# Generating code with IBM® Rational® Rhapsody®

Last update: April 2014

# Table of Contents

Introduction....4
Basic code generation concepts....5
  Components and Configurations....5
    Components....5
    Configurations....6
    Active components and configurations....7
    Specifying objects to create when application is run....7
    Directory structure of generated code....8
    Generation of instrumentation code - animation and tracing....9
  Generating code for different parts of a project....10
  Triggering code generation – manually and automatically....11
  Check model....12
  Output window - log tab....13
  Viewing generated code....14
  Code generation performance....15
    Parallel code generation....15
    Breaking code generation into chunks....15
  Customizing code generation with the Features window and properties....17
Customizing file headers and footers....18
Including comments in the generated code....19
  Element descriptions....19
  Including requirements as comments in the generated code....19
Code formatting....20
  Curly brace style....20
  Controlling indentation....20
  Generating operation arguments on separate lines....21
Controlling the order of elements in the generated code....22
  Changing the order of attributes in generated code....22
  Changing the order of operations in generated code....22
Code generated for dependencies....23
  Controlling how usage dependencies are represented in the code (C, C++)....23
  Code generated for friend dependencies....23
Generating standard operations....24
Statechart code....25
Language-specific code generation features....26
  C and C++....26
    Generating classes as structs....26
    Friend classes....26
    Wrapping code with #ifdef-#endif....26
    Generating operation descriptions in implementation files....27
    Fine-tuning code generated for arguments....27
    Fine-tuning code generated for return types....27
  Java....28
    Including Javadoc comments in the generated code....28
    Static imports....28

Defining static blocks........................................................................................29
Generating JAR files..........................................................................................29
Java 5 annotations.............................................................................................30
Multi-language projects...........................................................................................31
Using documentation generation tools such as Doxygen.........................................32
Code generation with the command line.................................................................33

# Introduction

One of Rational Rhapsody's most prominent features is its ability to generate code from your model. This includes both code that can be used for testing and debugging your model and production-level code. The code generation capability allows you to store your application as a Rational Rhapsody model and then generate code from the model as needed.

The purpose of this document is to introduce you to Rational Rhapsody's code generation capabilities, and to present the various options available to best adapt these capabilities to your needs.

In order to cover as many aspects of code generation as possible, this document will avoid going down to the level of step-by-step directions. Rather, it will address the various options available and point you to where you can find detailed descriptions in the help.

# Basic code generation concepts

## Components and Configurations

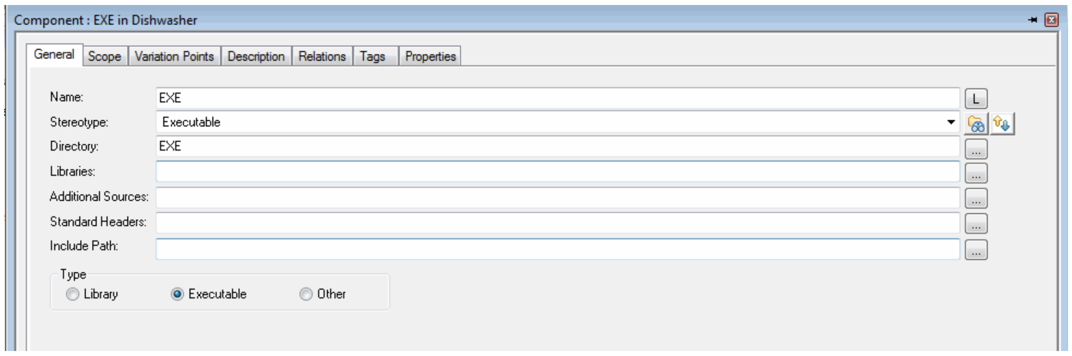
### Components

While container elements such as Packages are provided to help you group the elements in your model, the grouping of model elements for the purpose of code generation is carried out by assigning model elements to Components.

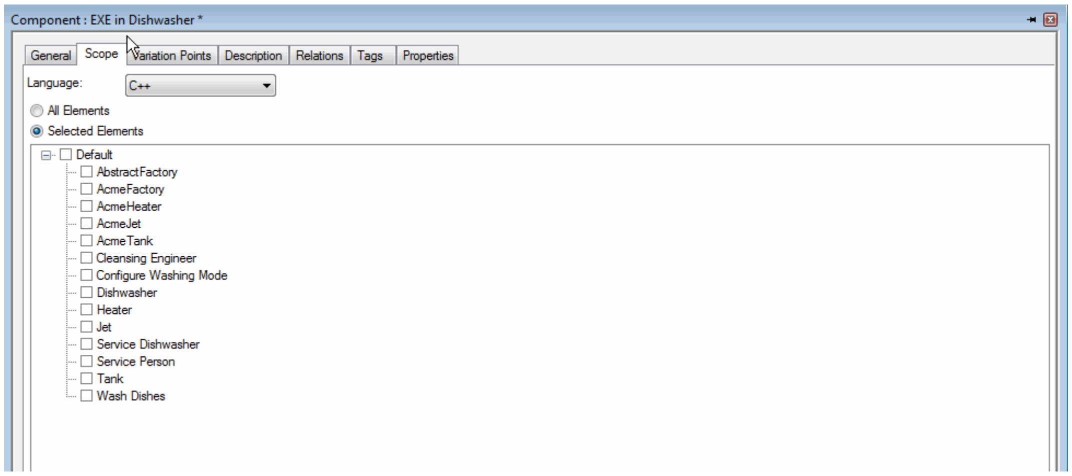
Components in your model are used to represent software components. This is reflected in the Features window for Components, where you can choose whether the Component represents an executable or a library.

On the Scope tab of the Features dialog, you indicate which elements of your model should be included in the Component so that they are grouped together both for code generation and when you build your application from the generated code.

By default, a new Component includes all the code-relevant elements in your model. To include only a subset of the model elements in the Component, you use the controls provided on the Scope tab in order to select only those elements that are relevant for the given software component.



Name: EXE
Stereotype: Executable
Directory: EXE
Libraries:
Additional Sources:
Standard Headers:
Include Path:
Type: Library [✓ Executable] Other



Language: C++
All Elements [✓ Selected Elements]

- Default
  - AbstractFactory
  - AcmeFactory
  - AcmeHeater
  - AcmeJet
  - AcmeTank
  - Cleansing Engineer
  - Configure Washing Mode
  - Dishwasher
  - Heater
  - Jet
  - Service Dishwasher
  - Service Person
  - Tank
  - Wash Dishes

## *Configurations*

For each Component, you can define multiple Configurations.

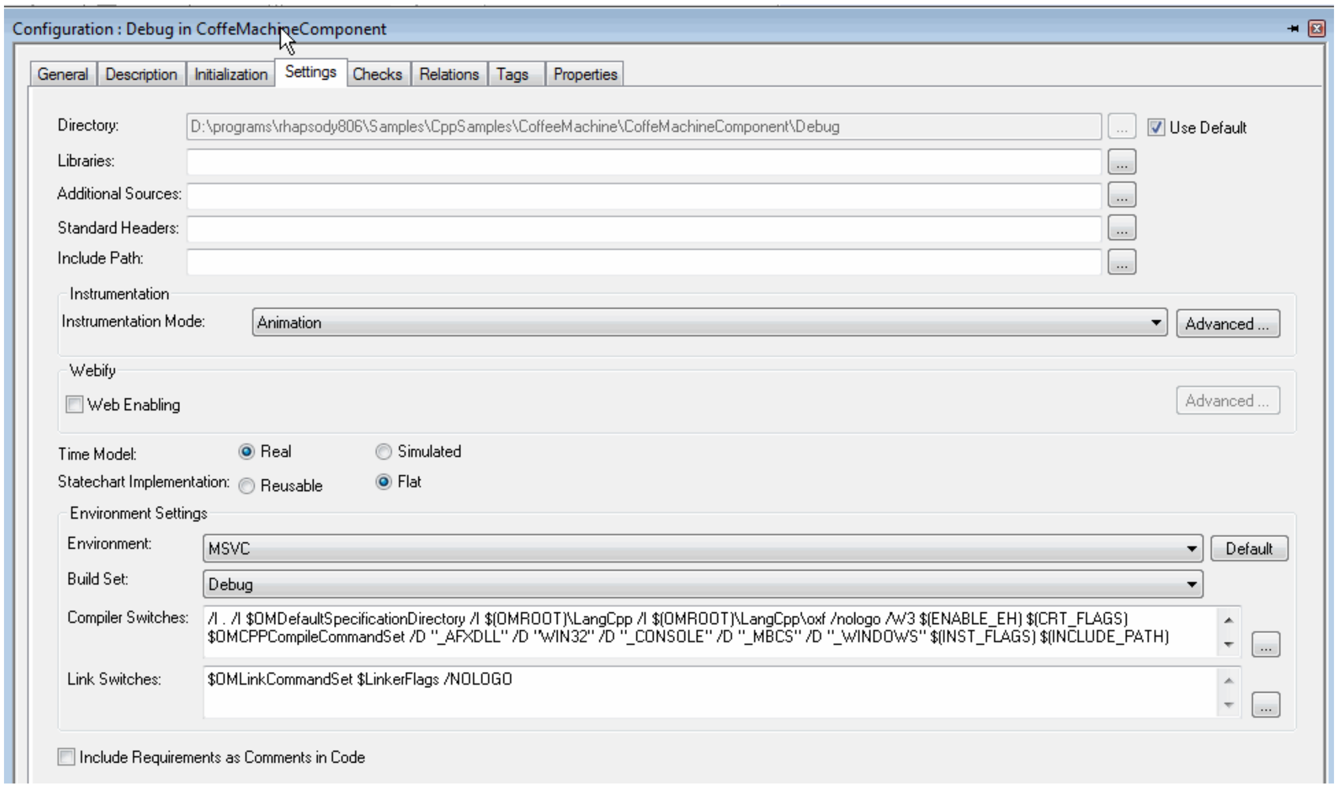
While Components are used to specify what elements code should be generated for, Configurations are used to instruct Rhapsody to generate the code differently for different purposes.

For example:

- You may want to have one Configuration to generate code that includes the instrumentation code that is necessary in order to animate your model in Rhapsody, and then use a second Configuration for generating your production code.
- You may want to have one Configuration for generating code for an application that will be run on a Windows-based system, and a second Configuration for generating code for an application that will be run on a VxWorks-based system.

If you look at the various options on the Settings tab of the Features window for Configurations, you'll probably discover some other ways in which you may want to use multiple Configurations for generating code for your model.

For each of your Configurations, Rhapsody generates an appropriate main file and makefile, based on the settings you have chosen.



Directory: D:\programs\rhapsody806\Samples\CppSamples\CoffeeMachine\CoffeMachineComponent\Debug ✓ Use Default

Libraries:

Additional Sources:

Standard Headers:

Include Path:

Instrumentation

Instrumentation Mode: Animation

Webify

☐ Web Enabling

Time Model: [✓ Real] Simulated

Statechart Implementation: Reusable [✓ Flat]

Environment Settings

Environment: MSVC

Build Set: Debug

Compiler Switches:

```
/I . /I $OMDefaultSpecificationDirectory /I $(OMROOT)\LangCpp /I $(OMROOT)\LangCpp\oxf /nologo /W3 $(ENABLE_EH) $(CRT_FLAGS)
$OMCPPCompileCommandSet /D "_AFXDLL" /D "WIN32" /D "_CONSOLE" /D "_MBCS" /D "_WINDOWS" $(INST_FLAGS) $(INCLUDE_PATH)
```

Link Switches:

```
$OMLinkCommandSet $LinkerFlags /NOLOGO
```

☐ Include Requirements as Comments in Code

## Active components and configurations

Since your model can contain multiple Components and multiple Configurations for each Component, you tell Rhapsody which Component and which Configuration to use for code generation by making one Component the "active" Component, and one Configuration the "active" Configuration.

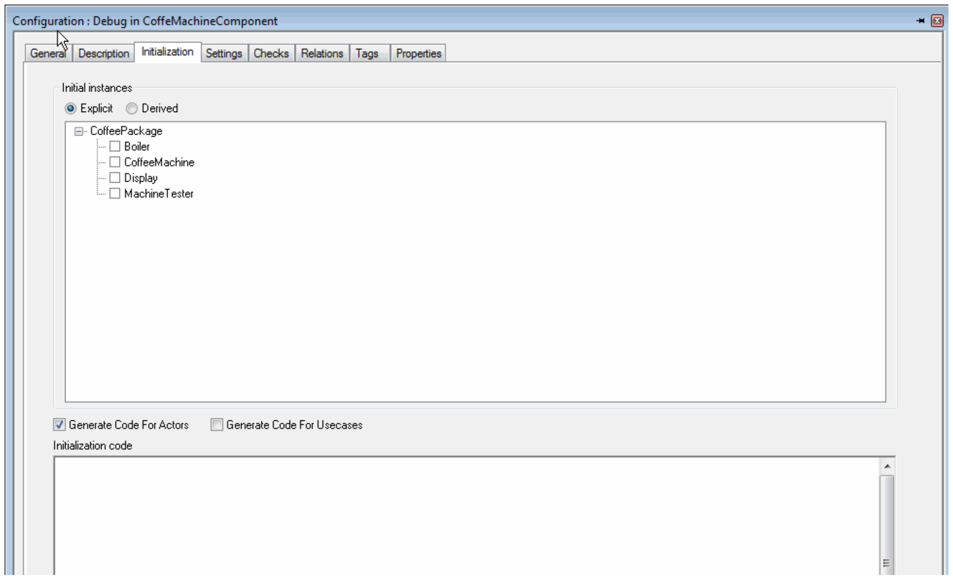
## Specifying objects to create when application is run

To specify the objects that you want to have created when the application is run, use the Initialization tab of the Features dialog for configurations.

The *Initial Instances* section of the tab lists all of the classes in the model. Use the check boxes to indicate the classes that should be instantiated when the application is run.

In addition to the *Initial Instances* section, you can use the *Initialization code* section of the tab to include your own code for creating objects when the application is run.

> *The Explicit and Derived options on the Initialization tab affect the code generation scope:*
> *If you select the Explicit option, code is generated for the classes that are specified on the Scope tab of the Features dialog for the code generation component.*
> *If you select the Derived option, code will be generated for the classes that you indicated should be instantiated, as well as for any classes which the selected classes require, for example classes that they have a usage dependency upon, or classes that they have an association to. The Derived option will also take into account a chain of dependencies. For example, if class A depends upon class B, and class B depends on class C, code will be generated for all three classes if you selected class A.*
> ***Note that when the Derived option is used, Rational Rhapsody ignores the code generation scope that was defined on the Scope tab of the Features dialog for the component.***



Initial instances

[✓ Explicit] Derived

- CoffeePackage
  - ☐ Boiler
  - ☐ CoffeeMachine
  - ☐ Display
  - ☐ MachineTester

☑ Generate Code For Actors ☐ Generate Code For Usecases

Initialization code

### *Directory structure of generated code*

By default, Rhapsody creates a subdirectory in your project directory for each Component. And under each Component, Rhapsody creates a directory for each Configuration that you have defined.

However, Rhapsody provides a number of settings that can be used to control the directory structure used for the generated code. This includes both:

- the ability to specify the output directory for code generation
- the ability to specify the directory structure within the output directory.

To specify the output directory for the generated code:

- Open the Features window for the relevant component.
- On the General tab, use the Directory field to specify the directory to which you want the code to be generated. You can enter a relative path (beneath the directory that contains your model), or an absolute path.

To specify the directory structure within the output directory, use one or both of the following properties:

- *[lang]_CG::Package::GenerateDirectory* - Set the value of GenerateDirectory to True to have a directory created for each package. The code generated for elements in the package will be generated to the directory representing that package.
- *[lang]_CG::Configuration::DefaultSpecificationDirectory* and *[lang]_CG::Configuration::DefaultImplementationDirectory* - These properties are available for C and C++ since these languages have separate specification and implementation files. Use these properties to specify the names of the subdirectories that should be used to separate the specification files from the implementation files. For example, you can have all the specification files generated to a subdirectory called *inc* and all the implementation files generated to a subdirectory called *src*. If you do not provide values for these properties, the specification and implementation files will be generated to the same directory.

  For C and C++ you can choose to use these properties in conjunction with the *GenerateDirectory* property. For example, if you specify "inc" as the specification directory and "src" as the implementation directory, and set the value of *GenerateDirectory* to True, you will end up with inc and src subdirectories under each package directory.

  *In addition to customizing the directory structure for the generated code, you can also customize the filename used for a generated file by modifying the value of the FileName property.*

If these standard options are not sufficient to create the directory structure you want for code generation, there is also an option to map individual model elements to specific output files and to map individual output files to specific directories. For more details, see Rational Rhapsody code generation: mapping elements to files and mapping files to folders.

## *Generation of instrumentation code - animation and tracing*

To use the animation and tracing features to test your application, select Animation or Tracing from the Instrumentation Mode drop-down list on the Settings tab of the Features window for configurations.

If you use animation or tracing, Rational Rhapsody must also generate the code that makes these features possible.

The following snippets of generated code illustrate this.

Code for configuration without animation:

```
Printer::Printer(IOxfActive* theActiveContext) {
  setActiveContext(theActiveContext, false);
  initStatechart();
}
```

Code for configuration that includes animation:

```
Printer::Printer(IOxfActive* theActiveContext) {
  NOTIFY_REACTIVE_CONSTRUCTOR(Printer, Printer(), 0, 
Default_Printer_Printer_SERIALIZE);
  setActiveContext(theActiveContext, false);
  initStatechart();
}
```

Since you can create multiple code generation configurations for each component in your model, you can create a configuration that includes the code necessary for animation or tracing, and a separate configuration which generates only the code necessary for your application.

## *Generating code for different parts of a project*

Rhapsody allows you to generate the code for your entire project at once.

However, as your model grows in size, it may not be very efficient to generate code for the entire project each time.

You can generate code for smaller slices of your project by selecting one of the following code generation options:

- the currently active configuration
- the currently active configuration, along with the corresponding configurations in all components on which the current component depends
- the currently selected classes (you can use the multiple-select feature to select multiple classes in the browser

## Triggering code generation – manually and automatically

You can use the various code generation options in the main menu and in the context menus in order to manually trigger code generation at any point in your work.

In addition, you can have code generation triggered automatically be using one of the DMCA options.

DMCA (Dynamic Model-Code Associativity) is designed to bind the model and code together such that changes made to the model are immediately reflected in the code if the code editor window is open. Similarly, changes made to the code in the code editor are automatically reflected in the model.

If you want to use DMCA, you can select from the following options:

- *Code Generation* – changes to the model are automatically reflected in the code but not vice versa
- *Roundtrip* – changes made directly to the code are automatically reflected in the model but not vice versa
- *Bidirectional* – changes to the model are automatically reflected in the code, and direct changes to the code are automatically reflected in the model

If you set DMCA to Code Generation or Bidirectional, then you do not have to trigger code generation manually if the relevant file is currently open in the code editor. The relevant code is generated or updated whenever you make a change to the model.

Note that even if you do not want to use DMCA on a regular basis, it is an excellent way to test what code changes will result from specific changes to the model – both changes made through the Features dialog and changes made by modifying the value of a property.

The most efficient way to use DMCA to test such model changes is to keep the relevant windows open simultaneously:

1. Set DMCA to Code Generation temporarily (if you do not always use it).
2. Open the section of code that represents the element you will be modifying.
3. Open the Features dialog to the relevant tab or find the relevant property in the Properties tab of the dialog.
4. Align the Features dialog and the code window so that you can see both simultaneously.
5. Make the change in the Features dialog and press Apply.
6. Click in the code window.

As soon as the code windows gets the focus, the code is updated to reflect the change.

## Check model

Rhapsody's Check Model feature checks your model for various types of modeling errors.

You can manually invoke model-checking at any time.

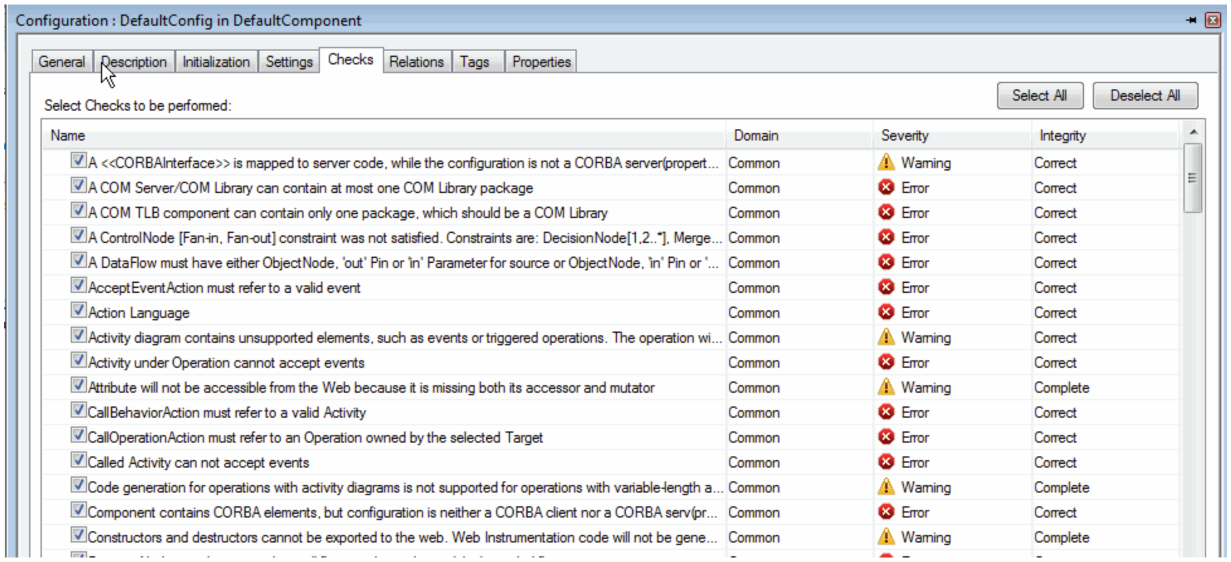
In addition, model-checking is automatically carried out before code is generated.

The Check Model tab in the Output window displays any problems found during model-checking. The problems detected are categorized as Errors or Warnings.

Some of these problems block the generation of code. For other problems, the details of the problem are displayed but code is still generated.

All of the model-checking results are also displayed in the Log tab of the Output window, however, the Check Model tab includes additional capabilities, such as the ability to jump to the problematic model element by double-clicking the relevant line in the Check Model results.

*For manually-invoked model checking, you can decide which of the predefined checks should be carried out. For the model check carried out prior to code generation, you can also control which checks should be carried out, however, some checks cannot be turned off.*



Select Checks to be performed:

| Name | Domain | Severity | Integrity |
|---|---|---|---|
| ✓ A <<CORBAInterface>> is mapped to server code, while the configuration is not a CORBA server(propert... | Common | ⚠ Warning | Correct |
| ✓ A COM Server/COM Library can contain at most one COM Library package | Common | ✖ Error | Correct |
| ✓ A COM TLB component can contain only one package, which should be a COM Library | Common | ✖ Error | Correct |
| ✓ A ControlNode [Fan-in, Fan-out] constraint was not satisfied. Constraints are: DecisionNode[1,2..*], Merge... | Common | ✖ Error | Correct |
| ✓ A DataFlow must have either ObjectNode, 'out' Pin or 'in' Parameter for source or ObjectNode, 'in' Pin or '... | Common | ✖ Error | Correct |
| ✓ AcceptEventAction must refer to a valid event | Common | ✖ Error | Correct |
| ✓ Action Language | Common | ✖ Error | Correct |
| ✓ Activity diagram contains unsupported elements, such as events or triggered operations. The operation wi... | Common | ⚠ Warning | Correct |
| ✓ Activity under Operation cannot accept events | Common | ✖ Error | Correct |
| ✓ Attribute will not be accessible from the Web because it is missing both its accessor and mutator | Common | ⚠ Warning | Complete |
| ✓ CallBehaviorAction must refer to a valid Activity | Common | ✖ Error | Correct |
| ✓ CallOperationAction must refer to an Operation owned by the selected Target | Common | ✖ Error | Correct |
| ✓ Called Activity can not accept events | Common | ✖ Error | Correct |
| ✓ Code generation for operations with activity diagrams is not supported for operations with variable-length a... | Common | ⚠ Warning | Complete |
| ✓ Component contains CORBA elements, but configuration is neither a CORBA client nor a CORBA serv(pr... | Common | ✖ Error | Correct |
| ✓ Constructors and destructors cannot be exported to the web. Web Instrumentation code will not be gene... | Common | ⚠ Warning | Complete |

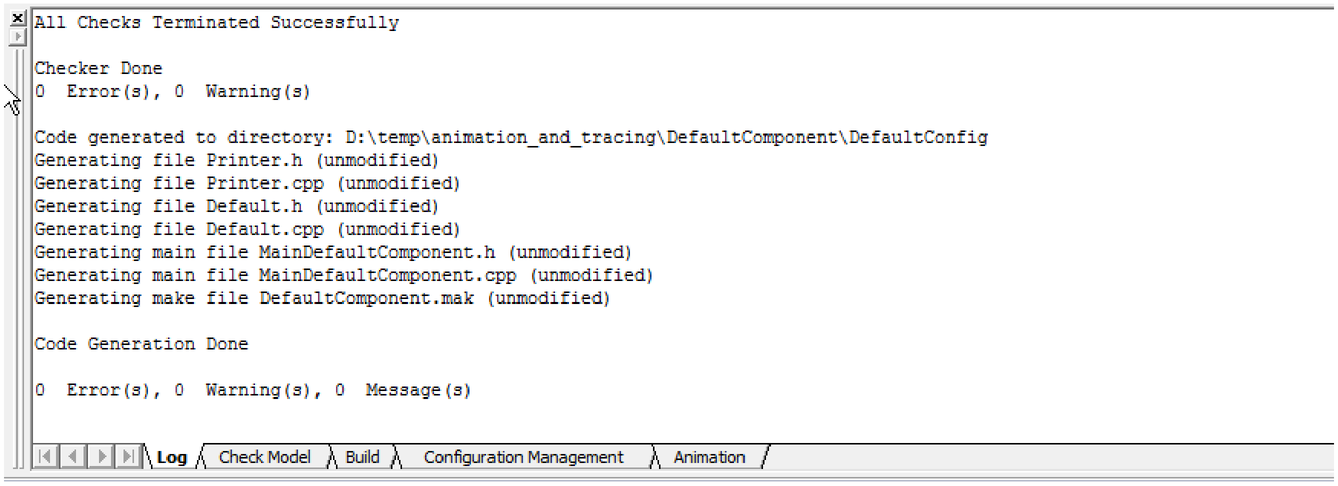
## Output window - log tab

Any time you generate code, Rhapsody displays messages in the log tab of the Output window.

These messages include:

- error messages (for errors that block code generation)
- warning messages (for issues that do not prevent code generation)
- code generation progress messages

Note that messages are displayed in the Output windows both for manually-invoked code generation and for automatic code generation triggered by the DMCA setting.



```
All Checks Terminated Successfully

Checker Done
0  Error(s), 0  Warning(s)

Code generated to directory: D:\temp\animation_and_tracing\DefaultComponent\DefaultConfig
Generating file Printer.h (unmodified)
Generating file Printer.cpp (unmodified)
Generating file Default.h (unmodified)
Generating file Default.cpp (unmodified)
Generating main file MainDefaultComponent.h (unmodified)
Generating main file MainDefaultComponent.cpp (unmodified)
Generating make file DefaultComponent.mak (unmodified)

Code Generation Done

0  Error(s), 0  Warning(s), 0  Message(s)
```

## *Viewing generated code*

Rhapsody contains an internal code editor that you can use to view generated code and to make changes to code.

To view the generated code in the internal editor, just select the relevant element in the browser, for example, a class, and select Edit Code from the pop-up menu.

You also have the option to specify an external code editor that should be opened whenever you select Edit Code. To specify the external editor to use, modify the value of the property General::Model::EditorCommandLine.

## Code generation performance

After initial code generation for a configuration, code is generated only for elements that were modified since the last generation.

This provides a significant performance improvement by reducing the time required for generating code.

Rational Rhapsody provides a number of options that can be used to further improve code generation performance. Depending on the size and nature of your model, you may want to use different combinations of these options.

### Parallel code generation

Beginning in version 8.0, the default code generation behavior is to use parallel processing in order to improve code generation performance. Parallel code generation is carried out by launching multiple RhapsodyCL processes.

Parallel code generation is used when you select one of the following code generation options:

- configuration "with dependencies"
- "entire project"

A number of properties are provided to allow you to:

- determine how many parallel code generation processes are launched
- specify a script that handles the distribution of these RhapsodyCL processes, for example, by having each such process run on a separate computer
- disable parallel code generation

You can control the use of parallel code generation by modifying the values of the following properties.

- CG::General::ParallelCodeGeneration
- CG::General::UserDefinedParallelProcesses
- CG::General::ParallelCodeGenerationCommand

For details on the use of these properties, see the Rational Rhapsody IC.

### Breaking code generation into chunks

When working with extremely large projects, you may encounter memory-related problems when trying to generate code for the entire model. To overcome such problems, you can use the [lang]_CG::Configuration::ClassesPerCGCall property to break up the code generation process into a number of distinct chunks.

Use the property to specify the maximum number of classes that should be included in a single code generation "chunk". Above this number, the code generation action will be broken into a number of smaller code generation actions.

For example, if you specify 500 for the value of the property, then if your model has 501-

1000 classes, Rational Rhapsody will try to break the code generation action into two smaller code generation actions.

Note: While the property name includes the term "classes", this number also takes into account similar model elements, such as actors and files.

## *Customizing code generation with the Features window and properties*

To customize the code that is produced by the Rational Rhapsody code generator, you can modify settings that are included in the Features window for the element, or you can modify the values of various code-generation properties.

In general, the options presented in the Features window represent modeling concepts from standards such as UML and SysML, while properties are used for options that are directly tied to the code generated, especially options that can vary between programming languages.

## Customizing file headers and footers

You can customize the headers and footers of the source files generated from your model. When you modify the values of the properties that control the content of the file headers and footers, you can use a number of keywords to include information such as the project name and the element name.

For languages that use separate specification and implementation files, the content of the file header is controlled by the properties [lang]_CG::File::SpecificationHeader and [lang]_CG::File::ImplementationHeader. For programming languages that don't have such a distinction, the content of the file header is controlled by the property [lang]_CG::File::Header.

For languages that use separate specification and implementation files, the content of the file footer is controlled by the properties [lang]_CG::File::SpecificationFooter and [lang]_CG::File::ImplementationFooter. For programming languages that don't have such a distinction, the content of the file footer is controlled by the property [lang]_CG::File::Footer.

The values of the header and footer properties can contain any of the following keywords:

- $ProjectName - the name of the project
- $ComponentName - the name of the component
- $ConfigurationName - the name of the configuration
- $ModelElementName - the name of the element mapped to the file. If there is more than one element mapped to the file, the text used is the name of the first element
- $FullModelElementName - the name of the element mapped to the file, including the full path. If there is more than one element mapped to the file, the text used is the name of the first element
- $CodeGeneratedDate - the date on which the file was generated
- $CodeGeneratedTime - the time at which the file was generated
- $RhapsodyVersion - the version of Rational Rhapsody that generated the file
- $Login - the user who generated the file
- $CodeGeneratedFileName - the name of the generated file
- $FullCodeGeneratedFileName - the name of the generated file, including the relative path, starting with the name of the directory that represents the component

# Including comments in the generated code

## *Element descriptions*

The Features window for model elements contains a Description tab that you can use to provide a description of the model element. These element descriptions are generated as comments that precede the code that is generated for the model element itself.

You can customize the content of these comments by modifying the value of the DescriptionTemplate property for the element.

In addition to providing plain text for the DescriptionTemplate property, you can use various keywords in order to include information that you have defined in the model, for example $FullName or $Arguments.

For detailed information on using the DescriptionTemplate property and the keywords that can be included, see the topic “Customize element descriptions in generated code” in the IC and the description of DescriptionTemplate in the property help pane of the Features window.

## *Including requirements as comments in the generated code*

To facilitate the tracing of requirements to code and vice versa, you can specify that the code generated should include comments that represent the requirements that are met by each code element.

Note: This feature can be used for C, C++, or Java code.

To include requirements as comments in the generated code:

1. Associate the requirement with the relevant model element
2. Open the Features window for the relevant configuration, and on the Settings tab select the Include Requirements as Comments in Code option. (This controls the value of the CG::Configuration::IncludeRequirementsAsComments property.)
3. Generate code.

The requirement comments can be generated in the specification file, the implementation file, or both.

For more information, see the topic titled “Including requirements as comments in generated code” in the IC.

## Code formatting

You can control the following aspects of code formatting in the code that Rational Rhapsody generates:

- curly brace style
- indentation
- operation arguments on separate lines

### *Curly brace style*

To accommodate various coding styles, Rational Rhapsody allows you to specify where curly braces should be placed for namespaces, classes, operations, enums, structs, and unions.

Curly brace style is controlled by the following properties:

<lang>_CG::Package::OpeningBraceStyle (for namespaces that have been defined)

<lang>_CG::Class::OpeningBraceStyle

<lang>_CG::Operation::OpeningBraceStyle

<lang>_CG::Type::OpeningBraceStyle

For each of these properties, the possible values are:

SameLine

NewLine

Each of these properties can be set from the project level downward. So if you want all the classes in your project to use a certain curly brace style, you would set the value of the <lang>_CG::Class::OpeningBraceStyle property at the project level.

### *Controlling indentation*

You can specify the number of spaces that should be used for indentation when code is generated.

This option is controlled by the Indentation property.

This property takes integer values representing the number of spaces you want to use for each indent.

Note: The Indentation property does not affect the indentation used in operation bodies since they consist of user-provided code.

## *Generating operation arguments on separate lines*

When generating code for operations, you have the option of having each operation argument generated on a separate line.

This option is controlled by the MultiLineArgumentList property.

If you set the value of the property to False, the generated code looks like this:

```
void displayBatteryInfo(double percentLeft, int minutesLeft, bool
isCharging);
```

If you set the value of the property to True, the generated code looks like this.

```
void displayBatteryInfo(double percentLeft,
             int minutesLeft,
             bool isCharging);
```

# Controlling the order of elements in the generated code

## *Changing the order of attributes in generated code*

By default, attributes are displayed in alphabetical order.

In some cases, you might want to define a specific order for the attributes when the code is generated. To change the order of appearance in the generated code:

1. Right-click Attributes on the browser and select Edit Order of Attributes.
2. In the window that opens, highlight the attribute that you want to move and use the Up and Down buttons to modify the order that is to be used in code generation. If the Up and Down buttons are disabled, clear the Use Default Order check box to enable them.
3. Click OK.

To restore the default order, select the Use Default Order check box and click OK.

## *Changing the order of operations in generated code*

By default, operations appear in the following order in generated code:

- Constructors and destructors
- User-defined operations
- Triggered operations

Within each of these categories, the operations are displayed in the following order: public, protected, private. Within these access subcategories, operations are listed alphabetically.

In some cases, you might want to define a specific order for the operations when the code is generated. To change the order of appearance in the generated code:

4. Right-click Operations (or Functions) on the browser and select Edit Operations Order (or Edit Functions Order).
5. In the window that opens, highlight the signature that you want to move and use the Up and Down buttons to modify the order that is to be used in code generation. If the Up and Down buttons are disabled, clear the Use Default Order check box to enable them.
6. Click OK.

To restore the default order, select the Use Default Order check box and click OK.

## Code generated for dependencies

If you create a “usage” dependency between a class and a class that it depends on, a corresponding #include directive will be generated for C or C++ code. In Java code, an import statement will be generated if the class is in a different package.

In C and C++, where the order of dependencies can be significant, you can modify the order of the dependencies in the generated code by right-clicking the Dependencies category in the browser and selecting Edit Dependencies Order.

Details on using the *Edit usage dependencies order* window can be found in the IC in the topic titled “Changing the order of usage dependencies in generated code”.

### *Controlling how usage dependencies are represented in the code (C, C++)*

The property CG::Dependency::UsageType can be used to control the way usage dependencies are represented in the generated code.

For example, you can generate:

- a forward declaration in the specification file
- an #include statement in the specification file
- a forward declaration in the specification file and an #include statement in the implementation file

Note that the code generated also depends on the value of the property [lang]_CG::Dependency::GenerateForwardDeclarations. If GenerateForwardDeclarations is set to True, code is generated as described above. However, if GenerateForwardDeclarations is set to False, then the specification file will not contain a forward declaration regardless of the value of the property UsageType.

For more details on using these properties, see the property documentation in the property help pane of the Features window.

### *Code generated for friend dependencies*

If you add a “friend” dependency from one class to another, the dependent class will contain a forward declaration for the friend class in addition to the declaration with the “friend” specifier within the class itself.

## Generating standard operations

Rational Rhapsody provides a mechanism for including standard operations in generated code.

The mechanism for generating standard operations involves:

- specifying the code for each standard operation in the site.prp file
- specifying which of the standard operations should be generated for a specific class by listing the operations in the value of the property CG::Class::StandardOperations

For more information, see the topic titled “Generating standard operations” in the IC.

## Statechart code

When you generate code, Rational Rhapsody generates code to represent statecharts that you have created.

In general, this is not code that you are going to modify directly, however, there are a number of points worth knowing about statechart code:

- All classes with statecharts inherit from OMReactive
- Statechart methods are preceded by an annotation such as:
  //## statechart_method
- The action code that you define for transitions is enclosed in annotations such as:

  ```
  //#[ transition 1
  print;
  //#]
  ```
- The action on entry and action on exit code that you define for an action is enclosed in annotations such as:

  ```
  //#[ state state_1.(Entry)
  entry_action_for_state_1();
  //#]
  ```

  and

  ```
  //#[ state state_1.(Exit)
  exit_action_for_state_1();
  //#]
  ```

# Language-specific code generation features

## *C and C++*

### *Generating classes as structs*

While class elements are ordinarily generated as C++ classes, you can specify that one or more classes should be generated as structs by changing the value of the property CPP_CG::Class::GenClassAsStruct.

### *Friend classes*

You can define friend classes and friend functions in your model, and the appropriate C++ code will be generated.

To define a friend class:

1. Add a dependency from the class whose data will be made available to the class that needs to access this data.
2. Apply the Friend stereotype to the dependency.

If the dependency is drawn from class A to class B, the code generated for class A will contain the following declaration:

```
friend class B;
```

To define a Friend function:

1. Add a dependency from the class whose data will be made available to the function that needs to access this data.
2. Apply the friend stereotype to the dependency.

If the dependency is drawn from class A to function `getInfo()`, the code generated for class A will contain the following declaration:

```
friend void getInfo();
```

*For more information on using friend classes and friend functions, see the topic "Code generation for friend classes and functions" in the Rhapsody IC.*

### *Wrapping code with #ifdef-#endif*

If you need to wrap elements such as classes or operations with an `#ifdef #endif` pair, add a compiler-specific keyword, or add a `#pragma` directive, you can set the `SpecificationProlog`, `SpecificationEpilog`, `ImplementationProlog`, and `ImplementationEpilog` properties for the element.

For more information, see the topic "Wrapping code with #ifdef-#endif" in the Rhapsody IC.

## *Generating operation descriptions in implementation files*

The text that was entered on the Description tab of the Features window for an operation is generated as a comment before the declaration of an operation.

Using the properties GenerateDescriptionInImplementation and DescriptionInImplementation, you can have a descriptive comment generated in the implementation file as well.

For more information on the use of these two properties to control operation descriptions in the generated code, see the topic "Generating operation descriptions in implementation files" in the Rhapsody IC.

## *Fine-tuning code generated for arguments*

You can use the Code pattern field in the Features window for arguments to fine-tune the code that is generated for the arguments of a specific operation or event.

If you want to modify the code generated when a given class, type, or event is used as an operation or event argument anywhere in your model, you can modify the value of the relevant In, Out, InOut, or TriggerArgument property at the class, type, or event level.

For more information on using the Code pattern field and the In, Out, InOut, and TriggerArgument properties, see the topic "Fine-tuning code generated for arguments" in the Rhapsody IC.

## *Fine-tuning code generated for return types*

The mechanism used for fine-tuning the code generated for return types is similar to that used for the code generated for arguments.

You can use the Code pattern field on the General tab of the Features window for operation to fine-tune the code that is generated for the return type of a specific operation.

If you want to modify the code that is generated when a given class, type, or event is used as the return type of an operation anywhere in your model, you can set the value of the ReturnType property at the class, type, or event level.

For more information on using the Code pattern field and the ReturnType property, see the topic "Fine-tuning code generated for return types" in the Rhapsody IC.

## Java

### Including Javadoc comments in the generated code

Rational Rhapsody provides a mechanism for including Javadoc comments when code is generated for a Java model.

This mechanism is turned on, by default, for Java projects.

In the generated code, you see Javadoc comments based on the descriptions you have provided for model elements. Comments for operations also include any descriptions you have provided for operation arguments.

In addition to generating these basic Javadoc comments, you can have Rational Rhapsody include the following standard Javadoc tags: author, deprecated, return, see, since, and version.

Once your code includes Javadoc comments, you can generate a Javadoc report using the standard Javadoc process.

For more details on including Javadoc comments in the generated code and customizing these comments, see the section titled “Javadoc comments” in the Rhapsody IC.

### Static imports

You can model static imports and Rational Rhapsody generates the appropriate code. In addition, the reverse engineering feature can handle static imports in Java code, and the roundtripping feature can handle changes to static import statements.

You can model both static import of individual class members (`import static java.lang.Math.PI`) and static import of all static members of a class (`import static java.lang.Math.*`).

To add static imports to your model:

1. Create a dependency in the browser or by drawing a dependency in an object model diagram. The dependency can be from a class to a class or from a class to an individual static attribute or operation.
2. Open the Features window for the dependency you created, and apply the StaticImport stereotype to it.

When you next generate code, the code for the dependent class contains the appropriate static import statement.

For more information on using static imports, see the section titled “Static import constructs” in the IC.

## *Defining static blocks*

Rational Rhapsody allows you to add static blocks to Java classes in your model, and generates appropriate code for such blocks.

To add a static block to a class:

1. Right-click the class in the browser and select Add New > StaticBlock. (Alternatively, right-click the class in an object model diagram and select New StaticBlock.)
2. Open the Features window for the newly created static block, and on the Implementation tab enter the code for the body of the block.

You can switch a static block to an operation and vice versa.

To change a static block to an operation, right-click the static block in the browser and select Change To > Primitive Operation.

To change a primitive operation to a static block, right-click the operation in the browser and select Change To > Static Block.

For more details on using static blocks, see the topic titled "Static blocks" in the IC.

## *Generating JAR files*

For Java projects, you have the option of specifying that a JAR file should be generated when you build your project.

To specify that a JAR file be created as part of the build process:

1. Open the Features window for the relevant configuration.
2. On the Settings tab, select the **Generate JAR File** option.

The JAR file generation mechanism is controlled by the following properties:

- `JAVA_CG::Configuration::JarFileGenerate` - a Boolean property that determines whether a JAR file is generated as part of the build process.
- `JAVA_CG::Configuration::JarFileGeneratorCommand` - specifies the jar command that is carried out if the `JarFileGenerate` property has been set to `True`.

## *Java 5 annotations*

Rational Rhapsody allows you to use Java annotations in your model and to have these annotations included in your generated code.

The detailed steps for using Java annotations can be found in the IC in the topic titled “Java 5 annotations”.

The basic steps for using this feature are as follows:

1. Create an AnnotationType by right-clicking a package or class and selecting the relevant option from the pop-up menu.
2. Open the Features window for the Annotation Type you created, and use the Elements tab to add the information you want to include for the annotation.
3. Create a JavaAnnotation by right-clicking a package and selecting the relevant option from the pop-up menu. When the Add Java Annotation window is displayed, select the relevant AnnotationType.
4. Open the Features window for the JavaAnnotation you created and assign values to each of the annotation elements.
5. Associate the JavaAnnotation you created with one or more model elements by drawing a dependency from the model element to the JavaAnnotation, and applying the  AnnotationUsage stereotype to the dependency.

After carrying out these steps, when you generate code, the annotation is generated for the relevant model element.

## Multi-language projects

If you have the necessary license for multi-language projects, you can create a single model that contains units that are associated with different programming languages. A model can include units associated with C, C++, or Java. Code can then be generated in the appropriate language for each unit.

When you use the Create Unit option to create a unit for a specific model element, the Unit Information window provides a list that allows you to select a specific language for the unit. The default language for a new unit is the language of its owner unit.

For code generation to work properly for multi-language projects, you have to adhere to the following rules:

- To generate code for units in a certain language, the appropriate language must be specified at the component level. You specify the language of the component by selecting a language on the Scope tab of the Features window for the component.
- Elements included in the scope of a component must be of the same language as the component.

For more information on using multi-language projects, see the topic titled “Planning multiple development language projects” in the IC.

## Using documentation generation tools such as Doxygen

For Java models, Javadoc comments are generated by default.

For other languages, if you would like to use documentation generation tools such as Doxygen, you can generate comments with the required format by modifying the value of the property <lang>_CG::Configuration::DescriptionBeginLine. There is also a corresponding property - <lang>_CG::Configuration::DescriptionEndLine - that can be used to specify the symbols for the end of comment lines.

## Code generation with the command line

You can carry out code-related actions from the command-line, using RhapsodyCL.exe.

RhapsodyCL allows you to use code-related functions, such as generate and make, in contexts where you do not require the GUI elements of Rational Rhapsody, for example, as part of a nightly build procedure.

The actions you can carry out include:

- buildentire
  Builds the entire project
- buildwithdep
  Builds a component with all its dependencies
- generate
  Generates code for a specific component and configuration
- genwithdep
  Generates a component with all its dependencies
- gmr
  Performs generate/make/run
- make
  Builds the application, using the current configuration
- rebuildentire
  Rebuilds the entire project
- rebuildwithdep
  Rebuilds a component with all its dependencies
- regenerate
  Regenerates the code for the specified component and configuration
- regenentire
  Regenerates the code for the entire project
- regenwithdep
  Regenerates the code for a component with all its dependencies
- roundtrip
  Roundtrips code changes back into the model
- setcomponent
  Sets the active component
- setconfiguration
  Sets the active configuration

These commands can be carried out individually from the command-line, or they can be combined in batch files.

For details of the syntax to use for these commands, see the topic titled “Command-line commands” in the IC.