IBM Rational Rhapsody Properties

Every model element in Rational Rhapsody has a set of properties associated with it which can be accessed through the features window of Rational Rhapsody. These properties can be used in your generation rules by specifying which properties you would like to have for which model element type.

C_ReverseEngineering

In addition to the ReverseEngineering subject, Rational Rhapsody provides language-specific subjects to control how Rational Rhapsody imports legacy code. Most of the properties are identical for each language. Any language-specific properties are clearly labeled. In general, most of the reverse engineering properties have graphical representation in the Reverse Engineering Advanced Options window. You should change the options by using the Reverse Engineering Advanced Options window instead of the corresponding properties.

C_ReverseEngineering::**Filtering**

The Filtering metaclass contains properties that control which items are analyzed during the reverse engineering operation.

C_ReverseEngineering::Filtering::**AnalyzeGlobalFunctions**

```
The AnalyzeGlobalFunctions property specifies whether to analyze global
functions.
```

```
Default = Checked
```

C_ReverseEngineering::Filtering::**AnalyzeGlobalTypes**

```
The AnalyzeGlobalTypes property specifies whether to analyze global
types.
```

```
Default = Checked
```

C_ReverseEngineering::Filtering::**AnalyzeGlobalVariables**

```
The AnalyzeGlobalVariables property specifies whether to analyze global
variables.
```

```
Default = Checked
```

C_ReverseEngineering::Filtering::**CreateReferenceClasses**

```
The CreateReferenceClasses property specifies whether to create
external classes for undefined classes that result from forward
declarations and inheritance. By default, reference classes are created
(as in earlier versions of Rational Rhapsody). If the incomplete class
cannot be resolved, the tool deletes the incomplete class if this
property is set to Cleared. In some cases, the class cannot be deleted
(for example, a class referenced by a typedef type).
```

```
Default = Checked
```

C_ReverseEngineering::Filtering::**IncludeInheritanceInReference**

```
The IncludeInheritanceInReference property specifies whether  to
include inheritance information in reference classes.
```

```
Default = Cleared
```

C_ReverseEngineering::Filtering::**ReferenceClasses**

```
The ReferenceClasses property specifies which classes to model as
reference classes. Reference classes are classes that can be mentioned
in the final design as placeholders without having to specify their
internal details. For example, you can include the MFC classes as
reference classes, without having to specify any of their members or
relations. They would simply be modeled as terminals for context, to
show that they are acting as superclasses or peers to other classes.
```

```
Default = empty string
```

C_ReverseEngineering::Filtering::**ReferenceDirectories**

```
The ReferenceDirectories property specifies which directories (and
subdirectories) contain reference classes.
```

```
Default = empty string
```

C_ReverseEngineering::**ImplementationTrait**

The ImplementationTrait metaclass contains properties that determine the implementation traits used during the reverse engineering operation.

C_ReverseEngineering::ImplementationTrait::**AnalyzeIncludeFiles**

```
The AnalyzeIncludeFiles property specifies which, if any, include files
should be analyzed during reverse engineering. The possible values are
as follows:
```

```
    ▪ AllIncludes - Analyze all include files.
    ▪ IgnoreIncludes - Ignore all include files.
    ▪ OnlyFromSelected - Analyze the specified include files only.
    ▪ OnlyLogicalHeader - Analyze the logical header files only.
```

```
(C Default = AllIncludes)
```

C_ReverseEngineering::ImplementationTrait::**AutomaticIncludePath**

```
When Rational Rhapsody reverse engineers a file, there might be cases
where the file references a header file but the path in the include
directive is not clear enough for the product to find the file. If you
set the value of the AutomaticIncludePath property to Checked, then in
such cases, Rational Rhapsody searches the list of files to be reverse
engineered to see if the list contains a header file with that name. If
```

there is such a file, Rational Rhapsody uses the full path that was provided for that header file, assuming that this is the header file that was being referenced in the original file.

Rational Rhapsody performs this search for ambiguous header files when it does macro collection. This means that if the value of the C_ReverseEngineering::ImplementationTrait::CollectMode property is set to None, then Rational Rhapsody doesl not search for ambiguous header files even if the value of the AutomaticIncludePath iproperty s set to Checked.

Default = Checked

C_ReverseEngineering::ImplementationTrait::**CreateBlackDiamondAssociations**

The CreateBlackDiamondAssociations property specifies how the reverse engineering feature should handle composition relationships. If the value of the property is set to False, then Rational Rhapsody creates parts. If the value of the property is set to Checked, Rational Rhapsody creates composition associations (black diamond).

Default = Cleared

C_ReverseEngineering::ImplementationTrait::**CreateDependencies**

The CreateDependencies property is used during reverse engineering (RE) for creating dependencies from include statements found in the imported code. This property determines whether the RE utility creates dependencies. Reverse engineering imports include statements as dependencies if the option Create Dependencies from Includes is set in the Rational Rhapsody GUI. This operation is successful if the reverse engineering utility analyzes both the included file and the source - and the source and included files contain class declarations for creating the dependencies between them. If there is not enough information, the includes are not converted dependencies. This can happen in the following cases:

- The include file was not found, or is not in the scope Input tab settings.
- A class is not defined in the include file or source file, so the dependency could not be created.

If the dependency is not created successfully, the include files that were not converted to dependencies are imported to the C_CG::Class::SpecIncludes or ImpIncludes properties so you do not have to re-create them manually. If the include file is in the specification file, the information is imported to the SpecIncludes property; if it is in the implementation file, the information is imported to the ImpIncludes property. If a file contains several classes, include information is imported for all the classes in the file. The possible values for this property are as follows:

- None - Nothing is imported from include statements.
- DependenciesOnly - Model dependencies are created from include statements when it is possible to do so. This is the RE behavior of previous versions of Rational Rhapsody.
- All - The reverse engineering utility attempts to map the include file as a dependency. If it fails, the information is written to a property.

In previous versions of Rational Rhapsody, this property was a Boolean value. For compatibility with earlier versions, the old values are mapped as follows:

Old Value	New Value
Checked	DependenciesOnly
Cleared	None

1. In addition to influencing reverse engineering, the CreateDependencies property also impacts the reverse engineering of user code added to model elements. The rules for interpreting #include and friend declarations for reverse engineering are as follows:

- Any #include OTHER in FILE is represented as a Uses dependency between each (outer) packages or classes in FILE to any (outer) packages or class in OTHER.
- If OTHER is not a specification file, the information is lost.
- If FILE is a specification file, the RefereeEffect is Specification. If FILE is an implementation file, the RefereeEffect is Implementation. Otherwise, the information is lost.

2. Any forward of a class or a package (by way of a namespace) E in FILE is represented as a Uses dependency between each (outer) packages/classes in FILE to E. The RefereeEffect is Existence.

3. This dependency is not added, if a Uses dependency can be matched.

4. Redundant Uses dependencies are removed. For example, when a relation is synthesized from a pointer to B, it is not necessary to add a Uses dependency.

5. A friend F (only when F is a class) of class C is represented as a dependency with DependencyType to be Friendship from F to C.

Default = All)

C_ReverseEngineering::ImplementationTrait::**CreateFilesIn**

The CreateFilesIn property is a placeholder for the reverse engineering option Create File-s In option. See the Rational Rhapsody help for more information. You should not set this value directly. The default value for C is Package.

C_ReverseEngineering::ImplementationTrait::**CollectMode**

The CollectMode property allows Rational Rhapsody to collect macros. The possible values are as follows:

- None - Macros are not collected from include files that are not on the reverse engineering list.
- Once - Macros are collected only if the model does not yet include a controlled file of collected macros.
- Always - Macros are collected each time reverse engineering is carried out. The controlled file that stores the macros are replaced each time.

(C Default = None)

C_ReverseEngineering::ImplementationTrait::**DataTypesLibrary**

The Mapping tab of the Reverse Engineering Options window allows you to specify a list of types that should be modeled as "Language" types. You can add individual types to the list or groups of types that you have previously defined as data types for a specific library.

If you select the option of adding a library, you are presented with a drop-down list of libraries to choose from. The libraries on this list are taken from the value of the DataTypesLibrary property. You can add a number of libraries to the drop-down list by using a comma-separated list of names as the value for this property.

When you select a library from the drop-down list, all of the types that were defined for that library are added to the list of types.

You define types for a library by carrying out the following steps:

- In the relevant .prp file, under the C_ReverseEngineering subject, add a metaclass with the name of the library (use the same name you used in the value of the DataTypesLibrary property).
- Under the new metaclass, add a property called DataTypes.
-

For the value of the DataTypes property that you added, enter a comma-separated list of the types that you want to include for that library.

- Now, if you select the library from the drop-down list displayed on the Mapping tab, the types you defined with the DataTypes property is automatically added to the list of types that should be modeled as "Language" types.

Default = Blank

C_ReverseEngineering::ImplementationTrait::**ImportAsExternal**

The ImportAsExternal property specifies whether the elements contained in the files you are reverse engineering should be brought into the model as "external" elements. This means that code is not generated for these elements during code generation.

This property corresponds to the Import as External check box on the Mapping tab of the Reverse Engineering Options window.

Default = Cleared

C_ReverseEngineering::ImplementationTrait::**ImportDefineAsType**

The ImportDefineAsType property is a Boolean value that specifies how to import a #define. Note that models created before Version 5.2 automatically have this property overridden (set to True) when the model is loaded. The possible values are as follows:

- True - Import a #define as a user type.
- False - Import a #define as a constant variable, constant function, or type according to the following policy:
- If the #define has parameters, Rational Rhapsody creates a constant function. This applies to Rational Rhapsody Developer for C only.
- If the #define does not have parameters and its value includes only one line, Rational Rhapsody creates a constant variable. In Rational Rhapsody Developer for C++, the CG::Attribute::ConstantVariableAsDefine property is set to True.
- If the #define was not imported as a variable or function, Rational Rhapsody creates a type (the behavior of Rational Rhapsody 5.0.1).

Default = False

C_ReverseEngineering::ImplementationTrait::**ImportGlobalAsPrivate**

The ImportGlobalAsPrivate property allows you to import C functions as public or private.
The possible values are as follows:

- Never - Import globals (functions) as public. The declaration remains in the specification file.
- InImplementation - Global functions are imported as private. Both the declaration and the implementation of the function are imported into the implementation (.c) file.
- StaticInImplementation - Globals are imported as private in the implementation (.c) file and the functions are marked as static. (same as "InImplementation" but the keyword "static" is added to the declaration and implementation of the function).

C_ReverseEngineering::ImplementationTrait::**ImportPreprocessorDirectives**

Before running reverse engineering or roundtrip operations, set the ImportPreprocessorDirectives property to preserve the order of preprocessor directives during code generation. This property is only used if the C_Roundtrip:General:RoundtripScheme property is set to the "Respect" scheme.

However, the order of #include and #define is always preserved regardless of the setting of the ImportPreprocessorDirectives property.

Default = Checked

C_ReverseEngineering::ImplementationTrait::**ImportStructAsClass**

The ImportStructAsClass property is a Boolean value specifies how structs in external code are imported during reverse engineering. The possible values are as follows:

- Checked - structs are imported as classes (as in Rational Rhapsody 5.0 and earlier).
- Cleared - structs are imported as types of kind Structure.

Default = Cleared

C_ReverseEngineering::ImplementationTrait::**LocalizeRespectInformation**

When reverse engineering code in Respect mode, Rational Rhapsody stores information such as the order of code elements so that when code is regenerated from the model, the code resembles as much as possible the original code.

When the LocalizeRespectInformation property is set to Checked, Rational Rhapsody stores this information as SourceArtifact elements below the relevant class. (These elements are not visible by default, but you can see them in the model if you set the value of the ShowSourceArtifacts property to True.)

If the value of the LocalizeRespectInformation property is set to Cleared, then Rational Rhapsody stores this "respect" information as File elements under the relevant Component.

Default = Checked

C_ReverseEngineering::ImplementationTrait::**MacroExpansion**

Early versions of Rational Rhapsody were not capable of importing macros in code such that they would be regenerated as macros. Rather, the code represented by the macro was stored in the model, and when the code was regenerated, the macro calls would be replaced with the relevant code.

Now, by default, Rational Rhapsody imports macros such that when the code is regenerated, the macro definition and macro calls are generated as they appeared in the original code that was reverse engineered.

If you would like the previous Rational Rhapsody behavior, that is, replacement of macro calls with the actual macro code, you can set the MacroExpansion property to Checked.

Note that the C_ReverseEngineering::Parser::ForceExpansionMacros property allows you to specify that individual macros should be expanded during reverse engineering even if the value of the MacroExpansion property is set to False.

Default = Cleared

C_ReverseEngineering::ImplementationTrait::**MapGlobalsToComponentFiles**

The MapGlobalsToComponentFiles property allows you to specify whether Rational Rhapsody should map global variables, functions, and types to component files, reflecting the original file location of these elements in the files that were reverse engineered. The property can take any of the following values:

- OnExternal - Global variables, functions, and types should be mapped to component files only if the user selected the reverse engineering option "Import as External"
- TypesOnly - Global types should be mapped to component files, but not global variables and functions
- TypesOnExternal - Only global types should be mapped to component files, and this should only be done if the user selected the reverse engineering option "Import as External"
- False - Global variables, functions, and types should not be

mapped to component files

Default = TypesOnExternal

C_ReverseEngineering::ImplementationTrait::**MapToPackage**

The MapToPackage property allows you to specify how the code elements you are reverse engineering should be divided into packages.

The property represents the options that appear in the Map to Package section of the Mapping tab in the Reverse Engineering Options window.

When the value of the property is set to Directory, a separate package is created for each subdirectory in the directory you have chosen to reverse engineer. The elements found in the files in each subdirectory is added to the package that corresponds to that subdirectory.

If you set the value of this property to User, then Rational Rhapsody puts all reverse engineered elements into a single package in the model. The name of the package is taken from the property C_ReverseEngineering::ImplementationTrait::UserPackage.

Default = Directory

C_ReverseEngineering::ImplementationTrait::**ModelStyle**

The ModelStyle property determines how model elements are opened in the browser after reverse engineering - by using a file-based functional approach or by using an object-oriented approach based on classes (the corresponding property values are Functional and ObjectBased).

This property corresponds to the Modeling Policy radio buttons on the Mapping tab of the Reverse Engineering Options window.

Note that for C++ and Java, the file-based approach can only be used for visualization purposes. Rational Rhapsody does not generate code from the model for elements imported that uses the Functional option. (Notice that in the Reverse Engineering Options window, you can only select the File radio button if you first select the Visualization Only

```
option.)
```

```
Default = Functional in RiC, ObjectBased in RiC++ and RiJ
```

C_ReverseEngineering::ImplementationTrait::**PackageForExternals**

```
   If the value of the UsePackageForExternals property is set to
Checked,  the Rational Rhapsody reverse engineering feature puts all
external elements in a separate package. You can control the name of
this package by changing the value of the PackageForExternals property.
```

```
   Default = Externals
```

C_ReverseEngineering::ImplementationTrait::**PreCommentSensibility**

```
  During reverse engineering, a comment that comes immediately before
the code for an element is considered a comment for that element, and
the comment text is brought into Rational Rhapsody as the description
for that element.
```

```
  The PreCommentSensibility property is used to specify the maximum
number of lines by which a comment can precede the code for an element
and still be considered a comment for that element. Any comment that
precedes an element by more than the number of lines specified is
considered a global comment.
```

```
  A value of 1 means that a comment must appear on the line prior to
the code for an element to be considered a comment for that element.
```

```
  Default = 2
```

C_ReverseEngineering::ImplementationTrait::**ReflectDataMembers**

```
  The ReflectDataMembers property determines how the visibility of
attributes is brought into the model when code is reverse engineered.
The property affects both the visibility of the attribute in the
regenerated code and the generation of get and set operations for the
attribute. The property can take any of the following values:
```

- None - The visibility used for attributes is the same as that specified in the code that was reverse engineered. However, Rational Rhapsody generates public get/set operations for the attributes regardless of the visibility specified.
- VisibilityOnly - The visibility used for attributes is the same as that specified in the code that was reverse engineered. In addition, Rational Rhapsody generates get/set operations for the attribute with the same visibility. For example, if the visibility for an attribute in the original code was private, the visibility is private in the regenerated code and the code also includes private get/set operations for the attribute.
- VisibilityAndHelpers - The visibility used for attributes is the same as that specified in the code that was reverse engineered. Rational Rhapsody does not generate get/set operations for the attribute if the original code did not contain such operations.

Note that when the property is set to VisibilityAndHelpers, get/set operations are not generated for attributes, and Rational Rhapsody does not generate any of its automatically-generated operations such as default constructors.

Default = VisibilityAndHelpers

C_ReverseEngineering::ImplementationTrait::**RespectCodeLayout**

The RespectCodeLayout property determines to what degree Rational Rhapsody attempts to save information about the code that is reverse engineered so that it is possible to match the original code when code is later regenerated from the model. The saved information includes:

- order of #includes and other code elements
- handling of preprocessor directives such as #ifdefs
- keeping macro calls as they were rather than expanding the macro in the regenerated code
- handling of global comments

The property can take any of the following values:

- None - Rational Rhapsody does not save information about the order of elements in the code that is imported, nor does it save the information necessary to regenerate all elements back to the files from which they were originally imported.
- Mapping - Rational Rhapsody saves partial information so that it can regenerate all elements back to the files from which they were originally imported.
- Ordering - Rational Rhapsody saves all the information it can so that the regenerated code matches the original code as much as possible. See the examples listed above.

Note that even if the value of this property is set to Ordering, Rational Rhapsody only attempts to match the regenerated code to the original code if the C_CG::Configuration::CodeGeneratorTool property is set to Advanced, which is the default value for that property.

Default = Ordering

C_ReverseEngineering::ImplementationTrait::**RootDirectory**

This property specifies the root directory for reverse engineering. This root directory might contain all the folders that should become package during the reverse engineering process. Rhapsody builds the package hierarchy according to the folder tree from the specified path.

Default = empty string

C_ReverseEngineering::ImplementationTrait::**UseCalculatedRootDirectory**

This property controls the use of the <lang>_ReverseEngineering::Implementation::RootDirectory property.

The possible values are:

- Never - Do not calculate the root directory.
- Always - Calculate the root directory and override the RootDirectory property.
- Auto - Ask the user if they want to override the value in the RootDirectory property if it is different from the calculated root directory. If the RootDirectory property is empty, Rational Rhapsody uses the calculated value without asking. This is the default value.

```
Default = Auto
```

C_ReverseEngineering::ImplementationTrait::**UsePackageForExternals**

```
When Rational Rhapsody generates code, it does not regenerate code for
elements that have been brought in as "external" elements. By default,
the reverse engineering feature puts all external elements in a
separate package in the model. You can change this behavior by changing
the value of the UsePackageForExternals property. When a separate
package is used, the name of the package is taken from the value of the
PackageForExternals property.
```

```
  Default = Checked
```

C_ReverseEngineering::ImplementationTrait::**UserDataTypes**

```
The UserDataTypes specifies classes to be modeled as data types. This
property corresponds to types entered in the Add Type window.
```

```
Default = empty string
```

C_ReverseEngineering::ImplementationTrait::**UserPackage**

```
  When reverse engineering files, Rational Rhapsody allows you the
option of having packages created for each subdirectory or having all
of the reverse-engineered elements placed in a single package. This
option is controlled by the property
C_ReverseEngineering::ImplementationTrait::MapToPackage.
```

```
  When MapToPackage is set to "User", you can use the UserPackage
property to provide the name that you would like Rational Rhapsody to
use for the single package that contains all of the reverse-engineered
elements.
```

```
  You can specify a nested package by using the following syntax:
package1::package2
```

```
  If the model already contains a package with the specified name, the
reverse-engineered elements are put in that package. If not, Rational
Rhapsody creates the package.
```

This property corresponds to the text field provided for the package name in the Map to Package section of the Mapping tab in the Reverse Engineering Options window.

Default = ReverseEngineering

C_ReverseEngineering::ImplementationTrait::**VisualizationUpdate**

The VisualizationUpdate property instructs Rational Rhapsody to use the code-centric approach during reverse engineering and roundtripping. This approach assumes that the code serves as the blueprint for the software, and that the visual modeling capabilities of Rational Rhapsody are being used primarily to visualize the code.

In general, in code-centric mode, Rhapsody allows more drastic code changes to be brought into the model, relative to the changes that are imported when using the model-centric mode.

Default = Checked (in code-centric settings)

C_ReverseEngineering::**Main**

The metaclass Main contains properties that define the file extensions used for filtering files in the reverse engineering file selection window, as well as properties that enable jumping to problematic lines of code by double-clicking messages in the Output window.

C_ReverseEngineering::Main::**ErrorMessageTokensFormat**

When errors are encountered during reverse engineering, they are displayed in the Rational Rhapsody Output window. If you double-click the error message, you are taken to the problematic line in the relevant source file.

This ability is made possible by the values provided for the ParseErrorMessage and ErrorMessageTokensFormat properties.

The value of the ParseErrorMessage property is a regular expression that extracts the relevant filename and line number information from the Rational Rhapsody-generated error message. The value of the ErrorMessageTokensFormat property is then used to interpret the information that was extracted from the error message.

The value of the ErrorMessageTokensFormat property consists of a comma-separated list of keyword-value pairs representing the number of tokens contained in the extracted information, which token represents the filename, and which token represents the line number.

Users should not change the value of this property.

Default = ToTalNumberOfTokens=2,FileTokenPosition=1,LineTokenPosition=2

C_ReverseEngineering::Main::**ImplementationExtension**

The ImplementationExtension property specifies the file extensions used to filter the list of files displayed in the Add Files window of the reverse engineering tool.

(C Default = .c)

C_ReverseEngineering::Main::**MakefileExtension**

This property specifies the list of file extensions that are displayed and analyzed in the Reverse Engineering window when "Makefiles" is selected.

In order to import another kind of makefile, use the property C_ReverseEngineering:MakefileImport:MakefileType and the set of properties under C_ReverseEngineering:MakefileUserDefined.

Default = mak,mk,makefile,gpj

C_ReverseEngineering::Main::**ParseErrorMessage**

When errors are encountered during reverse engineering, they are displayed in the Rational Rhapsody Output window. If you double-click the error message, you are taken to the problematic line in the relevant source file.

This ability is made possible by the values provided for the ParseErrorMessage and ErrorMessageTokensFormat properties.

The value of the ParseErrorMessage property is a regular expression that extracts the relevant filename and line number information from the Rational Rhapsody-generated error message. The value of the ErrorMessageTokensFormat property is then used to interpret the information that was extracted from the error message.

Users should not change the value of this property.

```
Default = "([a-zA-Z_]+[:0-9a-zA-Z_.\/]*)"[:][ ]*LINE[ ]*([0-9]+)
```

C_ReverseEngineering::Main::**SpecificationExtension**

The SpecificationExtension property is used to specify the filename extensions that should be used to filter files in the reverse engineering file selection window. This property is used in conjunction with the ImplementationExtension property.

You can specify a number of extensions. They should be entered as a comma-separated list.

Default = h,inl

C_ReverseEngineering::Main::**UseCodeCentricSettings**

The UseCodeCentricSettings property specifies whether the visualization result of running reverse engineering is in code-centric mode in Rational Rhapsody.

If this property is checked, the code-centric settings are added to the model during reverse engineering (if they do not already exist) and a dependency with the applied profile stereotype is added between the active component to the code-centric settings.

Default = Cleared

C_ReverseEngineering::**MFC**

The MFC metaclass contains a property that affects the MFC type library.

C_ReverseEngineering::MFC::**DataTypes**

The DataTypes property specifies classes to be modeled as MFC data types. There is only one predefined library (MFC) that contains only one class (Cstring). You can, however, expand this short list of classes by the addition of classes in this property or the creation of new libraries in the property files factory.prpfactory and site.prpsite.

```
Default = Cstring
```

C_ReverseEngineering::**MSVC60**

The MSVC60 metaclass contains properties used to control the Microsoft Visual C++ environment.

C_ReverseEngineering::MSVC60::**Defined**

```
The Defined property specifies symbols that are defined for the
Microsoft Visual C++ version 6.0 (MSVC60) preprocessor. These symbols
are automatically filled into the Name list of the Preprocessing tab of
the Reverse Engineering Options window when you select Add > Dialect:
MSVC60.
```

```
The default value is as follows:
```

```
 __STDC__,__STDC_VERSION__,__cplusplus,__DATE__,
__TIME__,_WIN32,_cdecl,__cdecl,__int64=int,__stdcall,
__export,_export,_AFX_PORTABLE,_M_IX86=500,__declspec,
__MSC_VER=1200,__inline=inline,__far,__near,_far,_near,
__pascal,_pascal,__asm,__finally=catch,__based,
__inline=inline,__single_inheritance,__cdecl,__int8=int,
__stdcall,__declspec,__int16=int,__int32=int,__try=try,
__int64=int,__virtual_inheritance,__except=catch,
__leave=catch,__fastcall,__multiple_inheritance)
```

C_ReverseEngineering::MSVC60::**IncludePath**

```
The IncludePath property specifies necessary include paths for the
Microsoft Visual C++ preprocessor. It is possible to specify the path
to the site installation of the compiler as part of the site.prp, thus
doing it only once and not for every project.
```

```
Default = empty string
```

C_ReverseEngineering::MSVC60::**Undefined**

```
The Undefined property specifies symbols that must be undefined for the
Microsoft Visual C++ preprocessor.
```

```
Default = empty string
```

C_ReverseEngineering::**Parser**

The metaclass Parser contains properties that can be used to modify the way the parser handles code during reverse engineering.

C_ReverseEngineering::Parser::**AdditionalKeywords**

```
The AdditionalKeywords property can be used to list non-standard
keywords that might appear in the code that you reverse engineer. This
allows Rational Rhapsody to parse this code correctly during reverse
```

engineering.

The value of this property should be a comma-separated list of the additional keywords you want to include.

Note that keywords with parameters are not supported, nor are keywords that consist of more than one word.

This property corresponds to the keywords listed on the Preprocessing tab of the Reverse Engineering Options window. Note that when you add additional keywords by using the controls on the Preprocessing tab, these keywords are included in the value of the AdditionalKeywords property at the level of the active configuration.

Default = far,near

C_ReverseEngineering::Parser::**Defined**

The Defined property specifies symbols and macros to be defined by using #define. For example, you can enter the following
to define name> as text with the appropriate intermediate character:
/D name{=|#}text

Default = empty string

C_ReverseEngineering::Parser::**Dialects**

The Dialects property specifies which symbols are added to the Preprocessing tab of the Reverse Engineering window
box when that dialect is selected. The default value is MSVC60, which is itself defined by a metaclass of the same name under subject C_ReverseEngineering. This dialect specifies the symbols that are defined for the Microsoft Visual C++ environment. You can define your own dialect (in the site.prp file) and select it in the Dialects property. The default value for C is an empty string.

C_ReverseEngineering::Parser::**ForceExpansionMacros**

By default, Rational Rhapsody reverse engineers macros such that when the code is regenerated, the macro definition and macro calls are generated as they appeared in the original code that was reverse engineered. (This behavior can be controlled with the C_ReverseEngineering::ImplementationTrait::MacroExpansion property.)

In some cases, you might find that you are not satisfied with the way that Rational Rhapsody imports the macro. For such situations, you can use the ForceExpansionMacros property to list specific macros that should be expanded during reverse engineering even if the value of the MacroExpansion property is set to False.

The value of this property should be a comma-separated list of the macros that you would like Rational Rhapsody to expand during reverse engineering.

Default = Blank

C_ReverseEngineering::Parser::**IncludePath**

The Preprocessing tab of the Reverse Engineering Options window allows you to specify an include path (classpath for Java) for the parser to use. The In propertycludePath represents this path.

For the value of this property, you can enter a comma-separated list of directories. Note that you have to specify subdirectories individually.

The directories you list here is combined with the directories specified in #include statements in order to find the necessary files. For example, if you have c:\d1\d2\d3\file.h, you can enter c:\d1\d2 as the value of this property and then use d3\file.h in the #include statement.

You should take into account that the value of this property also determines the structure of the source file directory when code is generated from the model. So, in the above example, the top-level directory created is d3.

Default = Blank

C_ReverseEngineering::Parser::**Undefined**

The Undefined property specifies symbols and macros to be undefined by using #undef.

Default = empty string

C_ReverseEngineering::**Promotions**

The metaclass Promotion contains a number of properties used to specify whether Rational Rhapsody should add various advanced modeling constructs to your model based on relationships/patterns uncovered during reverse engineering.

C_ReverseEngineering::Promotions::**EnableAttributeToRelation**

The EnableAttributeToRelation property is used to specify whether Rational Rhapsody should add Associations to the model for attributes whose type is another class in the model.

For example, if you have two classes, A and B, and B contains an attribute of type A, Rational Rhapsody adds an Association to the model reflecting this relationship.

Default = Checked

C_ReverseEngineering::Promotions::**EnableFunctionToObjectBasedOperation**

The EnableFunctionToObjectBasedOperation property specifies whether object-based promotion is enabled during reverse engineering. Object-based promotion “promotes” a global function to comply with the pattern specified in the C_CG::Operation::PublicName and C_CG::Operation::ProtectedName properties to be an operation of the class (object_type)
defined in the me parameter for the function.

Default = Cleared

C_ReverseEngineering::Promotions::**EnableResolveIncompleteClasses**

Sometimes, during reverse engineering, Rational Rhapsody is not able to find the base class for a given class. The EnableResolveIncompleteClasses property is used to specify that if Rational Rhapsody finds a class with the same name as the base class in a different location, it should assume that this class is the missing base class.

Default = Checked

C_ReverseEngineering::**Update**

The Update metaclass contains properties used to control various aspects of the Rational Rhapsody behavior during and after reverse engineering.

C_ReverseEngineering::Update::**CreateFlowcharts**

Use this property to specify whether or not Rational Rhapsody should automatically create flowcharts for operations during reverse engineering of code.

Set this property before you start the reverse engineering process.

Use this property in conjunction with the FlowchartCreationCriterion, FlowchartMinLOC, FlowchartMaxLOC, FlowchartMinControlStructures and FlowchartMaxControlStructures properties so that flowcharts are created only for operations that are within a given range in terms of lines of code or in terms of the number of control structures in the operation.

Default = Cleared

C_ReverseEngineering::Update::**FlowchartCreationCriterion**

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, you can use the FlowchartCreationCriterion property to select the criterion that should be used to decide what operations Rational Rhapsody should create flowcharts for.

The property can take the following values:

- Control Structures - the decision whether or not to generate a flowchart for an operation is based on the number of control structures in the operation. When this option is selected, the minimum and maximum number of control structures used to define the inclusion criterion are taken from the FlowchartMinControlStructures and FlowchartMaxControlStructures properties.
- LOC - the decision whether or not to generate a flowchart for an operation is based on the number of lines of code in the operation. When this option is selected, the minimum and maximum lines of code used to define the inclusion criterion are taken from the FlowchartMinLOC and FlowchartMaxLOC properties.

Default = LOC

C_ReverseEngineering::Update::**FlowchartMaxControlStructures**

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, and you have set the value of the FlowchartCreationCriterion property to Control Structures, then you can use the FlowchartMaxControlStructures property to specify the maximum number of control structures that an operation can have, above which Rational Rhapsody does not create a flowchart for it.

Default = 10

C_ReverseEngineering::Update::**FlowchartMaxLOC**

If you have set the CreateFlowcharts property to have Rational Rhapsody create flowcharts during reverse engineering, and you have set the value of the FlowchartCreationCriterion property to LOC, then you can use the FlowchartMaxLOC property to specify the maximum number of lines of code that an operation can have, above which Rational Rhapsody does not create a flowchart for it.

Default = 100

C_ReverseEngineering::Update::**FlowchartMinControlStructures**

```
If you have set the CreateFlowcharts property to have Rational Rhapsody
create flowcharts during reverse engineering, and you have set the
value of the FlowchartCreationCriterion property to Control Structures,
then you can use the FlowchartMinControlStructures property specify the
minimum number of control structures that an operation must have in
order to have Rational Rhapsody create a flowchart for it.
```

```
Default = 2
```

C_ReverseEngineering::Update::**FlowchartMinLOC**

```
If you have set the CreateFlowcharts property to have Rational Rhapsody
create flowcharts during reverse engineering, and you have set the
value of the FlowchartCreationCriterion property to LOC, then you can
use the FlowchartMinLOC property to specify the minimum number of lines
of code that an operation must have in order to have Rational Rhapsody
create a flowchart for it.
```

```
Default = 10
```

Feedback