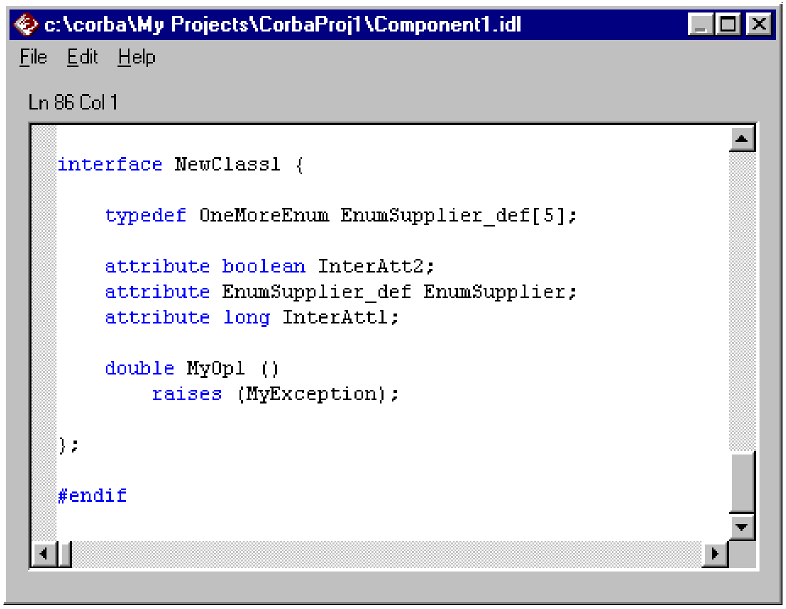
Aggregate Static Friend

By Value By Reference [✓ Unspecified]

Name	Type

Next, let's look at the CORBA IDL that corresponds to the definition of **NewClass1**, including its relationship to **OneMoreEnum**. Notice that:

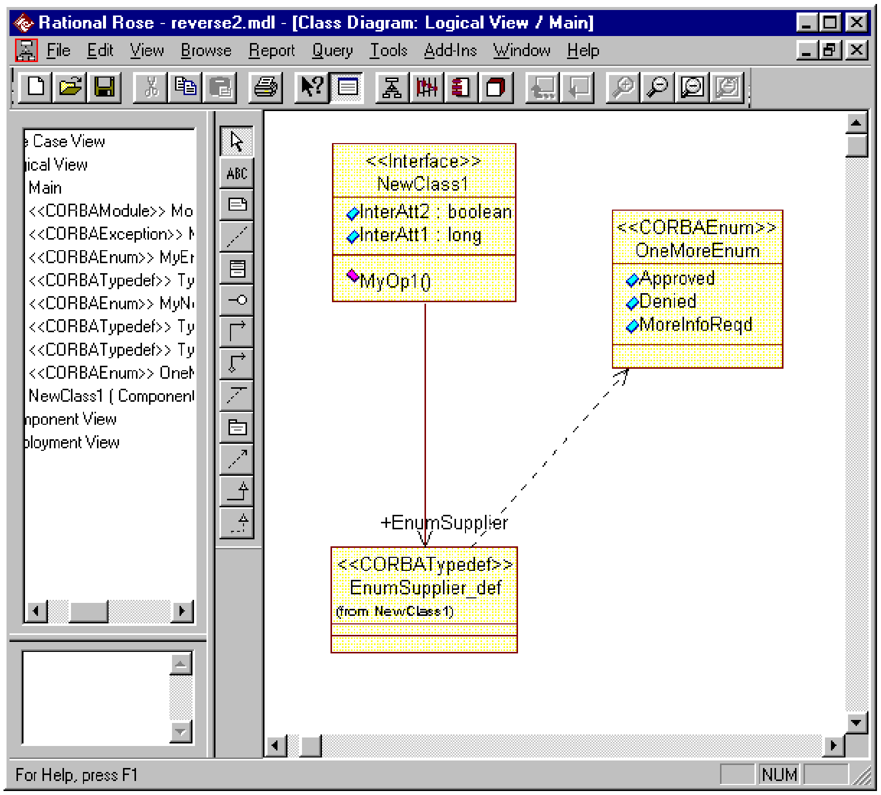
- NewClass1's attributes (**InterAtt1** and **InterAtt2**) in the model, correspond to CORBA attribute definitions in the IDL file. Also, its operation corresponds to a CORBA Interface operation definition.
- An IDL typedef statement defines the supplier class (**OneMoreEnum**) as a single-dimensioned array of type **EnumSupplier_def**, with a cardinality of 5. The name of the supplier role (**EnumSupplier**) becomes an attribute of Interface **NewClass1** and its type is the type of the supplier class (**EnumSupplier_def**).



```
interface NewClass1 {

    typedef OneMoreEnum EnumSupplier_def[5];

    attribute boolean InterAtt2;
    attribute EnumSupplier_def EnumSupplier;
    attribute long InterAtt1;

    double MyOp1 ()
        raises (MyException);

};

#endif
```

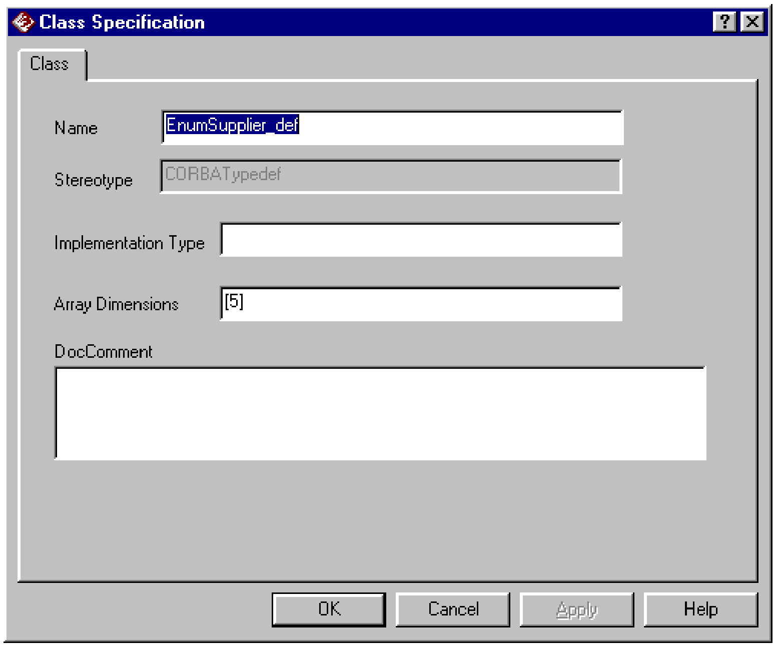
If you reverse engineer this special case, your new model will define:

- The supplier role as a nested Typedef of the Interface
- A dependency between the Typedef and the Enum



If you look in the Typedef's specification, you will see that the bounded cardinality you defined in the original model is now converted to the value of the ArrayDimensions property.

Remember, this new model no longer matches your original model.


Class Specification

Class

Name: EnumSupplier_def

Stereotype: CORBATypedef

Implementation Type:

Array Dimensions: [5]

DocComment:

Attributes Defined as a Bounded Sequence of User-Defined Types

You can represent an Interface attribute that is a bounded sequence of user-defined types by:

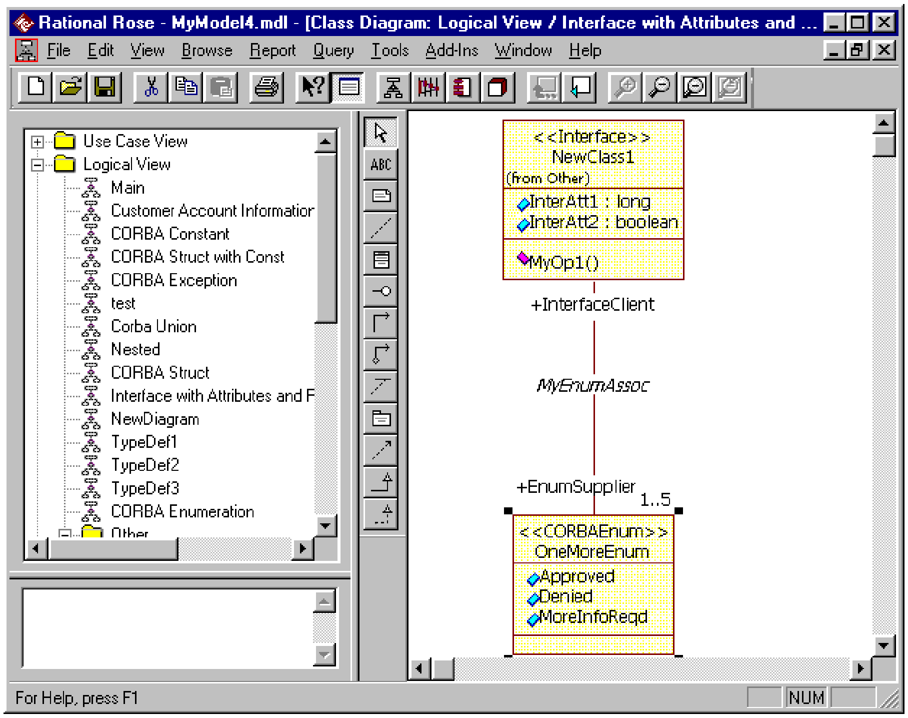
- Creating a relationship of bounded cardinality between the Interface and a supplier class that represents the user-defined type
- Setting the supplier class's BoundedRoleType property to Sequence

If you generate code from these elements and then reverse engineer the generated IDL code, Rational Rose CORBA creates a nested CORBA Typedef class. This class is the supplier in a relationship with the Interface class. Remember this will no longer match your original model.

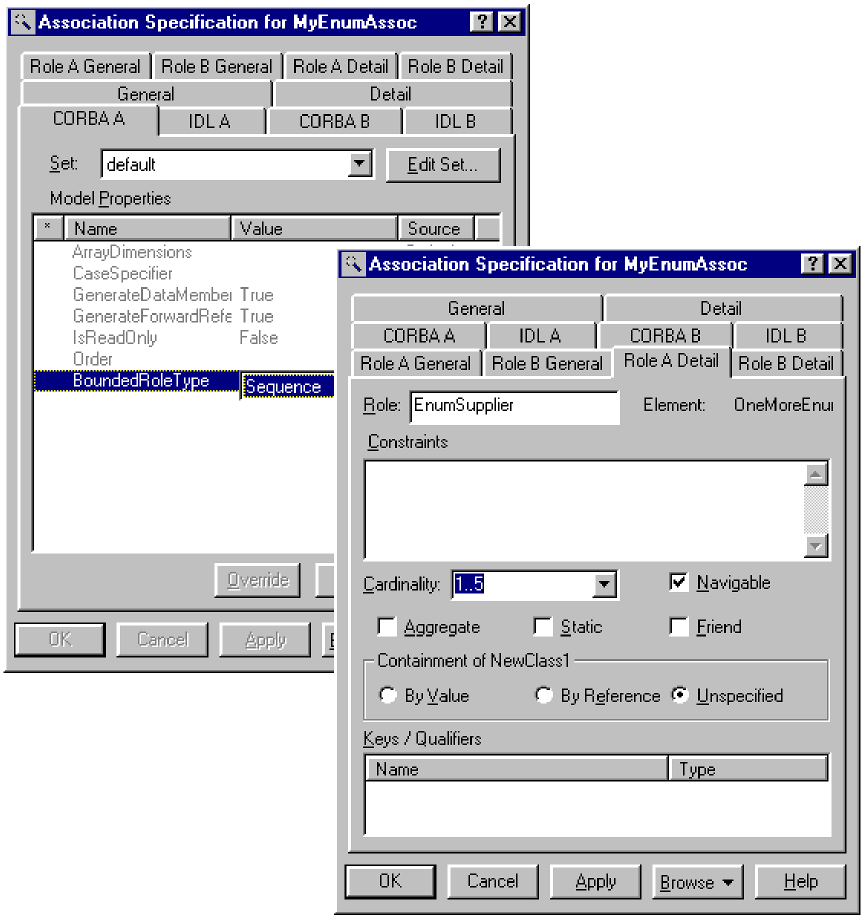
Example

The following example is a special case of Interface attribute definition. It shows an Interface (**NewClass1**) in an association relationship with a bounded sequence of a user-defined type called **OneMoreEnum**. You can create this relationship between an Interface and any user-defined type.

To create these model elements, you set the supplier's BoundedRoleType property to `Sequence`. Its cardinality must be bounded.



Here are the specifications for the supplier role in the association:



Association Specification for MyEnumAssoc

Set: default

Model Properties

*	Name	Value	Source
	ArrayDimensions		
	CaseSpecifier		
	GenerateDataMember	True	
	GenerateForwardRefe	True	
	IsReadOnly	False	
	Order		
	BoundedRoleType	Sequence	

Association Specification for MyEnumAssoc

Role: EnumSupplier

Element: OneMoreEnu

Cardinality: 1..5

[✓] Navigable

[ ] Aggregate [ ] Static [ ] Friend

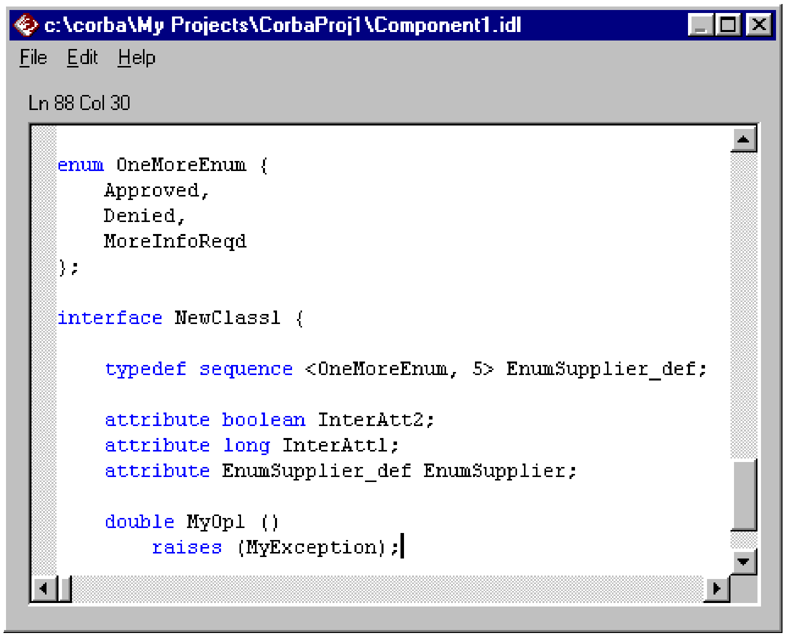
Containment of NewClass1: ( ) By Value ( ) By Reference (●) Unspecified

Keys / Qualifiers

Name	Type

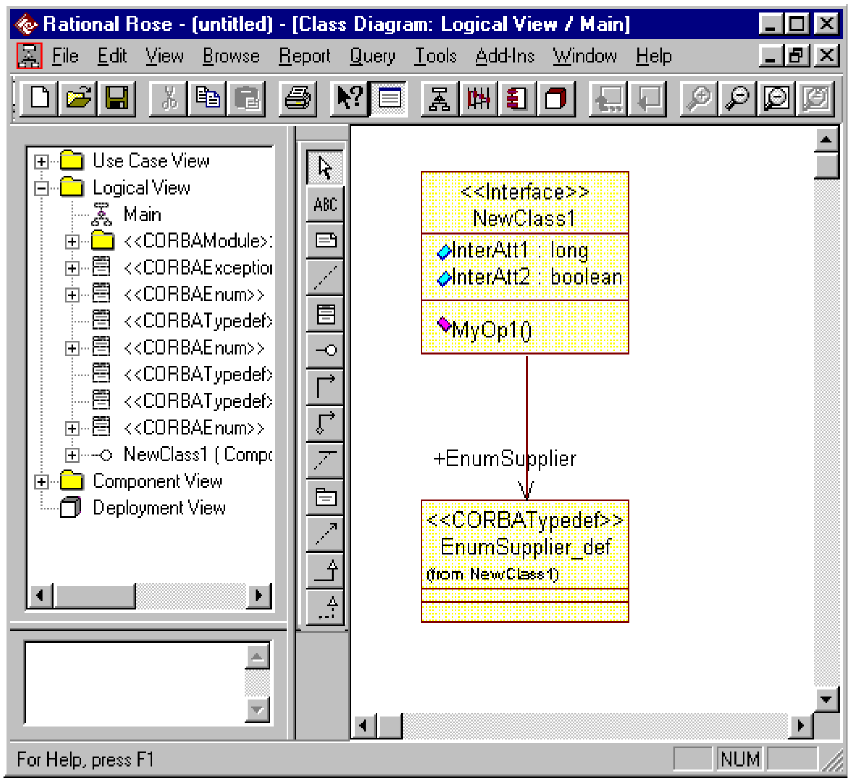
Now, look at the CORBA IDL that corresponds to the definition of NewClass1, including its relationship to OneMoreEnum, as well as the IDL definition for the Enum itself. Notice that:

- NewClass1's attributes (**InterAtt1** and **InterAtt2**) in the model, correspond to CORBA attribute definitions in the IDL file. Also, its operation corresponds to a CORBA Interface operation definition.
- An IDL typedef statement defines the supplier class (**OneMoreEnum**) as a bounded sequence of type **EnumSupplier_def**, with a dimension of **5** (to match the cardinality you defined in the specification).
- The name of the supplier role (**EnumSupplier**) becomes an attribute of Interface **NewClass1** and its type is the type of the supplier class (**EnumSupplier_def**).



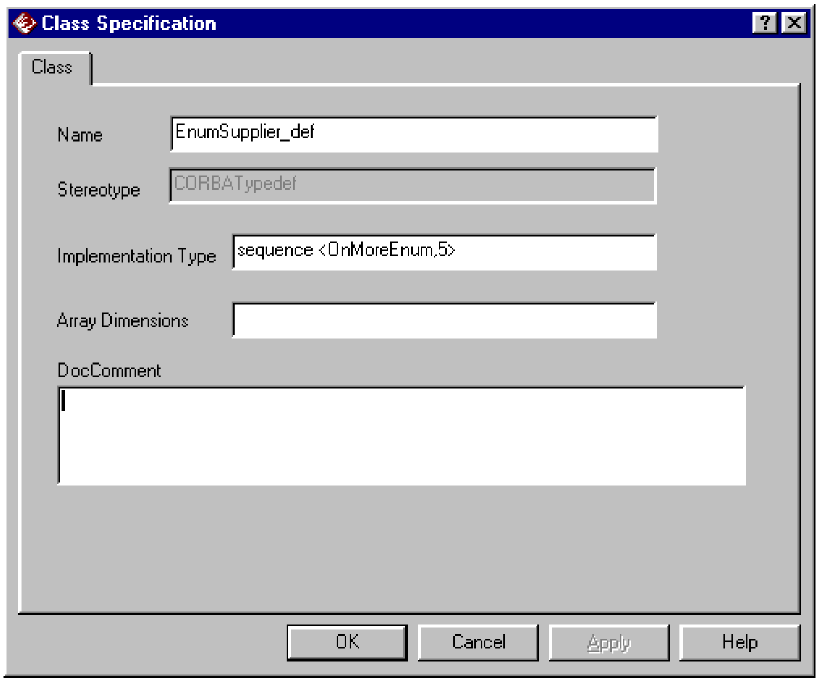
```
enum OneMoreEnum {
    Approved,
    Denied,
    MoreInfoReqd
};

interface NewClass1 {

    typedef sequence <OneMoreEnum, 5> EnumSupplier_def;

    attribute boolean InterAtt2;
    attribute long InterAtt1;
    attribute EnumSupplier_def EnumSupplier;

    double MyOp1 ()
        raises (MyException);
```

If you reverse engineer this special case, your new model will define the supplier role as a nested Typedef of the Interface.



If you look in the Typedef's specification, you will see that the ImplementationType property has been set to the name of the Enum and includes a dimension that matches the bounded cardinality you originally defined.

Remember, this new model no longer matches your original model.



Name: EnumSupplier_def
Stereotype: CORBATypedef
Implementation Type: sequence <OnMoreEnum,5>
Array Dimensions:
DocComment:

Attributes Defined as Unbounded Sequence of User-defined Types

You can represent an Interface attribute that is an unbounded sequence of user-defined types by:

- Creating a relationship of unbounded cardinality between the Interface and a supplier class that represents the user-defined type
- Setting the supplier class's BoundedRoleType property to `Sequence`

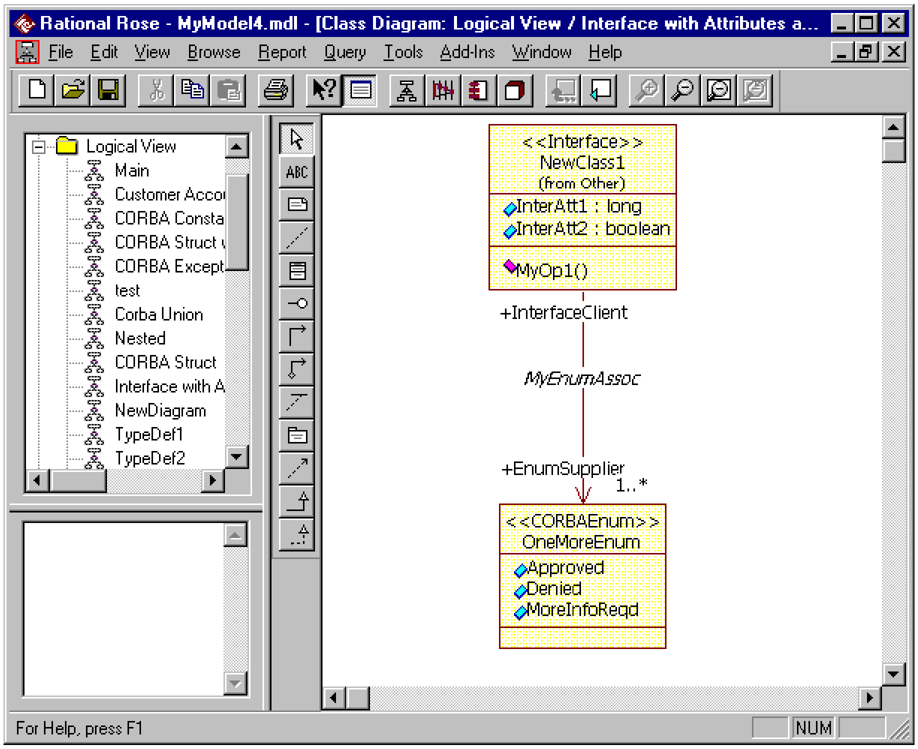
If you generate code from these elements and then reverse engineer the generated IDL code, Rational Rose CORBA creates a nested CORBA Typedef class. This class is the supplier in a relationship with the Interface class.

Remember this will no longer match your original model.

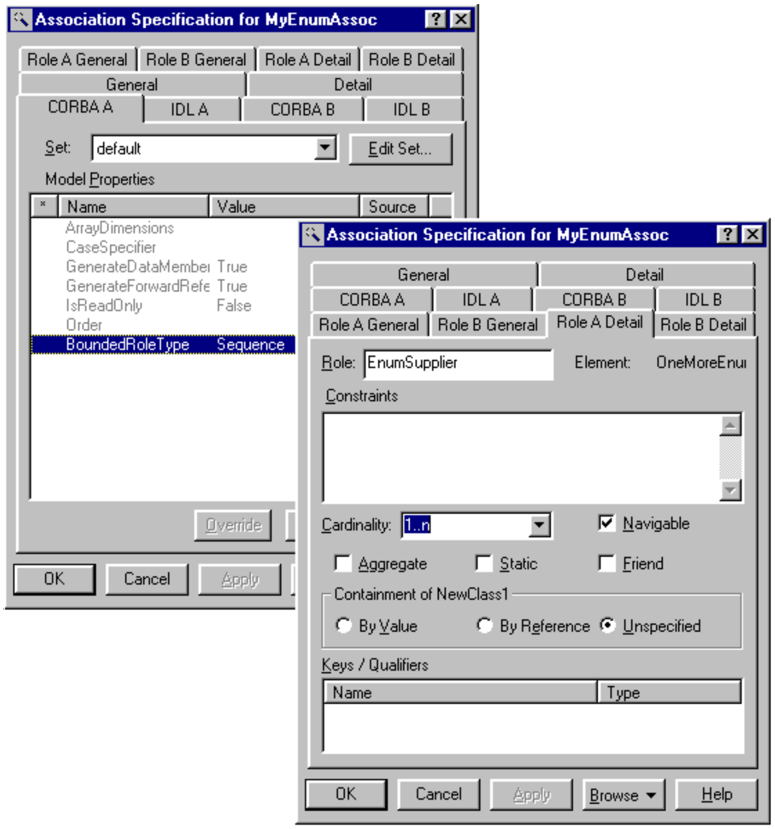
Example

The following example is a special case of Interface attribute definition. It shows an Interface (**NewClass1**) in an association relationship with an unbounded sequence of a user-defined type (**OneMoreEnum**). You can create this relationship between an Interface and any user-defined type.

To create these model elements, you set the supplier's BoundedRoleType property to Sequence. Its cardinality must be unbounded.



Here are the specifications for the supplier role in the association.



Set: default

Model Properties

*	Name	Value	Source
	ArrayDimensions		
	CaseSpecifier		
	GenerateDataMember	True	
	GenerateForwardRefe	True	
	IsReadOnly	False	
	Order		
	BoundedRoleType	Sequence	

Role: EnumSupplier Element: OneMoreEnu

Constraints

Cardinality: 1..n [✓ Navigable]

[ ] Aggregate [ ] Static [ ] Friend

Containment of NewClass1

By Value By Reference [✓ Unspecified]

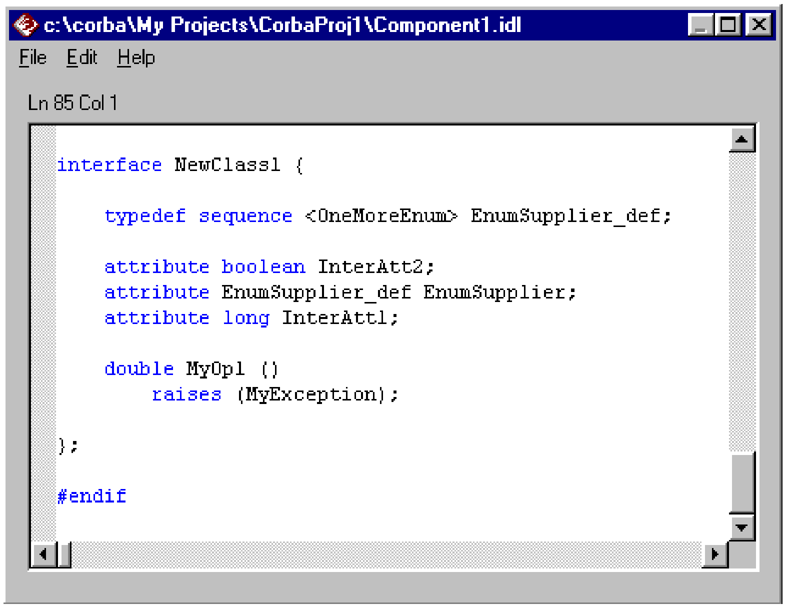
Keys / Qualifiers

Name	Type

The following figure shows the CORBA IDL that corresponds to the definition of NewClass1, including its relationship to OneMoreEnum.

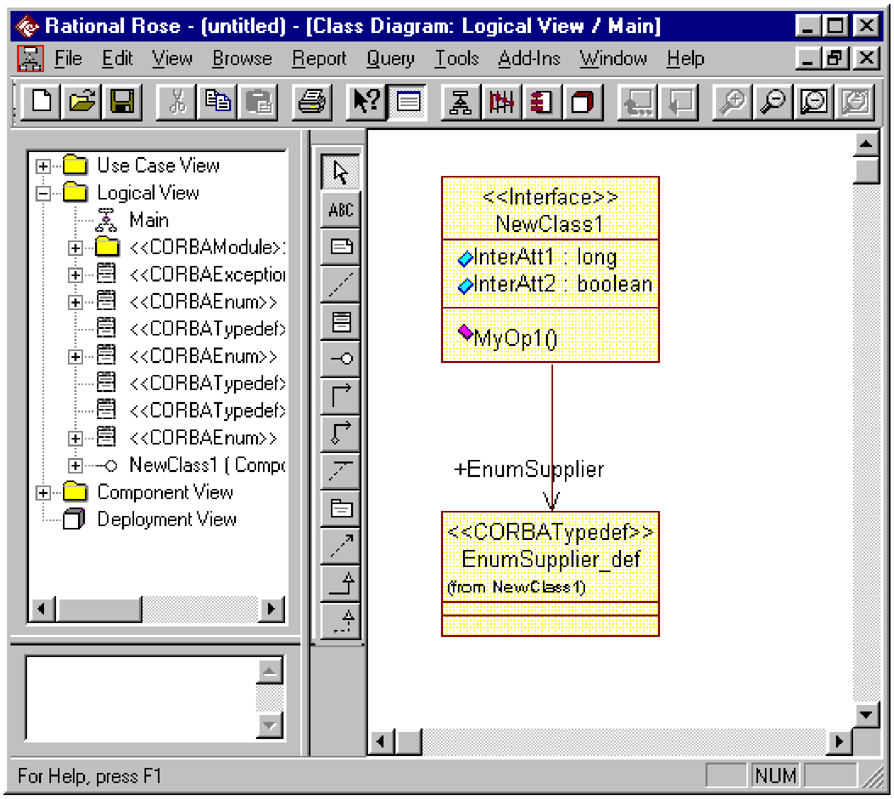
Notice that NewClass1's attributes (**InterAtt1** and **InterAtt2**) in the model, correspond to CORBA attribute definitions in the IDL file. Also, its operation corresponds to a CORBA Interface operation definition.

In addition, an IDL typedef statement defines the supplier class (**OneMoreEnum**) as an unbounded sequence of type **EnumSupplier_def**. The name of the supplier role (**EnumSupplier**) becomes an attribute of Interface NewClass1 and its type is the type of the supplier class (**EnumSupplier_def**).

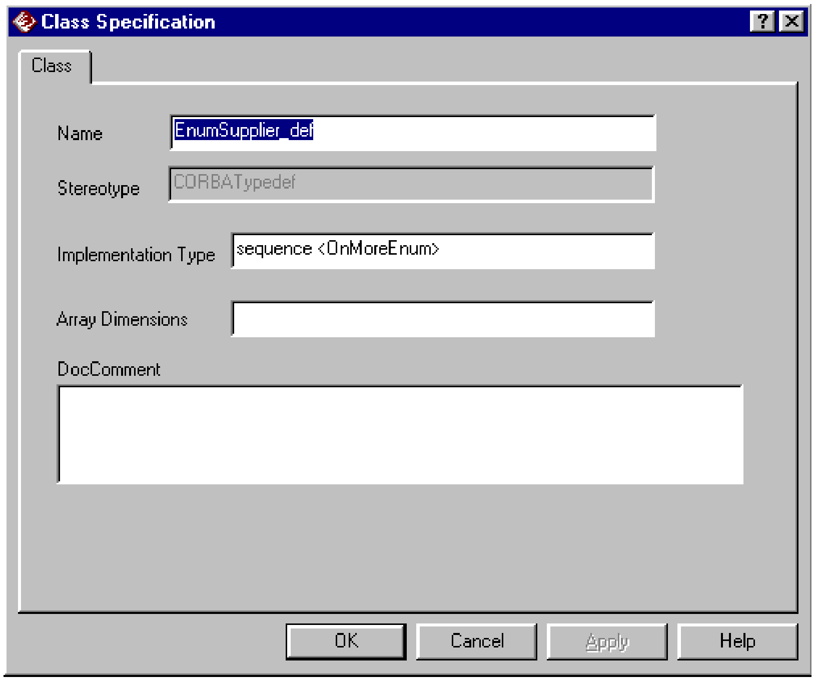


```
interface NewClass1 {

    typedef sequence <OneMoreEnum> EnumSupplier_def;

    attribute boolean InterAtt2;
    attribute EnumSupplier_def EnumSupplier;
    attribute long InterAtt1;

    double MyOp1 ()
        raises (MyException);

};

#endif
```

If you reverse engineer this special case, your new model will define the supplier role as a nested Typedef of the Interface.



If you look in the Typedef's specification you will see that the ImplementationType property has been set to a sequence whose name is the name of the Enum (**OneMoreEnum**). No Enum dimension is included because you reverse engineered an unbounded sequence. Remember, this new model no longer matches your original model.


Class Specification

Name: EnumSupplier_def
Stereotype: CORBATypedef
Implementation Type: sequence <OnMoreEnum>
Array Dimensions:
DocComment:

Chapter 5

Mapping CORBA Typedefs with Rational Rose Classes

Overview

A CORBA Typedef maps to a Rational Rose class with the stereotype **CORBATypedef**. The Typedef may have a dependency to a class that represents an IDL type.

The purpose of a Typedef is to create a synonym for another CORBA IDL element. Among other uses, this provides a way to define a sequence as an attribute of a CORBA Interface, something which you could not otherwise do.

A Typedef can define a single element or an array of elements (that is, an exception, struct, or union).

- If the Typedef defines a single element, you do not need to specify the ArrayDimensions property.
- If the Typedef defines an array, you specify a string of one or more array dimensions that apply to the Typedef class.

Ways to Model a TypeDef

There are two ways to model a CORBA TypeDef:

- Using the TypeDef's Class Specification

  Create a class with the **CORBATypeDef** stereotype. Open the class's custom specification and set the Implementation Type to a CORBA type. This TypeDef can then be used as the supplier in an association relationship with another CORBA class. In such cases, the TypeDef becomes an attribute (by reference) of the client class.
- As a client in an association relationship to another CORBA class

  Create a class with the **CORBATypeDef** stereotype, but do not set its Implementation Type. Then, create an association relationship in which the TypeDef is the client and another CORBA class is the supplier. The supplier class provides the attributes of the TypeDef by reference.

RoseTips

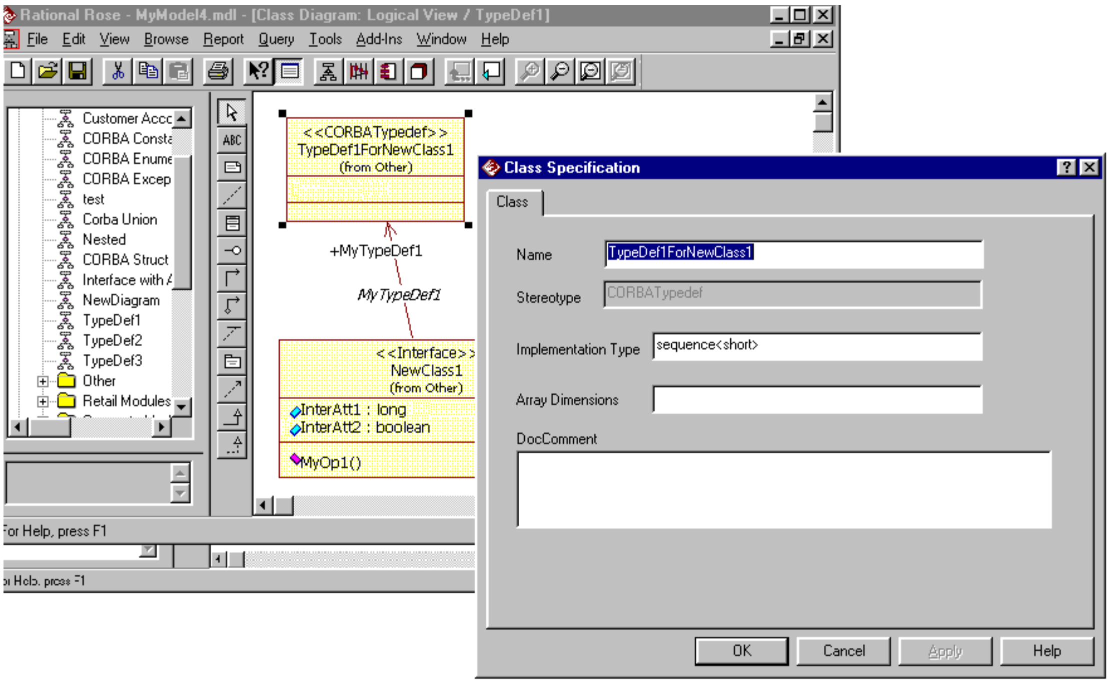
Do not directly specify attributes for a CORBA TypeDef, either in a class diagram or in a standard class specification. Standard Rose attributes will not generate correct CORBA IDL code. The TypeDef will get its CORBA attributes in one of the ways listed above; that is, from Implementation Type or from an association with a CORBA class.

Even though IDL grammar supports multiple declarators for a single Typedef, Rational Rose Corba mapping functions do not. If you reverse engineer IDL code that contains a Typedef with multiple declarators, Rational Rose CORBA creates separate Typedefs for each declarator. Conversely, if you model a Typedef with multiple declarators, you will get an error during code generation.

The following subsections provide mapping information and examples for the various forms of Typedef attributes.

Example 1 - TypeDef Mapping Using Implementation Type

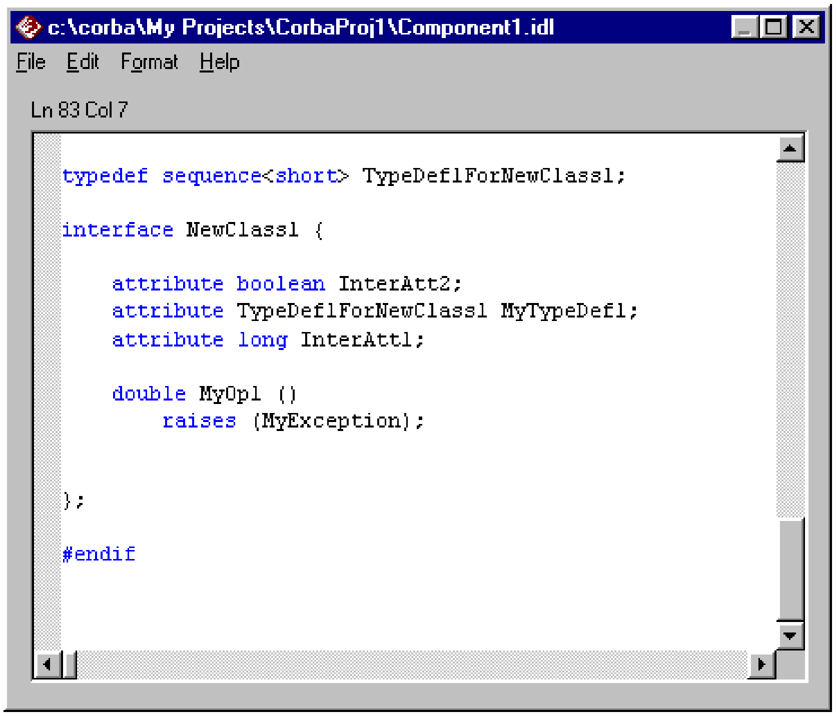
The following example shows how to map a typedef using a CORBA ImplementationType of `sequence<short>`. The typedef is then used in the supplier role of an association with a CORBA interface called NewClass1.



Name: TypeDef1ForNewClass1
Stereotype: CORBATypedef
Implementation Type: sequence<short>
Array Dimensions:
DocComment:

The following code fragment shows how these model elements map to CORBA IDL:

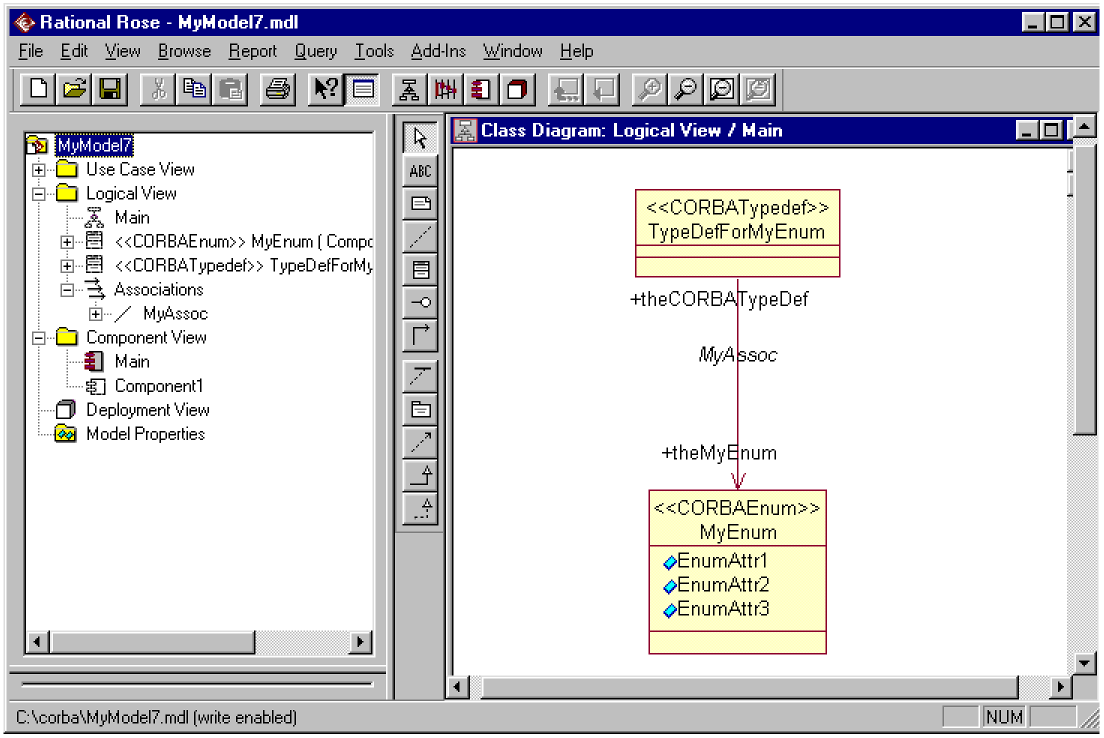
- TypeDef1ForNewClass1 maps directly to a typedef statement. Its characteristics come from the ImplementationType property in the class specification.
- The interface called NewClass1 is defined with attributes that come from:
    - The NewClass1 class specification
    - A reference to TypeDef1ForNewClass1, which comes from the reference to the association (MyTypeDef1) between the Typedef and the Interface



```
typedef sequence<short> TypeDef1ForNewClass1;

interface NewClass1 {

    attribute boolean InterAtt2;
    attribute TypeDef1ForNewClass1 MyTypeDef1;
    attribute long InterAtt1;

    double MyOp1 ()
        raises (MyException);


};

#endif
```

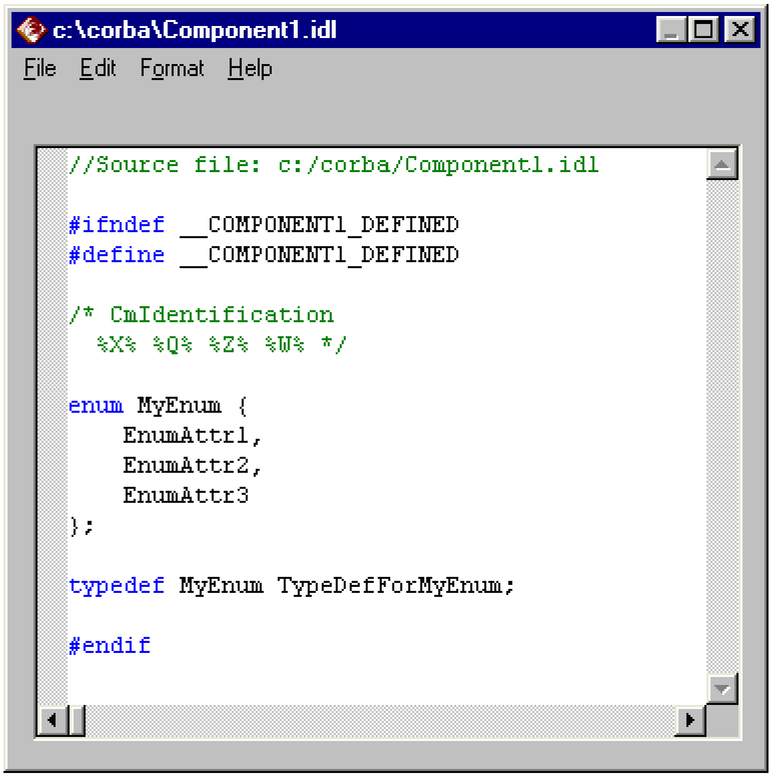
Example 2 - TypeDef Mapping Using an Association to an Unnested User-Defined Type

This example shows how to create a CORBA typedef whose attributes come from an association to unnested user-defined type, in this case a CORBAEnum.



The following code fragment shows how these model elements map to CORBA IDL:

Because the relationship is between TypeDefForMyEnum (client) and MyEnum (an unnested user-defined type in the supplier role), a reference to the enum becomes the attribute of the typedef.



```
//Source file: c:/corba/Component1.idl

#ifndef __COMPONENT1_DEFINED
#define __COMPONENT1_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

enum MyEnum {
    EnumAttr1,
    EnumAttr2,
    EnumAttr3
};

typedef MyEnum TypeDefForMyEnum;

#endif
```

Chapter 6

Mapping CORBA Exceptions, Structs, and Unions with Rational Rose Classes

Overview

The elements of CORBA Exceptions, Structs, and Unions can take several forms, and therefore map to various elements in a Rational Rose model:

- Rational Rose attributes map to CORBA fundamental and template types
- Rational Rose relationship roles map to one of the following:
    - References to user-defined types
    - In-Line definitions of user-defined types
    - Single- and multi-dimension arrays of user-defined types

In addition, there are multiple ways of creating the following special cases of CORBA element definitions. Some methods are applicable for one-time forward engineering only. For details, see the specific information on the following special cases:

- Single-Dimension array of user-defined types
- Bounded sequence of user-defined types
- Unbounded sequence of user-defined types

The first part of this Chapter presents simple mapping examples for an Exception, a Struct, and a Union. The remainder of the Chapter describes and provides examples for the more complex relationships that apply to all three CORBA types.

CORBA Exception Mapping

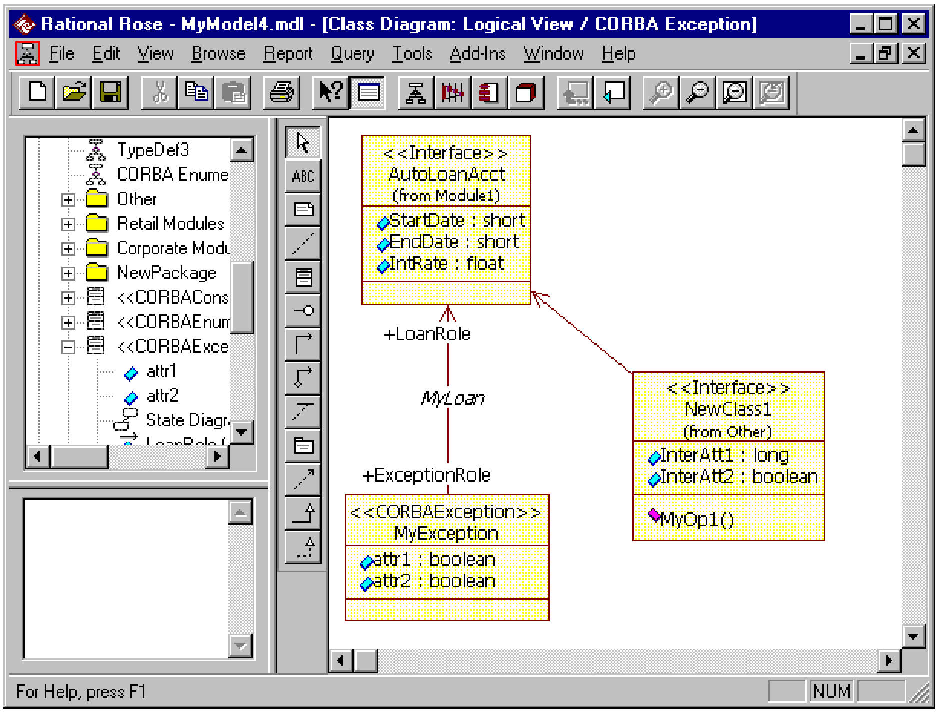
A CORBA Exception maps to a Rational Rose class with the stereotype **CORBAException**. The exception elements map to the class's attributes and relationships.

Rational Rose CORBA generates the attributes of the exception from a combination of the class's attributes, plus any user-defined types that have a relationship with the class.

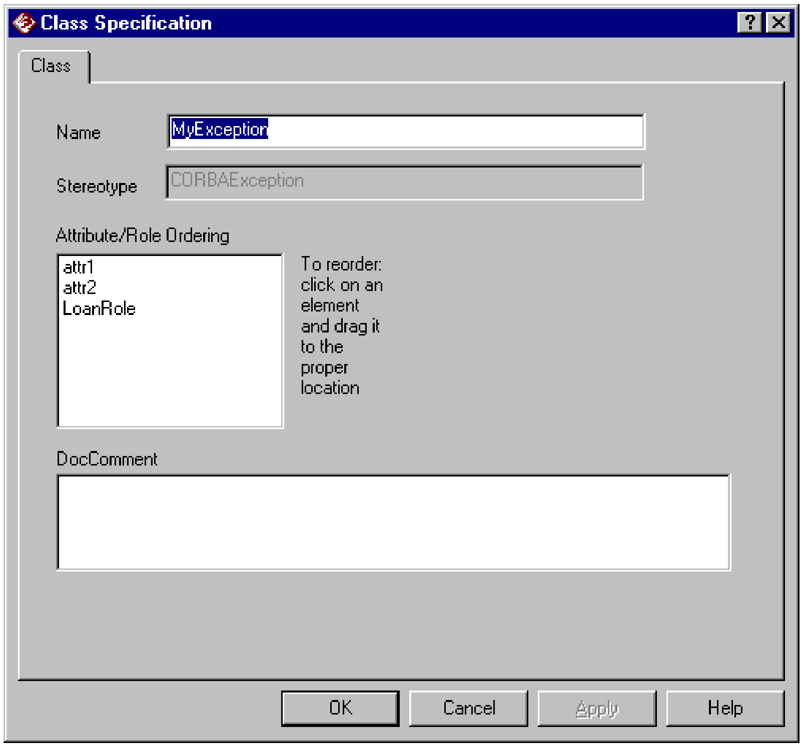
You generate the IDL code that raises the exception by setting the Raises property on an operation of an Interface and forward engineering the interface.

Example

The following example models a CORBA Exception whose attributes will come directly from its class definition, as well as from the association it has with an Interface. It also shows the Interface whose operation is defined to raise the Exception.



Here is the custom Class Specification for the Exception. Note that its attributes include the supplier role in the association relationship between **MyException** and **AutoLoanAcct**.


Class Specification

Class

Name: MyException

Stereotype: CORBAException

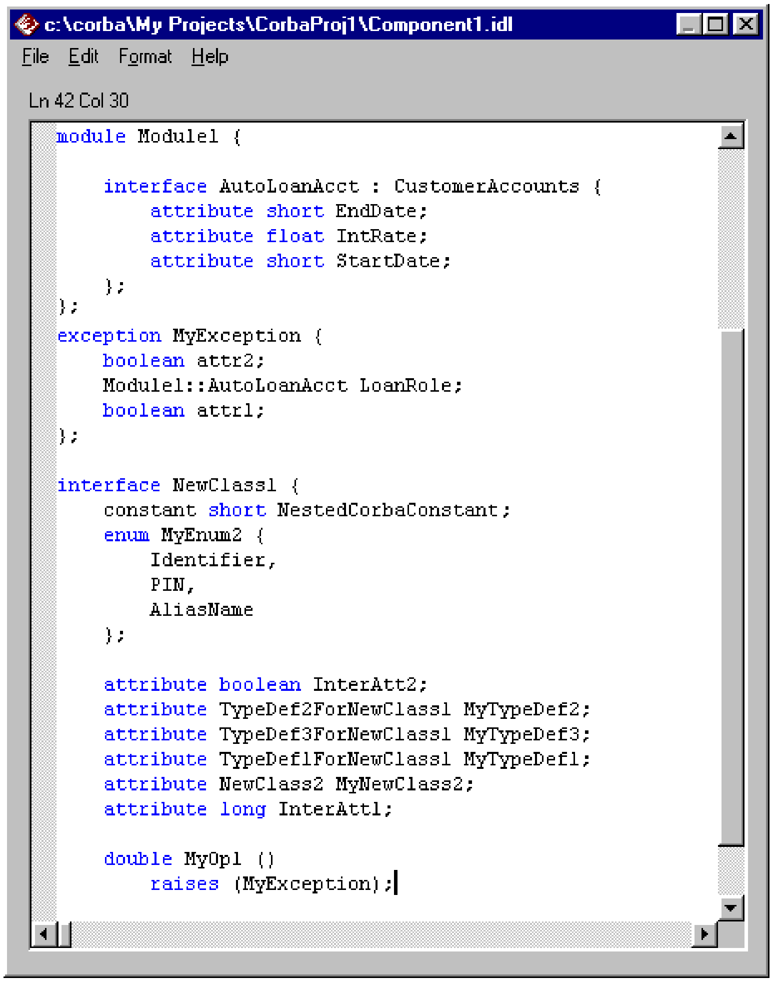
Attribute/Role Ordering

attr1
attr2
LoanRole

To reorder: click on an element and drag it to the proper location

DocComment

The following figure shows a fragment of CORBA IDL that corresponds to elements defined in the model. Notice that the attributes of MyException include its defined attributes, as well as a reference to the AutoLoanAcct interface class. The reference to AutoLoanAcct comes from the association between it and MyException. It also includes the definition of MyOp1, which (in NewClass1) raises MyException.



```
module Module1 {

    interface AutoLoanAcct : CustomerAccounts {
        attribute short EndDate;
        attribute float IntRate;
        attribute short StartDate;
    };
};
exception MyException {
    boolean attr2;
    Module1::AutoLoanAcct LoanRole;
    boolean attr1;
};

interface NewClass1 {
    constant short NestedCorbaConstant;
    enum MyEnum2 {
        Identifier,
        PIN,
        AliasName
    };

    attribute boolean InterAtt2;
    attribute TypeDef2ForNewClass1 MyTypeDef2;
    attribute TypeDef3ForNewClass1 MyTypeDef3;
    attribute TypeDef1ForNewClass1 MyTypeDef1;
    attribute NewClass2 MyNewClass2;
    attribute long InterAtt1;

    double MyOp1 ()
        raises (MyException);
```

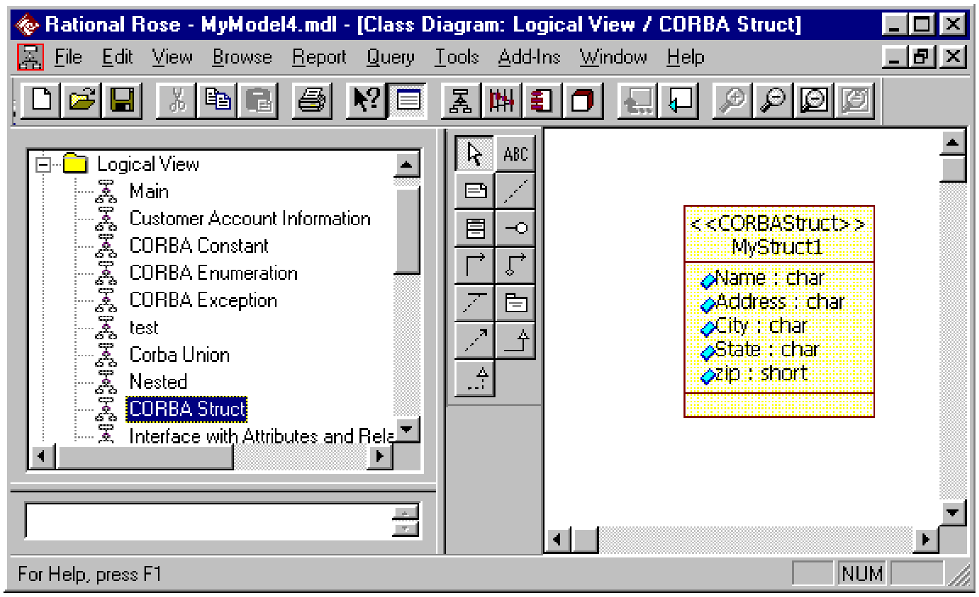
CORBA Struct Mapping

A CORBA Struct maps to a Rational Rose class with the stereotype **CORBAStruct**. The data members of the Struct map to the class's attributes and relationships.

Rational Rose CORBA generates the attributes of the Struct from a combination of the class's attributes and any user-defined types that have a relationship with the class.

Example

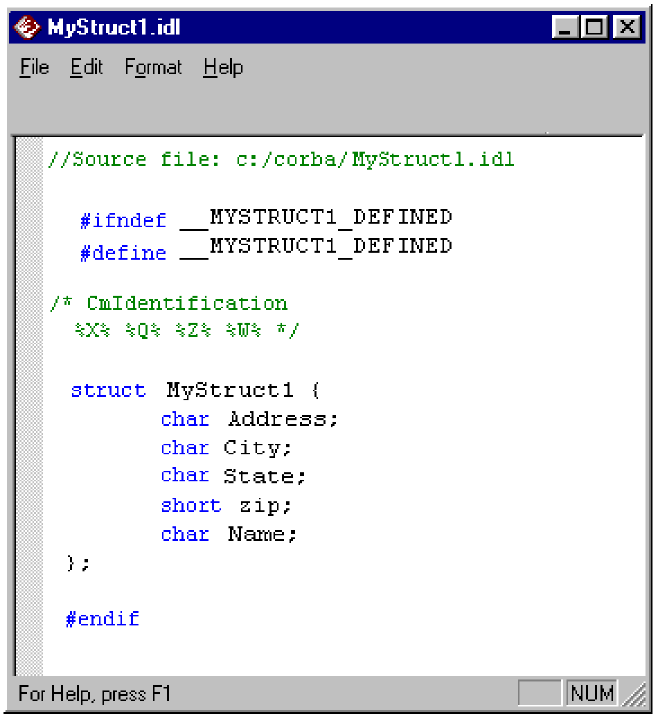
The following example defines **MyStruct1**, a simple CORBA Struct that holds some basic identification information:


Rational Rose - MyModel4.mdl - [Class Diagram: Logical View / CORBA Struct]

<<CORBAStruct>>
MyStruct1
Name : char
Address : char
City : char
State : char
zip : short

For Help, press F1

Here is the CORBA IDL code that maps to **MyStruct1**:



```
//Source file: c:/corba/MyStruct1.idl

  #ifndef __MYSTRUCT1_DEFINED
  #define __MYSTRUCT1_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

 struct MyStruct1 {
        char Address;
        char City;
        char State;
        short zip;
        char Name;
 };

 #endif
```

CORBA Union Mapping

A CORBA Union is a cross between a C language union and a switch statement. A CORBA Union maps to a Rational Rose class with the stereotype **CORBAUnion**.

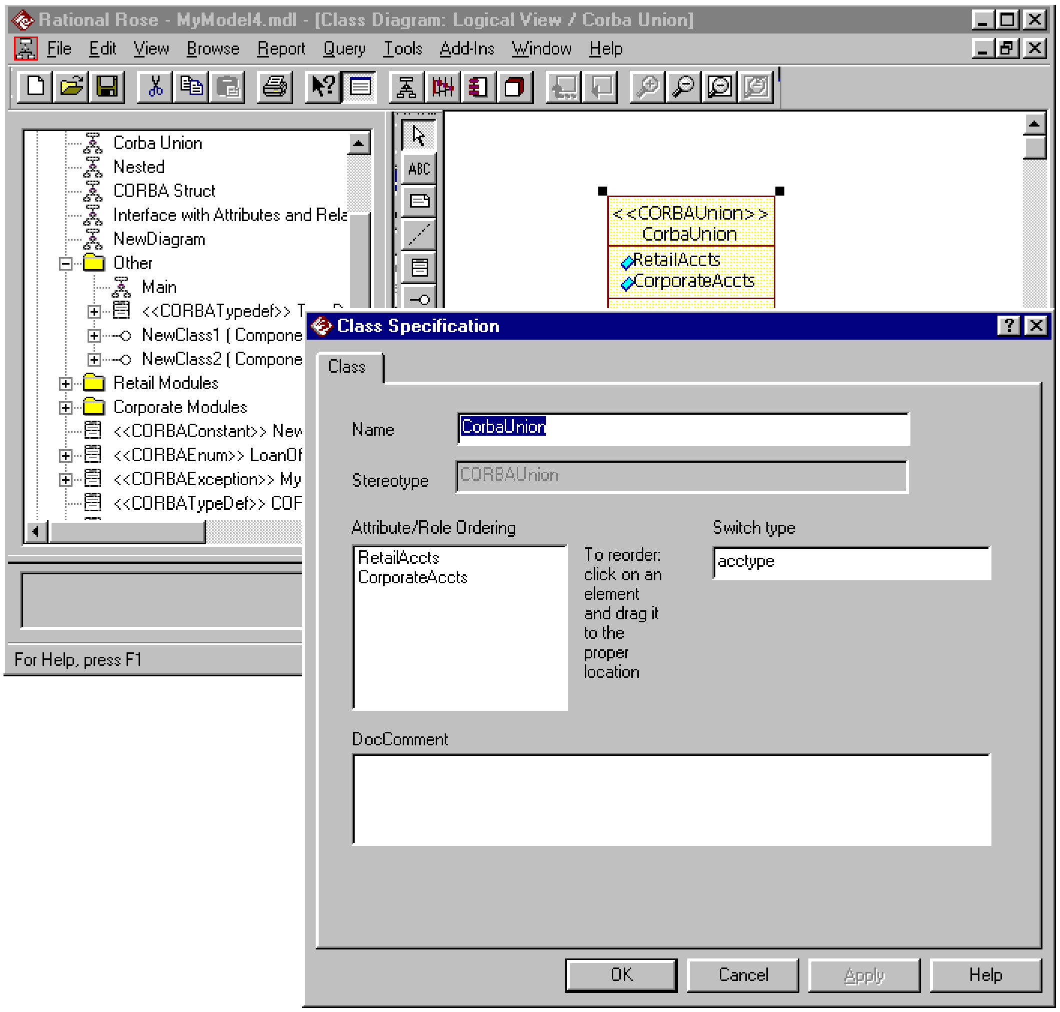
The Union elements map to the class's attributes and relationships. Rational Rose CORBA generates the attributes of the union from a combination of the class's attributes, plus any user-defined types that have a relationship with the class.

The name of the CORBA IDL switch is a user-defined variable that maps to the Union's ImplementationType property in the model. The elements of the Union are called *cases* and map to the attributes of the Union in the model.

Example

The following example shows the characteristics of a simple CORBA Union modeled in Rational Rose, as well as the CORBA IDL code that maps to it.

Notice that the class's stereotype is set to CORBAUnion. It has two attributes (**RetailAccts** and **CorporateAccts**) and its ImplementationType property is set to acctype.



Name: CorbaUnion

Stereotype: CORBAUnion

Attribute/Role Ordering

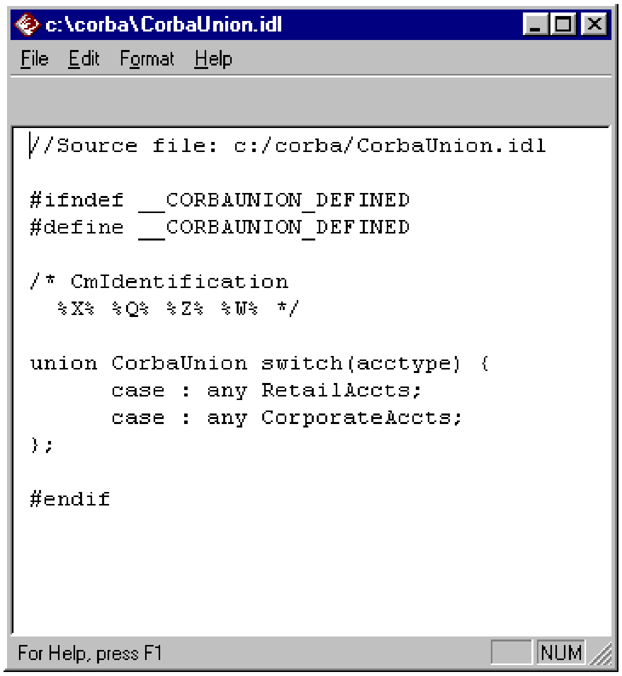
RetailAccts
CorporateAccts

To reorder: click on an element and drag it to the proper location

Switch type: acctype

DocComment

In the CORBA IDL code that maps to this CORBA Union, notice that the ImplementationType in Rational Rose maps to the switch variable in the IDL switch statement. In addition, each of the attributes in the Rational Rose class maps to a case statement that defines the possible values of the Union.



```
//Source file: c:/corba/CorbaUnion.idl

#ifndef __CORBAUNION_DEFINED
#define __CORBAUNION_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

union CorbaUnion switch(acctype) {
      case : any RetailAccts;
      case : any CorporateAccts;
};

#endif
```

More About CORBA Exception, Struct, and Union Mapping

The following subsections provide attribute mapping information and examples that apply equally to Exceptions, Structs and Unions.

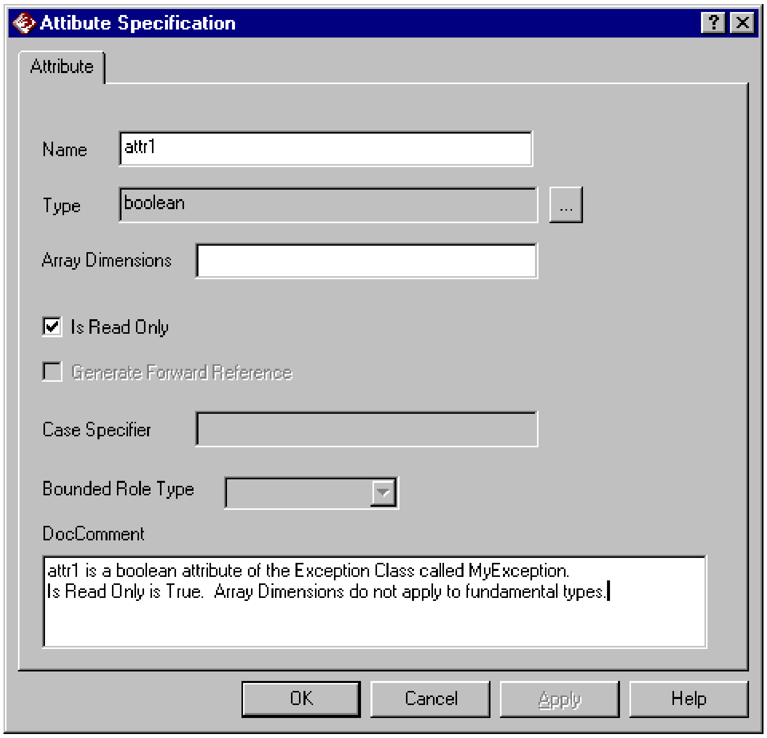
Attributes Defined as Fundamental and Template Types

An Exception, Struct, Or Union attribute that is a CORBA fundamental type or template type is represented in a Rational Rose model as an attribute whose Type is a CORBA fundamental type or template type. You assign this Type in the attribute specification.

Example

This example shows a custom specification for the attribute of a CORBA Exception (MyException).

Notice that the attribute’s **Type** is **boolean** (selectable from the list of CORBA fundamental and template types) and that it is marked as **Is Read Only**. Since the attribute is a fundamental type, the **ArrayDimensions** property does not apply.



Name: attr1

Type: boolean

Array Dimensions:

[✓] Is Read Only

[ ] Generate Forward Reference

Case Specifier:

Bounded Role Type:

DocComment:

attr1 is a boolean attribute of the Exception Class called MyException.
Is Read Only is True. Array Dimensions do not apply to fundamental types.

RoseTip

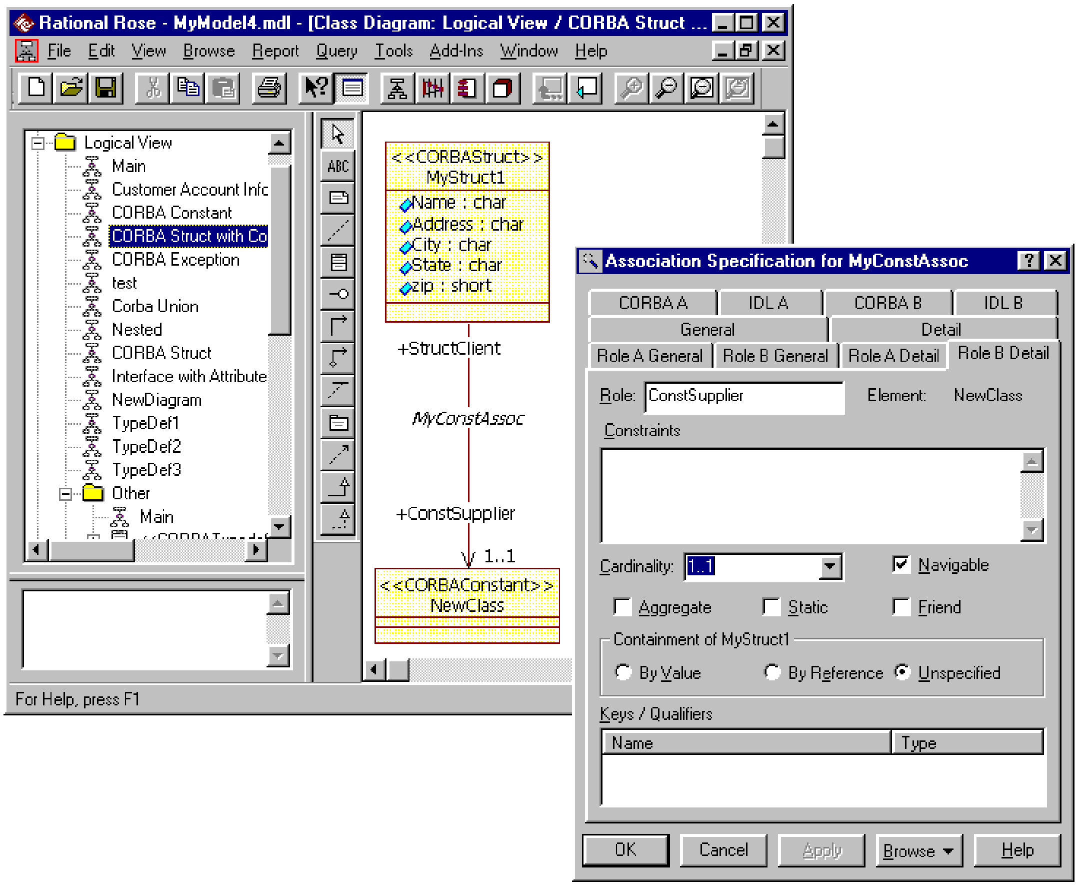
The IsReadOnly property is valid for Exception attributes, but not for attributes of Structs or Unions. If you open the custom specification for a Struct or Union attribute, you will find this field unavailable.

Attributes Defined as References to User-defined Types

An Exception, Struct, or Union element that is a reference to a user-defined type is represented in a Rose model as a relationship of cardinality 1. In this relationship, the Exception, Struct, or Union is the client class and the class that represents the user-defined type is the supplier.

Example

The following example shows a CORBA Struct (**MyStruct1**) in an association relationship with a CORBA Constant (**NewClass**). You can create this relationship between an Exception, Struct, or Union and any CORBA user-defined type. The Association Specification shows that the relationship is of cardinality 1. (See *CORBA IDL Types*, in Chapter 2, for information on user-defined types.)



Role: ConstSupplier Element: NewClass

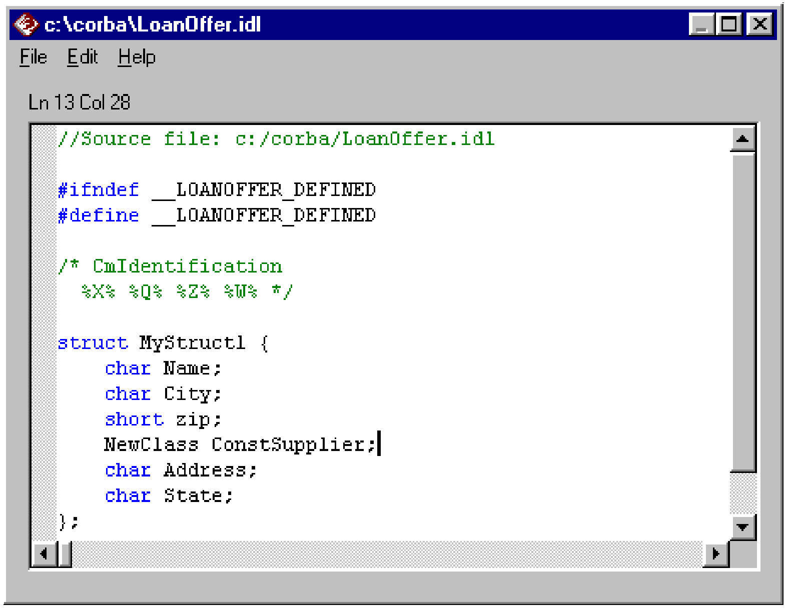
Cardinality: 1..1 [✓ Navigable]

Aggregate Static Friend

Containment of MyStruct1: By Value By Reference [✓ Unspecified]

The following CORBA IDL code corresponds to the definition of **MyStruct1**, including its relationship to **NewClass**. Notice that:

- MyStruct1's attributes (Name, Address, City, State, Zip) in the model, correspond to CORBA Struct elements in the IDL file.
- There is another CORBA Struct element whose name (called a *declarator* in CORBA IDL) is the name of the supplier role of the relationship (**ConstSupplier**) and whose type is the fully qualified name of the supplier class (**NewClass**).



```
//Source file: c:/corba/LoanOffer.idl

#ifndef __LOANOFFER_DEFINED
#define __LOANOFFER_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

struct MyStruct1 {
    char Name;
    char City;
    short zip;
    NewClass ConstSupplier;
    char Address;
    char State;
};
```

RoseTip

The supplier class can also be a single- or multi-dimension array. For more information, see *Attributes Defined as Single- or Multi-Dimension Arrays of User-defined Types.*

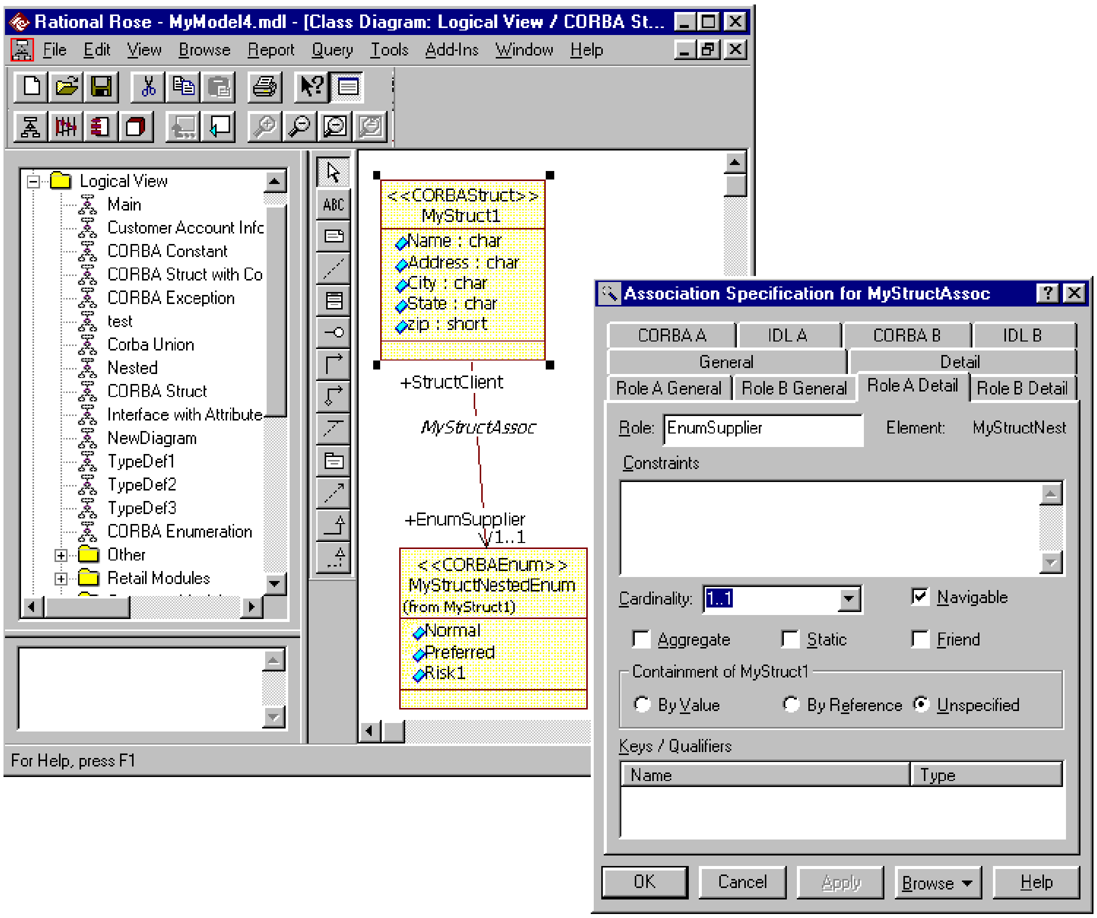
Attributes Defined as Inline Definitions of User-Defined Types

An Exception, Struct, or Union element that is comprised of an inline definition of a CORBA user-defined type is represented in a Rose model as an association relationship between the Exception, Struct, or Union and a nested class. The Exception, Struct, or Union is client class and the class that represents the user-defined type (the nested class) is the supplier class. The cardinality of the relationship is 1.

Example

The following example shows a CORBA Struct (**MyStruct1**) in an association relationship with a nested CORBA Enum (**MyStructNestedEnum**).

The purpose of this relationship is to include an inline definition of a user-defined type (the Enum) as an element of an Exception, Struct, or Union. You can create this relationship between an Exception, Struct, or Union and any CORBA user-defined type. The Association Specification shows that the relationship is of cardinality 1.



<<CORBAStruct>>
MyStruct1
Name : char
Address : char
City : char
State : char
zip : short

+StructClient
MyStructAssoc
+EnumSupplier
1..1

<<CORBAEnum>>
MyStructNestedEnum
(from MyStruct1)
Normal
Preferred
Risk1

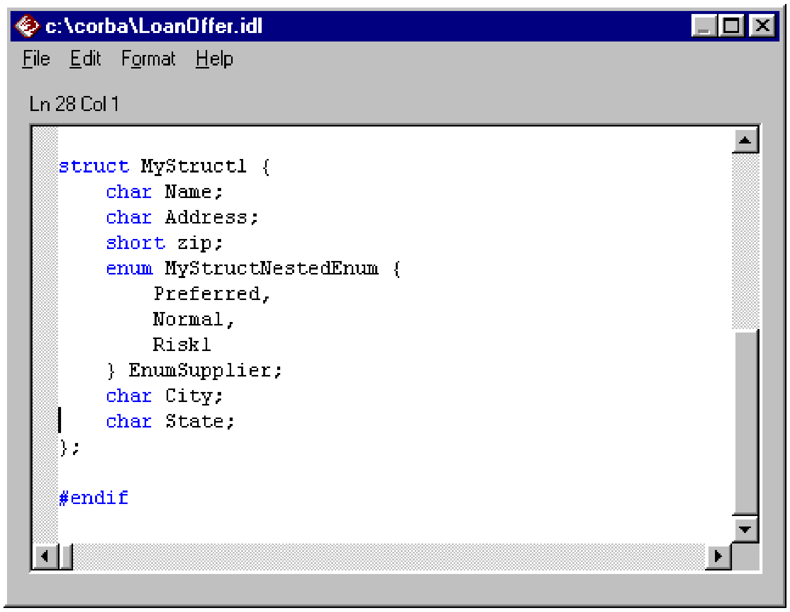
Role: EnumSupplier
Element: MyStructNest
Constraints
Cardinality: 1..1
[✓ Navigable]
Aggregate
Static
Friend
Containment of MyStruct1
By Value By Reference [✓ Unspecified]
Keys / Qualifiers

Name	Type

The following CORBA IDL code corresponds to the definition of **MyStruct1**, including its relationship to **MyStructNestedEnum**.

Notice that MyStruct1's attributes (Name, Address, City, State, Zip) in the model, correspond to the CORBA Struct elements in the IDL file.

In addition, MyStructNestedEnum maps to a complete, inline definition of the Enum within the IDL definition of MyStruct1.



```
struct MyStruct1 {
    char Name;
    char Address;
    short zip;
    enum MyStructNestedEnum {
        Preferred,
        Normal,
        Risk1
    } EnumSupplier;
    char City;
    char State;
};

#endif
```

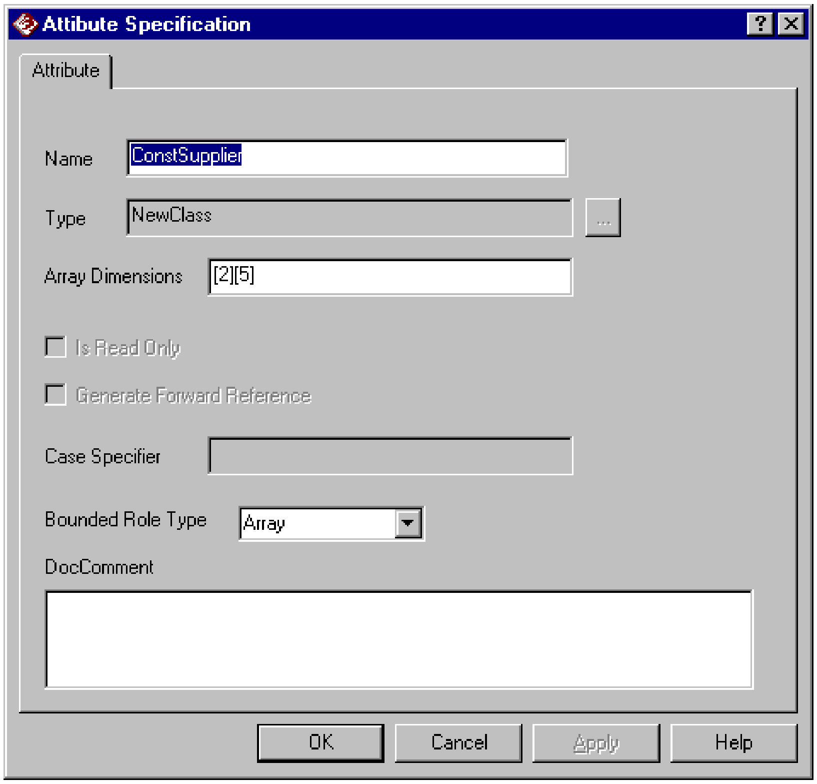
Attributes Defined as Single- or Multi-Dimension Arrays of User-defined Types

There are two ways to represent an exception, struct, or union element that is a single-dimension array of a user-defined type.

Method 1

The most straightforward way to define an attribute as a single- or multi-dimension array of a user-defined type is to create an association in which the supplier's **Bounded Role Type** is set to **Array** and its **Array Dimensions** property is set to the dimension(s) of the array.

For example, the following custom attribute specification defines an attribute called ConstSupplier. The attribute is defined as a multi-dimension array whose dimensions are `[2][5]` and whose **Type** is NewClass, a user-defined CORBA class in the model. Note that a single-dimension array would have only one value (for example, `[2]`) defined in the **Array Dimensions** field.



Name: ConstSupplier
Type: NewClass
Array Dimensions: [2][5]
☐ Is Read Only
☐ Generate Forward Reference
Case Specifier:
Bounded Role Type: Array
DocComment:

Method 2 (Special Case)

An alternative definition that applies only to single-dimension arrays calls for an association in which the supplier's **BoundedRoleType** is set to Array, but instead of defining the supplier's array dimensions, you define its Cardinality. The cardinality must be bounded.

If you use this method to generate code and then reverse engineer the generated IDL code, Rose CORBA creates a relationship in which the supplier's ArrayDimensions property is set appropriately.

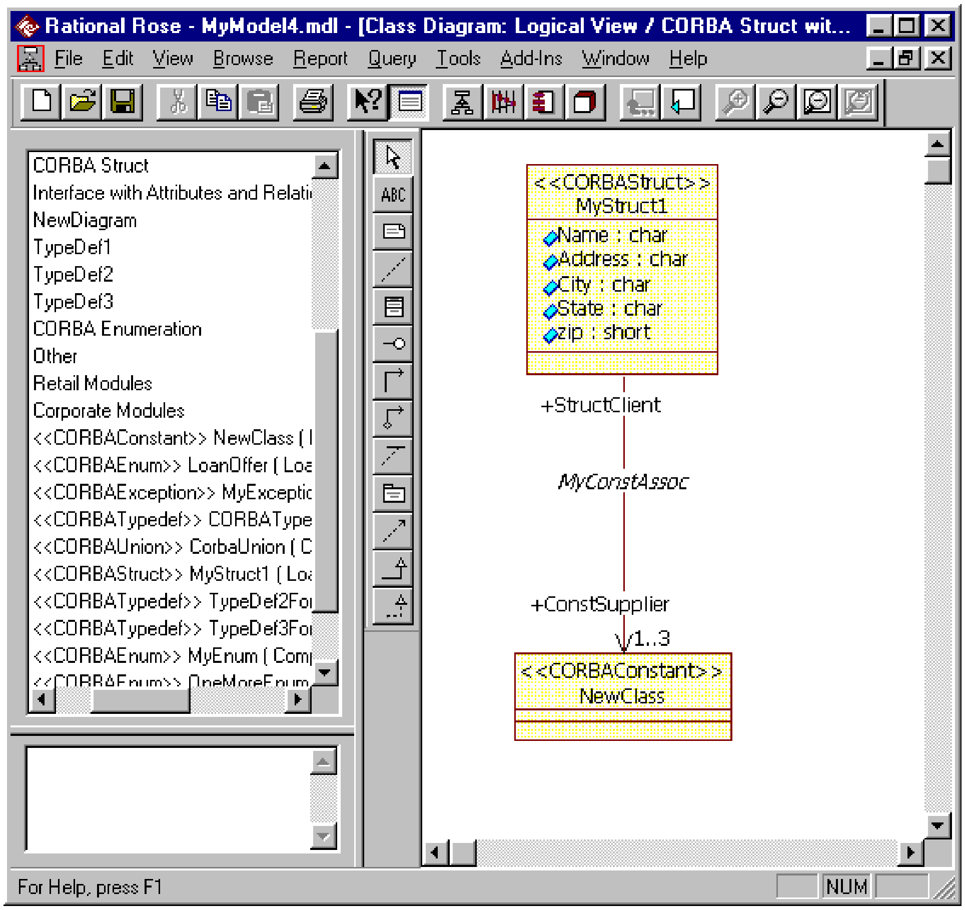
RoseTip

You can define the array as it will eventually be reverse-engineered (Method 1), if you prefer. However, you may find it easier to define it using bounded cardinality and the BoundedRoleType property. Just remember that, depending on your original definition, your reverse-engineered model may no longer match your original model.

Example

The following example is a special case of Exception, Struct, or Union attribute definition . It shows a Struct (MyStruct1) in an association relationship with a single-dimension array of a user-defined type (NewClass). You can create this relationship between an Exception, Struct, or Union and any user-defined type.

To create these model elements, you set the supplier's BoundedRoleType property to Array. Its Cardinality must be bounded. There is no need to set the ArrayDimensions property because the array is single-dimension, and its value will be derived from the cardinality you define.


Rational Rose - MyModel4.mdl - [Class Diagram: Logical View / CORBA Struct wit...
File Edit View Browse Report Query Tools Add-Ins Window Help
ABC

CORBA Struct
Interface with Attributes and Relati
NewDiagram
TypeDef1
TypeDef2
TypeDef3
CORBA Enumeration
Other
Retail Modules
Corporate Modules
<<CORBAConstant>> NewClass (I
<<CORBAEnum>> LoanOffer (Loa
<<CORBAException>> MyExceptic
<<CORBATypedef>> CORBAType
<<CORBAUnion>> CorbaUnion (C
<<CORBAStruct>> MyStruct1 (Loa
<<CORBATypedef>> TypeDef2Foi
<<CORBATypedef>> TypeDef3Foi
<<CORBAEnum>> MyEnum (Comp
<<CORBAEnum>> OneMoreEnum

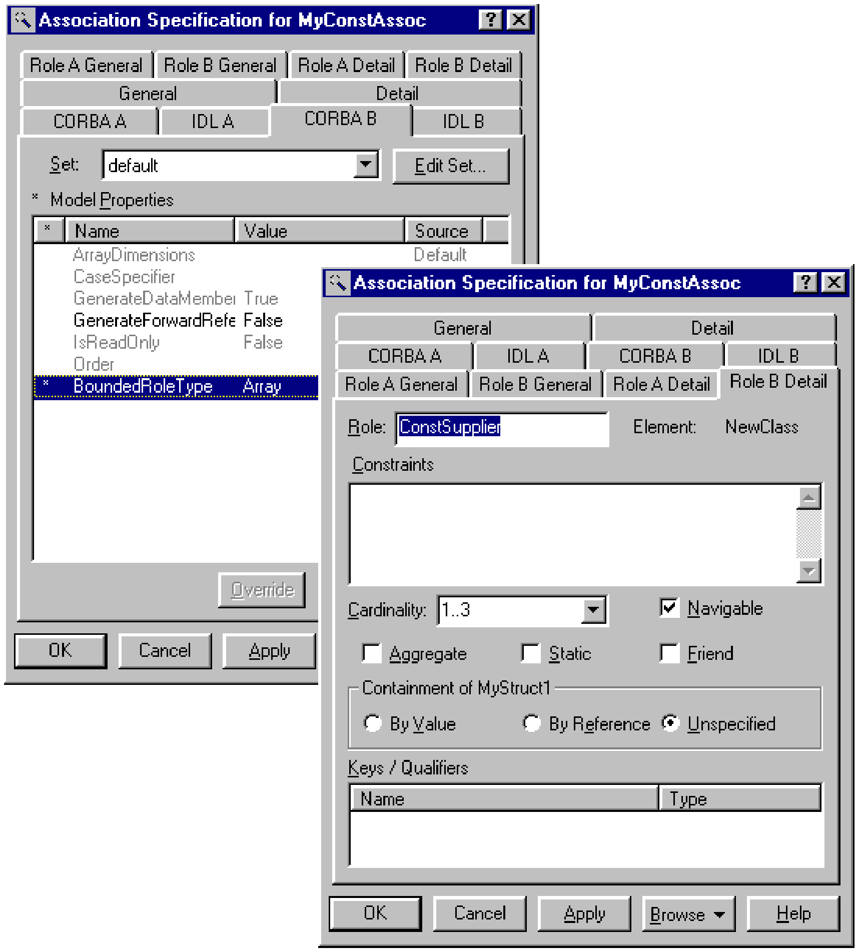
<<CORBAStruct>>
MyStruct1
Name : char
Address : char
City : char
State : char
zip : short

+StructClient
MyConstAssoc
+ConstSupplier
1..3

<<CORBAConstant>>
NewClass

For Help, press F1
NUM

The following figure shows the standard specifications for the supplier role in the association.



Set: default

Model Properties

Name	Value	Source
ArrayDimensions		Default
CaseSpecifier		
GenerateDataMember	True	
GenerateForwardRefe	False	
IsReadOnly	False	
Order		
BoundedRoleType	Array	

Role: ConstSupplier Element: NewClass

Constraints

Cardinality: 1..3

[✓ Navigable]

Aggregate Static Friend

Containment of MyStruct1

By Value By Reference [✓ Unspecified]

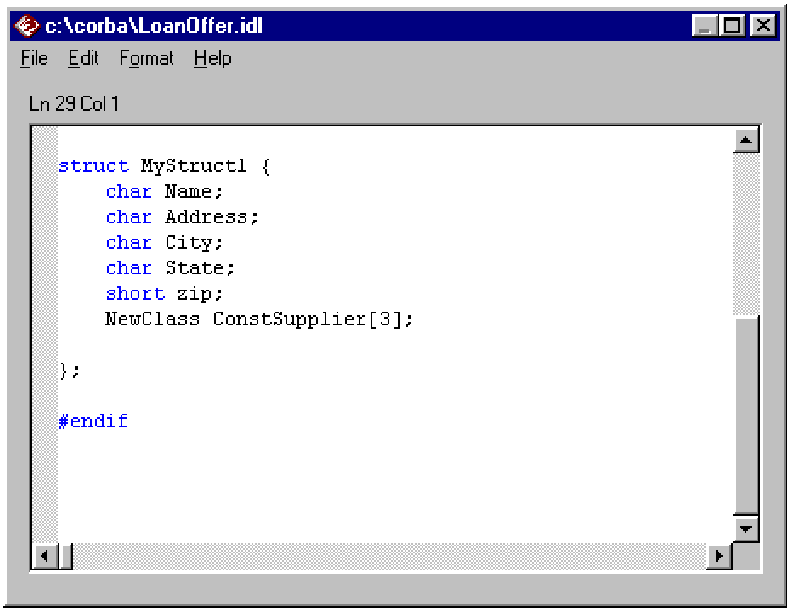
Keys / Qualifiers

Name	Type

When you generate code for this relationship, you will note that Rose CORBA uses the Cardinality you specified to create an array dimension in the CORBA IDL code.

The following CORBA IDL code corresponds to the definition of MyStruct1, including its relationship to NewClass. Notice that:

- MyStruct1's attributes (Name, Address, City, State, Zip) in the model correspond to elements of the Struct definition in the IDL file.
- The code for MyStruct1 defines an element that is a single-dimension array whose name (called a *declarator* in CORBA IDL) is the supplier role (ConstSupplier), whose type is the supplier class (NewClass) and whose single dimension is 3.



```
struct MyStruct1 {
    char Name;
    char Address;
    char City;
    char State;
    short zip;
    NewClass ConstSupplier[3];

};

#endif
```

When you reverse engineer this code, Rose CORBA will:

- Set the ArrayDimensions property as defined in the generated CORBA IDL code.
- Leave the Cardinality you originally defined on your diagram.

Remember that this new model no longer matches the original model. Even though the cardinality you specified remains on the diagram, it will be ignored in all future forward and reverse engineering activities.

Attributes Defined as a Bounded Sequence of User-defined Types

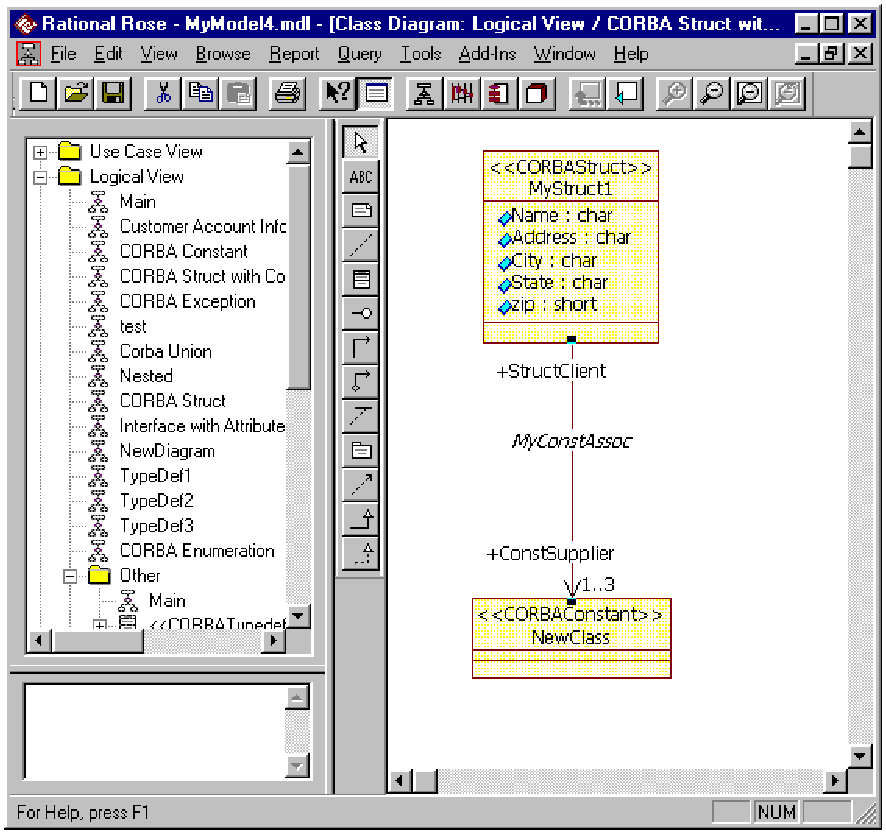
To define an attribute of an exception, struct, or union as a bounded sequence of a user-defined type, you create an association in which the supplier's Bounded RoleType is set to **`Sequence`** and its Cardinality is set to the dimension of the sequence.

If you generate code from these elements and then reverse engineer the generated IDL code, Rose CORBA creates an attribute that corresponds to the supplier role and whose type is set to the type and dimension of the supplier. Remember this will no longer match your original model.

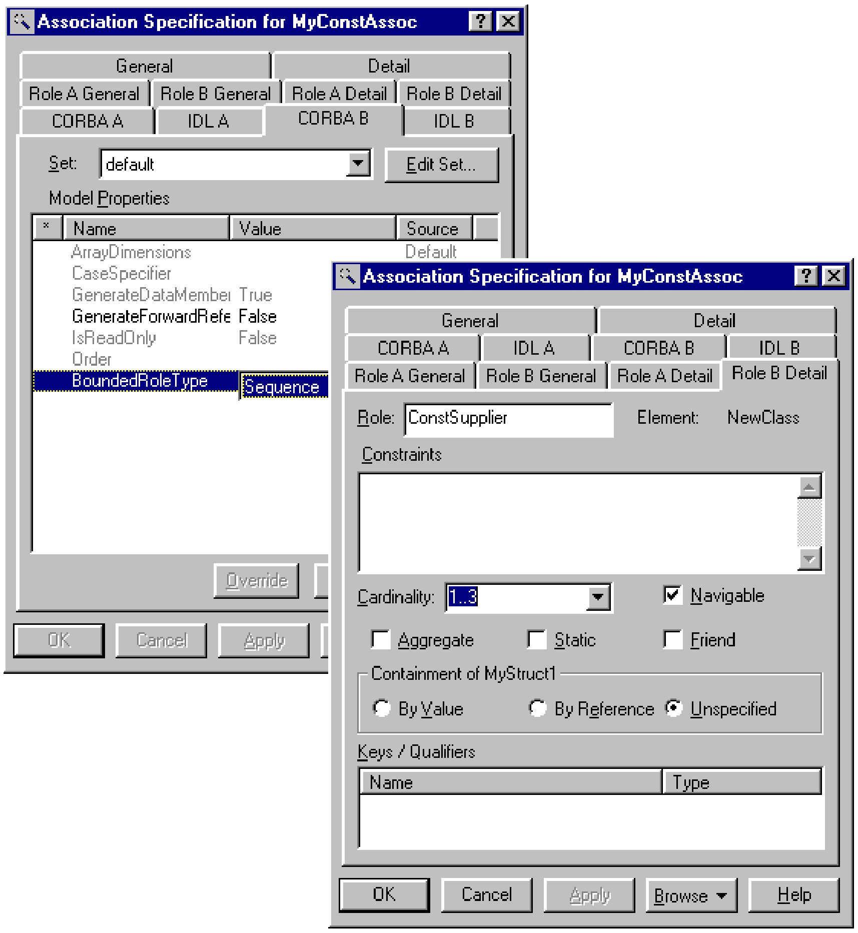
Example

The following example is a special case of Exception, Struct and Union element definition. It shows a CORBA Struct (**MyStruct1**) in an association relationship with a bounded sequence of a user-defined type (**NewClass**). You can create this relationship between an Exception, Struct, or Union and any user-defined type.

To create these model elements, you set the supplier's BoundedRoleType property to **Sequence**. Its cardinality must be bounded.



The following specifications define the supplier role in the association.



Association Specification for MyConstAssoc

General | Detail | Role A General | Role B General | Role A Detail | Role B Detail | CORBA A | IDL A | CORBA B | IDL B

Set: default

Model Properties

*	Name	Value	Source
	ArrayDimensions		Default
	CaseSpecifier		
	GenerateDataMember	True	
	GenerateForwardRefe	False	
	IsReadOnly	False	
	Order		
	BoundedRoleType	Sequence	

Association Specification for MyConstAssoc

Role: ConstSupplier Element: NewClass

Constraints

Cardinality: 1..3 [✓ Navigable]

☐ Aggregate ☐ Static ☐ Friend

Containment of MyStruct1

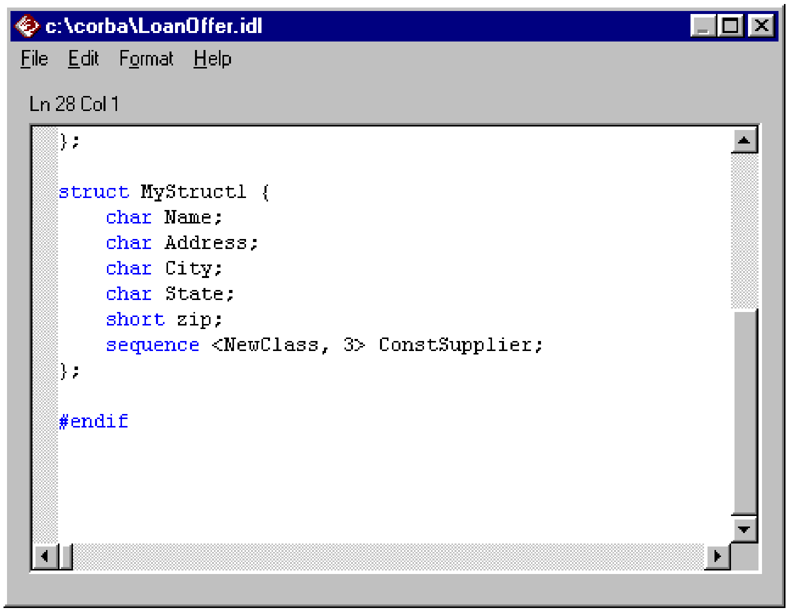
○ By Value ○ By Reference ● Unspecified

Keys / Qualifiers

Name	Type

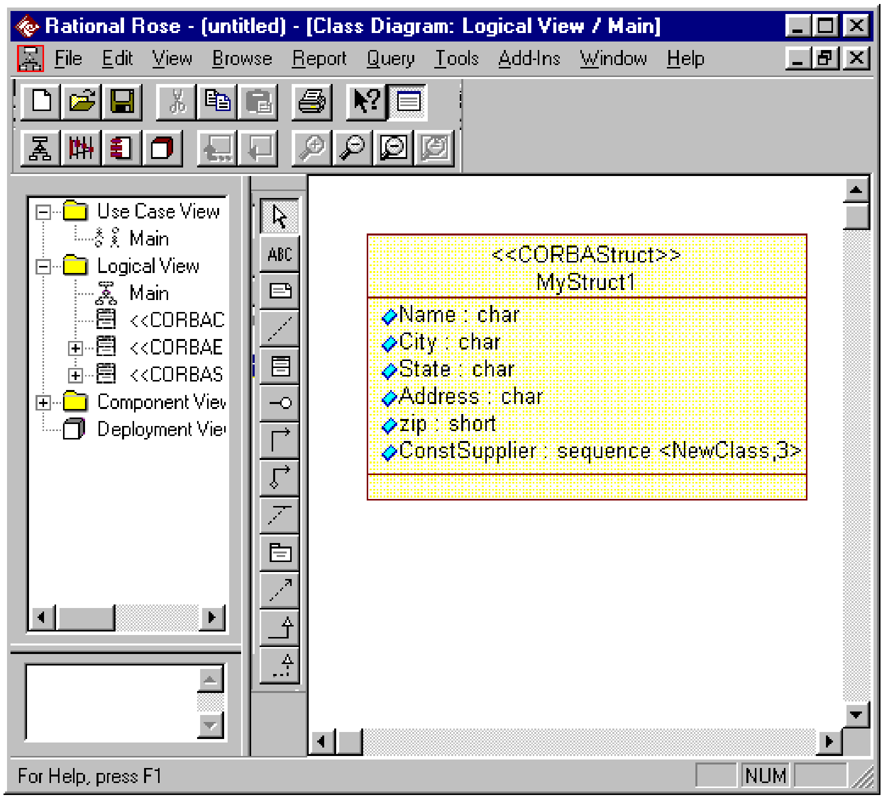
The following CORBA IDL code corresponds to the definition of **MyStruct1**, including its relationship to **NewClass**. Notice that:

- MyStruct1's attributes (Name, Address, City, State, Zip) in the model, correspond to elements of the Struct definition in the IDL file.
- The definition of MyStruct1 includes a sequence whose name (called a declarator in CORBA IDL) is the supplier role (**ConstSupplier**), whose type is the supplier class (**NewClass**), and whose dimension is **3**.



```
};

struct MyStruct1 {
    char Name;
    char Address;
    char City;
    char State;
    short zip;
    sequence <NewClass, 3> ConstSupplier;
};

#endif
```

If you reverse engineer this special mapping case, the new model matches the generated code and looks like the following model. Remember, this new model no longer matches the original model.


Rational Rose - (untitled) - [Class Diagram: Logical View / Main]
File Edit View Browse Report Query Tools Add-Ins Window Help
Use Case View
Main
Logical View
Main
<<CORBAC
<<CORBAE
<<CORBAS
Component View
Deployment View
For Help, press F1
NUM

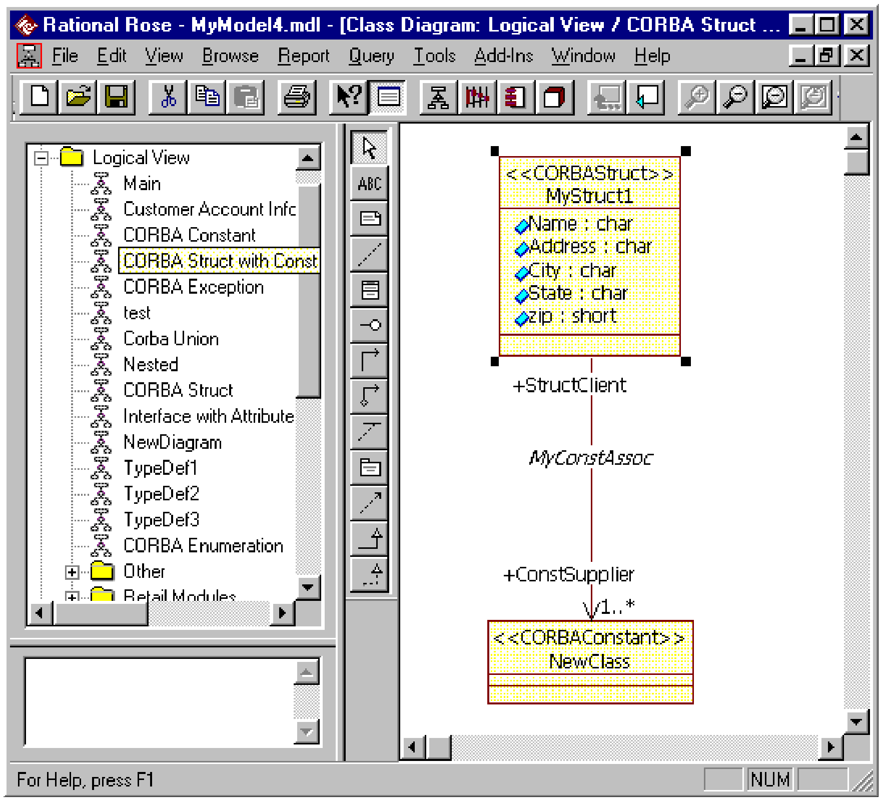
<<CORBAStruct>>
MyStruct1

Name : char
City : char
State : char
Address : char
zip : short
ConstSupplier : sequence <NewClass,3>

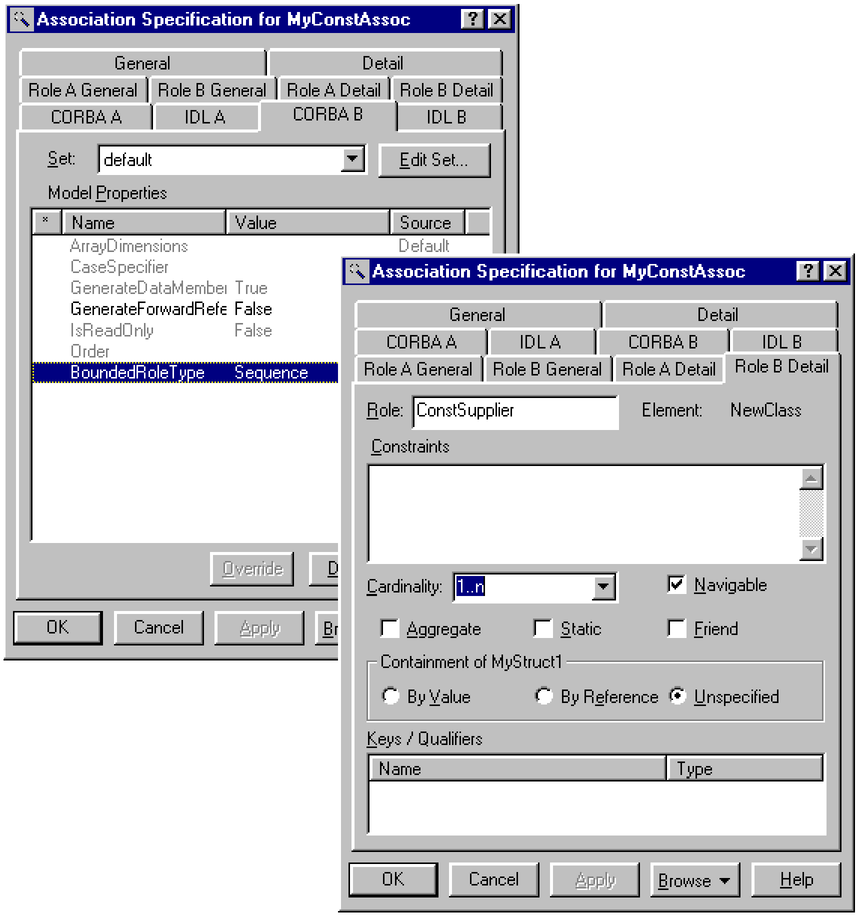
Attributes Defined as Unbounded Sequence of User-defined Types

The following example is a special case of Exception, Struct, and Union element definition. It shows a CORBA Struct (**MyStruct1**) in an association relationship with an unbounded sequence of a user-defined type (OneMoreEnum). You can create this relationship between an Exception, Struct, or Union and any user-defined type.

To create these model elements, you set the supplier's BoundedRoleType property to Sequence. Its cardinality must be unbounded.



The following specifications are for the supplier role in the association.



Association Specification for MyConstAssoc

Set: default

Model Properties

*	Name	Value	Source
	ArrayDimensions		Default
	CaseSpecifier		
	GenerateDataMember	True	
	GenerateForwardRefe	False	
	IsReadOnly	False	
	Order		
	BoundedRoleType	Sequence	

Association Specification for MyConstAssoc

Role: ConstSupplier Element: NewClass

Cardinality: 1..n

[✓ Navigable] Aggregate Static Friend

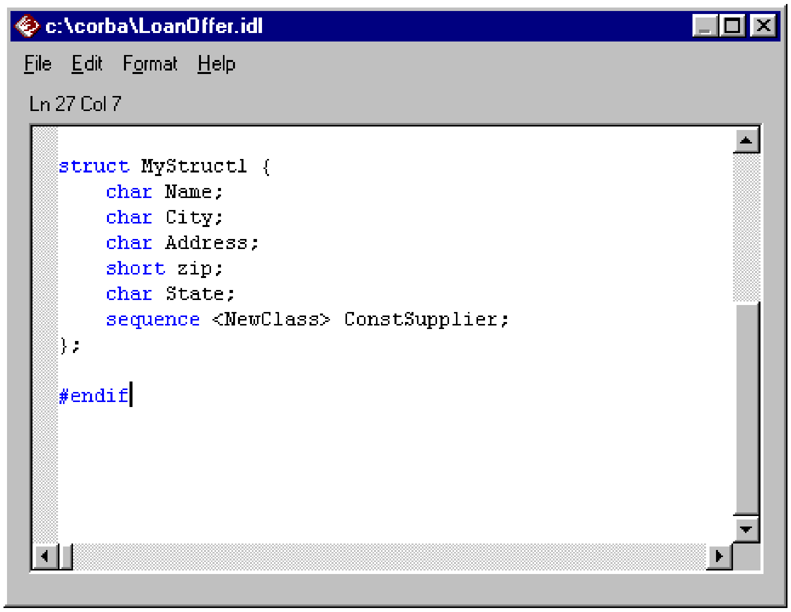
Containment of MyStruct1: By Value / By Reference / [✓ Unspecified]

Keys / Qualifiers

Name	Type

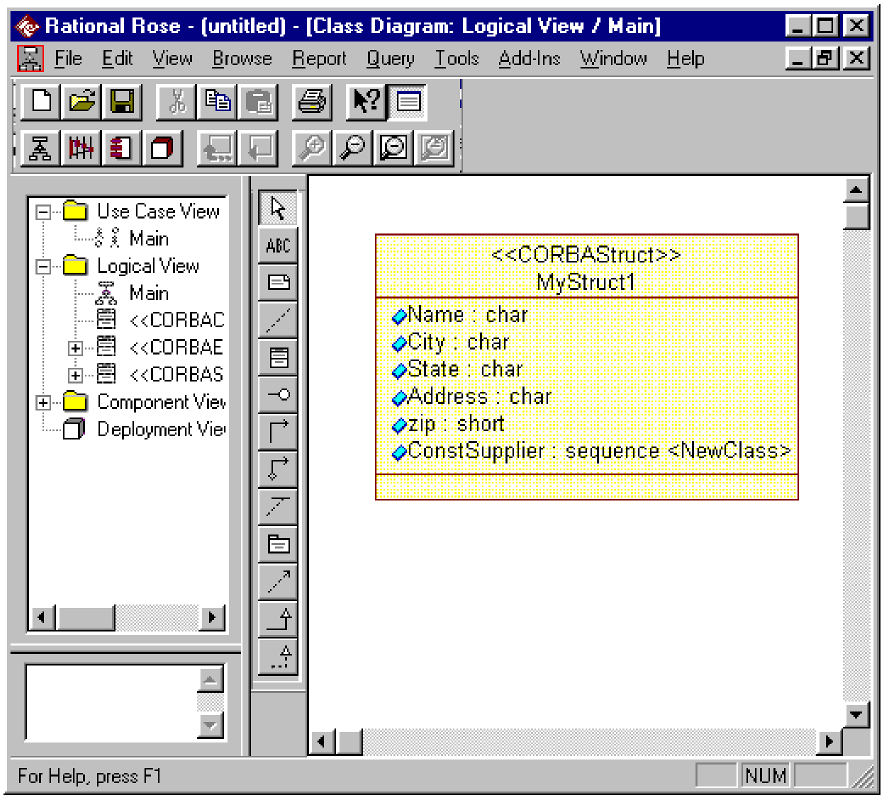
The following CORBA IDL code corresponds to the definition of **MyStruct1**, including its relationship to **NewClass**. Notice that:

- MyStruct1's attributes (Name, Address, City, State, Zip) in the model, correspond to elements of the Struct definition in the IDL file.
- The definition of MyStruct1 includes a sequence whose name (called a declarator in CORBA IDL) is the supplier role (**ConstSupplier**), whose type is the supplier class (**NewClass**), and whose dimension is unspecified.



```
struct MyStruct1 {
    char Name;
    char City;
    char Address;
    short zip;
    char State;
    sequence <NewClass> ConstSupplier;
};

#endif
```

If you reverse engineer this special mapping case, the new model matches the generated code and looks like the following figure. Remember, this new model no longer matches the original model.


Rational Rose - (untitled) - [Class Diagram: Logical View / Main]
File Edit View Browse Report Query Tools Add-Ins Window Help
Use Case View
Main
Logical View
Main
<<CORBAC
<<CORBAE
<<CORBAS
Component View
Deployment View
For Help, press F1
NUM

<<CORBAStruct>>
MyStruct1

- Name : char
- City : char
- State : char
- Address : char
- zip : short
- ConstSupplier : sequence <NewClass>

Chapter 7

Forward and Reverse Engineering with Rational Rose CORBA

Forward engineering is the process of generating CORBA IDL code from a Rational Rose model. Reverse engineering is the process of generating model elements from CORBA IDL files.

As part of these processes, you might also check the syntax of model elements, edit your code, and update your model.

Eventually, you will repeat the forward or reverse engineering process in order to carry model changes forward into existing code or to carry code changes back into your model. This last step is often referred to as Roundtrip Engineering.

You can repeat the forward and reverse engineering processes as necessary to keep your model and code in synch. Rational Rose CORBA will preserve your changes throughout these iterations.

This chapter explains how to accomplish the various tasks that comprise the forward and reverse engineering processes.

Setting Project Properties

You can use the CORBA custom Project Specification to set values for the following project-level properties, which affect forward and reverse engineering:

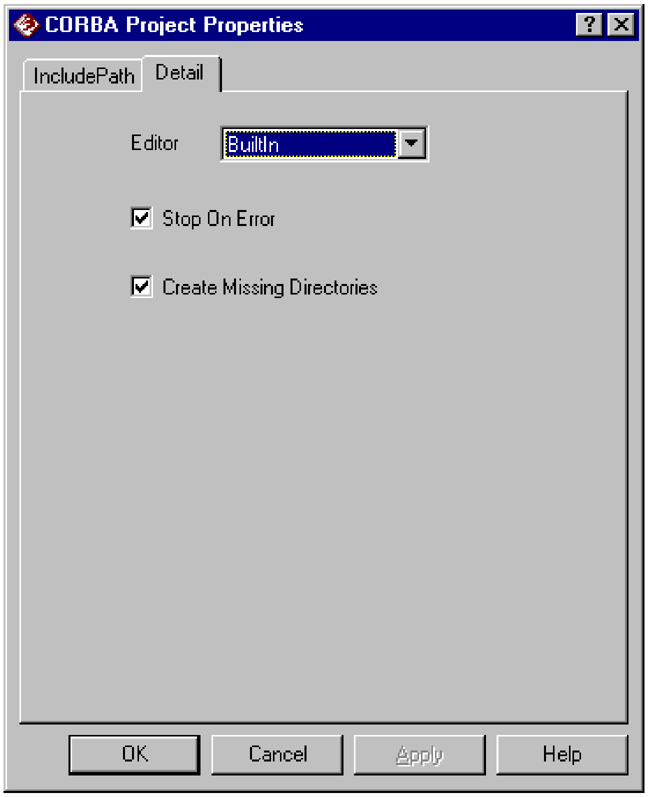
- StopOnError

  Indicates whether to stop code generation or reverse engineering the first time Rational Rose CORBA encounters an error
- CreateMissingDirectories

  Indicates whether to create directories as needed during forward engineering

Use the following procedure to set values for these project properties:

1. Click **Tools** > **CORBA** > **Project Specification**.
2. Click the **Detail** tab.
   - To set a property to True, select its checkbox.
   - To set a property to False, clear its checkbox.


CORBA Project Properties

IncludePath | Detail

Editor: BuiltIn

[✓] Stop On Error

[✓] Create Missing Directories

RoseTip

This page of the Project Specification also allows you to set the default source editor. For details, see *Setting the Default Editor*, later in this chapter.

For information about the IncludePath page of the Project Specification, see *Include Path Mapping* in Chapter 2.

Forward Engineering

Once your project properties are set, the tasks associated with forward engineering are:

- Assign classes to components
- Check model syntax (optional)
- Generate code
- Display and edit generated code

Assigning Classes to Components

Assigning classes to components is a basic function of the Rational Rose browser. You can assign multiple classes to the same component. Each component generates code to a single .idl file.

If you do not assign a class to a component, during forward engineering Rational Rose CORBA creates a component with the same name as the class, and automatically assigns the class to the new component. The disadvantage to allowing this automatic assignment is that you cannot generate multiple related classes into a single source file.

To assign a class to a component:

1. In the browser, click and drag the class from the logical view to an appropriate component in the component view.
2. Double-click the component to display its specification.
3. If it is not already set, set the **Language** field in the specification to **CORBA**.

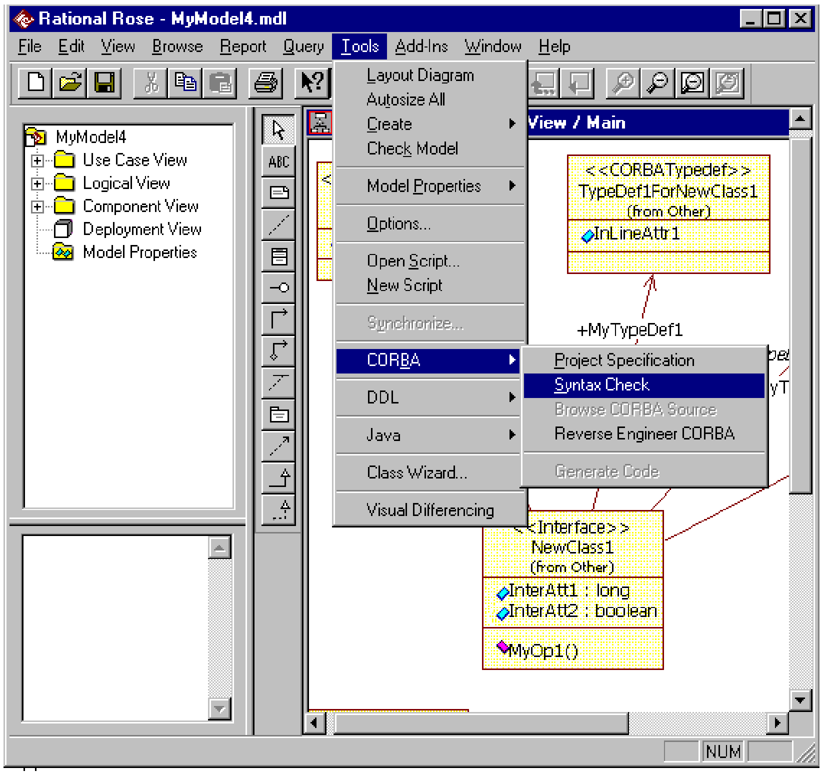
   Assigning the language to CORBA makes the CORBA custom specifications available for the component and its assigned classes. This is the only way to activate the custom specifications.

Checking Model Syntax

The CORBA syntax checker in Rational Rose finds rudimentary syntax errors before you generate code. However, more complicated syntax checking occurs automatically during code generation.

To check your model for CORBA syntax errors:

1. Click **Tools** > **CORBA** > **Syntax Check**.



2. Select **Log** from the **Window** menu to view information and errors.

Before You Generate Code

If your model uses controlled units, you should verify that all of them are loaded before you generate code. Although you are not required to load all units, any code you generate in this way may be incorrect. Some of the problems that can occur as a result of code generation with unloaded controlled units are:

- If the supplier class of an Association or the Association itself is in an unloaded controlled unit, no CORBA IDL Attribute will be generated.
- If the supplier class of a Generalization relation is in an unloaded controlled unit, no parent Interfaces will be generated.
- If the supplier component in a Dependency relation is in an unloaded controlled unit, no `#include` statement will be generated.

If you do not load all of your model's controlled units before beginning code generation, Rose CORBA displays the following message:

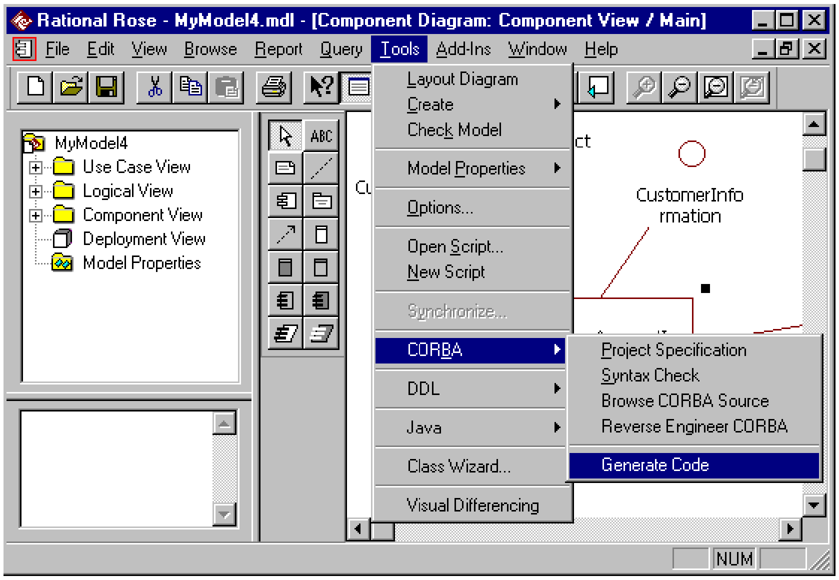
Not all units are loaded. Incorrect code might be generated. Do you want to continue?

You can continue at your own risk or load all units.

Generating CORBA IDL Code

Follow these steps to generate CORBA source from a component diagram in Rose:

1. Open your model and display a class or component diagram that contains the elements for which you want to generate code.
2. Select one or more packages, classes, or components in the diagram.
3. Click **Tools** > **CORBA** > **Generate Code**.

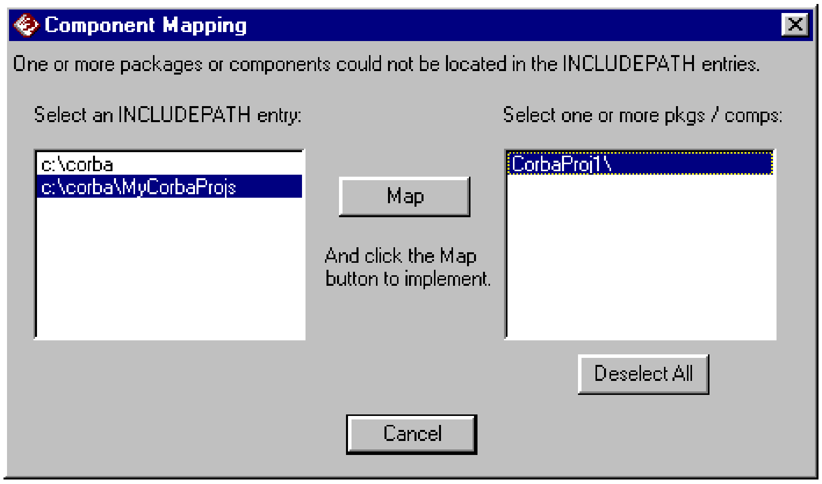


4. If you are prompted to map components during code generation, follow the instructions in *Mapping Components During Code Generation*, which immediately follows this procedure.
5. Check the Rose Log window to view the results of the CORBA generation, including any errors that occurred.
6. Correct any errors and then repeat steps 3 through 5 until no errors are returned.

Mapping Components During Code Generation

In order to successfully generate code for a component, the component must be mapped to one of your project's IncludePath entries.

If you created a component outside a valid IncludePath hierarchy, the Component Mapping dialog box will appear when you generate code, requiring you to map the component (or its parent package) at that time.


Component Mapping

One or more packages or components could not be located in the INCLUDEPATH entries.

Select an INCLUDEPATH entry:

c:\corba
c:\corba\MyCorbaProjs

Select one or more pkgs / comps:

CorbaProj1\

And click the Map button to implement.

Follow these steps to map packages or components during code generation:

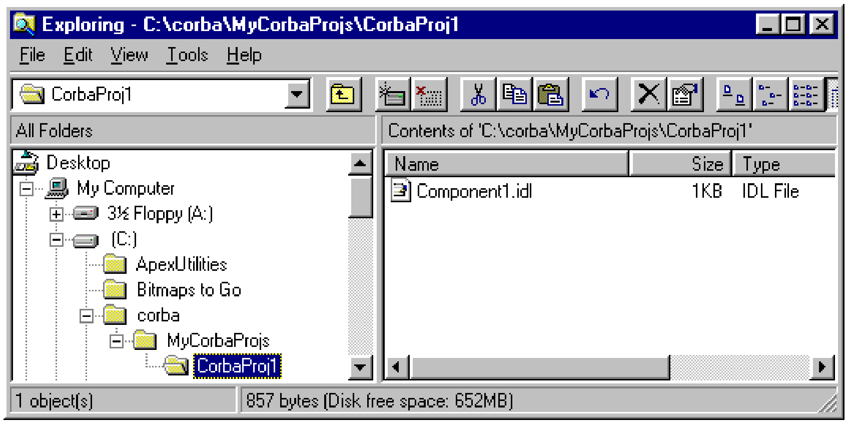
1. From the list of IncludePath entries, select the path (directory structure) in which to place the IDL file when it is generated.
2. From the list of unlocatable packages and components, select one or more items to map to the currently selected IncludePath.

   The list of unlocatable packages and components shows only the topmost unlocatable level. Once you locate this level relative to your chosen IncludePath, all of its subordinate packages and components in your model will be located within this hierarchy.

3. Click Map to map your selected package or component to the selected directory structure and complete code generation.

   In the sample dialog, the component package **CorbaProj1** maps to the IncludePath **c:\CORBA\MyCorbaProjs.** All packages and components defined under **CorbaProj1** will also be mapped within this directory structure, but subordinate to **CorbaProj1**.

4. Go to Windows Explorer to trace the location of the subdirectories and files that now belong to the hierarchy you defined. Notice that component1.idl resides in the CorbaProj1 folder, and CorbaProj1 is a subfolder of the IncludePath to which you mapped it.


Exploring - C:\corba\MyCorbaProjs\CorbaProj1
File Edit View Tools Help
CorbaProj1
All Folders
Contents of 'C:\corba\MyCorbaProjs\CorbaProj1'
Desktop
My Computer
3½ Floppy (A:)
(C:)
ApexUtilities
Bitmaps to Go
corba
MyCorbaProjs
CorbaProj1

Name	Size	Type
Component1.idl	1KB	IDL File

1 object(s) 857 bytes (Disk free space: 652MB)

Displaying and Editing Generated Code

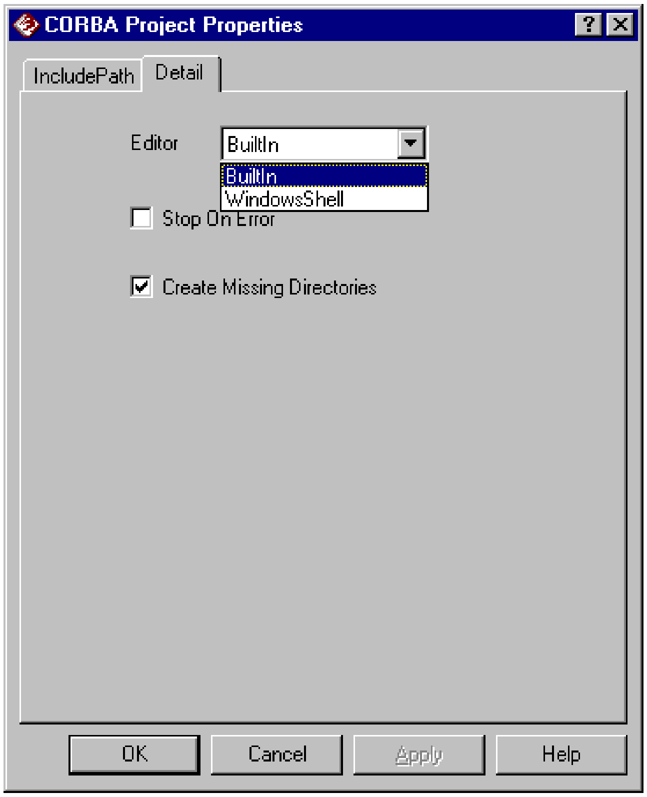
Setting the Default Editor

Rational Rose CORBA provides an editor for viewing and editing CORBA IDL (.idl) files. This BuiltIn editor is the default editor for viewing and changing IDL files generated from a model.

You can change the default editor for your model by setting or resetting the model's Editor property on the CORBA custom Project Specification.

Follow these steps to set the default editor:

1. Click **Tools** > **CORBA** > **Project Specification**. This opens the custom CORBA Project Specification.
2. Click the **Detail** tab.
3. Expand the Editor menu and select **BuiltIn** or **WindowsShell**, as shown.



Editor: BuiltIn

[ ] Stop On Error

[✓] Create Missing Directories

4. Click **OK** to set the **Editor** property.

RoseTips

If you set the Editor property to BuiltIn, CORBA IDL source is displayed using the built-in CORBA editor.

If you set the Editor property to WindowsShell, the source is displayed using the application with which the file type is associated.

To associate a CORBA source file with an editor other than **BuiltIn**, go to Windows Explorer and then click **View** > **Options**. On the **File Types** tab, specify the application to use when opening **.idl** files.

Check Windows Help if you need more information on working with file types.

Using the BuiltIn Editor

The generated CORBA source for each component is stored in a separate .idl file. The generated source contains CORBA programming elements based on the objects and relationships defined in your model.

After generating the CORBA IDL source, you will want to view, and perhaps edit, the generated source. If you use the CORBA BuiltIn editor, you will see the source displayed with the following characteristics:

- Keywords in blue
- Literal strings in red
- Comments in green

RoseTips

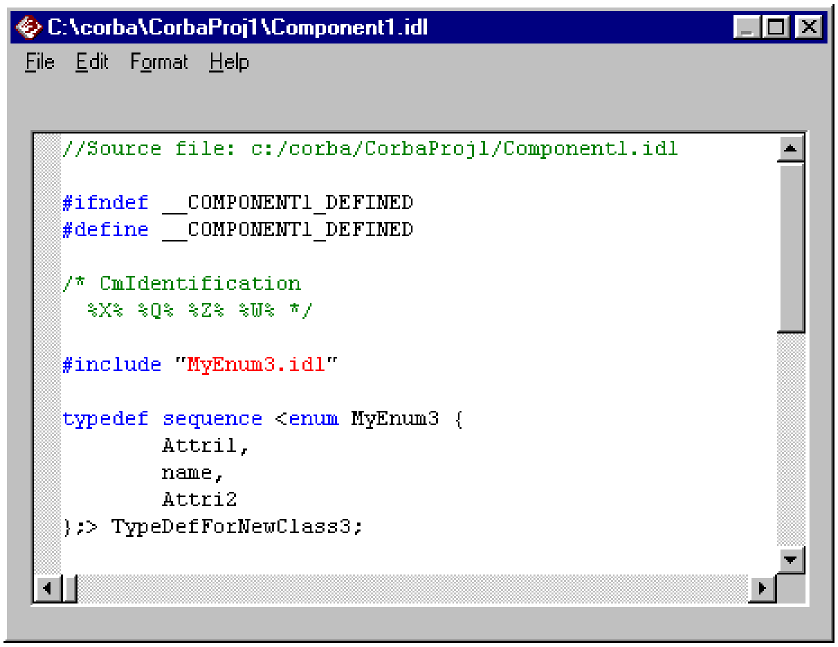
You can use the Format menu in the built-in editor to change the colors and fonts used to display the elements in your IDL code.

Follow these steps to display and edit code using the BuiltIn Editor:

1. Select one or more classes or components whose generated IDL you want to edit.
2. Click **Tools** > **CORBA** > **Browse Code**. One Editor window containing the CORBA IDL file opens for each selected class or component.

RoseTip

If you select multiple classes that are assigned to the same component, each open Editor window will contain the same IDL file. To avoid confusion in such cases, select the component rather than its assigned classes.



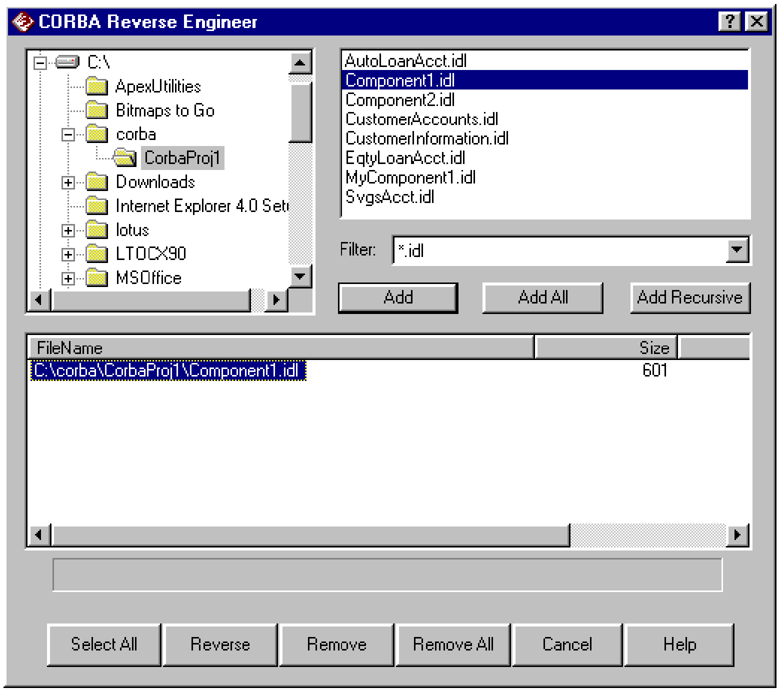
```
//Source file: c:/corba/CorbaProj1/Component1.idl

#ifndef __COMPONENT1_DEFINED
#define __COMPONENT1_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

#include "MyEnum3.idl"

typedef sequence <enum MyEnum3 {
        Attri1,
        name,
        Attri2
};> TypeDefForNewClass3;
```

Reverse Engineering

Reverse engineering is the process of creating or updating a model by analyzing CORBA source. As Rational Rose CORBA reverse engineers each IDL file, it finds the classes, components, attributes, roles, and operations in the file and includes them in your model.

Follow these steps to reverse engineer CORBA IDL files:

1. If you are updating an existing model, open the model.
2. Click **Tools** > **CORBA** > **Reverse Engineer**.



AutoLoanAcct.idl
Component1.idl
Component2.idl
CustomerAccounts.idl
CustomerInformation.idl
EqtyLoanAcct.idl
MyComponent1.idl
SvgsAcct.idl

Filter: *.idl

FileName	Size
C:\corba\CorbaProj1\Component1.idl	601

3. Select a folder in the Tree to display the list of files it contains.
4. Do one of the following to place .idl files into the **Selected Files** list:
   - In the list box, select one or more individual files and click **Add**.
   - Click **Add All** to add all of the files in the selected folder.
   - Click **Add Recursive** to add all of the files in the selected folder and its subfolders.

5. Select one or more files in the Selected Files box or click Select All to confirm the list of files to reverse engineer.
6. Click Reverse to create or update your model from the CORBA source you specified. An error dialog displays, if any errors occur during reverse engineering.
7. Check the Rose Log for a listing of any errors that might have occurred.
8. If necessary, correct errors in the CORBA IDL source and repeat steps 1 through 7.
9. Save the new or revised model.

Index