DocGen User's Guide

DocGen User's Guide

Publication date Sat Feb 11 13:04:57 PST 2017

Table of Contents

List of Figures

List of Tables

Chapter 1. DocGen Overview

The Document Generator (DocGen) is a module of MDK plug-in in MagicDraw. It provides the capability to generate formal documents from UML/SysML models in MagicDraw. A "document" is a view into a model, or a representation of model data, which may be structured in a hierarchical way. A document is a collection of paragraphs, sections, and analysis, and the order and layout of the content is important. DocGen operates within MagicDraw, traversing document's "outline", collecting information, performing analysis, and writing the output to a file. DocGen produces a DocBook XML file, which may be fed into transformation tools to produce the document in PDF, HTML, or other formats.

DocGen consists of:

- a UML profile with elements for creating a document framework
- a set of scripts for traversing document frameworks, conducting analysis, and producing the output
- a set of tools to help users validate the correctness and completeness of their documents.

Users need to invest the time to define the document content and format; however, once this is done the document can be produced with a button click whenever the model data is updated. The user never has to waste time numbering sections or fighting with reluctant formatting, as this is all performed automatically during the transformation to PDF, HTML, etc. DocGen can also be extended and queries may be added in a form of reusable analysis functions. Check with your project to see if they have a document framework to use.

Chapter 2. Install DocGen

DocGen is a part of the MDK plugin. If you have installed an EMS-provided version of MagicDraw (i.e. Crushinator), this is already included and the following install steps are not applicable. If you are using the SSCAE download (provided on the MBSE community of practice webpage), you will need to install the MDK plugin.

Install:

1. Download the latest MDK plugin from the
   <mms-transclude-name> [cf:Access and Downloads.name] </mms-transclude-name>
   table on the
   <a>MDK Site.</a>
2. Download the mdk-version.zip file from the
   <mms-transclude-name> [cf:Artifactory.name] </mms-transclude-name>
   link. **Do not unzip the file.**
3. Start MagicDraw, go to Help->Resource/Plugin Manager
4. Click on Import and choose the zip file. Restart MagicDraw.

Add module to project:

1. Open or create a project. Go to Options->Modules.
2. Click on Use Module and select the SysML Extensions.mdxml file (this should be in the md.install/profiles/MDK directory)
3. Now you can use the stereotypes from Document Profile, inside DocGen>MDK EMP Client

You should only need to add the module once per project. If you install an updated plug-in, you don't always have to re-add the module to the project. The model will prompt you if it cannot find the module.

Chapter 3. Create Offline Content Using DocGen

If a model is not available in EMS server, an offline document can be created using the DocGen module in MDK plug-in. This section describes how to produce offline, read-only documents in MagicDraw. Note that these documents are not editable like those published in EMS. All changes must be made directly in the MagicDraw model.

Generate Documents

DocGen allow generating a local version of your document without access to EMS server. Select a document, and use a right click menu "DocGen>Generate DocGen3 Document". It will generate a DocBook XML file. Check with your project to see if there is a preferred method of generation. Most practitioners use a .xml viewer such as oXygen to convert the .xml file to a PDF file (or other types of files if they prefer). You can install oXygen from the SSCAE. Note that the document is static. If you want to change the document, changes need to made to the MagicDraw model and document needs to be regenerated.

To open your .xml output with oXygen, first obtain a copy of the mgss.xsl stylesheet from MBEE (if you downloaded the bundled installation of Crushinator from MBEE, you can find mgss.xsl in the install folder under DocGenUserScripts -> DocGenStyleSheet). Open your oXygen install directory, navigate to <oxygen_dir>/frameworks/docbook/xsl/fo, and save mgss.xsl to that directory. This stylesheet will allow the front matter inputs from MagicDraw to be visible on the PDF.

Now you can open your saved .xml output file with oXgen. After opening your file, right click the "DocBook PDF" option within transformation scenarios, as pictured below. In the menu that appears, select "Duplicate." Rename your new scenario as desired. Select the XSL URL text box and replace "${frameworks}/docbook/xsl/fo/docbook_custom.xsl" with ${frameworks}/docbook/xsl/fo/mgss.xsl," as pictured below, and then click "OK."

To generate a new PDF, double-click on your newly created transformation in the Transformation Scenarios window. You can also click the red "play" button if your DocBook PDF is selected as shown below to generate a PDF.

Use the DocGen Stylesheet

The DocGen stylesheet allows the user to input front matter information to a document. To input the information you would like to add as front matter, navigate to the specification window of a document block. (The easiest way is to double click on the document block.) Scroll down in the specification window to the "FrontMatter" section, as shown below.

Here, you can specify a variety of fields such as logo size and location, approvers, legal footers, etc. A reference card for the tags is shown below.

Chapter 4. Create Viewpoint Methods

Creating a viewpoint involves two steps. First, create a viewpoint element. Second, define a viewpoint method diagram for the viewpoint. Then the viewpoint can be applied to a view. The following subsections explain each step in more details.

Collect/Sort/Filter Model Elements

Once exposed to a view, elements can can be operated on by three types of viewpoint operators (Collect, Sort, and Filter). These operations can be used to expand or narrow the collection of elements that are used in the viewpoint method.

The following views reveal the View Diagrams, Viewpoint Method Diagrams, and Block Definition Diagrams that - when generated - result in the contained subviews.

Collect

"Collect..." is a viewpoint operator that separates the elements in an exposed package into ones that will continue to be used and ones that will be ignored. There are a number of collection methods, each of which have their own call behavior action in the side bar of the viewpoint method diagram.

CollectOwnedElements

CollectOwnedElements gathers anything owned by the exposed element(s) in the model to be used in the viewpoint method. A common use case would be when a group of elements contained within a package need to be exposed to a view. Instead of individually exposing each element, a user can employ "CollectOwnedElements" within a viewpoint and then expose only the package. The collection method will look at the exposed element and return all the elements that it owns.

The views
<mms-view-link> [cf:Single Depth Example.vlink] </mms-view-link>
and
<mms-view-link> [cf:Infinite Depth Example.vlink] </mms-view-link>
show the output of the
<mms-transclude-name> [cf:Example View Diagram -.name] </mms-transclude-name>
as shown below. Both of these views expose the same
<mms-transclude-name> [cf:Lunchbox -.name] </mms-transclude-name>
, but they conform to different viewpoints. The only difference between the viewpoints is the "DepthChoosable" property:
<mms-transclude-name> [cf:CollectOwnedElements Single Depth -.name] </mms-transclude-name>
has a DepthChoosable of 1,
<mms-transclude-name> [cf:CollectOwnedElements Infinite Depth -.name] </mms-transclude-name>
has a DepthChoosable of 0.

Note that CollectOwnedElements collects part properties as well. For this example we use
<mms-transclude-name> [cf:FilterByStereotypes -.name] </mms-transclude-name>
to only display Block elements in the output.

Figure 4.1. Example View Diagram

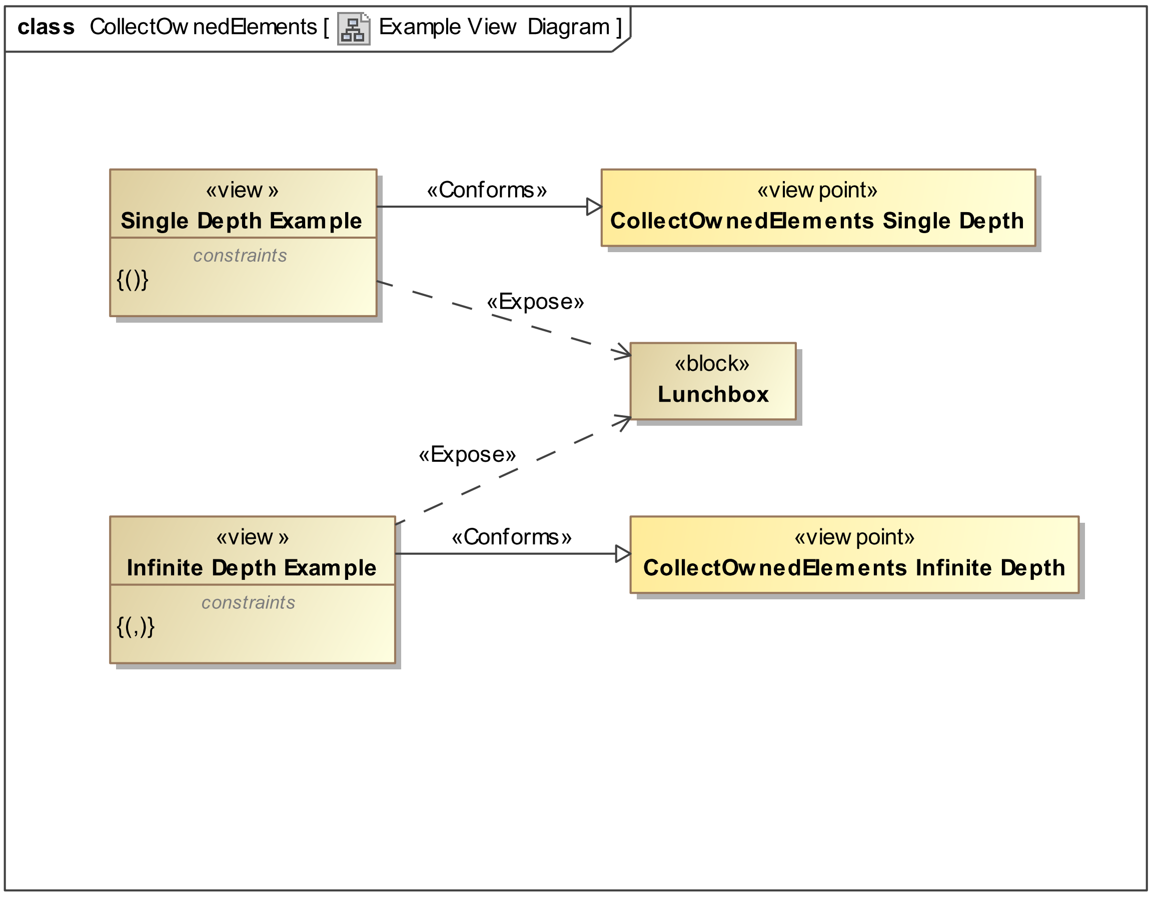


Figure 4.2. Example Diagram

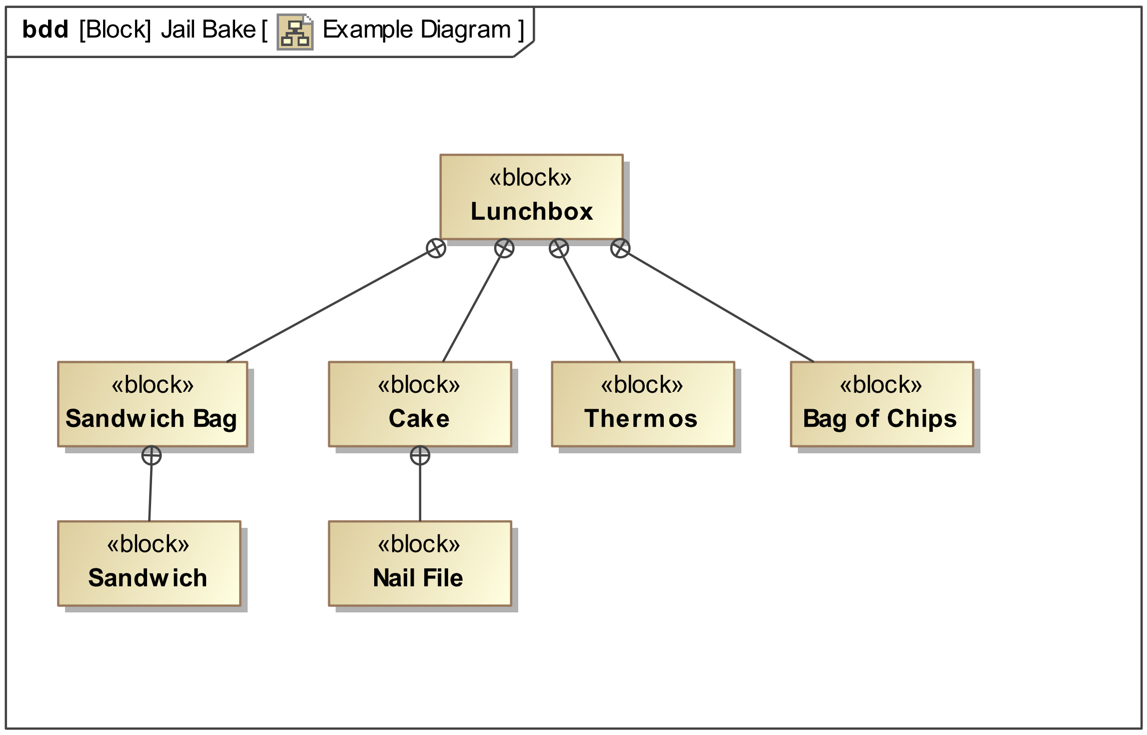


Figure 4.3. CollectOwnedElements Single Depth

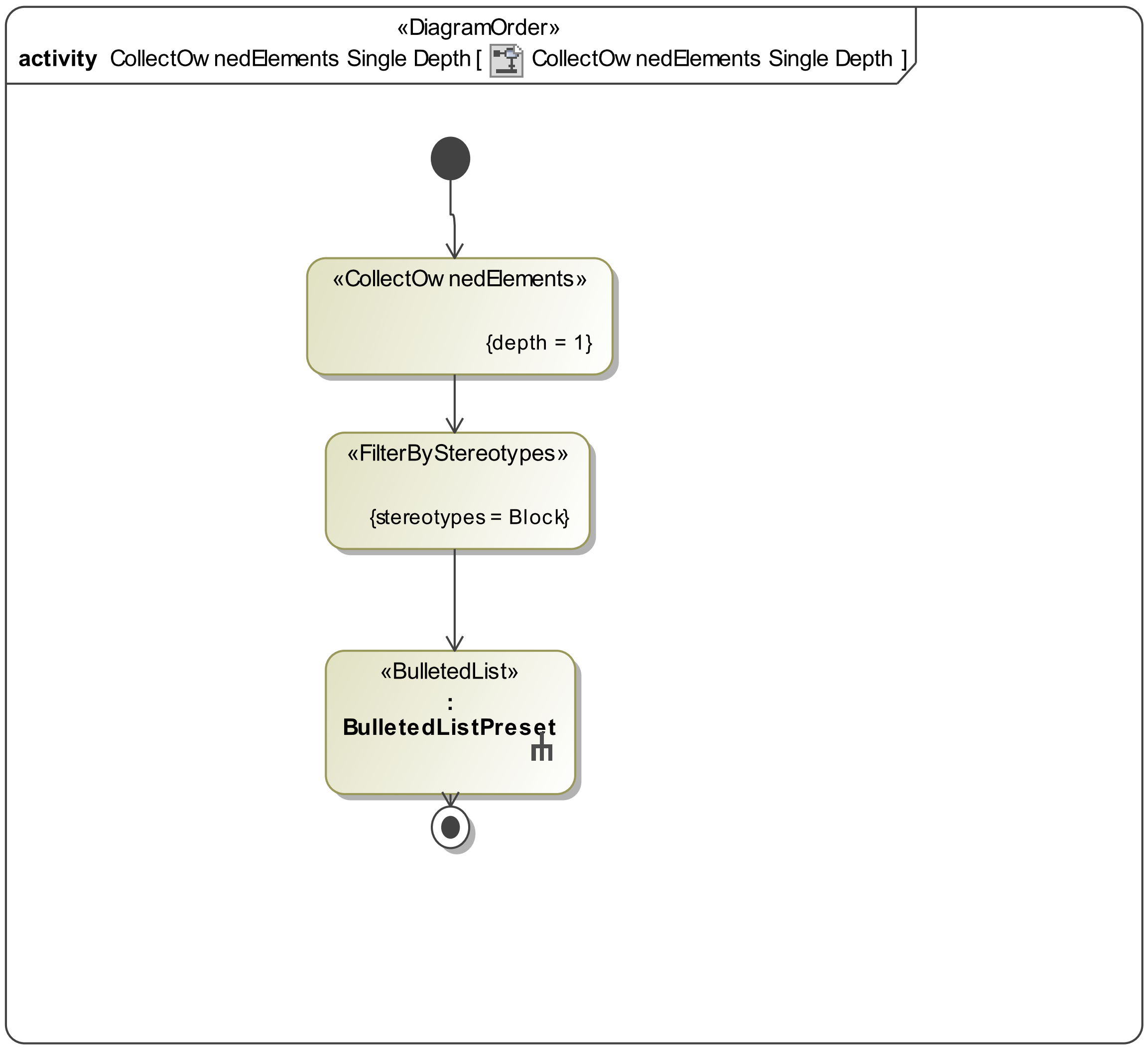
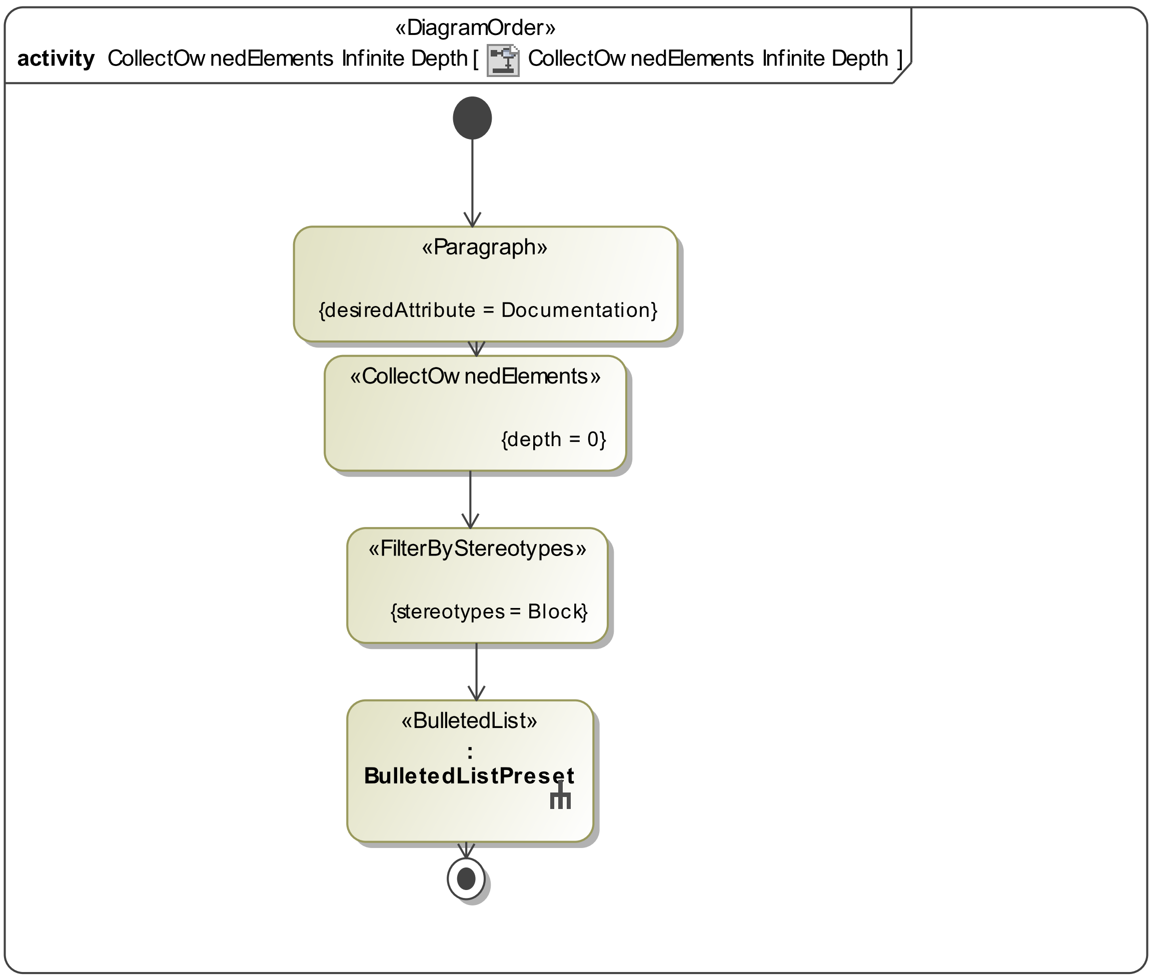


Figure 4.4. CollectOwnedElements Infinite Depth



Single Depth Example

In this example, we expose the owned elements only at a depth of one. The result is in the following bulleted list.

1. Sandwich Bag
2. Thermos
3. Bag of Chips
4. Cake

Infinite Depth Example

1. Sandwich Bag
2. Sandwich
3. Thermos
4. Bag of Chips
5. Cake
6. Nail File

CollectOwners

"CollectOwners" is essentially the opposite of "CollectOwnedElements". It gathers all of the owners (from the containment tree) of the exposed element(s) for use in the viewpoint method. Using a different block "Target", this returns the path of ownership in the containment tree. It starts with the direct ownership package (Example Elements) and progresses back to the entire model (Data) because the "DepthChoosable" was set at 0 (infinite).

<mms-view-link> [cf:Example View.vlink] </mms-view-link>
shows the output of the following diagrams.

Figure 4.5. Example View Diagram

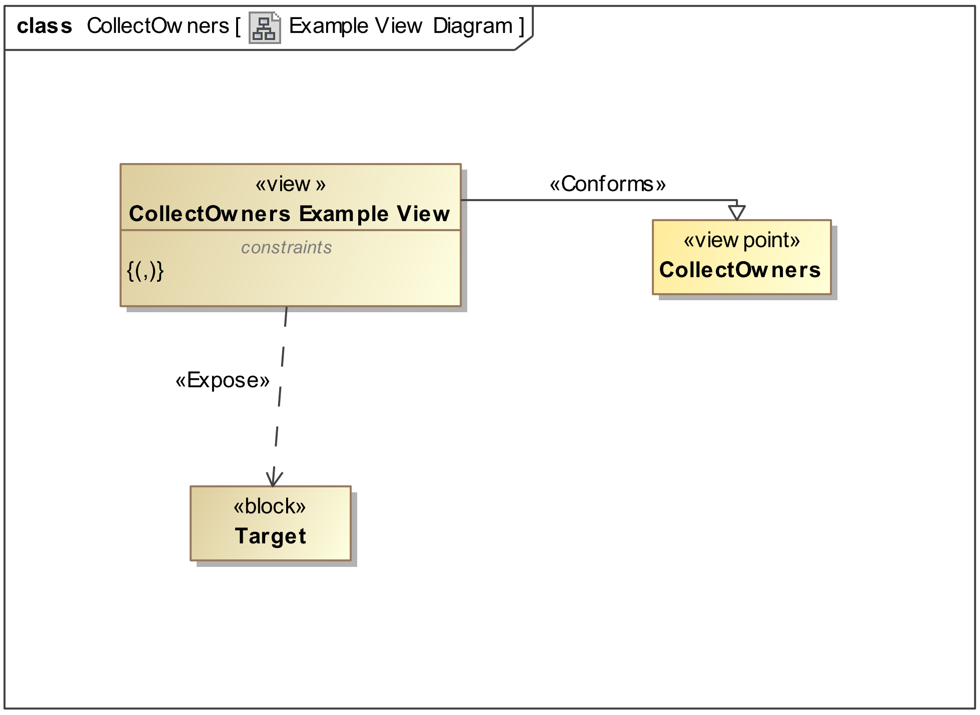


Figure 4.6. CollectOwners

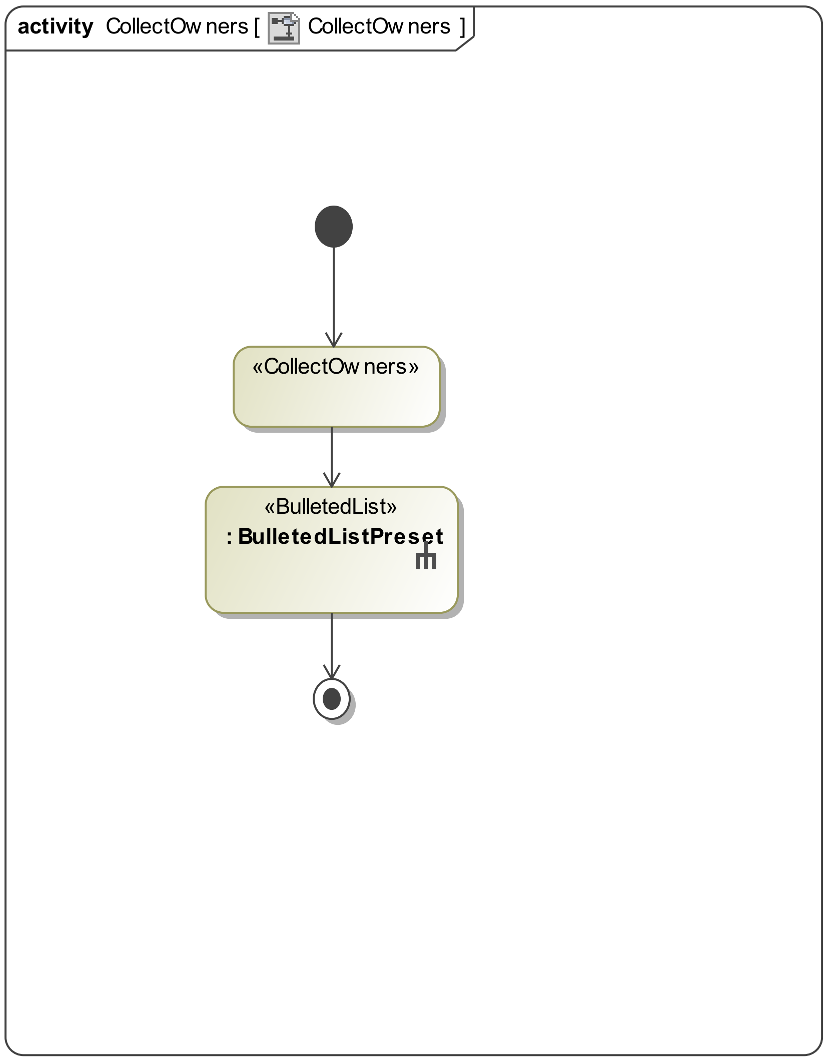
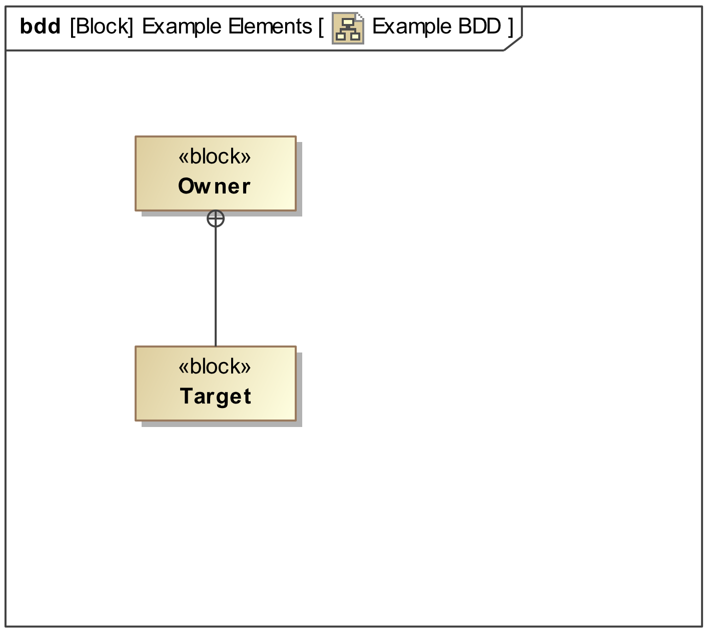


Figure 4.7. Example BDD



CollectOwners Example View

1. Owner
2. Example Elements
3. CollectOwners
4. Collect
5. Create Viewpoint Methods Examples
6. Models
7. DocGen Manual
8. Data

CollectThingsOnDiagram

CollectThingsOnDiagram will collect all the elements depicted on a diagram. To demonstrate this, the <mms-transclude-name> [cf:Example BDD -.name] </mms-transclude-name> was exposed. Note that the name of each element as well as the viewpoint method itself is listed below. Names are often not automatically created with an element, so these must be inserted in the model. Otherwise a "no content for..." message will appear in the name's spot.

Figure 4.8. Example View Diagram

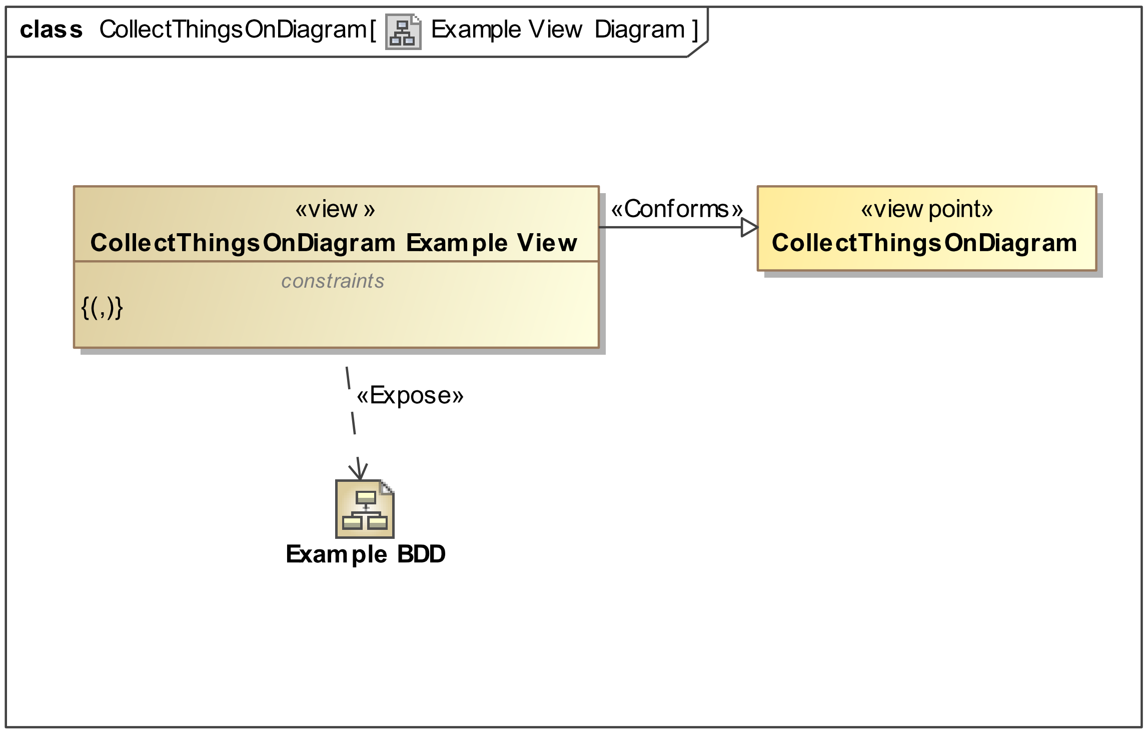

class CollectThingsOnDiagram [Example View Diagram]
«view»
CollectThingsOnDiagram Example View
constraints
{(,)}
«Conforms»
«view point»
CollectThingsOnDiagram
«Expose»
Example BDD

Figure 4.9. CollectThingsOnDiagram

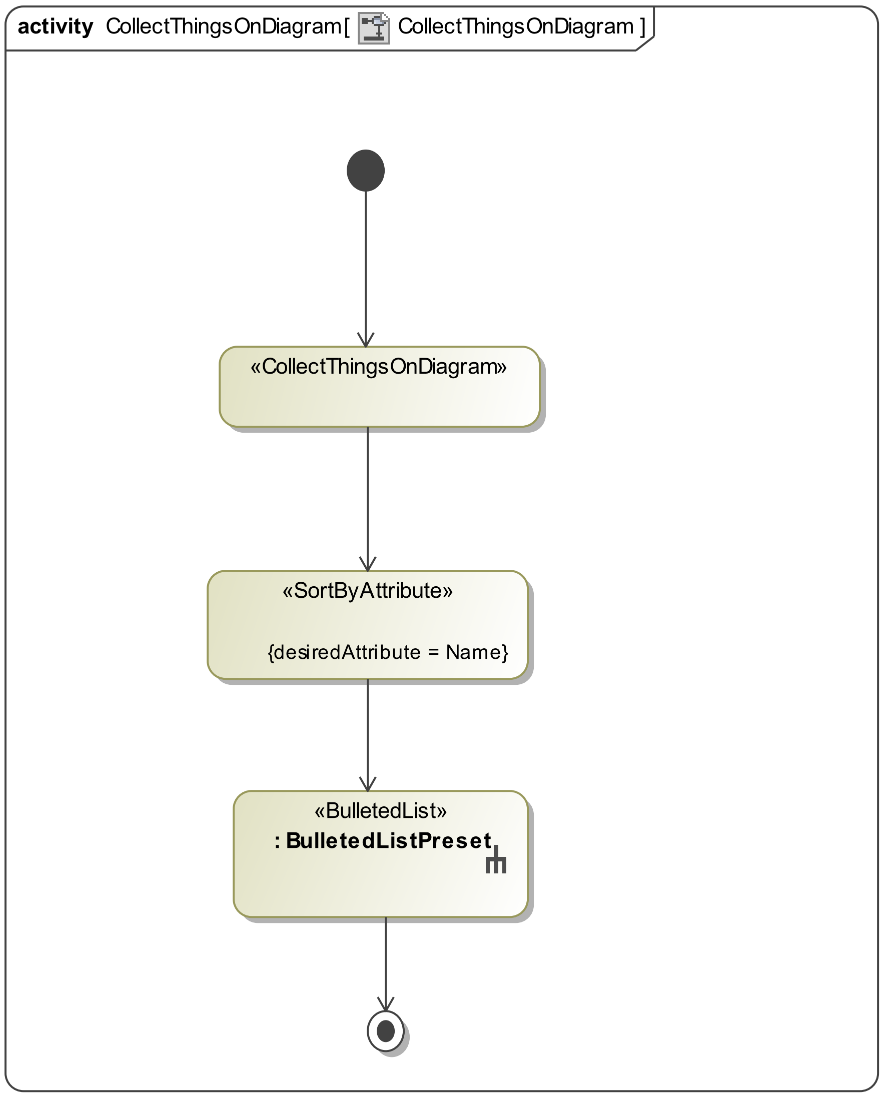
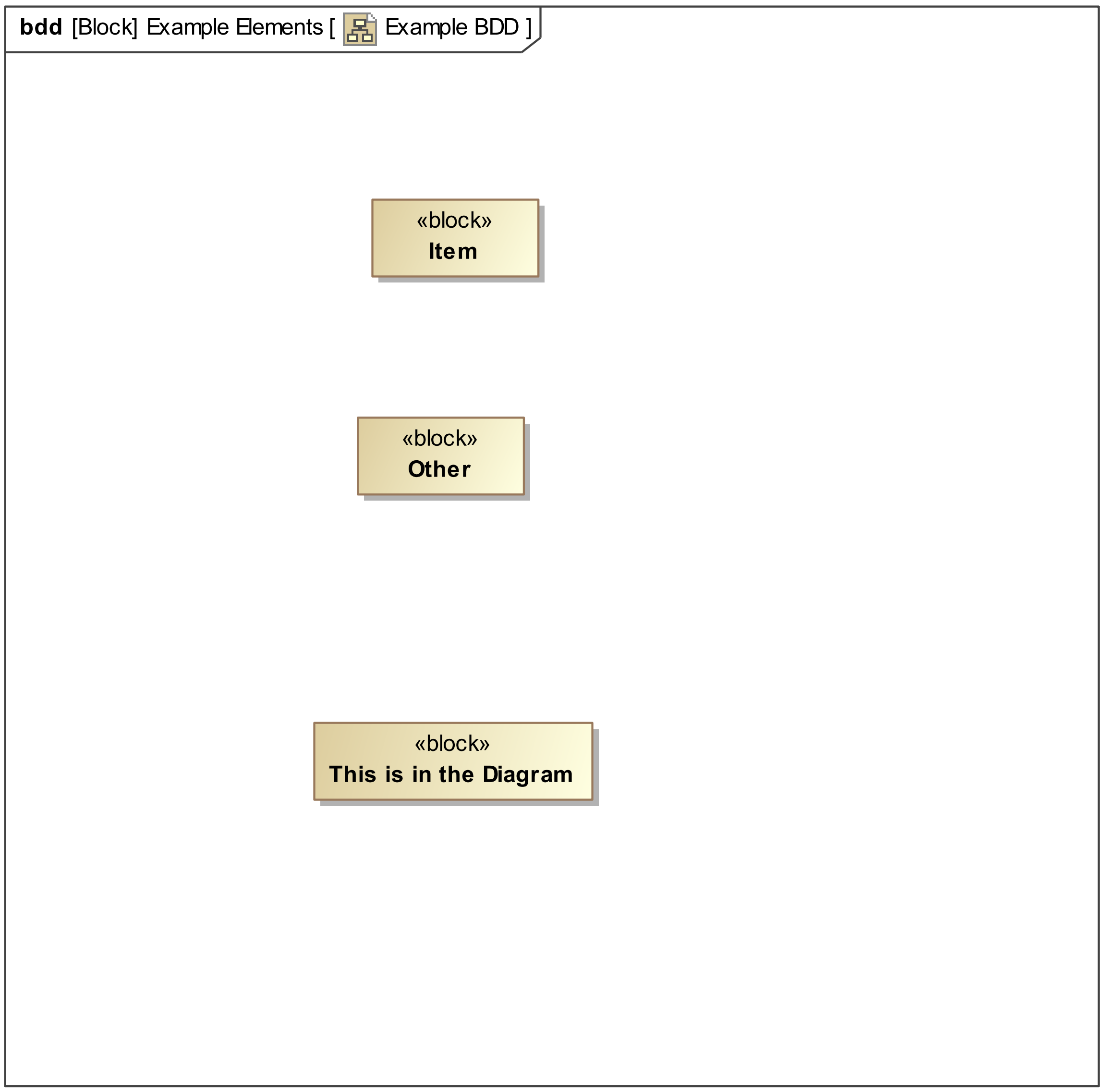

activity CollectThingsOnDiagram[CollectThingsOnDiagram]
«CollectThingsOnDiagram»
«SortByAttribute»
{desiredAttribute = Name}
«BulletedList»
: BulletedListPreset

Figure 4.10. Example BDD



Another element not shown on this diagram will not be picked up by this method.

CollectThingsOnDiagram Example View

1. Example BDD
2. Item
3. Other
4. This is in the Diagram

CollectByStereotypeProperties

This collection action allows a user to get at a specified property value(s) of a stereotype. This might be helpful when multiple different values are specified for a single stereotype property. It is important to note that this functionality does not work with string values and will return only one element value. In other words, if there are duplicate values for the same property, the duplicates will be removed from the returned values set. The list below shows the elements found in the "Example Elements" package. Several of these characters have the stereotype <<Human>>. This particular stereotype has a property (orange dot) named "Gender" and is typed "Element" (also shown below).

The character "Billy" has the <<Human>> stereotype and you would like to know the value of this stereotype's property "Gender". By exposing the list above to the activity diagram diagram below, you are able to extract the values of all characters stereotyped <<Human>> who's value has been specified. Notice, in the activity below, the CollectByStereotypeProperties action is specified to collect only the property "Gender". In this example ONLY Billy's gender value has been set, the rest of the characters were left the same, thus, the expected outcome should be one value.

CollectByStereotypeProperties collects the properties of the stereotypes. The following views
<mms-view-link> [cf:Bulleted List Example.vlink] </mms-view-link>
and
<mms-view-link> [cf:Table Structure Example.vlink] </mms-view-link>
show the output of the view structure described in the Example View Diagram below.

Figure 4.11. Example View Diagram

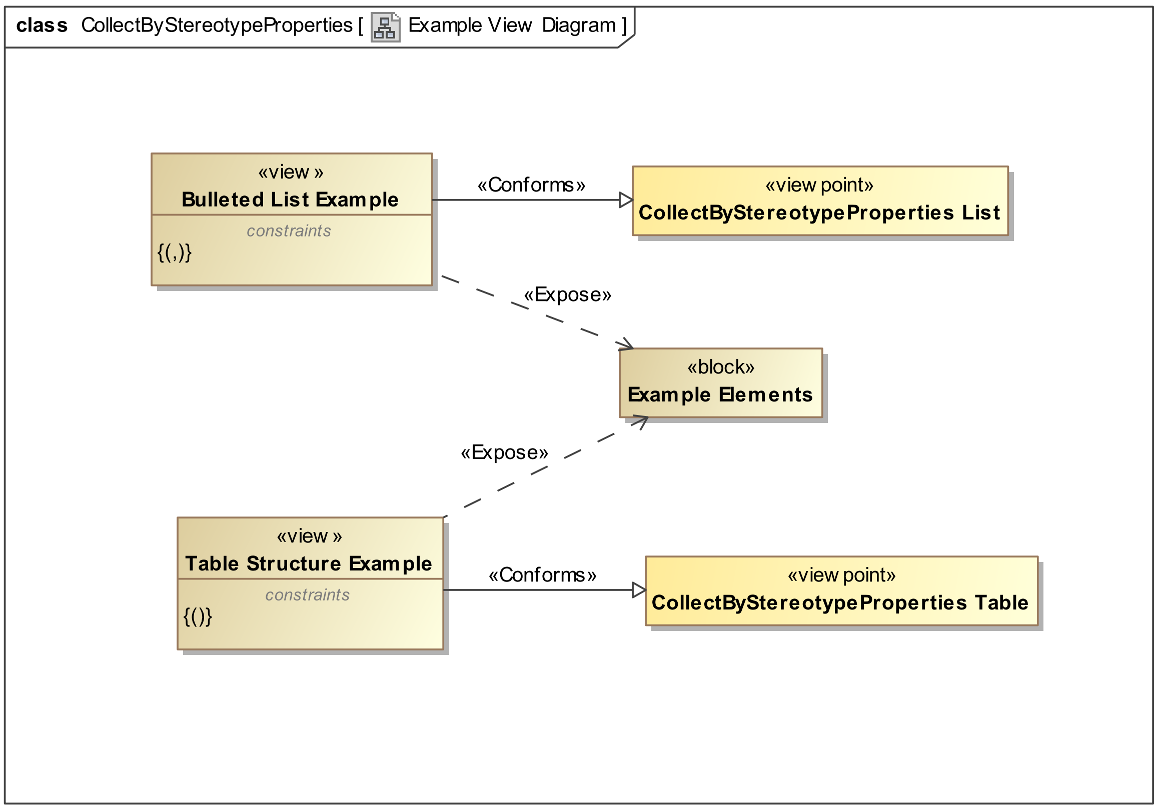


Figure 4.12. CollectByStereotypeProperties List

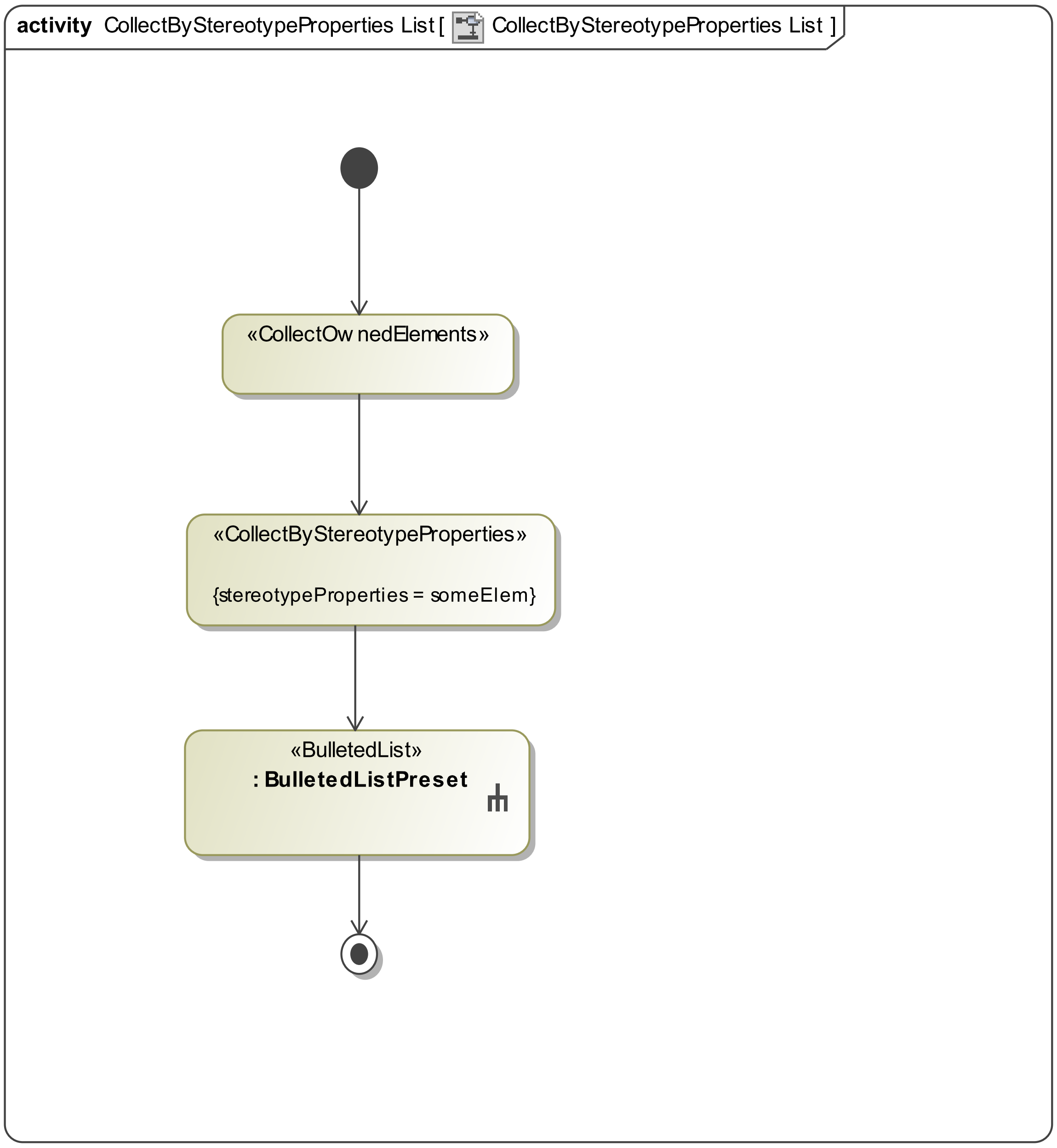

activity CollectByStereotypeProperties List [CollectByStereotypeProperties List]
«CollectOwnedElements»
«CollectByStereotypeProperties»
{stereotypeProperties = someElem}
«BulletedList»
: BulletedListPreset

Figure 4.13. Example BDD

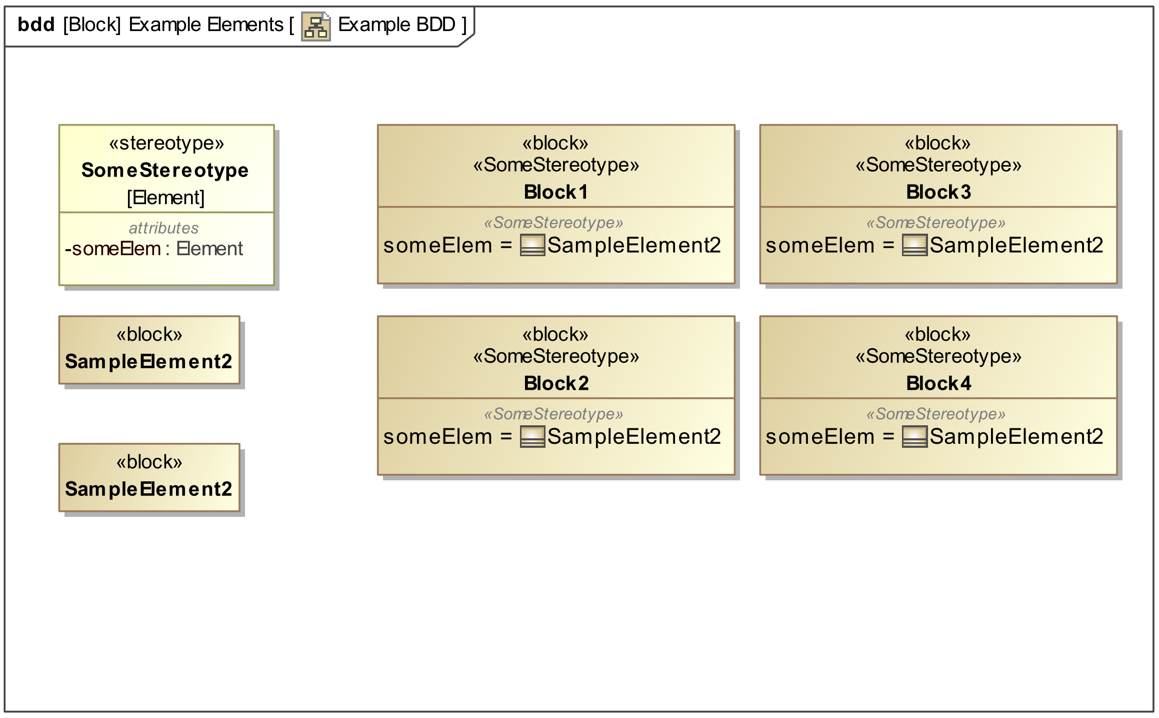
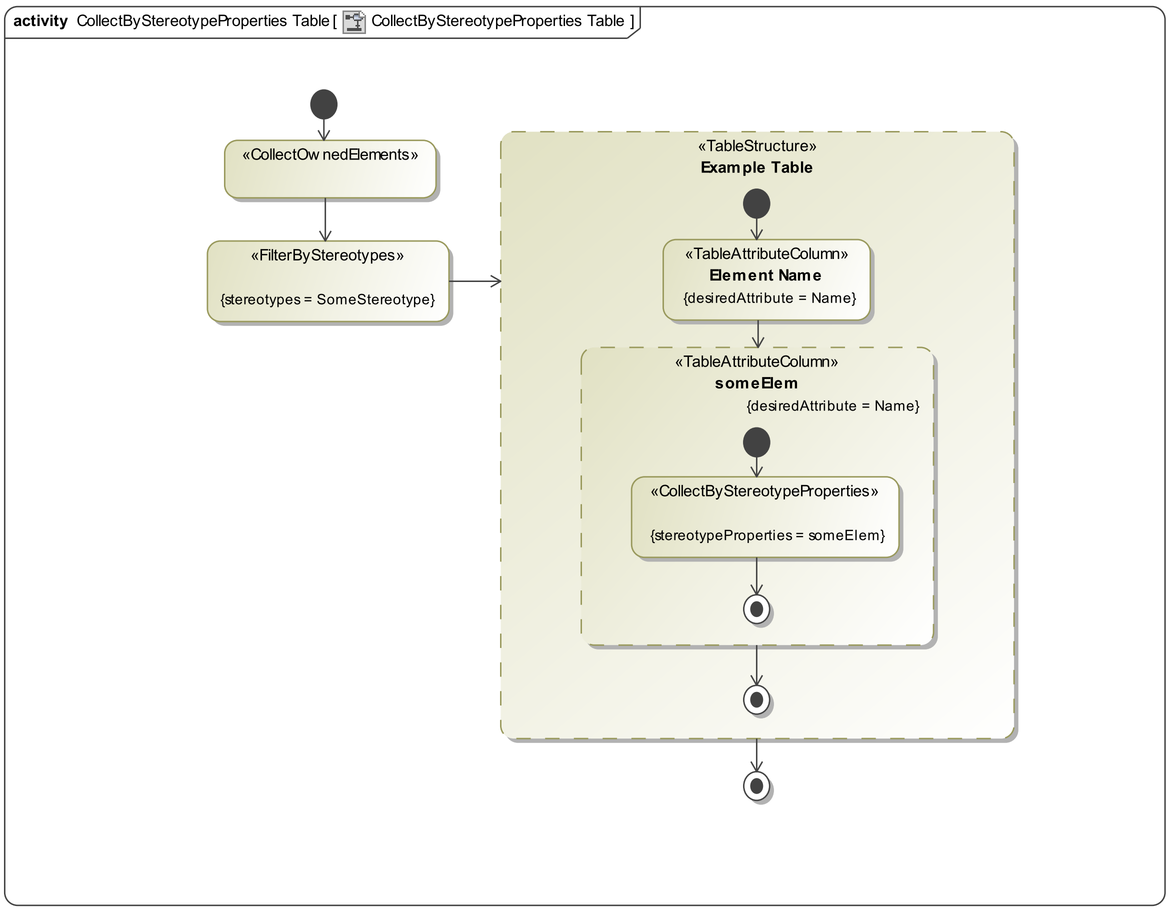


Figure 4.14. CollectByStereotypeProperties Table


activity CollectByStereotypeProperties Table [CollectByStereotypeProperties Table]
«CollectOwnedElements»
«FilterByStereotypes»
{stereotypes = SomeStereotype}
«TableStructure»
Example Table
«TableAttributeColumn»
Element Name
{desiredAttribute = Name}
«TableAttributeColumn»
someElem
{desiredAttribute = Name}
«CollectByStereotypeProperties»
{stereotypeProperties = someElem}

Bulleted List Example

1. SampleElement2
2. SampleElement2

Table Structure Example

Table 4.1. Example Table

Element Name	someElem
Block1	SampleElement2
Block2	SampleElement2
Block3	SampleElement2
Block4	SampleElement2

CollectByDirectedRelationshipMetaclasses

CollectByDirectedRelationshipMetaclasses action collects elements based on the relationships that other elements use to connect to them. Note that in the example, Helo and Athena are connected to Hera with a "Dependency" (the dashed arrowed line). In other words, Hera depends on Helo and Athena. These are the only dependencies in the example, so if I expose the Example Elements package, collect the owned elements, and then use this operator with the metaclass "Dependency", only Helo and Athena will be collected. Note the origin of the dependency - Hera - is NOT collected with this operator.

Selecting [...] under MetaclassChoosable in the specification will open the element selector, allowing the selection of the desired metaclass. Make sure the metaclass option is selected in the lower left corner or else metaclasses will not show in the search. Also keep in mind that depending on the extensions, plug-ins, etc. that you are using in Magic Draw, there might be multiple metaclasses with the same name (i.e. the built-in one and additional custom ones). Be careful to select the metaclass you are using in the model. The final results of this operator are shown in the following views.

Figure 4.15. Example View Diagram

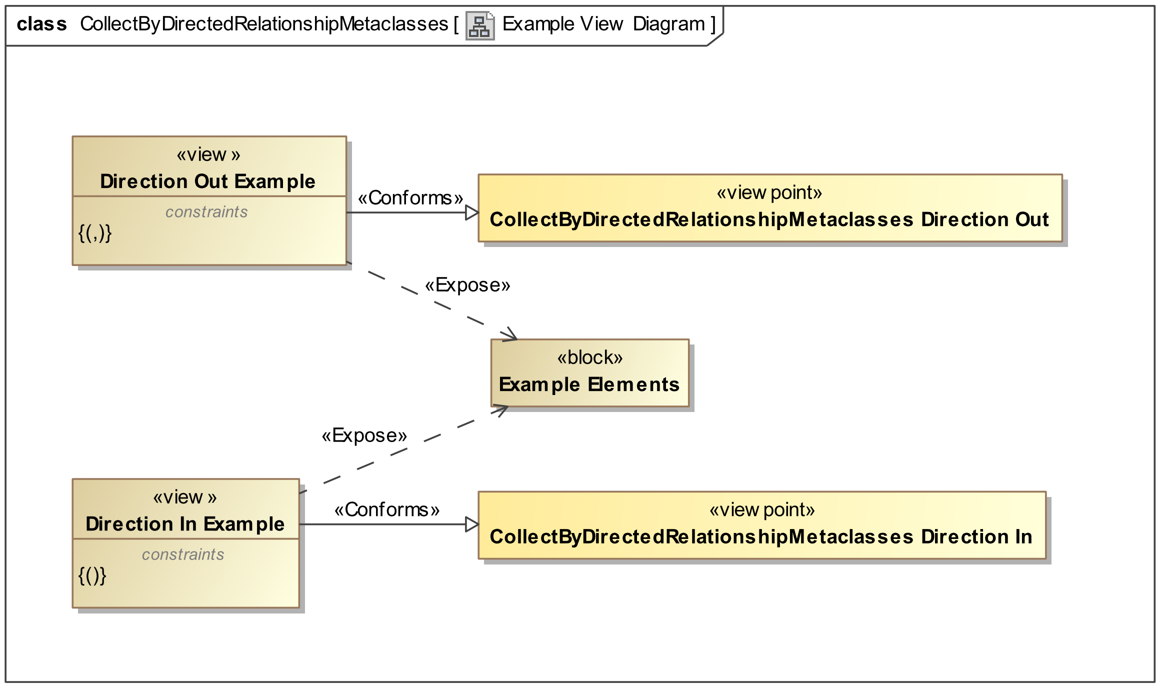

class CollectByDirectedRelationshipMetaclasses [Example View Diagram]
«view »
Direction Out Example
constraints
{(,)}
«Conforms»
«view point»
CollectByDirectedRelationshipMetaclasses Direction Out
«Expose»
«block»
Example Elements
«Expose»
«view »
Direction In Example
constraints
{()}
«Conforms»
«view point»
CollectByDirectedRelationshipMetaclasses Direction In

Figure 4.16. CollectByDirectedRelationshipMetaclasses Direction Out

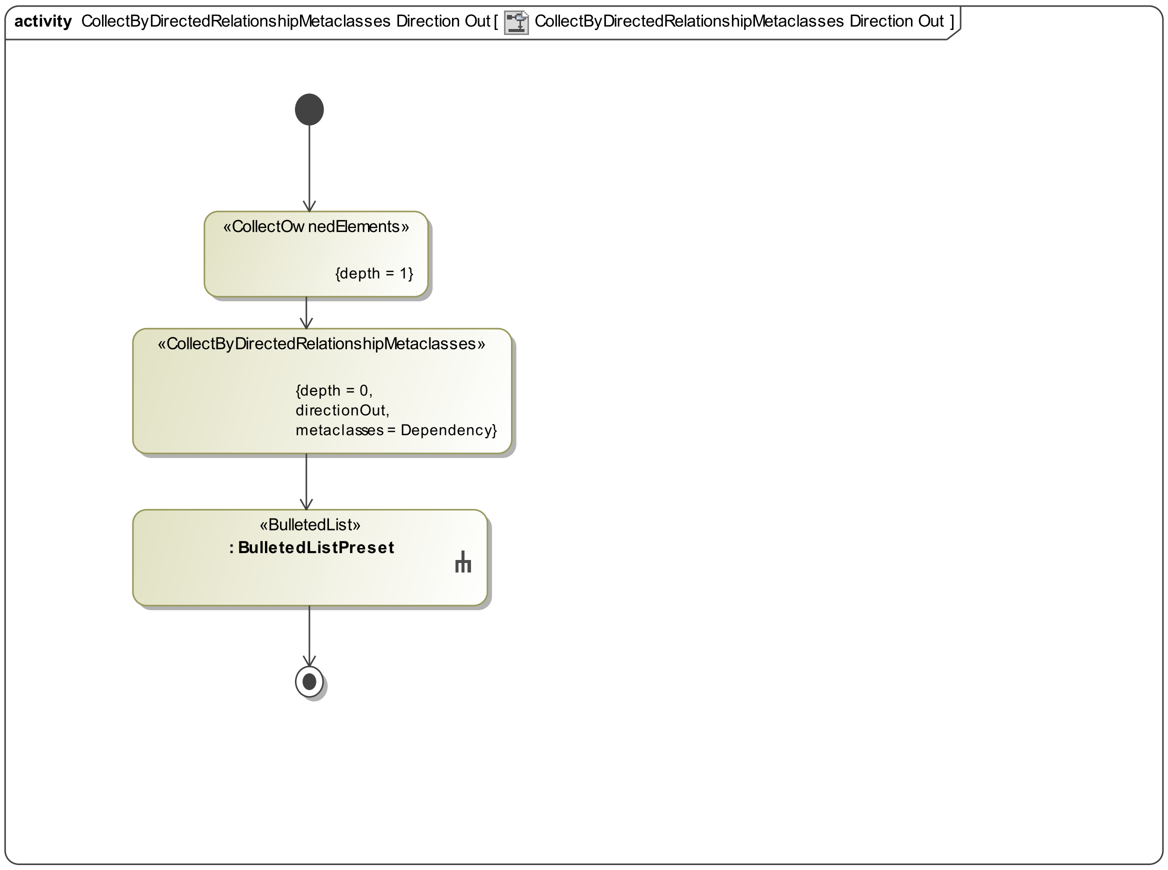

activity CollectByDirectedRelationshipMetaclasses Direction Out [CollectByDirectedRelationshipMetaclasses Direction Out]
«CollectOwnedElements»
{depth = 1}
«CollectByDirectedRelationshipMetaclasses»
{depth = 0, directionOut, metaclasses = Dependency}
«BulletedList»
: BulletedListPreset

Figure 4.17. Example BDD

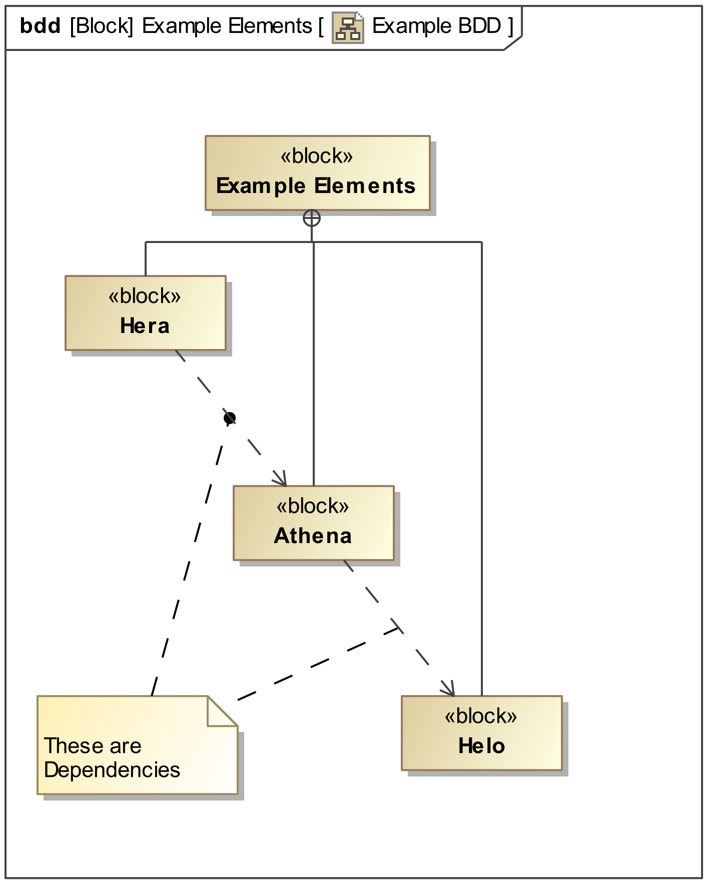
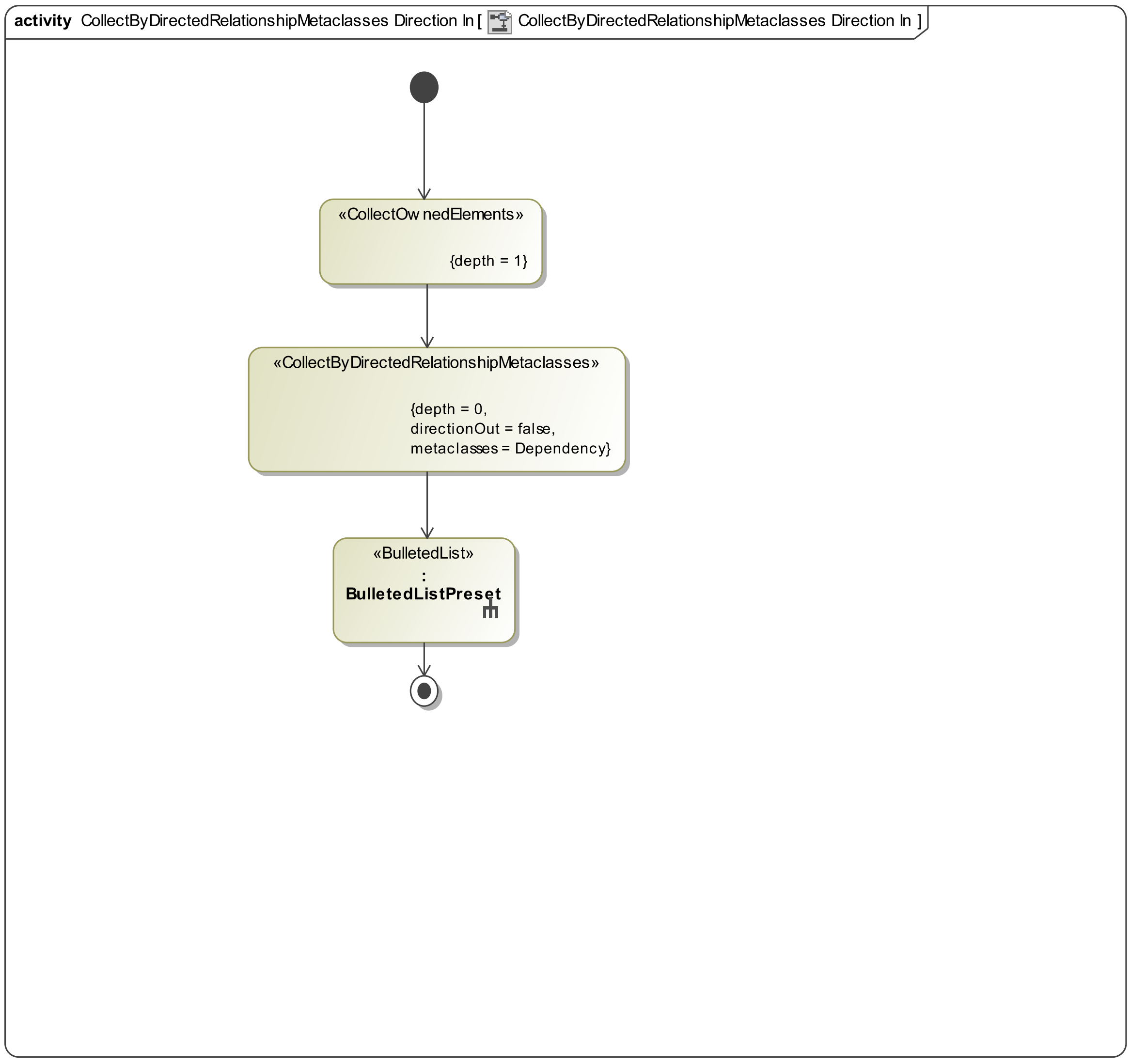


Figure 4.18. CollectByDirectedRelationshipMetaclasses Direction In


activity CollectByDirectedRelationshipMetaclasses Direction In [CollectByDirectedRelationshipMetaclasses Direction In]
«CollectOwnedElements»
{depth = 1}
«CollectByDirectedRelationshipMetaclasses»
{depth = 0,
directionOut = false,
metaclasses = Dependency}
«BulletedList»
: BulletedListPreset

Direction Out Example

1. Athena
2. Helo

Direction In Example

1. Hera
2. Athena

Direction Out

In the previous part of this example, the specification has the option "Direction Out" as undefined. This is the default setting but once you click on it you can chose between true and false. The default value, corresponding to undefined, is "true" and produces the results in the previous part of this example. This part of the example shows what happens when "Direction Out" is set to false.

When a directed relationship is used, one block is the "source" of the relationship and the other is the "target". The target can be thought of as the block with the "arrowhead" portion of the relationship. However, there is another set of terms where the block with the "arrowhead" is referred to as the "supplier" and the other end is referred to as the "client". When "Direction Out" was set to true, the "target" (also known as "supplier") values were returned. Thus logically, when "Direction Out" is set to false, the "source" (also known as "client") values should be returned. The directionality of the dependency relationship can be confusing for new users and it may be helpful to instead think of it in terms which are dependent and which are independent. The blocks at the arrowhead (target/supplier) end of the dependency relationship are independent while the blocks at the end with no arrow (source/client) are dependent. In this case, we would expect Hera, the dependent block, to be on this list.

However, if you look at the output list at the bottom of this page, more elements than just Hera are returned. What is really odd is that these elements correspond to sections of this document instead of the example character blocks we had been working with. This is due to how the model software conducts its search and we will explain below. It unfortunately makes this a complicated example, but this is an important point to understand. It will be especially helpful if you find yourself trying to troubleshoot a situation where the elements returned are not what you expected.

First if you recall in the initial discussion of how to set up a view point, the section
<mms-transclude-name> [cf:Link to View and Expose Content.name] </mms-transclude-name>
explains that you choose the section of the model you want to work with. In this example we were working with the "Example Elements" package. However, the examples for
<mms-transclude-name> [cf:CollectOwnedElements.name] </mms-transclude-name>
and
<mms-transclude-name> [cf:CollectOwners.name] </mms-transclude-name>
only exposed the Eights block from the Example Elements package.

For the first part of this collect by directed relationship metaclass example (the simple part), the only elements with a complete dependency relationship within the "Example Elements" package were Athena, Helo, and Hera. However, some other elements, such as the Eights block, had dependency relationships that were not completely contained in the Example Elements package. It was previously mentioned that the "expose" relationship is a stereotype of the "dependency" relationship. Thus since we had chosen the dependency relationship for this example, the dependency stereotypes, such as "expose", also meet the search criteria. When the viewpoints for the previous examples "exposed" the Eights block to create the portions of this document, they also created dependency relationships.

Now the next question is why parts of those relationships across the package boundary only appeared in this list output and not earlier when the "Direction Out" was "true". That requires an understanding of how the model conducts its search.

Search Steps

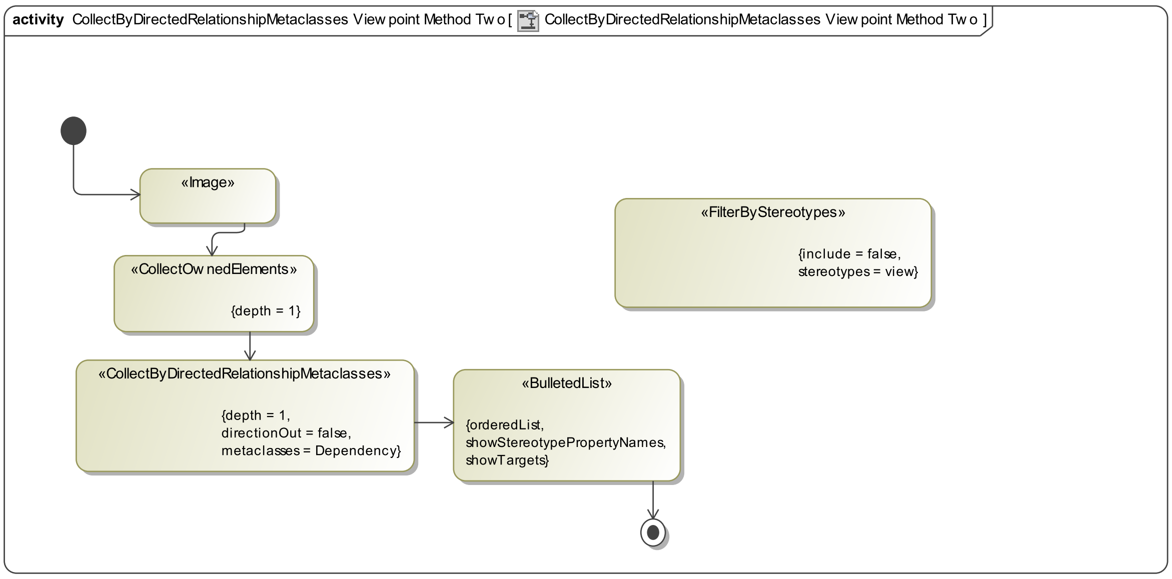
1. Within the content that is exposed by the viewpoint, the CollectByDirectedRelationshipMetaclasses action searches for the contained elements.
2. It then checks the relationships associated with the elements.
3. Next, it checks the direction of those relationships to see if it corresponds with the "Direction Out" value that you had specified.
4. The last step is that the search follows the relationship and outputs what is at the other end of the relationship.

When "Direction Out" was set to "true", the search went through the "Example Elements" package, found all the blocks with dependency relationships for which the block was the "source" of the relationship, followed the relationship connection, and listed the "target" elements.

When the "Direction Out" is set to "false", the search went through the "Example Elements" package, found all the blocks with dependency relationships for which the block was the "target" of the relationship, followed the relationship connection back, and listed the "source" elements. In this case, since the Eights block is listed as the "target" of the various "expose" dependency relationships, the search followed the connection, even though it was outside the "Example Elements" package. That is why the list at the bottom of the screen has entries other than just Hera as we would expect.

If you just wanted the elements within the "Example Elements" package, you could create a filter like the one shown in the image below. If the filter were to be connected, then the list would just output Hera.

Figure 4.19. CollectByDirectedRelationshipMetaclasses Viewpoint Method Two


activity CollectByDirectedRelationshipMetaclasses View point Method Tw o [CollectByDirectedRelationshipMetaclasses View point Method Tw o]
«Image»
«CollectOw nedElements»
{depth = 1}
«CollectByDirectedRelationshipMetaclasses»
{depth = 1, directionOut = false, metaclasses = Dependency}
«BulletedList»
{orderedList, showStereotypePropertyNames, showTargets}
«FilterByStereotypes»
{include = false, stereotypes = view}

CollectByDirectedRelationshipStereotypes

CollectByDirectedRelationshipStereotypes also collects elements based on the relationships that connect them to other elements. However, here the collection is by the stereotype of the relationship instead of the metaclass (as in the previous section).

Figure 4.20. Example View Diagram

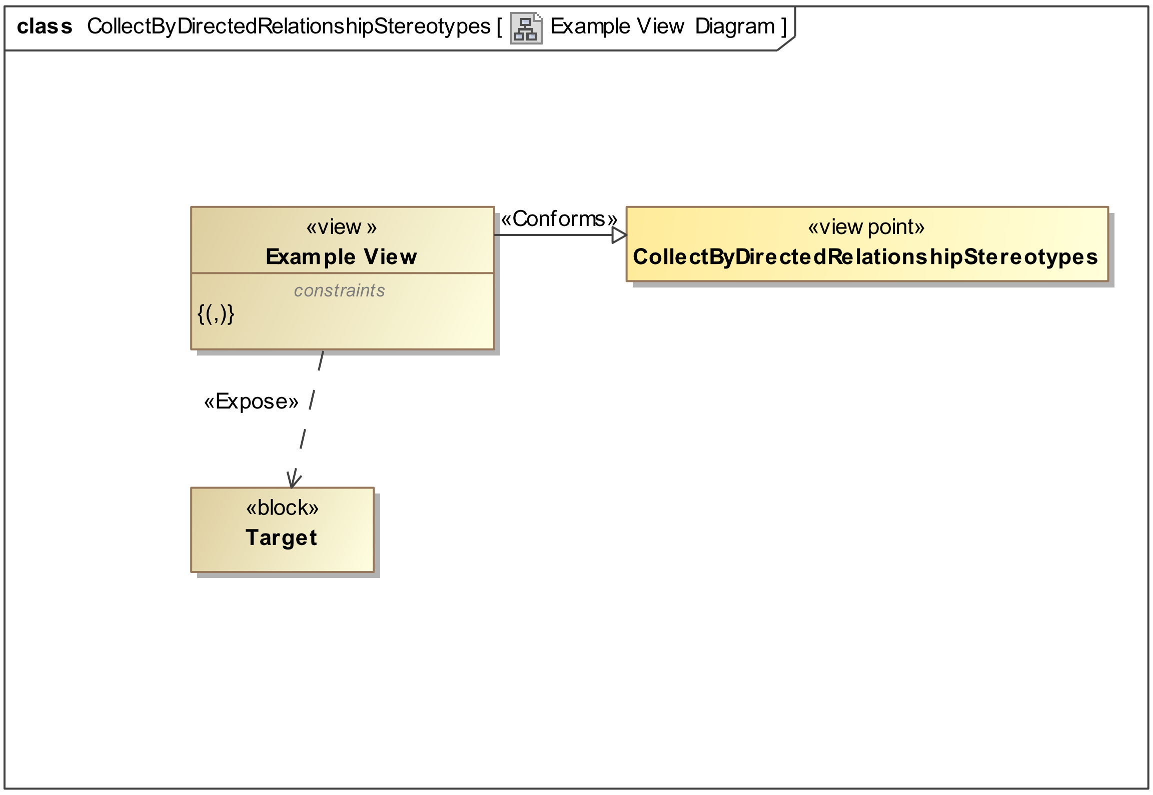

class CollectByDirectedRelationshipStereotypes [Example View Diagram]
«view »
Example View
constraints
{(,)}
«Conforms»
«view point»
CollectByDirectedRelationshipStereotypes
«Expose»
«block»
Target

Figure 4.21. Example BDD

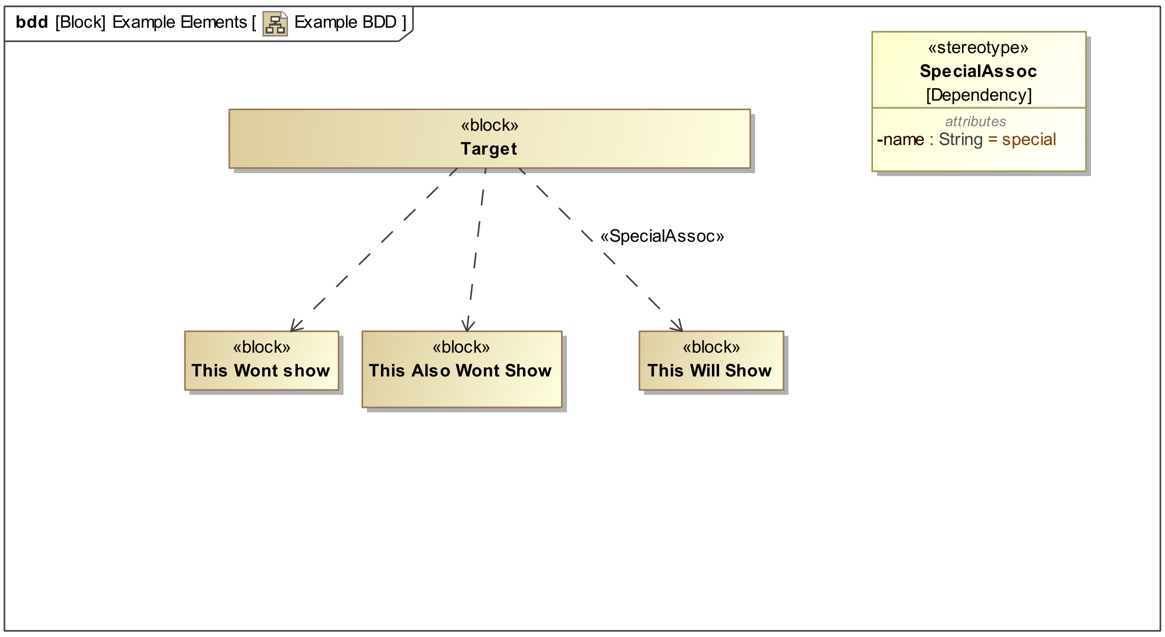
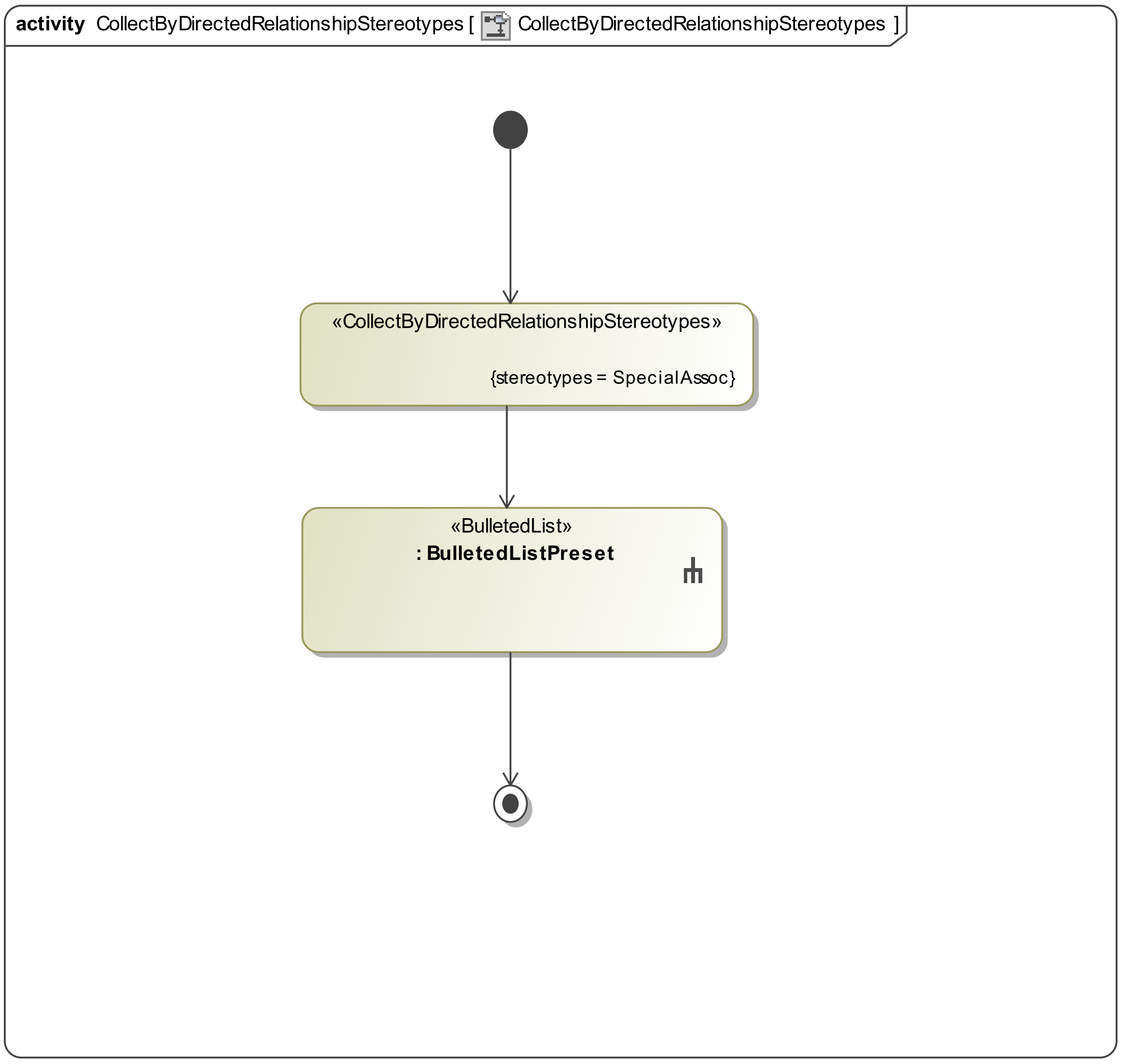


Figure 4.22. CollectByDirectedRelationshipStereotypes


activity CollectByDirectedRelationshipStereotypes [CollectByDirectedRelationshipStereotypes]
«CollectByDirectedRelationshipStereotypes»
{stereotypes = SpecialAssoc}
«BulletedList»
: BulletedListPreset

Example View

1. This Will Show

CollectByAssociation

CollectByAssociation collects the blocks with aggregation of either composite, shared, or none. In other words, both white diamond associations (shared) and black diamond associations (composite) can be collected by this action. The CollectByAssociation action will then collect only those blocks that have the aggregation type as noted in the specification for CollectByAssociation block.

In the view diagram, this view exposes the
<mms-transclude-name> [cf:Sample Element.name] </mms-transclude-name>
and conforms to the CollectByAssociation viewpoint method. The result of this viewpoint method is shown in the following view.

Figure 4.23. Example View Diagram

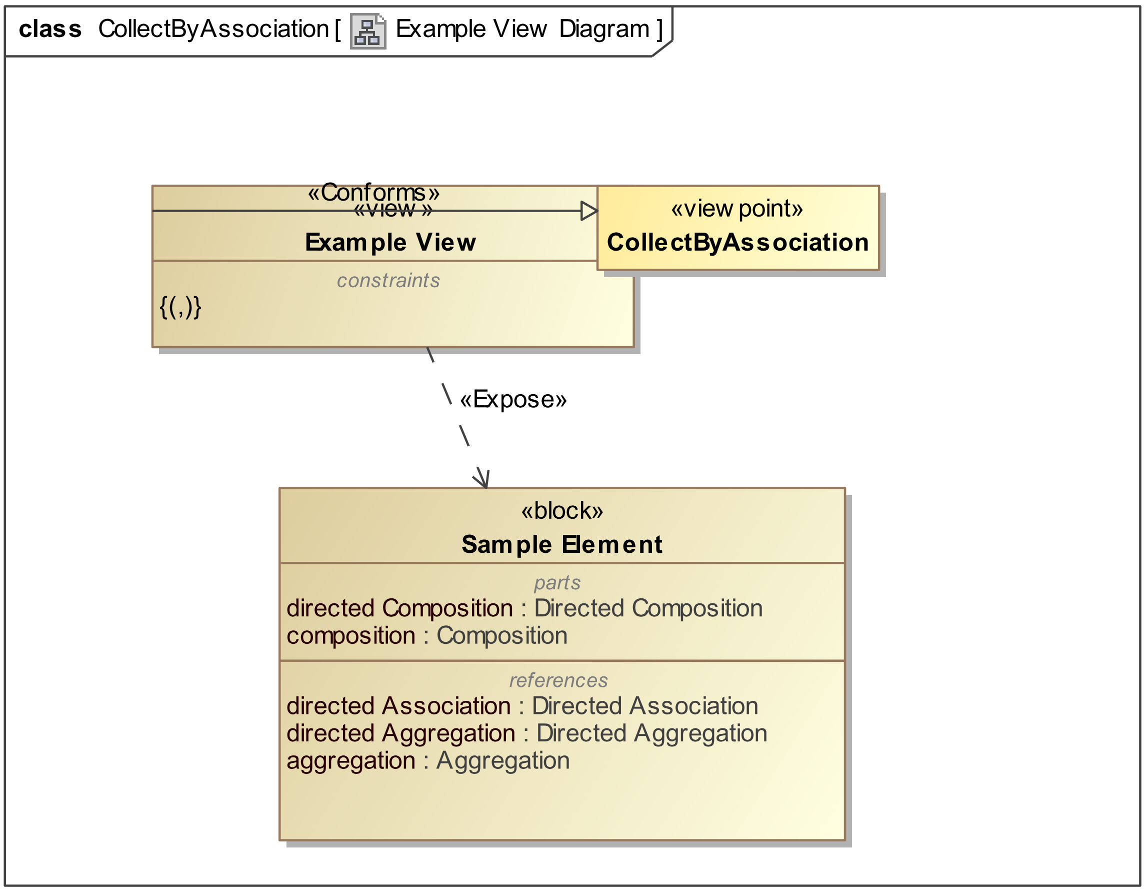

class CollectByAssociation [Example View Diagram]
«Conforms»
«view»
Example View
constraints
{(,)}
«view point»
CollectByAssociation
«Expose»
«block»
Sample Element
parts
directed Composition : Directed Composition
composition : Composition
references
directed Association : Directed Association
directed Aggregation : Directed Aggregation
aggregation : Aggregation

Figure 4.24. CollectByAssociation

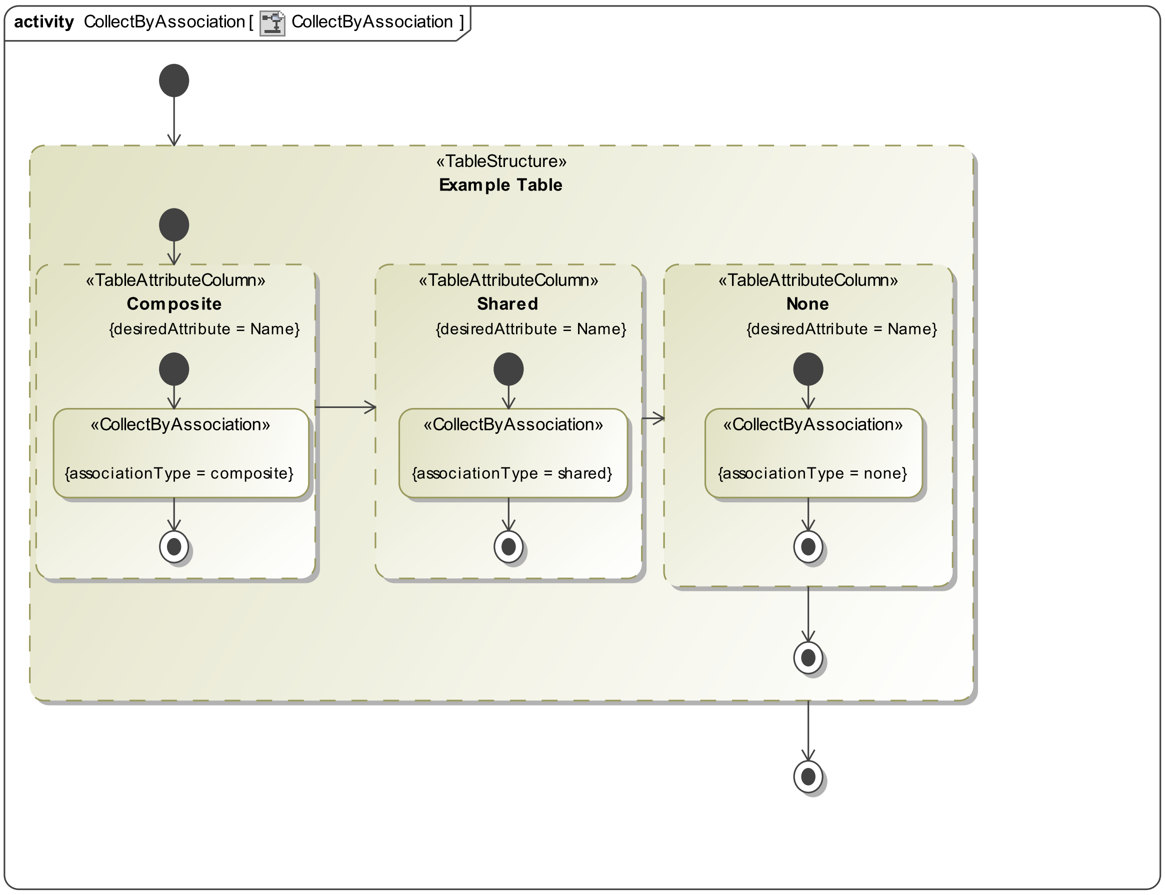
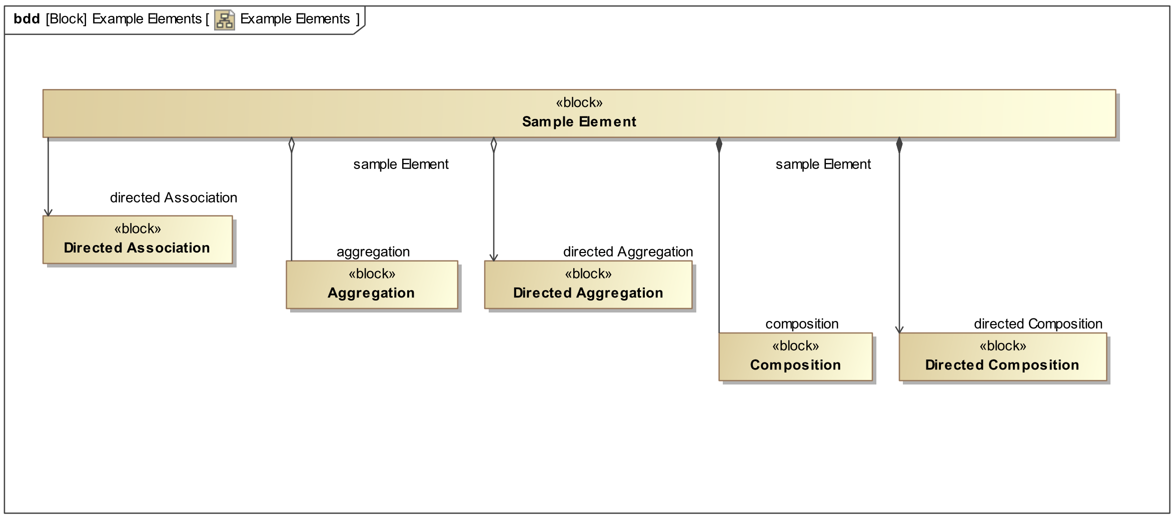


Figure 4.25. Example Elements



Example View

Table 4.2. Example Table

Composite	Shared	None
Directed Composition	Directed Aggregation	Directed Association

Composite	**Shared**	**None**
Composition	Aggregation	

CollectTypes

CollectTypes collects types. In this case, the CollectOwnedElements or CollectOwners needs to be used to collect elements before CollectTypes. Then, CollectTypes will collect the type associated with each element. The viewpoint method diagram is shown below.

Figure 4.26. Example View Diagram

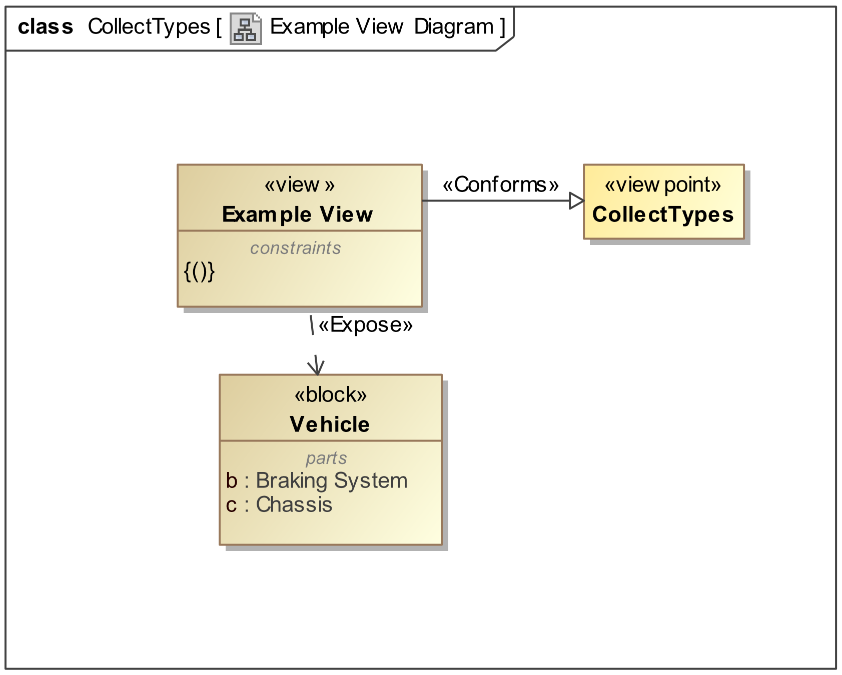


Figure 4.27. CollectTypes

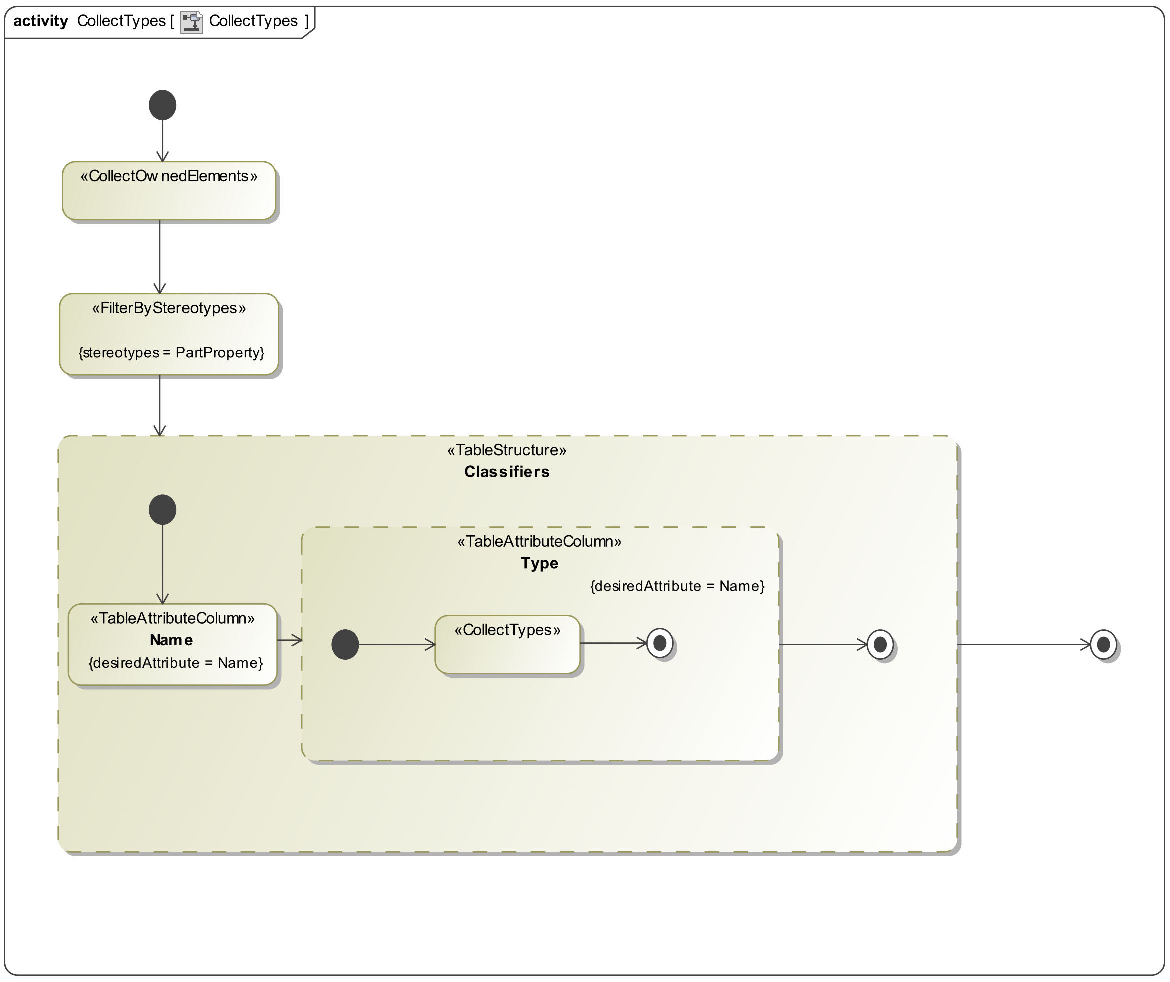
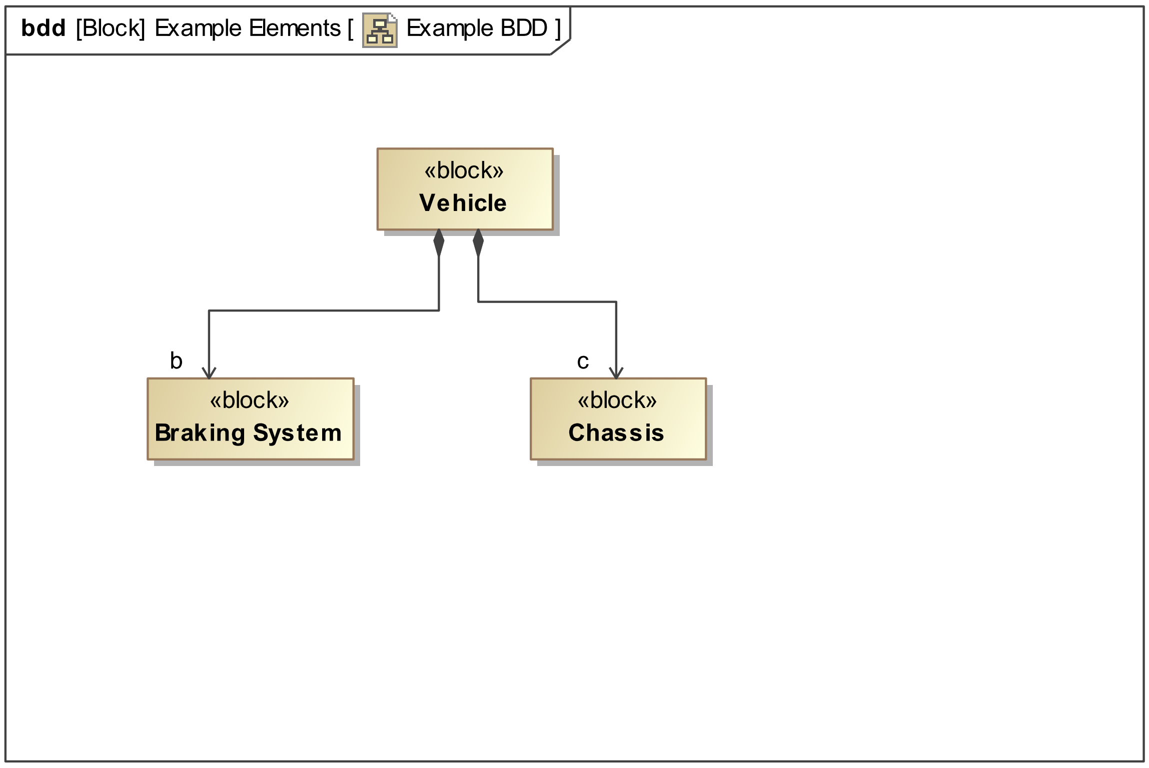


Figure 4.28. Example BDD



Example View

Table 4.3. Classifiers

Name	Type
b	Braking System
c	Chassis

CollectClassifierAttributes

CollectClassifierAttributes collects attributes of a class. To use this operation, we do not need to use CollectOwnedElements or CollectOwners, unlike in previous sections. The results are displayed in the following Example View table.

Figure 4.29. Example View Diagram

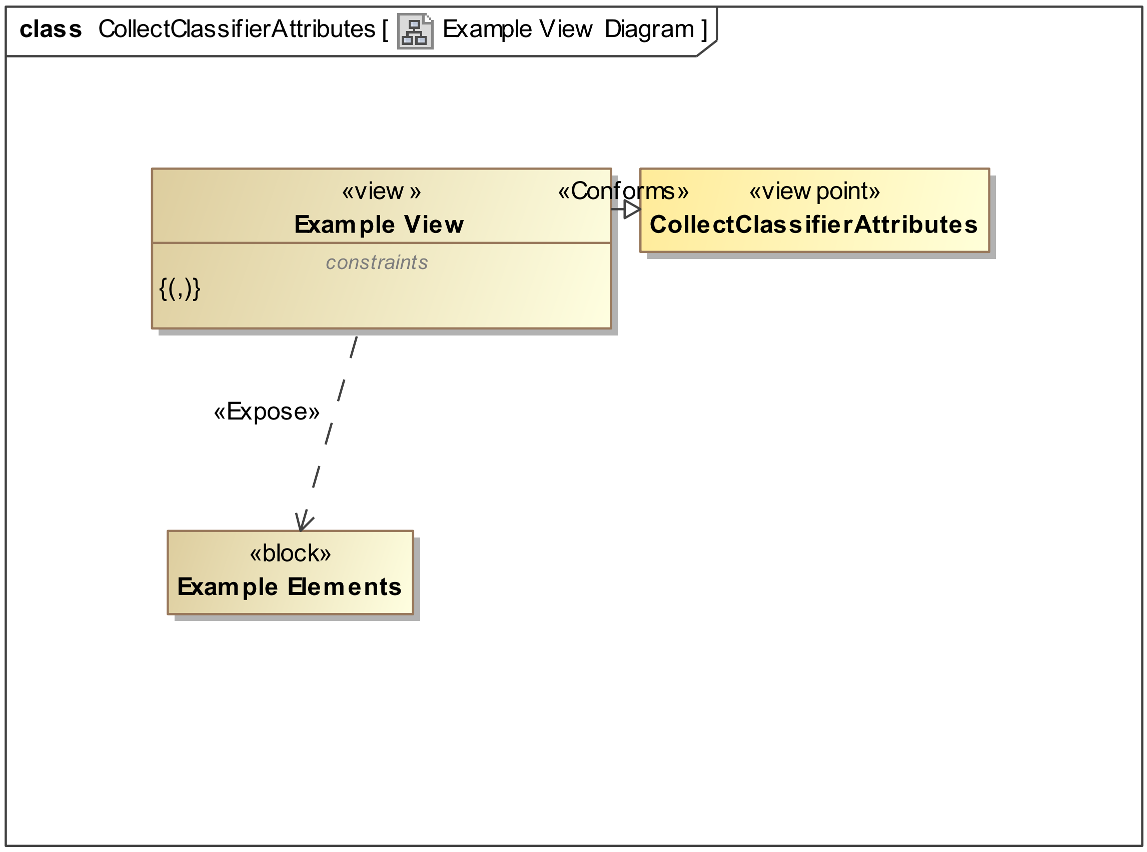


Figure 4.30. CollectClassifierAttributes

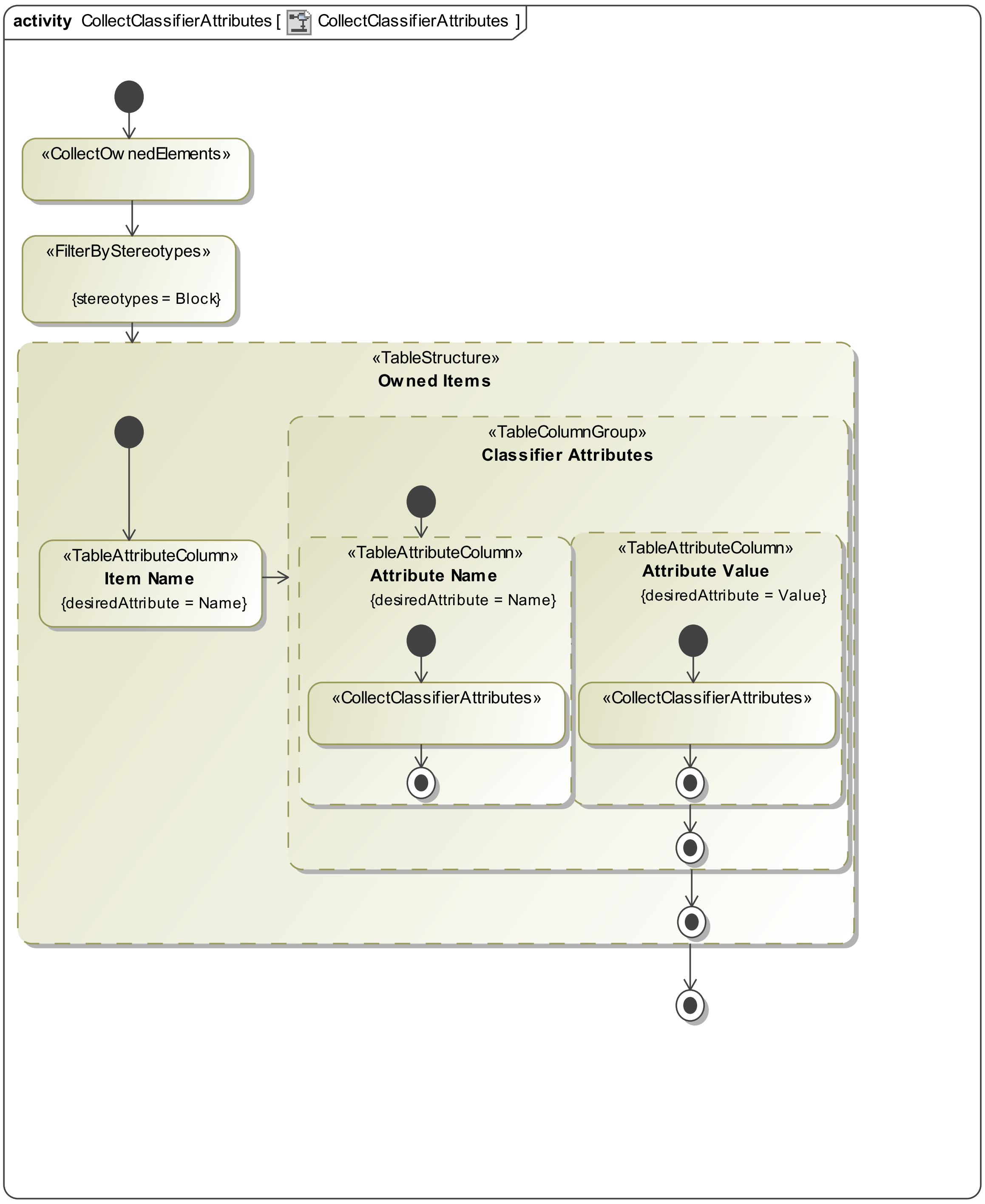
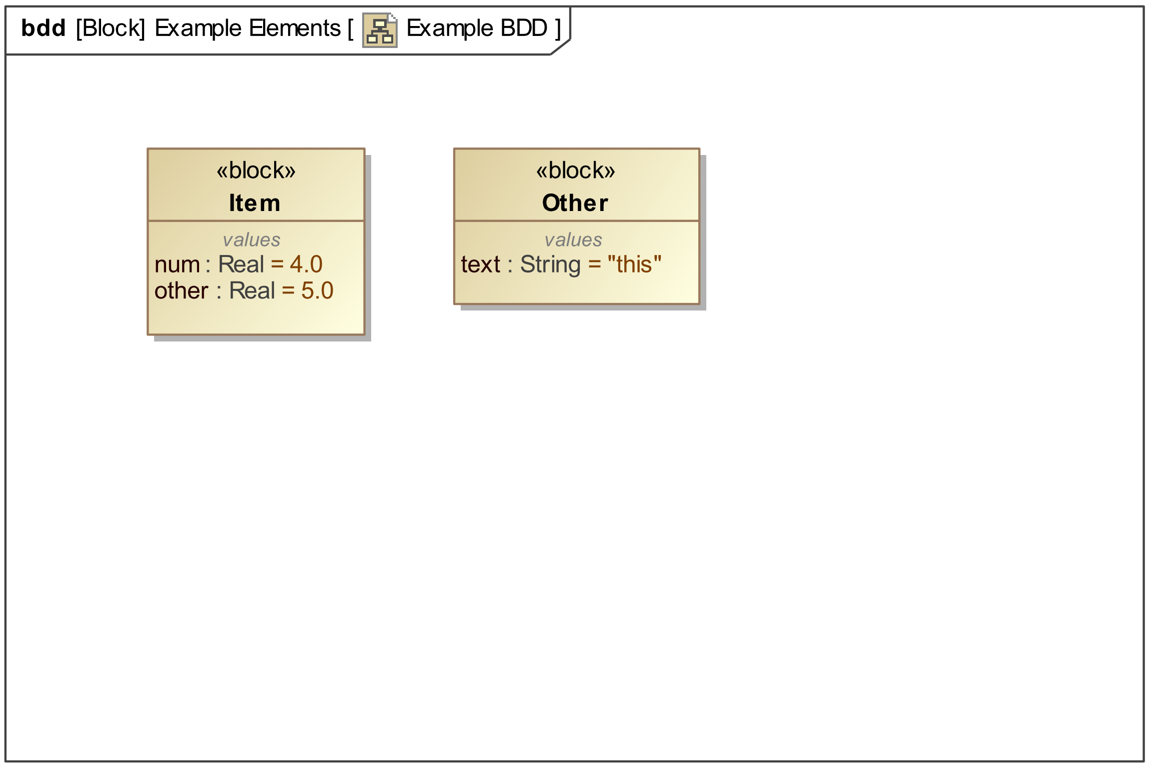


Figure 4.31. Example BDD



Example View

Table 4.4. Owned Items

Item Name	Classifier Attributes	
	Attribute Name	**Attribute Value**
Item	num other	4.0 5.0
Other	text	"this"

CollectByExpression

CollectByExpression is a more customized approach to querying a model using Object Constraint Language (OCL) .

Figure 4.32. Example View Diagram

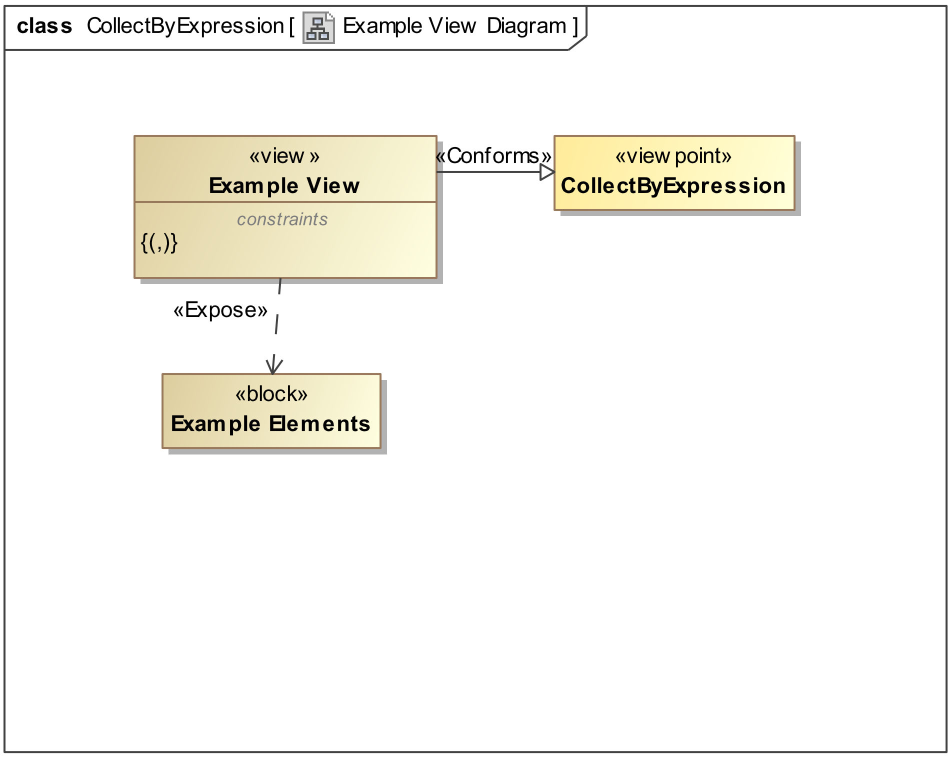


Figure 4.33. CollectByExpression

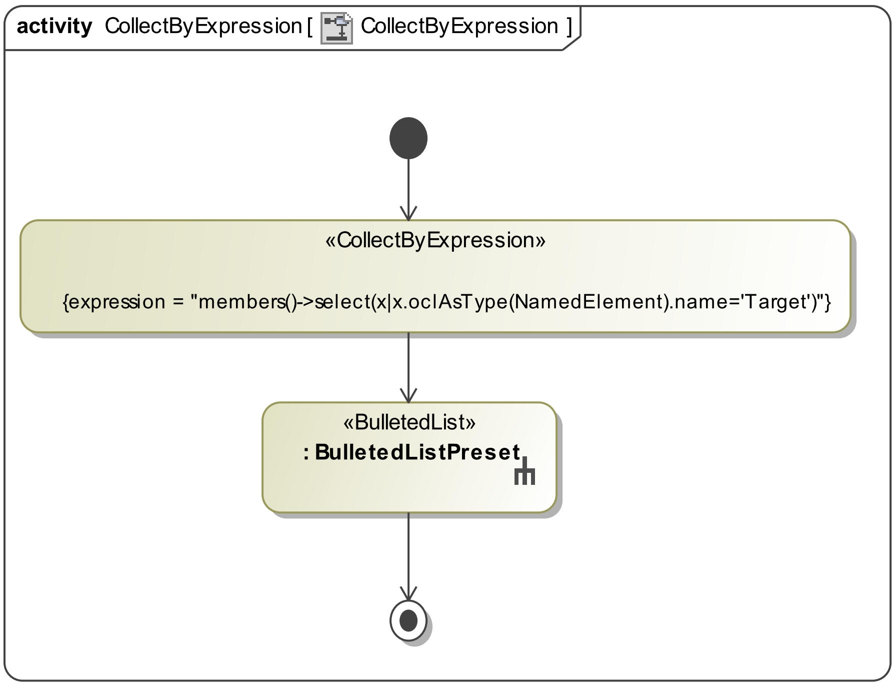
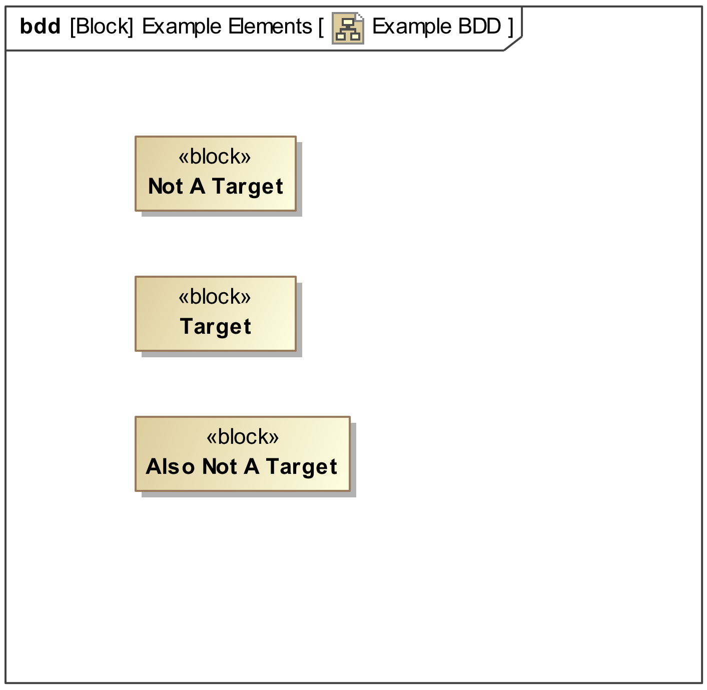

activity CollectByExpression [CollectByExpression]
«CollectByExpression»
{expression = "members()->select(x|x.oclAsType(NamedElement).name='Target')"}
«BulletedList»
: BulletedListPreset

Figure 4.34. Example BDD



Example View

1. Target

Filter

The Filter operations allow the user to narrow fields of data to the data of interest according to one or multiple filter criteria of various types (such as stereotype, name, metaclasses, etc). For most applications, the "FilterBy..." operation needs to have a "CollectBy..." operation preceding it so the filter operation has a data set to look through and filter.

The following sections will take a look at the various Filter operations, what they do, how to set up the operation, and what the output could look like. The examples to follow are simpler than you would likely use with a real project, and they are meant to be simple to explain the basic principles that can be used to create the more complicated outputs you may require.

FilterByDiagramType

The FilterByDiagramType activity goes through a data set and looks at the elements which are diagrams. The user can decide which type of diagram is of interest to them, and display the names of only those diagrams in their document.

Figure 4.35. Example View Diagram

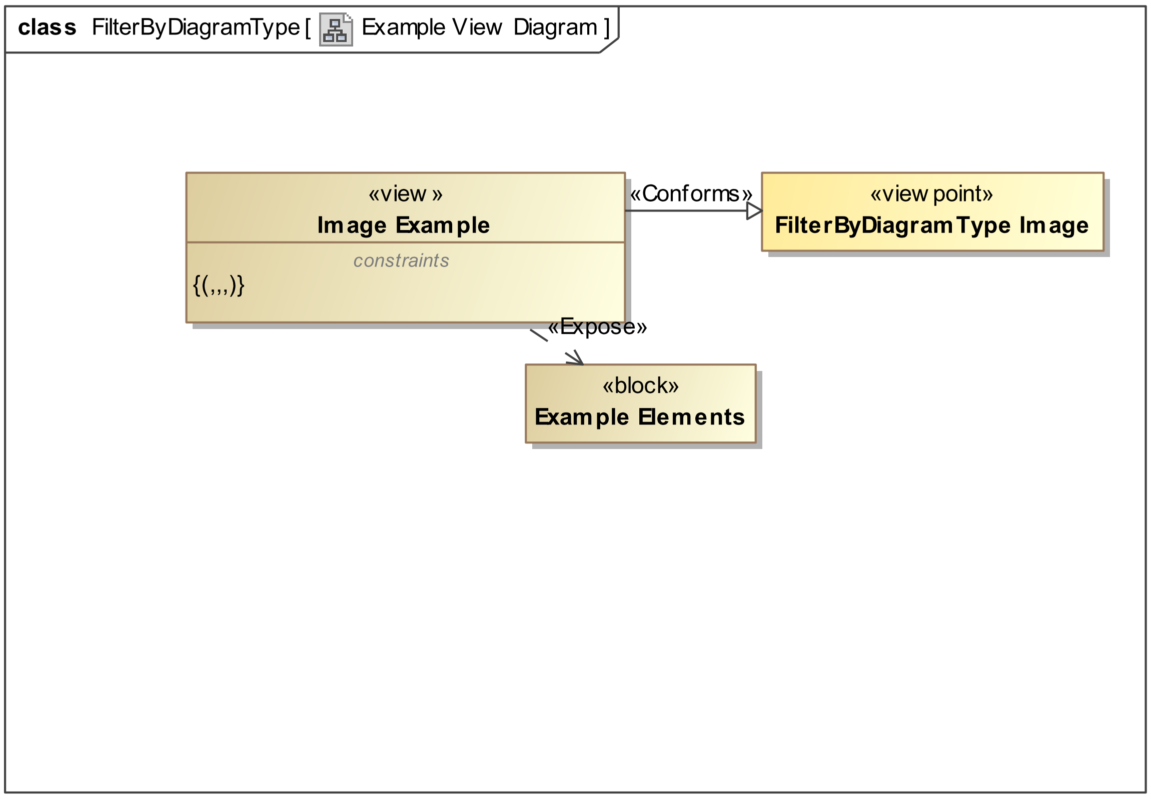


Figure 4.36. Diagrams

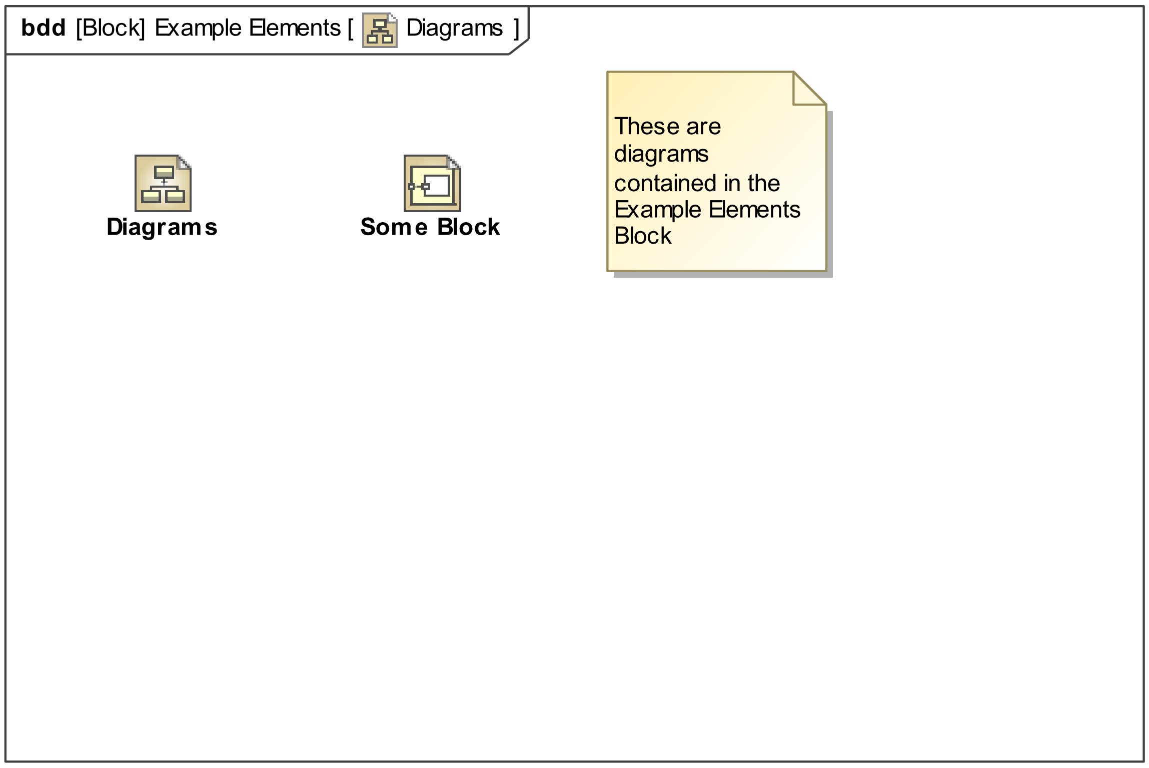

bdd [Block] Example Elements [Diagrams]
Diagrams
Some Block
These are diagrams contained in the Example Elements Block

Figure 4.37. FilterByDiagramType Image

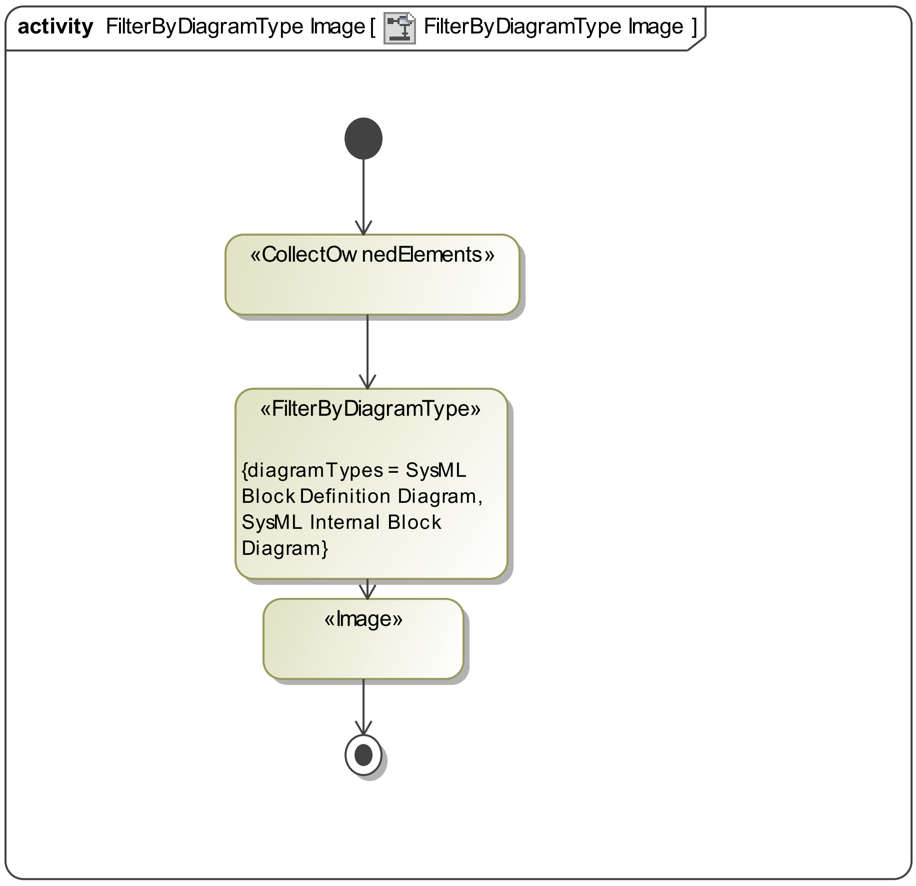

activity FilterByDiagramType Image [FilterByDiagramType Image]
«CollectOwnedElements»
«FilterByDiagramType»
{diagramTypes = SysML Block Definition Diagram, SysML Internal Block Diagram}
«Image»

Image Example

Figure 4.38. Diagrams

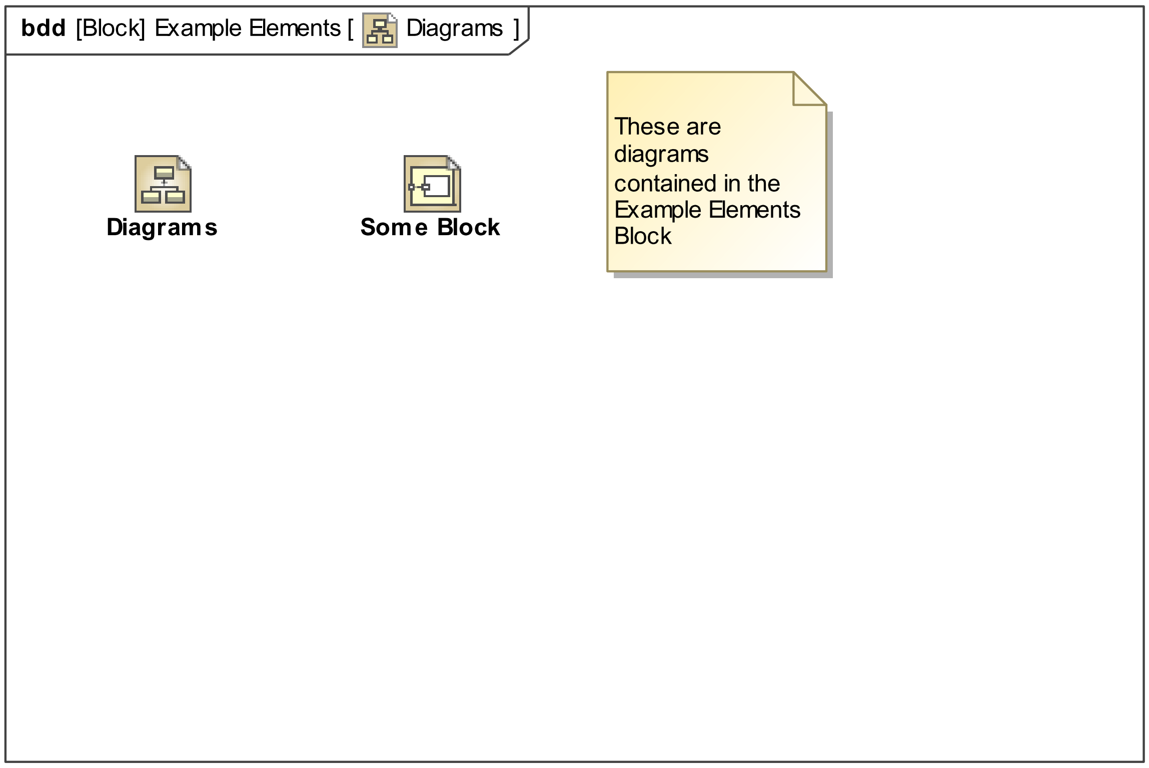
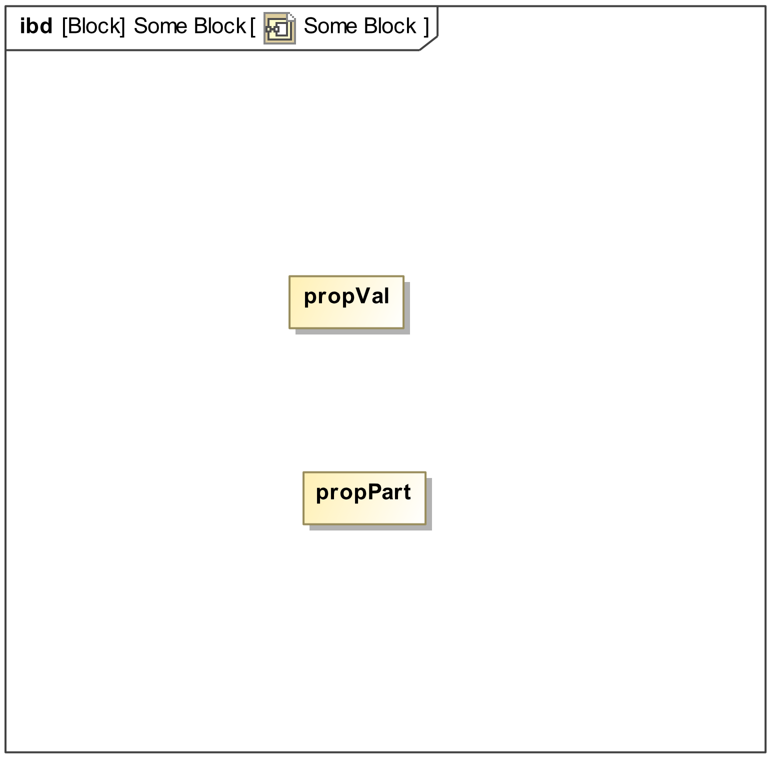

bdd [Block] Example Elements [Diagrams]
Diagrams
Some Block
These are diagrams contained in the Example Elements Block

Figure 4.39. Some Block



FilterByNames

We will now illustrate the use of the "FilterByNames" operation, which allows the user to find all the elements within the data set with a particular name or the elements connected to the element with the particular name. In this case, we will use the regular expression "Bat.*" to gather all elements whose names start with "Bat"

Figure 4.40. Example View Diagram

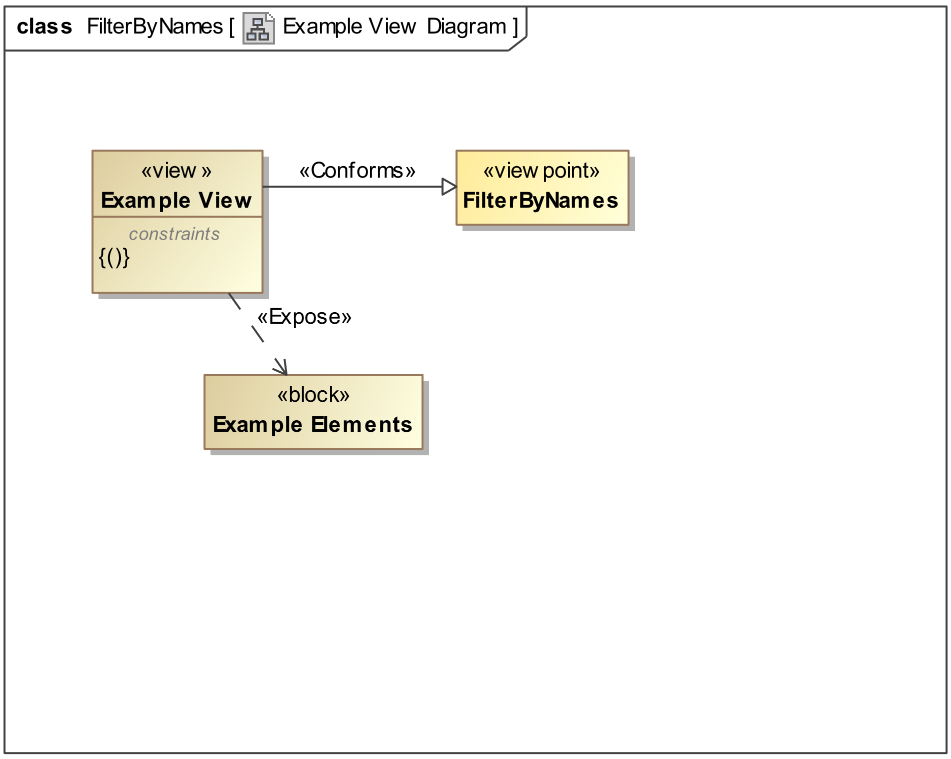


Figure 4.41. Example BDD

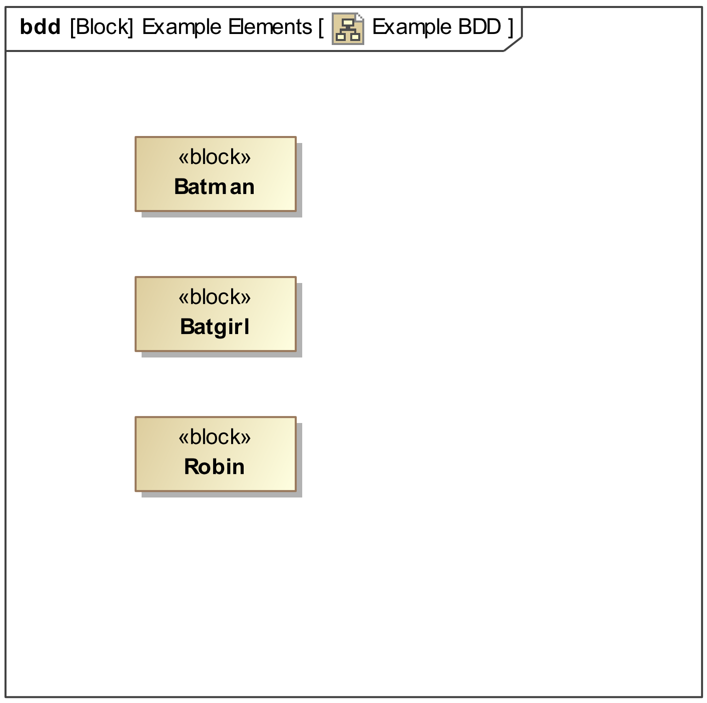
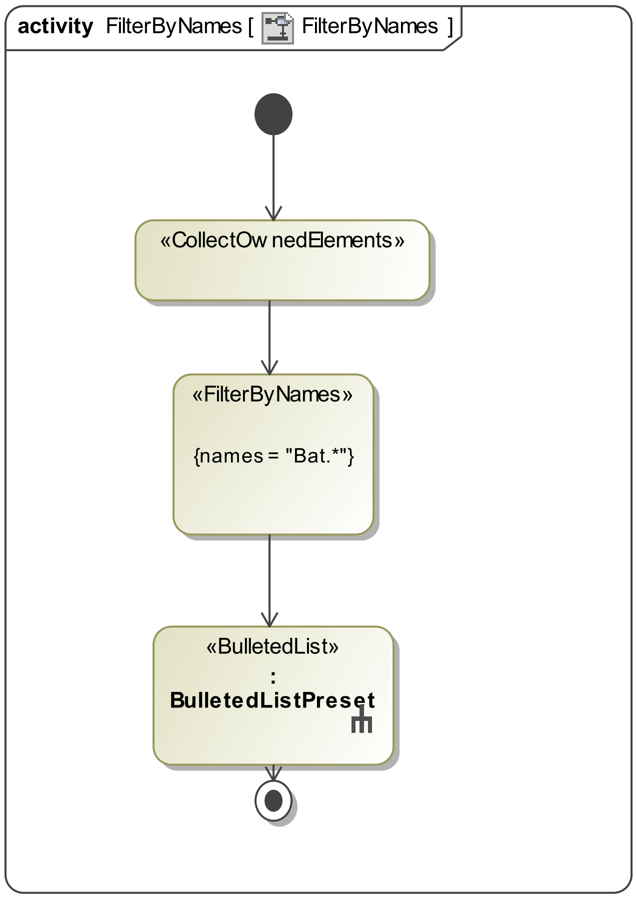


Figure 4.42. FilterByNames



Example View

1. Batman
2. Batgirl

FilterByMetaclasses

FilterByMetaclasses allows filtering of elements by metaclasses. As with the previous Filter operations, a Collect operation must be used first to collect elements desired to be filtered. In the example below, the CollectOwnedElements will be used again; this time, we will collect the elements owned by the block "Example Elements." The FilterByMetaclasses operation is then used to show the actors in this subset of elements.

After dragging a "FilterByMetaclasses" block onto the diagram, go to the specification window for the "FilterByMetaclasses" block. In this window, check the "true" box under "IncludeChoosable." As with the previous examples, checking this box means we want to include the metaclasses we are going to choose to filter. Next, we go to the "MetaclassesChoosable" section in the specification window, and click the "..." button . A "Select Class" window will appear. In this window, make sure the small box in the bottom left hand corner is selected to allow metaclasses to be shown in the selection box. Now, we can navigate to "UML Standard Profile" -> "UML2 Metamodel" -> select "Actor" -> and click the "+" button. We could also have chosen the "Stereotype" metaclass, for example, or a number of other metaclasses.

Figure 4.43. Example View Diagram

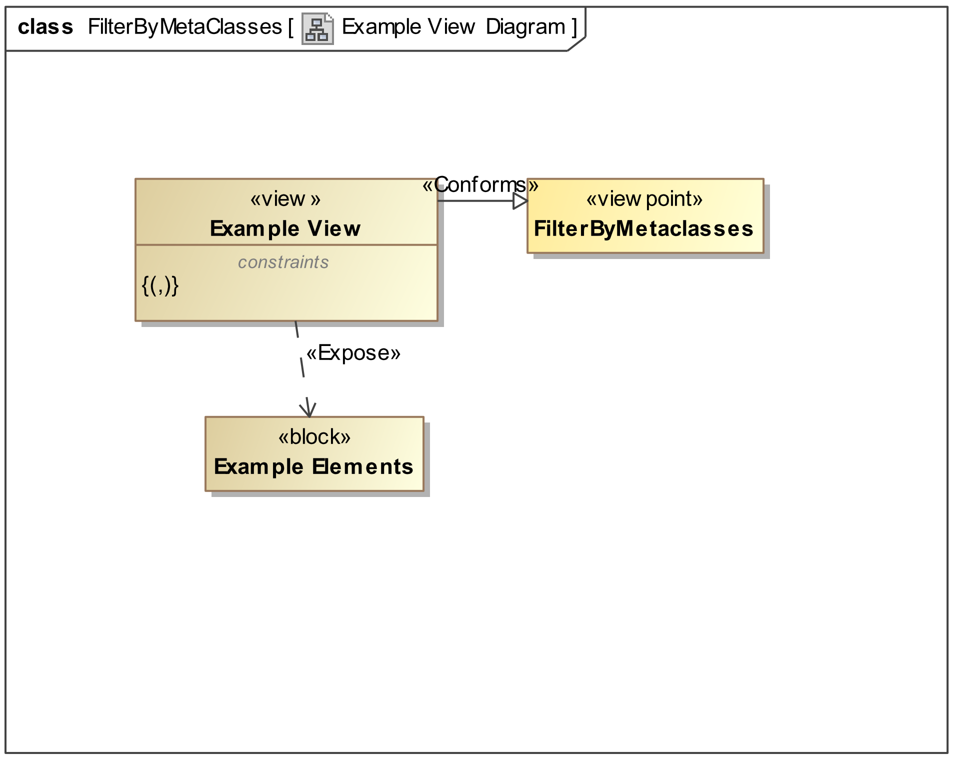


Figure 4.44. FilterByMetaclasses

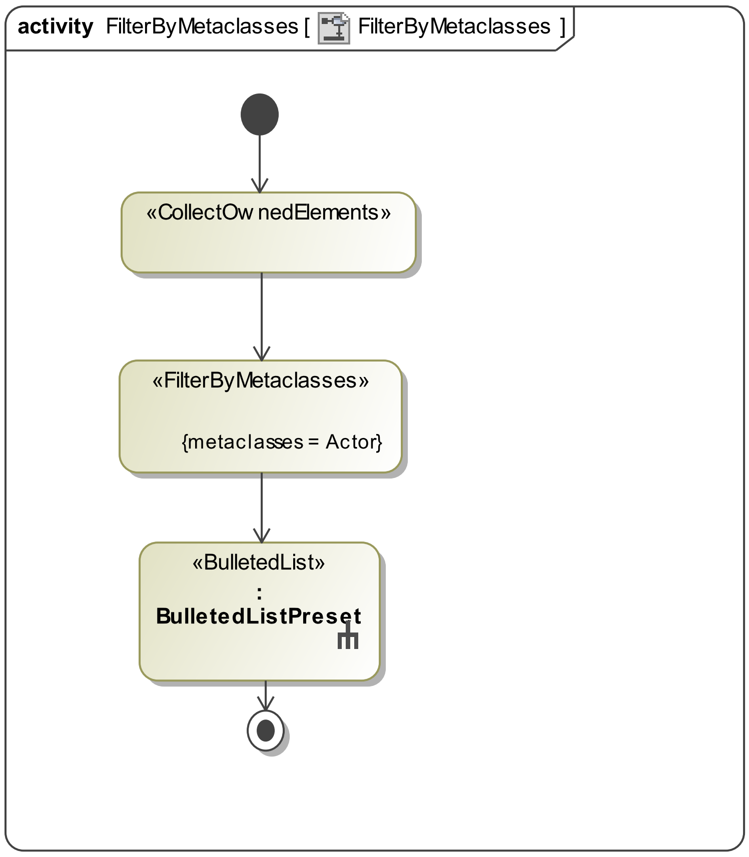
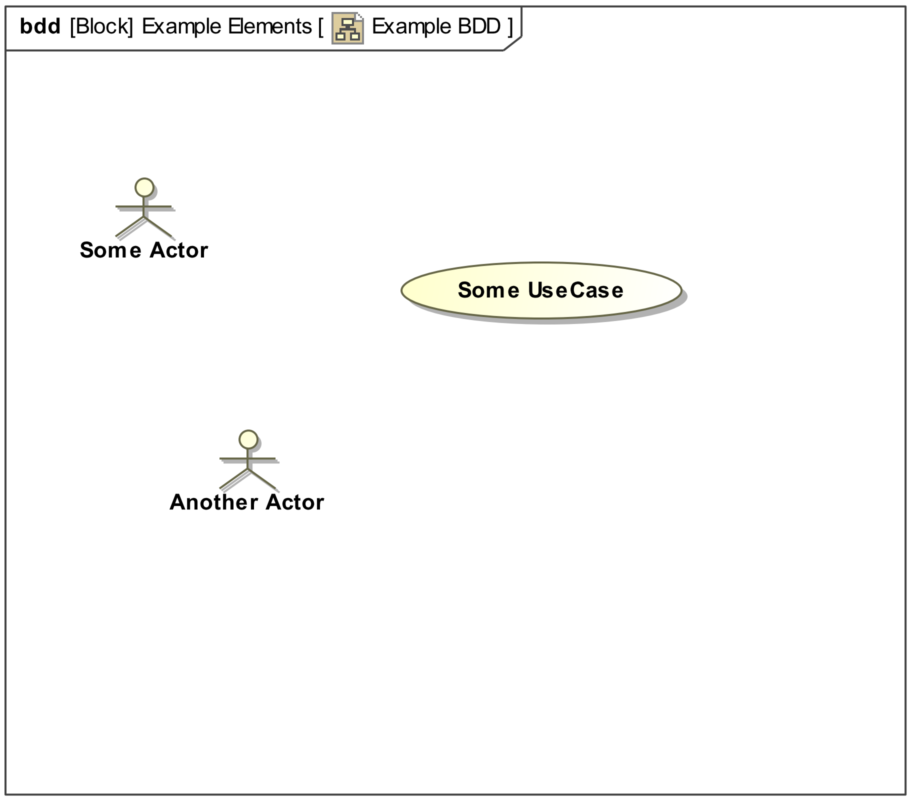


Figure 4.45. Example BDD



Example View

1. Some Actor
2. Another Actor

FilterByStereotypes

FilterByStereotypes allows filtering of elements by the stereotype(s) applied to them.

Figure 4.46. Example View Diagram

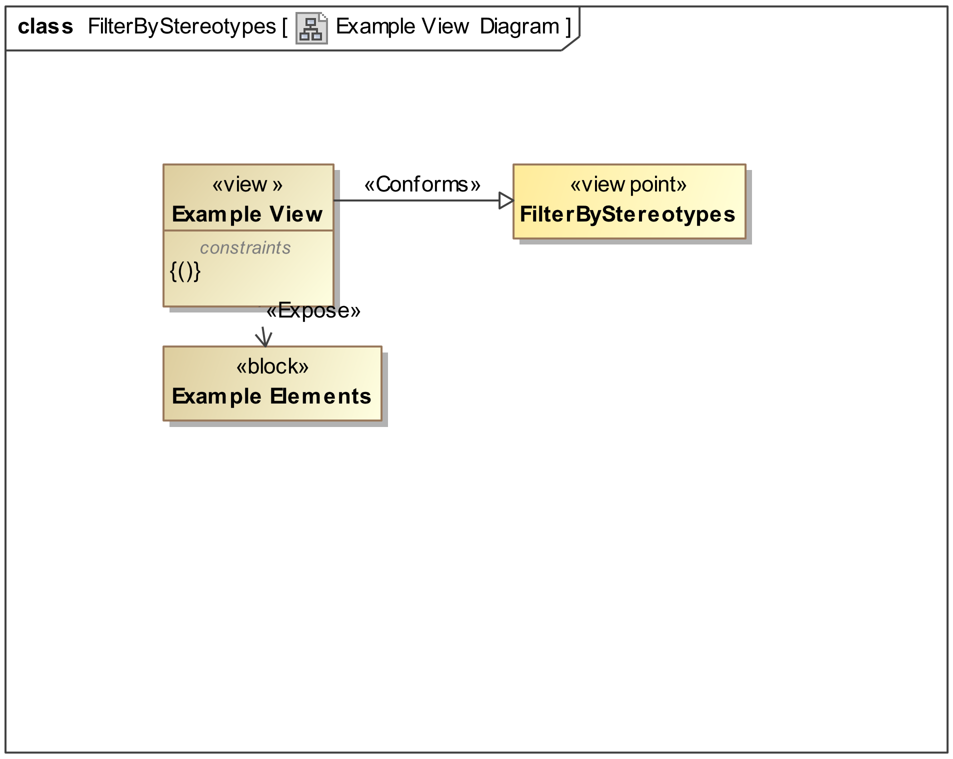


Figure 4.47. FilterByStereotypes

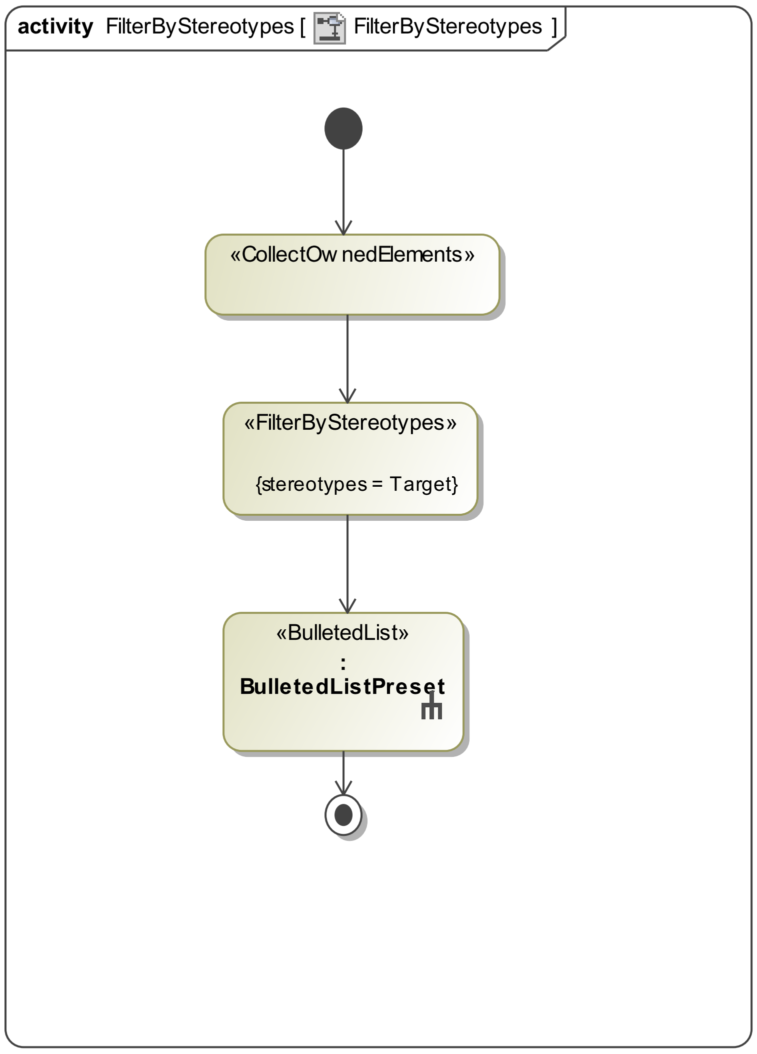
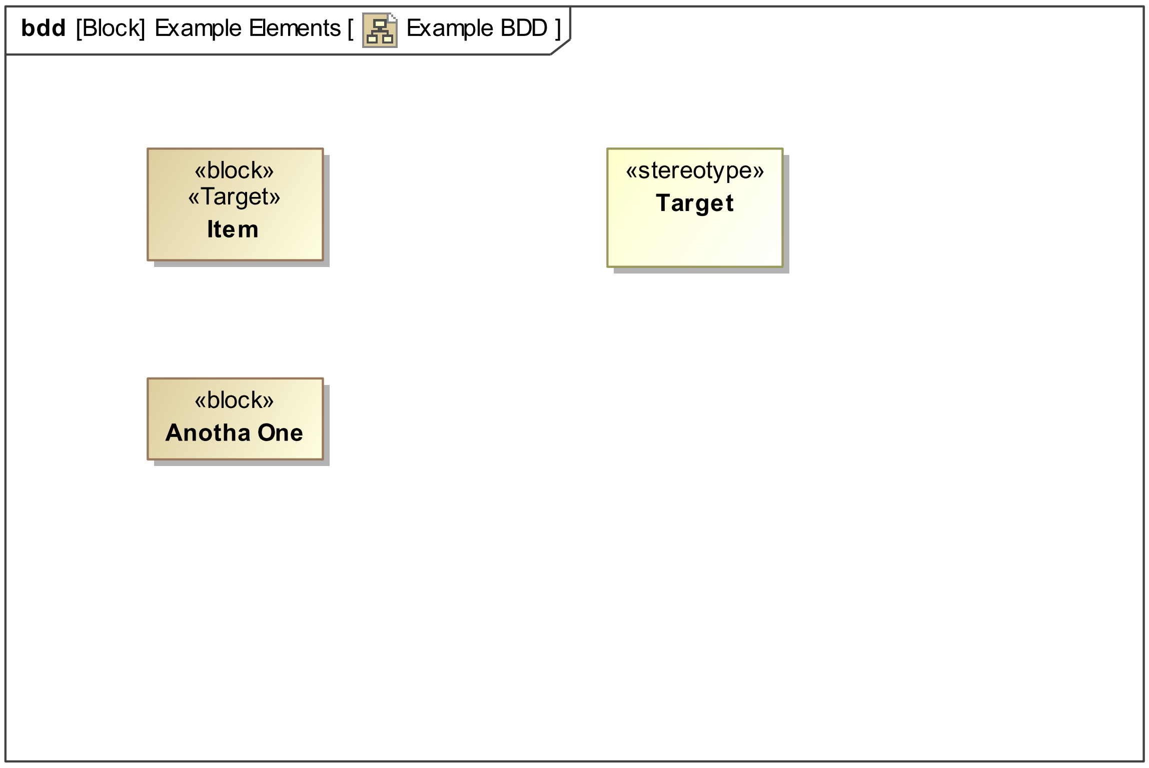


Figure 4.48. Example BDD



Example View

1. Item

FilterByExpression

FilterByExpression is a more customized approach to querying a model using Object Constraint Language (OCL). The expression value is simply a boolean written in OCL.

Figure 4.49. Example View Diagram

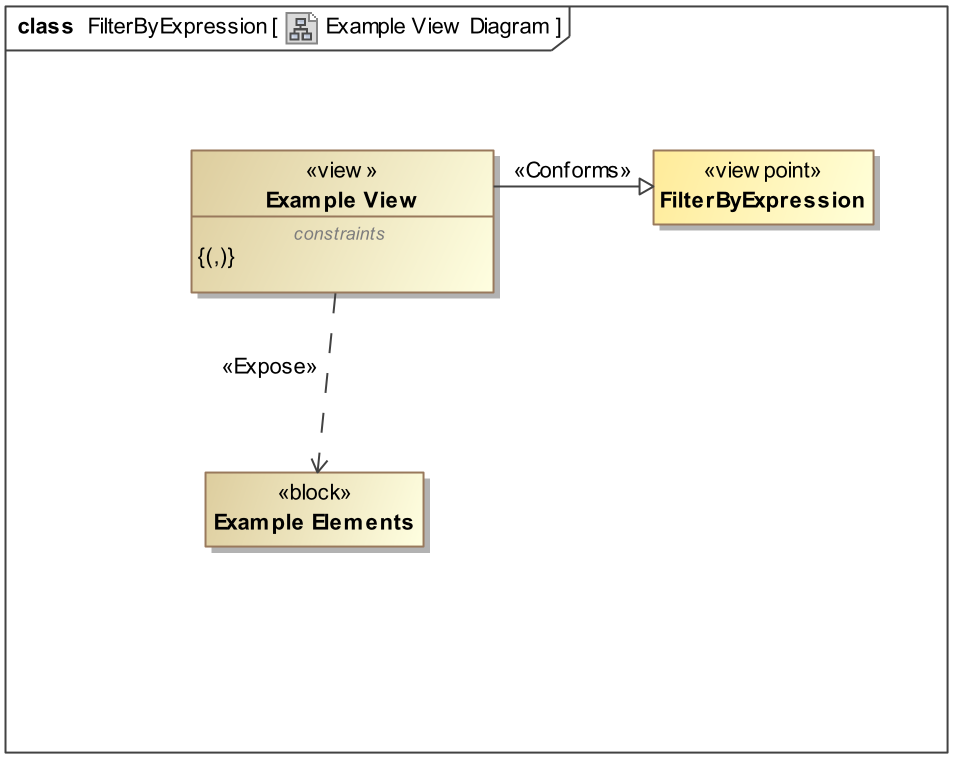


Figure 4.50. FilterByExpression

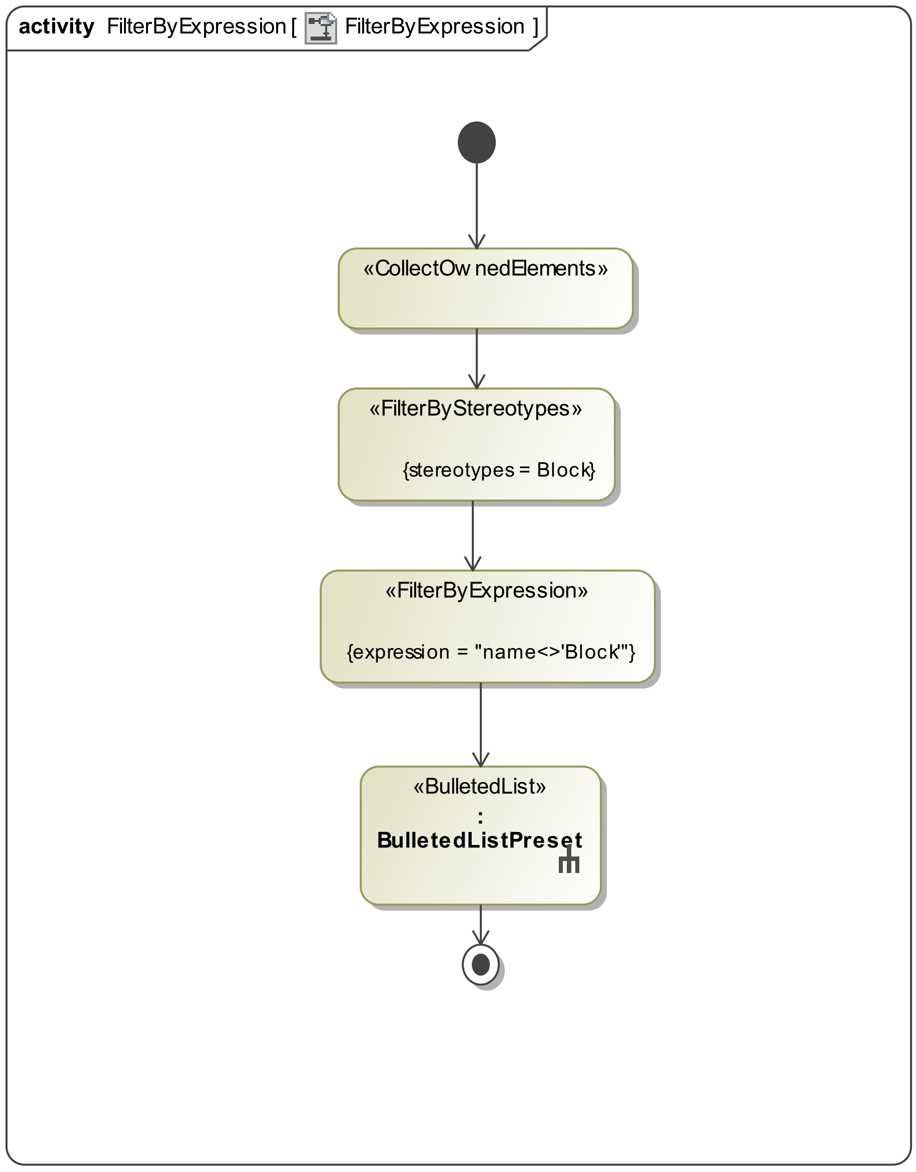
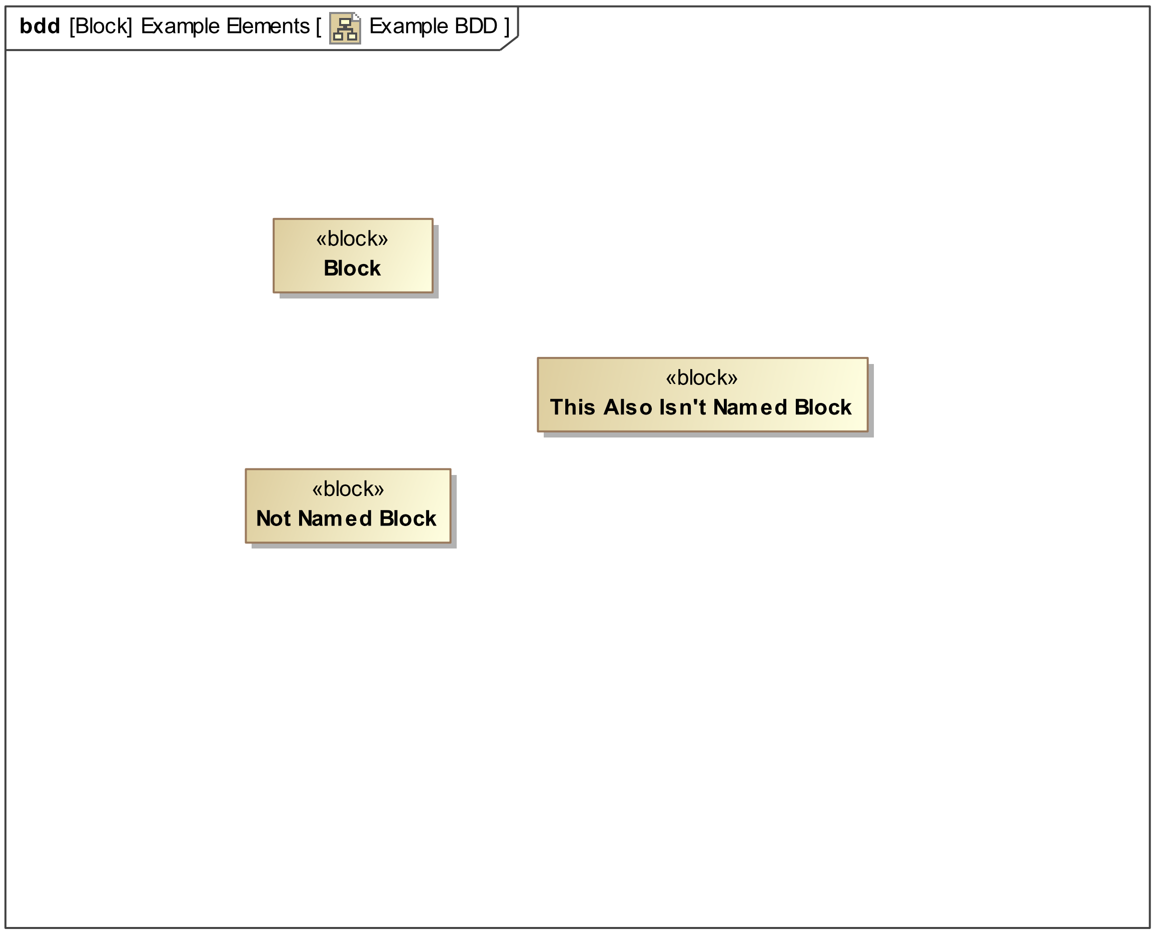


Figure 4.51. Example BDD


bdd [Block] Example Elements [Example BDD]
«block»
Block
«block»
This Also Isn't Named Block
«block»
Not Named Block

Example View

1. Not Named Block
2. This Also Isn't Named Block

Sort

The Sort operations allow a data set to be sorted according to a desired parameter. Data can be sorted either by attribute, property, or expression. These Sort operations are illustrated below. As with the Filter operations, it is necessary for us to Collect elements in some way in order to have a data set to sort.

SortByAttribute

"SortByAttribute" allows the user to sort a data set by the chosen attribute. The list of attributes that can be sorted include name, documentation, or value. With this operation, only one attribute is choosable at a time for sort. In the "FilterByStereotypes" example, it was highlighted that a Sort operation was necessary if a user would like to alphabetize the output of a viewpoint method.

In the specification window for "SortByAttribute," we choose the "Name" option from the drop-down menu under the "AttributeChoosable" section. Ensure "Reverse" is false.

Figure 4.52. Example View Diagram

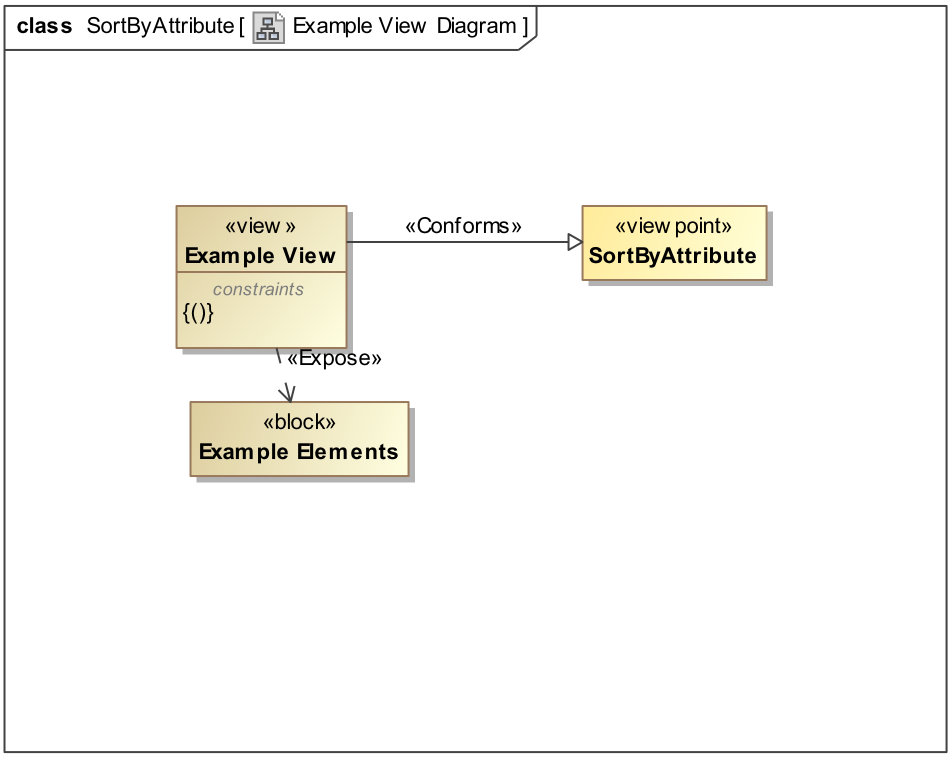


Figure 4.53. SortByAttribute

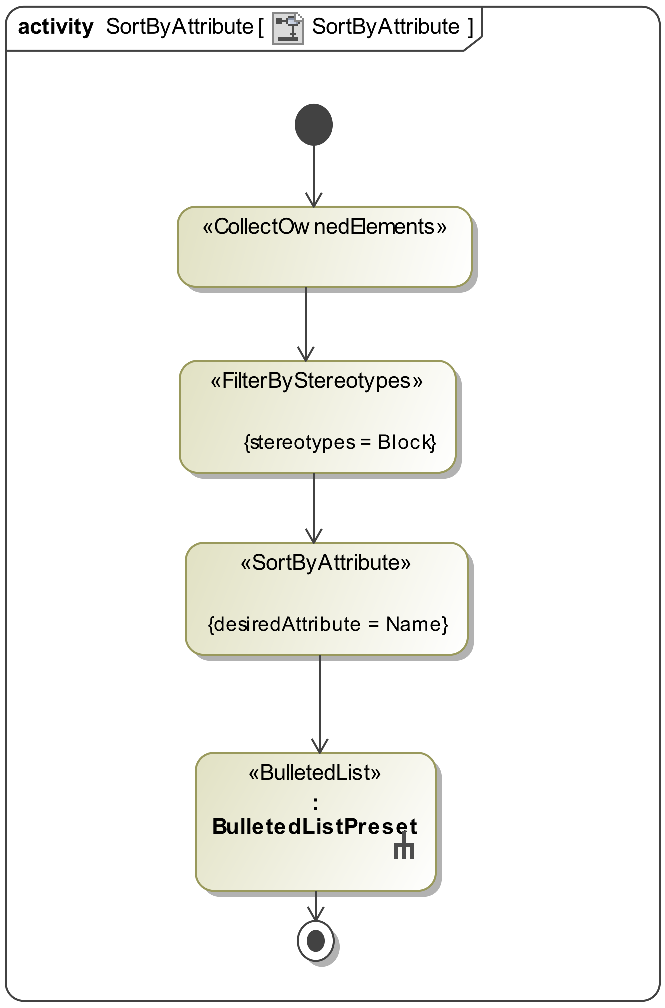
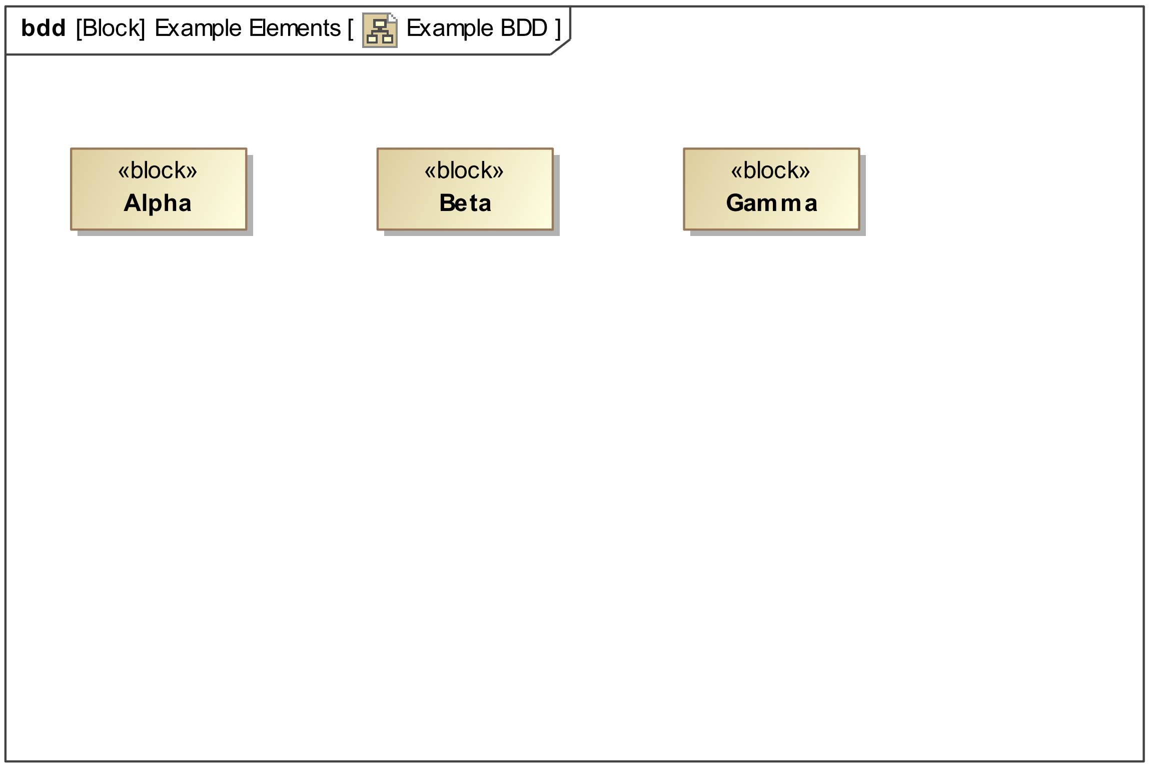


Figure 4.54. Example BDD



Example View

1. Alpha
2. Beta
3. Gamma

SortByProperty

Figure 4.55. Example View Diagram

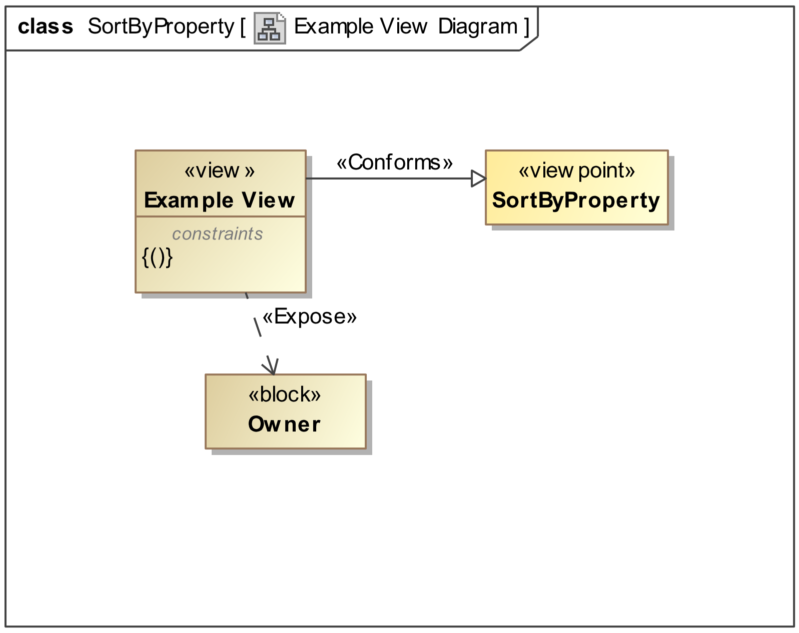

class SortByProperty [Example View Diagram]
«view» Example View
constraints
{()}
«Conforms»
«view point» SortByProperty
«Expose»
«block» Owner

Figure 4.56. Example BDD

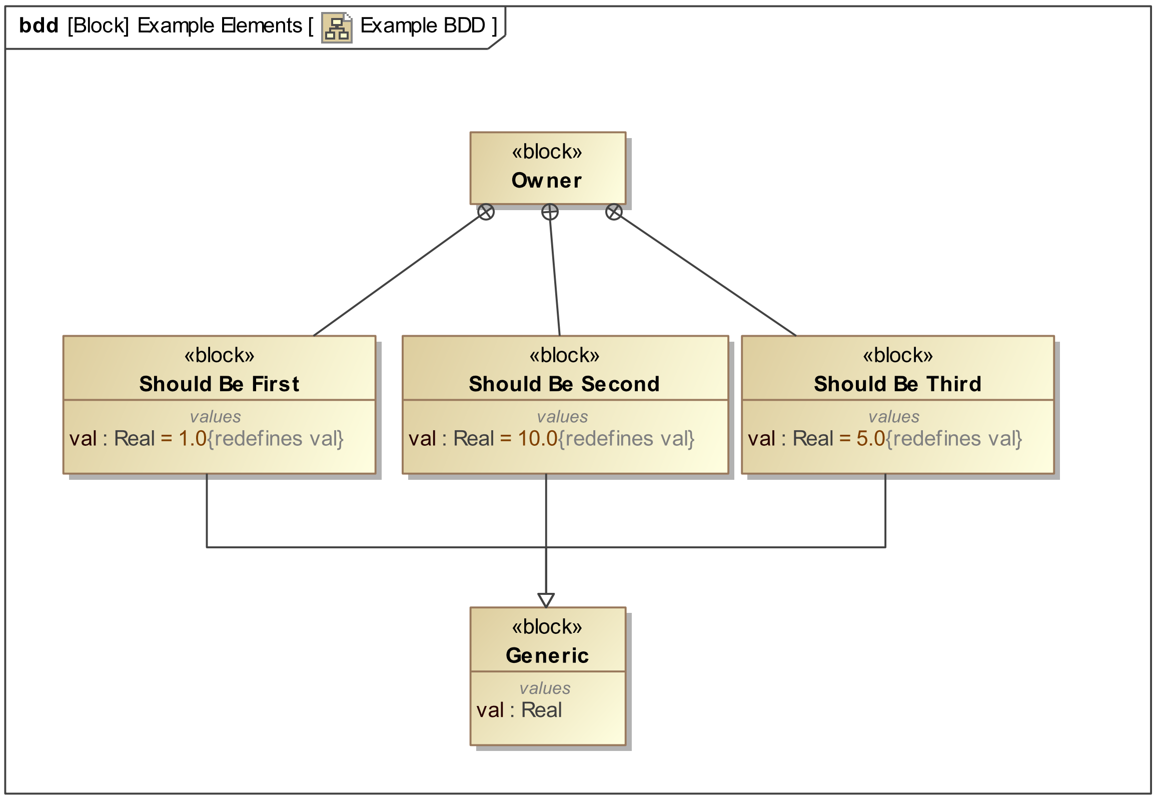
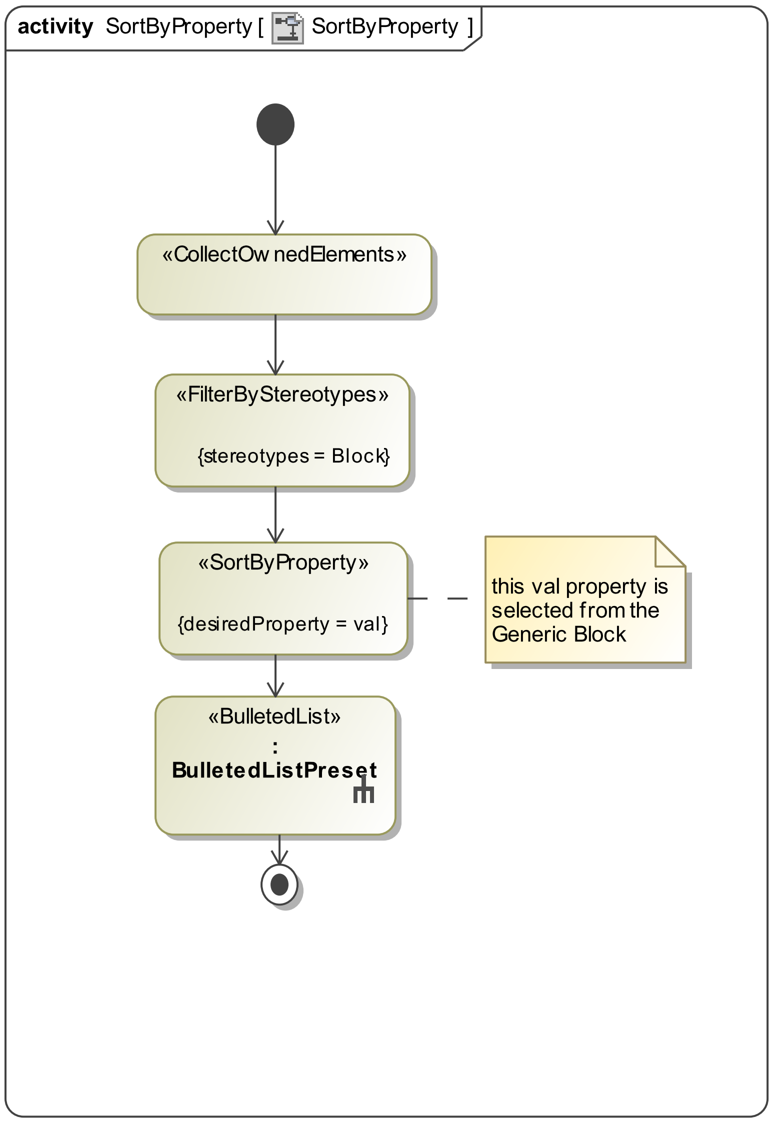

bdd [Block] Example Elements [Example BDD]
«block» Owner
«block» Should Be First
values
val : Real = 1.0{redefines val}
«block» Should Be Second
values
val : Real = 10.0{redefines val}
«block» Should Be Third
values
val : Real = 5.0{redefines val}
«block» Generic
values
val : Real

Figure 4.57. SortByProperty



Example View

1. Should Be First
2. Should Be Third
3. Should Be Second

SortByExpression

CollectByExpression is a more customized approach to querying a model using Object Constraint Language (OCL).

Figure 4.58. Example View Diagram

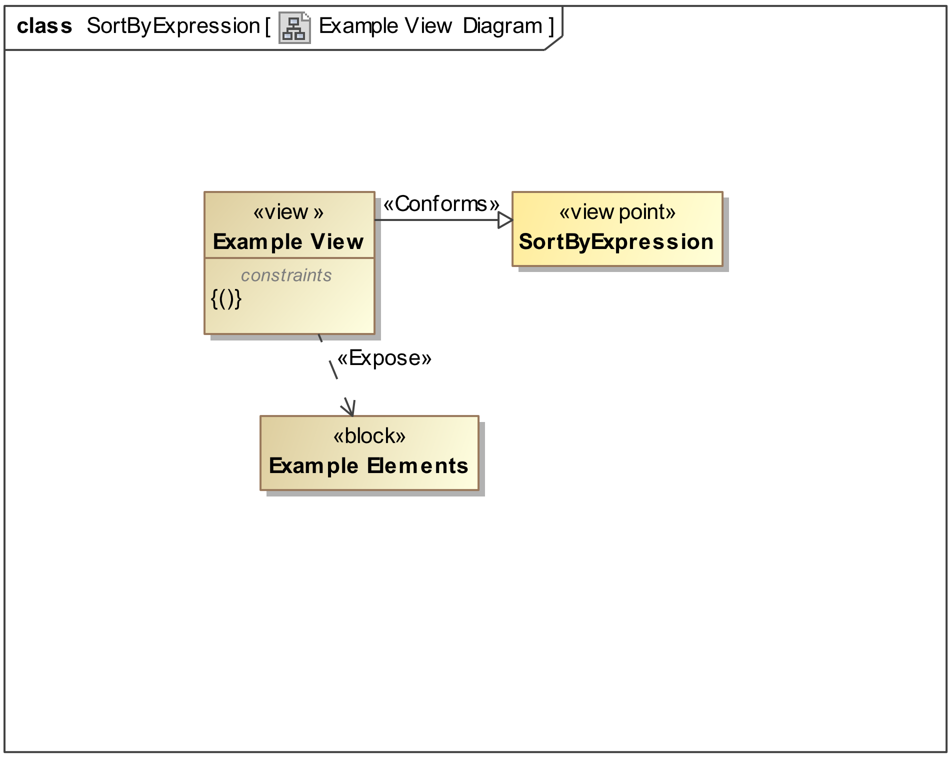


Figure 4.59. SortByExpression

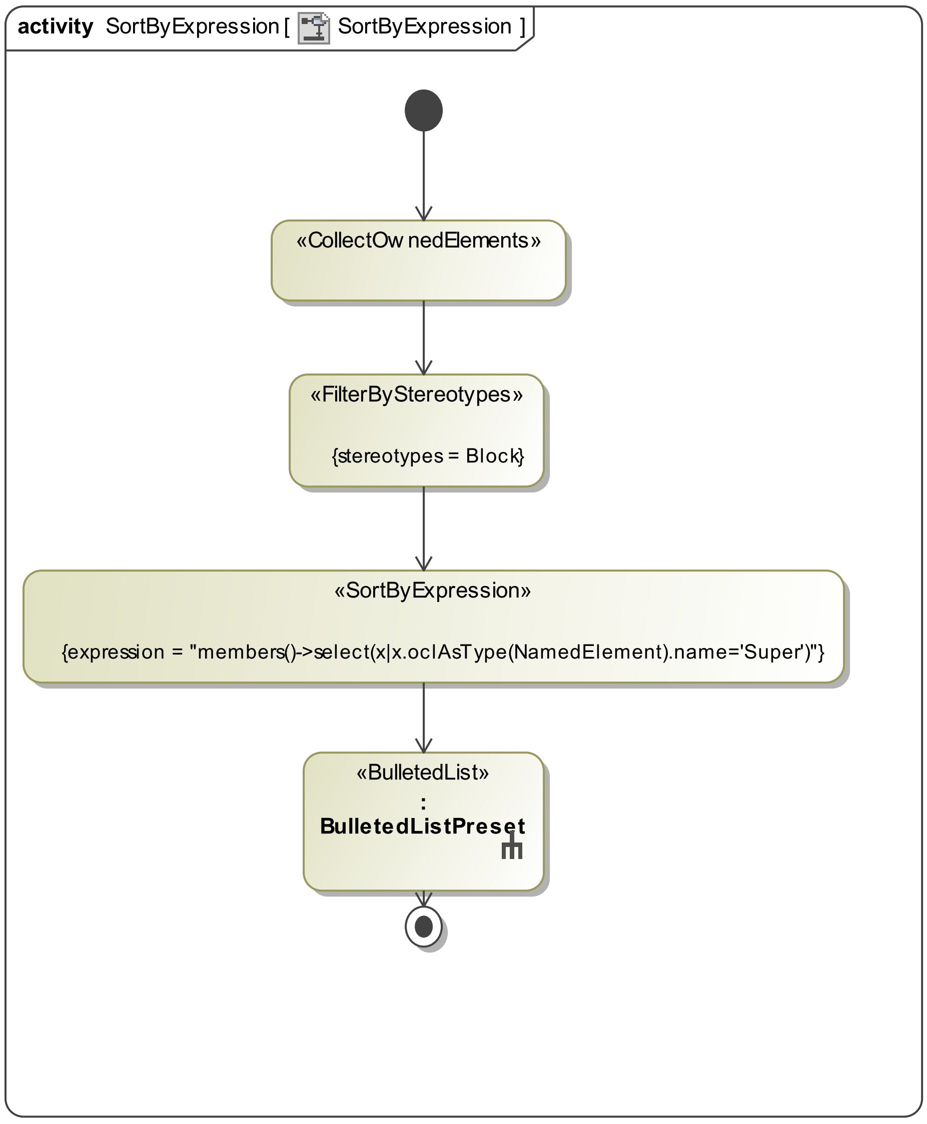
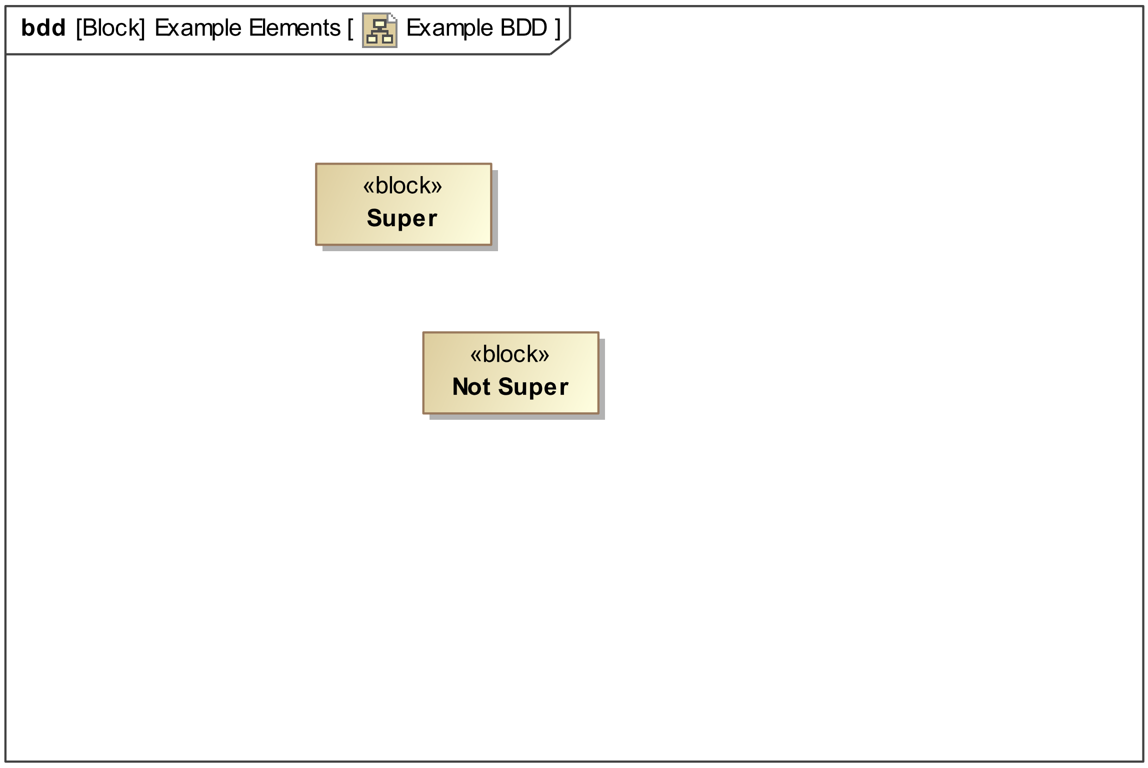

activity SortByExpression [SortByExpression]
«CollectOwnedElements»
«FilterByStereotypes»
{stereotypes = Block}
«SortByExpression»
{expression = "members()->select(x|x.oclAsType(NamedElement).name='Super')"}
«BulletedList»
: BulletedListPreset

Figure 4.60. Example BDD



Example View

1. Super
2. Not Super

Present Model Data

After the data is collected, filtered, and/or sorted, several operators can be used to adjust how the data will be displayed. The presentation element operators are table, image, paragraph, list, and sections. These are used in the viewpoint method diagram and determine the formatting that will be displayed in the document views.

The following examples all use a common Zoo package, that has an assortment of elements to serve as samples. The examples that deal with OCL use a Robot Zoo package in order to demonstrate different behavior.

Table

The sections below go over how to create tables. Tables of varying complexity can be created in MagicDraw, but these tables share some common base components. The left sidebar gives various components for use under "Table Structure."

The first option, TableStructure, is used in all cases to create the table base. The last three options shown under "Table Structure" in the sidebar allow the user to select a type of column based on the information they need to display in each column. TableExpressionColumn uses an OCL expression. TablePropertyColumn allows stereotype properties or value properties of a class to be displayed, depending on the what the user chooses. The TableAttributeColumn allows the user to display the name, documentation, or value of an element. Clicking the small black arrow on the right of these options on the sidebar opens a set of selections where the second one has a dotted outside line on the icon. This dotted outside line selection exists for each of the three column types and allows the user to configure a flow within the column. The TableColumnGroup allows creating a column group with a merged header, as will be illustrated in the Complex Table selection.

Figure 4.61. Example View Diagram

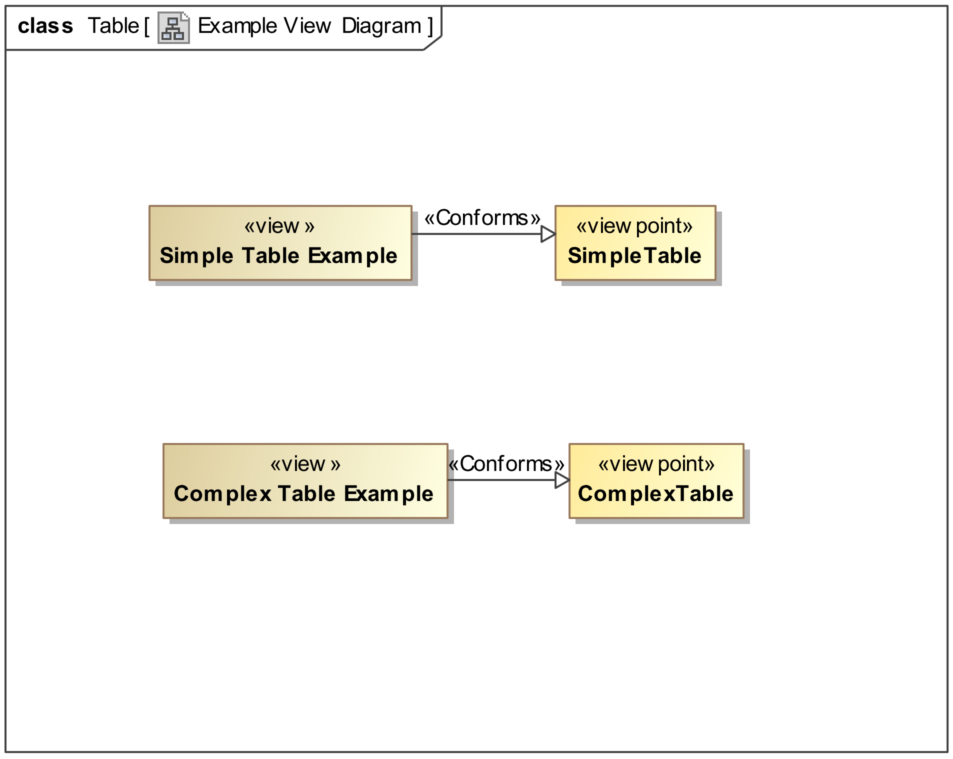


Figure 4.62. SimpleTable

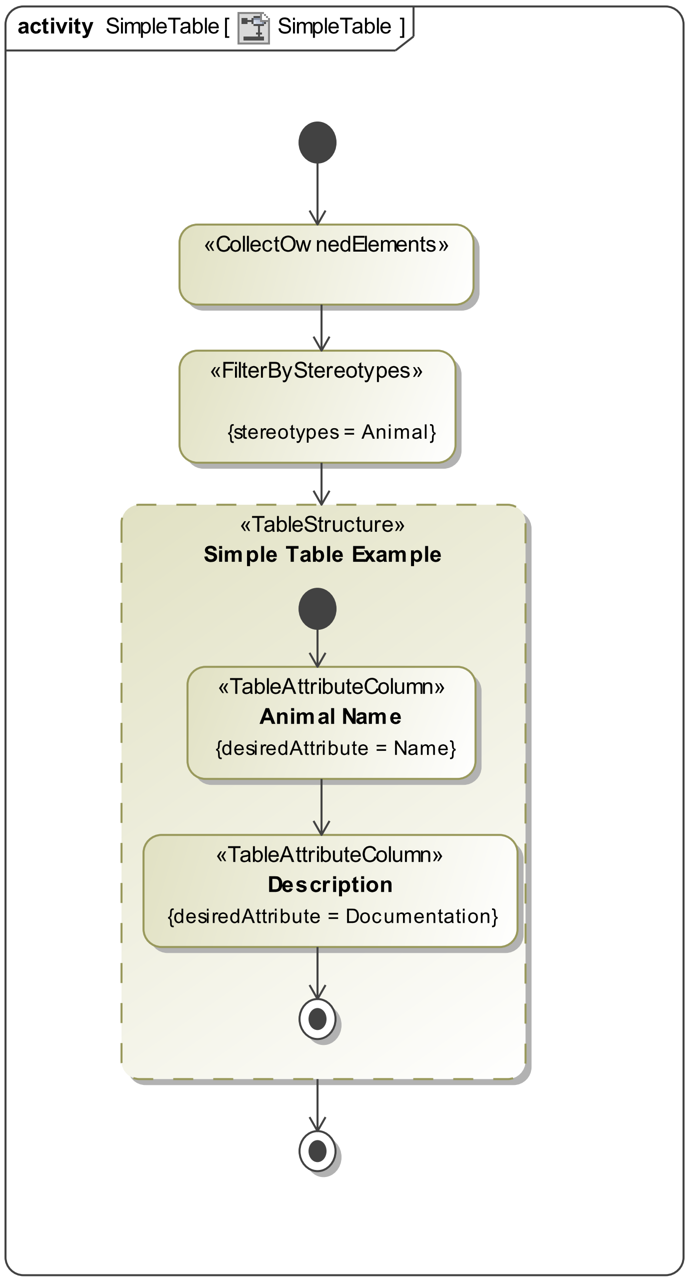
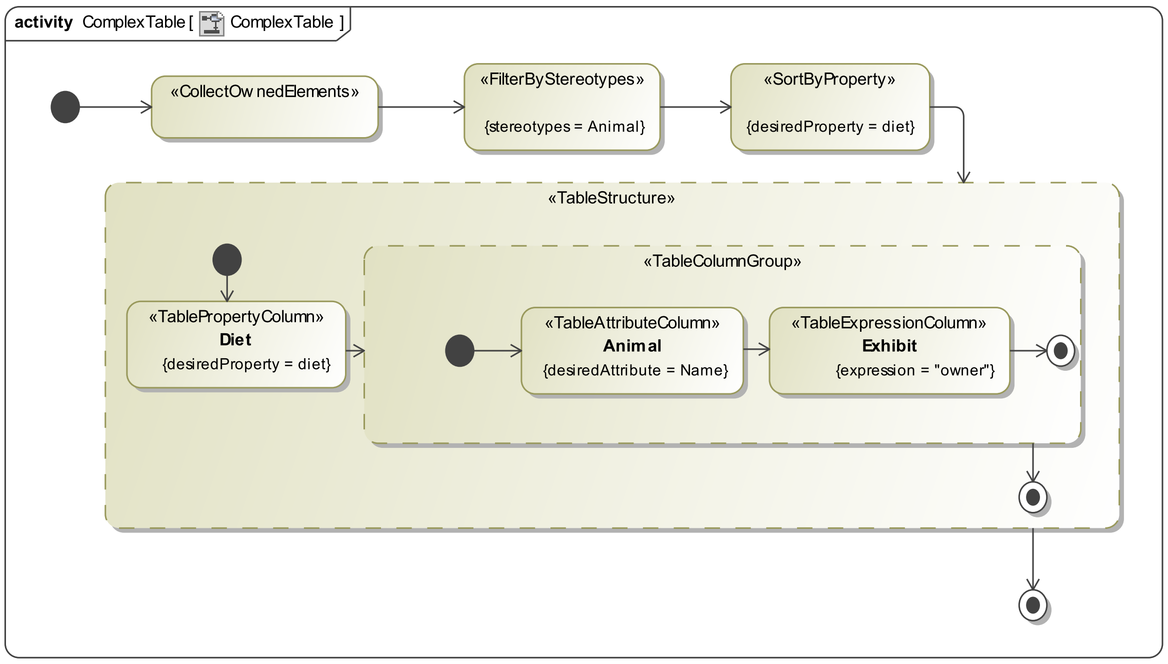


Figure 4.63. ComplexTable


activity ComplexTable [ComplexTable]
«CollectOwnedElements»
«FilterByStereotypes» {stereotypes = Animal}
«SortByProperty» {desiredProperty = diet}
«TableStructure»
«TablePropertyColumn» Diet {desiredProperty = diet}
«TableColumnGroup»
«TableAttributeColumn» Animal {desiredAttribute = Name}
«TableExpressionColumn» Exhibit {expression = "owner"}

Simple Table

This viewpoint method is used to create a simple table that shows the name and documentation of all the Animals in the Zoo package. First we collect all elements owned by the Zoo package. Then we filter out all elements that are not <<Animal>>. This leaves the five animal elements, which we pass into a TableStructure.

The first column we create shows the Name attribute of the animal elements. We call this column "Animal Name" which will become the header of the column when we generate the document or reveal on ViewEditor. We select "Name" as the desiredAttribute in the TableAttributeColumn Action.

We do the same thing for the next column "Description" except that we set desiredAttribute to "Documentation" to target a different attribute of the Animal element.

Table 4.5. Simple Table Example

Animal Name	Description
Ostrich	Ostriches can run up to 70 km/h, the fastest land speed of any bird.
Crocodile	Basically a dinosaur.
Zebra	Zebras feed almost entirely on grasses.
Seal	Mostly blubber.
Arctic Tern	What sound does an arctic tern make?

Complex Table

Table 4.6.

Diet		
	Animal	**Exhibit**
Carnivore	Crocodile	Africa Exhibit

Diet		
	Animal	**Exhibit**
Carnivore	Seal	Antarctica Exhibit
Carnivore	Arctic Tern	Antarctica Exhibit
Herbivore	Zebra	Africa Exhibit
Omnivore	Ostrich	Africa Exhibit

Image

Image Elements can be displayed in view editor by the use of the <<Image>> action. An example of an Image Element is a Diagram. This action will display the image associated with any Image Element followed by its documentation. If multiple diagrams are exposed or collected, a single <<Image>> action will iteratively display them all.

This image will then be updated when the model is updated without need for a new image to be put into the document to reflect changes. It is also possible to add captions and titles to the image, however, these functionalities are not currently working.

The View Diagram and Viewpoint Method Diagram for this operation are shown below.

Figure 4.64. Example View Diagram

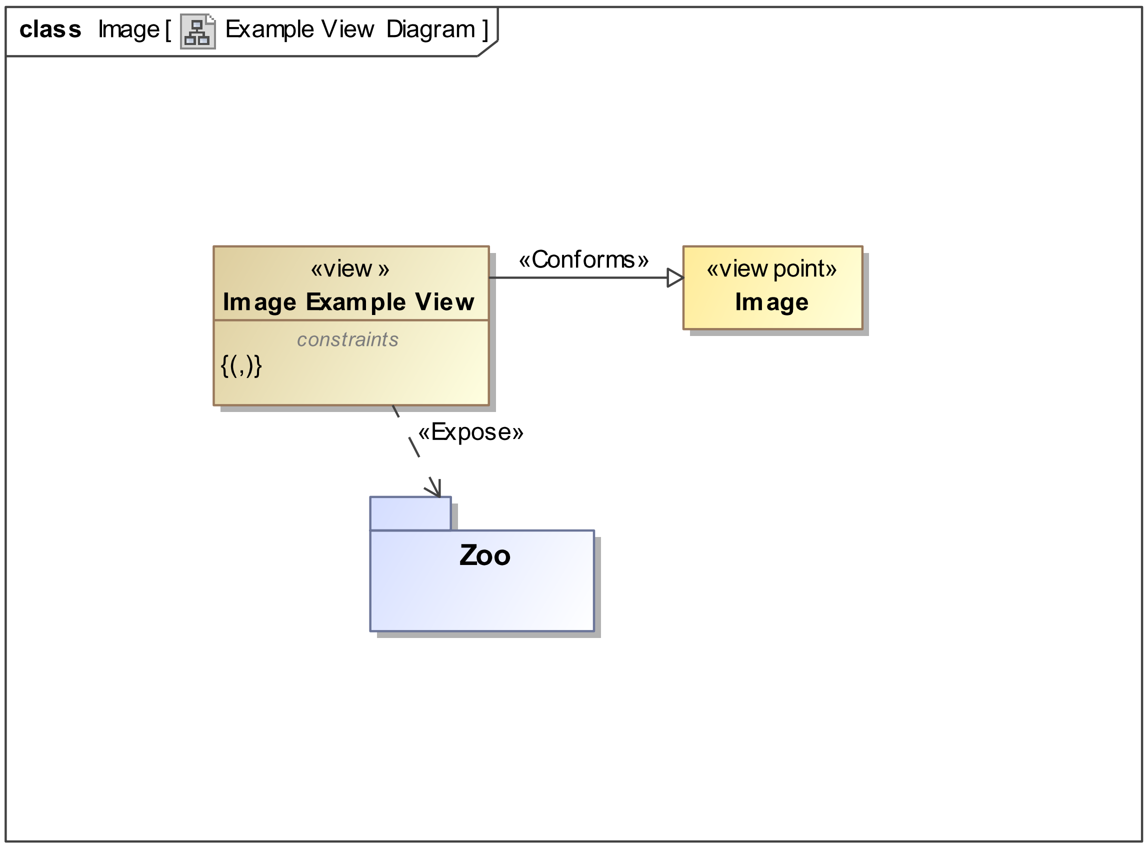


Figure 4.65. Image

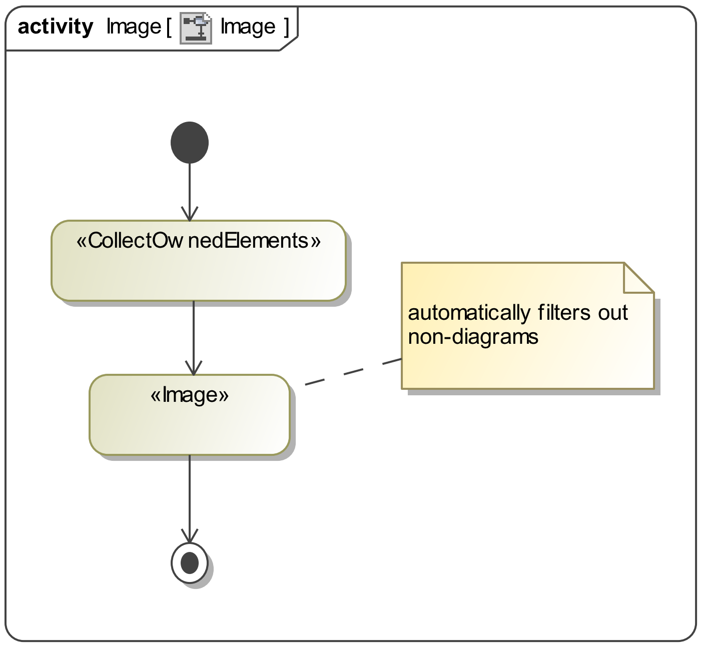


Image Example View

Figure 4.66. Zoo

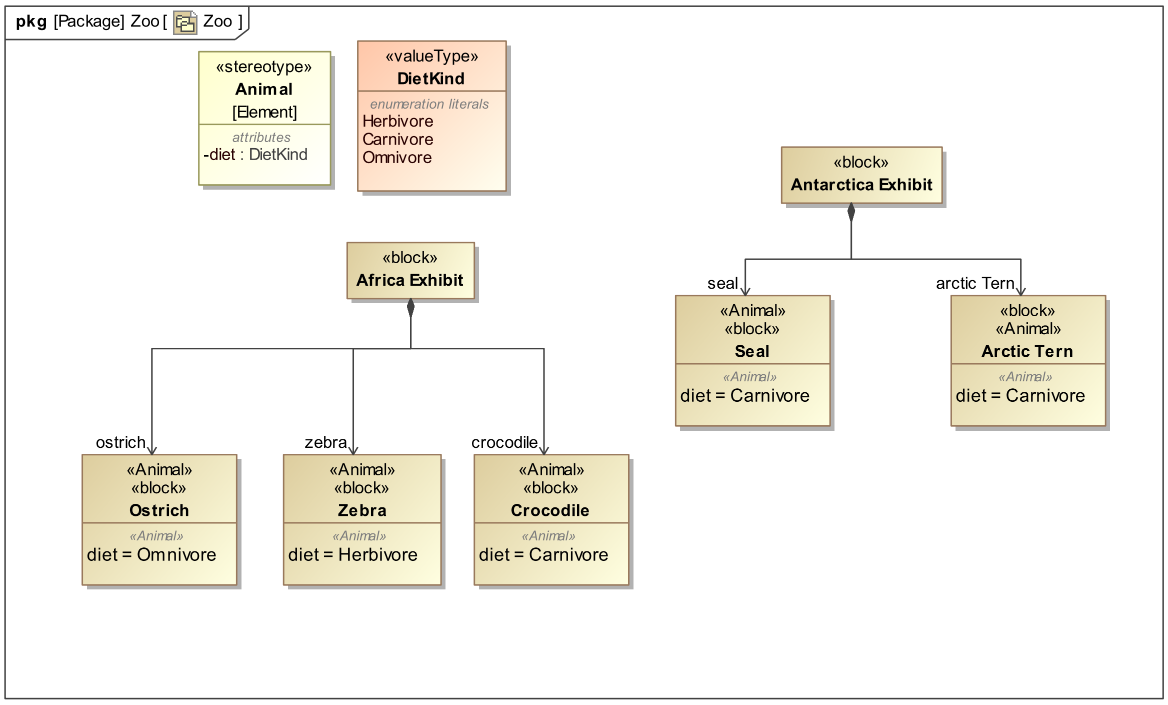
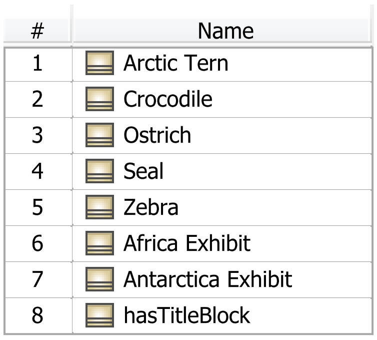


Figure 4.67. Zoo Animals



#	Name
1	Arctic Tern
2	Crocodile
3	Ostrich
4	Seal
5	Zebra
6	Africa Exhibit
7	Antarctica Exhibit
8	hasTitleBlock

Paragraph

Paragraph is a presentation element designed to display text. Every view requires a viewpoint method, if one is not specified the default method is a single paragraph action targeting the documentation of the view element. This is added automatically without being visible to the user.

This section describes advanced ways you can: create paragraphs, combine paragraphs with other viewpoint method actions, and generate paragraphs using OCL.

Paragraph is a combination of the features of Image and Table. It is similar to an image in that it will display the documentation of any element that is exposed by the view. It also has some advanced expression features that allow it to additionally display the attributes of an element exposed by a view (similar to TableAttributeColumn) or result of an expression (similar to TableExpressionColumn).

In general the results of a paragraph allow greater flexibility to display view editor editable content from that model than information displayed within the rigid constraints of an Image or Table.

The following sections describe various ways a paragraph can be used.

Paragraph Action with Targets

Figure 4.68. Generic Paragraph View Diagram

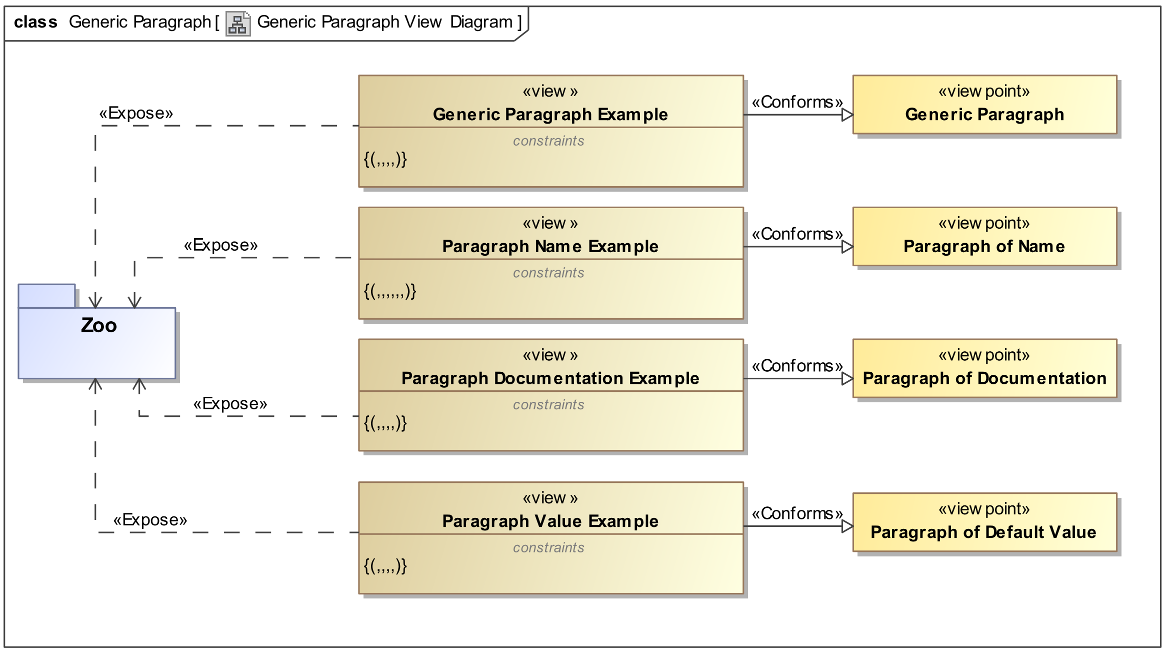


Figure 4.69. Generic Paragraph

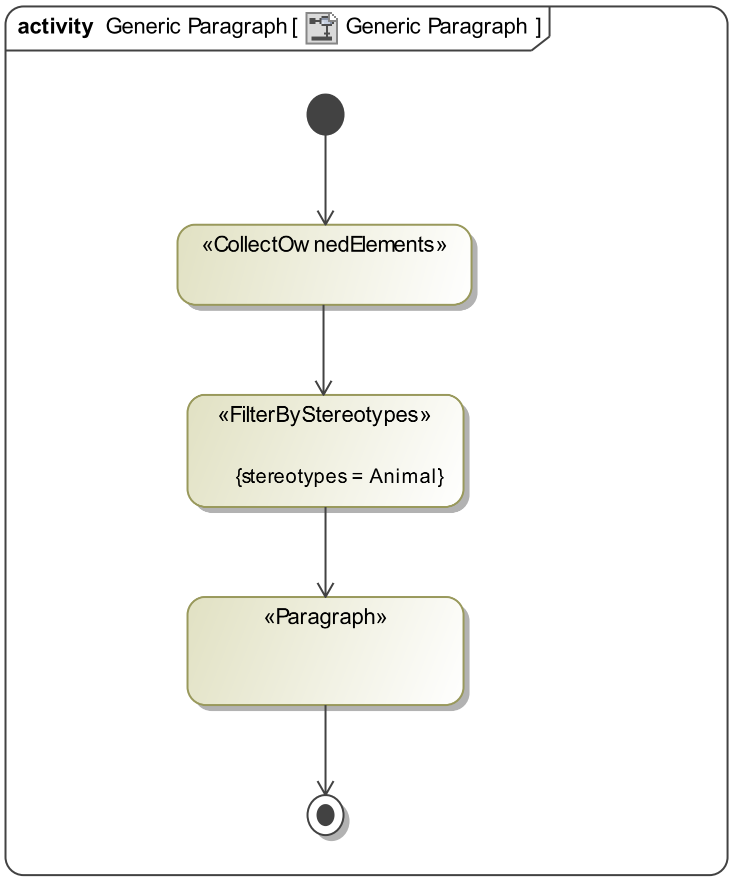


Figure 4.70. Paragraph of Name

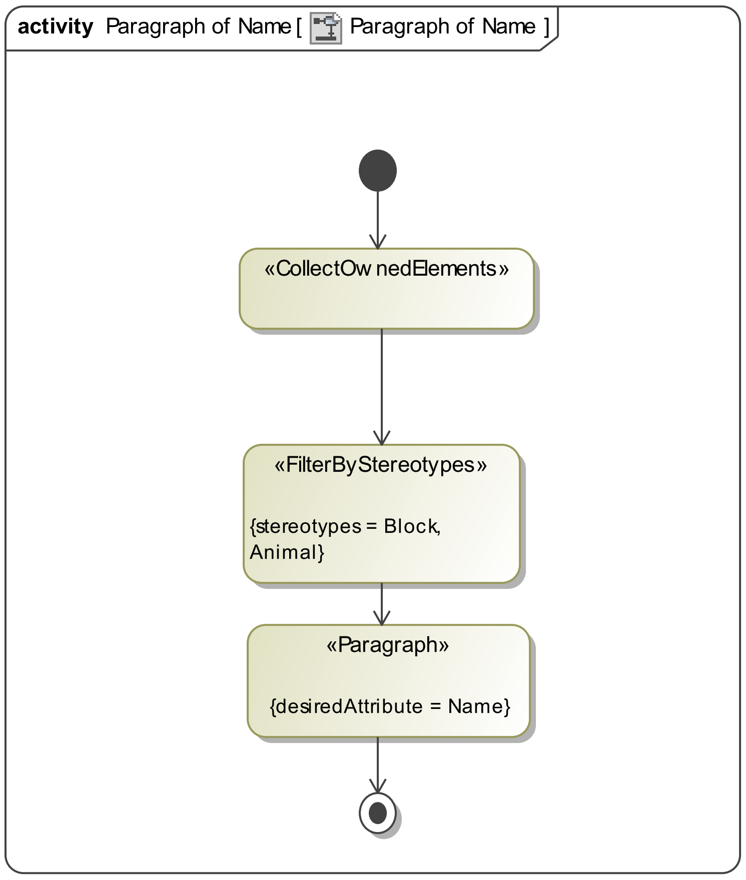

activity Paragraph of Name [Paragraph of Name]
«CollectOwnedElements»
«FilterByStereotypes»
{stereotypes = Block, Animal}
«Paragraph»
{desiredAttribute = Name}

Figure 4.71. Paragraph of Documentation

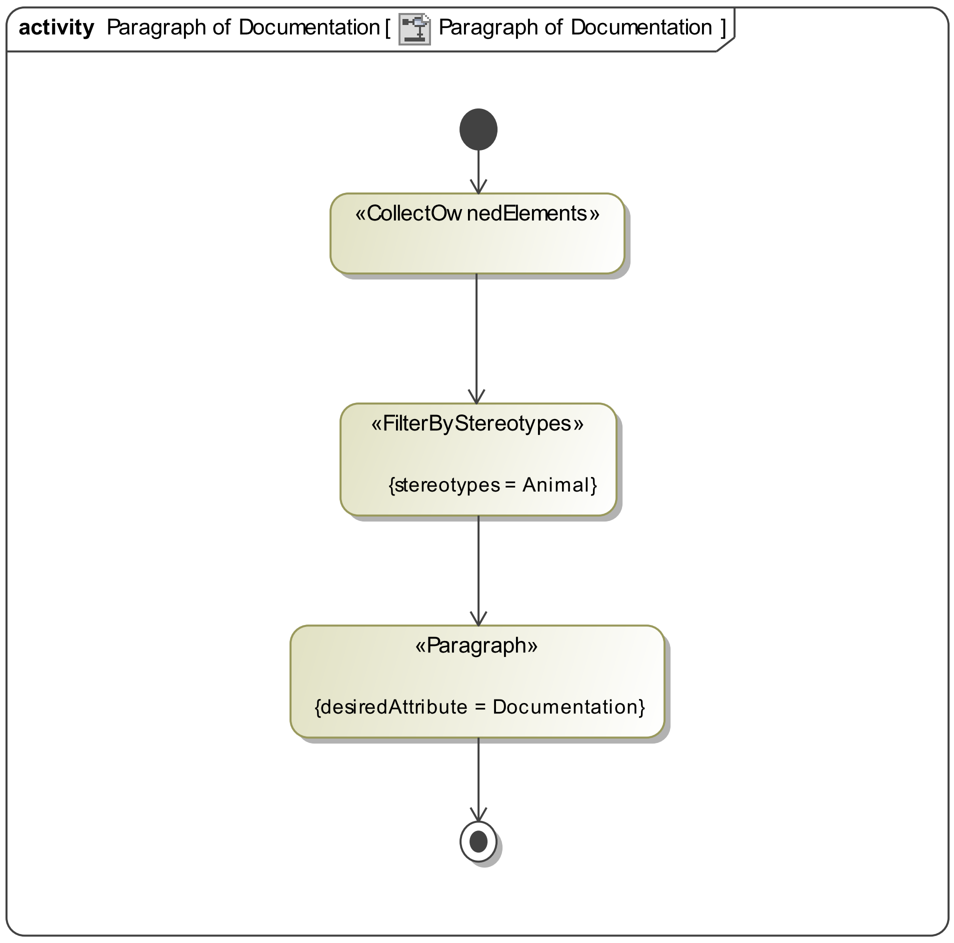
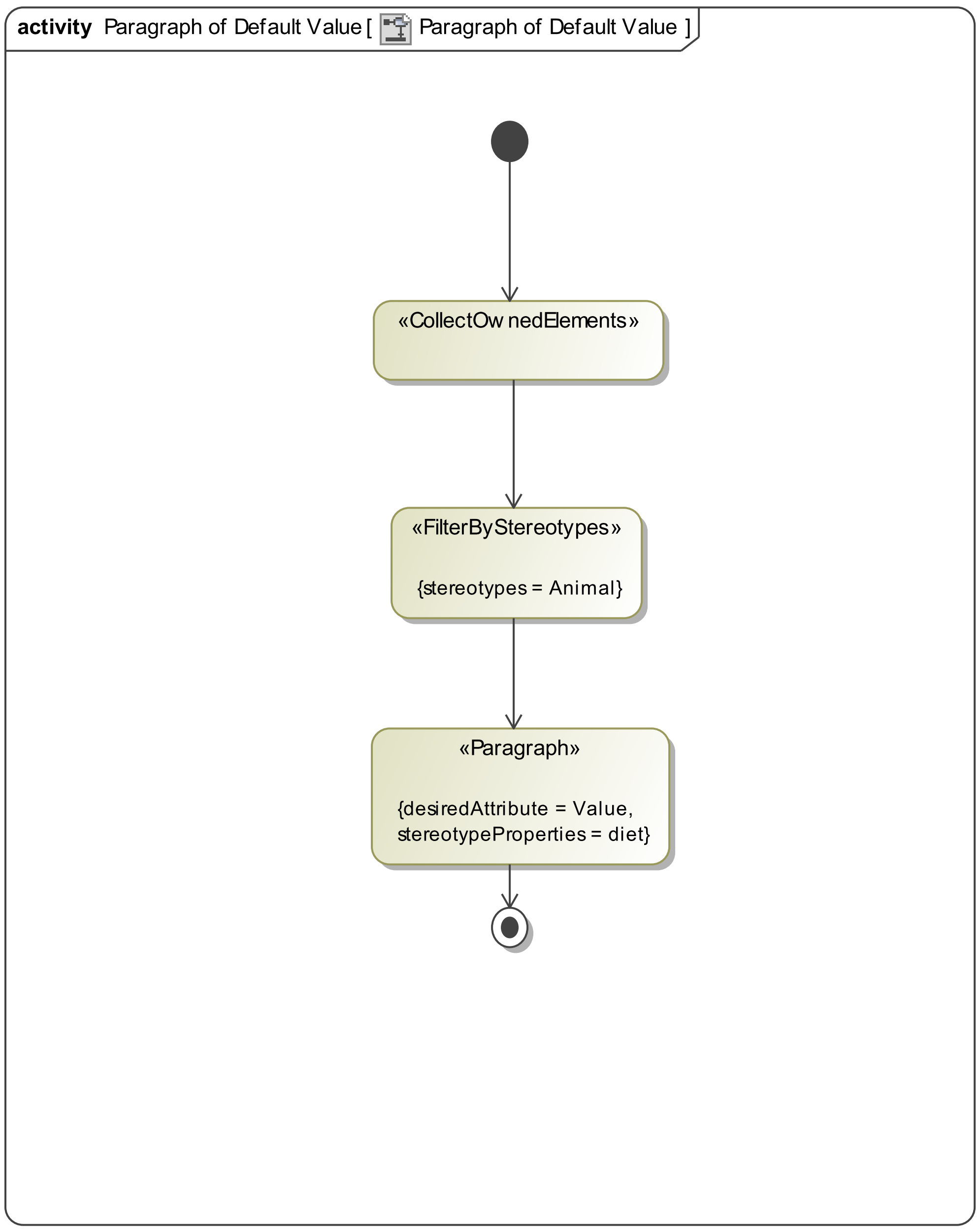

activity Paragraph of Documentation [Paragraph of Documentation]
«CollectOwnedElements»
«FilterByStereotypes»
{stereotypes = Animal}
«Paragraph»
{desiredAttribute = Documentation}

Figure 4.72. Paragraph of Default Value


activity Paragraph of Default Value [Paragraph of Default Value]
«CollectOwnedElements»
«FilterByStereotypes»
{stereotypes = Animal}
«Paragraph»
{desiredAttribute = Value,
stereotypeProperties = diet}

Generic Paragraph Example

Ostriches can run up to 70 km/h, the fastest land speed of any bird.

Basically a dinosaur.

Zebras feed almost entirely on grasses.

Mostly blubber.

What sound does an arctic tern make?

Paragraph Name Example

Africa Exhibit

Ostrich

Crocodile

Zebra

Antarctica Exhibit

Seal

Arctic Tern

hasTitleBlock

Paragraph Documentation Example

Ostriches can run up to 70 km/h, the fastest land speed of any bird.

Basically a dinosaur.

Zebras feed almost entirely on grasses.

Mostly blubber.

What sound does an arctic tern make?

Paragraph Value Example

Omnivore

Carnivore

Herbivore

Carnivore

Carnivore

Paragraph Action with Body

Figure 4.73. Example View Diagram

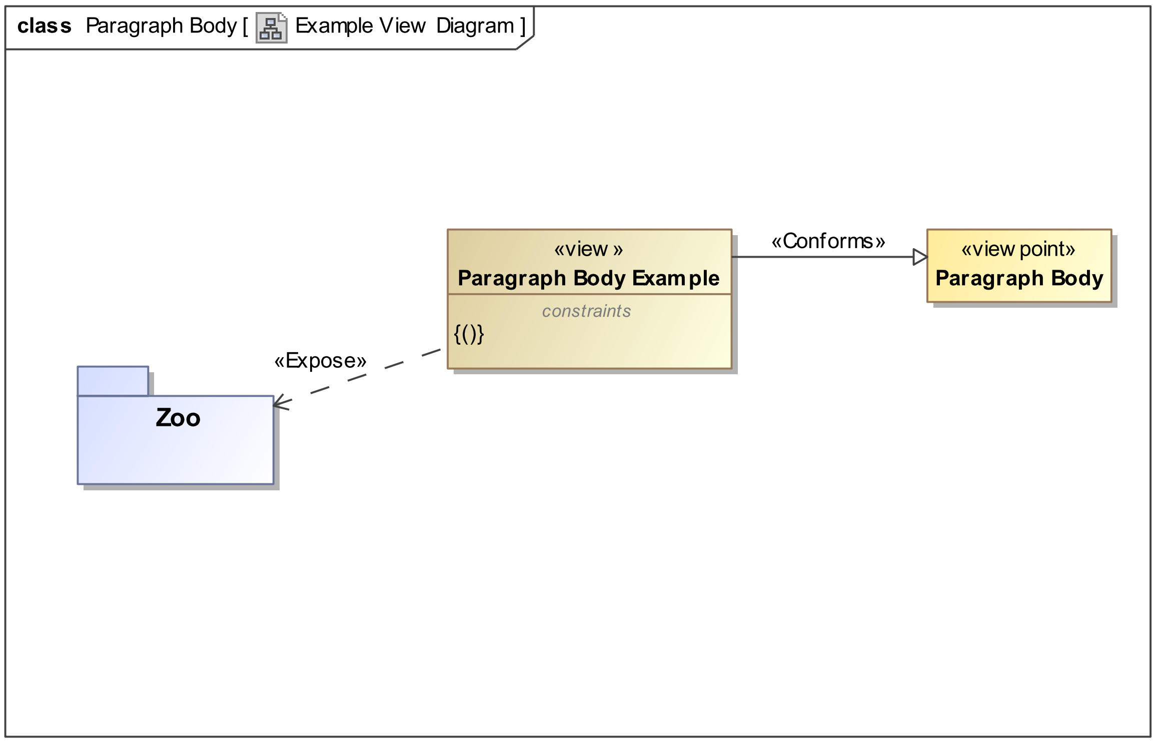
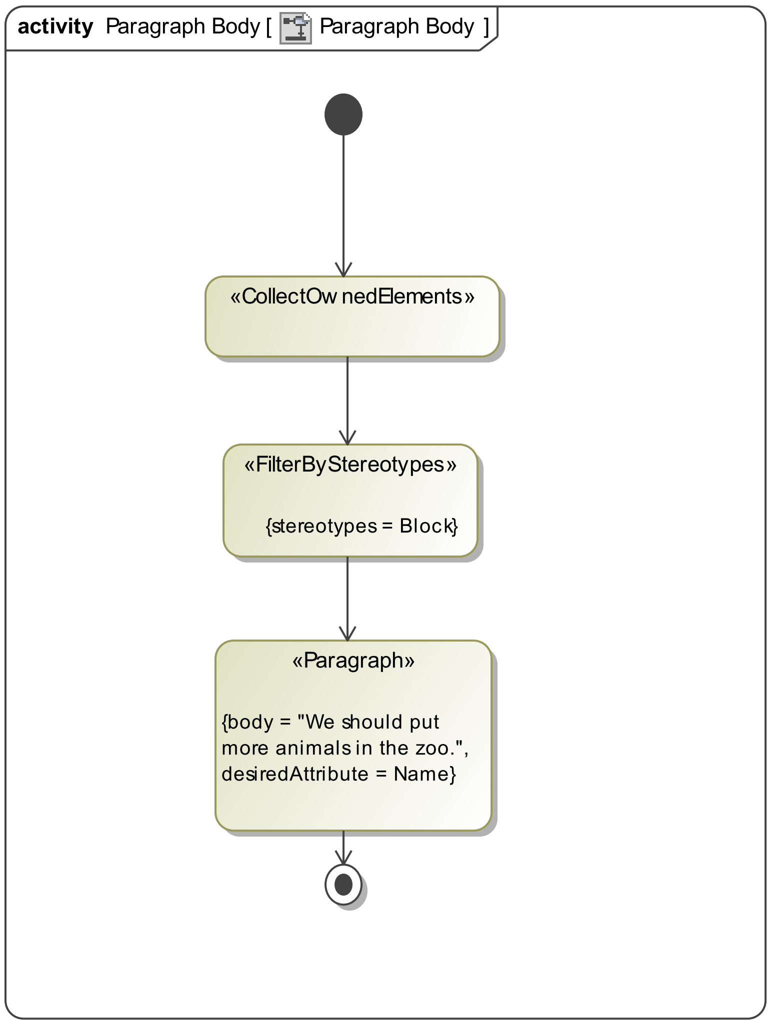


Figure 4.74. Paragraph Body


activity Paragraph Body [Paragraph Body]
«CollectOwnedElements»
«FilterByStereotypes»
{stereotypes = Block}
«Paragraph»
{body = "We should put more animals in the zoo.", desiredAttribute = Name}

Paragraph Body Example

We should put more animals in the zoo.

Paragraph Action Evaluate OCL

Figure 4.75. Example View Diagram

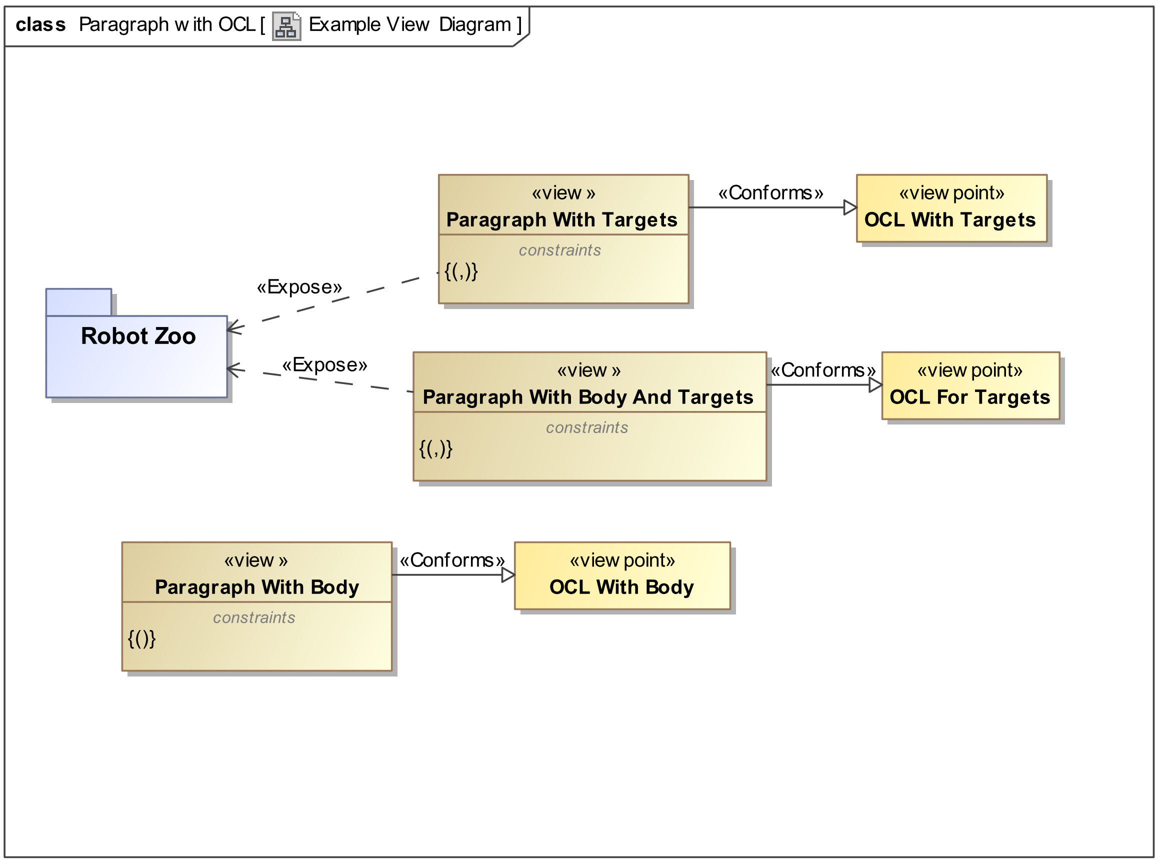


Figure 4.76. OCL With Targets

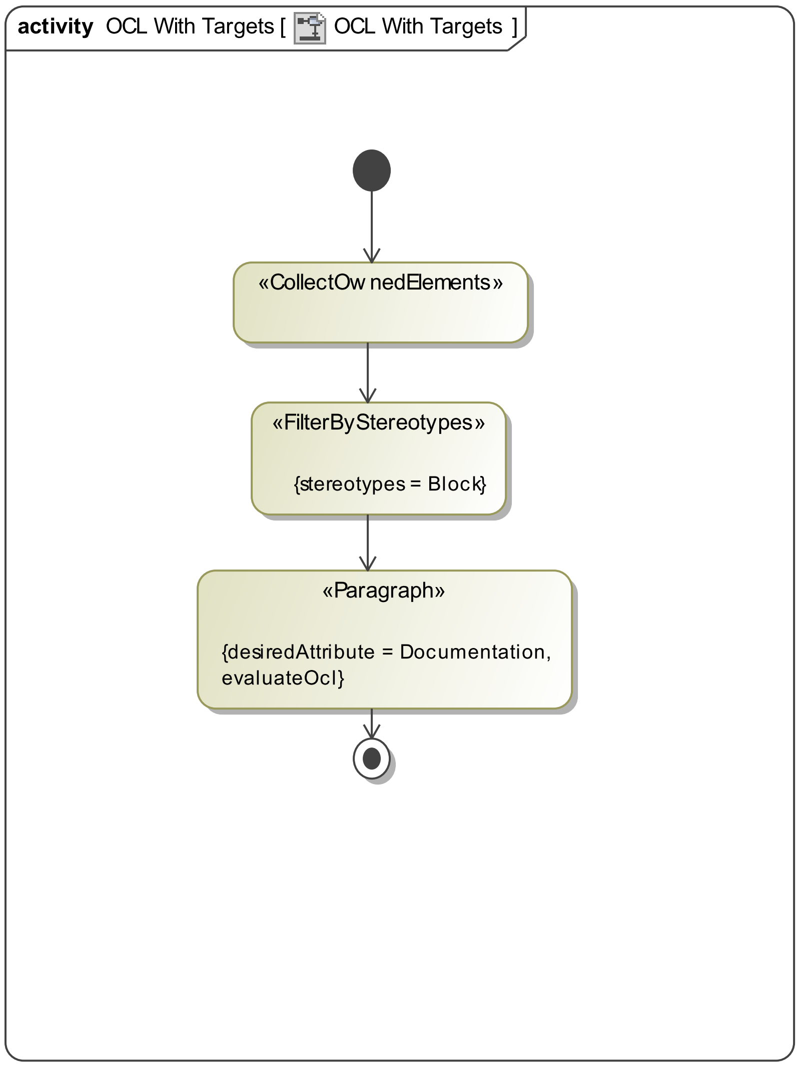

activity OCL With Targets [OCL With Targets]
«CollectOwnedElements»
«FilterByStereotypes»
{stereotypes = Block}
«Paragraph»
{desiredAttribute = Documentation, evaluateOcl}

Figure 4.77. OCL Without Targets

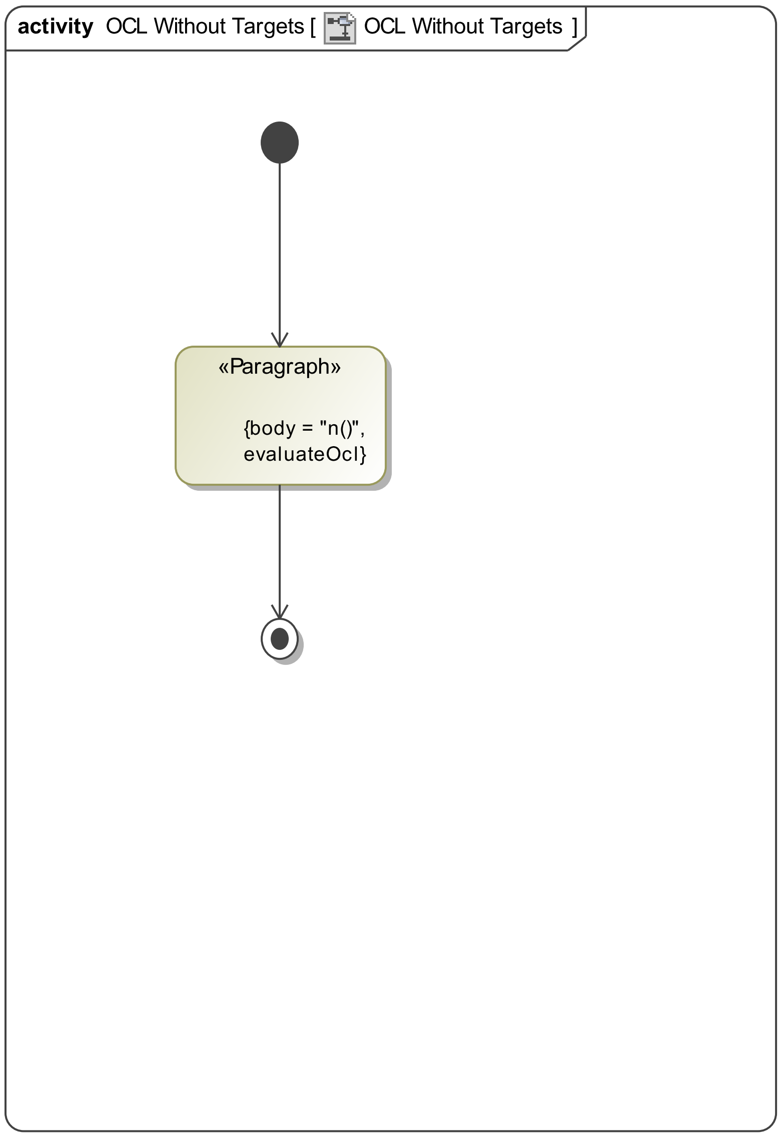
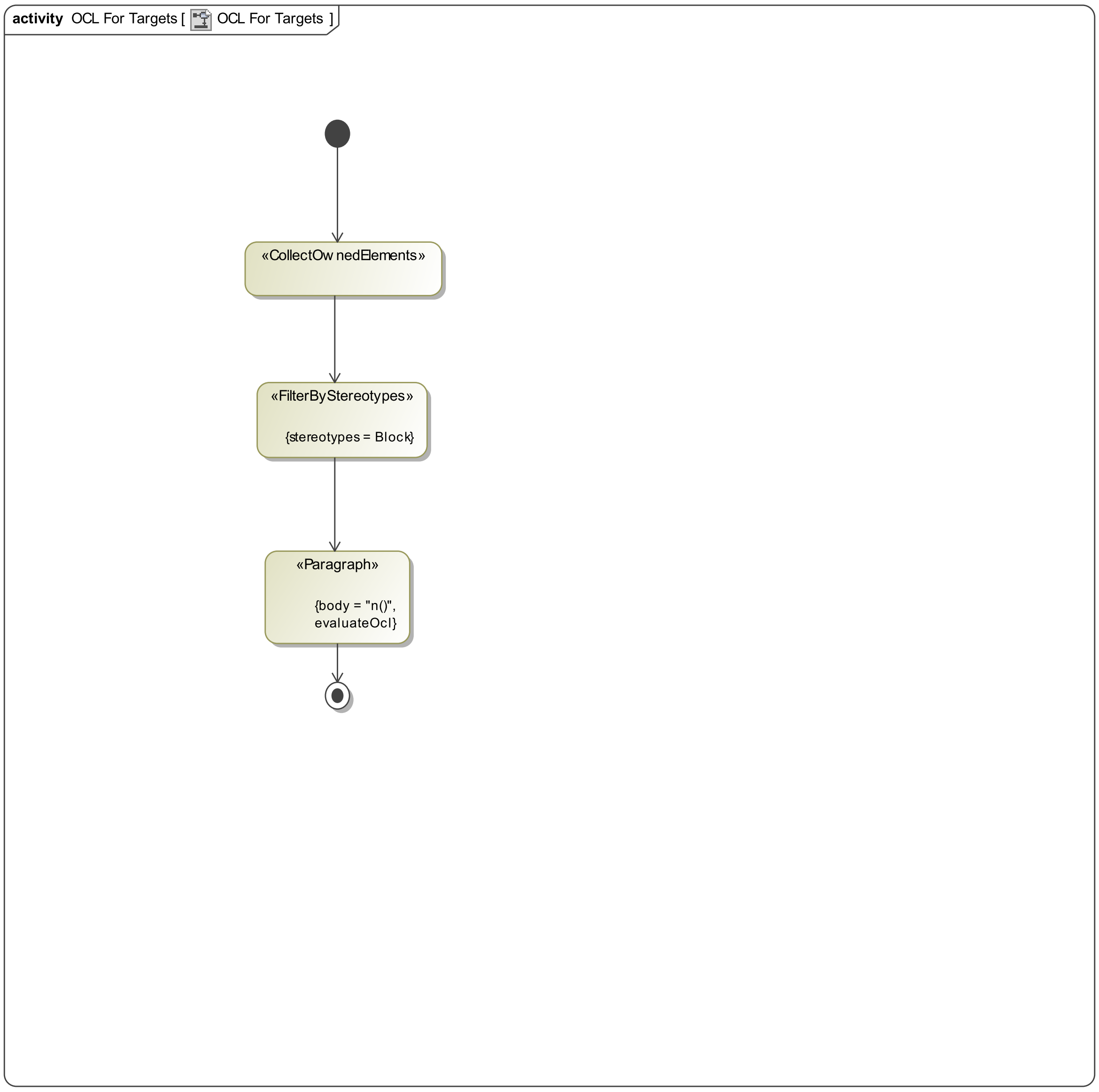


Figure 4.78. OCL For Targets



Paragraph With Body

Paragraph With Body

Paragraph With Targets

true

false

Paragraph With Body And Targets

Robot Moose

Robot Squirrel

List

"BulletedList" creates lists based on the model elements exposed to the behavior. This one presentation element can create either an ordered (numbered) list, like the one shown below, or a bulleted list like those that have been displayed in previous examples. What information is displayed (names, documentation, stereotype property values) depends on the options selected in the behavior's specification and the filters applied to the collected data. For example, when "Show Targets" is "true", the name of the element is listed.

The viewpoint method diagram below shows the diagram that was used to create the below example list. The exposed package was "Zoo" from previous examples and a filter was applied to only identify the <<Animal>> stereotypes. To create this example, "Ordered List" was selected to be "true" in order to create a numbered list. If that option had been false, then the list would be bulleted instead. Inside the bulleted list specification, there are a number of other options. If you click on an option, an explanation appears in the bottom box.

NOTE: "Show Stereotype Property Names" currently doesn't work. It theoretically prints out the stereotype property name before listing its values.

Figure 4.79. Example List View Diagram

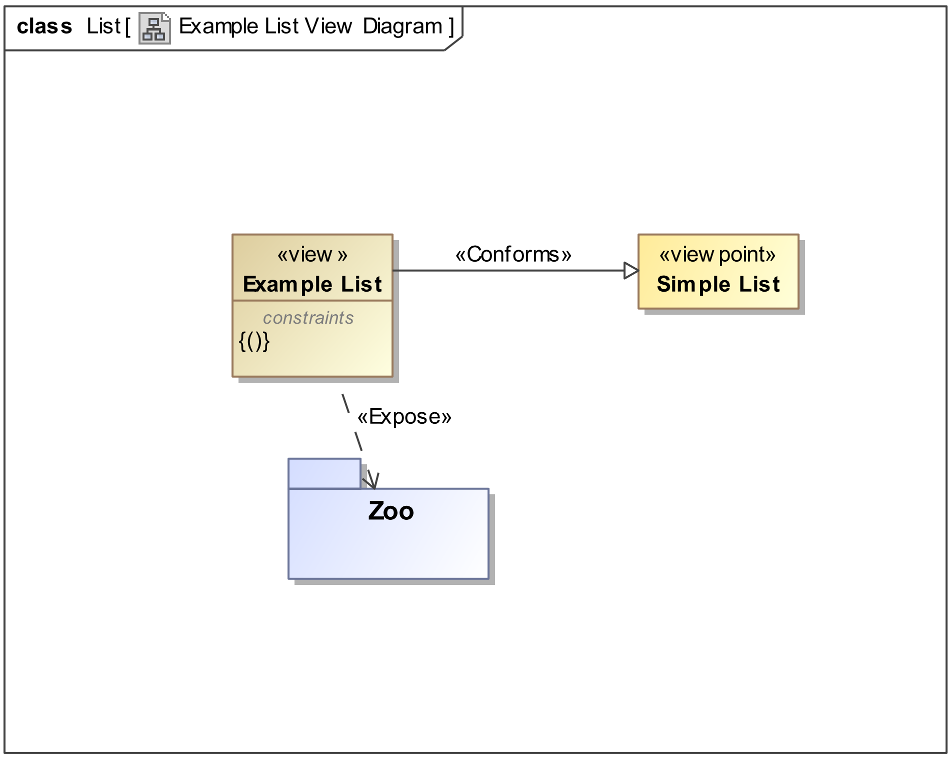
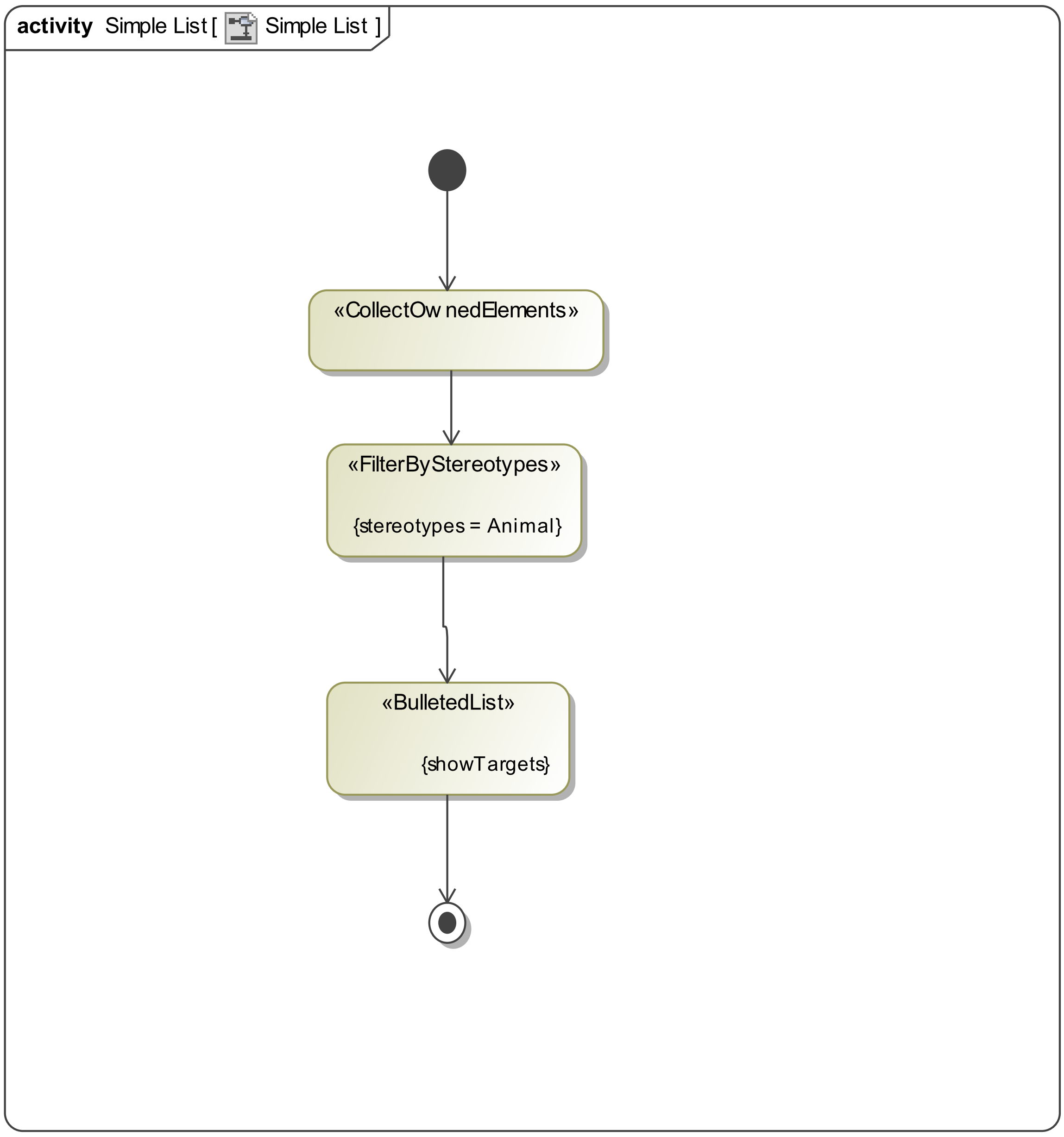


Figure 4.80. Simple List



Example List

- Ostrich
- Crocodile
- Zebra
- Seal
- Arctic Tern

Dynamic Sectioning

Dynamic sectioning is the creation of viewpoint method defined sections. They are created using the "Structured Query" activity in the viewpoint method diagram.

This section describes two main types of dynamic sectioning: the creation of a single section and the creation of multiple sections. The new dynamic section(s) can be distinguished from a standard

<<view>> in the table of contents by a small page symbol, §. Notice that the dynamic section(s) also load into the view at the same time as the parent view.

For these examples, the package "Animals", first introduced in the paragraph examples, was exposed. Each of the five animal elements consists of an element title and some documentation text.

These sections have two main uses, basic organization allowing sections to be broken up. Formally, dynamic sections allow further organization of views while preserving the canonical view hierarchy.

Figure 4.81. Example View Diagram

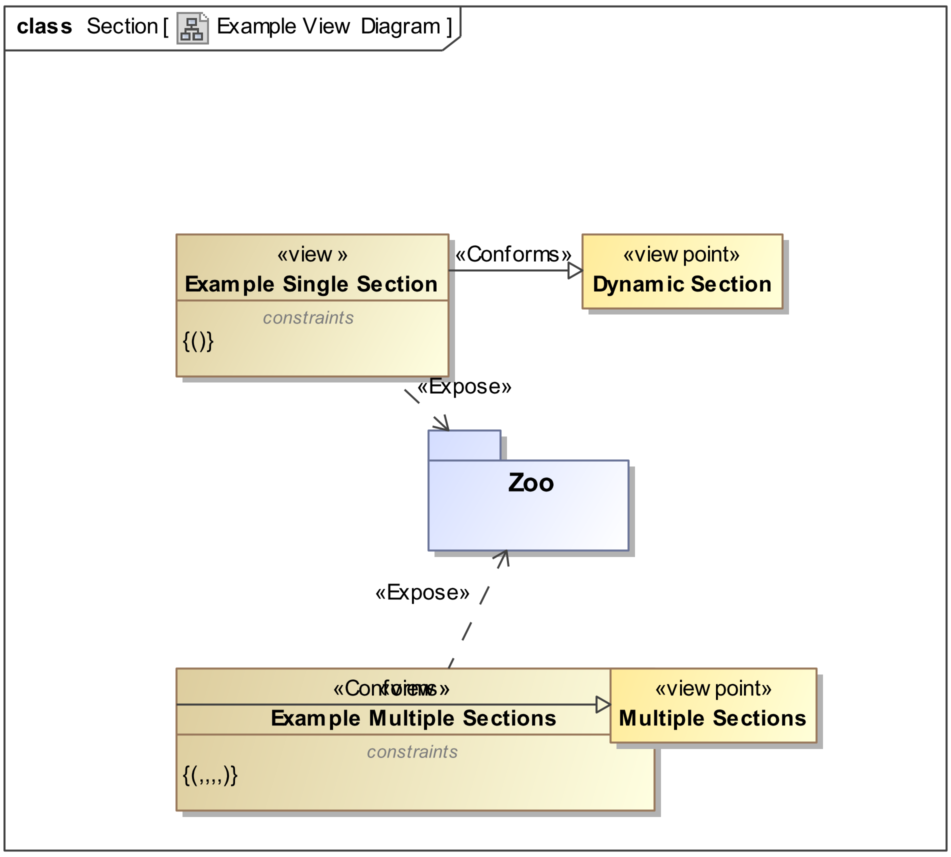


Figure 4.82. Dynamic Section

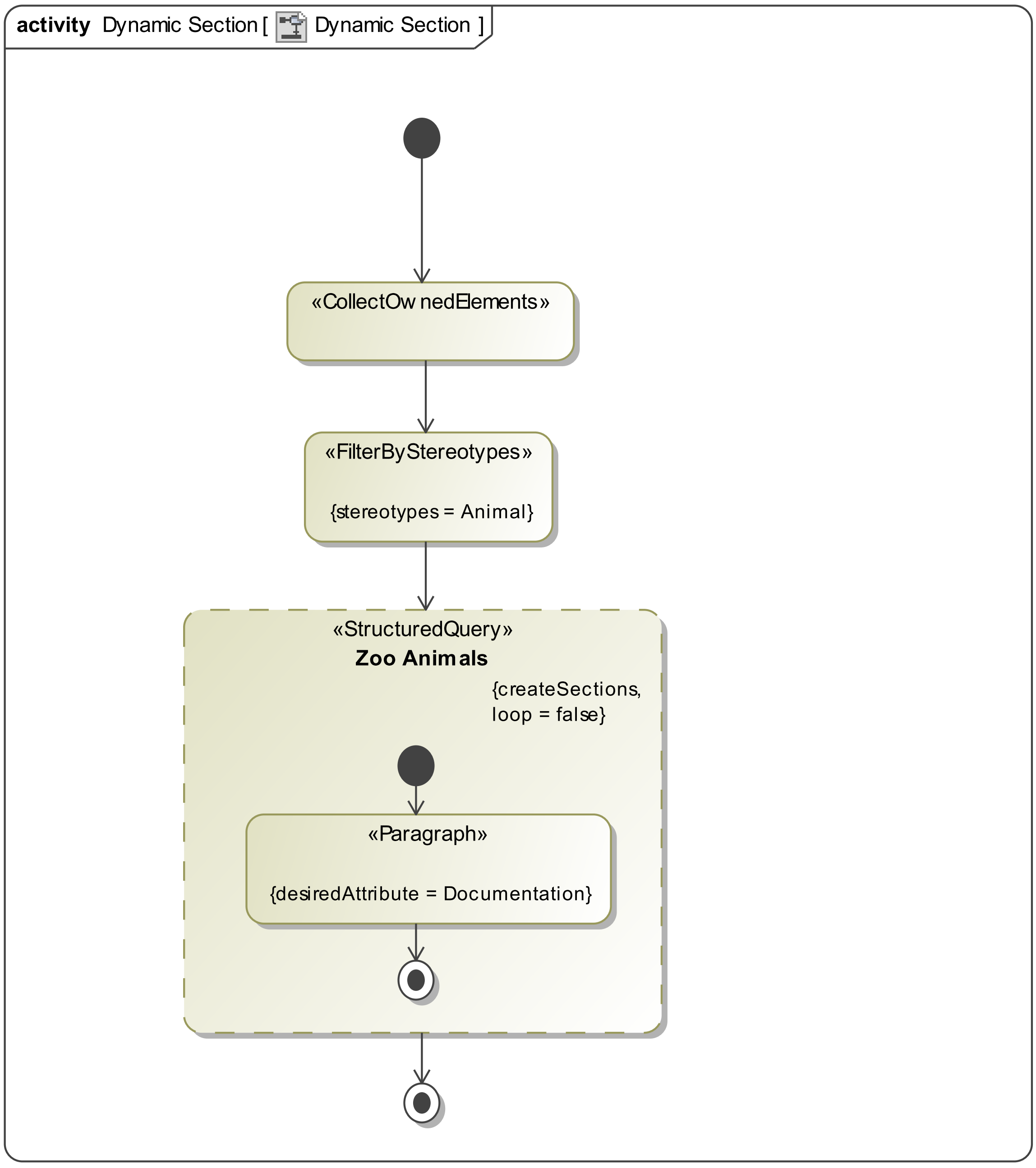
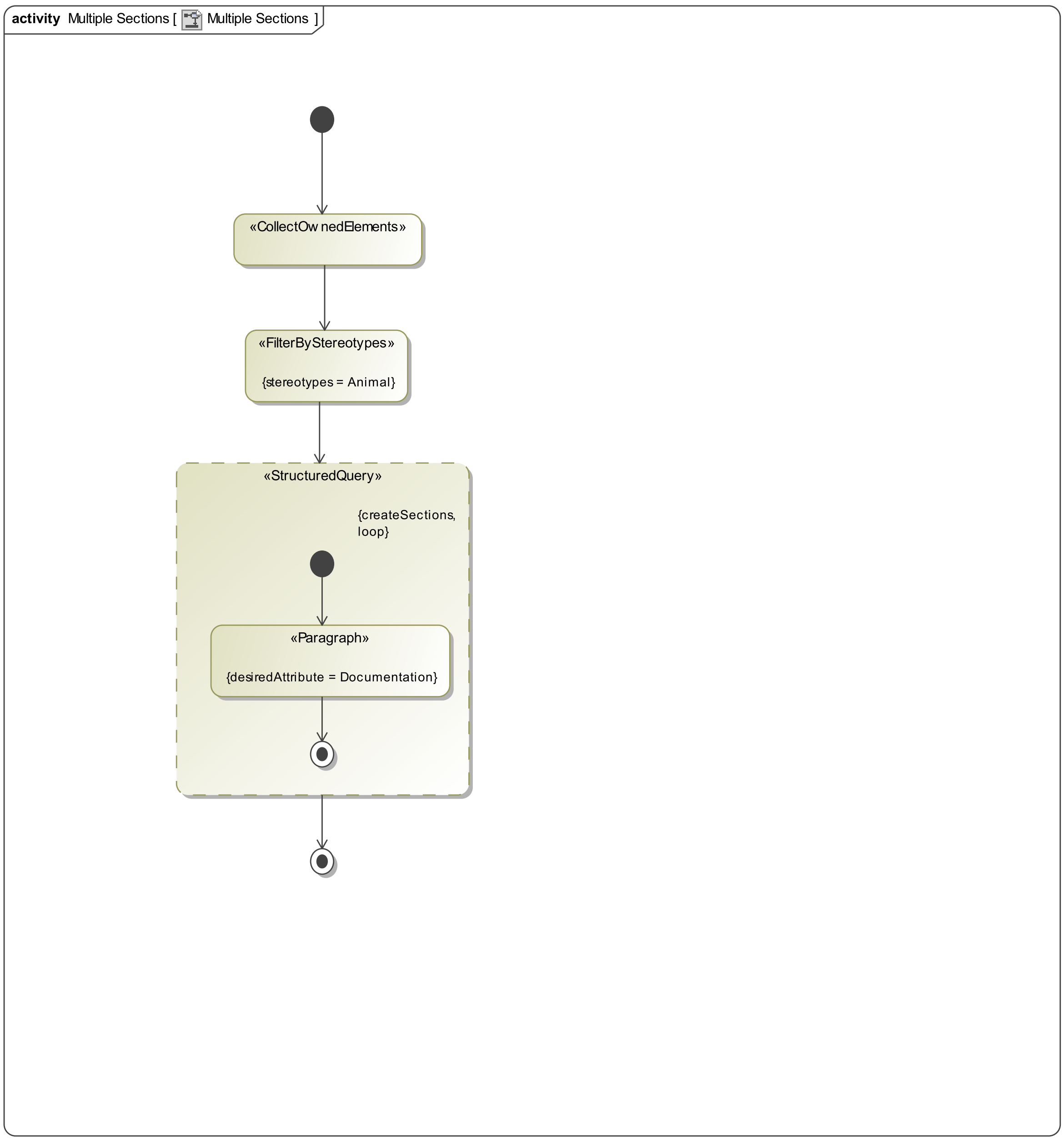


Figure 4.83. Multiple Sections



Example Single Section

Zoo Animals

Ostriches can run up to 70 km/h, the fastest land speed of any bird.

Basically a dinosaur.

Zebras feed almost entirely on grasses.

Mostly blubber.

What sound does an arctic tern make?

Example Multiple Sections

Ostrich

Ostriches can run up to 70 km/h, the fastest land speed of any bird.

Crocodile

Basically a dinosaur.

Zebra

Zebras feed almost entirely on grasses.

Seal

Mostly blubber.

Arctic Tern

What sound does an arctic tern make?

Chapter 5. DocGen UserScripts

This section details some of the advanced features of MDK that allow the integration of User-Scripts into your DocGen queries. Be sure to check your Project's Modeling Policy with regard to script permissions.

Creating UserScripts

1. Create a stereotype that specializes the UserScript stereotype in the SysML Extension profile
2. Name the stereotype using the format <namespace>.<scriptName>
3. Apply the <<DocGenScript>> stereotype from SysML Extension in order to change the default interpreter language of your UserScript stereotype. The default language without the <<DocGenScript>> stereotype is Jython
4. Create your script under <md.install.dir>/DocGenUserScripts/<namespace> The namespace and script name will be the same as the namespace and scriptName of your stereotype
5. Apply your new specialized UserScript stereotype to an Action. Use this action in a Viewpoint Method. You may also select the "Other (like UserScripts)" Template Button in the Viewpoint Method pallette, and then type the name of your newly created UserScript.
6. Create a View that conforms to your Viewpoint and exposes some Element
7. Set the "collectViewActions" tag in the View to "true" in order to invoke the UserScript during DocGen generation. This tag is part of the "view" stereotype, and by default is set to "false"
8. Generate the Document locally or Generate Views and commit to MMS in order to see the output as usual

Writing UserScripts

When executing a UserScript, DocGen makes available a mapping or dictionary called "scriptInput". This contains key-value mappings of information available to your script. The most important one is "DocGenTargets". This contains a list of MagicDraw elements that are being passed to your script and is the starting point in the model for your script. Below is a list of inputs available:

- DocGenTargets - list of MagicDraw elements
- md_install_dir - magicdraw installation directory as a string
- docgen_output_dir - if locally generating, this is the output directory as a string
- ForViewEditor - boolean, whether the current execution is for the view editor
- other properties defined by your stereotype*

* You can define tags on your stereotype just like the built in ones have tags defined, and those will become inputs as a list. For example, if you defined "booleanTag" on your stereotype and on the action, the tag is set to True, you'll see scriptInput['booleanTag'] = [True]

Presentation UserScripts

For output, DocGen would be looking for a "scriptOutput" object with a key-value pair of "DocGenOutput" to be a list of DB* objects. The DB* objects are detailed in the javadoc [http://docgen.jpl.nasa.gov/opsrev/downloads/docgen/javadoc/]. They are basically Java class representations of Table, Paragraph, or List. Below is example code that outputs a table:

Collect and Filter UserScripts

Collect/Filter UserScripts are custom scripts that provide collect/filter capability instead of the built in ones. You can use this anywhere you can use a built in collect/filter. The setup is similar to other UserScripts except your stereotype needs to specialize the <<Collect/Filter Userscript>> stereotype.

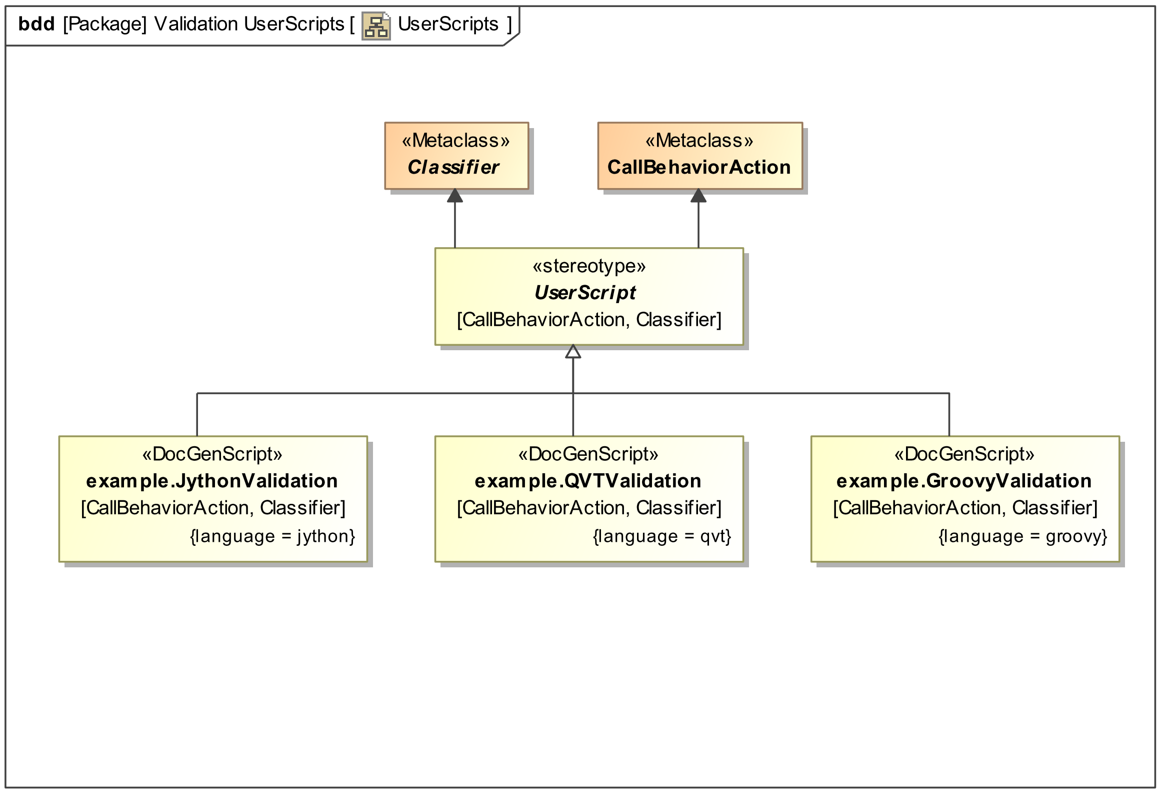
The passed in values to the script would be the same, but DocGen would be expecting a list of elements as the output.

```
#this is a script implementation of get owned elements with depth 1
targets = scriptInput['DocGenTargets']
output = []
for target in targets:
   for e in target.getOwnedElement():
      output.append(e)
scriptOutput = {"DocGenOutput": output}
```

Validation UserScripts

Validation scripts can be used to output a common table layout for validation suite, rules, and violations. A validation suite consists of one or more rules, and each rule can have one or more violations. A view that conforms to a viewpoint with some validation action can also be run inside Magicdraw that's tied into the Magicdraw validation window.

Figure 5.1. UserScripts



Jython Validation Example

Chapter 6. Create and Evaluate OCL Constraints

This section is a **brief** introduction to Object Constraint Language (OCL) and how it is used within the MDK. It is important to note that this is not a comprehensive explanation of OCL and at any point during this tutorial if you find yourself wanting a more in-depth explanation of the language, (i.e. specific syntax rules), there are plenty of resources to be found and several are offered in the next section.

What Is OCL and Why Do I Use It?

OCL Primer By Bradley Clement

What Is OCL?

The Object Constraint Language (OCL) [http://en.wikipedia.org/wiki/Object_Constraint_Language] is a text language (a subset of QVT [http://en.wikipedia.org/wiki/QVT]) that can be used to customize views and viewpoints in various ways that otherwise may require writing external code in Jython, QVT, Java, etc. OCL is used to define invariants of objects and pre-and post conditions of specified operations. This allows for more advanced querying of model elements and their properties. Throughout this entire section the words "expression" and "operations" will be used extensively. **An OCL "expression" is a statement of OCL "operations" that can be pieced together to get what you want**. These expressions can be quite simple or complex depending on the need.

For example:

1. **n()**

2.
r('analysis:characterizes').oclAsType(Dependency).source.m('mass').oclAsType(Property).defaultValue.oclAsTy

.value.toInteger()=eval(self._constraintOfConstrainedElement->asSequence()->at(1).oclAsType(Constraint)

.specification.oclAsType(OpaqueExpression)._body->at(1))

Both 1 and 2 are OCL expressions, however, it is obvious to see that number 2 is quite a bit more complex. Number 1 is a single **operation** that stands alone as an **expression**. It is important to note that these expressions are built by stringing along carefully selected operations and separating them with the proper syntax. In the complex example above you will notice a lot of periods (.) and arrows (->). Both of these symbols separate one piece of the expression from the next (there are more than just these two). It will take some practice to understand which to use and when to use it. For a more extensive/advanced explanation of OCL syntax check out this link [http://santos.cis.ksu.edu/771-Distribution/Reading/Richters_2001_OCL.pdf].

Why Do I Use OCL?

You can use OCL to specify how to collect, filter, sort, constrain, and present model data. Constraints on model elements are the driving force behind MDK and the inter-workings of certain elements presented earlier in this document will be explained via OCL. Several new elements that were created specifically for dealing with OCL expressions within a viewpoint will be introduced as well. For your everyday collect and filters of model elements OCL probably isn't the best choice, however, for more advanced queries of the model it can be your golden ticket. Examples outlining this idea will follow.

OCL syntax can be tricky since it accesses model information through the UML metamodel (see the UML metamodel manual in your Magicdraw installation directory, under manual). Here we try to give some tips on how to write OCL expressions to more easily customize views without having to write external scripts in Jython, Java, QVT, etc. Keep in mind, this topic is more advanced than previous ones discussed earlier in this document. At this point it is assumed that you understand how a viewpoint method diagram works and can walk step by step from one action to the next and comprehend what elements should be expected in the final result. This is important because OCL expressions work the same way.

OCL Resources

For help with OCL, you can try the
<a>OCL Cheat Sheet</a>
or search the web for example OCL. Some examples will also be given in the sections below. The OCL Cheat Sheet is very helpful once you get a basic understanding of how OCL works, however, until you have gained some understanding it might be confusing. In the following sections some references to this resource will be made with explanations of how it can be useful.

NoMagic's UML metamodel specifies what objects can be referenced in an OCL expression for the different UML types. A pdf manual of the metamodel can be found in your MagicDraw installation in the manual folder. If you use Eclipse, try opening the Metamodel Explorer view from the menu: Window -> Show View -> Other. There may be many metamodels, but look for the com.nomagic.uml2 metamodel. There is a button on the view where you can search for a type, but be careful since there may be more than one UML metamodel. You want MagicDraw's UML metamodel. There is also a button for showing inherited members that can be referenced in OCL.

What are the OCL Black Box Expressions?

Black box operations are often used to simplify expression structure. The following section will cover several of these so that you will recognize them. However, before discussing the black box operations it is important to talk about the viewpoint method action <<TableExpressionColumn>> since it will be used in the following example viewpoint methods.

<<**TableExpressionColumn**>> Applies an OCL expression to the target elements in a <<TableStructure>> that will populate a column of the table displayed in View Editor. This can be used to chain operations on elements and relationships that would otherwise be difficult or impossible.

Make sure to focus directly on this action in the examples. Understanding how they handle OCL expressions is important.

Another critical topic before moving on is the idea of "**casting**" in your OCL expressions. Many times when a result is returned from an OCL operation it will need to be cast as the element expected from the query in order for further operations to be carried out. This is the nature of OCL and just takes getting used to. Often times errors can be fixed by remembering to cast the returned elements. The cheat sheet discussed earlier gives the operations needed for casting. They are shown below and will be implemented and explained in the following examples. Remember to come back and reference these operations while working through the examples.

Below are the OCL black box operations used as shorthand in expressions. They will be demonstrated in the following sections.

- **m(), member(), or members()** returns the owned elements

- **r(), relationship(), or relationships()** returns all relationships owned or for which the element is a target
- **n(), name(), or names()** return the name
- **t(), types()** return all types (Stereotypes, metaclass, and Java classes and interfaces).
- **type()** returns just the type of the element selected.
- **s(), stereotype(), or stereotypes()** returns the Stereotypes. This is basically short for appliedStereotypeInstance.classifier. s() and stereotypes() should return all stereotypes, and stereotype() should just return one.
- **e(), evaluate(), or eval()** evaluates an ocl expression retrieved from an element

- **value()** returns the value of a property or slot
- **owners()** returns owner and owner's owners, recursively
- **log()** prints to Notification Window
- **run(View/Viewpoint)** runs DocGen on a single View or Viewpoint with specified inputs and gets the result

Many of these views reference results of the example viewpoints. These results are similar to those found in the
<mms-view-link> [cf:Present Model Data.vlink] </mms-view-link>
 view.

value()

The expression **value()** is a useful tool to get at the value of a property. In the example below the Animals package was exposed to the viewpoint method diagram shown. Inspecting the expression [**self.get('noise').v().v()**] shows **get()** collecting the elements named "noise," in this case all the properties. **v()** is then used twice, once to get the property default value, literal string. It is then used again to get the text of the literal string.

Relationships "r()"

The following is an outline of the operation r(). The viewpoint method below is designed in such a way that it will use r() by itself and also part of a more complex expression. The bullet list below will run through the explanations of the OCL expressions used in the <<TableExpressionColumn>> below.

- **r(), relationship(), or relationships()** returns all relationships owned.
  - **r('likes')** in the first <<TableExpressionColumn>> gets all of the relationships of name or type 'likes'
  - **r('likes').oclAsType(Dependency).target** in the second <<TableExpressionColumn>> gets the 'likes' relationship, casts it as Dependency then collects the target of the relationship (Cat). Keep in mind, when **r()** returns the relationship 'likes,' in order to keep performing operations we have to cast what was returned. This is done by **oclAsType()**. Look over the cheat sheet discussed earlier to become more familiar with these operations.
  - Note: ignore the filters in the viewpoint method diagram, they are unimportant to the concept covered in this section.
  - Notice the <<TableExpressionColumn>> activities are where the OCL expressions are defined in the viewpoint method diagram.

Names "n()"

- **n(), name(), or names()** returns the name

For this example, expression "n()" returns the name of the element passed to the action. In this case "Dog."

Stereotypes "s()"

- **s(), stereotype(), or stereotypes()** returns the Stereotypes. This is basically short for appliedStereotypeInstance.classifier. **s()** and **stereotypes()** should return all stereotypes, and **stereotype()** should just return one.

For this example, Dog is exposed and it's stereotypes populated into a table.

Members "m()"

- **m(), member(), or members()** returns the owned elements
  - m('text') gets the member whose name or type is 'text'

Example:

- Since the element Dog is again exposed, **m('noise')** will get the member whose name is 'noise' and post that name to the Attributes column of some particular table.
- Notice in the <<TableExpressionColumn>> actions of the viewpoint method are using different OCL expressions. The "Attributes" column is returning any member who's name is 'noise'. The expression in the "Attribute Default Value" action is collecting the same property, casting it as a Property and then returning the default value. This is done by: **m('noise').oclAsType(Property).defaultValue**.

Evaluate "eval()"

eval() or **e()** evaluates an OCL expression retrieved from an element. This example may seem overwhelmingly complex, however, the main point is to demonstrate the use of the black box expression. The image below shows an element (Propulsion Subsystem) which has a set constraint (red arrow).

This is a constrained element with the constraint specification:

The example expression is long but take the time to try and piece it together to help your understanding. A useful hint is to start with the oclAsType() operations since these will give you a clue as to what was returned by the previous operation.

This is eval() taking in the constrained element and evaluating the constraint:

This is the complete viewpoint method diagram:

Types "t()"

t() returns the type(s) of the element selected where as, **type()** only returns the type of the element exposed re. Element "Dog" is of type Class and has other types associated with it. We would like to return just the type of element Dog, therefore, we use **type()** in the <<TableExpressionColumn>>.

Note: the other information returned with Class is an artifact of MagicDraw.

owners()

Returns owner and owner's owners, recursively. In the example below, the OCL evaluation tool is used to collect the owners of the element "Dog" In the results of the evaluation box, the owning packages and model are shown. Note: the fist part of the expression (**self.**) is specifying the target element. The operation would work without it since it is just one element being exposed.

The red arrows below show the package hierarchy collected.

Table 6.1.

Owners
What are the OCL Black Box Expressions? How Do I Get Started Using OCL? Create and Evaluate OCL Constraints View Hierarchy Models DocGen Manual

Owners
Data

log()

The **log()** black box expression takes the elements collected at any point in an OCL expression and prints them in the notification window in Magic Draw. In this example, the OCL query shown in the OCL Evaluator (discussed later) was run on the "Animals" package. This expression collects the owned elements of the package. Notice, in the comparison of the OCL Evaluation (top) window and the Notification Window (bottom), the elements displayed are the same. The red and blue arrows correspond to the same elements.

run(View/Viewpoint)

One way to use **run()** is to first collect a viewpoint and then run it on the element selected. The first viewpoint method below has an OCL expression in the <<TableExpressionColumn>> activity: **self.get('testforrun').run(self).** Dissecting the expression shows the viewpoint titled "testforrun" was collected using get(). (**NOTE: get() is a very useful expression when selecting a specific element)**. Next, the operation **run()** took the viewpoint and ran it against all elements that were not filtered out earlier. At this point in the method, these elements would be classified as "**self**." The second viewpoint method diagram shown below is "testforrun." As each element (self) is passed to the viewpoint via **run()**, their name is collected and returned to the <<TableExpressionColumn>> action.

How Do I Use OCL Expressions in a Viewpoint?

OCL expressions can be used in viewpoint methods in various ways to do various things. The following sections will outline some of these functions.

Using Collect/Filter/Sort by Expression

To more easily introduce OCL Expressions into viewpoint methods, several custom actions have been provided and can be found in the tool bar of viewpoint method diagrams along with the other collect and filter actions discussed in previous sections. A brief description of each follows below. Each action's specification window has a designated tag to enter the desired OCL expression.

Viewpoint Elements

<<**CollectByExpression**>> Collect elements using an OCL expression. This works like other collect stereotypes.

<<**FilterByExpression**>> Filter a collection using an OCL expression. This works like other filter stereotypes.

<<**SortByExpression**>> Sort a collection using an OCL expression. This works like other sort stereotypes.

<<**Constraint**>> You can add this stereotype to a comment or an action in an activity diagram to evaluate an OCL expression on the action's results. The expression should return true or false. If false, the constraint is violated, and the violation will be added to the validation results panel. A constraint comment can be anchored to multiple actions to apply to all of the actions' results separately.

<<**TableExpressionColumn**>> Apply an OCL expression to the target elements. This can be used to chain operations on elements and relatonships that would be otherwise be difficult or impossible.

<<ViewpointConstraint>> Allows a constraint to be evaluated at any point in a viewpoint method diagram on any elements passed to the action.

zz<<**CustomTable**>> A CustomTable allows columns to all be specified as OCL expressions in one viewpoint element. Target elements each have a row in the table. Title, headings, and captions are specified as with other tables.

Advanced Topics

View Currently Under Construction

Use of the Iterate Flag

What is the OCL Evaluator and Why Do I Use It?

This tool is **EXTREMELY** helpful since many times your OCL query needs to be pieced together in order to get the results you want. All of the OCL expressions discussed in the above sections can be reproduced in the the OCL Evaluator directly in Magic Draw. For example, the image below demonstrates how the expression used in section "8.2.10 value()" would operate in the OCL Evaluator tool. Notice **.ownedElement** was added to collect the elements of the package. This was not needed in the viewpoint method of 8.2.10 because the elements were already collected and filtered.

As you can see, this feature allows you to see what results your expression will return without needing to export your view to View Editor. This can save you a lot of time.

To try it for yourself, follow these steps:

1. selecting some model element(s) in a diagram or the containment tree,
2. finding the MDK menu, and
3. selecting "Run OCL Query."
4. You may need to resize the popup window to see the entry fields.
5. Enter an expression in OCL, hit Evaluate, and see the result (or error).

Note: "Self" in the below examples is setting the target of the OCL expression.

There are four parts to the OCL Evaluator:

1. Expression entry

In this area you can enter an OCL query and recall previously evaluated queries by clicking on the arrow on the right hand side of the field.

2. Results

This is where the results of the query are displayed.

3. Query completion suggestions

This field suggests potential options for the current result of your query.

4. Selection location

This allows you to select from which location "Self" or the target of the query should be taken from.

How Do I Use OCL Viewpoint Constraints?

In this example the package OCL Viewpoint Contraints contains a <<requirement>> that has two dependency relationships stereotyped <<mission:specifies>>. One of these relationships violates a set pattern for what a requirement can specify. The viewpoint method shown below contains a <<ViewpointConstraint>> action which will validate elements passed in with a specified OCL expression. This is the expression used :

r('mission:specifies').oclAsType(Dependency).target.oclIsKindOf(Relationship).validationReport

A breakdown of the expression:

- starts with selecting the mission:specifies relationship using **r()**

- then casts the values returned as dependencies using .**oclAsType(Dependency)**

- then selects the targets of the dependencies with **.target**

- then casts the returned values as relationships using **.oclIsKindOf(Relationship)**

- finally, signals return of a validation report (first two tables show below) with **.validationReport**

The validation reports are automatically generated and populated with error data. One being a summary and one detailed.

Since the requirement in the OCL Viewpoint Constraints package has a dependency with another type of relationship as the target an error will be thrown.The element which violates this constraint is then sent to table structure and populated into the last table shown below. In this case, "Requirement Name!!" is expected to be the final result.

How Do I Create OCL Rules?

How Do I Create Rules on Specific Model Elements? (Constraint Evaluator)

The diagram below contains a <<block>> named "Commented Block." Attached to this element are two constraints...

1. oclIsKindOf(Comment)

2. oclIsKindOf(Class)

How Do I Create Rules Within Viewpoints?

How Do I Validate OCL Rules in my Model?

There are two methods for validating OCL queries in a model, via a right click and via the MD Validation Window.

To validate OCL rules using the right click menu, select the package containing the elements to be validated and select MDK -> Validate Constraints

How Do I Create Expression Libraries?

How Do I Use RegEx In my Queries?

http://www.vogella.com/tutorials/JavaRegularExpressions/article.html http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

How Do I Create Transclusions with OCL Queries?

Table 6.2.

Name	**Documentation**
How Do I Create Transclusions with OCL Queries?	

Table 6.3.

Name (from constructed Transclusion)	**Documentation (from constructed Transclusion)**