Visual Paradigm

Doc. Composer Writer’s Guide

Last update: Feb 19, 2020

Copyright Information

Disclaimer

Publication Date

Published 23 April 2015, based on Visual Paradigm 12.0.

Revised 25 Apr 2018, based on Visual Paradigm 15.0.

Feedback

Please direct any comments or suggestions about this document to:
support-team@visual-paradigm.com

Table of Contents

Chapter 1. Introducing Doc. Composer

1.1 Introduction

Doc. Composer is a document builder in Visual Paradigm. It provides the necessary tools development teams need to write their own project documentation with design specification embedded.

This is a comprehensive guide that provides detailed description of how to work with Doc. Composer, the two modes of Doc. Composer and how to write element templates for different documentation needs. Examples are widely used in this guide and most of them are ready for practical use. They provide a good start point for writers both in learning Doc. Composer and developing their own documents and element templates.

1.2 Terminologies

Here is a list of terms that we will use in this document, along with their definitions:

Term	Definition
Content block	A block of document content produced by dragging an element template from Element Template Pane to document.
Diagram elements	A shape or connector in a diagram.
Doc. Composer	A document building tool in Visual Paradigm.
Doc Template	A Doc Template is the source of Doc Base. Users who work in team environment can define Doc Template to use in document production. When create document, Doc Base can be replicated from a Doc Template.
Doc Base	A semi-completed version of your project documentation or report. It contains only background information, possibly filled by you or your colleague. The design details are leave empty and be filled by Doc. Composer.
Doc Field	A Doc Field is a special piece of text within a Doc Base. Doc Fields will be replaced by your actual project content when being read by Doc. Composer during document generation.

Element Template	Written in XML, element template defines what and how content gets output in a document.
Fill-in Doc	A mode of Doc. Composer that helps you “fill-in” the design details of your project documentation.
Model element	Fundamental unit of project data of a Visual Paradigm project. Use case, class, action are all examples of model element.
Visual Paradigm Online	Cloud-based service provided by Visual Paradigm that allows you to store your project online and to work collaboratively with your teammates by enjoying features like commit/update, branching and tagged, PostMania ,Tasifier, etc.
VP Online	Abbreviation of Visual Paradigm Online
Word document	Document saved in Microsoft Word format (.docx).
Teamwork server	Self-hosted collaborative modeling.

1.3 What is Doc. Composer?

Documentation is important in any software project. We write software requirement specification for describing requirements, database specification for detailing database structure, process specification for visualizing business activities, etc. Well written documentation can ensure quality software to be developed, and can make a good impression on customers and stakeholders. However, we understand that time is limited and writing good documentation is not something we relish doing. Therefore, we introduced Doc. Composer, a document builder that saves your previous time in writing documentation by providing a smooth integration between your documentation and your software design. As long as you need to have your design or design specification appear in your documentation, Doc. Composer can help.

1.4 Modes of Doc. Composer

When you open Doc. Composer, you will be prompted to select either to build a document from scratch, or to produce a document with a “Fill-in Doc”. Here is a description of the two modes of Doc. Composer.

1.4.1 Build from Scratch

To build a document with the **Build from Scratch** mode is to being from a blank document, and then make use of the tools and element templates to write and complete the document.

1.4.2 Fill-in Doc

Typically, a project documentation or report is a combination of background information like project goal, scope and constraints, and design details like use case details, database design, process design, etc. The Fill-in Doc mode of Doc. Composer is designed to help you “fill-in” the design details of your documentation.

Chapter 2. Build from Scratch

2.1 Introduction

To build a document with the **Build from Scratch** mode is to begin from a blank document, and then make use of the tools and element templates available in Doc. Composer to write and complete the document. As an overview, “Build from Scratch” works in this way:

1. You create a document in Doc. Composer.
2. Form the content by dragging and dropping element templates from the **Templates** pane onto the document.
3. Touch-up the document by adding TOC, defining headers & footers, configuring styles, etc.
4. Generate the document to a document file in HTML/PDF/MS Word format.

2.2 Understanding Element Template

An element template defines what and how content gets output in a document. For example, a Data Dictionary template is capable in producing a data dictionary in a document, with dedicated type of project data presented in the data dictionary (table). Each type of project data has its own set of element templates. You have Data Dictionary template for Entity Relationship Diagram and Sub Diagrams template for Use Case, etc.

Doc. Composer comes with a set of built-in element templates, but you can also create and edit custom templates. For example, you can create a custom template to output a table of actors’ details, and then a list of use cases, and then a list of sub-diagrams.

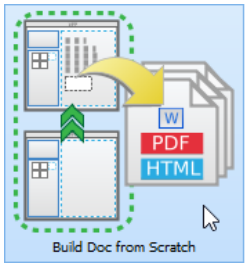
Element templates are written using an XML-based language called Doc. Composer Template Language. If you want to create your own templates, you can learn this language by reading Chapter 4. Writing Element Templates. In this chapter our focus is primarily on the use of built-in templates in creating a document.

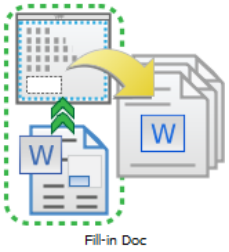
2.3 Creating a Document

To create a document:

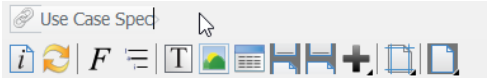
1. Select **Tools** > **Doc. Composer** from the toolbar.

2. Click **Build Doc from Scratch**.

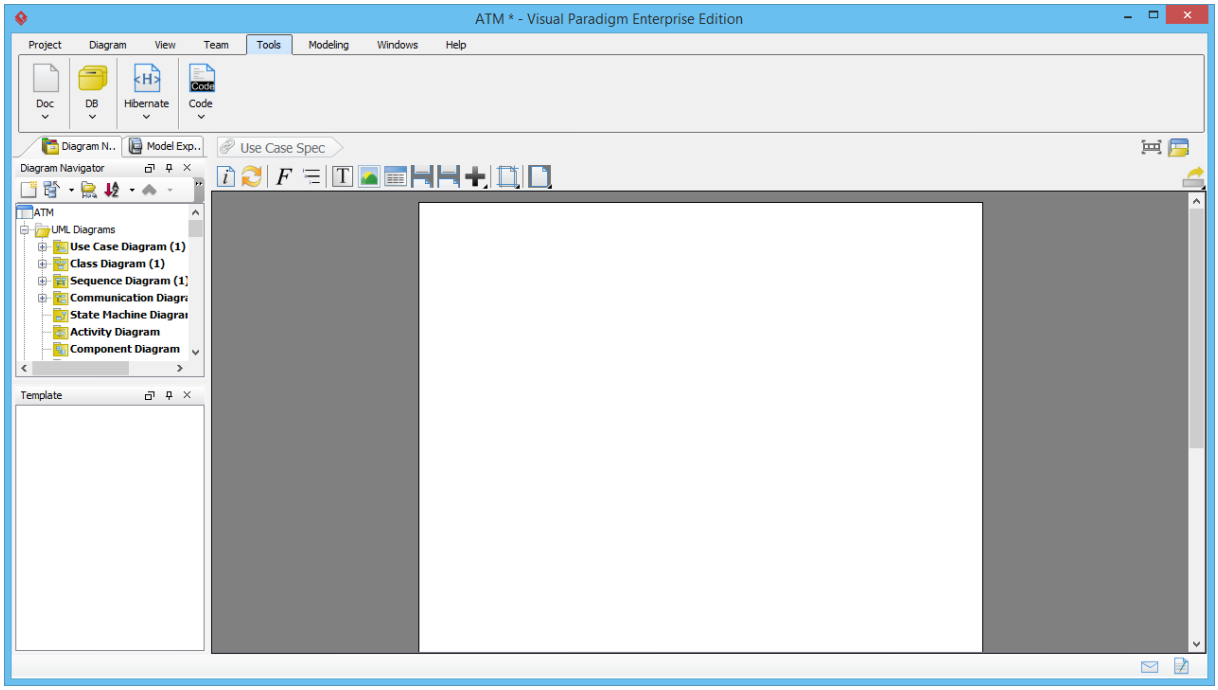




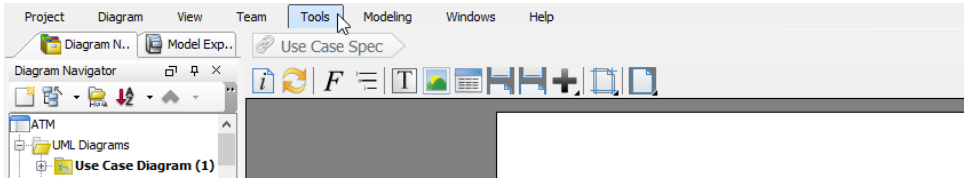
3. Name the document by double clicking on *Document1* in the breadcrumb and then typing in a new name.



4. Press the **Enter** key to confirm the naming. Your screen should look like this:

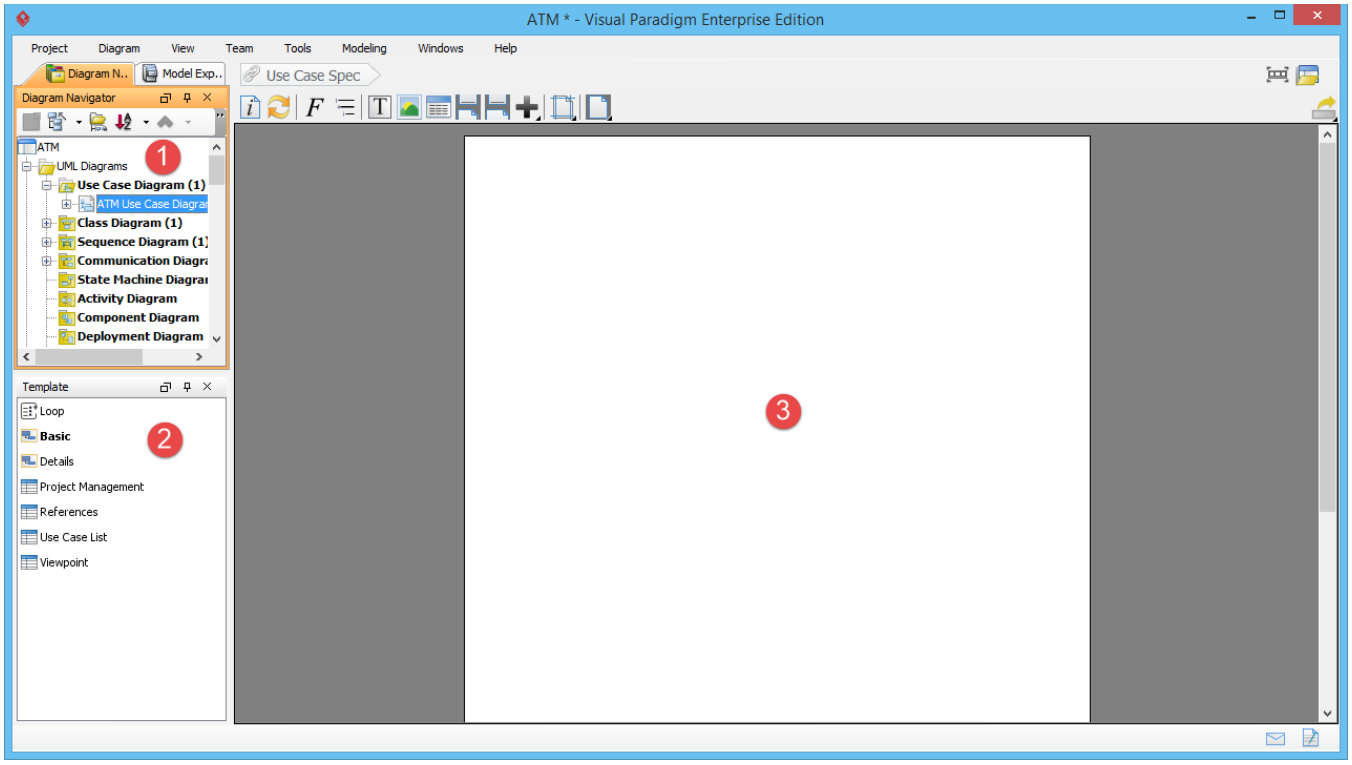


5. To have more editing space, we recommend you to collapse the toolbar temporarily by double clicking on the **Tools** tab.



Notes	If you don't see the **Doc. Composer** button, make sure you are running the Standard Edition (or higher) of Visual Paradigm, and have **Sleek** chosen as the UI style.

2.4 Overview of Doc. Composer



The “Build from Scratch” mode of Doc. Composer consists of three main parts:

Part	Name	Description
1	Diagram Navigator/ Model Explorer	The **Diagram Navigator** and **Model Explorer** (behind **Diagram Navigator**) provides you with access to different parts of your project– the project, diagram, diagram elements and model elements. If you want to output content for say a use case diagram named “My Diagram”, select “My Diagram” in **Diagram Navigator** (or **Model Explorer**) first.
2	Element Template List	A list of element templates available for the project data selected in **Diagram Navigator** or **Model Explorer**. As said earlier, each type of project data has its own set of element templates. Take the image above as an example, by selecting a use case diagram in **Diagram Navigator**, the templates **Basic**, **Details**, **Project Management**, etc. are listed. If you select other project data in **Diagram Navigator** or **Model Explorer**, a different list of element templates will be presented. Document creation is the process of dragging and dropping a suitable template from the Element Template List onto the document.

3	Document	Content of document. What you can see here will be what you will get when you generate the document as HTML/PDF/Word. On top of the document there is a toolbar that provides you will access to tools like format configuration, report configuration, page break insertion, etc. We will talk about the tools later on in this chapter.

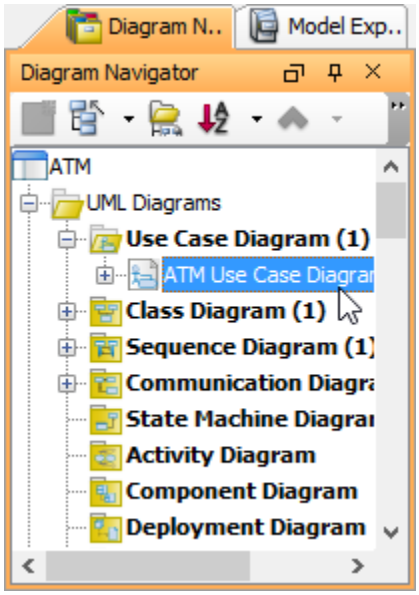
2.5 Developing a Document

Doc. Composer provides a flexible way for you to create project documentation. All you need to do is to select your target model element/diagram, drag the elements template(s) from the **Element Template Pane** onto the document.

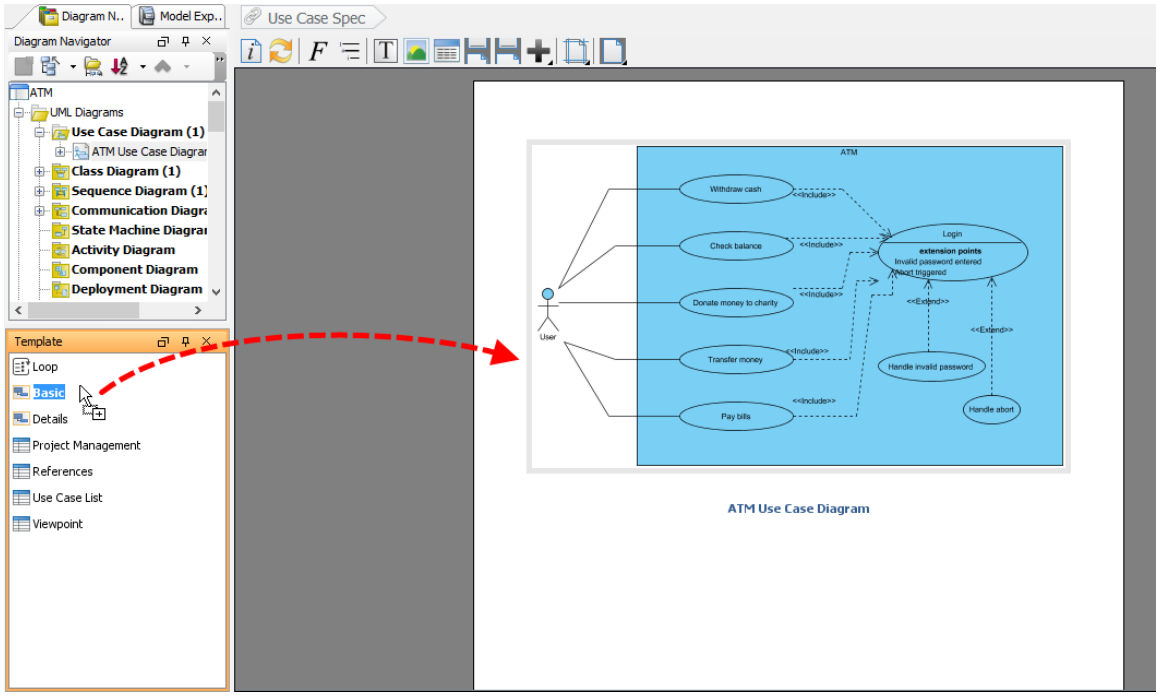
2.5.1 Using a Template

To add content into a document:

1. Select project / diagram / diagram element / model element on **Diagram Navigator** / **Model Explorer**.



2. Select the desired element template from the **Element Template Pane** and drop it/ them on the document.



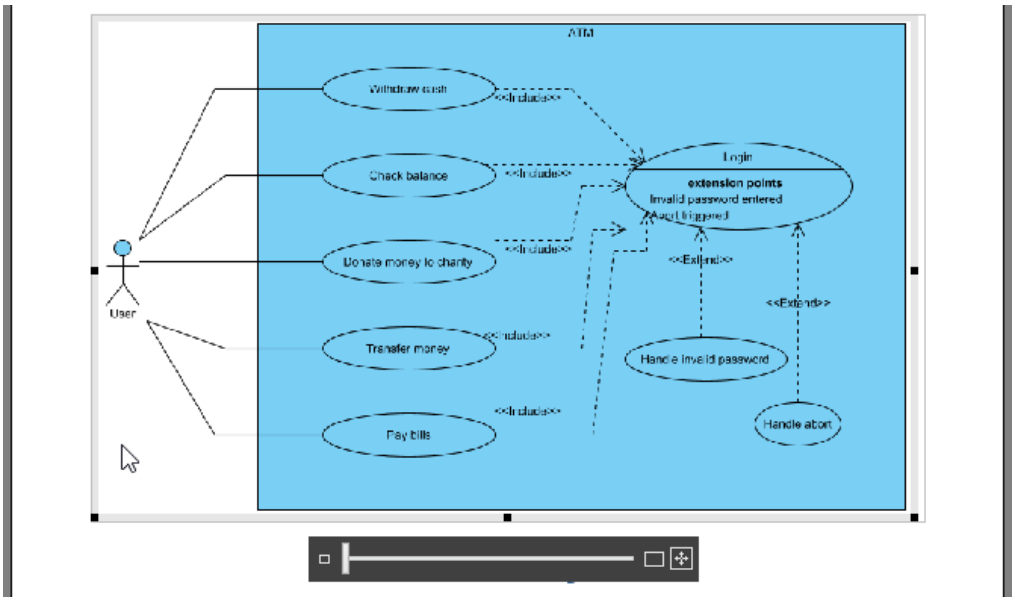
3. Repeat the previous step to build the document.

Notes	Instead of adding content element by element, you can select multiple elements at the same time to speed up the document creation process.

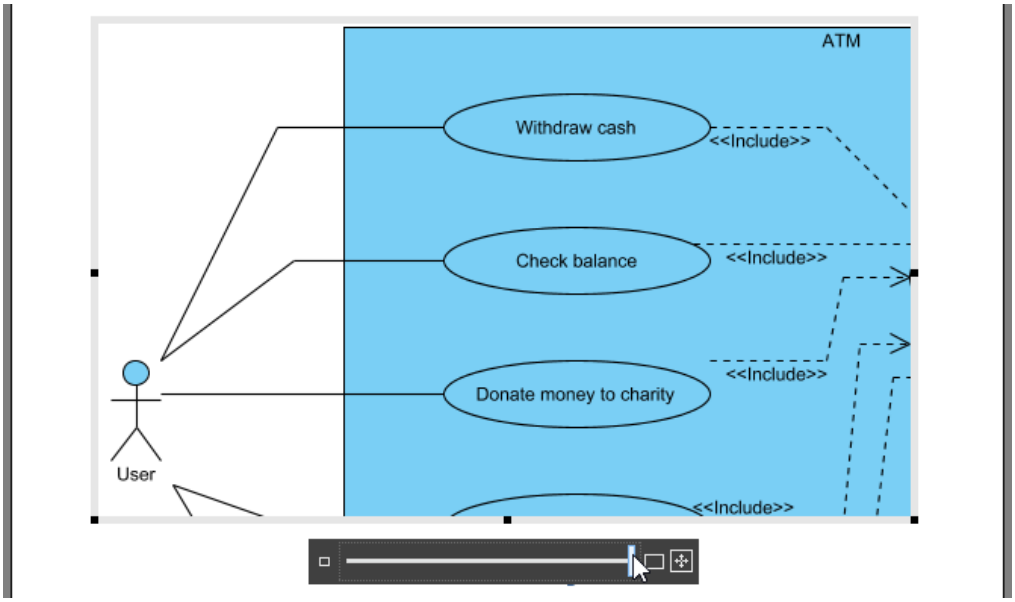
2.5.1.1 Editing image

You can add diagram images into a document with the use of element template. For diagrams presented on a document, instead of showing the entire diagram you may want to have it focused on a specific part of a diagram. In order to achieve this, edit the image by taking the steps below.

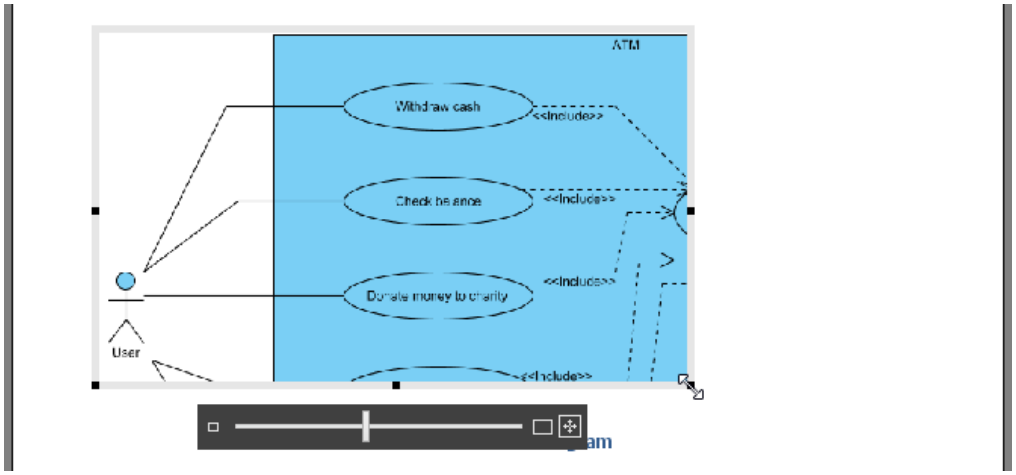
1. Click each image so that a bar will appear at the bottom of the image. You can edit the image through the bar. Initially, the whole diagram is displayed to fit the placeholder.



To zoom in the particular part of diagram, drag the slider to Zoom 100% (Actual Size).



2. To resize the image, drag the border of placeholder.



Notes	You can only edit the images produced by templates you dropped onto the document but not inserted images.

Notes	The default setting of image on document is **Disable Auto Fit Placeholder**. Nevertheless, once you zoom or resize, it will turn to be **Enable Auto Fit Placeholder** automatically.

2.5.1.2 Working with diagram layers

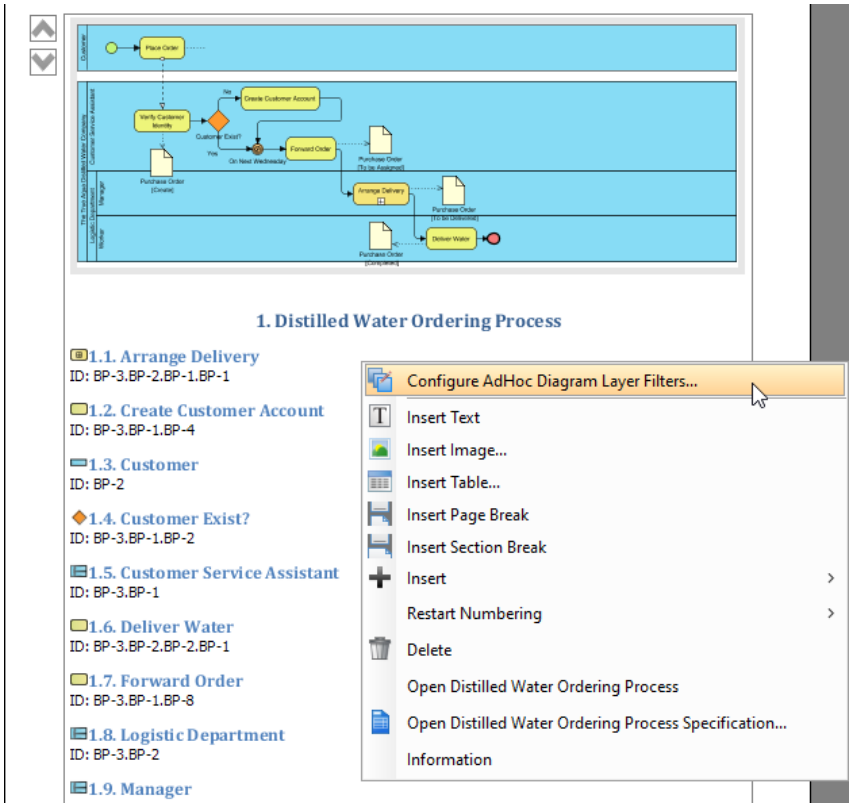
The layer feature allows you to group diagram content into different logical layers. For example, you may create an annotation layer for explanatory purpose. Such a layer contains shapes like callout and note, but not the other shapes. Then when you want not to show the annotation shapes temporarily, you just need to hide the layer.

Doc. Composer supports diagram layers. You can select the diagram layer to or not to process when outputting content to a document. Let's say if you need to produce a document for a business stakeholder, you may not want him to see the annotation shapes. What you have to do is to configure the layer filter by excluding the annotation layer. Doc. Composer engine will read the filter and not to process the annotation shapes.

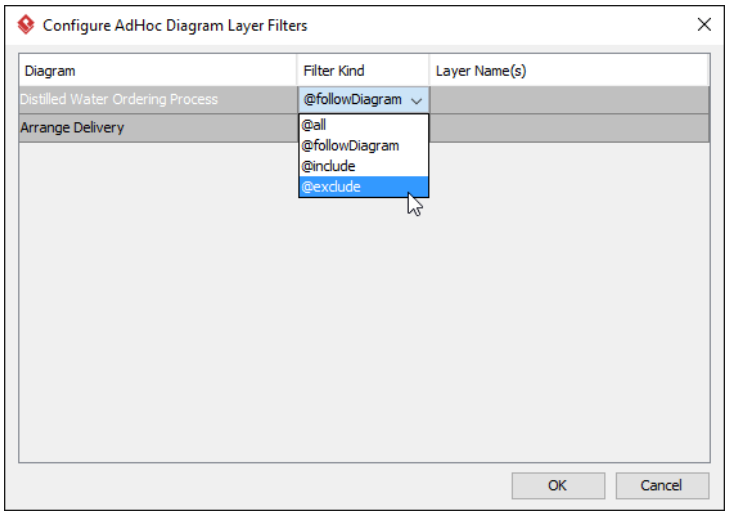
To configure filter:

1. Right click on a content block that involves processing diagram(s) and select **Configure AdHoc Diagram Layer Filters…** from the popup menu. Note that the word 'diagram' here is not restricted to 'diagram image'. If a content block only involves listing the name of the shapes in a

diagram, it is still processing a diagram.

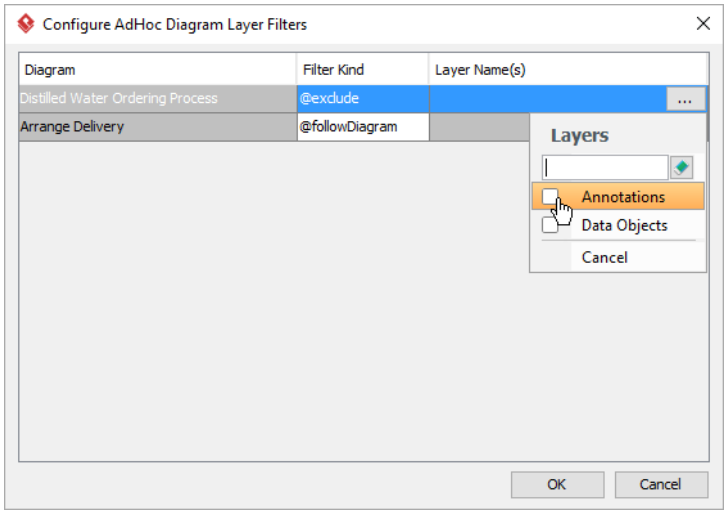


2. The diagrams being processed are listed in the **Configure AdHoc Diagram Layer Filters** window. For each diagram, select which layer to include in or exclude from processing. First, select the **Filter Kind**. There are four options.



@followDiagram - Follow the visibility of the layers set to the actual diagram. Layers that are set visible will be included here, likewise hidden layers will be excluded. Simply put, what you can see in the document will be exactly the same as the real diagram.
@all - Include all diagram layers in processing.
@include - Select the layer(s) to process.
@exclude - Select the layer(s) not to process.

3. If you have selected **@include** or **@exclude**, select the layer(s) to include in or exclude from processing.



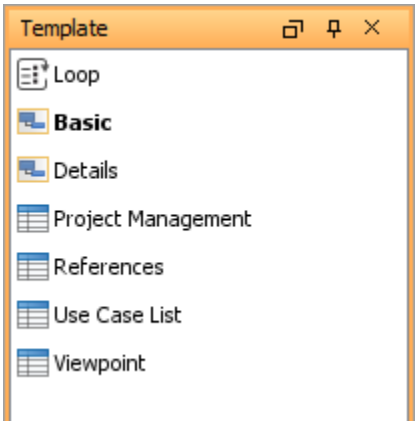
4. Click **OK**. The document will be refreshed immediately to reflect the filter configuration.

2.5.1.3 Understanding the default template

Each type of project data has its own set of element templates and, among these templates, one of them is the default template.

Normally, we add content into a document by first selecting an element in **Diagram Navigator** or **Model Explorer**, and then dragging a template from the **Element Template List** onto a diagram. This works well but wouldn't it be a bit faster if we can skip the template selection part? Default template was designed to serve this purpose. Besides dragging a template onto a document, you can also select a piece of project data in **Diagram Navigator** or **Model Explorer**, and then drag it directly onto the document to add content. If you do this, we will add content based on the default template of the selected element.

Default template is shown with its name bolded under the **Element Template Pane**, like the **Basic** template in the following image:



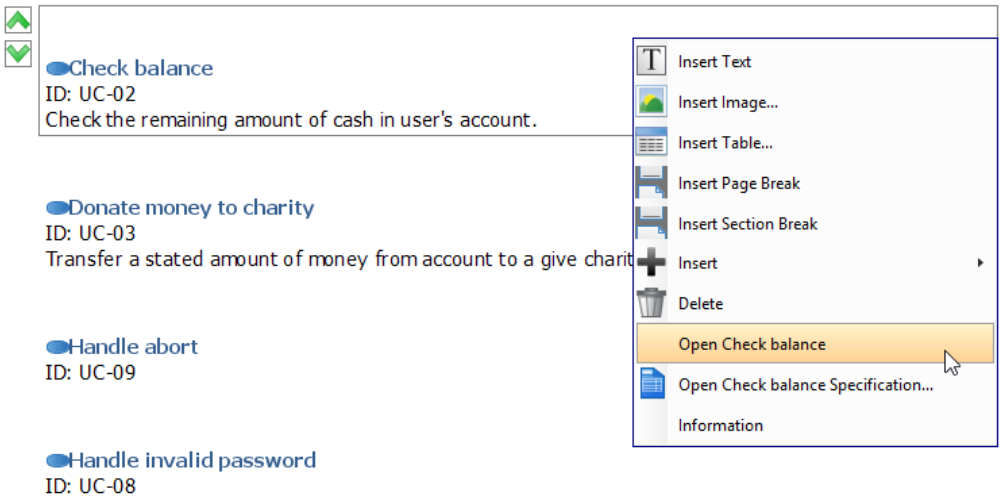
By default, the **Basic** template is chosen to be the default template. You can, however, set a user created template to be the default. Note that you can select either the **Basic** template, or any of the

user created templates as default. You cannot select a built-in template as the default, except the **Basic** template.

2.5.2 Working with Content Block

2.5.2.1 Opening the Underlying Element

If you want to open the source element from which a content was created, right click on the content block and select **Open %NAME%** from the popup menu where **%NAME%** is the name of the element. Very often you need this when you spot a design flaw when reading a document, and you want to correct it.



2.5.2.2 Opening the Specification of the Underlying Element

If you want to review or edit properties of an element from which a content was created, right click on the content block and select **Open %NAME% Specification** from the popup menu where **%NAME%** is the name of the element.

2.5.2.3 Repositioning a Content Block

You can always re-order content blocks in a document by performing these steps:

1. Press to select the blocks to move up or down. You can perform a multiple selection by pressing the **Shift** or **Ctrl** key.

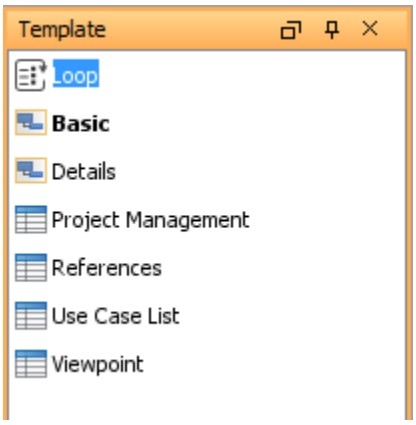
2. Click on **Move Up** or **Move Down** to re-order the selected blocks.

2.5.2.4 Deleting a Content Block

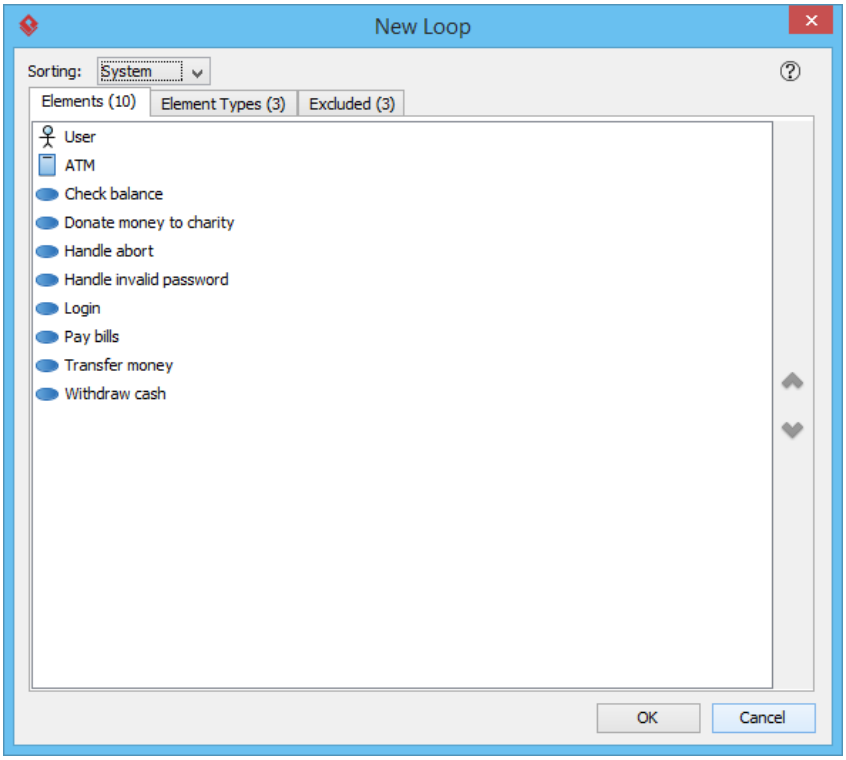
If you want to remove a content block from a document, simply select the blocks and press **Delete**.

2.5.3 Using Loop

Besides element template, there is another tool to query project data and place it onto a document, called the Loop tool. You can access the Loop tool under the **Element Template Pane**.

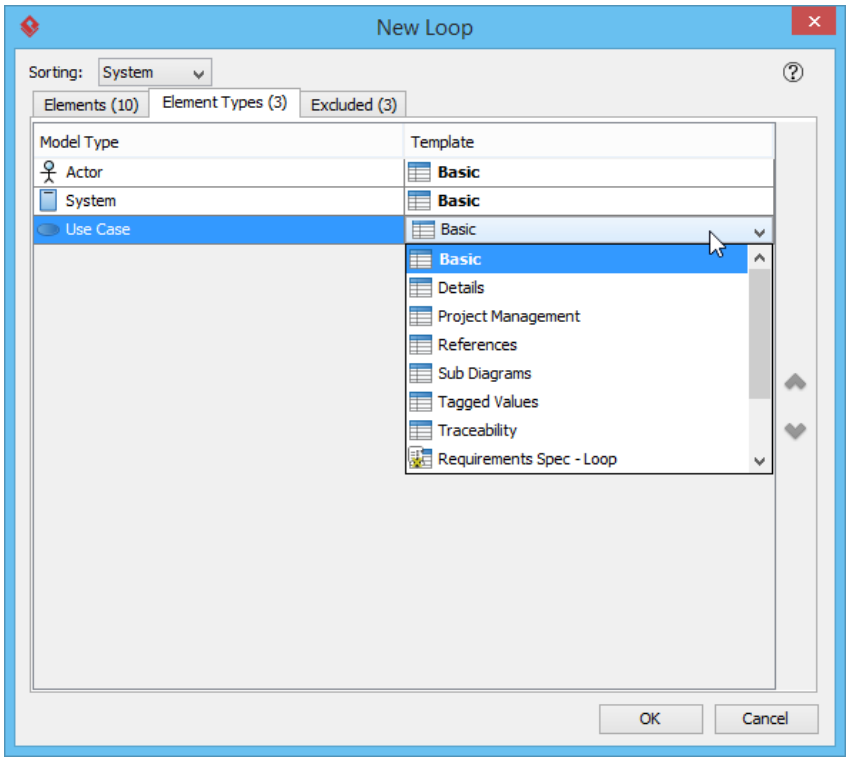


The Loop tool is capable in querying children element of element selected in **Diagram Navigator** / **Model Explorer**. When you select and drag the Loop tool onto a document, you will be prompted to configure the Loop, like this:



Items listed under the **Elements** tab are children elements of the element selected in **Diagram Navigator** / **Model Explorer**. For example, if you have selected a diagram, you can expect the diagram elements being listed here.

Content will be added to document for each of the elements listed. The content to produce is determined by an element template. You can select the template under the **Element Types** tab.

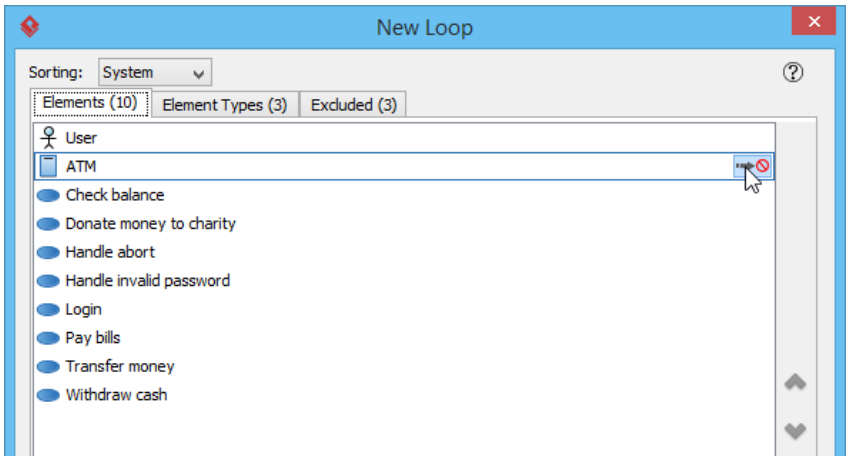


The left hand side of the **Element Types** tab lists the types of elements found by the loop while the right hand side lists the element template to be chosen for content creation. By default the **Basic** template is chosen for all element types. This means that for each of the elements under the **Elements** tab, content will be produced and added to the document based on the **Basic** template. You can select another template by click on **Basic** and make a selection.

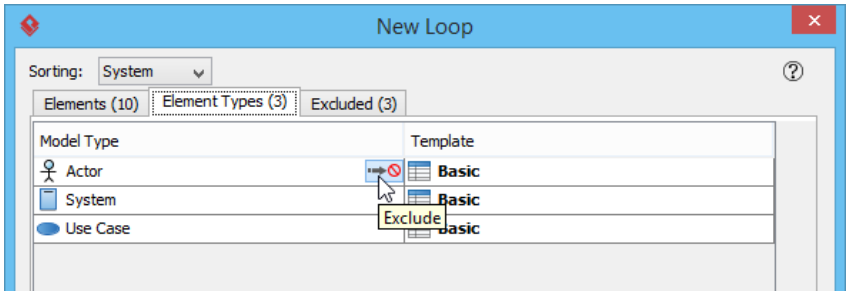
2.5.3.1 Excluding Elements and Element Types

Sometimes, you may not want to produce content for all elements returned by a loop, but only some of the elements. For example, when you create a use case specification, you may not want to process the actors when looping under a Use Case Diagram. For such cases, you can exclude the elements or types of element that are not needed in content creation.

You can exclude an element, or a type of element. To exclude an element, move your mouse pointer over that element under the **Elements** tab and the click on the **Exclude** button on the right hand side of the hovering row.



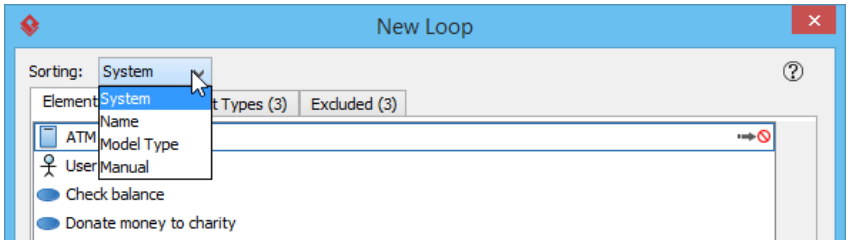
To exclude an element type, open the **Element Types** tab and move your mouse pointer over the type to exclude, then click on the **Exclude** button.



If you want to remove an element or element type from the exclude list, open the **Excluded** tab, move your mouse pointer over the item to be removed and then click on the **Include** button on the right hand side of the hovering row.

2.5.3.2 Sorting

To re-order elements, click on the **Sorting** drop down menu at the top of the **New Loop** window and select the way to sort.



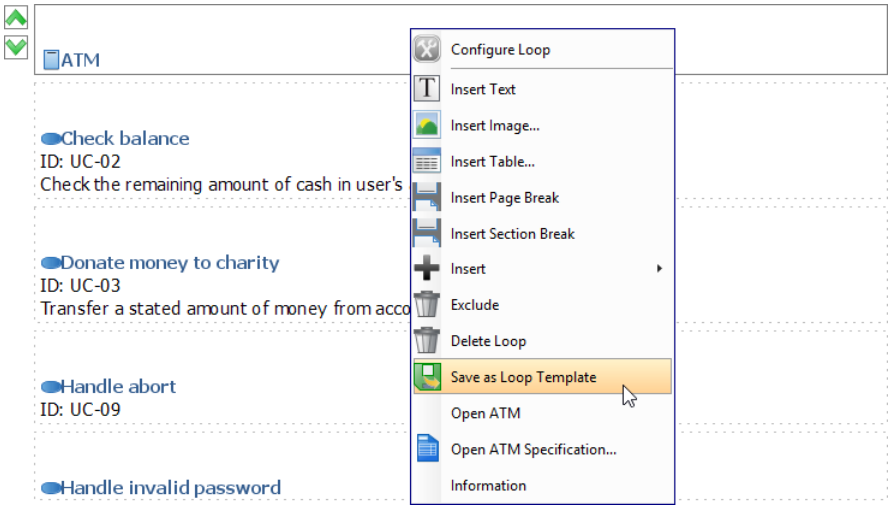
Here are a description of the various types of sorting method:

Type	Description
System	Elements are sorted according to the system's default setting.
Name	Sort by elements' name in alphabetical order.

Model Type	Sort by element type. You can customize the sort order in the Element Types tab.
Manual	Order the elements yourself.

2.5.3.3 Saving a Loop Template

Although Loop itself is tool instead of a template, you can produce a template from it. Doing so allows you to customize the template further by editing the loop in XML form. To produce a loop template, right click on any content block produced with the Loop tool and select **Save as Loop Template** from the popup menu.



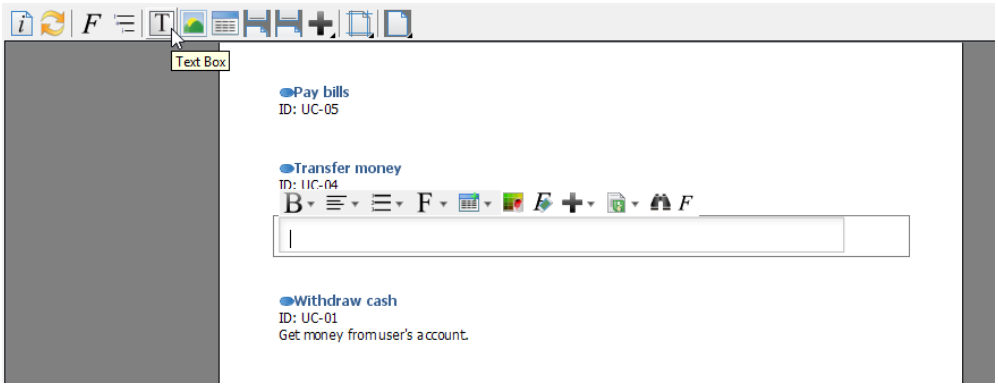
You will be prompted for a template name. Enter the name and confirm. When finished you will find a new template listed in the **Element Templates Pane**. You can customize it and re-use it in creating content. For details about writing a template, please read Writing Your Template.

2.5.4 Adding Custom Text

Although you cannot type in a document directly, you can add text through creating text boxes.

The text box is used for editing data on document. The significant characteristic is you can display many different types of data by applying RTF within the text boxes.

1. Select the space where you want to insert text beforehand.
2. Click Text box button on the document's toolbar.

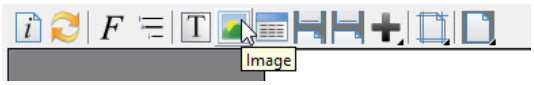


3. Enter text in the text box. You can use the pop-up formatting toolbar to convert your plain text into RTF when you want to emphasize some terms/ phrases.

2.5.5 Adding Image

Document supports inserting images. An image can be a logo or picture that is placed on the document. Not only you can place pictures on the empty space of document but also fit them inside table cells. In this sense, you can insert your company logo into any preferred place within the document when you are doing a company document. The advantage is you can spare no effort in arranging a series of images in document and then resize them. To add an image:

1. Select the content block where you want to insert an image beforehand.
2. Click Image button on the document's toolbar.

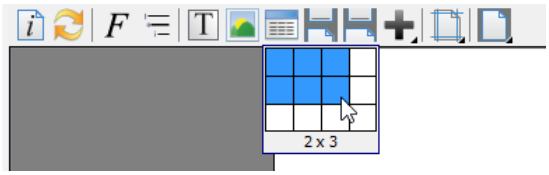


3. Select the directory of your target image and then click **Open** button in **Choose image(s)** dialog box. As a result, the selected image is inserted.

2.5.6 Adding Table

Table is one of the popular elements in structuring data. To create a table:

1. Select the content block where you want to insert a table beforehand.
2. Click on the button on the document's toolbar.
3. Select the number of rows and columns to be created.



4. Complete the table.

2.5.7 Adding Page Break

To end the current page and continue the document to the next page, add a page break. To add a page break:

1. Select the content block where you want to insert a page break beforehand.
2. Click on the button on the document's toolbar.

2.5.8 Using Section

A section is a number of continued pages that apply the same set of page properties. These properties include page size, page orientation, page margin, visibility of header/footer, content of header/footer, etc.

Because section allows you to define different layouts for different pages, you can make pages that consist of wide tables show in landscape, with the other pages remain in portrait. You can also add content-specific header and footer.

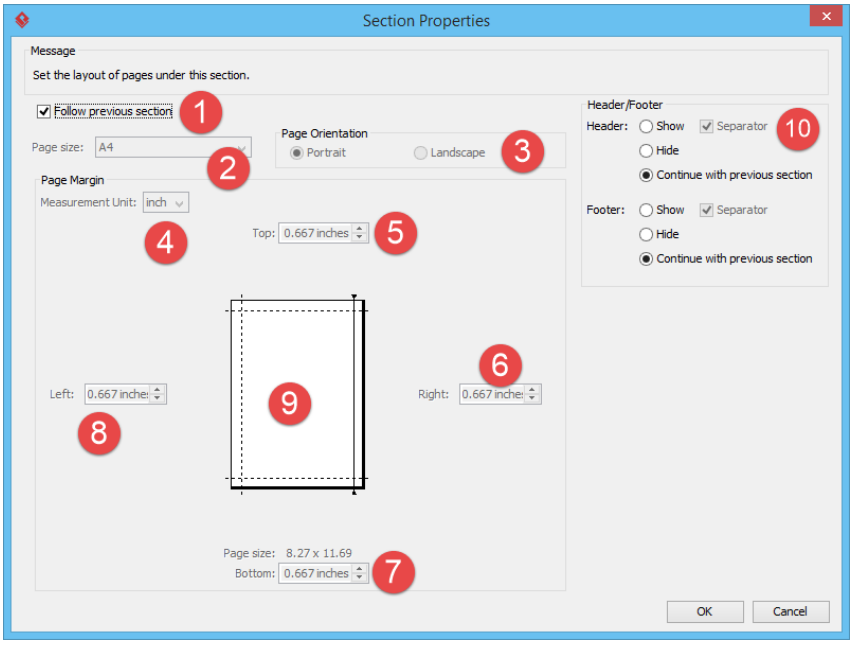
By inserting a section break, pages that appear after the break will apply the same set of page properties as defined in the break. To insert a section break:

1. Select the content block where you want to insert a section break beforehand.
2. Click on the button on the document's toolbar. A section page is inserted which moves the chosen content to a new page.

The layout of pages within a section are controlled by the setting configured in the section break. To configure a section break:

1. Right click on the section break.
2. Select **Edit...** from the popup menu. This shows the **Section Properties** window.
3. Edit the settings and click **OK** to confirm the change.

Here is a description of different parts of the **Section Properties** window.



Part	Name	Description
1	Follow previous section	When checked, section properties will follow that defined in previous section. For the first section, it follows the properties set to the whole document.
2	Page size	Determine the dimension of page.
3	Page Orientation	Orientation of page, either in portrait or landscape.
4	Measurement unit	Unit of margin.
5	Top margin	Determine the empty space at the top of a page. You can edit the margins size by entering the sizes into the text fields. Alternatively, click the spinner buttons to increase/ decrease the margin sizes.

6	Right margin	Determine the empty space on the right hand side of a page. You can edit the margins size by entering the sizes into the text fields. Alternatively, click the spinner buttons to increase/ decrease the margin sizes.
7	Bottom margin	Determine the empty space at the bottom of a page. You can edit the margins size by entering the sizes into the text fields. Alternatively, click the spinner buttons to increase/ decrease the margin sizes.
8	Left margin	Determine the empty space on the left hand side of a page. You can edit the margins size by entering the sizes into the text fields. Alternatively, click the spinner buttons to increase/ decrease the margin sizes.
9	Preview of page	Preview the effect of adjusting margin.
10	Header/Footer	**Show** - Show the header/footer in document **Hide** - Hide the header/footer in document **Separator** - When checked, a line will be shown between header/footer and the main content. **Continue with previous section** - Following the previous section means to have the visibility of header, footer and separator follow that defined in the previous section. If you have added page number to header/footer, the numbering will continue with the previous section, too. If not to follow the previous section, the numbering will reset to 1.

2.5.9 Adding Table of Contents

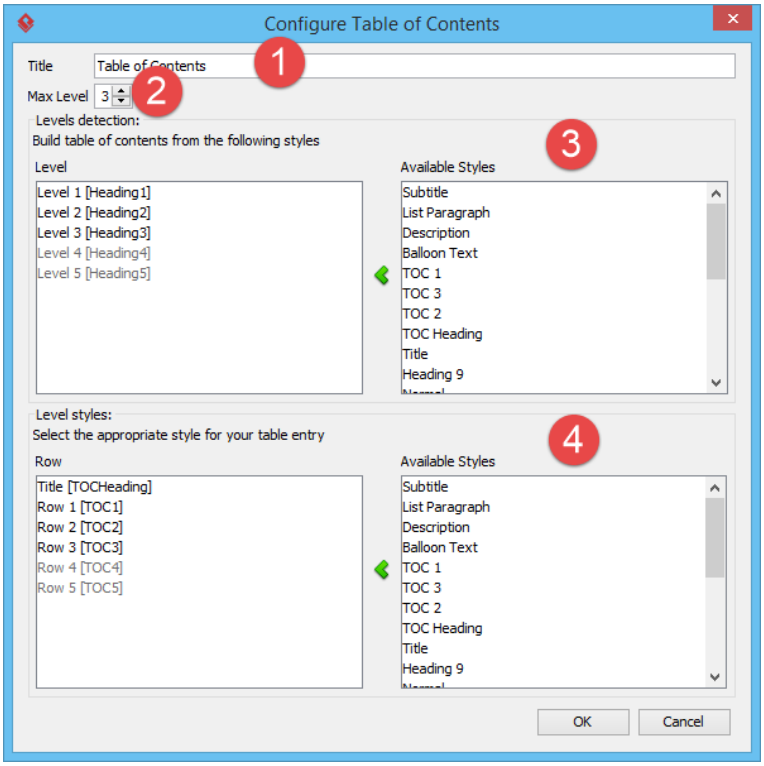
A table of contents is a list of key parts of a document. It is often constructed by headers or key titles in a document, to present readers with and outline of the whole document.

Doc. Composer allows you to insert a table of contents into a document. A table of contents can be formed not only from traditional headings styles like Heading 1 and Heading 2 but from any kind of style, even from user-defined styles. To insert a table of contents:

1. Select the content block where you want to insert a table of contents beforehand.
2. Click on the + button on the document's toolbar and then select **Table of Contents** from the drop down menu.

To change the title, maximum number of level, level detection or styles of a table of contents, to configure it. To configure a table of contents, right click on the table of contents and select **Configure Table of Contents...** from the popup menu.

Here is a description of different parts of the **Configure Table of Contents** window.



Part	Description
1	The title of the table of contents. This is the text that appear above the table of contents in document.
2	Determine the depth of the table of contents.
3	Specify the style to check for each level. If you want level 1 shows all content with Heading 1 as style, select Level 1 on the left hand side, Heading 1 on right hand side, and click < to match them up.
4	Specify the appearance of text in table of contents. You can apply different styles for different rows (levels).

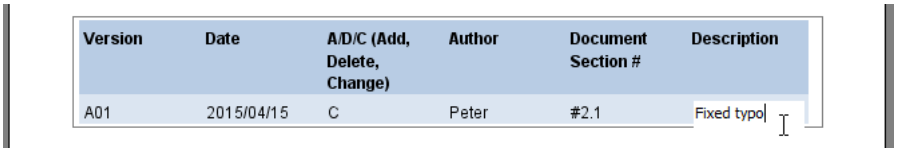
To update a table of contents to make it reflect the structure of the latest document content, right click on the table of contents and select **Update Table of Contents** from the popup menu.

2.5.10 Adding Revision Log

When your team attempt to or has made significant changes on a document, you may want to record the version of document, the date/time when the change took place, the person who made the change and other necessarily remarks regarding the changes. Revision log is a piece of content you can add to a document to record all these information. With a revision log, you fill in the revision detail as well as to add/remove columns to suit the requirement of your team. To insert a table of contents:

1. Select the content block where you want to insert a revision log beforehand.
2. Click on the ➕ button on the document's toolbar and then select **Revision Log** from the drop down menu.

3. To enter a revision, double click on the cells and enter the values one by one.



4. If you want to insert more rows or columns, right click on the revision log and select **Insert Row** or **Insert Column** from the popup menu.

2.5.11 Adding Cover Page

A cover page is a page that can be added at the beginning of a document. Whether or not to add such page is up to the writer. There are two kinds of cover page you can add into a document. The first one is to print the cover page in a program defined way. This approach requires you fill in some of the background information like the document title, organization name and author name, etc. The second kind of cover page is fully designed by you, the writer. It is called a free style cover page.

2.5.11.1 Built-in Cover Page

1. Click on the button on the document's toolbar.
2. Open the **Cover Page** tab.
3. Configure the cover page by specifying the file path of logo image, title, organization name, author name. You can preview the page at the right hand side of the **Document Properties** window.
4. Click **OK**.

Notes	Built-in cover page is only visible in generated document, not in Doc. Composer.

2.5.11.2 Free Style Cover Page

Free style cover page provides you with a page that appears at the beginning of a document for you to design the page. You can add any text and image freely on the cover page and position them in any position you like within the cover page. To insert a free style cover page:

1. Click on the button on the document's toolbar and then select **Cover Page** from the drop down menu.
2. When you insert a free style page the first time, you are prompted to override the generate cover page option. By default, the built-in cover page would be chosen as cover page. When you try to insert a free style cover page, the built-in cover page would be ignored. This option is to ask for your confirmation for ignoring the built-in cover page. Click **OK** to confirm.
3. You will see an empty cover page added to the beginning of the document. Note that the page MUST be added to the beginning of document and you cannot control its location. If you want to add a page of custom content in the middle of the document, insert a **Free Style Page** instead.
4. Start editing the page by inserting text and image. To insert text or image into the page, right click on the background of cover page and select **Insert Free Style Text** or **Insert Free Style Image** from the popup menu.
5. Fill in the text or select the image file to insert to the page. Repeat step 3 and 4 to complete the page.

2.5.12 Various Page Display Options

Page display is especially useful when you view the overview of document layout. Doc. Composer supports 4 display options: single page, single page continuous, two-up and two-up continuous.

Option	Description
Single Page	Display only one page at a time.
Single Page Continuous	Display pages in a consecutive and vertical column.
Two-Up	Display two pages side by side simultaneously.
Two-Up Continuous	Display pages side by side in two consecutive vertical columns.

Click the **Page Display Option** button to select a page display option from the drop-down menu.

2.6 Keeping Your Document Updated

Since document maintains the linkage between project data and document content, you can refresh the document upon project changes. As a result, it saves your time on repeating the steps for creating document.

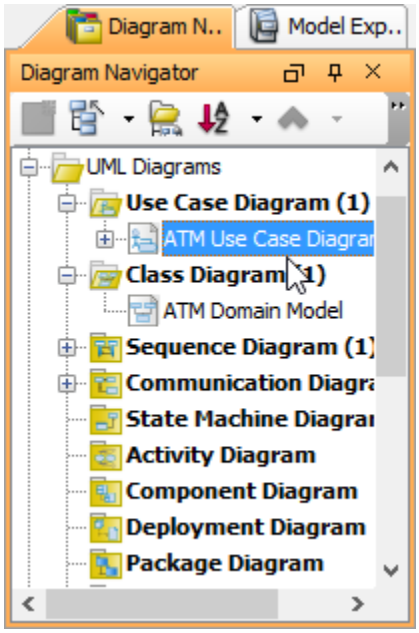
2.7 Writing Your Template

Although Doc. Composer comes with a complete set of built-in element templates, you may still want to customize or even to write your own templates for maximizing the efficiency of document design. You can accomplish this by writing and programming your own element templates.

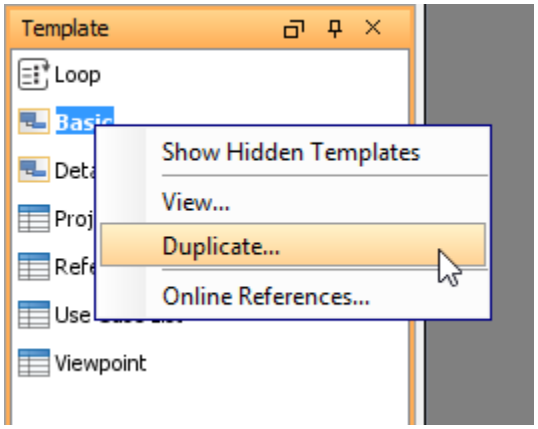
To create a template:

1. Open Doc. Composer.
2. Select the type of element to create template. For example, select ANY use case diagram in **Diagram Navigator** if you want to create a template to list specific shapes in use case diagram. You can select project / diagram / model element / diagram element in **Diagram Navigator** and

Model Explorer.



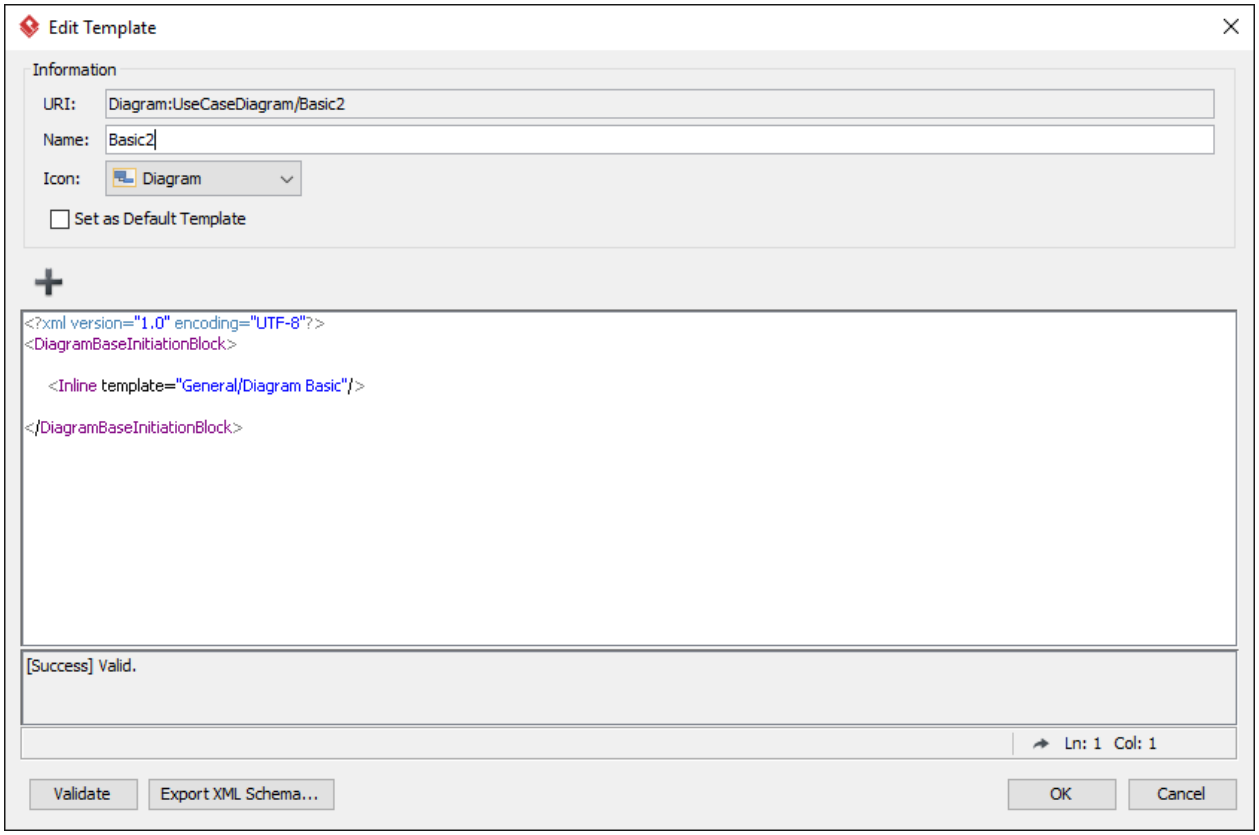
3. The **Element Template Pane** lists the templates available for the selected element type. If your Template pane is hidden, select **View > Panes > Property** in the toolbar (or press **Ctrl+Shift+P**) to show it.
4. To simplify the programming of template, you are suggested to duplicate an existing template and start editing it, rather than do everything from scratch. Target on a template that gives the closest outcome to what you want to show in document. If you want to start from an empty document, select any templates. Right click on your selection and select **Duplicate...** from the popup menu.



5. In the **Edit Template** window, specify the following information and click **OK** to continue.
   URI: A unique URL of the template, used in plugin development
   Name: The name of the element template, which is the name that shown under the Element Template Pane.
   Icon: An icon that best represent the layout of content that will be produced by using this template.
   Set as Default Template: Check this option if you want Doc. Composer to apply this template automatically when dragging elements directly from Diagram Navigator / Model Explorer onto

document.

Template content: Editor for programming the template.

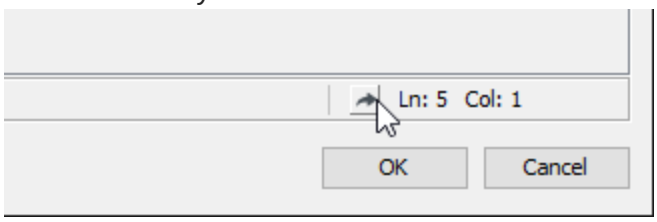


6. Customize your template and click **OK** to apply the changes. To learn how to write a template, read Chapter 4. Writing Element Templates.

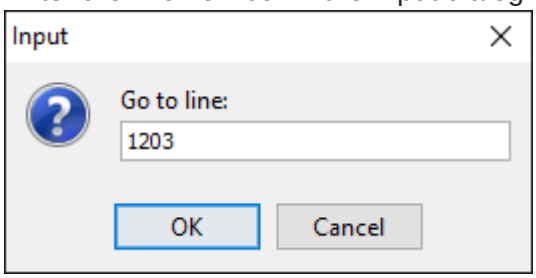
2.7.1 Go to line

If your template contains invalid content, by performing a validation (by clicking **Validate**), the line number of the problem content will be displayed in the log pane. To view the problem content, you can use the 'Go to line' function.

1. Click on the tiny arrow button next to the current line number.



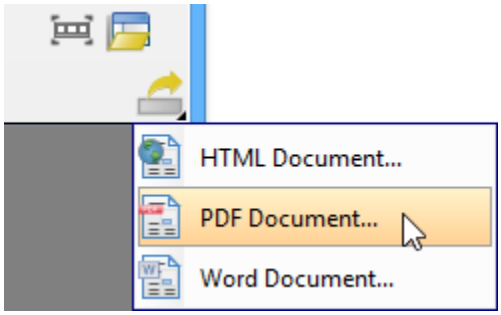
2. Enter the line number in the Input dialog box and click **OK**.



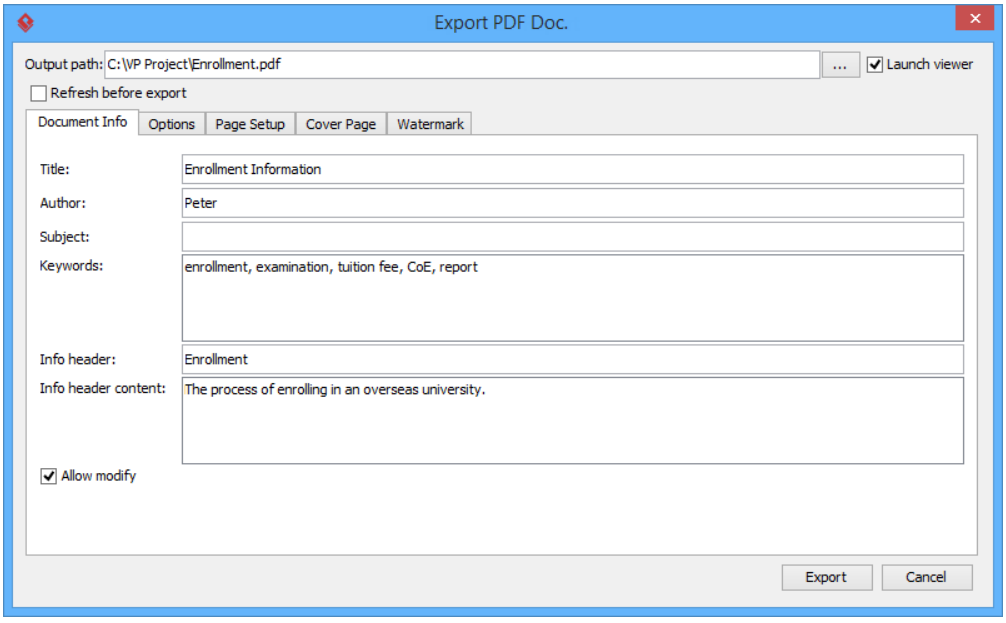
2.8 Exporting a Document

After you have customized your document template on document, you can export it into document. There are three types of document available for exporting: HTML, PDF and Word.

In document, click the **Export** button at the top right corner and select a type of document for exporting.



In the pop-up **Export [document type]** document window, specify output path and document info, and customize page setup, cover page and watermark.



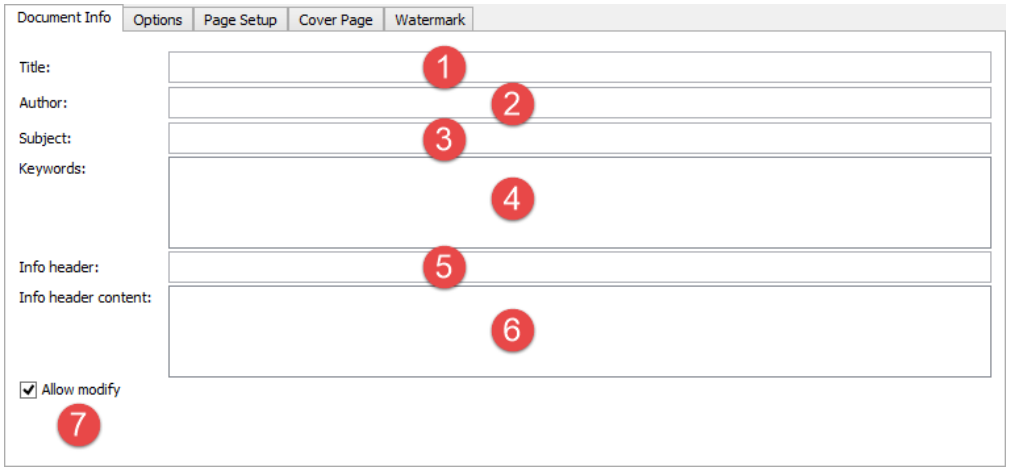
At last, click **Export** button.

2.8.1 The overview of export document window

No.	Name	Description
1	Output path	The output path of document to be generated.
2	Launch viewer	Check to open the document automatically after generation.
3	Refresh before export	Before proceed exporting, refresh the document content.
4	Document info	To define document information.
5	Options	To determine how data is to be printed in document by setting some of the configurable options.
6	Page Setup	To customize the layout of document.
7	Cover Page	To customize the first page of document.
8	Watermark	To customize the watermark on document.
9	Export	Confirm and export the document.
10	Cancel	Close the export document dialog box without exporting.

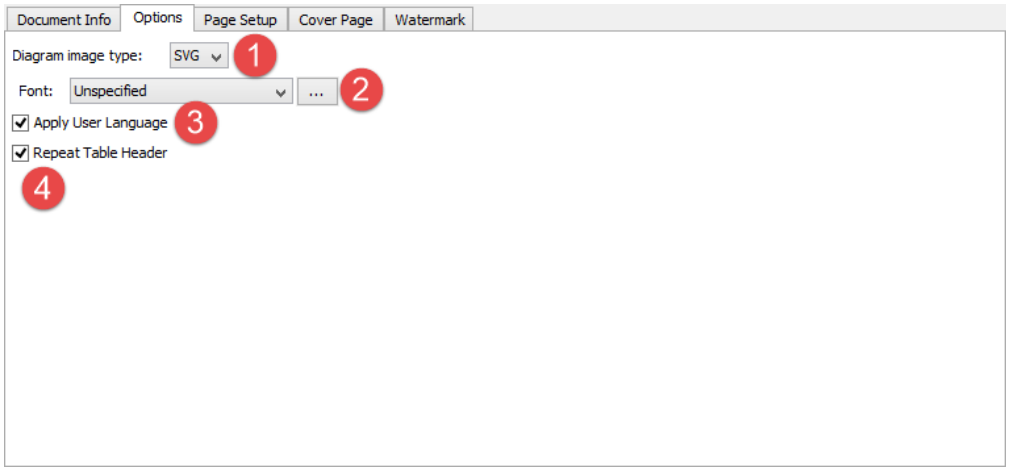
Notes	An additional **Content** tab is attached to **Export Word document** window.

2.8.2 The overview of Document Info



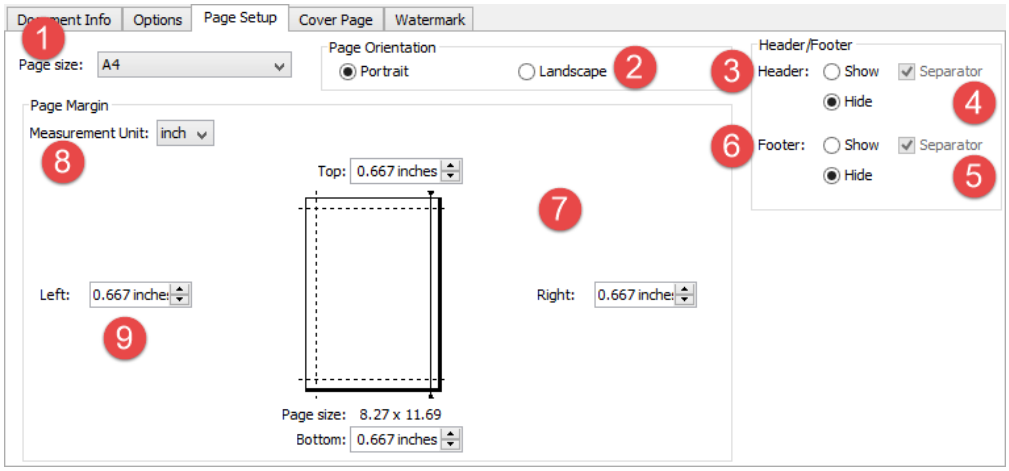
No.	Name	Description
1	Title	The title of document. This option is only available for exporting PDF document.
2	Author	The author of the document.
3	Subject	The subject of the document. This option is only available for exporting PDF and Word document.
4	Keywords	The keywords of the document.
5	Info header	The info header of the document. This option is only available for exporting PDF document.
6	Info header content	The info header content of the document. This option is only available for exporting PDF document.
7	Allow modify	Select to allow modification on the document. This option is only available for exporting PDF document.

2.8.3 The overview of Options Setup



No.	Name	Description
1	Diagram image type	Select the type of image format for image that appear in the exported document.
2	Font	Control the font of document text.
3	Apply User Language	By default, document content will be printed in English. By checking this option, it will follow the language setting chosen in global options.
4	Repeat Table Header	By checking this option, table header would be repeatedly printed when the table span multiple pages.

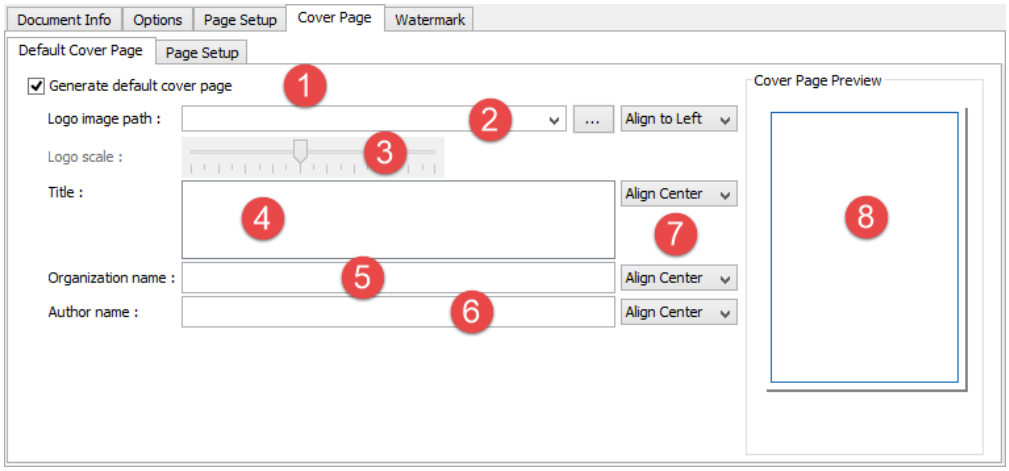
2.8.4 The overview of Page Setup



No.	Name	Description
1	Page size	To select the paper size of the exported document.
2	Page Orientation	This option is used to select the orientation of the document (portrait/ landscape). This option is only available to PDF and Word document.
3	Header	Check this option to insert header to the exported document. This option is only available to PDF and Word document.
4	Header Separator	Check this option to insert header separator to the exported document. This option is only available to PDF and Word document.
5	Footer Separator	Check this option to insert footer separator to the exported document. This option is only available to PDF and Word document.
6	Footer	Check this option to insert footer to the exported document. This option is only available to PDF and Word document.

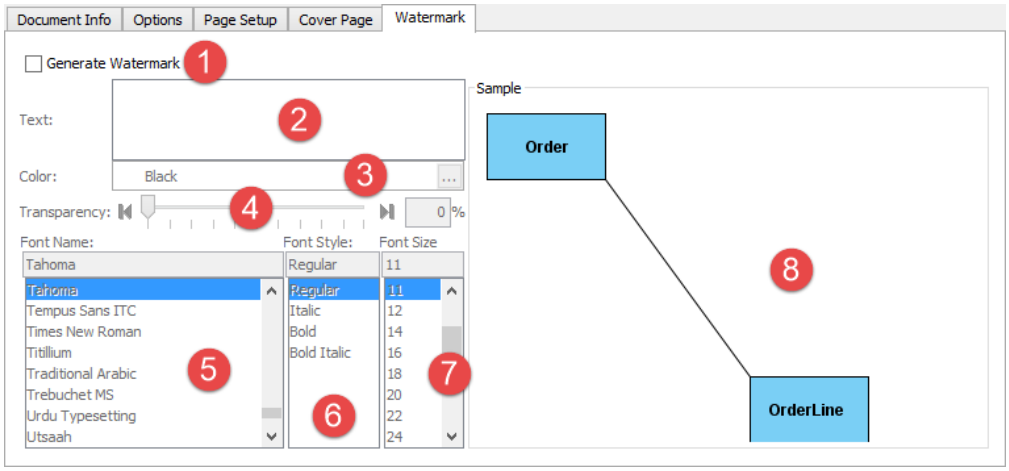
7	Page margin	To specify the page margins of the document: top, left, right and bottom. This option is only available to PDF and Word document.
8	Measurement Unit	To choose the measurement unit of page margin of the document: inch and cm. This option is only available to PDF and Word document.
9	Margin (Left/Top/Right/Bottom)	Specify the width of spaces between the content and the page border.

2.8.5 The overview of Cover Page



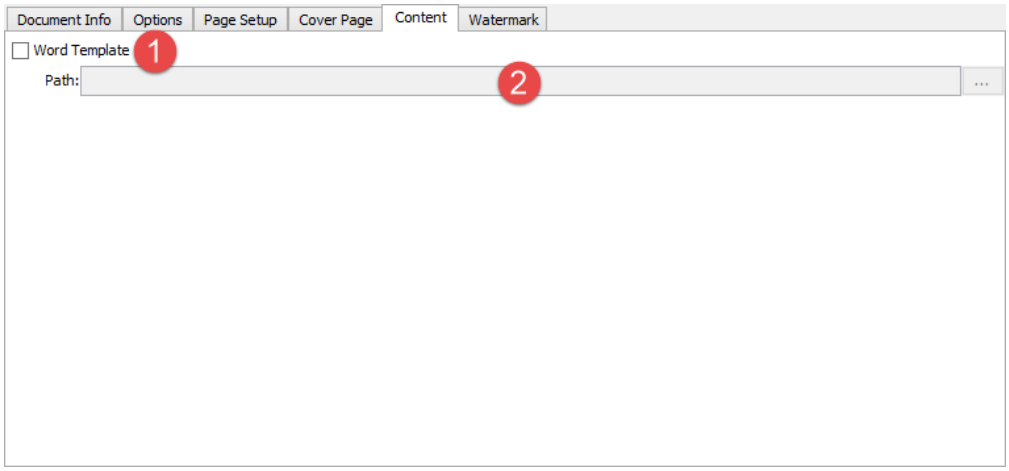
No.	Name	Description
1	Generate cover page	Check this option to generate a cover page to the document.
2	Logo image path	Insert an image to the cover page. You can specify the image's directory or select the directory by clicking the ... button.
3	Logo scale	Resize the inserted image by adjusting the slider.
4	Title	Specify the title of your document on cover page.
5	Organization name	Specify the organization name of your document on cover page.
6	Author name	Specify the author name on cover page.
7	Alignment	Control the position of content, whether to appear on the left, center or right hand side of the page.
8	Cover Page Preview	You can preview your cover page here.

2.8.6 The overview of Watermark



No.	Name	Description
1	Generate Watermark	Check this option to generate watermark on all diagrams of document.
2	Text	Specify the text will be used for watermark.
3	Color	Specify the color of text will be used for watermark by clicking the ... button.
4	Transparency	To change the background transparency for watermark, move the Transparency slider or specify the percentage of transparency directly.
5	Font Name	Select the font name for watermark.
6	Font Style	Select the font style for watermark.
7	Font Size	Select the font size for watermark.
8	Sample	Preview watermark here.

2.8.7 The overview of Content (Only for Word document)



No.	Name	Description
1	Word Template	Check this option to select a Word template for Word document.
2	Path	Specify the directory of word template by clicking the … button.

2.9 Managing Element Templates in Team Environment

If your team is using Visual Paradigm Online or Teamwork Server as collaborative modeling solution, you can share element templates among team members with the built-in management and synchronization features. Doing so allows the entire team to compose document based on a common set of element templates. Besides, this ensures the completeness of document when being viewed in any member's environment because all members have access to the same and most updated templates needed by the documents.

In server, element templates are stored in repository based. This means that all of your projects managed under the same repository have access to the same set of element templates. In this page, you will learn how to manage those element templates and share them among team members.

2.9.1 Managing element templates

Manage element templates is the process to create, edit or delete element templates stored in repository. Once you have made the desired changes in Visual Paradigm locally, you can synchronize the changes to server. Teammates can get the updated templates by synchronizing changes to server as well.

As said earlier, element templates are stored in repository based. Therefore, no matter which project you have opened, you are managing the same set of element templates.

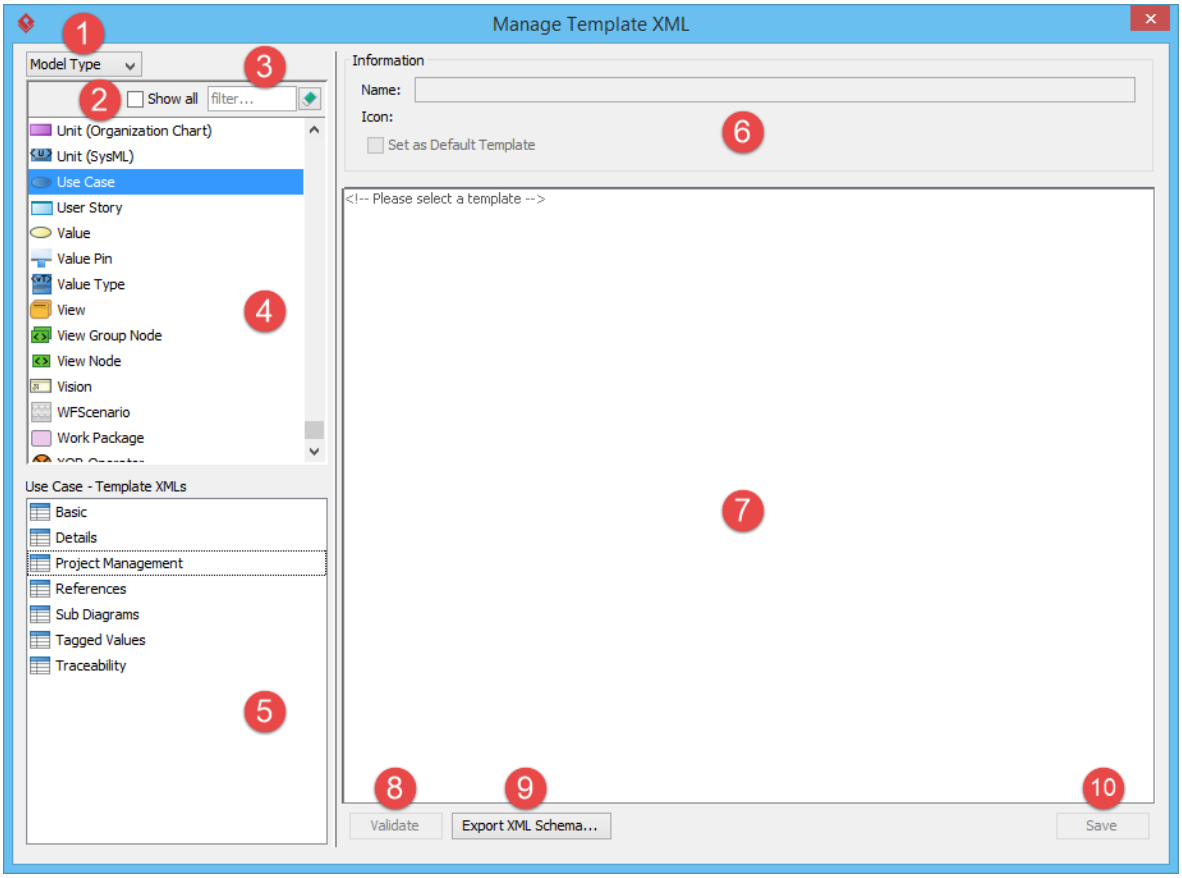
To manage element templates:

1. In Visual Paradigm, select **Tools > Doc. Composer > Manage Template XMLs…** from the toolbar. In order to access the management function, make sure you are opening a team project managed under either Visual Paradigm Online or Teamwork Server. Besides, make sure you are a team member and have been granted the right to **Change document template** in server.

You may need to contact your server administrator to confirm the permission settings made in server.

2. Now, you can manage element templates in the **Manage Template XML** window. Read the next section for details about what you can do in the **Manage Template XML** window.

2.9.1.1 Overview of Manage Template XML window



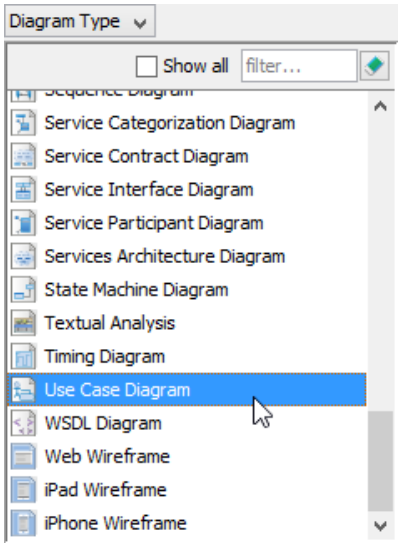
No.	Name	Description
1	Type of project data	Different templates are available for different project data. The drop-down menu there divide project data into four main types: **Project** - The entire project. You will see element templates mainly for querying details of diagrams in project. **Diagram Type** - The available types of diagram. (e.g. Use Case Diagram, Business Process Diagram) **Model Type** - The available types of model elements. (e.g Use Case, Class) **General** - Mainly for listing legacy or obsolete element templates. Normally you don't need to deal with templates under General section.

2	Show all	Visual Paradigm supports a large volume of model elements, but not all of them are well known or meaningful. By default, we hide away those elements that aren't popular. If you need to edit their templates you can check Show all to reveal them.
3	Filter	Filter the items to be displayed in element list.
4	List of elements	The elements that support template editing. If you have chosen to list Diagram Type (in Type of project data), you will see a list of diagram here. If you have chosen to list Model Type, you will see a list of model elements here.
5	List of templates	A list of element templates available for the element selected in the list of elements. Each type of project data has its own set of element templates. Take Use Case Diagram as example, you have templates like Basic, Details, Project Management, etc. With a different selection of element, a different list of element templates will be presented.
6	Background information	Background information of an element template. Note that you can only edit background information of a user-defined template, but not any built-in template. Here is a description of properties you can set: **Name**: The name of the element template, which is the name that shown under the Element Template Pane. **Icon**: An icon that best represent the layout of content that will be produced by using this template. **Set as Default Template**: Check this option if you want Doc. Composer to apply this template automatically when dragging elements directly from Diagram Navigator / Model Explorer onto document. **Template content**: Editor for programming the template.
7	XML Editor	Customize your template in the XML editor. Again, you can only edit a user-defined template, but not any built-in template.
8	Validate	Validate the XML against the built-in XML schema.
9	Export XML Schema	Export the XML schema (*.xsd) for validating the XML template content.
10	Save	Save the modifications made in XML editor.

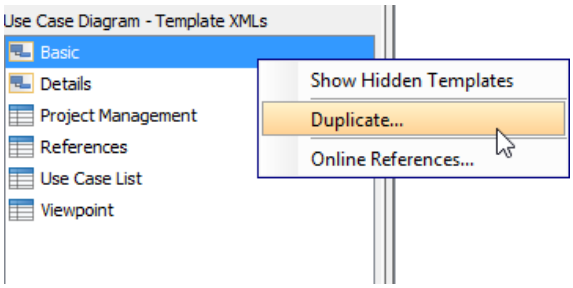
2.9.2 Creating a template

To simplify the programming of template, you are suggested to duplicate an existing template and start editing it, rather than do everything from scratch. Target on a template that gives the closest outcome to what you want to show in document. If you want to start from an empty document, select any templates.

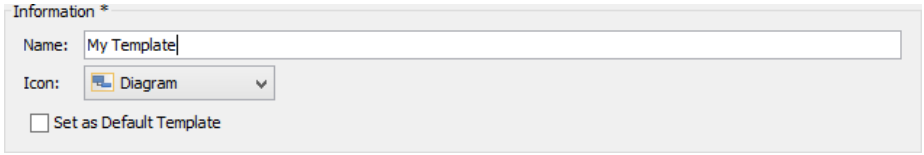
1. Select the type of element to create template. For example, select **Use Case Diagram** if you want to create a template to list specific shapes in use case diagram. You can select project / diagram type / model element type.



2. Right click on an element template and select **Duplicate...** from the popup menu.



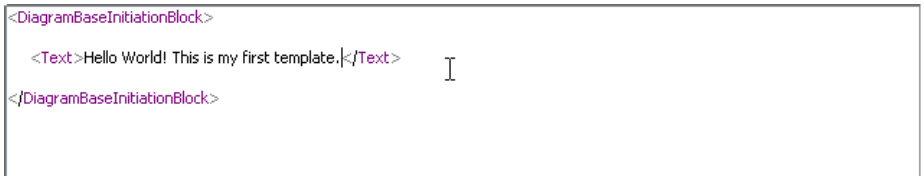
3. Enter the name of the new template.



4. Choose an icon that represents the presentation of content output with your template.

5. Check **Set as Default Template** if you want Doc. Composer to apply this template automatically when dragging elements directly from Diagram Navigator / Model Explorer onto document.

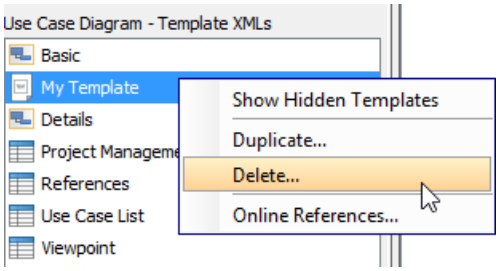
6. Compose the template in XML editor. If part of your template references content written in another element template, you can click + to add a reference to that template.



7. Click **Save** when finished editing. Now, you can use the new template in your document. You can also share it with teammates.

2.9.3 Deleting a template

Right click on an element template and select **Delete...** from the popup menu to remove it. Note that this action cannot be undone. Moreover, documents that used a deleted template in content will have the missing parts be replaced by <empty> tag(s). This may severely affect the completeness of your document so think twice before you delete a template. Make sure the template is not currently in-used by any document, or the documents that use the templates are not important anymore.



Notes	You can only delete a user-defined template.

2.9.4 Modifying a template

Click on the template in the list of template list and then modify it in XML editor. Click **Save** when finished editing. When finished, you can refresh your document to apply the changes.

2.9.5 Synchronizing element templates

Once you have finished editing element templates, you can synchronize the changes to server. Teammates can get the updated templates by synchronizing changes to server as well.

To synchronize changes to server manually, select **Tools > Doc. Composer > Sync. to VP Online/Teamwork Server** from the toolbar.

Note that your changes will be synchronized automatically when you perform commit.

2.10 Managing Styles in Team Environment

If your team is using Visual Paradigm Online or Teamwork Server as collaborative modeling solution, you can share styles configuration among team members with the built-in management and synchronization features. Doing so allows the entire team to compose document based on a common set of styles configuration. Besides, this ensures that documents are always up-to-date when being viewed in any member's environment because all members have access to the most updated styles configuration.

In server, styles configuration are stored in repository based. This means that all of your projects managed under the same repository have access to the same set of styles configuration. In this page, you will learn how to manage those styles and share them among team members.

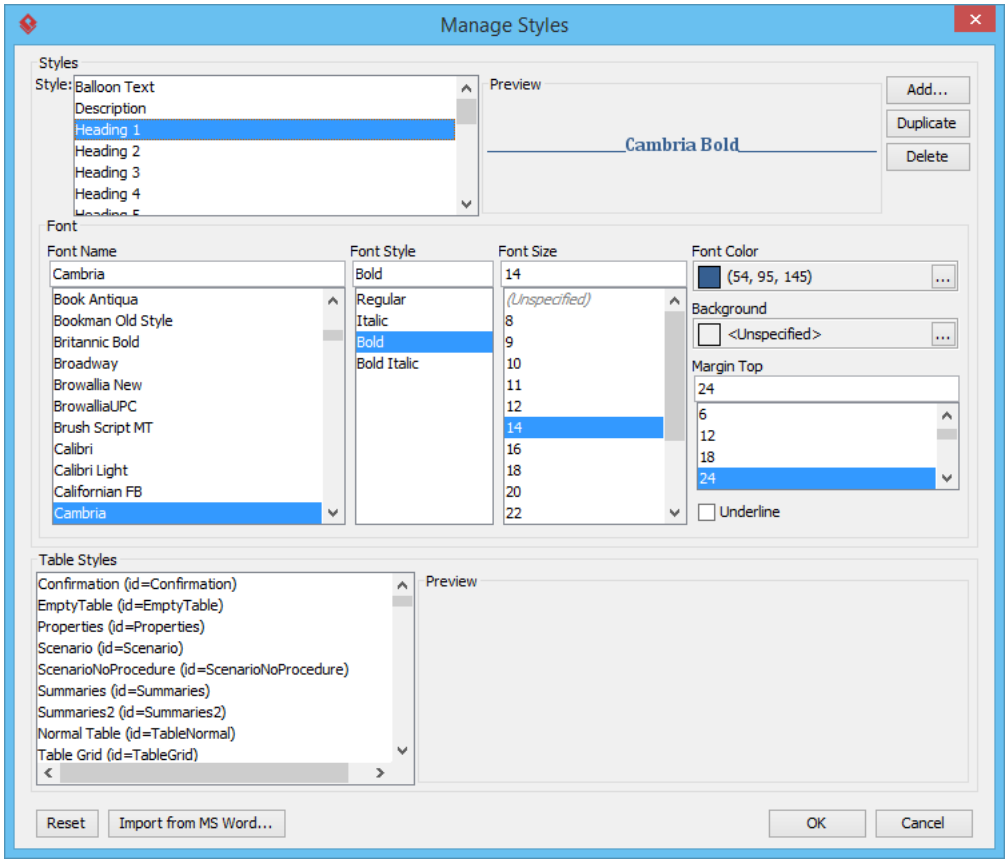
2.10.1 Managing Styles

Manage styles is the process to create, edit or delete styles configuration stored in repository. Once you have made the desired changes in Visual Paradigm locally, you can synchronize the changes to server. Teammates can get the updated styles configuration by synchronizing changes to server as well.

As said earlier, styles configuration are stored in repository based. Therefore, no matter which project you have opened, you are managing the same set of styles.

To manage styles:

1. In Visual Paradigm, select **Tools > Doc. Composer > Manage Styles...** from the toolbar. In order to access the management function, make sure you are opening a team project managed under either Visual Paradigm Online or Teamwork Server. Besides, make sure you are a team member and have been granted the right to **Change document template** in server. You may need to contact your server administrator to confirm the permission settings made in server.
2. Now, you can manage styles in the **Manage Styles** window.



2.10.2 Synchronizing styles configuration

Once you have finished editing styles configuration, you can synchronize the changes to server. Teammates can get the updated configuration by synchronizing changes to server as well.

To synchronize changes to server manually, select **Tools > Doc. Composer > Sync. to VP Online/Teamwork Server** from the toolbar.

Note that your changes will be synchronized automatically when you perform commit.

Chapter 3. Fill-in Doc

3.1 Introduction

Typically, a project documentation or report is a combination of background information like project goal, scope and constraints, and design details like use case details, database design, process design, etc. The Fill-in Doc mode of Doc. Composer is designed to help you “fill-in” the design details of your documentation. As an overview, Fill-in Doc works in this way:

1. You write your project documentation in Microsoft Word.
2. Place some special fields to the parts that requires the insertion of design details.
3. Load the document into Doc. Composer
4. Generate a document. Doc. Composer analyze the special fields in your document and replace them with actual project content.

3.2 Understanding Doc Base

A Doc Base is a semi-completed version of your project documentation or report. It contains only background information, possibly filled by you or your colleague. The design details are leave empty and be filled by Doc. Composer. As its name suggested, Doc Base provides a base for documentation production. You provide such a base to Doc. Composer and then Doc. Composer fill the empty parts for you by embedded model data extracted from your Visual Paradigm project into your documentation.

3.3 Understanding Doc Field

A Doc Field is a special piece of text within a Doc Base. Doc Fields will be replaced by your actual project content when being read by Doc. Composer during document generation. Here is an example of Doc Field:

```
${DIAGRAM, “List of use case diagrams”, “UseCaseDiagram”, LoopInProject, PROPERTY=name}
```

A Doc Field is written in this format: *${list_of_parameters}*. We will talk about the parameters in detail in coming sections.

Simply speaking, if you create a Word document, type the above text into the document, save the document as a Word file, import the file into Doc. Composer as a Doc Base and then generate a document from Doc. Composer, the generated document will look like this:

```
Use Case Diagram1, Use Case Diagram2, Use Case Diagram3
```

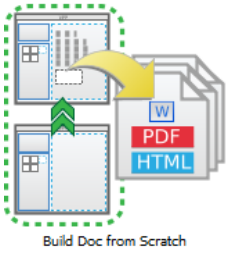
Here we assume that your project contains three use case diagrams, namely *Use Case Diagram1*, *Use Case Diagram2*, *Use Case Diagram3*.

This is how Doc Field basically works – You type the field text into your document at the places where you need to embed your project content, save the document, import it into Doc. Composer and let it produce a new document by replacing those fields with project content.

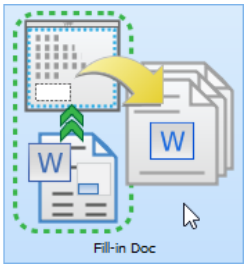
There are six kinds of fields. The following gives you the basic ideas of each fields, along with their required formats and capabilities. The detailed usage of these fields will be covered in next big sections.

3.4 Creating a Fill-in Doc

1. Select **Tools** > **Doc. Composer** from the toolbar.
2. Click **Fill-in Doc**. This shows the Specify Doc Base window.



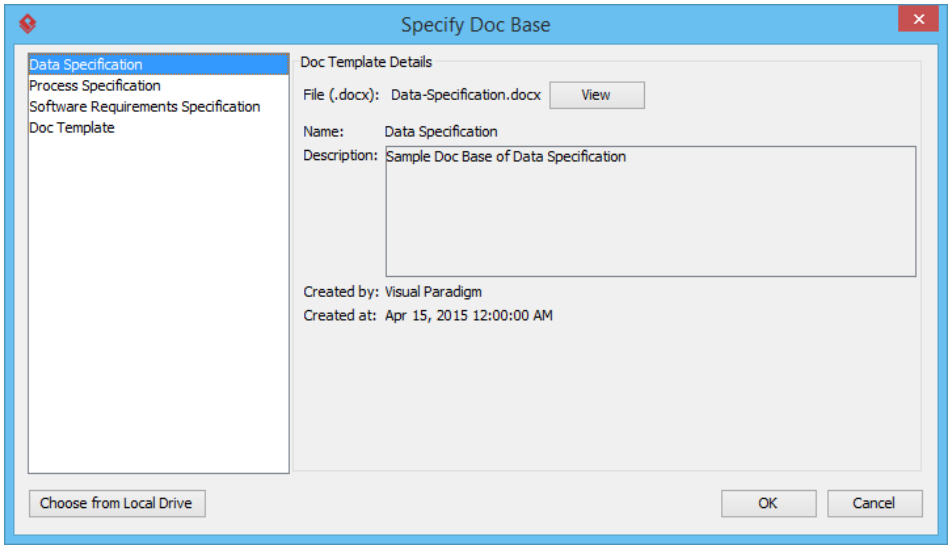
Build Doc from Scratch



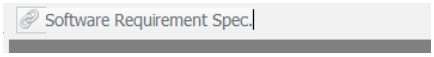
Fill-in Doc

3. Specify the Doc Base to use in document production. Doc base is a Word document file that contains both manually written content (e.g. Introduction, project scope, etc) and Doc Fields. If you are unclear about Doc Base and Doc Field, read the previous sections. There are two approaches from which Doc Base can be created from. One is to create from an external document file. If you take this approach, click **Choose from Local Drive** and then select the document file (*.docx). Another approach is to duplicate from an existing Doc Template. If you take this approach, select the Doc Template from the template list and click **OK**. You will then be prompted to save a copy of the Doc Template to your computer as Doc Base. Visual Paradigm provides three default templates for you to choose from. If your team uses Teamwork Server or Visual Paradigm Online, you can create your own set of templates and share them

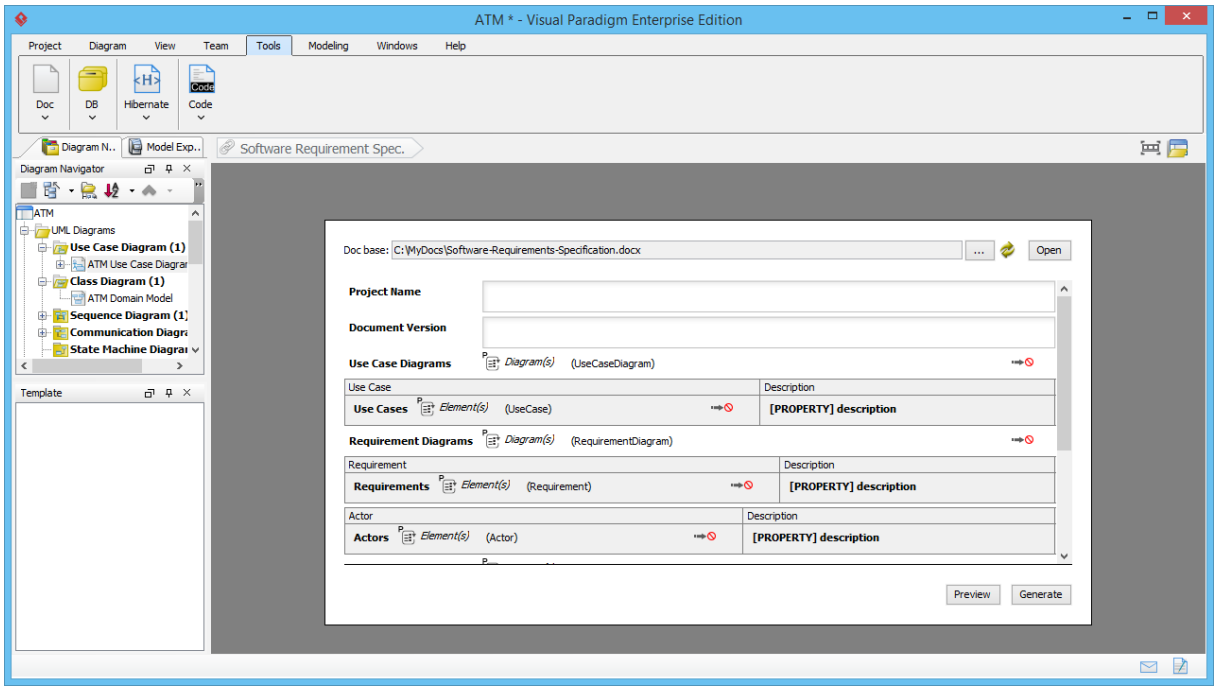
among the team.



4. If necessary, rename the document by double clicking on its name in breadcrumb and then typing in a new name.



5. Press the **Enter** key to confirm the naming.
6. Doc. Composer analyzes your Doc Base and presents the Doc Fields that exist in your document. Your screen should look like this:



To have more editing space, we recommend you to collapse the toolbar temporarily by double clicking on the **Tools** tab.

3.5 Touching-Up a Document

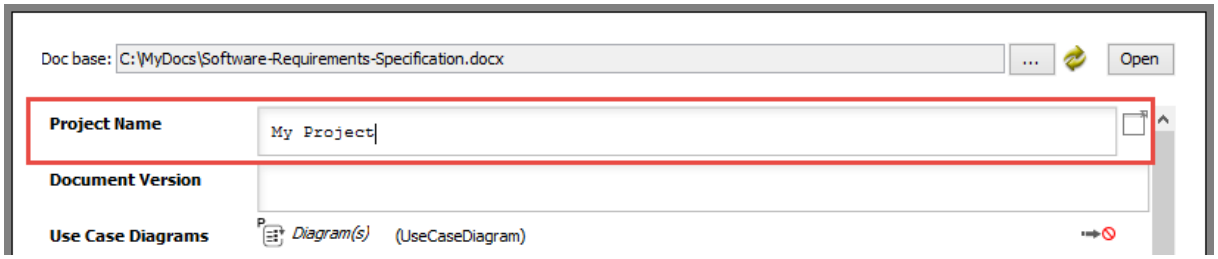
If your Doc Base contains any of the following kinds of Doc Fields, you have to touch-up the document in order to generate a document file from Doc. Composer

- Doc Field with Any as source
- Doc Field with One as source
- Doc Field with LoopInElement as source
- Doc Field with LoopInDiagram as source
- ${TEXT} field

To “touch-up” a document means to select diagram(s) or model element(s), or to enter the content required by Doc Fields in a document. For example, if in a Doc Base there exist a ${TEXT} like this:

```
<${TEXT, "Project Name"}>
```

You’ll have to provide the project name in Doc. Composer. Here is what you will see in Doc. Composer:

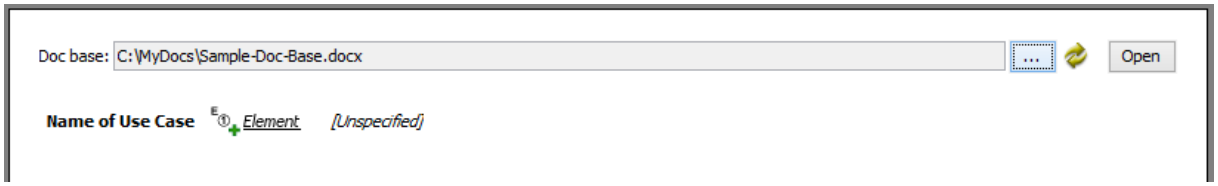


In the example above, we have entered *My Project* as the project name.

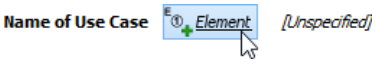
Another example would be the use of **${ELEMENT}** field, with *One* as source:

```
${ELEMENT, "Name of Use Case", "UseCase", One, PROPERTY=name}
```

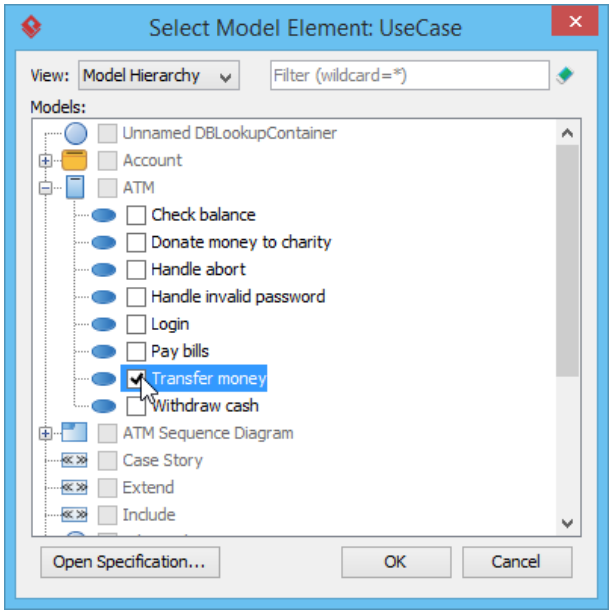
This example means to output the name of a use case to the document and such a use case shall be specified in Doc. Composer. Here is how the Doc. Composer will look like when applying a Doc Base that contains such an **${ELEMENT}** field.



What you need to do is to click on the **Element** link. Note that the title of this link varies depending on the type of field and source specified.



Then, select the desired element(s) and click **OK** to confirm.



3.6 Previewing a Document

If you want to take a quick look at the document file that will be generated with the applied Doc Base, click **Preview** at the bottom right corner of the document preview. A temporary document file will be opened for you to preview the outcome.

3.7 Generating a Document

When you are ready for producing the final document, click **Generate** at the bottom right corner of the document preview. Enter the filename of the document and confirm. A complete document file will then be generated.

3.8 The Doc Fields

3.8.1 ${PROJECT}

The **${PROJECT}** field is used to output a project property's value or based on a template written for the project.

Here is an example of **${PROJECT}**:

```
${PROJECT, PROPERTY=name}
```

Here is the sample output:

```
MyProject
```

This is the syntax of a **${PROJECT}** field:

```
${PROJECT,
  template_name | PROPERTY=property_name
}
```

This is a description of the various parts of a **${PROJECT}** field:

- PRJOECT is to indicate that this is a **${PROJECT}** field.
- ***template_name*** | PROPERTY=***property_name*** – The type of content to be extracted from the project and printed on the document.
    - ***template_name*** – Output content from the project based on the template ***template_name*** written for the project. For example, if you have a template *AllClassDiagrams* written for the project, by specifying *AllClassDiagrams* as ***template_name***, Doc. Composer will output content by following *AllClassDiagrams*.
    - PROPERTY=***property_name*** – Output a **specific** property (e.g. name) for the project. If you want to output multiple properties, try write a template and make reference to it by providing its name here.

3.8.2 ${DIAGRAM}

The **${DIAGRAM}** field is used to query diagram(s) from a project (or a specific place in a project), and to output content from the diagrams querying.

Here is an example of **${DIAGRAM}**:

```
${DIAGRAM, "List of Use Case Diagrams", "UseCaseDiagram", LoopInProject, PROPERTY=name}
```

Here is the sample output:

```
Use Case Diagram1, Use Case Diagram2, Use Case Diagram3
```

This is the syntax of a **${DIAGRAM}** field:

```
${DIAGRAM,
  field_name,
  ["diagram_type{, diagram_type...}", ]
  [SortBy="property{, property...}", ]

  One | Any | LoopInProject | LoopInElement,
  template_name | PROPERTY=property_name | ICON | IMAGE
}
```

This is a description of the various parts of a **${DIAGRAM}** field:

- DIAGRAM is to indicate that this is a **${DIAGRAM}** field.
- ***field_name*** is a short description of the field (e.g. “List of Use Case Diagrams”). In Doc. Composer, you can see the fields placed in an imported Doc Base. The fields are represented by the ***field_name*** typed here. ***field_name*** must be unique within a Doc Base. If there are two or more fields having the same field name, content may be produced wrongly.
- ***diagram_type*** indicates the type(s) of diagram that you want to query (e.g. “UseCaseDiagram”). If you want to query all types of diagram in a project, skip this parameter. If you want to query multiple types of diagram, enter their types respectively, separated by comma (e.g. “ClassDiagram,UseCaseDiagram”). Click here for the proper diagram types to use.

- SortBy is an optional part that supports the sorting of diagrams retrieved, based on the property or properties specified. If you want diagrams not to be sorted, use *SortBy=NoSort*.
- `One | Any | LoopInProject | LoopInElement` indicates the source from which to query diagrams.
    - `One` – Query a **specific** diagram in project. This option is often used when your document, or part of the document is written around a specific diagram. If you choose `One` here, you will have to select in Doc. Composer the diagram to query.
    - `Any` – Query a number of diagrams in project. If you choose `Any` here, you will have to select in Doc. Composer the diagram to query.
    - `LoopInProject` – Query all the diagrams in project.
    - `LoopInElement` – Query the sub-diagram(s) of a specific model element. If you choose `LoopInElement` here, you will have to select in Doc. Composer the model element from which to query sub-diagrams.
- ***`template_name`*** `| PROPERTY=`***`property_name`*** `| ICON | IMAGE` – The type of content to be extracted from the querying diagrams and printed on the document.
    - ***`template_name`*** – Output content from each of the querying diagrams, based on the template ***`template_name`*** written for the type of the querying diagram. For example, if you have a template *AllUseCases* written for Use Case Diagram, by specifying *AllUseCases* as ***`template_name`***, Doc. Composer will output content for each of the Use Case Diagrams by following *AllUseCases*.
    - `PROPERTY=`***`property_name`*** – Output a **specific** property (e.g. description) for each of the querying diagrams. If you want to output multiple properties, try write a template and make reference to it by providing its name here.
    - `ICON` – Output the icon for each of the querying diagrams.
    - `IMAGE` – Output the diagram image for each of the querying diagrams.

3.8.3 ${ELEMENT}

The **${ELEMENT}** field is used to query model element(s) or diagram element(s) from a project (or a specific place in a project), and to output content from the elements querying.

Here is an example of **${ELEMENT}**:

```
${ELEMENT, "List of Use Cases", UseCase, LoopInProject, PROPERTY=name}
```

Here is the sample output:

```
Use Case1, Use Case2
```

This is the syntax of a **${ELEMENT}** field:

```
${ELEMENT,
  field_name,
  ["element_type{,element_type...}", ]
  [SortBy="property{, property...}", ]
  One | Any | LoopInProject | LoopInElement | LoopInDiagram,
```

```
   template_name | PROPERTY=property_name | ICON
}
```

This is a description of the various parts of a **${ELEMENT}** field:

- `ELEMENT` is to indicate that this is a **${ELEMENT}** field.
- ***field_name*** is a short description of the field (e.g. “List of Use Cases”). In Doc. Composer, you can see the fields placed in an imported Doc Base. The fields are represented by the ***field_name*** typed here. ***field_name*** must be unique within a Doc Base. If there are two or more fields having the same field name, content may be produced wrongly.
- ***element_type*** indicates the type(s) of model/diagram element that you want to query (e.g. “UseCase”). If you want to query all types of model element in a project, skip this parameter. If you want to query multiple types of model element, enter their types respectively, separated by comma (e.g. “Actor,UseCase”).
- SortBy is an optional part that supports the sorting of elements retrieved, based on the property or properties specified. If you want elements not to be sorted, use *SortBy=NoSort*.
- `One | Any | LoopInProject | LoopInElement | LoopInDiagram` indicates the source from which to query elements.
    - `One` – Query a **specific** model element in project. This option is often used when your document, or part of the document is written around a specific model element. If you choose `One` here, you will have to select in Doc. Composer the model element to query.
    - `Any` – Query a number of model elements in project. If you choose `Any` here, you will have to select in Doc. Composer the model element to query.
    - `LoopInProject` – Query all the model elements in project.
    - `LoopInElement` – Query the child elements of a specific model element. If you choose `LoopInElement` here, you will have to select in Doc. Composer the model element from which to query child elements.
    - `LoopInDiagram` – Query the diagram elements from a specific diagram. If you choose `LoopInDiagram` here, you will have to select in Doc. Composer the diagram from which to query diagram elements.
- ***template_name*** | `PROPERTY=`***property_name*** | `ICON` | `IMAGE` – The type of content to be extracted from the querying elements and printed on the document.
    - ***template_name*** – Output content from each of the querying elements, based on the template ***template_name*** written for the type of the querying element. For example, if you have a template *UseCaseInfo* written for Use Case, by specifying *UseCaseInfo* as ***template_name***, Doc. Composer will output content for each of the Use Cases by following *UseCaseInfo*.
    - `PROPERTY=`***property_name*** – Output a **specific** property (e.g. description) for each of the querying elements. If you want to output multiple properties, try write a template and make reference to it by providing its name here.
    - `ICON` – Output the icon for each of the querying elements.

3.8.4 ${ICON}

When you are working with a table, you can place the **${ICON}** field in a table cell to let Doc. Composer replace it with the icon image of the querying diagram or element.

Note that **${ICON}** can only be used in a table cell.

Here is an example of **${ICON}**:

```
${ICON}
```

Here is the sample output:

This is the syntax of a **${ICON}** field:

```
${ICON}
```

3.8.5 ${IMAGE}

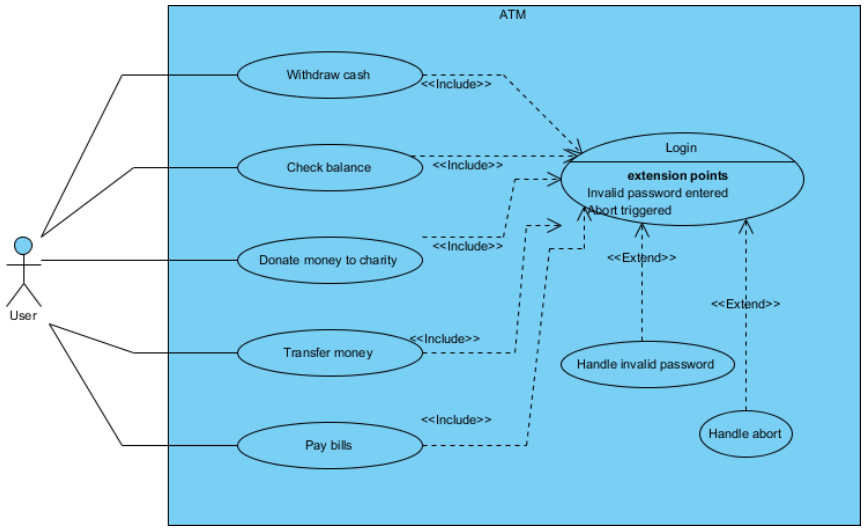
When you are working with a table, you can place the **${IMAGE}** field in a table cell to let Doc. Composer replace it with the diagram image of the querying diagram.

Note that **${IMAGE}** can only be used in a table cell.

Here is an example of **${IMAGE}**:

```
${IMAGE}
```

Here is the sample output:



This is the syntax of an **${IMAGE}** field:

```
${IMAGE}
```

3.8.6 ${PROPERTY}

When you are working with a table, you can place the **${PROPERTY}** field in a table cell to let Doc. Composer replace it with the property value of the querying diagram or element.

Note that **${PROPERTY}** can only be used in a table cell.

Here is an example of **${PROPERTY}**:

```
${PROPERTY}
```

Here is the sample output:

```
This is the description of use case.
```

This is the syntax of a **${ICON}** field:

```
${PROPERTY}
```

3.8.7 ${TEXT}

The **${TEXT}** field is used when you need to include information that should be or can only be provided when generating document. A typical usage of **${TEXT}** is to request for project name.

In Doc. Composer, **${TEXT}** are represented as text fields. User can enter the value required by the **${TEXT}** field. When generating document, those **${TEXT}** will be replaced by the text entered.

Here is an example of **${TEXT}**:

```
${TEXT, "Project name"}
```

Here is the sample output:

```
Online Banking
```

This is the syntax of a **${TEXT}** field:

```
${TEXT,
  field_name
}
```

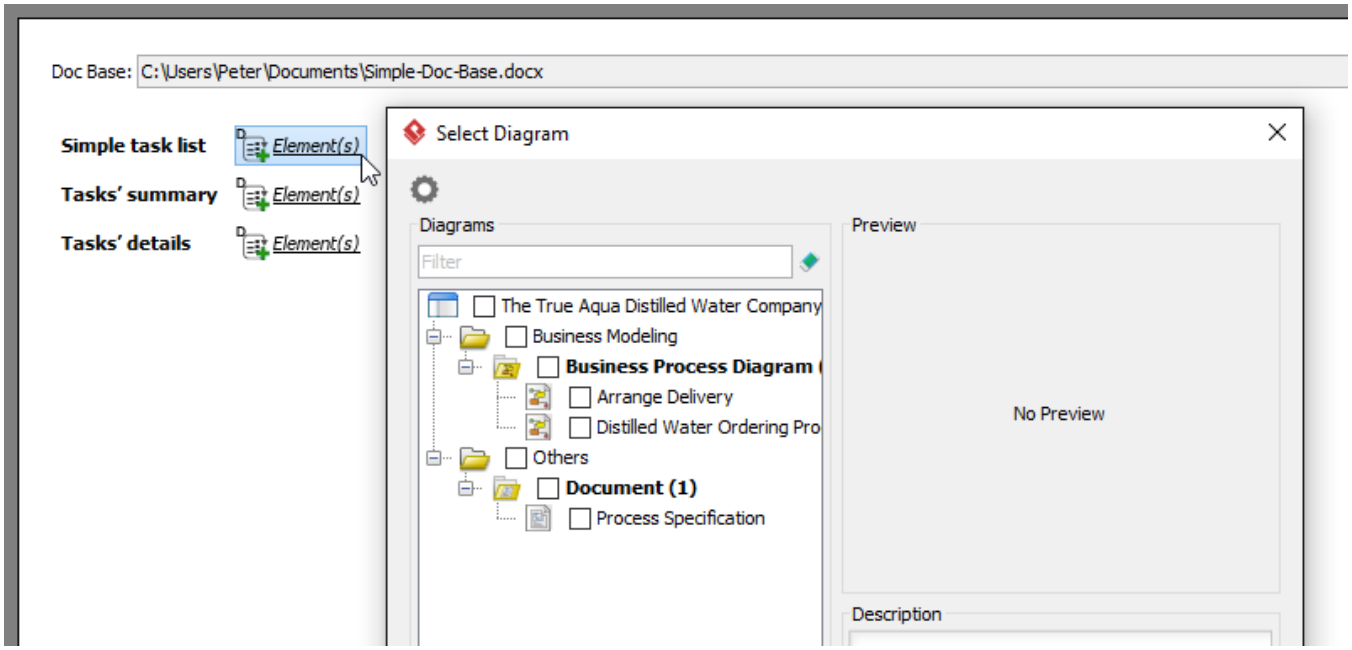
This is a description of the various parts of a **${TEXT}** field:

- `TEXT` is to indicate that this is a **${TEXT}** field.
- ***`field_name`*** is a short description of the field (e.g. "Project name"). It can also be used as a reminder to provide certain kind of information (e.g. "Please enter the project name.").

field_name must be unique within a Doc Base. If there are two or more fields having the same field name, content may be produced wrongly.

3.8.8 Reusability of Doc Fields

A Doc Base may contains many Doc Fields. If the Doc Fields use LoopInElement and LoopInDiagram, which require the selection of source in Doc. Composer, you (or the person who will generate document with the Doc Base) will then need to select model elements and diagrams again and again in Doc. Composer. This is not just time consuming but also error prone.



In order to solve this problem, you can reuse Doc Fields throughout a document. When you need to query data from a source that was expected by an earlier Doc Field, write the new Doc Field by using the same name as the previous one, like this:

${ELEMENT, "**Tasks in Main BPD**", "BPTask", LoopInDiagram, PROPERTY=name}

…

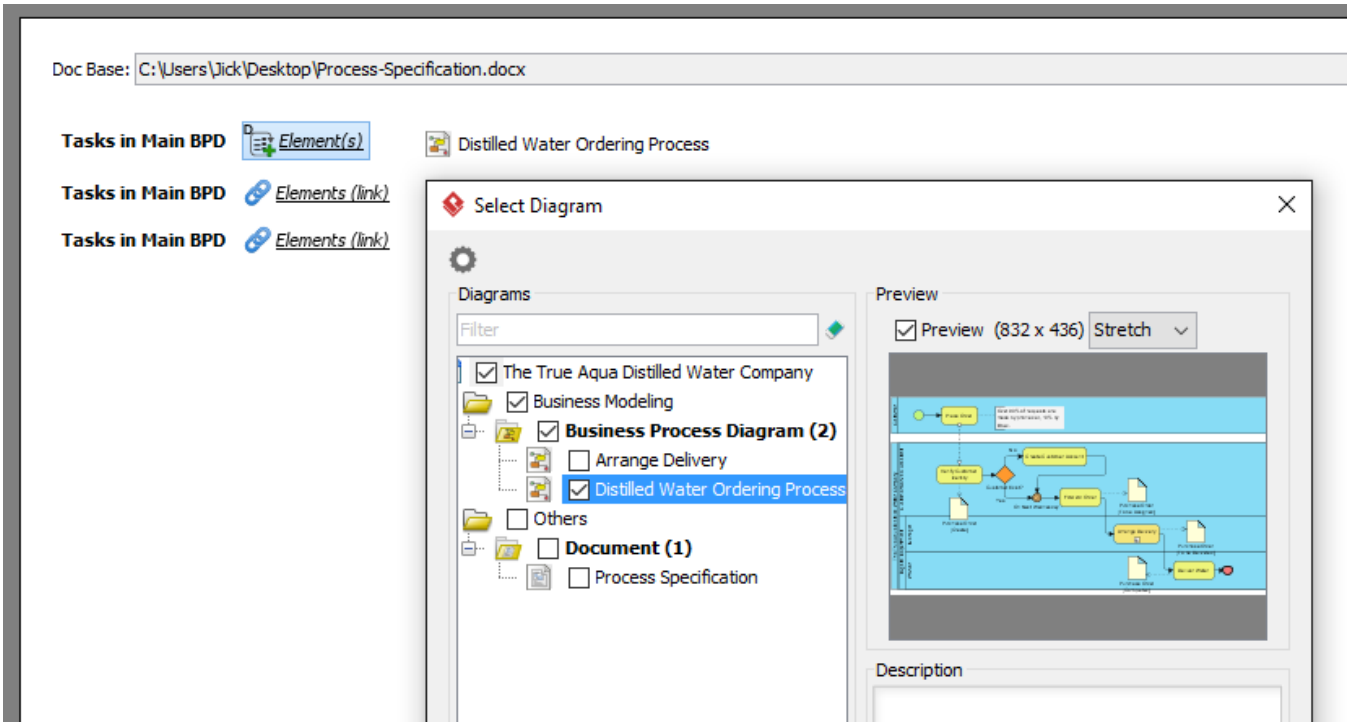
${ELEMENT, "**Tasks in Main BPD**", "BPSubProcess", LoopInDiagram, Basic}

…

${ELEMENT, "**Tasks in Main BPD**", "BPTask", LoopInDiagram, Details}

This three Doc Fields mean that I want to display the name of all tasks from a specific diagram (to be chosen in Doc. Composer). Then, I want to display the basic information of sub-processes on the same diagram. Finally, I want to display the details of tasks from again the same diagram.

By reusing Doc Fields, you just need to make the selection of source once. Subsequent Doc Fields will just apply the same selection. The key is to use same names for different Doc Fields.



Note that the same source must be supplied in order for the reusability to work. By “source”, we are referring to the argument of a Doc Field that indicates the source from which to query elements/diagrams, such as One | Any | LoopInProject | LoopInElement | LoopInDiagram.

3.9 Querying Diagrams

If you want to retrieve a diagram or diagrams, and to output content like the diagram’s image, name or a list of containing diagram elements, etc., read this section to learn the ways to retrieve diagrams.

3.9.1 Querying Diagrams in Project

If you want to output the image or any detail of **all the diagrams in project**, write a **${DIAGRAM}** field in your Word document with **LoopInProject** specified as diagram source. Here are several examples of such a **${DIAGRAM}** field:

```
${DIAGRAM, “Name of ALL Use Case Diagrams”, “UseCaseDiagram”, LoopInProject,
PROPERTY=name}

${DIAGRAM, “Description of ALL diagrams”, LoopInProject, PROPERTY=description}

${DIAGRAM, “Details of ALL Use Case Diagram and class diagram”,
“UseCaseDiagram,ClassDiagram”, LoopInProject, MyTemplate}
```

In the first example, the name of all Use Case Diagrams in the project will be output. Note that “Name of ALL Use Case Diagrams” is the field name, which is a required and unique value for identifying this field.

In the second example, the description of all the diagrams in the project will be output.

In the third example, content will be output for each of the Use Case Diagrams and Class Diagrams in the project, based on the template *MyTemplate*.

3.9.2 Querying Selected Diagrams in Project

The diagrams in your project may be created for different contexts or about different problem domains. When you write a documentation, you may want to focus on a specific context at a time, which requires the insertion of design specification for that specific context. In that case, you will want to query a selected set of diagrams in your project, instead of querying all diagrams.

If you want to output the image or any detail of **selected diagrams in project**, write a **${DIAGRAM}** field in your Word document with **Any** specified as diagram source. Here are several examples of such a **${DIAGRAM}** field:

```
${DIAGRAM, "Use Case Diagrams (Admin)", "UseCaseDiagram", Any, PROPERTY=name}

${DIAGRAM, "Diagram Images", Any, IMAGE}

${DIAGRAM, "Diagram Images", Any, MyTemplate}
```

In the first example, the name of selected Use Case Diagrams in the project will be output. Note that "Use Case Diagrams (Admin)" is the field name, which is a required and unique value for identifying this field.

In the second example, the image of selected diagrams in the project will be output.

In the third example, content will be output for each of the selected diagrams in the project, based on the template *MyTemplate*.

When you pick-up a Doc Base with such a **${DIAGRAM}** field in it, you can select the diagrams to query in Doc. Composer.

3.9.3 Querying Specific Diagram in Project

Let's say you have created multiple Use Case Diagrams for multiple sub-systems. When you write a documentation for a specific sub-system, you may want to insert the design specification related to that specific sub-system. In that case, you will want to query a specific Use Case Diagram in your project.

If you want to output the image or any detail of a **specific diagram in project**, write a **${DIAGRAM}** field in your Word document with **One** specified as diagram source. Here are several examples of such a **${DIAGRAM}** field:

```
${DIAGRAM, "ATM Overview", "UseCaseDiagram", One, PROPERTY=description}

${DIAGRAM, "Online Photo Album", One, IMAGE}

${DIAGRAM, "CS System – Use Cases", One, Children}
```

In the first example, the description of selected Use Case Diagram in the project will be output. Note that "ATM Overview" is the field name, which is a required and unique value for identifying this field.

In the second example, the image of selected diagram in the project will be output.

In the third example, content will be output for the selected diagram in the project, based on the template *Children*.

When you pick-up a Doc Base with such a **${DIAGRAM}** field in it, you can select the diagram to query in Doc. Composer.

3.9.4 Querying Sub-Diagrams from Specific Model Element

Let's say you have created several use cases and, for each use case, there are multiple sub-Business Process Diagrams that describe the business workflow in which the use case might happen. When you write a use case report, you may want to present the details of the Business Process Diagrams of a chosen use case. In that case, you will want to query the sub-diagrams of a selected use case.

If you want to output the image or any detail of **sub-diagrams from a specific model element**, write a **${DIAGRAM}** field in your Word document with **LoopInElement** specified as diagram source. Here are several examples of such a **${DIAGRAM}** field:

```
${DIAGRAM, "Related Business Workflow", "BusinessProcessDIagram", LoopInElement,
PROPERTY=name}

${DIAGRAM, "Images of Sub-Diagrams", LoopInElement, IMAGE}

${DIAGRAM, "Sub Diagram Details", LoopInElement, MySubDiagrams}
```

In the first example, the name of sub-Business Process Diagrams of the selected model element will be output. Note that "Related Business Workflow" is the field name, which is a required and unique value for identifying this field.

In the second example, the image of sub-diagrams of a selected model element will be output.

In the third example, content will be output for each of the sub-diagrams of a selected model element, based on the template *MySubDiagrams*.

When you pick-up a Doc Base with such a **${DIAGRAM}** field in it, you can select the model element to query in Doc. Composer.

3.10 Querying Model Elements

If you want to retrieve a model element or elements, and to output content like the element's name, description, or a list of member elements (e.g. attributes of class, columns of entity), etc., read this section to learn the ways to retrieve model elements.

3.10.1 Querying Model Elements in Project

If you want to output any detail of **all the model elements in project**, write an **${ELEMENT}** field in your Word document with **LoopInProject** specified as element source. Here are several examples of such an **${ELEMENT}** field:

```
${ELEMENT, "Name of ALL Use Cases", "UseCase", LoopInProject, PROPERTY=name}

${ELEMENT, "Description of ALL model elements", LoopInProject, PROPERTY=description}
```

```
${ELEMENT, "Details of ALL Use Cases and Business Processes",
"UseCase,BPTask,BPSubProcess", LoopInProject, MyTemplate}
```

In the first example, the name of all use cases in the project will be output. Note that "Name of ALL Use Cases" is the field name, which is a required and unique value for identifying this field.

In the second example, the description of all the model elements in the project will be output.

In the third example, content will be output for each of the use cases, BPMN tasks and sub-processes in the project, based on the template *MyTemplate*.

3.10.2 Querying Selected Model Elements in Project

The model elements in your project may be created for different contexts or purposes. When you write a documentation, you may want to focus on some of them at a time. In that case, you will want to query a selected set of model elements in your project, instead of querying all elements.

If you want to output any detail of **selected model elements in project**, write an **${ELEMENT}** field in your Word document with **Any** specified as element source. Here are several examples of such an **${ELEMENT}** field:

```
${ELEMENT, "Use Cases", "UseCase", Any, PROPERTY=name}

${ELEMENT, "Elements' Detail", Any, MyTemplate}
```

In the first example, the name of selected use cases will be output. Note that "Use Cases" is the field name, which is a required and unique value for identifying this field.

In the second example, content will be output for each of the selected model elements, based on the template *MyTemplate*.

When you pick-up a Doc Base with such an **${ELEMENT}** field in it, you can select the model elements to query in Doc. Composer.

3.10.3 Querying Specific Model Element in Project

If you need to write a document for a specific model element, like a use case report, you may need to insert the details of a specific use case into your document. In that case, you will want to query a specific model element in your project, instead of querying all model elements.

If you want to output any detail of a **specific model element in project**, write an **${ELEMENT}** field in your Word document with **One** specified as element source. Here are several examples of such an **${ELEMENT}** field:

```
${ELEMENT, "Desc of Use Case", "UseCase", One, PROPERTY=description}

${ELEMENT, "List of Class Members", "Class", One, Children}
```

In the first example, the description of selected use case will be output. Note that "Desc of Use Case" is the field name, which is a required and unique value for identifying this field.

In the second example, content will be output for the selected class, based on the template *Children*.

When you pick-up a Doc Base with such a **${DIAGRAM}** field in it, you can select the model element to query in Doc. Composer.

3.10.4 Querying Model Elements from Specific Model Element

If you need to write a document for a specific model element by detailing its children elements, like a business responsibility report that details the tasks contained by specific pool, or a use case report that details the use cases contained by a system, you will want to query the children elements of a selected model element.

If you want to output any detail of **model elements from a specific model element**, write an **${ELEMENT}** field in your Word document with **LoopInElement** specified as element source. Here are several examples of such an **${ELEMENT}** field:

```
${ELEMENT, "List of System Use Cases", "UseCase", LoopInElement, PROPERTY=name}

${ELEMENT, "List of Elements (Any type)", LoopInElement, PROPERTY=name}
```

In the first example, the name of the children use cases of selected element will be output. Note that "List of System Use Cases" is the field name, which is a required and unique value for identifying this field.

In the second example, the name of all children elements of selected element will be output.

When you pick-up a Doc Base with such an **${ELEMENT}** field in it, you can select the model element to query in Doc. Composer.

3.11 Querying Diagram Elements

If you want to retrieve a shape or shapes, and to output content like the shape's name, description, or a list of member elements (e.g. attributes of class, columns of entity), etc., read this section to learn the ways to retrieve shapes (i.e. diagram elements).

3.11.1 Querying Diagram Elements from Specific Diagram

If you need to write a document for a specific diagram, like an ERD report, you may need to insert the details of the containing shapes into your document. In that case, you will want to query diagram elements in a diagram.

If you want to output any detail of a **diagram elements in a diagram**, write an **${ELEMENT}** field in your Word document with **LoopInDiagram** specified as element source. Here are several examples of such an **${ELEMENT}** field:

```
${ELEMENT, "Tables in ERD", "DBTable", LoopInDiagarm, PROPERTY=name}

${ELEMENT, "List of Classes", "Class", LoopInDiagram, Details}
```

In the first example, the name of entities on a selected ERD will be output. Note that "Tables in ERD" is the field name, which is a required and unique value for identifying this field.

In the second example, content will be output for classes in the selected diagram, based on the template *Children*.

When you pick-up a Doc Base with such an **${ELEMENT}** field in it, you can select the diagram to query in Doc. Composer.

3.12 Using Custom Text

If you want to request the user of Doc Base to fill-in certain piece of content himself/herself, write a **${TEXT}** field. A **${TEXT}** field is a placeholder of content that can only be provided when generating a document, such as project name or author name. Here is an example of a **${TEXT}** field:

```
${TEXT, "Project Name"}
```

3.13 Working with Table

You can present project data neatly with the use of table. In this section we will introduce the various kinds of table you can create in a Doc Base, and explain how to create such tables by writing Doc Fields. We assume that you have the basic knowledge of Doc Field. If you don't, please read the previous sections.

Let's begin by studying the following example, which consists of a table in a Doc Base, with an ${ELEMENT} Doc Field placed in the second row of the table.

Use Cases in Project
${ELEMENT, "List of Use Cases", UseCase, LoopInProject, PROPERTY=name}

Suppose the Doc Base is applied on a project that contains the design specification of an ATM. Here is the sample outcome:

Use Cases in Project
Withdraw Cash
Transfer Cash
Donate Money
Pay Bills

Based on the outcome, you can see:

- The row that contains the Doc Field replicate itself to list out all the elements queried.
- In each row, the name of use case is output, which is the result of using **PROPERTY=name** in the Doc Field.

While this example output the name of use case, you can output complex content with the use of an element template. You just need to replace **PROPERTY=name** with the name of that template.

The example above is perhaps a bit simple. Let's extend it to make it a bit more complicated and closer to practical usage. Let's study this table:

Use Cases in Project	ID	Description
${ELEMENT, "List of Use Cases", UseCase, LoopInProject, PROPERTY=name}	**${PROPERTY, "userID"}**	${PROPERTY, "description"} ***(to be confirmed)***

We have added two more columns into the table, one for displaying the ID of use cases and another for displaying the description of use cases. Again, if we apply the Doc Base on an ATM project, here is the sample outcome:

Use Cases in Project	ID	Description
Withdraw Cash	**UC01**	Get cash from the ATM. ***(to be confirmed)***
Transfer Cash	**UC02**	Transfer cash from one account to another. ***(to be confirmed)***
Donate Money	**UC03**	Donate money to a chosen charity. ***(to be confirmed)***
Pay Bills	**UC04**	Settle bills. ***(to be confirmed)***

Based on the outcome, you can see:

- In order to output multiple properties of an element, add extra columns into the table and use ${PROPERTY} to output those properties.
- You can format table content, like the green text you see above.
- You can add your own text into table cells.

You may also want to present the icons of querying element. The following example gives you some ideas how to achieve it.

Model Elements in a Class Diagram	Or if you want a column of icons
${ELEMENT, "List of Model Elements", , LoopInDiagram, ICON} ${PROPERTY, "name"}	${ICON}

Here is the sample output when applying the above Doc Base on an ATM project, with a class diagram chosen to be the source of elements to query.

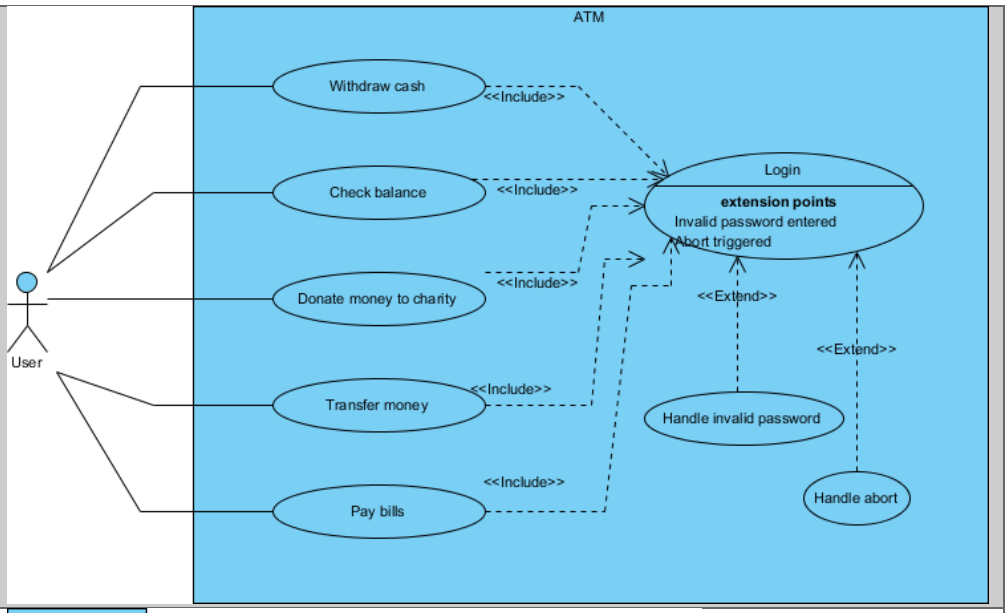
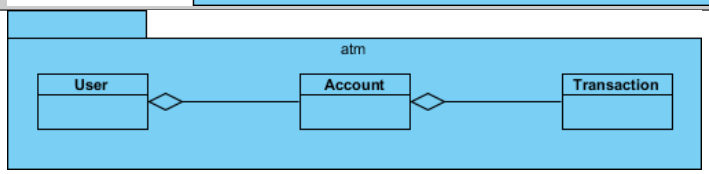
Model Elements in a Class Diagram	Or if you want a column of icons
Account	
atm	
Transaction	
User	

You may also output diagram images using **${IMAGE}**. Here is an example:

All Diagrams in Project	Diagram Image
${DIAGRAM, "All Diagrams", , LoopInProject, PROPERTY=name}	${IMAGE}

Here is the sample output when applying the above Doc Base on an ATM project.

All Diagrams in Project	Diagram Image

ATM Use Case Model	
Domain Class Model	

Notes	${PROPERTY}, ${ICON} and ${IMAGE} can only be used inside a table cell.

3.14 Managing Doc Templates in Team Environment

If your team is using Visual Paradigm Online or Teamwork Server as collaborative modeling solution, you can share Doc Templates among team members with the built-in management and synchronization features. Doing so allows the entire team to compose document based on a common set of Doc Templates. Besides, this ensures that documents are always up-to-date when being viewed in any member's environment because all members have access to the most updated templates.

In server, Doc Templates are stored in repository based. This means that all of your projects managed under the same repository have access to the same set of Doc Templates. In this page, you will learn how to manage those Doc Templates and share them among team members.

3.14.1 Managing Doc Templates

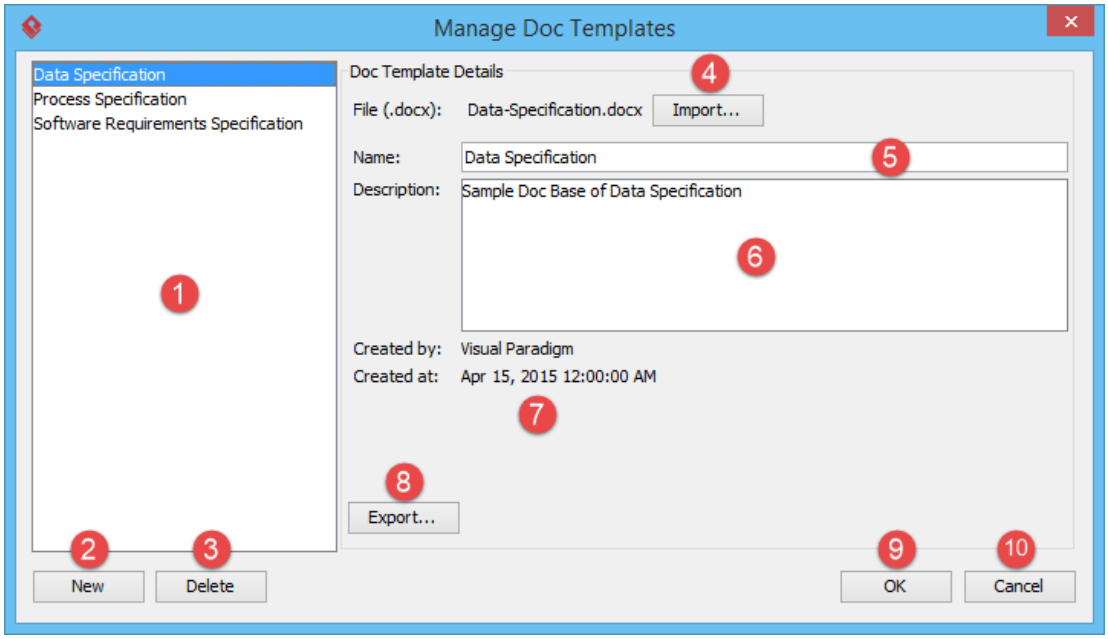
Manage Doc Templates is the process to create, edit or delete Doc Templates stored in repository. Once you have made the desired changes in Visual Paradigm locally, you can synchronize the changes to server. Teammates can get the updated templates by synchronizing changes to server as well.

As said earlier, Doc Templates are stored in repository based. Therefore, no matter which project you have opened, you are managing the same set of Doc Templates.

To manage Doc Templates:

1. In Visual Paradigm, select **Tools > Doc. Composer > Manage Doc Templates...** from the toolbar. In order to access the management function, make sure you are opening a team project managed under either Visual Paradigm Online or Teamwork Server. Besides, make sure you are a team member and have been granted the right to **Change document template** in server. You may need to contact your server administrator to confirm the permission settings made in server.
2. Now, you can manage Doc Templates in the **Manage Doc Templates** window. Read the next section for details about what you can do in the **Manage Doc Templates** window.

3.14.1.1 Overview of Manage Doc Template window



No.	Name	Description
1	List of Doc Templates	List of Doc Template available for use as Doc Base in document generation.
2	New	Create a Doc Template.
3	Delete	Delete the Doc Template.
4	Import	Import a Word document (.docx) file for this Doc Template. Note that the second time you import a .docx file into a Doc Template will have the original one be replaced by the importing one.
5	Name	Name of Doc Template.
6	Description	Description of Doc Template.
7	Create and modified date	The date of creation and modification of this Doc Template.
8	Export	Export the current or any earlier revisions of Doc Templates as .docx file.

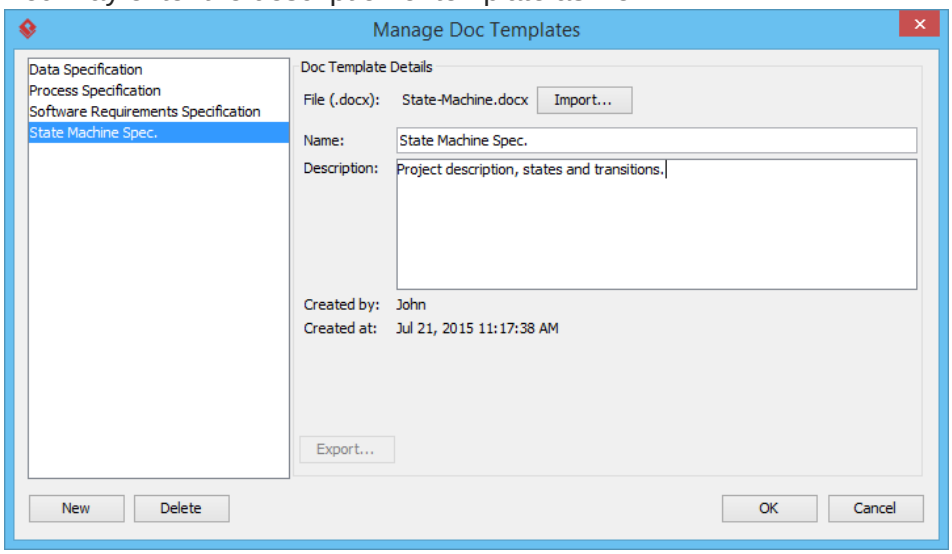
9	OK	Confirm the changes made and return to Doc. Composer.
10	Cancel	Discard the changes and return to Doc. Composer.

3.14.2 Creating a Doc Template

When you create a document under the Fill-in Doc mode of Doc. Composer, you will be asked to choose a Doc Base for document generation. There are two approaches of choosing Doc Base. One is to create from an external document file. Another one is to duplicate from an existing Doc Template. In this section, you will learn how to create a Doc Template. By creating a Doc Template, you can re-use it again and again in creating different documents.

By default, Visual Paradigm provides three default Doc Templates. You can also create your own set of Doc Templates in the Manage Doc Templates window by taking the steps below:

1. In the **Manage Doc Templates** window, click **New** at bottom left corner.
2. Choose your .docx file in the file chooser. The file you provide here should contain both manually written content (e.g. Introduction, project scope, etc) and Doc Fields.
3. Enter a meaningful name for the Doc Template.
4. You may enter the description of template as well.



5. Click **OK** to confirm the changes. Now, you can use the new template when you create a document with Fill-in Doc. You can also share it with teammates.

3.14.3 Deleting a Doc Template

In the **Manage Doc Templates** window, select the Doc Template to delete from the template list and click **Delete...** at bottom left corner to remove it permanently. Note that the deletion will NOT affect any of the document that used the deleted template. The only effect of deletion is that no one will be able to create documents with the Doc Template deleted.

3.14.4 Editing a Doc Template

If you have modified your project documentation, say, for updating its content or layout, you will need to replace the document file previously imported to a Doc Template with the new version of document. You can do this by editing the Doc Template.

In the **Manage Doc Templates** window, select the Doc Template to edit, and then modify its details on the right hand side. You can rename it, update its description or replace it with another document file by clicking **Import...**. Note that by importing a document file, the original one will be overwritten. However, you can always retrieve a previously imported file by clicking **Export...**. Click **OK** when finished editing. When finished, you can use the modified template when you create a document with Fill-in Doc. You can also share it with teammates.

3.14.5 Synchronizing Doc Templates

Once you have finished editing Doc Templates, you can synchronize the changes to server. Teammates can get the updated templates by synchronizing changes to server as well.

To synchronize changes to server manually, select **Tools > Doc. Composer > Sync. to VP Online/Teamwork Server** from the toolbar.

Note that your changes will be synchronized automatically when you perform commit.

Chapter 4.Writing Element Templates

4.1 What is Doc. Composer Template Language?

DCTL (Doc. Composer Template Language) is an XML-based language that enables the transformation of design specification into document content. DCTL comes with a well-defined structure and syntactic rules for writers to define what and how project data should be extracted from a Visual Paradigm project, and how these data should be presented in a document.

The following shows a basic template:

```
<DiagramBaseInitiationBlock>
	<TableBlock tableStyle="Summaries">
		<TableRow>
			<TableCell>
				<Text>Name</Text>
			</TableCell
		</TableRow>

		<IterationBlock modelType="UseCase">
			<TableRow>
				<TableCell>
					<Property property="name"/>
				</TableCell>

			</TableRow>
		</IterationBlock>
	</TableBlock>
</DiagramBaseInitiationBlock>
```

As the template shows, it tries to create a table, and add table rows for showing the names of use cases within a diagram. This example is a fairly simple one yet it outlines two main thing that every element template tries to achieve:

- Data retrieval - To query the use cases from a diagram, and then get the name of each use case.
- Layout of content – Table construction.

That's what you can do with a DCTL – You compose an element template with DCTL, drag the template into your document in Doc. Composer, let Doc. Composer interpret your template and output content accordingly.

In the coming sections you will see how to retrieve project data as well as to layout the content with the use of DCTL.

4.2 Template Root

Every element template mush has <ElementBaseInitiationBlock>, <DiagramBaseInitiationBlock> or <ProjectBaseInitiationBlock> as root element.

The use of <ElementBaseInitiationBlock> is to tell the template engine that the template will be applied to a model element. If you are writing a template for a model element (e.g. use case, package...), use <ElementBaseInitiationBlock> as template root.

The use of <DiagramBaseInitiationBlock> is to tell the template engine that the template will be applied to a diagram. If you are writing a template for a diagram (e.g. Class Diagram...), use <DiagramBaseInitiationBlock> as template root.

The use of <ProjectBaseInitiationBlock> is to tell the template engine that the template will be applied to a project. If you are writing a template for a project, use <ProjectBaseInitiationBlock> as template root.

If you check back the example used in the previous section, you will find that the template was written to query the use cases from a diagram. The template will be applied on a diagram so <DiagramBaseInitiationBlock> was used as root element.

The following examples show the use of <ElementBaseInitiationBlock> and <ProjectBaseInitiationBlock> in templates.

```
<ElementBaseInitiationBlock>

	<!—Output the name of selected class-->
	<Property property="name"/>

</ElementBaseInitiationBlock>
```

```
<ProjectBaseInitiationBlock>

	<!—Output the name of all use cases in the project-->
	<IterationBlock allLevel="true" modelType="UseCase">
		<Property property="name"/>
	</IterationBlock>

</ProjectBaseInitiationBlock>
```

4.3 Text and Property

When you want to output some text, you either use a <Text> or a <Property>.

You use <Text> when you want to output specific words, sentences or paragraphs, such as “Here is a list of business activities:”. The following example shows the use of <Text> in a template.

```
<Text>Here is a list of elements in my project:</Text>
<ParagraphBreak/>
<IterationBlock allLevel="true" >
       <Property property="name"/>
       <Text> : </Text>
       <Property property="modelType"/>
       <ParagraphBreak/>
</IterationBlock>
```

The first <Text> in the example above outputs a sentence “Here is a list of elements in my project: “.

The second <Text> outputs a colon between the name and type of elements. Note that the space before and after the colon will get output into the document.

Here is the outcome of the example above.

```
Here is a list of elements in my project:
Place Order : UseCase
PaymentController : Class
Cancel Order : BPTask
```

The following table lists the available attributes of <Text>.

Name	Description	Required?
isBold : boolean	Set the text to bold.	Optional
isItalic : boolean	Set the text italic.	Optional
isUnderline : boolean	Underline text.	Optional
fontFamily : string	Specify the name of font to apply to the text.	Optional
fontSize : integer	Set the font size.	Optional
foreColor : color	Set the color of text.	Optional
alignment : string {left | center | right}	Set the alignment of text.	Optional
style : string	Set the name of style. You can add and edit style in Doc. Composer.	Optional
numberingLevel : short	Determine the Numbering Level if the text is showing as a number or bullet list.	Optional
margin	Determine the left margin of text.	Optional
margin-top : integer	The top margin of a text. Note that only the first set of margin will be considered within a paragraph. The rest will	Optional

	be ignored. In Doc. Composer, a paragraph is ended by a <ParagraphBreak/>.	
margin-right : integer	The right margin of a text. Note that only the first set of margin will be considered within a paragraph. The rest will be ignored. In Doc. Composer, a paragraph is ended by a <ParagraphBreak/>.	Optional
margin-bottom : integer	The bottom margin of a text. Note that only the first set of margin will be considered within a paragraph. The rest will be ignored. In Doc. Composer, a paragraph is ended by a <ParagraphBreak/>.	Optional
margin-left : integer	The left margin of a text. Note that only the first set of margin will be considered within a paragraph. The rest will be ignored. In Doc. Composer, a paragraph is ended by a <ParagraphBreak/>.	Optional
hyperlink : boolean	Specify whether the text is a hyperlink or not. If true, the text will be linkable.	Optional
keepWithNext : boolean	Make sure the text will be shown in same page with next item. (Used for WORD document only)	Optional
href : string	The text will be linkable and the target will be the URL specified by this attribute. When this attribute is specified, @hyperlink will be ignored. Example: <Text href="https://www.visual-paradigm.com">Visual Paradigm</Text>	Optional
isWordFieldCode : boolean	When true, the text will be exported as a Word field code instead of plain text content. Example <Text isWordFieldCode="true">FILENAME</Text>	Optional

You may have noticed the use of <Property> in the example above. <Property> is another way to output text. You use <Property> when you want to output text by extracting the data from a property of querying diagram or element. The following example shows the use of <Property> in a template.

```
<IterationBlock allLevel="true" >
       <Text>Name: </Text>
       <Property property="name"/>
       <ParagraphBreak/>
       <Property property="description"/>
       <ParagraphBreak/>
       <ParagraphBreak/>
</IterationBlock>
```

The first <Property> outputs the name of the querying element, while the second <Property> outputs the description.

Here is the outcome of the example above.

```
Name: Place Order
The process to check out a shopping cart and finish the payment.

Name: PaymentController
A controller class that handles the payment logic.

Name: Cancel Order
The process to delete an order made within the last 7 days.
```

The following table lists the available attributes of <Property>.

Name	Description	Required?
property : string	The property to query.	Required
isIgnoreHTMLFontSize : boolean	Ignore the font size on the HTML text.	Optional
isIgnoreHTMLFontFamily : boolean	Ignore the font selection of the HTML text	Optional
forcePlainText : boolean	Force HTML text to show as plain text by removing formatting, if any.	Optional
defaultValue : string	The text to show when the value of property has not ever been specified.	Optional
isBold : boolean	Set the text to bold.	Optional
isItalic : boolean	Set the text italic.	Optional
isUnderline : boolean	Underline text.	Optional
fontFamily : string	Specify the name of font to apply to the text.	Optional
fontSize : integer	Set the font size.	Optional
foreColor : color	Set the color of text.	Optional
alignment : string {left | center | right}	Set the alignment of text.	Optional
style : string	Set the name of style. You can add and edit style in Doc. Composer.	Optional
numberingLevel : short	Determine the Numbering Level if the text is showing as a number or bullet list.	Optional

margin	The top, right, bottom, left margin of a text. Note that only the first set of margin will be considered within a paragraph. The rest will be ignored. In Doc. Composer, a paragraph is ended by a <ParagraphBreak/>. Example: margin="0, 0, 0, 0"	Optional
margin-top : integer	The top margin of a text. Note that only the first set of margin will be considered within a paragraph. The rest will be ignored. In Doc. Composer, a paragraph is ended by a <ParagraphBreak/>.	Optional
margin-right : integer	The right margin of a text. Note that only the first set of margin will be considered within a paragraph. The rest will be ignored. In Doc. Composer, a paragraph is ended by a <ParagraphBreak/>.	Optional
margin-bottom : integer	The bottom margin of a text. Note that only the first set of margin will be considered within a paragraph. The rest will be ignored. In Doc. Composer, a paragraph is ended by a <ParagraphBreak/>.	Optional
margin-left : integer	The left margin of a text. Note that only the first set of margin will be considered within a paragraph. The rest will be ignored. In Doc. Composer, a paragraph is ended by a <ParagraphBreak/>.	Optional
hyperlink : boolean	Specify whether the text is a hyperlink or not. If true, the text will be linkable.	Optional
keepWithNext : boolean	Make sure the text will be shown in same page with next item. (Used for WORD document only)	Optional
dateFormatString : string	Date value property will be formatted with the format pattern specified before displaying. Formatting will only occur when the property is a date value (e.g. pmLastModified). e.g. @dateFormatString ="yyyy-MM-dd"	Optional
typeWithFullyQualify : boolean	Output the fully qualified type of querying element. For example: true to output com.vp.MyClass instead of MyClass	Optional
returnTypeWithFullyQualify : boolean	Output the fully qualified return type of querying element. For example: true to output com.vp.MyClass instead of MyClass	Optional

4.3.1 Understanding Dynamic Heading Style

When you want to set a text produced by a <Text> or a <Property> to be a heading, add and specify the style attribute in the <Text> or <Property>.

There are two ways of specifying a heading – Static and Dynamic. The following example shows the static way of specifying heading:

```
<Text style="Heading 1">Text in heading 1</Text>
<Property property="name" style="Heading 2"/>
```

The <Text> in the example above outputs a sentence “Text in heading 1”, with Heading 1 as style. The <Property> outputs the name of a model element, with Heading 2 as style.

The static way of specifying heading requires you to provide the style name in the template.

In contrast to the static way, here is an example that shows the dynamic way of specifying heading:

```
<Text style="@heading+">Text in heading 1</Text>
<Property property="name" style="@heading+"/>
```

In the example above, we do not provide the name of the heading style. Instead, we use **@heading** to indicate the need to assign heading style, **@heading+** to indicate an increase of heading style level. By using **@heading**, the style Heading 1…N will be used in the output document.

Here is the outcome of the example above.

Text in heading 1

Place Order Use Case

The sentence *Text in heading 1* has Heading 1 applied, while the name *Place Order Use Case* has Heading 2 applied.

4.3.1.1 More about Heading Increment

By appending **+** to **@heading**, the leveling of heading style will be increased. For example, if the previous heading is a Heading 1, the use of **@heading+** will output a heading in Heading 2.

The use of **+** in **@heading** is optional though. If you want to add a heading that has the same heading level as the previous heading, skip + to reuse the style used by the previous heading.

4.4 Looping (Non Connector)

When you want to retrieve the children elements from a querying model element / diagram, write a loop element.

4.4.1 <IterationBlock>

Retrieve elements from project / model element / diagram. By iterating over project and model element, a list of model element will be returned. By iterating over diagram, a list of diagram element will be returned. The following example shows the use of <IterationBlock> in a template.

```
<IterationBlock modelType="class">
	<Property property="name"/>
	<ParagraphBreak/>
</IterationBlock>
```

Here is the outcome of the example above.

```
User
Account
AccountManager
Transaction
AccountController
```

The following table lists the available attributes of <IterationBlock>.

Name	Description	Required?
modelType : string	Filter the children by specified model element type (e.g. package).	Optional
modelTypes : string	Filter the children by a number of model element types. (e.g. actor, usecase)	Optional
stereotypes : string	Filter the children by a number of stereotypes.	Optional
name : string	Filter the children by their name.	Optional
filterHidden : boolean	Filter hidden children diagram element. This is for retrieving from diagram/diagram element only.	Optional
includeConnectors : boolean	Determines whether to retrieve shape or shape+connectors from diagram. This is for retrieving from diagram only.	Optional
allLevel : boolean	Determines whether to retrieve all model elements from project. When false, only the root level elements will be retrieved. This attribute is only useful when retrieving elements from project.	Optional
ignoreLastSeparator : boolean	Ignore the break for the last element of current for-each loop.	Optional
breakString : string	Insert a string between model elements of current for-each.	
identifier : string		Optional
suppressDuplicatedModelElement : boolean = true	When the same model element is being included twice in an iteration, having	Optional

	suppressDuplicatedModelElement set to true means to ignore those repeated occurrences.	

4.4.2 <ForEach>

Retrieve model elements from a model element's property. The following example shows the use of <ForEach> in a template.

```
<ForEach property = "stereotypes">
       <Property property="name"/>
       <ParagraphBreak/>
</ForEach>
```

Here is the outcome of the example above.

```
Control
ORM Persistable
```

The following table lists the available attributes of <ForEach>.

Name	Description	Required?
property : string	The property from which model elements can be retrieved.	Required
ignoreLastSeparator : boolean	Ignore the break for the last element of current for-each loop.	Optional
breakString : string	Insert a string between model elements of current for-each.	Optional

4.4.3 <ForEachSubDiagram>

Retrieve sub-diagram(s) from a model element. For example, retrieve sub-sequence-diagrams from a controller class. Note that you can only use <ForEachSubDiagram> to retrieve sub-diagram(s) of model element. If you want to retrieve diagrams from project, use <ForEachDiagram> instead.

The following table lists the available attributes of <ForEachSubDiagram>.

Name	Description	Required?
diagramType : string	The type of diagram to retrieve.	Optional
ignoreLastSeparator : boolean	Ignore the break for the last element of current for-each loop.	Optional
layerFilters : string	Select the diagram layer to or not to process when outputting content to a document. Let's say if you need to produce a document for a business stakeholder, you may not want him to see the annotation shapes. What you have to do is to configure the layer filter by excluding the annotation layer (assuming that such a layer exists). Doc. Composer engine will read the filter and not to process the annotation shapes.	Optional

	Possible values: **@followDiagram** - Follow the visibility of the layers set to the actual diagram. Layers that are set visible will be included here, likewise hidden layers will be excluded. Simply put, what you can see in the document will be exactly the same as the real diagram. **@all** - Include all diagram layers in processing. **name** - The name of the only layer to include in processing. **name1, name2, name3...** - The names of the layers to include in processing. ", " is used as a delimiter. **${...}** - '...' is the variable name. Use a variable to specify the name/names of layers to include in processing. User can specify the value of the variable in Doc. Composer. **@exclude:name** - The name of the only layer not to process. **@exclude:name1, name2, name3...** - The names of the layers not to process. **@exclude:${...}** - '...' is the variable name. Use a variable to specify the name/names of layers not to include in processing. User can specify the value of the variable in Doc. Composer. By not specifying layerFilters, 'AdHoc' filter will be used, which means that the end user will be responsible for configuring the filter in Doc. Composer. If not specified, it will behave as @followDiagram.	

4.4.4 <ForEachDiagram>

Retrieve diagram(s) from project. Like other for-each elements, you can specify the type of diagram to retrieve. For example, retrieve all class diagrams from project. Note that you can only use <ForEachDiagram> to retrieve diagram from project. If you want to retrieve sub-diagrams from model element, use <ForEachSubDiagram> instead.

The following table lists the available attributes of <ForEachDiagram>.

Name	Description	Required?
diagramType : string	The type of diagram to retrieve.	Optional
property : string	The property from which diagrams can be retrieved.	Optional

ignoreLastSeparator : boolean	Ignore the break for the last element of current for-each loop.	Optional
layerFilters : string	Select the diagram layer to or not to process when outputting content to a document. Let's say if you need to produce a document for a business stakeholder, you may not want him to see the annotation shapes. What you have to do is to configure the layer filter by excluding the annotation layer (assuming that such a layer exists). Doc. Composer engine will read the filter and not to process the annotation shapes. Possible values: @**followDiagram** - Follow the visibility of the layers set to the actual diagram. Layers that are set visible will be included here, likewise hidden layers will be excluded. Simply put, what you can see in the document will be exactly the same as the actual diagram. @**all** - Include all diagram layers in processing. **name** - The name of the only layer to include in processing. **name1, name2, name3...** - The names of the layers to include in processing. ", " is used as a delimiter. **${...}** - '...' is the variable name. Use a variable to specify the name/names of layers to include in processing. User can specify the value of the variable in Doc. Composer. @**exclude:name** - The name of the only layer not to process. @**exclude:name1, name2, name3...** - The names of the layers not to process. @**exclude:${...}** - '...' is the variable name. Use a variable to specify the name/names of layers not to include in processing. User can specify the value of the variable in Doc. Composer. By not specifying layerFilters, 'AdHoc' filter will be used, which means that the end user will be responsible for configuring the filter in Doc. Composer. If not specified, it will behave as @followDiagram.	Optional

4.4.5 <ForEachOwnerDiagram>

Retrieve the diagram(s) that owns a specific model element. For example, class diagram "Domain Diagram" and "Security" both contain class "Login" (same model element), by applying <ForEachOwnerDiagram> on the "Login" class, diagram "Domain Diagram" and "Security" will be returned.

The following table lists the available attributes of <ForEachOwnerDiagram>.

Name	Description	Required?
diagramType : string	The type of diagram to retrieve.	Optional
ignoreLastSeparator : boolean	Ignore the break for the last element of current for-each loop.	Optional
layerFilters : string	Select the diagram layer to or not to process. Let's say if you need to produce a document for a business stakeholder, you may not want him to see the annotation shapes. What you have to do is to configure the layer filter by excluding the annotation layer (assuming that such a layer exists). Doc. Composer engine will read the filter and not to process the annotation shapes. Possible values: **@followDiagram** - Follow the visibility of the layers set to the real diagram. Layers that are set visible will be included here, likewise hidden layers will be excluded. Simply put, what you can see in the document will be exactly the same as the real diagram. **@all** - Include all diagram layers in processing. **name** - The name of the only layer to include in processing. **name1, name2, name3...** - The names of the layers to include in processing. ", " is used as a delimiter. **${...}** - '...' is the variable name. Use a variable to specify the name/names of layers to include in processing. User can specify the value of the variable in Doc. Composer. **@exclude:name** - The name of the only layer not to process. **@exclude:name1, name2, name3...** - The names of the layers not to process.	Optional

	@exclude:${...} - '...' is the variable name. Use a variable to specify the name/names of layers not to include in processing. User can specify the value of the variable in Doc. Composer. By not specifying layerFilters, 'AdHoc' filter will be used, which means that the end user will be responsible for configuring the filter in Doc. Composer. If not specified, it will behave as @followDiagram.	

4.5 Looping (Connector)

4.5.1 <ForEachSimpleRelationship>

Retrieve SimpleRelationship elements from model element or connector from diagram element.

The following table lists the available attributes of <ForEachSimpleRelationship>.

Name	Description	Required?
modelType : string	Filter relationship by specified model element type (e.g. Generalization). Note that not all kind of relationship belongs to simple relationship. Here are the possible types of simple relationship: Abstraction, ActivityObjectFlow, AnalysisComposition, AnalysisDiagramTransitor, AnalysisParentChild, AnalysisReference, AnalysisRelationship, AnalysisSubDiagram, AnalysisTransitor, AnalysisUsed, AnalysisView, Anchor, ArchiMateAccess, ArchiMateAggregation, ArchiMateAssignment, ArchiMateAssociation, ArchiMateCommunicationPath, ArchiMateFlow, ArchiMateNetwork, ArchiMateProvide, ArchiMateRealization, ArchiMateRequire, ArchiMateSpecialization, ArchiMateTriggering, ArchiMateUsedBy, AssociationClass, BPAssociation, BPDataAssociation, BPMessageFlow, BPSequenceFlow, BindingDependency, BusinessRuleAssociation, Constraint, ControlFlow, ConversationLink, DBForeignKey, DFDataFlow, Dependency, Deployment, EPCControlFlow, EPCInformationFlow, EPCOrganizationUnitAssignment, ExceptionHandler, Extend, Generalization, GenericConnector, GlossaryFactTypeAssociation, Include, InteractionDiagramDurationConstraint, Link, MindConnector, MindLink, OCLine, ObjectFlow, PMProcessLink, Permission, RQRefine, RQTrace, Realization, RequirementDerive, Satisfy, Transition, Transition2, Usage, Verify	Optional

modelTypes : string	Filter relationships by modelTypes. If @modelType is defined, @modelTypes will be ignored.	Optional
direction {all | from | to}	Filter relationship by direction.	Optional
ignoreLastSeparator : boolean	Ignore the break for the last element of current for-each loop.	Optional
breakString : string	Insert a string between model elements of current for-each.	Optional

4.5.2 <ForEachRelationshipEnd>

Retrieve the from or to end of an association.

The following table lists the available attributes of <ForEachRelationshipEnd>.

Name	Description	Required?
modelType : string	Filter relationship end by specific model element type.	Optional
modelTypes : string	Filter relationship ends by modelTypes. If @modelType is defined, @modelTypes will be ignored.	Optional
endPointer {all | from | to | both | self | other}	Filter relationship ends based on the way they are attached to the querying element. For example, if 'from' is specified, only relationships that take the querying element as the source (i.e. from end) will be chosen.	Optional
ignoreLastSeparator : boolean	Ignore the break for the last element of current for-each loop.	Optional
breakString : string	Insert a string between model elements of current for-each.	Optional

4.6 Sorting in Loop

Add <Sortings> under a loop element (e.g. <IterationBlock>, <ForEach>) to sort retrieved elements. <Sortings> contains one or more <Sorting>. Each <Sorting> defines a way to sort the elements retrieved. The following example shows the use of <Sorting> in a template.

```
<IterationBlock modelType="Class">
	<Sortings>
		<Sorting by="property" property="name"/>
	</Sortings>
	<Property property="name"/>
	<ParagraphBreak/>
</IterationBlock>
```

Here is the outcome of the example above.

```
Account
AccountController
```

```
AccountManager
Transaction
User
```

The following table lists the available attributes of <Sorting>.

Name	Description	Required?
by : string {name | type | modelType | diagramType | property | followTree | level | businessProcessFlow}	Sort by any of the following options: - name : sort by name - type : sort by type (type of model element or diagram) - modelType : sort by model element's type - diagramType : sort by diagram type - property: sort by property, requires the definition of @property, @sortValues, @defaultPropertyValue - followTree: follows the order of elements in active tree – either Model Navigator or Diagram Navigator - level: sort by parent-child - businessProcessFlow: sort the BPD elements by the ordering in BPD (calculated by their ordering in sequence/message flow). ONLY AVIALABLE for sorting Diagram Elements in a BPD	Required
property : string	If @by="property", you have to specify @property to name the property to be sorted. You can also sort elements by their tagged values by specifying this: ${taggedValues.children(TAG_NAME).value} Replace TAG_NAME with the name of the tag to be sorted	Optional
sortValues : string	If @by="property", you can specify @sortValues to define the ordering of values to be sorted. e.g. @by="property" @property="visibility" @sortValues="public, protected, private" means 'public' model elements will list before 'protected' model elements, 'protected' will list before 'private'	Optional
defaultPropertyValue : string	If @by="property", @defaultPropertyValue can be specified for the default value of the model elements that don't have this property value.	Optional
descending : Boolean	true to sort elements in descending order, false to sort elements in ascending order.	Optional

dateFormatString : string	Date value property will be formatted with the format pattern specified before sorting. Formatting will only occur when the property is a date value (e.g. pmLastModified). e.g. @dateFormatString ="yyyy-MM-dd"	Optional

4.6.1 Suppress the default way of sorting

Without using <Sortings> and <Sorting>, elements in loop will still be sorted alphabetically. If you want to suppress the default way of sorting, write <Sortings noSort="true"/>. Here is an example:

```
<IterationBlock modelType="Class">
	<Sortings noSort="true"/>
	<Property property="name"/>
	<ParagraphBreak/>
</IterationBlock>
```

4.7 Conditional Expression

4.7.1 <DefaultValueChecker>

The <DefaultValueChecker> element evaluates the querying element to check if the property stated by the @property attribute equals to its default value. If the result of evaluation matches with the result stated by the attribute @flag, the child elements of <DefaultValueChecker> will be processed. Otherwise, the child elements will be skipped.

The following example shows the use of <DefaultValueChecker> in a template.

```
<IterationBlock modelType="Class">
	<DefaultValueChecker property="root" flag="false">
		<Text>The root property has been modified.</Text>
	</DefaultValueChecker>
</IterationBlock>
```

The following table lists the available attributes of <DefaultValueChecker>.

Name	Description	Required?
property : string	The property to check. **Note**: If you want to evaluate if a model is from a referenced project, use fromReferenceProject (i.e. property="fromReferenceProject" value="true")	Optional
flag : boolean	The expected result of checking. If the actual result matches the value specified by @flag, the child elements will be processed.	Optional

4.7.2 <ValueChecker>

The <ValueChecker> element evaluates the querying element to check if the value of the property stated by the @property attribute equals to the value stated by the @value attribute. If the result of evaluation is true, the child elements of <ValueChecker> will be processed. Otherwise, the child elements will be skipped.

The following example shows the use of <ValueChecker> in a template.

```
<IterationBlock modelType="Class">
	<ValueChecker property="name" value="ShapeCreator">
		<Text>ShapeCreator class found!</Text>
	</ValueChecker>
	<ValueChecker property="description" operator="not equals" value="">
		<Text>Description: </Text>
		<Property property="description"/>
	</ValueChecker>
</IterationBlock>
```

The following table lists the available attributes of <ValueChecker>.

Name	Description	Required?
property : string	The property to check.	Optional
operator : string {equals | not equals | less than | equals or less than | greater than | equals or greater than | like | not like | equal | not equal}	Specify the way to compare the property value of model against your expectation. equals - The value of property must be the same as the expected value not equals - The value of property must be different from the expected value less than - The value of property must be smaller than the expected value. equals or less than - The value of property must be the same or smaller than the expected value. greater than - The value of property must be larger than the expected value. equals or greater than - The value of property must be the same or larger than the expected value. like – The value of property must contain the expected value not like - The value of property must not contain the expected value	Optional
value : string	The value expected for the property. If @regularExpression is set to true, you can make use of '?' and '*' in the value field for representing wildcard characters. For example, use "*UI" as	Optional

	@value to find out all model elements with names end with "UI".	
length : int		Optional
caseSensitive : Boolean	Determine whether the checking of string property need to take care of the use of upper and lower case.	Optional
regularExpression : boolean	When true, you can make use of '?' and '*' in the value field for representing wildcard characters. For example, use "*UI" as @value to find out all model elements with names end with "UI".	Optional
id : string		Optional
dateFormatString : string	Date value property will be formatted with the format pattern specified before checking. Formatting will only occur when the property is a date value (e.g. pmLastModified). e.g. @dateFormatString ="yyyy-MM-dd"	Optional

4.7.3 <HasChildElementChecker>

The <HasChildElementChecker> element evaluates the querying element to check if it contains any child element, or type(s) of child elements specified by the @modelType or @modelTypes attributes. If the result of evaluation is true, the child elements of <HasChildElementChecker> will be processed. Otherwise, the child elements will be skipped.

The following example shows the use of <HasChildElementChecker> in a template.

```
<IterationBlock modelType="Class">
	<HasChildElementChecker modelType="Attribute" flag="true">
		<IterationBlock modelType="Attribute">
			<Property property="name"/>
			<ParagraphBreak>
		</IterationBlock>
	</HasChildElementChecker>
	<HasChildElementChecker modelType="Attribute,Operation" flag="true">
		<IterationBlock modelTypes="Attribute,Operation">
			<Property property="name"/>
			<ParagraphBreak>
		</IterationBlock>
	</HasChildElementChecker>
</IterationBlock>
```

The following table lists the available attributes of <HasChildElementChecker>.

Name	Description	Required?

flag : boolean	The expected result of checking. If the actual result matches the value specified by @flag, the child elements will be processed.	Optional
modelType : string	The type of model element you want the parent to contain or not contain.	Optional
modelTypes : string	The types of model element you want the parent to contain or not contain.	Optional
stereotypes : string	Filter the children by a number of stereotypes.	Optional
includeConnectors : boolean	Determine whether to retrieve shape or shape+connectors from diagram. This is for retrieving from diagram only.	
filterHidden : Boolean		
allLevel : Boolean	Determine whether to retrieve all model elements from project. When false, only the root level elements will be retrieved. This attribute is only useful when retrieving elements from project.	
valueConditionCheckId : string		

4.7.4 <HasRelationshipChecker>

The <HasRelationshipChecker> element evaluates the querying element to check if it contains any relationship, or type(s) of relationships specified by the @modelType or @modelTypes attributes. If the result of evaluation is true, the child elements of <HasRelationshipChecker> will be processed. Otherwise, the child elements will be skipped.

The following example shows the use of <HasRelationshipChecker> in a template.

```
<IterationBlock modelType="Class">
	<HasRelationshipChecker modelType="Association" flag="true">
		<ForEachRelationshipEnd modelType="AssociationEnd" endPointer="self">
			<RelationshipEndEndRelationship>
				<FromEnd>
					<ModelElementProperty property="EndModelElement">
						<Property property="name"/>
						<Text>, </Text>
					</ModelElementProperty>
				</FromEnd>
				<ToEnd>
					<ModelElementProperty property="EndModelElement">
						<Property property="name"/>
						<Text>, </Text>
					</ModelElementProperty>
				</ToEnd>
		</ForEachRelationshipEnd>
	</HasRelationshipChecker>
```

```
      <HasRelationshipChecker modelType="Generalization" direction="to">
            <ForEachSimpleRelationship type="Generalization">
                  <ModelElementProperty property="from">
                        <Property property="name"/>
                        <Text>, </Text>
                  </ModelElementProperty>
            </ForEachSimpleRelationship>
      </HasRelationshipChecker>
</IterationBlock>
```

The following table lists the available attributes of <HasRelationshipChecker>.

Name	Description	Required?
flag : boolean	Check whether you want any relationship to exist or not.	Optional
modelType : string	The type of relationship you want the querying model element to contain.	Optional
modelTypes : string	The types of relationship you want the querying model element to contain. If @modelType is specified, @modelTypes will be ignored.	Optional
direction {all | from | to}	Check if the querying model element belongs to a specific end of a relationship.	Optional

4.7.5 <HasDiagramChecker>

The <HasDiagramChecker> element evaluates the querying “thing” (project or model element) to check if it has specified any diagram for a given property specified by the @property attribute. If the result of evaluation is true, the child elements of <HasDiagramChecker> will be processed. Otherwise, the child elements will be skipped.

The following example shows the use of <HasDiagramChecker> in a template.

```
<ProjectBaseInitiationBlock>
      <HasDiagramChecker diagramType="ClassDiagram" flag="true">
            <Text>This project contains at least one class diagram.</Text>
      </HasDiagramChecker>
</ProjectBaseInitiationBlock>
```

The following table lists the available attributes of <HasDiagramChecker>.

Name	Description	Required?
flag : boolean	The expected result of checking. If the actual result matches the value specified by @flag, the child elements will be processed.	Optional
property : string	The property from which diagrams can be retrieved.	Optional

diagramType : string	The type of diagram you want the project to contain or not contain.	Optional

4.7.6 <HasValueChecker>

The <HasValueChecker> element evaluates the querying “thing” (project or model element) to check if it has specified a given property, specified by the @property attribute. If the result of evaluation is true, the child elements of <HasValueChecker> will be processed. Otherwise, the child elements will be skipped.

The following example shows the use of <HasValueChecker> in a template.

```
<IterationBlock modelType="Class">
	<HasValueChecker property="taggedValues" flag="true">
		<Text>This class contains at least one tagged value.</Text>
	</HasValueChecker>
</IterationBlock>
```

The following table lists the available attributes of <HasValueChecker>.

Name	Description	Required?
flag : boolean	The expected result of checking. If the actual result matches the value specified by @flag, the child elements will be processed.	Optional
property : string	The name of property to check.	Optional
modelType : string	The result of evaluation will return a true only if the querying model element contains the type of elements specified by @modelType.	Optional
name : string	The result of evaluation will return a true only if the querying model element contains the elements with same name as specified by @name.	Optional
stereotypes : string	The result of evaluation will return a true only if the querying model element contains the elements that are extended from the stereotypes specified by @stereotypes.	Optional

4.7.7 <HasParentModelChecker>

The <HasParentModelChecker> element evaluates the querying model element to check if it is being contained by a parent model element. If the result of evaluation is true, the child elements of <HasParentModelChecker> will be processed. Otherwise, the child elements will be skipped.

The following example shows the use of <HasParentModelChecker> in a template.

```
<IterationBlock modelType="Class">
	<HasParentModelChecker modelType="Package" flag="true">
		<Text>This class contains is contained by package: </Text>
```

```
            <ParentModel>
                  <Property property="name"/>
            </ParentModel>
      </HasParentModelChecker>
</IterationBlock>
```

The following table lists the available attributes of <HasParentModelChecker>.

Name	Description	Required?
flag : boolean	The expected result of checking. If the actual result matches the value specified by @flag, the child elements will be processed.	Optional
modelType : string	The type of parent model element you want the model element to be/not to be contained by.	Optional

4.7.8 <HasSubDiagramChecker>

The <HasSubDiagramChecker> element evaluates the querying model element to check if it contains any sub-diagram. If the result of evaluation is true, the child elements of <HasSubDiagramChecker> will be processed. Otherwise, the child elements will be skipped.

The following example shows the use of <HasSubDiagramChecker> in a template.

```
<IterationBlock modelType="Class">
      <HasSubDiagramChecker diagramType="StateMachineDiagram" flag="true">
            <ForEachSubDiagram>
                  <Property property="name"/>
            </ForEachSubDiagram>
      </HasSubDiagramChecker>
</IterationBlock>
```

The following table lists the available attributes of <HasSubDiagramChecker>.

Name	Description	Required?
flag : boolean	The expected result of checking. If the actual result matches the value specified by @flag, the child elements will be processed.	Optional
diagramType : string	The type of sub-diagram you want the model element to contain.	Optional

4.7.9 <HasOwnerDiagramsChecker>

The <HasOwnerDiagramsChecker> element evaluates the querying model element to check if it has been visualized in any diagram. If the result of evaluation is true, the child elements of <HasOwnerDiagramsChecker> will be processed. Otherwise, the child elements will be skipped.

The following example shows the use of <HasOwnerDiagramsChecker> in a template.

```
<IterationBlock modelType="BPTask">
	<HasOwnerDiagramsChecker diagramType="BusinessProcessDiagram" flag="true">
		<ForEachOwnerDiagram>
			<Property property="name"/>
		</ForEachOwnerDiagram>
	</HasOwnerDiagramsChecker>
</IterationBlock>
```

The following table lists the available attributes of <HasOwnerDiagramsChecker>.

Name	Description	Required?
flag : boolean	The expected result of checking. If the actual result matches the value specified by @flag, the child elements will be processed.	Optional
diagramType : string	The type of diagram you want the owner diagram to be.	Optional

4.7.10 Checking Multiple Conditions with <Conditions>

The use of <...Checker> enables you to perform checking on single condition. Sometimes, you may want to check for multiple conditions at a time. For example, you may want to output the name of all public and static attributes of a class. In order to check for multiple conditions, use <Conditions>. <...Checker> under <Conditions> will all be evaluated. If the result of evaluation are true for ALL checkers, the subsequent elements will be evaluated. Otherwise, the parent iteration will continue to next round.

Note that <Conditions> supports the type attribute, which enables you to specify the way how the result of checkers are evaluated. The value "and" means that the result of checkers must all be positive in order to continue, while the value "or" means that as long as there is one checker that returns a positive result, the flow can continue.

The following example shows the use of <Conditions> in a template.

```
<IterationBlock modelType="Attribute">
	<Conditions>
		<ValueChecker property="visibility" value="public" />
		<ValueChecker property="scope" value="classifier" />
	</Conditions>

	<Property property="name"/>
	<ParagraphBreak/>
</IterationBlock>
```

4.7.10.1 Using <...ConditionChecker> (e.g. <IterationBlockConditionChecker>, <ForEachConditionChecker>, etc.)

The following lists out a set of <...ConditionChecker>.

- <IterationBlockConditionChecker>

- <ForEachConditionChecker>
- <ForEachRelationshipConditionChecker>
- <ForEachSimpleRelationshipConditionChecker>
- <ForEachEndRelationshipConditionChecker>
- <ForEachRelationshipEndConditionChecker>
- <ForEachSubDiagramConditionChecker>
- <ForEachOwnerDiagramConditionChecker>
- <ForEachDiagramConditionChecker>
- <ForEachDiagramElementConditionChecker>
- <ModelElementPropertyConditionChecker>
- <FromEndConditionChecker>
- <ToEndConditionChecker>
- <RelationshipEndEndRelationshipConditionChecker>
- <RelationshipEndOppositeEndConditionChecker>
- <DiagramPropertyConditionChecker>
- <DiagramElementPropertyConditionChecker>
- <ParentModelConditionChecker>
- <ParentShapeConditionChecker>
- <OwnerDiagramConditionChecker>

They all share similar usage so let's explain them together. To make it simple, let's explain with <IterationBlockCondictionChecker>, using the example above.

Let's say we want to add a line "Public and static attributes" before we list out the attributes. Here is the problem: If we place a <Text>Public and static attributes</Text> before <IterationBlock>, the line will get output even without any public and static attributes. If we put the <Text> inside the <IterationBlock> and below <Conditions>, the line will get output multiple times if there are multiple public and static attributes.

To solve this problem, use <IterationBlockConditionChecker>. <IterationBlockConditionChecker> enables you to check if there exists an element in an iteration, when certain conditions are applied. The following example is the revised version of the above example. It will output a single line of title above the list of attributes output within the <IterationBlock>.

```
<IterationBlockConditionChecker>
	<Conditions>
		<ValueChecker property="visibility" value="public" />
		<ValueChecker property="scope" value="classifier" />
	</Conditions>

	<Text>Public and static attributes</Text>
	<ParagraphBreak/>

	<IterationBlock modelType="Attribute">
		<Conditions>
			<ValueChecker property="visibility" value="public" />
			<ValueChecker property="scope" value="classifier" />
		</Conditions>
```

```
            <Property property="name"/>
            <ParagraphBreak/>
      </IterationBlock>
<IterationBlockConditionChecker/>
```

4.7.10.2 Using <ConditionsChecker>

The use of <Conditions> allows the filtering of elements in a loop. But there are times that you want to check against a querying model element instead of filtering elements in a looping. In such case, use <ConditionsChecker>

The following example shows the use of <ConditionsChecker> in a template.

```
<ElementBaseInitiationBlock>
      <ConditionsChecker>
            <Conditions type="and">
            <ValueChecker property="visibility" value="public" />
           <ValueChecker property="scope" value="classifier" />
      </Conditions>

      <Property property="name"/>
      <ParagraphBreak/>
      </ConditionsChecker>
</ElementBaseInitiationBlock>
```

4.7.11 Using Nested Checkers in Propagated Checking

Sometimes, a checking requires the checking of not just the querying element, but certain property of the querying element. Let's say we want to retrieve all public attributes from a class whose attribute type has to be classes with names ended with 'Controller'. In order to handle such a complex checking, we have to create a nested checker structure which involves the nested use of <...Checker> and <Conditions>.

The following example shows the use of various checkers in a nested structure.

```
<IterationBlock modelType="Attribute">
      <Conditions>
            <ValueChecker property="visibility" value="public" />
            <ModelElementPropertyConditionChecker property="type">
                  <Conditions>
                        <ValueChecker property="name" operator="like"
value="Controller" />
                  <Conditions>
            </ModelElementPropertyConditionChecker>
      </Conditions>

      <Property property="name"/>
      <ParagraphBreak/>
</IterationBlock>
```

4.8 Working with Table

4.8.1 <TableBlock>

Insert a table to document. It is typically used to present table of elements or element properties. You must combine the use of <TableRow> and <TableCell> in order to form a complete table. Most of the build-in templates are formed by tables and contains <TableBlock>. You can look for references easily.

Name	Description	Required?
tableStyle : string	Specify the table style by its ID. You can find the ID of table styles in the **Formats** window of Doc. Composer.	Optional
tableWidth : string	The width of table. For example: '50%' means to occupy 50% of page width. There are four available units: % (e.g. 50%) cm (e.g. 10cm) mm (e.g. 800mm) px (e.g. 500px).	Optional
colWidths : integers	Specify the widths of table columns in ratio, separate by comma. Note that the number of columns specify in @colWidths must match the number of <TableCell> to add under <TableRow>, under this table. For example, specify "1, 1, 2" for a table with 20000 as width will result in creating a table with three columns, and have widths 5000, 5000, 10000.	Optional
rowBackgroundColors : color	Background color of rows in table.	Optional
repeatTableHeader {true | false | followOption}	True to repeat the table header row at the top of the next page for table that span multiple pages. Note that this option only works in PDF and Word document. If "followOption", it will follow the setting set in Doc. Composer's document export window.	Optional
singlePage : Boolean = false	True to ensure that a table will not be split in two or more pages. When the remaining space of a page cannot display the whole table, Doc. Composer will try to show it in the next page. If the table height is longer than a page, the exceeding part of the table will be cropped.	Optional

4.8.2 <TableRow>

Enables you to add rows to a <TableBlock>. Without <TableRow>, <TableBlock> is useless. You should add <TableCell> to <TableRow> in order to complete a table.

Name	Description	Required?
height : integer	How tall it is for the table row.	Optional
backgroundColor : color	Background color of row.	Optional
singlePage : boolean = false	True to ensure that a table row will not be split in two or more pages. When the remaining space of a page cannot display the whole row, Doc. Composer will try to show it in the next page.	Optional

4.8.3 <TableCell>

Enables you to add cells to a <TableRow>.

Name	Description	Required?
topBorderEnable : boolean	True to draw the top border of cell.	Optional
bottomBorderEnable : Boolean	True to draw the bottom border of cell.	Optional
leftBorderEnable : Boolean	True to draw the left border of cell.	Optional
rightBorderEnable : Boolean	True to draw the right border of cell.	Optional
verticalAlignment {top | center | bottom}	The vertical alignment of cell.	Optional
color : color	The background color of cell.	Optional
colspan : integer	Specify the number of cell this cell consumes horizontally. For example, a colspan of 2 means to consume this and the cell on the right. This is equivalent to HTML's colspan.	Optional

4.9 Image

4.9.1 <Image>

Name	Description	Required?
alignment {left | center | right}	Set the alignment of image.	Optional
width : string	Set the width of image. It can be an absolute value (e.g. "15500") or a scale to the original image width (e.g. "80%")	Optional
height : string	Set the height of image. It can be an absolute value (e.g. "15500") or a scale to the original image height (e.g. "80%")	Optional

maxWidth : integer	Set the maximum width of image.	Optional
maxHeight : integer	Set the maximum height of image.	Optional
rotate **{none | right | left}**	Rotate the image to right (90 degree) or left (270 degree)	Optional
keepWithPreviousInPDF : boolean	Make sure the previous item will be shown in same page with this item. (Used for PDF document only)	Optional
keepWithNext : boolean	Make sure this item will be shown in same page with next item. (Used for WORD document only)	Optional

4.9.2 <Icon>

Icon of a model element type.

Name	Description	Required?
alignment **{left | center | right}**	Set the alignment of icon image.	Optional
rotate **{none | right | left}**	Rotate the image to right (90 degree) or left (270 degree)	Optional

4.10 Break

4.10.1 <ParagraphBreak>

Enables you to add a break in document to separate text into paragraphs. <ParagraphBreak> does not carry any text.

4.10.2 <PageBreak>

Enables you to insert a new page at where <PageBreak> is processed.

4.11 Other Constructs

4.11.1 <OwnerDiagram>

Retrieve the diagram in which a diagram element or the master view of a model element reside. For example, class diagram "Domain Diagram" and "Security" both contain class "Login" (same model element), while the master view is placed inside "Login", by applying <OwnerDiagram> on the "Login" class, diagram "Domain Diagram" will be returned.

If you want to retrieve all the diagrams that own a model element, use <ForEachOwnerDiagram> instead.

4.11.2 <ParentModel>

<ParentModel> serves two purposes. First, to retrieve the immediate parent element of a model element or diagram. Second, to look for a specific type of parent element along the hierarchy.

Here is an example of how <ParentModel> can help you find the immediate parent element. For example, class "Circle" is in package "Shape", by applying <ParentModel> on the "Circle" class, the package "Shape" will be returned.

Name	Description	Required?
modelType : string	Used in finding a specific type of parent along the hierarchy. For example, if class "Shape" is in package "Shape" and "Shape" is in model "Main", by applying <ParentModel modelType="Model"> on "Circle", the model "Main" will be returned.	Optional

4.11.3 <ParentShape>

<ParentShape> serves two purposes. First, to retrieve the immediate parent shape of a shape. Second, to look for a specific type of parent shape along the hierarchy.

Here is an example of how <ParentDiagram> can help you find the immediate parent element. For example, class "Circle" is drawn in package "Shape", by applying <ParentParent> on the "Circle" class, the package "Shape" will be returned.

Name	Description	Required?
shapeType : string	Used in finding a specific type of parent along the hierarchy. For example, if class "Shape" is drawn in package "Shape" and "Shape" is drawn in model "Main", by applying <ParentShape modelType="Model"> on "Circle", the model "Main" will be returned.	Optional

4.12 Reusing Template with Inline or Reference

You may want to produce same content under different templates. If you duplicate the same template code in multiple templates, you need to spend extra time and effort in keeping them consistent with each other. In such a case, you can create one template, and reuse it in other templates. The reuse of template can be done by using <Reference> and <Inline>. The following example shows the use of <Reference> and <Inline> in a template.

```
<Inline template="Children (General)"/>
<Reference template="Children (General)"/>
```

The example above means that when <Inline> is met, substitute that part with content written in the Children (General) template. Same for <Reference>.

4.12.1 Inline vs Reference

Both <Inline> and <Reference> support the reuse of templates. They work nearly identically except one important difference – The way they handle dynamic heading style.

Dynamic heading style, as its name suggest, supports the dynamic assignment of heading style to text content. If you use <Inline> and in the referencing template a **@heading+** is used, the leveling of active heading style will be increased by one. Once the referencing template has ended and the flow flows back to the source template, the leveling of heading style will remain as-is, which means, same as the leveling used by the last heading defined in the referencing template.

What makes <Reference> different is that when its flow ends and flows back to the source template, the leveling of heading style will be reset to that before entering the referencing template.

Let's explain with an example. Here a template *Bar*.

```
<AnyBaseInitiationBlock>
   <Text style="@heading+">Heading</Text>
   <Text style="@heading+">Heading</Text>
   <Text style="@heading+">Heading</Text>
</AnyBaseInitiationBlock>
```

Here is another template *Foo*. It references the *Bar* template above.

```
<DiagramBaseInitiationBlock>
   <Text style="@heading">Init</Text>
   <Inline template="Bar"/>
   <Text style="@heading">Result</Text>
</DiagramBaseInitiationBlock>
```

In *Bar*, several **@heading+** has been used, which trigger the increases of the leveling of heading style. You can expect that in the end the text *Result* printed by *Foo* will be in Heading 4 because it follows last style used by the template referencing inline.

If we change <Inline> to <Reference>, the text *Result* will be in Heading 1, following the style last used within *Foo*.

4.13 Using Variable

To favor template reusability, some of the template elements support the use of variable. User who produce a document with a template that has variable defined will need to specify the value of the variable in runtime to generate the outcome he/she expect.

4.13.1 How does it work?

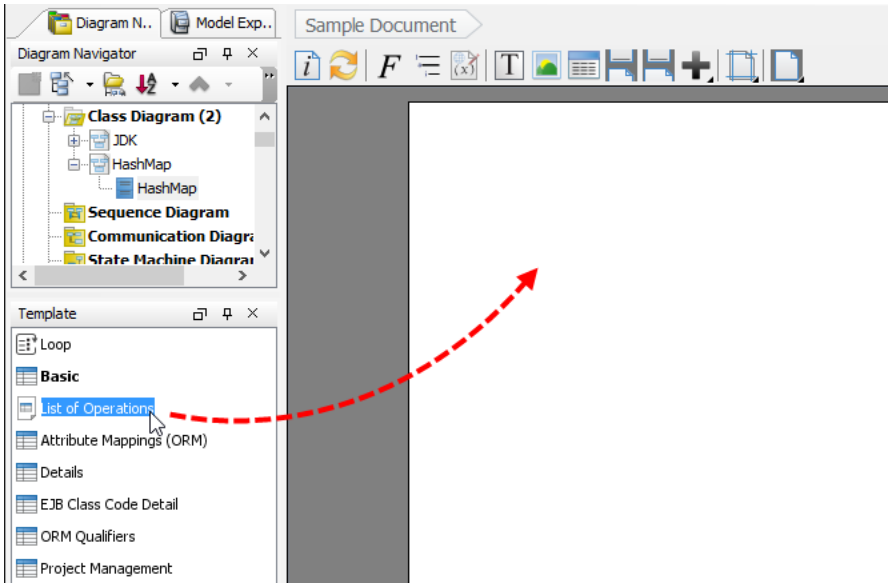
Let's take a look at the following example:

A template *List of Operations* has been written for element type 'Class' to output a list of operation names from a given class. Let's say we want to output only operations with a specific visibility to be decided by the person who produce the final document. This is the content of the template:

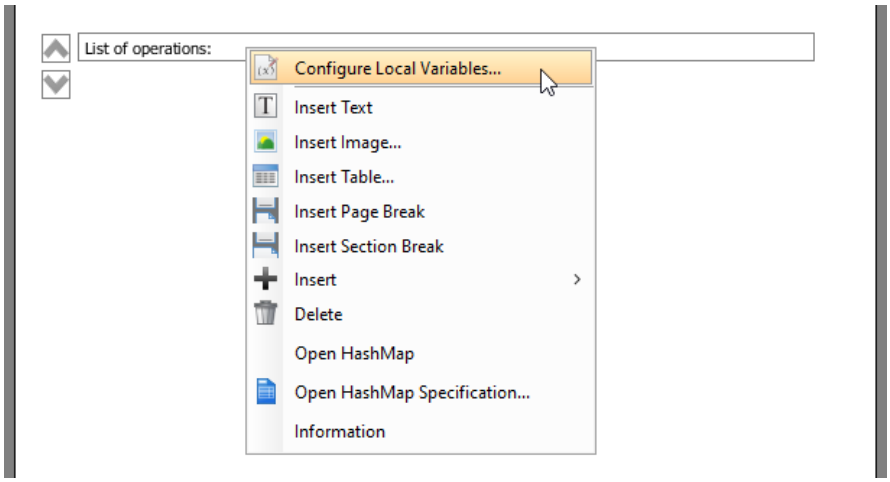
```
<?xml version="1.0" encoding="UTF-8"?>
<ElementBaseInitiationBlock>
   <Text>List of operations:</Text>
   <ParagraphBreak/>
   <IterationBlock modelType="Operation">
      <ValueChecker property="visibility" operator="equals" value="${oper-visibility}">
         <Property property="name"></Property>
         <ParagraphBreak/>
      </ValueChecker>
   </IterationBlock>

</ElementBaseInitiationBlock>
```

Pay attention to @value in <ValueChecker>. Instead of having the value of visibility hardcoded in the template, a variable ${oper-visibility} is used.

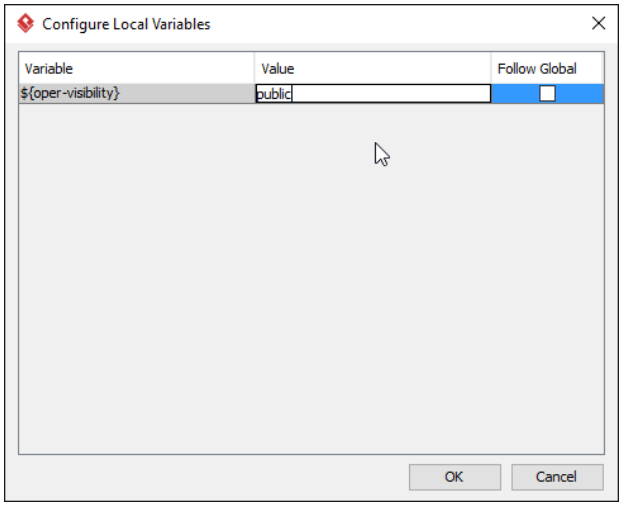
In Doc. Composer, the person who want to output a document with the template will use the template as usual by dragging it onto the document.



After that, he/she has to specify the value of the variable, which is, in this case the visibility of operation expected. To do this, right click on the content block on the document and select **Configure Local Variables...** from the popup menu.

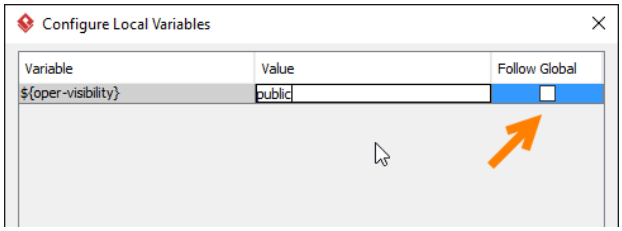


In the **Configure Local Variables** window, enter the value of the variable. In this case, *public* is entered. This means that the variable ${oper-visibility} will be replaced by the text 'public' when producing content for this block.

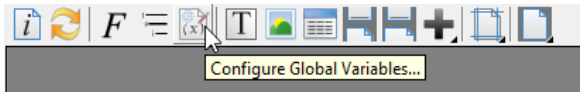


When finished, click **OK** to confirm.

Besides entering the value, there is an option **Follow Global** in the configuration window. The value of variable can be set locally and globally. Values set locally will be effective within a specific block of content, while values set globally will be effective to the entire document. To make a variable follows global setting, simply check **Follow Global**.



To enter the value of variables globally, click on the **Configure Global Variables...** button in the toolbar of Doc. Composer and enter the value in the **Configure Global Variables** window.



4.13.2 Why variable?

To use variable instead of hard-coding a value gives you the following benefits.

4.13.2.1 Unifying the outcome

Instead of writing and maintaining a set of similar templates, you just need to write a single template only. This ensures the consistency of similar content since they all come from a single source.

4.13.2.2 Let the 'user' of template decide what to output

The use of variable allows the person who produce the document decide what to output, instead of forcing the editor of template to make the decision when writing the template, which is sometimes impractical.

4.13.3 Elements that supports the use of variable

The following table lists the elements and the attributes that support the use of variable.

Element	Attribute
TableBlock	tableStyle
Text	style, href
Property	style
ForEachDiagram	diagramType
ForEachSubDiagram	diagramType
ForEachOwnerDiagram	diagramType
HasDiagramChecker	diagramType
HasSubDiagramChecker	diagramType
HasOwnerDiagramsChecker	diagramType
HasChildElementChecker	stereotypes
HasRelationshipChecker	modelTypes
HasParentModelChecker	modelType
HasValueChecker	stereotypes, modelType, name
IterationBlock	stereotypes, name, modelTypes
ValueChecker	value

Appendix A - DCTL Examples

Working with Use Case Scenario

```
<ForEach property="stepContainers">
   <Property property="name"/>
   <TableBlock>
      <FlowOfEventIterationBlock> <!-- Walk through each step (row) in a scenario -->
         <TableRow>
            <TableCell>
               <FlowOfEventIndent/> <!-- Apply proper indentation to the current step. You
don't have to specify the level of indentation. It's automatically done for you -->

               <Property property="index" foreColor="#848284" style="Table Contents"/> <!-
- The step number. We set its foreColor to a lighter one to make it looks like how it
looks in Visual Paradigm -->

               <Text style="Table Contents"> </Text>
               <ValueChecker property="type" operator="not equals" value="">
                  <ValueChecker property="type" operator="not equals" value="system">
                     <Property property="type" foreColor="#00B200" style="Table
Contents"/> <!-- @type here refers to 'control labels' like if, then, elseif -->
                  </ValueChecker>
                  <ValueChecker property="type" operator="equals" value="system">
                     <Property property="type" foreColor="#CA6400" style="Table
Contents"/> <!-- @type here refers to SYSTEM -->
                  </ValueChecker>
               </ValueChecker>
               <Property property="name" style="Table Contents"/> <!-- The content of step
-->
            </TableCell>
         </TableRow>
      </FlowOfEventIterationBlock>
   </TableBlock>
</ForEach>
```

Working with Sub-Diagrams

```
<HasSubDiagramChecker>
   <!-- Name of all sub-diagrams -->
   <ForEachSubDiagram>
      <Property property="name" />
      <ParagraphBreak/>
   </ForEachSubDiagram >

   <!-- Name of all sub-BPD -->
   <ForEachSubDiagram diagramType="BusinessProcessDiagram">
      <Property property="name" />
      <ParagraphBreak/>
   </ForEachSubDiagram >
</HasSubDiagramChecker>
```

Working with References

```
<!-- Output the name of all Use Case reference -->
<HasValueChecker property="references">
   <ForEach property="references">
      <ValueChecker property="type" operator="equals" value="Model Element">
         <ModelElementProperty property="url">
            <ValueChecker property="modelType" operator="equals" value="UseCase">
               <Property property="name" />
            </ValueChecker>
         </ModelElementProperty>
      </ValueChecker>
   </ForEach>
</HasValueChecker>

<!-- Output the name of all model elements that are referencing the querying element as
'Model Element Reference' -->
<HasValueChecker property="modelElementReferencedBys">
   <ForEach property="modelElementReferencedBys">
      <Property property="name"/>
   </ForEach>
</HasValueChecker>

<!-- Output the name of all model elements that are referencing the querying element as
'Shape Reference' -->
<HasValueChecker property="viewReferencedBys">
   <ForEach property="viewReferencedBys">
      <Property property="name"/>
   </ForEach>
</HasValueChecker>
```

Working with Stereotypes and Tagged Values

```
<!-- Output the name of all assigned stereotypes -->
<HasValueChecker property="stereotypes">
   <ForEach property="stereotypes" ignoreLastSeparator="true">
      <Text>&lt;&lt;</Text>
         <Property property="name" />
      <Text>&gt;&gt;</Text>
   </ForEach>
</HasValueChecker>
<ParagraphBreak/>

<!-- Output the name of tagged values added -->
<HasValueChecker property="taggedValues">
   <ModelElementProperty property="taggedValues">
      <HasChildElementChecker modelType="TaggedValue">
         <IterationBlock modelType="TaggedValue">
            <Property property="name" />
            <Text> - </Text>
            <Property property="value" />
         </IterationBlock>
      </HasChildElementChecker>
   </ModelElementProperty>
```

```
</HasValueChecker>
```

Working with Table Records of Entity

```
<!-- Check if an entity (i.e. DBTable) has record specified -->
<HasValueChecker property="records">
   <TableBlock>
      <!-- Create a header row that shows the columns' name in each cell -->
      <TableRow>
         <IterationBlock modelType = "DBColumn">
            <TableCell>
               <Property property="name" />
            </TableCell>
         </IterationBlock>
      </TableRow>
      <!-- Create one row for each record -->
      <ModelElementProperty property="records">
         <IterationBlock modelType = "EntityRecord">
            <TableRow>
               <IterationBlock modelType = "EntityRecordCell">
                  <TableCell>
                     <!-- Output the value directly if it's a general column -->
                     <Property property="value" />
                     <!-- Use the following method to output the value if it's a FK -->
                     <HasValueChecker property="value" flag="true">
                        <ModelElementProperty property="value">
                           <Property property="value" />
                        </ModelElementProperty>
                     </HasValueChecker>
                  </TableCell>
               </IterationBlock>
            </TableRow>
         </IterationBlock>
      </ModelElementProperty>
   </TableBlock>
</HasValueChecker>
```

Working with Working Procedures of BPMN Task/Sub-Process

```
<!-- Check if a BPMN task/sub-process has working procedure entered -->
<HasValueChecker property="bpProcedures">

   <!-- Retrieve the procedure steps -->
   <ForEach property="bpProcedures">
      <Property property="name" style="@heading"/>
      <ParagraphBreak/>

      <!-- Retrieve the steps in the querying procedure set -->
      <ForEach property="bpProcedureSteps">
         <Property property="name"/>
         <ParagraphBreak/>
      </ForEach>
   </ForEach>
```

```
</HasValueChecker>
```

Working with Action and Action's Type

```
<HasChildElementChecker flag="true">
   <TableBlock colWidths="1,2" tableWidth="14500" alignment="right">
      <TableRow>
         <TableCell leftBorderEnable="false" rightBorderEnable="false" color="230, 230,
230">
            <Text style="Column header 1">Element</Text>
         </TableCell>
         <TableCell leftBorderEnable="false" rightBorderEnable="false" color="230, 230,
230">
            <Text style="Column header 1">Description</Text>
         </TableCell>
      </TableRow>

      <IterationBlock modelType="ActivityAction">
         <!-- Sort the activities by name. -->
         <Sortings>
            <Sorting by="name"></Sorting>
         </Sortings>

         <ValueChecker property="modelType" operator="not equals" value="">
            <TableRow>
               <TableCell leftBorderEnable="false" rightBorderEnable="false">
                  <Property property="name" style="Table Contents"/>
               </TableCell>
               <TableCell leftBorderEnable="false" rightBorderEnable="false">
                  <ValueChecker property = "documentation" operator = "NOT EQUAL" value =
"">
                     <Property property="documentation" style="Table Contents"/>
                     <ParagraphBreak/>
                     <ParagraphBreak/>
                  </ValueChecker>

                  <ModelElementProperty property="actionType">

                     <!-- Call Behavior Action -->
                     <ValueChecker property="modelType" operator="equals"
value="CallBehaviorAction">
                        <Text style="Table Contents">Call action:</Text>
                        <HasValueChecker property="behavior" flag="true">
                           <ModelElementProperty property="behavior">
                              <Property property="name" style="TableContents"
isBold="true"/>
                          </ModelElementProperty>
                        </HasValueChecker>
                        <HasValueChecker property="behavior" flag="false">
                           <Text style="Table Contents">&lt;Unspecified&gt;</Text>
                        </HasValueChecker>
                     </ValueChecker>

                     <!-- Call Operation Action -->
```

```
                    <ValueChecker property="modelType" operator="equals"
value="CallOperationAction">
                       <Text style="Table Contents">Call operation:</Text>
                       <HasValueChecker property="operation" flag="true">

                          <ModelElementProperty property="operation">
                             <ParentModel>
                                <Property property="name" style="Table Contents"
isBold="true"/>
                             </ParentModel>
                             <Text style="Table Contents">.</Text>
                             <Property property="name" style="Table Contents"
isBold="true"/>
                          </ModelElementProperty>
                       </HasValueChecker>
                       <HasValueChecker property="operation" flag="false">
                          <Text style="Table Contents">&lt;Unspecified&gt;</Text>
                       </HasValueChecker>
                    </ValueChecker>
                 </ModelElementProperty>

              </TableCell>
           </TableRow>
        </ValueChecker>
     </IterationBlock>
  </TableBlock>
</HasChildElementChecker>
```

Working with Chart Relations

```
<!—Chart Relations -->
<HasValueChecker property="chartRelations">
   <Text style="@heading+">Chart Relations</Text>
   <ParagraphBreak/>

   <TableBlock tableStyle="Summaries" colWidths="20, 80">
      <TableRow>
         <TableCell>
            <Text>Code</Text>
         </TableCell>
         <TableCell>
            <Text>Begins</Text>
         </TableCell>
         <TableCell>
            <Text>Ends</Text>
         </TableCell>
      </TableRow>

      <ForEach property="chartRelations">
         <TableRow>
            <TableCell>
               <ModelElementProperty property="code">
                  <Property property="code"/>
               </ModelElementProperty>
            </TableCell>
```

```
            <TableCell>
               <ModelElementProperty property="from">
                  <Icon/>
                  <Property property="name"/>
               </ModelElementProperty>
            </TableCell>
            <TableCell>
               <ModelElementProperty property="to">
                  <Icon/>
                  <Property property="name"/>
               </ModelElementProperty>
            </TableCell>
         </TableRow>
      </ForEach>
   </TableBlock>
</HasValueChecker>
```

Working with Model Transitor

```
<!-- Check if the querying element has transition, either from/to -->
<HasValueChecker property="traceability">
   <Text>Traceability detected.</Text>
   <ParagraphBreak/>
</HasValueChecker>
<ParagraphBreak/>

<!-- List out the Transit From elements -->
<Text isBold="true">Transit From:</Text>
<ParagraphBreak/>
<ForEach property="transitFrom">
   <Property property="name"/>
   <ParagraphBreak/>
</ForEach>
<ParagraphBreak/>

<!-- List out the Transit To elements -->
<Text isBold="true">Transit To:</Text>
<ParagraphBreak/>
<ForEach property="transitTo">
   <Property property="name"/>
   <ParagraphBreak/>
</ForEach>
<ParagraphBreak/>

<!-- List out the Transit From diagrams -->
<Text isBold="true">Transit From (Diagram):</Text>
<ParagraphBreak/>
<ForEachDiagram property="transitFrom">
   <Property property="name"/>
   <ParagraphBreak/>
</ForEachDiagram>
<ParagraphBreak/>

<!-- List out the Transit To diagrams -->
```

```
<Text isBold="true">Transit To (Diagram):</Text>
<ParagraphBreak/>
<ForEachDiagram property="transitTo">
   <Property property="name"/>
   <ParagraphBreak/>
</ForEachDiagram>
<ParagraphBreak/>
```

Working with InstanceSpecification in an Object Diagram

```
<!-- Print the description of instanceSpecifications and their classifiers, in a given
object diagram-->
<DiagramBaseInitiationBlock>
    <IterationBlock modelType="InstanceSpecification">
        <!-- Name of instance (in ${instance_name} : ${classifier} format)-->
        <Property property="name"/>
        <Text> : </Text>
        <HasValueChecker property="classifiers">
            <ForEach property="classifiers">
                <Property property="name" />
            </ForEach>
        </HasValueChecker>

        <!-- Description of class-->
        <ParagraphBreak/>
        <Text isBold="true">Description of classifier: </Text>
        <HasValueChecker property="classifiers">
            <ForEach property="classifiers">
                <Property property="description" />
            </ForEach>
        </HasValueChecker>

        <!-- Description of instance-->
        <ParagraphBreak/>
        <Text isBold="true">Description of instance: </Text>
        <Property property="description"/>
        <ParagraphBreak/>
        <ParagraphBreak/>
    </IterationBlock>
</DiagramBaseInitiationBlock>
```

Appendix B – Diagram Types

The following table contains all the diagram types available in Visual Paradigm 15.0. When you need to query a specific type of diagram with DCTL, make sure the value presented in the second column is used. Please be aware that the list is version specific, so you may not find all of them in previous versions. The diagram types are ordered following the Diagram Navigator, so that you can easily locate the diagram type you want.

Display Diagram Type	Diagram Type to Use in Fields/Templates
Use Case Diagram	UseCaseDiagram
Class Diagram	ClassDiagram
Sequence Diagram	InteractionDiagram
Communication Diagram	CommunicationDiagram
State Machine Diagram	StateDiagram
Activity Diagram	ActivityDiagram
Component Diagram	ComponentDiagram
Deployment Diagram	DeploymentDiagram
Package Diagram	PackageDiagram
Object Diagram	ObjectDiagram
Composite Structure Diagram	CompositeStructureDiagram
Timing Diagram	TimingDiagram
Interaction Overview Diagram	InteractionOverviewDiagram
Textual Analysis	TextualAnalysis
Requirement Diagram	RequirementDiagram
Business Concept Diagram	FreehandDiagram2
CRC Card Diagram	CRCCardDiagram
Android Tablet Wireframe	WFAndroidTabletDiagram
Android Phone Wireframe	WFAndroidPhoneDiagram
Desktop Wireframe	WFDesktopDiagram
iPad Wireframe	WFIPadDiagram
iPhone Wireframe	WFIPhoneDiagram
Web Wireframe	WFWebDiagram

Entity Relationship Diagram	ERDiagram
ORM Diagram	ORMDiagram
Business Process Diagram	BusinessProcessDiagram
Conversation Diagram	ConversationDiagram
Data Flow Diagram	DataFlowDiagram
EPC Diagram	EPCDiagram
Process Overview Diagram	ProcessMapDiagram
Organization Chart	OrganizationChart
Fact Model	FactDiagram
Decision Table	DTBDecisionTableEditorDiagram
Block Definition Diagram	BlockDefinitionDiagram
Internal Block Diagram	InternalBlockDiagram
Parametric Diagram	ParametricDiagram
Zachman Framework	ZachmanDiagram
Migration Roadmap	MigrationRoadmap
Business Motivation Model Diagram	BusinessMotivationModelDiagram
ArchiMate Diagram	ArchiMateDiagram
Radar Chart	MaturityAnalysis
Implementation Plan Diagram	ArchitectureRoadmap
Enhanced PERT Chart	PERTChart
Breakdown Structure Diagram	BreakdownStructure
Cause and Effect Diagram	FishboneDiagram
Service Interface Diagram	SoaMLServiceInterfaceDiagram
Service Participant Diagram	SoaMLServiceParticipantDiagram
Service Contract Diagram	SoaMLServiceContractDiagram
Services Architecture Diagram	SoaMLServicesArchitectureDiagram
Service Categorization Diagram	SoaMLServiceCategorizationDiagram
CMMN Diagram	CMMNDiagram
Matrix Diagram	MatrixDiagram

Analysis Diagram	AnalysisDiagram
Chart Diagram	ChartDiagram
Overview Diagram	OverviewDiagram
User Interface Diagram	UserInterfaceDiagram
Mind Mapping Diagram	MindMapDiagram
Grid	GridDiagram
Brainstorm	Brainstorm
Profile Diagram	ProfileDiagram
Documentation Cabinet	DocumentationCabinet
Doc. Composer Document	ReportDiagram