Telelogic

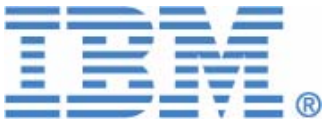
Rhapsody

ReporterPLUS Guide

IBM

Rhapsody®

ReporterPLUS Guide



Before using the information in this manual, be sure to read the “Notices” section of the Help or the PDF available from **Help > List of Books**.

This edition applies to Telelogic Rhapsody 7.4 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

ReporterPLUS Basics

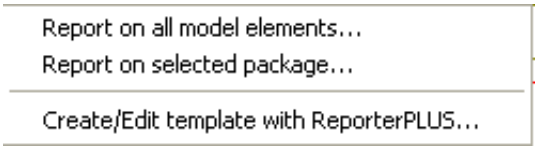
ReporterPLUS is a documentation tool that allows you to create Microsoft Word, Microsoft PowerPoint, HTML, RTF, and text documents from any Rhapsody model. ReporterPLUS' pre-defined templates extract text and diagrams from a model, arrange the text and diagrams in a document, and format the document. For a list of these templates, see **Standard ReporterPLUS Templates**. To customize any of these templates, you can use drag-and-drop techniques.

Note

If you are using ReporterPLUS on Linux, you must install the current version of OpenOffice and Firefox.

Methods for Starting ReporterPLUS

To start ReporterPLUS from inside of Rhapsody, select **Tools > ReporterPLUS**. This menu displays options for printing the model currently displayed in Rhapsody.



The **Report on selected package** option is not available from this menu unless a package in the model is highlighted in the Rhapsody browser. If one of the first two items is selected, a report can be generated using a predefined template without displaying the main ReporterPLUS GUI. If the last option is selected, ReporterPLUS starts with no model elements imported.

To start ReporterPLUS from outside of Rhapsody, select **Start > All Programs > Telelogic > Telelogic Rhapsody *version #* >Rhapsody ReporterPLUS *version #* > Rhapsody ReporterPLUS *version #***.

Note

The advantage of running ReporterPLUS from within Rhapsody is convenience and the ability to generate a report on a selected element. If you run the ReporterPLUS interface from outside Rhapsody, it can generate a report only for the full model.

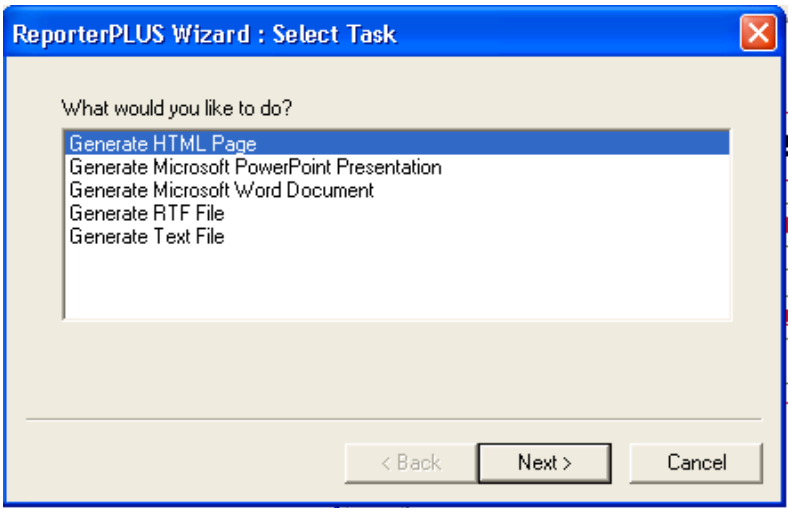
By default, ReporterPLUS runs as a DLL when it is invoked from inside Rhapsody. The disadvantage of running ReporterPLUS as a DLL is that ReporterPLUS always saves the model before generating the report and an unnecessary save on a large model is usually unnecessary and time consuming.

ReporterPLUS does not save the model when generating a report when it runs as an *executable*. Use one of these methods to run it as an executable:

- Set the `InvokeReporterDLL` variable in the `rhapsody.ini` file to `FALSE`
- Launch ReporterPLUS outside of Rhapsody

This approach is preferred for reports generated from large models, as described in the **Creating HTML Reports for Large Models** section.

After starting ReporterPLUS, either from the Tools menu or outside Rhapsody, select one of these report generation options.



The ReporterPLUS Interface

After selecting the report type, you then select a template for the report. If you cancel that selection, the interface appears so that you can examine or define one or more templates.

Across the top of the interface is a menu bar and icons providing the following options:

- **File** allows you to create templates, open and save templates, open and close models, and set template properties. It also shows you the recent template files used.
- **Edit** provides the standard Cut, Copy, Paste, Undo and Redo options with special new, move, and delete template node options and the important *Generate Document* option.

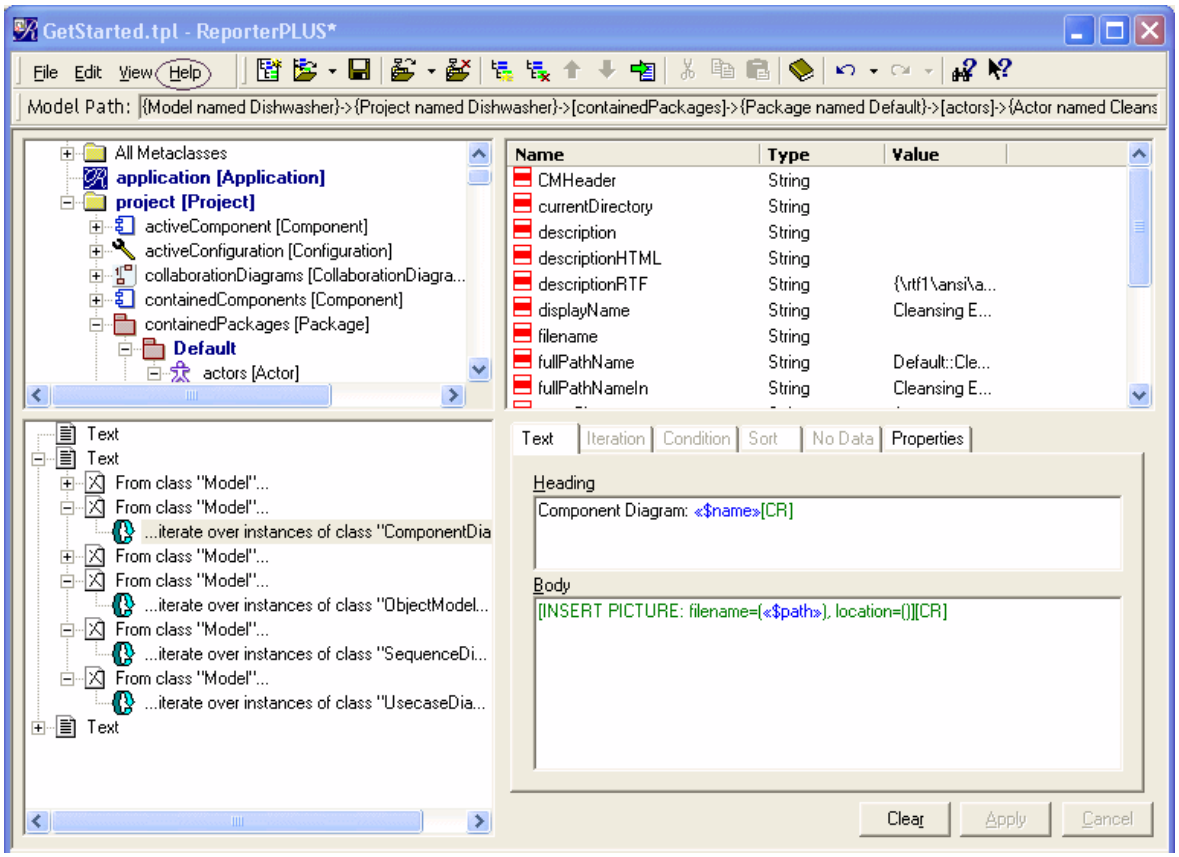
- **View** displays toolbars, interface display options, default document properties for the HTML, Word, PowerPoint, and RTF output. This menu also accesses the *Model View Guide* to view information on a selected element or attribute.
- **Help** provides a specific ReporterPLUS Help system.

The ReporterPLUS interface work area is divided into four panes (referred to as *views*):

- The model view
- The attribute view
- The template view
- The template node view

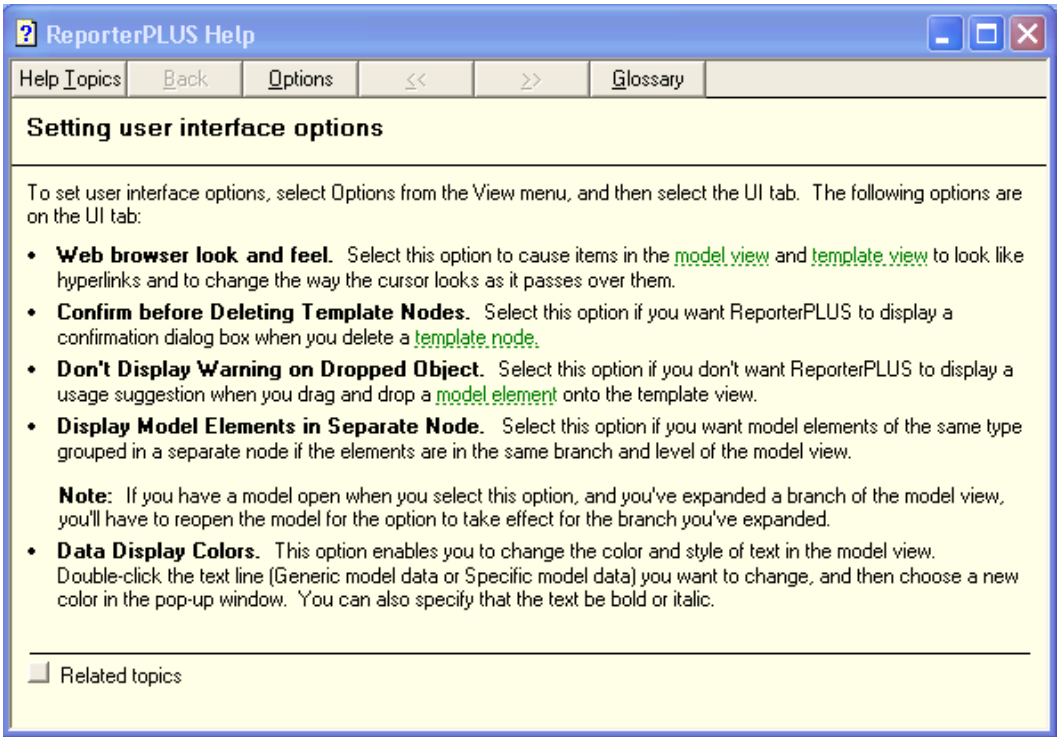
If you have just started ReporterPLUS, the model view displays data and the other three views are empty.

You can customize the colors in the model view. For more instructions, search the ReporterPLUS online Help for the topic “Setting user interface options.” The following figure shows the ReporterPLUS interface with the ReporterPLUS online Help menu option circled.



ReporterPLUS Online Help

The ReporterPLUS interface provides general descriptions of the interface features. The following illustration shows a topic from the ReporterPLUS online Help.



In addition to the Help system, Rhapsody provides this manual with special instructions to use ReporterPLUS with Rhapsody models. This additional documentation is designed to supplement the online Help (described above). There are two methods to access the *Rhapsody ReporterPLUS Guide*:

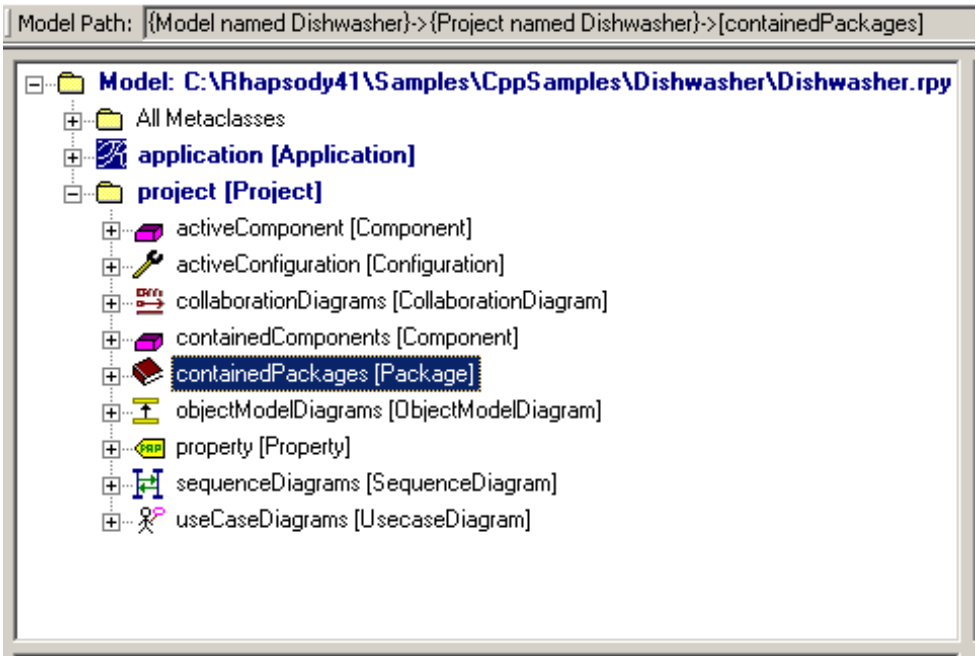
- In the Rhapsody interface, select **Help > List of Books** and click the manual's title in that list.
- For the Windows Start menu outside the Rhapsody interface, select **Start > All Programs > Telelogic > Telelogic Rhapsody *version #* > Rhapsody ReporterPLUS *version #* > Rhapsody ReporterPLUS Guide**.

Model View

The *model view* is the upper, left pane of the ReporterPLUS window. When you first start ReporterPLUS, the model view displays generic elements. Generic elements are types of model elements that can be found in any model. For more information on the difference between generic and model elements, see **Generic and Model Elements**.

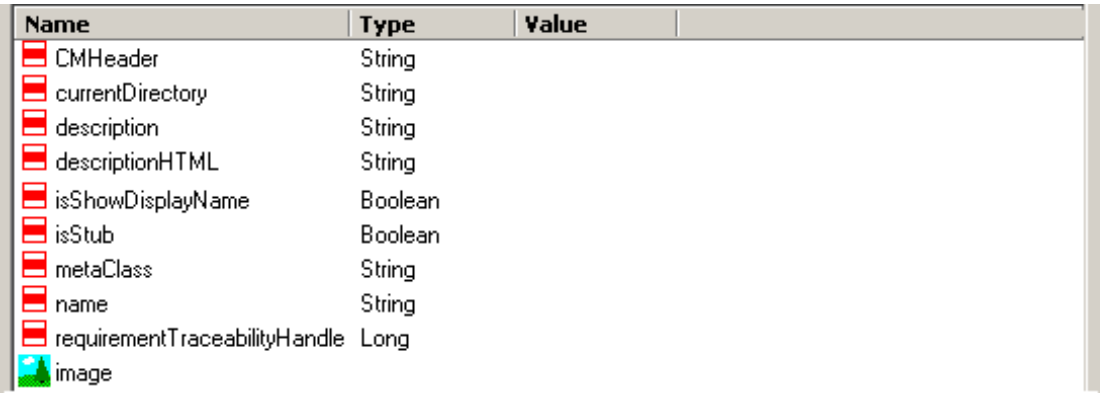
When you open a model, you can see elements from the actual model in addition to generic elements. By default, the generic elements are in black text and the model elements are in blue. These model elements are the same model elements you see when you open the model in Rhapsody. The various icons you see identify different types of elements.

When you select an element in the model view, the path to that element is displayed in the **Model Path** field, as shown in the following figure.



Attribute View

The *attribute view* is the upper, right pane in the ReporterPLUS window. It displays the attributes for the element selected in the model view. Attributes represent the pieces of *data* that you can extract for each element in the model. For example, for `Property`, you can extract the following attributes: `isOverridden`, `metaClass`, `name`, `propertyName`, `subject`, `type` and `value`. The following figure shows the `Property` attributes.



To extract text and diagrams from a model, you can simply *drag and drop* an attribute into the **Text Tab** area and then enter any addition information to add it to your ReporterPLUS template.

When you select a generic element in the model view, you see generic attributes in the attribute view. The value column on the far right of the attribute view is blank because generic elements represent *types* of model elements, not *actual* model elements. If you have a model open and select a specific model element, you see the actual values of the attributes. See **Generic and Model Elements** for more information on generic and model elements.

Template View

The *template view* is the lower, left pane of the ReporterPLUS window. The template view is blank when you first start ReporterPLUS. Once you open a ReporterPLUS template, it displays template nodes (or report sections), which present a graphical representation of the template's structure. The order and structure of a generated document is determined by the order and structure of the nodes in the template used to generate the document. Content (text and diagrams) is added to the template by **Adding Boilerplate Text** and attributes to template nodes.

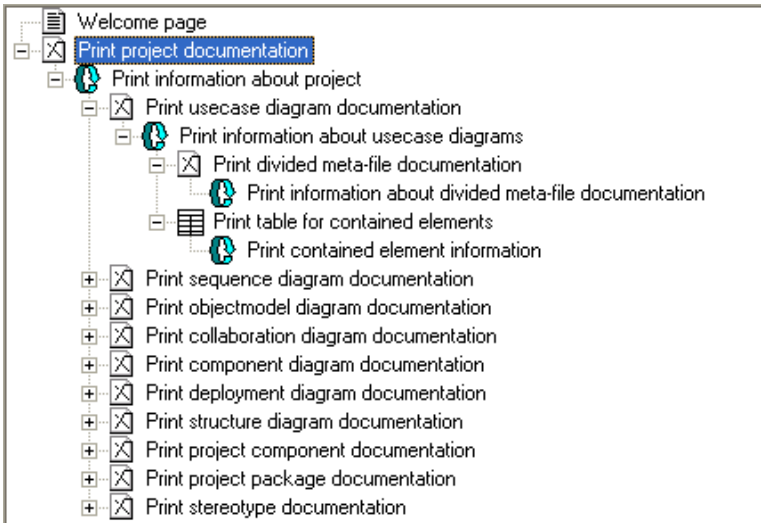
The nodes in the template view form a hierarchical structure. A node under an iteration is controlled by its parent iteration. This is true of both subnodes and nested iterations. In addition, the hierarchy of template nodes determines the heading level of the nodes in the generated document (see **Formatting You Can Specify in ReporterPLUS** for more information).

When you add a node to the template view, ReporterPLUS creates a label for the node. The label identifies what the node does and where in the model it came from. You can change this label on the Properties tab (see the **Properties Tab** and **Changing the Node Label** sections for more information).

Node (Section) Types

There are four kinds of nodes in the template view (shown below):

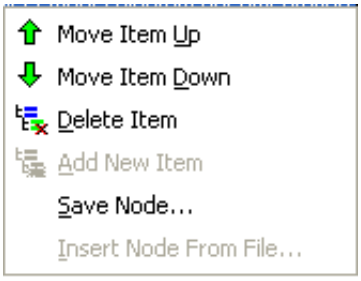
- **Iteration nodes**—Specify what class the iteration pertains to. ReporterPLUS loops (or iterates) through this class looking for elements to extract. Iteration nodes also specify any conditions that limit the iteration, how elements are sorted, and what happens when the iteration does not yield elements from the model. In addition, iteration nodes can contain boilerplate text and attributes.
- **Iteration subnodes**—Contains the information included in your document for each element extracted by the iteration. Subnodes contain attributes (such as name or image) that represent the text and diagrams you want to include in your document and usually contain boilerplate text as well. Whereas the iteration node tells ReporterPLUS which class to look at (iterate over) and what element to extract from that class, the subnode tells ReporterPLUS what attributes (what information about that element) to extract.
- **Table nodes**—Are iteration nodes that produce tables rather than paragraphs of text. The information on the table node becomes column headings and the information on the iteration subnodes beneath the table node becomes the body of the table. To add a table node, see the **Text Tab** section.
- **Text nodes**—Hold attributes and boilerplate text. They can also be used to hold headers and footers for your generated document. Text nodes can stand on their own, serve as subnodes under iteration subnodes, or serve as parent nodes to iteration nodes or table nodes, but they cannot hold iterations.



Template View Options

To perform operations on the items in the template, follow these steps:

1. Highlight an item in the template view.
2. Right-click to display this menu.



3. The options available on this menu depend on the type of template element you selected.

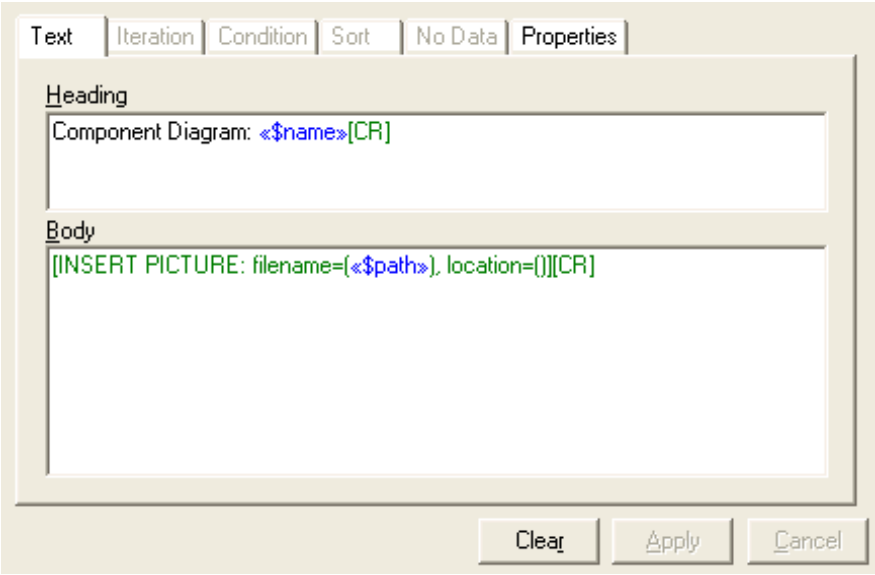
Use the menu options to perform this template operations:

- **Move Item Up** repositions the highlighted template element one position higher in the template structure. Select this option again to move the element again.
- **Move Item Down** repositions the highlighted template element one position lower in the template structure. Select this option again to move the element again.
- **Delete Item** removes a template element from the template.
- **Add New Item** allows you to insert a new element into the template.

- **Save Node** allows you to save a portion of the template as a subtemplate to be inserted in this or any other template. This option creates an “sbt” file and allows you to store it in a selected directory.
- **Insert Node from File** allows you to put a previously save subtemplate (“sbt” file) into the selected location in the currently displayed template.

Template Node View

The *template node view* is the lower, right pane of the ReporterPLUS window. It consists of six tabs, which allow you to view and modify the contents of template nodes (or sections). All six tabs are active when you select an iteration node or table node in the template view. When you select an iteration subnode, only the Text and Properties tabs are active. When you select a text node, the Text, Iteration, and Properties tabs are active. The following figure shows the template node view with the Text tab selected.



Text Tab

The Text tab shows the contents of the template node selected in the template view. The contents consist of boilerplate text, attributes (for example, «`$name`», «`$path`»), and formatting commands (for example, `[CR]`, `[INSERT PICTURE]`).

The Text tab has two sections where you can define the **Heading** and the **Body** of the text. Attributes and boilerplate text in the heading of the Text tab become a heading in the generated document. Attributes and boilerplate text in the body of the Text tab become part of the body of the generated document.

When you create a new template node, ReporterPLUS automatically adds some boilerplate text and attributes to the Text tab. You can leave this information unchanged if you want it to appear in generated documents, or you can delete it or modify it using right-click menu options.

Note

Remember that you can use the **Edit > Undo** option to remove a text change that is not what you intended.

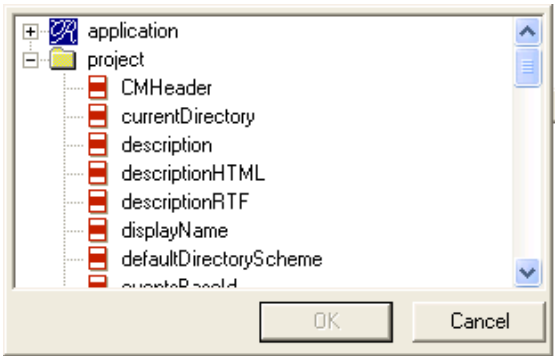
To modify the text, right-click in either of Text tab sections to display this menu.

Add Attribute

Selecting the **Add Attribute** command, displays a dialog box containing standard attributes and others specific to the project. For more information about attributes, see the **Attribute View** section. Use this option when you want to work with a single attribute from the attributes listed above.

To add an attribute to the text generation instructions, follow these steps:

1. Position the cursor where you want the attribute to be inserted.
2. Right-click to display the text modification menu (shown previously).
3. Select the **Add Attribute** option, and this dialog displays.



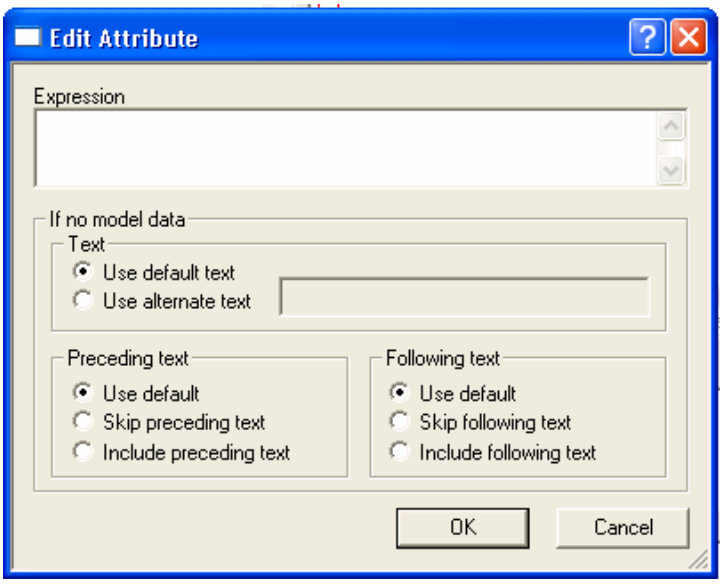
4. Highlight the desired attribute in the list.
5. Click **OK**.

Add Expression

Use the **Add Expression** option when you want to work with two or more attributes from the attributes listed above in an operation such as a comparison. This type of operation requires the use of the Q language to define.

To add an expression, follow these steps:

1. Position the cursor where you want the expression inserted in the **Heading** or **Body** areas.
2. Right-click to display the text modification menu (shown previously).
3. Select the **Add Expression** option, and this dialog displays.



4. Type the expression using the **Q Language** syntax.
5. If the Expression needs more information to complete the operation described, select any additional options using the radio buttons in the three areas below. For example, every time a phrase (described in the **Expression** area) appears in the model you want to substitute a different phrase (in the report), select the **Use alternate text** radio button to replace the <<No Model Data>> that appears when ReporterPLUS cannot find an element/attribute (such as a description).
6. Click **OK**. ReporterPLUS checks the syntax of the Q language expression and inserts it if is correct. If it is not, it displays an error message.

Commands

The Text tab menu supplies commands that define text. These commands simplify common types of modifications, such as adding page breaks and numbers. Select the **Commands** option from the Text tab's right-click menu to display this list of commands:

- **Add Carriage Return** breaks text at the specified location.
- **Add Page Break** inserts a manual page break at the specified location.
- **Add Date** inserts a field that automatically inserts the date that the report is generated into the results.
- **Add Time** inserts a field that automatically inserts the current time of day that the report is generated.
- **Add Filename** allows you to reference a file that is external to the model from a hyperlink that you add using the **Insert Link** command.
- **Add Page Number** inserts a field that automatically numbers the pages in a report each time it is generated.
- **Begin <Header>** marks the beginning of the report header text. See **Using Multiple Headers and Footers** for more information. This command changes depending on the highlighted item.
- **End <Header>** marks the end of the report header text. This command changes depending on the highlighted item.
- **Begin <Footer>** marks the beginning of the report footer text. This command changes depending on the highlighted item.
- **End <Footer>** marks the end of the report footer text.This command changes depending on the highlighted item.
- **Insert Picture** marks the location and provides the file name of a picture in an external file or model diagram is inserted when the report is generated.
- **Insert Bookmark** allows you to specify a particular location in the generated report to which a hyperlink can be joined.
- **Insert Link** defines a hyperlink to a position in the generated report identified with a bookmark or a file defined using the **Add Filename** command.
- **Start new file** allows you to start a new report file within the existing file, give it a separate name, and define it as well.

- **Convert** allows you to insert text that should be substituted for text as it appears in the model. For example, you want the report to refer to a component in the project by its commercial name in the report instead of the working name used in the project.
- **Locate** is only used for the HTML reports created, as described in the **Generating an HTML Exporter Report** section. For this template only, the command inserts a navigation button into the generated HTML display. This **Locate In Browser** button allows the user to highlight an item in the HTML report (right side), click the button, and jump to the display of the corresponding node in the HTML browser on the left side.

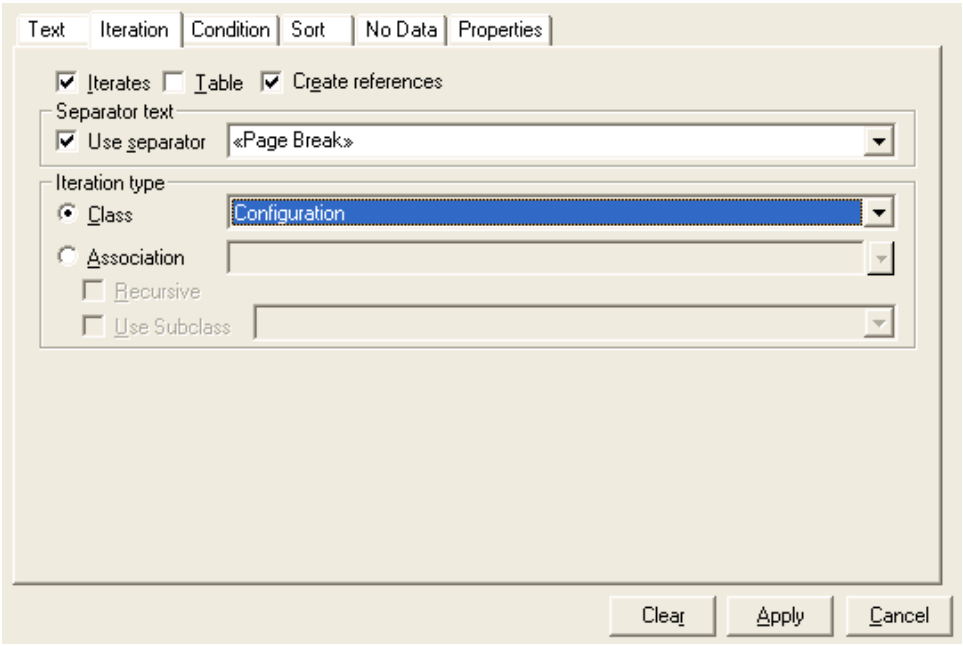
Formatting

The remaining options in the text modification menu support these common formatting processes:

- **Font** selections
- **Style** definition
- **Alignment** with margins
- **Table** addition and definition

Iteration Tab

The Iteration tab (below) displays details about an iteration or table node, such as iteration type Class (…) or Association (…) and whether recursion and subclasses are applied. It can also be used to switch the node type to text, iteration or table.



The Iteration tab displays details of an iteration node or a table node. To view these details, select the iteration or table node in the template view, and then select the Iteration tab. You can also modify the iteration on the Iteration tab, and (for some iterations) you can add recursion or attributes for subclasses. The following are the uses for each of the iteration tab options.

Iterates indicates whether the template node is a text node or an iteration or table node. If this option is selected, the node is either an iteration node or a table node. If you have selected a text node in the template view, you can select this option to convert the text node into an iteration node. (You must also specify which class or association you want to node to apply to; for information, see the Modifying iterations button below.) If you have selected an iteration node in the template view, this option is already selected. You can convert the iteration node to a text node by clearing this option. Note that clearing this option deletes all the iteration subnodes of the selected node, and invalidates nested iterations. (Invalid iterations are displayed in red text in the template view.)

Table converts an iteration node to a table node. If this option is selected, the node is a table node. To convert a table node to an iteration node, clear this option. For information on creating tables, see the How do I add tables? button below.

Use Separator indicates that you want a separator between subsections created by the iteration. Then select the type of separator (page break or new line) from the drop-down list, or enter text that you want to use as a separator.

Class selects a generic element in the adjacent drop-down list to modify the iteration node to extract all of the elements of the selected type in the entire model (rather than a branch). These are the same generic elements that you'll find under All Metaclasses in the model view. For example, if you select this option, and then select Class from the drop-down list, you're selecting all of the classes in the model. If this option is already selected, you can select a different element in the drop-down list.

Association selects a generic element in the adjacent drop-down list to modify the iteration node to extract all of the elements of the selected type in a branch of the model view (rather than the entire model). If this option is already selected, you can select a different element in the drop-down list.

Recursive instructs ReporterPLUS to carry out the current iteration node's instructions for all nested elements of the same type. This option is available for iteration nodes that use one of the following generic elements: nestedPackages, nestedActors, nestedUsecases, or Substates. (There are other reflexive associations, but these are the most useful.) To see what generic element an iteration node uses, select the iteration node, and then select the Iteration tab. The Class or Association field displays this generic element.

Use Subclass selects a subclass from the drop-down list. (Note that not every class on this list is applicable for all associations.) Once you have selected a subclass, you can add attributes for the subclass by right-clicking in the Text tab for the same node, selecting Add Attribute from the context menu, and then selecting the attribute you want to add.

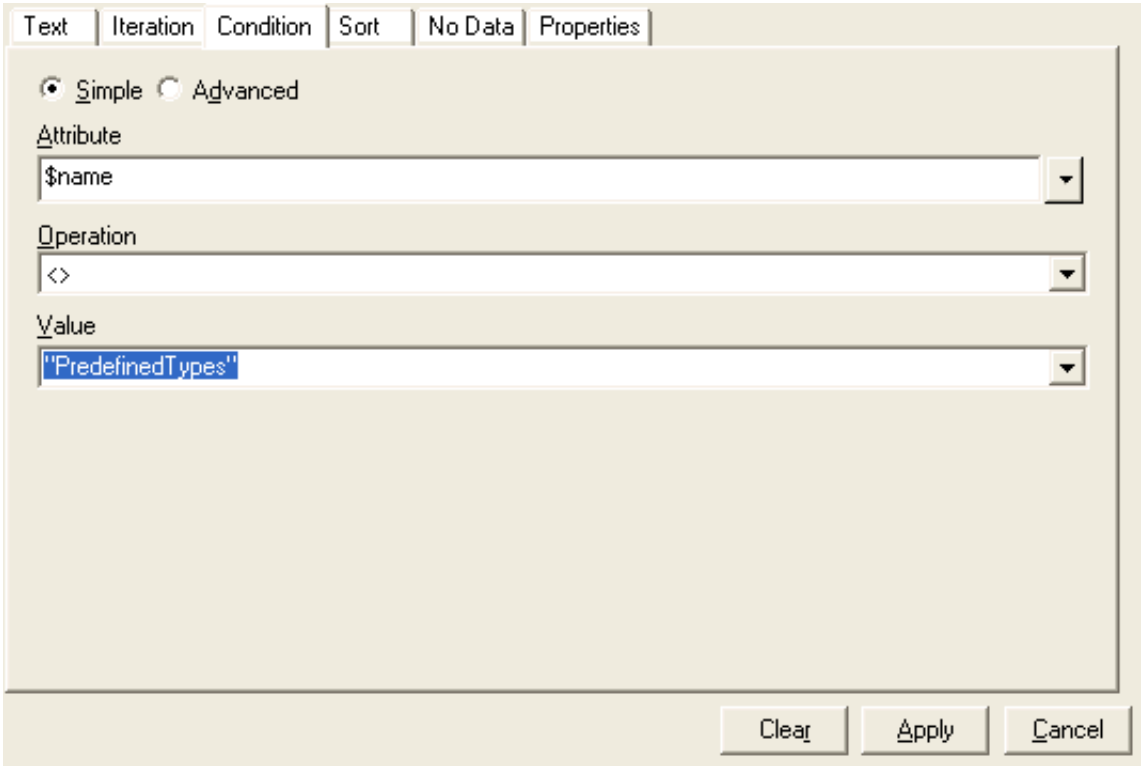
Condition Tab

The Condition tab, shown in the following figure, shows the conditions that apply to an iteration or table node. Conditions limit the model elements ReporterPLUS includes from an iteration. When you add a condition to an iteration, ReporterPLUS extracts only those model elements that meet the condition. If you do not specify a condition, ReporterPLUS extracts all the model elements in the iteration.

There are two types of conditions:

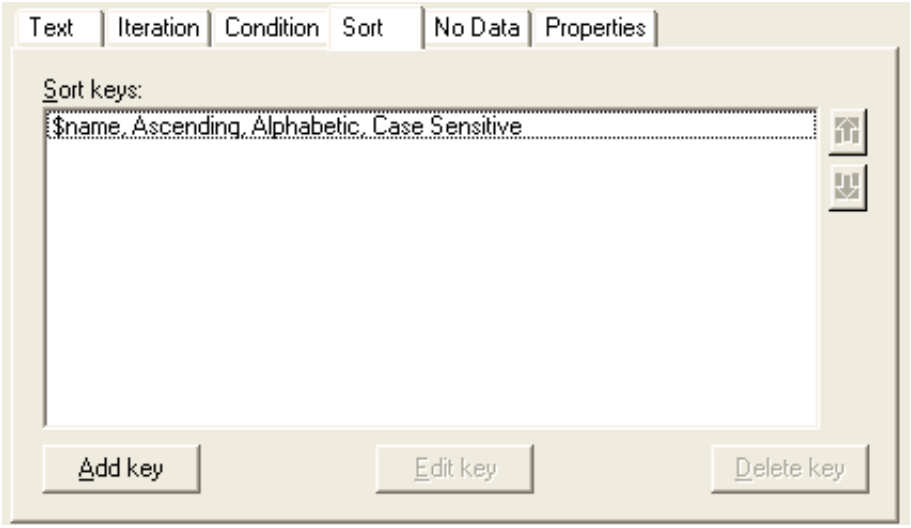
- **Simple condition**—Compares a single attribute with a single value.
- **Advanced condition**—provides an open field to enter Q language. to define the condition. See the **Q Language** section for detailed information. See **Adding a Condition** for instructions on adding conditions to iterations.

The example below illustrates a simple condition that excludes (<>) the names ($name) of the predefined types when the report is generated.



Sort Tab

The Sort tab, shown in the following figure, shows how model elements extracted by an iteration are sorted in the generated document.



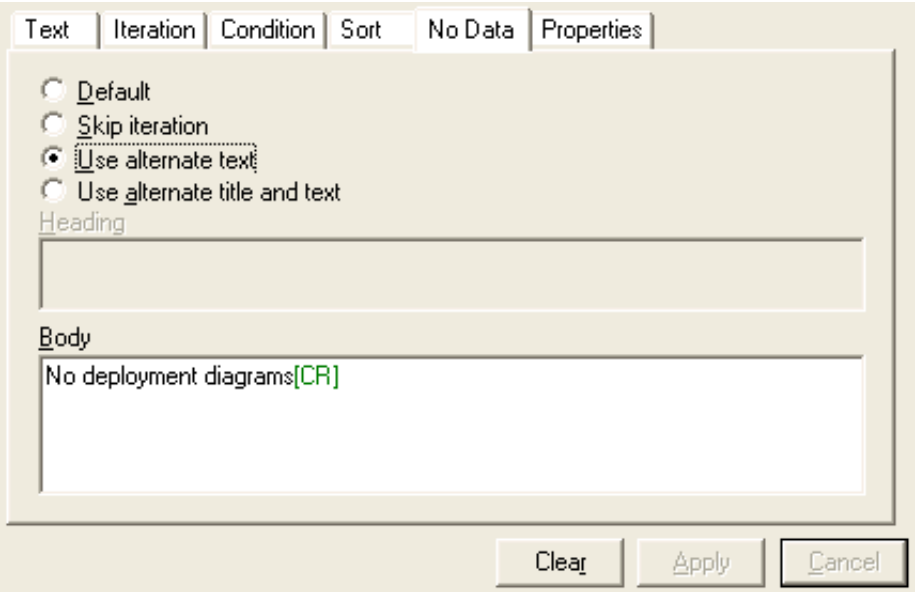
You can specify an unlimited number of keys for the sort. For each key, you can specify whether:

- The attributes are sorted in ascending or descending order
- The sort is alphanumeric or numeric
- The sort is case-sensitive

You can quickly re-order sort keys by selecting a key and clicking the up and down arrows on the right side of the Sort tab. See **Sorting Model Elements** for instructions on specifying a sort order for model elements.

No Data Tab

The information on the No Data tab tells ReporterPLUS what to do when there is no model data for an iteration. The default behavior is to skip that section of the template and not print anything in the generated document for that iteration. The No Data tab enables you to specify that ReporterPLUS print something instead.



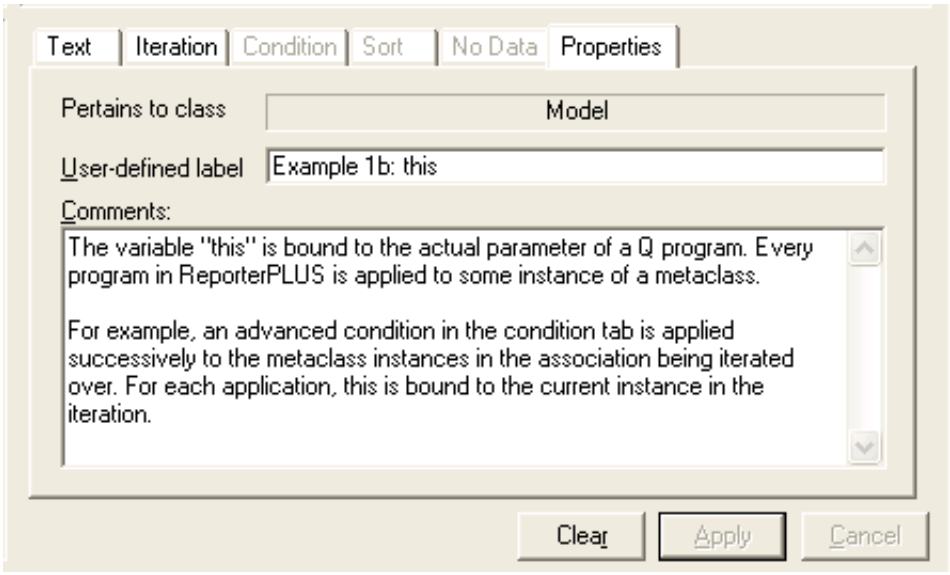
See **Coping with Missing Elements in an Iteration** for instructions on using the No Data tab.

Note

You can also specify how ReporterPLUS should handle a lack of model data for a particular attribute. See **Coping with a Lack of Model Data**.

Properties Tab

The Properties tab, shown in the following figure, shows what class the template node pertains to; this is the class that ReporterPLUS iterates over to extract data for the selected template node. The **User-defined label** field enables you to change the default label for this node. (The label is the text displayed next to the node icon in the template view.) The Properties tab also has a field for entering comments about the node. Neither the comments nor the label print in the generated document.



ReporterPLUS Template Basics

There are two basic ingredients required to create a document with ReporterPLUS:

- Rhapsody model
- ReporterPLUS template

The template tells ReporterPLUS what information to extract from the model, how to arrange and format the document, and what boilerplate text to include. Once you create a ReporterPLUS template, you can use it to generate documents from any model created in Rhapsody.

Creating Documents from Templates

ReporterPLUS creates documents using these techniques:

- Extracting text and diagrams from a model created in Rhapsody.
- Creating a *Microsoft Word*, *PowerPoint*, *HTML*, RTF, or text document.
- Adding text and diagrams from the model and images to the document. (Note that text files do not include diagrams.)
- Adding boilerplate text specified in the ReporterPLUS template to the document.
- Formatting the document according to the formatting commands in the ReporterPLUS template, as well as the specifications in a *Word* template (`.dot` file), a PowerPoint template (`.pot` file), or an HTML style sheet (`.css` file). Using a `.dot`, `.pot`, or `.css` file is optional. You can also use HTML tags to format HTML documents. See **Setting Standard Template Properties** for more information about this feature.

This section summarizes ReporterPLUS templates and provides general information about the steps you need to take to build them. Subsequent sections provide instructions on building your own template.

Note

If you do not want to create a ReporterPLUS template from scratch, you can use the templates included with the tool. You can use these templates as-is or modify them as needed. You can use them as starting points for creating your own templates and as nodes/iterations to import into a customized template.

Basic Steps in Building a New Template

Building a ReporterPLUS template involves four basic steps:

1. Create the template structure by adding nodes to a template.
2. Add attributes (model text and diagrams).
3. Add boilerplate text.
4. Add formatting.

The following sections describe each of these steps in more detail.

Adding Template Nodes (Sections)

You create the basic structure of a ReporterPLUS template by adding template nodes to the template view. The template's structure determines the structure of documents generated from it.

There are three ways to add report sections (also called "nodes") to the template view:

- Drag an element from the model view to the template view. This method creates an iteration node and an iteration subnode. You can add boilerplate text and attributes to either node. To create a table node, you first create an iteration node, then change it to a table node.
- Use the **New Template Node** command or toolbar button. This method creates a single text node (rather than an iteration and subnode), to which you can add boilerplate text and attributes.
- Right-click in the template view, then select **Add New Item**. This method also creates a single text node to which you can add text and attributes.

You can also save and then re-use template nodes or whole branches of nodes by using the **Save Node** and **Insert Node From File** options on the context menu in the template view. See the ReporterPLUS online help topic "Saving template nodes and sub-templates" for more information.

Adding Model Text and Diagrams

After you add nodes to the template, you need to add attributes to the nodes. Attributes represent the data—text and diagrams—that you want to extract from the model and include in your document.

ReporterPLUS adds some attributes automatically when you drag an element to the template view. For example, ReporterPLUS always adds the «`$name`» attribute. If you not want this attribute, you can delete it from the Text tab. To add attributes, drag them from the attribute view to the heading or body of the Text tab in the template node view.

Adding Boilerplate Text

Boilerplate text is text that you add to the template by typing in the heading or body of the Text tab—it does *not* come from the model. ReporterPLUS inserts some boilerplate text automatically when you add template nodes; you can keep, modify, or delete this text.

Specifying Formatting for the Document

The following factors affect the format of your generated document:

- The ReporterPLUS template and ReporterPLUS options
- The output type (Word, PowerPoint, HTML, RTF, or text)
- A Word or PowerPoint template or the HTML style sheet you may attach to the output from ReporterPLUS
- HTML tags (see to the Note below)

These factors can interact with one another. For example, when there are conflicting formatting commands, the formatting specified in the ReporterPLUS template overrides formatting in the Word template.

Note

You can type HTML tags into the heading or body of the Text tab. In addition, there can be HTML tags in your model, in an external document, or in an inserted file. If you want your generated document to use any of these tags, you must specify **Pass through HTML** in the Default Document Properties or the Template Properties dialog box. Refer to the ReporterPLUS online help topic “Using HTML tags to format your document” for more information.

Formatting You Can Specify in ReporterPLUS

As you add text and attributes to the heading or body of the Text tab, you begin to define the format of the generated document. The information in the heading of the Text tab becomes a heading in the generated document, and the information in the body of the Text tab becomes part of the body of the document.

As mentioned previously, the nodes in the template view form a hierarchical structure. By default, the level of the node in the template view determines the level of the heading applied to that information in the generated document (except for text documents, which do not support most formatting). That is, a first-level template node is formatted with a Heading 1 style in Word; a second-level template node is formatted with a Heading 2 style, and so on.

In addition, you can add the following formatting to a ReporterPLUS template:

- Headers and footers
- Page numbers
- Line and page breaks
- Tables
- Table of contents
- Font characteristics (for example, bold, italic, point size)
- Paragraph characteristics (such as styles and justification)

By using ReporterPLUS options, you can also specify HTML navigation features.

Output Type's Effect on Formatting

The same ReporterPLUS template can produce different results depending on the output type you specify when you generate a document.

For example, if your ReporterPLUS template includes a page break command:

- In Word, it creates a new page and sometimes a new section. See **Using Multiple Headers and Footers**.
- In PowerPoint, it creates a new slide.
- In HTML, it creates a new page (a separate `.html` file).
- In text, it has no effect at all.

Formatting in a Template or Style Sheet

If you are generating a Word, PowerPoint, or HTML document, your ReporterPLUS template produces different results depending on the template (`.dot` or `.pot` file) or HTML style sheet (`.css` file) you specify. You can specify default templates and style sheets for ReporterPLUS to use, as well as associate a particular ReporterPLUS template with a particular template and style sheet. For example, you can use a Word template that includes numbered headings and specifies a particular font for heading and body text. For PowerPoint, you could choose a template with a colored background and contrasting text.

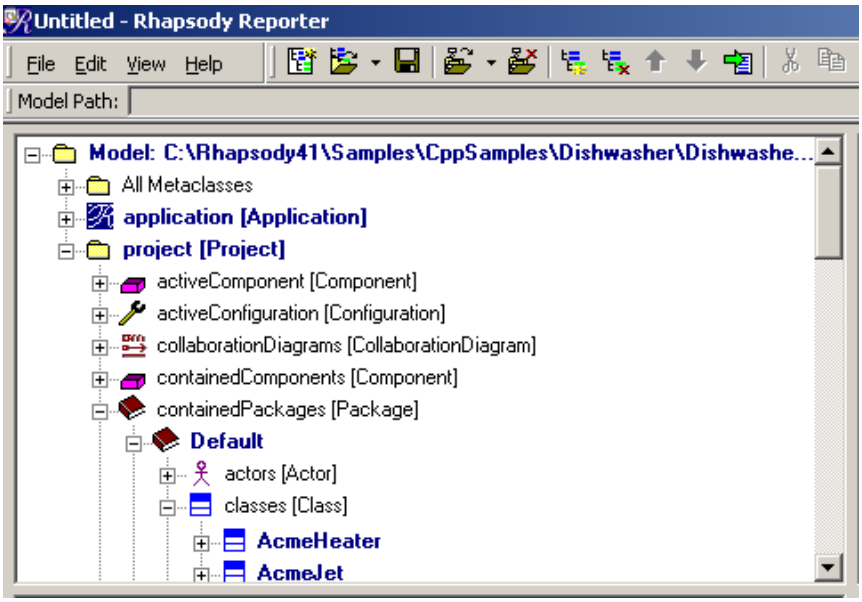
For more information, see **Using a Word Template to Add Formatting** and **Associating a Word Template with a ReporterPLUS Template**.

Note

Specifying a Word or PowerPoint template or HTML style sheet is optional. For more information, see **Differences among the Types of Templates**.

Generic and Model Elements

When you open a model in ReporterPLUS, you see actual elements from your model as well as generic elements. Generic elements represent types of model elements—such as classes and diagrams—that might exist in any Rhapsody model. Model elements are *actual* elements from a *specific* model. By displaying both generic and model elements, ReporterPLUS provides you flexibility in designing templates.



Generic elements enable you to create a template that refers to elements that might appear in any model, rather than to elements that are specific to a single model. It is usually faster to build a template with generic elements. For example, if you want to extract all the classes from a model, you can add a single generic class element to your template, rather than searching through your model to find every class and adding it to the template. This enables you to use the template with any model: when you use the generic class element to build your template, no matter what model you have open, ReporterPLUS extracts the classes.

Model elements enable you to add text or diagrams quickly from a specific model to your template. You can use model elements when designing a template to be used only with one specific model. If you use this template with a different model, ReporterPLUS might not extract any model data because the names of the model elements are not the same.

Creating a Simple Document

This section describes these basic ReporterPLUS facilities and operations:

- Using different template types
- Opening an existing template
- Definitions of sample templates
- Using a sample Word template
- Using a sample HTML template
- Using your corporate standards for reports
- Opening a model for report generation
- Making simple template modifications
- Navigating HTML report options

Differences among the Types of Templates

It is important to understand the difference among using a ReporterPLUS template and a Word or PowerPoint template or an HTML style sheet. The ReporterPLUS template is fundamental to ReporterPLUS: it contains the information that ReporterPLUS uses to extract elements from a model. The structure of the ReporterPLUS template controls the structure of the generated document. In addition, a ReporterPLUS template might contain formatting commands such as page breaks and text alignment.

Any ReporterPLUS template can be used to generate any kind of document (Word, PowerPoint, HTML, or text). However, some of the templates are structured for a specific type of output as noted in the **Standard ReporterPLUS Templates** section.

Using a Word or PowerPoint template or an HTML style sheet is not required. Yet you may use one of these to simplify your formatting work or comply with your company's report standards. See **Formatting You Can Specify in ReporterPLUS** for more information.

Opening a Model

Before you begin to work with the template, you must first open your model and display it in the *model view* (upper, left pane of the ReporterPLUS window).

To open a model, follow these steps:

1. Select **File > Open Model**.
2. In the **Open Model** dialog box, navigate to a model you have created or to the `Samples` for your development code (C++, C, or Java) under the main Rhapsody directory.
3. Select a model.
4. Click **Open**.

Alternatively, you can open a model by clicking the **Open Model** icon  or by selecting **Recent Model Files** from the File menu or toolbar.

Opening an Existing Template

To select an existing template, follow these steps:

1. If it is not already open, following the instructions in **Methods for Starting ReporterPLUS**.
2. Select **File** > **Open Template**.
3. In the Open Template dialog box, navigate to the `Templates` directory (under the main ReporterPLUS directory). For descriptions of all of the templates, see **Standard ReporterPLUS Templates**.
4. Select a template from the list and note the template description displayed in the text box to the right of the list of files.
5. Click **Open**. The template is displayed in the template view (below the model browser) and the name of the open template is displayed in the title bar of the ReporterPLUS window.

Alternatively, you can open a template by clicking the **Open Template** icon or by selecting **Recent Template Files** from the File menu or toolbar.

Note

If your template included a "block," this element is automatically changed to an "object" in the Rhapsody 7.2 or greater version of ReporterPLUS.

Examining a Selected Template with Your Model

If you want to see how the selected template works with your model, follow these steps for a quick test:

1. Be certain that your model is displayed in the upper left browser.
2. With selected template shown in the panel below the model, click the **Generate Document** icon or select **Edit** > **Generate Document**.
3. Select the type of output desired from the pull-down menu in the **Save as type** field in the **Generate Document** dialog.
4. Select and directory for the output and enter a name for this test file in the **File name** field.
5. Click **Generate**.
6. View the generated results so see what the generic template produces.
7. At this point, you may decide to select a different template, use this template, or save a section of this template that you intend to use in a customized template.

Standard ReporterPLUS Templates

ReporterPLUS provides a large number of standard templates. These templates can be used as they are or modified to meet your needs. The following table lists all of these pre-defined templates with a description of each and the preferred output formats. The rich text format (.rtf) and plain text (.txt) formats are not listed in the following table because the design capabilities of those output formats is simplified and can be used for any of these templates. However, HTML, Word and PowerPoint provide more formatting capabilities, so some of the generic templates look better in one or more of the other three formats.

Template Name	Preferred Output Formats	Description
class.tpl	HTML or Word	This template defines the information about the classes in the project. It prints the documentation of the following elements of the class: • Attributes • Operations • Relations • Events • Statechart and Activity Diagrams (including the diagrams)
ClassHierarchyBrowser.tpl	HTML	This template generates a list of the model's class hierarchies with descriptions and a hyperlinked index.
ClassHierarchyBrowser2.tpl	HTML	This template generates the model's root classes with a title page and hyperlinked index.
ClassHierarchyBrowser3.tpl	HTML	This template generates the model's class hierarchies including *Java script*, a title page, and a hyperlinked index.
ClassOverviewPresentation.tpl	PowerPoint	This template shows an overview of the classes in Package(s) as a presentation.
ClassReport.tpl	Word	This template extracts the attribute and operation information for all classes in the model.
DetailedClassReport.tpl	Word	This template produces a Table of Contents and lists all classes along with their attributes and operations.
DiagramOrientedReport.tpl	Word	This template shows all packages that have *object model diagrams*. For each package, it displays all of the object model diagrams, with each diagram followed by the contained elements appearing on that diagram.

Template Name	Preferred Output Formats	Description
DiagramReport_Hierarchy.tpl	Word	This template defines the information related to the diagrams of the project. It includes both project level and package level diagrams for the following: 1. Collaboration Diagrams 2. Component Diagrams 3. Deployment Diagrams 4. Object Model Diagrams 5. Sequence Diagrams 6. Use Case Diagrams 7. Statecharts and Activity Diagrams
Diagrams.tpl	Word & PowerPoint	Displays titles and all diagrams in the model.
EgalitarianPackageReport.tpl	Word	This template generates a diagram-oriented report. The diagrams represented in the report are object model, use case, sequence, deployment, collaboration and component diagrams.
EgalitarianPackageReport2.tpl	Word	This template generates a diagram-oriented report of the classes in the project. The diagrams represented in the report are object model, use case, sequence, deployment, collaboration and component diagrams.
FullDetailedProjectReport.tpl	Word, PowerPoint, or HTML	This generic template defines the information about the complete project. It includes all the model elements of the project and all the diagrams of the project.
GetStarted.tpl	Word & PowerPoint	Shows all diagrams and classes in a model.
HierarchicalPackageReport.tpl	Word	Recursively descend through packages showing their diagrams, classes, and nested packages.
IndexedClassReport.tpl	Word	Print object model diagrams and classes in a package.
MetamodelReport.tpl	Word	Show all packages that have object model diagrams. For each package, it displays all of the object model diagrams and classes within the package. For each class, the template extracts the its attributes, operations, and associations.
ModelMetrics.tpl	Word or HTML	This prints out metrics for the entire model and each package in the model

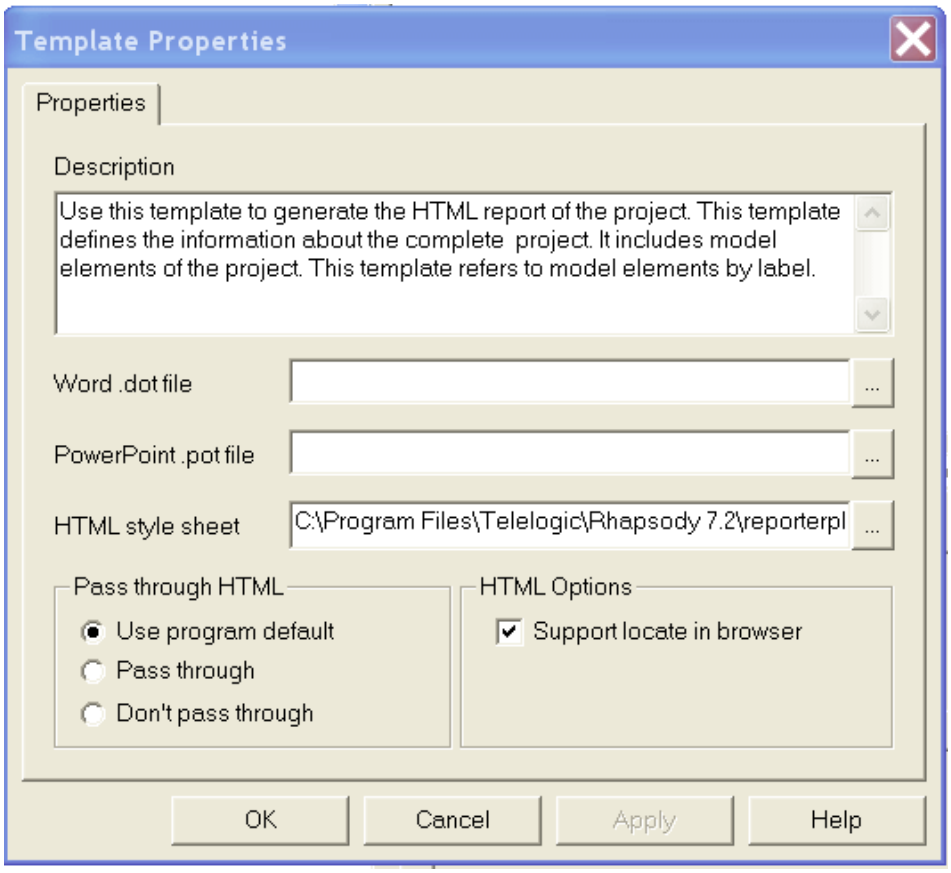
Template Name	Preferred Output Formats	Description
OverriddenProperties.tpl	Word	This template defines the information about the overridden properties. It includes overridden properties of all the metaclasses of the model. It prints the metaclass names followed by a table of overridden properties for each metaclass. It has Q expressions to exclude the metaclasses, which do not have any overridden properties.
PackageReport.tpl	Word	This template defines the information related to the packages and all the elements of in these packages: • All packages (nested structure) • All the diagrams in each package • Elements contained in the packages • Sub Packages and their elements It does not include project level information.
PackageReportFiles.tpl	Word	This template defines the information related to the packages and all the elements of in these packages: • All packages (nested structure) • All the diagrams in each package • Elements contained in the packages • Sub Packages and their elements It does not include project level information.
ProjectReport.tpl	Word	This generic template defines the information about the complete project. It includes all the model elements and all the diagrams in the project. It includes and title page and table of contents.
RequirementsTable.tpl	Word	This template lists the requirements, use cases, actors, and all diagrams.
Rhapsody HTML Exporter.tpl	HTML only	The template generates a comprehensive HTML report of the project. The template defines the information about the complete project including model elements by label.
SequenceDiagramWithClasses.tpl	Word & PowerPoint	This template provides information about the sequence diagrams and the classes participating in the sequence diagrams. It includes and title page and table of contents.
Statechart.tpl	Word	This template provides information about the statecharts of the project. It includes the statecharts, states, and the list of elements contained in each of these statecharts.

Template Name	Preferred Output Formats	Description
SysMLDataFlowInPackage.tpl	HTML or Word	This specialized template lists the data imported from System Architect (SA) into a Rhapsody package.
SysMLreport.tpl	Word or HTML	This template uses the SysML profile to provide the underlying stereotypes to generate a document. If SysML was not selected as the Project Type when it was created, you cannot take full advantage of the features of this template. The main sections the document produces are in the following order (if they exist): • Requirements Diagrams • Use case diagrams • Sequence Diagrams • Structure Diagrams • Object Model Diagrams • External Block Diagrams • Internal Block Diagrams • Parametric Diagrams • Element Dictionary • Model Configuration The document produced is hyperlinked where appropriate.
TabularViews.tpl	any selected format	This template displays all of the diagrams in the model.
UseCaseDiagramsDetailedReport.tpl	Word	This template supplies the information about use case diagrams, use cases and actors of the project. It lists all the use case diagrams followed by actors and use cases related to the use case diagram.
UseCaseReport.tpl	Word	This template prints all of the use cases and use case diagrams in a model.

Setting Standard Template Properties

After selecting one of the ReporterPLUS templates, you may set standard template properties with these steps:

1. Select the **File >Template** Properties option. The dialog box displays information about the selected template, as shown in this example of a HTML template.

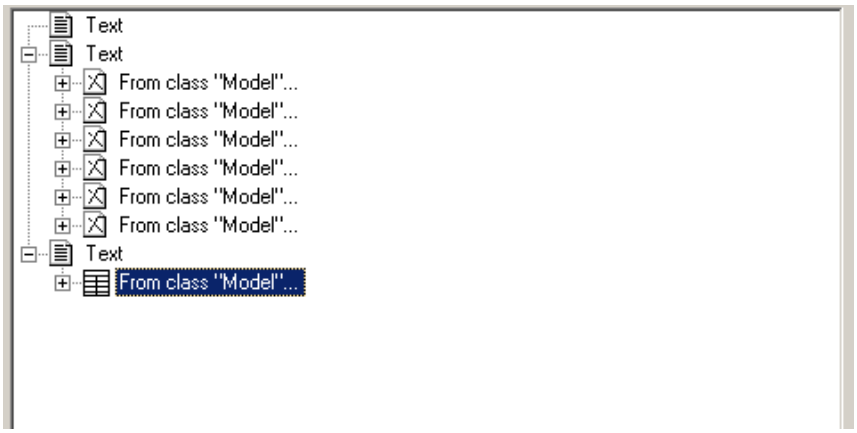


2. Depending on the type of template selected, you can navigate to an existing Word.dot file, PowerPoint template (.pot file), or an HTML style sheet (.css file) to format the text style and other design features available in the selected template.

3. When you have located the template you want to use, click **OK**.

Exploring the GetStarted Template

The GetStarted template creates a document that includes all the diagrams and all the classes in a model, arranged in alphabetical order by name. The template extracts the diagram names and images, and the class name and description. The template view displays the GetStarted template you just opened, as shown in the following example.



Follow these steps:

1. Expand the template nodes by clicking on the plus signs (+).
2. Click on the first node in the template view to select it. Note the text displayed on the Text tab. This is a text node, which contains only the Table of Contents for the document.
3. Click on the next node (`From class "Model"`…), expand the tree by clicking on the (+) sign and take a look at the information on the Text tab. You see boilerplate text (in black), attributes (in blue), and formatting commands (in green). This node is an iteration node; in conjunction with its subnode, it extracts every collaboration diagram from a model.
4. Click through the other tabs in the template node view to see what is there. The information on the Sort tab tells ReporterPLUS to sort the collaboration diagrams alphabetically by name. The information on the No Data tab tells ReporterPLUS to print “No collaboration diagrams” when you generate a document from a model that does not contain any collaboration diagrams.
5. Click on the subnode (…`iterate over instances of class "Collaboration Diagram"`) The «`$name`» attribute tells ReporterPLUS to print the diagram name in the generated document; the «`image`» attribute represented by the `[INSERT PICTURE]` command prints the diagram itself.
6. Click on the next node. It is a text node that holds a section heading for the table.

7. Click on the table node (also labeled `From class "Model"…`). A table node is simply an iteration node that displays information in a table. In conjunction with its subnode, the table node extracts every class from the model and prints the class name, description, and displayName in a three-column table. The headings for the table columns are in the body section of the Text tab.

If you click on the Iteration tab, you can see that the **Table** box is checked. To turn a regular iteration node into a table node, you check this box and set up the columns. See the online help for detailed information on using tables.

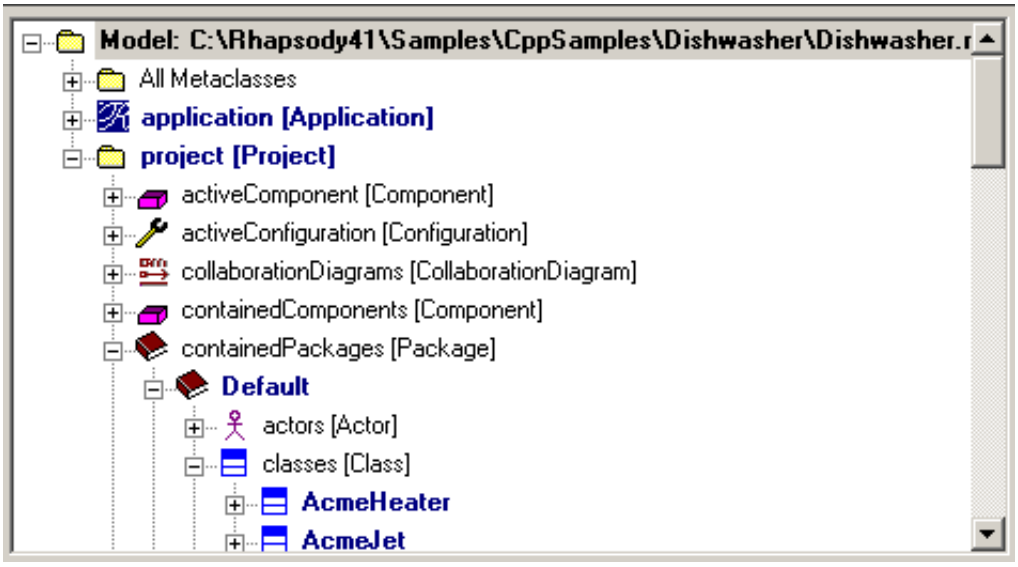
8. Click on the subnode (…`iterate over instances of class "Class"`) to see the definition for the rest of the table. The attributes on the Text tab tell ReporterPLUS to include the class name, description, and displayName in the table.

Exploring the Model View

Note how the model view changes when you open a model. The black text represents generic elements just as it did before opening the model. But much of the text is now blue. This blue text represents model elements.

Follow these steps:

1. Expand the `Project` node and the `containedPackages` within the `Project` node.
2. Expand the classes node under the `Default` package node. As shown in the following figure, you can see actual classes from the Dishwasher model (such as `AbstractFactory`, `AcmeFactory`, and `AcmeHeater`).



Generating Documents

Now that you have opened a ReporterPLUS template and a model, you can generate documents in any of the four output types that ReporterPLUS supports: PowerPoint, Word, HTML, RTF, and text.

Note

If you are generating reports in Linux and receive an error, you can correct this problem by starting up Rhapsody and exiting it. Then return to ReporterPLUS to generate the reports.

You can use the same ReporterPLUS template for all document types. However, there are differences among the output formats that affect the way the document looks and, in some cases, even the document content. As you generate different types of documents, note these differences.

This section describes using the `GetStarted.tpl` and `Dishwasher.rpy` to generate PowerPoint, Word, and HTML documents.

Note

You can also generate a text document from the GetStarted template by selecting **Text File** in the Generate Document dialog box. However, because the text format does not support graphics, you see only the diagram names and class information.

Generating a PowerPoint Presentation

To generate a PowerPoint presentation, follow these steps:

1. Select **Edit** > **Generate Document** to display the Generate Document dialog box.
2. Select a drive and folder, and type a name for the document.
3. In the **Save as type** drop-down list, select **Microsoft PowerPoint Presentation**.
4. Click **Generate**. Alternatively, you can generate a document by clicking the **Generate Document** icon  . ReporterPLUS generates the document, opens PowerPoint, and displays the resultant presentation.
5. When you are finished viewing the presentation, close PowerPoint.

Generating a Word Document

To use the same template and model to generate a Word document, follow these steps:

1. Select **Edit > Generate Document**.
2. Select a drive and folder, and type a name for the document.
3. In the **Save as type** drop-down list, select **Microsoft Word Document.**
4. Click the **Generate** button. ReporterPLUS generates the document, opens Word, and displays the document.
5. When you are finished viewing the document, close Word.

Using a Word Template to Add Formatting

When generating a Word document, ReporterPLUS looks in two places for formatting information:

- The ReporterPLUS template
- The Word template (`.dot` file)

When you install ReporterPLUS, `RhapRepDot.dot` is specified as the default Word template. You can change this default template, or you can associate a Word template with a particular ReporterPLUS template.

If there is no default Word template specified and no Word template associated with the ReporterPLUS template, Word uses its own default template—`normal.dot`. In addition to using the Word templates included with ReporterPLUS, you can develop your own templates.

Note

This section describes Word templates only, but the same principles apply to PowerPoint templates and HTML style sheets.

Change the Default Template

The Word document you just generated used `Reporter1.dot`, which employs a numbered heading style. The next template uses a heading style with colored text and no numbers. To change the default template, follow these steps:

1. Select **View > Default Document Properties**.
2. Select the Word tab.
3. Click the browse button to the right of the **Word template file** field.
4. In the Select Word Template File dialog box, navigate to the `Templates` folder (under the main ReporterPLUS directory).
5. Select `ReporterSimpleClassReport.dot`.
6. Click **Open**.
7. Click **OK**.
8. Select **Edit > Generate Document** and regenerate the Word document. You might want to give this document a different name to compare the results.
9. When you have finished viewing the document, close Word.
10. In ReporterPLUS, select **View > Default Document Properties** and change the default Word template back to `RhapRepDot.dot`.

Note

This last step applies to this example only. Normally, you can use any Word template as the default.

Generating an HTML Document

In this step, use the same ReporterPLUS template and model to create an HTML document. You should keep the following behavior in mind when generating HTML documents:

If your ReporterPLUS template has page breaks (as the GetStarted template does), ReporterPLUS creates a separate `.html` file for each page and adds one or both of the following for navigation:

- Next and back arrows at the top of each page
- A table of contents frame

By default, ReporterPLUS creates a subdirectory named `pix` in the directory you specify for the HTML files and puts the graphics in it. However, you can specify a different location for graphics using an absolute or relative path.

Associating an Image File with a Model Element

Rhapsody allows you to associate an image file with a model element. This image can then be used to represent the element in diagrams in place of the standard graphic representation.

To associate an image file with a model element, follow these steps:

1. Right-click the model element in the browser.
2. Select the **Add Associated Image** menu option.
3. Select the appropriate image file for the model element.

Specifying HTML Options

To define the HTML characteristics for the generated report, follow these steps:

1. Select **View > Default Document Properties**. The HTML tab displays the options you can set for your HTML documents.
2. In the **Navigation** section of the HTML tab, make sure **Table of contents** is selected (it is the default setting), and select **Back/next arrows** (which lets you see both types of navigation).
3. Click **OK**.

Generating an HTML Document

To generate the HTML file, follow these steps:

1. Select **Edit > Generate Document**. The Generate Document dialog box opens.
2. Select a drive and folder, and type a name for the document.
3. In the **Save as type** drop-down list, select **HTML Page**.
4. Click **Generate**.

ReporterPLUS generates the document, opens your default browser, and displays the first page. Click on the headings in the contents frame or click the next and back buttons to view other pages in the document.

Note

If you have an older browser and selected the **Support older browsers** option in the Default Document Properties dialog box, you see a message stating that a newer version of the browser yields better results, and suggesting several options. Selecting the non-resizable table of contents creates a dynamic table of contents with icons, but of a fixed size. Selecting the static table of contents creates a contents frame, but it is not dynamic and does not include icons.

Displaying Your Icons for Stereotypes

When generating an HTML report, the generic icon for the object type in the report browser is usually displayed in the report. For example, every class in the report includes the standard Rhapsody graphic for a class.

However, if you want to create and display customized icons for the stereotypes in your model, follow these steps:

1. Create your own icon in a .gif file format in 16x16 pixels.
2. Name the icon using the same name as the stereotype it represents. For example, if the stereotype is “Car,” the icon name should be “Car.gif.”
3. Add your new graphic to the `icons.zip` file located in your Rhapsody installation of ReporterPLUS (e.g., C:\Program Files\Telelogic\Rhapsody 7.4\reporterplus).
4. Generate the HTML report with your icons.

Rhapsody HTML Exporter Template

The `Rhapsody HTML Exporter.tpl` file is an easy-to-use template to generate an HTML report. Follow these steps to set up Rhapsody and ReporterPLUS to use this template:

1. In Rhapsody set the project property `General::Graphics::ExportedDiagramScale` to be "NoPagination." This defines the diagrams to be exported to ReporterPLUS as one picture as they look in Rhapsody with 100% zoom. Since the HTML browser can show bigger diagrams using scroll bars, there is no size limitation, as with a Microsoft Word document page size.

2. To enable HTML navigation for diagrams, add the following flags in the `rhapsody.ini` file:

```
[ReporterPLUS]
EnableLoadOptions=TRUE
LoadImageMaps=TRUE
```

The HTML report generated with this template uses a JAVA applet in the tree browser. This applet supports a **Locate In Browser** button for navigation from the right-side pane to the left-side browser tree.This applet requires the JAVA 2 Runtime Environment to be installed on your PC.

With the settings made and the correct version of Java available, follow these steps to locate and open the `Rhapsody HTML Exporter.tpl` file:

1. If it is not already open, following the instructions in **Methods for Starting ReporterPLUS**.

2. Select **File** > **Open Template**.

3. In the Open Template dialog box, navigate to the `Templates` directory (under the main ReporterPLUS directory).

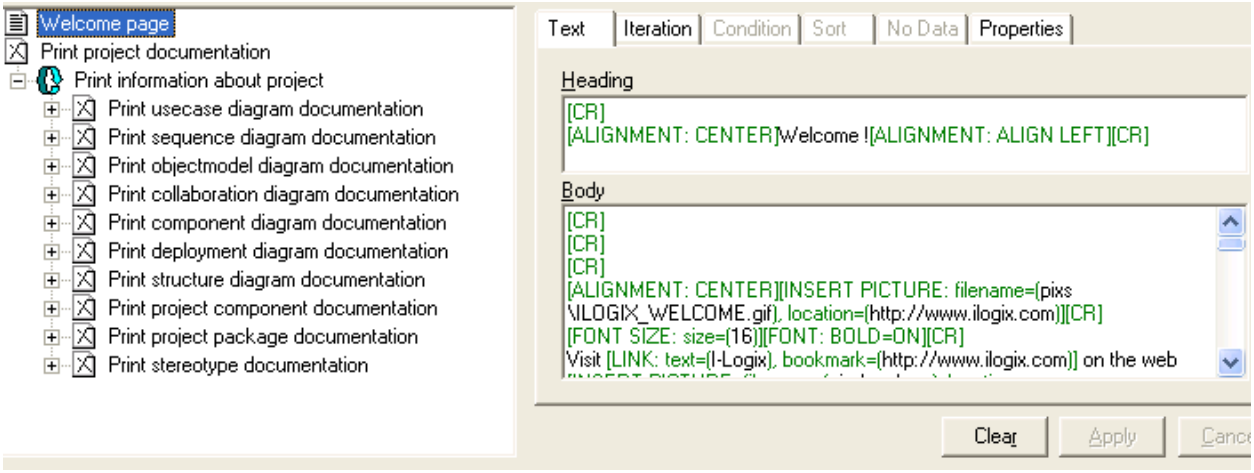
4. Select `Rhapsody HTML Exporter.tpl`. A description of the selected template displays in the text box to the right of the list of files.

5. Make any changes to the **HTML Exporter Template Structure** that are desired.

6. To insert the **Locate in Browser** button in the finished report, use the **Locate** command available from the **Text Tab**.

HTML Exporter Template Structure

The HTML Exporter template's basic structure is shown in the example below. You need to customize the `Welcome page` (highlighted on the left with the Q language displayed in the **Text** tab on the right). The Welcome page should contain any company and project information you need to identify and introduce your project's HTML report.



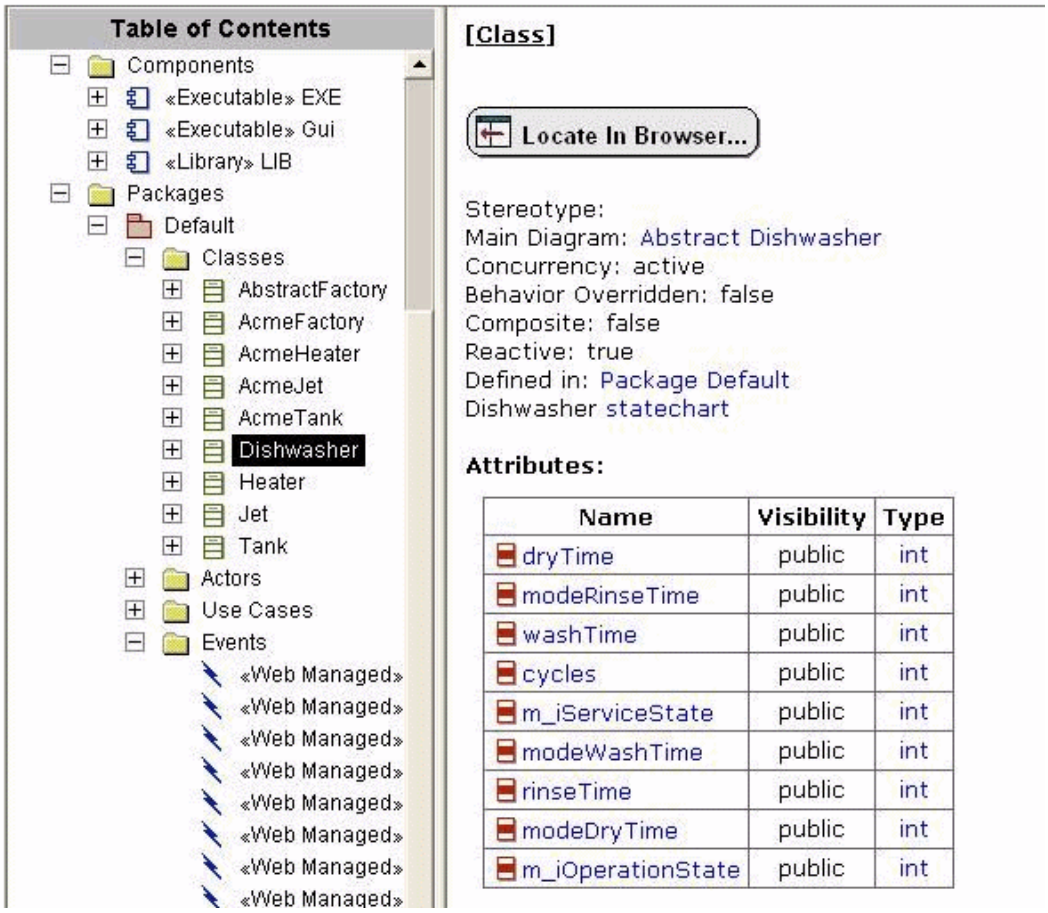
Examine the subheadings under the `Print information about project` to move, add, or delete any items from the report.

Generating an HTML Exporter Report

After you have set up the report to meet your needs, follow these steps to generate the report:

1. Select **Edit > Generate Document**. The Generate Document dialog box opens.
2. Select a drive and folder, and type a name for the document. In the **Save as type** drop-down list, select **HTML Page**.
3. Click **Generate**.

ReporterPLUS generates the HTML report, opens your default browser, and displays the report in two columns with a browser on the left and the report on the right, as shown in this example.



The report contains all of the usual hyperlinked information. In addition, for each description displayed in the report on the right, clicking the **Locate In Browser** button positions and highlights the node in the project (displayed in the browser on the left). This is particularly helpful for large reports.

Creating Diagram Hot Spots

Boxes, representing classes, comments, use cases and other elements in a diagram output as HTML, can be changed to hot spots to open that element's page. To switch on "hot spot" navigation make the following `rhapsody.ini` file changes:

```
[ReporterPLUS]
EnableLoadOptions=TRUE
LoadImageMaps=TRUE
```

The `EnableLoadOptions` flag enables reading of load optimization flags, such as `LoadImageMaps`.

Viewing Reports Online

After creating reports using ReporterPLUS templates and facilities, follow these guidelines for viewing the reports online:

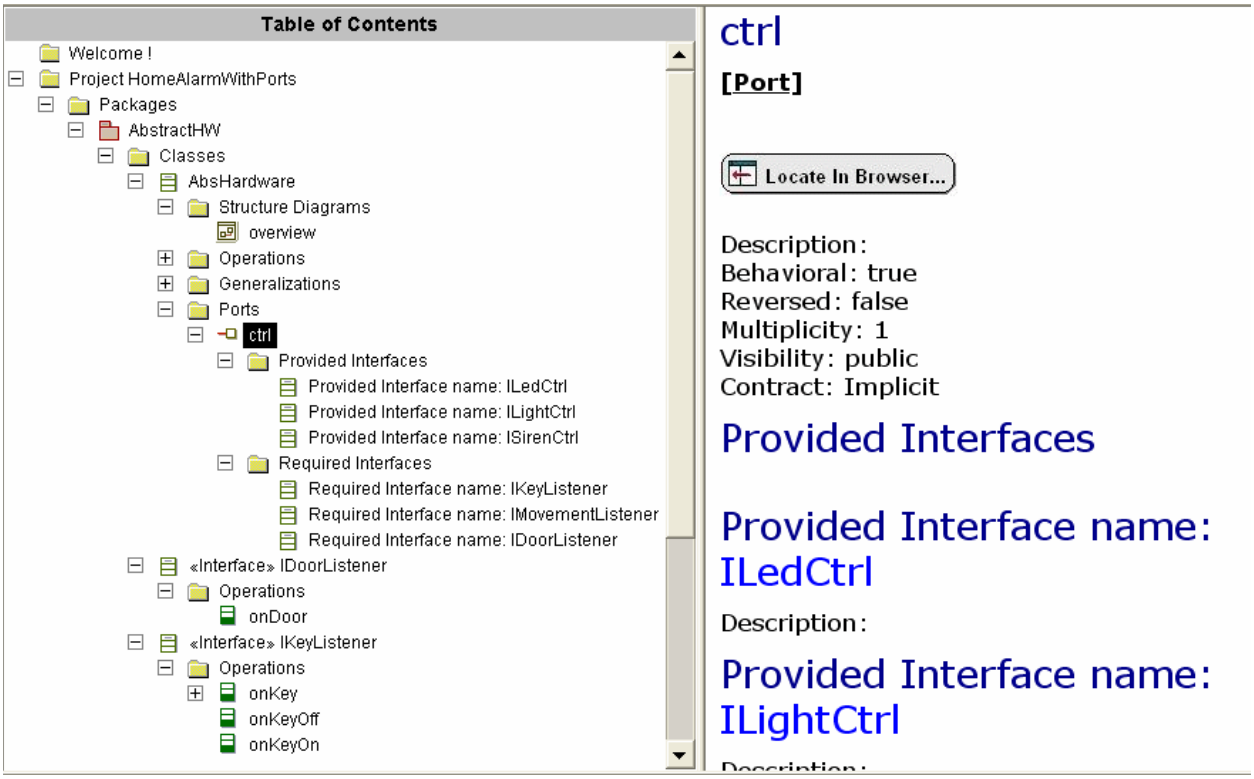
- For reports generated in Linux, view the HTML reports in Mozilla Firefox and the RTF reports in Open Office 2.0 or higher.
- For reports generated in Windows, view HTML reports in any standard browser available on the PC and for the other report formats, the appropriate programs for viewing these reports launch when the report files are clicked to launch.

Generating a List of Specific Items

If during development you want to generate a list of items, such as all of the ports using an interface, you can focus the generated report on that section of the model. To generate a list of specific items in a model, follow these steps:

1. Display the model in Rhapsody.
2. In the browser, select the section of the model containing the specific items that you need in a list.
3. Select **Tools > ReporterPLUS > Report on selected package**.
4. Select the template you want to use for the report and generate and save the report. The following sample, produced using the HTML Exporter Report template, lists both the

provided and required ports for the AbstractHW package in the Home Alarm with Ports project.



Creating HTML Reports for Large Models

If you are using the HTML Exporter template to generate reports for a large model, you have some special settings and adjustments to make for ReporterPLUS to handle these reports efficiently.

First, if you are using *Windows XP*, you should apply this Microsoft graphics patch so that ReporterPLUS can manage a large number of diagrams:

Update for Windows XP (KB319740)

Managing Long Paths in a Generated HTML Report

Microsoft Windows' system file name length limit is 256 characters. The hierarchy of a generated HTML report reflects the hierarchy of the model and, therefore, could exceed Microsoft's file name length limit.

ReporterPLUS guards against this problem. When the regular hierarchical path is likely to exceed the Windows' file name length limit, the sub-directory is generated to the parent's parent directory until the generated file name length does not exceed the system file name length limit.

Generating Large Model Reports in Multiple Directories

ReporterPLUS normally generates the files of an HTML report into a single directory. However, for very large models, this could exceed Microsoft Windows' limit for the number of files in one directory.

To avoid this problem, ReporterPLUS generates the HTML report files into multiple directories when the `GenerateMultifolderReport` flag is set to `TRUE`. To set up your ReporterPLUS, make the following setting changes to the `rhapsody.ini` file in the installation directory:

```
[General]
InvokeReporterDll = FALSE
[ReporterPLUS]
EnableLoadOptions=TRUE
LoadImageMaps=TRUE
ModelSize=10
GenerateMultifolderReport=TRUE
SupportedMultifolderReportMetaClasses=package
```

The `EnableLoadOptions` flag enables reading of load optimization flags, such as `GenerateMultifolderReport`.

Additionally, you can set the `SupportedMultifolderReportMetaClasses` variable to indicate the types of model elements to receive separate directories. Use these flag values to define those subdirectories:

- `package`
- `profile`
- `component`
- `class`
- `actor`
- `usecase`
- `object`
- `module`

For example, to generate separate subdirectories for packages and classes, set the flag as follows:

```
SupportedMultifolderReportMetaClasses=package,class
```

Generally, setting this flag to `package` is sufficient to generate the necessary subdirectories.

Optimizing Memory for Large Reports

The `ModelSize` flag controls the ReporterPLUS memory buffer size. The model size, an integer value from 1 to 10, defaults to “1” (minimum memory). For a large model, the model size value should be changed in the `rhapsody.ini` file to “10” (maximum memory).

```
[ReporterPLUS]
EnableLoadOptions=TRUE
ModelSize=10
```

The `EnableLoadOptions` flag enables reading of load optimization flags, such as `ModelSize`. In addition, very large reports should be generated on machines with at least 2 GBytes of memory.

Creating Your Own ReporterPLUS Template

In this section, you create your own ReporterPLUS template (which is similar to the GetStarted template, as described in **Creating a Simple Document**). This section describes how to add nodes to a template by dragging elements from the model view to the template view. In addition, you learn how to add attributes and boilerplate text to the nodes and specify some basic formatting.

This section describes the following tasks:

- Using **All Metaclasses** to add generic elements to a template.
- Adding a generic class from the model.
- Adding boilerplate text and attributes.
- Adding simple formatting commands.
- Saving a template.
- Generating and viewing a document with your template.

Before You Begin

Before you begin, do one of the following depending on your situation:

- If you are continuing from the previous section's tasks and ReporterPLUS is still open, do the following:
  - Close the model you have open. Select **File > Close Model**.
  - Open a new, blank template. Select **File > New Template**.

    If the GetStarted template was previously open, it closes, leaving the template view blank so you can start building a new template.
  - Continue with **Extracting All Diagrams from a Model**.
- If ReporterPLUS is not running, do the following:
  - Start ReporterPLUS. Select **Start > All Programs > Telelogic > Telelogic Rhapsody *version #* >Rhapsody ReporterPLUS *version #* > Rhapsody ReporterPLUS *version #***).
  - Click **Cancel** on the Select Task dialog box.
  - Continue with **Extracting All Diagrams from a Model**.

Extracting All Diagrams from a Model

The first step is to create a ReporterPLUS template that generates a document that includes all the diagrams from a model. There are two ways to do this:

- Open the model in ReporterPLUS, search the model view for all the diagrams, and add them one-by-one to the template view. This is referred to as using model elements to build a template.
- Add generic elements that represent all the diagrams in a model to the template view.

If you use the first method, your template selects only diagrams that have the same names as those in the model used to build the template. This works fine if you use the template with only a single model, but it is unlikely that you are able to use it effectively with other models.

The second method uses a generic element, which represents a type of model element that might exist in any Rhapsody model. By using generic elements, you create a template that can be used with any model. There is another advantage to using generic elements—it is usually faster to build a template with them than with model elements.

To extract diagrams from a model, follow these steps:

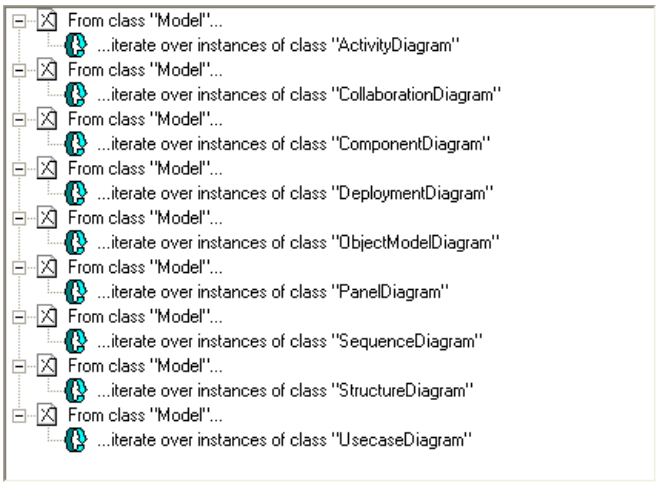
1. In the model view, expand **All Metaclasses**.

   The generic elements under **All Metaclasses** represent all the model elements of their type in a model. For example, **CollaborationDiagram** represents all the collaboration diagrams in a model, **ComponentDiagram** represents all the component diagrams in a model, and **Class** represents all the classes in a model.

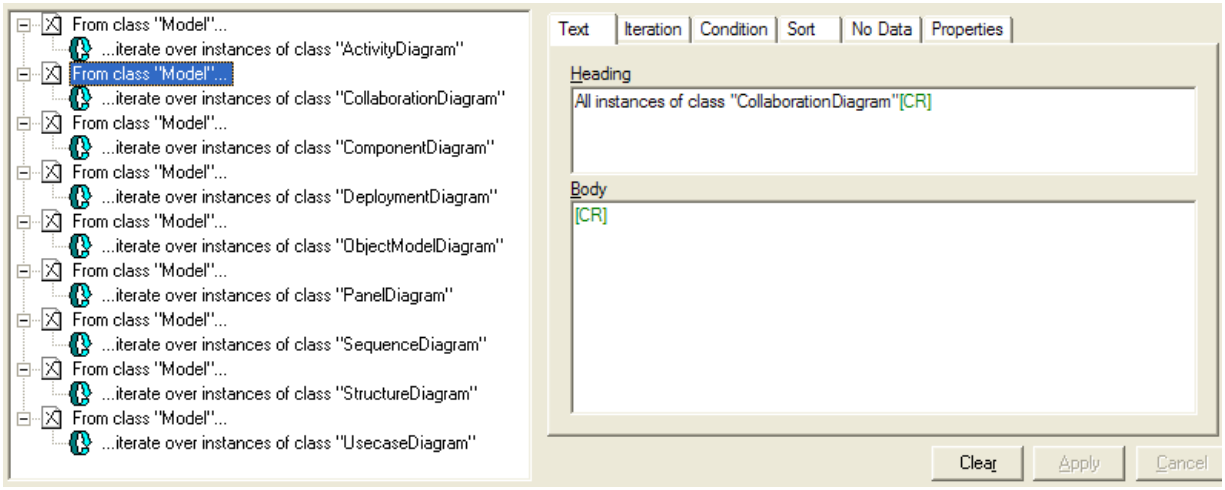
2. Click the **CollaborationDiagram** node and drag it to the template view.

3. Repeat for each of the diagram nodes (**ComponentDiagram**, **DeploymentDiagram**, **ObjectModelDiagram**, and so forth).

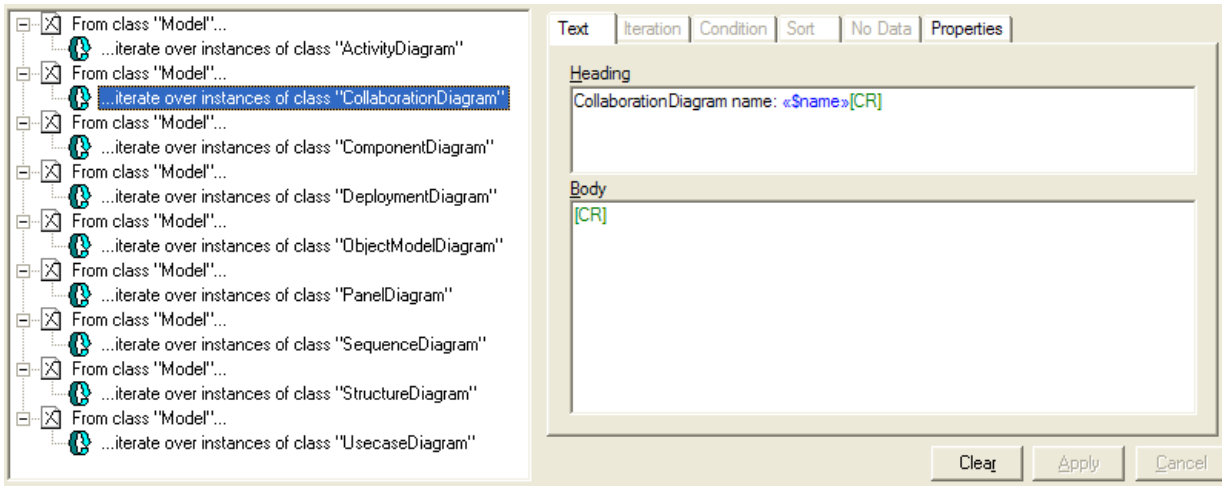
   This action creates an iteration node and an iteration subnode. Your template view should resemble the following figure:



4. Click an iteration node and notice the information displayed on the Text tab, as shown in the following figure. In the generated document, this information is displayed once, at the beginning of the section of the document that contains the diagrams.



5. Click an iteration subnode and notice again the Text tab. You see boilerplate text (in black) and attributes (in blue), as shown in the following figure. In the generated document, the information on this node displays for *each diagram* extracted from the model. In this case, for each diagram, the document has a heading that includes the diagram name.



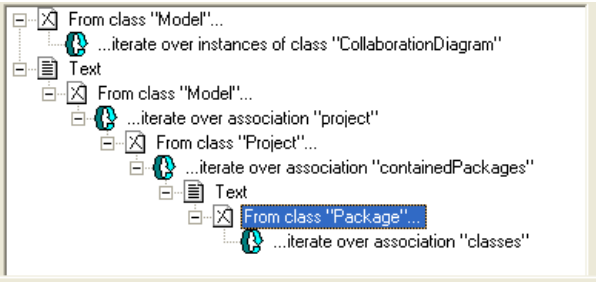
As you can see, ReporterPLUS automatically adds some default text and attributes to the Text tab. You can edit or remove the text. Attributes tell ReporterPLUS what information to extract from the model. For example, the `«$name»` attribute tells ReporterPLUS to include the name of the diagram in the generated document. You add additional text and attributes in a later task. Next, see **Extracting All Classes in the Package**

Extracting All Classes in the Package

In this task, you add a node to the template for a generic element that is not from **All Metaclasses**. There might be times when you want only elements from a specific part of the model, such as the **Package**, rather than all elements in the model. There are several ways to do this (all of which are covered in the ReporterPLUS online help). In this task, you build a series of iterations that creates a path to the generic element.

To extract all classes in the package, follow these steps:

1. In the model view, collapse **All Metaclasses** and expand **Project**.
2. Locate and expand the **containedPackages** node so that you can see the generic elements for it.
3. Drag **project** and then **containedPackages** to the template view, under the nodes you added for diagrams. This creates an iteration node and subnode under which you put an additional iteration.
4. Find **classes** under the **containedPackages** node. Drag **classes** onto the iteration subnode created in Step 3. This creates a third iteration and subnode, which are subnodes of the iteration created in Step 3. Your template view should resemble the following figure:



The nodes you just added tell ReporterPLUS to iterate through the model to find the project, then iterate through the **containedPackages** and extract all the classes. The labels that ReporterPLUS adds to the template view identify what the nodes do and what part of the model they are from.

As mentioned previously, there are several methods for adding nodes and attributes to templates. For details, refer to the ReporterPLUS online help topic “How do I add model text?”

Note

In this section, you added only generic elements to your template. You can also add model elements to a template. See **Building a ReporterPLUS Template for a Specific Model** for instructions. For more information on generic versus model elements, see **Generic and Model Elements**.

Adding Boilerplate Text and Attributes

ReporterPLUS adds some boilerplate text and attributes to the template automatically. In this section, you add additional text and attributes.

To add additional text and attributes, follow these steps:

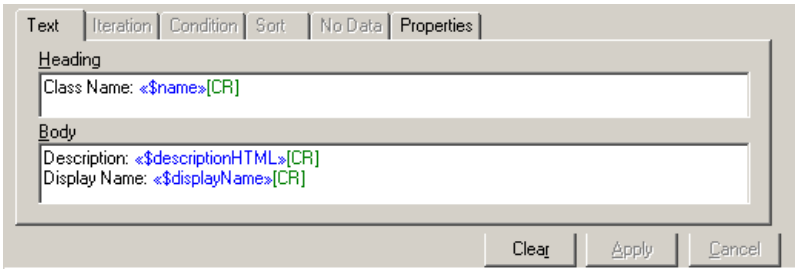
1. Click the class iteration subnode in the template view. This should be the last node in the template (…`iterate over association "classes"`).
2. Click to place the cursor in the body section of the Text tab, before the `[CR]` (carriage return) code.

   Note: If you have difficulty placing the cursor exactly where you want it, just click anywhere in the body of the Text tab, then use the arrow keys to move the cursor into position.

3. Type `Description:` followed by a space.
4. In the model view, click **classes** under the **containedPackages** node.

   The attribute view (upper, right pane in the ReporterPLUS window) now displays the attributes for the selected generic element. These are the items that you can extract from the model and include in your document for that particular element. The attributes available depend on the element selected in the model view.

5. In the attribute view, click-and-drag **descriptionHTML** to the body of the Text tab to the right of the “Description:” item.
6. Press Enter to add a `[CR]` code and move to the next line.
7. Type `Display Name:` followed by a space.
8. In the attribute view, click-and-drag **displayName** to the body of the Text tab to the right of the “Display Name:” item. Your Text tab should resemble the following figure:



When you generate a document from this template, ReporterPLUS includes the name, description, and display name for each class in the `Package`. The class name is a heading in the generated document; the description and display name is body text under that heading.

Note

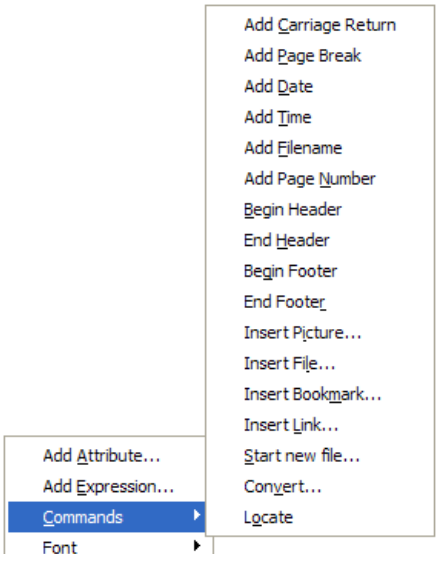
You can also add attributes from the context menu on the Text tab. Right-click in the Text tab, then select **Add Attribute**. See the ReporterPLUS online help for instructions on all the methods for adding attributes to templates.

Adding Formatting

There are numerous types of formatting you can add to your ReporterPLUS template: headings and footers, page numbers, page breaks, and font and paragraph characteristics.

To add formatting, follow these steps:

1. In the template view, click on a subnode; for example, the collaboration diagram iteration (`…iterate over instances of class "CollaborationDiagram"`).
2. Right-click in the body of the Text tab and select **Commands** and select an appropriate formatting commend from the context menu, as shown in the following figure:



3. Click **Apply** on the Text tab.

Note

For object model diagrams that are produced in PowerPoint, it is essential that you include a page break after each diagram. If you do not, some of the diagrams may not display properly.

Saving a ReporterPLUS Template

Until you save and name a template, ReporterPLUS displays the temporary name `Untitled` in the title bar. All ReporterPLUS templates are saved with the extension `.tpl`.

To save a ReporterPLUS template, follow these steps:

1. Select **File > Template Properties** to open the Template Properties dialog box.
2. In the **Description** box, type a brief description of what the template does, such as `extracts all diagrams; extracts classes from Package`.
3. Click **OK** to close the Template Properties dialog box.
4. In ReporterPLUS, select **File > Save Template As**.
5. In the Save Template As dialog box, type a name for the modified template. Although you can store templates anywhere, it is easier to find templates if they are all stored in your own ReporterPLUS `Templates` directory.
6. Click **Save**.

Generating and Viewing Your Document

To generate a report using the template you just created template, follow these steps:

1. Open the model `Dishwasher.rpy` (see **Opening a Model**).
2. Generate a Word document (see **Generating a Word Document**).
3. View the generated document. You should see all the diagrams from the Dishwasher model, each on a separate page. On the last page, there should be a list of classes for the package, with the description and display name for each class. For some of the classes, you see `«No Model Data»` where the documentation should be; you can learn more about this in **Sorting, Conditions, and Missing Data**.

You can try generating a PowerPoint or HTML document from this template.

Building a ReporterPLUS Template for a Specific Model

Although it is recommended that you use generic elements when you build a ReporterPLUS template so you can use it with any Rhapsody model, it is possible to use model elements to build a template for use with a specific model.

To build a template for use with a specific mode, follow these steps:

1. Open the model for which you want to create the template.
2. Locate the model elements (in blue text) in the model view.
3. Drag the specific model elements to the template view.

The rest of the procedures—adding attributes, text, and formatting—are the same as for generic elements.

Note

Even when you are designing a template for a specific model, it may be more efficient to use generic elements rather than model elements.

Q Language

This section describes the *Q Language* embedded in ReporterPLUS. This language is used to accomplish advanced documentation tasks such as the following:

- Placing conditions on iterations
- Verifying structural properties of a model

For example, a user could specify a condition that limits an iteration to only those classes that implement some interface.

Q Language Characteristics

Q is an expression-oriented language. Everything in Q is an expression. In fact, a “program” is nothing other than a complete expression. Expressions in Q very much resemble arithmetic expressions. Just like arithmetic expressions, expressions in Q are conceptually trees of subexpressions. Hence, the elements of the Q language naturally fall into two categories, the basic expressions at the leaves and the composite expressions at the internal nodes. Basic expressions are the fundamental building blocks of all expressions. They are like numbers in arithmetic expressions. Composite expressions are the means by which larger expressions are constructed from smaller expressions. Hence, they are like arithmetic operators.

Another parallel between arithmetic expressions and expressions in Q is in evaluation. Just as in arithmetic expressions, the evaluation of an expression proceeds recursively, with each composite expression evaluating its subexpressions.

Q is a side-effect free language. Evaluation involves the combination of values rather than a change of state. In particular, there are no variables whose value changes as a program runs. Another important consequence of this is that no program can alter the model currently loaded into ReporterPLUS.

Q is a typed language. Every well-formed expression in Q has a precise type. The Q language compiler infers this type and verifies that all expressions are compatible in type.

Model Representation

In general, expressions examine and gather information about a user's model. Therefore, all parts of a user's model must be easily accessible. Not only does this entail easy access to elements such as classes, but also to the relationships among them. In Q, we provide such a uniform view of user model data with a metamodel. All user data then appears as instances of the classes in the metamodel.

As an illustration, consider a model with a package named "Plant Structure" that contains classes named "Branch" and "Leaf." For an expression in Q, the package "Plant Structure" appears as an instance of the metaclass Package with the value of the attribute name of that instance equal to "Plant Structure." Similarly, the classes "Branch" and "Leaf" appears as instances of the metaclass Class with the name attributes equal to "Branch" and "Leaf," Finally, the containment relationship between the package and the classes appears as a link.

A user's model as a whole appears as an instance of the metaclass Model. All data in a user's model is ultimately reachable from this Model instance.

Basic Q Types

Every well-formed expression has a precise type. The two basic types in Q are the following:

- Primitive values
- User model data

The types string, integer, real, and boolean range over the corresponding primitive values; *regexp* ranges over regular expressions; and object ranges over instances of metaclasses.

Note

Object encompasses instances of all metaclasses. So instances of Package, Class and so on, all have the type object.

In addition to the basic types, the Q language supports the following three forms of constructed types: truples, collections, and functions.

Tuples

The first form is for tuples. For any types α, β, … , ω, the expression $\alpha \times \beta \times \ldots \times \omega$ denotes the type of a tuple with component types α, β, … , ω. For example, *string* $\times$ *integer* is the type of a pair with a string as the first component and an integer as the second component.

Collections

The second form is for collections (which are lists with possible duplicates). For any type α, α *collection* denotes the type of a collection of α. Some examples are *object collection* and *string collection*.

Functions

The third form is for functions. For any types α and β, $\alpha \rightarrow \beta$ denotes a function that takes a parameter of type α and returns a value of type β. For example, *string* $\rightarrow$ *integer* is the type of a function that takes a string and returns an integer.

Limitations

Although this scheme of type representation accommodates arbitrarily complex types, in practice there are limitations in the current implementation that restrict the complexity. Specifically, the current implementation supports the basic types, tuples of basic types, *object collection*, *string collection*, and functions that take and return any of these supported types, with the exception of tuples for return types. There is also special support for the types (*object* $\rightarrow$ *string*) $\times$ *object collection* $\rightarrow$ *string collection* (which is the type of the built-in function `map`), (*object* $\rightarrow$ *boolean*) $\times$ *object collection* $\rightarrow$ *object collection* (which is the type of the built-in function `filter`), and other such types for `traverse` and `sort`.

Complicated Type Examples

The following are some examples of more complicated types:

- *string* $\times$ *real* $\rightarrow$ *boolean* : A function that takes a tuple of two components—the first of type *string* and the second of type *real*—and returns a *boolean*.
- (object $\rightarrow$ string) $\times$ object collection $\rightarrow$ string collection : A function that takes a function (of type object $\rightarrow$ string) and an object collection and returns a string collection.

Basic Expressions

The *basic expressions* are the fundamental building blocks of all programs. There are several kinds of basic expressions. These are the expressions for constant literals and tuples; the expressions for the usual arithmetic, relational, and logical operations; the expressions for string concatenation, comparison, and pattern matching; the expressions for set operations; the expressions for object comparisons; and the expressions for variables.

Constant Literals

Constant literals represent values of the basic types string, integer, real, and boolean and values of the type regexp.

These expressions simply evaluate to their corresponding values.

Strings

Strings are enclosed in double quotes ("). Strings may contain character escapes, see the **Precedence and associativity of operators table**. Examples:

```
"abc"
"a\"bc\tdef\n"
```

Integer Examples

```
2
100
```

Real Examples

```
-2.0
3.4
```

Boolean

This constant may take one of these two values: `true` or `false`.

Regexp

For the syntax of *regular expressions*, see **Lexical Elements**. Regular expressions are enclosed in back quotes (`), as in this example:

```
`[a-zA-Z]+`
```

Tuples

Tuples represent ordered collections of values. These expressions take the form of a parenthesized list of *components* separated by commas, as in these examples:

```
(1, true)
("abc", true, -2.0)
```

Arithmetic Operations

Arithmetic operations on primitive values are basic expressions. The arithmetic operators +, -, *, and / have the usual meanings over integers and reals. The arguments to an arithmetic operator must be of the same type, either *integer* or *real*.

Relational Operations

Relational operations on primitive values are basic expressions. The relational operators =, <>, <, <=, >, and >= have the usual meanings over integers and reals, with <> representing the inequality operator. The arguments to a relational operator must agree in type.

Logical Operations

Logical operations on primitive values are basic expressions over Booleans. The logical operators are as follows:

```
not
and
or
implies (p implies q  ≡ not p or q)
```

String Operations

String concatenation, comparison, and pattern matching are basic expressions. These expressions take the form of an infixed operator.

String concatenation

The operator + concatenates two strings, as in this example:

```
"abc" + "def" ⇒ abcdef
```

String comparison

The relational operators (=,<>,<, <=, >, >=) implement lexicographical comparisons of strings, as in these examples:

```
"abc" = "abd" ⇒ false
"abc" <> "abd" ⇒ true
```

```
"abc" < "abd" ⇒ true
```

String pattern matching

The operators **~=** and **~<>** compare a string on the left-hand side against a regular expression on the right-hand side for a match or a mismatch, respectively. The right-hand side argument to **~=** and **~<>** should be a string literal rather than a regular expression literal, as in these examples:

```
"abc" ~= "a*" ⇒ true
"abc" ~<> "a*" ⇒ false
"10111010110101" ~= "[01]+" ⇒ true
"abcdefghijkl" ~= "[^01]+" ⇒ true
```

Object Comparisons

Object comparisons are basic expressions. These expressions take the form of an infixed operator over values of *object*. The operators = and <> compare two metaclass instances for identity. That is, *x* = *y* if, and only if, *x* and *y* refer to the same metaclass instance. Note that two metaclass instances can have the same values for corresponding attributes and yet still be distinct.

Variables

Variables are basic expressions. A variable name is an expression that evaluates to the value bound to that variable. For example, if `x` is bound to the expression `5 + 6`, then `x` ⇒ *11* and `x + 2` ⇒ *13*. Although variables are called "variables," they actually never vary. A variable simply stands for an expression. Hence, we say that a variable is "bound" to a value rather than that a variable "has" a value.

There are three predefined variables that pervade every program. These are `model`, `this`, and `current`, all of type *object*.

Predefined Variable: `model`

The variable `model` is bound to the unique instance of the metaclass *Model* that represents a user's model in ReporterPLUS. All data in the user's model is ultimately reachable from the *Model* instance bound to `model`.

Predefined Variable: `this`

The variable `this` is bound to the actual parameter of a *Q* program. Every program in ReporterPLUS is applied to some instance of a metaclass. For example, an advanced condition in the condition tab is applied successively to the metaclass instances in the association being iterated over. For each application, `this` is bound to the current instance in the iteration.

Predefined Variable: `current`

Unlike the other predefined variables, the variable `current` may become bound to different values at different textual locations in a program. Specifically, `current` is bound to whatever metaclass instance is "current" at any given textual location. At the outermost expression, for example, the "current" metaclass instance is the parameter to the whole program. Therefore, `current` is bound to the same object as `this` at the outermost expression. There are several expressions that temporarily rebind `current`. We make note of this fact in the descriptions of such expressions in the remainder of this document.

Composite Expressions

Composite expressions are the means by which larger expressions are constructed from smaller expressions. This section describes the composite expressions with the exception of functions, which we discuss in the following section.

Some composite expressions impose restrictions on the possible types of their subexpressions. Specifically, the composite expressions `let`, `if`, and `sort` require subexpressions of a small number of types. Therefore, to simplify the following discussions, use the symbols Π and Σ to stand for the permissible types: Π represents one of *string*, *integer*, *real*, *boolean*, *object*, *object collection*, or *string collection*, and Σ represents one of *string*, *integer*, *real*, or *boolean*. The symbol Π is used in the discussion of `let` and `if`. The symbol Σ is used in the discussion of `sort`.

Catalog of Composite Expressions

let : $\Pi_1 \rightarrow \Pi_2$

```
let x = expr1 in expr2
```

This expression evaluates to $expr_2$, with all (free) references to the variable *x* in $expr_2$ evaluating to $expr_1$. Each of $expr_1$ and $expr_2$ must have some type Π, but the types of these expressions need not be the same. That is, if $expr_1$ has type Π_1 and $expr_2$ has type Π_2, then it is not necessary for $\Pi_1 = \Pi_2$.

```
let x = 5 in x + 7 ⇒ 12

(let x = 5 in x + 7) + 3 ⇒ 15

let x = 5 in let y = 7 in x + y ⇒ 12

let x = 5 + 7 in x + 3 ⇒ 15

let x = 5 in let x = x * x in x ⇒ 25

let x = model in class x ⇒ Model

let x = all "Class" in size x <> 0 ⇒ true

let x = all "Class" in size x <> 0 implies
(there_exists y in x => $name of y = "Dishwasher") ⇒ true

let x = all "Class" in size x <> 0 implies
(there_exists y in x => $name of y = "FooBar") ⇒ false
```

if : ***boolean*** x Π x Π → Π

```
if test then expr1 else expr2
```

The value of this expression depends on the value of test. If test evaluates to true, then the whole expression evaluates to $expr_1$. Otherwise, the whole expression evaluates to $expr_2$. The

type of test must be *boolean*, and the type of both $expr_1$ and $expr_2$ must be some Π. Note that the types of $expr_1$ and $expr_2$ must be the same. That is, if $expr_1$ has type Π_1 and $expr_2$ has type Π_2, then it must be the case that $\Pi_1 = \Pi_2$.

```
if true then 1 else 0 => 1

if class x = "Operation"

then "Found Operation"

else "Found " + class x => Found Model

(if false then "abc" else "123") + "def" => 123def
```

for_all : *object collection* x (*object* → *boolean*) → *boolean*

```
for_all x in ocoll => predicate ≡ not (there_exists x in ocoll => not
predicate)
```

there_exists : *object collection* x (*object* → *boolean*) → *boolean*

```
there_exists x in ocoll => predicate ≡ not (for_all x in ocoll => not
predicate)
```

These expressions evaluate to the assertions $\forall x \in ocoll$ (*predicate*(*x*)) and $\exists x \in ocoll$ (*predicate*(*x*)). That is, `for_all` returns true if, and only if, either *ocoll* is empty or every object in *ocoll* satisfies the predicate *predicate*. And `there_exists` returns true if, and only if, there is at least one object in *ocoll* which satisfies the predicate *predicate*.

The actual evaluation of these expressions occurs in the following way. For both `for_all` and `there_exists`, the variable *x* is bound in turn to each object in *ocoll*, and *predicate* is evaluated in the context of such a binding of *x*. If *predicate* evaluates to false, then `for_all` evaluates to false. If *predicate* evaluates to true, then `there_exists` evaluates to true. If neither condition occurs before all objects in *ocoll* have been tested, then `for_all` evaluates to true, while `there_exists` evaluates to false. These expressions also bind the pervasive variable `current` to *x* at each binding of *x*.

The type of the expression *ocoll* must be *object collection*, and the type of *predicate* must be *boolean* (which we interpret as *object* → *boolean*, with *x* being the implicit parameter to *predicate*).

In the following examples, suppose that there are three classes, named Tree, Branch, and Leaf.

```
for_all x in all "Class" => class x = "Class" ≡
for_all x in all "Class" => class current = "Class" ⇒ true
there_exists x in all "Class" => $name = "Leaf" ⇒ true
for_all x in all "Class" => $name = "Leaf" ⇒ false
not (for_all x in all "Class" => $name <> "Leaf") ⇒ true
for_all x in anEmptyCollection => false ⇒ true
for_all x in anyCollection => current = x ≡ true ⇒ true
for_all class in all "Class" =>
   there_exists op in class->[operation] =>
      $name of op = "grow" ⇒ false
```

Functions

Functions encapsulate expressions that may be useful in many different situations. A function executes when it is applied to an actual parameter. The general syntax of a function application is a function name followed by an expression. Some examples are `all("Class")` and `class current`. Functions part of the *Q* language, and there is no way to define new functions.

All functions take a single parameter. Functions that appear to take more than one parameter really take a single tuple containing the parameters. Because all functions take a single parameter, it is not necessary to enclose the parameter in parentheses. Instead, the smallest complete expression following the name is taken to be the parameter. This rule of taking the “smallest complete” expression means that parentheses are sometimes necessary to indicate a larger expression for the parameter.

Two other aspects of functions are worth mentioning. First, function application is at the highest precedence level. Hence, function applications are usually evaluated earlier than operators, such as `+ and` in composite expressions. Second, function application associates to the right.

To illustrate these points, consider two functions named `inc` and `dec` (these functions are not part of the *Q* language - they are used merely as examples). The function `inc` takes an integer and returns one plus that integer. The function `neg` takes an integer and returns the negation of that integer. Use the symbol ≡ in the following examples to show an equivalent but fully-parenthesized counterpart to some expression.

```
inc 5 ≡ inc (5) ⇒ 6
neg 5 + 3 ≡ (neg (5)) + 3 ⇒ -2
neg (5 + 3) ⇒ -8
neg 5 + inc 3 ≡ (neg (5)) + (inc (3)) ⇒ -1
neg (5 + inc 3) ⇒ -9
neg inc 3 ≡ neg (inc (3)) ⇒ -4
neg inc 3 + 1 ≡ (neg (inc 3)) + 1 ⇒ -3
neg neg neg neg 3 ≡ neg (neg (neg (neg (3)))) ⇒ 3
```

Catalog of Built-In Functions

all : *string* → *object collection*

```
all name
```

This returns all instances of the metaclass named *name*. For example, `all "Class"` returns all instances of the metaclass *Class*.

alternation : *string* x *string* → *string*

|=> : *string* x *string* → *string*

```
alternation (str, alt) ≡ str |=> alt ≡ if str <> "" then str else alt
```

If *str* is not the empty string, then this returns *str*; otherwise, this returns *alt*.

```
alternation ("abc", "def") ⇒ abc
alternation ("", "def") ⇒ def
attribute : string x object → string
$ : string x object → string
attribute (name, obj)
```

This returns the value of the attribute named *name* of *obj*. If the metaclass of *obj* does not have an attribute named *name*, then this signals an execution error. Note that the first parameter to $ is not enclosed in double quotes whereas this is the case for the first parameter to `attribute`.

This always returns the value of an attribute as a string. Attributes in the metamodel, however, can be of other types—*integer*, *real*, *boolean*, and various enumerations. But `attribute` always returns a string because this is the safest assumption to make, given that the type *object* encompasses instances of all metaclasses. In essence, the type *object* is untyped with respect to the metamodel. Although this always returns the value as a string, no information is lost. The returned string can always be converted to the appropriate type, using one of the conversion operators. The only exception to this occurs with attributes that are enumerations. Enumerations are treated like strings, with the values being the textual names of the enumerators that make up an enumeration.

```
attribute ("name", aClass) ≡ $name of aClass ⇒ Tree
attribute ("type", anOperation) ⇒ std::string
$name ≡ $name of current ⇒ Leaf
boolean $isAbstract of aClass ⇒ true
```

attribute : *string* x *object collection* → *string*

$: *string* x *object collection* → *string*

```
attribute (name, ocoll) ≡ $name' of ocoll ≡ comma map {$name} over ocoll
```

This returns a string formed from the values of the attribute named *name* of each object in *ocoll*, with the values separated by commas. If *ocoll* contains no objects, then this returns the empty string. If *ocoll* contains just one object, say *obj*, then `attribute(name, ocoll)` is equivalent to `attribute (name, obj)`. If the metaclass of an object in *ocoll* does not have an attribute named *name*, then this signals an execution error.

```
attribute ("name", all "Class") ⇒ Tree, Branch, Leaf
attribute ("type", all "Operation") ⇒ grow
$name of all "Class" ⇒ Tree, Branch, Leaf
```

class : *object* → *string*

```
class obj
```

This returns the name of the metaclass of *obj*.

```
class aClass ⇒ Class
class model ⇒ Model
```

comma : *string collection* → *string*

```
comma scoll
```

This returns the concatenation of all the strings in *scoll*, with the strings separated by commas.

concat : *string collection* → *string*

```
concat scoll
```

This returns the concatenation of all the strings in *scoll*.

find : *regexp* x *string* → *string*

```
find (re, s)
```

This returns the first substring of *s* that matches the regular expression *re*. If there is no substring of *s* that matches *re*, then this returns the empty string (λ). When matching substrings, this always matches as many characters as possible.

```
find (`[01]+`, "---010111--111") ⇒ 010111
find (`[a-z]+`, "--010111") ⇒ λ
```

first : *object collection* → *object*

```
first ocoll
```

This returns the first object in *ocoll*. If *ocoll* is empty, then this signals an execution error.

```
$name of first all "Class" ⇒ Node
```

match : *regexp* x *string* → *boolean*

```
match (re, s) ≡ s ~= re'
```

This returns true if, and only if, there is a substring of *s* that matches the regular expression *re*.

```
match (`[01]+`, "abc010111def") ⇒ true
match (`^[01]+$`, "abc010111def") ⇒ false
match (`[01]*`, "abc") ⇒ true
```

Note how this last example is true even though it would seem otherwise. The match succeeds because the closure * admits the empty string and because the empty string is a valid substring of "abc."

object : *object collection* → *object*

```
object ocoll ≡ only ocoll
```

See the entry for function **only : *object collection* → *object*** (below).

only : *object collection* → *object*

```
only ocoll
```

This returns the first and only object in *ocoll*. If *ocoll* is empty or has more than one object, then this signals an execution error.

```
$name of only all "Class" ⇒ Node
```

replace : *regexp* x *string* x *string* → *string*

```
replace (re, s, r)
```

This returns a string equal to *s* with the first substring matching *re* replaced with *r*. If there is no substring of *s* that matches *re*, then this simply returns *s*. When matching substrings, this always matches as many characters as possible.

```
replace (`[01]+`, "---010111--111", "A") ⇒ ---A--111
replace (`[a-z]+`, "010111", "A") ⇒ 010111
```

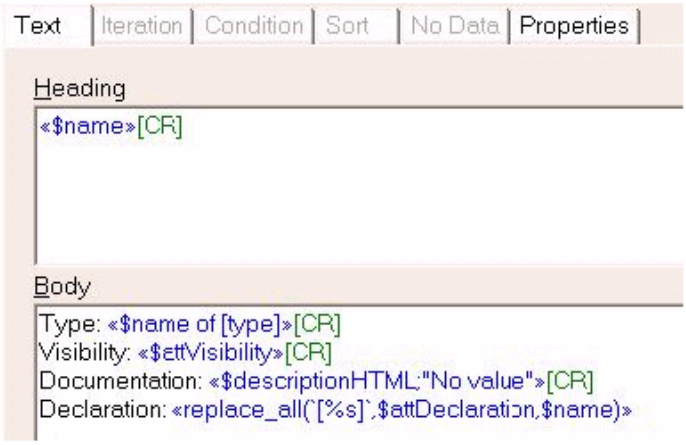
replace_all : *regexp* x *string* x *string* → *string*

```
replace_all (re, s, r)
```

This returns a string equal to *s* with all substrings matching *re* replaced with *r*. If there is no substring of *s* that matches *re*, then this simply returns *s*. When matching substrings, this always matches as many characters as possible.

```
replace_all (`[01]+`, "---010111--111", "A") ⇒ ---A--A
replace_all (`[a-z]+`, "010111", "A") ⇒ 010111
```

The following example illustrates the use of "`%s`" with the `replace_all` keyword.



reverse : *object collection* → *object collection*

```
reverse ocoll
```

This returns the collection *ocoll* with the elements in reverse order.

```
$name of (all "Class") ⇒ "Class_0, Class_1, Class_2"
$name of (reverse all "Class") ⇒ "Class_2, Class_1, Class_0"
```

size : *object collection* → *string*

```
size ocoll
```

This returns the number of objects in *ocoll*.

```
size all "Class" ⇒ 27
```

tolower : *string* → *string*

```
tolower s
```

This returns the string *s* with all upper case letters changed to lower case.

```
tolower "aBcDeF" ⇒ abcdef
```

toupper : *string* → *string*

```
toupper s
```

This returns the string *s* with all lower case letters changed to upper case.

```
toupper "aBcDeF" ⇒ ABCDEF
```

trim : *string* → *string*

```
trim s ≡ replace (`^[ \t\f\r\n]+`, replace (`[ \t\f\r\n]+$`, s, ""),
"")
```

This returns the string *s* with all leading and trailing spaces, tabs, form feeds, carriage returns, and new lines removed.

```
trim " \tabc\n" ⇒ abc
```

uid : *object* → *string*

```
uid obj
```

This returns a unique ID for *obj*. No two objects in a given model has the same ID. Two objects in two different models, however, may have the same ID, and a given object may have different IDs during different sessions. There is no structure to the ID strings, and the actual format could change in the future.

```
uid anObject ⇒ 04240120
```

Note: This uid is NOT the Rhapsody object uid and is valid within the context of ReporterPLUS session.

Conversion Operators

In addition to the above built-in functions, *Q* includes several conversion operators. The syntax of these operators is the same as that of function application. There are four conversion operators—named `string`, `integer`, `real`, and `boolean`–that support the conversion of values from any basic type to any other basic type, where the basic types are *string, integer, real*, and *boolean*. Most of these conversions are obvious. The following is a list of the non-obvious conversions (*b*, *s*, *i*, and *r* are *boolean, string, integer*, and *real* expressions, respectively):

```
integer b ≡ if b then 1 else 0
real b ≡ if b then 1.0 else 0.0
boolean s ≡ if s = "true" then true else false
boolean i ≡ if i = 0 then false else true
boolean r ≡ if r = 0.0 then false else true
```

Here are some examples of applying the conversion operators:

```
integer "77" ⇒ 77
string true ⇒ true
integer true ⇒ 1
boolean "true" ⇒ true
boolean "foo" ⇒ false
boolean 25 ⇒ true
integer string 5 ≡ integer (string (5)) ⇒ 5
integer string 5 + 7 ≡ (integer (string (5))) + 7 ⇒ 12
integer boolean 5 + 7 ⇒ 8
```

Paths

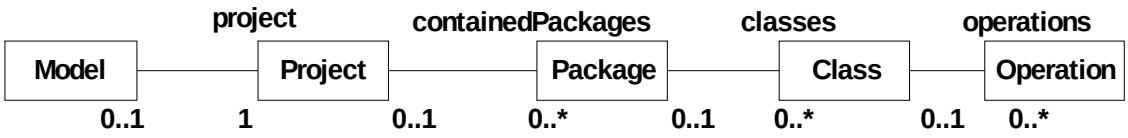
Paths are expressions that describe a traversal through a user's model. Conceptually, a path is a function, from *object* to *object collection*, that returns some of the objects reachable along a specified set of associations. The term "anchor" denotes the initial object (the parameter of a path). In general, a path in *Q* can be any arbitrary directed graph. This is a linear representation of a directed graph for the syntax of paths.

Basic Paths

The following is an example of a simple path:

```
model->[project]->[containedPackages]->[classes]->[operation]
```

This path describes a simple linear traversal of three associations. This says that, starting from the object `model`, we should follow the association "project" and then "containedPackages" and then "classes" and then "operation." The end result of applying this path to the anchor `model` is a collection of all instances of the metaclass *Operation* that belong to some class in the package at the end of the association "project." This illustration below shows the portion of the metamodel that this path traverses.

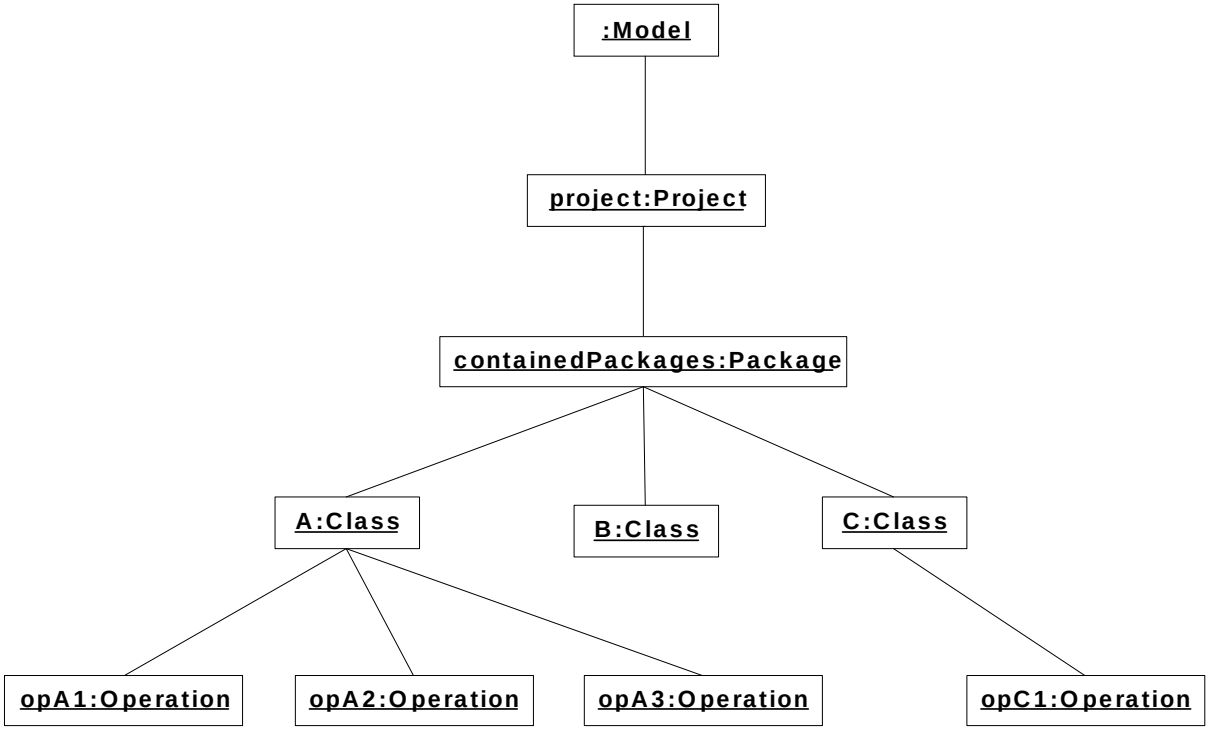


Note

The path corresponds directly to the metamodel.

For example, *Project* has the role "project" in an association with *Model*, and *Class* has the role "classes" in an association with *Package*.

The following illustration shows an object structure to which you can apply this path. In this case, the path would evaluate to a collection of the objects opA1, opA2, opA3, and opC1



So, in the syntax of paths, a node (such as `[project]`) represents an association to traverse, and an edge (`->`) between two nodes specifies the order of traversal. In the syntax, the symbol `->` also separates the anchor from the actual path, and the anchor is optional. When an anchor is omitted, the implied anchor is the object currently bound to the variable `current`. So the path `[project]` is a shorthand for `current->[project]`.

Paths with Cycles

The following example shows a path with a cycle:

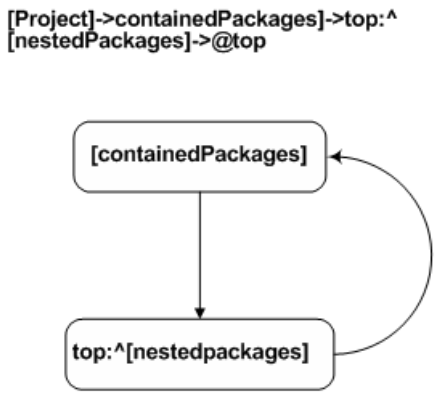
```
model->[project]->[containedPackages]->top:^[nestedPackages]->@top
```

This example introduces three new syntactical elements. These are node labels (`top:`), add-to-result annotations (^), and reference nodes (`@top`). Labels and reference nodes provide a way of encoding the two-dimensional structure of cycles, and other nonlinear structures, of complex paths.

A graphical representation of the above path is shown in **Sample Path with a Cycle**. In this example, there is a cycle from the node labelled “top” to itself. This cycle corresponds to the reflexive association `nestedPackages` of `Package`, which is an association to all of the immediate nested packages of a `Package`.

The purpose of this cycle is to traverse all nested packages of a package, including those packages in the `nestedPackages` of a package. The end result of applying this path to `model` is a collection of all instances of `Package` that are nested packages, directly and indirectly, of the package at the end of the association `containedPackages` of `project` in `model`. But in order for this path to work in this way, we have to use the add-to-result annotation ^. This annotation on the node `[nestedPackages]` means that all objects found at the end of the association `nestedPackages` should be added to the result set. All nodes that do not have any outgoing edges have the add-to-result annotation implicitly. So in our first example, the node `[operation]` has the add-to-result annotation implicitly. In the case of the second example, however, all nodes have outgoing edges because of the cycle. Therefore, no node has the add-to-result annotation implicitly.

Sample Path with a Cycle



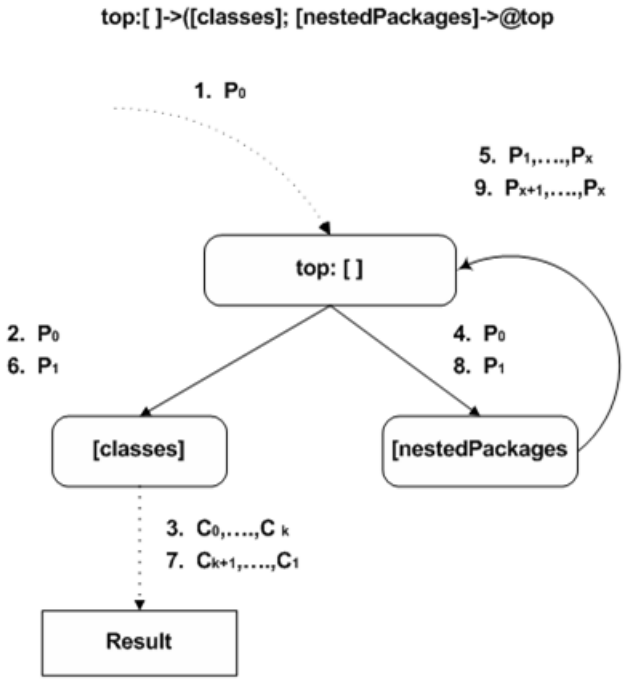
Path Nodes with Multiple Outgoing Edges

This next example shows a node with more than one outgoing edge:

```
model->[project]->[containedPackages]->top:[]->([classes]; [nestedPackages]->@top)
```

In this example, the node labelled “top” has two successors, the nodes [classes] and [nestedPackages]. In this case, the node “top” is also an empty association ([]). So “top” only serves to structure the nodes in the path. The node [classes] doesn’t have any outgoing edges (the arrow to Result in **Node with More than One Outgoing Edge** is not related to the structure of the path), so it has the add-to-result annotation implicitly.

Node with More than One Outgoing Edge



Hence, the result of applying this path to model is a collection of all instances of Class that belong either to the package at the end of the association `containedPackage` of project in model or to a nested package of that package.

Paths Nodes with Conditions

This final example shows two nodes with conditions:

```
aPackage->[nestedPackages]{$name="PkgA"}->[classes]{$name~="*Impl"}
```

Any node in a path may be accompanied by a condition. The condition, which appears in curly braces after the association name, specifies a predicate that objects in an association must satisfy before such objects are allowed to pass through a node. In this example, the condition `{$name="PkgA"}` allows only the objects named “PkgA” to reach the node `[classes]`, and the condition `{$name~="*Impl"}` allows only the objects that have a name matching the regular expression “*Impl” to appear in the result set. The effect of `[nestedPackages]{$name="PkgA"}` is exactly that of `filter {$name="PkgA"} over [nestedPackages]`. See the documentation of `filter` for the details of how the condition expression is evaluated.

Execution Model of Paths

Paths are functions from *object* to *object collection.* They yield the objects reachable along a set of associations. Paths encode this computation in their graph structure—in essence, the structure *is* the computation. To be more precise, paths are dataflow graphs, and they resemble dataflow in their execution. So in the dataflow view, paths now consist of data links and operations. Data links correspond to edges, and operations correspond to nodes. Data links are conduits along which objects flow, and operations are basic computations that accept objects from incoming data links and produce objects on outgoing data links.

Consider, for instance, the following path:

```
model->[project]->[containedPackages]->[classes]->[operation]
```

We can analyze this in terms of dataflow in the following way. This path consists of the operations `[project]`, `[containedPackages]`, `[classes]`, and `[operation]` linked together linearly. When the object `model` flows into `[project]`, all the objects in the association “project” of `model` flow out. Then all these objects flow into `[containedPackages]`, which causes the objects in the association `containedPackages` to flow out. Then all these objects flow into `[classes]`, and the objects in the association “classes” flow out. Then all these objects flow into `[operation]`, and the objects in the association “operation” flow out. Finally, these objects flow into the (implicit) result node for the complete path. All objects produced by leaf nodes implicitly flow into the result node, which collects the objects produced by the complete path.

The evaluation of a path ends when the flow of objects ceases. Hence, some care is needed to ensure that a path terminates. Stated precisely, a path will *not* terminate if, and only if, an object revisits an operation, either because the object itself completes a cycle or because a cascade of objects completes a cycle.

There are some boundary cases in the behavior of operations. First, the operation [ ] is the identity operation. All objects simply flow through. Second, an operation [α] doesn’t produce any output if the input object does not have any objects in the association α. Third, an operation [α] also does not produce any output if the metaclass of the input object doesn't have the association α. So, for example, the operation `[nestedPackages]` would not produce any output if the input object is an instance of `Class` rather than `Package`.

Note

The path should not include cycles.

Precedence and Associativity of Operators

The table below summarizes the precedence and associativity of all operators and special functions. The first entry has the highest precedence, and precedence decreases from top to bottom. In this table, "ID *expr*" represents function application, and "*type expr*" represents the application of a conversion operator. Note that the alternation operator (| =>) and the relational operators (=, <>, and so on) do not associate.

Precedence and associativity of operators table

Operator	Associativity
$, `map`, `filter`, `traverse`, `sort`, ID *expr*, *type expr*	R
(unary) `-`, `not`	R
`*`, `/`	L
`+`, `-`	L
=, <>, ~=, ~<>, <, <=, >, >=	–
And	L
Or	L
Implies	R
let, if, for_all, there_exists	R
|=>	–

Lexical Elements

This section describes the lexical elements (regular expressions) in *Q*. Comments, spaces, tabs, carriage returns, and new lines are ignored between tokens. A comment is a sequence of characters delimited by `(*` and `*)`. A comment may span more than one line.

Note

Comments may not be nested.

Punctuation

The following are the punctuation symbols:

```
= <> ~= ~<> < <= > >= + - * / -> ( ) { } [ ] " => |=> , $
```

Identifiers (ID)

An identifier is a letter or an underscore followed by any number of letters, digits, or underscores. The case of an identifier is significant.

Keywords

The following are the keywords. Keywords may not be used as names of variables or functions.

and	in	real
boolean	integer	sort
current	let	string
else	map	then
false	model	there_exists
filter	not	this
for_all	of	traverse
if	or	true
implies	over	

Integer Literals (INTEGER_LITERAL)

An integer literal is an optional minus sign followed by one or more decimal digits.

Real Literals (REAL_LITERAL)

A real literal is an optional minus sign followed by one or more decimal digits, a decimal point, and one or more decimal digits.

Boolean Literals (BOOLEAN_LITERAL)

The boolean literals are `true` and `false`.

Association Literals (ASSOC_LITERAL)

Association literals are essentially string literals with square brackets (`[]`) in place of double quotes. The sequence `\]` escapes the closing delimiter.

String Literals (STRING_LITERAL)

A string literal is a sequence of characters enclosed in double quotes. A string may not extend across lines (that is, a string may not contain an unescaped new line). The supported escape sequences are listed in the following table.

Escape Sequence	Replacement Text
\"	"
\\	\
\t	tab
\r	carriage return
\n	new line

Regular Expression Literals (REGEXP_LITERAL)

A *regular expression literal* is a sequence of characters enclosed in back quotes. Several characters have special meanings within regular expression literals. These characters have their special meaning unless preceded by a back slash. The special characters are `.`, `\`, `[`, `]`, `?`, `*`, `+`, `^`, and `$`. A regular expression literal may contain character escape sequences. The allowed escape sequences are `\t`, `\r`, `\n`, and `\xdd`, which stand for a tab, a carriage return, a new line, and the character with the ASCII code equal to the hexadecimal number dd, respectively.

Regular expressions have a recursive structure. They are formed from smaller subexpressions. The building blocks of all regular expressions are the expressions to match a single character. These fundamental expressions have the following three forms:

- Period (`.`) is a regular expression that matches any single character. For example, `` `.` `` matches *a*, *5*, #, \n, ", and so on.
- Any character other than a special character, or a special character preceded by a back slash, is a regular expression that matches that character. For example, `` `a` ``, `` `5` ``, and `` `\*` `` match *a*, *5*, and *, respectively.
- A set of characters enclosed in square brackets is a regular expression that matches any one character in the set. For example, `` `[abc]` `` matches any one of *a*, *b*, or *c*. If the first character in the set is a caret (^), then the regular expression matches the complement of the given set of characters. So `` `[^abc]` `` matches any character other than *a*, *b*, and *c*. As a matter of convenience, a range of characters can be specified with a dash. The range includes all characters between the lower and upper bounds, inclusively. For example, `` `[a-zA-Z0-9]` `` matches any letter or digit.

From these building blocks, larger regular expressions may be formed in the following way:

- If re_1 and re_2 are regular expressions, then $re_1\ re_2$ (concatenation) is a regular expression that matches all strings of the form s_1s_2, where s_1 is matchable by re_1 and s_2 is matchable by re_2. For example, `` `[ab][01]` `` matches *a0*, *a1*, *b0*, and *b1*.
- If *re* is a regular expression, then *re?* is a regular expression that matches zero or one occurrence of *re*. For example, `` `ab?` `` matches *a* or *ab*, and `` `a[01]?` `` matches *a*, *a0*, or *a1*.
- If *re* is a regular expression, then *re** is a regular expression that matches zero or more occurrences of *re*. For example, `` `ab*` `` matches *a*, *ab*, *abb*, *abbb*, …, and `` `a[01]*` `` matches *a*, *a0*, *a1*, *a00*, *a01*, *a11*, ….
- If *re* is a regular expression, then *re+* is a regular expression that matches one or more occurrences of *re*. For example, `` `ab+` `` matches *ab*, *abb*, *abbb*, …, and `` `a[01]+` `` matches *a0*, *a1*, *a00*, *a01*, *a11*, ….

Note

The postfix operators `?`, `*`, and `+` bind more tightly than concatenation. Therefore, `ab*` means `a(b*)` and not `(ab)*`.

Finally, a complete regular expression may be anchored to the beginning or end of a string with `^` and `$`, respectively. If *re* is a regular expression, then `^`*re* is a regular expression that matches all strings matchable by *re* but only if they occur at the beginning of a string. Similarly, *re*`$` is a regular expression that matches all strings matchable by *re* but only if they occur at the end of a string. For example, `` `^[01]+` `` matches *0* and *0110* but not *a0* or *a0110*; and `` `[01]+$` `` matches *0* and *0110* but not *0a* or *0110a*.

Q Expression Tester

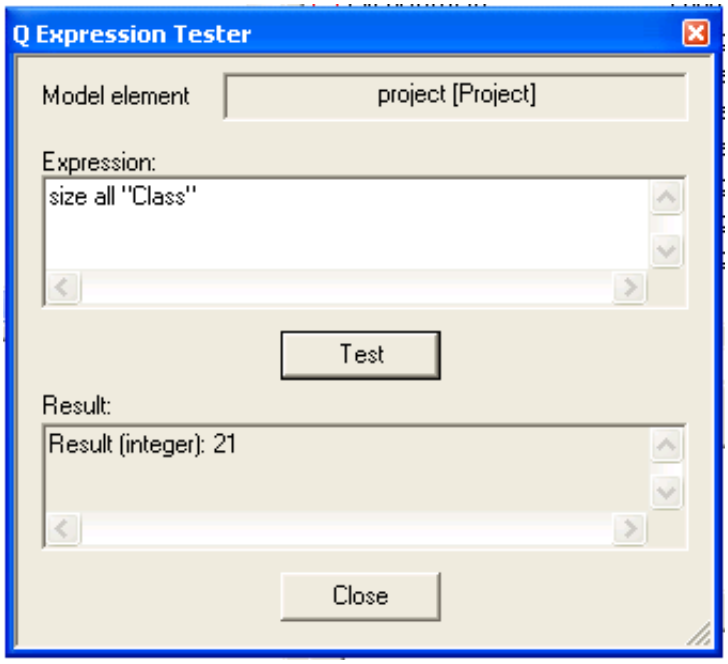
The Q Expression Tester allows you to perform the following operations:

- Pre-test **Q Language** expressions for validity before incorporating them into a report template
- Gather metrics on a project or a portion of a project
- Check the values associated with selected elements

To launch the Q Expression Tester, follow these steps:

1. In the Model View, highlight an element in the browser.
2. Right-click to display the **View Q Tester** option and select it.
3. Type the Q language expression to perform the type of test you need.
4. Click **Test** to run the test. The results display below the Expression area or an error message displays indicating that the Q language was no correct in either syntax or for the selected element.

The following example shows a test to determine how many classes are in the selected project. For this project (Model element), the total number of classes is 21.



Q Language Grammar

This section lists the complete Q language grammar. See **Lexical Elements** for the structure of the tokens.

Examine the following table for a summary of the typographical conventions used in the presentation of the grammar.

Style	Meaning
[ ]	Optional element
( )+	One or more occurrences
( )*	Zero or more occurrences
ID, STRING_LITERAL	Tokens
let, for_all, =>, ~<>	Reserved words or punctuation

program

```
: expr
```

expr

```
: expr′
| expr′ |=> expr′
```

expr′

```
: implies-expr
| let-expr
| if-expr
| universal-expr
| existential-expr
```

let-expr

```
: let ID = expr in expr′
```

if-expr

```
: if expr then expr else expr′
```

universal-expr

```
: for_all ID in expr => expr′
```

existential-expr

```
:  there_exists ID in expr => expr'
```

implies-expr

```
:  or-expr
|  or-expr implies implies-expr
```

or-expr

```
:  and-expr
|  or-expr or and-expr
```

and-expr

```
:  relational-expr
|  and-expr and relational-expr
```

relational-expr

```
:  additive-expr
|  additive-expr = additive-expr
|  additive-expr <> additive-expr
|  additive-expr ~= additive-expr
|  additive-expr ~<> additive-expr
|  additive-expr < additive-expr
|  additive-expr <= additive-expr
|  additive-expr > additive-expr
|  additive-expr >= additive-expr
```

additive-expr

```
:  multiplicative-expr
|  additive-expr + multiplicative-expr
|  additive-expr - multiplicative-expr
```

multiplicative-expr

```
:  unary-expr
|  multiplicative-expr * unary-expr
|  multiplicative-expr / unary-expr
```

unary-expr

```
:  primary-expr
```

```
  | - unary-expr
  | not unary-expr
```

primary-expr

```
  :  constant
  |  $ ID
  |  $ ID of primary-expr
  |  map { expr } over primary-expr
  |  filter { expr } over primary-expr
  |  traverse { expr } over primary-expr
  |  sort-expr
  |  ID primary-expr
  |  conversion-expr
  |  parenthesized-expr
  |  path
```

constant

```
  :  REGEXP_LITERAL
  |  STRING_LITERAL
  |  INTEGER_LITERAL
  |  REAL_LITERAL
  |  BOOLEAN_LITERAL
```

sort-expr

```
  :  sort-component ( & sort-component )* over primary-expr
```

sort-component

```
     :  sort-direction { expr }
```

sort-direction

```
  :  sort
  |  sortd
```

conversion-expr

```
  :  conversion-operator primary-expr
```

conversion-operator

```
  :  string
  |  integer
```

| **real**
| **boolean**

parenthesized-expr

: **( )**
| **(** expr **)**
| **(** expr (, expr)+ **)**

path

: anchor
| anchor **->** association-chain
| association-chain

anchor

: **this**
| **current**
| **model**
| ID

association-chain

: simple-association (**->** simple-association)* [association-chain-tail]

simple-association

: [ID **:**] [^] ASSOC_LITERAL [**{** expr **}**]

association-chain-tail

: **->** reference-association
| **->** composite-association

reference-association

: **@** ID

composite-association

: **(** component-association (**;** component-association)+ **)**

component-association

: reference-association
| *association-chain*

Footers and Other Formatting

This section describes the process to continue building the template started in **Creating Your Own ReporterPLUS Template**. It demonstrates adding a title page and a footer with a page number, changing a node label, and associating a *Word* template with the template.

After completing this section's tasks, you should be able to:

- Add a text node.
- Add headers, footers, and page numbers.
- Create a title page.
- Change a node label.
- Associate a Word template with a ReporterPLUS template.

Adding a Text Node

In **Creating Your Own ReporterPLUS Template**, you added iteration nodes and iteration subnodes to your template by dragging elements from the model view. In this section, you add a text node, which is used to hold a document title and a footer with a page number.

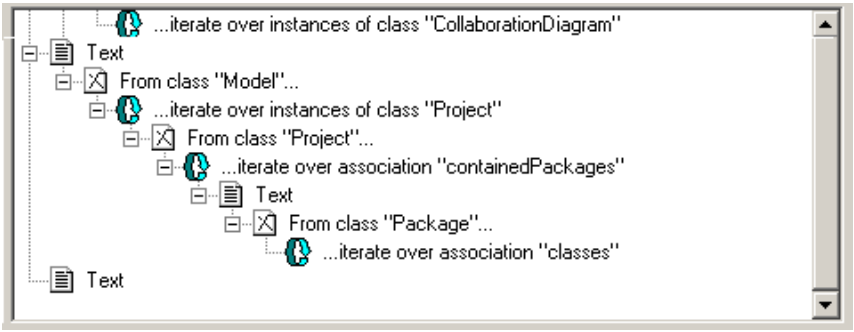
If your ReporterPLUS template is not open, open it now (as described in **Opening an Existing Template**).

Do the following:

1. In the template view, click in the blank area below the existing template nodes.
2. Select **Edit > New Template Node**.

Alternatively, you can add a text node by clicking the **New Template Node** icon  or by right-clicking in the template view and selecting **Add New Item** from the context menu.

A node labeled “Text” is displayed in the template view below the other nodes. The template view looks like the following figure.



Adding a Footer and Page Number

Place the commands for headers and footers for a node at the very top of your template if you want them to apply to the whole document. If desired, you can change the header and footer for different parts of the document (see Using Multiple Headers and Footers). First, you move the text node you just added, add a footer with a page number, and follow these steps.

1. In the template view, click on the text node you just added.
2. Click the **Move Node Up** toolbar button (the green up-arrow) until the text node is at the top of the template view.
3. With the text node still selected, right-click in the body of the Text tab, before the `[CR]` command, to display the context menu.
4. Select **Commands > Begin Footer**. You see the `[BEGIN FOOTER]` formatting command on the Text tab.
5. Without moving the cursor, right-click again.
6. Select **Commands > Add Page Number**.
7. Right-click again, then select **Commands > End Footer**.
8. Click **Apply**.

The procedure for adding a header is similar to that for adding a footer: just choose the **Begin Header** and **End Header** commands from the context menu. It is best to place the header and footer commands in the body of the Text tab, rather than the heading section.

In Word, the header and footer text is assigned the header or footer style. The actual appearance of the header or footer depends on how the header and footer styles are defined in the Word template used to create the document.

Note

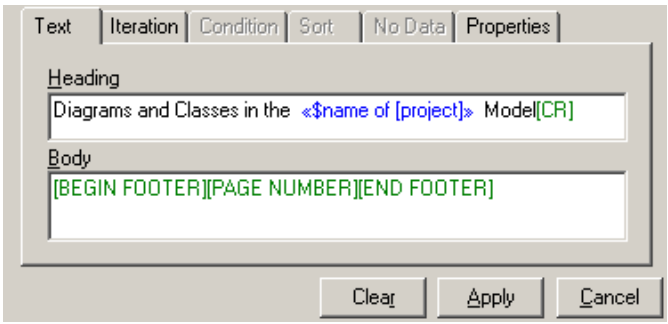
You can add additional formatting commands or text to your footer. For example, you can center the page number, bold it, or add the date and time. All of these formatting commands are accessed from the context menu. See the online help topic “What formatting can I specify?” for more information. In addition, you can add attributes, such as the model file name, to the header or footer by selecting **Add Attribute** from the context menu.

Adding a Title Page

In this section, use the same text node to add a title page to the generated document by following these steps:

1. Click on the text node.
2. In the heading of the Text tab, select the text that reads “New Text Node” and the [CR] command, and delete them (by pressing the `Delete` key).
3. With your cursor still in the heading of the Text tab, type “Diagrams and Classes in the Model.” Leave two spaces between “the” and “Model.”
4. Right-click on the Heading window, click on the (+) next to `project`, select **name**, and insert it between “the” and “Model.” Use the `«$name of project»` attribute instead of typing in the actual name of the project so you can use this template with other models.
5. Right-click at the end of the title to display the context menu.
6. Select **Commands > Add Page Break**. By placing a page break after the title, you ensure that the title appears on a page by itself.

7. Click **Apply**. The Text tab should now look like the following figure.



Changing the Node Label

The label displayed next to the template node icon in the template view is created by ReporterPLUS when you create the node. If desired, you can change the default label to a user-defined label. The node label does not affect the generated document. Instead, it simply makes the template easier to use by identifying the template nodes.

Do the following:

1. Click on the text node.
2. In the heading of the Text tab, select the text and attribute you added in the previous section. Do not select the page break command.
3. Select **Edit > Copy** (or click the **Copy** button on the toolbar).
4. Select the Properties tab.
5. Click to place the cursor in the **User-defined label** field.
6. Select **Edit > Paste**. Alternatively, click the **Paste** button on the toolbar, or right-click and select **Paste** from the context menu.
7. Click **Apply**. You should see the new label text display next to the text node icon in the template view.

Using Multiple Headers and Footers

You can use multiple headers and footers when you generate Word, PowerPoint, or HTML documents. For example, you might want to change the footer for each section of a document—one footer for a section that covers the `Component`, another footer for a section that covers the `Package`, and so on.

You can define headers and footers anywhere in a ReporterPLUS template. When ReporterPLUS encounters a new header or footer code in the template, it changes the header or footer in the generated document. Page numbering remains consecutive throughout the document.

Note

If you are planning on generating a Word document, you must place a page break before the second (or any additional) header or footer. (Page breaks are optional when the template has only one header or footer.) Word creates a new section—and start using the new header or footer—at the page break immediately preceding the new header or footer command. For this reason, you should place the page break as close as is practical to the header or footer command. You might want to place the page break at the very beginning of the heading section of the Text tab for the node where the header or footer changes.

Associating a Word Template with a ReporterPLUS Template

Previously, you learned how to change the default Word template used by ReporterPLUS when generating documents. In this section, you learn how to associate a particular Word template with a ReporterPLUS template.

Do the following:

1. Select **File** > **Template Properties**.

   In addition to the **Description** field, the Template Properties dialog box has fields for entering the name of a Word or PowerPoint template or an HTML style sheet, and for indicating how you want to handle HTML tags.

2. Click the browse button next to the **Word .dot File** field, navigate to the `Templates` directory, and select `ReporterSimpleClassReport.dot`.

3. Click **OK**.

Every time you generate a Word document from this template, ReporterPLUS uses the `ReporterSimpleClassReport.dot` template. If there is no template specified in the Template Properties dialog box, ReporterPLUS uses the Word template specified in the Default Document Properties dialog box.

Generating and Viewing Your Document

To generate and view the resultant document, follow these steps:

1. Select **File > Save Template** to save your ReporterPLUS template. Although you do not have to save before generating, it is a good habit to develop.

   Alternatively, you can save a template by clicking the **Save Template** icon  .

2. If the `Dishwasher.rpy` model is not open, open it now (see **Opening a Model**).

3. Generate a Word document (see **Generating a Word Document**).

4. View the generated document. Note that the document has a title page and footer, and that it no longer uses the Word template with numbered headings.

Sorting, Conditions, and Missing Data

This section describes how to continue building the ReporterPLUS template that was described in the **Creating Your Own ReporterPLUS Template**. You specify a sort order for the diagrams and classes, add a condition to an iteration, and add some instructions that tell ReporterPLUS what to do when there is no model data for an attribute or iteration.

After completing this section's tasks, you should be able to accomplish the following:

- Specify sort orders for classes and diagrams.
- Add a condition to an iteration.
- Specify what you want ReporterPLUS to do when a model is missing data.

Sorting Model Elements

If you just completed the previous section, your ReporterPLUS template should still be open. If it is not already open, following the instructions in **Methods for Starting ReporterPLUS**.

To specify that ReporterPLUS should sort diagrams and classes by name, follow these steps:

1. In the template view, click on the `Package` iteration node (the first iteration node, labeled `'From class "Model"…'`). Be sure to select the *iteration node*, not the *iteration subnode*.
2. Click the Sort tab.
3. Click **Add key**.
4. Click the browse button next to the **Attribute** field to display a list of attributes. These are the keys by which you can sort.
5. Select **name** from the list, then click **OK**. Because you want the sort to be alphanumeric, ascending, and case sensitive, do not need to select any of the check boxes.
6. Click **OK**, and then click **Apply**.
7. Click on the `classes` iteration node in the template view and repeat Steps 2 through 6.

You can generate a Word document now to check the sort order, or you can wait until after you add a condition and then generate.

Adding a Condition

Conditions limit the model elements that ReporterPLUS includes from an iteration. When you add a condition, ReporterPLUS extracts only those model elements that meet the condition. If you do not specify a condition, ReporterPLUS extracts all the model elements in the iteration.

For example, the `classes` iteration in your template extracts *all* classes from the model. But what if you do not want all of the classes? If you want only classes named "Dishwasher," for example, you can add a condition to the iteration that limits the classes to those whose names are equal to "Dishwasher."

To specify a condition, you first choose the attribute you want to base the condition on, such as `name` or `metaClass`. Next, you choose the operator, such as = (equal to) or <> (not equal to). Finally, you specify the value you want to compare to the attribute. In this section, you add a condition so that only the overridden properties in the class are extracted.

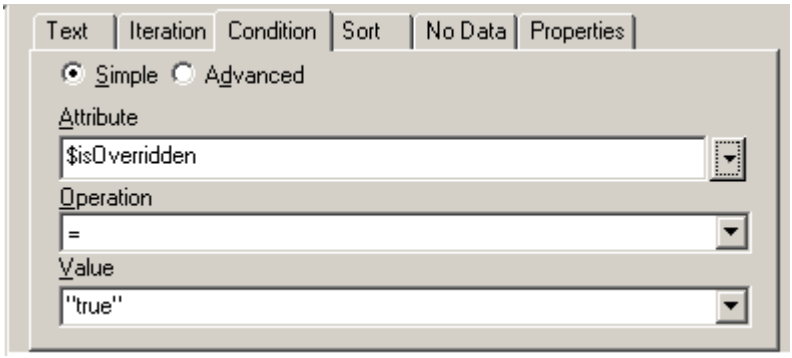
Note

ReporterPLUS supports two types of conditions: simple and advanced. This section describes simple conditions, which suffice for many purposes. Advanced conditions are written using the Q Language. For more information about Q, see the **Q Language** section.

Continue as follows:

1. In the template view, click on the property iteration node (`From class "Class"…`).
2. Click the Condition tab.
3. Make sure the **Simple** option is selected.
4. In the **Attribute** field, click the down arrow to display the drop-down list. These are the attributes on which you can base conditions for this iteration.
5. Select **isOverridden**, then click **OK**.
6. In the **Operation** field, click the down arrow and select the equal sign (=).

7. In the **Value** field, click the down arrow and enter **true**. Click **Apply**. The Condition tab should look like the following figure.



8. Save your ReporterPLUS template.

9. If the `Dishwasher.rpy` is not already open, open it now.

10. Generate a Word document, and go to the last page to check the results.For class `AcmeJet` in the model, you see only one property, `CG::Class::Concurrency`, because it is the only overridden property.

Coping with a Lack of Model Data

The «`No Model Data`» message is printed when the template attempts to extract information about an attribute that is not present in the model. For example, in the Dishwasher model, not all of the classes have descriptions.

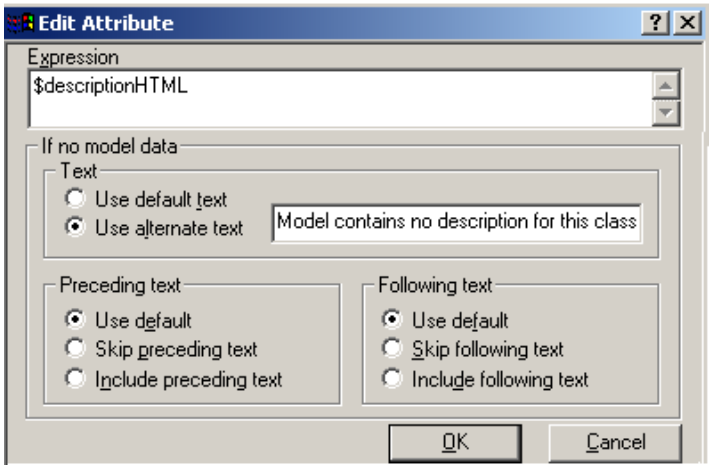
Missing data can occur on two levels: attribute and iteration. The missing description in the Dishwasher classes shows what happens at the attribute level: ReporterPLUS is able to extract classes from the model, but not all of the classes have data for all of the attributes specified in the template. There might also be missing data on the iteration level. For example, if you open a model that has no object model diagrams, then generate a document with a template iterating over object model diagrams, the template tries to extract object model diagrams, but there are none.

In this section, you learn how to control what ReporterPLUS does when data is missing from a model at either the attribute or the iteration level.

Coping with Missing Attributes

The «`No Model Data`» message is the default behavior of ReporterPLUS when attributes are missing from a model. To change that behavior to something more specific, follow these steps:

1. Click on the class subnode (the last node in the template view, labeled '`…iterate over association "classes"`').
2. On the Text tab, right-click on the «`$descriptionHTML`» attribute to display the context menu. Be sure to click directly on top of the attribute.
3. Select **Edit Attribute** from the context menu to display the Edit Attribute dialog box. This dialog box enables you to specify alternate text and to control what happens to text preceding and following the missing attribute.
4. Select **Use alternate text** and type **Model contains no description for this class** in the text field. ReporterPLUS prints this text instead of «`No Model Data`».
5. From the **Preceding text** section of the dialog box, select **Skip preceding text**. This means that ReporterPLUS does not print the word "Description:" for the missing attributes. The Edit Attribute dialog box should look like the following figure.



6. Click **OK** to dismiss the dialog box, then click **Apply** in the template node view.
7. Save your template.
8. Generate a *Word* document and go to the last page to look at the Classes section of the document. For the `AbstractFactory` and `AcmeFactory` classes, you should see the message "Model contains no documentation for this class."

Note

In the Edit Attribute dialog box, you can also specify whether you want ReporterPLUS to include any boilerplate text following the attribute. Refer to the online ReporterPLUS help topic “Coping with missing attributes” for details on exactly what text is included in “preceding” and “following” selections. In addition, you can create advanced statements about the attribute using the Q Language. For example, you can create a statement that specifies that ReporterPLUS should print a Boolean attribute only if its value equals “true.” For more information about Q, see the **Q Language** section.

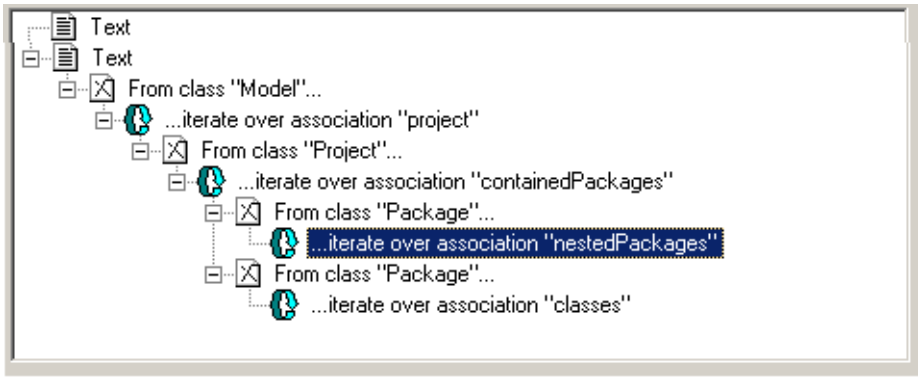
Coping with Missing Elements in an Iteration

When the model is missing an element that an iteration is attempting to extract, the default behavior for ReporterPLUS is to skip the iteration and print nothing for that iteration in the generated document. However, there might be instances when you want to know that the model does *not* contain an element. You can tell ReporterPLUS to print heading or body text when an iteration cannot find the specified model data.

Because your template is fairly simple—it extracts only diagrams and classes—in order to see how this option works, you need to add a generic element to the template that you know does not exist in the Dishwasher model.

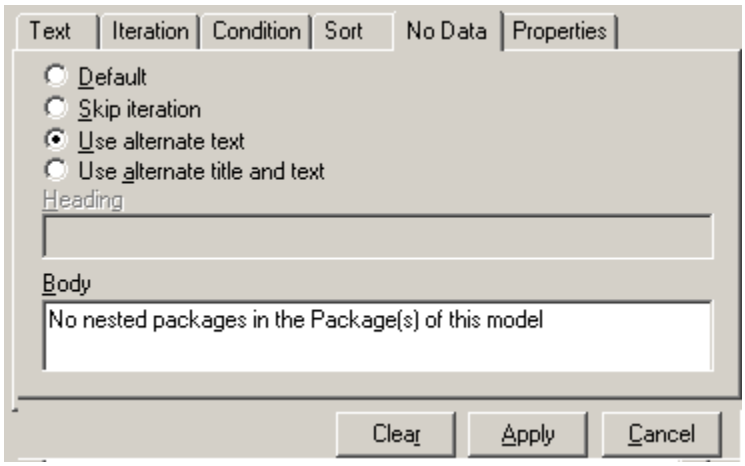
Do the following:

1. In the model view, expand the project node and the `containedPackages` node.
2. Select the `nestedPackages` node, and drag it to the template view on top of the `containedPackages` iteration subnode (…`iterate over association "containedPackages"`). ReporterPLUS puts the new iteration node at the end of the template, under the `containedPackages` classes subnode.



Note: As you drag `nestedPackages` over the nodes in the template view, the mouse pointer changes to the symbol ⃠. This means you cannot drop the item at that location.

3. Click on the new `nestedPackages` iteration node.
4. Click the **No Data** tab.
5. Select **Use alternate text**.
6. Click to place your cursor in the text box, then type "No nested packages in the Package(s) of this model." The No Data tab should be similar to the following example.



7. Click **OK** to dismiss the dialog box.
8. Save your template.
9. Generate a Word document, and go to the last page to check the results. You should see the heading "Package information for Package «$name»" followed by "No nested packages in the Package(s) of this model."

Next Steps

To learn more about ReporterPLUS and templates, you may wish to use the GetStarted template or the template you built in previous sections with some of your own models. Read the descriptions displayed in the Open Template dialog box to produce the desired reports.

For detailed instructions on other ReporterPLUS procedures, examine the **ReporterPLUS Online Help**.

Command-line Operation

The ReporterPLUS tool can be invoked using the following command-line interface (CLI). Developers prefer to use commands, entered onto a DOS prompt line, to run ReporterPLUS under these circumstances:

- Running Rhapsody in *batch mode*
- Creating a nightly build process that does not require any developer interaction

Launching ReporterPLUS from an MSDOS Shell

To control ReporterPLUS from the command line, follow these steps:

1. Open an MSDOS shell window.
2. Change to the directory containing the current version of Rhapsody and the ReporterPLUS program with this command:

   ```
   cd \<Rhapsody directory name>\reporterplus
   ```

3. Type the launch command (below) with any of the **Command-line Options** that are required.

   ```
   reporter.exe <options>
   ```

4. Press **Enter**.

Command-line Options

The options for the `reporter.exe` commands are listed below. These options may also include **Command-line Example**. The following table lists the ReporterPLUS CLI options.

Option	Description
/q or /quiet	Silent mode=Yes or GUI mode = No
/m or /model	Rhapsody model file name
/t or /template	Rhapsody template file name
/fn or /filename	Output file name
/ft or /filetype	Output file type (doc/html/ppt/txt)
/d or /display	Display the generated document
/s or /scope	Defines the portion of the model (shown by a path as a colon separated string) on which a command will be performed
/l or /license	License to be used (ReporterPLUS)

For some examples of how the "/scope" command is used, see the **Command-line Example** section.

Parameter Values

Any of these three parameters may be used with any of the options, listed previously.

Note

These command-line parameters are not case sensitive.

`/$param-name $param-value` (for example, /ft HTML)

`/$param-name=$param-value` (for example, /ft=html)

`/$param-name:$param-value` (for example, /ft:HTML)

Option Guidelines

The following guidelines describe the preferred uses for the command-line options for ReporterPLUS CLI:

1. With the `/q=Yes` (quiet=Yes) option, the developer should also use `/m`, `/l`, `/t`, `/fn`, and `/ft`. If any one of these are not included in the command, ReporterPLUS terminates and displays an error message. The use of `/s` is optional in either case. If `/s` is not provided, the default is the full scope.

2. The use of `/d` is optional when the developer runs ReporterPLUS CLI with a `/q=no`.

3. If the developer runs ReporterPLUS CLI with `/q=Y`, that option overrides `/d` and does not display the generated document.

4. If the developer does not use `/q`, then the ReporterPLUS CLI prompts the developer with a wizard asking for the following:

   a. Missing values (for example: output file type is not provided in the command-line interface)

   b. Undefined or Invalid values (for example: model file name passed in the interface is not existing)

   c. If all the required values were found, then the ReporterPLUS CLI generates the document (does not display the wizard), and based on the `/d` value, it may or may not show the generated document.

5. If the developer passes the scope indicator with `/s` along with a string to indicate the full path name to a package, then only that package, the model elements contained within, and sub-packages are loaded, as shown in this example:

   ```
   /s TopPkg:TargetPkg:Package
   ```

   `TopPkg` is the start of the path defining the scope.
   `TargetPkg` is the package within TopPkg.
   `Package` is the metatype.

   Note: The scope is limited to the package level only, and the metatype is limited to packages only.

6. If the scope is not passed, the full model is loaded.

7. If the developer does not use an `/l` command, then by default, the ReporterPLUS license is checked, and if it is not available, the ReporterPLUS CLI terminates and displays an error message.

Command-line Example

The following example shows a full example of the `reporter.exe` command with the options and parameters:

```
reporter.exe/m
C:\Rhapsody\Benchmark\Sample1\Benchmark1\benchmark1.rpy\
/s Package:ATMTransactionCtrl:Subsystems:benchmark1\
/t C:\Rhapsody\Templates\Drawing.tpl\
/ft html\
/fn C:\Test.html\
/q yes
```

1. In this example, the model to be used is identified with the “/m” command.

   ```
   /m C:\Rhapsody\Benchmark\Sample1\Benchmark1\benchmark1.rpy.
   ```

2. The scope of the model to be loaded is defined with “/s” command.

   ```
   /s Package:ATMTransactionCtrl:Subsystems:benchmark1.
   ```

3. The template file name to be used is identified with the “/t” command.

   ```
   /t C:\Rhapsody\Templates\Drawing.tpl.
   ```

4. The type of the document to be generated is defined with the “/ft” command.

   ```
   /ft html
   ```

5. The output file to be generated is defined with the “/fn” command.

   ```
   /fn C:\Test.html.
   ```

6. The ReporterPLUS CLI runs in silent mode and does not show the document generated because of this command: `/q yes`.

Execute Command

With the `Execute` command, you can specify the name of the DLL with a predefined function that it calls. Therefore, it is also possible to specify the arguments to be passed by referencing model meta attributes such as `$name` and `$GUID`.

The syntax for this command is as follows:

```
char* Execute(char* arguments)
```

This command provides the following capabilities:

- Load user-specified DLLs specifying a predefined entry point
- Generate a complex matrix and insert it into the generated report

Glossary

attribute view

The upper, right pane in the ReporterPLUS window. See also **attributes**.

attributes

The items listed in the attribute view. Attributes represent the pieces of data that can be extracted from a model. To add model text or diagrams to your document, you add attributes to the Text tab. ReporterPLUS adds some attributes automatically when you drag elements to the template view.

boilerplate text

Text that has been added to a template by typing in the Text tab. ReporterPLUS adds some boilerplate text automatically when you drag elements to the template view. Boilerplate text does not come from the model.

condition

A statement that limits the elements ReporterPLUS extracts from a model for an iteration.

expressions

In the **Q Language**, expressions examine and gather information about the model. Basic expressions are the fundamental building blocks of all Q language expressions. They are similar to numbers in arithmetic expressions. Composite expressions are the means by which larger expressions are constructed from smaller expressions. Hence, they are similar to arithmetic operators.

Another parallel between arithmetic expressions and expressions in Q is in evaluation. Just as in arithmetic expressions, the evaluation of an expression proceeds recursively, with each composite expression evaluating its subexpressions.

generic element

An element that represents a *type* of element that might be found in any Rhapsody model. See also **model element**.

iteration node

A template node formed by dragging a generic or model element to the template view. (An iteration subnode is created at the same time.) The iteration node specifies what class the iteration pertains to and what element to extract from that class. Iteration nodes can also specify conditions applied to the iteration, how elements are sorted, what happens when the iteration does not yield elements from the model, and can contain attributes and boilerplate text.

iteration subnode

A template node formed by dragging a generic or model element to the template view. The iteration subnode specifies the information included in the generated document for each element extracted by the iteration. Subnodes can contain attributes and boilerplate text, and might be the parent node to further iterations.

node label

The text next to the template node icon in the template view. The node label describes what the node does and what part of the model it is from. ReporterPLUS adds a default label automatically; you can change it in the **User-defined label** field on the Properties tab.

model element

An element from a specific model. See also **generic element**.

model view

The upper, left pane in the ReporterPLUS window. When a model is open, the model view displays generic and model elements. When no model is open, it displays only generic elements.

node

This is basically a section in a report that is specified at a heading level in the report.

output type

The format of the generated document. ReporterPLUS can produce documents in five output types: Microsoft Word, Microsoft PowerPoint, HTML, Rich Text Format (RTF), and text. You select the output type in the Generate Document dialog box.

Q Language

The language used to write advanced conditions and create advanced statements about attributes. For more information, see the **Q Language** section in this manual.

ReporterPLUS template

A set of instructions that tells ReporterPLUS what data to extract from a model and how that data should be formatted. You can build your own ReporterPLUS templates or use the ones provided with ReporterPLUS.

table node

An iteration node that produces a table rather than paragraphs of text. The information on the table node is formatted into column headings in the generated document, whereas the information on the iteration subnodes beneath the table node is formatted into table rows.

template

See **ReporterPLUS template**.

template node (or section)

Refers to any node in the template view. Template nodes form the structure of the generated document, and hold the attributes, boilerplate text, format commands, and other information that ReporterPLUS needs to generate a document from a model.

template node view

The lower, right pane in the ReporterPLUS window. The template node view has six tabs, which show information about the nodes in the template view.

template view

The lower, left pane in the ReporterPLUS window. The template view displays the currently open ReporterPLUS template. This view is empty when ReporterPLUS first starts.

text node

A template node that holds attributes and boilerplate text. Text nodes can stand on their own, serve as subnodes under iteration subnodes, or serve as parent nodes for iteration nodes. They cannot hold iterations.

Index

A

B

C

D

E

F

G

N

O

P

Q

R

S

T

U

V

W