# Rational Rose 2000e
# Using Rose C++

# *Contents*

# *List of Figures*

# *List of Tables*

# *Preface*

Rational Software corporation's Rational Rose® family of products provides easy-to-use support for object oriented analysis and design, and for controlled iterative development. Rational Rose C++ provides the interface between the Rational Rose modeling environment and the C++ programming language.

This guide is intended for the experienced C++ developer. Familiarity with Rational Rose modeling tools is strongly advised.

This guide is a companion to *Rational Rose 2000e*, *Using Rose*, which provides the conceptual and reference information needed to use the Rational Rose modeling tools.

*Using Rose C++* explains how to:

- Generate C++ source code
- Reverse engineer C++ source code
- Identify differences between Rational Rose models
- Update a Rational Rose model to reflect source code changes
- Apply round-trip engineering to C++ applications

## How this Guide is Organized

Chapter 1 introduces the features of Rational Rose C++, and the basic concepts needed to use it.

Chapter 2 explains the process of generating C++ source code from Rational Rose model elements. It explains the mapping between Rational Rose model elements and C++ code elements.

Chapter 3 explains how to reverse engineer a C++ source code project into a Rational Rose model.

Chapter 4 discusses Analyzer projects and maintaining the Analyzer lists.

Chapter 5 explains the Analyzer export options.

Chapter 6 discusses the Analyzer project file and the annotation process.

Chapter 7 explains how to update a Rational Rose model with the model file updated by the Analyzer.

Chapter 8 explains the Rational rose round-trip engineering process as it applies to C++.

Appendixes A through K list the model properties included with Rational Rose C++, and how they control C++ code and model generation.

Appendix L explains how to set up the Analyzer for proper operation.

## Related Documentation

The following documents are included with Rational Rose C++.

- Comprehensive on-line help with hypertext links and a two-level search index. To activate on-line help, go to the **Help** menu on the Rational Rose menu bar.
- Online user manuals. Please refer to the README.txt file, found in the Rational Rose installation directory, for more information.
- Release Notes, a help file containing additional information about Rational Rose C++. You access this file from the Windows **Start** menu by selecting **Programs**, then **Rational Rose 2000e**, and then **Release Notes**.
- A README.txt file, containing last-minute information about Rational Rose C++. You access this plain-text file from the Windows **Start** menu by selecting **Programs**, then **Rational Rose 2000e**, and then **ReadMe**.
- A tutorial that introduces UML and guides you through Rational Rose's functionality. Each section includes an exercise where you can perform the exercise and compare your solution to that of a Rational Rose model. To run the tutorial, start the Tutorial item in the Rational Rose program group. For Windows users, you can see the Rational Rose solution demonstrated in a “movie”.

### References

The following books are excellent references to the concepts, semantics, and process of object-oriented analysis and design, and the Unified Modeling Language (UML):

- *Visual Modeling with Rational Rose and UML* by Terry Quatrani, Addison Wesley, 1998, available from Rational Software Corp.
- *Object-Oriented Development*, by Grady Booch and Jim Rumbaugh, available from Rational Software Corp.
- **UML Notation:** *Unified Modeling Language* by James Rumbaugh, Grady Booch, and Ivar Jacobson, available from http://www.rational.com

- **Booch Notation:** *Object-Oriented Analysis and Design with Applications* (second edition) by Grady Booch, Benjamin-Cummings Pub. Co., Redwood City, California, 1993
- **OMT Notation:** *Object-Oriented Modeling and Design*, by J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Prentice-Hall Inc., Engelwood Cliffs, New Jersey, 1991

## File Names

Where file names appear in examples, Windows syntax is depicted. To obtain a legal UNIX file name, eliminate any drive prefix and change the backslashes to forward slashes:

```
c:\project\username
```

becomes

```
/project/username
```

# *Chapter 1*

# *Introduction*

Controlled, iterative, and incremental development processes produce a sequence of implementations where functionality increases towards that specified by the user requirements. An explicit iteration driver provides the focus required by a controlled, iterative process.

Rational's development process specifies critical risks as its iteration driver, prescribing continuous active identification, prioritization, and elimination of risk items throughout the project.

This process begins with requirements definition and analysis, followed by the development of an initial architecture. Critical-risk aspects of the design are then identified and prioritized. A design element can involve risk intrinsically or because it is derived from requirements that are ambiguous, poorly understood, or likely to change.

The highest-priority critical-risk item is chosen as the driver for the first iteration, which is planned as the minimum implementation that, when constructed, can objectively demonstrate the resolution of the targeted risk. Ideally, this demonstration is accomplished by successful execution of a test suite, thereby providing both objectivity and a means of detecting regression during subsequent iterations. This approach also institutes testing early in the cycle, avoiding the typical "big bang" integration and testing efforts that overload teams in the late phases of development projects.

When the implementation of the initial iteration has been completed, it must then be assessed to determine whether the targeted critical-risk has in fact been eliminated. If an iteration eliminates its targeted critical-risk, it must also be assessed to determine whether architectural changes were introduced during its implementation, and whether these changes are acceptable. If so, the application architecture must be updated to reflect the changes instituted during implementation. When the targeted critical-risk has been eliminated, and the architecture and implementation are consistent, then the iteration has been successfully completed.

Subsequent iterations begin with a re-examination of the requirements based on knowledge and user feedback acquired during completed iterations. Experience gained during completed iterations can also result in the exposure of new critical-risk items. This analysis can result in architectural changes or reprioritization of known risk items. This process drives out the most critical-risks, and architectural flaws, during the earliest iterations allowing for detection and correction when it is least expensive to do so.

Each iteration is conducted on the assumption that its entire implementation can be incorporated in the final product.

Specifically designed to support controlled, iterative, and incremental development, Rose C++ provides:

- A code generator that automatically carries forward user-supplied declarations and definitions from the previous iteration, and annotates generated code with design information not representable in C++.
- An analyzer that reverse engineers an implementation’s classes, class relationships, logical packages, components, modules, relationships, and component packages from implementation source code, utilizing information contained in annotations when present.
- A visual differencing mechanism that reports architectural differences between an iteration’s initial design and the “as-built,” reverse engineered, implementation design.
- A design update mechanism that updates the iteration’s initial design to reflect architectural changes introduced in its implementation.

Collectively, these mechanisms facilitate round-trip engineering architecturally driven, controlled, iterative, and incremental development with tool support provided by Rose C++.

# *Chapter 2*

# *C++ Code Generation*

Using the Rational Rose C++ code generator, you can produce C++ source code from the information contained in a Rational Rose model. The code generated for each selected model component is a function of that component's specification and code generation properties, and the project's properties. These properties provide the language-specific information required to map your model onto C++, and allow you to control the code generated for each component.

Rational Rose C++ preserves the declarations and definitions you add to generated source code from one iteration to the next. You place these declarations and definitions in protected code regions which are preserved whenever code is regenerated.

A Rational Rose model contains information that cannot be expressed in C++. To support round-trip engineering, information which logically belongs with the generated source code is placed in annotations, which are structured comments. The Documentation field in an Operation Specification, for example, is inserted into an annotation preceding the declaration of the member function.

## Generated Source Code

The specifications and properties of the model components you select determine the code produced by the C++ code generator. One or more header and implementation files contain the generated code.

In this section, the term module specification refers to a subprogram specification, package specification, or task specification. Similarly, the term module body refers to a subprogram body, package body, or task body. The module kind does not modify the C++ code generated.

## Classes Assigned to Modules

When a class you have selected for code generation is assigned to both a module specification and a module body, the C++ code generator places its declarations and definitions in the header and implementation files associated with the module specification and module body, respectively. You assign a class to a module in the module specification.

***Note:*** *To assign a class to a module, the Rational Rose model must contain a corresponding module body*.

If the class is only assigned to a module specification, the C++ code generator places only the class definition in the header file corresponding to the module specification. It also warns that because the class is not assigned to a module body, no implementation source is generated.

If the class is only assigned to a module body, the C++ code generator places the class definition and implementations in the implementation file corresponding to the module body.

The C++ code generator generates a class assigned to a module body as local to that module, by placing both its declaration and definition in the implementation file associated with the module body.

The name of a module specification's header file is controlled by the module specification's FileName property. The name of a module body's implementation file is controlled by the module body's FileName property. File name extensions are controlled by the model's HeaderFileExtension and CodeFileExtension properties; the default values of these properties are `.h` and `.cpp`, respectively.

When the C++ code generator produces a header or implementation file that already exists, the previous version is first copied to a backup file. The file name extension of this backup file is specified by the HeaderFileBackupExtension or CodeFileBackupExtension project properties; the default values of these properties are `.h~` and `.cp~`, respectively.

## Classes Not Assigned to Modules

If a class you have selected for code generation has not been assigned to a module, the C++ code generator assigns it to an implicit pair of modules: a module specification and a module body. When you generate code for this class:

- The implicit module specification generates a header file that contains the class declaration, including declarations of its data members and member functions.
- The implicit module body generates an implementation file that contains the definitions of member functions and static data members.

The default code generation properties for module specifications and module bodies determine the generated code for implicit modules. The names of these header and implementation files are derived from the class name. File name extensions are controlled by the model's HeaderFileExtension and CodeFileExtension properties; the default values of these properties are `.h` and `.cpp`, respectively.

When the C++ code generator produces a header or implementation file that already exists, the previous version is first copied to a backup file. The file name extension of this backup file is specified by the HeaderFileBackupExtension project property or CodeFileBackupExtension project property; the default values of these properties are `.h~` and `.cp~`, respectively.

## Generated Header Files

The generated header files contain annotations, inclusion protection directives, `#include` directives, global declarations, and code regions. Each header file also contains one or more of the following:

- Class declarations, including class names and base lists
- Member function declarations for standard operations, get and set operations for data members, and user-defined operations

## Generated Implementation Files

The generated implementation files contain annotations, `#include` directives, global declarations, static data members, skeletal member function definitions, and code regions.

The C++ code generator does not generate source code which implements member functions, but instead produces skeletal member function definitions containing protected code regions. If you place each member function's implementation within its code region, this implementation code is preserved whenever code is regenerated from the model.

## Logical and Component Packages

If the logical package containing the classes selected for code generation are assigned to component packages, all generated files are placed in a hierarchy of directories that correspond to the model's component package hierarchy. The root of this hierarchy is specified by the Directory project property.

If a logical package containing a class selected for code generation is not assigned to a component package, the code generator assigns it to an implicit component package and creates a corresponding directory to contain its generated files. The name of this implicit component package is taken from the logical package.

## Source Code Annotations

Source code annotations are structured comments used to express model information that cannot be expressed in C++ but should be associated with generated source code. Annotations also identify Protected Code Regions, which preserve user-supplied declarations and definitions when code is regenerated. Annotations begin with the string `//##`.

## Protected Code Regions

Protected code regions are pairs of annotations generated by the C++ code generator to delimit declarations or definitions you provide so they can be retained when code is regenerated. Generated code regions are:

- At the head of each module spec and body, allowing you to provide auxiliary `#include` directives
- At the head of each module spec and body, allowing you to provide auxiliary global declarations

- Within each section of a class declaration (`public`, `protected`, `private`, and `implementation`), allowing you to provide auxiliary declarations
- Preceding each set of definitions for a class (within a module body), allowing you to provide auxiliary declarations and definitions associated with that class
- Within skeletal member function definitions, allowing you to provide the member function's implementation

Protected code regions have the form:

```
//## begin code_region preserve=yes
//## end code_region
```

where *`code_region`* is a unique identifier constructed by the code generator. The *`preserve`* argument indicates whether the contents of the code region are to be preserved when code is regenerated. Changing the argument from its default `yes` to `no` causes the code generator to discard any declarations or definitions contained in the code region when source code is regenerated.

When editing generated source files:

- Add declarations and definitions only between the `//##begin` and `//##end` annotations. Any code added outside of these annotations is not preserved when code is regenerated.
- Do not create your own code regions—you can only use those provided by the C++ code generator.
- Do not move code regions within their module.
- Do not textually delete code regions; if the contents of a code region should be discarded, change `preserve=yes` to `preserve=no` in the `//##begin` annotation.

Generating code adds information describing newly-created protected code regions to your current model. If after generating code, you load a new model or exit Rational Rose, a dialog box appears enabling you to first save your current model. If you choose not to save your model, code subsequently added to these protected regions is not maintained.

Code regions associated with model components that are later deleted or renamed become orphan code, which is moved to the end of its implementation file with its preserve argument set to `no`.

### Namespaces and Protected Code Regions

Elements of a namespace other than class definitions, class forward declarations, and typedef and enum declarations must be enclosed in a protected region. Declarations and directives, in particular, must go in protected regions. They are not recognized during round-trip engineering. For this purpose a new protected region is defined for the initial portion of a namespace (before the first allowed declaration). The marker on this region is the name of the namespace with the suffix `.initialDeclarations`. Whether or not this protected region is generated when empty is controlled by the GenerateEmptyRegions property on the module. The permitted values for this property and their meaning are the same as for other instances of this property on other model items.

Since the use of directives and declarations are not recognized in this release, all inter-namespace references are fully qualified. Intra-namespace references are not qualified.

## Code Generation—Step by Step

**To generate code from your Rational Rose model:**

1. Set the Directory project property to name the directory in which generated header and implementation files are to be created.
2. Select the classes for which you want to generate code. If you select a logical package, code is generated for every class it contains.

   Check each classes' specification for the desired operations. The C++ code generator uses this information to generate data members and member functions.

   Check that all units are loaded. If code generation is applied to a class that has 'uses' links to classes in units not yet loaded, the generated code has no 'with' clause for the missing classes.
3. Click **Tools > C++ > Code Generation**.
4. If errors are reported during step 3, open the log icon at the lower left side of the application window to view the error message.
5. Edit the generated source files and add implementation code to complete the protected code regions for skeletal member function definitions and auxiliary declarations.
6. Compile, link, and test the edited files.

7. Click **File > Save** to preserve any additional information created during code generation.

## Code Generation Properties

Code generation properties provide language-specific information that is not expressed in the Rational Rose notation, but is necessary for generating source code.

Properties also allow you to control the code generated for each of the model components. For example, a class' GenerateCopyConstructor property determines whether a copy constructor is automatically generated.

When a model component is created, Rational Rose sets each of its properties to a default value that can be modified.

Certain code generation properties are model-wide in scope and are referred to as *project properties*.

The Property Set mechanism facilitate helps you to manage code generation properties. it allow you to create a named set of properties whose default values control the code generation for a specific model element. For example, you could establish a class property set whose property settings inhibit automatic generation of all operations. Assigning this property set to selected classes in your model is much more convenient than individually setting the properties of each of these classes. Code generation property sets are described in more detail later in this chapter.

## Accessing the Property Editor

**To access the Property Editor:**

1. Display a diagram that contains an icon representing a model component.
2. Select the model component in the diagram, or click on a clear space in the diagram (no icon selected) to modify the project.
3. Click **Tools > Model Properties > Edit**. Rational Rose displays the Code Generation/C++ tab for the selected model component's specification. If you did not specifically select a model component, Rational Rose takes you to the **C++** tab on the **Options** dialog box.

If a model component is not selected in the diagram, the Type drop-down box lists the individual components.

The Set field lists the property set that is currently attached to the model component. The drop-down box contains the names of all currently-defined property sets for the selected component kind. If a collection of components was selected and a variety of property sets are related to those components, this field remains blank. If a collection of components was selected and each component is attached to the same property set, the name of that property set is displayed in the Set field. This field can be disabled if property sets do not exist for the selected component type.

The Model Properties section of the dialog box lists the available code generation properties and their values.

## Display or Edit Properties

**To display or edit properties:**

1. Display a diagram that contains an icon representing a model component.
2. Select the model component in the diagram.
3. Open the model component's specification by double-clicking on an item in a diagram.
4. Select a diagram item and click **Browse > Specification**, or right-click on the item and click **Specification**.
5. Select the **C++** tab. The Set field displays the property set attached to the item. The properties related to the model item are displayed in the Model Properties list.
6. To modify one of the property values, select it and click on it a second time. This places the property in edit mode. A drop-down box displays all the property's values.

   ***Note:*** *The scrollbar becomes inactive while in edit mode. To use the scrollbar, select either* ***OK*** *or* ***Cancel*** *to switch the setting to view mode and enable scrolling.*

7. To complete the edit, click outside the edit box.
8. Click **OK** or **Apply** to commit the changes to the item.

Properties that are specified explicitly by the item, and hence override the attached property set value, are drawn in normal text. Properties that have been changed since the last apply are indicated by an asterisk in the left column.

***Note:*** *Changes made to a property are accepted whenever you activate any control in the editor. For example, after editing a property, you can select another property to both accept the changes to the original property and begin editing the newly selected property.*

## Remove an Overriding Item-Level Property

Editing a property in the **C++** tab of a specification automatically makes it an overriding item-level property. To remove the overriding value from the item and return to the default value:

1. Select the property(s) and click **Default**.
2. Click **OK** or **Apply** to commit the changes to the item.

## Force a Property to be Item Specific

**To force a property to be item specific:**

1. Select the property(s) and click **Override**.
2. Click **OK** or **Apply** to commit the changes to the item.

## Reinstate the State and Value of the Last Committed Change

**To reinstate the state and value for the last committed change:**

1. Select the property(s) and click **Revert**.

## Attach a Property Set

**To attach a property set to a single component or a collection of components:**

1. Display a diagram that contains an icon representing a model component.
2. Select the model component in the diagram.
3. Open the model component's specification by double-clicking on an item in a diagram.
4. Select a diagram item and click **Browse > Specification**, or right-click on the item and click **Specification**.

5. Select the **C++** tab. The property set attached to the item is displayed in the Set field. The Model Properties list displays the properties of the model item.
6. Select a different property set from the Set list box.

   Commits are made as you move from page to page. Also, as you move from set to set or type to type within the set-level property page, any changes you have made to the currently displayed set are committed.

## Display or Edit a Specific Property Set

To display or edit a specific property set:

1. Select the component from the diagram. If you are selecting a collection of components, ensure that all the components are of the same type. A warning displays if you select different model components.
2. Click **Tools > Model Properties > Edit**, or press the **F4** key. The code generator displays the **C++** page of the **Options** dialog box. The kind of model item chosen is displayed in the Type field.
3. Select the property set name in the Set list box to display all the properties and values.
4. Modify property set values by following instructions to edit a specific property set, as previously listed.
5. Click **Apply** or **OK** to accept your changes.

   ***Note:*** *Changes made to a property are accepted whenever you activate any control in the editor. For example, after editing a property, you can select another property to both accept the changes to the original property and begin editing the newly selected property.*

## Create a New Property Set

**To create a new property set:**

1. Select a property set from the Set list box with which to base your new property set.
2. Click **Clone**.
3. Enter the new property set name in the dialog box and click **OK**. A new property set is created as a copy of the current property set.

4. Modify property set values by following instructions to edit a specific property set, as previously listed.

## Delete a Property Set

**To delete a property set:**

1. Select a property set from the Set list box.
2. Click **Remove** to delete the property set from the model. An attempt is made to find all the components in the model that reference that set and change those components to reference the default property set.

## Managing Property Sets

Whenever you create a new Rational Rose model, the C++ code generator loads its default property sets from a specially-formatted *property file* named `rosecpp.pty`. This file is created during installation in the directory that contains the Rational Rose C++ executable.

If you modify this default property set for your model you can save and reuse the modified set in several ways. You can:

- Designate a property set as a controlled unit to support multi-user development.
- Export your property sets to a file for use in other models.
- Replace your current model's property sets with those contained in a previously exported file.
- Add properties from exported property sets to the current model.
- Specify a property file from which property sets are loaded whenever a new model is created.

**To designate a property set as a controlled unit:**

***Note:*** *A Controlled Unit is any package (or file) that can be loaded and saved independently and that is subject to revision control within a configuration management system.*

1. Click **Browse > Units** to display the **Units** dialog box.
2. Click **Others** in the **Group** frame.
3. Click **<properties>** in the **Other Units** list.
4. Click **Control**, and specify a file for persistent storage using the **Filename For** dialog box.

**To export your property sets to a file for use in other models:**

1. Click **Tools > Properties > Export Properties**.
2. Specify a destination file using the **Export Properties** dialog box.

**To replace your current model's property sets with those contained in a previously exported file:**

1. Click **Tools > Properties > Import Properties.**
2. Select a source file using the **Read Petal** dialog box.

**To add properties from exported property sets to the current model:**

***Note:*** *The properties you are adding must not be present in the model.*

1. Click **Tools > Properties > Add Properties.**
2. Select a source file using the **Add Properties** dialog box.

**To specify a property file from which property sets are loaded whenever a new model is created:**

1. Click **Tools > Properties > Set Default Properties File.**
2. Select a property file using the **Set Default Property File** dialog box.

The property sets in this file initialize new models, rather than those in `codegen.pty`.

## Class Properties

Class properties control the code generated for each selected class. Refer to *Appendix B* for more detailed information on each property.

### Implementation

Use the ImplementationType property to implement a class as an elemental data type instead of as a C++ class.

## Standard Operations

Use the following code generation properties to specify the kinds of standard operations you want the C++ code generator to generate in the declaration and definition of a class:

***Table 1 Standard Operations***

| To generate: | Use these properties: |
|---|---|
| Data members | CodeName<br>ImplementationType |
| Default Constructor | GenerateDefaultConstructor<br>DefaultConstructorVisibility<br>InlineDefaultConstructor |
| Copy Constructor | GenerateCopyConstructor<br>InlineCopyConstructor<br>CopyConstructorVisibility |
| Destructor | GenerateDestructor<br>InlineDestructor<br>DestructorVisibility<br>DestructorKind |
| Assignment Operation | GenerateAssignmentOperation<br>InlineAssignmentOperation<br>AssignmentVisibility<br>AssignmentKind |
| Equality Operations | GenerateEqualityOperations<br>InlineEqualityOperations<br>EqualityVisibility<br>EqualityKind |
| Relational Operations | GenerateRelationalOperations<br>InlineRelationOperations<br>EqualityVisibility<br>EqualityKind |
| Storage Operations | GenerateStorageMgmtOperations<br>InlineStorageMgmtOperations<br>StorageMgmtVisibility |
| Subscript Operation | GenerateSubscriptOperation<br>SubscriptVisibility<br>SubscriptKind<br>SubscriptResultType |

| To generate: | Use these properties: |
|---|---|
| Dereference Operation | GenerateDeferenceOperation<br>InlineDeferenceOperation<br>DeferenceVisibility<br>DereferenceKind<br>DereferenceResultType |
| Indirection Operation | GenerateIndirectionOperation<br>InlineIndirectionOperation<br>IndirectionVisibility<br>IndirectionKind<br>IndirectionResultType |
| Stream Operation | GenerateStreamOperation<br>StreamVisibility |

In the above table, Visibility can be public, protected, private or implementation, and Kind can be common, virtual, abstract, or friend.

## User-Defined Operation Properties

Operation properties control the code generated for the user-defined operations in each selected class. Refer to *Appendix G* for more detailed information on each property.

When generating code for a class, the code generator produces a skeletal member function for each user-defined operation enumerated in the class specification:

- Use the EntryCode operation property to specify code or comments to be inserted at the beginning of the generated member function body; this property is useful for inserting instrumentation, or adhering to documentation standards.
- Use the ExitCode operation property to specify code or comments to be inserted at the end of the generated member function body; this property is useful for inserting instrumentation, or adhering to documentation standards.
- Use the OperationIsConst operation property to add the `const` keyword to the generated member function declaration.
- Use the OperationKind operation property to specify whether the operation is `common`, `virtual`, `abstract`, or `friend`.

- Use the CodeName operation property when you want the class to be named differently than it is in the Rational Rose model.
- Use the Inline operation property to specify whether to inline an operation.

## Module Specification Properties

Use the following properties to control the code generated for a module specification. Refer to *Appendix F* for more detailed information on each property.

***Table 2 Module Specification Properties***

| To specify: | Use these properties: |
|---|---|
| Commented information | CMIdentification<br>CopyrightNotice |
| Module file name | FileName<br>Generate |
| Preprocessor directives | InclusionProtectionSymbol<br>AdditionalIncludes<br>IncludeBySimpleName<br>InliningStyle |

## Module Body Properties

Use the following properties to control the code generated for module bodies. Refer to *Appendix F* for more detailed information on each property.

***Table 3 Module Body Properties***

| To specify: | Use these properties |
|---|---|
| Commented information | CMIdentification<br>CopyrightNotice |
| Module file name | FileName<br>Generate |
| Preprocessor Directives | AdditionalIncludes<br>IncludeBySimpleName<br>InliningStyle |

## Component Package Properties

Use the Directory subsystem property to specify the pathname of the directory in which the code generator places source code files generated from modules assigned to a subsystem. Refer to *Appendix J* for more detailed information on the Directory property.

## Project Properties

Project Properties control various aspects of code generation that apply to an entire model rather than to specific kinds of model components in the model, such as classes. Refer to *Appendix H* for more detailed information.

### Code Location

Use the Directory project property to specify the root directory into which generated header and implementation files are placed.

### File Name Extensions

Use these properties to control the file name extensions when constructing the names of header files, implementation files, backup header files, and backup implementation files:

***Table 4 File Name Extensions***

| **To set extensions for:** | **Use these properties:** |
|---|---|
| Header files | HeaderFileBackupExtension<br>HeaderFileExtension<br>HeaderFileTemporaryExtension |
| Implementation files | CodeFileBackupExtension<br>CodeFileExtension<br>CodeFileTemporaryExtension<br>FileNameFormat |

### Specifying Compiler-Specific Alternatives

You use the following project properties to permit or suppress code constructs that are supported by some C++ compilers:

- AllowTemplates
- AllowProtectedInheritance

## Specifying the Boolean Type

Use the BooleanType project property to specify the return value type for Boolean functions.

## Specifying Container Classes for Associations

Use the following project properties to specify the default container classes that the C++ code generator uses when generating data members for associations. Container classes generate data members for associations with certain combinations of cardinality and containment.

***Table 5 Specify Container Class***

| To specify containers for: | Use these properties: |
|---|---|
| By-value relationship | OptionalByValueContainer<br>OneByValueContainer<br>FixedByValueContainer<br>UnorderedFixedByValueContainer<br>BoundedByValueContainer<br>UnorderedBoundedByValueContainer<br>UnboundedByValueContainer<br>UnorderedQualifiedByValueContainer |
| By-reference relationships | FixedByReferenceContainer<br>UnorderedFixedByReferenceContainer<br>OneByReferenceContainer<br>OptionalByReferenceContainer<br>UnboundByReferenceContainer<br>UnorderedUnboundedByReferenceContainer<br>QualifiedByReferenceContainer<br>UnorderedQualifiedByReferenceContainer |

## Controlling Error Behavior

You use the following project properties to specify how the C++ code generator responds to errors:

- StopOnError
- ErrorLimit

Use the CreateMissingDirectories project property to specify whether the C++ code generator must generate code in existing directories.

## Class Attribute Properties

When generating code for a class, the C++ code generator produces a data member for each class attribute. A common practice in C++ is to make data members private and to provide get and set member functions to read and modify member data. By default, the C++ code generator produces a basic implementation for get and set member functions.

You use the following code generation properties to control the code generated for data members and their corresponding get and set member functions. Refer to *Appendix B* for more detailed information on each property.

***Table 6 Class Attribute Properties***

| **To control code for:** | **Use these properties:** |
|---|---|
| Data members | GenerateDataMember<br>CodeName<br>DataMemberName<br>DataMemberVisibility |
| Get and set operations | GenerateGetOperation<br>GenerateSetOperation<br>GetSetKinds<br>GetName<br>SetName<br>InlineGet<br>InlineSet<br>GetIsConstant<br>GetSetByReference<br>SetReturnsValue |

## Association Properties

When generating code for a class, the C++ code generator produces a data member for each role. The properties on a role control the code generated to support traversal in one direction. Traversal from a client class to a supplier class is controlled by properties on the role on the supplier end.

A common practice in C++ is to make data members private and to provide get and set member functions to read and modify member data. By default, the C++ code generator produces a basic implementation for get and set member functions.

The C++ code generator implements association roles only if they are marked navigable, regardless of their properties.

There are two cases of association traversals. One case implements an association with an association class, and the other implements an association without an association class. If the case involves an association without an association class, the data member and get and set functions implement traversal from each client class directly to its supplier in each navigable direction. If the association traversal includes an association class, the data member and member functions in each client class support traversal to the association class, and the data members and member functions in the association class support traversal from the association class to each supplier class in each navigable direction.

***Note:*** *There is no navigability property in Rational Rose that determines navigability from a client class to its association class. These traversals are always implemented.*

You use the following code generation properties to control the code generated for data members and their corresponding get and set member functions. Refer to *Appendix I* for more detailed information on each property.

***Table 7 Association Properties***

| To control code for: | Use these properties: |
|---|---|
| Data members | NameIfUnlabeled<br>GenerateDataMember<br>CodeName<br>DataMemberName<br>DataMemberVisibility* |
| Get and set operations | GenerateGetOperation<br>GenerateSetOperation<br>GenerateQualifiedGetOperation<br>GenerateQualifiedSetOperation<br>GetSetKinds<br>GetName<br>SetName<br>QualifiedGetName<br>QualifiedSetName<br>InlineGet<br>InlineSet<br>InlineQualifiedGet<br>InlineQualifiedSet<br>GetIsConst<br>QualifiedGetIsConst<br>GetSetByReference<br>SetReturnsValue<br>QualifiedSetReturnsValue |
| Header File Inclusion | ForwardReferenceOnly |

## Property Sets

Code generation properties are grouped by the model component to which they apply. A set of property values for each property in such a group is referred to as a *property set*. For example, a class property set contains property values for all class properties.

The C++ code generator provides a default property set for each kind of model component, and attaches the appropriate default property set to a component when it is created. To change a component's code generation properties:

- Directly modify one or more of its property values
- Modify one or more property values in its attached default property set
- Attach a named property set, setting of all of the component's code generation properties to the values in that property set

### Project Property Sets

Rational Rose provides three project property sets:

- Compiler 2.1
- Compiler 3.0
- Default

The Compiler 2.1 project property set contains reasonable property values for use with a C++ Version 2.1 compiler, one that implements neither templates nor protected inheritance. The Compiler 3.0 project property set contains reasonable property values for use with a C++ Version 3.0 compiler, which implements both templates and protected inheritance. The Default project property set contains the same property values as 3.0, with the addition of UseMSVC and CommentWidth properties.

## Model-to-Code Correspondences

The C++ code generator uses the specifications and code generation properties of components in the current model to produce C++ source code. For each class in a Rational Rose model, the C++ code generator produces a corresponding C++ class. Class associations—those representing aggregation and those representing attributes—are translated to data members of the class.

The C++ code generator produces member functions for three kinds of operations: standard operations, get and set operations for data members, and user-defined operations. For get and set operations, the C++ code generator produces member functions with basic implementations. For standard and user-defined operations, the C++ code generator produces skeletal member functions; you must edit the generated code to add the member function bodies.

When the C++ code generator produces source code files for classes in a Rational Rose model, they are stored in a directory determined by the value you specify for the Directory project property. Component packages in a Rational Rose model are mapped to subdirectories.

## Code Generated for Classes

Code is generated for classes in both a header file and an implementation file.

### In the Header File

For each class, the C++ code generator produces the following code constructs in the header file—unless you have assigned the class to a module body (in which case these constructs are placed in the implementation file):

- Class Annotations extracted from the class specification.
- A Class definition and base list, taken from the class' name and generalization relationships.
- A Class Access specification for public, protected, and private declarations that are generated for the class.
- Code regions within each section of a class declaration (public, protected, private, and implementation), allowing you to provide auxiliary declarations.
- Member function declarations for:
    - User-defined operations defined in the class specification.
    - Standard operations as specified by the class properties.
    - Get and set operations provided for data members generated from the class associations.
- Class `friend` declarations for the class' friends.

### In the Implementation File

For each class, the C++ code generator produces the following code constructs in the implementation file:

- Declarations of static data members, taken from the class associations.
- A Code region that allows you to provide auxiliary declarations and definitions associated with the class.
- Skeletal member function bodies for standard operations.
- Skeletal member function bodies for user-defined operations.
- Minimally-implemented member function bodies for get and set operations provided for data members generated from the class associations.

### Class Definition and Base List

Following the annotations for a class, the C++ code generator produces the class definition. The head of the definition is the keyword class followed by the name you gave the class in the Rational Rose model.

The class definition can include a base list that contains one entry for each of the class' inheritance relationships. Each entry in the base list can include additional keywords to indicate visibility and whether or not the inheritance is virtual. In a Rational Rose model, visibility and virtual inheritance are specified in the specification for the inheritance relationship.

### Class Access Specification

Within the class definition body, the C++ code generator produces declarations for the class data members and member functions, grouped according to their access. The C++ code generator produces the appropriate access keyword preceding each of four separate access levels:

```
{
   public:
   // Public members...
   protected:
   // Protected members...
   private:
   // Private members...
   private:  // implementation
```

```
    // Private implementation members...
};
```

The Rational Rose notation has four levels of visibility, while C++ has only three. The private implementation level is realized as a second private level. By default, all data members are generated in this part. While there is nothing to prevent a friend function from accessing private implementation members directly, the intent of private implementation access is to indicate that members should never be accessed directly outside the class, even by friends.

The C++ code generator determines the direct access of a class member using information from specifications and code generation properties.

The access of data members is determined by the value of the association DataMemberVisibility property. By default, data members are generated in private implementation.

The access of get and set operations for data members is determined by the value of the Visibility field in the corresponding Association Specification. By default, member functions for get and set operations are generated with public access.

The access of each standard operation member function is determined by a class property with the name “standardopVisibility” where standardop is the name of the standard operation. By default, member functions for standard operations are generated with public access.

The access of each user-defined operation is determined by the value of the Visibility field in the operation’s specification. By default, member functions for user-defined operations are generated with public access.

## Class Friend Declarations

After the private implementation members, the C++ code generator produces declarations for friends of the class. A friend declaration is generated for any relationship in which the class is the supplier and the Friendship Required field in the relationship specification is set to True. The name of the friend is the name of the client in the relationship.

In the following example, class `Rectangle` is declared as friend of class `Screen` and has access to private members of `Screen`.

```
class Rectangle : virtual public Shape
{
   public:
   // Public members...
   protected:
   // Protected members...
   private:
   // Private members...
   private:  // implementation
   // Private implementation members...
   friend class Screen;
};
```

## Code Generated for Class Attributes

By default, a class attribute is represented in code as a data member and a pair of get and set member functions for accessing the data member. The default code that is generated consists of:

- A private implementation data member whose type specified in the Rose model and whose name is based on the DataMemberName value.
- A pair of private get and set member functions whose names are based on the GetName and SetName values, respectively.

The C++ code generator declares the data member and the get and set member functions in the class definition. If the class has not been assigned to a module, this code is generated in the header file; if the class has been assigned to a module, this code is generated in the implementation file.

The C++ code generator produces the definitions for the get and set member functions in the implementation file using a form of the code region. These code regions specify that the definition source code is to be regenerated each time code generation is invoked. If you modify a definition and wish this modification to be preserved in subsequent iterations, change the `preserve` argument in the code region's `begin` annotation from its default setting of `no` to `yes`.

A number of factors affect the actual code that is generated for a class attribute:

- The GenerateDataMember property controls whether or not a data member is generated.
- The type of the data member is affected by:
  - The type specified in the Rational Rose model.
  - The containment of the attribute.
- The `static` adornment generates the `static` keyword for the data member.

An initial value can be specified for the attribute in the Rational Rose model:

- Visibility adornments and properties affect access of the data member and its get and set operations.
- The get and set operations for the data member are affected by:
  - The GenerateGetOperation and GenerateSetOperation properties, which control whether the member functions are generated.
  - The GetSetKinds, GetIsConst, GetSetByReference, and SetReturnsValue properties, which control other details about the member functions.
- If the type of the attribute references declarations of other classes, you must draw an association relationship from the class containing the attribute to the referenced class. This ensures that the referencing and referenced classes are in the correct order if they are in the same module, or that the appropriate include gets generated if they are not. In the latter case, the same effect can be achieved by adding a module dependency relationship to the appropriate module diagram. A module dependency relationship is also required if the attribute references a declaration in the declarations or additional declarations protected code region of another module.

## Code Generated for Class Utilities

For a class utility in a Rational Rose model, the C++ code generator produces the same header and implementation code as a class. However, since class utilities do not have instances, all operations for a class utility are static.

Though class property sets can be attached to class utilities, many of the code generation properties for classes are invalid for class utilities. For example, if you set the GenerateDestructor property to True for a class utility and then generate code, the C++ code generator displays an error message in the log.

The C++ code generator produces the following code constructs in the header file for the class utility:

***Note:*** *This code is generated in the implementation file if you have assigned the class utility to a module body.*

- The class definition, including the class name and base list.
- Declarations for static member functions from the operations listed in the class specification. Operations need not be marked `static` in their specification.
- Declarations for static data members.

The C++ code generator produces the following code constructs only in the implementation file:

- Definitions for static data members.
- Skeletal definitions for static member functions.

## Code Generated for Parameterized Classes, Instantiated Classes and Class Utilities

The generation of code for parameterized and instantiated classes and utilities depends on the value of the AllowTemplates project property. This property controls whether the C++ code generator produces templates for parameterized classes. This is significant because some C++ compilers do not support templates.

If the AllowTemplates project property is set to False, parameterized and instantiated classes and class utilities are not supported. If you try to generate a parameterized or instantiated class or class utility, the C++ code generator displays an error message in the log.

If the AllowTemplates project property is set to True, the C++ code generator produces code for parameterized and instantiated classes and class utilities.

## Parameterized Classes

A class template definition is generated for each parameterized class or class utility. For example, a parameterized class with the name Stack generates:

```
template <class T>
class Stack { }
```

To specify template parameters, also called arguments—in this example, `class T`—you enter them in the Attributes field in the specification for the parameterized class (right-click in the Attributes list and click **Insert**, then enter the arguments in the Name field). As in C++, you specify a type-valued parameter using the typename class. Every parameterized class must have at least one parameter.

The C++ code generator produces data members and member functions as for a class.

## Instantiated Classes

An instantiated class or class utility generates a `typedef` in the header file for the instantiated class (or in the implementation file, if you have assigned the class to a module body). For example, the C++ code generator generates the following `typedef` for `Int_Stack`, an instantiation of the parameterized class `Stack`:

```
typedef Stack<int> Int_Stack;
```

To specify arguments for an instantiated class, enter them in the Parameters field in the specification for the instantiated class (right-click in the Attributes list and click **Insert**, then enter the arguments in the Name field).

The C++ code generator produces data members and member functions as for a class. The C++ code generator ignores any operations in an instantiated class that are not defined in the corresponding parameterized class.

## Parameterized and Instantiated Class Utilities

The code generated for parameterized and instantiated class utilities is the same as that for parameterized and instantiated classes, except that member functions are generated as `static`.

## Code Generated for Association Relationships

An association is a relationship among two or more classes. Code can be produced to efficiently traverse the relationship in neither, one, or both directions.

The ends of each association are called a role. Roles can be labeled with an identifier that describes the role that an associate class plays in the association. A role has both Rational Rose model and code generation properties that affect the generated code which traverses to that role. For example, marking a role navigable means that traversal from the opposite role's class to this role's class is to be implemented. Relative to a given direction of traversal, the roles can be designated as the client role and the supplier role, respectively. Of course, these designations are exchanged when considering traversal in the opposite direction.

By default, an association relationship is represented in code as a data member and a pair of get and set member functions for accessing the data member in each client class whose corresponding supplier role is marked navigable. In the simplest case, when each client class is associated with exactly one supplier class, the default code that is generated consists of:

- A private implementation data member whose type is the supplier class and whose name is based on the DataMemberName value.
- A pair of public get and set member functions whose names are based on the GetName and SetName values, respectively.

The C++ code generator declares the data member and the get and set member functions in the class definition.

The C++ code generator produces the definitions for the get and set member functions in the implementation file using a form of code region; these code regions specify that the definition source code is to be re-generated each time the code generator is invoked. If you modify a definition and wish this modification to be preserved in subsequent iterations, change the preserve argument in the code region's begin annotation from its default setting of `no` to `yes`.

A number of factors affect the actual code that is generated for an association relationship:

- The GenerateDataMember property controls whether or not a data member is generated.
- The type of the data is affected by:
  - The value of the ContainerClass and QualifiedContainer association-role properties of the supplier role of the relationship.
  - The ordered constraint of the supplier role.
  - The qualifiers of the supplier role.
  - The cardinality and containment adornments of the association relationship.
  - The value of the ImplementationType class property for the supplier class.
- The static adornment generates the `static` keyword for the data member.
- Visibility adornments and properties affect access of the data member and its get and set operations.
- The get and set operations for the data member are affected by:
  - The GenerateGetOperation, GenerateSetOperation, GenerateQualifiedGetOperation, GenerateQualifiedSetOperation, InlineGet, InlineSet, InlineQualifiedGet, and the InlineQualifiedSet properties, which control whether the member functions are generated.
  - The qualifiers of the supplier role.
  - The GetSetKinds, GetIsConst, GetSetByReference, SetReturnsValue, and QualifiedSetReturnsValue properties, which control other details about the member functions.
- The ForwardReferenceOnly property controls whether a `#include` directive or a forward declaration is generated for the supplier class of the association relationship.

There are two pairs of get and set operators:

- The unqualified get and set operators allow you to get or set the entire container of supplier objects associated with a single client object.

- The qualified get and set operators allow you to get or set the container of supplier objects associated with a single client object and a fixed value for each of the supplier role’s qualifiers.

## The Association Class

Normally, the get and set operations in a client class implement traversal directly from a client object to supplier objects. If there is an association class, these operations implement traversal to zero or more association class objects. The association class has data members and get and set operations which implement traversal from its objects to an object of each associate class.

The properties which affect the data members and get and set operations generated in the association class are:

- AssocClassContainer
- GenerateAssocClassDataMember
- GenerateAssocClassGetOperation
- GenerateAssocClassSetOperation
- AssocClassGetIsConst
- AssocClassSetReturnsValue
- InlineAssocClassGet
- InlineAssocClassSet

## Code Example of an Association Relationship

This example shows data member `width` of class `Rectangle` and its corresponding get and set member functions.

The C++ code generator produces the data member and declarations of the get and set member functions in the header file for a class.

***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body:*

```
public:
...
   // Get and Set Operations for Associations
   const int get_width() const;
   void set_width(const int value);
...
private: // implementation
// Data Members for Associations
...
//##begin Rectangle::width:$:assoc%.has preserve=no
   int width;
//##end Rectangle::width:$:assoc%.has
...
```

The C++ code generator produces the definitions for the get and set member functions in the implementation file:

```
// Get and Set Operations for Associations
const int Rectangle::get_width() const
{
//##begin Rectangle::get_width%.get preserve=no
   return width;
//##end Rectangle::get_width%.get
}
void Rectangle::set_width(const int value)
{
//##begin Rectangle::set_width%.set preserve=no
   width = value;
//##end Rectangle::set_width%.set
}
```

## Code Generated for Generalization/Inheritance Relationships

When the C++ code generator produces a definition for a class (the client) that inherits from another class (the supplier), the name of the supplier class is included in the base list of the class definition. The access of the inheritance's relationship and whether or not the inheritance is virtual are determined by information in the inheritance's relationship specification.

When generating code for the client class, the C++ code generator also generates a `#include` directive referencing the file that contains the supplier class. If the definition of the client class is generated in a header file, the `#include` directive is generated in the header file; otherwise, it is generated in the implementation file.

Some C++ compilers do not support protected inheritance. You can control the access that the C++ code generator produces for protected inheritance relationships by setting the AllowProtectedInheritance project property.

## Code Generated for Instantiates Relationships

An instantiates relationship documents the relationship between an instantiated class and the parameterized class from which it is instantiated. The generation of code for parameterized and instantiated classes and utilities depends on the value of the AllowTemplates project property. This property controls whether the C++ code generator produces templates for parameterized classes. This is significant because some C++ compilers do not support templates.

If the AllowTemplates project property is set to False, parameterized and instantiated classes and class utilities are not supported. If the AllowTemplates project property is set to True, the C++ code generator produces a class template definition for each parameterized class or class utility.

For an instantiates relationship, the C++ code generator produces a typedef in the header file for the instantiated template.

***Note:*** *If you have assigned the class to a module body, this code is generated in the implementation files.*

## Code Generated for Cardinality and Containment Adornments for Associations

When a data member is generated for an association, and the ContainerClass property for that relationship is empty (the default), the C++ code generator looks at the supplier cardinality, the kind of containment of the association, and the presence or absence of the ordered constraint to determine what kind of data member to generate. (The client cardinality of the association is ignored.) In general:

- The cardinality of the supplier in the relationship determines the data member type. By default:
  - A supplier cardinality of 1 generates a simple field.
  - A supplier cardinality greater than 1 usually requires an array or container class.
- The relationship type determines whether the data member holds the actual data or pointers to the data:
  - An aggregation relationship generates a “by-value” data member—the data member contains instances of the supplier class.
  - A non-aggregation relationship generates a “by-reference” data member—the data member contains pointers to instances of the supplier class.

  ***Note:*** *If you assign containment model properties that do not conform to the above, they are ignored. For example, assigning the* OneByReference *property to an aggregation relationship.*

The following table summarizes the properties that determine the types generated for the various combinations of supplier cardinality, containment, and ordering. Most of these types either are, or are derived from, the supplier class (T). Other types are container classes provided by project properties (click on the container class names in the table to see which properties generate them).

***Note:*** *You can use the default container classes if you provide implementations for them. Otherwise, you should set the appropriate properties to specify container classes of your own.*

***Table 8 Combination of Supplier Cardinality, Containment and Ordering***

| Supplier Cardinality | Order Constraint | By Value (aggregation relationships) | By Reference (non-aggregation relationships) |
|---|---|---|---|
| 1 | yes | OneByValueContainer | OneByReferenceContainer |
| 0..1 | yes | OptionalByValueContainter | OptionalByReferenceContainer |
| Exact Number, >1 | yes | FixByValueContainer | FixedByContainerReference |
| Exact Number, >1 | no | UnorderedFixByValueContainer | UnorderedFixByReferenceContainer |
| Bounded Range | yes | BoundedByValueContainer | BoundedByReferenceContainer |
| Bounded Rante | no | UnorderedBoundedByValueContainer | UnorderedBoundedByReferenceContainer |
| UnboundedRange | yes | UnboundedByValueContainer | UnboundedByReferenceContainer |
| UnboundedRante | no | UnorderedUnboundedByValueContainer | UnorderedUnboundedByReferenceContainer |
| Qualified | yes | QualifiedByValueContainer | QualifiedByReferenceContainer |
| Qualified | no | UnorderedQualifiedByValueContainer | UnorderedQualifiedByReferenceContainer |

For example, set the UnboundedByReferenceContainer property to `MyHeap`. Then, whenever the C++ code generator encounters a relationship with by-reference containment and a supplier cardinality of `n`, the generated data member has the type `MyHeap`.

## Code Generated for Standard Operations

When generating code for a class, the C++ code generator can also generate skeletal member functions for one or more standard operations. The C++ code generator determines whether and how to generate member functions for standard operations from class property values. For example, the GenerateAssignmentOperation and AssignmentKind class properties determine whether or not the C++ code generator produces an assignment operation for a class and if so, its definition and visibility.

For each enabled standard operation, the code generator produces:

- A member function declaration in the header file for the class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeletal function body in the implementation file containing a code region. Complete the member function by inserting implementation code between the annotations delimiting the code region.

This example shows a member function declaration and corresponding skeletal function body for an assignment operation for class `Rectangle`:

```
// Assignment Operation
const Rectangle & operator=(const Rectangle &right);
// Assignment Operation
const Rectangle & Rectangle::operator=(const Rectangle
&right)
{
//##begin Rectangle::operator=%.body preserve=yes
//##end Rectangle::operator=%.body
}
```

**To overload a standard operation:**

1. Generate the default standard operation by setting the relevant class properties.
2. Create one or more additional operations with the same name but different parameters in the Class Specification.

## Code Generated for User-Defined Operations

When generating code for a class, the C++ code generator produces a skeletal member function for each operation listed in the class specification. For each such operation, the C++ code generator produces:

- A member function declaration in the header file for the class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeletal function body in the implementation file containing a code region. You complete the member function by inserting implementation code between the annotations delimiting the code region.

The C++ code generator uses the information in the operation's specification, as well as operation code generation properties, to generate the member function.

***Note:*** *The C++ code generator automatically generates code for skeletal member functions for standard operations, which are generated based on the values of class properties.*

Consequently, you do not need to list these operations in the class specification unless you want to overload or override them.

## Declarations Generated for User-Defined Operations

For each user-defined operation of a class, the C++ code generator produces the following code in the header file for the class:

***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body:*

- Comments generated from fields in the operation specification, such as Documentation, Time and Space Complexity, and Exceptions.
- The member function declaration, generated from fields in the operation specification, such as Name, Formal Parameters, and Return Class.

The declaration is generated under the appropriate access keyword, according to the value of the Visibility field of the operation specification.

Additional keywords, such as `static`, `virtual`, or `const` can be generated for the member function declaration based on its OperationKind and OperationIsConst values.

## Implementation Code Generated for User-Defined Operations

For each user-defined operation of a class, the C++ code generator produces the following in the implementation file:

- A member function definition, generated from fields in the operation specification, such as Name, Formal Parameters, and Result.
- Additional keywords, such as `static`, `virtual`, or `const` can be generated for the member function declaration based on its OperationKind and OperationIsConst values.
- Entry code, generated from text specified for the operation EntryCode property.
- A preserved code region for specifying the body of the member function.
- Exit code, generated from text specified for the operation ExitCode property.

## Code Generated for Modules

For each module specification or module body in a Rational Rose model, the C++ code generator produces:

- Module Annotations extracted from the module specification and module properties.
- Directives for Inclusion Protection (in header files only).
- #include Directives based on relationships in class and module diagrams, as well as module properties.
- A code region for user-specified auxiliary #include directives.
- Module Declarations, if any, extracted from the module specification.
- A code region for user-specified auxiliary global declarations.
- Class Declarations Definitions for classes assigned to the module, which include code regions as described in "Code Generated for Classes" on page 24.
- Orphan Code, if any, resulting from code regions that no longer correspond to model components due to changes in the model.

### Module Inclusion Protection

Before the `#include` statements in a header file, the C++ code generator produces a set of preprocessor directives that prevents duplicate `#include` directives in the file. This mechanism eliminates the need to compute module dependencies. The preprocessor directives are of the form:

```
# ifndef symbol
# define symbol
   ...#include directives...
   ...module contents...
# endif
```

where `symbol` is based on the module's InclusionProtectionSymbol value. A corresponding `#endif` directive is generated at the end of the file.

For example, these are the preprocessor directives generated by the C++ code generator for a module specification named Array:

```
# ifndef Array_H
# define Array_H
   ...#include directives...
   ...module contents...
# endif
```

## Module #include Directives

The code generator writes `#include` directives from up to three sources:

- Relationships in class and module diagrams
- Module properties
- A preserved code region for `#include` directives

The C++ code generator produces the `#include` directives in the header file for a class, unless you have assigned the class to a module body; in this case, the code is generated in the implementation file.

In addition, when generating an implementation file that has a corresponding header file, the C++ code generator produces a single `#include` directive for the header file in the implementation file. For example, the following code is generated in the implementation file for module-body array:

```
// array
# include "array.h"
```

The IncludeBySimpleName module specification property and IncludeBySimpleName module-body property control whether the `#include` directives that the C++ code generator produces specify the path of the included files or only the file names.

## Relationships in Class and Module Diagrams

The C++ code generator produces a `#include` directive for each header file on which the module depends.

The C++ code generator determines which header files to include in a module based on:

- Direct dependencies of the module, as shown in module diagrams.

- Generalization relationships and associations for each class that is defined in the module.

## Module Properties

The C++ code generator produces a group of `#include` directives based on text specified in the AdditionalIncludes module property.

The generated code is of the form:

```
//Additional Includes:
#include "a.h"
#include "b.h"
...
```

## Preserved Code Region for `#includes`

The C++ code generator produces a code region in which you can enter additional `#include` directives when you edit the file. The code region is delimited by annotations of the form:

```
//##begin includes preserve=yes
//##end includes
```

You can insert `#include` directives between the annotations. In this code region, include any header files that are specific to the module. The `#include` directives in the code region are preserved when the C++ code generator regenerates code.

## Module Declarations

After the `#include` directives, the C++ code generator produces any additional declarations that you have specified in the Declarations field of the module specification.

Following the declarations from the module specification, the C++ code generator produces a code region in which you can enter additional declarations when you edit the file. The code region is delimited by annotations of the form:

```
//##begin declarations preserve=yes
//##end declarations
```

You can insert additional declarations between the annotations. The declarations in the code region are preserved when the C++ code generator re-generates code.

### Module Orphan Code

If you delete or rename a model component for which code has been previously generated, the code generator treats all code regions associated with that model component as “orphaned.” Rather than deleting these code regions, the C++ code generator moves them to a special section at the end of its file.

Depending the setting of the AlwaysKeepOrphanedCode project property, the C++ code generator sets the preserve argument of these regions to `yes` or `no`. If AlwaysKeepOrphanedCode is False (the default), the preserve argument of the orphaned regions is set to `no`. This means the orphaned code in these regions is discarded the next time code is generated—you should first intercede with a text editor if you plan to salvage this code. If AlwaysKeepOrphanedCode is True, the preserve argument of the orphaned regions is set to `yes`. In this case, unless you remove the orphaned code with a text editor or change the value of AlwaysKeepOrphanedCode to False, the orphaned code is preserved through subsequent code generations.

The orphan code is of the form:

```
#if 0
//##begin code_region preserve=no
   <your implementation code>
//##end code_region
#endif
```

where `code_region` identifies the model component that was deleted or renamed.

## Code Generation and Component Packages

When the C++ code generator produces code for the classes and modules in a project, the resulting files are stored in a directory structure. The root of this directory structure is the directory specified by the Directory project property.

By default, each top-level component package in a Rational Rose model is stored in a separate subdirectory of the project directory. A nested component package is stored in a subdirectory of its parent’s component package’s directory. This is recursive; nested component packages in the model are mapped to nested subdirectories, for example, module files in subsystem `E` are stored in the subdirectory `c:\myproj\d\e`, where `c:\myproj` is the name of the project directory:

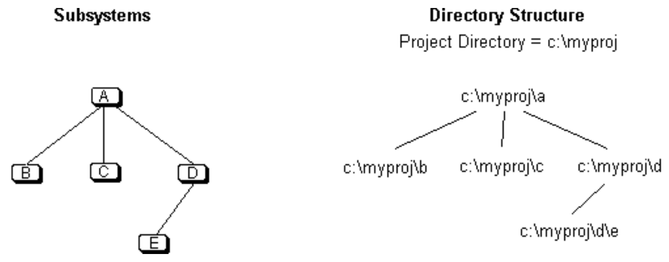


***Figure 1 Naming Project Directories***

The name of each subdirectory is determined by the value of the Directory subsystem property. By default, the subdirectory name is derived from the name of the corresponding component package.

If the CreateMissingDirectories project property is set to True, as the C++ code generator produces module files, it creates any subdirectories it needs automatically.

If, in its specification, a logical package is assigned to a particular component package, the module files that correspond to that logical package are stored in that component package. Otherwise, the C++ code generator implicitly maps the logical package to a component package by creating a subdirectory in the project file.

# *Chapter 3*

# *C++ Reverse Engineering*

Reverse engineering is the process of examining a program's source code to recover information about its design. The Rational Rose C++ Analyzer extracts design information from a C++ application's source code and uses it to construct a model representing the application's design—its logical and physical structure. You can then use Rational Rose to view and manipulate this model.

The Rational Rose C++ Analyzer is packaged as a separate executable that is invoked independently of the Rational Rose executable.

## Key Concepts

To reverse engineer a program, you must specify:

- The source files which represent the program.
- The location of any source code libraries referenced by these source files.
- How the C++ constructs in the source files should be mapped to the UML, OMT, or Booch notation.
- What diagrams should be created.

The Analyzer captures this information for a particular program in a *project* that you access whenever you wish to analyze that program. A project contains several *components*—each of which maintains a specific subset of this information. The Directory list component, for example, enumerates the directories containing the program's source files. You can name a project, save it in a file, and recall it whenever you reverse engineer the program it describes.

## An Example Program

Consider the following program, represented by two source files located in directory `c:\rose\example`. The file `propel.h` contains:

```
class engine
{
   int start();
   int throttle(int setting);
   //control speed
   int stop();
};
```

and the file `move.h` contains:

```
#include "propel.h"
class vehicle
{
   private:
      engine motive_power;
};

//for use on land
class car : public vehicle
{
};

//for use on sea
class boat : public vehicle
{
};
```

These files illustrate many of the operations described in this section. You can create this directory and these two files and try the operations.

## Creating a New Project

When you start the Analyzer, it displays its application window:

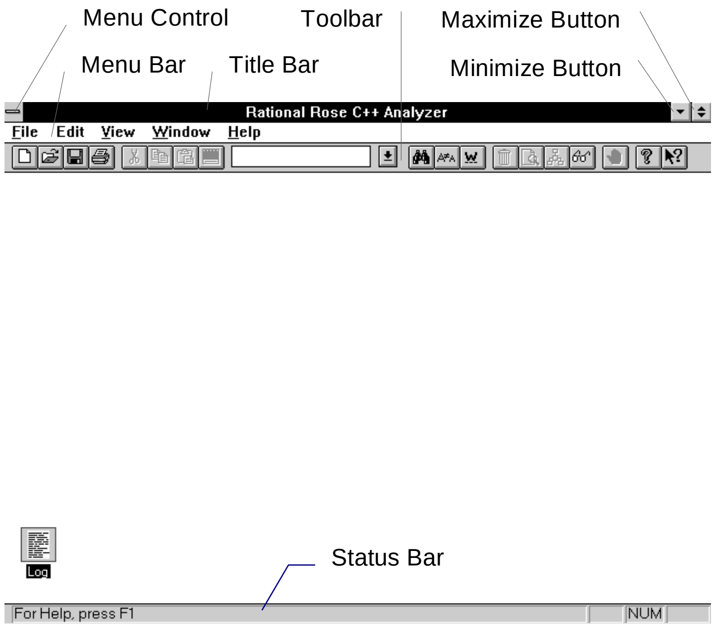
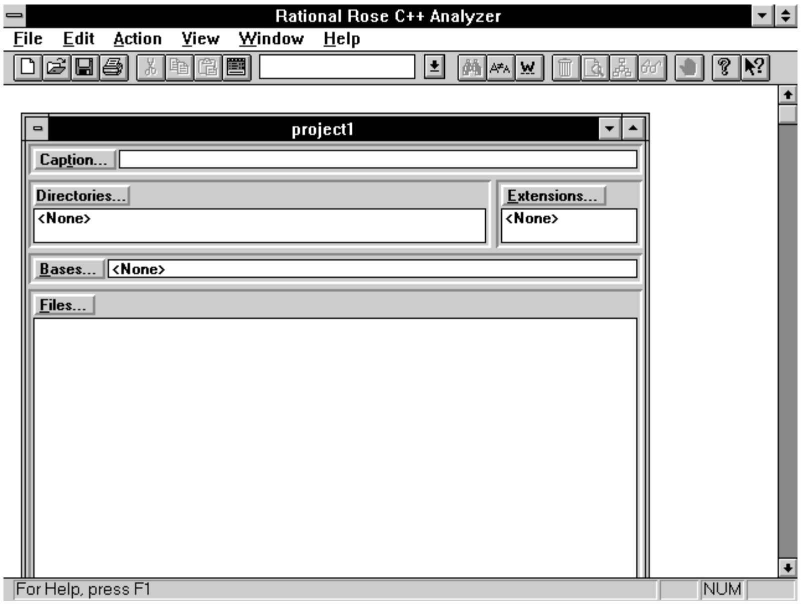


***Figure 2 Application Window***

At startup, the application window contains a menu bar, a toolbar, a status bar, and an icon representing the Analyzer's Log window. Though many of the toolbar icons are inactive at this point, letting the cursor rest on an icon briefly produces a tool tip describing the icon's function.

**To create an empty new project:**

1. While depressing the **SHIFT** key, click **File > New**. The analyzer displays a project window named **project 1** within the application window:



<None>

<None>

<None>

For Help, press F1

*Figure 3 Project Window—Simple View*

The project window is a child of the application window. The application window can simultaneously display multiple child windows using horizontal and vertical scroll bars. To display these windows, click **Window > Cascade** or **Window > Tile**. You can use this feature to compare two project windows.

2. Click **Windows > New Window** to make multiple copies of the same window. The Analyzer automatically maintains consistency between these copies. When working with a single project, click on the project window's maximize button to provide a complete display:

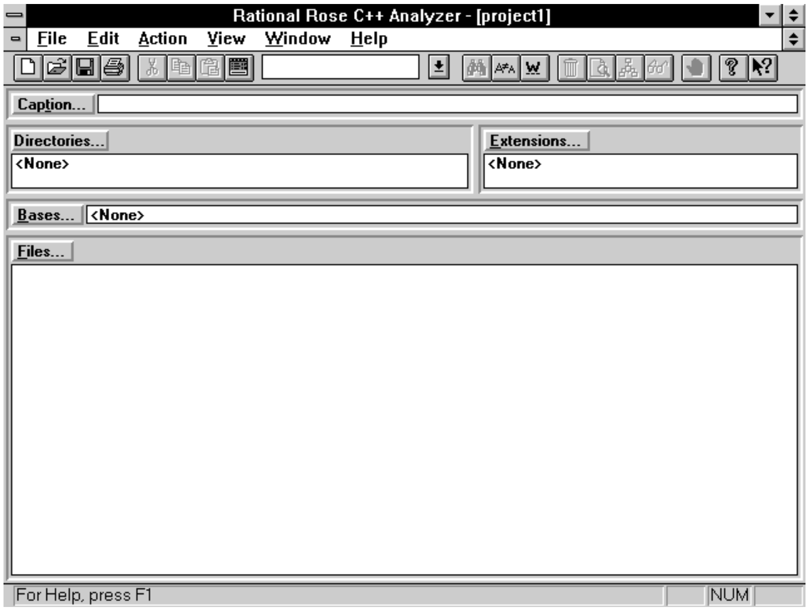
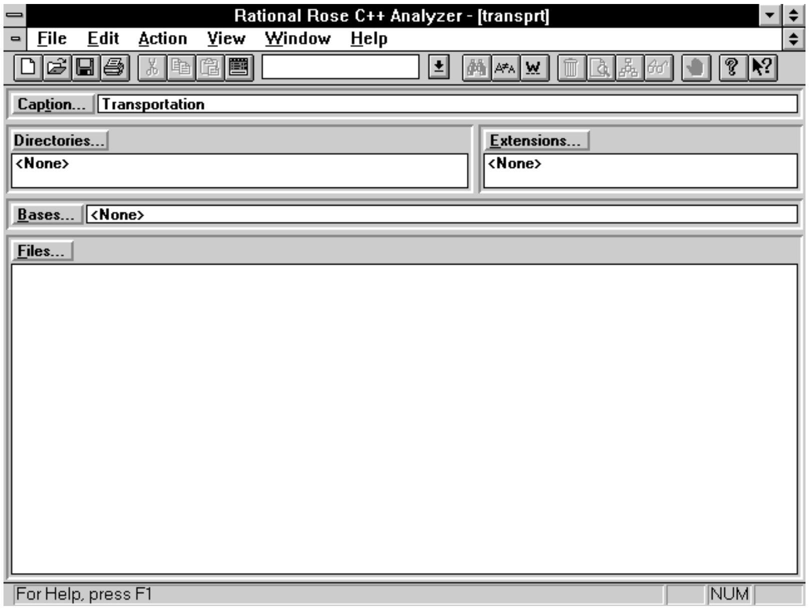


***Figure 4 Maximize a Project Window***

The Analyzer can display several different views of a project in a project window. The view shown on the previous page is the Simple view; other views are described in the Analyzer User Interface section that follows.

3. Click **Caption** and enter the descriptive caption "Transportation" in the resulting **Caption** dialog box. Click **File > Save As** and use the **Save Project As** dialog box to save the project in file `c:\rose\example\transprt.pjt`. The Analyzer updates the project window accordingly:



Caption... Transportation

Directories... <None>

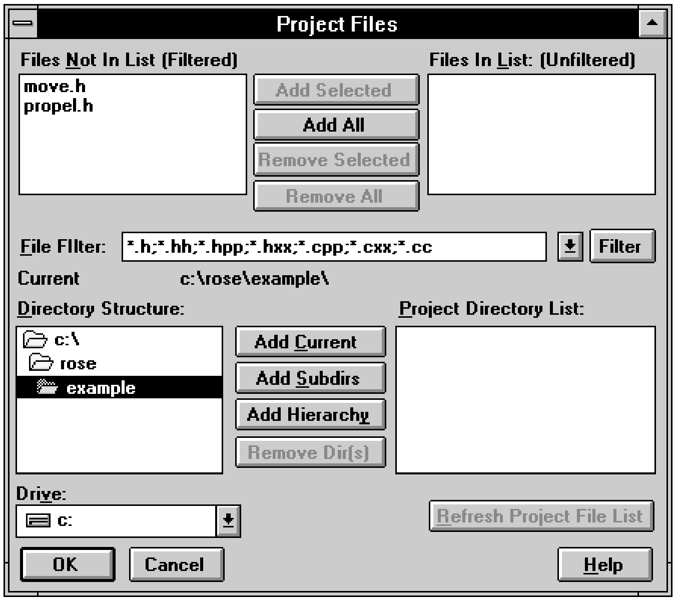
Extensions... <None>

Bases... <None>

***Figure 5 Project Window with Caption***

## Selecting Source Code to Reverse Engineer

To identify source files for reverse engineering, place them on the file list by clicking **Files** in the project window:



Files Not In List (Filtered)
move.h
propel.h

Files In List: (Unfiltered)

File Filter: *.h;*.hh;*.hpp;*.hxx;*.cpp;*.cxx;*.cc

Current c:\rose\example\

Directory Structure:
c:\
rose
example

Project Directory List:

Drive:
c:

***Figure 6   Project Files Dialog Box***

You can navigate the Directory Structure list by clicking with the mouse. Double-clicking on a directory entry both selects it and displays any subdirectories. The currently selected directory is named after the Current Directory label in the dialog box.

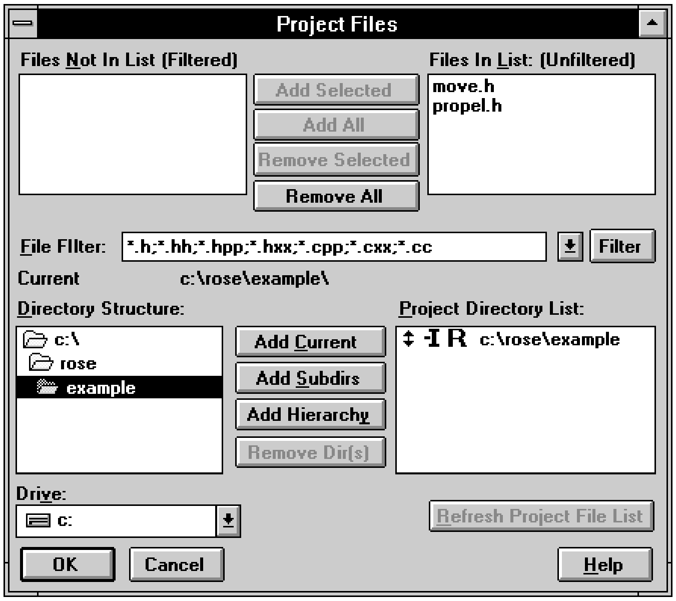
The patterns in the File Filter box determine which files in the Current directory are candidates for addition to the file list; such candidates are displayed in the Files Not In List box. Since the Analyzer's default File Filter patterns include `*.h`, the files `move.h` and `propel.h` are shown in the Files Not In List box.

To add a file shown in the Files Not In List box to the files list, click on its entry to select it, and then click **Add Selected**. The file is moved from the Files Not In List box to the Files In List box. To add all files in the Files Not In List box to the files list, click **Add All**.

To remove a file from the file list, click on its entry in the Files In List box, and then click **Remove Selected**. The file is moved from the Files In List box to the Files Not In List box. To remove all files in the Files In List box from the files list, click **Remove All**.

To move multiple files between the Files Not In List and Files In List boxes, select multiple contiguous entries in either the Files In List or Files In List list boxes by holding the shift key while clicking on the first and last entries of the selection. Clicking on an entry in either list box while depressing the **CONTROL** key switches that entry's selection status without deselecting any other entries. **Add Selected** and **Remove Selected** operate on all selected file entries.

To add both `move.h` and `propel.h` to the file list, click **Add Selected**:



Files Not In List (Filtered)

Files In List: (Unfiltered)
move.h
propel.h

File Filter: *.h;*.hh;*.hpp;*.hxx;*.cpp;*.cxx;*.cc

Current c:\rose\example\

Directory Structure:
c:\
rose
example

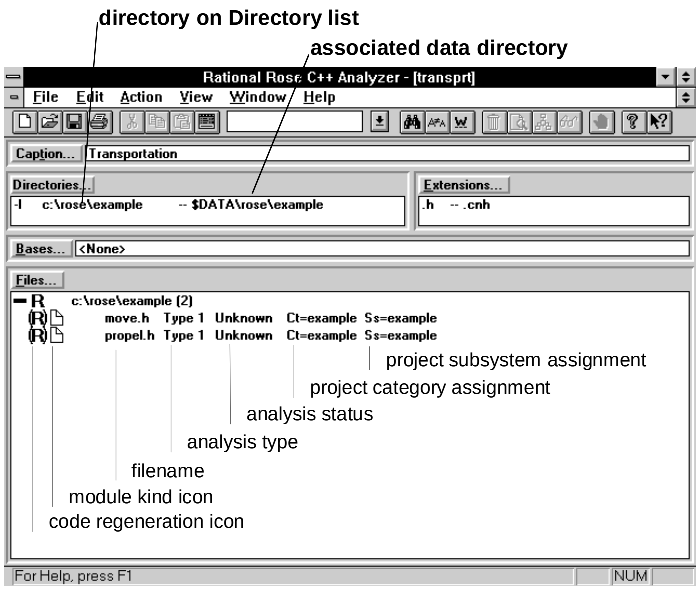
Project Directory List:
‡ I R c:\rose\example

Drive:
c:

***Figure 7 Working in the Project File Dialog Box***

***Note:*** *The directory containing* `move.h` *and* `propel.h` *has been automatically added to the project Directory list.*

Click **OK** to accept these changes. The Analyzer updates the project window accordingly:


directory on Directory list
associated data directory
Rational Rose C++ Analyzer - [transprt]
File Edit Action View Window Help
Caption...
Directories...
Extensions...
Bases...
Files...
project subsystem assignment
project category assignment
analysis status
analysis type
filename
module kind icon
code regeneration icon
For Help, press F1
NUM

```
Caption...     Transportation
Directories... -I   c:\rose\example      -- $DATA\rose\example
Extensions...  .h   -- .cnh
Bases...       <None>
Files...
- R   c:\rose\example (2)
      move.h    Type 1  Unknown  Ct=example  Ss=example
      propel.h  Type 1  Unknown  Ct=example  Ss=example
```

***Figure 8 Project Directories List***

***Note:*** *The project directory list entry contains two paths:*

`c:\rose\example`—the source directory path

`$DATA\rose\example`—the associated data directory

When the Analyzer performs semantic analysis of a source file, it creates a data file containing the results of this analysis. Using incremental compilation technology, the Analyzer uses the information in these data files to greatly reduce the time required to reverse engineer a program that was previously reverse engineered but later modified. The data file associated with a source file is stored in a data directory that the Analyzer creates and associates with the source directory containing the source file.

To facilitate team development, the Analyzer's default data directory naming pattern incorporates the virtual symbol $DATA; virtual symbols are distinguished by a leading **$** (dollar sign). Using the Analyzer's Path Map, each developer can specify their own physical path-name mappings for virtual symbols used anywhere in a project.

The procedure for establishing a Path Map entry for $DATA is described in the following subsection. You can directly specify a directory list entry's data directory using the project **Directory List** dialog box displayed when you click **Directories** from the project window. This dialog box modifies the data directory naming pattern used by the Analyzer when associating a data directory with a source directory. Procedures for utilizing the project Directory List dialog box to perform these actions are described in "Editing the Directory List" on page 106.

***Note:*** *Adding* `move.h` *and* `propel.h` *to the file list also automatically updates the project's Extension list, adding an entry for* `.h`.

The entry for each file on the file list indicates its code regeneration policy, module kind, analysis type, analysis status, project category assignment, and project subsystem assignment.

## Code Regeneration Policy

Code regeneration policy designates source files associated with existing class libraries or legacy code. Classes reverse engineered from such files should have their regeneration properties set to False, preventing Rational Rose C++ from subsequently generating code that overwrites the existing code. When you add a file to the file list, its code regeneration policy defaults to follow the policy of its parent directory, which defaults to enabled. Thus, you need only modify a file's regeneration policy if it is associated with existing code that doesn't change in subsequent round-trip iterations. See "Code Regeneration Policy" on page 103 for more information.

## Module Kind

A file's module kind identifies it as a specification (header) or body (implementation). This information is used when reverse engineering the physical part of the generated model. When you add a file to the file list, its module kind is set by its file name extension, which defaults to specification for `.h` file name extensions, and body for all others.

## Analysis Type

The analysis type designation allows the analyzer to handle source files that are context-dependent or syntactically incomplete; a source file is context-dependent if it references symbols defined in another source file without explicitly declaring a `#include` for that source file.

A source file's analysis type characterizes its context dependence and syntactic completeness:

***Table 9 Source File's Analysis Type***

| Analysis Type | Syntactically Complete | Context-Independent |
|---|---|---|
| 1 | Yes | Yes |
| 2 | Yes | No |
| 3 | No | Unknown |

When a file is added to the file list, the Analyzer designates it as Type 1. Additional information about analysis types and their management is provided in "Analysis" on page 73.

## Analysis Status

The analysis status designation indicates the Analyzer's current understanding of each source file's semantic correctness. Typical values include:

***Table 10 Analysis Status***

| Analysis Status | Semantic Correctness |
|---|---|
| Unknown | The source file has not been analyzed since being added to the File list, or has not been updated since the project was opened. |
| Analyzed | The last semantic analysis found no errors. |
| CodeCycled | The last semantic analysis found no errors, and the source file can support round-trip engineering. |
| Has Errors | The last semantic analysis found errors; these can be displayed by double-clicking on the File list entry. |

## Project Category and Subsystem Assignments

Each class and module reverse engineered from source code can be assigned to a category (logical package) and subsystem (component package), respectively. If the source code being reverse engineered was generated by the Rational Ros eC++ code generator, it contains annotations that accurately specify these assignments. If such annotations are not present, assign each source file to a category and subsystem (logical and component package). The Analyzer utilizes these assignments—referred to as the project category assignment and project component package assignment respectively—to properly assign model components extracted from a source file to the correct category and subsystem (logical and component package). The Analyzer initializes these project assignments based on directory structure. To modify them, click **Action > Set File Properties.**

The Analyzer has assigned `move.h` and `propel.h` to a category (logical package) named `example` and a subsystem (component package) named `example`.

## Establishing a Path Map Entry

Rational Rose provides the Path Map mechanism to support parallel development by teams of analysts, architects, and engineers. The Path Map enables Rational Rose to create model files whose embedded paths are relative to a user-defined symbol. This allows Rational Rose to work with models moved or copied among workspaces and archives by redefining the actual directory associated with the user-defined symbol. Refer to the *Team Development* chapter in your *Rational Rose 2000e, Using Rose,* for more information.

The Analyzer utilizes this same mechanism to:

- Convert *physical paths* to *virtual paths* when generating model files containing an exported design.
- Convert *physical paths* to *virtual paths* when saving a project to a file.
- Convert *virtual paths* to *physical paths* when loading a project from a file.
- Convert *virtual paths* to *physical paths* in a project's File list, Base list, Directory list, and export options.

The project Directory list for `transprt` contains a single source directory, `c:\rose\example`, with which the Analyzer has associated a data directory whose path is `$DATA\rose\example`. The symbol $DATA in this path is a virtual symbol, designated by its leading **$**. The Analyzer provides a Path Map mechanism to allow each developer to uniquely map each virtual symbol to a physical path.

To define a mapping for $DATA, click **File > Edit Path Map** to display the **Virtual Path Map** dialog box:

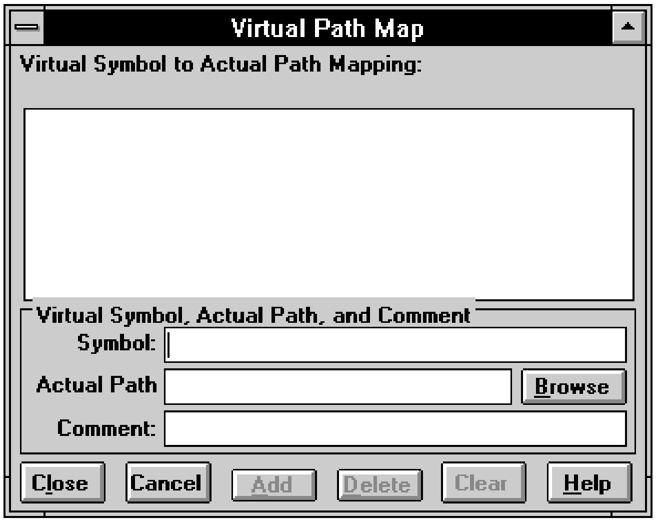

Virtual Path Map
Virtual Symbol to Actual Path Mapping:
Virtual Symbol, Actual Path, and Comment
Symbol:
Actual Path
Browse
Comment:
Close
Cancel
Add
Delete
Clear
Help

***Figure 9 Virtual Path Map Dialog Box***

This dialog box shows a Path Map with no current entries, but your Path Map might already contain entries for several virtual symbols such as $DATA and $DESIGN. Such entries were established during your installation of Rational Rose.

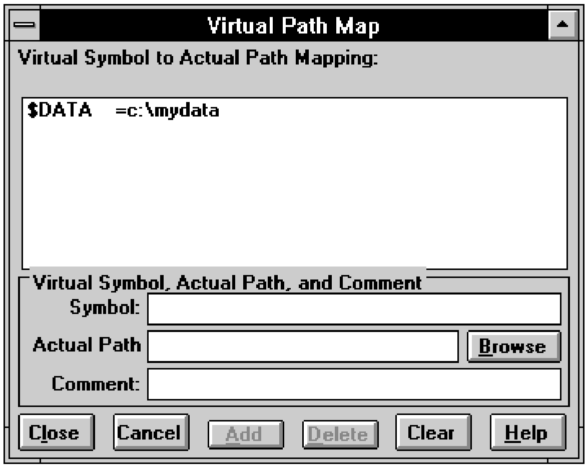
**To add an entry to the Path Map:**

1. Enter the virtual symbol in the Symbol text box; this symbol must begin with a **$** character.
2. Enter the actual path in the Actual Path text box, or click **Browse** to display a dialog box containing a file selector.
3. Optionally, enter a comment in the Comment text box.
4. Click **Add**.

To delete an entry from the Path Map, select it in the Virtual Symbol, Actual Path, or Comment text boxes and click **Delete**.

To update the Path Map, click **OK**.

If there is no entry for $DATA, create one by entering **`$DATA`** in the Symbol text box, then enter **`c:\mydata`** in the Actual Path text box, and then click **Add**:


Virtual Symbol to Actual Path Mapping:

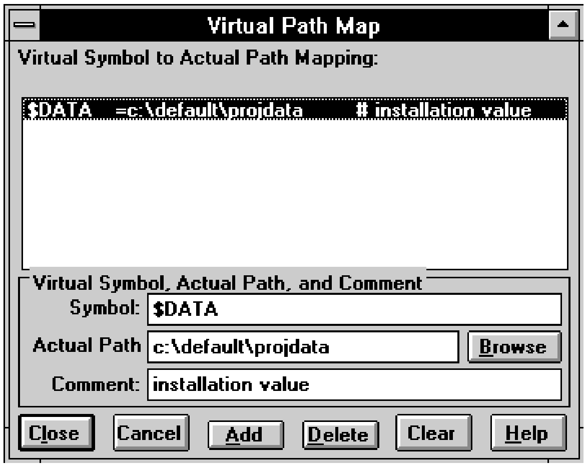
```
$DATA   =c:\mydata
```

Virtual Symbol, Actual Path, and Comment

| | |
|---|---|
| Symbol: | |
| Actual Path | |
| Comment: | |

***Figure 10 Virtual Symbols***

If a mapping for $DATA is present, click on its entry to place its parameters in the **Symbol, Actual Path**, and **Comment** text boxes where you can modify them:


Virtual Symbol to Actual Path Mapping:

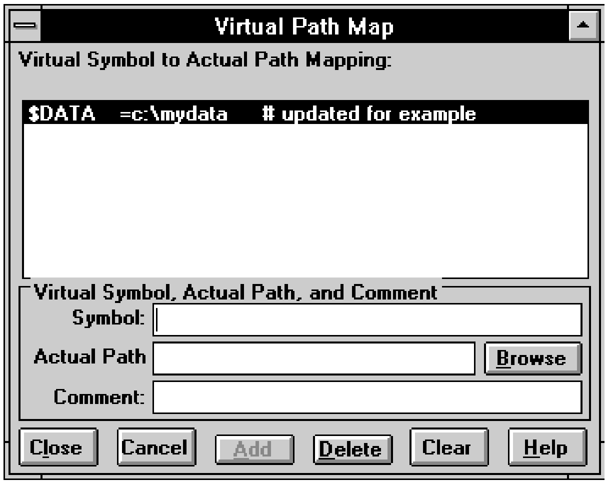
```
$DATA   =c:\default\projdata        # installation value
```

Virtual Symbol, Actual Path, and Comment

| | |
|---|---|
| Symbol: | $DATA |
| Actual Path | c:\default\projdata |
| Comment: | installation value |

***Figure 11 Virtual Symbols and Parameters***

Change the Actual Path text box to **`c:\mydata`**, update the **Comment** appropriately, and then click **Add**:


Virtual Path Map

Virtual Symbol to Actual Path Mapping:

```
$DATA   =c:\mydata     # updated for example
```

Virtual Symbol, Actual Path, and Comment

Symbol:

Actual Path

Comment:

***Figure 12 Mapping Virtual Symbols***

Click **Close** to exit the dialog box. Whether you created a new Path Map entry, or modified an existing one, you have now specified that the virtual symbol `$DATA` is mapped to `c:\mydata`. As it semantically analyzes the source files in `transprt`, the Analyzer encounters the data directory path `$DATA\rose\example`, whereupon it creates a directory named `c:\mydata\rose\example` (if one does not already exist) and places in this directory the data file generated by the analysis of each source file in `c:\rose\example`.

## $DATA and Drive Names

When $DATA is used in the initial position of the data specification, the drive letter from the source is preserved. This assures the data on different drives to be in unique directories.

For example, if `$DATA = c:\rose_data` but your source is on your `d:` drive, the data path defaults to `d:\rose_data`.

If you are keeping data files on a different drive than your source files, remove the drive letter and colon from the **$DATA** definition and begin the data specification with the desired drive letter. For example:

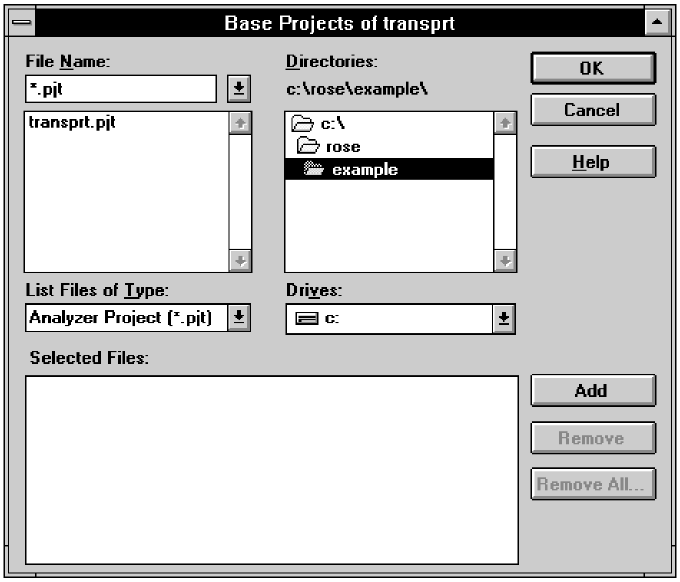
**`c:$DATA\...`**

## Resolving References to Libraries

It is often the case that the source code representing the program you wish to reverse engineer references symbols defined in external source code libraries, such as those provided with C++ compilers or domain-specific class libraries. You could add the directories containing these external source files to your project and reverse engineer them along with your source files, but this approach is inefficient when the libraries are used by several engineers.

Instead, you can create an Analyzer project for each external library. You then reference these library-specific projects in your project by adding them your project's *base list*. The Analyzer can then successfully resolve references from source files in your project to source files in the external libraries. A project that appears on another project's base list is referred to as a *base project*.

To add a project to your base list, click **Bases** in the project window to display the **Base Projects** dialog box:



File Name: *.pjt

transprt.pjt

Directories: c:\rose\example\

c:\
rose
example

List Files of Type: Analyzer Project (*.pjt)

Drives: c:

***Figure 13 Base Projects Dialog Box***

The upper half of this dialog box is a file requester that allows you to select files containing projects representing external libraries. Click **Add** to add a selected project file to the Selected Files box. **Remove** and

**Remove All** delete selected project files from the Selected Files box. When you have added a project for each external library your source files reference, click **OK** to update the base list.

During the Rational Rose C++ installation process, a default project named `analyzer.pjt` is created in your Rational Rose installation directory. A compiler-specific project, representing the source libraries associated with your site's C++ compiler, can also be installed and placed on the base list of `analyzer.pjt`. This compiler-specific project references the Path Map virtual symbols, which it resolves to the directories containing these source libraries. During installation, you identify the locations of these directories. Path Map entries defining the appropriate virtual symbols are created using this information.

Click **File > New** without depressing the **SHIFT** key, and the Analyzer creates the new project by copying `analyzer.pjt`. The new project contains a base list entry referring to the project representing the source libraries associated with your site's C++ compiler. This is necessary when the source code you intend to analyze references your compiler's source libraries.

You can use this mechanism to facilitate the use of one or more class libraries used at your site: create a project for each class library, and then add these projects to the base list of `analyzer.pjt`.

While depressing the **SHIFT** key, click **File > New**, to empty the new project.

## Analyzing Source Code

The process of reverse engineering is split into two steps: analysis (syntactic and semantic), and export. This division is for efficiency—it allows you to analyze a program once but create several different Rational Rose models from it, varying the diagrams produced, or focusing on different parts of the program as you learn more about it.

To analyze source files on the file list, select the files you want to analyze, then click **Action > Analyze**. The Analyzer displays its progress as it proceeds in the status bar, reporting the number of classes, members, and errors encountered. The analysis status of any file containing errors is set to Has Errors in the file list. Double-clicking on such a file list entry expands the entry to display one line describing each error found. Double-clicking on one of these error lines opens a

**File Viewer** window on the source file, positioned on the line containing the error. When errors are detected during Analysis, you can review the Analyzer’s log by clicking **Window > Log**.

To select multiple files on the file list in the project window, drag the mouse from one entry to another. Clicking on an entry while depressing the CONTROL key switches that entry’s selection status without deselecting any other entries. You can select every entry on the file list by clicking **Edit > Select All**.

Select **move.h** and **propel.h**, and then click **Action > Analyze**. After analysis, the status of `move.h` should be set to Analyzed, and the status of `propel.h` should be set to CodeCycled.

## Dealing with Common Errors

There are three common sources of errors encountered during analysis:

- Unresolved references
- Language extensions
- Context-sensitive source files

***Note:*** *It is not necessary to achieve an error-free analysis before proceeding to export design information to Rational Rose. Especially when first reverse engineering large bodies of pre-existing code, it is often more effective to proceed once most of the “important” source code analyzes without error, deferring elimination of errors in less-important code to later iterations.*

### Unresolved References

The Analyzer can report errors in source code that compiles correctly. Error messages containing the phrase "cannot #include" usually indicate an incomplete base list, which prevents the analyzer from resolving references to declarations in your compiler-specific or library-specific source files. Eliminate such errors by determining which directories contain the missing header files, and adding these directories to a project listed in your project’s base list. To force re-analysis, while depressing the SHIFT key, click **Action > Analyze**.

## Language Extensions

Another cause of analysis errors in source code that compiles cleanly is that your code can contain C++ language extensions supported by your compiler but not recognized by the analyzer. Alternatively, your source code might use improper or obsolete C++ constructs that are accepted by the compiler but reported as errors by the Analyzer. In many cases, you can work around such problems by adding symbols to the Defined Symbol list, thereby enabling the Analyzer to successfully preprocess and parse “non-standard” C++ source code.

## Context Dependent Source Files

Error messages that contain the phrase "cannot find" are usually caused by a source file whose semantic analysis is context dependent. The Analyzer provides the analysis type designation to permit these source code files to be handled properly. Such errors are frequently caused by source files which reference symbols defined in another file without explicitly including that file. You correct the problem by adding the appropriate `#include` directive, thereby making the source file context independent. When this is not possible, click **Action > Set Type** to designate the offending source file as Type 2.

Click **Edit > Type 2 Contexts** to modify the project’s Type 2 context by adding directives that include the required source file. After closing the Type 2 context, select the offending source file and then click **Action > Analyze**.

Suppose, for example, that analysis of source file `balance.h` reports the error "cannot find units," and you determine that units is defined in source file `currency.h`. You observe that `balance.h` does not contain the directive:

```
#include "currency.h"
```

**To add the directive:**

1. Select `balance.h`, and mark it as analysis Type 2.
2. Edit the project’s Type2 Context, and append the following directives:

   ```
   #if defined (t2_balance_h)
   #include "currency.h"
   #endif
   ```

3. Select `balance.h`, and execute the Analyze command. The Analyzer processes the directives in the Type 2 Context before analyzing `balance.h`, and includes `currency.h`, defining `units`.

## Deciding What to Export

Exporting a design to Rational Rose means creating a model file that specifies model components and, optionally, diagrams. Before exporting, you must specify:

- How the C++ constructs in the source files should be mapped to the UML, Booch, or OMT notation.
- What diagrams should be created.

To provide flexibility, the Analyzer specifies this information for each kind of C++ construct and relationship:

- Classes
- Utilities
- Modules
- Typedefs
- Fundamental types
- Data members
- Member functions
- Containment relationships
- Visibility relationships
- Inheritance relationships
- Friend relationships

The Analyzer provides export options that enable you to control the handling of each of these constructs. The Classes export option specifies whether each C++ class should be:

- Represented in the exported model, and whether this representation should be conditioned on the class' visibility.
- Drawn on a class diagram created in the exported model, and whether this appearance should be conditioned on the class' visibility.
- Documented in its specification by comments extracted either before or after the class header.

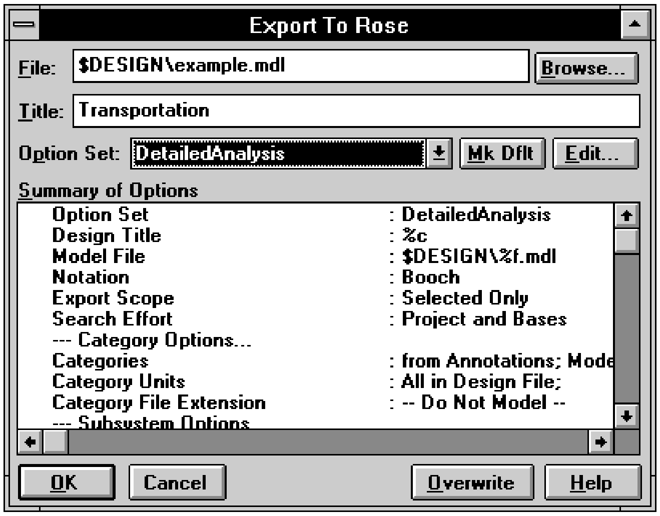
To facilitate the management of export options, the Analyzer provides export option sets, which are named collections of export option settings that can be saved as part of a project. This creates an appropriate export option set that can be easily reused, or shared among team members.

The Analyzer provides three predefined Option Sets to support common reverse engineering usages:

***Table 11 Option Sets***

| | |
|---|---|
| FirstLook | For high-level examination of preexisting source code |
| DetailedAnalysis | For in-depth examination of preexisting source code |
| RoundTrip | For Round-Trip Engineering |

To create a design from `move.h` and `propel.h` using **Action > Detailed Analysis**, click **Export To Rose** to display the **Export To Rose** dialog box:



File: $DESIGN\example.mdl

Title: Transportation

Option Set: DetailedAnalysis

Summary of Options

```
Option Set                 : DetailedAnalysis
Design Title               : %c
Model File                 : $DESIGN\%f.mdl
Notation                   : Booch
Export Scope               : Selected Only
Search Effort              : Project and Bases
--- Category Options...
Categories                 : from Annotations; Mode
Category Units             : All in Design File;
Category File Extension    : -- Do Not Model --
--- Subsystem Options
```

***Figure 14 Export To Rose Dialog Box***

Use the Option Set box to select the **Detailed Analysis** export Option Set, which instructs the Analyzer to create both class and module diagrams for the program being exported.

The **Export To Rose** dialog box provides two text boxes— File and Title—that reflect current export option settings.

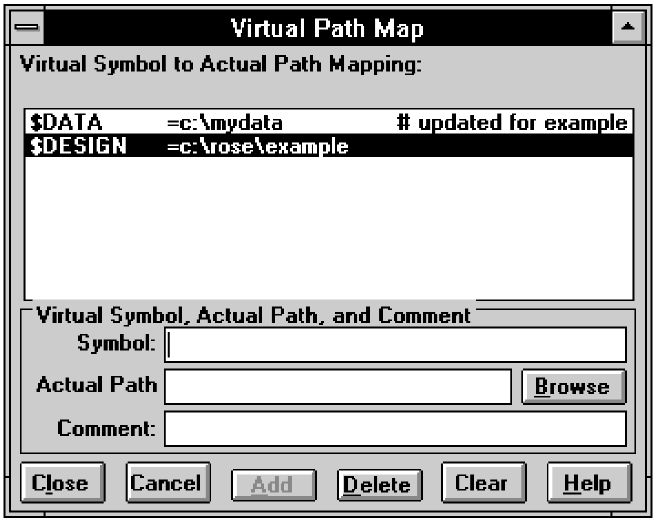
The File text box, which specifies the path to the generated model file, is similarly initialized from the **Model File** export option. This value is shown on the third line of the Summary of Options box. This value includes the format code `%f`, which expands to a simple name computed from the selected file list entries and the virtual symbol $DESIGN.

The Title text box is initialized from the **Design Title** export option, whose value is shown on the second line of the Summary of Options box. The `%c` format code expands to the project caption. To modify the title of diagram's **Design Title** export option, click **Action > Export To Rose**.

You can cause $DESIGN to resolve to `c:\rose\example` by adding:

```
$DESIGN = c:\rose\example
```

to the Path Map, then clicking **Cancel** in the **Export To Rose** dialog box, and then clicking **File > Edit Path Map**.

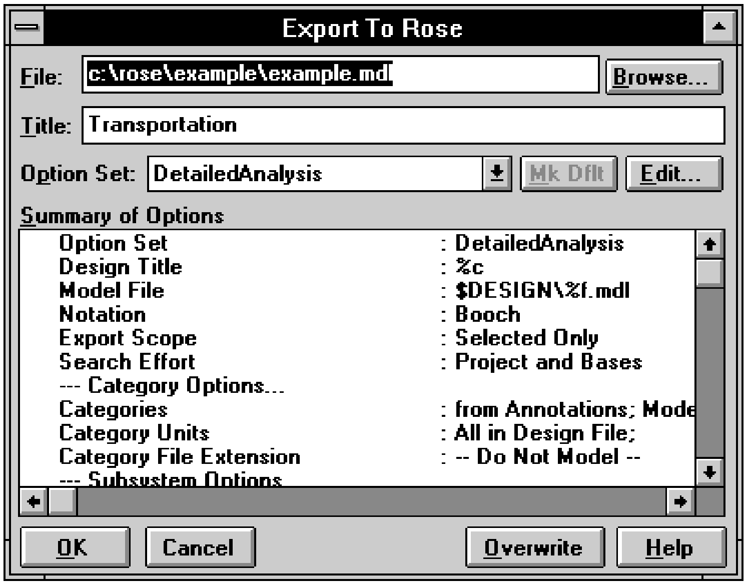

Virtual Path Map
Virtual Symbol to Actual Path Mapping:
Virtual Symbol, Actual Path, and Comment
Symbol:
Actual Path
Browse
Comment:
Close
Cancel
Add
Delete
Clear
Help

```
$DATA      =c:\mydata           # updated for example
$DESIGN    =c:\rose\example
```

*Figure 15 Mapping the $DESIGN Symbol*

This allows many users to utilize the same export Option Set, since the mapping from $DESIGN to directory is driven by the Path Map, which is unique to each user. Alternatively, you could directly edit the **Export To Rose** dialog box's File text box to specify a path to be used only during this export operation.

After $DESIGN is defined, click **Actions > Export To Rose**. The File text box in the **Export To Rose** dialog box now displays the actual path to the generated model file after the virtual symbol is replaced as directed by its Path Map entry:



File: c:\rose\example\example.mdl

Title: Transportation

Option Set: DetailedAnalysis

Summary of Options

```
Option Set                          : DetailedAnalysis
Design Title                        : %c
Model File                          : $DESIGN\%f.mdl
Notation                            : Booch
Export Scope                        : Selected Only
Search Effort                       : Project and Bases
--- Category Options...
Categories                          : from Annotations; Mode
Category Units                      : All in Design File;
Category File Extension             : -- Do Not Model --
--- Subsystem Options
```

***Figure 16 Actual Path Displayed***

Click **OK** to initiate generation of the model file. The Analyzer displays its progress in the status bar, reporting the number of classes, members, and errors encountered. If errors are produced, click **Window > Log** to inspect the error messages.

Load `c:\rose\example\example.mdl` into Rational Rose by clicking **File > Open**.

If your default notation is UML, the logical package contains the class diagram named “Reverse Engineered” as shown:

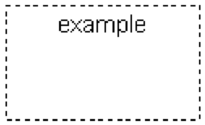


***Figure 17    Reverse Engineered Class Diagram***

If you clicked **Options > Double Click To Diagram**, double-clicking on this logical package icon displays the following class diagram:

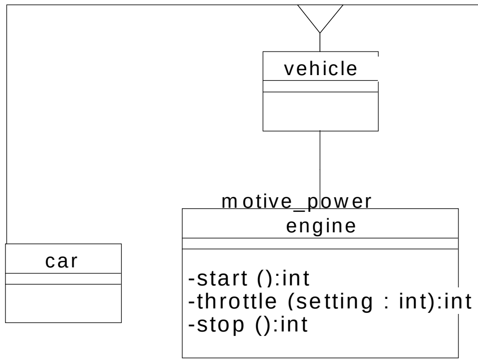


***Figure 18    Example Class Diagram***

Click **Tools > Layout** to produce the following class diagram:

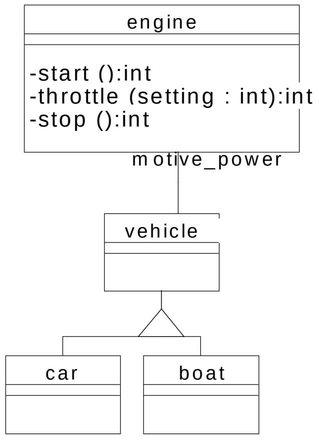
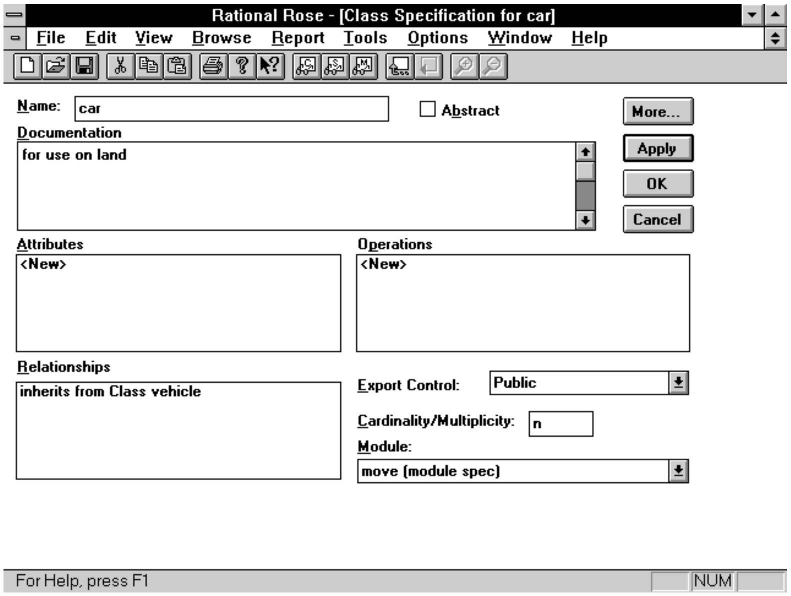


***Figure 19    Class Diagram After Running Layout***

Select the icon representing class **car**, and click **Browse > Specification**:


Rational Rose - [Class Specification for car]

File Edit View Browse Report Tools Options Window Help

Name: car
☐ Abstract

Documentation
for use on land

Attributes
<New>

Operations
<New>

Relationships
inherits from Class vehicle

Export Control: Public

Cardinality/Multiplicity: n

Module: move (module spec)

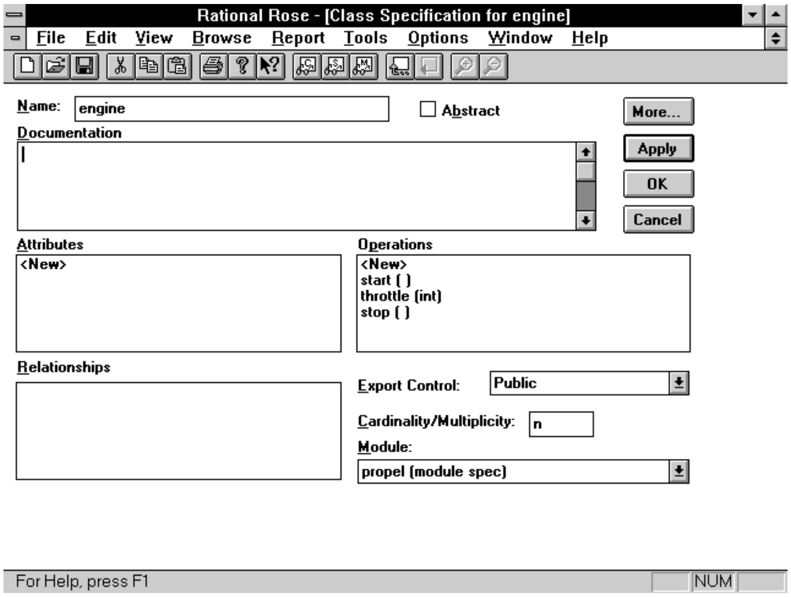
More... Apply OK Cancel

For Help, press F1 NUM

***Figure 20 Class Specification for Car***

***Note:*** *The comment preceding the declaration of this class in* `move.h` *has been captured in the specification's Documentation field. Also note that the class* `car` *is assigned to module specification* `move`*.*

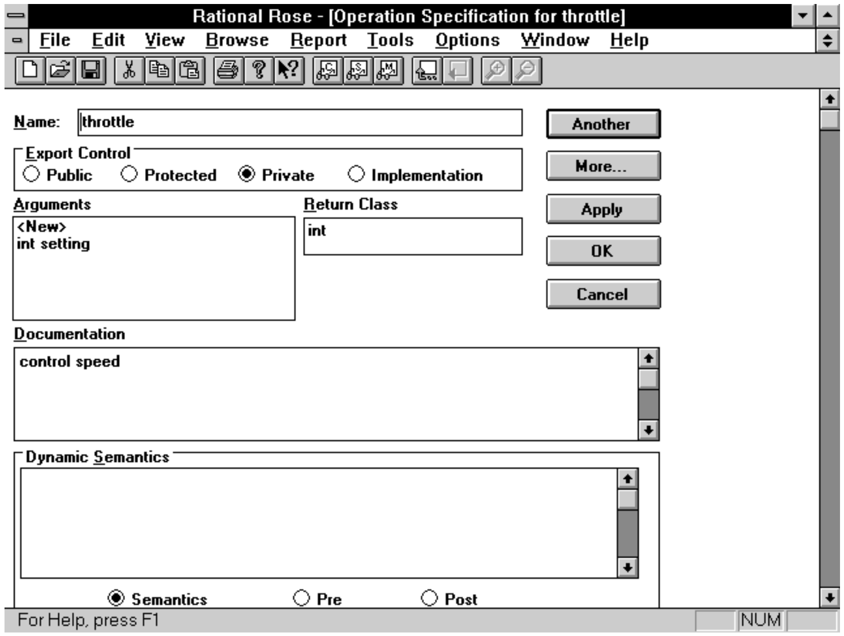
Select the **engine** class icon, and then click **Browse > Specification:**


Rational Rose - [Class Specification for engine]
File Edit View Browse Report Tools Options Window Help
More...
Apply
OK
Cancel
For Help, press F1
NUM

Name: engine
☐ Abstract
Documentation
Attributes
<New>
Operations
<New>
start ( )
throttle (int)
stop ( )
Relationships
Export Control: Public
Cardinality/Multiplicity: n
Module: propel (module spec)

***Figure 21 Class Specification for Engine***

***Note:*** *Each of* ***`engine`****'s operations has been captured, and the class* ***`engine`*** *is assigned to module specification* ***`propel`****. Double-click on operation* ***`throttle`*** *to display its specification:*



Rational Rose - [Operation Specification for throttle]

Name: throttle

Export Control: ○ Public ○ Protected ● Private ○ Implementation

Arguments:
<New>
int setting

Return Class: int

Documentation: control speed

Dynamic Semantics: ● Semantics ○ Pre ○ Post

For Help, press F1 NUM

***Figure 22 Operation Specification for Throttle***

***Note:*** *The comment preceding the declaration of* ***`throttle`*** *in* ***`propel.h`*** *has been captured in the specification's Documentation field.*

To inspect the static architecture, click **Browse > Module Diagrams** to display the module diagram named "Reverse Engineered" in the <top level> component view:

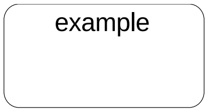


***Figure 23 Reverse Engineered Icon***

Click **Browse > Module Diagrams** to display the module diagram named "Reverse Engineered" in the `example` subsystem (component package). Then click **Tools > Layout**, to display the following static view of the example program:

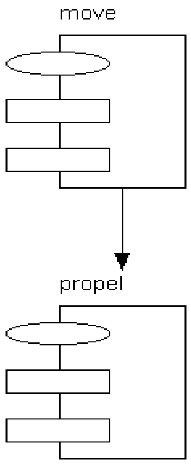


***Figure 24 Example Module Diagram***

## Summary

Reverse engineering in the sample program required the following steps:

1. Create a project that identified the program's source files.
2. Establish Path Map entries for $DATA and $DESIGN.
3. Semantically analyze the source files.
4. Choose an export option set and export a model file.
5. Load the exported model file into Rational Rose.

These steps represent the typical process used to reverse engineer an existing application and to better understand its design. When using the Analyzer in support of an iterative software development process, all five steps listed need only be performed during the first iteration. Subsequent iterations typically involve only steps 3, 4, and 5, with the condition that new source files must be added to the project in whatever iteration they first appear.

## Analysis

Analysis is the syntactic and semantic processing of a program which extracts design information. To analyze a source file, select it from a project's File list and then click **Action > Analyze**. When you analyze a file, the Analyzer preprocesses it, parses it, and then writes the results in a data file. The Analyzer also updates the file's analysis status in the File list and posts progress and error messages to the Log window, which you can inspect by clicking **Window > Log**. The Analyzer also attaches a file's error messages to its entry in the File list.

In addition to the selected files, the Analyzer recursively analyzes any files referenced in `#include` directives contained in the selected source files. The project's Directory list and the Directory lists of projects on its Base list are searched for included files when attempting to resolve the targets of these `#include` directives.

Because the Analyzer seeks only to extract design information, the source files being analyzed are not required to have been compiled without errors. In particular, function bodies are not parsed.

### Re-analysis

When you make changes to source files that you have already analyzed, you must re-analyze them to update the design information in their data files. The Analyzer automatically accelerates re-analysis by analyzing only those files that are affected by the changes.

To determine whether to re-analyze a source file, the Analyzer looks at the time stamp and reference information stored in its associated data file, as well as the actual changes made in the source file itself and in any `#include` files it references. If the changes semantically impact the source file, the Analyzer rebuilds the data file; otherwise, it reuses the data file.

If the source file's analysis status is Has Errors the Analyzer re-analyzes it only if it might have changed since the last time it was analyzed; that is, if the file or the `#include` files it references now have later time stamps. If these time stamps are unchanged, the Analyzer assumes that the errors still exist and does not update the data file. If the file's analysis type is Type 2, however, the analyzer re-analyzes it if you have modified the project's Type 2 context, or if a source file included as a result of directives in the Type 2 context has been modified.

You should force re-analysis if you correct the errors in a source file without advancing its time stamp—for example, by overwriting the source file with a version with an earlier time stamp obtained from your configuration management system.

To force the re-analysis of a specific source file, select the file, and then depress the **SHIFT** key while clicking **Action > Analyze**.

## Analysis Status

The analysis status designation indicates the Analyzer's understanding of the source file's last semantic analysis. Typical values include:

***Table 12 Analysis Status***

| | |
|---|---|
| Unknown | The source file has not been analyzed since being added to the File list, or has not been updated since the project was opened. |
| Analyzed | The last semantic analysis found no errors, but the source file cannot support round-trip engineering. |
| CodeCycled | The last semantic analysis found no errors, and the source file can support round-trip engineering. |
| Has Errors | The last semantic analysis found errors; these can be displayed by double-clicking on the File list entry. |
| Excluded | Has no data file because the source file is Type 3—the Analyzer stores design information Type 3 files in the data files of the source files which reference them. |
| No Source | Cannot find a file in the file system for this entry in the File list—if you have deleted the file since you added it to the File list, click **Edit > Cut** to remove the entry. |
| Stale Data | Has a potentially out-of-date data file for the file—this status is set by clicking **Actions > Update Status**. |
| Unanalyzed | Has no data file—this status can be set by clicking **Action > Update Status** or **> Delete Data**. |

A source file's analysis status and error information is stored in its associated data file. After opening an existing project, the analyzer does not automatically display this information in the project's File list, since changes to source files might have invalidated the information stored in the data files. Select the project's source files and click **Action > Update Status** to verify the validity of each data file and update the File list with error and analysis status information. If displaying error information is your only concern, click **Action > Show Errors** for a quicker response.

## Analysis Errors

The Analyzer reports:

- Missing or incorrect definitions for classes and types.
- Missing `#include` files.
- Errors in referenced `#include` files.

The Analyzer reports all errors in the Log window, which you can inspect by clicking **Window > Log**. In addition:

- If a file on the File list contains errors, the Analyzer attaches the error messages for that file to its File list entry. Double-click on the entry to expand or collapse it, thereby displaying or hiding associated error messages. Double-click on an error message to display a **File Viewer** window on the source file positioned at the point of error.
- If a referenced `#include` file found through the Directory list contains errors, the Analyzer adds a File list entry for that file and attaches its errors to the new entry. The Analyzer marks such entries as `untyped` and leaves them on the File list until you re-analyze the files. You can make the entry permanent by setting an analysis type for it.

In each error message, the Analyzer displays the error location and the applicable message text in the following format:

```
line:column message text
```

The message text briefly describes the error and references any conflicts that the Analyzer encountered during analysis. This information helps you determine whether the file has missing or incorrect information. For example, the following error message reports a conflict between the function in line 55:9 and the code in line 30:

```
line 55:9 Function 'item' has a different return type than item
   in line 30
```

To eliminate errors, consider the following:

- Error messages containing the phrase "cannot #include" usually indicate an incomplete Base list, which prevents the analyzer from resolving references to declarations in your compiler-specific or library-specific header files. Eliminate such errors by determining which directories contain the missing header files, and adding these directories to a Base project in your project's Base list.
- Your code might contain C++ language extensions supported by your compiler, but not recognized by the analyzer. Alternatively, your source code might use improper or obsolete C++ constructs which are accepted by that compiler, but reported as errors by the Analyzer. Error messages of the form "expected '...', saw '...'" typify these situations. In many cases, you can add symbols to the Defined Symbol list, thereby enabling the Analyzer to successfully preprocess and parse "non-standard" C++ source code.
- Error messages that contain the phrase "cannot find" are usually caused by a source file whose semantic analysis is context dependent. Click **Action > Set Type** to designate such files as Type 2, and then click **Edit > Type2Contexts** to provide the context needed to analyze the source file.
- Are include directories being searched? If not, add them to the project's Directory list and specify that they are to participate in the include search.
- Are include directories being searched in the proper order? If not, change their order in the Directory list.

***Note:*** *It is not necessary to achieve an error-free analysis before proceeding to export design information to Rational Rose. Especially when first reverse engineering large bodies of pre-existing code, it is often more effective to proceed after most of the "important" source code analyzes without error, deferring elimination of errors in less-important code to later iterations.*

## Analysis Types

Some files require special handling during analysis because they are context dependent or syntactically incomplete. To ensure correct analysis, every file is tagged with an analysis type. A source file's analysis type characterizes its context dependence and syntactic completeness:

***Table 13 Analysis Type***

| **Analysis Type** | **Syntactically Complete** | **Context-Independent** |
|---|---|---|
| 1 | Yes | Yes |
| 2 | Yes | No |
| 3 | No | Unknown |

Files that do not require special handling should be designated as Type 1 files; when you initially select a file for analysis, the Analyzer assumes it is Type 1.

### Type 1 Source Files

A Type 1 source file is:

- Syntactically complete—the file is a list of complete C++ declarations at file scope.
- Semantically context-independent—the file either contains its own symbol definitions or obtains definitions from files targeted by its `#include` directives: definitions are visible for all referenced symbols.

The header files in a well-engineered C++ program should meet these Type 1 criteria.

A Type 1 source file is analyzed either when it is found on the File list or when it is referenced by another source file being analyzed, whichever comes first.

Because a context independent Type 1 file has a consistent meaning throughout the program, the Analyzer stores the results of its analysis in an associated data file which it references when processing `#include` directives that name the source file.

## Type 2 Source Files

A Type 2 source file is:

- Syntactically complete: the file is a list of complete C++ declarations at file scope.
- Semantically context dependent: the file contains symbols whose definitions are provided by the context into which it is included.
- Interpreted the same way no matter where it is included: the symbols in the file produce the same definition from every `#include` context in the program.

Type 2 source files are usually designed to be included along with some companion `#include` file that provides the required symbol definitions.

A Type 2 file is context dependent. It must be analyzed in an appropriate context; otherwise, the Analyzer might not successfully resolve its symbols. Consequently, the Analyzer analyzes Type 2 files in the context defined by the project's Ty pe2 Context component. Because a context dependent Type 2 file has a consistent meaning throughout the program, the Analyzer stores the analysis results in a data file and reuses the stored data each time a `#include` directive references the file.

Each project provides a Type 2 Context component that enables you to define a standard set of definitions or inclusions to be logically prepended to every Type 2 file contained in that project. Modifying the Type 2 Context causes all Type 2 files to require re-analysis. Modifying a source file included in the Type 2 Context causes all Type 2 files that reference the modified file to require re-analysis.

Error messages that contain the phrase "cannot find" are often caused by source files that should be designated as Type 2. Click **Action > Set Type** to designate the offending source file as Type 2. Click **Edit > Type2Contexts** to modify the project's Type 2 Context by adding directives that include the required source file. After closing the Type 2 Context, click **File > Save** to make your changes permanent.

Suppose, for example, that analysis of source file `balance.h` reports the error "cannot find units", and you determine that `units` is defined in source file `currency.h`. You observe that `balance.h` does not contain the directive `#include "currency.h"`.

Select `balance.h`, and mark it as analysis Type 2. Edit the project's Type 2 Context, and append the following directives:

```
#if defined (T2_BALANCE_H)
#include "currency.h"
#endif
```

Select `balance.h`, and execute the Analyze command. The Analyzer processes the directives in the Type 2 Context before analyzing `balance.h`, and will therefore include `currency.h`, defining `units`.

## Type 3 Source Files

Any source file that is in isolation syntactically incomplete—containing a bare list of statements, or some portion of a declaration—should be set to analysis Type3. Files of this sort are usually designed to introduce additional text into the file that includes them, producing a result that is syntactically complete.

Any file that meets the following criteria should also be set to analysis Type 3:

- The file is syntactically complete, consisting of a list of complete declarations at file scope.
- The file is semantically context dependent, containing symbols whose definitions are provided by the context in which the file is included.
- The symbols contained in the file are interpreted differently as a function of where in the program the file is included.

Source files meeting these criteria are sometimes encountered in simulations of templates in early versions of C++.

Because Type 3 source files are either syntactically or semantically context dependent, the Analyzer analyzes them in context. Without information from the files that `#include` them, Type 3 files might fail to successfully parse or might contain symbols which cannot successfully be resolved. Consequently, the Analyzer ignores any selected Type 3 files in the File list—it analyzes them only when resolving `#include` directives.

Because the meaning of a Type 3 source file varies throughout a program, the file must be re-analyzed each time it is referenced by another file. The Analyzer stores the results of each analysis in the data file associated with the source file that included it, rather than the Type 3 file's own data file.

## Preprocessing

Preprocessing is the first phase of analyzing a C++ source file. During preprocessing, the Analyzer:

- Expands preprocessing macros.
- Executes preprocessing directives such as `#include` directives and conditional compilation.

During macro expansion, the Analyzer consults the entries in the project's Defined Symbols list and the Undefined Symbols list.

When the Analyzer encounters `#include` directives in a file, it searches the directories in the project's Directory list and, if necessary, the Directory lists of the base projects in its Base list for the referenced `#include` files. Files which are included are recursively analyzed.

The Analyzer's handling of an `#include` file referenced by multiple source files depends on that `#include` file's analysis type. Unless the included file is Type 3, the Analyzer preprocesses and parses it only once, using information stored in the included file's associated data file for subsequent encounters. If the referenced `#include` file is Type 3, then the Analyzer re-analyzes it each time it is referenced.

## Parsing

Parsing is the second phase of analyzing a file, after preprocessing. Parsing performed by the Analyzer is similar to parsing during compilation, except that the Analyzer ignores the code in function bodies. Thus, you can reverse engineer files that do not currently compile without error, or are semantically incomplete.

The Analyzer extracts design information from the following constructs at any scope outside of function bodies:

- Struct definitions
- Class definitions
- Enumeration definitions
- Type definitions
- Instantiations
- Class template definitions
- References to types and classes in forward declarations, derivation lists, friend declarations, and function signatures

### Order of Analysis

The Analyzer determines the order in which to analyze files by:

- The order of the files on the File list.
- The order in which `#include` directives are resolved during preprocessing.

The Analyzer uses the File list order as a starting point. When you select multiple files on the File list, the Analyzer starts by preprocessing the first file in the selection. The Analyzer does not parse this file until it has resolved all of its `#include` directives. When the file has been parsed, the Analyzer moves to the next un-analyzed file in the selection. Consequently, the Analyzer might analyze an arbitrary number of files before starting preprocessing on the second file in the selection.

Some of the files found during `#include` resolution might also be among those selected on the File list. By default, the Analyzer analyzes a file whenever it is first encountered—as the next selection on the File list or during `#include` resolution for another file.

Syntactically incomplete files must be analyzed only when they are found during `#include` resolution. Errors result if you allow them to be analyzed in the wrong order. You can prevent this by clicking **Action > Set Type** to designate such files as Type 3.

## Design Exporting

After you analyze the source files specified in a project, you can selectively extract design information and generate one or more model files; these model files can then be displayed or manipulated with Rational Rose. Rational Rose can merge the generated model files into an existing model, updating it to reflect design changes instituted in source code derived from that existing model—refer to the chapter titled *C++ Round-Trip Engineering* for more information about this approach.

The export operation can be directed to map specific C++ constructs found in your source code into components of the notation, and selectively place these components into the generated model file. The export operation can optionally fabricate a class diagram for each exported category (logical package), and can optionally fabricate a module diagram for each exported subsystem (component package).

These diagrams can be selectively populated with icons representing exported model components and the relationships between them. Diagrams fabricated by the Analyzer are placed in the generated model file, enabling viewing and manipulation with Rational Rose.

There are many choices which drive exporting—the selection of source files, the specification of language-to-notation mapping, the selection of components to be exported, the grouping of classes into categories (logical packages), the grouping of modules into subsystems (component packages), and the fabrication, naming, and population of diagrams. These choices are controlled by Export Options, which are maintained in the project's Export Options component. Refer to Chapter 5, *Analyzer Export Options* for more information.

The Export Options appropriate for one set of software engineering activities can be inappropriate for another. For example, the Export Options needed to reverse-engineer pre-existing C++ source code are different from those one would use during a round-trip iteration. The Analyzer therefore enables you to define Export Option sets—named collections of settings for each Export Option—that enable you to rapidly and accurately establish Export Option settings appropriate for your current activity. The Analyzer provides three predefined Export Options sets:

***Table 14 Export Options Sets***

| | |
|---|---|
| FirstLook | For high-level examination of pre-existing source code. |
| DetailedAnalysis | For in-depth examination of pre-existing source code. |
| RoundTrip | For round-trip engineering. |

## Code Cycling

In a round-trip engineering iteration, new operation definitions and file-scope declarations added to the source code must be annotated to preserve them through subsequent code generation. When reverse engineering the source code, you can instruct the Analyzer to insert these annotations by clicking **Action > Code Cycle**. This command performs the same analysis steps performed by clicking **Action >**

**Analyze**. If no errors are encountered during analysis, the Analyzer inserts the required annotations into the source code in place, creating an unmodified backup copy of each modified source file.

If the command is successful, the selected file's analysis status is set to **CodeCycled**. If a source code file is write-protected, or if the Analyzer cannot create a backup copy, it does not add the annotations, and sets the file's analysis status to **Analyzed**.

Click **Action > Analyze** to set a file's analysis status to **CodeCycled** if no errors are found and if the necessary annotations are already present; this typically occurs when you reverse engineer source code generated by the Rational Rose C++ code generator.

Click **Action > Code Cycle** to annotate operation definitions and file-scope declarations when initially reverse engineering source code that was not produced by the Rational Rose C++ code generator. During subsequent code generation, the properties for automatic generation of operations—such as constructors, destructors, and equality—should be disabled; otherwise, the automatically generated operations might duplicate those already present in the source code.

## The Analyzer User Interface

### Application Window

The main window displays the name Rational C++ Analyzer in its title bar. When you start the C++ Analyzer, the main window is the first window that you see. This window remains on the screen until you exit the application. The main window is initially empty except for the menu bar, toolbar, status bar, and an icon representing the Log window. The menu bar contains all of the C++ Analyzer's command menus.

The toolbar appears below the menu bar, and contains buttons that can activate frequently-used commands; the toolbar also contains a text box that specifies the search string used by clicking **Edit > Find** or **Edit > Find Next**. Allowing the mouse cursor to remain over a toolbar button for a few seconds displays a tool tip naming the command associated with the button. Click **View > Toolbar** to enable or disable the display of the toolbar.

The status bar appears at the bottom of every File Viewer window, and at the bottom of a Project window while a command from the **Action** menu is executing. Click **View > Status Bar** to enable or disable the display of the status bar.

The Analyzer application window can simultaneously display multiple child windows: the Log window, one or more Project windows, and one or more File Viewer windows. Click **Window > Cascade** or **Window > Tile** to arrange child windows.

## Project Window

The Analyzer displays the components of a project in a Project window; the project's file name is displayed in the title bar. When you enlarge a Project window to the maximum size, it shares its borders and title bar with the Analyzer Application window.

A Project window displays some or all of the project's components. Each displayed component appears as a rectangular region bordered by dark lines within which you can scroll the component's contents. Using sizing bars, you can resize a displayed component to display more of its contents without scrolling. You can click a component's edit button to change its contents.

The particular set of project components shown in the Project window depends on the currently selected view of the project.

For example, to make **Use On Open** the default project view:

1. Click **View > Use On Open**.
2. Click **File > Save**.

The Analyzer provides the following Project window views:

***Table 15 Project Window Views***

| Choose this view: | To display these components: |
|---|---|
| Full | All components |
| Export | Caption, File list, Directory list, Category list, Subsystem list, Export Options, Defined Symbols list |
| Simple | Caption, File list, Directory list, Extension list, Base list |
| Files Only | Caption, File list |

If you use the sizing bars to change the layout of a view, and want to preserve these changes:

1. Click **View > MemorizeChanges**.
2. Click **File > Save**.

To restore all views to their original layouts:

1. Click **View > Restore Standard Views**.
2. Click **File > Save**.

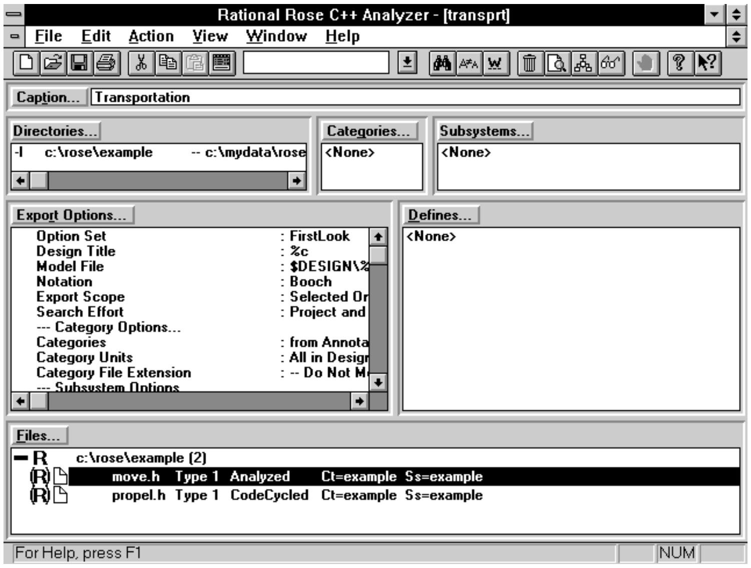
Click **Window > New Window** to create a new copy of the currently active Project window. The Analyzer automatically maintains consistency between the multiple copies of a Project window. Thus you can set up window copies to present specific project views or file list sort orders to facilitate rapid access to the desired presentation, or both.

## Full View

To display the full view of a Project window, click **View > Full.**

## Export View

To display the full view of a Project window, click **View > Export**:

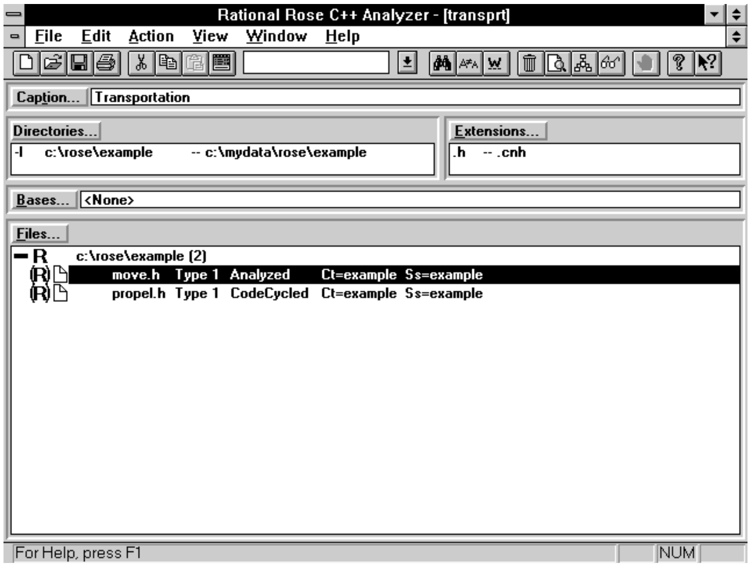


```
Caption...  Transportation

Directories...
-I    c:\rose\example        -- c:\mydata\rose

Categories...   <None>
Subsystems...   <None>

Export Options...
    Option Set                     : FirstLook
    Design Title                   : %c
    Model File                     : $DESIGN\%
    Notation                       : Booch
    Export Scope                   : Selected Or
    Search Effort                  : Project and
    --- Category Options...
    Categories                     : from Annota
    Category Units                 : All in Desigr
    Category File Extension        : -- Do Not M
    --- Subsystem Options

Defines...   <None>

Files...
- R     c:\rose\example (2)
  (R)       move.h    Type 1  Analyzed     Ct=example  Ss=example
  (R)       propel.h  Type 1  CodeCycled   Ct=example  Ss=example

For Help, press F1                                    NUM
```

*Figure 25 Full View of Project Window, Export View*

## Simple View

To display the full view of a Project window, click **View > Simple**:

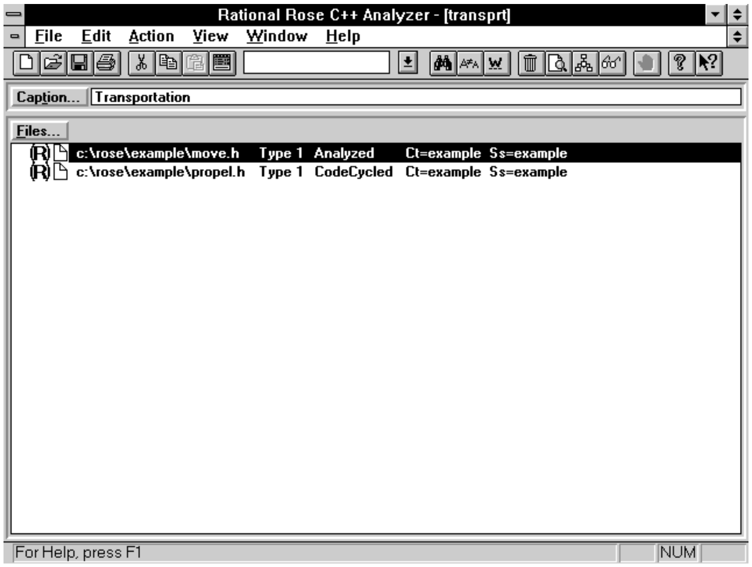


```
Caption...     Transportation
Directories...
-I    c:\rose\example        -- c:\mydata\rose\example
Extensions...
.h    -- .cnh
Bases...       <None>
Files...
R     c:\rose\example (2)
        move.h   Type 1  Analyzed     Ct=example  Ss=example
        propel.h Type 1  CodeCycled   Ct=example  Ss=example
```

*Figure 26 Full View of Project Window, Simple View*

## Files Only View

To display the full view of a Project window, click **View > Files Only**:


Rational Rose C++ Analyzer - [transprt]
File Edit Action View Window Help
Caption...
Files...
For Help, press F1
NUM

Transportation

```
c:\rose\example\move.h    Type 1  Analyzed     Ct=example  Ss=example
c:\rose\example\propel.h  Type 1  CodeCycled   Ct=example  Ss=example
```

***Figure 27 Full View of Project Window, Files Only View***

## File Viewer Window

The File Viewer window is a read-only window in which you can display the contents of a text file. You can have more than one File Viewer window open at a time. You can divide a File Viewer window into two or four panes either by clicking **Window > Split**, or by dragging the split controls located to the left of the horizontal scroll bar and above the vertical scroll bar.

Click **Window > New Window** to create a new copy of the currently active File Viewer window. The Analyzer automatically maintains consistency between the multiple copies of a File Viewer window.

Each File Viewer window contains the name of the displayed file in the title bar. The status bar displays:

- The text of a corresponding analysis error message, if applicable.
- The current line number.

The rest of the window displays the contents of the selected file. You can use the window's scroll bars to display any text that might be hidden from view. When you enlarge a File Viewer window to the maximum size, it shares its borders and title bar with the C++ Analyzer main window.

You can access the File Viewer window by selecting a file in the File list and clicking **File > Open Selected**, or by double-clicking on an analysis error displayed in the File list.

When you access the File Viewer window by double-clicking on an analysis error message, the C++ Analyzer highlights the text that corresponds to the selected analysis message and displays the error message in the status bar. If a text line has more than one associated error, the C++ Analyzer displays the text for the first error message. You can see the text for the remaining error messages by looking in the File list.

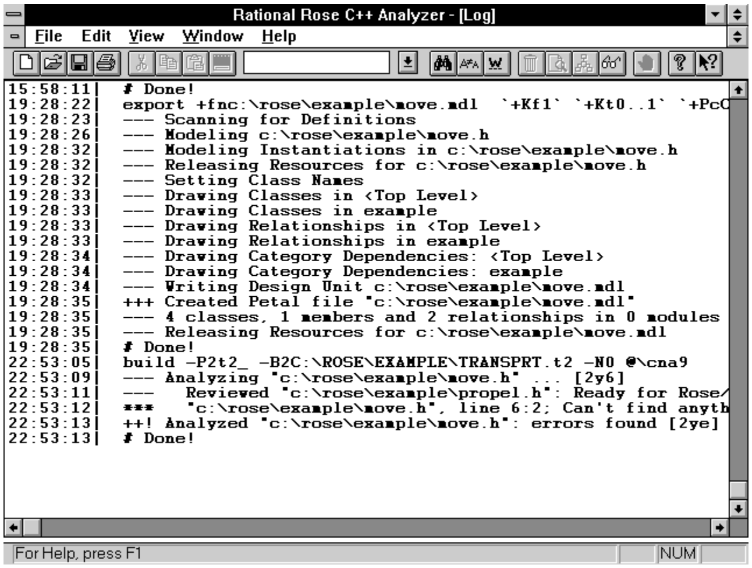
Click **View > Next Message** to navigate between error messages in the File Viewer window.

## Log Window

The C++ Analyzer posts all progress and error messages produced by **Action** menu commands in the Log window. The Log window is minimized until you double-click on its icon in the Main window, or until you click **Window > Log**. The C++ Analyzer posts messages to the Log window whether it is minimized, obscured, or visible. You cannot close the Log window—clicking on its upper-left-corner close control results in it being minimized.

Messages posted to the log can be prefixed with a time stamp. Click **View > Time Stamp** to enable or disable this time stamp. The **Time Stamp** command is only present when the Log window is active.

Use the Log window to inspect messages produced by clicking **Action > Analyze** or **Action > Code Cycle**. You can monitor progress during execution of these commands, or to help you diagnose and correct any errors afterwards:



```
15:58:11|   # Done!
19:28:22|   export +fnc:\rose\example\move.mdl  `+Kf1` `+Kt0..1` `+PcC
19:28:23|   --- Scanning for Definitions
19:28:26|   --- Modeling c:\rose\example\move.h
19:28:32|   --- Modeling Instantiations in c:\rose\example\move.h
19:28:32|   --- Releasing Resources for c:\rose\example\move.h
19:28:32|   --- Setting Class Names
19:28:33|   --- Drawing Classes in <Top Level>
19:28:33|   --- Drawing Classes in example
19:28:33|   --- Drawing Relationships in <Top Level>
19:28:33|   --- Drawing Relationships in example
19:28:34|   --- Drawing Category Dependencies: <Top Level>
19:28:34|   --- Drawing Category Dependencies: example
19:28:34|   --- Writing Design Unit c:\rose\example\move.mdl
19:28:35|   +++ Created Petal file "c:\rose\example\move.mdl"
19:28:35|   --- 4 classes, 1 members and 2 relationships in 0 modules
19:28:35|   --- Releasing Resources for c:\rose\example\move.mdl
19:28:35|   # Done!
22:53:05|   build -P2t2_ -B2C:\ROSE\EXAMPLE\TRANSPRT.t2 -N0 @\cna9
22:53:09|   --- Analyzing "c:\rose\example\move.h" ... [2y6]
22:53:11|   ---   Reviewed "c:\rose\example\propel.h": Ready for Rose/
22:53:12|   ***   "c:\rose\example\move.h", line 6:2; Can't find anyth
22:53:13|   ++! Analyzed "c:\rose\example\move.h": errors found [2ye]
22:53:13|   # Done!
```

*Figure 28 Log Window*

Each log message is prepended with a 3-character code to identify its nature:

***Table 16 Character Codes for Log Messages***

| | |
|---|---|
| ::: | Analyzer revision information |
| --- | note |
| !!! | warning |
| *** | error |
| +++ | step completed with no errors |
| ++! | step completed with non-fatal errors |
| ++* | step completed with fatal errors |

Click **File > Save Log** to save the contents of the Log window to a file. You can also choose to automatically save messages to a file as they are posted by clicking **File > Auto Save Log**. These commands are only present when the Log window is active.

## Analyzer Scripts

An Analyzer Script is an ASCII text file containing a series of commands that perform certain menu actions. The recommended file extension for a script is `.scr`, although this is not required. A script file is processed by providing its path as an argument in the command line that invokes the Analyzer.

### Invocation

Assuming that your search path is set up so that the **Analyzer** shell command invokes the Rational Rose C++ Analyzer, then

```
Analyzer -Scripts- <script-pathname1> <script-pathname2> ...
```

directs the Analyzer to process each specified script in succession, terminating only when a **quit** command is executed. If no **quit** command is encountered in a script, the Analyzer can be controlled through its user interface after the last script command is executed.

When invoked this way, the Analyzer provides two additional commands on the **File** menu: **Run Script**, and **Check Script**.

Clicking **File > Run Script** enables you to interactively select an Analyzer script to be processed. Control returns to the Analyzer's user interface after the last script command is executed, unless the script contains a **quit** command.

Clicking **File > Check Script** enables you to interactively select an Analyzer script to be checked for errors (but not processed); any errors detected are noted in the Log.

Invoking the Analyzer with the shell command

```
Analyzer -Scripts-
```

enables **Run Script** and **Check Script** without first executing a script.

## Format

Each command in a script file is written on a line by itself. The line begins with a command name and any arguments to the command follow the command name on the line. Blanks and tabs are significant and are used to delimit command names and arguments. Except in Unix file names, alphabetic case is ignored in category (logical package) names, subsystem (component package) names, and option set names.

All file names encountered during script processing are considered virtual paths; the Path Map is used to construct physical paths prior to use of the file name during command execution.

Lines beginning with `//` are treated as comments—they are listed in the Log, but otherwise ignored.

For the most part, script command names are derived from the interactive (menu-invoked) commands that they emulate.

## Selection/Deselection Patterns

Several commands select or deselect File list entries based on pattern arguments. Such arguments can contain the following wildcard characters:

***Table 17 Wildcard Characters***

| | |
|---|---|
| ? | Matches a single character. |
| * | Matches zero or more contiguous characters not including the path segment delimiter ('/' or '\'). |
| # | Matches zero or more contiguous characters. |

## Command Reference

### analyze

Semantically analyze the files specified by the selected File list entries. Execution of this command with no active project or no selected File list entries yields an error.

### autosave [<file name>]

If a file name is provided, activate the Analyzer's autosave function, which automatically spools Log contents to the designated file. If no file name is specified and the autosave function has been previously activated, then the autosave function is deactivated.

### clearlog

Clear the contents of the Log window.

### close [all]

Close the active project; if all is specified, close all projects. After execution of this command, no project is active. Execution of this command with no active project yields an error.

### codecycle

Code Cycle the files associated with the selected File list entries. Execution of this command with no active project or no selected File list entries yields an error.

### collapse

Collapses the currently-selected File list entries; if no File list entries are selected, no operation is performed. Collapsed File list entries cannot be selected or deselected. Execution of this command with no active project yields an error.

### deletedata

Delete the data files associated with the selected File list entries. Execution of this command with no active project or no selected File list entries yields an error.

### deselect all

Deselect all visible (not collapsed) File list entries. Execution of this command with no active project yields an error.

### deselect category <pattern>

Deselect all visible (not collapsed) File list entries assigned to the specified category (logical package). Execution of this command with no active project yields an error.

### deselect file name <pattern>

Deselect all visible (not collapsed) File list entries associated with the specified simple file name. Execution of this command with no active project yields an error.

### deselect line <pattern>

Deselect all visible (not collapsed) lines in the File list that match the specified pattern. This command is the only way to deselect lines that do not correspond to files in the File list—such as the lines for a directory or status category in a sorted display. Execution of this command with no active project yields an error.

### deselect [pathname] <pattern>

Deselect all visible (not collapsed) File list entries associated with the specified full path. Execution of this command with no active project yields an error.

### deselect status <pattern>

Deselect all visible (not collapsed) File list entries whose analysis status matches the specified pattern. Execution of this command with no active project yields an error.

### deselect subsystem <pattern>

Deselect all visible (not collapsed) File list entries assigned to the specified subsystem (component package). Execution of this command with no active project yields an error.

### deselect unit <pattern>

Deselect all visible (not collapsed) File list entries assigned to the specified controlled unit. Execution of this command with no active project yields an error.

### expand

Expands the currently-selected File list entries; if no File list entries are selected, no operation is performed. Execution of this command with no active project yields an error.

### export <modelfile> [<option set name>]

Export a model file using design information extracted from the files associated with the selected File list entries according to the Export Options specified in the named Export Option Set. If no Export Option Set is specified, the active project's default Export Option Set is utilized. Execution of this command with no active project or no selected File list entries yields an error.

### project <file name>

Establish the project in the named file as the active project.

### quit

Terminate the Analyzer application. The Analyzer is left running—awaiting interactive direction through its user interface—at the end of a script unless the script executes this command.

### run <file name>

Run the specified script file in the current context. The context at the completion of the specified script is retained for the rest of the script containing this command.

### savelog <file name>

Save the contents of the Log window to the named file.

## select all

Select all visible (not collapsed) File list entries. This command does not clear the current selection; it adds File list entries matching the specified pattern to the set of selected entries. Use the **deselect all** command before a select command to achieve an exclusive selection. Execution of this command with no active project yields an error.

## select category <pattern>

Select all visible (not collapsed) File list entries assigned to the specified category (logical package). This command does not clear the current selection; it adds File list entries matching the specified pattern to the set of selected entries. Use the **deselect all** command before a select command to achieve an exclusive selection. Execution of this command with no active project yields an error.

## select file name <pattern>

Select all visible (not collapsed) File list entries associated with the specified simple file name. This command does not clear the current selection; it adds File list entries matching the specified pattern to the set of selected entries. Use the **deselect all** command before a select command to achieve an exclusive selection. Execution of this command with no active project yields an error.

## select line <pattern>

Select all visible (not collapsed) lines in the File list that match the specified pattern. This command is the only way to select lines that do not correspond to files in the File list—such as the lines for a directory or status category in a sorted display. This command does not clear the current selection; it adds File list entries matching the specified pattern to the set of selected entries. Use the **deselect all** command before a select command to achieve an exclusive selection. Execution of this command with no active project yields an error.

## select [pathname] <pattern>

Select all visible (not collapsed) File list entries associated with the specified full path. This command does not clear the current selection; it adds File list entries matching the specified pattern to the set of selected entries. Use the **deselect all** command before a select command to achieve an exclusive selection. Execution of this command with no active project yields an error.

### select status <pattern>

Select all visible (not collapsed) File list entries whose analysis status matches the specified pattern. This command does not clear the current selection; it adds File list entries matching the specified pattern to the set of selected entries. Use the **deselect all** command before a select command to achieve an exclusive selection. Execution of this command with no active project yields an error.

### select subsystem <pattern>

Select all visible (not collapsed) File list entries assigned to the specified subsystem (component package). This command does not clear the current selection; it adds File list entries matching the specified pattern to the set of selected entries. Use the **deselect all** command before a select command to achieve an exclusive selection. Execution of this command with no active project yields an error.

### select unit <pattern>

Select all visible (not collapsed) File list entries assigned to the specified controlled unit. This command does not clear the current selection; it adds File list entries matching the specified pattern to the set of selected entries. Use the **deselect all** command before a select command to achieve an exclusive selection. Execution of this command with no active project yields an error.

### sortby <sort-kind>

Selects the File list display format. Legal values for `<sort-kind>` are category (logical package), directory, errors, extension, path name, status, simple name, subsystem (component package), or type. Execution of this command with no active project yields an error.

### show <project component>

Writes the contents of the selected project component to the Log, with spaces and special characters elided. Legal values for `<project component>` are bases, captions, categories (logical packages), defines, directories, Export Options, extensions, files, subsystems (component packages), Type 2contexts, and undefines. Execution of this command with no active project yields an error.

### showerrors

Refresh the error messages associated with each selected File list entry. Execution of this command with no active project or no selected File list entries yields an error.

### timestamp [on | off]

Enable or disable generation of time-of-day stamps at the beginning of each Log entry.

### updatestatus

Refresh the error messages and analysis status associated with each selected File list entry. Execution of this command with no active project or no selected File list entries yields an error.

# *Chapter 4*

# *Analyzer Projects*

A *project* is a specially formatted file that is the focal point for reverse engineering a program whose source code is contained in a set of files. Conceptually, projects used by the Analyzer are similar to project files used by some compilers. Just as a compilation project contains the necessary information for producing an executable from source-code files, a reverse-engineering project contains the necessary information for extracting a Rational Rose design from those same source-code files. Since projects are stored in named files, it is easy to reverse-engineer a set of source files repeatedly—for example, during each iteration of your program's development.

A project stores its information as a set of *components*. Components are lists that the Analyzer consults when it analyzes a program or exports design information to Rational Rose. These lists provide the information required for the Analyzer to preprocess the code, parse it, write the results into data files, and export the information from design files into model files:

- The Caption provides text describing the project's title and purpose.
- The Directory list identifies each directory containing source code files, associating each source code directory with a data directory in which its data files are maintained.
- The Extension list enumerates the file name extensions that identify files containing source code, and specifies a data file extension for each.
- The File list identifies the source code files to be analyzed from among candidates contained by directories in the Directory list.

- The Base list identifies base projects required to resolve source code `#include` references to compiler-specific or library-specific header files.
- The Defined Symbols and Undefined Symbols lists specify how preprocessor symbols found in source code files are to be expanded.
- The Categories list provides a list of categories (logical packages) to which source files can be assigned.
- The Subsystem list provides a list of subsystems (component packages) to which source files can be assigned.
- The Type 2 Context provides preprocessor directives to be executed before analyzing those context-sensitive source code files designated as analysis Type 2.
- Export options govern the selection of source files from which design information is exported, the specification of Code-to-Design Correspondences, the selection of components to be exported, the grouping of classes into categories (logical packages), the grouping of modules into subsystems (component packages), and the fabrication, naming, and population of diagrams.

The project window's currently selected view determines which components are visible. Each component can be edited to add or delete entries from its list.

Projects whose file lists contain source code for an application being reverse engineered are referred to as program-specific projects. This distinguishes them from compiler-specific or library-specific projects constructed to encapsulate compiler-specific or library-specific header files. When the Analyzer is directed to analyze the source files in an initial program-specific project, it performs the following actions:

1. Recursively analyzes each base project named in the Base list.
2. Processes the Defined Symbols and Undefined Symbols list in each base project before processing the symbols listed in the initial project.
3. If references remain unresolved after searching the directories listed in the initial project, searches the directories in the Directory list of each base project.
4. Determines whether files specified in the File list of the initial project are context-sensitive, and thus require special handling.

5. Records any errors found during analysis in the log, and in the File list of the initial project.
6. Extracts design information and places it in data files in the initial project's data directories.

When the Analyzer is directed to export design information from a program-specific project, it performs the following actions:

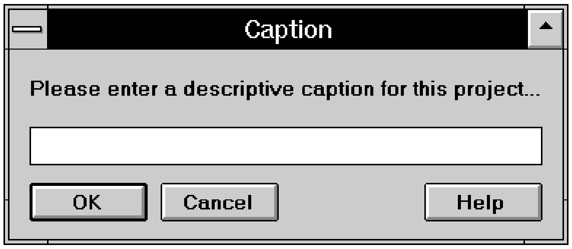
1. Uses the project's export options to identify those files from which design information is to be exported.
2. Processes each exported file's data file, using the project's export options to filter the design information, map it to the UML, COM, OMT, or Booch notation, group classes into categories (logical packages), group modules into subsystems (component packages), construct diagrams, and generate model files.

   ***Note:*** *Refer to the Analyzer Export Options chapter for more information.*

## Caption

This text string identifies a project's purpose and contents. A project's caption establishes the default value for the title of any exported diagrams, as specified by the **Title** export option. A diagram's title and name are placed in its upper-left corner. Captions also appear in entries for the base projects in a Base list.

To edit the project's caption, click **Caption** in the project window, or click **Edit > Caption**:


Caption

Please enter a descriptive caption for this project...

***Figure 29 Caption Dialog Box***

## Directory List

A project's Directory list component specifies:

- The path for each directory containing source files to be reverse engineered and, for each directory:
  - The data directory in which to create the data files that result from the analysis of source files in that directory.
  - Its code regeneration policy.
- The path for each directory to be searched when resolving `#include` directives during preprocessing. These are directories that usually appear in the include path option for many compilers.

### Data Files

When the Analyzer performs semantic analysis of a source file, it creates a data file containing the results of this analysis. When you export a design to Rational Rose, the Analyzer combines information extracted from data files to construct model files. Using incremental compilation technology, the Analyzer exploits the information in these data files to greatly reduce the time required to reverse engineer a program that was previously reverse engineered but later modified. Data files enable the Analyzer to determine what changed and assess the impact of these changes, eliminating the need to recompute valid information. This approach is particularly beneficial during the iterative development of large applications where only a portion of the source files change during each iteration.

When analyzing a source file, the Analyzer generates the name and location of the file's associated data file by:

- Giving the data file the same simple name as its associated source file.
- Finding the source file's extension in the project's Extension list and giving the data file the associated data extension.
- Finding the source file's closest containing directory in the project's Directory list and building the data file in that directory's associated data directory.

Data files enable the Analyzer to minimize preprocessing and parsing. Before the Analyzer preprocesses a file, it determines whether a valid data file already exists. Preprocessing proceeds only if no valid data file exists, if the data file is obsolete, or if the file's analysis type indicates that it should be re-analyzed at each reference. To determine whether a data file is obsolete, the Analyzer examines time stamp and reference information, as well as semantic changes made to the associated source file.

In addition to design data, time stamps, and reference information, each data file stores error status and analysis status for its associated source file. This information is displayed in the project's file list after semantic analysis.

## Data Directories

The data file associated with a source file is stored in a data directory that the Analyzer creates and associates with the directory containing the source file. When a directory is added to the project's Directory list—either as a side effect of your adding a file to the file list, or because you explicitly added the directory for `#include` resolution purposes—the Analyzer automatically creates an associated data directory. The Analyzer requires that each directory be associated with a unique data directory—no two directories can be associated with the same data directory. The Analyzer's default behavior builds the data directory structure under directory named by the virtual symbol $DATA.

After defining a Path Map entry for $DATA, you can generally leave the management of data directories and data files to the Analyzer. If necessary, however, you can change the Analyzer's behavior in creating data directories by clicking **Edit > Directories** and modifying the Default Data Directory Pattern text box in the **Directory** dialog box.

## Code Regeneration Policy

When incorporating an existing source code class library in your application, you can reverse engineer its design for inclusion in your application's design. To disable code generation for those parts of your application's design that were reverse engineered from class libraries, Rational Rose C++ provides the Generate code generation property. By setting this property to False for each class library module, subsequent code generation to that module is disabled.

To automate the setting of this code generation property for reverse engineered design elements, the C++ Analyzer allows you to enable or disable code regeneration at both directory and file granularity. By default, the directories on a project's directory list and the files on its file list are set to enable code generation, meaning that modules extracted from the files on its file list have their Generate property set to True. By sequentially clicking on a file's code regeneration icon in its file list entry, its code regeneration policy can be set to enabled, disabled, or specified by parent directory.

By clicking on a directory's code regeneration icon in its **Project Directory List** dialog box entry, its code regeneration policy is switched between enabled and disabled. If a file's code regeneration policy is disabled, or if its code regeneration policy is specified by the parent directory and that parent directory's code regeneration policy is disabled, then modules extracted from this file have their Generate property set to False; otherwise, these modules have their Generate property set to True.

## Resolving #Include Directives

When the Analyzer is attempting to resolve a `#include` directive during preprocessing, it searches each directory in the Directory list until it finds a file with the name referenced in the directive. The Analyzer searches directories in the order they appear in the Directory list.

If the Analyzer does not resolve the directive by searching the Directory list of the program-specific project, it recursively searches the Directory list of each of the base projects in the base list in the order that they appear. If a `#include` is not resolved by this extended search, the Analyzer posts an analysis error message for the file that contains the unresolved directive.

To ensure that the Directory list contains directory paths that enable resolution of `#include` directives, you can inspect individual `#include` directives to see how they reference the included files. The following cases contain examples that refer to the directory hierarchy as shown:

```
c:
   my_project
      my_subsys
         my_source.h
```

- For a simple file name, for example:

  ```
  #include "my_source.h"
  ```

  add the path of the directory that contains the file name to the Directory list, in this example `c:\my_project\my_subsys`.

- For a relative file name, for example:

  **`#include "my_subsys\my_source.h"`**

  add the path of the enclosing directory to the Directory list, in this example `c:\my_project`. Note that `c:\my_project\my_subsys` should also be added to the directory list if `my_source.h` is to be reverse engineered.
- For an absolute name, for example:

  **`#include "c:\my_project\my_subsys\my_source.h"`**

  no paths are added to the Directory list.

## Directory List Entries

In the project window, each entry in the Directory list has the following form:

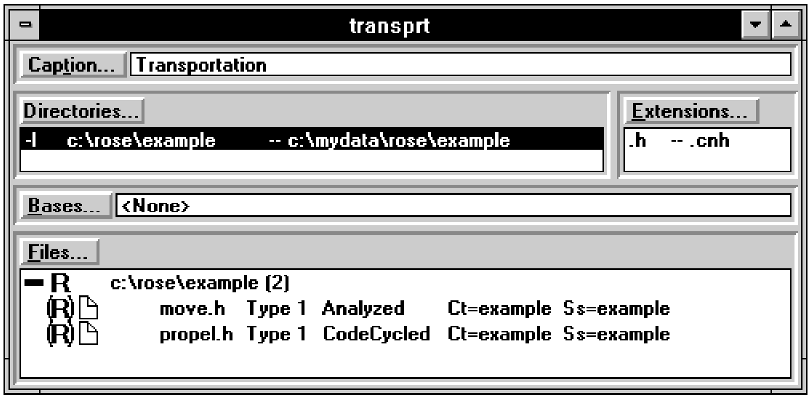
[-I] *directory name — data-directory name*

where:

- *directory name* is the path to a directory containing source-code files or files containing `#include` directives.
- *data-directory* name is the path to the data directory assigned to *directory name.*
- **-I** (if present) indicates that the Analyzer searches the named directory during `#include` resolution.

In the highlighted Directory list entry shown, the directory `c:\rose\example` is searched during `#include` resolution, and is associated with a data directory named `c:\mydata\rose\example`.

All data files produced during the analysis of source files in `c:\rose\example` are placed in `c:\mydata\rose\example`.



Transportation

-I c:\rose\example -- c:\mydata\rose\example

.h -- .cnh

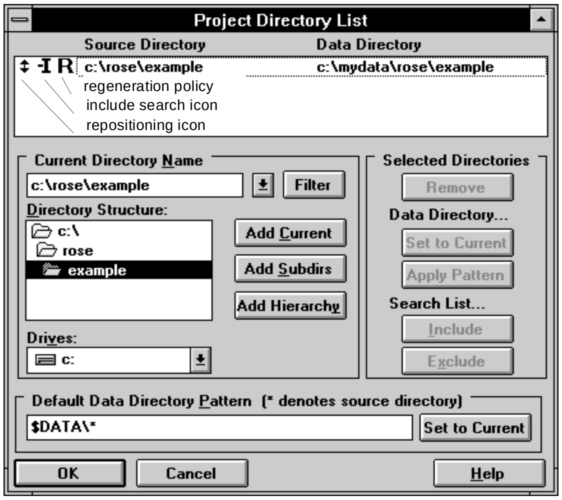
<None>

```
R      c:\rose\example (2)
(R)       move.h   Type 1  Analyzed    Ct=example  Ss=example
(R)       propel.h Type 1  CodeCycled  Ct=example  Ss=example
```

***Figure 30 Example Project Directory List***

## Editing the Directory List

To edit the project Directory list, click **Directory** in the project window, or click **Edit > Directory.** Either action displays the **Project Directory List** dialog box, which presents an entry in its Source Directory and Data Directory lists for each directory currently on the project's Directory list:



c:\rose\example c:\mydata\rose\example

Current Directory Name: c:\rose\example

Directory Structure: c:\ rose example

Drives: c:

Default Data Directory Pattern: $DATA*

***Figure 31 Project Directory List***

Each entry in the Source Directory and Data Directory lists includes:

- A positioning icon
- An include search icon
- A generation policy icon
- The directory’s path
- The path to its associated data directory

By dragging an entry’s positioning icon, you can change the order of directories in the Directory list. This order is relevant to `#include` resolution, since the Analyzer searches directories in their Directory-list order.

An entry’s include search icon indicates whether the associated directory is searched during `#include` resolution. Clicking on an entry’s icon switches it between two settings:

**I - directory's files included in search**
**~~I~~ - directory's files not included in search**

An entry’s regeneration policy icon indicates the associated directory’s code regeneration policy. Clicking on an entry’s icon switches it between two settings:

**R - enabled**
**~~R~~ - disabled**

To add a directory to the Directory list, locate it with the Directory Structure browser. Double-clicking on a directory in this browser displays any subdirectories and places its path in the Current Directory Name box. Alternatively, you can directly enter a directory name into the Current Directory Name box. When the Current Directory Name box contains the name of the directory you wish to add, click **Add Current**. To add each subdirectory of the Current Directory Name to the Directory list, click **Add Subdirs**. To add every subdirectory in the directory tree rooted at Current Directory Name, click **Add Hierarchy**.

When a directory is added to the Source Directory and Data Directory lists by the action of **Add Current, Add Subdirs** or **Add Hierarchy**—or by the action of adding a file to the file list—it’s `#include` search and code regeneration states are set to Included and Enabled Respectively, and its data directory is set using the pattern in the Default Data Directory Pattern box.

The Default Data Directory Pattern box contains a path that includes an asterisk (*****); a source directory's data directory name is created by substituting the source directory's name for the asterisk in the pattern. This mechanism ensures that each data directory be uniquely associated with one source directory, as required by the C++ Analyzer. If you specify a Default Data Directory Pattern that does not include an asterisk—which results in the same data directory being assigned to every source directory—the C++ Analyzer automatically appends an asterisk to the end of your pattern to ensure that unique data directories are assigned.

The Default Data Directory Pattern **Set To Current** button sets the Default Data Directory Pattern to the Current Directory Name suffixed by *.

A newly-created project's Default Data Directory Pattern is set to $DATA*, which means that the data files for each source directory are maintained in a subdirectory of a directory whose path is specified by the Path Map entry for the virtual symbol $DATA.

To change the `#include` search state or data directory for one or more source directories, first select the Source Directory and Data Directory list entries associated with these source directories. To select a single entry in this list box, click on it. To select several contiguous entries, click on the first, depress the **Shift** key, and click on the last. To switch an entry's selection, depress the **Ctrl** key and click on that entry.

To change selected entries' `#include` search states, click Search List **Include** or Search List **Exclude**.

To update selected entries' data directories to match the Default Data Directory Pattern, click Data Directory **Apply Pattern**.

To assign a specific data directory to a source directory, first select the Source Directory and Data Directory entry associated with the source directory. Enter the path to the data directory in the Current Directory Name box (or use the Directory Structure browser to navigate to this directory) and click Data Directory **Set to Current**.

To remove one or more a directories from the Directory list, select their entries in the Source Directory and Data Directory list and click **Remove**.

Directory list changes made through the **Project Directory List** dialog box are abandoned unless you exit the box by clicking **OK**.

## Extension List

The Analyzer uses the Extension list component of a project to:

- Provide an initial filter for files presented as candidates for addition to the project's file list. Typical file name extensions are `.h`, `.hh`, and `.cpp`.
- Associate a module kind with each file name extension used in the project.
- Determine which file name extensions to use when naming the data files that are built during analysis. The Extension list associates a data file extension with each source file extension on the Extension list. When the Analyzer semantically analyzes a source file, it builds a data file whose file name utilizes the extension associated with the source file's extension. Although you can view the associated data file extensions in the Extension list, you cannot change them.

### Module Kind

A source file can represent a module specification or module body; file name extensions of `.h` or `.hh` usually designate a file representing a module specification. The Extension list associates a module kind—specification or body—with each file name extension used in the project. When you add a source file to the file list, the Analyzer sets its module kind to the module kind associated with its file name extension. You can set a file's module kind to override its file name extension's module kind.

When a new file name extension is added to the Extension list—either as a side effect of adding a file to the file list, or through your use of the **Project File Extensions** dialog box—the Analyzer sets the extension's module kind to specification if the extension contains the letter `h`; otherwise, the Analyzer sets the extension's module kind to body. You can edit the Extension list to change a file name extension's module kind if your file name extensions follow a different convention.

## Extension List Entries

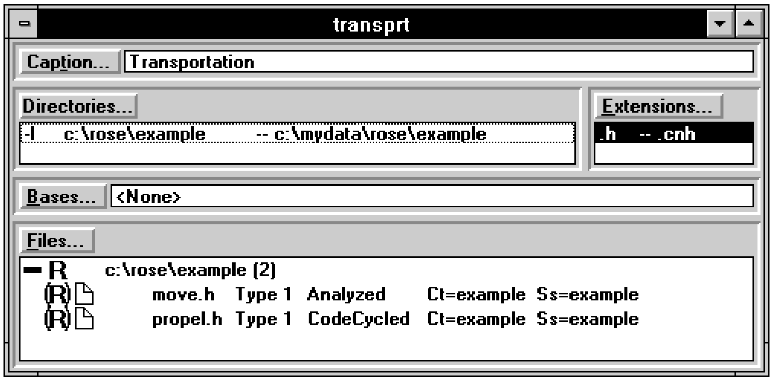
Each entry has the form

*source file extension — data file extension*

where:

- *source file extension* is a file name extension added to the Extension list.
- *data file extension* is the data extension the Analyzer assigned to source file extension.

The highlighted entry shown specifies the data files for source files whose file name extension is `.h` and is assigned a file name extension of `.cnh`:


transprt

Caption... Transportation

Directories...
```
-I     c:\rose\example        -- c:\mydata\rose\example
```

Extensions...
```
.h    -- .cnh
```

Bases... <None>

Files...
```
R      c:\rose\example (2)
           move.h   Type 1  Analyzed     Ct=example  Ss=example
           propel.h Type 1  CodeCycled   Ct=example  Ss=example
```

***Figure 32 Extension List Entries***

## Editing the Extension List

To edit an Extension list, click **Extensions** in the project window, or click **Edit > Extensions**, to display the **Project File Extension** dialog box:

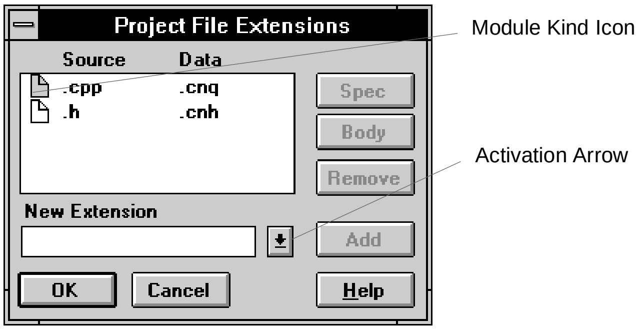


***Figure 33 Extension Dialog Box***

To add a file name extension, you can directly enter it in the New Extension list box and click **Add**, or you can choose from an existing set by clicking on the activation arrow.

Each entry in the **Source—Data** list box includes a module kind icon, the file name extension, and its associated data file name extension. An entry's module kind icon indicates whether files matching the extension are considered specification or body modules. Clicking on an entry's icon switches it between two settings:

🗋 **- specification**
▨ **- body**

You can also change an extension's module kind by selecting its entry and clicking **Spec** or **Body**. You can use this mechanism to change the module kind of several selected entries.

To remove a file name extension from the Extension list, select its entry and click **Remove**.

## File List

A project's File list component specifies the name of each source file to be reverse engineered and, for each file, contains:

- Code regeneration policy
- Module kind

- Analysis type
- Analysis status
- Project category assignment
- Project subsystem assignment

To successfully analyze a program, you should add the following source files to the File list:

- All the header files in the program, or in the portion of your program you want to reverse engineer.
- Any implementation files that contain class definitions at file scope (the Analyzer ignores definitions that are local to function bodies) or for which you are generating code.
- Any files that are context dependent or syntactically incomplete.

Header files that are required only for compilation—for example, system header files like `iostream.h`—do not need to be on the File list. However, the directories containing these files must be on the project's Directory list, or must be on the Directory list of a base project on the project's Base list.

From the File list, you select specific source files for the Analyzer to semantically analyze and extract design data from—you can select all source files in your program, or focus on a particular subset. In addition to the source files you select, the Analyzer recursively analyzes files referenced by `#include` directives in those selected source files.

## Code Regeneration

When incorporating an existing source code class library in your application, you can reverse engineer its design for inclusion in your application's design. To disable code generation for those parts of your application's design that were reverse engineered from class libraries, Rational Rose C++ provides the Generate code generation property. By setting this property to False for each class library module, subsequent code generation from that module is disabled.

To automate the setting of this code generation property for reverse engineered design elements, the C++ Analyzer enables or disables code regeneration at both directory and file granularity. By default, the directories on a project's Directory list and the files on its File list are set to enable code regeneration, meaning that modules extracted from the files on its File list have their Generate property set to True. By sequentially clicking on a file's code regeneration icon in its File list entry, its code regeneration policy can be set to enabled, disabled, or specified by the parent directory. By clicking on a directory's code regeneration icon in its **Project Directory List** dialog box entry, its code regeneration policy is switched between enabled and disabled. If a file's code regeneration policy is disabled, or if its code regeneration policy is specified by a parent directory and that parent directory's code regeneration policy is disabled, then modules extracted from this file have their Generate property set to False; otherwise, these modules have their Generate property set to True.

## Module Kind

A source file can represent a module specification or module body; file name extensions of `.h` or `.hh` usually designate a file representing a module specification. The Extension list associates a module kind—specification or body—with each file name extension used in the project. When you add a source file to the File list, the Analyzer sets its module kind to the module kind associated with its file name extension. You can set a file's module kind to override its file name extension's module kind.

When a new file name extension is added to the Extension list—either as a side effect of adding a file to the File list, or through your use of the **Project File Extensions** dialog box—the Analyzer sets the extension's module kind to `specification` if the extension contains the letter `h`; otherwise, the Analyzer sets the extension's module kind to `body`. You can edit the Extension list to change a file name extension's module kind if your file name extensions follow a different convention.

## Analysis Type

A file that is context sensitive or syntactically incomplete requires special handling during analysis. The File list allows you to identify such files by maintaining an analysis type designator for each File list entry. The analysis type designation allows the analyzer to handle source files which are context-dependent or syntactically incomplete; a source file is context-dependent if it references symbols defined in another source file without explicitly `#include` declaring that source file. A source file's analysis type characterizes its context dependence and syntactic completeness:

***Table 18 Analysis Type Characteristics***

| Analysis Type | Syntactically Complete | Context-Independent |
|---|---|---|
| 1 | yes | yes |
| 2 | yes | no |
| 3 | no | unknown |

When you add a file to the File list, the Analyzer assumes it requires no special handling, and sets its analysis type designator to Type 1; you can change the analysis type designator if you determine that special handling is required.

## Analysis Status

The analysis status designation indicates the Analyzer's knowledge of any previous analysis. Defined values are:

***Table 19 Analysis Status Designation***

| Defined Value | Definition |
|---|---|
| Unknown | The file has not been examined since the project has been opened. |
| Analyzed | The last semantic analysis found no errors, but the source file cannot support round-trip engineering. |
| CodeCycled | The last semantic analysis found no errors, and the source file can support round-trip engineering. |
| Has Errors | The last semantic analysis found errors; these can be displayed by double-clicking on the File list entry. |
| Excluded | Has no data file because the source file is Type3—the Analyzer stores design information for Type 3 files in the data files of the source files that reference them. |

| Defined Value | Definition |
|---|---|
| No Source | Cannot find a file in the file system for this entry in the File list—if you have deleted the file since you added it to the File list, to remove the entry, click **Cut > Edit**. |
| Stale Data | Has potentially out-of-date data file for the file—this status is set during analysis, or by clicking **Action > UpdateStatus**. |
| Unanalyzed | Has no data file—this status can be set by clicking **Action > UpdateStatus** or **Action > DeleteData.** |

The Analyzer does not permanently store analysis status and error information in a project because it can become invalid when you change your program. You must refresh this information by clicking **Action > ShowErrors** after you open a project. Or clicking **Action > UpdateStatus** or **Action > Analyze** also refreshes this information, but their additional functionality is slower than clicking **Action > ShowErrors**.

## Project Category and Project Subsystem Assignments

When you add a source file to the File list, the Analyzer creates a project category assignment and project subsystem assignment by assuming that the simple name of the file's parent directory represents both the category (logical package) and subsystem (component package) to which the file is assigned. This parent directory's simple name is added to the project's Category list and Subsystem list, which can be edited if desired. A file's project category assignment and project subsystem assignment are displayed in its File list entry. You can change these assignments by selecting the entry and clicking **Action > Set File Properties**.

When you export design information to Rational Rose, the **Assign Class To Category Based On Export** option in the **Output** tab of the **Export Options** dialog box can be set to the project category assignment that assigns classes to categories (logical packages). If you instead set this Export Option to either **Class Annotations** or **Directory Containing Definition**, the File list entries continue to display their project category assignments. Similarly, the **Assign Module to Subsystem Based On Export Option** in the **Output** tab of the **Export Options** dialog box can be set to use the project subsystem assignment to associate modules with subsystems (component packages).

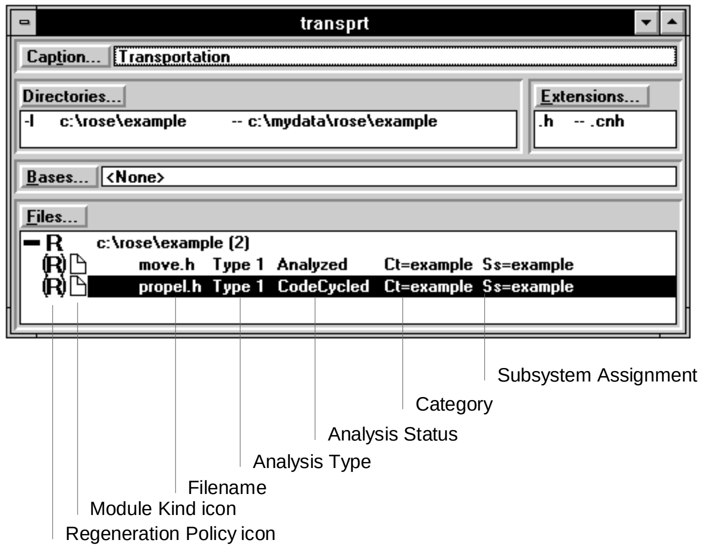
## Naming Conflicts

Rational Rose requires the names of classes and categories (logical packages), and the names of modules and subsystems (component packages) to be unique. When the Analyzer encounters a module or subsystem (component package) that has the same name as an existing subsystem (component package) or module, a new name is generated by appending `.n` to the original name, where `n` is an integer greater than 0 selected to produce a unique name. When the Analyzer encounters a class whose name conflicts with an existing category (logical package) name, or when it encounters a category (logical package) whose name conflicts with an existing class name, the category (logical package) name is made unique by appending ".0" to the original name. The names of classes are never changed.

If errors are detected while analyzing a file, the Analyzer attaches the error messages to the file's entry; you can expand or collapse the entry to display or hide its error messages by clicking **Edit > Expand**. Double-clicking on the entry of a source file with errors expands the entry to display one line for each error message. Double-clicking on one of these error messages displays a File Viewer window on the source file positioned at the point of error.

## File List Entries

The File list maintains an entry for each of its files. You can change the File list's display of its entries by choosing different sort orders using the **View** menu's **Sort** commands, as described in Figure 34. As shown, the fields in a file list entry display its file's regeneration policy icon, module kind icon, simple name, location in the file system, analysis type, analysis status, assigned category, assigned subsystem, and the number of analysis errors found when the file was last analyzed:



Caption... Transportation

Directories... -I c:\rose\example -- c:\mydata\rose\example

Extensions... .h -- .cnh

Bases... <None>

```
R      c:\rose\example (2)
(R)🗋      move.h   Type 1  Analyzed     Ct=example  Ss=example
(R)🗋      propel.h Type 1  CodeCycled   Ct=example  Ss=example
```

*Figure 34 Sort Order*

An entry's regeneration policy icon indicates the associated file's code regeneration policy. Clicking on an entry's icon switches it between four settings:

**(~~R~~) - specified by parent directory, which is disabled**
**(R) - specified by parent directory, which is enabled**
**R - enabled**
**~~R~~ - disabled**

An entry's module kind icon indicates whether the file considers *specification* or *body* modules. Clicking on an entry's icon switches it between four settings:

**🗋 - specified by filename extension, which is a specification**
**🗎 - specification**
**🗋 - specified by filename extension, which is a body**
**🗎 - body**

## Sort Order

The **View** menu's **Sort** commands provide several arrangements of File list entries. To make one of these sorts the default File list order, select it, then click **View > Memorize Changes**, and then click **File > Save**.

### Sort By Pathname

Choose this sort order to arrange File list entries alphabetically by the full paths of the listed files.

A check mark next to **Sort by Pathname** identifies it as the current sort order. In the File list, each entry has four fields:

***Table 20 Sort By Pathname***

| Pathname | Analysis Type | Analysis Status | Category Name |
|---|---|---|---|
| c:\testsrc\obj_key.h | Type 2 | Unanalyzed | keys |
| c:\testsrc\options.h | Type 1 | Analyzed | heaps |
| c:\testsrc\srcblock.h | Type 1 | Has Errors:4 | heaps |

### Sort By Directory

Choose this sort order to arrange the File list entries in groups according to the directories that contain the listed files. Within each group, files are listed alphabetically by their simple names.

A check mark next to **Sort by Directory** identifies it as the current sort order. In the File list, directory names appear as group headings and each entry has four fields:

***Table 21 Sort By Directory***

| File Name | Analysis Type | Analysis Status | Category Name |
|---|---|---|---|
| **-q:\testsrc (3)** | | | |
| obj_key.h | Type 2 | Unanalyzed | keys |
| options.h | Type 1 | Analyzed | heaps |
| srcblock.h | Type 1 | Has Errors:4 | heaps |

### Sort By Simple Name

Choose this sort order to arrange the File list entries alphabetically by the simple names of the listed files.

A check mark next to **Sort by Simple Name** identifies it as the current sort order. In the File list, each entry has five fields:

***Table 22 Sort By Simple Name***

| File Name | Analysis Type | Analysis Status | Category Name | Directory |
|---|---|---|---|---|
| obj_key.h | Type 2 | Unanalyzed | keys | q:\testsrc |
| options.h | Type 1 | Analyzed | heaps | q:\testsrc |
| srcblock.h | Type 1 | Has Errors:4 | heaps | q:\testsrc |

### Sort By Status

Choose this sort order to arrange the File list entries in groups according to their analysis status. Within each group, files are listed alphabetically by their simple names. You might need to refresh the File list before choosing this sort order; if so, click **Actions > UpdateStatus**.

A check mark next to **Sort by Status** identifies it as the current sort order. In the File list, each analysis status appears as a group heading and each entry has five fields, where the number of errors field might be blank:

***Table 23 Sort By Status***

| Number of Errors | File Name | Analysis Type | Category Name | Directory |
|---|---|---|---|---|
| **-Unanalyzed (1)** | | | | |
| 0 | obj_key.h | Type 2 | keys | q:\testsrc |
| **-Analyzed (1)** | | | | |
| 0 | options.h | Type 1 | heaps | q:\testsrc |
| **-Has Errors (1)** | | | | |
| +4 | srcblock.h | Type 1 | heaps | q:\testsrc |

### Sort By Errors

Choose this sort order to arrange the File list entries according to the number of analysis errors associated with them.

This sort order groups File list entries according to their analysis status, which indicates whether errors are present. Entries with Has Errors status have one or more associated errors, and entries with other status have no errors. Within the Has Errors group, files appear in descending order by the number of errors. You might need to refresh the File list before choosing this sort order; if so, click **Actions > ShowErrors**.

A check mark next to **Sort by Errors** identifies it as the current sort order. In the File list, analysis status appears as group headings and each entry has five fields, where the number of errors field might be blank:

***Table 24 Sort By Errors***

| **Number of Errors** | **File Name** | **Analysis Type** | **Category Name** | **Directory** |
|---|---|---|---|---|
| **-Has Errors (2)** | | | | |
| +4 | srcblock.h | Type 1 | heaps | q:\testsrc |
| +1 | heapid.h | Type 1 | heaps | q:\testsrc |
| **-Analyzed (1)** | | | | |
| 0 | options.h | Type 1 | heaps | q:\testsrc |

### Sort By Type

Choose this sort order to arrange the File list entries in groups according to the assigned analysis type. Within each group, files are further grouped by the directory that contains them. Within each subgroup, files appear alphabetically by file name.

A check mark next to **Sort by Type** identifies it as the current sort order. In the File list, analysis types and directory names appear as group headings and each entry has three fields:

***Table 25 Sort By Type***

| File Name | Analysis Status | Category Name |
|---|---|---|
| **-Type 1 (2)** | | |
| **-q:\testsrc (1)** | | |
| options.h | Analyzed | heaps |
| srcblock.h | Has Errors:4 | heaps |
| **-Type 2 (1)** | | |
| **-q:\testsrc (1)** | | |
| obj_key.h | Unanalyzed | heaps |

### Sort By Category

Choose this sort order to arrange the File list entries in groups according to their assigned categories (logical packages). Within each group, files are listed alphabetically by their simple names.

A check mark next to **Sort by Category** identifies it as the current sort order. In the File list, category names appear as group headings and each entry has five fields, where the Number of Errors field might be blank:

***Table 26 Sort By Category***

| Number of Errors | File Name | Analysis Type | Analysis Status | Directory |
|---|---|---|---|---|
| **-heaps (2)** | | | | |
| 0 | options.h | Type 1 | Analyzed | q:\testsrc |
| +4 | srcblock.h | Type 1 | Has Errors | q:\testsrc |
| **-keys (1)** | | | | |
| 0 | obj_key.h | Type 2 | Unanalyzed | q:\testsrc |

### Sort By Subsystem

Choose this sort order to arrange the File list entries in groups according to their assigned subsystems. Within each group, files are listed alphabetically by their simple names.

A check mark next to **Sort by Subsystem** identifies it as the current sort order. In the File list, subsystem names appear as group headings and each entry has five fields, where the number of errors field might be blank:

***Table 27 Sort By Subsystem***

| **Number of Errors** | **File Names** | **Analysis Type** | **Analysis Status** | **Director y** |
|---|---|---|---|---|
| **-heaps (2)** | | | | |
| 0 | options.h | Type 1 | Analyzed | q:\testsrc |
| +4 | srcblock.h | Type 1 | Has Errors | q:\testsrc |
| **-keys (1)** | | | | |
| 0 | obj_key.h | Type 2 | Unanalyzed | q:\testsrc |

### Sort By Extension

Choose this sort order to arrange the File list entries in groups according to their file name extensions. Within each extension group, files are grouped by directory; within directory groups, files are listed alphabetically by their simple names.

A check mark next to **Sort by Extension** identifies it as the current sort order. In the File list, file extensions appear as major group headings, directory names appear as minor group headings, and each entry has four fields:

***Table 28 Sort By Extension***

| File Name | Analysis Status | Analysis Name | Category Type |
|---|---|---|---|
| **.cpp (2)** | | | |
| **q:\testsrc** | | | |
| options.cpp | Type 1 | Analyzed | heaps |
| srcblock.cpp | Type 1 | Has Errors:1 | heaps |
| **.h (3)** | | | |
| **q:\testsrc** | | | |
| obj_key.h | Type 2 | Unanalyzed | keys |
| options.h | Type 1 | Analyzed | heaps |
| srcblock.h | Type 1 | Has Errors:2 | heaps |

## Editing the File List

To edit the project's File list, click **Files** in the **Project** window, or click **Edit > File List** to display the **Project Files** dialog box. This dialog box adds and removes directories from the Directory list:

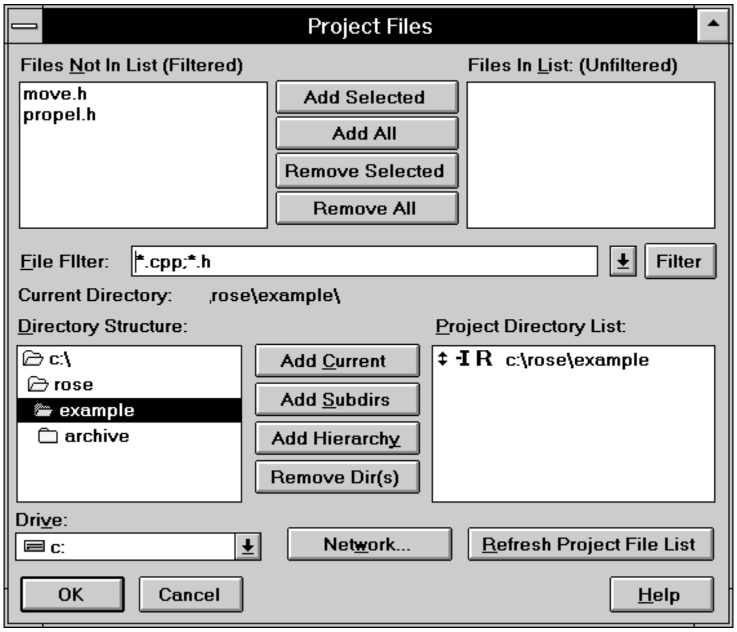


*Figure 35 Edit the File List*

**To add one or more files to the File list:**

1. If the directory containing the files you want to add is already on the Directory list, double-click on its entry in the **Project Directory List** box. Otherwise, use the directory system browser in the **Directory Structure** box to double-click on the directory that contains the files you want to add. On Windows platforms, use the **Drive** box and **Network** button to re-root the **Directory Structure** browser if necessary.
2. Inspect the **Files Not In List** box, which contains files from the selected directory that match the criteria in the **File Filter** box. This box is initialized with a set of patterns from the project's Extension

list. An ***** (asterisk) in these patterns matches 0 or more characters, and a **?** (question mark) matches a single character. To specify multiple patterns, separate them with semicolons.

If your Extension list contains `.h` and `.cpp`, for example, the File Filter is initialized to `*.h;*.cpp`. If you want to add files with a `.hh` file name extension, then you would change the **File Filter** to `*.h;*.hh;*.cpp`.

After modifying the **File Filter**, click the **Filter** button to update the **Files Not In List** box.

3. Select one or more files in the **Files Not In List** box, and click **Add Selected** to transfer the selections to the **Files In List** box.

   You can select multiple files in the **Files Not In List** box by dragging the mouse from one entry to another. Clicking on an entry in either list box while depressing the **Ctrl**-key switches that entry's selection status without deselecting any other entries.

   To add all of the files in the **Files Not In List** box to the File list, click **Add All**.

4. Click **OK** to add your selections to the File list, and update the Directory and Extension lists with any additions resulting from your choice of files.

   The regeneration policy of each added file is set to specified by parent directory; the module kind of each added file is set to specified by file name extension.

   The regeneration policy of each added directory (if any) is set to enabled, and the `#include` search state of each added directory is set to included. The module kind of each added file name extension (if any) is set to specification if the file name extension contains the letter `h`. Otherwise, the module kind is set to `body`.

   Click **Cancel** to dismiss the dialog box without making changes to the project.

5. Click **File > Save** to save the changes to the project.

The **Refresh Project File List** button—activated if one or more directories in the project Directory list are selected—adds every qualified source file in each selected directory to the File list. A source file is qualified if it meets the current **File Filter**.

**To remove one or more files from the File list:**

1. In the project Directory list box, double-click on the directory that contains the files you want to remove.
2. Select the unwanted files in the **Files In List** box and click **Remove Selected**. Note that if the selected files do not match the **File Filter** criteria, they are removed from the **Files In List** box but do not appear in the **Files Not In List** box.

   You can select multiple files in the **Files In List** box by dragging the mouse from one entry to another. Clicking on an entry in either check box while depressing the **Ctrl**-key switches that entry's selection status without deselecting any other entries.
3. To remove all of the files in the **Files In List** box, click **Remove All**.
4. Click **OK** to update the File list. The project's Directory list and Extension list are not modified by this procedure.

   Click **Cancel** to dismiss the dialog box without making changes to the project.
5. Click **File > Save** to save the changes to the project.

**To add a directory to the Directory list:**

1. Locate it with the **Directory Structure** browser; double-clicking on a directory in this browser displays its subdirectories (if any) and places its path in the Current Directory text box.
2. When the Current Directory text box contains the name of the directory you wish to add, click **Add Current**.

   To add each subdirectory of the Current Directory to the Directory list, click **Add Subdirs**.

   To add every subdirectory in the directory tree rooted at Current Directory, click **Add Hierarchy**.

   The regeneration policy of each added directory is set to **enabled**, and the `#include` search state of each added directory is set to **included**.
3. Click **OK** to update the project with these changes.

   Click **Cancel** to dismiss the dialog box without making changes to the project.
4. Click **File > Save** to save the changes to the project.

**To remove a directory to the Directory list:**

1. In the project Directory list box, click on the directory that you want to remove.
2. Click **Remove Dir(s)**.
3. Click **OK** to update the project with these changes.

   Click **Cancel** to dismiss the dialog box without making changes to the project.
4. Click **File > Save** to save the changes to the project.

## Base List

It is often the case that the source code representing the program you wish to reverse engineer references files (in `#include` directives) defined in external source code libraries, such as those provided with C++ compilers or domain-specific class libraries. You could add the directories containing these external source files to your project and reverse engineer them along with your source files, but this approach is inefficient when the libraries are used by several engineers.

Instead, you can create an Analyzer project for each external library. You then reference these library-specific projects in your project by adding them your project's Base list. The Analyzer can then successfully resolve references from source files in your project to source files in the external libraries. A project that appears on another project's Base list is referred to as a base project.

If, during preprocessing, the Analyzer cannot resolve all symbols and `#include` directives after searching the files in the directories in the project's Directory list, it searches the directories identified by each base project on the Base list. Each base project's Defined Symbols list and Undefined Symbols list are taken into account during this search.

Each entry in a Base list consists of the file name and caption of a base project. The order in which Base projects appear in the Base list entries is important, since it determines the order in which the C++ Analyzer searches header files, and defines or undefines preprocessor symbols; the first Base project on the Base list is processed first.

### The Default Project: analyzer.pjt

During the Rational Rose C++ installation process, a default project named `analyzer.pjt` is created in your Rational Rose installation directory. A compiler-specific project, representing the source libraries associated with your site's C++ compiler, can also be installed and placed on the Base list of `analyzer.pjt`. This compiler-specific project references PathMap virtual symbols that it resolves to the directories containing these source libraries. During installation, you identify the locations of these directories; PathMap entries defining the appropriate virtual symbols are created using this information. Click **File > New** *without* depressing the **SHIFT** key, and the Analyzer creates the new project by copying `analyzer.pjt`. The new project contains a Base list entry referring to the project representing the source libraries associated with your site's C++ compiler. This is appropriate when the source code you intend to analyze references your compiler's source libraries.

You can use this mechanism to facilitate the use of one or more class libraries used at your site: create a project for each class library, and add these projects to the Base list of `analyzer.pjt`.

Click **File > New** while depressing the **SHIFT** key, the new project is empty, rather than a copy of `analyzer.pjt`.

## Editing the Base List

To edit the project's Base list, click **Bases** in the **Project** window or Click **Edit > Base List** to display the **Base Projects** dialog box:

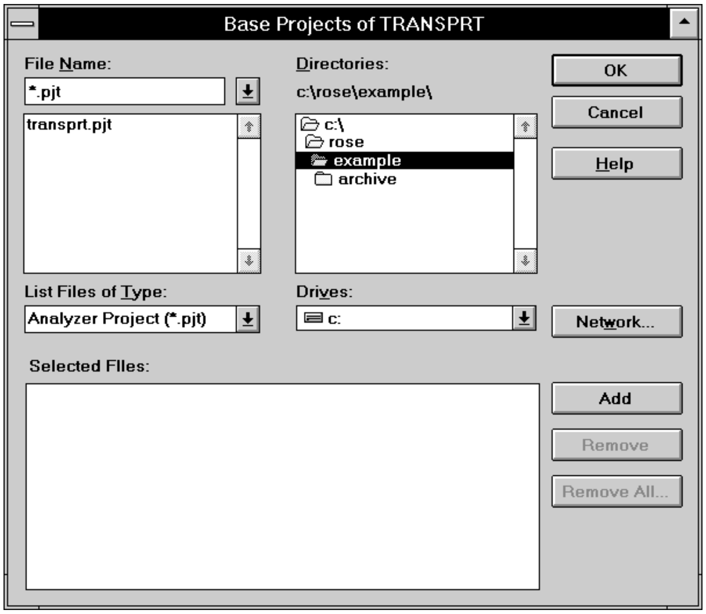


***Figure 36   Editing the Base List***

To add a project to the Base list, use the file requester in the dialog box to select an Analyzer project representing the external library, and then click **Add**. The Analyzer will not permit you to add a project to the Base list if this action would introduce a circular dependency.

To remove a project from the Base list, select its entry in the **Selected Files** list box and click **Remove**.

To change the order of Base list entries, use the mouse to drag them into the appropriate order.

## Defined Symbols List

Use a project's Defined Symbols list to define any preprocessor symbols you want the Analyzer to expand when it semantically analyzes the files in your program; however, the symbols you define should not already be defined in the source files.

The Analyzer utilizes the C++ language definition in ARM draft standard. However, the program you are analyzing might make use of language extensions or predefined macros that the compiler or a class library provides. Furthermore, your program can contain macros that you define via the `-D` compiler option instead of using `#define` directives. In order to analyze such source files, the Analyzer relies on the Defined Symbols list to define any language extensions and undefined macros it is likely to encounter.

In the Defined Symbols list for a compiler-specific base project, you should list:

- A symbol with an empty value for each compiler-defined language extension. For example, keywords such as `far`.
- A symbol with an appropriate value for each compiler-defined preprocessor symbol. For example, predefined macros like `__M_I86__`.

In the Defined Symbols list for a program-specific project, list:

- Any symbol you normally pass to your compiler via the `-D` option.

If the same symbol is defined in a program-specific project and in any of the base projects it references, the Analyzer takes the symbol definition from the program-specific project. Furthermore, if the same symbol is defined in more than one base project, the Analyzer uses the last definition for that symbol, and processes the base projects, in order, from first to last on the Base list.

## Editing the Defined Symbols List

To edit the project's Defined Symbols list, click **Edit > Defined Symbols** to display the **Defines** dialog box:

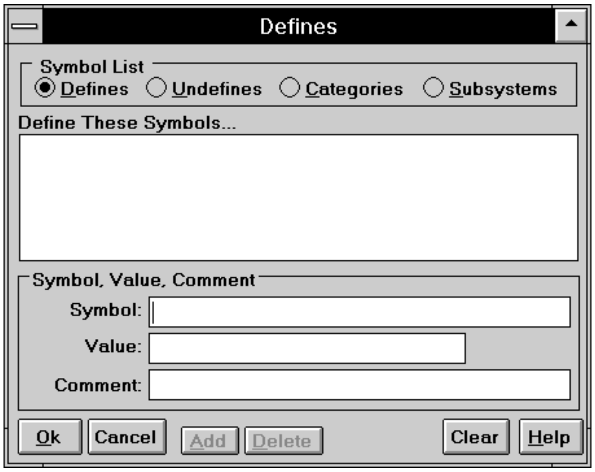


***Figure 37 Defines Dialog Box***

**To add a symbol to the Defined Symbol list:**

1. Enter the symbol's name in the **Symbol** text box.
2. Enter the symbol's value in the **Value** text box.
3. Optionally, enter a comment in the **Comment** text box.
4. Click **Add**.

To remove a symbol from the Defined Symbol list, select its entry in the Define These Symbols text box and click **Delete**.

***Note:*** *You can directly edit the project's Undefined Symbols list, Categories list, or Subsystems list by clicking the appropriate option button in the Symbol List frame.*

## Undefined Symbols List

Use the Undefined Symbols list to specify which symbols are undefined during analysis.

The Undefined Symbols list for a program-specific project should list any symbol that you pass to your compiler via the `-U` option. The Undefined Symbols list for a compiler-specific base project is usually empty.

Each entry in the Undefined Symbols list consists of a symbol followed by an optional comment:

```
symbol #comment
```

The symbol in each entry must be a single valid C++ identifier. For example: `far` or `_M_I86_` or `MYSYM`.

## Editing the Undefined Symbols List

To edit the project's Undefined Symbols list, click **Edit > Undefined Symbols** to display the **Undefines** dialog box:

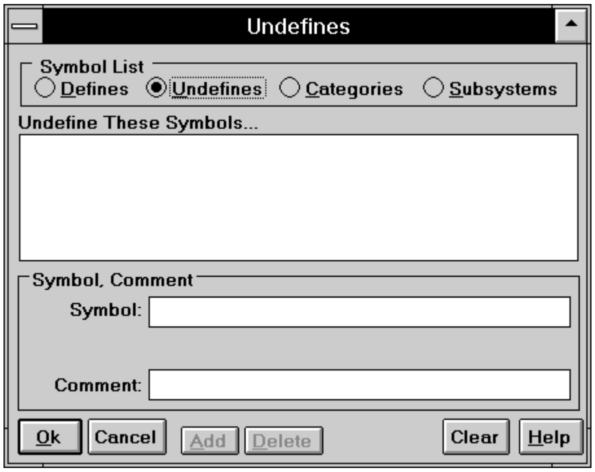


*Figure 38 Undefines Dialog Box*

To add a symbol to the Undefined Symbol list:

1. Enter the symbol's name in the **Symbol** text box.
2. Optionally enter a comment in the **Comment** text box.
3. Click **Add**.

To remove a symbol from the Undefined Symbol list, select its entry in the Undefine These Symbols text box and click **Delete**.

***Note:** You can directly edit the project's Defined Symbols list, Categories list, or Subsystems list by clicking the appropriate option button in the Symbol List frame.*

## Categories List

A project's Categories list contains all categories (logical packages) to which source files in the File list have been assigned. When a source file is added to the File list, the Analyzer uses the simple name of the source file's parent directory as the assigned category (logical package), and adds this simple name to the Categories list. The project's assignment of a source file to a category is referred to as the *project category assignment*.

Each entry in the project's Categories list contains three components:

- The name of a category (logical package).
- If the category (logical package) is to be a controlled unit, the path to the petal file that provides its persistent storage.
- Optionally, the name of the subsystem to which the category (logical package) is assigned.

When you export a project's design information, the **Assign Class to Category Export Option** in the **Output** tab of the **Export Options** dialog box can be set to use the **Project** category assignment to assign classes to categories (logical packages). Alternatively, this **Export Option** can be set to drive this assignment based on **Class Annotation**, or from the **Directory Containing Definition**.

## Editing the Categories List

To edit the project’s Categories list, click **Edit > Categories** to display the **Categories** dialog box:

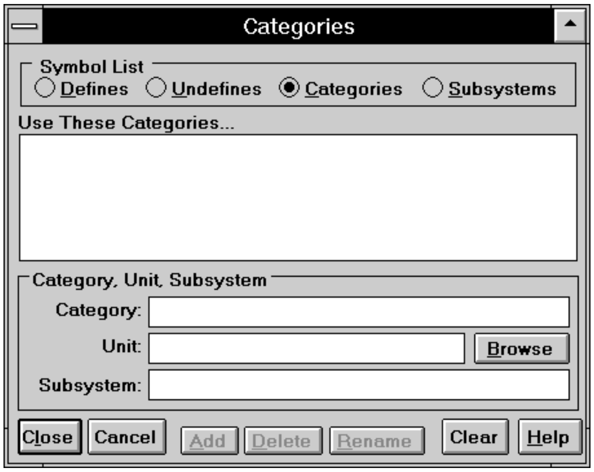


***Figure 39 Categories Dialog Box***

To add a category (logical package) to the Categories list:

1. Enter the category’s (logical package) name in the Category text box.
2. If the category (logical package) is a controlled unit, enter the path to the petal file that provides persistent storage into the Unit text box, or click **Browse** to use a file requester dialog box.
3. If the category (logical package) is assigned to a subsystem, enter the subsystem’s name in the Subsystem text box.
4. Click **Add**.

To remove a category (logical package) from the Categories list, select its entry in the Use These Categories list box and click **Delete**.

***Note:*** *You can directly edit the project’s Defined Symbols list, Undefined Symbols list, or Subsystems list by clicking the appropriate option button in the Symbol List frame.*

## Subsystems List

A project's Subsystems list contains all subsystems to which source files in the File list have been assigned. When a source file is added to the File list, the Analyzer use the simple name of the source file's parent directory as the assigned subsystem, and adds this simple name to the Subsystem list. The project's assignment of a source file to a subsystem is referred to as the project subsystem assignment.

Each entry in the project's Subsystems list contains three components:

- The name of a subsystem.
- If the subsystem is to be a controlled unit, the path to the petal file which provides its persistent storage.
- Optionally, a comment describing the subsystem.

When you export a project's design information to Rational Rose, the **Assign Module to Subsystem Export Option** in the **Output** tab of the **Export Options** dialog box can be set to use the **Project** subsystem assignment to assign modules to subsystems. Alternatively, this **Export Option** can be set to drive this assignment based on **Module Annotation**, or from the **Directory Containing Definition**.

## Editing the Subsystems List

To edit the project's Subsystems list, click **Edit > Subsystems** to display the **Subsystems** dialog box:

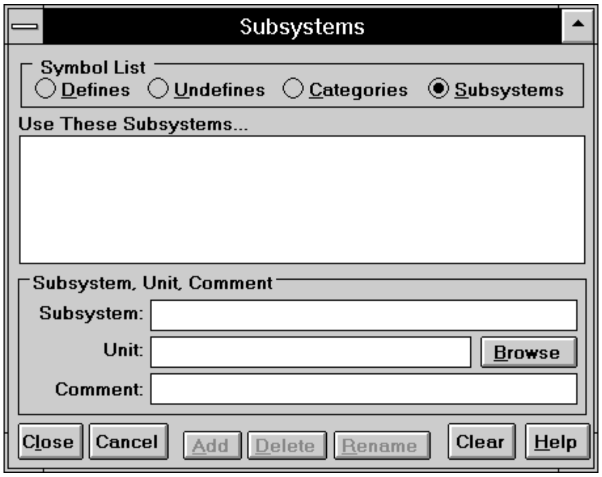


*Figure 40 Subsystems Dialog Box*

**To add a component package to the Subsystems list:**

1. Enter the subsystem's (component package) name in the Subsystem text box.
2. If the subsystem (component package) is a controlled unit, enter the path to the petal file that provides persistent storage into the Unit text box, or click the **Browse** button to use a file selector dialog box.
3. Optionally, enter a comment in the Comment text box.
4. Click **Add**.

To remove a subsystem (component package) from the Subsystems list, select its entry in the Use These Subsystems list box and click **Delete**.

***Note:*** *You can directly edit the project's Defined Symbols list, Undefined Symbols list, or Categories list by clicking the appropriate option button in the Symbol List frame.*

## Type 2 Context

A project's Type 2 Context component is a list of preprocessor directives to be executed before analyzing a Type 2 file contained by that project. When the Analyzer encounters a Type 2 file during semantic analysis, it creates a temporary file containing:

- An identification `#define` directive which defines a symbol derived from the name of the Type 2 file being analyzed.
- The preprocessor directives that you placed in the project's Type 2 Context component.
- A `#include` directive referencing the Type 2 file being analyzed.

This temporary file is then semantically analyzed instead of the Type 2 file. This temporary file is created in the directory containing the current project; if files cannot be created in this directory—due to filesystem access control settings—Type 2 files cannot be successfully semantically analyzed.

Without conditional compilation in the Type 2 Context component's directives, every Type 2 file would be analyzed in the same context; this is usually not appropriate. Each Type 2 file usually needs a different set of `#include` directives to provide the right context. To facilitate this, the Analyzer provides a `#define` symbol derived from the name of the Type 2 file being analyzed, which can be used in the Type 2 Context component directives to conditionally include the header files needed for each Type 2 file in the project. This symbol is constructed as specified in the **Type 2 Context File** dialog box.

Suppose your project includes the Type 2 files `foo.h` and `bar.h`. Further suppose that `foo.h` requires inclusion of `footext.h`, and `bar.h` requires inclusion of `bartext.h`. A Type 2 Context component that supports analysis of both Type 2 files is:

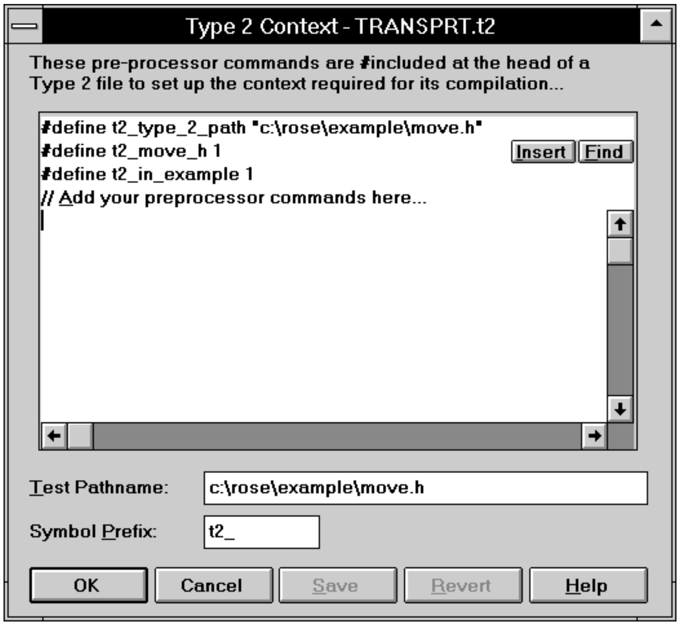
```
#if defined (t2_foo_h)
   #include "footext.h"
#elif defined (t2_bar_h)
   #include "bartext.h"
#endif
```

where `t2_foo_h` is defined only when compiling `foo.h` and `t2_bar_h` is defined only when compiling `bar.h`.

Modifying the Type 2 Context causes all Type 2 files to require re-analysis. Modifying a source file included in the Type 2 Context causes all Type 2 files to reference the modified file to require re-analysis.

## Editing the Type 2 Context

To edit the project's Type 2 Context, click **Edit > Type2 Context** to display the **Type 2 Context** dialog box:



Type 2 Context - TRANSPRT.t2

These pre-processor commands are #included at the head of a Type 2 file to set up the context required for its compilation...

```
#define t2_type_2_path "c:\rose\example\move.h"
#define t2_move_h 1
#define t2_in_example 1
// Add your preprocessor commands here...
```

Test Pathname: c:\rose\example\move.h

Symbol Prefix: t2_

***Figure 41 Type 2 Context Dialog Box***

To identify the special `#include` or `#define` directives that the C++ Analyzer needs to compile Type 2 files:

1. Click **Edit > Type2Context** or click **Type 2** in the **Project** window to display the **Type 2 Context** dialog box.
2. Type one or more `#include` or `#define` statements in the editing window after the **Add your preprocessor commands here** comment. You can use **Insert** to enter the identification symbol for the currently selected File list entry where needed.

   ***Note:*** *Type 2 Context dialog box is non-modal—you can click* **File > Save** *to have changes take effect without dismissing the dialog box.*

3. Click **OK** to add the preprocessor information, or click **Cancel** to discard your additions.
4. Click **File > Save** to save your changes to the project.

## Changing Preprocessor Commands

To modify the `#include` or `#define` directives for Type 2 context files:

1. Click **Edit > Type2Contexts** or click **Type 2** in the **Project** window to display the **Type 2 Context** dialog box.
2. Select the text below the **Add your preprocessor commands.**
3. Click **OK** to add the preprocessor information, or click **Cancel** to discard your additions.
4. Click **File > Save** to save your changes to the project.

*Chapter 5*

# *Analyzer Export Options*

*Export options* enable you to map specific C++ source code constructs into model-notation elements, and then selectively place these elements into one or more exported model files. Rational Rose can then use the exported model files to update an existing model, so that it reflects design changes instituted in source code derived from that existing model. These export options are maintained in a project's export options component.

Export options allow you to control things like: the selection of source files, the specification of language-to-notation mapping, the selection of elements to be exported, the grouping of classes into class categories, the grouping of modules into subsystems, and the fabrication, naming, and population of diagrams.

The export operation can optionally fabricate a class diagram for each exported class category, and can optionally fabricate a module diagram for each exported subsystem. These diagrams can be selectively populated with icons representing exported model elements and the relationships between them. Diagrams fabricated by the C++ Analyzer are placed in the generated model file, enabling viewing and manipulation with Rational Rose.

## Export Option Sets

The export options appropriate for reverse-engineering pre-existing C++ source code are different from those one would use for round-trip engineering. The C++ Analyzer therefore lets you to define *export options sets*—named collections of export option settings—and attach a set to your project. These sets enable you to rapidly and accurately establish the settings appropriate for your current activity. To specify the export option set whose export options govern the export process, click **Action > Export To Rose**.

The Analyzer predefines three standard export option sets:

- FirstLook
- DetailedAnalysis
- RoundTrip

You can modify the export option settings of these predefined sets, or you can create and name new sets. You specify an export option set as your project default by clicking **Action > Export To Rose** and selecting an option set. See "Predefined Export Option Sets" on page 143 for more information.

## Editing Export Options and Sets

To display or modify the export options for a set, click **Edit > Export Options** or, if your current project view shows the Export Options component, click **Export Options** to display the **Export Options** dialog box. Select the export option set you wish to display or modify in the Option Set list box, then click the tab that contains the options of interest.

If you modify one or more export option settings, an asterisk is appended to the export options set name in the Option Set list box, and the **Delete** button changes to **Update**. Clicking **Update** saves your modifications when the dialog box is dismissed. When you click **Update**, the appended asterisk is removed to indicate that the export option set was updated.

The **Export To Rose** dialog box (click **Action > Export To Rose**) also provides an Option Set list box, along with a **Mk Dflt** (make default) button that enables you to designate a specific option set as the project default. A project's default export option set can always be selected by choosing the special symbol **<default>** in an Option Set list box.

Clicking this dialog box's **Edit** button, or clicking on a specific export option in its **Summary of Options** scroll box enables you to modify the export options used during the export operation without modifying the settings of any export option set. The **Export Option** dialog box presented in this situation is identical to those described, except that the title is “One-Use Export Option Set” and the confirmation button is labeled **Use-Set.**

To choose the most-recently modified export option set, select **<latest>** in the Option Set list box.

You can modify the export option settings of a predefined export option set. To restore the settings of the predefined export option sets to their original values, select **<restore standard option settings>** in the Option Set list box.

To create a new export option set, select a predefined set in the Export Options box. Choose the predefined set whose export option settings are closest to the desired values in the new set.

Click **Clone**, and name the new export option set in the **Name Option Set** dialog box the Analyzer then displays. Use the dialog tabs to modify the new set’s export options as desired. To preserve these definitions, click **Update**, click **OK**, and then click **File > Save**.

To delete an export option set, select it in the Export Options box and click **Delete**.

## Predefined Export Option Sets

In addition to allowing you to define your own export option sets, the Analyzer provides three predefined sets:

- FirstLook
- DetailedAnalysis
- RoundTrip

### FirstLook

The FirstLook option set is optimized for high-level examination of pre-existing source code.

- The complete model is exported in one file with extension `.mdl`.
- Classes are assigned to categories (logical packages) based on the project category assignment.
- Categories (logical packages), classes, typedefs, and enumerations are modeled and drawn.
- Object functions and static functions are modeled.

- Aggregate, generalization, and instantiates relationships are modeled and drawn.
- Default cardinality is 1 to 0..1.
- Default protocol is C++.
- Generated class diagrams are named “Reverse Engineered.”
- Neither subsystems (component packages) nor modules are modeled or drawn.
- Class, operator, member, and module annotations are modeled.

## DetailedAnalysis

The DetailedAnalysis option set is optimized for in-depth examination of pre-existing source code.

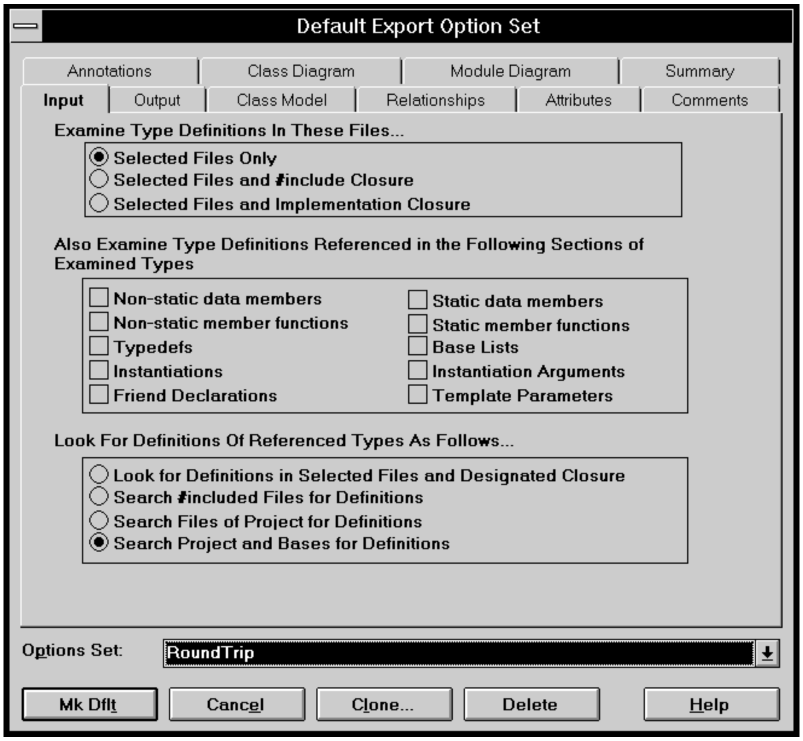
- The complete model is exported in one file with extension `.mdl`.
- Classes are assigned to categories (logical packages) based on the project category assignments.
- Categories (logical packages), classes, typedefs, and enumerations are modeled and drawn.
- Object functions and static functions are modeled.
- Object aggregate, static aggregate, generalization, instantiates, and friendship required relationships are modeled and drawn.
- All dependency relationships are modeled, but not drawn.
- Default cardinality is 1 to 0..1.
- Default protocol is C++.
- Generated class and module diagrams are named “Reverse Engineered.”
- Modules are assigned to class categories based on the project subsystem assignments.
- Module names are derived from directories.
- Class categories and modules are modeled and drawn.
- Visibility relationships are drawn for direct `#include` declarations, `#include` declarations needed from context, and `#include` declarations of type 3 headers.
- Module -> subsystem (component package) and subsystem -> module visibility relationships are drawn.
- Class, operator, member, and module annotations are modeled.

## RoundTrip

The RoundTrip option set is optimized for round-trip engineering.

- The complete model is exported in one file with extension `.red`.
- Classes are assigned to categories (logical packages) based on annotations.
- Categories (logical packages), classes, and typedefs are modeled and drawn.
- Object functions and static functions are modeled.
- Aggregate and instance dependency relationships are modeled, drawn, and named.
- Object dependency, static dependency, generalization, instantiates, and friendship required relationships are modeled and drawn.
- Default cardinality is 1 to 0..1.
- Default protocol is C++.
- Generated class and module diagrams are named “Reverse Engineered.”
- Modules are assigned to subsystems (component packages) based on annotations.
- Module names are derived from annotations.
- Subsystems (component packages) and modules are modeled and drawn.
- Visibility relationships are drawn for direct `#include` declarations, `#include` declarations needed from context, and `#include` declarations of type 3 headers.
- Module -> subsystems (component packages) and subsystems -> module visibility relationships are drawn.
- Class, operator, member, and module annotations are modeled.

## Input Export Options


Default Export Option Set

Examine Type Definitions In These Files...

- [✓ Selected Files Only]
- Selected Files and #include Closure
- Selected Files and Implementation Closure

Also Examine Type Definitions Referenced in the Following Sections of Examined Types

- ☐ Non-static data members
- ☐ Non-static member functions
- ☐ Typedefs
- ☐ Instantiations
- ☐ Friend Declarations
- ☐ Static data members
- ☐ Static member functions
- ☐ Base Lists
- ☐ Instantiation Arguments
- ☐ Template Parameters

Look For Definitions Of Referenced Types As Follows...

- Look for Definitions in Selected Files and Designated Closure
- Search #included Files for Definitions
- Search Files of Project for Definitions
- [✓ Search Project and Bases for Definitions]

Options Set: RoundTrip

*Figure 42 Default Export Option Set Dialog Box*

### Examine Type Definitions In These Files

This export option identifies the source files from which design information is extracted.

### Selected Files Only

This option indicates that design information should be extracted only from files selected in the project's File List.

### Selected Files and #include Closure

This option indicates that design information should be extracted from files selected in the project's File List and the include closure of each selected file.

A file’s include closure is the combination of two sets of files:

1. The set of all files referenced in its `#include` directives.
2. The set of all files contained in the `#include` closure of each file contained in the first set.

This recursive definition produces a list of files within which a referenced declaration must be located.

## Selected Files and Implementation Closure

This option indicates that design information should be extracted from files selected in the project’s File List and the implementation closure of each selected file.

A file’s implementation closure is the combination of three sets of files:

1. The set of all files referenced by its `#include` directives.
2. The set of implementation files associated with the header files contained in the first set.
3. The set of all files contained in the implementation closure of each file contained in the first and second sets.

This recursive definition produces a list of files within which a referenced definition must be located.

## Also Examine Type Definitions

The option, **Also Examine Type Definitions Referenced in the Following Sections of Examined Types** has the following check boxes:

Non-static data members
Static data members
Non-static member functions
Static member functions
Typedefs
Base Lists
Instantiations
Instantiation Arguments
Friend Declarations
Template Parameters

## Look for Definitions of Referenced Types

This option controls the Analyzer's effort to find the definition of types referenced in C++ elements selected in the **Also Examine Type Definitions Referenced in the Following Sections of Examined Types** option.

## Look for Definitions in Selected Files and Designated Closure

This option searches for definitions only where specified by the **Examine Type Definitions in These Files** setting.

## Search #included Files for Definitions

The option, **Search #included Files for Definitions** searches the `#include` closure of an exported file in an attempt to find the definition for each referenced class. Since this is exactly what a C++ compiler does to compile a file, a file that compiles without errors should enable the Analyzer to resolve all references to base classes and templates in the file as well as all aggregation and dependency relationships by value.

Because C++ allows incomplete forward declarations, the Analyzer might not find some dependency and aggregation relationships by reference.

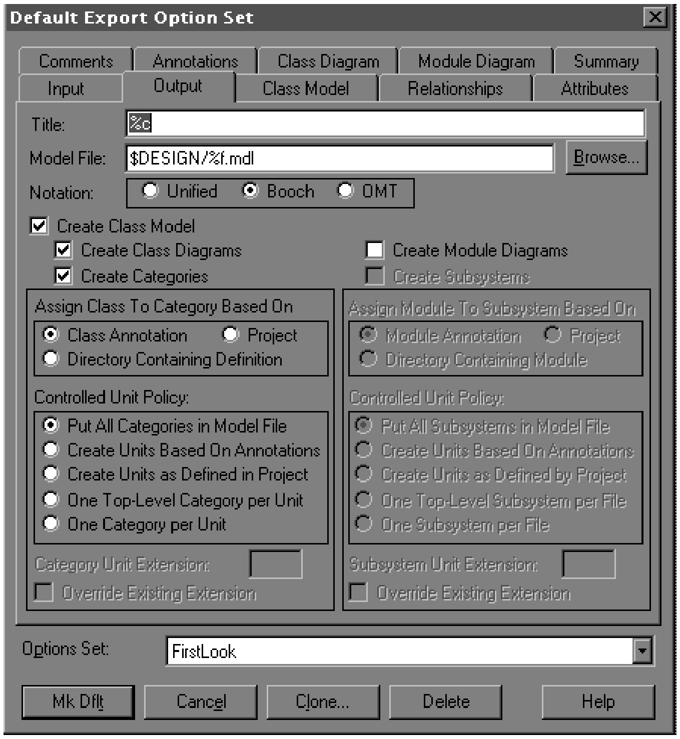
## Search Files of Project for Definitions

The option, **Search Files of Project for Definitions**, searches the `#include` closure of an exported file in an attempt to find the definition for each referenced class. If this search fails, the Analyzer then searches all analyzed files in the project.

## Search Project and Bases for Definitions

The option, **Search Project and Basis for Definitions**, searches the `#include` closure of an exported file in an attempt to find the definition for each referenced class. If this search fails, the Analyzer then searches all analyzed files in the project and its base projects. This search should find all referenced definitions; if not, a warning message is generated.

## Output Export Options


Default Export Option Set

Tabs: Comments, Annotations, Class Diagram, Module Diagram, Summary, Input, Output, Class Model, Relationships, Attributes

Title: %c

Model File: $DESIGN/%f.mdl

Notation: Unified [✓ Booch] OMT

[✓] Create Class Model

[✓] Create Class Diagrams

[✓] Create Categories

[ ] Create Module Diagrams

[ ] Create Subsystems

Assign Class To Category Based On: [✓ Class Annotation] Project, Directory Containing Definition

Controlled Unit Policy: [✓ Put All Categories in Model File] Create Units Based On Annotations, Create Units as Defined in Project, One Top-Level Category per Unit, One Category per Unit

Category Unit Extension:

[ ] Override Existing Extension

Assign Module To Subsystem Based On: [✓ Module Annotation] Project, Directory Containing Module

Controlled Unit Policy: [✓ Put All Subsystems in Model File] Create Units Based On Annotations, Create Units as Defined by Project, One Top-Level Subsystem per File, One Subsystem per File

Subsystem Unit Extension:

[ ] Override Existing Extension

Options Set: FirstLook

***Figure 43 Output Export Options***

### Title

The string contained in this text box initializes the **Title** box in the **Export To Rose** dialog box, which specifies the title placed on every generated class and module diagram.

The following format code substitutions are supported:

***Table 29 Format Code Substitutions***

| | |
|---|---|
| %c | Replace by the project Caption. |
| %d | Replaced by the simple name of the directory that contains the project. |

| | |
|---|---|
| %e | Replaced by the simple name generated from the selected File List entries.<br><br>If a single file is selected, the resulting name is the simple name of the selected file. If multiple files are selected, and every selected file is assigned to the same category (logical package), then the resulting name is the name of the category (logical package); otherwise, the resulting name is the name of the project. |
| %p | Replaced by the project name. |

## Model File

The string contained in this text box initializes the **File** box in the **Export To Rose** dialog box, which specifies the pathname of the generated petal file(s).

The following format code substitutions are supported:

***Table 30 Model File Format Code Substitutions***

| | |
|---|---|
| %c | Replaced by the project Caption. |
| %d | Replaced by the simple name of the directory that contains the project. |
| %e | Replaced by the full name of the directory that contains the project. |
| %f | Replaced by a simple name generated from the selected File List entries.<br><br>If a single file is selected, the resulting name is the simple name of the selected file. If multiple files are selected, and every selected file is assigned to the same category (logical package), then the resulting name is the name of the category (logical package); otherwise, the resulting name is the name of the project. |
| %p | Replaced by the project name. |

## Notation Buttons

These option buttons determine whether diagrams are generated with the UML, OMT, Booch notation.

## Browse Button

This button displays the **Specify Design File** dialog box which enables you to select the **Model File** by traversing the file system.

## Create Class Model

This check box enables the inclusion of classes and related elements in the generated model containing the reverse engineered design, subject to fine-grained control by export options on the **Class Model** tab. If you do not create a class model, then module diagrams can be exported.

## Create Class Diagrams

If this check box is enabled, the Analyzer generates a class diagram in each category (logical package) it encounters. On each class diagram, the Analyzer places icons representing model elements contained in the diagram's parent category (logical package), as directed by the export options on the **Class Diagram** tab.

## Create Categories

This check box enables the inclusion of categories (logical packages) in the generated model. If this check box is disabled, all classes are placed in the <top level> Logical View.

## Create Module Diagrams

If this check box is enabled, the Analyzer generates a module diagram in each subsystem (component package) it encounters. On each module diagram, the Analyzer places icons representing model elements contained in the diagram's parent subsystem (component package), as directed by the export options on the **Module Diagram** tab.

## Create Subsystems

This check box enables the inclusion of subsystems (component packages) in the generated model. If this check box is disabled, all modules are placed in the Component View.

## Assign Class to Category Based On

### Class Annotation

Classes are assigned to categories (logical packages) based on information extracted from source code annotations. If a class without the necessary annotation is encountered, the Analyzer will assign it a category (logical package) based on its project category assignment.

### Project

Classes are assigned to categories (logical packages) based on the project category assignment, which is maintained by each source file's entry in the project's File List. If there is no project category assignment for the file, it will be assigned to a category (logical package) whose name is the simple name of the file's parent directory. Any category (logical package) annotation present is ignored.

#### Directory Containing Definition

Classes will be assigned to categories (logical packages) whose names are the simple names of their file's parent directory. The project category assignment is ignored.

### Controlled Unit Policy (Categories)

#### Put All Categories in Model File

Categories (logical packages) are not designated as controlled units.

#### Create Units Based On Annotations

Categories (logical packages) are assigned to controlled units as specified by unit annotations in the source code; each exported controlled unit is represented by a separate petal file.

#### Create Units as Defined in Project

Categories (logical packages) in the project's Categories List are designated as controlled units if they have been assigned a unit name; each exported controlled unit is represented by a separate petal file. Any unit annotations present are ignored.

#### One Top-level Category per Unit

Each category (logical package) is designated as a controlled unit; each exported controlled unit is represented by a separate petal file.

#### One Category per Unit

Every category (logical package) in the design is designated as a controlled unit; each exported controlled unit is represented by a separate petal file.

### Category Unit Extension

This text box specifies the file name extension used for separate petal files containing a category (logical package).

## Override Existing Extension (Categories)

If enabled, the user-specified extension in the **Category Unit Extension** text boxes will replace any extension specified in an annotation or in the project. The user-specified extensions can also be used if a unit in an annotation or the project has no specified extension.

## Assign Module to Subsystem based on

The following determine how to assign a module to a subsystem:

### Module Annotation

Modules will be assigned to subsystems (component packages) based on information extracted from source code annotations. If a module is missing the required annotation, the Analyzer will assign the module to the subsystem (component package) specified by the module's project-subsystem assignment.

### Project

Modules will be assigned to subsystems (component packages) based on the project-subsystem assignment, which is maintained by each source file's entry in the project's File List. If there is no project subsystem-assignment for the file, it is assigned to the subsystem (component package) whose name is the simple name of the file's parent directory. Any subsystem (component package) annotation present is ignored.

### Directory Containing Module

Modules are assigned to subsystems (component packages) whose names are the simple names of their file's parent directory. The project-subsystem assignment is ignored.

## Controlled Unit Policy (Subsystems)

### Put All Subsystems in Model File

Subsystems (component packages) are not designated as controlled units.

**Create Units as Defined by Project**

Subsystems (component packages) in the project's Subsystems list are designated as controlled units if they have been assigned a unit name; each exported controlled unit is represented by a separate petal file.

**One Component View per Unit**

Each component view will be designated as a controlled unit; each exported controlled unit is represented by a separate petal file.

**One Subsystem per Unit**

Every subsystem (component package) in the design is designated as a controlled unit; each exported controlled unit is represented by a separate petal file.

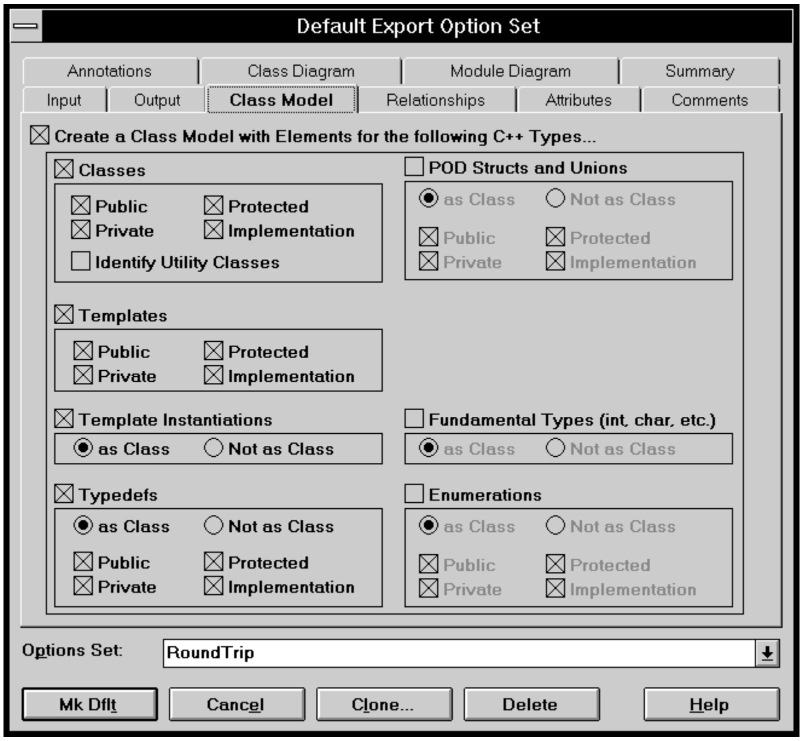
**Subsystem Unit Extension**

This text box specifies the file name extension used for separate petal files containing a subsystem (component package).

**Override Existing Extension (Subsystems)**

If enabled, the user-specified extension in the **Subsystem Unit Extension** text boxes will replace any extension specified in an annotation or in the project. The user-specified extensions are also used if a unit in an annotation or the project has no specified extension.

## Class Model Export Options


Default Export Option Set

Annotations | Class Diagram | Module Diagram | Summary
Input | Output | Class Model | Relationships | Attributes | Comments

[✓ Create a Class Model with Elements for the following C++ Types...]

[✓ Classes]
[✓ Public] [✓ Protected]
[✓ Private] [✓ Implementation]
Identify Utility Classes

POD Structs and Unions
[✓ as Class] Not as Class
[✓ Public] [✓ Protected]
[✓ Private] [✓ Implementation]

[✓ Templates]
[✓ Public] [✓ Protected]
[✓ Private] [✓ Implementation]

[✓ Template Instantiations]
[✓ as Class] Not as Class

Fundamental Types (int, char, etc.)
[✓ as Class] Not as Class

[✓ Typedefs]
[✓ as Class] Not as Class
[✓ Public] [✓ Protected]
[✓ Private] [✓ Implementation]

Enumerations
[✓ as Class] Not as Class
[✓ Public] [✓ Protected]
[✓ Private] [✓ Implementation]

Options Set: RoundTrip

*Figure 44 Class Model Export Options*

## Create a Class Model with Elements

The check box, **Create a Class Model with Elements for the following C++ Types**, enables the inclusion of classes and related elements in the generated model containing the reverse engineered design, subject to fine-grained control by the option settings on this tab.

Several of the following export options include check boxes that provide control based on access—**Public, Private, Protected**, and **Implementation**. Unnested types in specification files have public access; those in body files have implementation access. The access of nested types is determined by the rules of C++.

### Classes

These export options determine which C++ classes are represented as classes in the generated model.

### Templates

These export options determine which C++ templates are represented as classes in the generated model.

### Template Instantiations

These export options determine whether C++ template instantiations are represented as classes in the generated model.

### Typedefs

These export options determine which C++ typedefs are represented as classes in the generated model.

### POD Structs and Unions

These export options determine which C++ “Plain Old Data” structs and unions are represented as classes in the generated model.

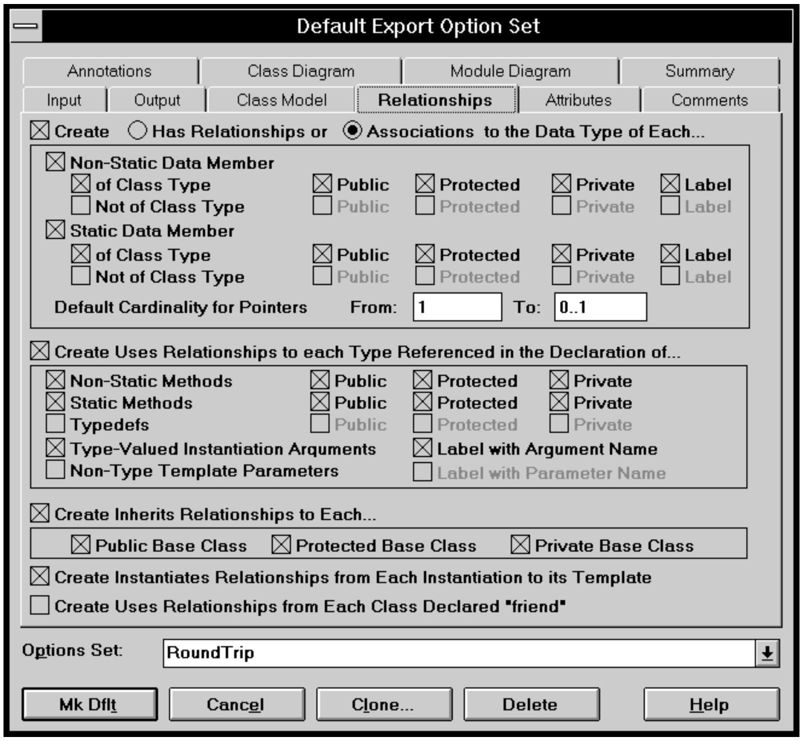
### Fundamental Types (int, char, etc.)

These export options determine whether C++ fundamental types are represented as classes in the generated model.

### Enumerations

These export options determine which C++ enumerations are represented as classes in the generated model.

## Relationships Export Options



[✓ Create] Has Relationships or [✓ Associations] to the Data Type of Each...

[✓ Non-Static Data Member]
- [✓ of Class Type] [✓ Public] [✓ Protected] [✓ Private] [✓ Label]
- Not of Class Type Public Protected Private Label

[✓ Static Data Member]
- [✓ of Class Type] [✓ Public] [✓ Protected] [✓ Private] [✓ Label]
- Not of Class Type Public Protected Private Label

Default Cardinality for Pointers From: 1 To: 0..1

[✓ Create Uses Relationships to each Type Referenced in the Declaration of...]
- [✓ Non-Static Methods] [✓ Public] [✓ Protected] [✓ Private]
- [✓ Static Methods] [✓ Public] [✓ Protected] [✓ Private]
- Typedefs Public Protected Private
- [✓ Type-Valued Instantiation Arguments] [✓ Label with Argument Name]
- Non-Type Template Parameters Label with Parameter Name

[✓ Create Inherits Relationships to Each...]
- [✓ Public Base Class] [✓ Protected Base Class] [✓ Private Base Class]

[✓ Create Instantiates Relationships from Each Instantiation to its Template]

Create Uses Relationships from Each Class Declared "friend"

Options Set: RoundTrip

***Figure 45 Relationships Export Options***

## Create

This check box enables the mapping of C++ data members to relationships in the generated model, subject to fine-grained control by the following export options:

### Has (Aggregate) Relationships or Associations

The export option, **Has Relationships** or **Associations to the Data Type of Each**, determine whether C++ data members are represented as aggregate (has) relationships or as associations.

### Non-Static Data Member

These export options determine which non-static C++ data members are represented in the generated model.

### Static Data Member

These export options determine which static C++ data members are represented in the generated model.

### Default Cardinality for Pointers

These export options specify the default cardinality that you want the Analyzer to use for relationships representing C++ pointer objects. The C++ Analyzer cannot determine the cardinality of a C++ pointer object from its definition. Set this option to the dominant usage of pointers in the application you are modeling, typically **From:** *1* **To:** *0..1* .

***Note:*** *Each pointer definition generated by the Rational Rose C++ is accompanied by an annotation that specifies its pointer's cardinality. Enable modeling of class annotations for cardinality in the* **Annotation** *export options dialog tab if you want the cardinality from annotations used instead of the default cardinality—this is the recommended setting for round-trip engineering.*

## Create Uses Relationships

The check box, **Create Uses Relationships to Each Type Referenced in the Declaration of** enables the creation of dependency relationships in the generated model containing the reverse engineered design, subject to fine-grained control by the following export options:

### Non-Static Methods

These export options determine whether to create dependency relationships to each type referenced in the declaration of non-static methods from the classes containing these methods.

### Static Methods

These export options determine whether to create *dependency* relationships to each type referenced in the declaration of static methods from the classes containing these methods.

### Typedefs

These export options determine whether to create dependency relationships to each type referenced in the declaration of typedefs from the classes containing these typedefs.

### Type-Valued Instantiation Arguments

These export options determine whether to create dependency relationships to each type referenced in the declaration of type-valued instantiated arguments from the classes representing these instantiations, and how these relationships are named.

### Non-Type Template Parameters

These export options determine whether to create dependency relationships to each type referenced in the declaration of non-type template parameters from the classes representing these templates, and how these relationships are named.

## Create Inherits Relationships to Each

This check box enables the creation of generalization/inheritance relationships in the generated model containing the reverse engineered design, subject to fine-grained control by the following export options:

### Public Base Class

This export option determines whether to create generalization/inheritance relationships between classes and their public base classes.

### Protected Base Class

This export option determines whether to create generalization/inheritance relationships between classes and their protected base classes.

### Private Base Class

This export option determines whether to create generalization/inheritance relationships between classes and their private base classes.

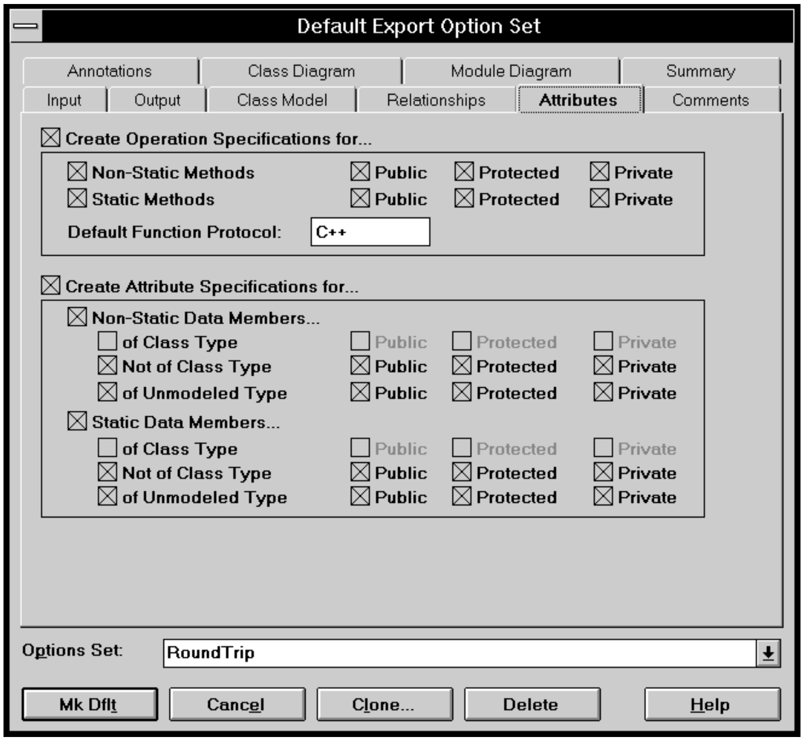
## Create Instantiates Relationships from Each Instantiation to its Template

This check box enables the creation of instantiates relationships in the generated model.

## Create Uses Relationships from Each Class Declared "friend"

This check box enables the creation of *dependency* relationships between classes and their "friends".

## Attributes Export Options


Default Export Option Set

☒ Create Operation Specifications for...

|  |  |  |  |
|---|---|---|---|
| ☒ Non-Static Methods | ☒ Public | ☒ Protected | ☒ Private |
| ☒ Static Methods | ☒ Public | ☒ Protected | ☒ Private |

Default Function Protocol: C++

☒ Create Attribute Specifications for...

|  |  |  |  |
|---|---|---|---|
| ☒ Non-Static Data Members... |  |  |  |
| ☐ of Class Type | ☐ Public | ☐ Protected | ☐ Private |
| ☒ Not of Class Type | ☒ Public | ☒ Protected | ☒ Private |
| ☒ of Unmodeled Type | ☒ Public | ☒ Protected | ☒ Private |
| ☒ Static Data Members... |  |  |  |
| ☐ of Class Type | ☐ Public | ☐ Protected | ☐ Private |
| ☒ Not of Class Type | ☒ Public | ☒ Protected | ☒ Private |
| ☒ of Unmodeled Type | ☒ Public | ☒ Protected | ☒ Private |

Options Set: RoundTrip

***Figure 46 Attributes Export Options***

## Create Operation Specifications for

This check box enables the mapping of C++ methods (member functions) to operations in the generated model containing the reverse engineered design, subject to fine-grained control by the following export options:

#### Non-Static Methods

These export options determine which non-static methods are represented as operations in the generated model.

#### Static Methods

These export options determine which static methods are represented as operations in the generated model.

#### Default Function Protocol

This export option specifies the protocol field in each generated operation specification; usually, **C++** is the proper value.

***Note:*** *Each operation definition generated by the Rational Rose C++ code generator is accompanied by an annotation that specifies its operation's protocol. Enable modeling of operation annotations for protocol in the Annotation export options dialog tab if you want the protocol from annotations used instead of the default protocol—this is the recommended setting for round-trip engineering.*

### Create Attribute Specifications for

This check box enables the mapping of C++ data members to attributes in the generated model containing the reverse engineered design, subject to fine-grained control by the following export options:

***Note:*** *The Analyzer does not enforce mutual exclusivity between export options governing whether a data member is to be modeled as an attribute and export options governing whether that same data member is to be modeled as a relationship.*

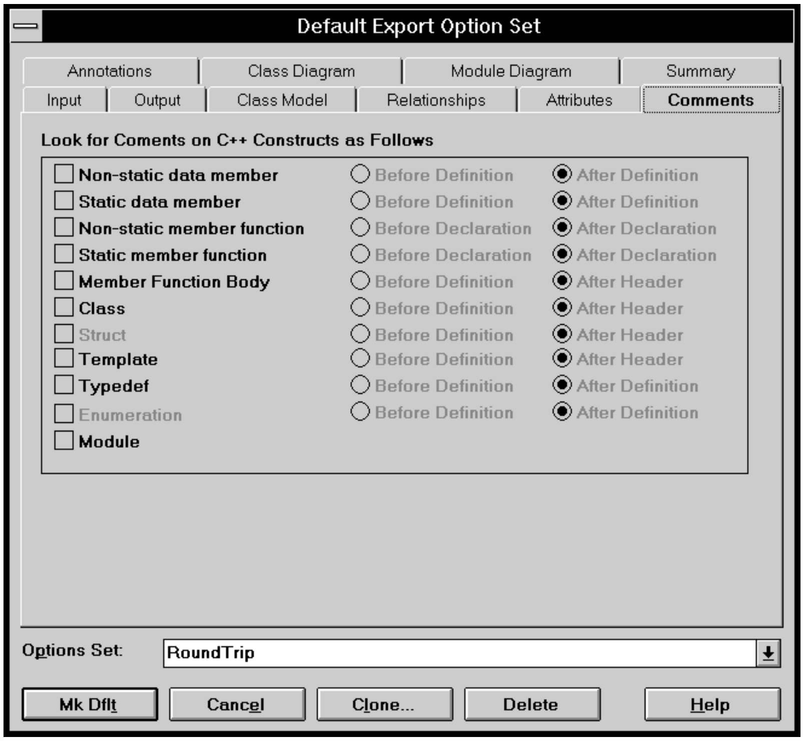
#### Non-Static Data Members

These export options determine which non-static data members are represented as attributes in the generated model.

#### Static Data Members

These export options determine which static data members are represented as attributes in the generated model.

## Comments Export Options



Default Export Option Set

Tabs: Annotations | Class Diagram | Module Diagram | Summary | Input | Output | Class Model | Relationships | Attributes | Comments

Look for Coments on C++ Constructs as Follows

| | | |
|---|---|---|
| ☐ Non-static data member | ○ Before Definition | ● After Definition |
| ☐ Static data member | ○ Before Definition | ● After Definition |
| ☐ Non-static member function | ○ Before Declaration | ● After Declaration |
| ☐ Static member function | ○ Before Declaration | ● After Declaration |
| ☐ Member Function Body | ○ Before Definition | ● After Header |
| ☐ Class | ○ Before Definition | ● After Header |
| ☐ Struct | ○ Before Definition | ● After Header |
| ☐ Template | ○ Before Definition | ● After Header |
| ☐ Typedef | ○ Before Definition | ● After Definition |
| ☐ Enumeration | ○ Before Definition | ● After Definition |
| ☐ Module | | |

Options Set: RoundTrip

*Figure 47 Comments Export Options*

### Look for Comments on C++ Constructs as Follows

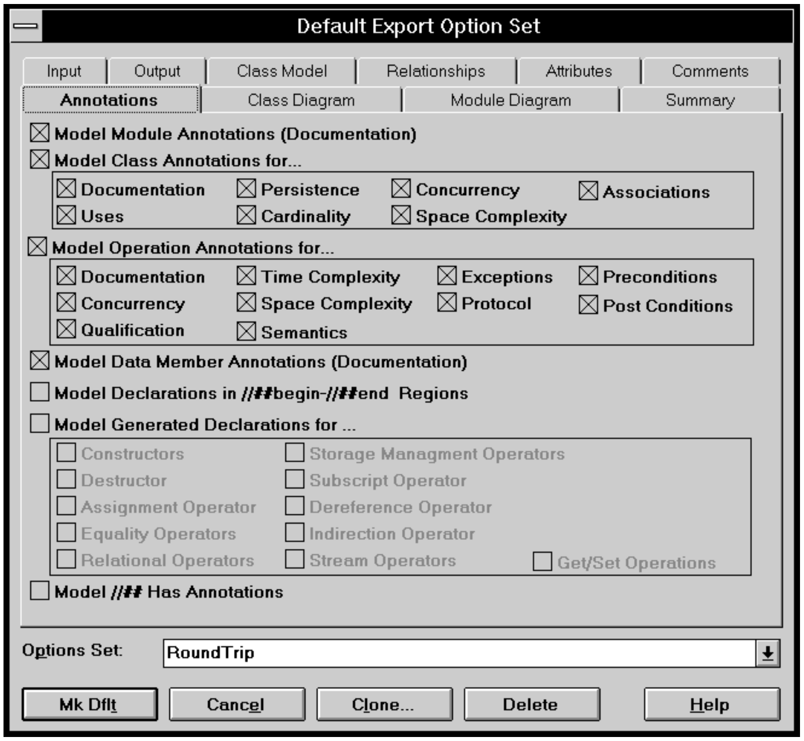
The check box in each entry enables the capture of comments from the designated C++ language construct for inclusion in the documentation field of the generated component specification. If the designated construct is not being modeled, its associated check box is disabled.

The option buttons in each entry control how the comments are extracted:

***Table 31 Comments on C++ Constructs***

| | |
|---|---|
| Non-static data member | before definition or after definition |
| Static data member | before definition or after definition |
| Non-static member function | before declaration or after declaration |
| Static member function | before declaration or after declaration |
| Member function body | before definition or after header |
| Class | before definition or after header |
| Struct | before definition or after header |
| Template | before definition or after header |
| Typedef | before definition or after definition |
| Enumeration | before definition or after definition |
| Module | |

## Annotation Export Options


Default Export Option Set

Input | Output | Class Model | Relationships | Attributes | Comments
Annotations | Class Diagram | Module Diagram | Summary

☒ Model Module Annotations (Documentation)
☒ Model Class Annotations for...
☒ Documentation ☒ Persistence ☒ Concurrency ☒ Associations
☒ Uses ☒ Cardinality ☒ Space Complexity
☒ Model Operation Annotations for...
☒ Documentation ☒ Time Complexity ☒ Exceptions ☒ Preconditions
☒ Concurrency ☒ Space Complexity ☒ Protocol ☒ Post Conditions
☒ Qualification ☒ Semantics
☒ Model Data Member Annotations (Documentation)
☐ Model Declarations in //##begin-//##end Regions
☐ Model Generated Declarations for ...
☐ Constructors ☐ Storage Managment Operators
☐ Destructor ☐ Subscript Operator
☐ Assignment Operator ☐ Dereference Operator
☐ Equality Operators ☐ Indirection Operator
☐ Relational Operators ☐ Stream Operators ☐ Get/Set Operations
☐ Model //## Has Annotations

Options Set: RoundTrip

*Figure 48 Annotation Export Options*

### Model Module Annotations (Documentation)

This check box enables module documentation annotations to be placed in the documentation field of reverse engineered module specifications.

### Model Class Annotations for

The following check boxes are available in **Model Class Annotations for**.

#### Documentation

This check box enables class documentation annotations to be placed in the documentation field of reverse engineered class specifications.

### Persistence

This check box enables class persistence annotations to be placed in the persistence field of reverse engineered class specifications.

### Concurrency

This check box enables class concurrency annotations to be placed in the concurrency field of reverse engineered class specifications.

### Associations

This check box enables class association annotations to create their specified association relationships in the reverse engineered model.

### Dependency

This check box enables class association annotations to create their specified relationships in the reverse engineered model.

### Cardinality

This check box enables class cardinality annotations to be placed in the cardinality field of reverse engineered class specifications.

### Space Complexity

This check box enables class space complexity annotations to be placed in the space field of reverse engineered class specifications.

## Model Operation Annotations for

The following check boxes are available in **Model Operation Annotations for**.

### Documentation

This check box enables documentation operation annotations to be placed in the documentation field of reverse engineered operation specifications.

### Time Complexity

This check box enables time complexity operation annotations to be placed in the time field of reverse engineered operation specifications.

### Exceptions

This check box enables exceptions operation annotations to be placed in the exceptions field of reverse engineered operation specifications.

### Preconditions

This check box enables preconditions operation annotations to be placed in the preconditions field of reverse engineered operation specifications.

### Concurrency

This check box enables concurrency operation annotations to be placed in the concurrency field of reverse engineered operation specifications.

### Space Complexity

This check box enables space complexity operation annotations to be placed in the space field of reverse engineered operation specifications.

### Protocol

This check box enables protocol operation annotations to be placed in the protocol field of reverse engineered operation specifications.

### Post Conditions

This check box enables post conditions operation annotations to be placed in the post conditions field of reverse engineered operation specifications.

### Qualification

This check box enables qualification operation annotations to be placed in the qualification field of reverse engineered operation specifications.

### Semantics

This check box enables semantics operation annotations to be placed in the semantics field of reverse engineered operation specifications.

## Model Data Member Annotations (Documentation)

This check box enables data member annotations to be placed in the documentation field of aggregate relationship specifications.

## Model Declarations in //##begin—//##end Regions

This check box enables the reverse engineering of declarations in protected regions.

***Note:*** *This export option should not be enabled when practicing round-trip engineering.*

## Model Generated Declarations

***Note:*** *The export options for model generated declarations should not be enabled when practicing round-trip engineering.*

The following check boxes are available in **Model Generated Declarations for**.

### Constructors

This check box enables the reverse engineering of constructors.

### Destructor

This check box enables the reverse engineering of destructors.

### Assignment Operator

This check box enables the reverse engineering of assignment operators.

### Equality Operators

This check box enables the reverse engineering of equality operators.

### Relational Operators

This check box enables the reverse engineering of relational operators.

### Storage Management Operators

This check box enables the reverse engineering of storage management operators.

### Subscript Operator

This check box enables the reverse engineering of subscript operators.

### Dereference Operators

This check box enables the reverse engineering of dereference operators.

### Indirection Operator

This check box enables the reverse engineering of indirection operators.

### Stream Operators

This check box enables the reverse engineering of stream operators.

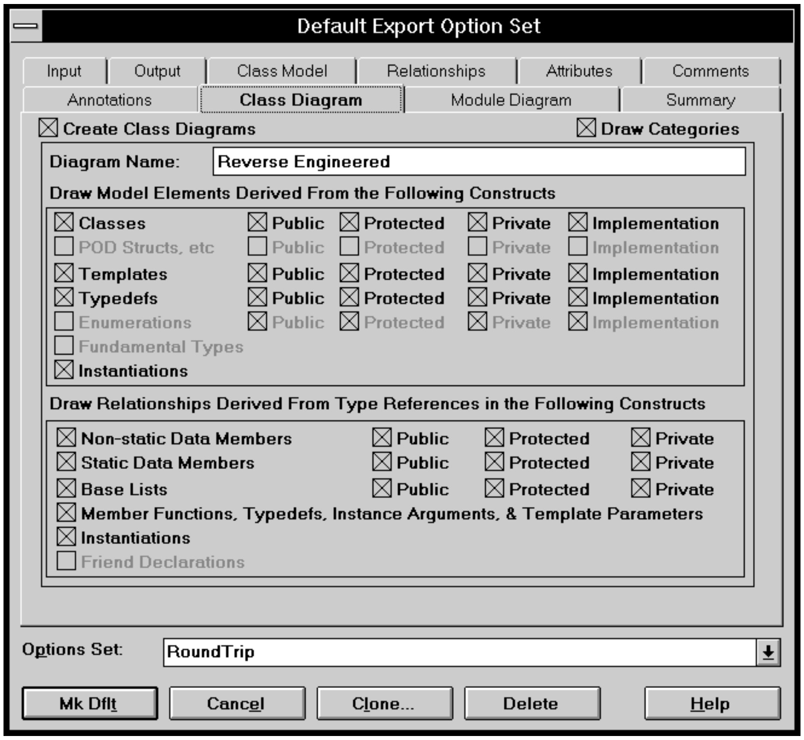
### Get/Set Operations

This check box enables the reverse engineering of get and set operators.

## Model //##aggregate Annotations

This check box enables aggregate annotations to create their specified aggregate relationships in the reverse engineered model.

## Class Diagram Export Options



☒ Create Class Diagrams ☒ Draw Categories

Diagram Name: Reverse Engineered

Draw Model Elements Derived From the Following Constructs

|  |  |  |  |  |
|---|---|---|---|---|
| ☒ Classes | ☒ Public | ☒ Protected | ☒ Private | ☒ Implementation |
| ☐ POD Structs, etc | ☐ Public | ☐ Protected | ☐ Private | ☐ Implementation |
| ☒ Templates | ☒ Public | ☒ Protected | ☒ Private | ☒ Implementation |
| ☒ Typedefs | ☒ Public | ☒ Protected | ☒ Private | ☒ Implementation |
| ☐ Enumerations | ☒ Public | ☒ Protected | ☒ Private | ☒ Implementation |
| ☐ Fundamental Types |  |  |  |  |
| ☒ Instantiations |  |  |  |  |

Draw Relationships Derived From Type References in the Following Constructs

|  |  |  |  |
|---|---|---|---|
| ☒ Non-static Data Members | ☒ Public | ☒ Protected | ☒ Private |
| ☒ Static Data Members | ☒ Public | ☒ Protected | ☒ Private |
| ☒ Base Lists | ☒ Public | ☒ Protected | ☒ Private |
| ☒ Member Functions, Typedefs, Instance Arguments, & Template Parameters |  |  |  |
| ☒ Instantiations |  |  |  |
| ☐ Friend Declarations |  |  |  |

Options Set: RoundTrip

*Figure 49 Class Diagram Export Options*

### Create Class Diagrams

If this check box is enabled, the Analyzer generates a class diagram, named as specified by the **Diagram Name** export option, in each category (logical package) it encounters. On each such class diagram, the Analyzer places icons representing model elements contained in the diagram's parent category (logical package), as directed by the following export options:

### Draw Categories

If this check box is enabled, separate class diagrams depict the model elements contained by each category (logical package); if not enabled, all model elements are depicted on the Logical View.

## Diagram Name

All generated class diagrams have the same name, as specified by the contents of this text box.

## Draw Model Elements Derived From the Following Constructs

The check boxes in each entry determine whether icons representing model elements derived from the entry's C++ construct will be depicted in generated class diagrams. If a construct is not being modeled, its associated check box is disabled.

Classes

POD ("Plain Old Data") Structs and Unions

Templates

Typedefs

Enumerations

Fundamental Types

Instantiations

## Draw Relationships Derived From Type References in the Following Constructs

The check boxes in each entry determine whether icons representing relationships derived from type references in the entry's C++ construct will be depicted in generated class diagrams. If a relationship is not being modeled, its associated check box is disabled.

Non-static Data Members

Static Data Members

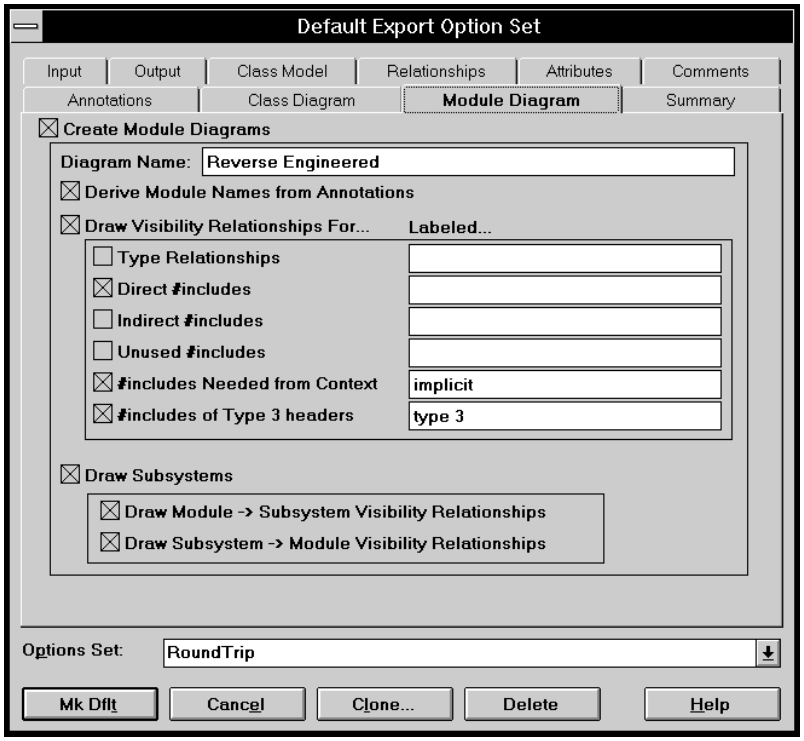
Base Lists

Member Functions, Typedefs, Instance Arguments, and Template Parameters

Instantiations

Friend Declarations

## Module Diagram Export Options



Default Export Option Set

Input | Output | Class Model | Relationships | Attributes | Comments
Annotations | Class Diagram | Module Diagram | Summary

[✓] Create Module Diagrams

Diagram Name: Reverse Engineered

[✓] Derive Module Names from Annotations

[✓] Draw Visibility Relationships For... Labeled...

| | |
|---|---|
| [ ] Type Relationships | |
| [✓] Direct #includes | |
| [ ] Indirect #includes | |
| [ ] Unused #includes | |
| [✓] #includes Needed from Context | implicit |
| [✓] #includes of Type 3 headers | type 3 |

[✓] Draw Subsystems

[✓] Draw Module -> Subsystem Visibility Relationships

[✓] Draw Subsystem -> Module Visibility Relationships

Options Set: RoundTrip

*Figure 50 Module Diagram Export Options*

### Create Module Diagrams

If this check box is enabled, the Analyzer generates a module diagram, named as specified by the Diagram Name export option, in each subsystem (component package) it encounters. On each such module diagram, the Analyzer places icons representing model elements contained in the diagram’s parent subsystem (component package), as directed by the following export options:

### Diagram Name

All generated class diagrams have the same name, as specified by the contents of this text box.

## Derive Module Names from Annotations

This option indicates that module names are specified by annotations in the selected source code files; if this option is not selected, module names are derived from file names.

## Draw Visibility Relationships For... Labeled...

The check box in each entry determines whether visibility relationship icons representing the entry's C++ construct are shown in generated module diagrams; the entry's text box specifies how such relationship icons are to be labeled.

Type Relationships

Direct `#include` declarations

Indirect `#include` declarations

Unused `#include` declarations

`#include` declarations needed from context

`#include` declarations of Type 3 headers

## Draw Subsystems

If this check box is enabled, separate module diagrams will depict each subsystem's modules; if not enabled, all modules are shown on a single top level module diagram.

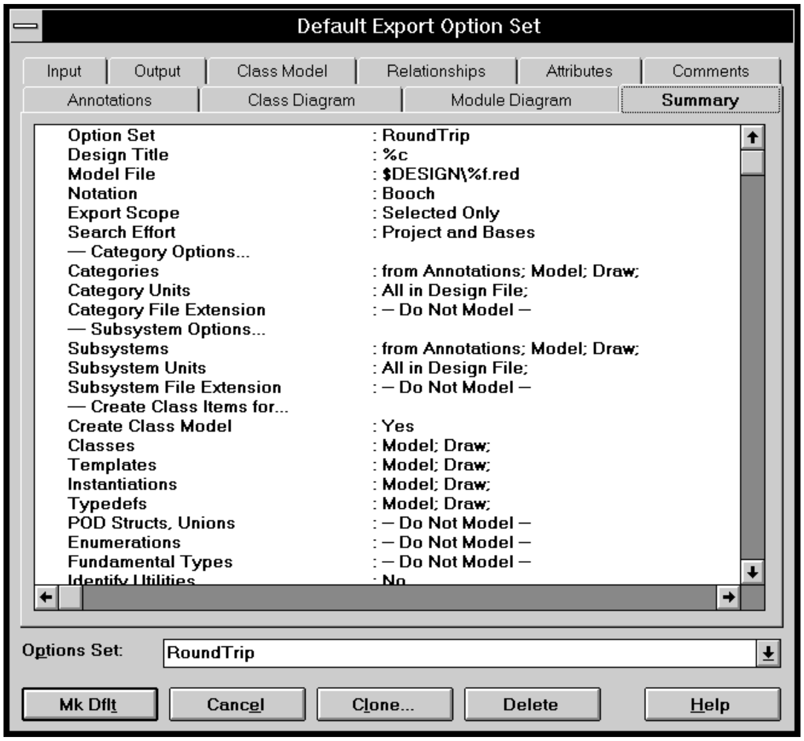
### Draw Module -> Subsystem Visibility Relationships

If this check box is enabled, instances of Module to Subsystem visibility relationships are shown in generated module diagrams.

### Draw Subsystem -> Module Visibility Relationships

If this check box is enabled, instances of Subsystem to Module visibility relationships are shown in generated module diagrams.

## Summary of Export Options



| | |
|---|---|
| Option Set | : RoundTrip |
| Design Title | : %c |
| Model File | : $DESIGN\%f.red |
| Notation | : Booch |
| Export Scope | : Selected Only |
| Search Effort | : Project and Bases |
| — Category Options... | |
| Categories | : from Annotations; Model; Draw; |
| Category Units | : All in Design File; |
| Category File Extension | : — Do Not Model — |
| — Subsystem Options... | |
| Subsystems | : from Annotations; Model; Draw; |
| Subsystem Units | : All in Design File; |
| Subsystem File Extension | : — Do Not Model — |
| — Create Class Items for... | |
| Create Class Model | : Yes |
| Classes | : Model; Draw; |
| Templates | : Model; Draw; |
| Instantiations | : Model; Draw; |
| Typedefs | : Model; Draw; |
| POD Structs, Unions | : — Do Not Model — |
| Enumerations | : — Do Not Model — |
| Fundamental Types | : — Do Not Model — |
| Identify Utilities | : No |

Options Set: RoundTrip

*Figure 51 Summary of Export Options*

The summary of export options provided by this dialog tab is identical to the project’s export options component.

## The Export Options

The available export options and their purpose are defined below. They are listed in the order in which they appear on the Summary tab of the Export Option Set dialog box.

***Note:*** *Some of the field names are quite long so, for clarity, the names may be truncated by an ellipsis (...).*

Double-clicking on an export option listed on the Summary tab takes you to the tab and field where the option is set.

- The Option Set option identifies the name of the option set.

- The Design Title option defines the identifying text string that is placed on each generated diagram.
- The Model File option defines the path for the generated model file. The option is set by the field on the tab.
- The Notation option determines whether generated diagrams utilize UML, COM, OMT, or Booch notation.
- The Export Scope option determines the closure for type definitions.
- The Search Effort option determines the search breadth for referenced type definitions.
- The Categories option specifies the basis on which classes are assigned to categories (logical packages), and whether categories (logical packages) appear in generated class diagrams. This option is enabled by the Create Class Model check box and has three settings on the **Output** tab:
  - The first (from) is set by the Assign Class To radio buttons, but is enabled by the Create Categories check box
  - The second (Model) is automatically set when the Create Class Model and Create Categories check boxes are checked
  - The third (Draw) is set by the Create Class Diagrams check box
- The Category Units option determines how categories (logical packages) are mapped to controlled units. This option is enabled by the Create Class Model check box and is set by the (Class) Controlled Unit Policy option buttons on the **Output** tab.

  When one of the two Create option buttons is selected, it enables the Override check box, which allows you to overwrite existing model-file extensions with the value for Category File Extension.
- The Category File Extension option specifies the file name extension used to generate model files representing categories (logical packages) that are controlled units. This option is enabled by all but the first (All) Controlled Unit Policy option buttons and is set by the Category Unit Extension text box on the **Output** tab.
- The Subsystem option determines the basis on which modules are assigned to subsystems (*component packages)*. This option is enabled when both Create Class Model and Create Module Diagrams check boxes are checked on the **Output** tab. This option has three settings:
  - The first (from) is set by the Assign Module To option buttons

  - The second (Model) is automatically set when the Create Class Model and Create Module Diagrams check boxes are checked
  - The third (Draw) is set by the Create Module Diagrams check box
- The Subsystem Units option determines how subsystems (component packages) are mapped to controlled units.

  When one of the two Create option buttons is selected, it enables the Override check box, which allows you to overwrite existing model-file extensions with the value for Subsystem File Extension.
- The Subsystem File Extension option specifies the file name extension used to generate model files representing subsystems (component packages) that are controlled units.
- The Create Class Model option determines whether classes are generated in the model, whether categories and subsystems (logical and component packages) are created, and whether class or module diagrams are generated.
- The Classes option determines the kinds of classes that are generated in the model, whether comments associated with these classes are captured in their specification's documentation field, and whether these classes appear in class diagrams.
- The Templates option determines what kinds of templates are represented as classes in the generated model, including associated comments, visibility, and class diagrams.
- The Instantiations option determines whether template instantiations are represented as classes in the generated model, and whether these classes appear in class diagrams.
- The Typedefs option determines what kinds of typedefs are represented as classes in the generated model, including associated comments, visibility, and class diagrams.
- The POD Structs, Unions option determines what kinds of Structs and Unions are represented as classes in the generated model, including associated comments, visibility, and class diagrams.
- The Enumerations option determines what kind of enumerations will be represented as classes in the generated model, whether comments associated with these enumerations are captured in their class specification's documentation field, and whether these classes will appear in class diagrams.

- The Fundamental Types option determines whether fundamental types are represented as classes in the generated model and in class diagrams.
- The Identify Utilities option determines whether classes, templates, or template instantiations that have no non-static members are modeled as class utilities, parameterized class utilities, or instantiated class utilities.

  If the option is disabled and there are no annotations, these language elements are modeled as normal, parameterized, or instantiated classes, respectively.
- The Object Methods option determines whether non-static methods are represented as operations in the generated model, including comments associated with the declarations of these methods.
- The Static Methods option determines whether static methods are represented as operations in the generated model, including comments associated with the declarations of these methods.
- The Method Body Comments option determines whether comments associated with the definitions of these methods are captured in their operation specification's documentation field (appended to comments captured from the declarations of these methods).
- The Class Object Data option determines whether non-static data members of class type are represented as attributes in the generated model.
- The Class Static Data option determines whether static data members of class type are represented as attributes in the generated model.
- The Non-Class Object Data option determines whether non-static data members of non-class type are represented as attributes in the generated model.
- The Non-Class Static Data option determines whether static data members of non-class type are represented as attributes in the generated model.
- The Unmodeled Object Data option determines whether non-static data members of unmodeled type are represented as attributes in the generated model.
- The Unmodeled Static Data option determines whether static data members of unmodeled type are represented as attributes in the generated model.

- The Class Object Has option determines whether to create *has* relationships or associations to the data types of non-static data members of class type.
- The Class Static Has option determines whether to create has relationships or associations to the data types of static data members of class type.
- The Non-Class Object Has option determines whether to create has relationships or associations to the data types of non-static data members of non-class type.
- The Non-Class Static Has option determines whether to create has relationships or associations to the data types of static data members of non-class type.
- The Object Method Uses option determines whether to create uses relationships to each type referenced in the declaration of non-static methods, and whether such uses relationships appear on class diagrams.
- The Static Method Uses option determines whether to create uses relationships to each type referenced in the declaration of static methods, and whether such uses relationships appear on class diagrams.
- The Instance Arg Uses option determines whether to create uses relationships to each type referenced in the declaration of type-valued instantiation arguments, whether such uses relationships appear on class diagrams, and whether these relationships are labeled with the name of their instantiation argument.
- The Template Param Uses option determines whether to create uses relationships to each type referenced in the declaration of non-type-valued template parameters, whether such uses relationships appear on class diagrams, and whether these relationships are labeled with the name of their parameter.
- The Typedef Uses option determines whether to create uses relationships to each type referenced in the declaration of typedefs.
- The Inherits option determines whether to create generalize relationships from classes to their base classes, and whether such relationships appear on class diagrams.
- The Instantiate option determines whether to create instantiates relationships from each template to its instantiation, and whether such instantiates relationships appear on class diagrams.

- The Friendship Required option determines whether to create uses relationships from classes to classes declared as friends, and whether such uses relationships appear on class diagrams.
- The Default Cardinality From and Default Cardinality To options specify the cardinality assigned to has or association relationships representing pointers.
- The Default Protocol option specifies the function protocol of each operation included in the generated model.
- The Class Diagram Name option specifies the name of the class diagram generated for each logistical package.
- The Modules option specifies the basis on which modules are assigned to subsystems (component packages), and whether subsystems (component packages) appear in generated module diagrams.
- The Module Diagram Name option specifies the name of the module diagram generated for each subsystem (component package).
- The Model Type Relationships option determines whether to create visibility relationships for type relationships, and how such relationships are labeled.
- The Model Direct #includes option determines whether to create visibility relationships for direct `#include` references, and how such relationships are labeled.
- The Model Indirect #includes option determines whether to create visibility relationships for indirect `#include` references, and how such relationships are labeled.
- The Model Unused #includes option determines whether to create visibility relationships for unused `#include` references, and how such relationships are labeled.
- The Model Contextual #includes option determines whether to create visibility relationships for `#include` references needed from context, and how such relationships are labeled.
- The Model Type-3 #includes option determines whether to create visibility relationships for `#include` references to type-3 headers, and how such relationships are labeled.
- The Module-Subsystem Rels option determines whether to create module-to-subsystem and subsystem-to-module visibility relationships.

- The Model Module Annotations option determines whether module annotations are captured in the documentation field of generated module specifications.
- The Model Class Annotations option determines what class annotations are captured in the documentation field of generated class specifications.
- The Model Operator Annotations option determines what operation annotations are captured in the documentation field of generated operation specifications.
- The Model Member Annotations option determine whether data member annotations are captured in the documentation field of generated attribute specifications.
- The Model Protected Regions option, which determines whether to ignore declarations found in protected regions.
- The Model Generated Declarations options, which determine whether to ignore generated declarations.
- The Version option cannot be set by the user.

In the interest of information density, several abbreviations are used in presenting the settings of the previously listed export options:

- **Modeled**—A specification for the component is included in the generated model.
- **Drawn**—An icon representing the component appears on the appropriate class or module diagram.
- **Name**—The icon representing the relationship is labeled.
- **Comments before**—Comments immediately preceding the declaration or definition are included in the documentation field of the component's specification.
- **Comments after**—In a class export option, indicates comments immediately following the declaration's left brace are included in the documentation field of the class's specification.
- **Comments after**—In a non-class export option, indicates comments following the declaration's (or definition's) right brace or semicolon are included in the documentation field of the component's specification.

# *Chapter 6*

# *Analyzer Project File*

The project files include settings for Visual C++ style macros and container classes from the MFC library. If you are an advanced user and you are planning to use other macro settings or different container classes than those provided, you might want to modify the project files. The help topics titled Macro File Parameters, Container Class Specifications, and Inline Annotations offer a detailed description of these project file settings.

The Analyzer project file is a text file with numerous project parameters specified one per line. The first characters of each line identify the parameters being specified on the rest of the line. These first characters are either of the form “#xxx>” or a simple letter code preceded by a “+” or “–”. The project file must start with the “#v001>” command, but there are a few order-dependencies between the parameter specifications. It is always safe to add new commands at the end of the project file.

## Macro File Parameters

The parameter that supports a macro file is:

`+J`<macro file name>

The language recognized by the Rational Ros eC++ Analyzer has been extended to include uniquely recognizable constructs for expressing Wizard information and its structures unambiguously within C++ source code to support the macros used by Visual C++. The Wizard macros generate this extended syntax. The macros are used only in reverse-engineering user code and, hence, are seen only by the Analyzer.

The language extensions are called inline annotations. The **+J** parameter identifies to the Analyzer the file that contains the Visual C++ macros. This file is implicitly included in every compilation. Without this project entry, none of the Visual C++ macros would be captured for RTE. The supplied **MSVC42.pjt, MSVC42b.pjt**, and **MSVC50.pjt** projects have this **+J** parameter pointing to **$ROSECPPHOME\projects\msvc.h**.

## Container Class Specification

If you use container classes other than the ones provided for MFC as defined in this release, you might want to change the container class specifications. The new parameter to support container classes is:

**+ct**<container class specification>

Un-annotated data members of a class are modeled as associations if the type of the data member is a type modeled as a class. In many cases, this technique generates a model that correctly captures the intended association between two classes. For example:

```
class Label {
   CString m_sName;
};
```

is modeled as a by-value aggregation between the class **Label** and the class **CString**. In more complex cases, however, this approach generates a poor model for the C++ code. For example, consider

```
class Message {
   CList<CString, int> To;
   CString From;
   CString Subject;
   CString Text;
};
```

Here, **To** would be modeled as a 1-1 by-value aggregate relation between **Message** and the instantiated class **CList<CString,int>**, but probably should be modeled as an unbounded aggregation of **CStrings** within **Message** since, in MFC, **CList** is a container class that implements a list of objects.

To model the use of container classes correctly, the Analyzer must be told what the container classes are and for a given instance, what the target class is and what cardinalities may be implied. This is the purpose of the **+ct** project parameter.

## +ct Project Parameter

The `+ct` project parameter has the form:

```
+ct<container class name> = <target class> [[<key type1>,...]]
{<card-from> -> <card-to> <flags>} [<imp>]
```

The left half of the equation represents the C++ name for a type. The rest of the `+ct` project parameter specifies how a given use of the container class is to be mapped to an association in a Rational Rose model.

**<container class name>** is the C++ name (fully qualified as needed) for a type, usually a template class, a class or a typedef naming an instantiation or class. When the <container class name> is found in the declaration of a class data member, the <container class name> and any template arguments, if present, will become the ContainerClass property for the role item (or has relationship) in the model that represents the data member. The ContainerClass property is used to regenerate the data member the next time Rational Rose generates source.

**<target class>** is the name of the class to be associated with the class that contains the data declaration. If the collection is indexed or qualified, the types of the keys should be listed in square brackets following the <target class>.

Within the specification of <target class>, special symbols of the form `$n` (for `n` = 1,...,9) may be used to reference the template arguments for a specific instance of the <container class name>. When the model is constructed, each `$n` will be replaced by the text of the $n^{th}$ template argument. The `$n` symbols can be used with a typedef if the typedef ultimately references an instantiation. The symbol `$0` is replaced by the first of the entered instances of the container class in the declarations.

**<card-to>** and **<card-from>** specify the cardinality of the relationship on both ends using any of the cardinality specifications allowed by Rational Rose (including a blank or unspecified cardinality).

Within the specification of <card-to> and <card-from>, special symbols of the form `$n` (for `n` = 1,...,9) may be used to reference the template arguments for a specific instance of the <container class name>. When the model is constructed, each `$n` will be replaced by the text of the $n^{th}$ template argument. The `$n` symbols can be used with a typedef if the typedef ultimately references an instantiation. The symbol `$0` is replaced by the first of the entered instances of the container class in the declarations.

**<flags>** is a sequence of one-letter codes that provide supplemental information about the relationship:

***Table 32 Flag Codes***

| Code | Meaning |
|---|---|
| R | Containment is by reference |
| V | Containment is by value |
| U | Containment is unspecified |
| F | Relationship is forward-reference only (definition of target class is not needed for this declaration) |
| G | Relationship is an aggregate |
| O | Relationship is ordered |
| D | Relationship is derived |

**<imp>** specifies what methods are to be generated to access the data. Its syntax is:

[P(<set>)] [M |V |A |T|F ] [BR|BV] [C |N ] [[I ] [K ] [-] G<name>] [[J ] [R ] [-] S<name>][br|bv ] [c |n ] [[i ] [k ] [-] g<name> <exp>] [[j ] [r ] [-] s<name> <exp>] q<name>

The elements of this specification string indicate how the project properties for the generated relationship are to be set for this collection class.

***Table 33 IMP Syntax***

| Code | Meaning | CG Properties affected |
|---|---|---|
| P<set> | Use the pre-defined property set identified by the string <set>. If this is not specified, then the default property set is used for the relationship. The remaining codes may be used to override this set or the default property set. If the code for a specific property is not present, then the value of that property forms the specified property set is used. | |
| M | Declare all functions as non-virtual Member functions. | GetSetKinds |
| V | Declare all functions virtual. | GetSetKinds |
| A | Declare all functions abstract. | GetSetKinds |
| T | Declare all functions static. | GetSetKinds |
| F | Declare all functions friend. | GetSetKinds |
| BR & BV | Container values are passed By Reference or By Value | GetSetByReference |
| C | The value/results are const. | GetResultIsConst<br>QualifiedGetResultIsConst |
| N | The value/result are not const. | GetResultIsConst<br>QualifiedGetResultIsConst |
| I | Declare function inline. | InlineGet<br>InlineQualifiedGet |
| K | Function is const. | GetIsConst<br>QualifiedGetIsConst |

| Code | Meaning | CG Properties affected |
|---|---|---|
| G<name> | Generate a get function for retrieving the entire container. In the absence of other codes, the function is not in-lined and not declared const.<br><name> following a **G** or **g** code is an optional name for the function enclosed in parentheses. If specified, the name becomes the value of the SetName, GetName, QualifiedSetName, and QualifiedSetName properties, respectively, for the relationship, The name can use the special symbols allowed in these project properties as well as the **`$n`** symbols. The default name is **`get_$target`**. | GenerateQualifiedGetOperation<br>QualifiedGetName |
| <exe> | <exe> following an **g** code is a C++ expression enclosed in braces that specifies how to get a value into or out of the collection. In this expression **`$data`** represents the data member that is the collection, and **`$keyn`** (for **`n`** in 1..9) represents the key values that identify the element of the collection to be set or retrieved. **`$key`** is a list of the key names separated by commas (,). **`$value`** represents the value to be set into or retrieved from the collection. The use of **`$value`** is optional in the specification of the <exp> for retrieval. If it is not present, it is assumed that the <exp> evaluates directly to the desired value.<br><exp> can be omitted. The default value <exp> for **g** is {**`$data.get ($keys, $value)`**}. | |
| J | | InlineSet<br>InlineQualifiedSet |
| R | A set function returns a value. | SetReturnsValue<br>QualifiedSetReturnsValue |

| Code | Meaning | CG Properties affected |
|---|---|---|
| S<name> | Generate a qualified Set function for storing a single item into the container. In the absence of other codes, the function is not in-lined and does not return a value. <name> following an **S** or **s** code is an optional name for the function enclosed in parentheses. If specified, the name becomes the value of the SetName, GetName, QualifiedSetName, and QualifiedSetName properties, respectively, for the relationship. The name can use the special symbols allowed in these project properties as well as the **$n** symbols defined above. The default name is **set_$target**. | GenerateQualifiedSetOperation<br>QualifiedSetName |
| <exp> | <exp> following an **s** code is a C++ expression enclosed in braces that specifies how to get a value into or out of the collection. In this expression **$data** represents the data member that is the collection, and **$keyn** (for **n** in 1..9) represents the key values that identify the element of the collection to be set or retrieved. **$keys** is a list of the key names separated by commas (,). **$value** represents the value to be set into or retrieved from the collection. The use of **$value** is optional in the specification of the <exp> for retrieval. If it is not preset, it is assumed that the <exp> evaluates directly to the desired value. Either <exp> can be omitted. The default <exp> for s is **{$data.set ($keys,$value)}.** | |
| Q<name> | a qualified container type. | QualifiedContainer |

## +ct Project Parameter Example

The +ct project parameter for CList might look like this:

```
+ctCList=$1[int]{1->nVGN} M BR C IGK -S
igk{$data.GetAt($data.FindIndex($key1))}
js{$data.SetAt($data.FindIndex($key1),$value)}
```

For the other MFC container classes, the +ct project parameters are as follows:

```
+ctCArray= $1[int]{1->nVGN} M BR C IGK -S
igk{$data.GetAt($key1)}js{$data.SetAt($key1,$value)}
```

```
+ctCMap= $3[$1] {1->nVGN} M BR C IGK -S ig{$data[$key1]}
js{$data.SetAt($key1,$value)}
```

```
+ctCTypedPtrList= $2 {1->nRGN} M BR C IGK -S -g -s
```

```
+ctCTypedPtrArray=$2[int]{1->nRGN} M BV C -G -S ig{$data[$key1]}
js{$data[$key1]=$value}
```

```
+ctCTypedPtrMap= $3[$2] {1->nRGN} M BV C -G -S
ig{$data[$key1]}js{$data[$key1]=$value}
```

## Inline Annotations

The language recognized by the Rational Ros eC++ Analyzer has been extended to include uniquely recognizable constructs for expressing MFC information and structures unambiguously within C++ source code. Substitute definitions for all of the MFC macros used to build these constructs have been created to generate this extended syntax. The language extensions are called *inline annotations*.

Specifically, the Rational Rose C++ Analyzer has been changed as follows:

***Table 34 Analyzer Changes***

| | |
|---|---|
| ✓ | The data file generated by the Analyzer been augmented to represent inline annotations. |
| ✓ | The Analyzer's preprocessor and parser have been modified to recognize inline annotations and the macro definitions that use them. |
| ✓ | The Analyzer's Rational Rose model builder has been modified to transfer the inline annotations from the data file to the Rational Rose model. |
| ✓ | The relevant Wizard macros have been identified and redefined to generate inline annotations. |

## Inline Annotation Syntax

***Table 35 Inline Annotation Syntax***

| | |
|---|---|
| <inline annotation> | ::= **%%[** <token list> **]** \| **%%{** {<inline annotation>} **}** |
| <token list> | ::= <token> \| <token> <token list> |
| <token> | ::= **%%(** <token list> **)** \| **%%()** \| <any C++ language token> |

The **%%**-token constructs are called inline annotations because, like the existing code generator annotations, they convey model information through the source, but, in addition, they can appear inline such as in a macro definition. They are recognized by the Analyzer's C++ scanner/parser and data file entries are generated for them. The parser determines where these inline annotations are permitted and to which language entity they are attached. Currently inline annotations are permitted wherever standard annotations are allowed and also as type specifiers in a declaration.

## Simple Annotations

The **%%[** token introduces a simple inline annotation, whose arguments are all individual tokens bracketed by a matching **]**. The syntax for an inline annotation is Lisp-like in that an annotation is simply a list of tokens, the first of which identifies the type of annotation and the remaining tokens are the arguments to the annotation. Parsing an annotation requires no knowledge of its semantics.

## Hierarchical Annotations

The **%%{** token introduces a list of inline annotations. The list is terminated by the next matching **}**. The list represents a hierarchy of inline annotations. The first inline annotation defines the root node of the hierarchy and the remaining inline annotations define its immediate children.

An empty annotation list is a valid construct. It generates no nodes in the data file, but counts as an inline annotation when the scanner decides if a macro definition is an overriding macro definition.

## Annotation Literals

The `%%(` token begins an annotation literal and causes all the tokens up to the next matching `)` to be collected into a single token similar to the way quotation marks are used to form characters into a string literal. In an annotation literal, however, the characters are tokenized and macro argument substitution is performed, but macro names are not expanded. An annotation literal thus behaves much like an extended form of the ## concatenation operator.

Annotation literals are allowed only in macro definitions and then only as an argument to a simple inline annotation.

The image of an annotation literal is the concatenated images of its contained tokens. The spacing of tokens in the source file is lost during preprocessing, so to preserve the identity of tokens within the annotation literal, a single blank is inserted between most token images when forming the image of the annotation literal. No space is inserted before the tokens `,`, `.`, `;`, `:`, `::`, `]` or `)` and no space is inserted after the tokens `.`, `::`, `(` or `[`. Within an annotation literal, the `#` and `##` tokens can be used to modify the insertion of spaces: `##` inhibits insertion of spaces and `#` forces a space to be inserted.

The empty annotation literal `%%()` has special meaning. It is not ignored like empty hierarchical annotations, `%%{}`. Instead, `%%()` generates an annotation literal that is the image of the macro call in which it is embedded. The image of the call is constructed by concatenating the tokens of the macro call using the spacing rules defined in the previous paragraph. This special annotation literal is the principle mechanism by which macro calls are captured from the source and copied into the model.

## Inline Annotation Semantics

The parsing of inline annotations and the constructing of the corresponding data file entries is based purely on the annotation syntax. The semantics of the various inline annotations is determined only by the export operation. The following inline annotations are currently implemented. Others may be added in future releases.

- The prop Annotation
- The map Annotation
- The open Annotation
- The close Annotation

- Wizard Comments
- Overriding Macro Definitions
- Implicit Include Option
- An Example of Inline Annotations

## The prop Annotation

The **prop** annotation causes project properties to be set in the exported model. The general form of this annotation is:

```
%%[prop [<class name> | <member name> | * ]
{<property specification> <property value>}]
```

Any code generation property presented to the Analyzer via the **prop** inline annotation will be attached to the indicated model element, but the property will not show up in the specification dialog within Rational Rose for the item unless it is also defined in the property file associated with the model.

**The first argument to prop** specifies the model element whose project properties are to be set.

- If the first argument is an annotation literal, then it is interpreted as a <class name>. The <class name> may be a C++ qualified name, which is evaluated in the context of the inline annotation according to normal C++ name lookup rules.
- The ***** token as the first argument indicates that the project property is to be set on the class that encloses the inline annotation; or, if the inline annotation is outside a class, on the module that encloses the inline annotation.
- If the first argument is neither an annotation literal nor an *****, it is assumed to be a <member name>. The <member name> is the simple name of the class member. It can be the name of a nested class, or an Operation, Attribute, or Role of the class. The class to which the member belongs is determined from the context of the inline annotation, which may be a class definition, a class member function definition, or a containing inline annotation that specifies a class (such as the **map** annotation described below).
- If the <member name> is specified as a string literal, and a member of that name does not exist, a Class Attribute of that name is created. (If the member name were not a string literal and the member did not exist, a warning message would be generated

during the export operation.) This feature is used in the MFC macros to construct a pseudo class member to represent OLE properties implemented via Get/Set functions rather than as data members.

- If the optional first argument is omitted (i.e. if there are an even number of tokens in the argument list), then it is assumed that the inline annotation is attached to the declaration or definition of the class member whose project property is to be set.
- If <member name> is overloaded, the properties are set on all of the overloaded members.

**<property specification>** is a single token of the form

[<tool name>::] [<type>]<property name>

A <property specification> must be an annotation literal if it contains a <tool name> and it may be an annotation literal even if it doesn't.

The default <tool name> is "`cg`".

The **type** of the code generation attribute is specified by a single, lowercase letter. If the first character of the <property specification>, following :: is present, and it is not one listed in the following table, the property is assumed to be a String property.

***Table 36 Character Codes***

| <type> Code | Project Property Type | Derivation of Property Value from <property value> |
|---|---|---|
| a | | The <property name> is ignored for this <type>. The inline annotation must associate with a class attribute. The built-in type property of the associated class attribute is set by this inline annotation. <property value> is the name of the member function of the class that contains this annotation. If it exists, it is the first argument type is the value assigned to the class attribute's type. |
| b | Boolean | The property is set to "True" if the <property value> image begins with 'T' or 't'; if it begins with 'W' or 'w' it is set "True" or "False" based on the location of the `prop` annotation inside or outside Wizard comments; otherwise it is set to "False". |
| c | Char | The first character of the <property value> image becomes the value of the project property. |

| <type> Code | Project Property Type | Derivation of Property Value from <property value> |
|---|---|---|
| e | Enum | The <property value> image consists of an attribute set name and an integer value defined in that set written as a single C++ identifier, such as **`WizardTypeSet3`** or **`OleFactorySet21`**. |
| l | Text | The <property value> image is appended to the existing value of the project property and then a new line sequence is appended. A multi-line CG Text property is thus constructed, one line at a time. |
| m | String | The <property value> image is appended to the existing value of the project property and then a space is appended. A multi-segment CG String property is thus constructed one segment at a time. |
| n | Int | The <property value> image is parsed (via **`atoi()`**) as an integer literal and the resulting integer value becomes the value of the project property. |
| r | | The <property name> is ignored for this <type>. The inline annotation must associate with a class attribute. The built-in type property of the associated class attribute is set by this inline annotation. <property value> is the name of a member function of the class that contains this annotation. If it exists, its return type is the value assigned to the class attribute's type. |
| s | String | The <property value> image is copied verbatim to the project property. |
| t | Text | The <property value> image is copied verbatim to the project property. |

<property name> can be any string of characters, but it usually conforms to the syntax for a C++ identifier.

A special naming convention is used in the implementation of the code generation properties that capture the macros that appear between Wizard comments. Using this convention one <property name> string actually names one of two possible code generation properties to be set. If the inline annotation appears between Wizard comments, the first property is set; if the inline annotation is not between Wizard comments, the second property is set. The convention is to delimit the second property name by "__AFX__" and "__." Any text that precedes the "__AFX__" or follows the "__" is common to both property names. To form the first property name (when the inline annotation is between

Wizard comments), the sequence beginning with "__AFX__" and ending with "__" is replaced by the name that appears in the Wizard comment (such as AFX_MSG_MAP or AFX_DISPATCH). To form the second property name, the "__AFX__" and "__" sequences are simply elided.

For example "__AFX__MESSAGE_MAP___Entries" generates "AFX_MSG_MAP_Entries" when it appears between //{{AFX_MSG_MAP and //}}AFX_MSG_MAP comments and "MESSAGE_MAP_Entries" when not between Wizard comments.

<property value> strings have special interpretations:

***Table 37 Property Value Strings***

| **Special <property value> Token** | **Value stored in CG Property** |
|---|---|
| %%(#) | The source text between the braces of the definition that is associated with the inline annotation of which this <property value> is a part; e.g., the body of a function definition. |
| %%(::;) | The source text of the entire declaration that is associated with the inline annotation of which this <property value> is a part. |
| %%(,) | The source text of the macro call (including embedded comments) that generated the inline annotation of which this <property value> is a part. |

## The map Annotation

**`%%[map`** <class name> [<base class name>] **`]`**

The **`map`** annotation denotes an MFC map implementation, such as a message map or dispatch map. The <class name> argument of the **`map`** annotation specifies the class with which the map is associated. It must be an annotation literal and may be a qualified name. The entries in the map are represented by the **`prop`** annotations that are subordinate to the **`map`** annotation.

## The open Annotation

**`%%[open`** < property specification > <class> {<property specification> <property value>}**`]`**

**`%%[open`** < property specification > <class> **`this]`**

These **`open`** annotations mark the start of a Wizard section in the source. A **`close`** annotation marks the end of a Wizard section. Wizard sections do not nest.

Within a Wizard section, if the image of any Boolean <property value> begins with a **`'W'`** or **`'w',`** then the value actually set in the property is True. Outside of a Wizard section, such Boolean values actually set the property to False.

Also in a Wizard section, any <property name> containing the sequence "__**`AFX`**__" … "__" is modified by replacing the sequence with the <property name> from the first <property specification> of the **`open`** annotation. Outside a Wizard section, both the "__AFX__" and "__" segments are elided (but the segment in between is retained).

If the first form of the **`open`** annotation is used, the specified <property specification> <property value> pairs are applied to each declaration in the Wizard region opened by the annotation as if there were a **`prop`** annotation associated with the declaration with the same list of <property specification> <property value> pairs. Typically the prop annotation sets a property that indicates in which type of Wizard region the declaration appeared.

If the second form of the **`open`** annotation is used, "**`_Entries`**" is appended to the <property specification> and a code generation property with that name is attached to the class that contains the annotation. The value of that property is the source text for the entire declaration that the annotation is attached to. This is obviously not a general-purpose annotation argument. It is used to capture the **`AFX_DISP_ID`** Wizard region in a class definition.

## The close Annotation

```
%%[close]
```

## Wizard Comments

The Microsoft Class Wizard uses unique comments of the form **`//{{AFX_xxx`** and **`//}}AFX_xxx`** to bracket the member declarations and map entries that it understands and manipulates for the user. The Analyzer now recognizes these as well. When the Analyzer sees a comment beginning "**`//{{ xxx`**" it replaces it internally with "**`__OPEN_xxx`**" and continues scanning the source. This could lead to

Analysis errors were it not for the fact that for each possible `//{{AFX_xxx` comment there is a corresponding `__OPEN_AFX_xxx` macro defined. Each `__OPEN_xxx` macro introduces an `open` inline annotation into the token stream.

Similarly, each comment of the form "`//}} xxx`" is transformed internally to "`__CLOSE_xxx`" and for each `//}}AFX_xxx` comment there is a `__CLOSE_AFX_xxx` macro defined. The `__CLOSE_AFX_xxx` macros introduce a `close` inline annotation into the token stream.

Also, each comment of the form "`//{{ xxx }}`" is transformed internally to "`__NOTE_xxx`." The defined `__NOTE_xxx` macros currently generate no tokens in this release.

The export operation keeps track of which inline annotations appear between Wizard comments and which ones don't. If the image of a Boolean <property value> begins with a 'W' or 'w', then the value actually set in the property is True if the annotation is between Wizard comments and False if it is not.

## Overriding Macro Definitions

Any macro definition containing an inline annotation (simple or hierarchical) or annotation literal is considered an overriding macro definition. That is, an overriding macro definition is any macro definition that contains one of the tokens `%%[`, `%%{`, or `%%(`. It does not matter how the macro definition is presented to the Analyzer — it may be from the source text or from the project's Defined Symbols list or via the `+J` project parameters.

Normally, the "defined symbols" specified in a project follow the rules for a symbol specified in the -D option of `cpp` (the UNIX preprocessor) and provide only an initial definition for the symbol. If the symbol is defined in the source being analyzed, then the definition in the source takes precedence. If, however, by the above mechanism, the symbol is defined as an overriding macro definition, any definitions of the symbol in the source will be ignored by the analyzer's preprocessor.

We use overriding macro definitions to replace the normal expansion of VC++ macros with expansions that generate annotations, which will allow the Analyzer to preserve the use of the macro in the source so that its semantics can be transmitted to the Rational Rose model.

## Implicit Include Option

The Analyzer option **+J**<file name> causes <file name> to be implicitly declared (`#include`) into each source file analyzed by the Analyzer (unless the source file is, itself, a **+J** file). Currently there is no GUI support for adding such lines to an Analyzer project, they must be added using a text editor. Since the primary use of this feature is to introduce our substitute MFC macros when analyzing MFC source, there seems to be little need for GUI support.

### Example of Inline Annotations

As an example of how inline annotations are used to recognize MFC constructs, consider the ubiquitous MFC message map. The message map is declared in the class specification via the **DECLARE_MESSAGE_MAP()** macro and is defined in the **.CPP** file by a list of macro calls such as **ON_COMMAND()**, **ON_WM_LBUTTONDOWN()**, etc. There is one macro call per message map entry and they are all sandwiched between the macros **BEGIN_MESSAGE_MAP** and **END_MESSAGE_MAP**. Unique comments of the form **//{{AFX_xxx** and **//}}AFX_xxx** bracket for the Class Wizard both the member function declarations in the spec (**xxx = MSG**) and the message map entries in the body (**xxx = MSG_MAP**). To complicate things further, the ClassWizard does not understand all of the macros that can appear in a message map, so there are some map entries that appear between **BEGIN_MESSAGE_MAP** and **END_MESSAGE_MAP** but not within the Wizard brackets.

To regenerate the message map for a class, the Rational Rose C++ code generator needs to know which member functions are message handlers and what their message map entries should look like. It also needs to know if the message map entry is understood by the ClassWizard. This information is defined in MSVC-specific project properties:

***Table 38 Message Map Entries***

| **MVCS Property Name** | **Type** | **Item Found On** | **Comments** |
|---|---|---|---|
| GenerateMessage Map | Boolean | Class | Causes the code generator to generate a message map for the class. |
| GenerateMessage Group | Boolean | Class | Causes the code generator to generate a **`//{{AFX_MSG`** section in the class definition. |
| Type | Enum | Operation | When an Operation has the Type MessageHandler, the code generator places the declaration in the **`//{{AFX_MSG`** section of the class definition. |
| AFX_MSG_MAP_ Entries | Text | Class or Operation | On an Operation item, the property contains the image of the actual message map entries referencing that Operation and appearing within the **`//{{AFX_MSG_MAP`** section of the message map. On a Class item, the property contains the image of the actual message map entries referencing an Operation not defined in the class (presumably a member of some base class) and appearing within the **`//{{AFX_MSG_MAP`** section of the message map. |

To fill in these project properties, the redefined overriding MFC macros would look like this:

```
// First some macros to simplify defining the multitude of
  ON_xxx macros
#define __MessageHandler MSVCOperationsTypeSet2
#define __WIZMAP(Member, Map, Entry) \
%%[prop %%(Member) %%(MSVC::l__AFX__##Map##___Entries) Entry]
#define __MSGMAP(Member,Entry) %%{} __WIZMAP(Member,
  MESSAGE_MAP, Entry)

// Macros to record the Wizard regions for the message map
#define __OPEN_AFX_MSG(Class) \
%%[open %%(MSVC::lAFX_MSG) %%(Class) %%(MSVC::eType)
  __MessageHandler] \
%%[prop * %%(MSVC::bGenerateMessageGroup) True]
#define __CLOSE_AFX_MSG %%[close]
#define __OPEN_AFX_MSG_MAP(Class) %%[open AFX_MSG_MAP
  %%(Class)]
#define __CLOSE_AFX_MSG_MAP %%[close]

// Elide the code normally generated by afx_msg from the source
  the Analyzer sees.
#define afx_msg %%{}

// Declare and define the message map entries in the Analyzer's
  data file
#define DECLARE_MESSAGE_MAP() %%[prop *
  %%(MSVC::bGenerateMessageMap) True]
#define BEGIN_MESSAGE_MAP(theClass, baseClass) %%{ %%[map
  %%(theClass) %%(baseClass)]
#define END_MESSAGE_MAP() %%{}}

// Capture the message map entries and associate them with
  there Message Handlers
#define ON_COMMAND(id, memberFxn) __MSGMAP (memberFxn, %%())
#define ON_COMMAND_RANGE(id, idLast, memberFxn) __MSGMAP
  (memberFxn, %%())
#define ON_UPDATE_COMMAND_UI(id, memberFxn) __MSGMAP
  (memberFxn, %%())
#define ON_UPDATE_COMMAND_UI_RANGE(id, idLast, memberFxn)
  __MSGMAP (memberFxn, %%())
… more of the same…
#define ON_WM_CREATE() __MSGMAP (OnCreate, %%())
#define ON_WM_DESTROY() __MSGMAP (OnDestroy, %%())
#define ON_WM_MOVE() __MSGMAP (OnMove, %%())
… and so forth for all macros that can appear in a message map.
```

*Chapter 7*

# *Design Update*

The Rational Rose C++ Analyzer produces model files containing design information extracted from application source files. In round-trip engineering practice, these source files are generated by a Rational Rose Code Generator from a Rational Rose model. Clicking **File > Update** inserts the reverse-engineered information contained in a specified model file (the update source) into the currently open model (the update destination). The **Update** command updates the currently open model to reflect design changes introduced through source file modification. Since this operation is performed "in place," you should be certain that the model has been appropriately versioned before clicking **File > Update**.

The **Update** option is not active (is grayed out) if the update destination is not a monolithic file—is separated into controlled units represented by several model files—and any of the controlled units of the update destination are:

- Not loaded.
- Or not write-enabled.

By default, clicking **File > Update** expects the model file containing the update source to have a **`.red`** file name extension; the Analyzer generates model files with this extension when its RoundTrip export option set is selected.

Clicking **File > Update** preserves the following information in the current model:

- The documentation field in each logical package specification.
- The documentation field in each component package specification.
- The deployment diagram.

- All interaction diagrams.
- All state diagrams.

The Analyzer can optionally generate class and module diagrams and place them into the generated model file. Clicking **File > Update** inserts these diagrams into the model, replacing any existing diagrams with the same name. Clicking **File > Update** preserves all class and module diagrams in the model except any whose names match diagrams contained in the specified model file.

New components identified during reverse engineering are noted in the log and added to the currently open model; icons representing these new components are not automatically added to diagrams. Components that have been relocated are also noted in the log without modification to diagrams.

Classes present in a logical package in the currently open model but absent from the reverse engineered version of that logical package are noted in the log as legacy components, but are not automatically removed from the currently open model or diagrams. Components present in a component package in the currently open model but absent from the reverse engineered version of that component package are similarly noted in the log.

The following code generation properties are updated from the Reverse Engineered source code:

- FileName
- CopyrightNotice
- AdditionalIncludes
- OperationKind
- OperationIsConst
- ForwardReferenceOnly
- DataMemberVisibility
- ImplementationType
- Directory

# *Chapter 8*

# *C++ Round-Trip Engineering*

The tool support required by controlled iterative development is known as *round-trip engineering*. Each iteration begins with the use of Rational Rose to assess the current iteration model, capture new scenarios, and extend the design to achieve specific objectives. Rational Rose generates source code frameworks for each class in this extended model, preserving user-supplied definitions and auxiliary declarations from the previous iteration's source code. New method definitions are then added, and this extended implementation is validated against the iteration's objectives. In the course of these activities, intentional and unintentional design changes are frequently introduced.

During acceptance, the C++ Analyzer is used to reverse engineer the extended implementation and annotate all newly-added operation definitions and file-scope declarations, producing a new as-built model that can be examined with Rational Rose. The Rational Rose C++ Visual Differencing tool is used to reveal design changes by comparing the reverse engineered model to the extended iteration model.

After the implementation is accepted, the Analyzer is used to produce a round-trip model that is used to update the iteration model to reflect the design changes instituted in the source code.

The overall process is illustrated below:

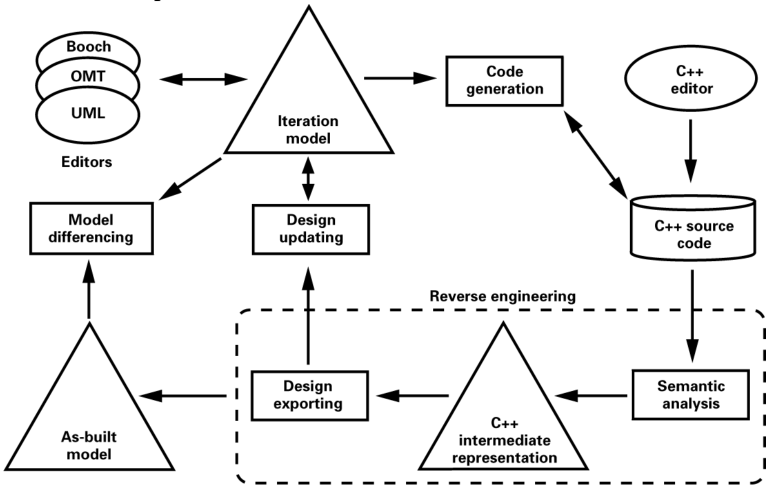


***Figure 52 Round-Trip Engineering Process***

## Round-Trip Engineering—Step By Step

All of the facilities required to practice round-trip engineering have already been described in earlier chapters of this guide. To illustrate their use, an example will be pursued through one iteration.

Consider the Tracker/Main class diagram from the iteration's initial model:

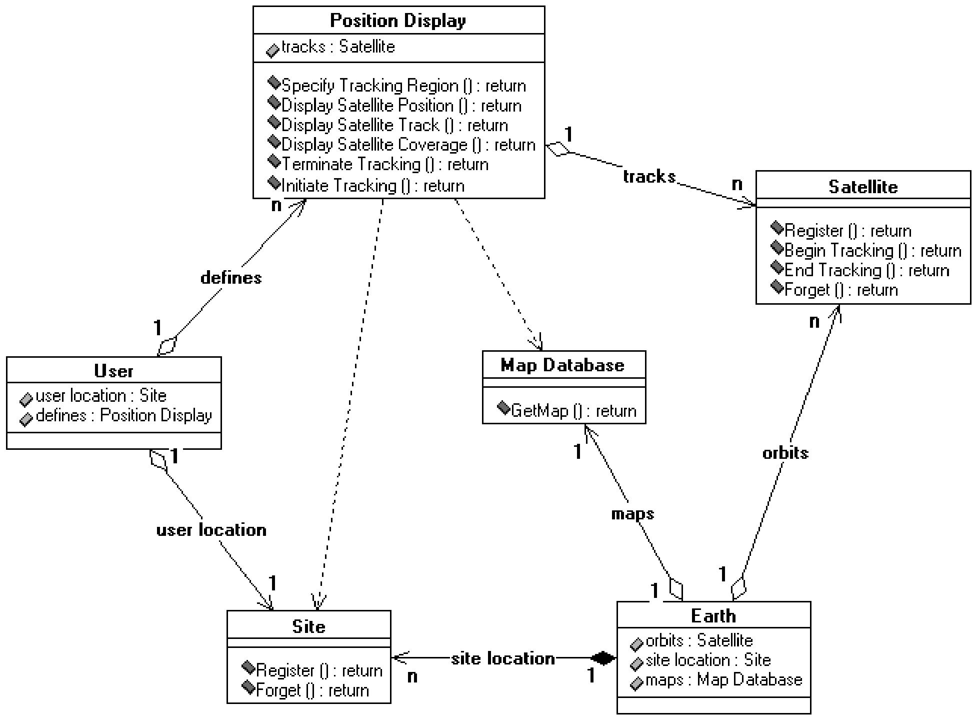


***Figure 53 Tracker Class Diagram***

At the beginning of the iteration, generate code by selecting these icons and clicking **Tools > C++ > Code Generation**. In the course of testing the iteration, it is discovered that a Satellite must be able to report position on demand, and that a Position Display needs a Data Link.

The code for module `satllite.h`, to which class `Satellite` is assigned follows. The code added to define operation ObtainPosition is underlined:

```
//## begin module.cm preserve=no
//   %X% %Q% %Z% %W%
//## end module.cm

//## begin module.cp preserve=no
//## end module.cp
```

```
//## Module: Satellite; Pseudo Package specification
//## Component Package: tracker
//## Source file: c:\rose\demo\ootrack\tracker\satllite.h
#ifndef Satllite_h
#define Satllite_h 1

//## begin module.additionalIncludes preserve=no
//## end module.additionalIncludes

//## begin module.includes preserve=yes
//## end module.includes

// Result
#include "utilties\Result.h"
// String
#include "utilties\String.h"
// KeplerianParameters
#include "utilties\KplrnPrm.h"

//## begin module.additionalDeclarations preserve=yes
//## end module.additionalDeclarations

//## Class: Satellite
//## Unit: c:\rose\demo\ootrack\tracker.cat
//## Logical package: tracker
//## Component package: tracker
//## Concurrency: Sequential
//## Persistence: Transient
//## Cardinality/Multiplicity: n
//## Dependency <Result>: public: Result { -> }
//## Dependency <String>: public: String { -> }
//## Dependency <KeplerianParameters>: public:
KeplerianParameters { -> }

class Satellite
{
  //## begin Satellite.initialDeclarations preserve=yes
  //## end Satellite.initialDeclarations

  public:
    //## Constructors (generated)
      Satellite();
      Satellite(const Satellite &right);

    //## Destructor (generated)
      ~Satellite();
//## Assignment Operation (generated)
      const Satellite & operator=(const Satellite &right);

    //## Equality Operations (generated)
      int operator==(const Satellite &right) const;
      int operator!=(const Satellite &right) const;

    //## Other Operations (specified)
      //## Concurrency: Sequential
      Result Register(String Name, KeplerianParameters Params);

      //## Concurrency: Sequential
      Result BeginTracking();

      //## Concurrency: Sequential
      Result EndTracking();

      //## Concurrency: Sequential
      Result Forget();

     //$$ Documentation
        //   report satellite position
        Result ObtainPosition();

    // Additional Public Declarations
      //## begin Satellite.public preserve=yes
      //## end Satellite.public

  protected:
    // Additional Protected Declarations
      //## begin Satellite.protected preserve=yes
      //## end Satellite.protected

private:
    // Additional Private Declarations
      //## begin Satellite.private preserve=yes
      //## end Satellite.private

  private:  //## implementation
    // Additional Implementation Declarations
      //## begin Satellite.implementation preserve=yes
      //## end Satellite.implementation
};
// Class Satellite
#endif
```

The code for module `pstndspl.h`, to which class `Position Display` is assigned follows. The code added to show its containment of a data link is underlined:

```
//## begin module.cm preserve=no
//   %X% %Q% %Z% %W%
//## end module.cm

//## begin module.cp preserve=no
//## end module.cp

//## Module: Position Display; Pseudo Package specification
//## Component package: tracker
//## Source file: c:\rose\demo\ootrack\tracker\pstndspl.h

#ifndef PstnDspl_h
#define PstnDspl_h 1

//## begin module.additionalIncludes preserve=no
//## end module.additionalIncludes

//## begin module.includes preserve=yes
//## end module.includes
// Result
#include "utilties\Result.h"
// Satellite
#include "tracker\Satllite.h"
// Map Database
#include "tracker\MapDtbse.h"
// Site
#include "tracker\Site.h"
// ReferenceList
#include "utilties\RfrncLst.h"

//## begin module.additionalDeclarations preserve=yes
//## end module.additionalDeclarations

class DataLink
//    provide data communications
{
public:
      int open();
      //   open link
      int close();
      //   close link
};
```

```
//## Class: Position Display
// This class manages a one-Window geographic display
//## Unit: c:\rose\demo\ootrack\tracker.cat
//## Logical package: tracker
//## Component package: tracker
//## Concurrency: Sequential
//## Persistence: Transient
//## Cardinality/Multiplicity: n
//## Dependency <Site>: public: Site { -> }
//## Dependency <Map_Database>: public: Map_Database { -> }
//## Dependency <Result>: public: Result { -> }
//## Dependency <ReferenceList>: public: ReferenceList { -> }

class Position_Display
{
  //## begin Position_Display.initialDeclarations preserve=yes
  //## end Position_Display.initialDeclarations

  public:
    //## Constructors (generated)
      Position_Display();
      Position_Display(const Position_Display &right);

    //## Destructor (generated)
      ~Position_Display();

    //## Assignment Operation (generated)
      const Position_Display & operator=(const Position_Display
&right);

    //## Equality Operations (generated)
      int operator==(const Position_Display &right) const;
      int operator!=(const Position_Display &right) const;

  //## Other Operations (specified)
      //## Documentation:
      //this operation specifies the region within which
      //the satellite is to be tracked
      //## Concurrency: Sequential
      Result SpecifyTrackingRegion();

      //## Documentation:
      //this operation shows the satelitte's current
      //       position
      //## Concurrency: Sequential
      Result DisplaySatellitePosition();
```

```
//## Documentation:
      //this operation shows the satellite’s ground
      //       track
      //## Concurrency: Sequential
      Result DisplaySatelliteTrack();

      //## Documentation:
      //this operation shows the Satellite’s current
      //ground coverage
      //## Concurrency: Sequential
      Result DisplaySatelliteCoverage();

      //## Documentation:
      //this operation terminates the display
      //## Concurrency: Sequential
      Result TerminateTracking();

      //## Documentation:
      //Begin tracking the specified satellite
      //## Concurrency: Sequential
      Result InitiateTracking();

    //## Get and Set Operations for Has Relationships
(generated)
      const ReferenceList get_tracks() const;
      void set_tracks(const ReferenceList value);

    // Additional Public Declarations
      //## begin Position_Display.public preserve=yes
      //## end Position_Display.public

  protected:
    // Additional Protected Declarations
      //## begin Position_Display.protected preserve=yes
      //## end Position_Display.protected

private:
    // Additional Private Declarations
      //## begin Position_Display.private preserve=yes
      //## end Position_Display.private

  private:  //## implementation
    // Data Members for Aggregate Relationships
      //## begin Position_Display::tracks.has preserve=no
```

```
public: Satellite {1 -> nRO}
      ReferenceList tracks;
      //## end Position_Display::tracks.has

      DataLink connects;

    // Additional Implementation Declarations
      //## begin Position_Display.implementation preserve=yes
      //## end Position_Display.implementation
};

// Class Position_Display

//## Get and Set Operations for Aggregate Relationships
(inline)
inline const ReferenceList Position_Display::get_tracks() const
{
  //## begin Position_Display::get_tracks%.get preserve=no
  return tracks;
  //## end Position_Display::get_tracks%.get
}

inline void Position_Display::set_tracks(const ReferenceList
value)
{
  //## begin Position_Display::set_tracks%.set preserve=no
  tracks = value;
  //## end Position_Display::set_tracks%.set
}

#endif
```

To assess these changes, the implementation is reverse engineered using the **DetailedAnalysis** export option set. The resulting model is loaded into Rational Rose by clicking **File > Open**. The generated class diagram is re-arranged by clicking **Tools > Layout**:

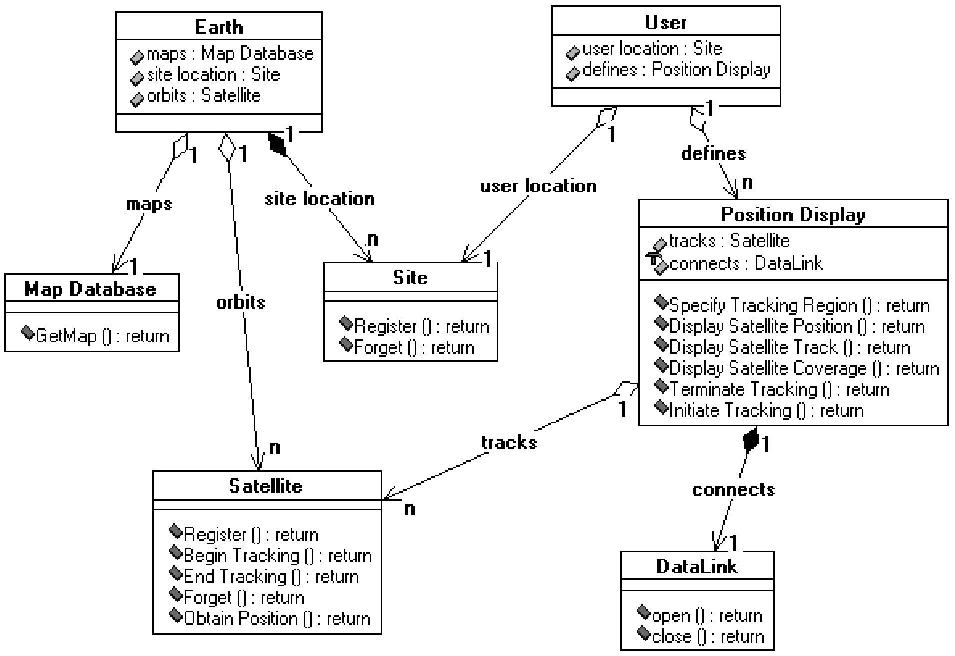


***Figure 54 Generated Class Diagram***

***Note:*** *The new ObtainPosition operation in class Satellite, and class DataLink, its operators, and the appropriate aggregate relationship from class Position Display to class DataLink were added. With the reverse engineered model, these changes are acceptable.*

To update the iteration model to reflect the changes instituted in the source code, the modified source code is reverse engineered using the **RoundTrip** export option set; since none of the source files changed after being reverse engineered to support assessment, semantic analysis proceeds quickly.

The iteration model is loaded into Rational Rose by clicking **File > Open**, and the reverse engineered model is updated by clicking **File > Update**. The Tracker/Main class diagram is displayed. Click **Query > Add Classes** to add an icon representing the new DataLink class. Right-clicking on the icon representing class Satellite updates its compartment to reflect its new ObtainPosition operation. The updated Tracker/Main class diagram follows:

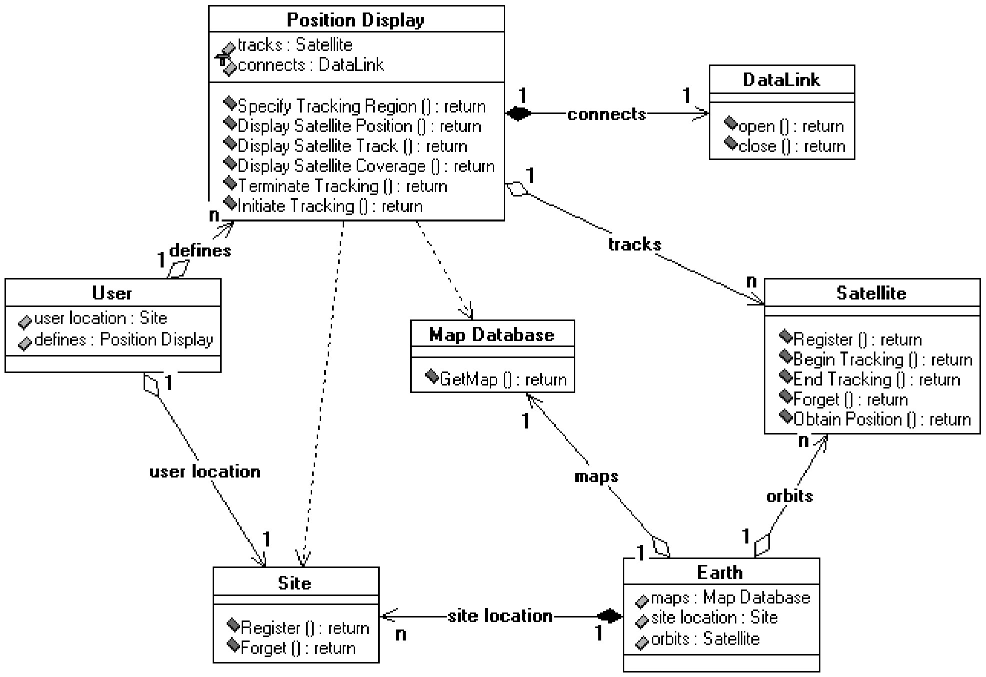


***Figure 55 Updated Class Diagram***

Double-clicking the **Data Link** icon displays the specification for its class.

Double-clicking the **Satellite** icon to display its specification, and then double-clicking on the ObtainPosition operation displays the specification for this new operation.

The iteration has been completed with the iteration model updated to match the implementation.

## Starting with Existing Source Code

If you begin round-trip engineering with an existing body of source code, the existing operation definitions and file-scope declarations within this source code must be annotated to preserve them through subsequent code generation. Click **Action > Code Cycle** to both analyze the source files and insert the required annotations while initially reverse engineering with the **DetailedAnalysis** export option set. Logical and Component Package assignments will be taken from the project category assignment and project subsystem assignment.

Rational Rose's default code generation properties automatically generate standard operations, such as constructors, destructors, equality operations, and get and set operations. Generating code from a model produced by reverse engineering existing source code will, by default, result in duplicate declarations for standard operations originally present in the source code. To avoid this, load the reverse engineered model into Rational Rose by clicking **File > Open**. Then click **Tools > Properties > Edit Properties** to create a code generation property set whose code generation properties disable generation of standard operations. Attach this property set to all model components whose standard operations were declared in the original source code. Setting the properties prevents duplicated operations in all subsequent iterations.

Before proceeding with a first iteration, generate code from the model, and reverse engineer this code using the **DetailedAnalysis** export option set. Use the Rational Rose model Integrator tool to verify that there are no unexpected variances between the first reverse engineered model and the second.

## Changing Between the UML, OMT, and Booch Notations

The Analyzer's Notation Export Option allows the generation of either UML, OMT, or Booch models. When practicing round-trip engineering, the setting of this Export Option should always match the notation you are using in Rational Rose to generate code and update your model. If you wish to change from one notation to the other, click**Tools > Options**, and then click the **Notations** tab. From the Default Notation box choose OMT, Booch or UML. Subsequent code generation, reverse engineering and update operations can then consistently utilize the new notation.

# *Appendix A*

# *Attribute Properties*

## CodeName

The CodeName property specifies the name for the class attribute in the generated code.

You need to set this property only if you want the class attribute to be named differently than it is in the Rational Rose model. This is especially useful when the Rational Rose model and code are expressed in different natural languages. The CodeName value should be a valid C++ identifier. If it is not a valid C++ identifier, the C++ code generator performs the following substitutions:

- White space characters and hyphens are changed to underscores
- All other characters except letters, digits, and underscores are changed to "X"
- If the first character is a number, an "N" precedes the number

***Table 39 CodeName Values***

| If you select: | The action is: |
|---|---|
| C++ Identifier | The class attribute is assigned the name of the identifier. This is especially useful when supporting dual languages. |
| <blank> | (Default) The C++ code generator produces a name for the class attribute from the model. |

## DataMemberFieldSize

The DataMemberFieldSize property specifies the number of bits as the size of the designated data member.

The following table lists the values for DataMemberFieldSize:

***Table 40 DataMemberFieldSize Values***

| If you select: | The action is: |
|---|---|
| <integer> | The C++ code generator specifies <integer> bits as the size of the data member. |
| <blank> | (Default) The C++ code generator does not produce the size for the data member. |

## DataMemberIsVolatile

If a data member is generated for this attribute, the declaration will be adorned with the v keyword.

The following table lists the values for DataMemberIsVolatile:

***Table 41 DataMemberIsVolatile Values***

| If you select: | The action is: |
|---|---|
| True | A data member generated for this attribute has its declaration adorned with the v keyword. |
| False | The declaration is not adorned with the v keyword. |

## DataMemberMutability

The DataMemberMutability property identifies how the declaration is adorned if a data member is generated for this attribute.

The following table lists the values for DataMemberMutability:

***Table 42 DataMemberMutability Values***

| If you select: | The action is: |
|---|---|
| Mutable | A data member generated for this attribute, has its declaration prefixed with the `mutable` keyword. |
| Const | A data member is generated for this attribute, has its declaration adorned with the `const` keyword. |
| Unrestricted | (Default) No `const` or `mutable` adornments are generated. |

## DataMemberName

The DataMemberName property specifies the name the C++ code generator produces for a data member for a class attribute. The default value is:

```
$attribute
```

When the C++ code generator produces a data member, $attribute expands to the label of the class attribute in the model or the name specified in the attribute's CodeName property.

The following class diagram and code example illustrate a class attribute and the data member that the C++ code generator produces for it by default:

```
Name:  high_temperature
Type:  Temperature
Default Value:  70
Containment:  By-Value

// Data Members for Attributes
...
Temperature high_temperature;
```

You can change the form of the names the C++ code generator uses for class attributes by changing the format of DataMemberName in each constructor. You can also refer to $attribute in DataMemberName. Note that if $attribute is followed by a character that can appear in an identifier, you must enclose "attribute" in braces {}.

For example, if you set DataMemberName to:

```
${attribute}_data
```

the C++ code generator produces the following data member for the class attribute:

```
// Data Members for Attributes
...
Temperature the_temperature_data;
```

You can control the case of the name derived from $attribute. This table describes the possible options:

***Table 43 DataMemberName $attribute Values***

| If you enter: | The action is: |
|---|---|
| ${attribute:l} | All characters in the attribute name are converted to lower case. |
| ${attribute:u} | All characters in the attribute name are converted to upper case. |
| ${attribute:f} | The case of the first character in the attribute name is inverted. |
| ${attribute:i} | The case of all characters in the attribute name is inverted. |

## DataMemberVisibility

The DataMemberVisibility property specifies the visibility of the data member generated from the class attribute. The C++ code generator uses this information to determine access for the generated data member.

The following table lists the values for DataMemberVisibility:

***Table 44 DataMemberVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | The C++ code generator produces the data member with public access. |
| Protected | The C++ code generator produces the data member with protected access. |

## GenerateDataMember

The GenerateDataMember property specifies whether or not the C++ code generator produces a data member corresponding to the class attribute. The name of the data member is determined by the class attribute's DataMemberName value. The data member type is specified in Rational Rose if the attribute's containment is by-value. The data member type is a pointer to the type specified in Rational Rose if the attribute's containment is by-reference.

You may also specify a default value for the data member in Rational Rose. If the data member is `static`, Rational Rose generates a declaration in the class header and a definition in the class body. If the data member is not `static`, Rational Rose generates a declaration in the class header and a constructor initializer in each constructor. All attributes are `static` in a class utility.

You set GenerateDataMember to False if you want to provide your own data member definition or if you want to implement the class attribute in some other way. After you generate code, you edit the generated file and add your own data member definition (or some other implementation of the class attribute) between the source markers for the data member. Be sure to change the preserve setting in the source marker to `yes`.

The following table lists the values for GenerateDataMember:

***Table 45 GenerateDataMember Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator produces a data member for the class attribute. |
| False | The C++ code generator does not generate a data member for the class attribute. |

## GenerateGetOperation

The GenerateGetOperation property specifies whether the C++ code generator produces a get member function which accesses the value of the data member generated from the class attribute. The get member function name is determined by the class attribute's GetName value. The result type of the get member function is the type of the data member.

You set GenerateGetOperation to False if you do not want to provide a member function for accessing the data member. If you want to create a custom get operation, set GenerateGetOperation to False and then define your get operation in the **Class Specification**, as you would for any user-defined operation.

The following table lists the values for GenerateGetOperation:

***Table 46 GenerateGetOperation Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator produces a get operation for the data member. |
| False | The C++ code generator does not generate a get operation for the data member. |

## GenerateSetOperation

The GenerateSetOperation property specifies whether the C++ code generator produces a set member function that modifies the value of the data member generated from the class attribute. The set member function name is determined by the class attribute's SetName value. By default, the result type of the set member function is `void`. You can change the result type of the set member function by setting the class attribute's SetReturnsValue property.

You set GenerateSetOperation to False if you do not want to provide a member function for modifying the data member. If you want to create a custom set operation, set GenerateSetOperation to False and then define your set operation in the **Class Specification**, as you would for any user-defined operation.

The following table lists the values for GenerateSetOperation:

***Table 47 GenerateSetOperation Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator produces a set operation for the data member. |
| False | The C++ code generator does not generate a set operation for the data member. |

## GetIsConst

The GetIsConst property specifies whether the C++ code generator produces the get member function for a data member with the `const` keyword. `const` member functions cannot modify class data members.

The following table lists the values for GetIsConst. In this table, `Temperature` is the attribute type:

***Table 48 GetIsConst Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True (Default) | **`const Temperature get_high_temperature const`** |
| False | **`Temperature get_high_temperature`** |

## GetName

The GetName property specifies the name the C++ code generator produces for a get member function for a class attribute. The default value is:

```
get_$attribute
```

When the C++ code generator produces a get member function for a data member, $attribute expands to the label of the class attribute in the model, or the name specified in the attribute's CodeName property.

The following class diagram and code example illustrate a class attribute and the get member function that the C++ code generator produces for it by default:

```
Name:  high_temperature
Type:  Temperature
Default Value:  70
Containment:  By-Value

// Get and Set Operations for Attributes
const Temperature get_high_temperature() const;
```

You can change the form of the names that the C++ code generator produces for get member functions by changing the GetName format. Note that if $attribute is followed by a character that can appear in an identifier, you must enclose "attribute" in braces {}.

For example, if you set GetName to:

```
${attribute}_get
```

the C++ code generator produces the following get member function for the class attribute:

```
// Get and Set Operations for Attributes
const Temperature high_temperature_get () const;
```

You can control the case of the name derived from $attribute. This table describes the possible options:

***Table 49 GetName $attribute Values***

| If you enter: | The action is: |
|---|---|
| ${attribute:l} | All characters in the attribute name are converted to lower case. |
| ${attribute:u} | All characters in the attribute name are converted to upper case. |
| ${attribute:f} | The case of the first character in the attribute name is inverted. |
| ${attribute:i} | The case of all characters in the attribute name is inverted. |

## GetResultIsConst

The GetResultIsConst property returns a `const` value if a get function is generated for this element.

The following table lists the values for GetResultIsConst:

***Table 50 GetResultIsConst Values***

| If you select: | The action is: |
|---|---|
| True | If a get function is generated for this item, it returns a `const` value. |
| False | If a get function is generated for this item, it returns a non-`const` value. |
| Same_As_Function | If a get function is generated for this item, it returns a `const` value if the function is `const` and a non-`const` value if the function is not `const` (as set by GetIsConst). |

## GetSetByReference

The GetSetByReference property specifies whether values in the get and set member functions are passed by reference or by value. By default, the C++ code generator produces get and set member functions for a class attribute to pass arguments and return values by value. You set GetSetByReference to True if you want the get and set member functions to pass arguments and return values by reference.

The following table lists the values for GetSetByReference. In this table, `Temperature` is the name of the supplier class of the class attribute and `the_high_temperature` is the name of the class attribute:

***Table 51 GetSetByReference Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | **`Temperature & get_high_temperature;void set_high_temperature(const Temperature &value)`** |
| False (Default) | **`Temperature get_high_temperature; void set_high_temperature(const Temperature value)`** |

## GetSetKinds

The GetSetKinds property specifies the kind of member functions that are generated for the get and set operations for a data member. The C++ code generator produces additional keywords in the declarations of the get and set member functions based on the value of GetSetKinds, such as `static` or `virtual`.

The following table lists the values for GetSetKinds. In this table, `T` is the name of the supplier class in the class attribute and `the_T` is the name of the class attribute:

***Table 52  GetSetKinds Values***

| If you select: | The C++ code generator produces: |
|---|---|
| Common (Default) | **`T get_the_T();`**<br>**`void set_the_T(const T value)`** |
| Virtual | **`virtual T get_the_T();`**<br>**`virtual void set_the_T(const T value)`** |
| Static | **`static T get_the_T(A &client);`**<br>**`static void set_the_T(A &client, const T value);`** |
| Abstract | **`virtual T get_the_T() = 0;`**<br>**`virtual void set_the_T(const T value) = 0;`** |
| Friend | **`friend T get_the_T(A &client);`**<br>**`friend void set_the_T(A &client, const T value);`** |

Note that if the class attribute itself is `static`, the only legal values for GetSetKinds are “Common” and "Static." In both cases, the C++ code generator produces:

```
static T get_the_T();
static void set_the_T(const T value);
```

## InlineGet

The InlineGet property specifies whether the C++ code generator inlines get operations.

***Table 53  InlineGet Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator inlines get operations. |
| False | The C++ code generator does not inline get operations. |

## InlineSet

The InlineSet property specifies whether the C++ code generator inlines set operations.

***Table 54 InlineSet Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator inline sets operations. |
| False | The C++ code generator does not inline set operations. |

## SetName

The SetName property specifies the name the C++ code generator produces for a set member function for a class attribute. The default value is:

```
set_$attribute
```

When the C++ code generator produces a set member function for a data member, $attribute expands to the label of the class attribute in the model, or the name specified in the attribute's CodeName property.

The following code example illustrates a class attribute and the default set member function produced for it by the C++ code generator:

```
Name:  high_temperature
Type:  Temperature
Default Value:  70
Containment:  By-Value

// Get and Set Operations for Attributes
void set_high_temperature(const Temperature value);
```

You can change the form of the names that the C++ code generator produces for set member functions by changing the format of SetName. Note that if $attribute is followed by a character that can appear in an identifier, you must enclose "attribute" in braces {}.

For example, if SetName is:

```
${attribute}_set
```

the C++ code generator produces the following set member function for the class attribute:

```
// Get and Set Operations for Attributes
void high_temperature_set (const Temperature value);
```

You can control the case of the name derived from $attribute. This table describes the values.

***Table 55  SetName $attribute Values***

| If you enter: | The action is: |
|---|---|
| ${attribute:l} | All characters in the attribute name are converted to lower case. |
| ${attribute:u} | All characters in the attribute name are converted to upper case. |
| ${attribute:f} | The case of the first character in the attribute name is inverted. |
| ${attribute:i} | The case of all characters in the attribute name is inverted. |

## SetReturnsValue

The SetReturnsValue property specifies whether the C++ code generator produces the set member function for a class attribute with a non-`void` return type. By default, the C++ code generator produces the set member function with return type `void`. However, sometimes it is convenient for the set member function to return the value to which the data member is set in the function.

The following table lists the values for SetReturnsValue. In this table, `Temperature` is the name of the supplier class of the class attribute and `high_temperature` is the name of the class attribute:

***Table 56  SetReturnsValue Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | **`const Temperature set_high_temperature (const Temperature value)`** |
| False (Default) | **`void set_high_temperature (const Temperature value)`** |

# *Appendix B*

# *Class Properties*

## AssignmentKind

The AssignmentKind property specifies the kind of member function that is generated for the assignment operation (`operator=`) of a class. The C++ code generator produces additional keywords in the declaration of the assignment member function based on the value of AssignmentKind.

The following table lists the values for AssignmentKind. In this table, `T` is the name of the class for which the assignment operation is defined:

***Table 57 AssignmentKind Values***

| If you select: | The C++ code generator produces: |
|---|---|
| Common (Default) | **`const T & operator=(const T &right)`** |
| Virtual | **`virtual const T & operator=(const T &right`** |
| Abstract | **`virtual const T & operator=(const T &right) = 0;`** |

## AssignmentVisibility

The AssignmentVisibility property specifies the visibility of the assignment member function (`operator=`) that the C++ code generator produces for the class.

The following table lists the values for AssignmentVisibility:

***Table 58 AssignmentVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | (Default) The C++ code generator produces the assignment member function with public access. |
| Protected | The C++ code generator produces the assignment member function with protected access. |
| Private | The C++ code generator produces the assignment member function with private access. |
| Implementation | The C++ code generator produces the assignment member function in a second private area called private implementation. |

## ClassKey

If the ClassKey property is non-blank, its value is used to generate the definition and any forward declarations of this class. If it is blank, then the `class` keyword is used.

The possible values of ClassKey include any valid C++ class key: `class`, `struct`, or `union`.

## CodeName

The CodeName property specifies the name for the class in the generated code.

You need to set this property only if you want the class to be named differently than it is in the Rational Rose model. This is especially useful when the Rational Rose model and code are expressed in different natural languages. The value of CodeName should be a valid C++ identifier. If it is not a valid identifier, the C++ code generator performs the following substitutions:

- White space characters and hyphens are changed to underscores
- All other characters except letters, digits, and underscores are changed to "X"
- If the first character is a number, an "N" precedes the number

***Table 59 CodeName Values***

| If you select: | The action is: |
|---|---|
| C++ Identifier | The class is assigned the name of the identifier. |
| <blank> | (Default) The C++ code generator produces a class name from the model. |

## CopyConstructorVisibility

The CopyConstructorVisibility property specifies the visibility of the copy constructor that the C++ code generator produces for the class.

The following table lists the values for CopyConstructorVisibility:

***Table 60 CopyConstructorVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | (Default) The C++ code generator produces the copy constructor with public access. |
| Protected | The C++ code generator produces the copy constructor with protected access. |
| Private | The C++ code generator produces the copy constructor with private access. |
| Implementation | The C++ code generator produces the copy constructor in a second private area called private implementation. |

## DefaultConstructorVisibility

The DefaultConstructorVisibility property specifies the visibility of the default constructor that the C++ code generator produces for the class.

The following table lists the values for DefaultConstructorVisibility:

***Table 61  DefaultConstructorVisibility Values***

| **If you select:** | **The action is:** |
|---|---|
| Public | (Default) The C++ code generator produces the default constructor with public access. |
| Protected | The C++ code generator produces the default constructor with protected access. |
| Private | The C++ code generator produces the default constructor with private access. |
| Implementation | The C++ code generator produces the default constructor in a second private area called private implementation. |

## DereferenceKind

The DereferenceKind property specifies the kind of member function that is generated for the dereference operation (`operator*`) of a class. The C++ code generator produces additional keywords in the declaration of the dereference member function based on the value of the Dereference Kind property.

The following table lists the values for DereferenceKind. In this table, `TPtr` is the name of the class for which the dereference operation is defined and `T` is the result type as specified by DereferenceResultType:

***Table 62  DereferenceKind Values***

| **If you select:** | **The C++ code generator produces:** |
|---|---|
| Common (Default) | **`T operator*() const;`** |
| Virtual | **`virtual T operator*() const;`** |
| Abstract | **`virtual T operator*() const = 0;`** |

## DereferenceResultType

The DereferenceResultType property specifies the result type of the dereference member function. The following table lists the values for DereferenceResultType:

***Table 63 DereferenceResultType Values***

| If you enter: | The action is: |
|---|---|
| literal | The C++ code generator produces the dereference operation (operator*) with *literal* as the result type. |
| <blank> | (Default) The C++ code generator produces the dereference operation (operator*) with result type void. It is recommended that you do not leave this property blank. |

## DereferenceVisibility

The DereferenceVisibility property specifies the visibility of the dereference member function (operator*) that the C++ code generator produces for the class.

The following table lists the values for DereferenceVisibility:

***Table 64 DereferenceVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | (Default) The C++ code generator produces the dereference member function with public access. |
| Protected | The C++ code generator produces the dereference member function with protected access. |
| Private | The C++ code generator produces the dereference member function with private access. |
| Implementation | The C++ code generator produces the dereference member function in a second private area called private implementation. |

## DestructorKind

The DestructorKind property specifies the kind of member function that is generated for the destructor of a class. The C++ code generator produces additional keywords in the destructor's declaration based on the value of DestructorKind.

The following table lists the values for DestructorKind. In this table, `T` is the name of the class for which the destructor is defined:

***Table 65 DestructorKind Values***

| If you select: | The C++ code generator produces: |
|---|---|
| Common (Default) | `~T( )` |
| Virtual | `virtual ~T( )` |

## DestructorVisibility

The DestructorVisibility property specifies the visibility of the destructor member function that the C++ code generator produces for the class.

The following table lists the values for DestructorVisibility:

***Table 66 DestructorVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | (Default) The C++ code generator produces the destructor with public access. |
| Protected | The C++ code generator produces the destructor with protected access. |
| Private | The C++ code generator produces the destructor with private access. |
| Implementation | The C++ code generator produces the destructor in a second private area called private implementation. |

## EqualityKind

The EqualityKind property specifies the kind of member functions that are generated for the equality operations (operator== and operator!=) of a class. The C++ code generator produces additional keywords in the declarations of the equality member functions based on the value of EqualityKind.

The following table lists the values for EqualityKind. In this table, `T` is the name of the class for which the equality operations are defined:

***Table 67 EqualityKind Values***

| If you select: | The C++ code generator produces: |
|---|---|
| Common (Default) | `int operator==(const T &right) const;`<br>`int operator!=(const T &right) const;` |
| Virtual | `virtual int operator==(const T &right) const;`<br>`virtual int operator!=(const T &right) const;` |
| Abstract | `virtual int operator==(const T &right) const = 0;`<br>`virtual int operator!=(const T &right) const = 0;` Friend `friend int operator==(const T &left, const T &right);`<br>`friend int operator!=(const T &left, const T &right);` |

The type returned by the equality functions is determined by the BooleanType Project property.

## EqualityVisibility

The EqualityVisibility property specifies the visibility of the equality member functions (`operator==` and `operator!=`) that the C++ code generator produces for the class.

The following table lists the values for EqualityVisibility:

***Table 68 EqualityVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | (Default) The C++ code generator produces the equality member functions with public access. |
| Protected | The C++ code generator produces the equality member functions with protected access. |
| Private | The C++ code generator produces the equality member functions with private access. |
| Implementation | The C++ code generator produces the equality member functions in a second private area called private implementation. |

## ExplicitCopyConstructor

If a copy constructor is generated for the class, the ExplicitCopyConstructor property makes it an `explicit` constructor.

The following table lists the values for ExplicitCopyConstructor:

***Table 69 ExplicitCopyConstructor Values***

| If you select: | The action is: |
|---|---|
| True | If a copy constructor is generated for the class, the Explicit Copy Constructor property makes it an `explicit` constructor. |
| False | If a copy constructor is generated for the class, it is a non-`explicit` constructor. |

## ExplicitDefaultConstructor

If a default constructor is generated for the class, the ExplicitDefaultConstructor property makes it an `explicit` constructor.

The following table lists the values for ExplicitDefaultConstructor:

***Table 70 ExplicitDefaultConstructor Values***

| If you select: | The action is: |
|---|---|
| True | If a default constructor is generated for the class, the ExplicitDefaultConstructor property makes it an `explicit` constructor. |
| False | If a default constructor is generated for the class, it is a non-`explicit` constructor. |

## GenerateAssignmentOperation

The GenerateAssignmentOperation property specifies whether the C++ code generator produces an assignment member function (operator=) for a class.

When the C++ code generator produces an assignment operation, it generates:

- A member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

The following table lists the values for GenerateAssignmentOperation:

***Table 71 GenerateAssignmentOperation Values***

| If you select: | The action is: |
|---|---|
| Declare and Define | (Default) The C++ code generator produces a declaration and a skeleton definition for the operator= member function. |
| Declare Only | The C++ code generator produces a declaration for the operator= member function to prevent C++ from supplying a default assignment operation. |
| Do Not Declare | The C++ code generator does not generate a declaration or a definition for the operator= member function. C++ supplies a default assignment operation. |

## Example of GenerateAssignmentOperation

This example shows a member function declaration and corresponding skeleton function definition generated for the assignment operation for class T:

- Declaration

```
// Assignment Operation
const T & operator=(const T &right);
```

- Function Definition

```
// Assignment Operation
const T & T::operator=(const T &right)
{
//##begin T::operator=%.body preserve=yes
//##end T::operator=%.body
}
```

## GenerateCopyConstructor

The GenerateCopyConstructor property specifies whether the C++ code generator produces a copy constructor for a class.

When the C++ code generator produces a copy constructor, it generates:

- A member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

The following table lists the values for GenerateCopyConstructor:

***Table 72 GenerateCopyConstructor Values***

| **If you select:** | **The action is:** |
|---|---|
| Declare and Define | (Default) The C++ code generator produces a declaration and a skeleton definition for the copy constructor. |
| Declare Only | The C++ code generator produces a declaration for the copy constructor to prevent C++ from supplying one. |
| Do Not Declare | The C++ code generator does not generate a declaration or a definition for the copy constructor. C++ will supply a copy constructor. |

***Note:*** *If you prefer, you may enter the default constructor into Rational Rose's list of operations. This allows you to supply documentation and alter information. If you choose to do this, set this property to* **Do Not Declare** *so that Rational Rose does not automatically generate a default constructor.*

### Example of GenerateCopyConstructor

This example shows a member function declaration and corresponding skeleton function definition generated for the copy constructor for class `T`:

- Declaration

```
T(const T& right);
```

- Function Definition

```
T::T(const T &right)
//##begin T::T%copy.hasinit preserve=no
//##end T::T%copy.hasinit
//##begin T::T%copy.initialization preserve=yes
//##end T::T%copy.initialization
{
//##begin T::T%copy.body preserve=yes
//##end T::T%copy.body
}
```

There are two code regions in which you may place a constructor initializer: hasinit and initialization. If you specify an initial value for an attribute, or if a container class object generator for a has relationship or association requires initialization, Rational Rose generates the appropriate constructor initializer in the `hasinit` section. Normally, this region is not preserved and Rational Rose regenerates it each time. Rational Rose does not generate constructor initializers for superclasses; the initialization section is provided for you to enter them.

## GenerateDefaultConstructor

The GenerateDefaultConstructor property specifies whether the C++ code generator produces a default constructor for a class.

When the C++ code generator produces a default constructor, it generates:

- A member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

The following table lists the values for GenerateDefaultConstructor:

***Table 73 GenerateDefaultConstructor Values***

| If you select: | The action is: |
|---|---|
| Declare and Define | (Default) The C++ code generator produces a declaration and a skeleton definition for the default constructor. |
| Declare Only | The C++ code generator produces a declaration for the default constructor to prevent C++ from supplying one. |
| Do Not Declare | The C++ code generator does not generate a declaration or a definition for the default constructor. C++ will supply a default constructor. |

## Example of GenerateDefaultConstructor

This example shows a member function declaration and corresponding skeleton function definition generated for the default constructor for class T:

- Declaration

```
T();
```

- Function Definition

```
T::T()
//##begin T::T%.hasinit preserve=no
//##end T::T%.hasinit
//##begin T::T%.initialization preserve=yes
//##end T::T%.initialization
{
//##begin T::T%.body preserve=yes
//##end T::T%.body
}
```

There are two code regions in which you may place a constructor initializer: hasinit and initialization. If you specify an initial value for an attribute, or if a container class object generator for a has relationship or association requires initialization, Rational Rose generates the appropriate constructor initializer in the hasinit section.

Normally, this region is not preserved and Rational Rose regenerates it each time. Rational Rose does not generate constructor initializers for superclasses; the initialization section is provided for you to enter them.

## GenerateDereferenceOperation

The GenerateDereferenceOperation property specifies whether the C++ code generator produces a dereference member function (`operator*`) for dereferencing a pointer to an object of a class.

When the C++ code generator produces a dereference operation, it generates:

- A member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

The following table lists the values for GenerateDereferenceOperation:

***Table 74 GenerateDereferenceOperation Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces a declaration and a skeleton definition for the operator* member function. |
| False | (Default) The C++ code generator does not generate an operator* member function. |

### Example of GenerateDereferenceOperation

This example shows a member function declaration and corresponding skeleton function definition generated for a dereference operation for class TPtr. In this example, the DereferenceResultType property is set to `T`.

- Declaration

```
// Dereference Operation
T operator*() const;
```

- Function Definition

  ```
  // Dereference Operation
  T TPtr::operator*() const
  {
  //##begin TPtr::operator*%.body preserve=yes
  //##end TPtr::operator*%.body
  }
  ```

## GenerateDestructor

The GenerateDestructor property specifies whether the C++ code generator produces a destructor for a class.

When the C++ code generator produces a destructor, it generates:

- A member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

The following table lists the values for GenerateDestructor:

***Table 75   GenerateDestructor Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator produces declarations and skeleton definitions for the destructor. |
| False | The C++ code generator does not generate a destructor. |

### Example of GenerateDestructor

This example shows a member function declaration and corresponding skeleton function definition generated for the destructor for class `T`:

- Declaration

  ```
  // Destructor
  ~T();
  ```

- Function Definition

  ```
  // Destructor
  ```

```
 T::~T()
{
//##begin T::~T%.body preserve=yes
//##end T::~T%.body
}
```

## GenerateEmptyRegions

The GenerateEmptyRegions property identifies the type of protected region associated with the item.

The following table lists the values for GenerateEmptyRegions:

***Table 76 GenerateEmptyRegions Values***

| If you select: | The action is: |
|---|---|
| None | No empty protected region associated with the item will be generated in the source file. |
| Preserved | An empty protected region associated with the item will be generated only if it has a “preserve=yes” clause. |
| Unpreserved | An empty protected region associated with the item will be generated only if it has a “preserve=no” clause. |
| All | An empty protected region associated with the item will always be generated in the source file. |

## GenerateEqualityOperation

The GenerateEqualityOperations property specifies whether the C++ code generator produces equality and inequality member or friend functions (`operator==` and `operator!=`) for a class.

When the C++ code generator produces equality operations, it generates the following for each operation:

- A member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

The following table lists the values for GenerateEqualityOperations:

***Table 77 GenerateEqualityOperations Values***

| If you select: | The action is: |
|---|---|
| Declare and Define | (Default) The C++ code generator produces declarations and skeleton definitions for the operator== and operator!= member functions. |
| Declare Only | The C++ code generator produces declarations for the operator== and operation!= member functions to prevent C++ from supplying default equality operations. |
| Do Not Declare | The C++ code generator does not generate declarations or definitions for the operator== and operator!= member functions. C++ will supply default equality operations. |

The type returned by the equality functions is determined by the BooleanType Project property.

## Example of GenerateEqualityOperations

This example shows member function declarations and corresponding skeleton function definitions generated for equality operations for class `T`. In this example, the BooleanType Project property is set to `int`:

- Declarations

```
// Equality Operations
int operator==(const T &right) const;
int operator!=(const T &right) const;
```

- Function Definitions

```
// Equality Operations
int T::operator==(const T &right) const
{
//##begin T::operator==%.body preserve=yes
//##end T::operator==%.body
}
int T::operator!=(const T &right) const
```

```
{
//##begin T::operator!=%.body preserve=yes
//##end T::operator!=%.body
}
```

## GenerateIndirectionOperation

The GenerateIndirectionOperation property specifies whether the C++ code generator produces an indirection member function (`operator->`) for accessing an object of a class through a pointer to that object.

When the C++ code generator produces an indirection operation, it generates:

- A member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

The following table lists the values for GenerateIndirectionOperation:

***Table 78 GenerateIndirectionOperation Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces a declaration and a skeleton definition for the `operator->` member function. |
| False | (Default) The C++ code generator does not generate an `operator->` member function. |

### Example of GenerateIndirectionOperation

This example shows a member function declaration and corresponding skeleton function definition generated for an indirection operation for class `TPtr`. In this example, the IndirectionResultType property is set to `T*`.

- Declaration

```
// Indirection Operation
T* operator->() const;
```

- Function Definition

  ```
  // Indirection Operation
  T* TPtr::operator->() const
  {
  //##begin TPtr::operator->%.body preserve=yes
  //##end TPtr::operator->%.body
  }
  ```

## GenerateRelationalOperations

The GenerateRelationalOperations property specifies whether the C++ code generator produces relational member or friend functions (`operator<`, `operator<=`, `operator>`, and `operator>=`) for comparing class objects.

When the C++ code generator produces relational operations, it generates the following for each operation:

- A member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

The following table lists the values for GenerateRelationalOperations:

***Table 79 GenerateRelationalOperations Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces declarations and skeleton definitions for the `operator<`, `operator<=`, `operator>`, and `operator>=` member functions. |
| False | (Default) The C++ code generator does not generate relational member functions. |

The type returned by the relational functions is determined by the BooleanType Project property.

## Example of GenerateRelationalOperations

This example shows member function declarations and corresponding skeleton function definitions generated for relational operations for class `T`. In this example, the BooleanType Project property is set to `int`:

- Declarations

```
// Relational Operations
int operator<(const T &right) const;
int operator>(const T &right) const;
int operator<=(const T &right) const;
int operator>=(const T &right) const;
```

- Function Definitions

  ```
  // Relational Operations
  int T::operator<(const T &right) const
  {
  //##begin T::operator<%.body preserve=yes
  //##end T::operator<%.body
  }
  int T::operator>(const T &right) const
  {
  //##begin T::operator>%.body preserve=yes
  //##end T::operator>%.body
  }
  int T::operator<=(const T &right) const
  {
  //##begin T::operator<=%.body preserve=yes
  //##end T::operator<=%.body
  }
  int T::operator>=(const T &right) const
  {
  //##begin T::operator>=%.body preserve=yes
  //##end T::operator>=%.body
  }
  ```

## GenerateStorageMgmtOperations

The GenerateStorageMgmtOperations property specifies whether the C++ code generator produces storage management member functions (operator `new` and operator `delete`) for allocating and freeing class objects.

When the C++ code generator produces storage management operations, it generates the following for each operation:

- A static member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

The following table lists the values for GenerateStorageMgmtOperations:

***Table 80 GenerateStorageMgmtOperations Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces declarations and skeleton definitions for the operator `new` and operator `delete` member functions. |
| False | (Default) The C++ code generator does not generate operator `new` and operator `delete` member functions. |

## Example of GenerateStorageMgmtOperations

This example shows member function declarations and corresponding skeleton function definitions generated for storage management operations for class `T`:

- Declarations

  ```
  // Storage Management Operations
  static void * operator new(size_t size);
  static void operator delete(void *object);
  ```

- Function Definitions

  ```
  // Storage Management Operations
  void * T::operator new(size_t size)
  {
  //##begin T::operatornew%.body preserve=yes
  //##end T::operatornew%.body
  }
  void T::operator delete(void *object)
  {
  //##begin T::operatordelete%.body preserve=yes
  //##end T::operatordelete%.body
  }
  ```

## GenerateStreamOperations

The GenerateStreamOperations property specifies whether the C++ code generator produces stream friend functions (`operator>>` and `operator<<`) for reading and writing class objects. Stream operations are always generated as friend functions.

When the C++ code generator produces stream operations, it generates the following for each operation:

- A member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

You can overload `operator<<` and `operator>>` to implement your own left and right shift operations. To do this, you define left and right shift operations in the **Class Specification**, as for any user-defined operation. Note that the C++ code generator places all `operator<<` and `operator>>` member functions following the comment line `// Stream Operations`.

The following table lists the values for GenerateStreamOperations:

***Table 81 GenerateStreamOperations Values***

| **If you select:** | **The action is:** |
|---|---|
| True | The C++ code generator produces declarations and skeleton definitions for the operator>> and operator<< member functions. |
| False | (Default) The C++ code generator does not generate operator>> and operator<< member functions. |

## Example of GenerateStreamOperations

This example shows member function declarations and corresponding skeleton function definitions generated for stream operations for class `T`:

- Declarations:

```
// Stream Operations
friend ostream & operator<<(ostream &s, const T &right)
const;
friend istream & operator>>(istream &s, T &object);
```

- Function Definitions

```
// Stream Operations
friend ostream & T::operator<<(ostream &s, const T &right)
const
{
//##begin T::operator<<%.body preserve=yes
//##end T::operator<<%.body
}
friend istream & T::operator>>(istream &s, T &object)
{
//##begin T::operator>>%.body preserve=yes
//##end T::operator>>%.body
}
```

## GenerateSubscriptOperation

The GenerateSubscriptOperation property specifies whether the C++ code generator produces a subscript member function (`operator[]`) for accessing a particular object of a class.

When the C++ code generator produces a subscript operation, it generates:

- A member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

The following table lists the values for GenerateSubscriptOperation:

***Table 82 GenerateSubscriptOperation Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces a declaration and a skeleton definition for the `operator[]` member function. |
| False | (Default) The C++ code generator does not generate an `operator[]` member function. |

## Example of GenerateSubscriptOperation

This example shows a member function declaration and corresponding skeleton function definition generated for a subscript operation for class `T`. In this example, the SubscriptResultType property is set to `T&`.

- Declaration

```
// Subscription Operation
void operator[](const int index) const;
```

- Function Definition

```
// Subscription Operation
T& T::operator[](const int index) const
{
//##begin T::operator[]%.body preserve=yes
//##end T::operator[]%.body
}
```

## ImplementationType

The ImplementationType property allows you to implement a class using either a class definition or an elemental data type. By default, the C++ code generator generates a class definition for a class in a Rational Rose model. You set the ImplementationType property when you want to implement a class as an elemental data type instead of as a C++ class. The C++ code generator produces a `typedef` that maps the class to its implementation type.

For example, you might create a class called `SmallString` whose ImplementationType property is set to `char[3]`. When the C++ code generator produces code for class Small String, it generates a `typedef` instead of a class definition:

```
typedef char SmallString[3];
```

To define an enumeration type, set ImplementationType to an enumeration type definition, for example:

```
enum {red = 10, blue = 20, green = 30, yellow = 40}
```

The following table lists the values for ImplementationType:

***Table 83 ImplementationType Values***

| If you enter: | The action is: |
|---|---|
| *type name* | The C++ code generator produces a `typedef` that maps the class to its implementation type and does not generate a class definition for the class. |
| <blank> | (Default) The C++ code generator produces a class definition for the class. |

## IndirectionKind

The IndirectionKind property specifies the kind of member function that is generated for the indirection operation (`operator->`) of a class. The C++ code generator produces additional keywords in the declaration of the indirection member function based on the IndirectionKind value.

The following table lists the values for IndirectionKind. In this table, `TPtr` is the name of the class for which the indirection operation is defined and `T*` is the result type as specified by the IndirectionResultType property:

***Table 84 IndirectionKind Values***

| If you select: | The C++ code generator produces: |
|---|---|
| Common (Default) | **`T* operator->() const;`** |
| Virtual | **`virtual T* operator->() const;`** |
| Abstract | **`virtual T* operator->() const = 0;`** |

## IndirectionResultType

The IndirectionResultType property specifies the result type of the indirection member function.

The following table lists the values for IndirectionResultType:

***Table 85 IndirectionResultType Values***

| If you enter: | The action is: |
|---|---|
| *literal* | The C++ code generator produces the indirection operation (operator->) with *literal* as the result type. |
| <blank> | (Default) The C++ code generator produces the indirection operation (`operator->`) with result type `void`. It is recommended that you do not leave this property blank. |

## IndirectionVisibility

The IndirectionVisibility property specifies the visibility of the indirection member function (`operator->`) that the C++ code generator produces for the class.

The following table lists the values for IndirectionVisibility:

***Table 86 IndirectionVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | (Default) The C++ code generator produces the indirection member function with public access. |
| Protected | The C++ code generator produces the indirection member function with protected access. |
| Private | The C++ code generator produces the indirection member function with private access. |
| Implementation | The C++ code generator produces the indirection member function in a second private area called private implementation. |

## InlineAssignmentOperation

The InlineAssignmentOperation property specifies whether the C++ code generator inlines an assignment member function (`operator=`) for a class.

The following table lists the values for InlineAssignmentOperation:

***Table 87 InlineAssignmentOperation Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator inlines the assignment operations for the class. |
| False | (Default) The C++ code generator does not inline the assignment operations for the class. |

## InlineCopyConstructor

The InlineCopyConstructor property specifies whether the C++ code generator inlines a copy constructor for a class.

The following table lists the values for InlineCopyConstructor:

***Table 88 InlineCopyConstructor Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator inlines the copy constructor for the class. |
| False | (Default) The C++ code generator does not inline the copy constructor for the class. |

## InlineDefaultConstructor

The InlineDefaultConstructor property specifies whether the C++ code generator inlines the default constructor for the class. If no default constructor is generated, this property has no effect. See GenerateDefaultConstructor.

The following table lists the values for InlineDefaultConstructor:

***Table 89 InlineDefaultConstructor Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator inlines the default constructor for the class. |
| False | (Default) The C++ code generator does not inline the default constructor for the class. |

## InlineDereferenceOperation

The InlineDereferenceOperation property specifies whether the C++ code generator inlines a dereference member function (`operator*`) for dereferencing a pointer to an object of a class.

The following table lists the values for InlineDereferenceOperation:

***Table 90 InlineDereference Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator inlines the dereferences for the class. |
| False | (Default) The C++ code generator does not inline dereferences for the class. |

## InlineDestructor

The InlineDestructor property specifies whether the C++ code generator inlines a destructor for a class.

The following table lists the values for InlineDestructor:

***Table 91 InlineDestructor Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator inlines the destructors for the class. |
| False | (Default) The C++ code generator does not inline destructors for the class. |

## InlineEqualityOperations

The InlineEqualityOperations property specifies whether the C++ code generator inlines equality and inequality member or friend functions (`operator==` and `operator!=`) for a class.

When the C++ code generator produces equality operations, it generates the following for each operation:

- A member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

The following table lists the values for InlineEqualityOperations:

***Table 92 InlinedEqualityOperations Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator inlines the equality operation for the class. |
| False | (Default) The C++ code generator does not inline the equality operation for the class. |

## InlineIndirectionOperation

The InlineIndirectionOperation property specifies whether the C++ code generator inlines an indirection member function (`operator->`) for accessing an object of a class through a pointer to that object.

The following table lists the values for InlineIndirectionOperation:

***Table 93 InlineIndirectionOperation Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator inlines the indirection operations for the class. |
| False | (Default) The C++ code generator does not inline the indirection operations for the class. |

## InlineRelationalOperations

The InlineRelationalOperations property specifies whether the C++ code generator inlines relational member or friend functions (`operator<`, `operator<=`, `operator>`, and `operator>=`) for comparing class objects.

When the C++ code generator produces relational operations, it generates the following for each operation:

- A member function declaration in the header file for a class.

  ***Note:*** *This code is generated in the implementation file if you have assigned the class to a module body.*

- A skeleton function definition with preserved code regions in the implementation file. You complete the member function by inserting implementation code between the source markers for the code regions.

The following table lists the values for InlineRelationalOperations:

***Table 94 InlineRelationalOperations Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator inlines the relational operation for the class. |
| False | (Default) The C++ code generator does not inline the relational operation for the class. |

The type returned by the relational functions is determined by the BooleanType Project property.

## InlineStorageMgmtOperations

The InlineStorageMgmtOperations property specifies whether the C++ code generator inlines storage management member functions (operator `new` and operator `delete`) for allocating and freeing class objects.

The following table lists the values for InlineStorageMgmtOperation:

***Table 95 InlineStorageMgmtOperation Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator inlines the storage management operations for the class. |
| False | (Default) The C++ code generator does not inline the storage management operations for the class. |

## InlineStreamOperations

The InlineStreamOperations property specifies whether the C++ code generator inlines stream friend functions (`operator>>` and `operator<<`) for reading and writing class objects. Stream operations are always generated as friend functions.

The following table lists the values for InlineStreamOperations:

***Table 96 InlineStreamOperations Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator inlines the stream operations for the class. |
| False | (Default) The C++ code generator does not inline the stream operations for the class. |

## InlineSubscriptOperation

The InlineSubscriptOperation property specifies whether the C++ code generator inlines a subscript member function (`operator=`) for a class.

The following table lists the values for InlineSubscriptOperation:

***Table 97 InlineSubscriptOperation Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator inlines the subscript operations for the class. |
| False | (Default) The C++ code generator does not inline the subscript operations for the class. |

## PutBodiesInSpec

If the value of PutBodiesInSpec is True, the implementation for the class is placed in the specification file (header). This would be used for compilers that need the definition of the template in every compilation unit.

## RelationalKind

The RelationalKind property specifies the kind of member functions that are generated for the relational operations (`operator<`, `operator<=`, `operator>`, and `operator>=`) of a class. The C++ code generator produces additional keywords in the declarations of the relational member functions based on the value of its RelationalKind property.

The following table lists the values for RelationalKind. In this table, `T` is the name of the class for which the relational operations are defined:

***Table 98 RelationalKind Values***

| If you select: | The C++ code generator produces: |
|---|---|
| Common | `int operator<(const T &right) const;` |
| (Default) | `int operator>(const T &right) const;` |
| | `int operator<=(const T &right) const;` |
| | `int operator>=(const T &right) const;` |
| Virtual | `virtual int operator<(const T &right) const;` |
| | `virtual int operator>(const T &right) const;` |
| | `virtual int operator<=(const T &right) const;` |
| | `virtual int operator>=(const T &right) const,` |
| Abstract | `virtual int operator<(const T &right) const =` |
| | `0;` |
| | `virtual int operator>(const T &right) const =` |
| | `0;` |
| | `virtual int operator<=(const T &right) const` |
| | `= 0;` |
| | `virtual int operator>=(const T &right) const` |
| | `= 0;` |
| Friend | `friend int operator<(const T &left, const` |
| | `  T&right);` |
| | `friend int operator>(const T &left, const` |
| | `  T&right);` |
| | `friend int operator<=(const T &left, const` |
| | `  T&right);` |
| | `friend int operator>=(const T &left, const` |
| | `  T&right);` |

The type returned by the relational functions is determined by the BooleanType Project property.

## RelationalVisibility

The RelationalVisibility property specifies the visibility of the relational member functions (`operator<`, `operator<=`, `operator>` and `operator>=`) that the C++ code generator produces for the class.

The following table lists the values for RelationalVisibility:

***Table 99 RelationalVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | (Default) The C++ code generator produces the relational member functions with public access. |
| Protected | The C++ code generator produces the relational member functions with protected access. |
| Private | The C++ code generator produces the relational member functions with private access. |
| Implementation | The C++ code generator produces the relational member functions in a second private area called private implementation. |

## StorageMgmtVisibility

The StorageMgmtVisibility property specifies the visibility of the storage management member functions (operator `new` and operator `delete`) that the C++ code generator produces for the class.

The following table lists the values for StorageMgmtVisibility:

***Table 100 StorageMgmtVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | (Default) The C++ code generator produces the `new` and `delete` member functions with public access. |
| Protected | The C++ code generator produces the `new` and `delete` member functions with protected access. |
| Private | The C++ code generator produces the `new` and `delete` member functions with private access. |
| Implementation | The C++ code generator produces the `new` and `delete` member functions in a second private area called private implementation. |

## StreamVisibility

The StreamVisibility property specifies the visibility of the stream member functions (`operator<<` and `operator>>`) that the C++ code generator produces for the class.

The following table lists the values for StreamVisibility:

***Table 101 StreamVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | (Default) The C++ code generator produces the stream member functions with public access. |
| Protected | The C++ code generator produces the stream member functions with protected access. |
| Private | The C++ code generator produces the stream member functions with private access. |
| Implementation | The C++ code generator produces the stream member functions in a second private area called private implementation. |

## SubscriptKind

The SubscriptKind property specifies the kind of member function that is generated for the subscript operation (`operator[]`) of a class. The C++ code generator produces additional keywords in the declaration of the subscript member function based on the value of SubscriptKind.

The following table lists the values for SubscriptKind. In this table, `T` is the name of the class for which the subscript operation is defined and `T&` is the result type as specified by SubscriptResultType:

***Table 102 SubscriptKind Values***

| If you select: | The C++ code generator produces: |
|---|---|
| Common (Default) | **`T& operator[ ](const int index) const;`** |
| Virtual | **`virtual T& operator[ ](const int index) const;`** |
| Abstract | **`virtual T& operator[ ](const int index) const = 0;`** |

## SubscriptResultType

The SubscriptResultType property specifies the result type of the subscript member function.

The following table lists the values for SubscriptResultType:

***Table 103 SubscriptResultType Values***

| If you enter: | The action is: |
|---|---|
| *literal* | The C++ code generator produces the subscript operation (operator[]) with *literal* as the result type. |
| <blank> | (Default) The C++ code generator produces the subscript operation (operator[]) with result type void. It is recommended that you do not leave this property blank. |

## SubscriptVisibility

The SubscriptVisibility property specifies the visibility of the subscript member function (operator[]) that the C++ code generator produces for the class.

The following table lists the values for SubscriptVisibility:

***Table 104 SubscriptVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | (Default) The C++ code generator produces the subscript member function with public access. |
| Protected | The C++ code generator produces the subscript member function with protected access. |
| Private | The C++ code generator produces the subscript member function with private access. |
| Implementation | The C++ code generator produces the subscript member function in a second private area called private implementation. |

# *Appendix C*

# *Class Category Properties*

## CodeName

The CodeName property indicates the C++ name for the namespace.

You need to set this property only if you want the class to be named differently than it is in the Rational Rose model. This is especially useful when the Rational Rose model and code are expressed in different natural languages. This property value should be a valid C++ identifier. If it is not a valid C++ identifier, the C++ code generator performs the following substitutions:

- White space characters and hyphens are changed to underscores
- All other characters except letters, digits, and underscores are changed to "X"
- If the first character is a number, an "N" precedes the number

***Table 105 CodeName Values***

| If you select: | The action is: |
|---|---|
| C++ Name | The string that identifies the C++ name for the namespace. |
| <blank> | (Default) The package does not represent a namespace. |

The following substitution symbols are defined for CodeName:

***Table 106 CodeName Substitution Symbols***

| Code | Explanation |
|---|---|
| $name | :model name |
| $identifier | :model name edited to conform to C++ rules for an identifier |
| $packageName | :same as name |
| $packageIdentifier | :same as identifier |
| $subsystemName | :model name for associated subsystem |
| $subsystemDir | :name of directory associated with subsystem |

## GenerateEmptyRegions

The GenerateEmptyRegions property identifies the type of protected region associated with the class.

The following table lists the values for GenerateEmptyRegions:

***Table 107 GenerateEmptyRegions Values***

| If you select: | The action is: |
|---|---|
| None | No empty protected region associated with the class is generated in the source file. |
| Preserved | An empty protected region associated with the class is generated only if it has a preserve=yes clause. |
| Unpreserved | An empty protected region associated with the class is generated only if it has a `preserve=no` clause. |
| All | An empty protected region associated with the class is always generated in the source file. |

## Indent

The Indent property specifies the indentation for a class category when the class category is a namespace. This property takes effect only when IsNamespace is True. The C++ code generator produces the specified indentation for the first-level items inside the namespace.

***Table 108 Indent Values***

| If you select: | The action is: |
|---|---|
| <integer> | The C++ code generator produces <integer>-space indentation for the first-level items inside the namespace. The default value is 2. |

## IsNamespace

The IsNamespace property specifies whether or not a class category is treated as a namespace in C++. For a class inside a class category, the C++ code generator normally produces a top-level C++ class. If a class category IsNamespace property is True, its C++ class is generated inside a C++ namespace with category name as the namespace name.

The following table lists the values for IsNamespace:

***Table 109 Is Namespace Property Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces a namespace for the class category, and places all its enclosing items inside the namespace. |
| False | (Default) The C++ code generator does not treat the class category as a namespace. |

# *Appendix D*

# *Dependency Properties*

## BodyReferenceOnly

The BodyReferenceOnly property specifies whether the reference due to the Dependency only takes effect on the body (implementation) file. The C++ code generator uses this property to decide where to produce the reference for the Dependency.

The following table lists the values for BodyReferenceOnly:

***Table 110 BodyReferenceOnly Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces the reference for the Dependency in the implementation file. |
| False | (Default) The C++ code generator produces the reference for the Dependency in the header file. |

## ForwardReferenceOnly

The ForwardReferenceOnly property specifies whether the C++ code generator produces a forward declaration for the supplier class of the association relationship before the class definitions in the module. If, in the Rational Rose model, there is a cycle consisting of two or more association, uses, inherits, and instantiates relationships, and navigable association roles, you must set ForwardReferenceOnly to True for at least one of the association or uses relationships, or navigable roles to break the cycle. You may also set this property to break cycles which include other kinds of dependencies, such as those specified by the module dependency relationship on a component diagram, or by includes you put in the AdditionalIncludes protected code region.

***Note:*** *This property does not apply to inherits or instantiates relationships because a forward declaration never provides sufficient information to the compiler in these cases.*

Each associate class is always dependent on its association class. An association class always has forward references to its associate class.

For example, if two classes are assigned to the same module and each class is the supplier in an association relationship with the other, as shown here, then a forward declaration is needed for one of the classes.

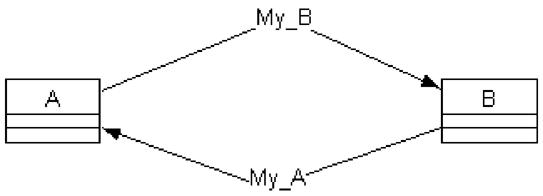


***Figure 56 Example of an Association Relationship***

Setting ForwardReferenceOnly to True for the "A to B" direction of the association (`My_B`) causes the C++ code generator to generate a forward declaration for class `B`, as shown by the following code example:

```
class B;
class A
{...
   // Data Members for Association Relationships
   B *My_B;
...};
class B
{...
   // Data Members for Association Relationships
   A *My_A;
...};
```

Note that this restricts how `A` can contain `B`. In particular, `A` cannot contain an instance of class `B`, although it can contain a pointer to an instance of `B`. It seldom makes sense to set ForwardReferenceOnly to True for an association-by-value because such a relationship usually implies containment. If the association role's GenerateDataMember property is False, however, it is possible to implement the association in a way that simulates containment without actual containment.

If the classes are assigned to different modules, then, by default, A's header file would contain B's header file and vice versa, creating a circular inclusion. To eliminate the circular inclusion, you set ForwardReferenceOnly to True for one of the association roles. The C++ code generator produces a forward declaration for the supplier class in the client module and suppresses the `#include` directive in the client module. In the following code example, ForwardReferenceOnly is set to True for the "A has B" relationship:

***Table 111 Eliminate Circular Inclusion***

| A's Header File | B's Header File |
|---|---|
| `class B;` | `#include "a.h"` |
| `...` | `...` |
| `class A{` | `class B {` |
| `...` | `...` |
| `};` | `};` |

The following table lists the values for ForwardReferenceOnly:

***Table 112 ForwardReferenceOnly Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces a forward declaration for the supplier class of the association relationship. If the supplier and the client are assigned to different modules, the C++ code generator suppresses the `#include` directive for the supplier module in the client module. |
| False | (Default) The C++ code generator does not generate a forward declaration for the supplier class of the association relationship. |

# Appendix E

# Has Properties

## CodeName

The CodeName property specifies the name for the has relationship in the generated code.

You set this property only if you want the has relationship to be named differently than it is in the Rational Rose model. This is especially useful when the Rational Rose model and code are expressed in different natural languages. This property value should be a valid C++ identifier. If it is not a valid C++ identifier, the C++ code generator performs the following substitutions:

- White space characters and hyphens are changed to underscores
- All other characters except letters, digits, and underscores are changed to "X"
- If the first character is a number, an "N" precedes the number

***Table 113 CodeName Values***

| **If you select:** | **The action is:** |
|---|---|
| C++ Identifier | The has relationship is assigned the name of the identifier. |
| &lt;blank&gt; | (Default) The C++ code generator produces a name for the has relationship from the model. |

## ContainerClass

The ContainerClass property specifies a data type for the data member generated from the has relationship. You can set ContainerClass to refer to your own container classes, which the C++ code generator will use to generate types for data members.

If you leave this property blank, the data members are generated whose types are container classes organized as lists, sets, or dictionaries. You can use this property when you want to generate other types of container classes, or if you want to choose the container class based on other criteria.

The following table lists the values for ContainerClass:

***Table 114 ContainerClass Values***

| If you enter: | The action is: |
|---|---|
| <blank> | (Default) The C++ code generator produces a type for the data member based on the supplier cardinality and containment of the has relationship. |
| *literal* | The C++ code generator uses *literal* as the data member type. Usually, *literal* is an instance of a template. The *literal* text may contain the variables $supplier and $limit. |

## DataMemberFieldSize

The DataMemberFieldSize property specifies the number of bits as the size of the designated data member.

The following table lists the values for DataMemberFieldSize:

***Table 115 DataMemberFieldSize Values***

| If you select: | The action is: |
|---|---|
| <integer> | The C++ code generator specifies <integer> bits as the size of the data member. |
| <blank> | (Default) The C++ code generator does not specify the size for the data member. |

## DataMemberIsVolatile

If a data member is generated for this relationship, the declaration will be adorned with the `v` keyword.

The following table lists the values for DataMemberIsVolatile:

*Table 116 DataMemberIsVolatile Values*

| If you select: | The C++ code generator produces: |
|---|---|
| True | If a data member is generated for this relationship, the declaration is adorned with the `v` keyword. |
| False | The declaration is not adorned with the `v` keyword. |

## DataMemberMutability

The DataMemberMutability property identifies how the declaration is adorned if a data member is generated for this relationship.

The following table lists the values for DataMemberMutability:

*Table 117 DataMemberMutability Values*

| If you select: | The C++ code generator produces: |
|---|---|
| Mutable | If a data member is generated for this relationship, the declaration is adorned with the `mutable` keyword. |
| Const | If a data member is generated for this relationship, the declaration is adorned with the `const` keyword. |
| Unrestricted | (Default) No `const` or `mutable` adornments are generated. |

## DataMemberName

The DataMemberName property specifies the name the C++ code generator produces for a data member for a has relationship. The default value is:

```
$relationship
```

When the C++ code generator produces a data member, $relationship and $target expand to the label of the has relationship in the model. If the has relationship is unlabeled, $relationship and $target expand to the value of the has relationship’s NameIfUnlabeled value.

The following class diagram and code example illustrate a has relationship and the data member that the C++ code generator produces for it by default:

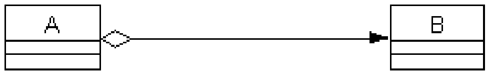


***Figure 57 Data Members for Has Relationships***

```
...
B the_B;
```

You can change the form of the names the C++ code generator uses for unlabeled has relationships by changing the DataMemberName format. You can also refer to $supplier in DataMemberName. Note that if either $relationship or $supplier is followed by a character that can appear in an identifier, you must enclose "relationship" in braces {}.

For example, if you set DataMemberName to:

```
${relationship}_data
```

the C++ code generator produces the following data member for the has relationship:

```
// Data Members for Has Relationships
...
B the_B_data;
```

You can control the case of the name derived from $relationship. This table describes the possible options (note that these options also can be used with $supplier, $targetClass, and $target):

***Table 118 DataMemberName Case Options***

| If you enter: | The action is: |
|---|---|
| ${relationship:l} | All characters in the relationship name are converted to lower case. |
| ${relationship:u} | All characters in the relationship name are converted to upper case. |
| ${relationship:f} | The case of the first character in the relationship name is inverted. |
| ${relationship:i} | The case of all characters in the relationship name is inverted. |

## DataMemberVisibility

The DataMemberVisibility property specifies the visibility of the data member generated from the has relationship. The C++ code generator uses this information to determine access for the generated data member.

The following table lists the values for DataMemberVisibility:

***Table 119 DataMemberVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | The C++ code generator produces the data member with public access. |
| Protected | The C++ code generator produces the data member with protected access. |

| If you select: | The action is: |
|---|---|
| Private | The C++ code generator produces the data member with private access. |
| Implementation | (Default) The C++ code generator produces the data member in a second private area called private implementation. |
| At Relationship Visibility | The C++ code generator produces the data member at the level of access specified for the relationship in the Rational Rose model. This choice is useful when you want to implement the relationship direct access to the data member, rather than protecting it with Get and Set, or other operations. Normally when the DataMemberVisibility is set to this value, you should set GenerateGetOperation and GenerateSetOperation to False. |

## ForwardReferenceOnly

The ForwardReferenceOnly property specifies whether the C++ code generator produces a forward declaration for the supplier class of the has relationship before the class definitions in the module. If, in the Rational Rose model, there is a cycle consisting of two or more has, uses, inherits, and instantiates relationships, and navigable association roles, you must set ForwardReferenceOnly to True for at least one of the has, uses relationships, or association roles to break the cycle. You may also use this property to break cycles which include other kinds of dependencies, such as those specified by the module dependency relationship on a component diagram, or by includes you put in the AdditionalIncludes protected code region.

***Note:*** *This property does not apply to inherits or instantiates relationships because a forward declaration never provides sufficient information to the compiler in these cases.*

For example, if two classes are assigned to the same module and each class is the supplier in a has relationship with the other, as shown here, then a forward declaration is needed for one of the classes.

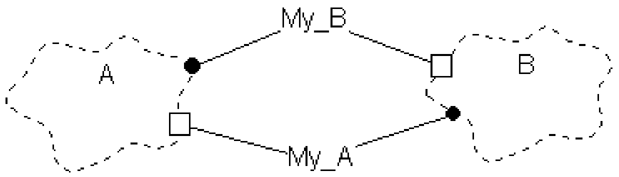


***Figure 58   Example of Forward Reference***

Setting ForwardReferenceOnly to True for the "A has B" relationship causes the C++ code generator to generate a forward declaration for class `B`, as shown by the following code example:

```
class B;
class A
{...
   // Data Members for Has Relationships
   B *My_B;
...};
class B
{...
   // Data Members for Has Relationships
   A *My_A;
...};
```

Note that this restricts how `A` can contain `B`. In particular, `A` cannot contain an instance of class `B`, although it can contain a pointer to an instance of `B`. It seldom makes sense to set ForwardReferenceOnly to True for a has-by-value relationship because such a relationship usually implies containment. However, if the has relationship's GenerateDataMember property is False, it is possible to implement a relationship that simulates containment without actual containment.

If the classes are assigned to different modules, then, by default, A's header file would contain B's header file and vice versa, creating a circular inclusion. To eliminate the circular inclusion, you set ForwardReferenceOnly to True for one of the relationships. The C++ code generator produces a forward declaration for the supplier class in the client module and suppresses the `#include` directive in the client module. In the following code example, ForwardReferenceOnly is True for the "A has B" relationship:

***Table 120 Eliminate Circular Inclusion***

| A's Header File | B's Header File |
|---|---|
| `class B;` | `#include "a.h"` |
| `...` | `...` |
| `class A{` | `class B {` |
| `...` | `...` |
| `};` | `};` |

The following table lists the values for ForwardReferenceOnly:

***Table 121 ForwardReferenceOnly Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces a forward declaration for the supplier class of the has relationship. If the supplier and the client are assigned to different modules, the C++ code generator suppresses the `#include` directive for the supplier module in the client module. |
| False | (Default) The C++ code generator does not generate a forward declaration for the supplier class of the has relationship. |

## GenerateDataMember

The GenerateDataMember property specifies whether the C++ code generator produces a data member corresponding to the has relationship. The name of the data member is determined by the has relationship’s DataMemberName value. The data member type is one that contains or designates objects of the supplier class.

You set GenerateDataMember to False if you want to provide your own data member definition or if you want to implement the has relationship in some other way. After you generate code, you edit the generated file and add your own data member definition (or some other implementation of the has relationship) between the source markers for the data member. Be sure to change the preserve setting in the source marker to `yes`.

The following table lists the values for GenerateDataMember:

***Table 122 GenerateDataMember Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator produces a data member for the has relationship. |
| False | The C++ code generator does not generate a data member for the has relationship. |

## GenerateGetOperation

The GenerateGetOperation property specifies whether the C++ code generator produces a get member function which accesses the value of the data member generated from the has relationship. The name of the get member function is determined by the has relationship's GetName value. The result type of the get member function is the type of the data member.

You set GenerateGetOperation to False if you do not want to provide a member function for accessing the data member. If you want to create a custom get operation, set GenerateGetOperation to False and then define your get operation in the class specification, as you would for any user-defined operation.

The following table lists the values for GenerateGetOperation:

***Table 123 GenerateGetOperation Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator produces a basic get operation for the data member. |
| False | The C++ code generator does not generate a get operation for the data member. |

## GenerateSetOperation

The GenerateSetOperation property specifies whether the C++ code generator produces a set member function which modifies the value of the data member generated from the has relationship. The name of the set member function is determined by the has relationship's SetName value. By default, the result type of the set member function is void. You can change the result type of the set member function by setting the has relationship's SetReturnsValue property.

You set GenerateSetOperation to False if you do not want to provide a member function for modifying the data member. If you want to create a custom set operation, set GenerateSetOperation to False and then define your set operation in the class specification, as you would for any user-defined operation.

The following table lists the values for GenerateSetOperation:

***Table 124 GenerateSetOperation Values***

| **If you select:** | **The action is:** |
|---|---|
| True | (Default) The C++ code generator produces a basic set operation for the data member. |
| False | The C++ code generator does not generate a set operation for the data member. |

## GetIsConst

The GetIsConst property specifies whether the C++ code generator produces the get member function for a data member with the `const` keyword. `const` member functions cannot modify class data members.

The following table lists the values for GetIsConst. In this table, `T` is the supplier class of the has relationship:

***Table 125 GetIsConst Values***

| **If you select:** | **The C++ code generator produces:** |
|---|---|
| True (Default) | **`const T get_the_T() const;`** |
| False | **`T get_the_T();`** |

## GetName

The GetName property specifies the name the C++ code generator produces for a get member function for a has relationship. The default value is:

```
get_$relationship
```

When the C++ code generator produces a get member function for a data member, $relationship and $target expand to the label of the has relationship in the model. If the has relationship is unlabeled, $relationship and $target expand to the has relationship's NameIfUnlabeled value.

The following class diagram and code example illustrate a has relationship and the get member function that the C++ code generator produces for it by default:

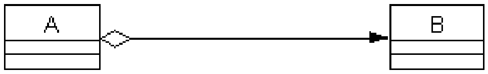


***Figure 59   Get Operations for Has Relationships***

```
const B get_the_B() const;
```

You can change the form of the names that the C++ code generator produces for get member functions by changing the format of GetName. You can also refer to $supplier and $targetClass in GetName. Note that if either $relationship, $target, $target Class, or $supplier is followed by a character that can appear in an identifier, you must enclose "relationship" or "supplier" in braces {}.

For example, if you set the Get Name property to:

```
${relationship}_get
```

the C++ code generator produces the following get member function for the has relationship:

```
// Get and Set Operations for Has Relationships
const B the_B_get() const;
```

You can control the case of the name derived from $relationship. This table describes the possible options (note that these options also can be used with $supplier, $targetClass and $target):

***Table 126 GetName Case Options***

| If you enter: | The action is: |
|---|---|
| ${relationship:l} | All characters in the relationship name are converted to lower case. |
| ${relationship:u} | All characters in the relationship name are converted to upper case. |
| ${relationship:f} | The case of the first character in the relationship name is inverted. |
| ${relationship:i} | The case of all characters in the relationship name is inverted. |

## GetResultIsConst

The GetResultIsConst property returns a `const` value if a get function is generated for this element.

The following table lists the values for GetResultIsConst:

***Table 127 GetResultIsConst Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | If a get function is generated for this item, it will return a const value. |
| False | If a get function is generated for this item, it will return a non-const value. |
| Same_As_Function | If a get function is generated for this item, it will return a const value if the function is const and a non-const value if the function is not const (as dictated by the Get Is Const property). |

## GetSetByReference

The GetSetByReference property specifies whether values in the get and set member functions are passed by reference or by value. By default, the C++ code generator produces get and set member functions for a has relationship to pass arguments and return values by value. If you want the get and set member functions to pass arguments and return values by reference, set GetSetByReference to True.

The following table lists the values for GetSetByReference. In this table, `T` is the name of the supplier class of the has relationship and `the_T` is the name of the has relationship:

***Table 128 GetSetByReference Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | **`T & get_the_T();`**<br>**`void set_the_T(const T &value);`** |
| False (Default) | **`T get_the_T();`**<br>**`void set_the_T(const T value);`** |

## GetSetKinds

The GetSetKinds property specifies the kind of member functions that are generated for the get and set operations for a data member. The C++ code generator produces additional keywords in the declarations of the get and set member functions based on the value of GetSetKinds, such as `static` or `virtual`.

The following table lists the values for GetSetKinds. In this table, `T` is the name of the supplier class in the has relationship and `the_T` is the name of the has relationship:

***Table 129 GetSetKind Values***

| If you select: | The C++ code generator produces: |
|---|---|
| Common (Default) | `T get_the_T();`<br>`void set_the_T(const T value);` |
| Virtual | `virtual T get_the_T();`<br>`virtual void set_the_T(const T value)` |
| Static | `static T get_the_T(A &client);`<br>`static void set_the_T(A &client, const T value);` |
| Abstract | `virtual T get_the_T() = 0;`<br>`virtual void set_the_T(const T value) = 0;` |
| Friend | `friend T get_the_T(A &client);`<br>`friend void set_the_T(A &client, const T value);` |

Note that if the has relationship itself is static, then the only legal values for GetSetKinds are "Common" and "Static." In both cases, the C++ code generator produces static member functions, such as:

```
static T get_the_T();
static void set_the_T(const T value);
```

## InitialValue

The InitialValue property specifies the initial value for the supplier of a Has relation. When the C++ code generator produces the declaration of the supplier, it also produces the initial value for the declaration.

The following table lists the values for InitialValue:

***Table 130 InitialValue Values***

| If you select: | The action is: |
|---|---|
| <string> | The C++ code generator produces a declaration for the has relation with <string> as its initial value. |
| <blank> | (Default) The C++ code generator produces a declaration for the has relation with no initial value. |

## InlineGet

The InlineGet property specifies whether the C++ code generator inlines the C++ code generator get operations.

***Table 131 InlineGet Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator inlines get operations. |
| False | The C++ code generator does not inline get operations. |

## InlineSet

The InlineSet property specifies whether the C++ code generator inlines the C++ code generator set operations.

***Table 132 InlineSet Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator inlines set operations. |
| False | The C++ code generator does not inline set operations. |

## NameIfUnlabeled

The NameIfUnlabeled property specifies the name to be used for an unlabeled has relationship. This property is not used if the relationship is labeled in Rational Rose or if a name is specified in the relationship's CodeName property. The C++ code generator uses the has relationship name to construct names for the corresponding data member and get and set member functions.

The default value of NameIfUnlabeled is:

```
the_$supplier
```

When the C++ code generator needs the name of the has relationship to generate a name for a data member or a get or set member function, $supplier expands to the name of the supplier class in the has relationship.

By default, the C++ code generator uses "the_B" as the name of the unlabeled has relationship shown in the following class diagram:

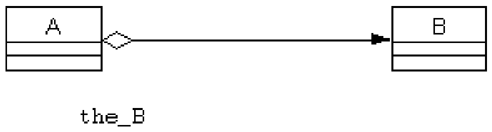


***Figure 60 Naming the Has Relationship***

You can change the form of the names the C++ code generator uses for unlabeled has relationships by changing the format of NameIfUnlabeled. Note that if $supplier is followed by a character that can appear in an identifier, you must enclose "supplier" in braces {}.

For example, if you set NameIfUnlabeled to:

```
${supplier}_rel
```

The C++ code generator uses "B_rel" as the name of the unlabeled has relationship.

You can control the case of the name derived from $supplier. This table describes the possible options:

***Table 133 NameIfUnlabled Case Options***

| If you enter: | The action is: |
|---|---|
| ${supplier:l} | All characters in the supplier class name are converted to lower case. |
| ${supplier:u} | All characters in the supplier class name are converted to upper case. |
| ${supplier:f} | The case of the first character in the supplier class name is inverted. |
| ${supplier:i} | The case of each character in the supplier class name is inverted. |

The $targetClass symbol is the same as the $supplier symbol.

***Note:*** *The* NameIfUnlabeled *property is always interpreted in the context of one of the roles of the association.*

## Ordered

The Ordered property specifies whether the objects on the supplier side are ordered.

This setting affects the choice of containers when generating code for has relationships. The type of the data member selected is based on the various combinations of supplier cardinality, containment, and order.

***Table 134 Ordered Property Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The objects in the supplier side of the has relationship are ordered. |
| (blank) | The objects in the supplier side of the has relationship are not ordered. |

## SelectorName

The SelectorName property allows you to implement direct get and set member functions for data members whose type is a container class. In particular, you use this property to specify a name for the selector argument of the generated get and set member functions. In addition, you need to specify a type for the selector argument in the SelectorType property.

Direct get and set member functions implement a direct method of accessing values in the container value of the relationship's data member.

- A direct get member function takes a selector as an argument and returns the corresponding element in the container.
- A direct set member function takes two arguments, a selector and a value, and sets the element identified by the selector to be the value.

By default, the C++ code generator produces indirect get and set member functions, which implement an indirect method of accessing values in a container class:

- The get member function returns a reference to or a copy of the entire container value of the relationship's data member. You then use member functions in the interface of the container class to modify the values in the copy.
- The set member function takes a complete container as its argument and replaces the contents of the entire container with the modified copy.

***Note:*** *You can set* GetSetByReference *to True to cause the get member function to return a reference to the container instead of a copy. You then use the container's member functions to modify the container's contents in-place.*

The following table lists the values for SelectorName:

***Table 135 SelectorName Values***

| If you enter: | The action is: |
|---|---|
| *literal* | The C++ code generator produces a selector argument for the get and set member functions with name *literal*. The type of the selector argument is determined by the value of SelectorType. For the set member function, the C++ code generator also generates a value argument whose name is "value" and whose type is the name of the supplier in the has relationship. |
| <blank> | (Default) The C++ code generator produces indirect get and set member functions for data members whose type is a container class. |

## SelectorType

The SelectorType property allows you to specify a corresponding type for the selector argument specified by SelectorName. You use SelectorName and SelectorType to implement direct get and set member functions for a data member whose type is a container class. To learn more about direct and indirect get and set member functions, see *SelectorName Property* in the online help.

If you specify a value for SelectorName but do not specify a value for SelectorType, the selector argument is generated with type `void`.

The following table lists the values for SelectorType:

***Table 136 SelectorType Values***

| If you enter: | The action is: |
|---|---|
| *literal* | The C++ code generator produces the selector argument for the get and set member functions with type *literal*. |
| <blank> | (Default) If a value has been specified for SelectorName, the C++ code generator produces the get and set member functions with a selector argument of type `void`. If SelectorName is also blank, the C++ code generator produces indirect get and set member functions. |

## Example of SelectorName and SelectorType

This example shows an association for which the C++ code generator produces a container class data member and the effect of SelectorName and SelectorType on the get and set member functions generated for the data member:

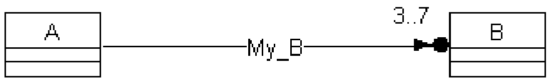


*Figure 61 Example of Selector Name and Selector Type*

By default, the C++ code generator produces the following data member for the association:

```
// Data Members for Associations
BoundedListByValue My_B<B,7>;
```

If no values are specified for SelectorName and SelectorType of the association, the C++ code generator produces indirect get and set member functions for the data member:

```
// Get and Set Operations for Associations
BoundedListByValue <B,7> get_My_B();
void set_My_B(const BoundedListByValue <B,7> value);
```

If SelectorName is set to "index" and SelectorType is set to "const int," the C++ code generator produces direct get and set member functions:

```
// Get and Set Operations for Associations
B get_My_B(const int index);
void set_My_B(const int index, const B value);
```

Note that in this case the implementation generated for the get and set member functions assume that the container class indicated by BoundedListByValue has a get function that takes a const int selector argument and a set function that takes a const int selector argument and a value argument:

```
// Get and Set Operations for Associations
B A::get_My_B(const int index)
{
//##begin A::get_My_B%.get preserve=no
   return My_B.get(index);
//##end A::get_My_B%.get
}
void A::set_My_B(const int index, const B value)
```

```
{
//##begin A::set_My_B%.set preserve=no
   My_B.set(index, value);
//##end A::set_My_B%.set
}
```

## SetName

The SetName property specifies the name the C++ code generator produces for a set member function for a has relationship. The default value is:

```
set_$relationship
```

When the C++ code generator produces a set member function for a data member, $relationship and $target expand to the label of the has relationship in the model. If the has relationship is unlabeled, $relationship and $target expand to the has relationship's NameIfUnlabeled value.

The following class diagram and code example illustrate a has relationship and the set member function that the C++ code generator produces for it by default:

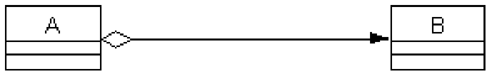


***Figure 62 Set Operations for Has Relationships***

```
void set_the_B(const B value);
```

You can change the form of the names that the C++ code generator produces for set member functions by changing the SetName format. You can also refer to $supplier and $targetClass in SetName. Note that if either $relationship, $target, $targetClass or $supplier is followed by a character that can appear in an identifier, you must enclose "relationship" or "supplier" in braces {}.

For example, if you set SetName to:

```
${relationship}_set
```

the C++ code generator produces the following set member function for the has relationship:

```
// Get and Set Operations for Has Relationships
```

```
void the_B_set(const B value);
```

You can control the case of the name derived from $relationship. This table describes the possible options.

***Note:*** *These options can also be used with $supplier, $target, and $targetClass.*

***Table 137 SetName Case Options***

| **If you enter:** | **The action is:** |
|---|---|
| ${relationship:l} | All characters in the relationship name are converted to lower case. |
| ${relationship:u} | All characters in the relationship name are converted to upper case. |
| ${relationship:f} | The case of the first character in the relationship name is inverted. |
| ${relationship:i} | The case of all characters in the relationship name is inverted. |

## SetReturnsValue

The SetReturnsValue property specifies whether the C++ code generator produces the set member function for a has relationship with a non-void return type. By default, the C++ code generator produces the set member function with return type `void`. However, sometimes it is convenient for the set member function to return the value to which the data member is set in the function.

The following table lists the values for SetReturnsValue. In this table, `T` is the name of the supplier class of the has relationship and `the_T` is the name of the has relationship:

***Table 138 SetReturnsValue Values***

| **If you select:** | **The C++ code generator produces:** |
|---|---|
| True | **`const T set_the_T(const T value);`** |
| False (Default) | **`void set_the_T(const T value);`** |

# *Appendix F*

# *Module Body and Module Specification Properties*

## AdditionalIncludes

The C++ code generator produces `#includes` in a file based on class relationships and module dependencies in your Rational Rose model. The AdditionalIncludes property specifies any additional `#include` directives to be generated in the implementation and header files. By default, AdditionalIncludes is empty.

The value of AdditionalIncludes must be a list of header files, one per line. Each file name should be enclosed by the delimiters `""` or `<>`, as in C++. Typically, you use quotes to specify user-defined header files that are in your inclusion search path. You use `<>` to specify system header files such as `<iostream.h>` and `<stddef.h>`. Consult your compiler manual for the search rules for your preprocessor. Conditional directives are not permitted.

The Rational Rose C++ code generator produces the additional `#include` directives in the form:

```
//Additional Includes:
#include "a.h"
#include <b.h>
...
```

A line beginning with `$include` is interpreted as a directive to insert the content of the file named after the `$include` symbol in the source.

## AllowExtensionlessFileName

The AllowExtensionlessFileName Module Body and Module Specification Properties specify whether the C++ code generator can produce an implementation file header whose full name does not have an extension. Normally, the C++ code generator uses the File Name property to create the name for the file.

If AllowExtensionlessFileName is set to False, the C++ code generator makes sure the specified file name has a proper extension. In this case, if the content of the FileName property does not have an extension, the C++ code generator appends an extension to that value to form a new file name.

The following table lists the values for AllowExtensionlessFileName:

***Table 139 AllowExtensionlessFileName Values***

| If you enter: | The action is: |
|---|---|
| True | The C++ code generator uses the value of FileName to create the file name, regardless of whether the file name has an extension. |
| False | (Default) The C++ code generator creates a file name with proper extension for the module specification or module body. |

## CmIdentification

The CmIdentification property specifies text that can be interpreted by a configuration management system to manage a file. The C++ code generator inserts the value of the CmIdentification property in the module annotation at the beginning of each implementation and header files.

The default value of the CmIdentification property is `"%X% %Q% %Z% %W%."` If you are using a configuration management system to manage your generated code files, change the value of CmIdentification property to a format that your configuration management system understands. If you are not using a configuration management system, you can change the CmIdentification property to any value or make it empty.

A line beginning with `$include` is interpreted as a directive to insert the content of the file named after the `$include` symbol in the source.

If $date is used in the CMIdentification property, it expands to the date the code was generated. If $time is used, it expands to the time the code was generated. If $module is used, it expands to the module name of the module. If $file is used, it expands to the fully qualified path of the module's file. If $relativeFile is used, it expands to the Apex view-independent name of the module's file. If $subsystem is used, it expands to the fully qualified name of the subsystem that contains the module. If $moduleKind is used, it expands to the specification or body. If $specificModuleKind is used, it expands to one of the following:

> Generic subprogram, Main program, Subprogram Specification, Subprogram Body, Package Specification, Package Body, Generic Package, Task Specification, Task Body, Pseudo Package Body, Pseudo Package Specification or Unknown.

## CopyrightNotice

The CopyrightNotice property is a text property that specifies copyright information for the generated implementation and header files. You can use the CopyrightNotice property to specify text that you want to place in every implementation and header file, such as copyright notices, project identification information, and so on. The C++ code generator inserts the value of the CopyrightNotice property in the annotation at the beginning of the implementation and header files. By default, the CopyrightNotice property is empty.

A line beginning with `$include` is interpreted as a directive to insert the content of the file named after the `$include` symbol in the source.

If $date is used in CopyrightNotice, it expands to the date the code was generated. If $time is used, it expands to the time the code was generated. If $module is used, it expands to the module name of the module. If $file is used, it expands to the fully qualified path of the module's file. If $relativeFile is used, it expands to the Apex view-independent name of the module's file. If $subsystem is used, it expands to the fully qualified name of the subsystem that contains the module. If $moduleKind is used, it expands to the specification or body. If $specificModuleKind is used, it expands to one of the following:

> Generic subprogram, Main program, Subprogram Specification, Subprogram Body, Package Specification, Package Body, Generic Package, Task Specification, Task Body, Pseudo Package Body, Pseudo Package Specification or Unknown.

## FileName

The FileName property specifies the file name for the implementation and header files that is generated for the module. The following table lists the possible values for FileName:

***Table 140    FileName Values***

| **If you enter:** | **The action is:** |
|---|---|
| Auto Generate | (Default) The C++ code generator produces a file name based on the name of the module. This name is the name of the module shortened to no more than the maximum number of characters permitted by the operating system, or 32, whichever is less. Shortening names can result in name conflicts. If this happens you must specify a name explicitly for all but one of the conflicting modules. |
| *literal* | The C++ code generator creates a file whose name is *literal.extension*, where *extension* is the file name extension specified by the module Implementation or HeaderFileExtensions property if one is not already specified. *literal* must be a valid file name. |
| <blank)> | The C++ code generator is unable to create a file and displays an error in the log. |

## Generate

The Generate property specifies whether the C++ code generator will generate a code file for the module.

This property allows you to prevent code from ever being generated for a module, such as modules in third party libraries, even if it is selected when the C++ code generator is invoked.

***Table 141    Generate Property Values***

| **If you enter:** | **The action is:** |
|---|---|
| True | (Default) The C++ code generator produces a code file for the module. |
| False | The C++ code generator does not produce a code file. |

## GenerateEmptyRegions

The GenerateEmptyRegions property identifies the type of protected region associated with the item.

The following table lists the possible values for GenerateEmptyRegions:

***Table 142 GenerateEmptyRegions Values***

| If you select: | The action is: |
|---|---|
| None | No empty protected region associated with the item is generated in the source file. |
| Preserved | An empty protected region associated with the item is generated only if it has a `preserve=yes` clause. |
| Unpreserved | An empty protected region associated with the item is generated only if it has a `preserve=no` clause. |
| All | An empty protected region associated with the item is always generated in the source file. |

## IncludeBySimpleName

The IncludeBySimpleName property controls whether the `#include` directives that the C++ code generator produces in the implementation and header files specify the path of the included files or just the file names.

If IncludeBySimpleName is False, the C++ code generator produces `#include` directives that specify the included file's path relative to the project directory.

For example, suppose a module "city" exists in a subsystem "county," which is nested in subsystem "state." The module's file name and the subsystem's directory properties have their default values (AUTO GENERATE). If IncludeBySimpleName is True, `highway.h` contains:
`#include "city.h"`

If IncludeBySimpleName is False, it contains:

```
#include "\state\county\city.h."
```

If IncludeBySimpleName is True, the C++ code generator produces `#include` directives with the included file names only. In this case, if your project generates code in multiple subdirectories, you must use a compilation option to tell the compiler to search for `#include` files in each subsystem. Consult your compiler manual for the search options for your compiler.

The following table lists the values for IncludeBySimpleName:

***Table 143 IncludeBySimpleName Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces `#include` directives with the file name only. |
| False | (Default) The C++ code generator produces `#include` directives with the path of the file, relative to the project directory. |

## IncludeClosure

The IncludeClosure property specifies the header files that are included (directly or indirectly) by the module. It is used by the code generator to determine if a given header file is in the closure of the pre-compiled header file.

The IncludeClosure property is computed by the Analyzer when it reverse-engineers the pre-compiled header and should not have to be modified by the user. It is computed only for the pre-compiled header.

The acceptable value for IncludeClosure is a list of source file names (including the path), one name per line.

## IncludeFormat

The IncludeFormat property specifies the source generated for `#include` directives.

Two symbols are available for substitution in the text: $package is replaced by the name of the module in the model and $file is replaced by the name of the file (in the format requested by IncludeBySimpleName for the module being included). The default value is:

```
// $package
#include “$file”
```

## IncludeOrder

Rational Rose C++ defines six regions that may contain `#include` directives. These regions are generated in the order specified by the IncludeOrder property. If the letter is not specified in IncludeOrder, the region is not generated. The regions are designated by letters, as follows:

`A` – `module.AdditionalIncludes` region

`I` – `module.Includes` region

`M` – AFX_INCLUDES group (Visual C++ only)

`H` – Header file

`P` – Pre-compiled header file

`R` – Rational Rose-generated `#include` directives

The `A`, `I`, and `M` regions are not modified by the code generator, but they are searched for `#include` directives and when the `R` region is generated, it will not duplicate any `#include` directives found in the other regions.

If `P` or `H` is omitted, the `#include` directives for the header and pre-compiled header, respectively, are placed at the start of the `#include` directives in the `R` section.

Note that even if `P` is specified in an IncludeOrder module property, the `#include` directives for the pre-compiled header are generated only if IncludePrecompiledHeader for the module is also True.

The possible values for IncludeOrder are any combination of at most one instance of each of the following letters: `A`, `I`, `M`, `H`, `P`, or `R`.

## IncludePrecompiledHeader

If IncludePrecompiledHeader is True and the module references something in the closure of the precompiled header, generate a `#include` for the pre-compiled header instead of a `#include` for the file that actually contains the referenced declaration.

If IncludePrecompiledHeader is False, and the module references something in the closure of the precompiled header, no `#include` is generated to satisfy the reference. (The precompiled header might be made visible to the compiler by some other method than `#include` directives.)

## InclusionProtectionSymbol (Module Spec Only)

The InclusionProtectionSymbol property specifies the symbol that the C++ code generator uses to generate preprocessor directives that prevent a header file from being included multiple times in another file.

The preprocessor directives are of the form:

```
# ifndef symbol
# define symbol
   ...#include directives...
   ...file contents...
# endif
```

where *`symbol`* is based on the value specified for InclusionProtectionSymbol. The following table lists the values for InclusionProtectionSymbol:

***Table 144 InclusionProtectionSymbol Values***

| **If you enter:** | **The action is:** |
|---|---|
| Auto Generate | (Default) The C++ code generator produces a symbol based on the name of the module. |
| *literal* | The C++ code generator produces the directives with *literal* as the symbol. |
| <blank> | The C++ code generator displays an error in the log. |

## InliningStyle

The InliningStyle property specifies the style of inlining the C++ code generator produces. The following table lists the values for InliningStyle:

***Table 145 InliningStyle Values***

| If you enter: | The action is: |
|---|---|
| In Class Declaration | The C++ code generator produces definitions of inlined member functions directly in the class header. |
| Following Class Declaration | (Default) The C++ code generator produces declarations of inline member functions in the class header. The definitions of these inline member functions are produced following the class header in the same file. The code is generated with the `inline` keyword. |

## TypesDefined

The TypesDefined property specifies the types defined in a module. If a module references a type, but that type is not assigned to a module, the code generator scans these lists of defined types to identify the module to `#include` into the referencing module.

The DefinedTypes property is computed by the Analyzer when it reverse-engineers C++ source code and should not have to be modified by the user.

The acceptable values for TypesDefined is a list of type names, one name per line.

*Appendix G*

# *Operation Properties*

## BodyAnnotations

The BodyAnnotations property allows annotations to be placed in the body. It specifies the annotations that are generated from the implementation (body) of the operation rather than from the specification. Any annotations not specified in BodyAnnotations are generated from the specification.

The BodyAnnotations format is a list of keywords separated by semicolons. The keywords are drawn from the following list, which correspond to the text boxes on the operation specification:

> Documentation, Concurrency, Space Complexity, Time Complexity, Qualification, Exceptions, Preconditions, Postconditions, and Semantics.

Only enough letters of a keyword to uniquely identify it are required. These letters are underlined in the above list.

## CodeName

The CodeName property specifies the name for the operation in the generated code.

You need to set this property only if you want the class to be named differently than it is in the Rational Rose model. This is especially useful when the Rational Rose model and code are expressed in different natural languages. The value of this property should be a valid C++ identifier. If it is not a valid C++ identifier, the C++ code generator performs the following substitutions:

- White space characters and hyphens are changed to underscores
- All other characters except letters, digits, and underscores are changed to "X"
- If the first character is a number, an "N" precedes the number

***Table 146 CodeName Values***

| If you select: | The action is: |
|---|---|
| C++ Identifier | The operation is assigned the name of the identifier. |
| <blank> | (Default) The C++ generator produces an operation name from the model. |

## EntryCode

The EntryCode property is a text property that specifies additional code or comments for a user-defined operation. You use EntryCode to specify code that you want to insert in a number of operations, such as instrumentation code. By default, EntryCode is empty.

The C++ code generator inserts the contents of the Entry Code property in the member function body before the source markers for the preserved code region. Note that the C++ code generator does not check the correctness of the code specified in EntryCode.

To learn more about the implementation code that is generated for user-defined operations, see *Implementation Code Generated for Operations* in the online help.

## ExitCode

The ExitCode property is a text property that specifies additional code or comments for a user-defined operation. You use ExitCode to specify code that you want to insert in a number of operations, such as instrumentation code. By default, ExitCode is empty.

The C++ code generator inserts the contents of ExitCode in the member function body following the source markers for the preserved code region. Note that the C++ code generator does not check the correctness of the code you specify in ExitCode. You must ensure that the flow of control passes through the exit code when appropriate.

## GenerateEmptyRegions

The GenerateEmptyRegions property identifies the type of protected region associated with the item.

The following table lists the values for GenerateEmptyRegions:

***Table 147 GenerateEmptyRegions Values***

| If you select: | The action is: |
|---|---|
| None | No empty protected region associated with the item is generated in the source file. |
| Preserved | An empty protected region associated with the item is generated only if it has a `preserve=yes` clause. |
| Unpreserved | An empty protected region associated with the item is generated only if it has a `preserve=no` clause. |
| All | An empty protected region associated with the item is always generated in the source file. |

## Inline

The Inline property is a boolean property that specifies whether to inline an operation.

***Table 148 Inline Property Values***

| If you enter: | The action is: |
|---|---|
| True | The C++ code generator inlines the operation. |
| False | (Default) The C++ code generator does not inline the operation. |

## OperationIsConst

TheOperationIsConst property specifies whether the C++ code generator produces the member function for a user-defined operation with the `const` keyword. `const` member functions cannot modify class data members.

The following table lists the values for OperationIsConst. In this table, `result` is the return type of the member function, `fname` is the name of the member function, and `params` is the formal parameter list:

***Table 149 OperationIsConst Values***

| If you select: | The C++ Code Generator produces: |
|---|---|
| True | **`result fname (params) const;`** |
| False (Default) | **`result fname (params);`** |

## OperationIsExplicit

The OperationIsExplicit property identifies the `explicit` keyword used to prefix the declaration of this operation. This option is valid only for operations that are constructors.

The following table lists the values for OperationIsExplicit:

***Table 150 OperationIsExplicit Values***

| If you select: | The action is: |
|---|---|
| True | The `explicit` keyword prefaces the declaration of this operation. |
| False | The explicit keyword is not used to prefix the declaration of this operation. |

## OperationKind

The OperationKind property specifies the kind of member function that is generated for a user-defined operation. The C++ code generator produces additional keywords in the declaration of the member function based on the value of OperationKind, such as `static` or `virtual`.

The following table lists the values for OperationKind. In this table, `result` is the result type of the member function, `fname` is the name of the member function, and `params` is the formal parameter list:

***Table 151 OperationKind Values***

| If you select: | The C++ Code Generator produces: |
|---|---|
| Common (Default) | **`result fname (params);`** |
| Virtual | **`virtual result fname (params);`** |
| Static | **`static result fname (params);`** |
| Abstract | **`virtual result fname (params) = 0;`** |
|  | **`Friend friend result fname (params);`** |

# *Appendix H*

# *Project Properties*

## AllowExplicitInstantiations

The AllowExplicitInstantiations property specifies whether the C++ code generator produces an explicit instantiation of a parameterized class. Note that some C++ compilers (for example, Visual C++ 6.0) may not properly handle this kind of instantiation.

The following table lists the values for AllowExplicitInstantiations:

***Table 152 AllowExplicitInstantiations Property Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces explicit instantiation of a parameterized class. |
| False | (Default) The C++ code generator does not produce explicit instantiation of a parameterized class. |

## AllowProtectedInheritance

The AllowProtectedInheritance property controls whether the C++ code generator produces protected derivation for classes with protected *inherits* relationships in the Rational Rose model. This is significant because some C++ compilers do not support protected inheritance.

If the AllowProtectedInheritance property is False, the C++ code generator produces protected inheritance relationships as public derivation.

The following table lists the values for AllowProtectedInheritance:

***Table 153 AllowProtectedInheritance Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces protected inheritance relationships as protected derivation. |
| False | (Default) The C++ code generator produces protected inheritance relationships as public derivation. |

## AllowTemplates

The AllowTemplates property controls whether the C++ code generator produces templates for parameterized classes. This is significant because Version 2.1 C++ compilers do not support templates.

The following table lists the values for AllowTemplates:

***Table 154 AllowTemplates Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces templates for parameterized classes. |
| False | The C++ code generator displays an error message in the log if you try to generate code for a parameterized or instantiated class or class utility. |

## AlwaysKeepOrphanedCode

The AlwaysKeepOrphanedCode property specifies whether the C++ code generator moves the orphaned code into a preserved region. A preserved region is a protected region whose "preserve" option is set to Yes. Normally, the C++ code generator moves the orphaned code into a non-preserved region that may be removed in the subsequent code generations.

The following table lists the values for AlwaysKeepOrphanedCode:

***Table 155 The AlwaysKeepOrphanedCode Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator always moves the orphaned code into a preserved region. |
| False | (Default) The C++ code generator moves the orphaned code into a non-preserved region. |

## BooleanType

The BooleanType property specifies the data type that you want the C++ code generator to generate as a result type for member functions that return Boolean values, such as equality and relational operations. The default value for BooleanType property is `int`.

## BoundedByReferenceContainer

The BoundedByReferenceContainer property indicates the default container class that the C++ code generator uses to generate a data member for a bounded has by reference relationship or a navigable bounded has by reference association. The value of this project property is used only if the ContainerClass property of the association or has relationship is empty.

The default value of the BoundedByReferenceContainer property is:

BoundedListByReference<$targetClass,$limit>

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The C++ code generator does not provide an implemented container class for bounded by reference relationships. However, you can either set the BoundedByReferenceContainer property to refer to a container class of your own, or you can provide an implementation for the default container class (BoundedListByReference).

### Example of BoundedByReferenceContainer

This is an example of a bounded by reference association and the data member that is generated for it by default, using Rational Rose default project properties:

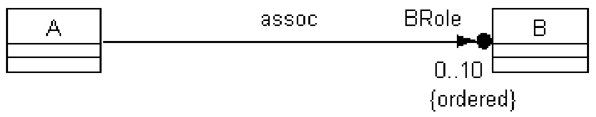


***Figure 63 Example of a Bounded By Reference Association***

The Rational Rose C++ code generator produces the data member as an instance of a template:

```
// Data Members for Associations
...
BoundedListByReference<B,10> BRole;
...
```

## BoundedByValueContainer

The BoundedByValueContainer property indicates the default container class that the C++ code generator uses to generate a data member for a bounded has by value relationship or a navigable bounded has by value association. The value of this project property is used only if the ContainerClass property of the association or has relationship is empty.

The default value of the BoundedByValueContainer property is:

BoundedListByValue<$targetClass,$limit>

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The C++ code generator does not provide an implemented container class for bounded by value relationships. However, you can either set the BoundedByValueContainer property to refer to a container class of your own, or you can provide an implementation for the default container class (BoundedListByValue).

### Example of BoundedByValueContainer

This is an example of a bounded by value association and the data member that is generated for it by default, using Rational Rose default project properties:

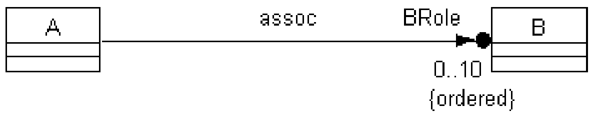


***Figure 64 Example of a Bounded By Value Association***

The Rational Rose C++ code generator produces the data member as an instance of a template:

```
// Data Members for Associations
...
BoundedListByValue<B,10> BRole;
...
```

## CodeFileBackupExtension

If the C++ code generator produces an implementation file that already exists, the previous version of the file is renamed to a backup file. The CodeFileBackupExtension property specifies the file name extension that the C++ code generator uses when creating implementation backup files. The default value of the CodeFileBackupExtension property is "**cp~**".

The following table lists the values for CodeFileBackupExtension:

***Table 156 CodeFileBackupExtension Values***

| **If you enter:** | **The action is:** |
|---|---|
| cp~ | (Default) The C++ code generator creates code backup files with the extension "**.cp~**". |
| *literal* | The C++ code generator creates code backup files with the extension ".*literal*". *literal* must be a valid file name extension, without the "**.**" Use caution when working with literal file extensions to insure that unique file names are assigned to the code files, header files, inline files, and backup files. |

## CodeFileExtension

The CodeFileExtension property specifies the file name extension that the C++ code generator uses when creating implementation files. The default value of the CodeFileExtension property is `cpp`.

The following table lists the values for the CodeFileExtension:

***Table 157 CodeFileExtension Values***

| If you enter: | The action is: |
|---|---|
| cpp | (Default) The C++ code generator creates code files with the extension `.cpp`. |
| *literal* | The C++ code generator creates code files with the extension ".*literal*". *literal* must be a valid file name extension, without the "**.**" Use caution when working with literal file extensions to insure that unique file names are assigned to the code files, header files, inline files, and backup files. |

## CodeFileTemporaryExtension

When the C++ code generator writes a code file, it actually writes the code to a temporary file. Once the code is completely written, the following steps are taken:

- The backup file (see CodeFileBackupExtension) is deleted, if there is one.
- The existing code file is renamed to the backup file, assuming an existing code file is present.
- The temporary file is renamed to be the new code file.

The CodeFileTemporaryExtension specifies the file name extension that the C++ code generator uses when creating temporary code files. The default value is `.c#`.

The following table lists the values for CodeFileTemporaryExtension:

***Table 158 CodeFileTemporaryExtension Values***

| If you enter: | The action is: |
|---|---|
| c# | (Default) The C++ code generator creates temporary code files with the extension `.c#`. |
| *literal* | The C++ code generator creates temporary code files with the extension ".*literal*". *literal* must be a valid file name extension, without the "**.**" Use caution when working with literal file extensions to insure that unique file names are assigned to the code files, header files, and backup files. |

## CommentWidth

The CommentWidth property specifies the maximum number of characters per line in generated comments. Lines in the model that are longer than this are split at word boundaries into multiple lines. The default Comment Width is 60.

## CreateMissingDirectories

By default, each package in a Rational Rose model is stored in a separate subdirectory of the project directory, as specified by the Directory project property. The CreateMissingDirectories property indicates whether or not the C++ code generator should create missing directories as it generates code. The C++ code generator can create directories only if you have proper access to the project directory and any parent directories that already exist.

The following table lists the values for CreateMissingDirectories:

***Table 159 CreateMissingDirectories Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator creates any needed directories automatically. |
| False | The C++ code generator does not create directories. The C++ code generator produces an error if a needed directory is missing. |

## Directory

The Directory property specifies the project directory, which is the directory in which all subdirectories and files for a project are generated. You can set this project property to an absolute or relative path.

If the model containing this project property is supporting multiuser development and thus must contain only relative paths, use virtual symbols from your Path Map to construct the directory path.

The following table lists the values for Directory:

***Table 160 Directory Property Values***

| If you enter: | The action is: |
|---|---|
| Auto Generate | (Default) The C++ code generator uses the current working directory as the project directory. |
| *literal* | The C++ code generator produces a directory with the name *literal*. *literal* must be a valid file system directory name. *literal* may be a relative or an absolute path; if *literal* is a relative path, the C++ code generator assumes that it is relative to the current working directory. The path specified by *literal* may include virtual path symbols. |
| (blank) | The C++ code generator uses the root directory as the project directory. |

## ErrorLimit

The ErrorLimit property indicates whether or not the C++ code generator stops generating code when it encounters the specified error limit. The following table lists the values for ErrorLimit:

***Table 161 ErrorLimit Values***

| If you select: | The action is: |
|---|---|
| *number* | If the C++ code generator detects an error while generating code for a module, it displays the error in the log and continues. Once the error limit is reached, the C++ code generator stops generating code immediately. The default value is 30. |

## FileNameFormat

The FileNameFormat property controls the automatic generation of directory and file names when the value of the Directory project property, a Directory subsystem property, or a module FileName property is “Auto Generate.”

The value is expected to be an integer followed by zero or more flag characters. The integer is the maximum number of characters in a file or directory name. The flags are:

***Table 162 FileName Format Flags***

| Flag: | The action is: |
|---|---|
| _ | retain underscores |
| v | retain vowels |
| u | convert all letters to upper case |
| l | convert all letters to lower case |
| x | retain case |

The default, if the value is blank, is to limit the length to 128 characters, retain vowels, eliminate white space, and eliminate underscores. When a blank or underscore is eliminated, the next character is capitalized. So, for example, if a project specification is named my_strange module, the file name for it is myStrangeModule.h.

## FixedByReferenceContainer

The FixedByReferenceContainer property indicates the default container class that the C++ code generator uses to generate a data member for a fixed has by reference relationship or a navigable fixed has by reference association. The value of this project property is used only if the ContainerClass property of the has relationship is empty.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The C++ code generator does not provide an implemented container class for fixed by reference relationships. However, you can set FixedByReferenceContainer to refer to a container class of your own. You can use $supplier, $limit, $target, and $targetClass variables when you provide your own value for this project property.

By default, FixedByReferenceContainer is empty, which causes the C++ code generator to generate the data member as an array of pointers to the supplier class T:

```
T*[5]
```

### Example of FixedByReferenceContainer

This is an example of a fixed by reference association and the data member that is generated for it by default:

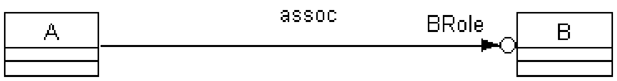


***Figure 65 Example of a Fixed By Reference Association***

The C++ code generator produces the following data member:

```
// Data Members for Has Relationships
...
b *my_b[4];
...
```

## FixedByValueContainer

The FixedByValueContainer property indicates the default container class that the C++ code generator uses to generate a data member for an ordered fixed has by value relationship or a navigable fixed has by value association. The value of this project property is used only if the ContainerClass property of the *has* relationship is empty.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The C++ code generator does not provide an implemented container class for fixed by value relationships. However, you can set FixedByValueContainer to refer to a container class of your own. You can use $supplier, $limit, $target, and $targetClass variables when you provide your own value for this project property.

By default, FixedByValueContainer is empty, which causes the C++ code generator to generate a data member which is an array of the supplier class T:

```
T[5]
```

## Example of FixedByValueContainer

This is an example of a fixed by value association and the data member that is generated for it by default:

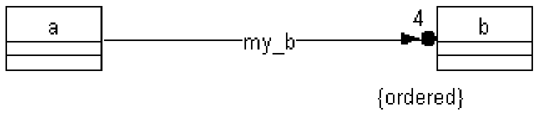


***Figure 66 Example of a Fixed By Value Association***

The C++ code generator produces the following data member:

```
// Data Members for Has Relationships
...
b my_b[4];
...
```

## HeaderFileBackupExtension

If the C++ code generator writes a header file that already exists, the previous version of the file is renamed to a backup file. The HeaderFileBackupExtension property specifies the file name extension that the C++ code generator uses when creating header backup files. The default value is "**h~**".

The following table lists the values for HeaderFileBackupExtension:

***Table 163 HeaderFileBackupExtension Values***

| If you enter: | The action is: |
|---|---|
| h~ | (Default) The C++ code generator creates header backup files with the extension "**.h~**". |
| *literal* | The C++ code generator creates header backup files with the extension ".*literal*". *literal* must be a valid file name extension, without the "**.**" Use caution when working with literal file extensions to insure that unique file names are assigned to the code files, header files, and backup files. |

## HeaderFileExtension

The HeaderFileExtension property specifies the file name extension that the C++ code generator uses when creating header files. The default value is "**h**".

The following table lists the values for HeaderFileExtension:

***Table 164 HeaderFileExtension Values***

| If you enter: | The action is: |
|---|---|
| h | (Default) The C++ code generator creates header files with the extension `.h`. |
| *literal* | The C++ code generator creates header files with the extension ".*literal*". *literal* must be a valid file name extension, without the "**.**" Use caution when working with literal file extensions to insure that unique file names are assigned to the code files, header files, inline files, and backup files. This caution need not apply if the file is conditionally coded for Unix. |

## HeaderFileTemporaryExtension

When the C++ code generator writes a header file, it actually writes the code to a temporary file. Once the code is completely written, the following steps are taken:

- The backup file (see the HeaderFileBackup Extension property) is deleted, if there is one.
- The existing header file is renamed to the backup file, assuming an existing header file is present.
- The temporary file is renamed to be the new header file.

The HeaderFileTemporaryExtension specifies the file name extension that the C++ code generator uses when creating temporary header files. The default value is `.h#`.

The following table lists the values for HeaderFileTemporaryExtension:

***Table 165 HeaderFileTemporaryExtension Property Values***

| If you enter: | The action is: |
|---|---|
| h# | (Default) The C++ code generator creates temporary header files with the extension `.h#`. |
| *literal* | The C++ code generator creates temporary header files with the extension ".*literal*". *literal* must be a valid file name extension, without the "." Use caution when working with literal file extensions to insure that unique file names are assigned to the code files, header files, and backup files. |

## OneByReferenceContainer

The OneByReferenceContainer property indicates the default container class that the C++ code generator uses to generate a data member for a one has by reference relationship or a navigable one has by reference association. The value of this project property is used only if the ContainerClass property of the association or has relationship is empty.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The default value of OneByReferenceContainer is simple containment of a target object:

```
$targetClass
```

The C++ code generator does not provide an implemented container class for one by reference relationships and one is usually not necessary in this case. However, you can set the One By Reference Container property to refer to a container class of your own.

### Example of OneByReferenceContainer

This is an example of an one by reference association and the data member that is generated for it by default, using Rational Rose default project properties:

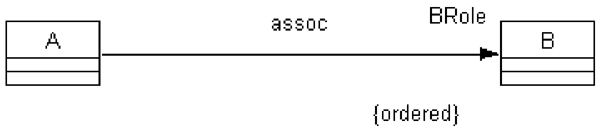


***Figure 67 Example of a One By Reference Association***

The Rational Rose C++ code generator produces the following data member:

```
// Data Members for Associations
...
B *BRole;
...
```

## OneByValueContainer

The OneByValueContainer property indicates the default container that the C++ code generator uses to generate a data member for a one has by value relationship or a navigable one has by value association. The value of this project property is used only if the ContainerClass property of the association or has relationship is empty.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The default value of OneByValueContainer is simple containment of a target object:

    $targetClass

When the C++ code generator produces the data member, $targetClass expands to the name of the supplier class in the *association* or *has* relationship. When this project property is empty, the generated data member has a void data type.

The C++ code generator does not provide an implemented container class for one by value relationships and one is not usually necessary in this case. However, you can set OneByValueContainer to refer to a container class of your own.

## Example of OneByValueContainer

This is an example of a one by value association and the data member that is generated for it by default:

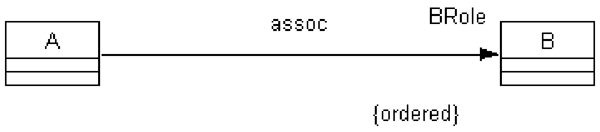


***Figure 68 Example of a One By Value Association***

The C++ code generator produces the following data member:

```
// Data Members for Associations
...
B BRole;
...
```

## OptionalByReferenceContainer

The OptionalByReferenceContainer property indicates the default container class that the C++ code generator uses to generate a data member for an optional has by reference relationship or a navigable optional has by reference association. The value of this project property is used only if the ContainerClass property of the association or has relationship is empty.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The default value of OptionalByReferenceContainer is:

OptionalByReference<$targetClass>

When the C++ code generator produces the data member, this project property is empty. $targetClass expands to the name of the supplier class when there is no association class. However, when an association class is present, $targetClass expands to the name of the association class. When this project property is empty, the generated data member has a void data type.

The C++ code generator does not provide an implemented container class for optional by reference relationships. However, you can either set OptionalByReferenceContainer to refer to a container class of your own, or you can provide an implementation for the default container class (optional by reference).

## Example of OptionalByReference Container

This is an example of an optional by reference association and the data member that is generated for it by default, using Rational Rose default project properties:

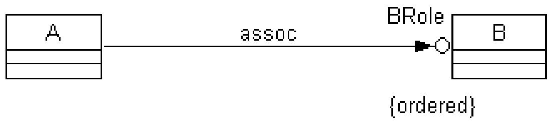


***Figure 69 Example of an Optional By Reference Association***

The Rational Rose C++ code generator produces the following data member:

```
// Data Members for Associations
...
B *BRole;
...
```

## OptionalByValueContainer

The OptionalByValueContainer property indicates the default container class that the C++ code generator uses to generate a data member for an optional has by value relationship or a navigable optional has by value association. The value of this project property is used only if the ContainerClass property of the *association* or *has* relationship is empty.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The default value of the OptionalByValueContainer property is:

```
OptionalByValue<$targetClass>
```

When the C++ code generator produces the data member, $targetClass expands to the name of the supplier class when there is no association class. However, when an association class is present, $targetClass expands to the name of the association class. When this project property is empty, the generated data member has a void data type.

The C++ code generator does not provide an implemented container class for optional by value relationships. However, you can either set OptionalByValueContainer to refer to a container class of your own, or you can provide an implementation for the default container class (optional by value).

### Example of OptionalByValueContainer

This is an example of an optional by value association and the data member that is generated for it by default, using Rational Rose default project properties:

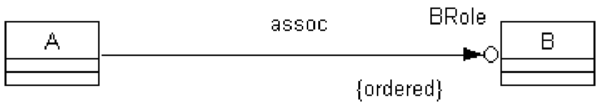


***Figure 70   Example of an Optional By Value Association***

The Rational Rose C++ code generator produces the following data member:

```
// Data Members for Associations or Has Relationships
...
OptionalByValue<B> BRole;
...
```

## PathSeparator

By default, the C++ code generator generates path separators in `#include` directives as preferred by the operating system. Windows uses a backslash or a slash as the path separator. Unix uses a slash as the path separator.

The following table lists the values for PathSeparator:

***Table 166 PathSeparator Values***

| If you enter: | The action is: |
|---|---|
| <blank> | (Default) The C++ code generator generates path separators in `#include` directives as preferred by the operating system. |
| *literal* | The C++ code generator uses the value to separate path elements in `#include` directives. This is especially useful when you specify a "/" which causes the C++ code generator to generate `#include` directives with slashes, even under Windows. This is helpful if you need to generate platform-independent code. |

## PrecompiledHeader

The PrecompiledHeader property identifies the header file that represents a pre-compiled header (or any commonly included file). Any module that needs visibility to a definition in the `#include` closure of this header file, will `#include` the pre-compiled header file rather than the file that defines the needed type.

The possible value of PrecompiledHeader is the simple name of the pre-compiled header file. The name of the pre-compiled header file is saved in each project.

The pre-compiled header file can be changed in a simple dialog accessible from the PrecompiledHeader option on the **Edit** menu, or **CTRL-H**.

## QualifiedByReferenceContainer

The QualifiedByReferenceContainer property indicates the default container class that the C++ code generator uses to generate a data member for a navigable qualified has by reference association. The value of this project property is used only if the ContainerClass property of the association is empty.

The default value of QualifiedByReferenceContainer is:

```
AssociationByReference<$qualtype,$qualcont>
```

An association is an ordered list of (key,value) pairs that can be efficiently accessed by key value.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

You can use $qualtype and $qualcont when you provide your own value for this project property. When this project property is empty, the generated data member has a void data type.

The C++ code generator does not provide an implemented container class for qualified by reference relationships. However, you can either set QualifiedByReferenceContainer to refer to your own container class, or you can provide an implementation for the default container class (association by reference).

## Example of QualifiedByReferenceContainer

This is an example of a qualified by reference association and the data member that is generated for it by default, using Rational Rose default project properties:

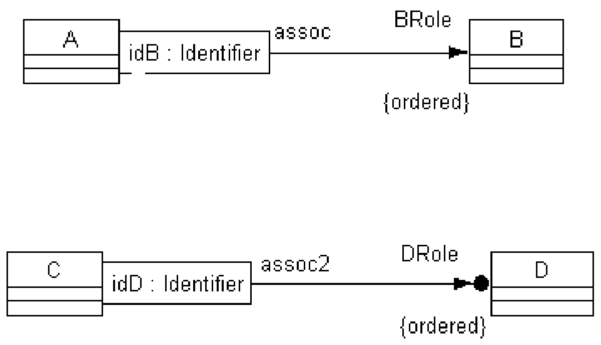


***Figure 71   Example of a Qualified By Reference Container***

The Rational Rose C++ code generator produces the data member as an instance of a template:

```
// Data Members for Associations
...
AssociationByReference<Identifier,B*> BRole;
...
AssociationByReference<Identifier,UnboundedSetByReference<D>
DRole;
```

## QualifiedByValueContainer

The QualifiedByValueContainer property indicates the default container class that the C++ code generator uses to generate a data member for a navigable, singly qualified has by value association. The value of this project property is used only if the ContainerClass property of the *association* is empty.

The default value of QualifiedByValueContainer is:

```
AssociationByValue<$qualtype, $qualcont>
```

An association is an ordered list of (key,value) pairs that can be efficiently accessed by key value.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The C++ code generator does not provide an implemented container class for qualified by value relationships. However, you can either set QualifiedByValueContainer to refer to a container class of your own, or you can provide an implementation for the default container class (association by value).

## Example of QualifiedByValueContainer

This is an example of a qualified by value association and the data member that is generated for it by default, using Rational Rose default project properties:

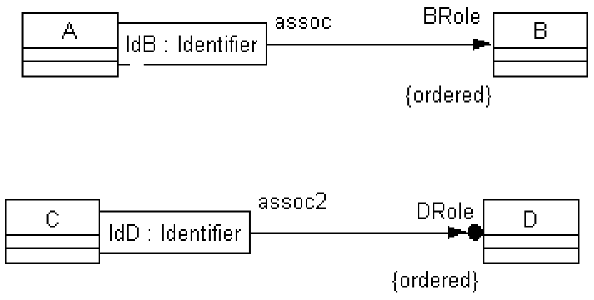


***Figure 72 Example of a Qualified By Value Association***

The Rational Rose C++ code generator produces the data member as an instance of a template:

```
// Data Members for Associations
...
AssociationByValue<Identifier,B> BRole;
...
AssociationByValue<Identifier, UnboundedSetByValue<B> >
DRole;
```

## StopOnError

The StopOnError property indicates whether or not the C++ code generator stops generating code when it encounters an error. This project property applies when you select multiple classes, modules, logical packages, or component packages and then initiate code generation. The following table lists the values for StopOnError:

***Table 167    StopOnError Values***

| If you select: | The action is: |
|---|---|
| True | If the C++ code generator detects an error while generating code for a module, it displays the error in the log and stops immediately. |
| False | (Default) If the C++ code generator detects errors while generating code for a module, it displays the errors in the log and continues. |

***Note:*** *This project property is obsolete and will be removed in a future release. We recommend that StopOnError be left at its default value and that you use the ErrorLimit property instead.*

## UnboundedByReferenceContainer

The UnboundedByReferenceContainer property indicates the default container class that the C++ code generator uses to generate a data member for an unbounded has by reference relationship or a navigable unbounded has by reference association. The value of this project property is used only if the ContainerClass property of the association or has relationship is empty.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The default value of UnboundedByReferenceContainer is:

    UnboundedListByReference<$targetClass>

You can use $targetClass when you provide your own value for this project property. When this project property is empty, the generated data member has a void data type.

The C++ code generator does not provide an implemented container class for unbounded by reference relationships. However, you can either set UnboundedByReferenceContainer to refer to your own container class, or you can provide an implementation for the default container class (unbounded list by reference).

### Example of UnboundedByReferenceContainer

This is an example of an unbounded by reference association and the data member that is generated for it by default, using Rational Rose default project properties:

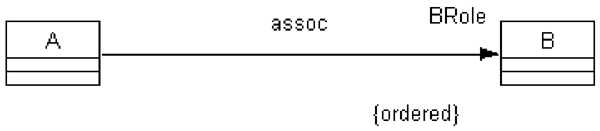


***Figure 73 Example of an Unbounded By Reference Association***

The Rational Rose C++ code generator produces the data member as an instance of a template:

```
// Data Members for Associations
...
UnboundedSetByReference<B> BRole;
...
```

## UnboundedByValueContainer

The UnboundedByValueContainer property indicates the default container class that the C++ code generator uses to generate a data member for an unbounded has by value relationship or a navigable unbounded has by value association. The value of this project property is used only if the ContainerClass property of the association or has relationship is empty.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The default value of UnboundedByValueContainer is:

UnboundedListByValue<$targetClass>

When the C++ code generator produces the data member, $targetClass expands to the name of the association class if one exists. If there is not an association class, it expands to the name of the supplier class. When this project property is empty, the generated data member has a void data type.

The C++ code generator does not provide an implemented container class for unbounded by value relationships. However, you can either set the UnboundedByValueContainer property to refer to a container class of your own, or you can provide an implementation for the default container class (unbounded list by value).

### Example of UnboundedByValueContainer

This is an example of an unbounded by value association and the data member that is generated for it by default, using Rational Rose default project properties:

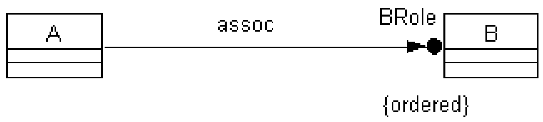


***Figure 74   Example of an Unbounded By Value Association***

The Rational Rose C++ code generator produces the data member as an instance of a template:

```
// Data Members for Associations
...
UnboundedListByValue<B> BRole;
...
```

## UnorderedBoundedByReferenceContainer

The UnorderedBoundedByReferenceContainer property indicates the default container class that the C++ code generator uses to generate a data member for an unordered bounded has by reference relationship or a navigable unordered bounded has by reference association. The value of this project property is used only if the ContainerClass property of the association or has relationship is empty.

The default value of UnorderedBoundedByReferenceContainer is:

    BoundedSetByReference<$targetClass,$limit>

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The C++ code generator does not provide an implemented container class for unordered bounded by reference relationships. However, you can either set the UnorderedBoundedByReferenceContainer property to refer to a container class of your own, or you can provide an implementation for the default container class (bounded set by reference).

### Example of UnorderedBoundedByReferenceContainer

This is an example of an unordered bounded by reference association and the data member that is generated for it by default:

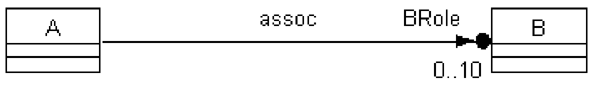


***Figure 75   Example of an Unordered Bounded By Reference Association***

The C++ code generator produces the following data member:

```
// Data Members for Associations
...
BoundedSetByReference<B,10> BRole;
...
```

## UnorderedBoundedByValueContainer

The UnorderedBoundedByValueContainer property indicates the default container class that the C++ code generator uses to generate a data member for an unordered bounded has by value relationship or a navigable unordered bounded has by value association. The value of this project property is used only if the ContainerClass property of the association or has relationship is empty.

The default value of UnorderedBoundedByValueContainer is:

BoundedSetByValue<$targetClass,$limit>

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The C++ code generator does not provide an implemented container class for unordered bounded by value relationships. However, you can either set UnorderedBoundedByValueContainer to refer to a container class of your own, or you can provide an implementation for the default container class (bounded set by value).

### Example of UnorderedBoundedByValueContainer

This is an example of an unordered bounded by value association and the data member that is generated for it by default:

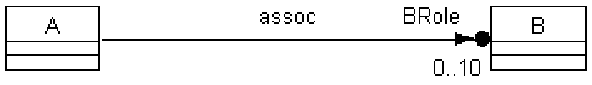


***Figure 76 Example of an Unordered Bounded By Value***

The C++ code generator produces the following data member:

```
// Data Members for Associations
...
BoundedSetByValue<B,10> BRole;
...
```

## UnorderedFixedByReferenceContainer

The UnorderedFixedByReferenceContainer property indicates the default container class that the C++ code generator uses to generate a data member for an unordered fixed has by reference relationship or an unordered association role. The value of this project property is used only if the ContainerClass property of the has relationship or association is empty.

The default value of this project property is:

$targetClass *[$limit]

which implements the data member as an array of pointers.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The C++ code generator does not provide a container class for unordered fixed by reference relationships. However, you can set UnorderedFixedByReferenceContainer to refer to a container class of your own. You can use $supplier and $limit variables when you provide your own value for this property.

### Example of UnorderedFixedByReferenceContainer

This is an example of an unordered fixed by reference association and the data member that is generated for it by default:

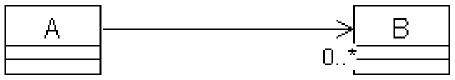


***Figure 77 Example of an Unordered Fixed By Reference Association***

The C++ code generator produces the following data member:

```
// Data Members for Associations
...
B *BRole [1];
...
```

## UnorderedFixedByValueContainer

The UnorderedFixedByValueContainer property indicates the default container class that the C++ code generator uses to generate a data member for an unordered fixed has by value relationship or a navigable unordered fixed has by value association. The value of this project property is used only if the ContainerClass property of the *has* relationship is empty.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The C++ code generator does not provide an implemented container class for unordered fixed by value relationships. However, you can set UnorderedFixedByValueContainer to refer to a container class of your own. You can use $supplier, $limit, $target and $targetClass variables when you provide your own value for this project property.

By default, UnorderedFixedByValueContainer is empty, which causes the C++ code generator to generate a data member as an array of the supplier class `T`:

```
T[5]
```

## Example of UnorderedFixedByValueContainer

This is an example of an unordered fixed by value association and the data member that is generated for it by default:

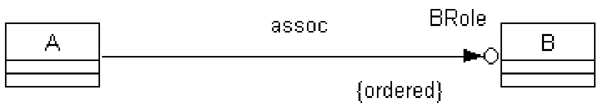


***Figure 78 Example of an Unordered Fixed By Value Association***

The C++ code generator produces the following data member:

```
// Data Members for Associations
...
B BRole [1];
...
```

## UnorderedQualifiedByReferenceContainer

The UnorderedQualifiedByReferenceContainer property indicates the default container class that the C++ code generator uses to generate a data member for a navigable unordered qualified has by reference association. The value of this project property is used only if the ContainerClass property of the association is empty.

The default value of UnorderedQualifiedByReferenceContainer is:

DictionaryByReference<$qualtype, $qualcont>

A dictionary is an unordered set of (key,value) pairs that can be efficiently accessed by key value.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

You can use $qualtype and $qualcont when you provide your own value for this project property. When this project property is empty, the generated data member has a void data type.

The C++ code generator does not provide an implemented container class for unordered qualified by reference relationships. However, you can either set UnorderedQualifiedByReferenceContainer property to refer to your own container class, or you can provide an implementation for the default container class (dictionary by reference).

## Example of UnorderedQualifiedByReferenceContainer

This is an example of an unordered qualified by reference association and the data member that is generated for it by default, using Rational Rose default project properties:

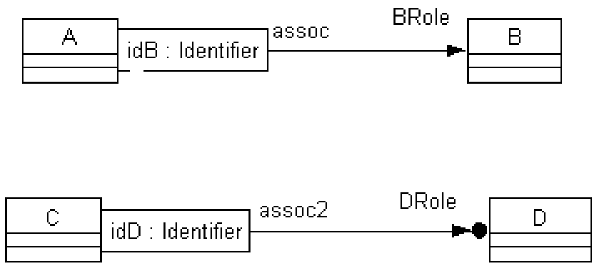


***Figure 79 Example of an Unordered Qualified By Reference Association***

The Rational Rose C++ code generator produces the following data member:

```
// Data Members for Associations
...
DictionaryByReference<Identifier,B*> BRole;
...
DictionaryByReference<Identifier,UnboundedSetByReference<B>>
, BRole;
...
```

## UnorderedQualifiedByValueContainer

The UnorderedQualifiedByValueContainer property indicates the default container class that the C++ code generator uses to generate a data member for a navigable unordered qualified has by value association. The value of this project property is used only if the ContainerClass property of the association is empty.

The default value of UnorderedQualifiedByValueContainer is:

```
DictionaryByValue<$qualtype,$qualcont>
```

An association is an ordered list of (key,value) pairs that can be efficiently accessed by key value.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The C++ code generator does not provide an implemented container class for unordered qualified by value relationships. However, you can either set the UnorderedQualifiedByValueContainer property to refer to a container class of your own, or you can provide an implementation for the default container class (dictionary by value).

### Example of UnorderedQualifiedByValueContainer

This is an example of an unordered qualified by value association and the data member that is generated for it by default, using Rational Rose default project properties:

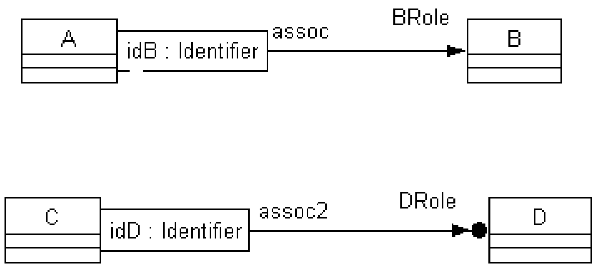


*Figure 80 Example of an Unordered Qualified By Value Association*

The Rational Rose C++ code generator produces the following data member:

```
// Data Members for Associations
...
DictionaryByValue<Identifier,B> BRole;
...
DictionaryByValue<Identifier, UnboundedSetByValue<D> >
DRole;
...
```

## UnorderedUnboundedByReferenceContainer

The UnorderedUnboundedByReferenceContainer property indicates the default container class that the C++ code generator uses to generate a data member for an unordered unbounded has by reference relationship or a navigable unordered unbounded has by reference association. The value of this project property is used only if the ContainerClass property of the association or has relationship is empty.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The default value of UnorderedUnboundedByReferenceContainer is:

UnboundedSetByReference<$targetClass>

You can use $targetClass when you provide your own value for this project property. When this project property is empty, the generated data member has a void data type.

The C++ code generator does not provide an implemented container class for unordered unbounded by reference relationships. However, you can either set UnorderedUnboundedByReferenceContainer property to refer to your own container class, or you can provide an implementation for the default container class (unbounded set by reference).

## Example of UnorderedUnboundedByReferenceContainer

This is an example of an unordered unbounded by reference association and the data member that is generated for it by default, using Rational Rose default project properties:

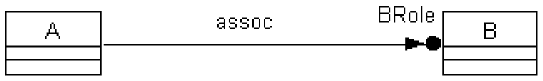


***Figure 81 Example of an Unordered Unbounded By Reference Association***

The Rational Rose C++ code generator produces the following data member:

```
// Data Members for Associations
...
UnboundedSetByReference<B,10> BRole;
...
```

## UnorderedUnboundedByValueContainer

The UnorderedUnboundedByValueContainer property indicates the default container class that the C++ code generator uses to generate a data member for an unordered unbounded has by value relationship or a navigable unordered unbounded has by value association. The value of this project property is used only if the ContainerClass property of the association or has relationship is empty.

The default value of UnorderedUnboundedByValueContainer is:

UnboundedSetByValue<$targetClass,$limit>

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The C++ code generator does not provide an implemented container class for unordered unbounded by value relationships. However, you can either set UnorderedUnboundedByValueContainer to refer to a container class of your own, or you can provide an implementation for the default container class (unbounded set by value).

### Example of UnorderedUnboundedByValueContainer

This is an example of an unordered unbounded by value association and the data member that is generated for it by default, using Rational Rose default project properties:

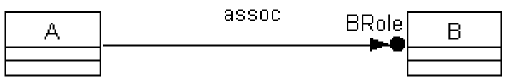


***Figure 82 Example of an Unordered Unbounded By Value Association***

The Rational Rose C++ code generator produces the following data member:

```
// Data Members for Associations
...
UnboundedSetByValue<B> BRole;
...
```

## UseMSVC

The UseMSVC property specifies whether the C++ code generator produces MSVC (Microsoft Visual C++) source code.

When the C++ code generator produces MSVC source code, it also uses the MSVC code generation properties. The MSVC properties are specified in the MSVC tab of the **Options** dialog.

The following table lists the values for UseMSVC:

***Table 168 The UseMSVC Values***

| **If you select:** | **The action is:** |
|---|---|
| True | The C++ code generator uses MSVC code generation properties. |
| False | (Default) The C++ code generator does not use MSVC properties. |

# *Appendix I*

# *Association Role Properties*

## AssocClassContainer

The AssocClassContainer property specifies a data type for a data member generated for the association relationship in an association class. You can set AssocClassContainer to reference your own container class and the C++ code generator uses your container class to generate the data member type.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The default value of this property is $supplier, indicating the C++ code generator generates a pointer to a single instance of the supplier class.

The following table lists the values for AssocClassContainer:

***Table 169 AssocClassContainer Values***

| **If you enter:** | **The action is:** |
|---|---|
| <blank> | (Default) The C++ code generator produces a type for the data member based on the supplier cardinality and containment of the association relationship. |
| *type expression* | The C++ code generator uses *type expression* as the data member type. Usually, *type expression* is an instance of a template. The *type expression* text may contain the variables $supplier, $target, and $targetClass. |

## AssocClassDataMemberIsVolatile

The AssocClassDataMemberIsVolatile property specifies whether a data member for an association role in the association class is volatile.

The following table lists the values for AssocClassDataMemberIsVolatile:

***Table 170 AssocClassDataMemberIsVolatile Values***

| If you enter: | The action is: |
|---|---|
| True | If a data member is generated for the association role in the association class, the declaration is adorned with the `v` keyword. |
| False | (Default) The declaration is not adorned with the `v` keyword. |

## AssocClassDataMemberMutability

The AssocClassDataMemberMutability property identifies how the declaration is adorned if a data member is generated for an association role in the association class.

The following table lists the values for AssocClassDataMemberMutability:

***Table 171 AssocClassDataMemberMutability Values***

| If you enter: | The action is: |
|---|---|
| Mutable | If a data member is generated for the association role in the association class, the declaration is prefixed with the `mutable` keyword. |
| Const | If a data member is generated for the association role in the association class, the declaration will be prefixed with the `const` keyword. |
| Unrestricted | (Default) No `const` or `mutable` adornments are generated. |

## AssocClassDataMemberName

The AssocClassDataMemberName property specifies the name the C++ code generator produces for a data member for an association role in the association class. The default value is:

```
$target
```

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The following class diagrams and code examples illustrate an association role (with and without an association class) and the data member that the C++ code generator produces for it by default:

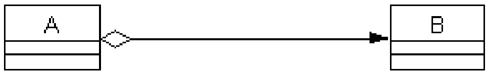


***Figure 83 Association Role without an Association Class***

```
// Data Members for Associations
```

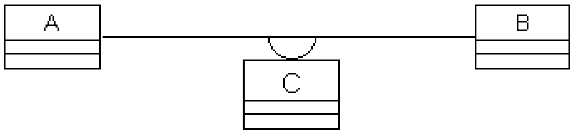


***Figure 84 Association Role with an Association Class***

```
// Data Members for Associations with Association Classes
The Data Member in C:  B the_B
```

You can change the form of the names the C++ code generator uses for the name of the data member in the association class by changing the AssocClassDataMemberName format. Note that if any of the symbols above are followed by a character that can appear in an identifier, you must enclose the symbol name in braces {}.

You can control the case of the name derived from $target. This table describes the options (note that these options also can be used with $supplier):

***Table 172 AssocClassDataMemberName Case Options***

| **If you enter:** | **The action is:** |
|---|---|
| ${target:l} | All characters in the relationship name are converted to lower case. |
| ${target:u} | All characters in the relationship name are converted to upper case. |
| ${target:f} | The case of the first character in the relationship name is inverted. |
| ${target:i} | The case of all characters in the relationship name is inverted. |

***Note:*** *If the association does not have an association class, this property has no effect on the generated code.*

## AssocClassDataMemberVisibility

The AssocClassDataMemberVisibility property specifies the visibility of the data member generated from the association relationship in the association class. The C++ code generator uses this information to determine access for the generated data member.

The following table lists the values for AssocClassDataMemberVisibility:

***Table 173 AssocClassDataMemberVisibility Values***

| **If you select:** | **The action is:** |
|---|---|
| Public | The C++ code generator produces the data member with public access. |
| Protected | The C++ code generator produces the data member with protected access. |

| If you select: | The action is: |
|---|---|
| Private | The C++ code generator produces the data member with private access. |
| Implementation | (Default) The C++ code generator produces the data member in a second private area called private implementation. |
| At Relationship Visibility | The C++ code generator produces the data member at the level of access specified for the relationship in the Rational Rose model. This choice is useful when you want to implement the relationship by direct access to the data member, rather than protecting it with Get and Set, or other operations. |

Normally, when AssocClassDataMemberVisibility is set to this value, you should set the association role's GenerateGetOperation and GenerateSetOperation properties to False.

***Note:*** *If the association does not have an association class, this property has no effect on the generated code.*

## AssocClassForwardReferenceOnly

The AssocClassForwardReferenceOnly property specifies whether the C++ code generator produces a forward declaration for an associate class before the class definition of the association class. The default value is True. If this property is True, a forward declaration is produced for the associate class before the definition of the association class. If the property is False, a `#include` of the header file defining the association class is produced.

As a side effect, this property also controls whether a forward declaration or a `#include` is produced for the association class in each associate class. A `#include` is normally produced unless that would produce a direct cycle in combination with a `#include` generated because AssocClassForwardReferenceOnly is False. (A cycle refers to header files that include each other in a cyclic manner.)

You must take the necessary steps to avoid cycles in the C++ code generator. Some rules must be adhered to when breaking cycles. A cycle may exist in a Rational Rose model consisting of two or more uses, inherits, or instantiates relationships. These relationships may involve dependencies implied by navigable association roles, either between associate classes or between an association class and one or more association classes.

You must break the cycle using one of two properties. Set ForwardReferenceOnly to True for at least one of the has or uses relationships, or for a navigable association role. Or you can set AssocClassForwardReferenceOnly to True for at least one of the association roles. You can also set AssocClassForwardReferenceOnly to True to break cycles that include other kinds of dependencies, such as those specified by the module dependency relationship on a component diagram, or by includes you put in the AdditionalIncludes protected code region.

If the associate class and the association class are in the same module, this property controls their order of declaration.

The following table lists the values for AssocClassForwardReferenceOnly:

***Table 174 AssocClassForwardReferenceOnly Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator produces a forward declaration for an associate class before the class definition of the association class. |
| False | The C++ code generator produces a `#include` of the header file defining the associate class in the module containing the association class. The `#include` of the module defining the association class in the module containing the associate class, if any, is replaced by a forward declaration. |

## AssocClassGetIsConst

The AssocClassGetIsConst property specifies whether the C++ code generator produces get member functions in the association class for a data member with the `const` keyword. `const` member functions cannot modify class data members.

The following table lists the values for AssocClassGetIsConst. In this table, `T` is the supplier class of the association relationship in the association class:

***Table 175 AssocClassGetIsConst Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True (Default) | `const T get_the_T() const;` |
| False | `T get_the_T();` |

***Note:*** *If the association does not have an association class, this property has no effect on the generated code.*

## AssocClassGetName

The AssocClassGetName property specifies the name the C++ code generator produces for a get member function for an association role in the association class. The default value is:

```
get_$target
```

The following class diagram and code example illustrate an association role and the get member function that the C++ code generator produces for it by default:

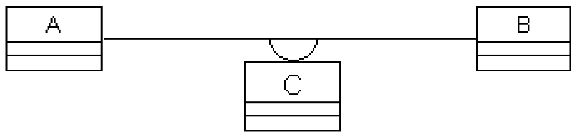


***Figure 85 Association Role and the Get Member Function***

```
// Get and Set Operations for Association Roles
const B get_the_B() const;
```

You can change the form of the names that the C++ code generator produces for get member functions by changing the AssocClassGetName format. You can also refer to $supplier in AssocClassGetName. Note that if either $target or $supplier is followed by a character that can appear in an identifier, you must enclose the variable name in braces {}.

For example, if you set GetName to:

```
${target}_get
```

the C++ code generator produces the following get member function for the association role:

```
// Get and Set Operations for Association Roles:
const B the_B_get() const;
```

You can control the case of the name derived from $target. This table describes the possible options (Note that these options also can be used with $supplier):

***Table 176 AssocClassGetName Case Options***

| **If you enter:** | **The action is:** |
|---|---|
| ${target:l} | All characters in the target name are converted to lower case. |
| ${target:u} | All characters in the target name are converted to upper case. |
| ${target:f} | The case of the first character in the target name is inverted. |
| ${target:i} | The case of all characters in the target name is inverted. |

***Note:** If the association does not have an association class, this property has no effect on the generated code.*

## AssocClassGetResultIsConst

If a get function is generated in the line class for the item, the `const`-ness of the returned value will be determined by the value of this property in the same way GetResultIsConst dictates the `const`-ness of the result for the get function.

The following table lists the values for AssocClassGetResultIsConst:

***Table 177 AssocClassGetResultIsConst Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | If a get function is generated for this item, it will return a const value. |
| False | If a get function is generated for this item, it will return a non-const value. |
| Same_As_Function | If a get function is generated for this item, it will return a const value if the function is const and a non-const value if the function is not const (as dictated by the Get Is Const property). |

## AssocClassGetSetKinds

The AssocClassGetSetKinds property specifies the kind of member or friend functions that are generated for the get and set operations for a data member in the association class. The C++ code generator produces additional keywords in the declarations of the get and set member functions based on the value of GetSetKinds, such as `static` or `virtual`.

The following table lists the values for AssocClassGetSetKinds. In this table, `T` is the name of the supplier class in the association relationship and `the_T` is the name of the supplier role of the association relationship in the association class:

***Table 178 AssocClassGetSetKinds Values***

| If you select: | The C++ code generator produces: |
|---|---|
| Common (Default) | **`T get_the_T();`**<br>**`void set_the_T(const T value);`** |
| Virtual | **`virtual T get_the_T();`**<br>**`virtual void set_the_T(const T value)`** |
| Static | **`static T get_the_T(A &client);`**<br>**`static void set_the_T(A &client, const T value);`** |
| Abstract | **`virtual T get_the_T() = 0`**<br>**`virtual void set_the_T(const T value) = 0;`** |
| Friend | **`friend T get_the_T(A &client);`**<br>**`friend void set_the_T(A &client, const T value);`** |

***Note:** If the association does not have an association class, this property has no effect on the generated code.*

## AssocClassInitialValue

The AssocClassInitialValue property specifies the initial value for an association role in the association class. When C++ code generator produces the declaration of the association role, it also produces the initial value for the declaration.

The following table lists the values for AssocClassInitialValue:

***Table 179 AssocClassInitialValue Values***

| If you select: | The action is: |
|---|---|
| <string> | The C++ code generator produces a declaration for the association role with <string> as its initial value. |
| <blank> | (Default) The C++ code generator produces a declaration for the association role with no initial value. |

## AssocClassSetName

The AssocClassSetName property specifies the name the C++ code generator produces for a set member function for an association role. The default value is:

```
set_$target
```

If the association relationship is unlabeled, $target expands to the value of the association's NameIfUnlabeled property.

The following class diagram and code example illustrate an association role and the set member function that the C++ code generator produces for it by default:

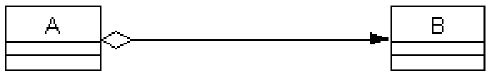


***Figure 86 Association Role and the Set Member Function***

```
// Get and Set Operations for Association Roles
void set_the_B(B value);
```

You can change the form of the names that the C++ code generator produces for set member functions by changing the AssocClassSetName format. You can also refer to $supplier in AssocClassSetName. Note that if either $target or $supplier is followed by a character that can appear in an identifier, you must enclose the variable name in braces {}.

For example, if you set the Set Name property to:

```
${target}_set
```

the C++ code generator produces the following set member function for the association role:

```
// Get and Set Operations for Association Roles
void the_B_set(const B value);
```

You can control the case of the name derived from $target. This table lists the options (note that these options also can be used with $supplier):

***Table 180 AssocClassSetName Case Options***

| If you enter: | The action is: |
|---|---|
| ${target:l} | All characters in the target name are converted to lower case. |
| ${target:u} | All characters in the target name are converted to upper case. |
| ${target:f} | The case of the first character in the target name is inverted. |
| ${target:i} | The case of all characters in the target name is inverted. |

***Note:*** *If the association does not have an association class, this property has no effect on the generated code.*

## AssocClassSetReturnsValue

The AssocClassSetReturnsValue property specifies whether the C++ code generator produces the member function for an association relationship with a non-`void` return type in the association class. By default, the C++ code generator produces the set member function with return type `void`. However, sometimes it is convenient for the set member function to return the value to which the data member is set in the function.

The following table lists the values for AssocClassSetReturnsValue. In this table, `T` is the name of the supplier class of the association relationship and `the_T` is the name of the association relationship:

***Table 181 AssocClassSetReturnsValue Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | **`const T set_the_T(const T value);`** |
| False (Default) | **`void set_the_T(const T value);`** |

***Note:*** *If the association does not have an association class, this property has no effect on the generated code.*

## CodeName

The CodeName property specifies the name for the association role in the generated code.

You need to set this property only if you want the association role to be named differently than it is in the Rational Rose model. This is especially useful when the Rational Rose model and code are expressed in different natural languages. The value of this property should be a valid C++ identifier. If it is not a valid C++ identifier, the C++ code generator will perform the following substitutions:

- White space characters and hyphens are changed to underscores
- All other characters except letters, digits, and underscores are changed to "X"
- If the first character is a number, an "N" precedes the number

*Table 182 Code Name Property Values*

| If you select: | The action is: |
|---|---|
| C++ Identifier | The association role is assigned the name to be used for the role in the generated code. This is especially useful when the design and code are written in different natural languages. |
| <blank> | (Default) The C++ code generator uses the name or the association role from the model. |

## ContainerClass

The ContainerClass property specifies a data type for the data member generated for the association relationship. You can set ContainerClass to reference your own container classes and the C++ code generator would then use them to generate types for data members.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The container class property also specifies the return type of the get operation and the parameter type of the value parameter of the set operation.

If you leave this property blank, the data members are generated whose types are container classes organized as lists, sets, or dictionaries. You can use this property to generate any other container class.

The following table lists the values for ContainerClass:

*Table 183 ContainerClass Values*

| If you enter: | The action is: |
|---|---|
| <blank> | (Default) The C++ code generator produces a type for the data member based on the supplier cardinality and containment of the association relationship. |
| *type expression* | The C++ code generator uses *type expression* as the type of the data member. Usually, *type expression* is an instance of a template. The *type expression* text may contain the variables $supplier and $limit. |

## ContainerGet

When a qualified "Get" member function is generated for a role (when the role is qualified and GenerateQualifiedGet is True), the value of ContainerGet is used to generate the body of the Get method, which takes a set of keys and returns the element of the collection associated with that set of keys.

Within the prototype the special symbols $data, $value, and $key*n* (for ***n*** = 1..9) represent the name of the collection, the selected element value, and the qualifying keys, respectively. If $value is not present in the prototype it is assumed that the C++ expression evaluates to the selected collection element.

The possible value for ContainerGet is the prototype C++ expression for retrieving an element from the collection that implements a qualified association.

## ContainerSet

When a qualified "Set" member function is generated for a role (when the role is qualified and GenerateQualifiedSet is True), the value of ContainerSet is used to generate the body of the Set method, which takes a set of keys and a value and stores the value into the element of the collection associated with that set of keys.

Within the prototype the special symbols $data, $value, and $key*n* (for ***n*** = 1..9) represent the name of the collection, the selected element value, and the qualifying keys, respectively.

The possible value of the ContainerSet property is the prototype C++ expression for storing an element into the collection that implements a qualified association.

## DataMemberFieldSize

The DataMemberFieldSize property specifies the number of bits as the size of the designated data member.

The following table lists the values for DataMemberFieldSize:

***Table 184 DataMemberFieldSize Values***

| If you select: | The action is: |
|---|---|
| <integer> | The C++ code generator specifies <integer> bits as the size of the data member. |
| <blank> | (Default) The C++ code generator does not specify the size for the data member. |

## DataMemberIsVolatile

If the DataMemberIsVolatile property is True, a data member is generated for this relationship and the declaration is adorned with the v keyword.

The following table summarizes the values for DataMemberIsVolatile:

***Table 185 DataMemberIsVolatile Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | If a data member is generated for this relationship, the declaration is adorned with the v keyword. |
| False | The declaration will not be adorned with the v keyword. |

## DataMemberMutability

The DataMemberMutability property identifies how the declaration is adorned if a data member is generated for this relationship.

The following table lists the values for DataMemberMutability:

***Table 186 DataMemberMutability Values***

| If you select: | The C++ code generator produces: |
|---|---|
| Mutable | If a data member is generated for this relationship, the declaration is adorned with the `mutable` keyword. |
| Const | If a data member is generated for this relationship, the declaration is adorned with the `const` keyword. |
| Unrestricted | (Default) No `const` or `mutable` adornments are generated |

## DataMemberName

The DataMemberName property specifies the name the C++ code generator produces for a data member for an association role. The default value is:

```
$target
```

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The following class diagrams and code examples illustrate an association role (with and without an association class) and the data member that the C++ code generator produces for it by default:

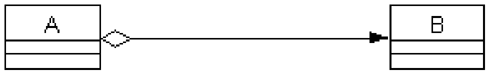


***Figure 87 Data Members for Associations***

```
...
B the_B;
```

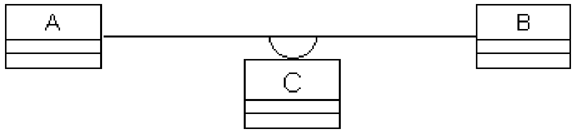


***Figure 88 Data Members for Associations with Association Classes***

```
...
The Data Member in A:  C* the_C
The Data Member in C:  B the_B
```

You can change the form of the names the C++ code generator uses for unlabeled association roles by changing the DataMemberName format. Note that if any of the symbols above are followed by a character that can appear in an identifier, you must enclose the symbol name in braces {}.

You can control the case of the name derived from $target. This table describes the possible options (note that these options also can be used with $supplier):

***Table 187 DataMemberName Case Options***

| If you enter: | The action is: |
|---|---|
| ${target:l} | All characters in the relationship name are converted to lower case. |
| ${target:u} | All characters in the relationship name are converted to upper case. |
| ${target:f} | The case of the first character in the relationship name is inverted. |
| ${target:i} | The case of all characters in the relationship name is inverted. |

## DataMemberVisibility

The DataMemberVisibility property specifies the visibility of the data member generated from the association relationship. The C++ code generator uses this information to determine access for the generated data member.

The following table lists the values for DataMemberVisibility:

***Table 188 DataMemberVisibility Values***

| If you select: | The action is: |
|---|---|
| Public | The C++ code generator produces the data member with public access. |
| Protected | The C++ code generator produces the data member with protected access. |

| If you select: | The action is: |
|---|---|
| Private | The C++ code generator produces the data member with private access. |
| Implementation | (Default) The C++ code generator produces the data member in a second private area called private implementation. |
| At Relationship Visibility | The C++ code generator produces the data member at the level of access specified for the relationship in the Rational Rose model. This choice is useful when you want to implement the relationship by direct access to the data member, rather than protecting it with Get and Set, or other operations. Normally, when DataMemberVisibility is set to this value, you should set GenerateGetOperation and GenerateSetOperation to False. |

## ForwardReferenceOnly

The ForwardReferenceOnly property specifies whether the C++ code generator produces a forward declaration for the supplier class of the association relationship before the class definitions in the module. If, in the Rational Rose model, there is a cycle consisting of two or more association, uses, inherits, and instantiates relationships, and navigable association roles, you must set ForwardReferenceOnly to True for at least one of the association or uses relationships, or navigable roles to break the cycle. You may also set ForwardReferenceOnly to break cycles that include other kinds of dependencies, such as those specified by the module dependency relationship on a component diagram, or by includes you put in the AdditionalIncludes protected code region.

***Note:*** *This property does not apply to inherits or instantiates relationships because a forward declaration never provides sufficient information to the compiler in these cases.*

Each associate class is always dependent on its association class. An association class always has forward references to its associate class.

For example, if two classes are assigned to the same module and each class is the supplier in an association relationship with the other, as shown here, then a forward declaration is needed for one of the classes.

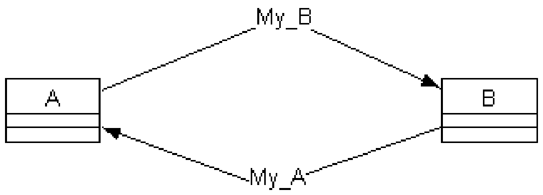


***Figure 89 Example of an Association Relationship***

Setting ForwardReferenceOnly to True for the "A to B" direction of the association (`My_B`) causes the C++ code generator to generate a forward declaration for class `B`, as shown by the following code example:

```
class B;
class A
{...
   // Data Members for Association Relationships
   B *My_B;
...};
class B
{...
   // Data Members for Association Relationships
   A *My_A;
...};
```

Note that this restricts how `A` can contain `B`. In particular, `A` cannot contain an instance of class `B`, although it can contain a pointer to an instance of `B`. It seldom makes sense to set ForwardReferenceOnly to True for an association-by-value because such a relationship usually implies containment. If the association role's GenerateDataMember property is set to False, however, it is possible to implement the association in a way that simulates containment without actual containment.

If the classes are assigned to different modules, then, by default, A's header file would contain B's header file and vice versa, creating a circular inclusion. To eliminate the circular inclusion, you set ForwardReferenceOnly to True for one of the association roles. The C++ code generator produces a forward declaration for the supplier class in the client module and suppresses the `#include` directive in the client module. In the following code example, ForwardReferenceOnly is set to True for the "A has B" relationship:

***Table 189 Eliminate Circular Inclusion***

| A's Header File | B's Header File |
|---|---|
| `class B;` | `#include "a.h"` |
| `...` | `...` |
| `class A{` | `class B {` |
| `...` | `...` |
| `};` | `};` |

The following table lists the values for ForwardReferenceOnly:

***Table 190 ForwardReferenceOnly Values***

| If you select: | The action is: |
|---|---|
| True | The C++ code generator produces a forward declaration for the supplier class of the association relationship. If the supplier and the client are assigned to different modules, the C++ code generator suppresses the `#include` directive for the supplier module in the client module. |
| False | (Default) The C++ code generator does not generate a forward declaration for the supplier class of the association relationship. |

## GenerateAssocClassDataMember

The GenerateAssocClassDataMember property specifies whether the C++ code generator produces a data member corresponding to the association role in the association class. Regardless of the setting of this property, the C++ code generator generates code for associations only in directions that are marked as navigable in the Rational Rose model. The name of the data member is determined by the AssocClassDataMemberName value for the association role. The data member type is one that designates an instance of the supplier class.

You set GenerateAssocClassDataMember to False if you want to provide your own data member definition or if you want to implement the association in some other way. After you generate code, you edit the generated file and add your own data member definition (or some other implementation of the association) between the source markers for the data member. Be sure to change the preserve setting in the source marker to "yes."

The following table lists the values for GenerateAssocClassDataMember:

***Table 191 GenerateAssocClassDataMember Values***

| **If you select:** | **The action is:** |
|---|---|
| True | (Default) The C++ code generator produces a data member for the association in the association class. |
| False | The C++ code generator does not generate a data member for the association in the association class. |

***Note:*** *If the association does not have an association class, this property has no effect on the generated code.*

## GenerateAssocClassGetOperation

The GenerateAssocClassGetOperation property specifies whether the C++ code generator produces a get member function that accesses the value of the data member generated from the association relationship in the association class. The name of the get member function is determined by the value of the association relationship Assoc ClassGetName property. The result type of the get member function is the type of the data member.

You set GenerateAssocClassGetOperation to False if you do not want to provide a member function for accessing the data member. If you want to create a custom get operation, set GenerateAssocClassGetOperation to False and then define your get operation in the association class specification, as you would for any user-defined operation.

The following table lists the values for GenerateAssocClassGetOperation:

***Table 192  GenerateAssocClassGetOperation Values***

| **If you select:** | **The action is:** |
|---|---|
| True | (Default) The C++ code generator produces a get operation for the data member in the association class. |
| False | The C++ code generator does not generate a get operation for the data member in the association class. |

***Note:*** *If the association does not have an association class, this property has no effect on the generated code.*

## GenerateAssocClassSetOperation

The GenerateAssocClassSetOperation property specifies whether the C++ code generator produces a set member function that modifies the value of the data member generated from the association relationship in the association class. The name of the set member function is determined by the AssocClassSetName value of the association relationship. By default, the result type of the set member function is void. You can change the result type of the set member function by setting the association relationship's AssocClassSetReturnsValue property.

You set GenerateAssocClassSetOperation to False if you do not want to provide a member function for modifying the data member. If you want to create a custom set operation, set GenerateAssocClassSetOperation to False and then define your set operation in the class specification, as you would for any user-defined operation.

The following table lists the values for GenerateAssocClassSetOperation:

***Table 193  GenerateAssocClassSetOperation Value***

| **If you select:** | **The action is:** |
|---|---|
| True | (Default) The C++ code generator produces a set operation for the data member in the association class. |
| False | The C++ code generator does not generate a set operation for the data member in the association class. |

***Note:*** *If the association does not have an association class, this property has no effect on the generated code.*

## GenerateDataMember

The GenerateDataMember property specifies whether the C++ code generator produces a data member corresponding to the association role. Regardless of the setting of this property, the C++ code generator generates code for associations only in directions that are marked as navigable in the Rational Rose model. The name of the data member is determined by the DataMemberName value for the association role. The data member type is one that contains or designates zero or more instances of the supplier class.

You set GenerateDataMember to False if you want to provide your own data member definition or if you want to implement the association in some other way. After you generate code, you edit the generated file and add your own data member definition (or some other implementation of the association) between the source markers for the data member. Be sure to change the preserve setting in the source marker to `yes`.

The following table lists the values for GenerateDataMember:

***Table 194 GenerateDataMember Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator produces a data member for the association. |
| False | The C++ code generator does not generate a data member for the association. |

## GenerateGetOperation

The GenerateGetOperation property specifies whether the C++ code generator produces a get member function, which accesses the value of the data member generated from the association relationship. The name of the get member function is determined by the GetName value for the association relationship. The result type of the get member function is the type of the data member.

You set GenerateGetOperation to False if you do not want to provide a member function for accessing the data member. If you want to create a custom get operation, set GenerateGetOperation to False and then define your get operation in the class specification, as you would for any user-defined operation.

This get operation accesses the entire data member. It does not reduce the subset of the suppliers returned by qualification.

The following table lists the values for GenerateGetOperation:

***Table 195 GenerateGetOperation Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator produces a get operation for the data member. |
| False | The C++ code generator does not generate a get operation for the data member. |

## GenerateQualifiedGetOperation

The GenerateQualifiedGetOperation property specifies whether the C++ code generator produces a get member function, which accesses the value of the data member generated from the association relationship. The name of the get member function is determined by the Get Name value for the association relationship. The result type of the get member function is a container suitable to hold the set of suppliers associated with a single client with a single set of qualifier values. See the QualifiedContainer property for more information. The parameters of this operation are the association qualifiers.

You set GenerateQualifiedGetOperation property to False if you do not want to provide a member function for accessing the data member. If you want to create a custom get operation, set GenerateQualifiedGetOperation to False and then define your get operation in the Class Specification, as you would for any user-defined operation.

The following table lists the values for GenerateQualifiedGetOperation:

***Table 196 GenerateQualifiedGetOperation Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator produces a basic get operation for the data member. |
| False | The C++ code generator does not generate a get operation for the data member. |

## GenerateQualifiedSetOperation

The GenerateQualifiedSetOperation property specifies whether the C++ code generator produces a set member function, which modifies the value of the data member generated from the association relationship. The name of the set member function is determined by the SetName value for the association relationship. By default, the result type of the set member function is `void`. You can change the result type of the set member function through SetReturnsValue for the association relationship. The parameters of this operation are the container of values to be associated with a single client and set of qualifier values, along with the values of the qualifiers.

You set GenerateQualifiedSetOperation to False if you do not want to provide a member function for modifying the data member with qualifications. If you want to create a custom set operation, set GenerateQualifiedSetOperation to False and then define your set operation in the Class Specification, as you would for any user-defined operation.

The following table lists the values for GenerateQualifiedSetOperation:

***Table 197 GenerateQualifiedSetOperation Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator produces a basic set operation for the data member. |
| False | The C++ code generator does not generate a set operation for the data member. |

## GenerateSetOperation

TheGenerateSetOperation property specifies whether the C++ code generator produces a set member function, which modifies the value of the data member generated from the association relationship. The name of the set member function is determined by the SetName value for the association relationship. By default, the result type of the set member function is `void`. You can change the result type of the set member function through SetReturnsValue for the association relationship.

You set GenerateSetOperation to False if you do not want to provide a member function for modifying the data member. If you want to create a custom set operation, set GenerateSetOperation to False and then define your set operation in the Class Specification, as you would for any user-defined operation.

This set operation accesses the entire data member. It does not reduce the subset of the suppliers returned by qualification.

The following table lists the values for GenerateSetOperation:

***Table 198 GenerateSetOperation Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator produces a set operation for the data member. |
| False | The C++ code generator does not generate a set operation for the data member. |

## GetIsConst

The GetIsConst property specifies whether the C++ code generator produces the unqualified get member functions for a data member with the `const` keyword. Const member functions cannot modify class data members.

The following table lists the values for GetIsConst. In this table, `T` is the supplier class of the association relationship:

***Table 199 GetIsConst Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True (Default) | **`const T get_the_T() const;`** |
| False | **`T get_the_T();`** |

## GetName

The GetName property specifies the name the C++ code generator produces for a get member function for an association role. The default value of GetName is:

```
get_$target
```

When the C++ code generator produces a get member function for a data member and there is no associate class, $target expands to the label of the supplier role in the model. If there is an association class, $target expands to the role of the association. If there is no association class, $target expands to the value of NameIfUnlabeled for the association role. If the association relationship is unlabeled, $target expands to the NameIfUnlabeled value for the association role.

If $targetClass is used in GetName, it is the same as $supplier when generating code in the association class. If there is no association class, it is the same as $supplier. Otherwise, it is the code name for the association.

If $role is used in GetName, it is the code name for the role. If $association is used in GetName, it is the code name for the association

The following class diagram and code example illustrate an association role and the get member function that the C++ code generator produces for it by default:

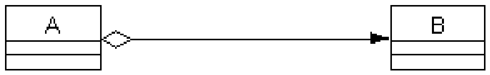


***Figure 90 Association Role and the Set Member Function***

```
// Get and Set Operations for Association Roles
const B get_the_B() const;
```

You can change the form of the names that the C++ code generator produces for get member functions by changing the GetName format. You can also refer to $supplier in GetName. Note that if either $target or $supplier is followed by a character that can appear in an identifier, you must enclose the variable name in braces {}.

For example, if you set GetName to:

```
${target}_get
```

the C++ code generator produces the following get member function for the association role:

```
// Get and Set Operations for Association Roles:
const B the_B_get() const;
```

You can control the case of the name derived from $target. This table describes the possible options (note that these options also can be used with $supplier):

***Table 200 GetName Case Options***

| **If you enter:** | **The action is:** |
|---|---|
| ${target:l} | All characters in the target name are converted to lower case. |
| ${target:u} | All characters in the target name are converted to upper case. |
| ${target:f} | The case of the first character in the target name is inverted. |
| ${target:i} | The case of all characters in the target name is inverted. |

## GetResultIsConst

The GetResultIsConst property returns a `const` value if a get function is generated for this element.

The following table lists the values for GetResultIsConst:

***Table 201 GetResultIsConst Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | If a get function is generated for this item, it returns a `const` value. |
| False | If a get function is generated for this item, it returns a non-`const` value. |
| Same_As_Function | If a get function is generated for this item, it returns a `const` value if the function is `const` and a non-`const` value if the function is not `const` (as dictated by GetIsConst). |

## GetSetByReference

The GetSetByReference property specifies whether values in the unqualified and qualified get and set member functions are passed by reference or by value. By default, the C++ code generator produces get and set member functions for an association relationship to pass arguments and return values by value. You set the value of GetSetByReference to True if you want the get and set member functions to pass arguments and return values by reference.

The following table lists the values for GetSetByReference. In this table, `T` is the name of the supplier class of the association relationship and `the_T` is the name of the association relationship:

***Table 202 GetSetByReference Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | **`T & get_the_T();`**<br>**`void set_the_T(const T &value);`** |
| False (Default) | **`T get_the_T();`**<br>**`void set_the_T(const T value);`** |

## GetSetKinds

The GetSetKinds property specifies the kind of member or friend functions that are generated for the get and set operations for a data member. The C++ code generator produces additional keywords in the declarations of the get and set member functions based on the value of GetSetKinds, such as `static` or `virtual`.

The following table lists the values for GetSetKinds. In this table, `T` is the name of the supplier class in the association relationship and `the_T` is the name of the supplier role of the association relationship:

***Table 203 GetSetKinds Values***

| If you select: | The C++ code generator produces: |
|---|---|
| Common (Default) | **`T get_the_T();`**<br>**`void set_the_T(const T value);`** |
| Virtual | **`virtual T get_the_T();`**<br>**`virtual void set_the_T(const T value)`** |
| Static | **`static T get_the_T(A &client);`**<br>**`static void set_the_T(A &client,`**<br>**`const T value);`** |
| Abstract | **`virtual T get_the_T() = 0;`**<br>**`virtual void set_the_T(const T value)`**<br>**`= 0;`** |
| Friend | **`friend T get_the_T(A &client);`**<br>**`friend void set_the_T(A &client,`**<br>**`const T value);`** |

***Note:*** *If the association relationship itself is static, then the only legal values for* GetSetKinds *are "Common" and "Static." In both cases, the C++ code generator produces:*

```
static T get_the_T();
static void set_the_T(const T value);
```

## InitialValue

The InitialValue property specifies the initial value for the supplier role of an association. When the C++ code generator produces the declaration of the supplier role, it also produces the initial value for the declaration.

The following table lists the values for InitialValue:

***Table 204 InitialValue Property Values***

| If you select: | The action is: |
|---|---|
| <string> | The C++ code generator produces a declaration for the role with <string> as its initial value. |
| <blank> | (Default) The C++ code generator produces a declaration for the role with no initial value. |

## InlineAssocClassGet

The InlineAssocClassGet property specifies whether the C++ code generator inlines get operations in the association class.

***Table 205 InlineAssocClassGet Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator inlines get operations. |
| False | The C++ code generator does not inline get operations. |

***Note:*** *If the association does not have an association class, this property has no effect on the generated code.*

## InlineAssocClassSet

The InlineAssocClassSet property specifies whether the C++ code generator inlines set operations in the association class.

***Table 206 InlineAssocClassSet Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator inlines set operations. |
| False | The C++ code generator does not inline set operations. |

***Note:*** *If the association does not have an association class, this property has no effect on the generated code.*

## InlineGet

The InlineGet property specifies whether the C++ code generator inlines get operations.

***Table 207 InlineGet Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator inlines get operations. |
| False | The C++ code generator does not inline get operations. |

## InlineQualifiedGet

The InlineQualifiedGet property specifies whether the C++ code generator inlines qualified get operations.

***Table 208 InlineQualifiedGet Values***

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator inlines qualified get operations. |
| False | The C++ code generator does not inline qualified get operations. |

## InlineQualifiedSet

The InlineQualifiedSet property specifies whether the C++ code generator inlines qualified set operations.

*Table 209 InlineQualifiedSet Values*

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator inlines qualified set operations. |
| False | The C++ code generator does not inline qualified set operations. |

## InlineSet

The InlineSet property specifies whether the C++ code generator inlines set operations.

*Table 210 InlineSet Values*

| If you select: | The action is: |
|---|---|
| True | (Default) The C++ code generator inlines set operations. |
| False | The C++ code generator does not inline set operations. |

## NameIfUnlabeled

The NameIfUnlabeled property specifies the name to be used for an unlabeled role. The C++ code generator uses the name of the role to construct names for the corresponding data member and get and set member functions. If the role is not named, the C++ code generator uses this property to determine the name of the role.

The default value of NameIfUnlabeled is:

```
the_$targetClass
```

When the C++ code generator needs the name of the role to generate a name for a data member or a get or set member function, $targetClass expands to the name of the association class or the association if there is one. Otherwise it expands to the name of the supplier class. If

$association is used in NameIfUnlabeled, it expands to the name of the association. If the $supplier is used in NameIfUnlabeled, it expands to the code name for the supplier class of the relevant Role.

By default, the C++ code generator uses `the_B` as the name of the unlabeled *role* shown in the following class diagram:

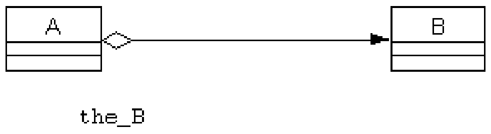


***Figure 91 Naming a Role***

You can change the form of the names the C++ code generator uses for unlabeled roles by changing the NameIfUnlabeled format. Note that if $targetClass is followed by a character that can appear in an identifier, you must enclose "targetClass" in braces {}.

For example, if you set NameIfUnlabeled to:

```
${targetClass}_rel
```

The C++ code generator uses `B_rel` as the name of the unlabeled role.

You can control the case of the name derived from $targetClass. This table describes the possible options:

***Table 211 NameIfUnlabeled Case Options***

| If you enter: | The action is: |
|---|---|
| ${targetClass:l} | All characters in the targetClass class name are converted to lower case. |
| ${targetClass:u} | All characters in the targetClass class name are converted to upper case. |
| ${targetClass:f} | The case of the first character in the targetClass class name is inverted. |
| ${targetClass:i} | The case of each character in the targetClass class name is inverted. |

## QualifiedContainer

The QualifiedContainer property specifies the data type of a container that contains or designates the supplier objects associated with a single client plus a particular setting for each qualifier. If you set QualifiedContainer to reference your own container classes, the C++ code generator uses your container classes to generate types for data members and for the container class property.

When the C++ code generator produces a data member, some symbols are expanded. Refer to the online help for information on the expanded symbol table.

The C++ code generator specifies the return data type for the data member generated from the qualified get operation. It also specifies the parameter type for the data member generated from the qualified set operation.

If you leave this property blank, the data members are generated whose types are container classes organized as sets. You can use this property to generate a non-set container class, such as a list or a bag. Note that containers are always unbounded if there is a qualifier.

The following table lists the values for QualifiedContainer:

***Table 212 QualifiedContainer Values***

| If you enter: | The action is: |
|---|---|
| (blank) | (Default) The C++ code generator produces a type for the data member based on the supplier cardinality and containment of the association relationship. |
| *type expression* | The C++ code generator uses *type expression* as the type of the data member. Usually, *type expression* is an instance of a template. The *type expression* text may contain the variables $supplier, $target, and $targetClass. |

## QualifiedGetIsConst

The QualifiedGetIsConst property specifies whether the C++ code generator produces the qualified get member function for a data member with the `const` keyword. `const` member functions cannot modify class data members.

The following table lists the values for QualifiedGetIsConst. In this table, `T` is the supplier class of the association relationship:

***Table 213 QualifiedGetIsConst Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True (Default) | `const T get_the_T() const;` |
| False | `T get_the_T();` |

## QualifiedGetName

The QualifiedGetName property specifies the name the C++ code generator produces for a qualified get member function for an association role. The default value is:

```
get_$target
```

When the C++ code generator produces a get member function for a data member, $target expands to the label of the association relationship in the model. If the association role is unlabeled, $target expands to the NameIfUnlabeled value for the association role.

If $targetClass is used in QualifiedGetName, it is the same as $supplier when generating code in the association class. If there is no association class, it is the same as $supplier. Otherwise, it is the code name for the association.

If $role is used in QualifiedGetName, it is the code name for the role. If $association is used in QualifiedGetName, it is the code name for the association.

The following class diagram and code example illustrate an association role and the get member function that the C++ code generator produces for it by default:

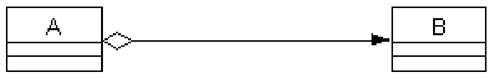


***Figure 92 Producing a Get Member Function***

```
// Get and Set Operations for Association Roles
const B get_the_B() const;
```

You can change the form of the names that the C++ code generator produces for get member functions by changing the QualifiedGetName format. You can also refer to $supplier in QualifiedGetName. Note that if either $target or $supplier is followed by a character that can appear in an identifier, you must enclose the variable name in braces {}.

For example, if you set QualifiedGetName to:

```
${target}_get
```

the C++ code generator produces the following get member function for the association relationship:

```
// Get and Set Operations for Association Roles
const B the_B_get() const;
```

You can control the case of the name derived from $target. This table describes the possible options (Note that these options also can be used with $supplier):

***Table 214 QualifiedGetName Case Options***

| If you enter: | The action is: |
|---|---|
| ${target:l} | All characters in the target name are converted to lower case. |
| ${target:u} | All characters in the target name are converted to upper case. |
| ${target:f} | The case of the first character in the target name is inverted. |
| ${target:i} | The case of all characters in the target name is inverted. |

## QualifiedGetResultIsConst

If a qualified get function is generated for the item, the `const`-ness of the returned value is determined by QualifiedGetResultIsConst in the same way GetResultIsConst dictates the `const`-ness of the result for a get function.

The following table lists the values for QualifiedGetResultIsConst:

***Table 215 QualifiedGetResultIsConst Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | If a get function is generated for this item, it will return a const value. |
| False | If a get function is generated for this item, it will return a non-const value. |
| Same_As_Function | If a get function is generated for this item, it will return a const value if the function is const and a non-const value if the function is not const (as dictated by GetIsConst). |

## QualifiedGetSetByReference

The QualifiedGetSetByReference property specifies whether values in the qualified get and set member functions are passed by reference or by value. By default, the C++ code generator produces get and set member functions for an association relationship to pass arguments and return values by value. You set the value of QualifiedGetSetByReference to True if you want the get and set member functions to pass arguments and return values by reference.

The following table lists the values for QualifiedGetSetByReference. In this table, `T` is the name of the supplier class of the association relationship and `the_T` is the name of the association relationship:

***Table 216 QualifiedGetSetByReference Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | **`T & get_the_T();`**<br>**`void set_the_T(const T &value);`** |
| False (Default) | **`T get_the_T();`**<br>**`void set_the_T(const T value);`** |

## QualifiedSetName

The QualifiedSetName property specifies the name the C++ code generator produces for a qualified set member function for an association role. The default value of the QualifiedSetName property is:

```
set_$target
```

When the C++ code generator produces a set member function for a data member, $target expands to the label of the association role in the model. If the association role is unlabeled, $target expands to the NameIfUnlabeled value for the association role.

If $targetClass is used in the QualifiedSetName property, it is the same as $supplier when generating code in the association class. If there is no association class, it is the same as $supplier. Otherwise, it is the code name for the association.

If $role is used in the QualifiedSetName property, it is the code name for the role. If $association is used in QualifiedSetName, it is the code name for the association.

The following class diagram and code example illustrate an association role and the set member function that the C++ code generator produces for it by default:

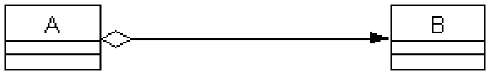


***Figure 93 Association Role and the Set Member Function***

```
// Get and Set Operations for Association Roles
void set_the_B(const B value);
```

You can change the form of the names that the C++ code generator produces for set member functions by changing the QualifiedSetName format. You can also refer to $supplier in QualifiedSetName. Note that if either $target or $supplier is followed by a character that can appear in an identifier, you must enclose the variable name in braces {}.

For example, if you set QualifiedSetName to:

```
${target}_set
```

the C++ code generator produces the following set member function for the association role:

```
// Get and Set Operations for Association Roles
void the_B_set(const B value);
```

You can control the case of the name derived from $target. This table describes the possible options (note that these options also can be used with $supplier):

***Table 217 QualifiedSetName Case Options***

| If you enter: | The action is: |
|---|---|
| ${target:l} | All characters in the target name are converted to lower case. |
| ${target:u} | All characters in the target name are converted to upper case. |
| ${target:f} | The case of the first character in the target name is inverted. |
| ${target:i} | The case of all characters in the target name is inverted. |

## QualifiedSetReturnsValue

The QualifiedSetReturnsValue property specifies whether the C++ code generator produces the qualified set member function for an association relationship with a non-void return type. By default, the C++ code generator produces the set member function with return type `void`. However, sometimes it is convenient for the set member function to return the value to which the data member is set in the function.

The following table lists the values for QualifiedSetReturnsValue. In this table, `T` is the name of the supplier class of the association relationship and `the_T` is the name of the association relationship:

***Table 218 QualifiedSetReturnsValue Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | `const T set_the_T(const T value);` |
| False (Default) | `void set_the_T(const T value);` |

## SetName

The SetName property specifies the name the C++ code generator produces for a set member function for an association role. The default value is:

```
set_$target
```

When the C++ code generator produces a set member function for a data member and there is no association class, $target expands to the label of the supplier role in the model. If there is an association class, $target expands to the name of the association. If the association relationship is unlabeled, $target expands to the NameIfUnlabeled value for the association.

If $targetClass is used in SetName, it is the same as $supplier when generating code in the association class. If there is no association class, it is the same as $supplier. Otherwise, it is the code name for the association.

If $role is used in SetName, it is the code name for the role. If $association is used in SetName, it is the code name for the association.

The following class diagram and code example illustrate an association role and the set member function that the C++ code generator produces for it by default:

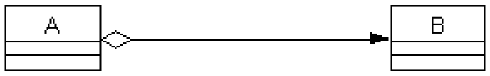


***Figure 94 Naming a Set Member Function***

```
// Get and Set Operations for Association Roles
void set_the_B(const B value);
```

You can change the form of the names that the C++ code generator produces for set member functions by changing the format of the SetName property. You can also refer to $supplier in the SetName property. Note that if either $target or $supplier is followed by a character that can appear in an identifier, you must enclose the variable name in braces {}.

For example, if the SetName value is:

```
${target}_set
```

the C++ code generator produces the following set member function for the association role:

```
// Get and Set Operations for Association Roles
void the_B_set(const B value);
```

You can control the case of the name derived from $target. This table describes the possible options (note that these options also can be used with $supplier).

***Table 219 SetName Case Options***

| If you enter: | The action is: |
|---|---|
| ${target:l} | All characters in the target name are converted to lower case. |
| ${target:u} | All characters in the target name are converted to upper case. |
| ${target:f} | The case of the first character in the target name is inverted. |
| ${target:i} | The case of all characters in the target name is inverted. |

## SetReturnsValue

The SetReturnsValue property specifies whether the C++ code generator produces the unqualified set member function for an association relationship with a non-void return type. By default, the C++ code generator produces the set member function with return type `void`. However, sometimes it is convenient for the set member function to return the value to which the data member is set in the function.

The following table lists the values for SetReturnsValue. In this table, `T` is the name of the supplier class of the association relationship and `the_T` is the name of the association relationship:

***Table 220 SetReturnsValue Property Values***

| If you select: | The C++ code generator produces: |
|---|---|
| True | **`const T set_the_T(const T value);`** |
| False (Default) | **`void set_the_T(const T value);`** |

# *Appendix J*

# *Subsystem Properties*

## Directory

The Directory property specifies the name of the directory to which the subsystem is mapped during code generation. The following table lists the values for Directory:

***Table 221 Directory Property Values***

| **If you enter:** | **The action is:** |
|---|---|
| Auto Generate | (Default) The C++ Generator produces a directory name based on the name of the subsystem. This name is the name of the module, shortened to no more than the maximum number of characters permitted by the operating system, or 32, whichever is less. Shortening names can result in name conflicts. If this happens, you must specify a name explicitly for all but one of the conflicting modules. The subsystem's directory is created as a subdirectory of the project directory, as specified by the Directory project property. |
| *literal* | The C++ Generator produces a directory for the subsystem with the name *literal*. The subsystem's directory is a subdirectory of the project directory. *literal* must be a valid directory name. If *literal* includes a path, the path must be relative, not absolute. |
| <blank> | The C++ Generator does not create a subdirectory for the subsystem. Any files generated for the subsystem are created in the enclosing subsystem's subdirectory, if any; otherwise, the files are created in the project directory. |

If the model containing this property is supporting multiuser development and thus must contain only relative paths, use virtual symbols from your Path Map to construct the directory path.

## DirectoryIsOnSearchList

When a `#include` is generated in a module for which IncludeBySimpleName is False and that `#include` references a module in this subsystem, the path that appears in that `#include` will be relative to the directory associated with this subsystem.

The following table lists the values for DirectoryIsOnSearchList:

***Table 222 DirectoryIsOnSearchList Values***

| If you enter: | The action is: |
|---|---|
| True | The path that appears in the `#include` is relative to the directory associated with this subsystem. |
| False | The path that appears in the `#include` must be an absolute path. |

## PrecompiledHeader

The PrecompiledHeader property specifies the file name for the precompiled header file used in the model. The C++ code generator uses the specified file name to locate a module in the model whose FileName property yields to the same name.

It then sets up the closure of the precompiled header file from the IncludeClosure property of the module and uses that for the code generation.

The following table lists the values for PrecompiledHeader:

***Table 223 PrecompiledHeader Values***

| If you enter: | The action is: |
|---|---|
| <string> | The C++ code generator sets up the closure of the precompiled header based on the file <string>. |
| <blank> | (Default) There is no precompiled header in the model. |

# *Appendix K*

# *Symbols*

The intended use of these symbols is that $mode, $ordered, $keyCount, $cardinality, and (possibly) $keyn can be used to form the name of the template or preprocessor macro and the remaining symbols can be used as arguments to the template instantiation or macro call.

***Note:*** *Because a C++ constructor call and a preprocessor macro call are syntactically indistinguishable and because the Rational Rose code generator sometimes needs to modify constructor calls, a special syntax must be used to pass a macro call through the code generator unmodified. To protect the macro call, don't delimit the macro's argument list with parentheses, but use the special character sequence "<{" and ">}" instead. These special character sequences will be converted to parentheses before being placed in the generated code. Using this convention, the generic example written:*
***$ordered${cardinality}Bag$mode<{$qualcont,$keys $bounds}>***

generates macro calls rather than template instantiations.

For example:

```
$ordered${cardinality}Bag$mode< $qualcont,$keys $bounds >
```

***Note:*** *Exactly one of $keys or $bounds will always be null so a syntactically correct instantiation is always generated.*

## $mode

The $mode symbol indicates whether the mode is by reference or by value.

The possible values you can enter are By Reference or By Value.

## $ordered

The $ordered symbol indicates whether the ordered constraint has been specified in the Role specification.

The possible values you can enter are Ordered or null.

## $cardinality

The $cardinality symbol corresponds to the cardinality **`0…1, 1`** or **`1…1,`** *x*…*x*, for any integer, $x > 1$, *x*…*y*, for any integers $y > x > 1$, **`0…n`**, **`1…n`** or **`n`** , or when keys are defined.

The possible values you can enter are Optional, One, Fixed, Bounded, Unbounded, and Qualified.

## $keyCount

The $keyCount symbol indicates the number of qualifiers or keys specified for the role.

The possible values are any non-negative integer.

## $keyn

The $keyn symbol identifies the code name for the key for the first nine keys in a qualified role.

The possible values are an identifier or null.

## $typen

The $typen symbol indicates the key type for the first nine keys in a qualified role.

The possible values are C++ type specification or null.

## $types

The $types symbol lists all of the key types in the form of an argument list to a template instantiation.

The possible values are null or a list of C++ type specifications separated by commas.

## $upper

For an unqualified association, the $upper symbol identifies the upper bound in the cardinality specification. It is the same as **`$limit`**. You must **`#define _Unbounded`** to be something meaningful in your implementation.

The possible values are integer, "_Unbounded", or null. This is null for qualified associations.

## $lower

For an unqualified association, the $lower symbol identifies the lower bound in the cardinality specification. It will be the same as **`$upper`** if only one bound is specified.

The possible values are integer, "_Unbounded", or null. This is null for qualified associations.

## $bounds

The possible values for the $bounds symbol are **`$lower`**, **`$upper`** or null.

## $targetClass

The $targetClass symbol is the same as **`$supplier`** unless there is an association class associated with the association, in which case the **`$targetClass`** is the association class.

The possible values are the fully qualified name of the class of elements to be stored in the container.

## $target

The $target symbol is the code name for **`$targetClass.`**

## $supplier

The $supplier symbol is always the class associated with the role opposite the role being coded—the class on the opposite end of the association from the end being implemented.

The possible values are the fully qualified name of the class on the other end of the relationship.

## $limit

For an unqualified association, the $limit symbol identifies the upper bound in the cardinality specification. It is the same as **`$upper`**. You must **`#define _Unbounded`** to be something meaningful in your implementation.

The possible values are integer, “_Unbounded”, or null. This is null for qualified associations.

## $qualcont

In a qualified association, the $qualcont symbol indicates the type of the elements selected by a particular set of key values—the first argument and result type for the qualified set and get functions, respectively. This is the QualifiedContainer class specified for the role as selected by the cardinality of the supplier role.

In an unqualified association this is the same as **`$targetClass`**.

For complete generality, any expression in the Container Class property should use **`$qualcont`** as the type of the element stored in the collection defined by that Container Class property.

The possible value is the C++ type specification.

## $qualname

The $qualname symbol is the key name. It is the same as $key1.

## $qualtype

The $qualtype symbol is the key type. It is the same as $type1.

## $starIfByRef

The $starIfByRef symbol is “**`*`**” only if **`$mode`** is By Reference.

The possible values are **`*`** or Null.

## $commaIfKeys

The $commaIfKeys symbol is “**`,`**” only if **`$keyCount`** > 0.

The possible values are “,” or Null.

## $commaIfNoKeys

The $commaIfNoKeys symbol is “`,`” only if **`$keyCount`** = 0.

The possible values are “,” or Null.

# *Appendix L*

# *Analyzer Setup*

This section outlines how you should use Rational Rose C++ with Microsoft Visual C++. It includes information on how to configure your Analyzer, start a new project, and add classes, data members, and member functions to your project.

## Configuring the Analyzer

To use Rational Rose C++ with Microsoft Visual C++, you must set the precompiled header file to stdafx.h using the **Precompiled Header** dialog from the **Edit** menu.

## Preference Settings for New C++ Features

New reserved words are required to support the new C++ features. The Analyzer controls whether a feature is supported through Analyzer preferences. If you choose to use the latest set of enhancements, set the following preferences to be compatible with the features supported by your compiler. This is done from the **Precompiled Header** dialog, which you reach from the **Edit** menu.

***Table 224 Preference Settings***

| **Preference Option** | **Introduces as reserved words or punctuation...** |
|---|---|
| NamespacesOK=True | **`namespace, using`** |
| TypeBoolOK=True | **`book, true, false`** |
| TypeWCharTOK=True | **`wchar_t`** |
| MutableOK=True | **`mutable`** |

| Preference Option | Introduces as reserved words or punctuation... |
|---|---|
| TypeIdOK=Trye | **typeid** |
| NewStyleCaseOK=True | **const_cast, dynamic_case, static_cast, reinterpret_cast** |
| AlternativeOperators OK=True | **bitand** (&), **and** (&&), **bitor** (\|), **xor** (^), **compl** (~), **and_eq** (&=), **xor_eq** (^=), **not** (!), **not_eq** (!=) |
| CatchOK=True | **catch, try** |
| ExplicitOK-True | **explicit** |
| ExportOK=True | **export** |
| TrigraphicsOK=True | **??=** (#), **??/** (\\), **??’** (^), **??(** ({), **??)** (}), **??!** (\|), **??<** ({), **??>** (}), **??-** (~) |
| DigraphsOK=True | **<%** ({), **%<** (}), **<:** ([), **:>** (]), **%:** (#), **%:%:** (##) |

To turn any of these features off, set the value to **False** instead.

## Starting a New Project

To start development of a new Visual C++ application, do the following:

- Use one of the Application Wizards in Visual C++ to create the initial set of files for your application.
- Create a new Analyzer project and populate it with the files created by the Application Wizard.
- All Wizard-generated header files, except for **stdafx.h,** should be established as Type 2 files—they all assume the presence of **stdafx.h** in the compilation unit, but do not include it directly.
- The Regenerate attribute for the following files should be turned off. They contain no classes and should not be regenerated by Rational Rose or Code Cycled by the Analyzer: **stdafx.h, stdafx.cpp**, and **resource.h**.
- Now Code Cycle all files and export an initial model to Rational Rose using the RoundTrip export options.

- Open the **`.red`** file in Rational Rose.
- The reverse-engineered model has no properties associated with it, so execute the Replace option from the **Properties** tab on the **Tools** dialog box and select the **`rosevcpp.pty`** file as the new property file.
- Save the model into a **`.mdl`** file.

## Adding Classes, Data Members, and Member Functions

To add a class to a Visual C++ application:

1. Use the Class Wizard in Visual C++ to create the new class if it will be using any of the mechanisms supported by the Class Wizard, such as message maps or ActiveX functions, properties or events; otherwise create the class in Rational Rose, generate code, and add the new files to the Visual C++ project.
2. After the Class Wizard has created the files for the class, add them to the Analyzer project.
3. New header files should be established as Type 2 files.
4. The Regenerate attribute should be turned off for any files created by the Class Wizard as wrapper classes for ActiveX controls. These files are linked to pre-compiled code and must not be changed by you, Rational Rose, or the Analyzer.
5. Code Cycle the project files to bring the Analyzer database up to date with the changes.
6. Export the files that have changed using a modified version of the RoundTrip export option set.
7. Modify the RoundTrip export option set
   a. Change the name of the class diagram, or turn off the creation of class diagrams altogether. Usually when you update the model in this fashion, you are reverse-engineering only a portion of the full model. The class diagram generated will reflect only the portion reverse-engineered. Most likely you will want to throw the diagram away once you have ruminated a while on its contents. If you don't change the name of the class diagram, Rational Rose will replace the diagram for the full model with this partial one.
   b. Change the name of the module diagram, but do not turn its creation off entirely. You can suppress the creation of subsystems, however. You need to transfer the new file

information into the model. This can only be done through the creation of a module diagram. By suppressing the creation of subsystems, you will get only one diagram. After updating the model, delete this diagram.

8. Click **File > Update**.

Adding a new method or data member to an existing file proceeds the same as adding a new class. Use the Class Wizard to add methods and data members normally handled by the Class Wizard, then Code Cycle and Export via the Analyzer, and Update the Rational Rose model. Use Rational Rose to add other methods and data items, then regenerate code and compile and link using Visual C++.

## Rational Rose Project Files

A Rational Rose project file is a line-oriented ASCII file. The only truly cryptic entries in a project file are the encodings for window and page layouts and export options. These components are not critical and can, in fact, be omitted entirely.

The first few characters of each line identify the line's contents. The only required line in a project is the first line, which must begin with the version number of the project file format:

```
#v001>
```

The rest of the line is the project caption.

With the few exceptions noted below, the line prefix takes the form "#xxxx>", where xxxx is a unique 4-character abbreviation for the information contained on that line. the allowable prefixes are:

#stup> – Description of virtual symbol used in project

#dlft> – Name of default view

#dpfx> – Prefix for default data directory

#dsfx> – Suffix for default data directory

#xnam> – Name of export set

#xopt> – Export options

#xddd> – Default design file

#dt2p> – Type 2 #define symbol prefix

#t2fc> – Line in the type 2 context file

#flf>, #sky>, #hky> #dky>, and #flf> – Define a file list display-sort option

#lyt>, #ste> #brs>, and #lyt> – Define a project window layout

#ssys> – Subsystem name

## Exceptions

Line entries that are prefixed by Unix-like command line options, such as -D or +X.

+C <category name>=<unit name>=<subsystem name>

-I <searched source lib>=<data lib>

-I!<unsearched source lib>=<data lib>

+I <non-regenerated searched source lib>=<data lib>

+I!<non-regenerated unsearched source lib>=<data lib>

+X*<source body extension>=.<data extension>

+X!<source header extension>=.<data extension>

-D<#define symbol definition>

-U<#undef symbol>

@<name of base project>

-X<type 3 file name>=<header/body/regenerate code>

-Y<type 2 file name>=<header/body/regenerate code>

The <header/body/regenerate code> is a string of letters indicating the file-level override for the file, where: L = don't regenerate, R = regenerate, B = body file, S = header file.

A line with no prefix indicates a type 1 file name, which may also be followed by the sequence =<header/body/regenerate code>.

A file is assigned to the last mentioned category and subsystem. Thus the category and subsystem lines (+C and #ssys>) can be repeated as necessary to assign the files appropriately.

# Index