MagicDraw MDK Manual

Jet Propulsion Laboratory
California Institute of Technology

Welcome to the Model Development Kit (MDK) manual. MDK is a MagicDraw Plug-In that was developed by JPL. It connects MagicDraw SysML authoring tool with JPL's Web-based model management system (EMS).

The goal of the EMS is to define a language which models can be collaboratively developed on the web. Using these features allows the user all new powers for documentation and description. While many very capable features are available to users of the EMS Applications most advanced model work must be done using a more feature-rich modeling tool. While are many modeling tools available for use the current standard for EMS is MagicDraw by NoMagic. MagicDraw can interact with the EMS environment using the Model Development Kit (MDK), a plug-in specially designed by the EMS team to interface MagicDraw with EMS Servers.

For more information about entry-level Model Based Systems Engineering Practice (MBSE) please refer to the EMS MBSE Training compendium here (want to contribute? let us know)

For more information about the current and future state of JPL MBSE tools , see the Reference Architecture Description for MBEE here

Section 1. Install MagicDraw with MDK Plug-In

The [cf:Welcome to EMS Support.vlink] page has download links to MagicDraw that is bundled with MDK plug-in and to a standalone MDK plug-in zip file. Use the MagicDraw bundle to install both MagicDraw and MDK plug-in. If you already have MagicDraw and need to install or update MDK plug-in, use the standalone MDK plug-in.

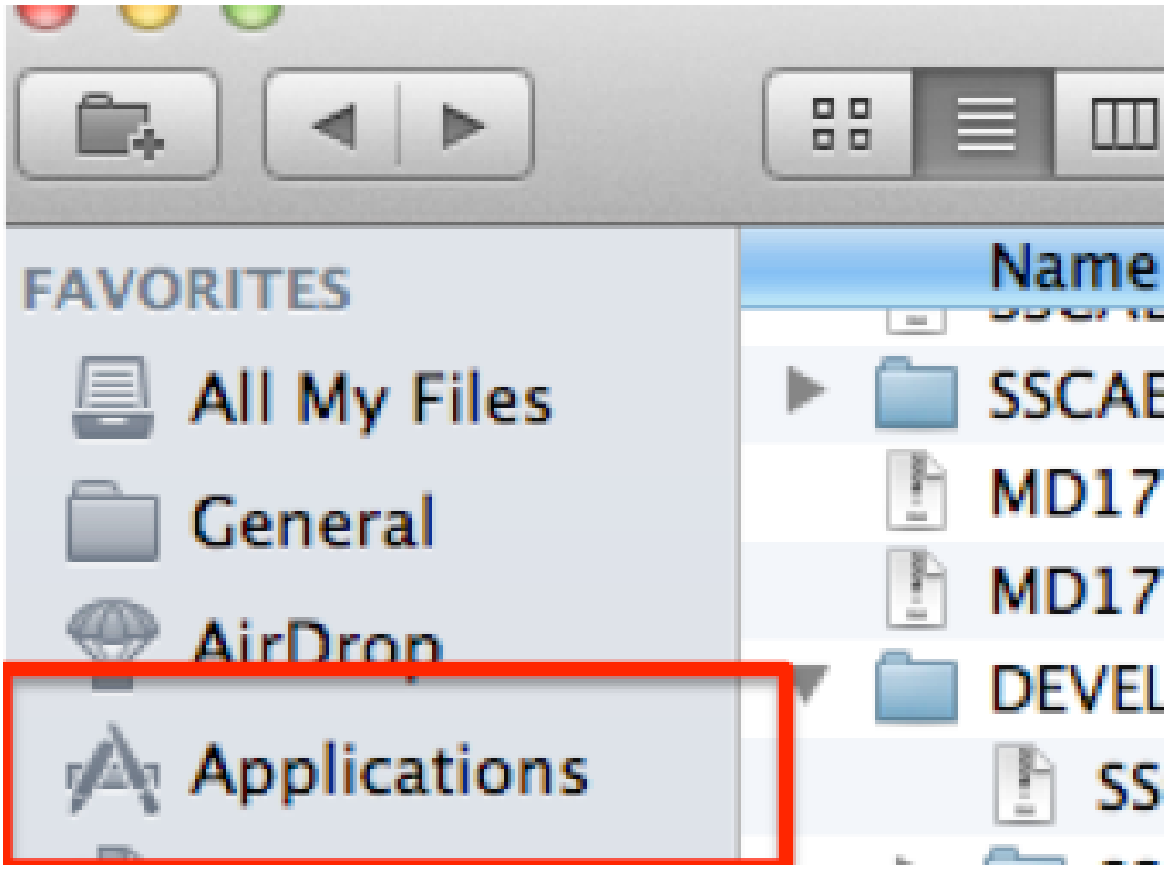
First locate the build of MagicDraw that you would like to install. Most likely you will use the community release located on the EMS Support page at "https://ems.jpl.nasa.gov". If you have not been to the EMS Support site before, once you login to Alfresco, then type "EMS Support" in the site finder. Other projects may have their own specific builds.

Note that if you are using a beta version, you should check back periodically for updates.

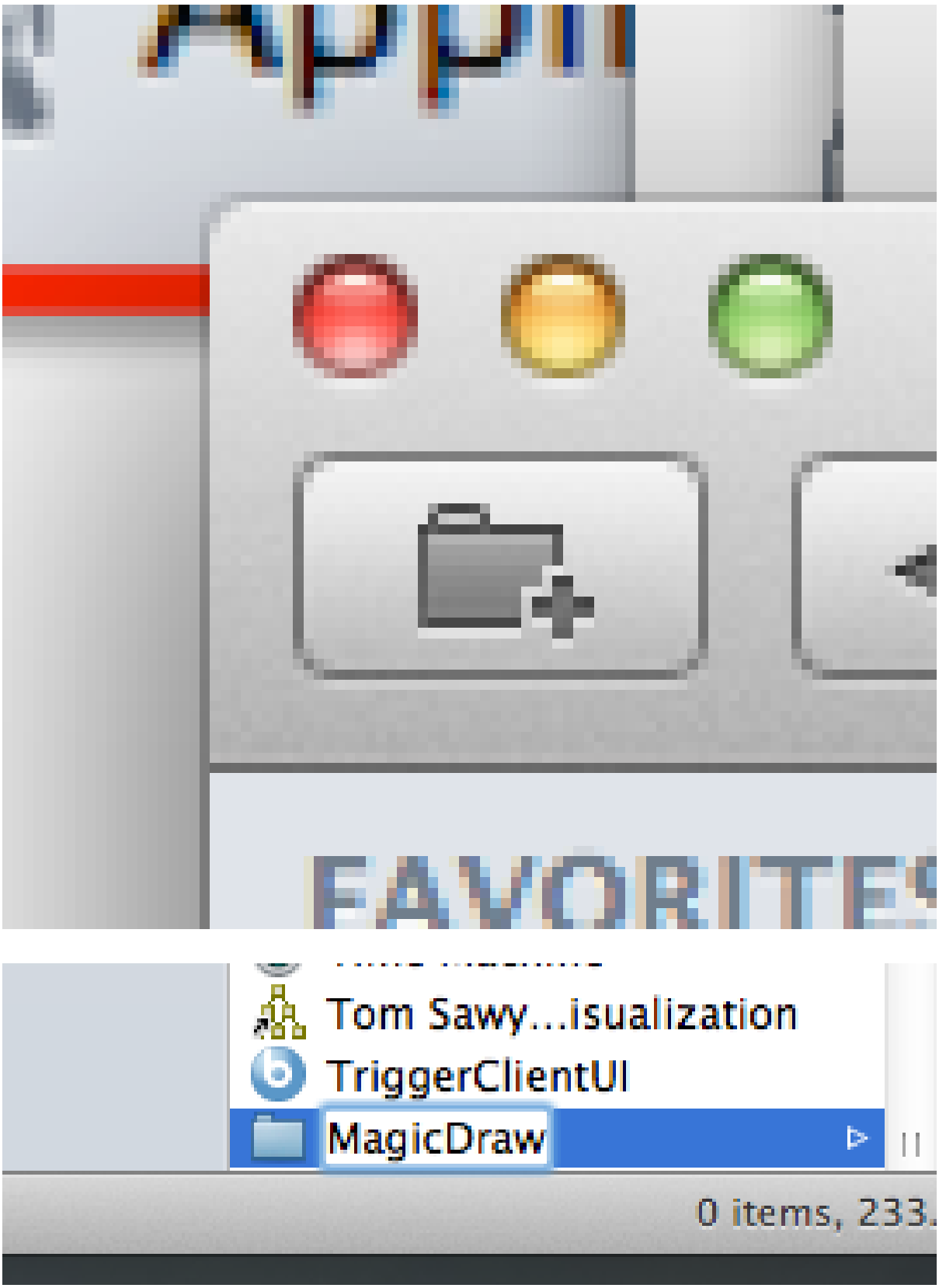
Mac Users -

Optional: This set of instructions is a recommended best practice only, location of MD install does not impact custom icon for Mac Users

1. Open the Finder and Click on Applications in the left hand side



2. Click on the new folder icon in the upper left corner of the window and name the new folder "MagicDraw". If you already have a Magic Draw installation in a "MagicDraw" folder, create a NEW folder in the Applications folder. Either rename your old MagicDraw folder or name this new folder something else (i.e. "Crushinator_2.1"). Do NOT install multiple Magic Draw installations in the same folder



1. Download Latest MD Release from the EMS Support site on "ems.jpl.nasa.gov"
2. Copy the downloaded .zip to \Applications\MagicDraw\ (or corresponding folder) and double click to extract it

3. Open the newly extracted Folder and locate "MagicDraw UML". Clicking this starts MagicDraw. Select the default option for all the pop-up boxes except for the application perspective, which we recommend to use "Full Featured - SSCAE (Current)" and "Expert". Also, never enable automatic log in.

4. It is recommended to create a shortcut in your OSX Dock. To do so, select "MagicDraw" and drag it down to your Dock.

Note that when you first open the MagicDraw installation, your Mac might say it is damaged and cannot be opened. This is an artifact of the Mac's security. Under System Preferences, open up Security and Privacy and select "Anywhere" for "Allow apps downloaded from:". Open your installation of MagicDraw. Your system will remember the developer once you have opened the program once. Remember to switch your security setting back to "Mac App Store and identified developers" after this initial opening!

Usually the icon is preinstalled, but if you get the generic MagicDraw icon (blue box with white x), you can add the icon manually.

1. Download IconPack from EMS-Support Document Library

2. Unzip Pack

3. Navigate to MD Install Directory

4. Right Click on MagicDraw UML select "Get Info..."

5. From the Icon Pack open (Version Name)_MD_Icon.png in Preview

6. Hit Command + A to select all and Command + C to copy

7. Return to the Get Info.. pane and click the icon in the upper left corner (It should highlight blue when selected)

8. Hit Command + V to paste the new icon

You might have to reattach MagicDraw to the dock to get the full effect.

Windows Users -

1. Download Latest MD Release from the EMS Support site on "ems.jpl.nasa.gov"

2. NOTE : If the file name is too long it will cause problems in Windows. Shorten the file name to remedy this problem but be certain to maintain the build version information contained in the file name.

3. Extract MagicDraw to the location of your choice (Recommended C:\MagicDraw\). Do NOT install multiple Magic Draw installations in the same folder. If you already have a Magic Draw installation in a "MagicDraw" folder, create a NEW folder in the Applications folder. Either rename your old MagicDraw folder or name this new folder something else (i.e. "Crushinator_2.1").

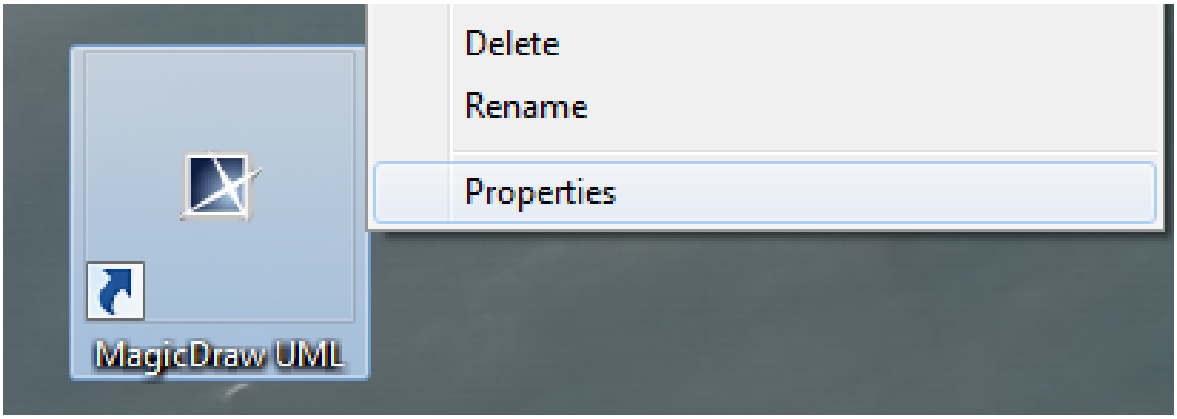
4. Open the bin folder in the extracted Magic Draw installation folder.

5. Click on magicdraw.exe to open Magic Draw.

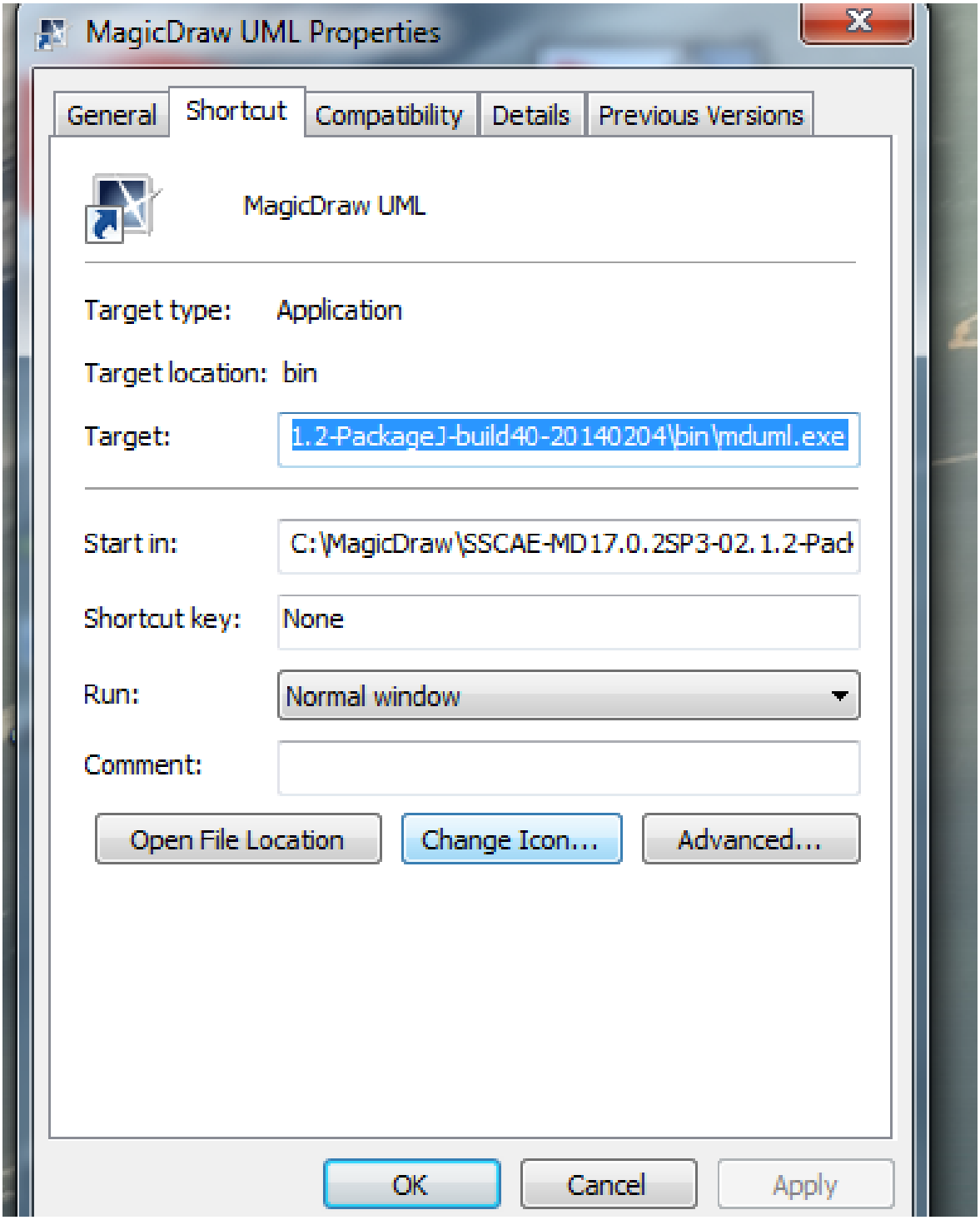
6. It is recommended to create a shortcut (either on your Desktop or your task bar). To do this, right click on magicdraw.exe and select Send To...->Desktop (create shortcut).

7. Navigate to your desktop and right click on on the Shortcut and click Rename. Enter "MagicDraw UML" and then click enter.

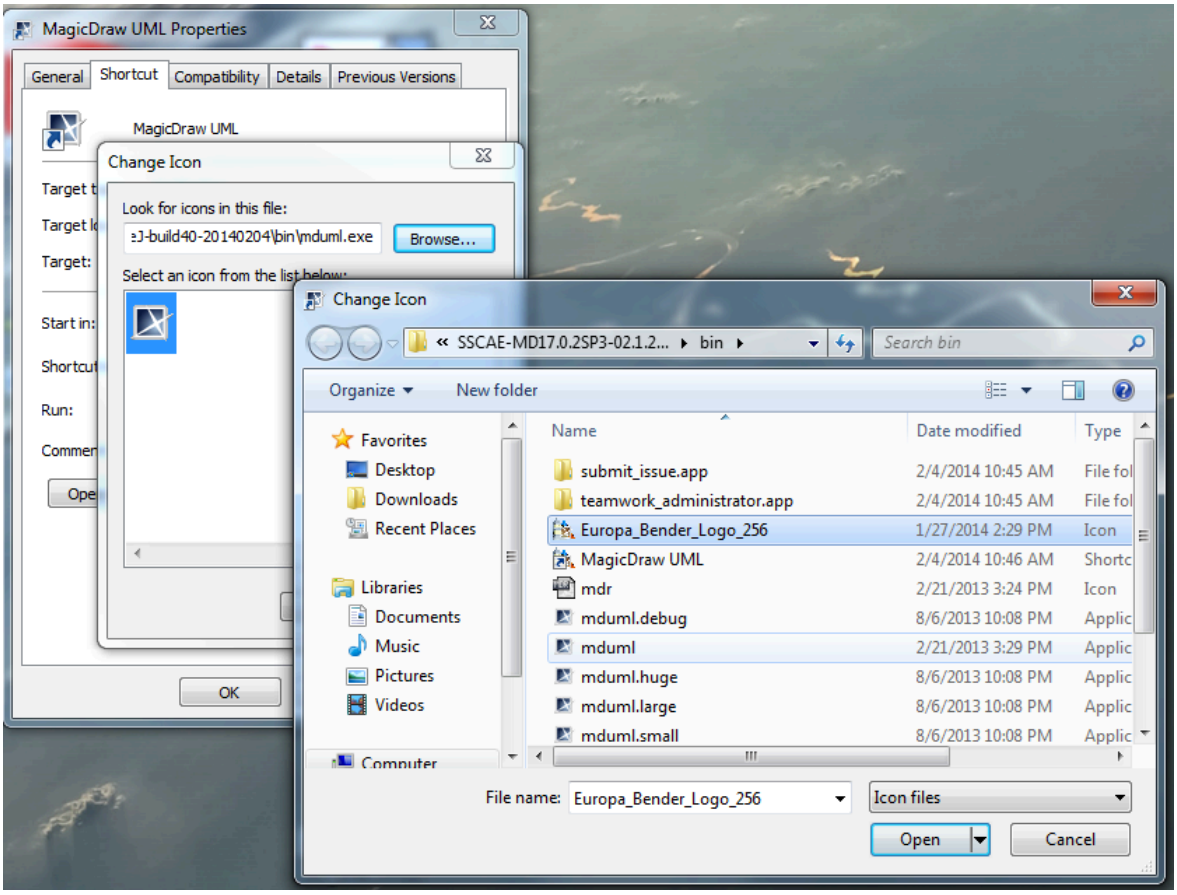
8. Now right click on "MagicDraw UML" and select Properties



9. In the Properties window click "Change Icon..."



10.The Change Icon dialog will open, select Browse... and Navigate to the installation's bin folder. Select the \Europa_%ReleaseName%_Logo_256.ico and click Open.



11.Hit Apply and OK

12.Now either leave the shortcut on your Desktop or drag it to your taskbar.

13.Double-click on the shortcut to open MagicDraw

How to install updates

Return to the site where you found the build originally. In most cases this would be the EMS Support page. If there is an update after the build you have installed, follow the instructions below.

1. Download the update as a zip file, but do not unzip the file. (Note: the file location you download to does not matter; however it is a good idea to keep track of which update(s) you have installed.) Generally you do not have to re-download the full installation of Magic Draw, but you can if you wish.

2. Open MagicDraw to the main blue screen.

3. Under the Help menu, click Resource/Plugin Manager.

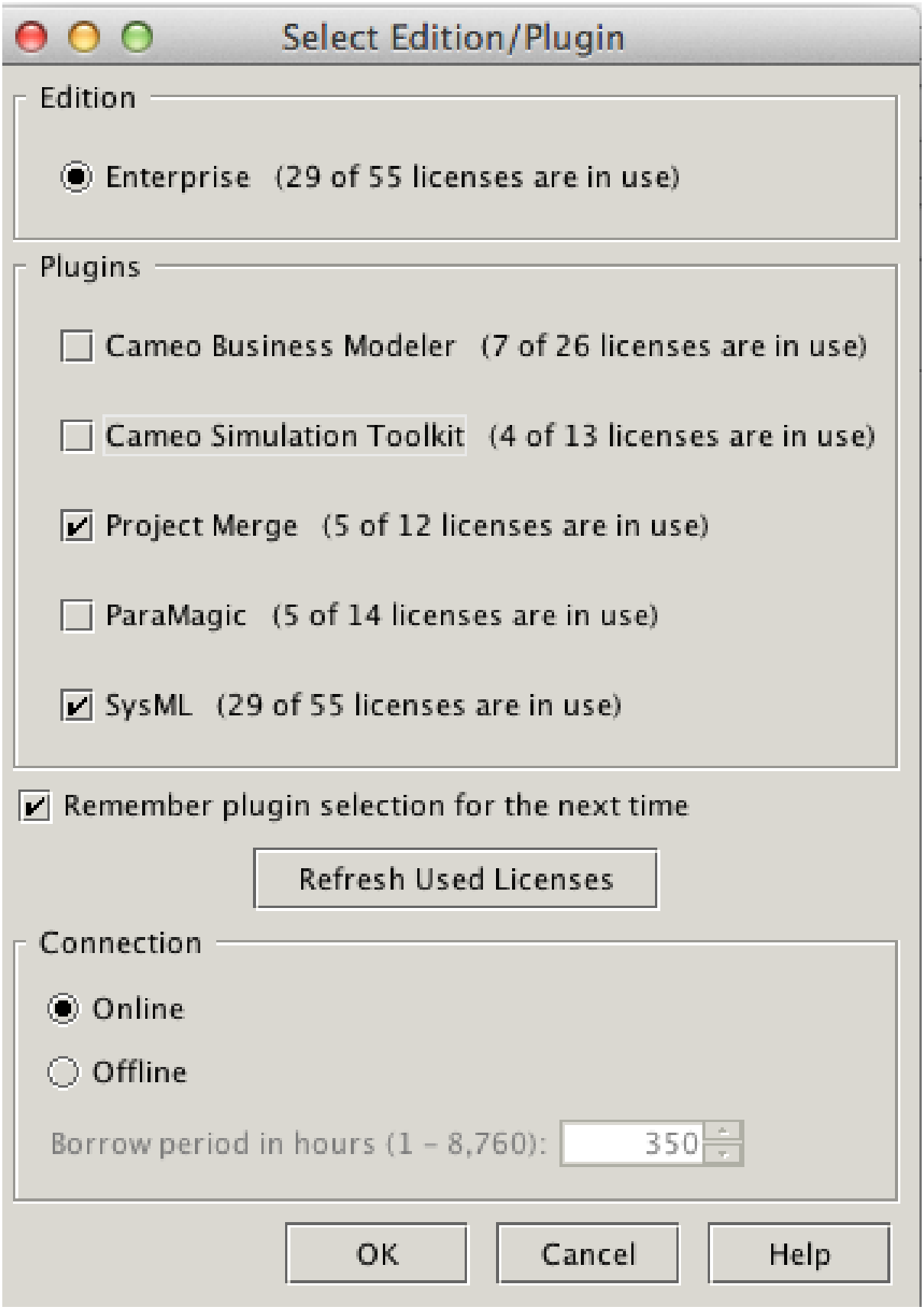
4. Click the Import button and select the zip file that you just downloaded.

5. Once the import is completed, close and restart MagicDraw.

Section 2. Create an EMS Project in MagicDraw

NOTE: Users who also use the Institutional Model Centric Engineering (IMCE) Profiles, installation of MDK no longer automatically mounts the IMCE profiles as part of the installation (please refer to IMCE documentation for more information). However the Project Usage Integrity Checker is included by default in all models created using SSCAE released MD products.

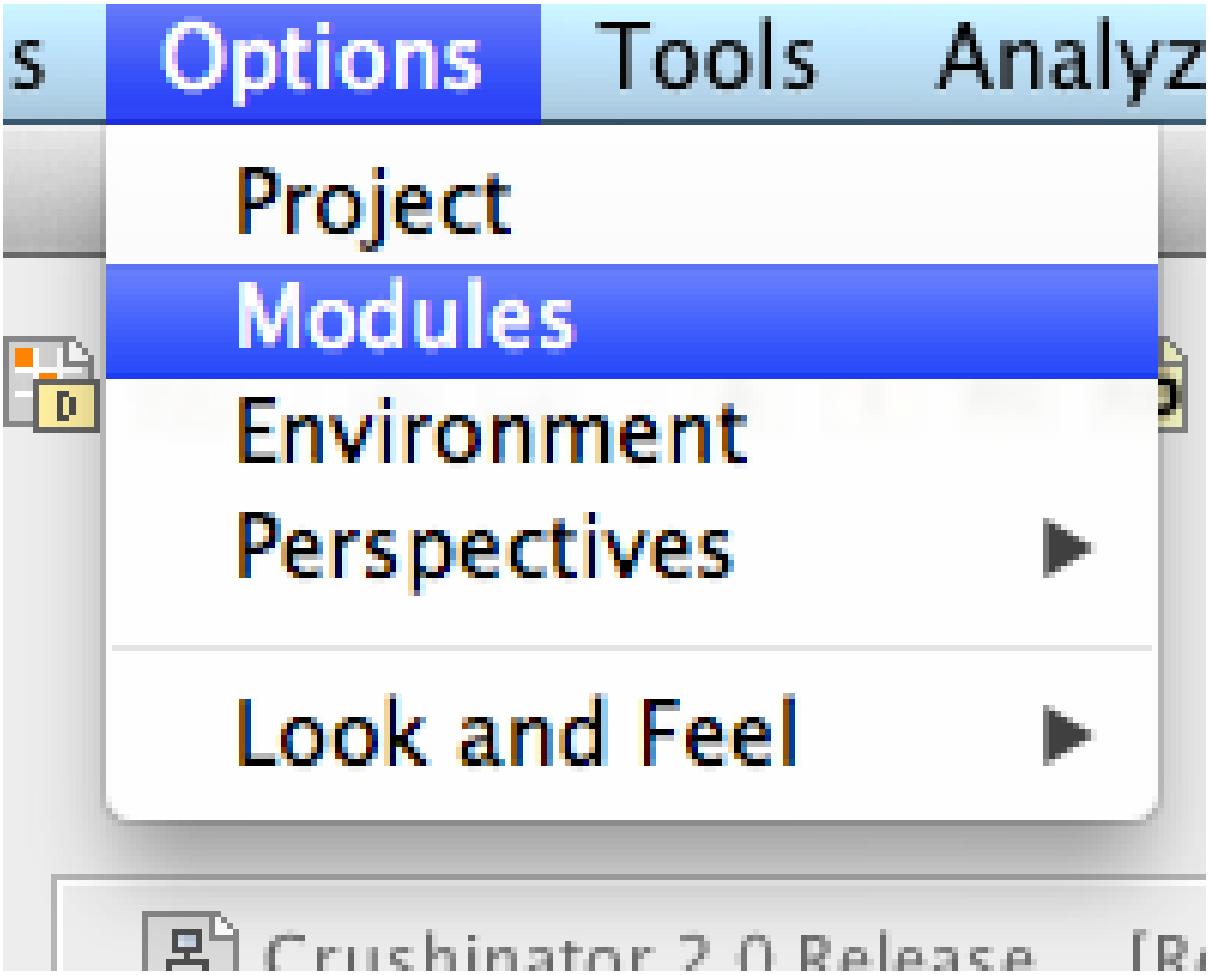
Open MagicDraw selecting the default options, note that if you plan to use EMS's branching and merging capability (check your project modeling policies if you are unsure) you must enable the 'Project Merge' tool at start-up to be able to merge an EMS Project using MagicDraw (Figure MDK 2-1).



[Circular reference!] : Magic Draw plugin selection dialog on startup, users should select Project Merge and SysML at minimum to take full advantage of EMS

To create a new model click File->New Project... A dialog will open, select "Systems Enigneering->SysML Project" and enter your save location and file name. Click 'Ok'. Once your new project has opened you must first add the "SysML Extensions" profile to make it an EMS Project. The first step is to open a new (or existing model) and load the profile. This will allow you to access the diagrams and stereotypes needed for MDK to function properly.

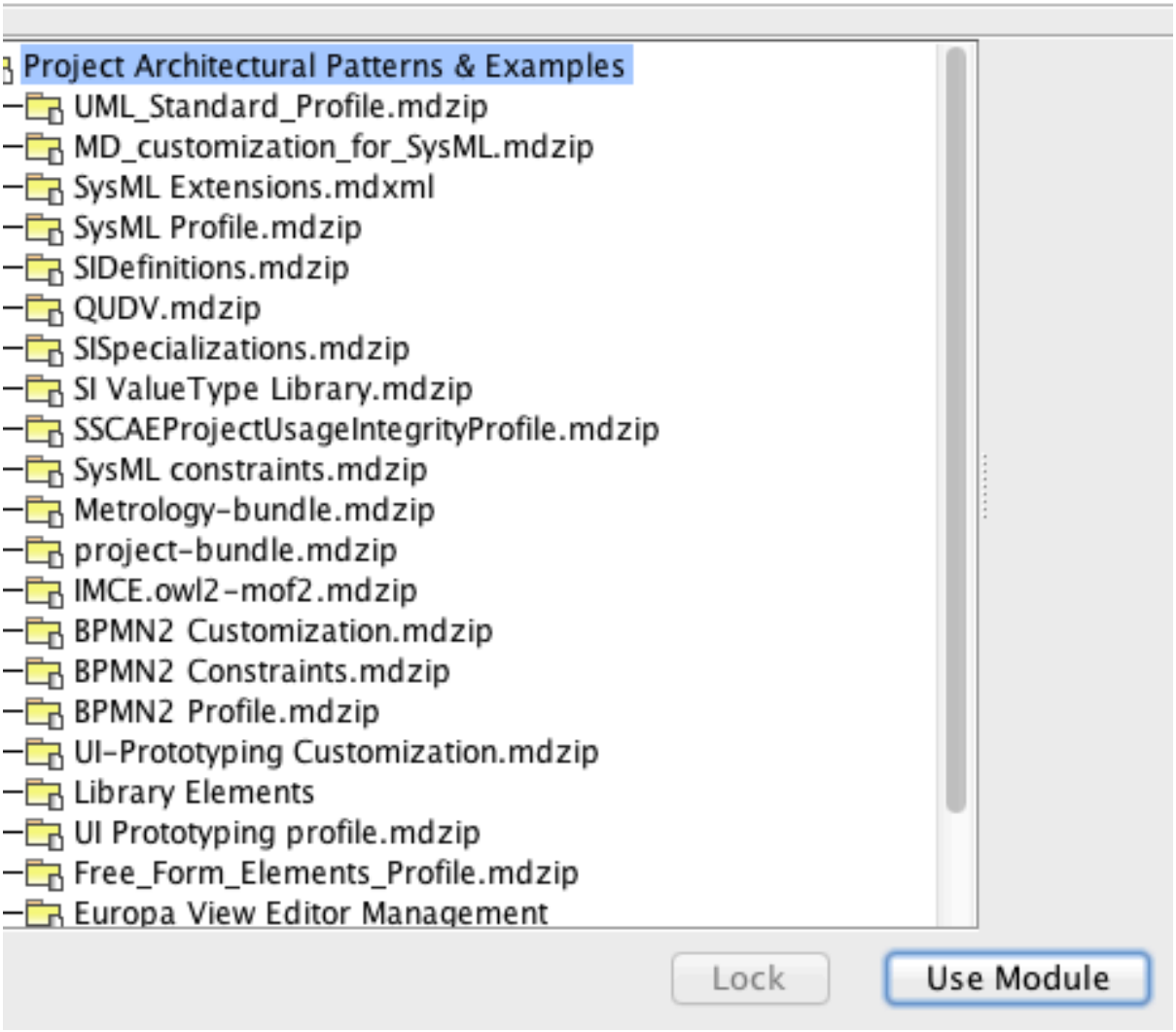
Add the profile to your model by clicking Options> Modules in the upper menu bar,



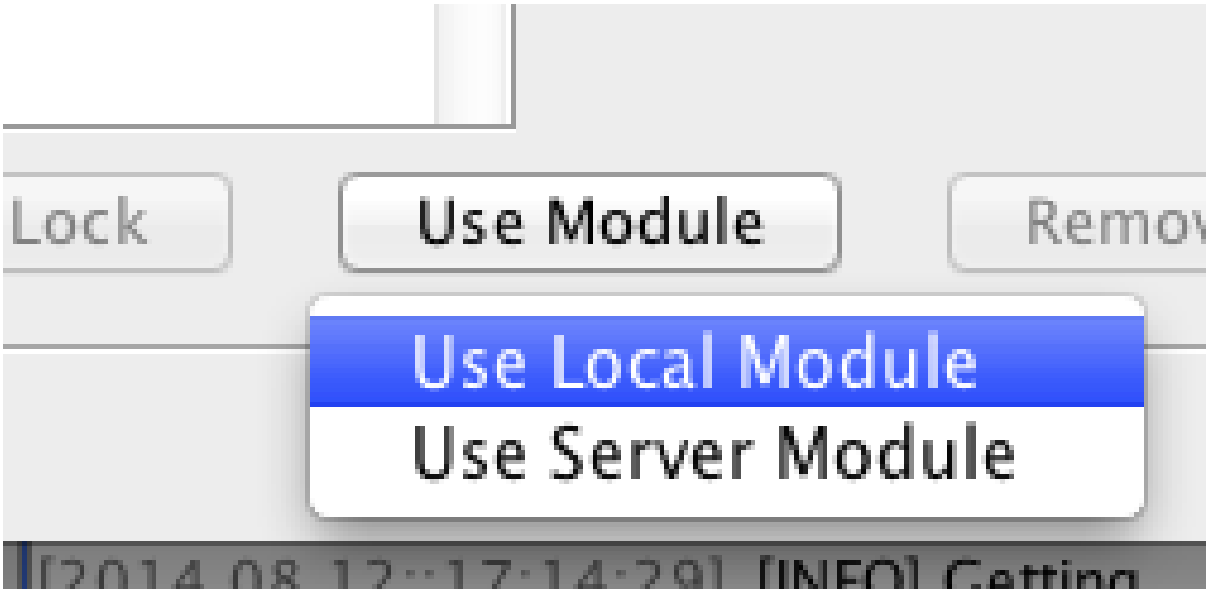
When the dialog opens look for "Use Module"

pecify module usage options

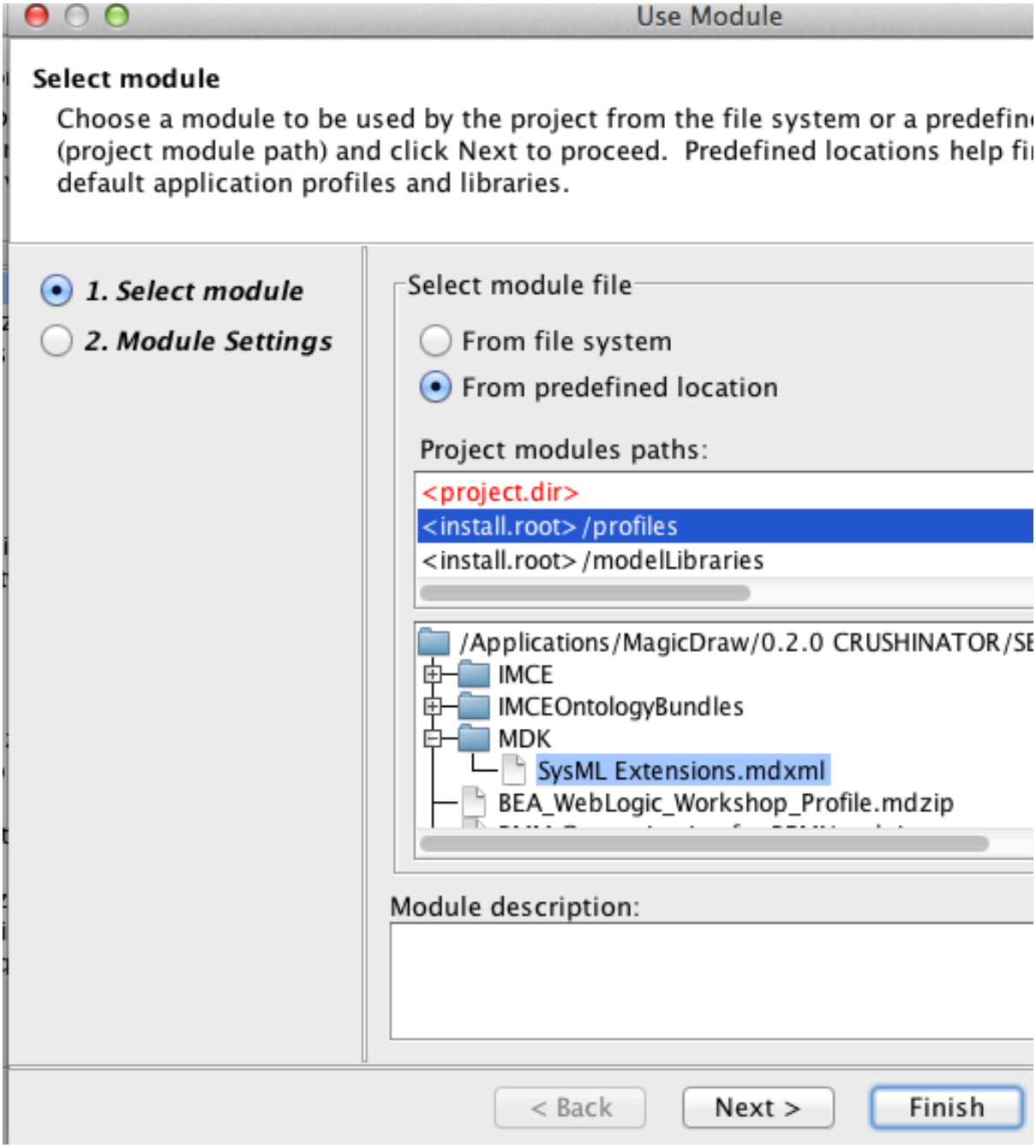
Modules allow the separation of projects into parts to allow the reuse of p
options (accessibility, load mode, index usage, and shared package mour
Remove buttons respectively.



Click it and select "Use Local Module"



When the new dialog box opens, you will need to navigate to the predefined location for the MDK profile. The profile is located in <install.root>/profiles/MDK/SysML Extensions.mdxml. Once this file is located click 'Finish'.



Your model will be configured to use both the MDK and IMCE profiles over the next 20-30 seconds. Once it is done, save your model.

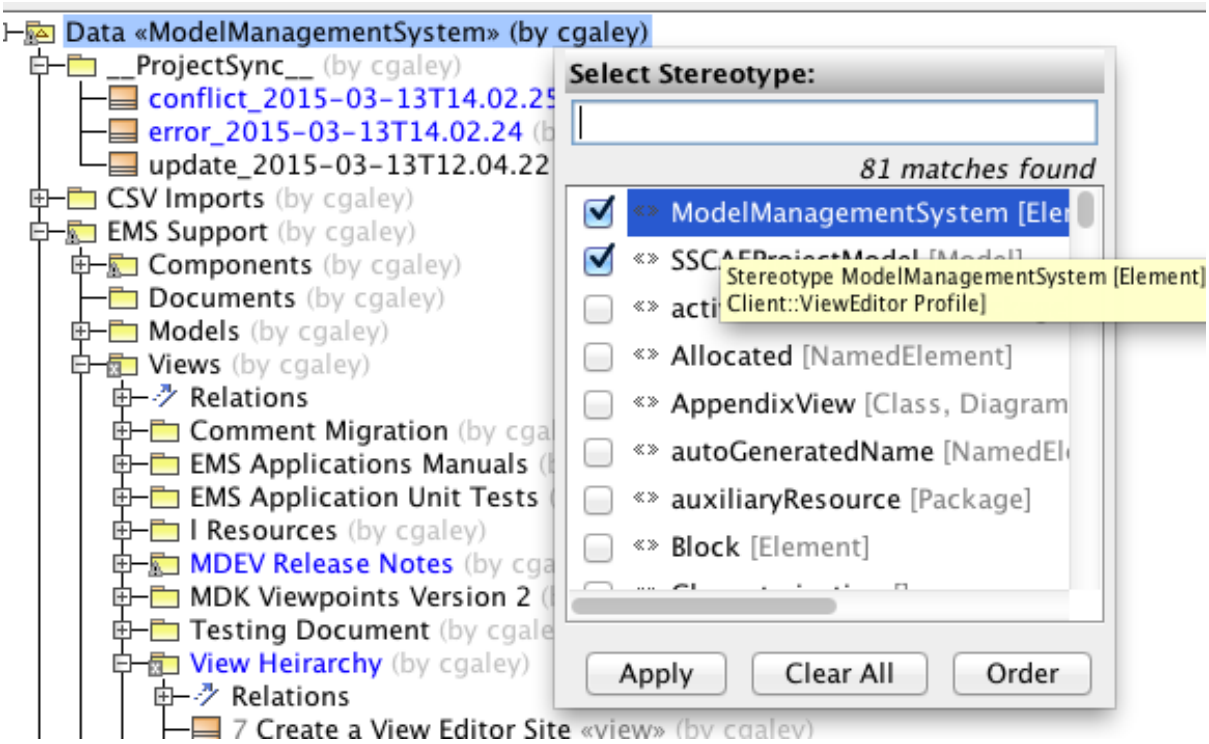
To save your model, check the menu bar to see if the IMCE Project Usage Integrity Checker (PUIC) is properly configured. You can identify this by a RED X or GREEN Check-Mark. If you have a GREEN Check-mark no further action is required and you may save your model normally.

If you have a RED X you must configure the PUIC. First, click the red 'X', this will open a validation window. For each item that appears right-click and perform the fix suggested. There will be several items and once complete you should see a green check-mark. If you are complete but still see a red X try clicking it again, this should update the status or bring additional validation errors to your attention.

For more information on collaboration and adding your model to Teamwork contact the document POC.

2.1. Configure EMS Project in MagicDraw for EMS Server

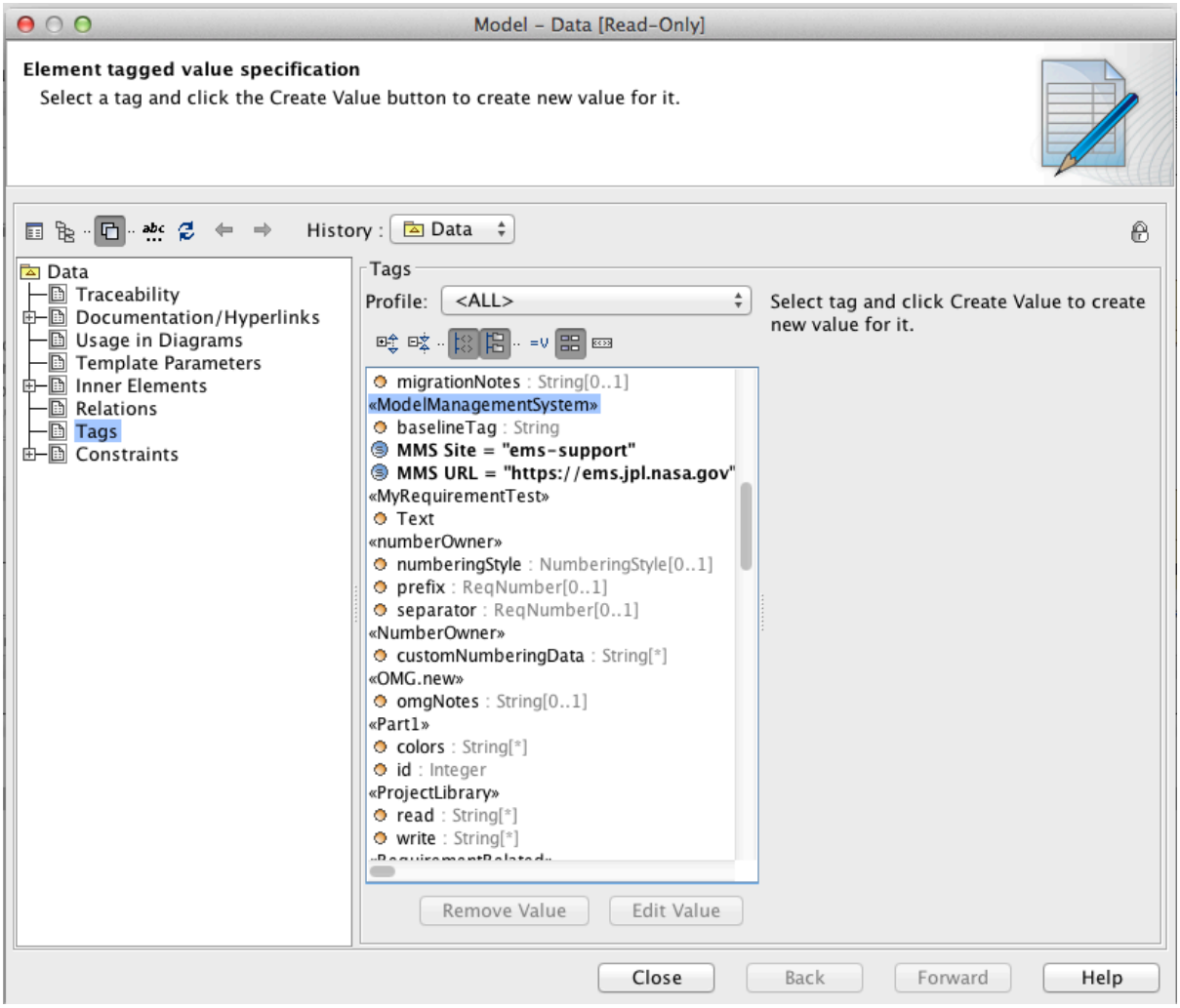
Before configuring the EMS Project the Project must be created in MagicDraw and a site created on the EMS Server (link). The EMS Project can be configured for an EMS Server by selecting the "Data" model at the top of the containment tree and stereotyping it with <<ModelManagementSystem>>. If the EMS Project model was loaded from a MagicDraw TeamWork server, you may need to lock the element for edit by using a right click menu "Lock>Lock Element". To apply the stereotype, use a right click menu "Stereotype". In the dialog that opens, put a check next to "ModelManagementSystem" and then click "Apply". Note you can start typing the name of the stereotype in the box above the dialog to filter available stereotypes.



Now you can link the model to the desired EMS server. Open the specification dialog of "Data" by double-clicking it in the containment tree. Select "Tags" node in the left tree and scroll down to find tags of <<ModelManagementSystem>> stereotype. There are two tags that must be defined.

- MMS URL: the URL location of the EMS Server (e.g., https://ems.jpl.nasa.gov)
- MMS Site: the name of a site in the EMS Server. Sites are the mechanism EMS server (in fact the underlying Alfresco server) uses to organize related model elements. There is an one-to-one relationship between a MagicDraw model (or project) and a EMS site. See the Alfresco Dashboard Manual for how to create a site.

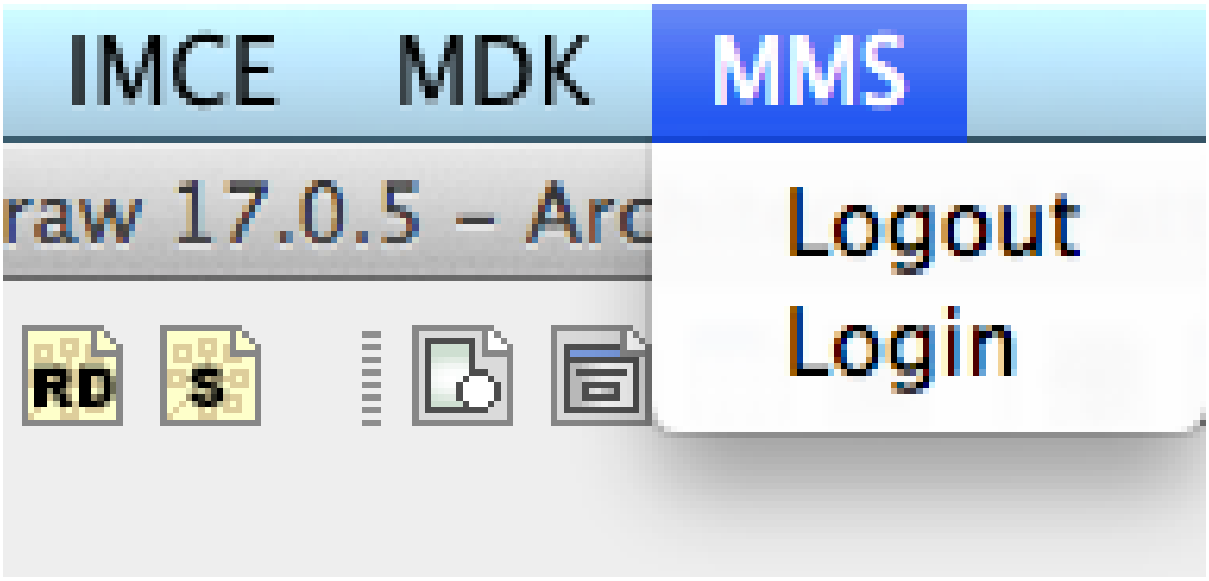
Note that baseline tag will not affect the upload; either leave it blank or fill in with "baseline".



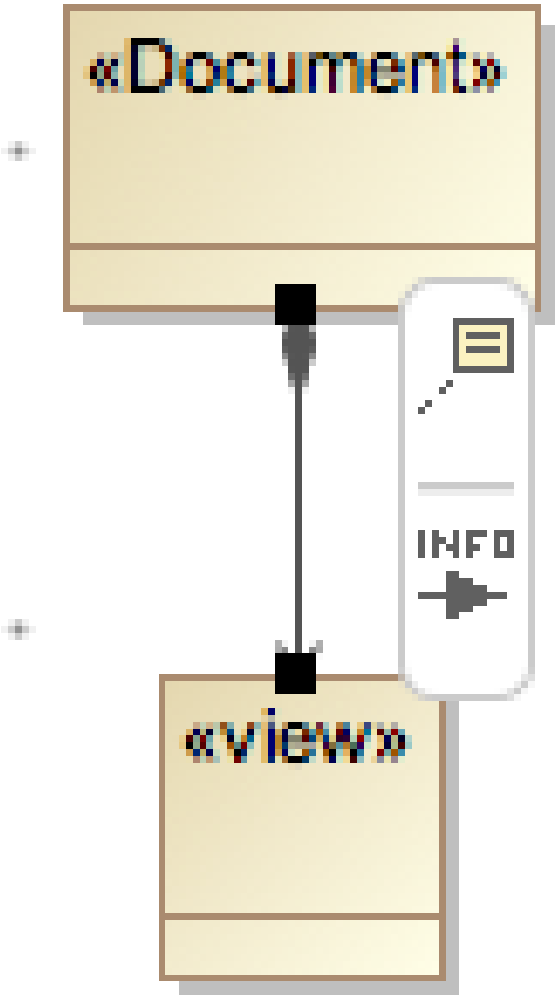
The URL should match the EMS server (JPL community users should use " https://ems.jpl.nasa.gov"), individual tasks may have their own servers (ie Research Network uses https://rn-ems.jpl.nasa.gov). The site is the "url-version" name of the site that was created on the EMS server (i.e. the version in the url of the Alfresco page). For example, this site for this document is https://ems.jpl.nasa.gov/alfresco/mmsapp/mms.html#/workspaces/master/sites/ems-support . As you can see in the image, the corresponding entry for "MMS Site" is just "ems-support".

2.2. Initialize EMS Project on EMS Server

Once the EMS Project in MagicDraw has been configured for the EMS server, it must be initialized on the server. In order to initialize the model, first the user will need to log in to the Model Management System (MMS) which is the EMS Server's service for exchanging information between tools. To activate the MMS Service click MMS->Login in the MagicDraw Menu.

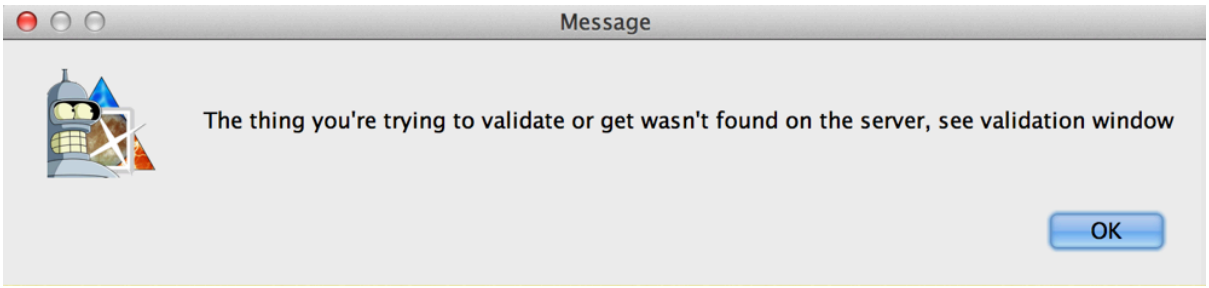


Clicking Login will initialize an MMS connection to the EMS Server, it will then ask you to authenticate your link using your institutional username and password:

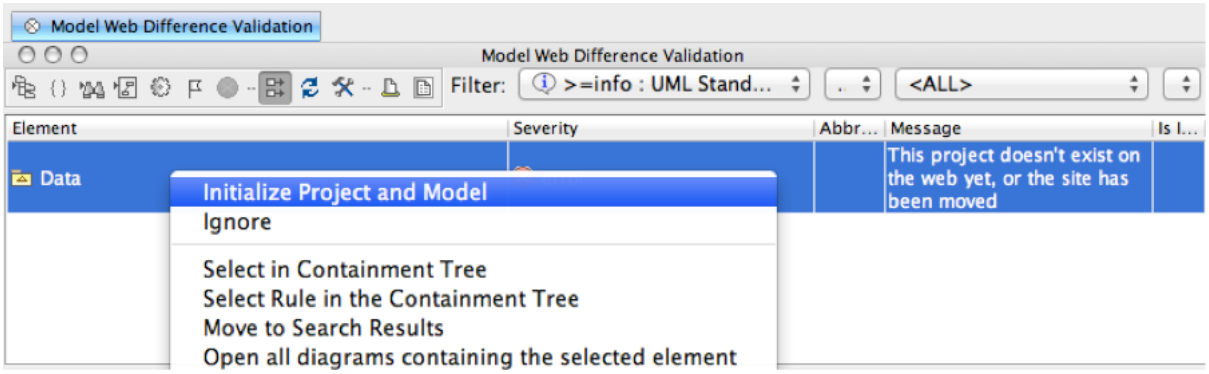


Now the Project can be initialized, right click on 'Data' and select 'MMS -> Validate Model':

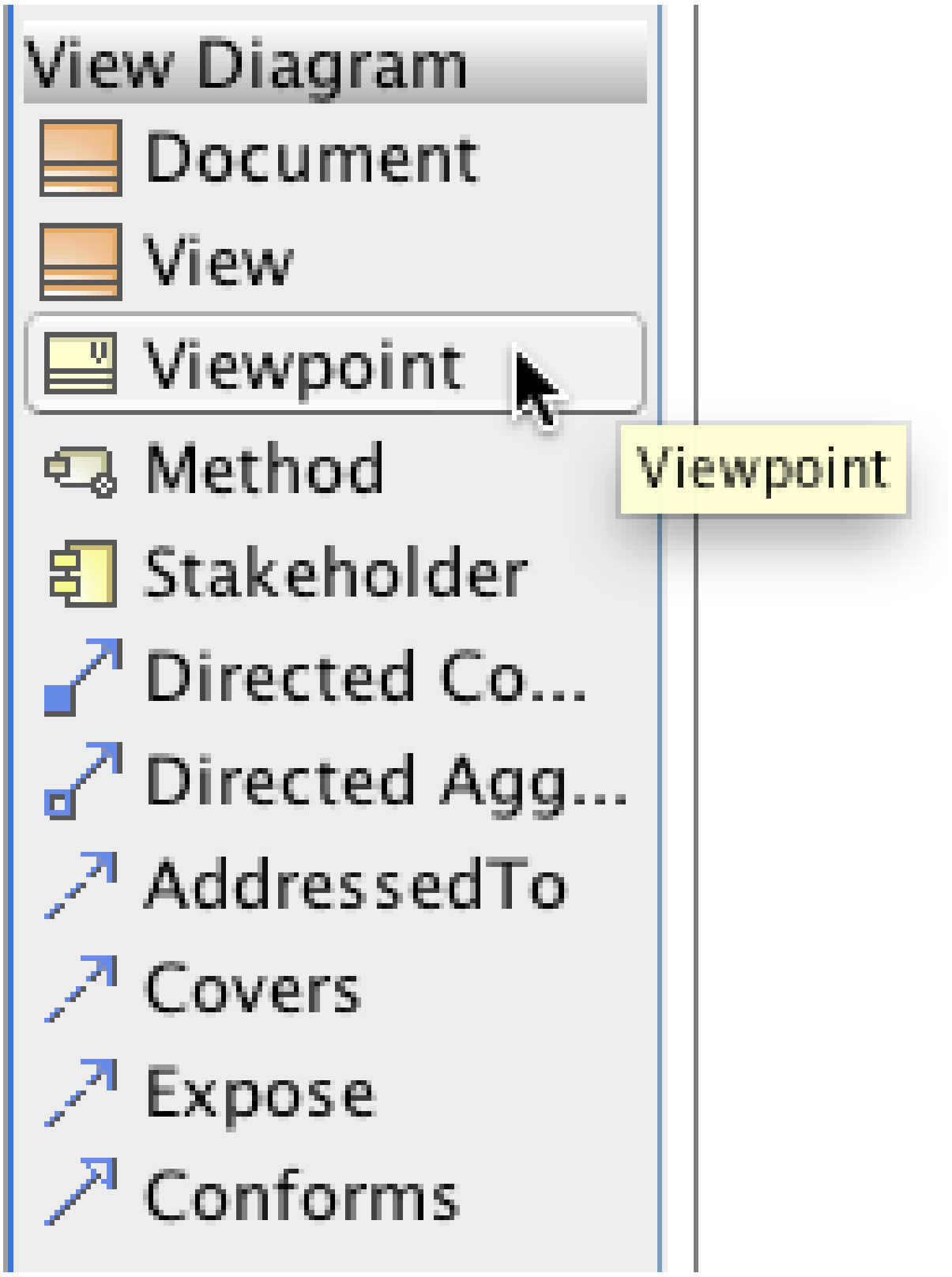
If the project is not already on the EMS Server the validation report should return with "The thing you're trying to validate or get wasn't found on the server, see validation window". (The Icon may vary depending on your version of MDK) :



Click 'OK' and check the MagicDrawvalidation window:



To send the model to the server, right click on the rule violation in the validation screen and select "Initialize Project and Model". The system will offer the option to upload the items in the background or foreground depending on the number of elements in the model users creating new EMS Projects can safely click no. However if the user is upgrading an existing MagicDraw model to an EMS Project they are strongly recommended to use background export:



A quick rule of thumb is that 1500 model elements will take approximately 1 min. Note that once uploaded it may take an additional 30 min or more for the server to index all the newly updated elements.

At this point no further action should be taken until you receive a notice that the export is complete. Users creating new EMS Projects should receive notice within 5 minutes.

After receiving the notice your EMS Project is now fully configured and ready for use. Now all collaborators can interact with your EMS Project via web or MagicDraw. In the following Views we discuss how to modify your EMS Project in MagicDraw. If you are interested in how to modify your EMS Project on the web see the EMS Applications Manual

Section 3. Modify an EMS Project in MagicDraw

3.1. Create Models

Creation and semantics of model creation are outside the scope of this manual, however there were several models that were developed in the course of documentation that might serve as useful examples and are shown below. For more detailed modelling techniques and methods please consult your Project's Modeling Policy, the MBSE Training site, or the JPL MBSE Community of Practice .

3.2. Create Documents and Views

This View assumes the user has an EMS Project in MagicDraw and is ready to start producing documents from it. A SysML model is not required to produce a document. A standalone document hierarchy (e.g., outline) is completely possible and could later be expanded to include Model Based Systems Engineering (MBSE) model content including SysML.

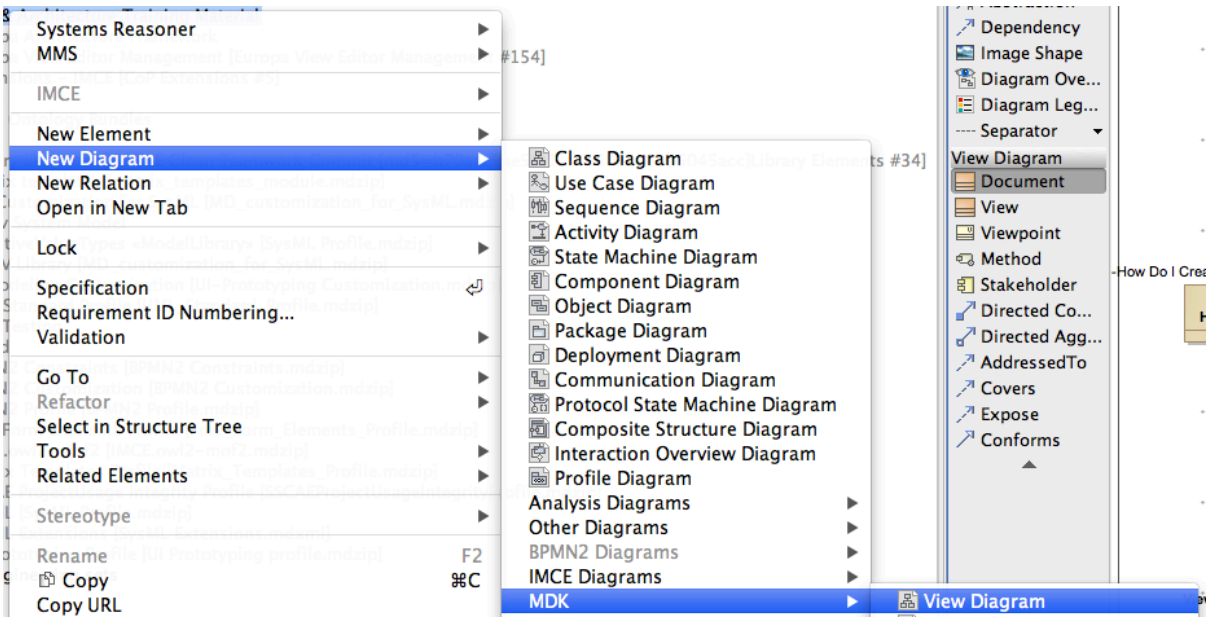
Documents are a key part of systems engineering. One of the defining moments for widespread adoption of SysML at JPL was when the community created "DocGen", a precursor of MDK plug-in, to enable MBSE practitioners to produce documents from their models. In order to create these documents, the view and viewpoint pattern was introduced. A pattern is a set of rules that govern model construction to give standardization and consistency across models. How you create documents is one such pattern adopted by the Object Management Group (OMG, the standards body behind SysML) and was incorporated into SysML 1.4.

In this pattern, a document is outlined by a hierarchy where each "View" element (based on a Class in UML) is a document section. At the root is a "Document" element, which contains all the information about the document including authors and other formatting metadata. Manipulation and input of this metadata is discussed here. Check your Project Modeling Policy before making any changes to the document format.

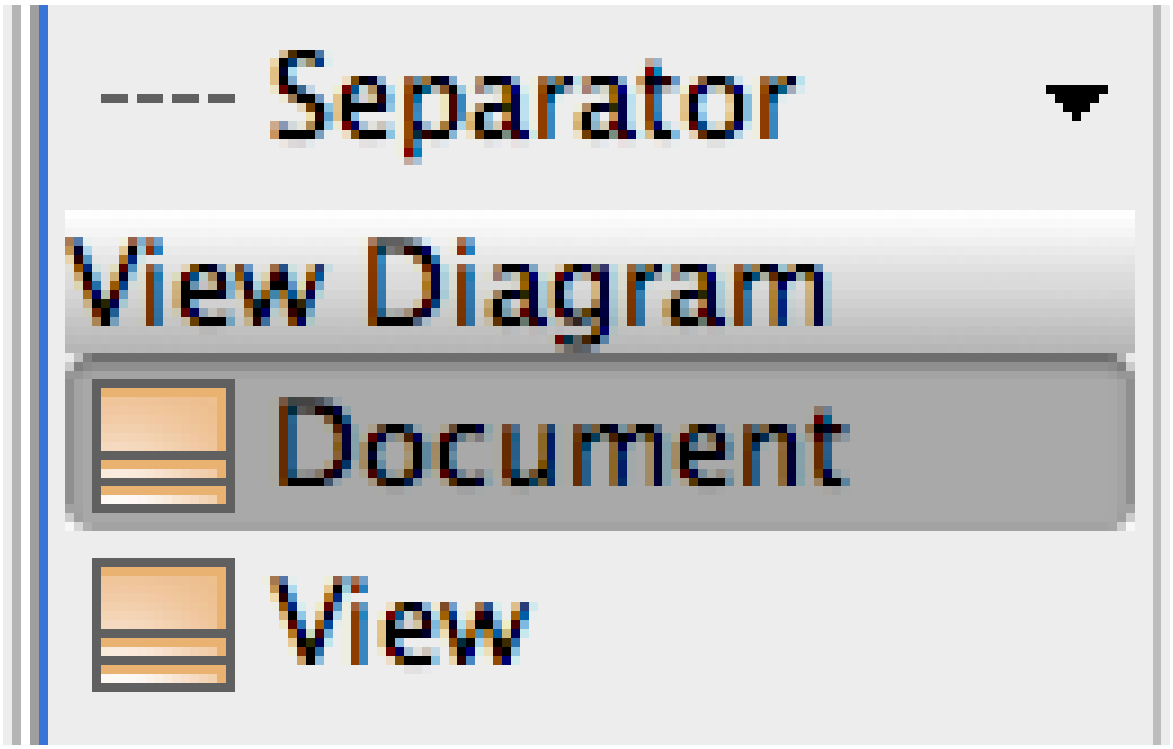
Once created, this document hierarchy can be manipulated by MDK to produce professional documents equivalent to word-processor created documents used in standard JPL practice. Unlike these current documentation tools, MDK creates documents that can be updated in real time with automatically configured templates, content, and styling. Use the DocGen Stylesheet

3.2.1. Create a Document

Creating a View Diagram is the first step to building a document or view model. This new diagram type provided by MDK simplifies access to all the tools required for making document models. To create a view diagram right click on the package in MagicDraw's containment tree where you would like the document model to reside and select 'New Diagram -> MDK -> View Diagram'.



The root of the document model is a Class stereotyped by <<Document>>. To insert this element click "Document" in the diagram toolbar, then click in the diagram work area.

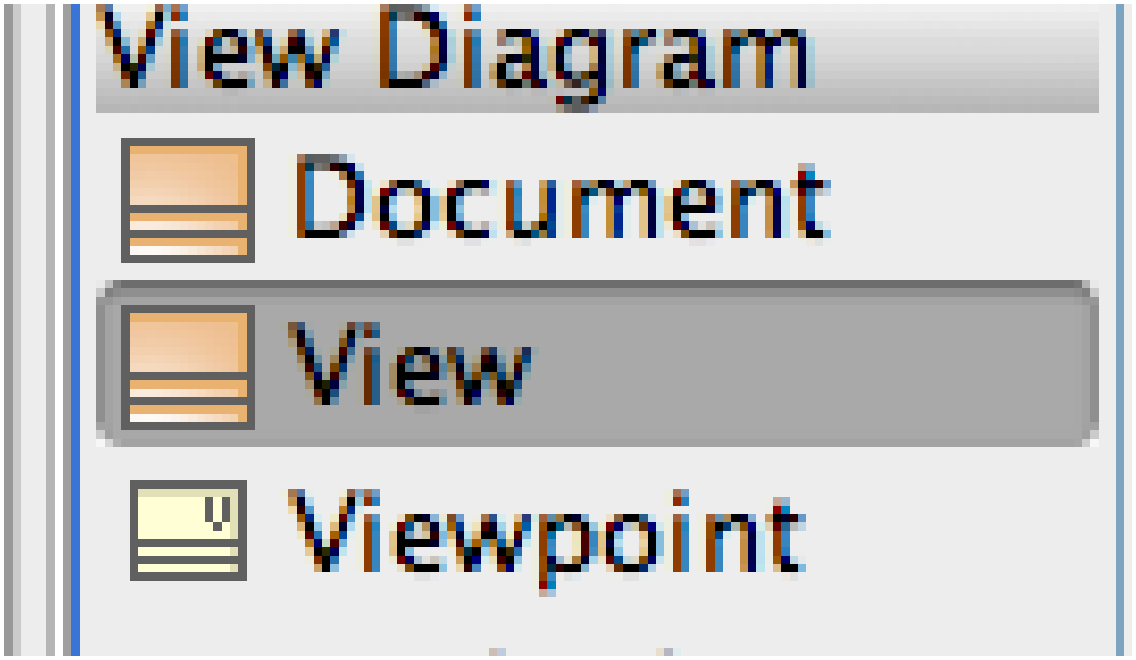


Your document title is defined by the name of the document element. Additional details about the document can be specified in the document element's Specification window. The full range of these options is discussed in How Do I Use the Docgen Stylesheet?. Remember to check your Project's Modeling Policy for specific project permissions and guidelines. As a good practice the "View Diagram" should also be titled the same as the document it describes.

The main content portion of the document is made up of "views". In formal modelling terms there is a difference between a "view" (lowercase) and a "View" (uppercase). A "View" is a complete description of a part of your model from a specific perspective. For the purposes of document creation a "view" is a section of your document. A document could be composed entirely of views defined here in the "View Diagram" or it could be composed of Views created elsewhere in your model to form a more detailed description. For more information about Views,

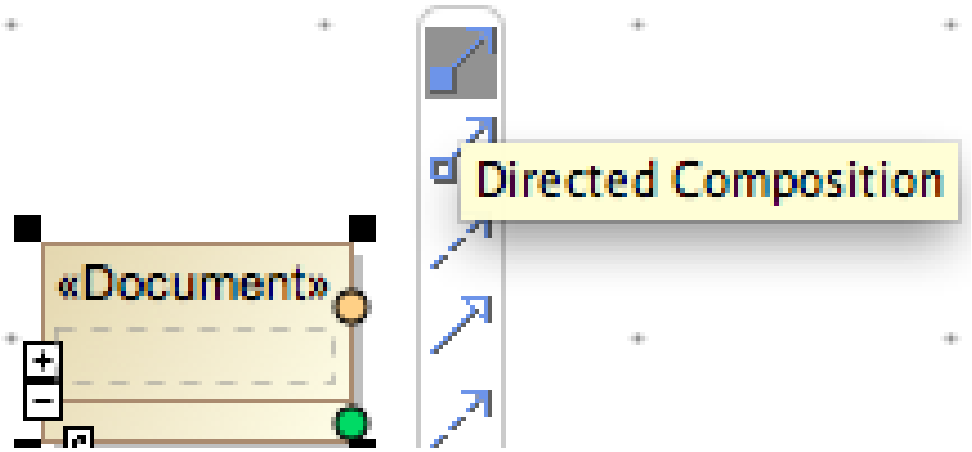
consult the System Architecture Framework. The content and layout of a view can be defined by a Viewpoint. For more information see the section on Viewpoints here.

To create a view click on the "View" button below "Document" on the diagram toolbar, then click again in the diagram work area.

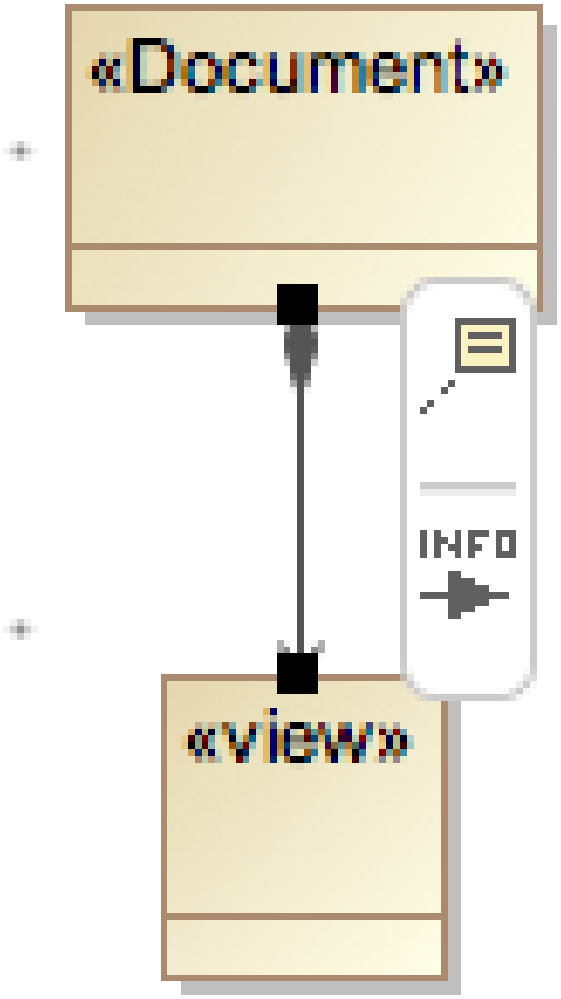


A view is connected to the document using either directed composition association or aggregation association. A view connected to a document directly utilizes a directed composition (black-diamond) association. While a view or View found elsewhere in the model is a reference and is connected via an aggregation (white-diamond) association.

To add either relationship click on the Document in the 'View Diagram' and select the desired association from the tool-tip:



Then drag the relationship down to the v/View and click. When creating these relationships remember to always start from the higher level element and finish at the lower level. When properly constructed your document should look something like this:



Congratulations! You just created your first document model! Continue on to the next part for how to publish and add more content to your document.

3.2.2. Publish a Document

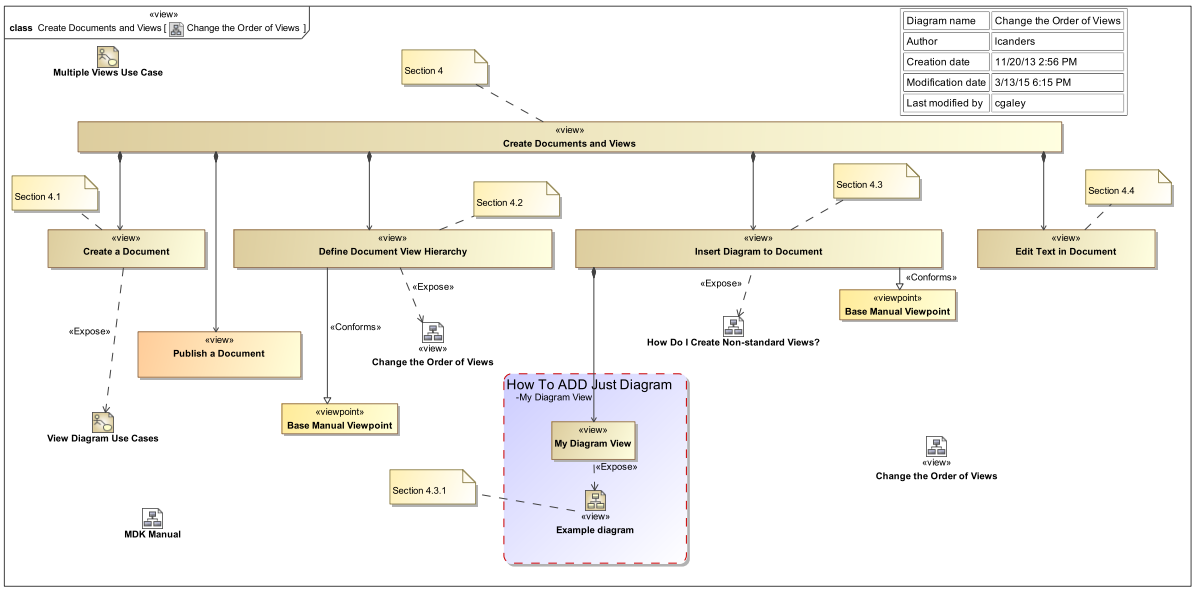
Once you have created the document model, many users find it helpful to publish it to the EMS Server to view how it will look in a published form. The MDK plug-in has multiple ways to publish your content online which are discussed in the [cf:How Do I Publish my Documents to View Editor?.vlink] View below. If users are interested in creating content in an offline mode see [cf:Create offline content using DocGen.vlink]

After following the instructions in those sections, you should be able to view your document, which at this point will mainly consist of a title and some sections. As you follow the instructions in later sections for organizing the document and adding content, it is helpful to troubleshoot by iteratively publishing the document and confirming that the results are what you expected. Project Modeling Policy's may defer, so remember to check yours before making any edits to the MagicDraw model or View Editor.

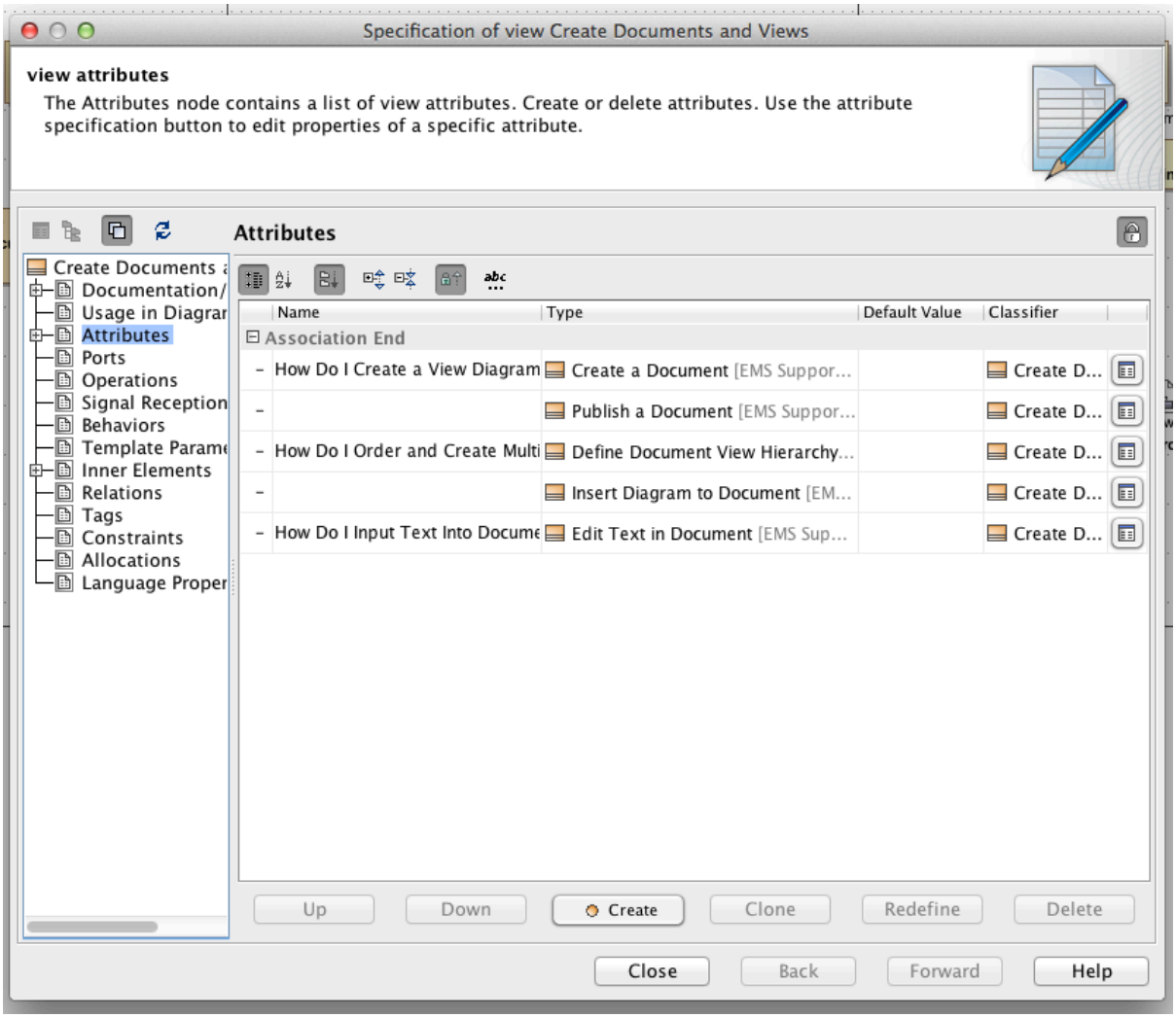
3.2.3. Define Document View Hierarchy

In a previous section we discussed creation of document sections. In this section we cover how to add additional sections and organize them. For example, the 'View Diagram' for the Create Documents and Views section of this manual is shown below. This section of the document has multiple sub-views connected to it.

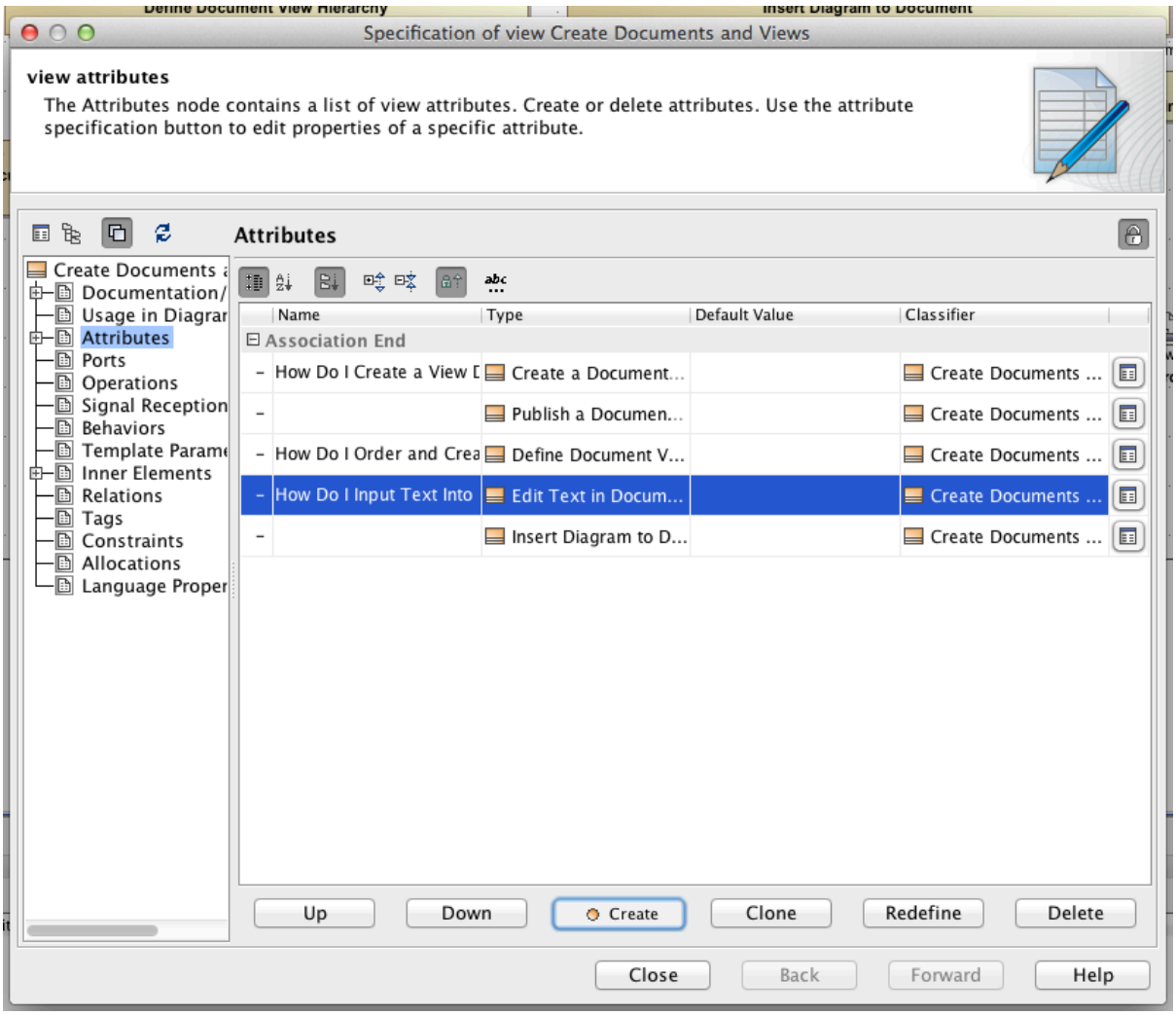
Figure 3.1. Change the Order of Views



While they may appear in a certain order in the diagram, the actual order of sub-views in the document is set by the order of view attributes of the parent view. Open the specification dialog of the parent view and select the 'Attributes' node in the left tree. For example, the sub-sections for Section 3.2 are listed in the attributes table below.



To reorder sub-views, change the order in the attributes table. For example, to have "Insert Diagram to Document" before "Edit Text in Document", select "Edit Text in Document" in the attributes table and click "Down" button.



The sub-sections will now be rendered in the updated order.

3.2.4. Insert Diagram to Document

To insert a diagram to document, create a diagram only view. This view is helpful for displaying model diagrams as sections or sub-sections within your document. To create a diagram only view, take following steps:

First, locate a parent view where you like the diagram to appear. In the example below, the parent view is "Insert Diagram to Document".

Second, create a placeholder view for diagram. In this example, it is called "My Diagram View". Connect it to the parent view.

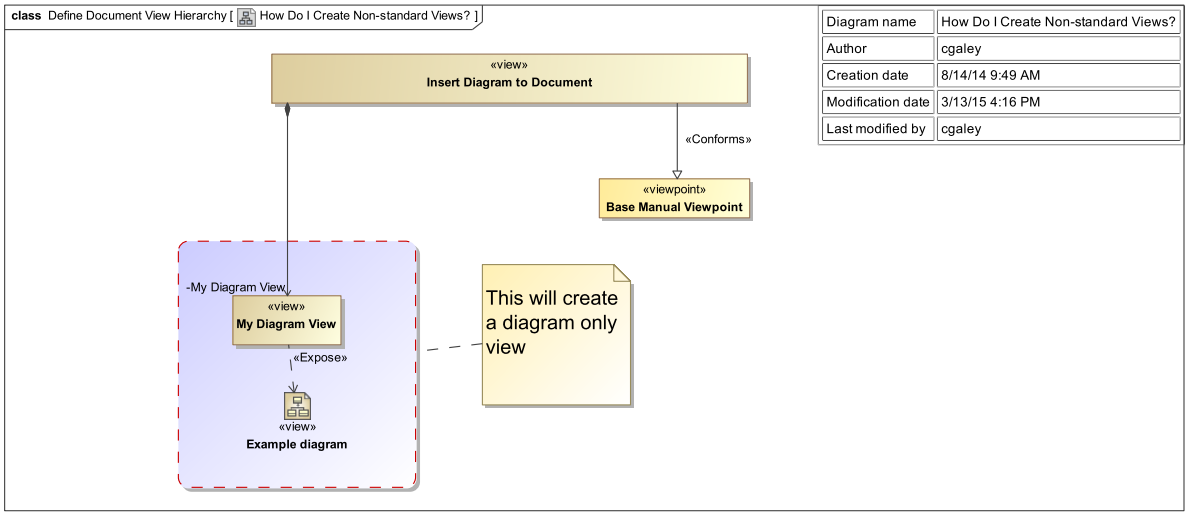
Third, locate the diagram you would like to display in the containment tree. Select it in the containment tree and drag it onto the diagram containing the placeholder view.

Fourth (optional), right click on the diagram symbol (in the diagram or in the containment tree). Select "Stereotype" from the list and in the resulting search box enter "view." Select "view [Class, Diagram, Package]" and click apply.

Fifth, define "Expose" relationship from the placeholder view to the diagram. This relationship allows the view to see the image of the diagram.

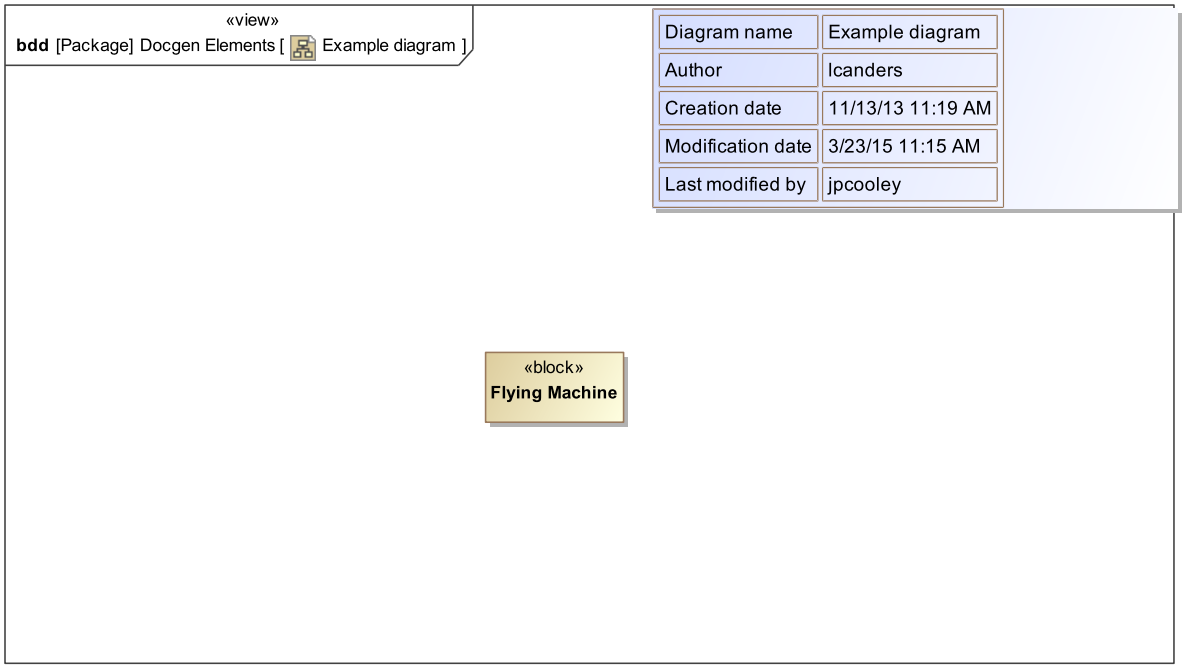
This setup will result in the default behavior of the view showing any exposed diagrams and their documentations. The image below highlights what the view relationships should look like.

Figure 3.2. How Do I Create Non-standard Views?



3.2.4.1. My Diagram View

Figure 3.3. Example diagram



This is the example diagram for the Diagram Only view.

3.2.5. Edit Text in Document

Text in your document can be edited either within MagicDraw environment or EMS Web environment (e.g., View Editor). When texts are changed using either of the methods, the changes will need to be synchronized between MagicDraw model and EMS model. If texts are changed in a MagicDraw model, the changes need to be committed to EMS. If texts are changed using EMS View Editor, the changes need to be accepted into the MagicDraw model after model validation. While the end results will be the same, most people find View Editor easier to use and provide more word processing features. Check your Project's Modeling Policy for preferred editing methods.

Editing Text in View Editor

View Editor is the recommend tool and it offers more advanced text entry and editing capabilities. First, within your View Editor document, first select the section you would like to edit from the table of contents. Then select the edit button which will bring up the word processing toolbar and the text entry area. (If you do not have a write privilege for the document, you may not see the edit button. Consult with an administrator for editing permission.) For detailed descriptions of the View Editor toolbar and, see the EMS Applications Manual.

Editing Text in MagicDraw

First, select a document view that contains text to edit. Then double click on the view to open up the specification dialog. Select "Documentation/Hyperlinks" node from the left tree. Documentation texts will be available in the Documentation box. Content can be entered as plain text or html if the "HTML" check box is selected. One significant limitation of this method is that no newlines are recognized when it is displayed in View Editor. For a simple sentence or paragraph, this approach should be enough.

DocBook Tags

Docbook is a markup language for technical documents. If you plan to generate documents locally using MDK plug-in, it is possible to use Docbook tags in text. Simply type in the docbook markup (make sure you have valid docbook tags!) in the model. Note that this method is not applicable to EMS View Editor.

HTML Tags

Check the "HTML" checkbox in MagicDraw on the text/documentation field. DocGen will convert html tags to docbook. For the html conversion, there are some limitations to what html tags you can use. In the Magicdraw Advance HTML Editor, you can look at the HTML source tab to see if your html source can be successfully converted into DocBook. Below are a list of tags that can be converted:

- p
- ul
- ol
- li
- b
- i
- u
- a
- sub
- sup
- pre

Examples:

A paragraph

- unordered list item
  - nested list item

1. ordered list item

Bold Italic underline or hyperlinked text , email , and $_{\text{subscripts}}$ and $^{\text{superscripts}}$.

Other preformmatted text like code and xml (the < chars in xml still have to be escaped in the html source). <xml> <sometag> hellllloo and other stuff <> </sometag> </xml>

All other styles or font setting will be stripped off as docgen processes it.

Tables made in the Advanced HTML Editor may also be supported, but there must be a <caption></caption> as the first child of the table. The caption will be used as the title of the table generated. You'll have to add this in the HTML source tab.

Table 3.1. HTML Table

example	html
table	in html editor

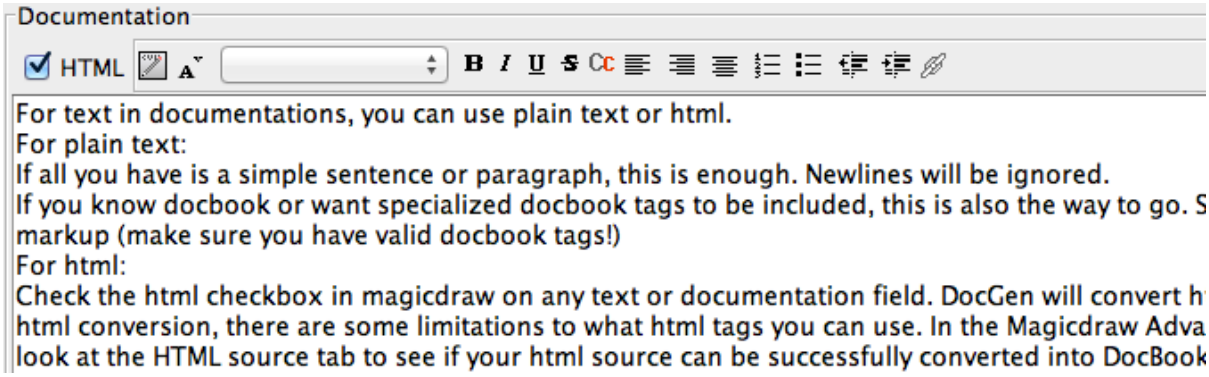
If you don't want a title for your table, make it an informal table. In the HTML source tab, put in <table class="informal" at the table tag. Don't put in the caption. This also means this table won't get an entry in the table of contents.

[cannot find [cf:name] with Id: _17_0_5_1_87b0275_1404621092813_395474_25797]

Table 3.2.

html	table
without	caption (title)

You can access html on the MagicDraw side by going to the specification window and documentation and selecting html. You can also directly insert html on EMS View Editor.



3.3. Query and Visualize EMS Project Content

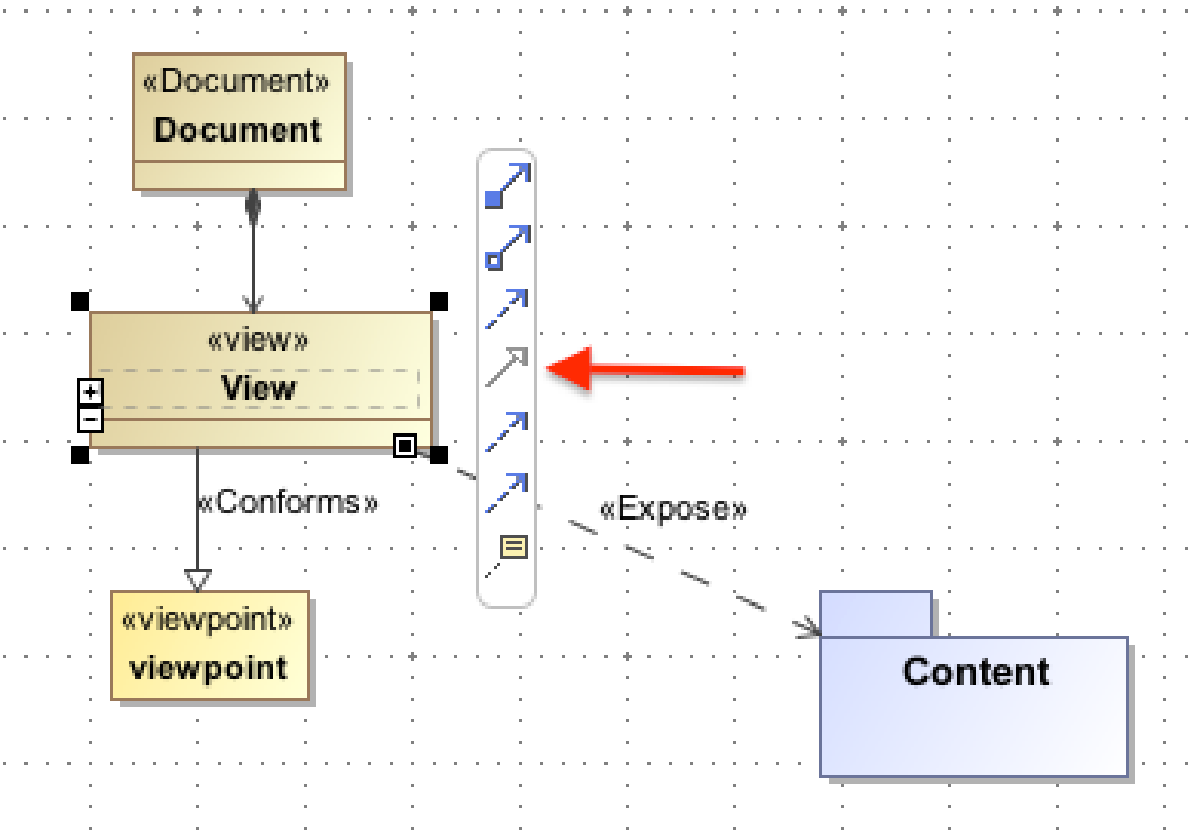
The instructions so far showed how to create a document and add views manually. One common need in systems engineering is to generate documents of the same format repeatedly. It would be very helpful to automatically extract data from a model, and lay out its content in a pre-defined format. This is made possible by using viewpoints.

A viewpoint specifies conventions and rules for constructing a view and can be thought of as a template for the view that generates contents and their format in a document. Viewpoints enable the inclusion of images, text,

tables, and other elements queried from the model into a document. Once a template is defined, it can be reused in different parts of a model. Viewpoints are defined using Viewpoint Method diagrams provided by MDK plug-in. Your project may have standard viewpoints for its documents; check your Project Modeling Policy for further details.

3.3.1. Create Viewpoints

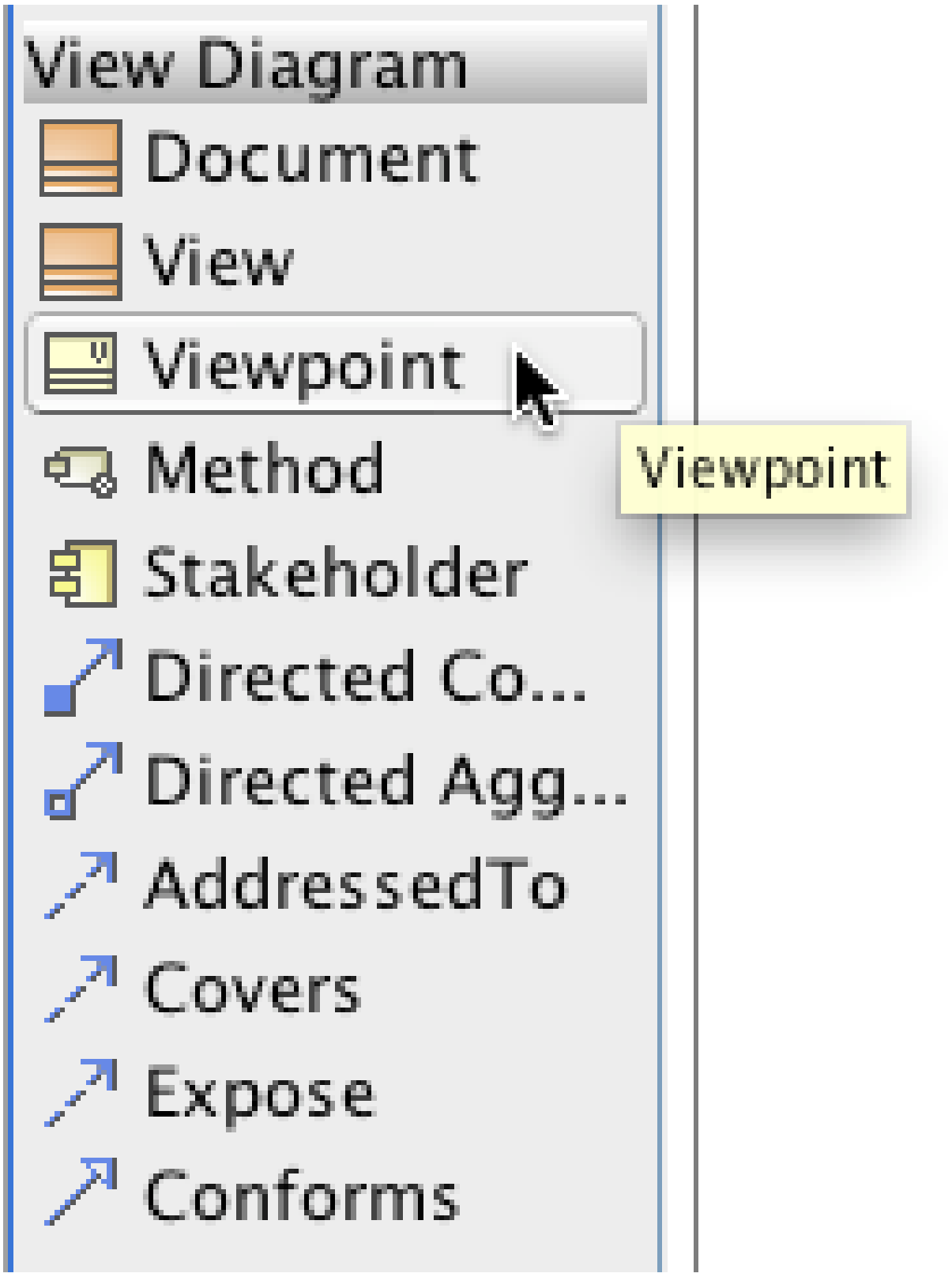
Once a viewpoint is defined along with its viewpoint method diagram, it can be applied to a view using a <<conforms>> generalization relationship. Each view can only have one viewpoint. But a viewpoint can be applied to more than one view. Proper modeling practice requires every view to conform to viewpoint, and warnings will appear if this practice isn't followed. The <<conforms>> generalization can be created through the side menu bar or by using the menu that appears when the element is selected on the diagram.



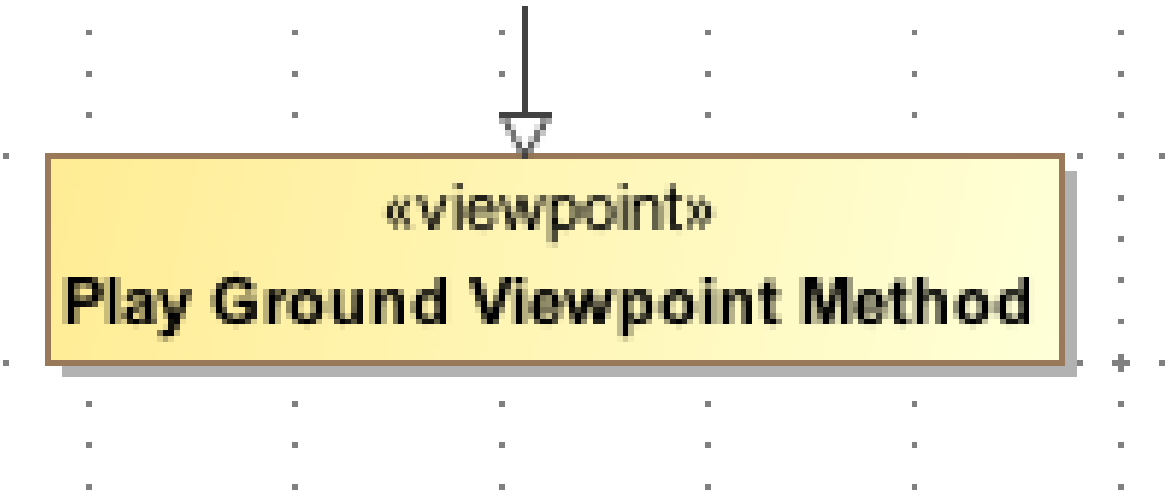
The view with a viewpoint is a custom view that can be applied to a certain type of element as defined in the viewpoint method diagram. So, the next step is to select a model element to which the view will be applied. Select a model element in the containment tree, and drag it onto your view diagram. In the image above, the selected element is a package called "Content." Then, connect the view to the package with the "expose" relationship. The "expose" relationship is a stereotype of the "dependency" relationship.

3.3.1.1. Create Viewpoint Element

To create a viewpoint, navigate to the side menu bar in Magic Draw, as shown in the picture below. Select the viewpoint icon, and drag the element onto the view diagram.

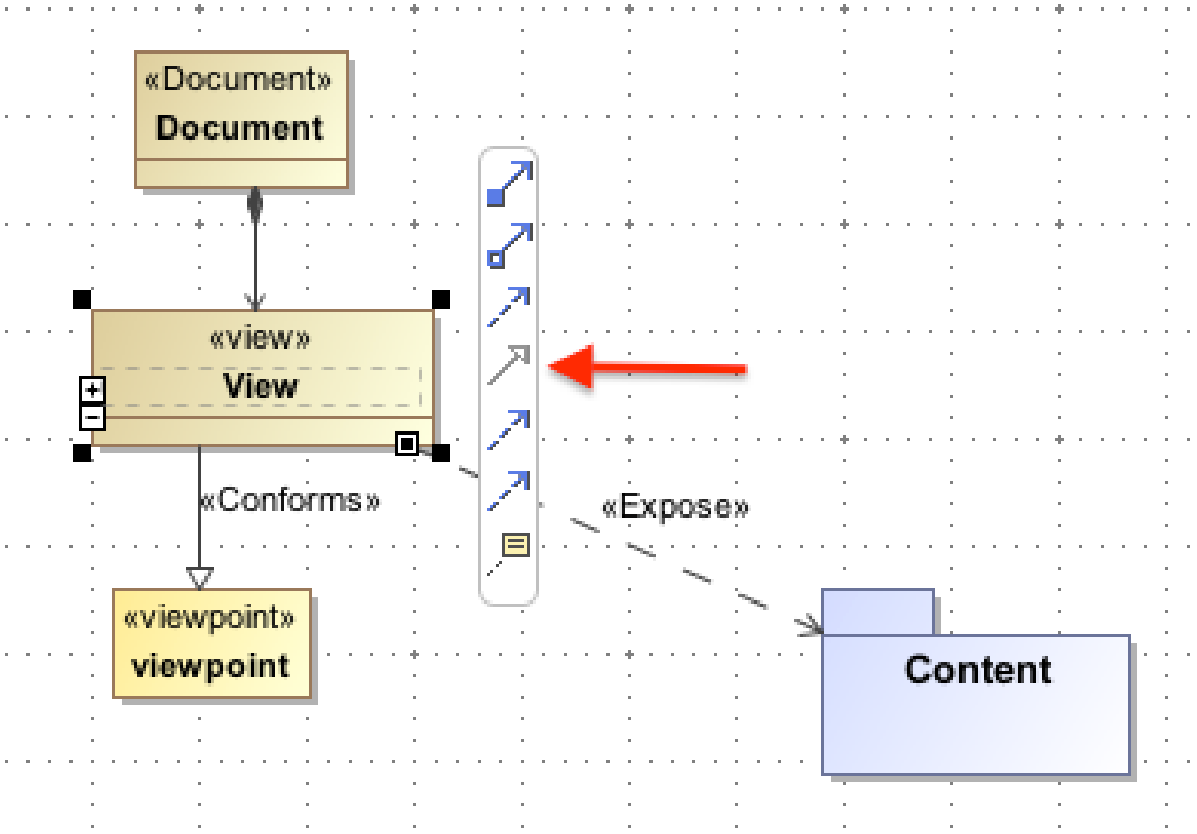


Below is an example of what a viewpoint looks like on a diagram.



3.3.1.2. Link to View and Expose Content

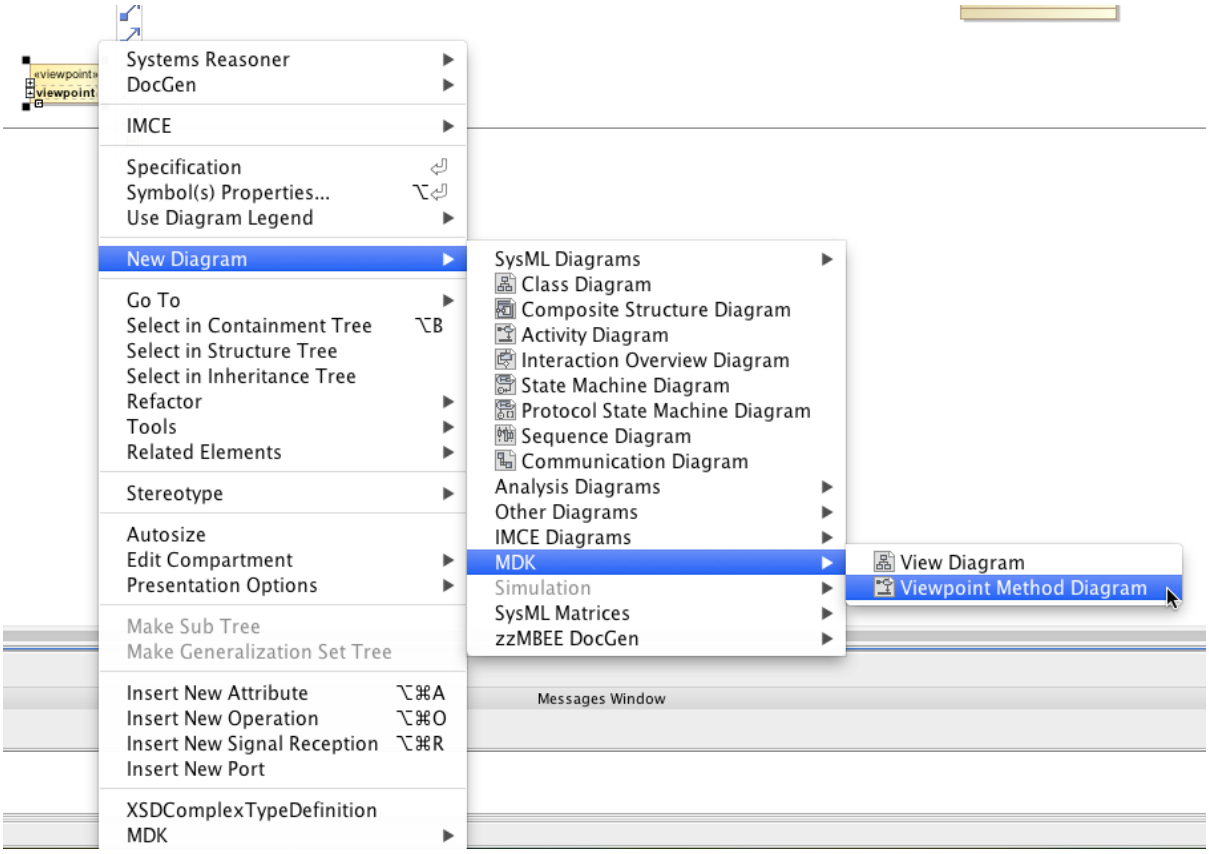
A view is linked to the viewpoint element using a <<conforms>> generalization. Each view can only have one viewpoint. Proper modeling practice requires every view to conform to viewpoint, and warnings will appear if this practice isn't followed. The <<conforms>> generalization can be created through the side menu bar or by using the menu that appears when the element is selected on the diagram.



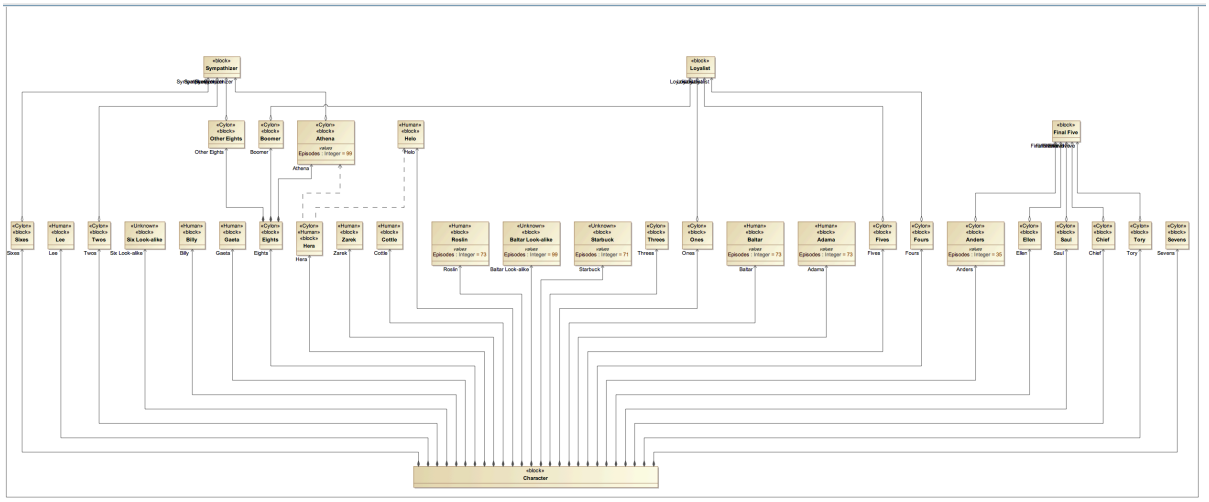
Since the viewpoint is used to automatically generate document content from the model, at this point you must determine which model content you would like to work with. Once identified in the containment tree, drag over the applicable model package and place it onto your diagram. In the image above, the selected package is named "Content." Then connect the view to the package with the "expose" relationship. The "expose" relationship is a stereotype of the "dependency" relationship.

3.3.1.3. Create Viewpoint Method Diagram

Right click on the viewpoint element on the view diagram and create an MDK viewpoint method diagram as shown below. A viewpoint method is an activity that creates the viewpoint, and the viewpoint method diagram is a specialized activity diagram.



The next section goes into more detail on how to populate the viewpoint method diagram to define a customized document view. To illustrate the approach, an example model has been created (shown below for context).



With the viewpoint method diagram that you created, you can add various actions to customize how you would like the content in the document section displayed. In every viewpoint method diagram, each activity begins with an "Initial Node" and either ends with an "Activity Final" or "Flow Final". "Control Flows" connect the different nodes and behaviors. These tools can be found in the side bar and/or in the menu that appears when the element is selected on the diagram.

The following two sections describe elements that can be used for viewpoint method diagrams. The first section describes how the data is obtained from the existing model using several types of operators: collect, filter, and sort. The second section describes how you format or present the obtained data. The presentation elements and formatting types discussed are tables, images, paragraphs, lists, and sections.

In the following examples, you will see completed viewpoint method diagrams which will show both the data manipulation and the content presentation. In fact, the viewpoint method diagrams show the viewpoint methods that were used to generate the sections they are shown in.

(Note that if you have questions on something that is presented in the context of the example but that hasn't been explained yet, you can navigate to the section describing that feature for further information.)

3.3.2. 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.

3.3.2.1. Collect/Sort/Filter Model Elements

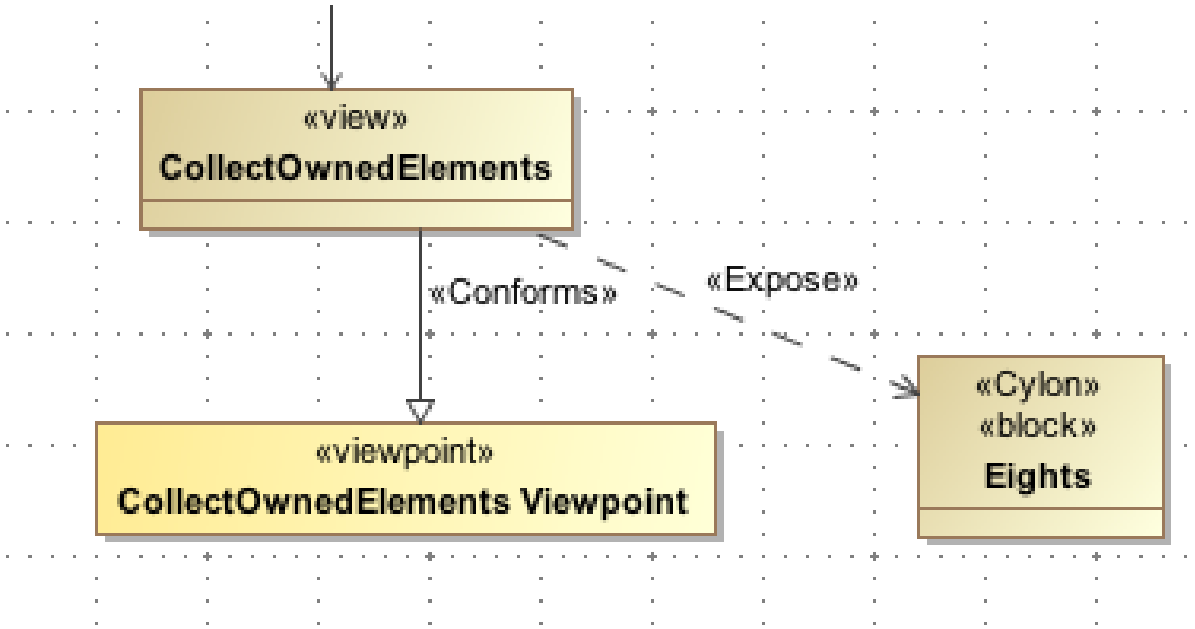
Once exposed to a view, elements can can be operated on by three types of viewpoint operators (Collect, Sort, and Filter). These operations can be used to expand or narrow the collection of elements that are used in the viewpoint method.

3.3.2.1.1. 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.

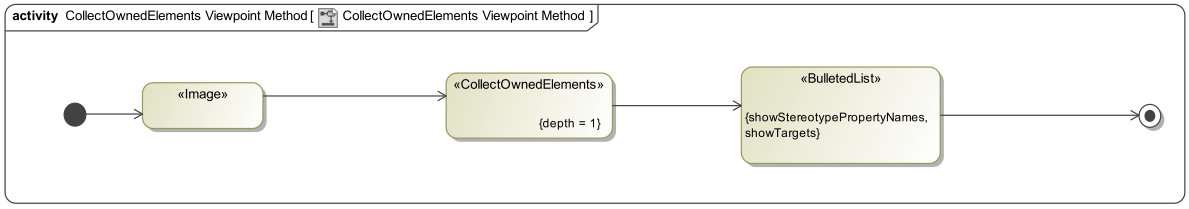
3.3.2.1.1.1. CollectOwnedElements

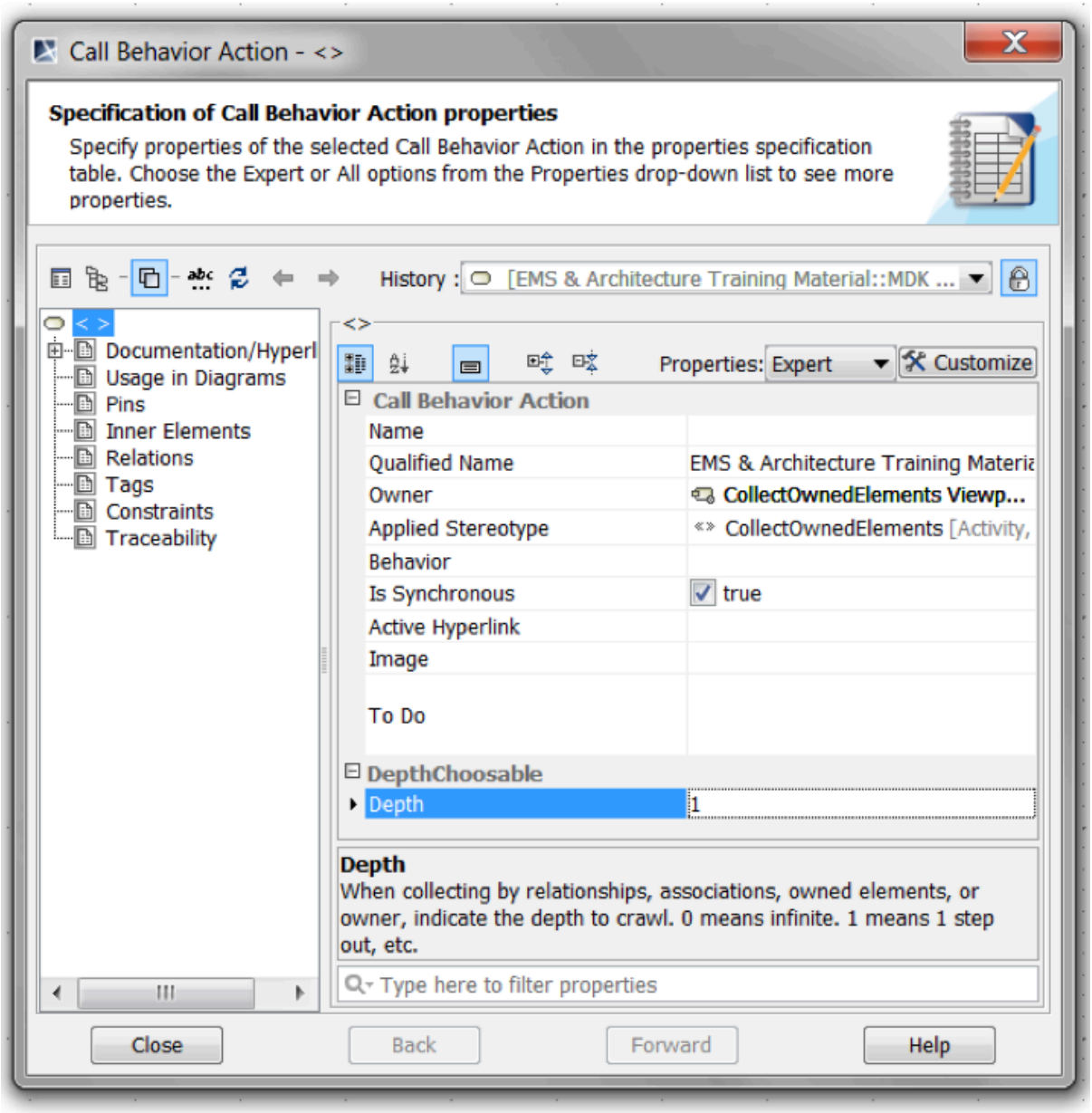
"CollectOwnedElements" gathers anything owned by the exposed element(s) in the model to be used in the viewpoint method. A common use case would be when a group of elements contained within a package need to be exposed to a view. Instead of individually exposing each element, a user can employ "CollectOwnedElements"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



In the image above, the block "Eights" is exposed. The viewpoint method diagram shown in the image below was then created. Note that "CollectOwnedElements" gives the part properties of "Eights". Also note that "CollectOwnedElements" has a specification that can be used to further define the behavior. The most useful is "DepthChoosable" at the bottom. It allows the user to dictate how many levels of ownership are used; this example uses one level.

Figure 3.4. CollectOwnedElements Viewpoint Method





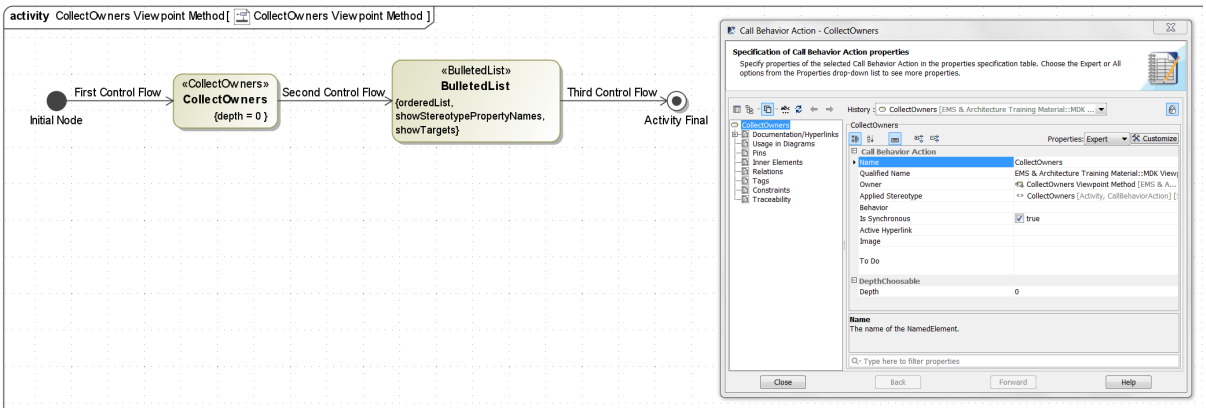
As you can see in the below list "CollectOwnedElements" used by itself may return elements that are not desired in your table or list. In this example this can be seen by the empty bullet as well as the generic "name" entries. Different collect operations or combining "CollectOwnedElements" with a filter operator (described later) can

help resolve these issues. Note, in this example the "BulletedList" behavior, also described later, is used by the viewpoint to display the results.

- Boomer
- Athena
- Other Eights
- test

3.3.2.1.1.2. CollectOwners

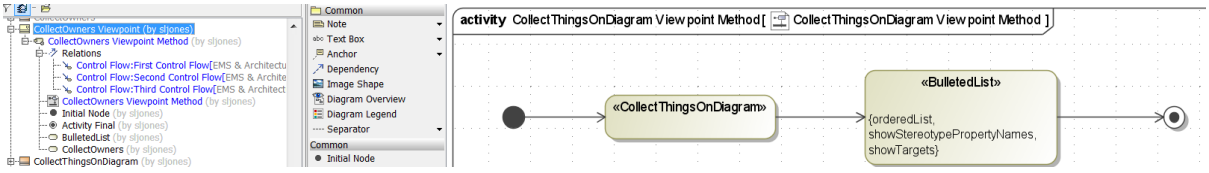
"CollectOwners" is essentially the opposite of "CollectOwnedElements". It gathers all of the owners (from the containment tree) of the exposed element(s) for use in the viewpoint method. Using the "Eights" block again, this returns the path of ownership in the containment tree. It starts with the direct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



1. Example Elements
2. Viewpoint Testing
3. Unit Testing
4. Components
5. EMS Support
6. Architectural Patterns & Examples

3.3.2.1.1.3. CollectThingsOnDiagram

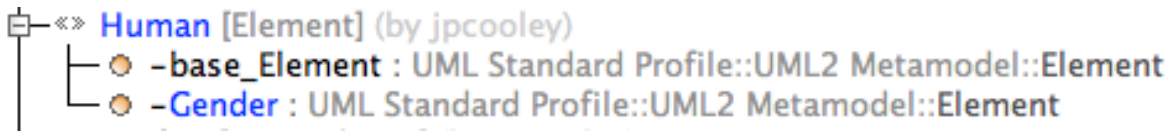
CollectThingsOnDiagram will collect all the elements depicted on a diagram. To demonstrate this, the previous CollectOwners viewpoint method diagram was exposed. Note that the *name* of each element as well as the viewpoint method itself is listed below. Names are often not automatically created with an element, so these must be inserted in the model. Otherwise a "no content for..." message will appear in the name's spot.

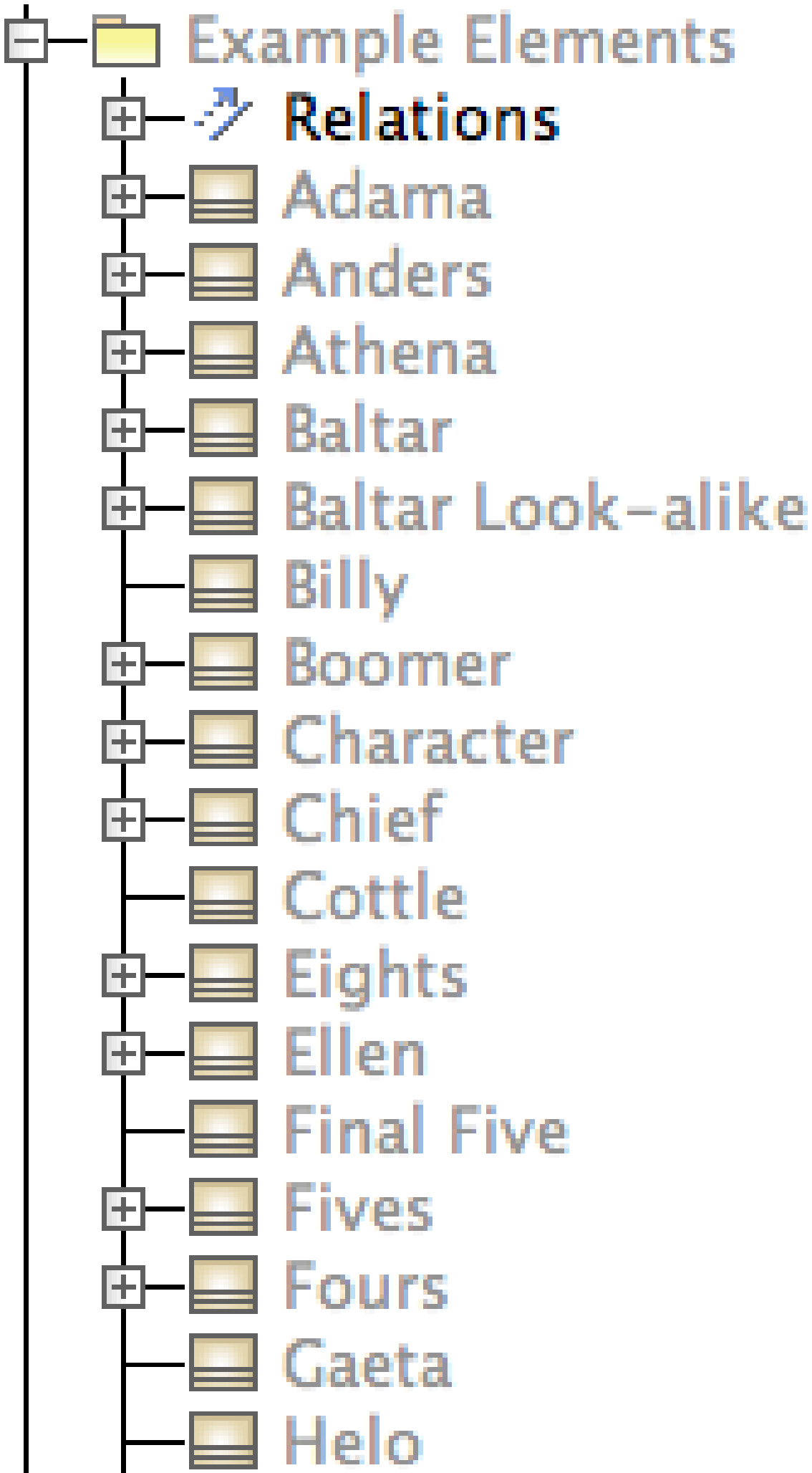


1. Activity Final
2. Third Control Flow
3. Initial Node
4. Second Control Flow
5. First Control Flow
6. CollectOwners
7. CollectOwners Viewpoint Method
8. BulletedList

3.3.2.1.1.4. CollectByStereotypeProperties

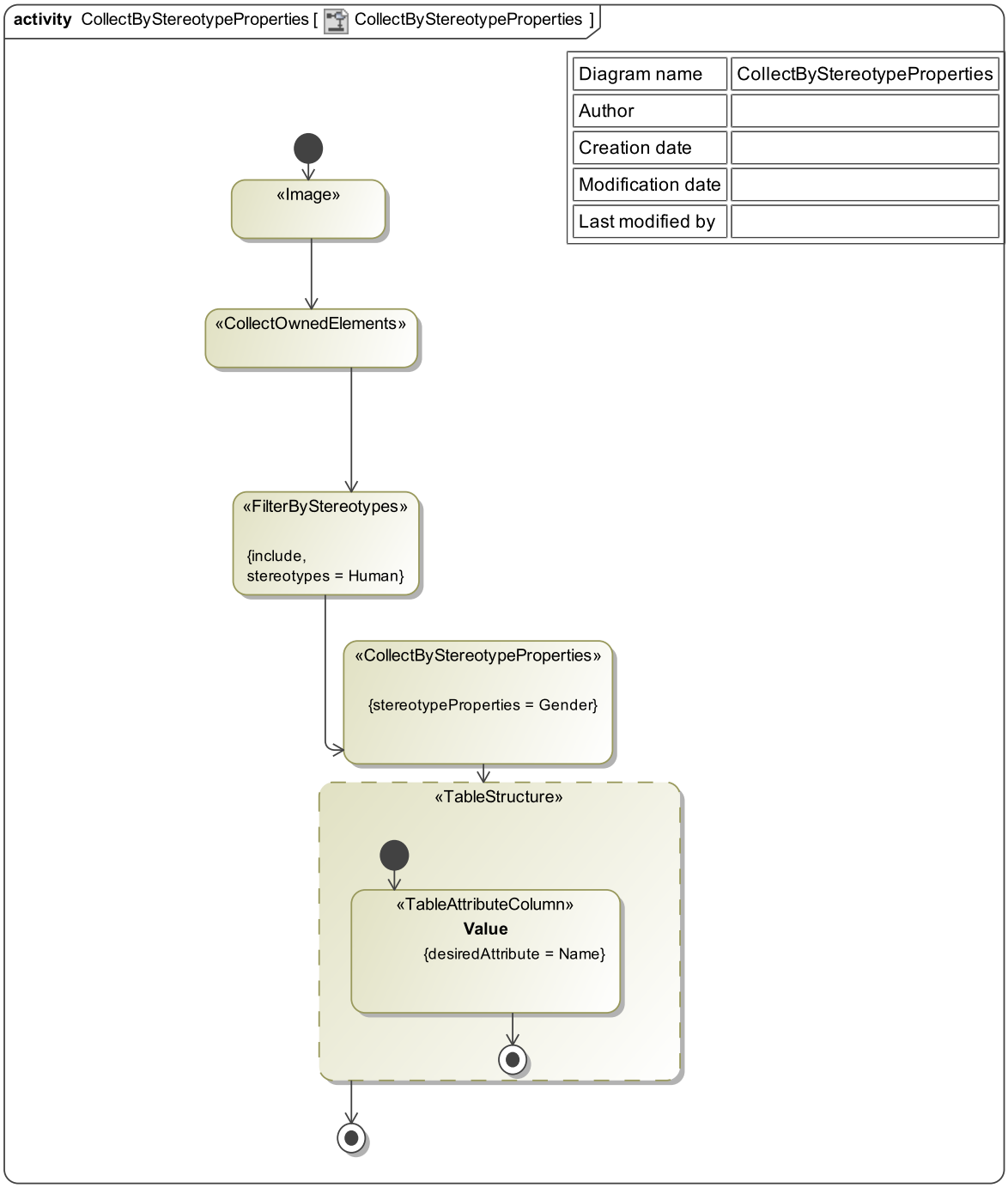
This collection action allows a user to get at a specified property value(s) of a stereotype. This might be helpful when multiple different values are specified for a single stereotype property. It is important to note that this functionality does not work with string values and will return only one 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).



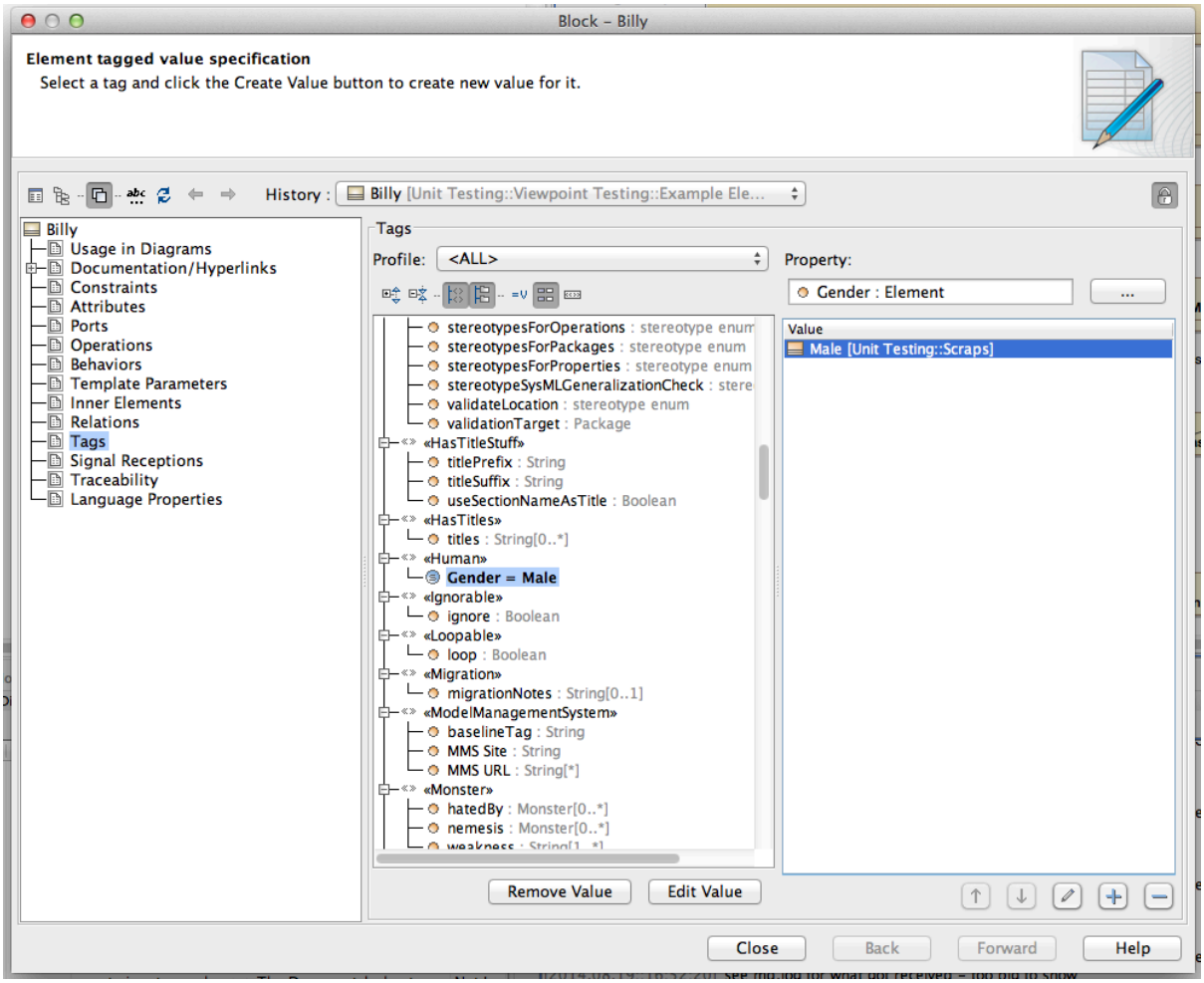

Example Elements
Relations
Adama
Anders
Athena
Baltar
Baltar Look-alike
Billy
Boomer
Character
Chief
Cottle
Eights
Ellen
Final Five
Fives
Fours
Gaeta
Helo

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.

Figure 3.5. CollectByStereotypeProperties



The image below shows the specification window for the block "Billy". In order to set the property value for the stereotype <<Human>>, first select Tags, find the stereotype in the list, and then create the value. In this example, since the property type is element, the Class element named "Male" is designated as the value of the property.



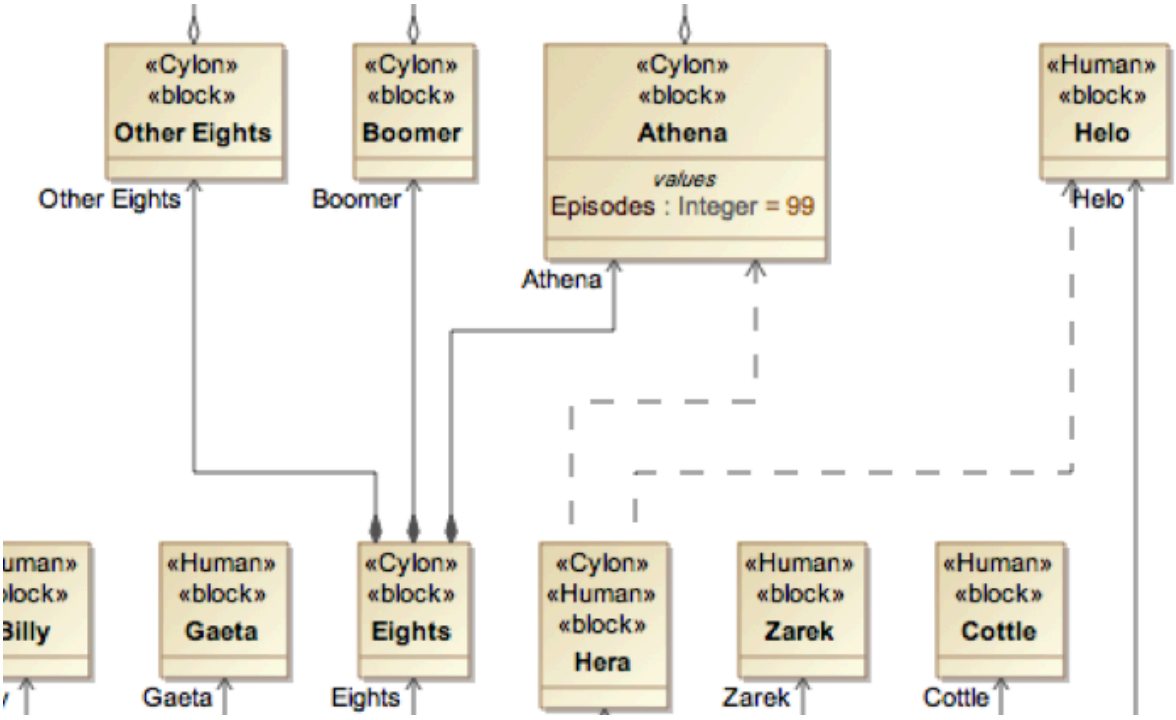
The table below displays the outcome of the activity diagram shown above. As expected the value of the stereotype property named "Gender" of the <<Human>> stereotype for the character Billy is shown. If other characters from the list, stereotyped <<Human>>, had specified values for the Gender stereotype property, then their values would appear in the table as well.

Table 3.3.

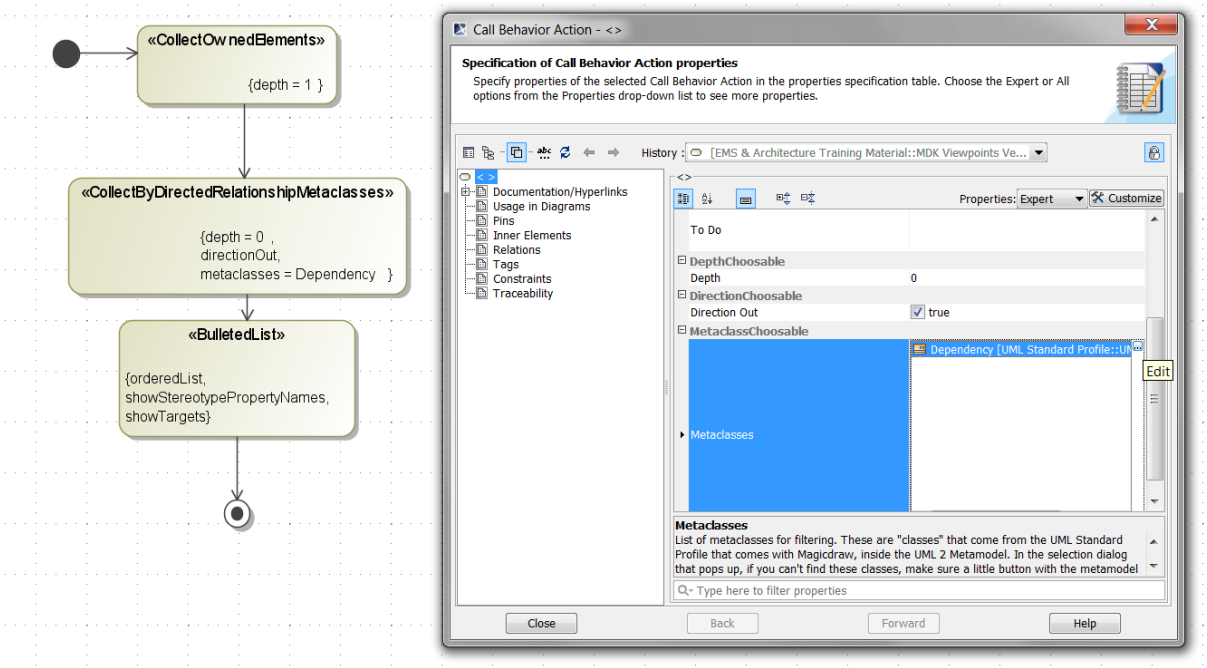
Value
Male

3.3.2.1.1.5. CollectByDirectedRelationshipMetaclasses

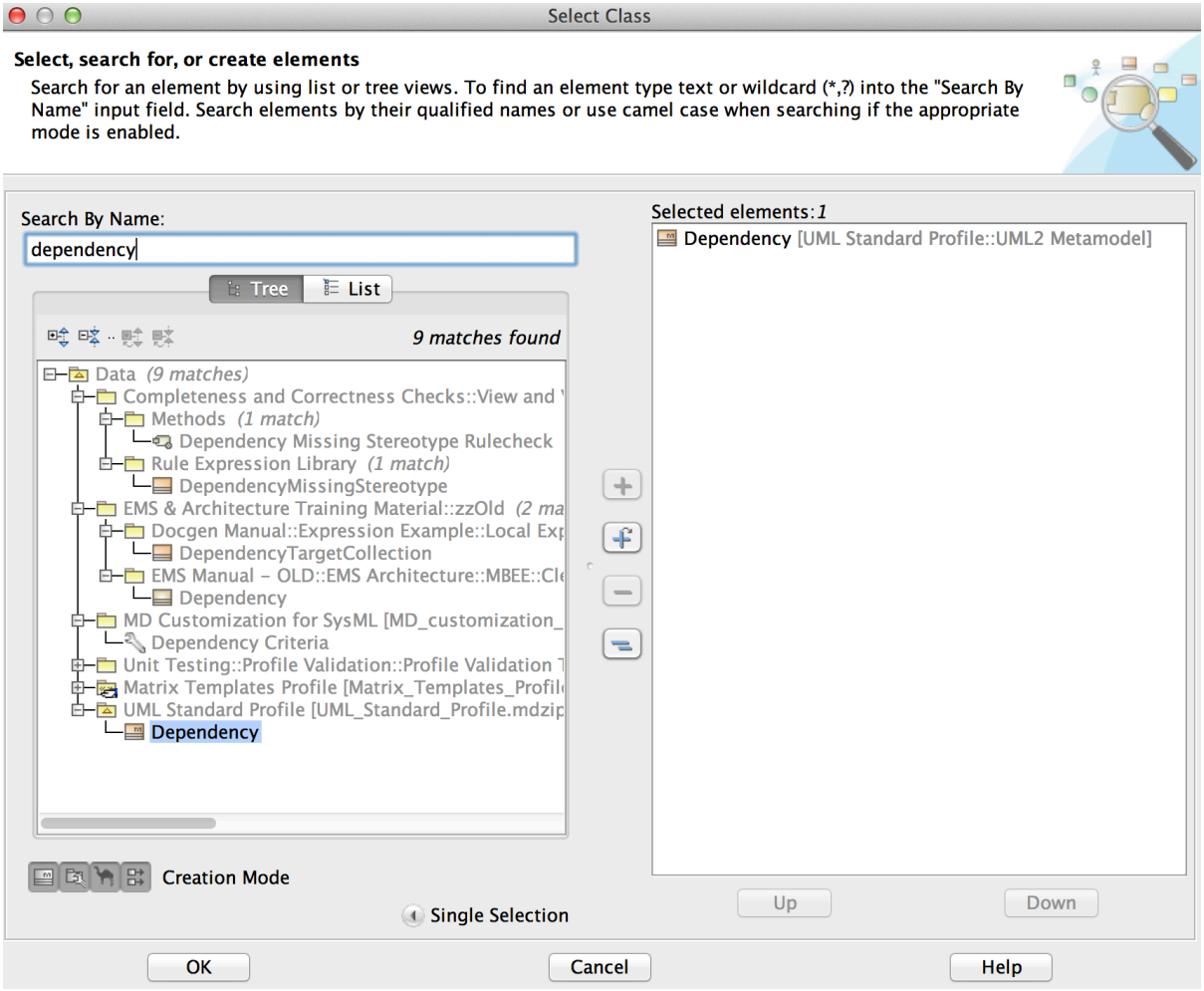
CollectByDirectedRelationshipMetaclasses action collects elements based on the relationships that other elements use to connect to them. Note that in the example, Helo and Athena are connected to Hera with a "Dependency" (the dashed arrowed line). In other words, Hera depends on Helo and Athena. These 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 bulleted list generated below.

1. SortByProperty
2. CollectClassifierAttributes
3. CollectOwners
4. SortByAttribute
5. CollectOwnedElements
6. Hera

3.3.2.1.1.5.1. 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 Link to View and Expose Content 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 CollectOwnedElements and CollectOwners 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.

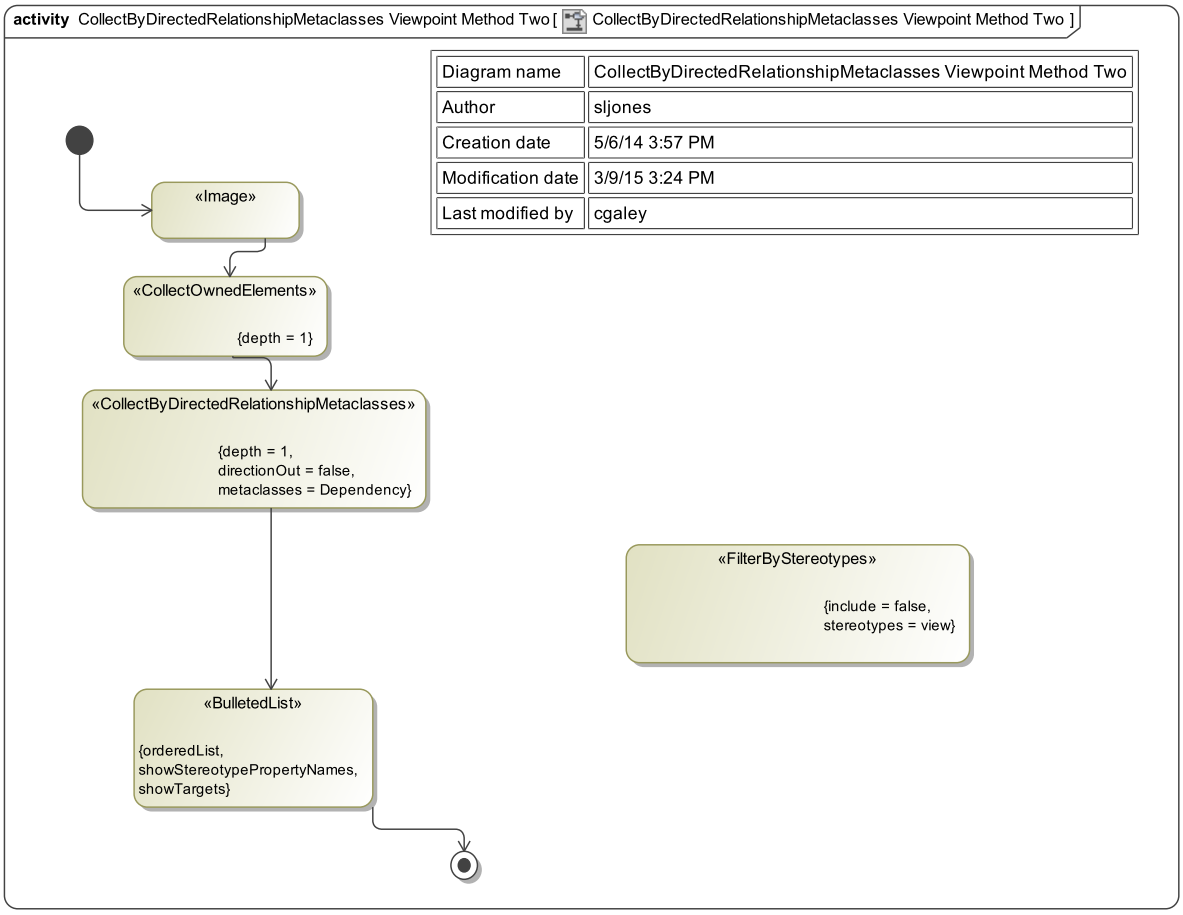
4. The last step is that the search follows the relationship and outputs what is at the other end of the relationship.

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 3.6. CollectByDirectedRelationshipMetaclasses Viewpoint Method Two

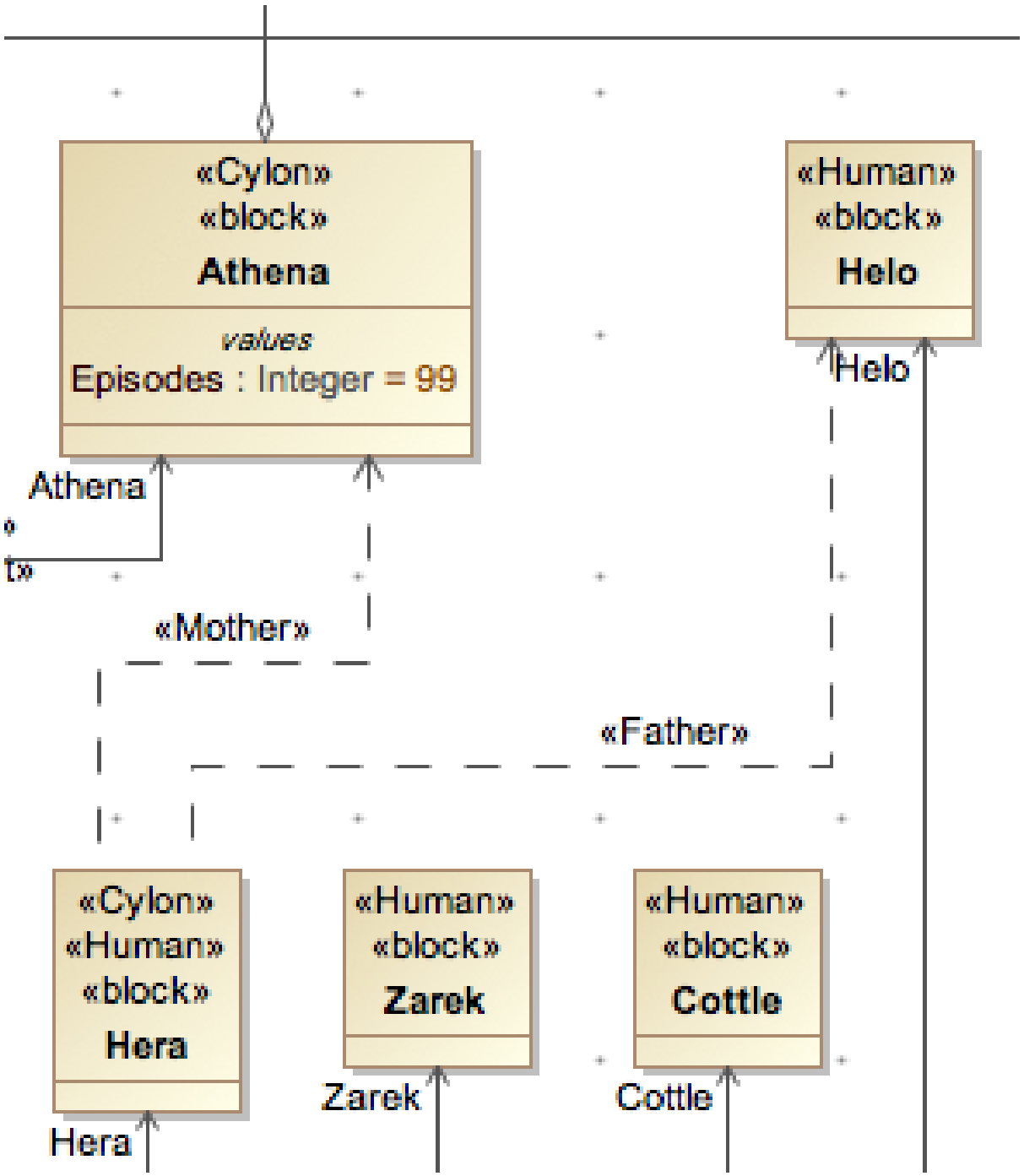


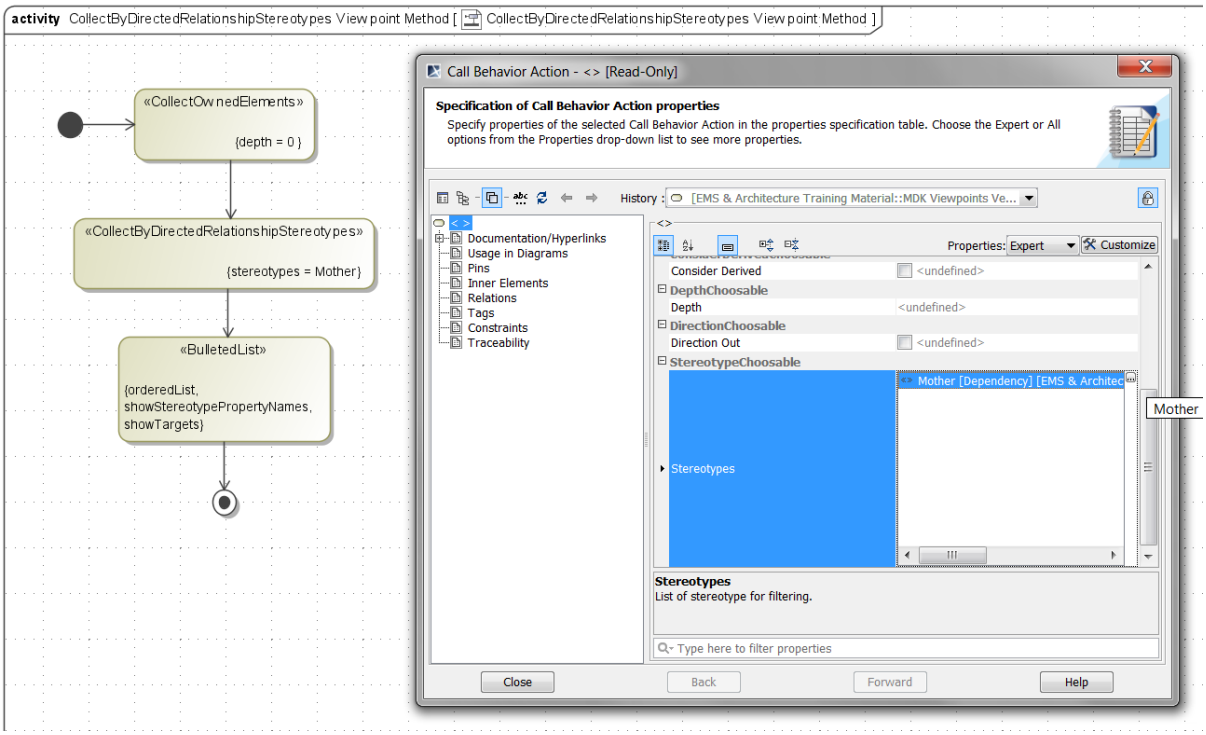
1. SortByProperty
2. CollectClassifierAttributes
3. CollectOwners
4. SortByAttribute
5. CollectOwnedElements
6. Hera

3.3.2.1.1.6. 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).

For this example, the collection is done by the stereotype <<Mother>>, which is on only one of the relationships shown in the previous example. Therefore, only the element at the arrow end of that relationship is listed below. (Note that Direction Out is once again undefined.)


«Cylon»
«block»
Athena
values
Episodes : Integer = 99
«Human»
«block»
Helo
Helo
Athena
t»
«Mother»
«Father»
«Cylon»
«Human»
«block»
Hera
«Human»
«block»
Zarek
«Human»
«block»
Cottle
Hera
Zarek
Cottle

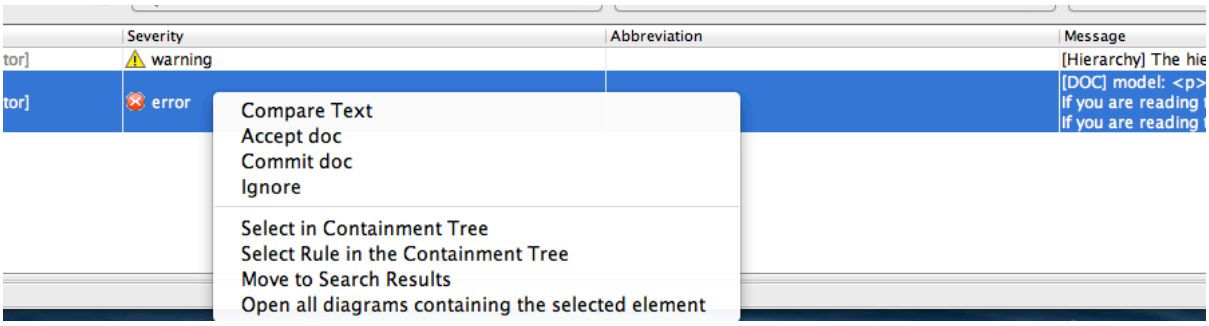


Selecting [...] under StereotypeChoosable in the specification will open the element selector, allowing the selection of the desired stereotype. Athena is the target of the dependency stereotyped <<Mother>> thus she appears in the list. Note: Hera is dependent on Athena, thus the arrow points from Hera (the source) to Athena (the target).

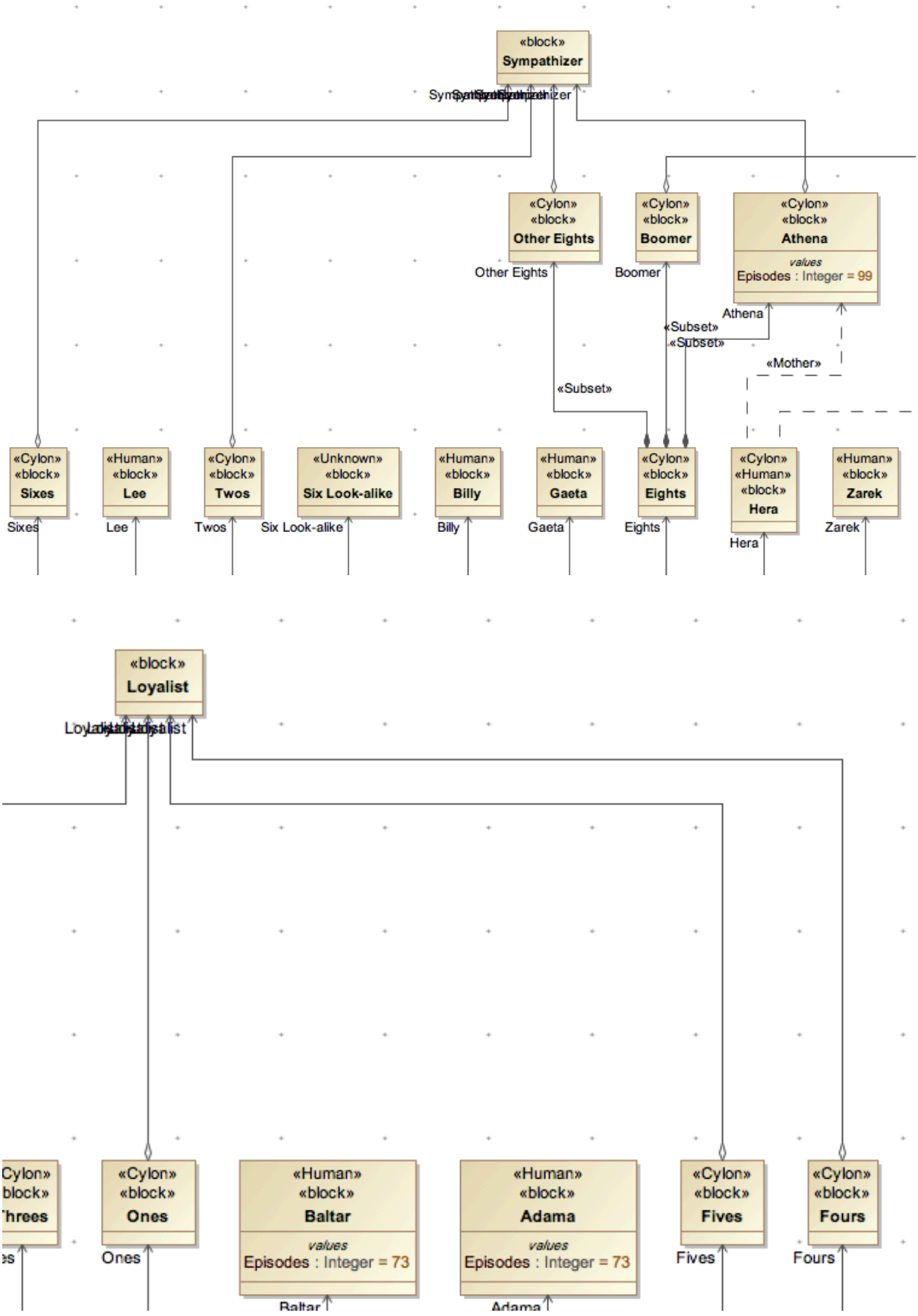
1. Athena

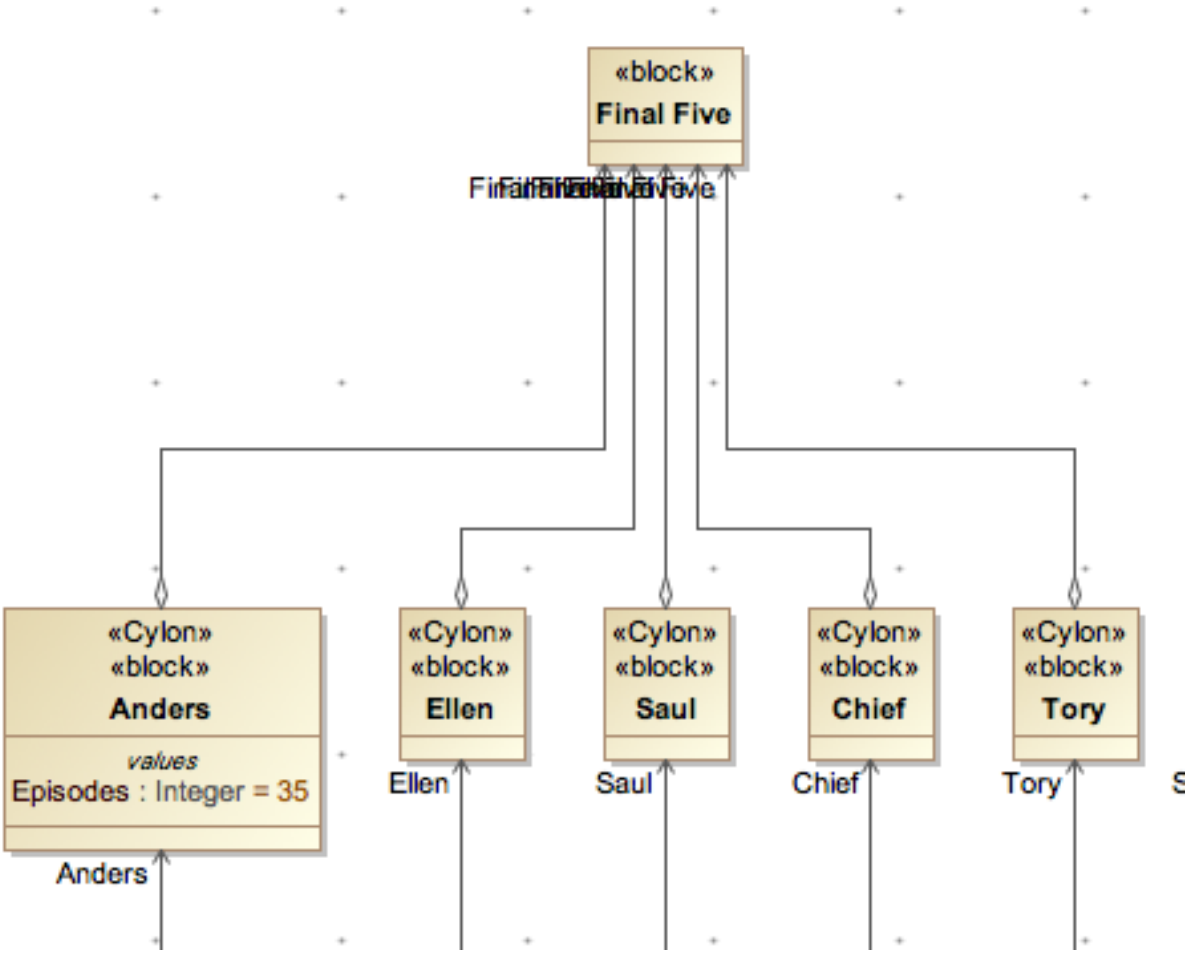
3.3.2.1.1.7. CollectByAssociation

CollectByAssociation collects the blocks with aggregation of either composite or shared. In other words, both white diamond associations (shared) and black diamond associations (composite) can be collected by this action. The CollectOwnedElements or CollectOwners operation needs to be used before CollectByAssociation. The CollectByAssociation action will then collect only those blocks that have the specified aggregation (composite or shared). The viewpoint method to collect shared properties is shown below. In the specification for CollectByAssociation block, "shared" is selected in the AssociationType section.



In the view diagram, this view exposes the Example Elements package and conforms to the CollectByAssociation viewpoint method. The result of this viewpoint method is shown in the list below. The following images show the shared (white diamond) associations with the targets (Sympathizer, Loyalist, Final Five) that appear in the list at the bottom of the page.

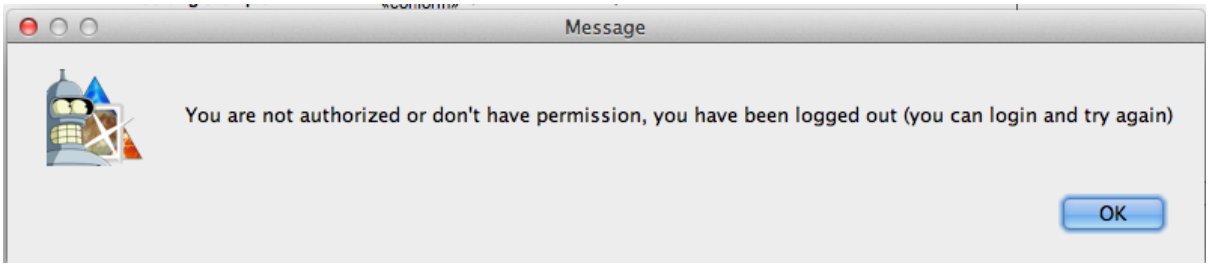

«block»
Sympathizer
«Cylon»
«block»
Other Eights
Other Eights
«Cylon»
«block»
Boomer
Boomer
«Cylon»
«block»
Athena
values
Episodes : Integer = 99
Athena
«Subset»
«Subset»
«Mother»
«Subset»
«Cylon»
«block»
Sixes
Sixes
«Human»
«block»
Lee
Lee
«Cylon»
«block»
Twos
Twos
«Unknown»
«block»
Six Look-alike
Six Look-alike
«Human»
«block»
Billy
Billy
«Human»
«block»
Gaeta
Gaeta
«Cylon»
«block»
Eights
Eights
«Cylon»
«Human»
«block»
Hera
Hera
«Human»
«block»
Zarek
Zarek
«block»
Loyalist
«Cylon»
«block»
Ones
Ones
«Human»
«block»
Baltar
values
Episodes : Integer = 73
«Human»
«block»
Adama
values
Episodes : Integer = 73
«Cylon»
«block»
Fives
Fives
«Cylon»
«block»
Fours
Fours



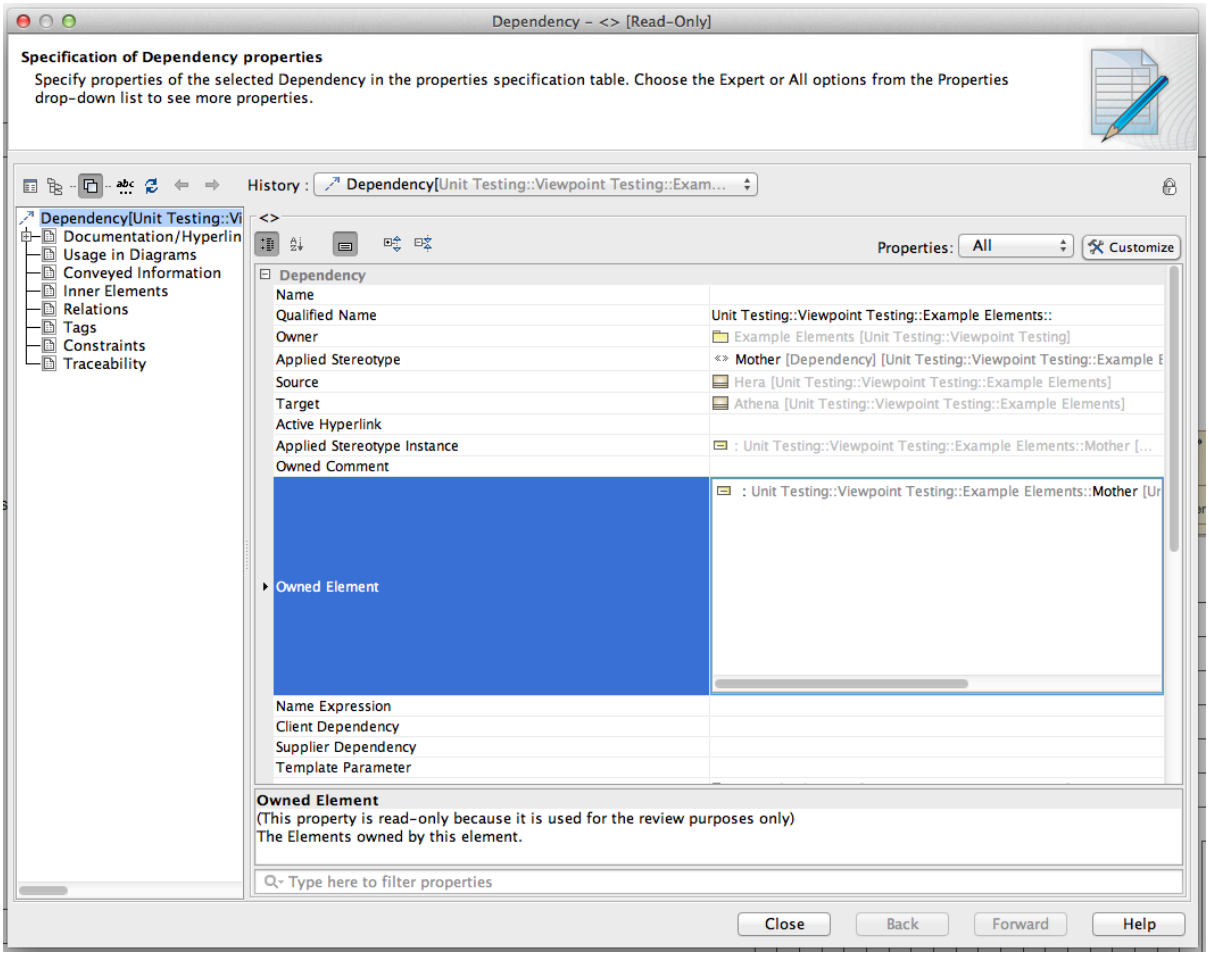
1. Loyalist
2. Sympathizer
3. Final Five

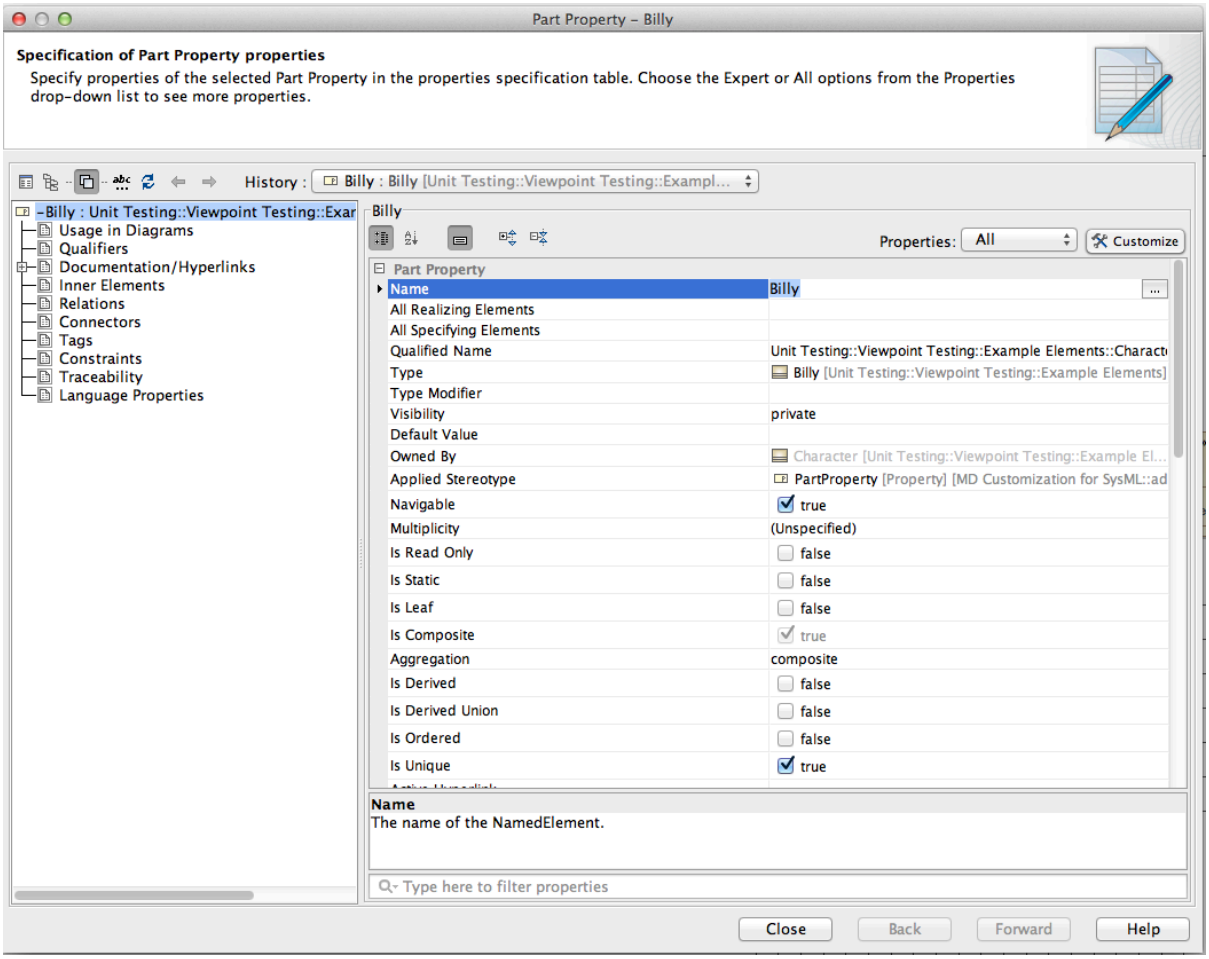
3.3.2.1.1.8. CollectTypes

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. The specification for the CollectTypes operation is also shown below.



In the view diagram, the Example Elements package is exposed again. Notice in the images below that all types are collected. For example, Mother is type of dependency and Billy is the type of part property created by the composite association connected with the "Billy" block.


Dependency – <> [Read-Only]
Specification of Dependency properties
Specify properties of the selected Dependency in the properties specification table. Choose the Expert or All options from the Properties drop-down list to see more properties.
History :
Dependency[Unit Testing::Viewpoint Testing::Exam...
Dependency[Unit Testing::Vi
Documentation/Hyperlin
Usage in Diagrams
Conveyed Information
Inner Elements
Relations
Tags
Constraints
Traceability
<>
Properties:
All
Customize
Dependency
Name
Qualified Name
Unit Testing::Viewpoint Testing::Example Elements::
Owner
Example Elements [Unit Testing::Viewpoint Testing]
Applied Stereotype
Mother [Dependency] [Unit Testing::Viewpoint Testing::Example
Source
Hera [Unit Testing::Viewpoint Testing::Example Elements]
Target
Athena [Unit Testing::Viewpoint Testing::Example Elements]
Active Hyperlink
Applied Stereotype Instance
: Unit Testing::Viewpoint Testing::Example Elements::Mother [...
Owned Comment
Owned Element
: Unit Testing::Viewpoint Testing::Example Elements::Mother [Ur
Name Expression
Client Dependency
Supplier Dependency
Template Parameter
Owned Element
(This property is read-only because it is used for the review purposes only)
The Elements owned by this element.
Type here to filter properties
Close
Back
Forward
Help



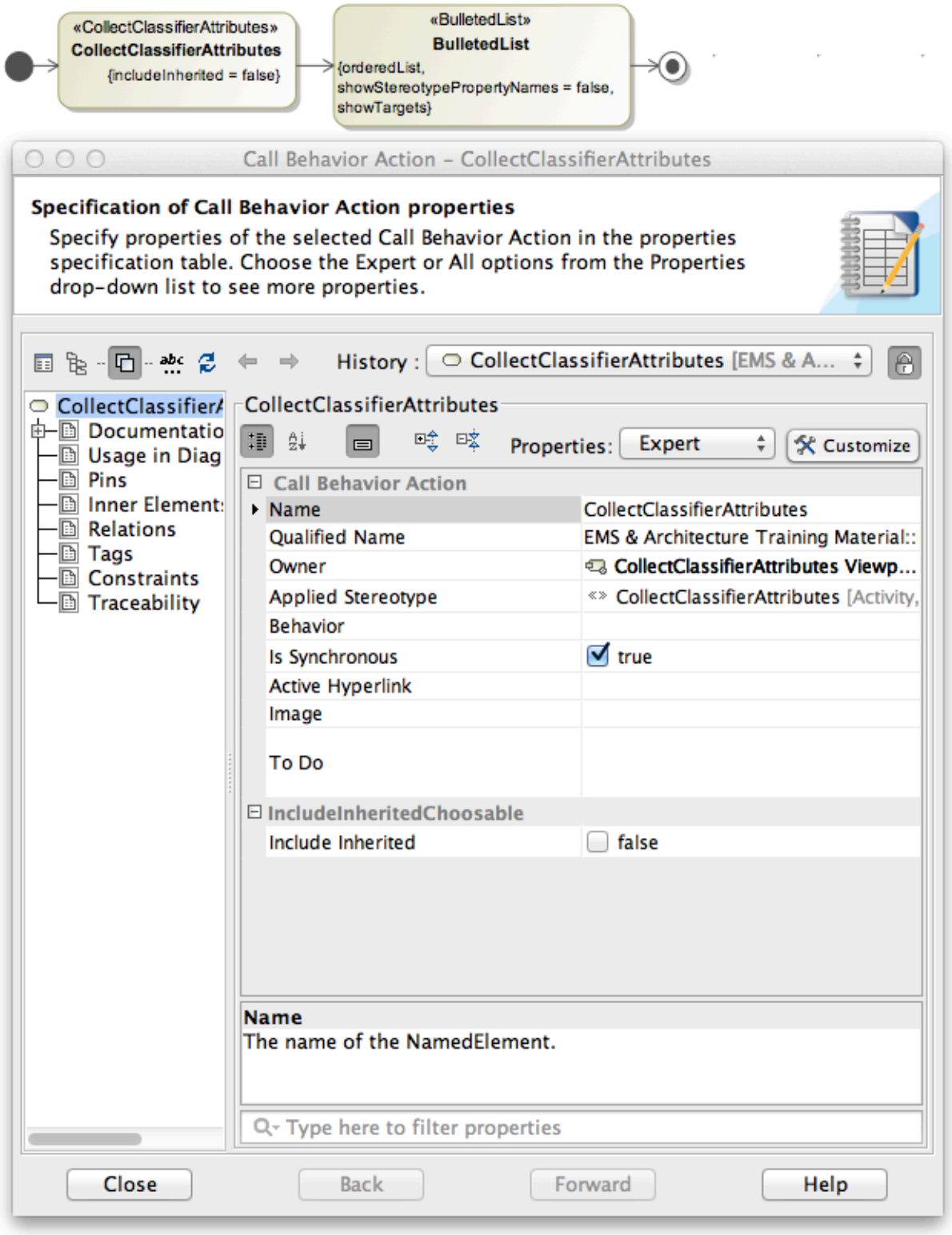
The result of the viewpoint method is as follows.

1. Sixes
2. Threes
3. Eights
4. Twos
5. Saul
6. Tory
7. Anders
8. Chief
9. Ellen
10. Fours
11. Ones
12. Fives
13. Sevens
14. Six Look-alike
15. Baltar Look-alike
16. Starbuck
17. Lee
18. Adama
19. Helo
20. Gaeta
21. Zarek
22. Cottle
23. Baltar
24. Billy
25. Roslin

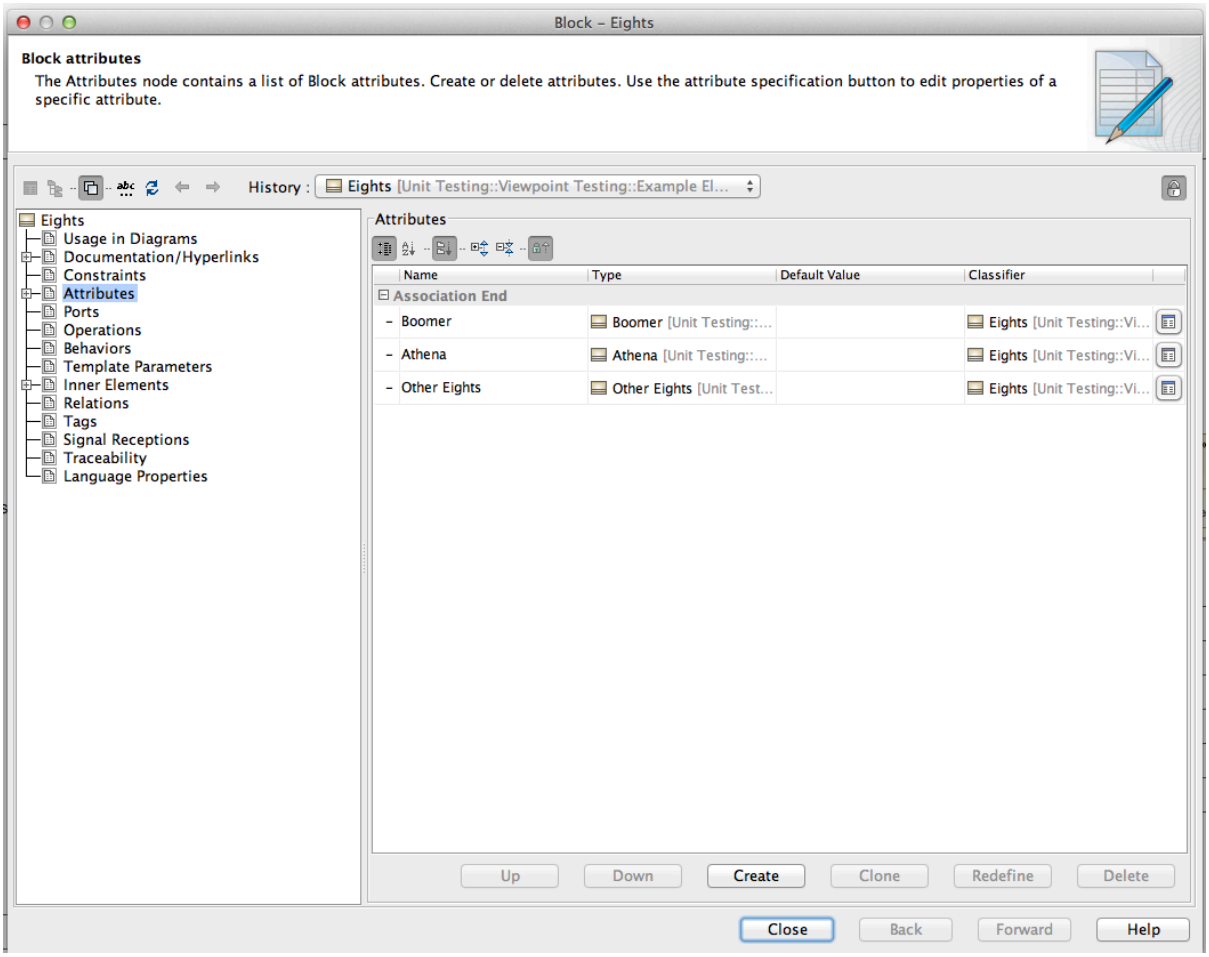
26.Hera
27.Loyalist
28.Sympathizer
29.Boomer
30.Athena
31.Other Eights
32.Final Five
33.Integer
34.Character
35.Element
36.Human
37.Cylon
38.Unknown
39.Association
40.Subset
41.Dependency
42.Mother
43.Father

3.3.2.1.1.9. 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 viewpoint method diagram below shows the CollectClassifierAttributes is used with "Include Inherited" set to false in the specification window. The results are then assembled into a bulleted list.



In the view diagram, the Eights block is exposed. The image below shows the specification window for the "Eights" block. Notice that Boomer, Athena, and Other Eights are its attributes.



The result of the viewpoint method is as follows.

1. Boomer
2. Athena
3. Other Eights

3.3.2.1.1.10. CollectByExpression

CollectByExpression is a more customized approach to querying a model using Object Constraint Language (OCL) . See section 8.3.1.1 below.

3.3.2.1.2. Filter

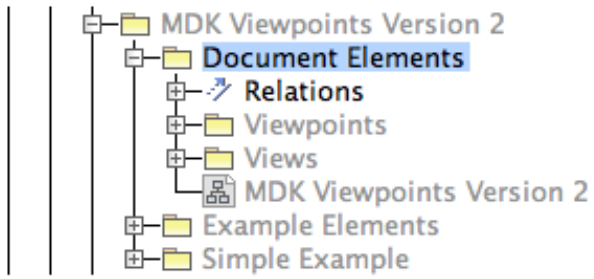
The Filter operations allow the user to narrow fields of data to the data of interest according to one or multiple filter criteria of various types (such as stereotype, name, metaclasses, etc). For most applications, the "FilterBy..." operation needs to have a "CollectBy..." operation preceding it so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

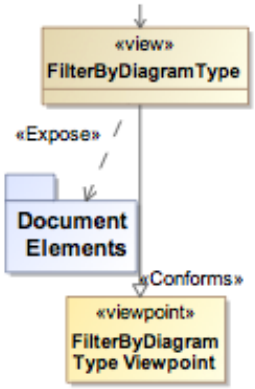
The following sections will take a look at the various Filter operations, what they do, how to set up the operation, and what the output could look like. The examples to follow are simpler than you would likely use with a real project, and they are meant to be simple to explain the basic principles that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

3.3.2.1.2.1. FilterByDiagramType

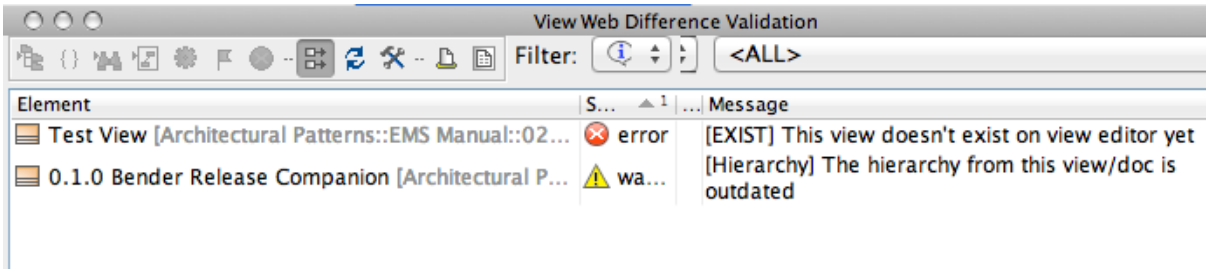
The FilterByDiagramType activity goes through a data set and looks at the elements which are diagrams. The user can decide which type of diagram is of interest to them, and display the names of only those diagrams in their document.

We will illustrate this operation using the model where this document and its elements reside. We know that within the package (called "Document Elements) are many viewpoint method diagrams that are used to display the examples so that is the package that is exposed. The pictures below show the containment tree (left) with the Document Elements package and the diagram (right) where the view for this section is linked to the FilterByDiagramType Viewpoint.





For this viewpoint method diagram, a collect action does need to be used first. In this case, we chose "CollectOwnedElements" and set the depth to 0. Since "Document Elements" is the exposed package, this activity will collect everything owned by "Document Elements." The viewpoint method diagram with the "FilterByDiagramType" block onto the diagram, as shown in the picture below.



In the specification for "FilterByDiagramType," go to the "DiagramTypeChoosable" section and select the viewpoint method diagram via the drop-down menu. As you can see, there are many diagram types that a user could filter by to suit their needs. It is also possible to filter by more than one diagram type (by clicking the "+" button), if necessary. Once this is complete, make sure that include is checked under the "IncludeChoosable" section. That specifies that we would like to include only the viewpoint method diagrams in our output, instead of everything but the viewpoint method diagrams.

The output of this viewpoint method is shown in the list below:

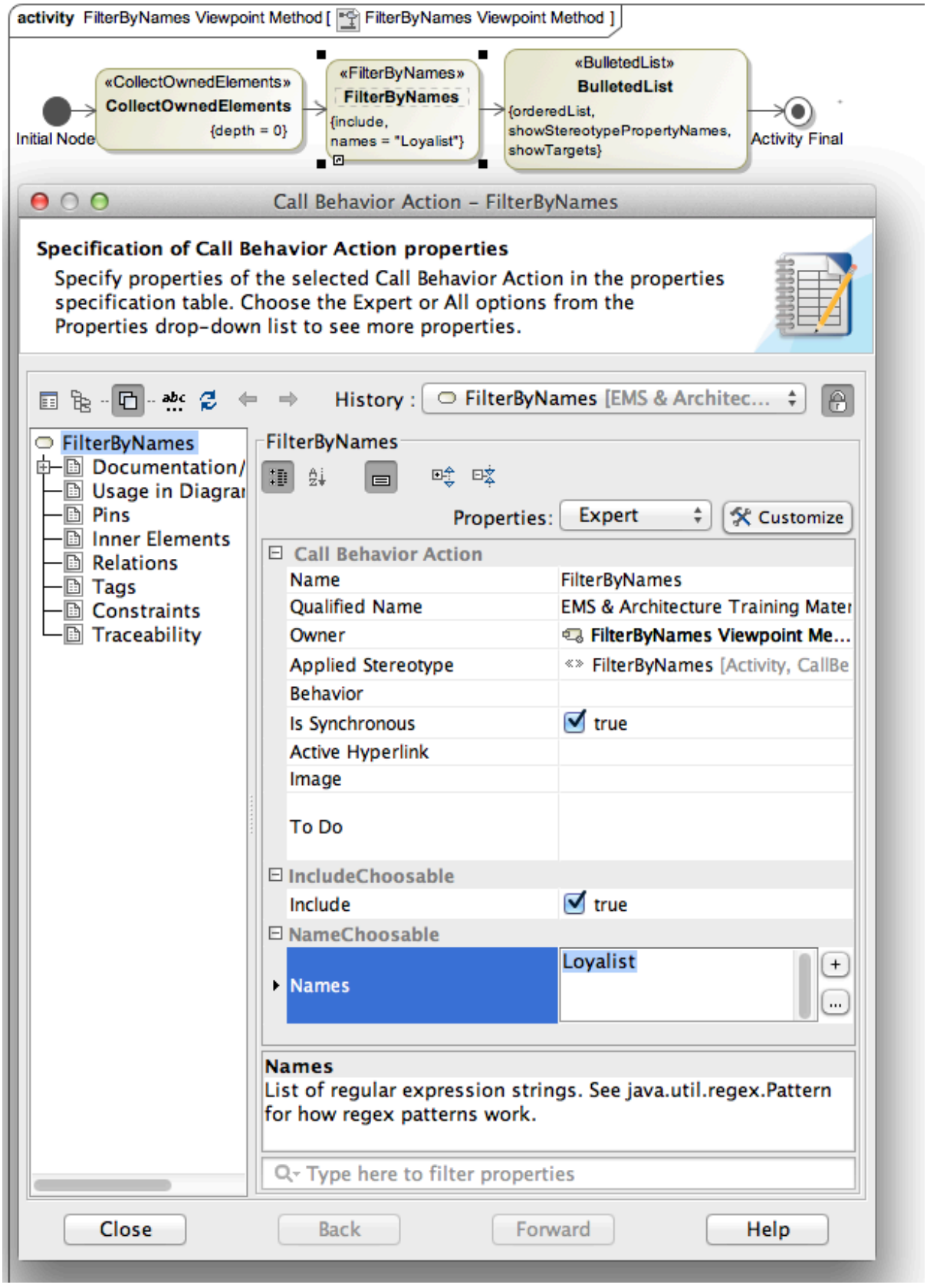
- Validation UserScripts
- Base Manual Viewpoint
- CollectOwnedElements Viewpoint Method
- CollectTypes Viewpoint Method
- SortByAttribute Viewpoint Method
- Multiple Sections
- CollectByStereotypeProperties
- CollectClassifierAttributes Viewpoint Method

- OrderedList Viewpoint
- SortByProperty Viewpoint Method Diagram
- FilterByDiagramType Viewpoint Method
- CollectOwners Viewpoint Method
- Complex Table Viewpoint
- Dynamic Section
- FilterByStereotypes Viewpoint Method
- CollectByAssociation Viewpoint Method
- CollectByDirectedRelationshipStereotypes Viewpoint Method
- Image Viewpoint Method
- CollectByDirectedRelationshipMetaclasses Viewpoint Method Two
- CollectByExpression Viewpoint Method
- Simple Table Viewpoint Method
- CollectByDirectedRelationshipMetaclasses Viewpoint Method
- CollectThingsOnDiagram Viewpoint Method
- FilterByMetaclasses Viewpoint Method
- FilterByNames Viewpoint Method
- Paragraph Viewpoint
- text in body/text in documentation
- Use Case 1
- Use Case 2
- Document Method
- Use Case 3
- Document Method
- Use Case 4
- Document Method
- Use Case 5
- Document Method
- Use Case 6
- Document Method
- Use Case 7
- Document Method
- Use Case 8
- Document Method
- Use Case 9
- Document Method
- Use Case 10
- Document Method
- Use Case 11
- Document Method
- Main Paragraph
- Paragraph 1
- Paragraph 2
- Document Method
- CollectTable
- Collect Document Diagrams
- Document Method bad
- Car Example Table

3.3.2.1.2.2. 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 Battlestar Galactica example that was baselined above. Specifically, we want to find how many characters are loyalists.

We first create a FilterByNames Viewpoint Method diagram. This diagram is set to expose the "Example Elements" package which stores the Battlestar Galactica example's data.



In the "FilterByNames" specification window, we need to enter the name we want to filter by under the "NamesChoosable" section. We are able to specify multiple names if necessary using the "+" button. In this case,

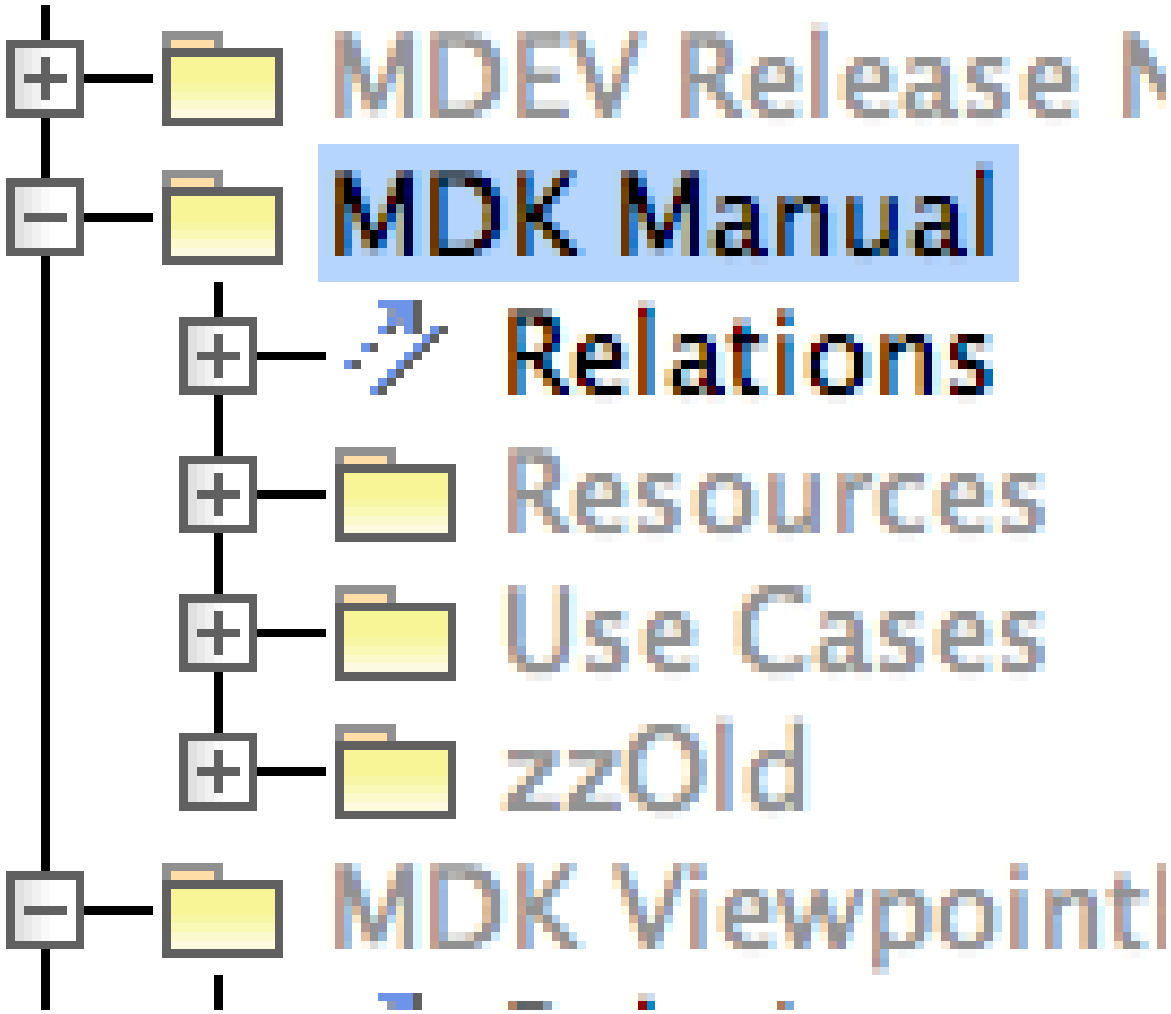
we enter the name "Loyalist" to accomplish our desire to find out how many characters are loyalists. Note, we know there is a block named "Loyalist," and this block is a shared property of four cylons. Refer to the Characters BDD at the beginning of section 5.2. Also note, this is a simple example, and this function would likely be more valuable in the case where there are many more than four components (or loyalists in this case). Again, ensure that "Include" is checked. The final diagram is shown in the picture above.

The output from this operation is below. Note, we expected to get four loyalists, not five. However, since there are four loyalists and the loyalist block itself, the operation returns all five. Additionally, the output does not tell us who the loyalists are, just the number (and incorrectly). The reason for this is that the "FilterByNames" operation is not enough to do everything we need. The "FilterByNames" operation needs to be combined with other Filters, and a table is a better way to convey all the information. This example is refined in the Simple Table section below to illustrate the more desirable, and useful, output.

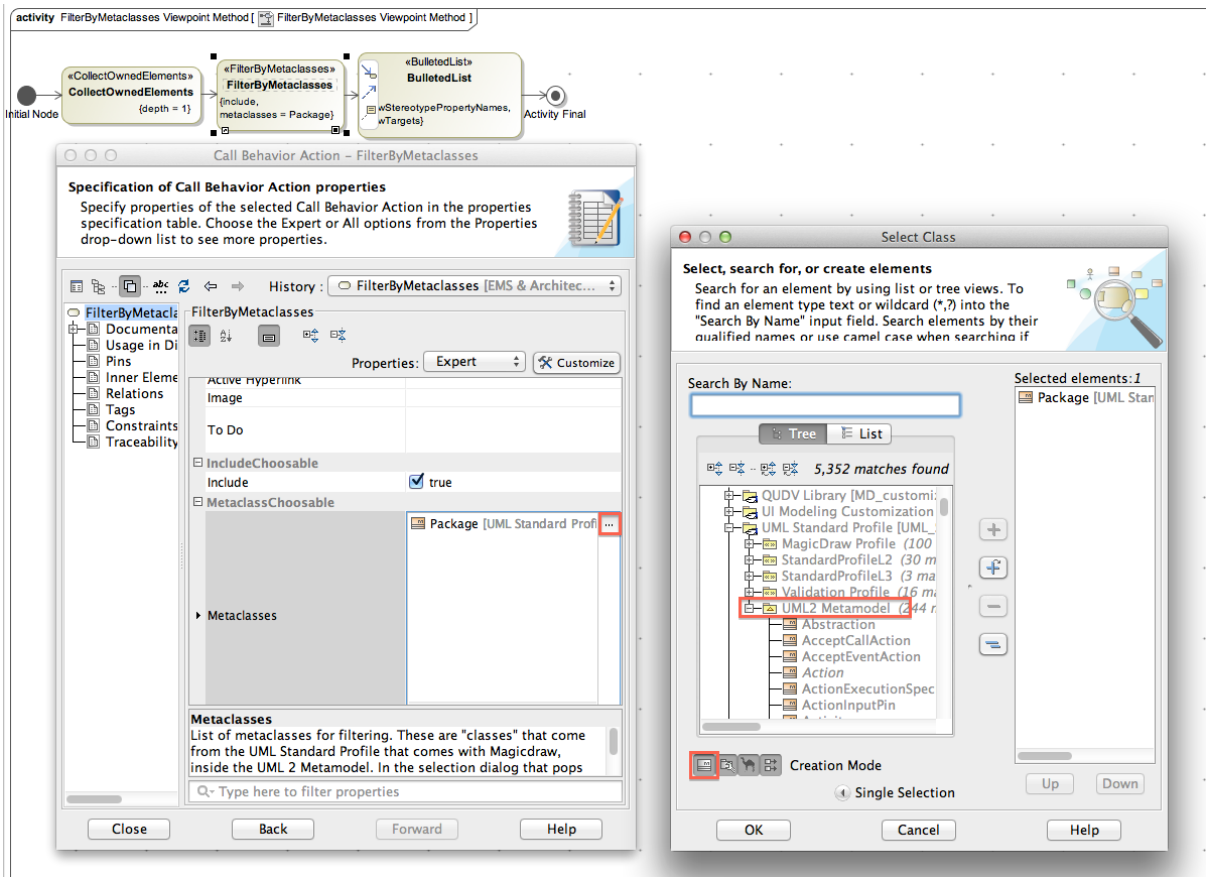
1. Loyalist
2. Loyalist
3. Loyalist
4. Loyalist
5. Loyalist

3.3.2.1.2.3. 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 package "MDK Manual," but only with a depth of 1. The FilterByMetaclasses operation is then used to show the packages (at the first level) under the "MDK Manual" package.



The completed viewpoint method diagram for this operation is shown below.



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 (circled in red in the picture above). A "Select Class" window will appear. In this window, make sure the small box in the bottom left hand corner (circled in red in the picture above) is selected to allow metaclasses to be shown in the selection box. Now, we can navigate to "UML Standard Profile" -> "UML2 Metamodel" -> select "Package" -> and click the "+" button. We could also have chosen the "Stereotype" metaclass, for example, or a number of other metaclasses.

The result of the viewpoint method with the FilterByMetaclasses operation is shown below, and the output clearly displays the three packages directly (level 1) underneath the "MDK Manual" package. Note, we used an un-ordered "BulletedList" to display the output.

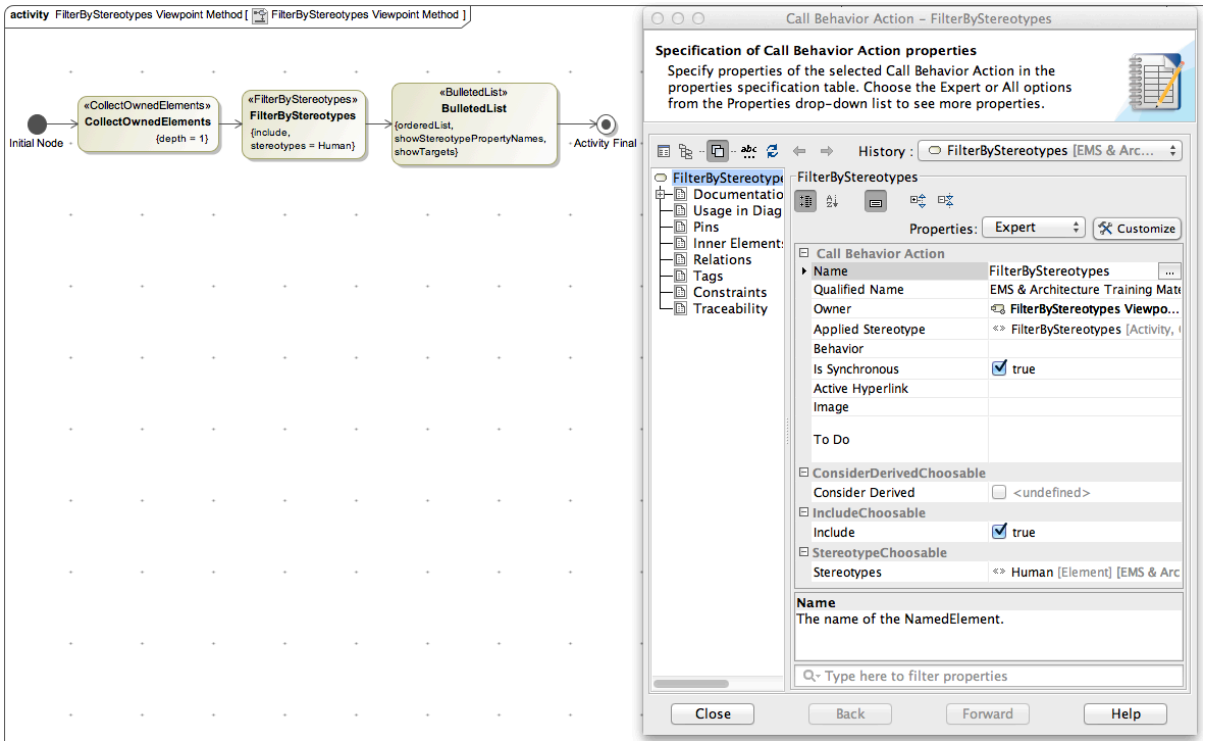
- l Resources
- MDEV Release Notes
- EMS Applications Manuals
- MDK Viewpoints Version 2
- Testing Document
- EMS Application Unit Tests
- Comment Migration
- View Heirarchy
- Viewpoints

3.3.2.1.2.4. FilterByStereotypes

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

In the example illustrated below, the CollectOwnedElements is utilized to collect the elements owned by the package "Example Elements." Once the elements have been collected, they are filtered to show only the elements stereotyped "Human," so that we can find out who the human characters are in Battlestar Galactica.

The finished viewpoint method diagram for "FilterByStereotypes" is shown below. In the specification of FilterByStereotypes , under Stereotypes Choosable, the stereotype Human is chosen. Then, under "IncludeChoosable," Include is set to true to show only the Humans in the bulleted list. We could also filter by the stereotypes Cylon and Unknown if desired. In addition, it is also possible to filter by more than one stereotype.



The bulleted list output from the FilterByStereotypes Viewpoint Method shows us who the humans are. Note, this list is not alphabetized or otherwise ordered in any meaningful way. An additional operation, a Sort operation, would be required if the user needs to alphabetize the output, for example.

1. Starbuck
2. Boomer
3. Adama
4. Lee
5. Helo
6. Gaeta
7. Roslin
8. Baltar
9. Zarek
10. Billy
11. Cottle
12. Hera

3.3.2.1.2.5. FilterByExpression

FilterByExpression is a more customized approach to querying a model using Object Constraint Language (OCL) . See section 8.3.1.2 below.

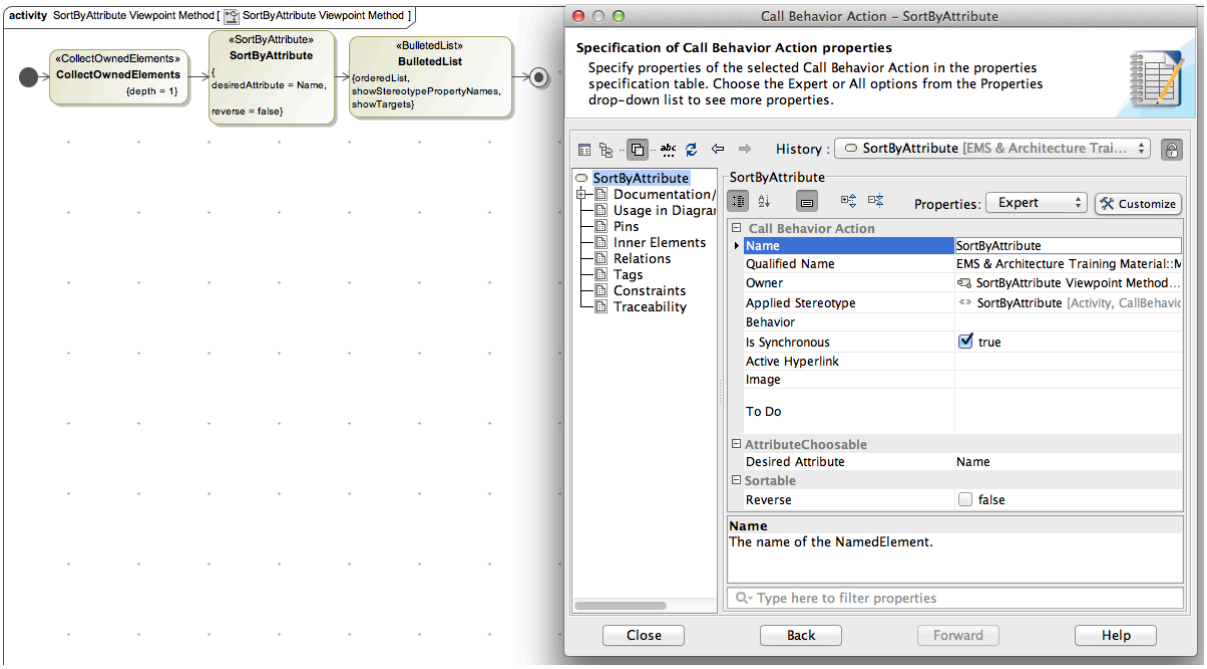
3.3.2.1.3. Sort

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.

3.3.2.1.3.1. 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. Thus, in this example, we will sort by name, and in this case we will sort the Eights from Battlestar Galactica.

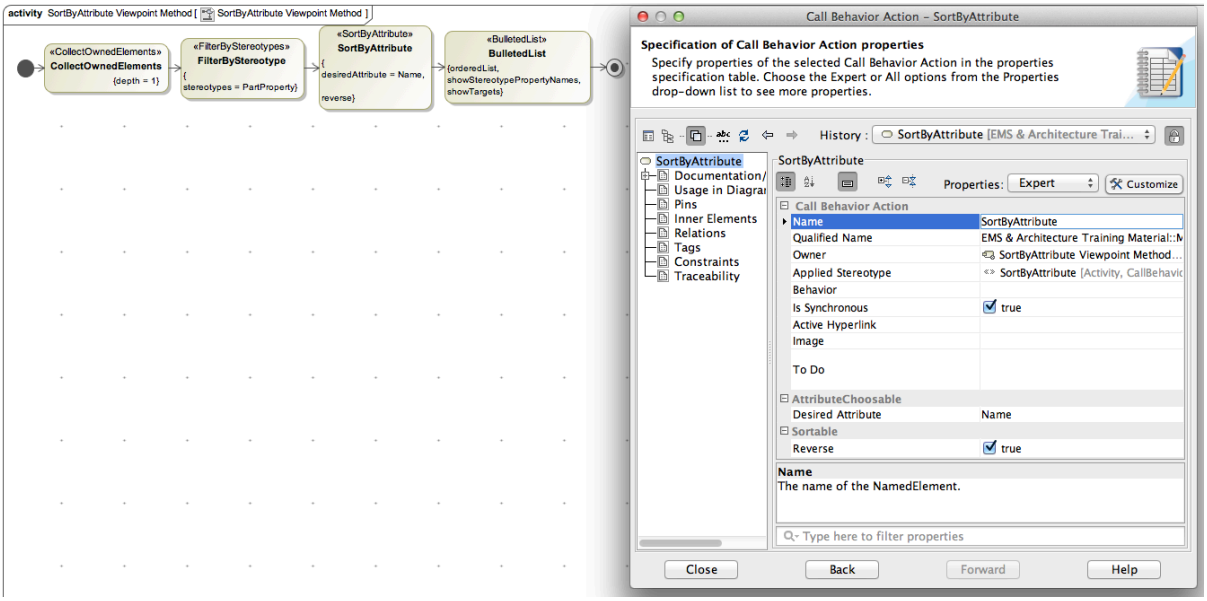
The finished "SortByAttribute Viewpoint Method" diagram is shown below.



In the specification window for "SortByAttribute," we choose the "Name" option from the drop-down menu under the "AttributeChoosable" section. Ensure "Reverse" is false. The output from the viewpoint method is shown below as an image to demonstrate that the first result is "reference missing," which is an artifact of Magic Draw.

1. reference missing
2. Athena
3. Boomer
4. Other Eights

To filter these artifacts, we need to use another filter. In this case, we use the FilterByStereotype operation with the PartProperty stereotype. This will filter all the elements collected by the CollectOwnedElements to the elements which are part properties (which includes our characters) before then being sorted by name. The viewpoint method with the additional filter is shown below. To test the reversibility function of the sort, we have checked reverse to be true in the specification window of SortByAttribute.



As before, we have exposed the Eights block, and the result of the modified viewpoint method is shown below. The Magic Draw artifact is no longer present, and the results are in reverse alphabetical order, as expected.

1. Boomer
2. Athena
3. Other Eights

3.3.2.1.3.2. SortByProperty

This operator currently does not work. The issue is expected to be resolved in a future version.

3.3.2.1.3.3. SortByExpression

CollectByExpression is a more customized approach to querying a model using Object Constraint Language (OCL) . See section 8.3.1.3 below.

3.3.2.2. 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.

3.3.2.2.1. Table

The sections below go over how to create tables. Tables of varying complexity can be created in Magic Draw, 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.

3.3.2.2.1.1. Simple Table

The viewpoint method below is used to create a simple table that will display the Battlestar Galactica characters that are loyalists.

First, CollectOwnedElements is used to collect all elements within the Example Elements package. Then, the FilterByNames operation is used to filter out all but the elements that have the name Loyalist, which would be the four shared properties and one block as illustrated by FilterByNames above. Then, since we want to know the names of the characters that have Loyalist as a shared property, we collect the owners of those five results from FilterByNames. However, as in FilterByNames , one of the filtered results is actually the Loyalist block and has the owner Example Elements (package), and we don't want to include that result. Thus, FilterByMetaclasses is used to filter out the Example Elements package that was collected as an owner of the Loyalist block.

Now, we begin the table with TableStructure. Within the table structure, we start a flow to create the table content with an initial node. Then, the TableAttributeColumn (no dotted line) is placed into the flow. This column is named "Character," which becomes the column header, with desiredAttribute in the specification window selected to be Name. This means the name of the previously collected owners of the loyalist shared property will be displayed in the first column.

Next, we want to show these characters are loyalists, however, this requires another flow to now collect their shared property. Thus, we use the TableAttributeColumn with the dotted line, and start another flow with the initial node. Then, we use CollectByAssociation to collect the shared association of the characters and end with the flow final. Within the specification for this second TableAttributeColumn, we input "Shared Property" as the name, which again will become the column header, and choose Name as desiredAttribute. This will display the name of the shared association. Finally, we end the flow within the TableStructure with a flow final, and then end the activity with an activity final.

This below image shows the completed viewpoint method diagram and the resulting table shows the four characters that are loyalists.

Figure 3.7. Simple Table Viewpoint Method

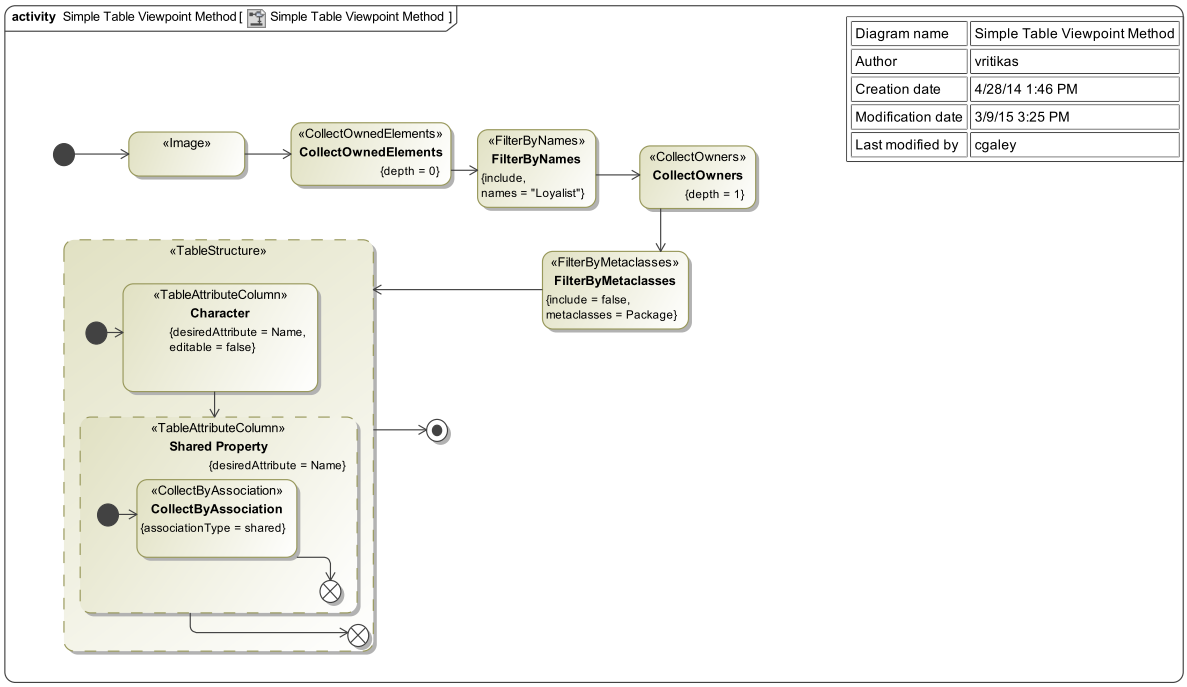


Table 3.4.

Character	Shared Property
Ones	Loyalist
Fours	Loyalist
Fives	Loyalist
Boomer	Loyalist

3.3.2.2.1.2. Complex Table

The above example shows the basics of tables in the model. However, different tools allow for more complexity. In the example below, the class elements of "Example Elements" are put into a table. Note that the generic "Table Structure" is used in the viewpoint. This allows more flexibility in designing the table. "TableColumnGroup" is used to create subcolumns. As can be seen in the example below, the name of the "TableColumnGroup" is the overall header, while the name of each column is the individual column header. While simpler columns such as "TableAttributeColumn" can be used, more complex results can be extracted from the model using "TableExpressionColumn". In the "ExpressionChooseable" portion of its specification, you can enter a code for the type of information you want. For example, "owner" returns the name of the element's owner, and "s()" returns all of the element's stereotypes. Note that each flow has it's own initial node and flow final, except for the overall activity, which has an activity final.

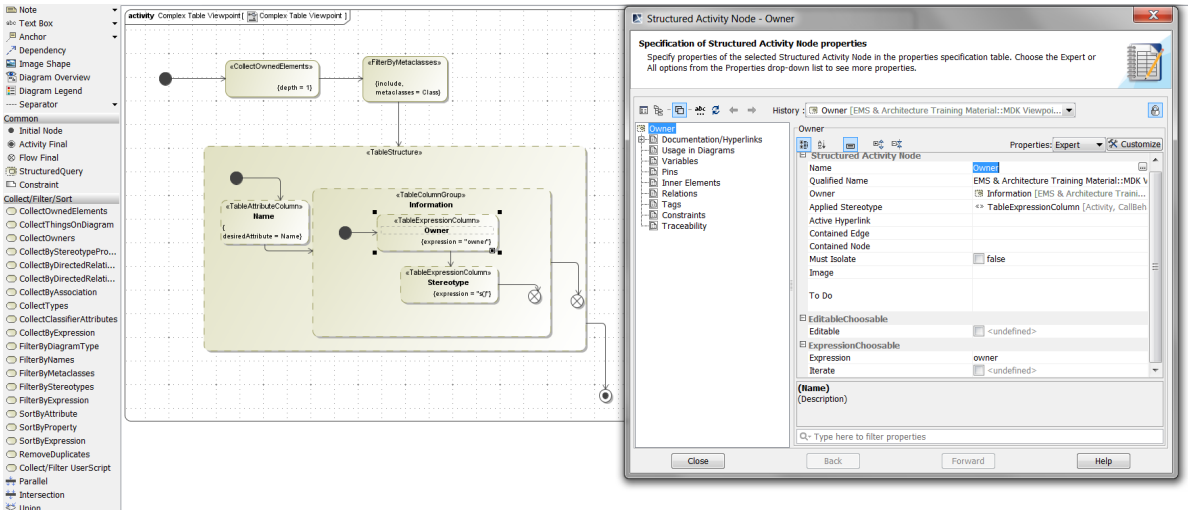


Table 3.5.

Name	Information	
	Owner	Stereotype
Character	Example Elements	Block
Ones	Example Elements	Cylon Block
Twos	Example Elements	Cylon Block
Threes	Example Elements	Cylon Block
Fours	Example Elements	Cylon Block
Fives	Example Elements	Cylon Block
Sixes	Example Elements	Cylon Block
Sevens	Example Elements	Cylon Block
Eights	Example Elements	Cylon Block
Anders	Example Elements	Cylon Block
Chief	Example Elements	Cylon Block
Saul	Example Elements	Cylon Block

Name	Information	
	Owner	**Stereotype**
Ellen	Example Elements	Cylon Block
Tory	Example Elements	Cylon Block
Starbuck	Example Elements	Unknown Block Human
Six Look-alike	Example Elements	Unknown Block
Baltar Look-alike	Example Elements	Unknown Block
Boomer	Example Elements	Cylon Block Human
Athena	Example Elements	Cylon Block
Other Eights	Example Elements	Cylon Block
Adama	Example Elements	Human Block
Lee	Example Elements	Human Block
Helo	Example Elements	Human Block
Gaeta	Example Elements	Human Block
Roslin	Example Elements	Human Block
Baltar	Example Elements	Human Block
Zarek	Example Elements	Human Block
Billy	Example Elements	Human Block

Name	Information	
	Owner	Stereotype
Cottle	Example Elements	Human Block
Human	Example Elements	
Cylon	Example Elements	
Unknown	Example Elements	
Hera	Example Elements	Cylon Human Block
Loyalist	Example Elements	Block
Sympathizer	Example Elements	Block
Final Five	Example Elements	Block
Subset	Example Elements	
Mother	Example Elements	
Father	Example Elements	

3.3.2.2.2. 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.

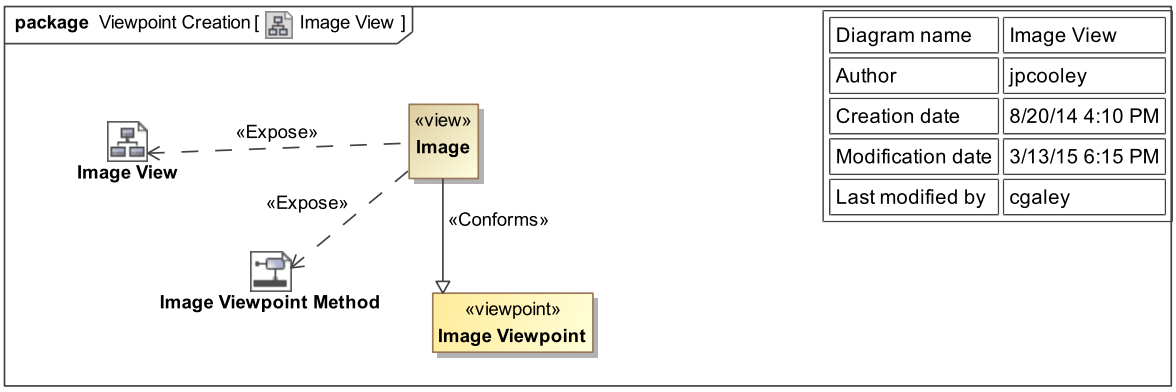
This image will then be updated when the model is updated without need for a new image to be put into the document to reflect changes. It is also possible to add captions and titles to the image, however, these functionalities are not currently working.

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

Figure 3.8. Image Viewpoint Method

This is the Viewpoint Method Diagram used to demonstrate the <<Image>> action.

Figure 3.9. Image View



Above is the View Diagram that was used to demonstrate the <<Image>> action.

3.3.2.2.3. Paragraph

NOTE: Several of the sections below require some knowledge of OCL (Object Constraint Language). If this is new to you please refer to section 8 below.

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.

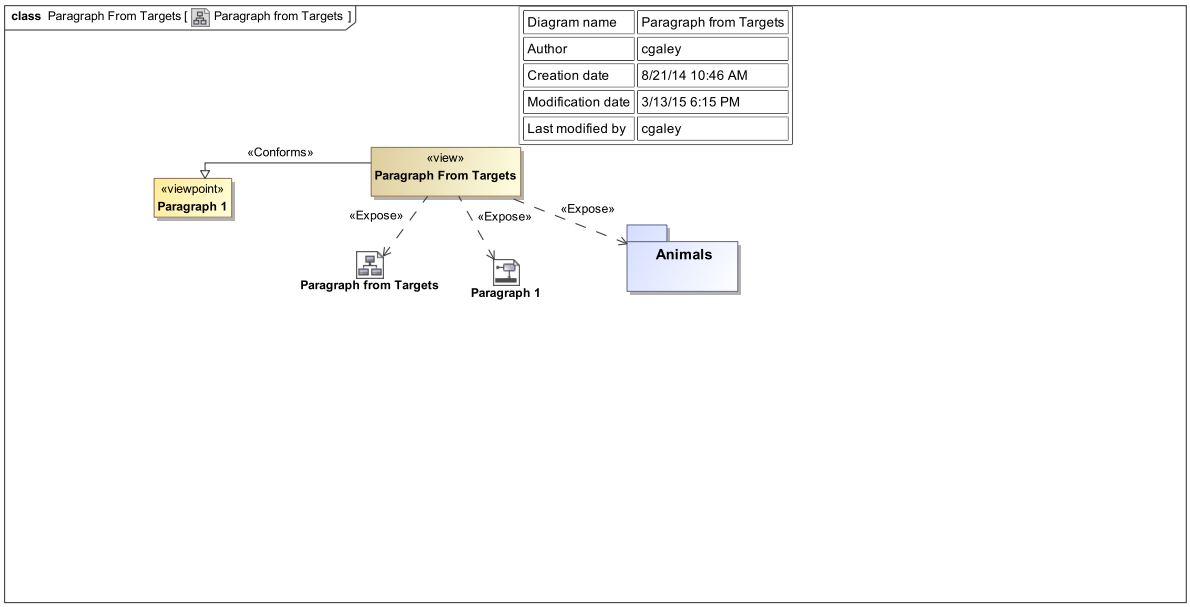
3.3.2.2.3.1. Paragraph From Targets

Basic paragraphs can be created by exposing a package full of elements. In this case a package is exposed that contains a group of 5 animals:

- Cat
- Cow
- Dog
- Duck
- Frog

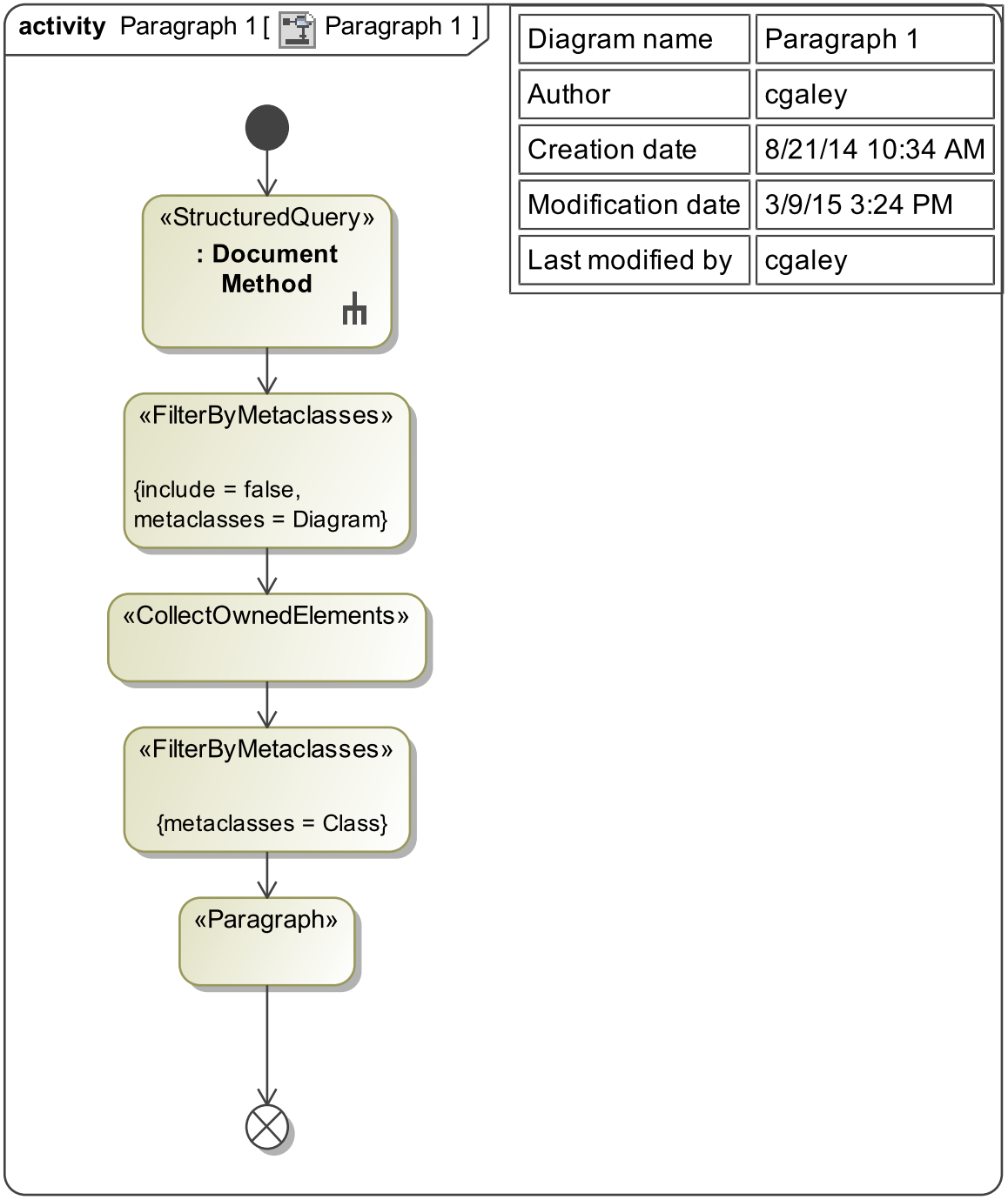
By clicking on the cross references in each bullet you can see what the documentation is for each animal. It is also displayed in the bottom which is the output of the Paragraph 1 viewpoint.

Figure 3.10. Paragraph from Targets



This is the view diagram used to create the Paragraph From Targets section.

Figure 3.11. Paragraph 1



This is the viewpoint method diagram used to create the Paragraph From Targets section. Note that the contents of the <<Paragraph>> action are blank. Because the <<Paragraph>> action is blank, the viewpoint method created individual paragraphs containing the documentation of each exposed element. Thus below you can see 5 separate paragraphs corresponding to the 5 animal elements exposed in the Animals package.

Below this line is the output of the <<Paragraph>> activity:

A Cat goes meow. 'Cat Documentation: '.concat(name)

A Duck goes quack 'Duck Documentation: '.concat(name)

A Cow goes MOOOOOO. 'Cow Documentation: '.concat(name)

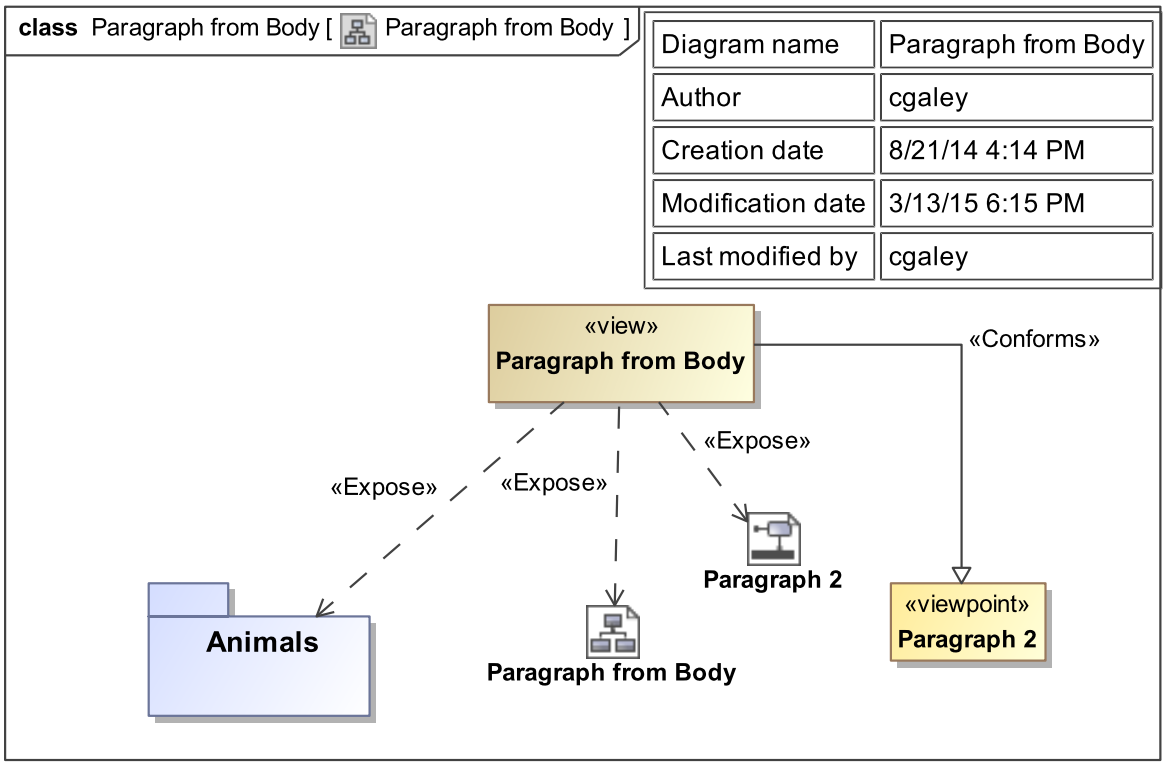
A Frog goes Ribbit. 'Frog Documentation: '.concat(name)

A Dog goes Woof. 'Dog Documentation: '.concat(name)

3.3.2.2.3.2. Paragraph from Body

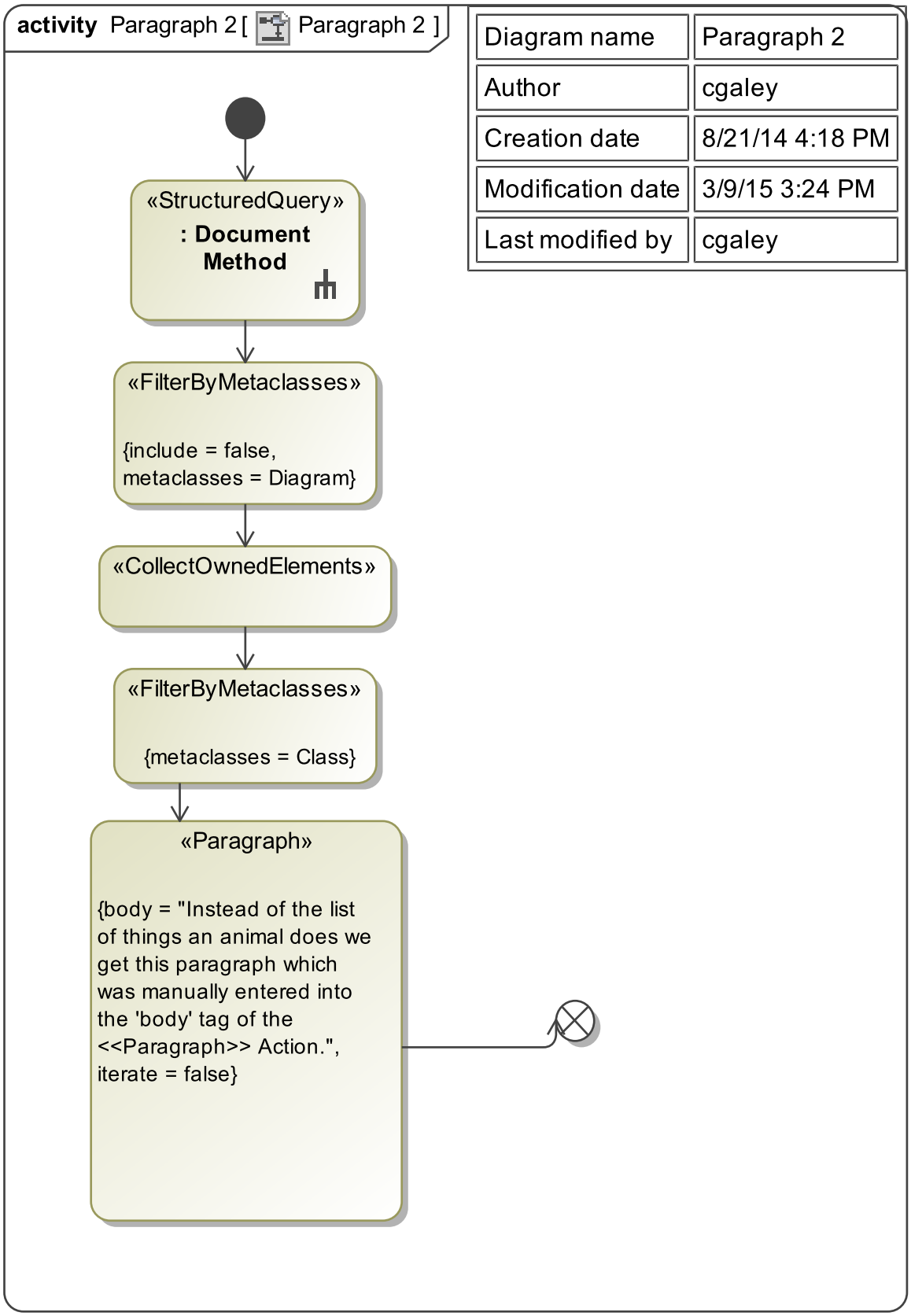
Using the <<Paragraph>> action allows you to better control where content appears in your document. For example in some situations you may want to use a Diagram in multiple views. Following the previous text entry methods we have described, additional text after the image would be placed in the documentation of the Diagram. If the diagram is used in multiple places that documentation will appear in the other locations as well. Using the <<Paragraph>> action you can create a specific paragraph for each instance through the use of the "body" tag and its location through the activity flow.

Figure 3.12. Paragraph from Body



The View Diagram shown above is very similar to the example for Paragraph From Targets. The only difference is a new viewpoint method. The Viewpoint Method is also very similar to the Paragraph From Targets except that the paragraph has information in the body tag. So instead of creating the same set of paragraphs from targets (the animals) the action will insert the text specified by the "body". Essentially text in the "body" tag causes the <<Paragraph>> to ignore any targets and only insert its value (nothing is changed about the information passed on for subsequent actions).

Figure 3.13. Paragraph 2



From the previous example we would expect to see a paragraph associated with each animal's documentation. However due to the text entered in the "body" tag this is suppressed and the value of the "body" is displayed instead.

Instead of the list of things an animal does we get this paragraph which was manually entered into the 'body' tag of the < > Action.

3.3.2.2.3.3. Paragraph Target-Property Pair

<<Paragraph>> can be used to display stereotype properties of targets. The expected result should be a paragraph for each stereotype property. Passing in the animal elements and specifying the stereotypeProperties tag of the <<Paragraph>> action, produces the results in the paragraphs shown below.

Figure 3.14. Animals

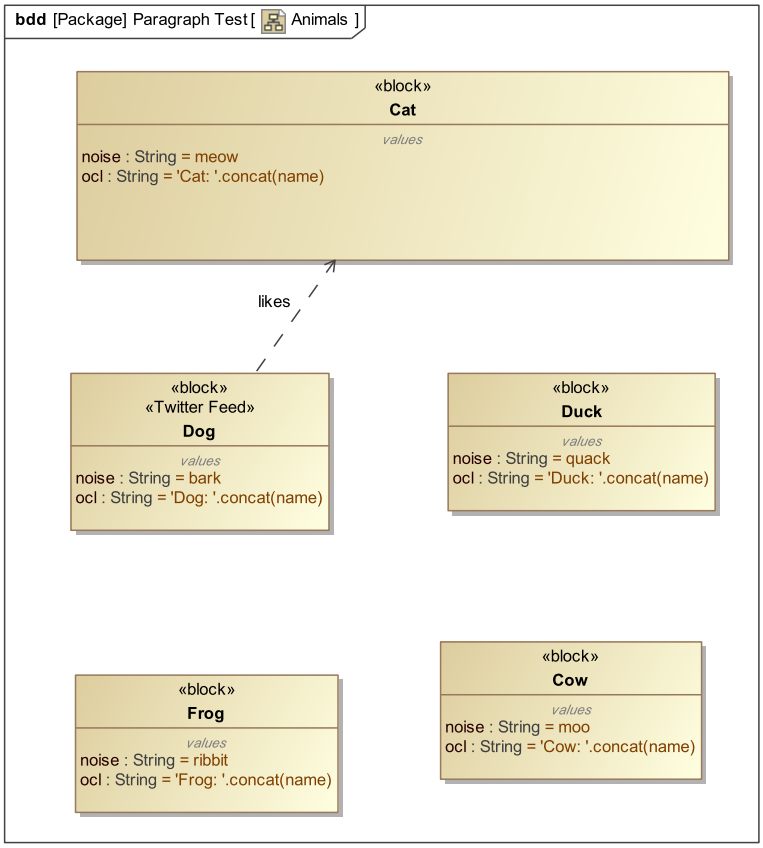
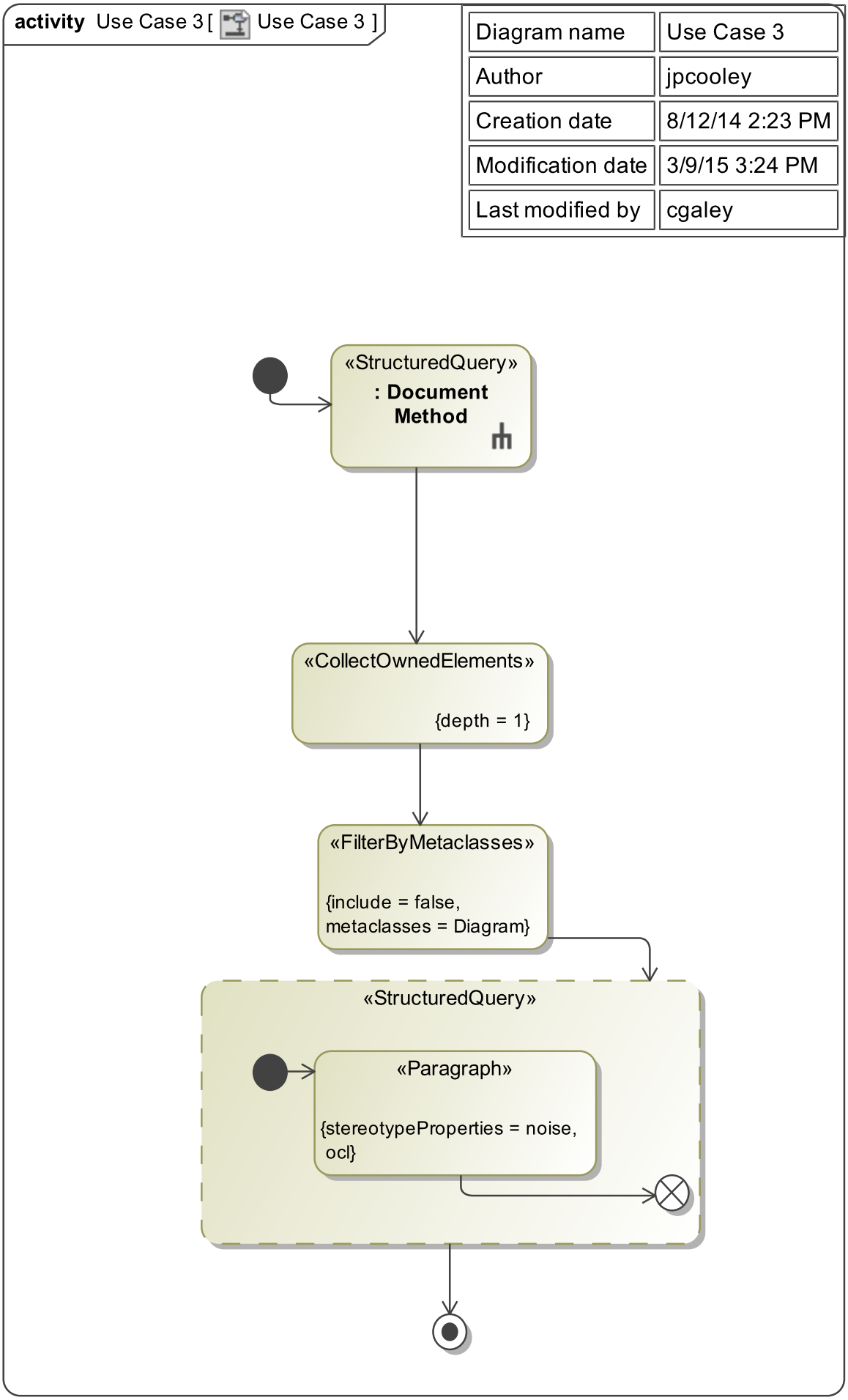


Diagram name	Animals
Author	igomes
Creation date	6/4/14 4:06 PM
Modification date	9/25/14 6:19 PM
Last modified by	cgaley


activity Use Case 3 [Use Case 3]
Diagram name
Use Case 3
Author
jpcooley
Creation date
8/12/14 2:23 PM
Modification date
3/9/15 3:24 PM
Last modified by
cgaley
«StructuredQuery»
: Document
Method
«CollectOwnedElements»
{depth = 1}
«FilterByMetaclasses»
{include = false,
metaclasses = Diagram}
«StructuredQuery»
«Paragraph»
{stereotypeProperties = noise,
ocl}

meow

'Cat: '.concat(name)

quack

'Duck: '.concat(name)

moo

'Cow: '.concat(name)

ribbit

'Frog: '.concat(name)

bark

'Dog: '.concat(name)

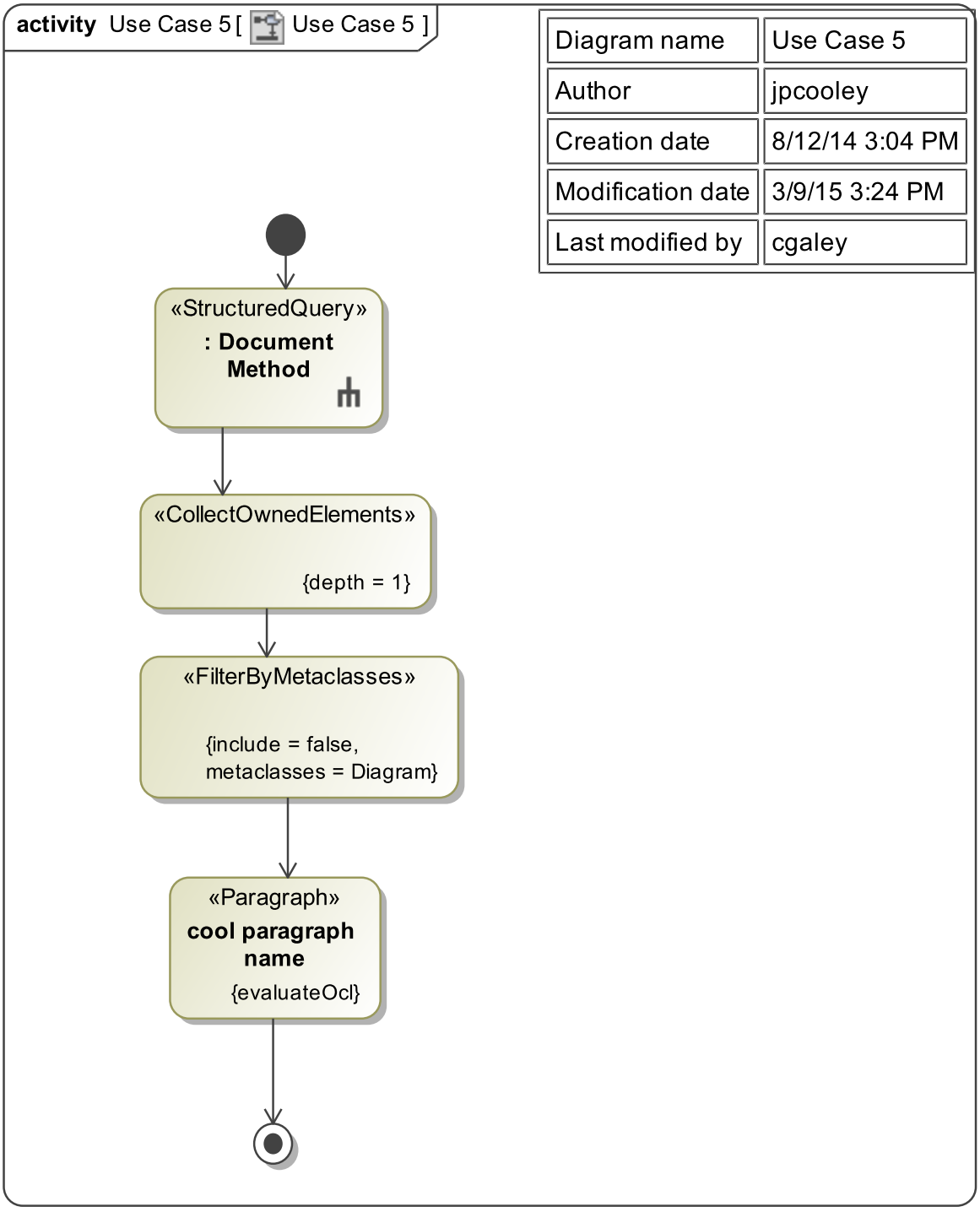
3.3.2.2.3.4. Paragraph Using OCL Only (View Under Construction)

Use Case: Has targets, has no text in body, and uses OCL

Expected: Evaluates OCL expression from the documentation of the targets on the Paragraph template (output varies)

(Not Currently Working)

Figure 3.16. Use Case 5



false

false

3.3.2.2.3.5. (View Under Construction)

Use Case: No targets, has text in body, and uses OCL

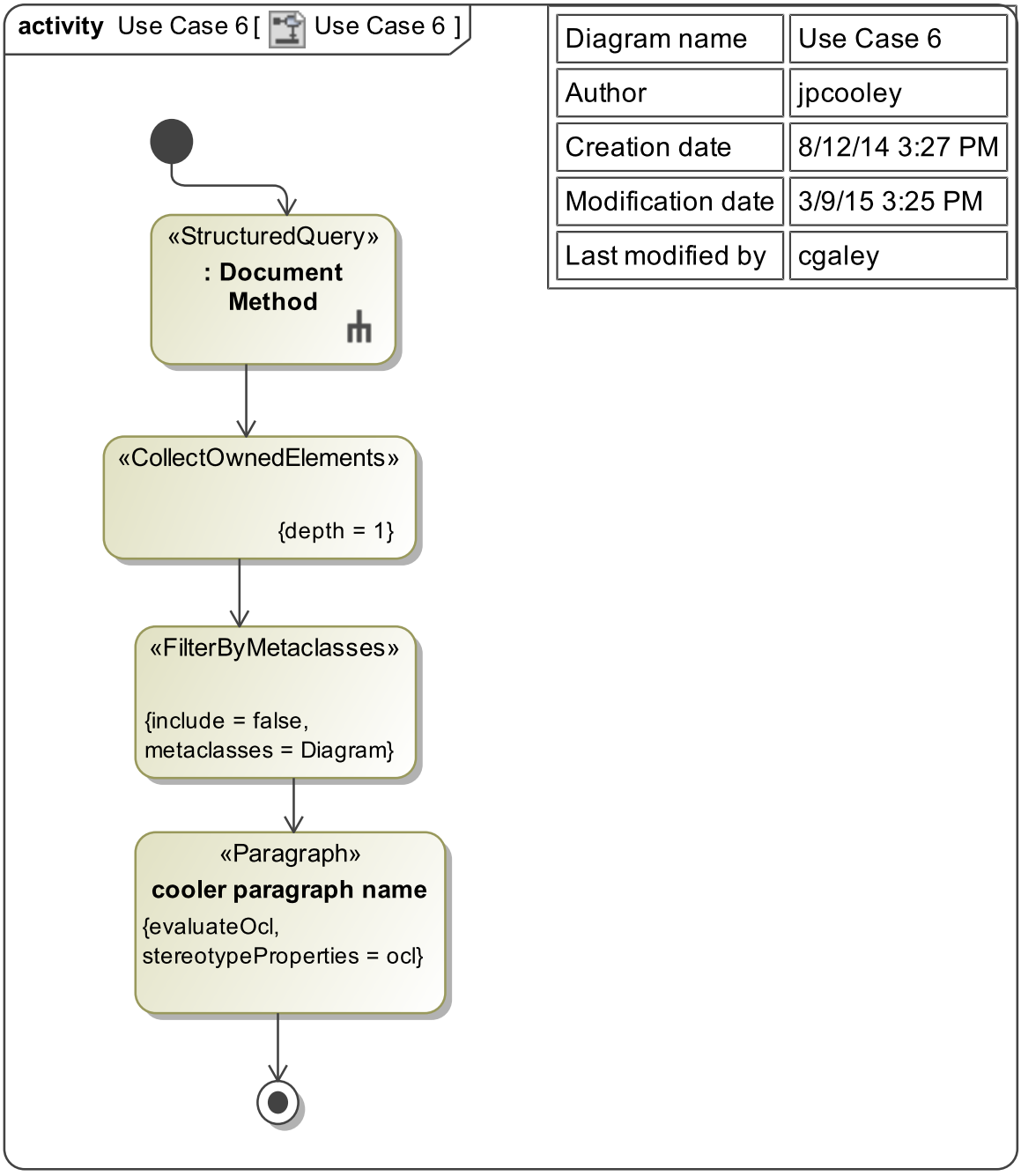
Expected: Evaluates OCL expression from body text with the Paragraph template as the context

(Not Currently Working)

3.3.2.2.3.6. Paragraph from OCL in Stereotype Properties

Currently you are able to build paragraphs using OCL in default value tags of properties. In this examples the Animal package was exposed and the property "ocl" was specified. Looking at the element "Cat" the default value of the property called "ocl" is set to, 'Cat: '.concate(name), we see that this indeed is the first entry of the listed paragraphs below.

Figure 3.17. Use Case 6



Cat: cooler paragraph name

Duck: cooler paragraph name

Cow: cooler paragraph name

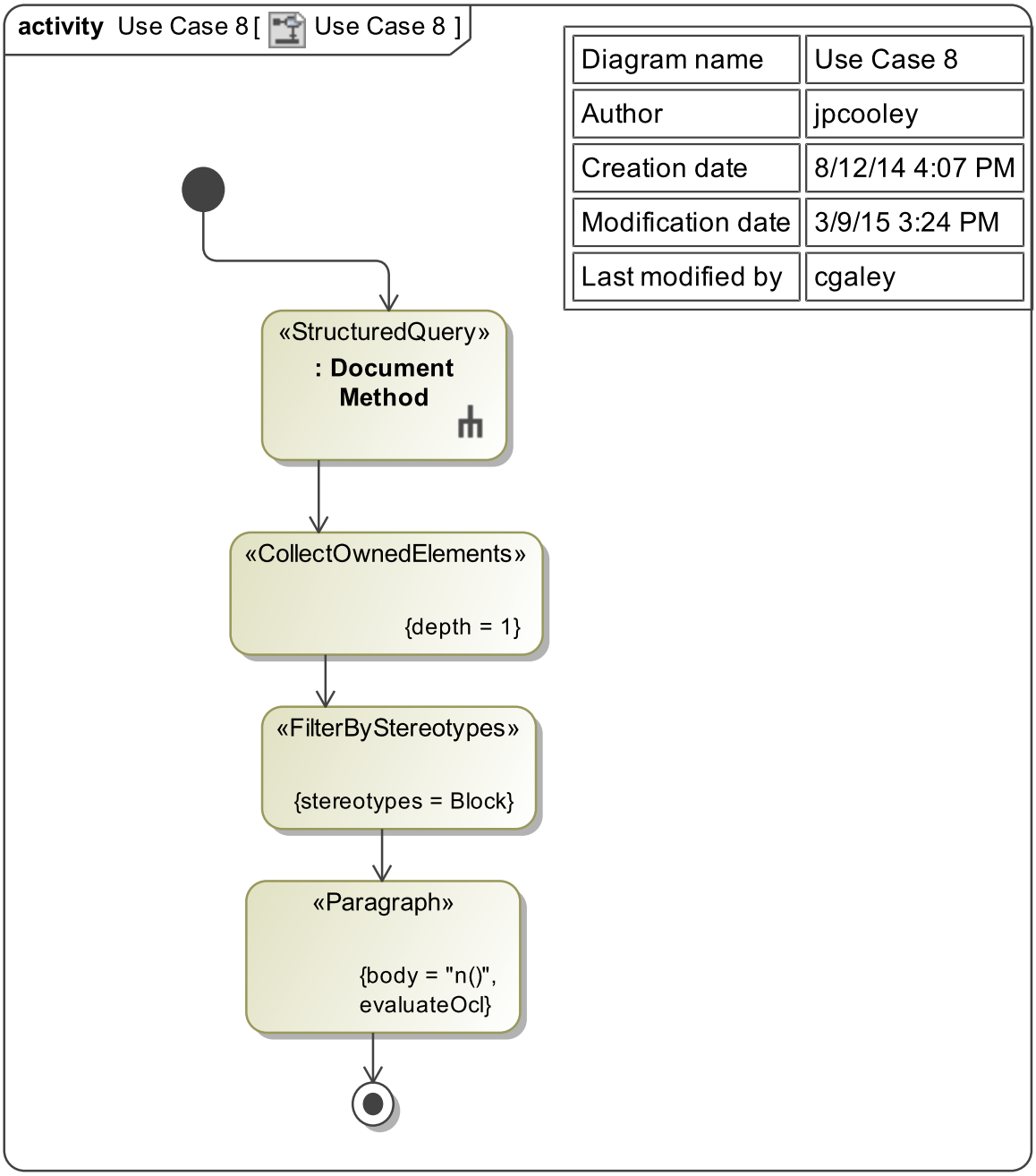
Frog: cooler paragraph name

Dog: cooler paragraph name

3.3.2.2.3.7. Paragraph from OCL in Body

It is possible to build paragraphs based on an OCL expression within the body tag of <<paragraph>>. Shown in this example, the Animal elements are collected and then their name is collected because of the OCL expression, n(), shown in the body. expected results shown below.

Figure 3.18. Use Case 8



Cat

Duck

Cow

Frog

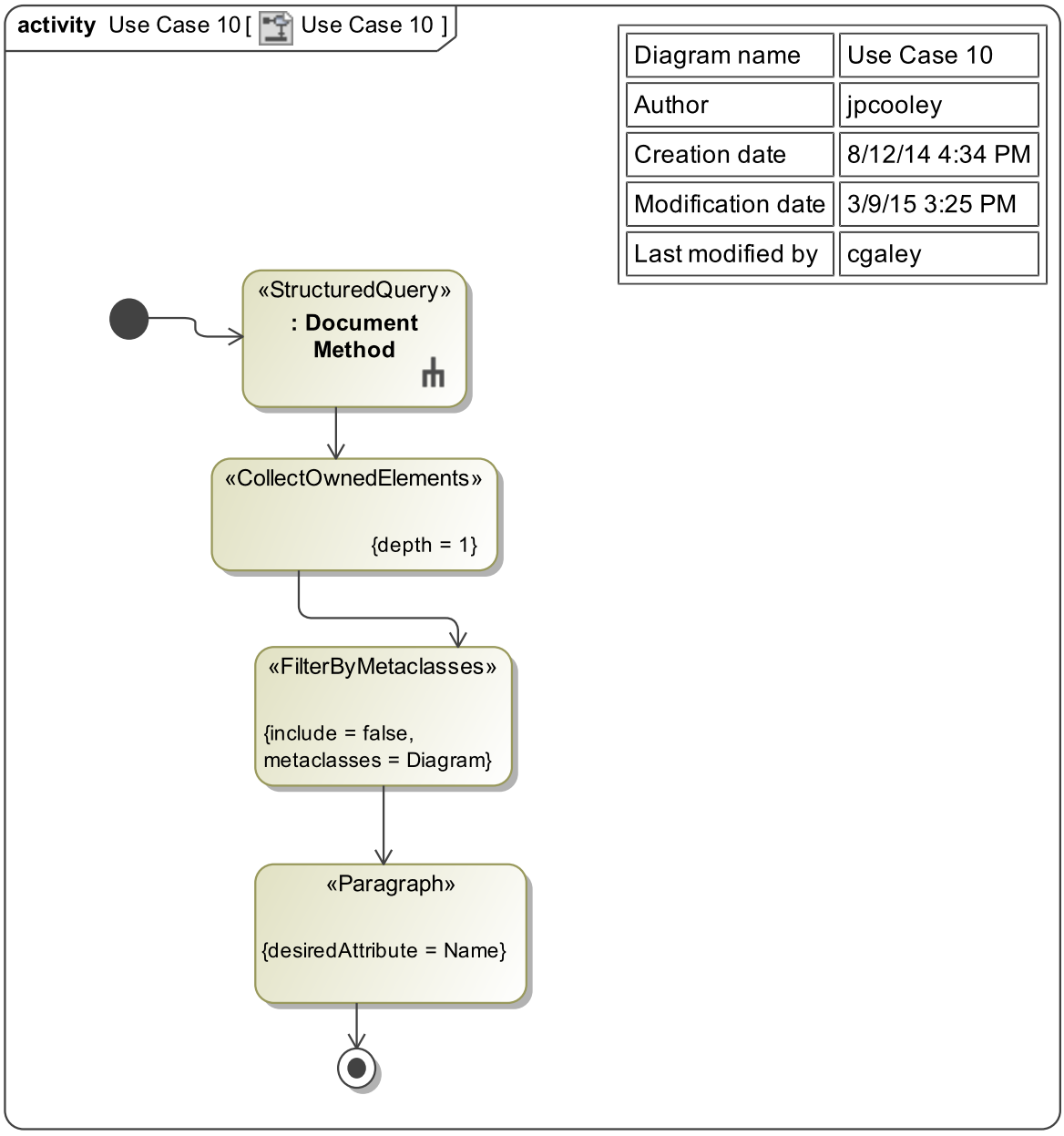
Dog

3.3.2.2.3.8. (View Under Construction)

Use Case: Has targets and desired attribute set as "Name"

Expected: Paragraph for each target's name

Figure 3.19. Use Case 10



A Cat goes meow. 'Cat Documentation: '.concat(name)

A Duck goes quack 'Duck Documentation: '.concat(name)

A Cow goes MOOOOOO. 'Cow Documentation: '.concat(name)

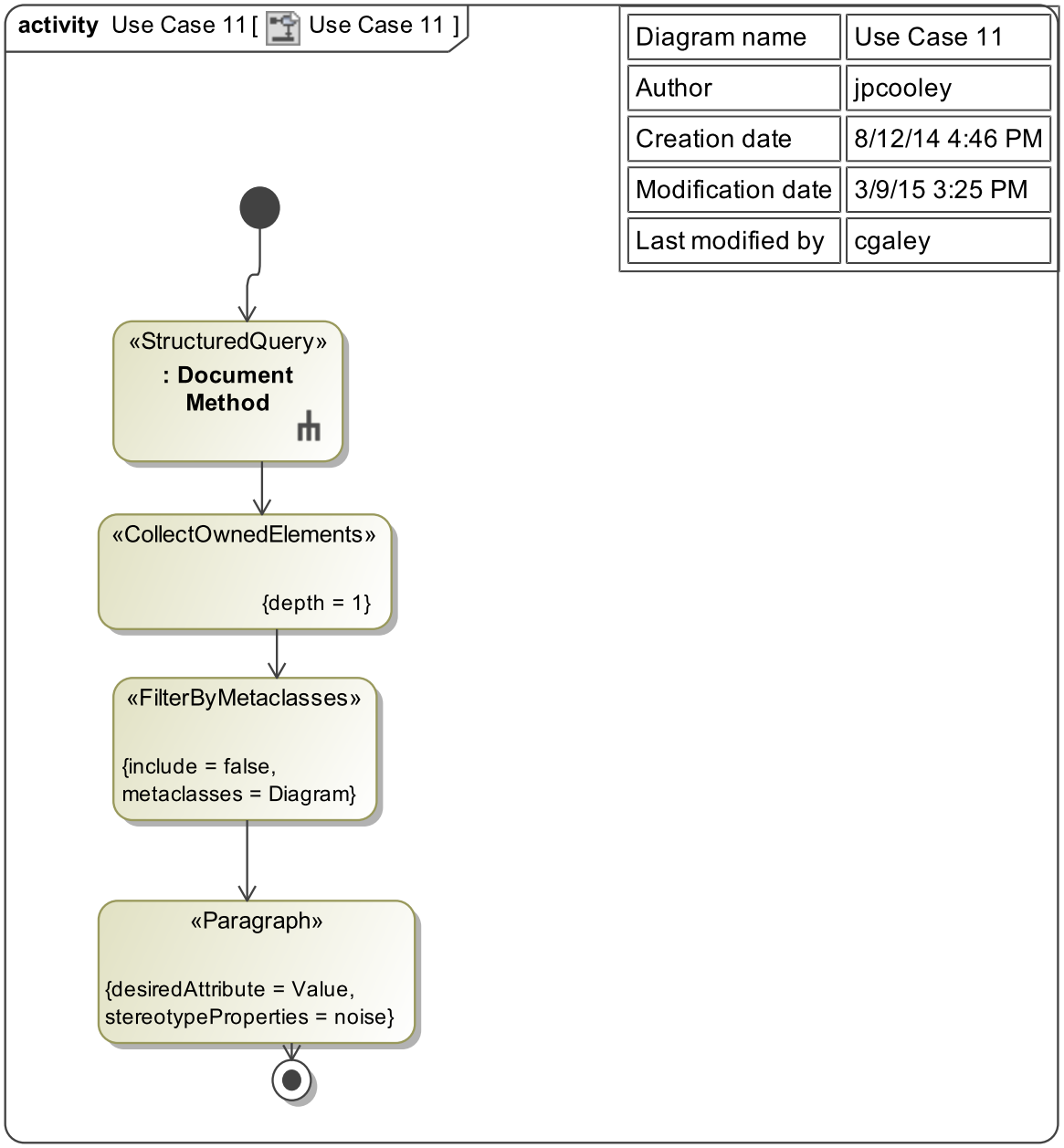
A Frog goes Ribbit. 'Frog Documentation: '.concat(name)

A Dog goes Woof. 'Dog Documentation: '.concat(name)

3.3.2.2.3.9. Paragraph Using Default Value of Value Properties

By specifying desiredAttribute and stereotypeProperties of <<Paragraph>>, it is possible to build paragraphs with the set values. In this example, The Animal elements are collected and desiredAttribute is set to Value and the stereotypeProperties tag is set to noise. This results in the "noise" property being selected and the value being displayed. Results shown below.

Figure 3.20. Use Case 11



meow

quack

moo

ribbit

bark

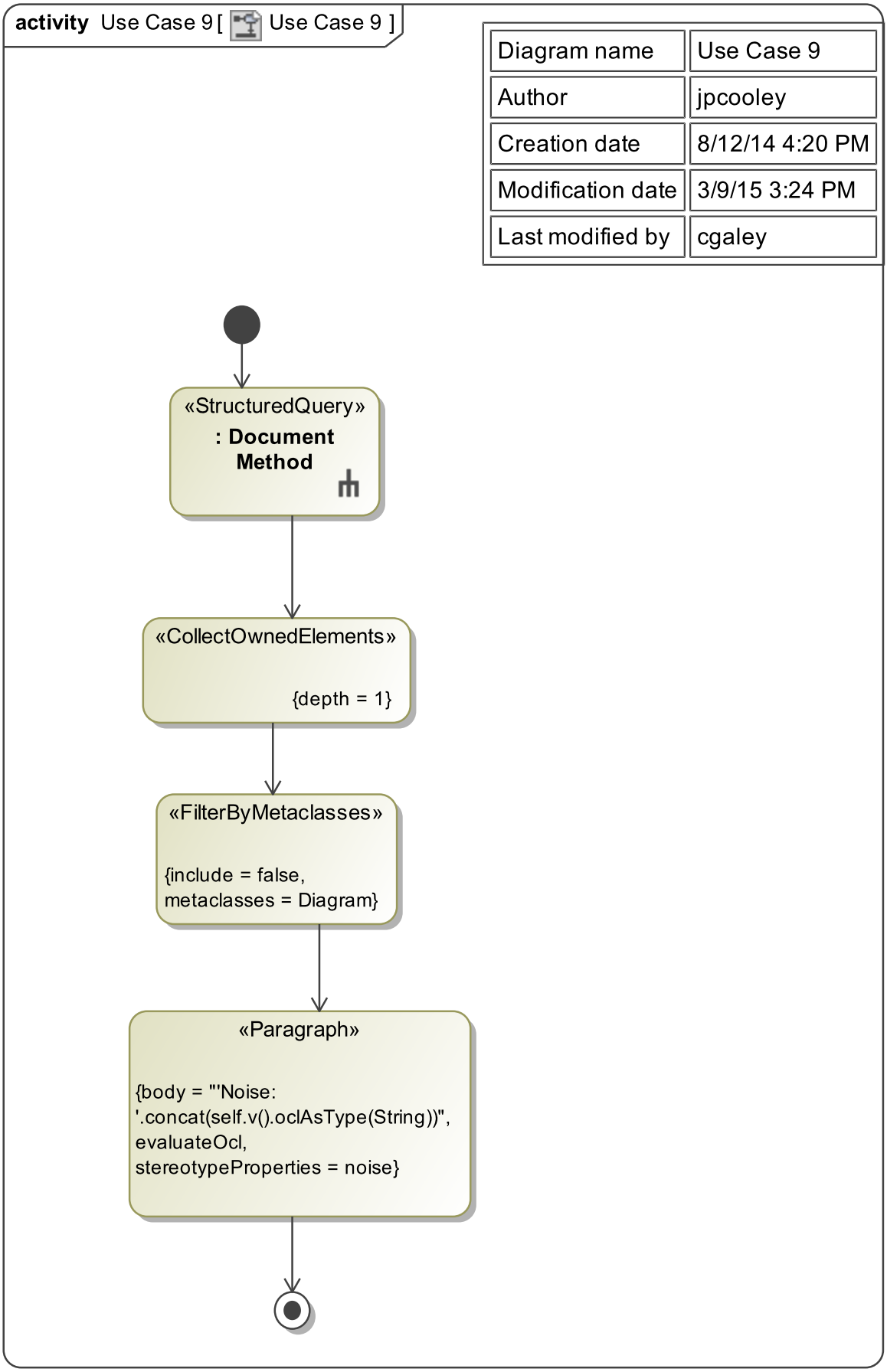
3.3.2.2.3.10. (View Under Construction)

Use Case: Has targets, has text in body, gets stereotype properties, and uses OCL

Expected: Evaluates OCL expression from body text on target-property pairs (Output varies)

NOTE: Currently non-functional. Known bug.

Figure 3.21 Use Case 9

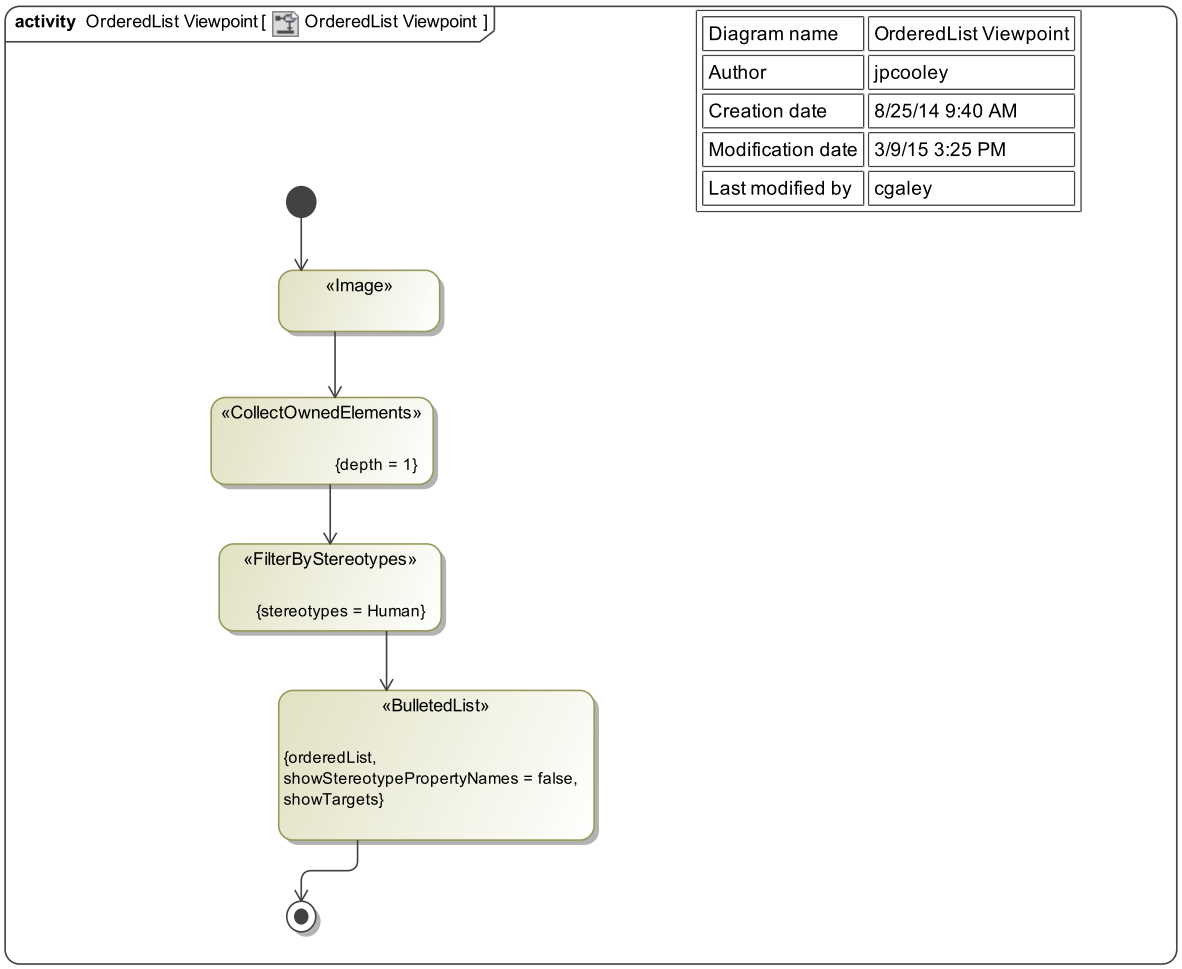


3.3.2.2.4. List

"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 "Example Elements" from previous examples and a filter was applied to only identify the human 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 3.22. OrderedList Viewpoint



This is the view diagram used to create the List section.

Below is the list of characters identified with a human stereotype.

1. Starbuck
2. Boomer
3. Adama
4. Lee

5. Helo
6. Gaeta
7. Roslin
8. Baltar
9. Zarek
10. Billy
11. Cottle
12. Hera

3.3.2.2.5. 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 cannonical view hierarchy.

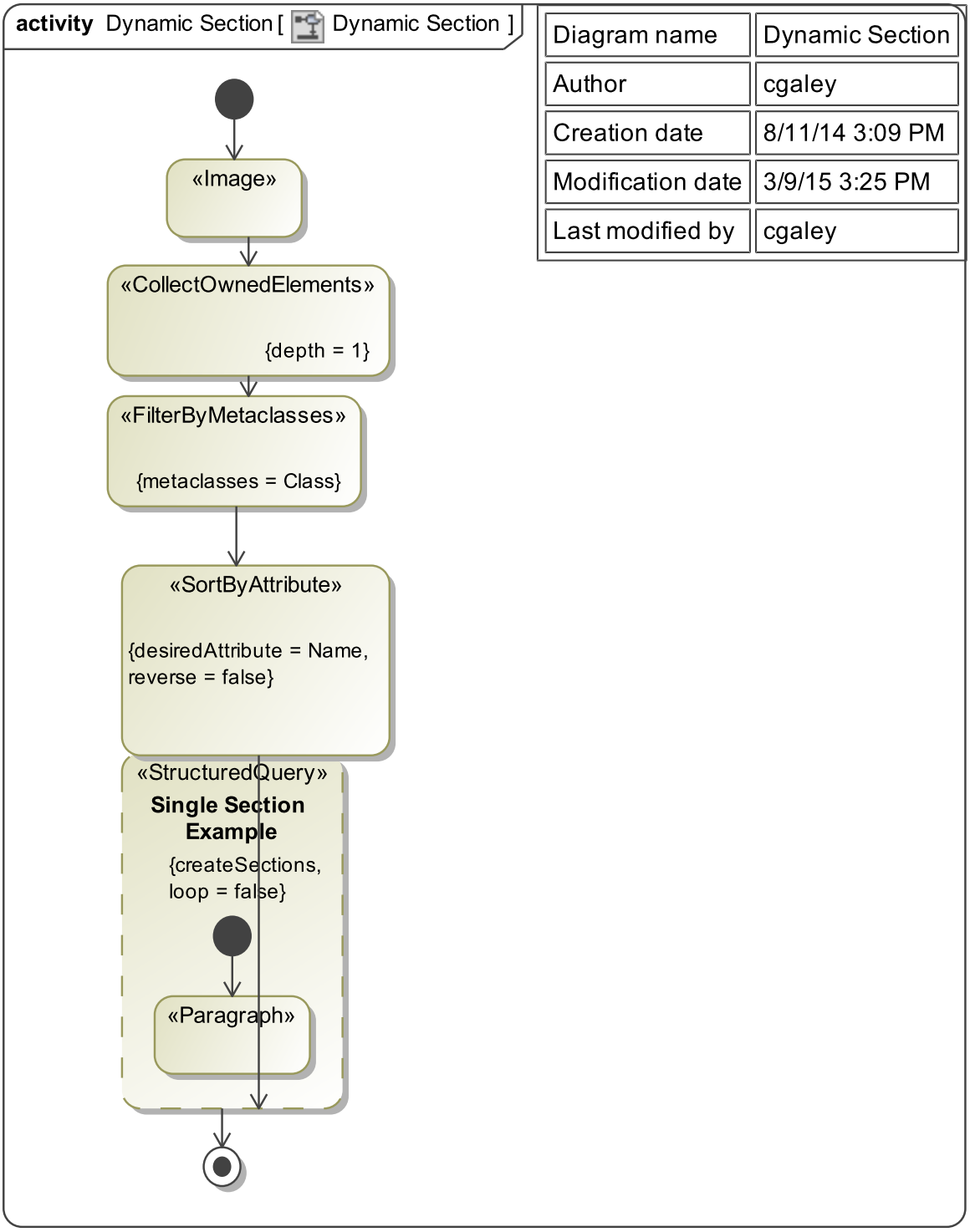
3.3.2.2.5.1. Single Section

A single dynamic section can be created by inserting a Structured Query and setting the "createSections" tag value to "true". A new section will be created and the method will then execute. The name of the section is set by naming the Structured Query.

As you can see in the viewpoint method diagram, within the Structured Query is a blank paragraph action. Because the paragraph is blank, as discussed previously, it will display all the documentation of the collected elements. Thus in the below section, the output will be the documentation for each of the 5 example animal elements.

The single dynamic section can be used to categorize or organize paragraphs automatically created by the model. For example the "Single Section Example" could have been named "Animals and their Corresponding Noises". So long as the documentation and the title match, it would not be distracting or glaringly obvious that this was document content automatically generated from the model. Additionally, if we wanted this to look more like professional document paragraphs, we would remove "'[Animal] Documentation: '.concat(name)" from the element documentation. However, since that portion of the documentation is present to help with other examples, it will remain.

Figure 3.23. Dynamic Section



This is the view method diagram used to create this section and the single dynamic section below.

3.3.2.2.5.1.1. Single Section Example

A Cat goes meow. 'Cat Documentation: '.concat(name)

A Duck goes quack 'Duck Documentation: '.concat(name)

A Cow goes MOOOOOO. 'Cow Documentation: '.concat(name)

A Frog goes Ribbit. 'Frog Documentation: '.concat(name)

A Dog goes Woof. 'Dog Documentation: '.concat(name)

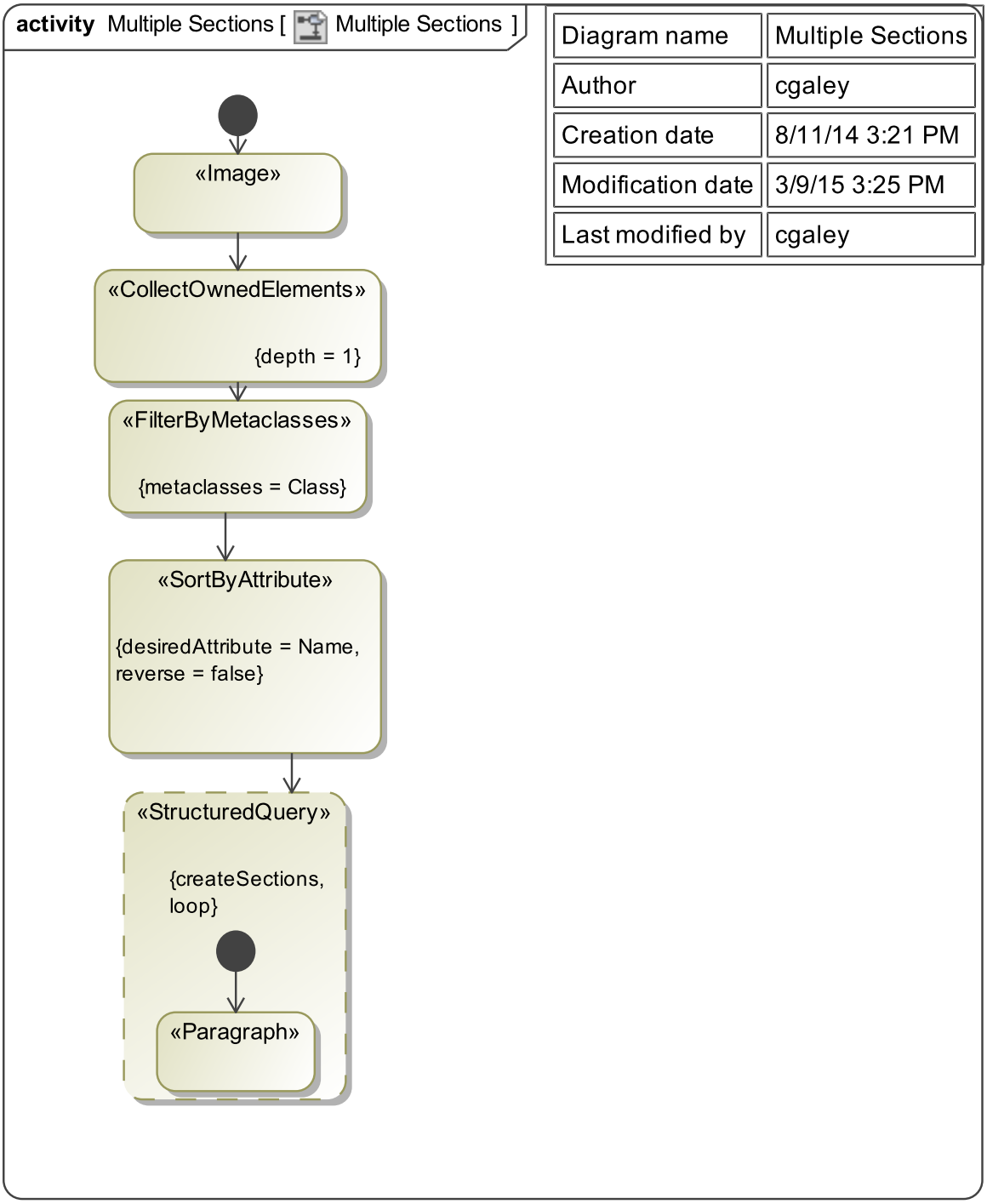
3.3.2.2.5.2. Multiple Sections

Just as with the single dynamic section example, multiple dynamic sections are created with a Structured Query in which "createSections" is selected as true. The only difference is that for multiple sections, the "loop" tagged value is also set to "true". Loop will create a new section and content underneath for each element passed.

As you can see in the below viewpoint method diagram, the only differences between the single section viewpoint method diagram is the addition of the "loop" tagged value, and the removal of the Structured Query title. In this case, the section names will come from the element names and the section paragraphs will the be element documentation. Thus in the below sections, each of the five animal elements is represented by their own section and the paragraph of each is again the element documentation.

Multiple dynamic sections can also be used to categorize or organize paragraphs automatically created by the model. The utility of this example could be a little more clear if the animal elements had more documentation under them. In that case each animal would have its own section, with possibly multiple paragraphs underneath. For example, if each animal's documentation had been consistently formatted, such as with paragraphs on animal description, animal noises, and animal habitat, then the resulting document paragraphs could be used to rapidly organize and fill out document sections.

Figure 3.24. Multiple Sections



This is the view method diagram that was used to create this section and the multiple animal sections shown below.

3.3.2.2.5.2.1. Cat

A Cat goes meow. 'Cat Documentation: '.concat(name)

3.3.2.2.5.2.2. Duck

A Duck goes quack 'Duck Documentation: '.concat(name)

3.3.2.2.5.2.3. Cow

A Cow goes MOOOOOO. 'Cow Documentation: '.concat(name)

3.3.2.2.5.2.4. Frog

A Frog goes Ribbit. 'Frog Documentation: '.concat(name)

3.3.2.2.5.2.5. Dog

A Dog goes Woof. 'Dog Documentation: '.concat(name)

3.4. Sync EMS Project in MagicDraw with EMS Server

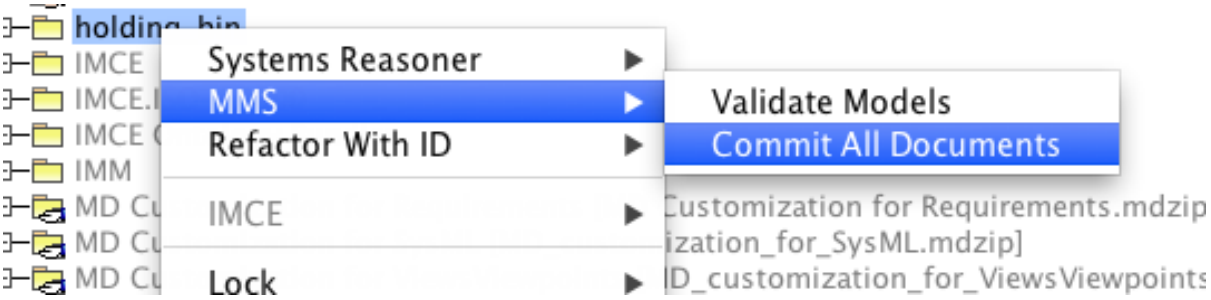
Once the user has an EMS Project configured for an EMS Server they can use the MDK plug-in to publish content the server. This section will outline how to work with and collaborate with others now that your project is online.

There are now three ways to sync the EMS Server with EMS Projects in MagicDraw. They are a real-time dynamic sync and a transactional upload/commit sync. Additionally, a manual sync is available for handling synchronization at an element by element granularity. Both of these are described in this section.

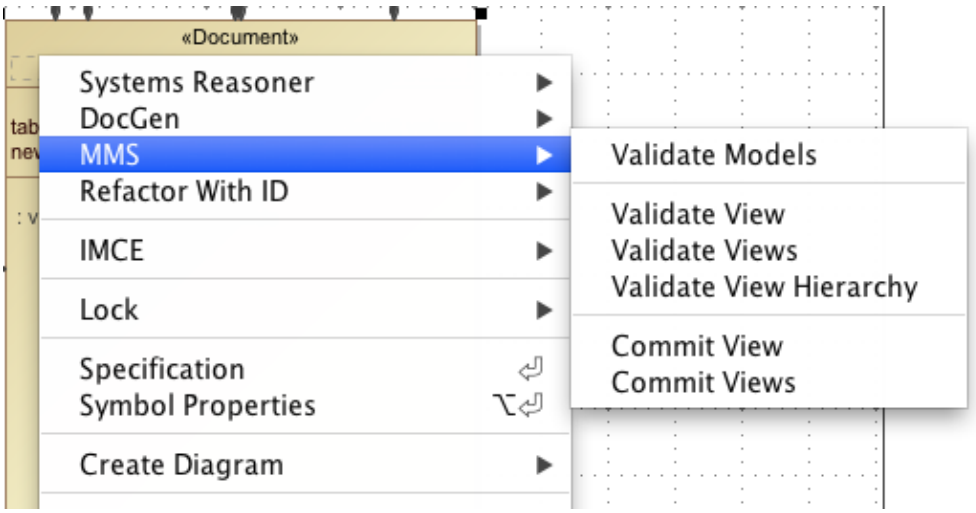
Note: A number of the items in this section are involved in various Project Policies (appendix). These are items in the workflow that may be altered by individual projects as needed for their own individual workflows. The information presented here is for the general use of the tools.

3.4.1. Synchronizing Views and Viewpoints

Once the **EMS Project** in MagicDraw is synchronized to the EMS server, its documents can then be published to the View Editor. This can happen automatically, by right-clicking on a package and selecting 'Commit All Owned Documents'. (Note this will commit all view structures, but document hierarchy still needs to be validated and exported individually since this can be edited on the web)



this action will export every single document's view structures recursively within the scope of the Package to View Editor. There are many options for syncing and committing views and documents. These are reached by right-clicking on any view



Commit View/Commit Views

To publish documents manually, right click on a particular document and select 'MMS -> Commit Views'. This will commit the view structure for the view clicked on and all children views (Commit View will do only the clicked on view). This will NOT commit the document hierarchy since the hierarchy can be changed on the web and requires a validation to make changes.

Validate Views/Validate Views

If the document or view is not already on the EMS server the validation window will return with "[EXIST] This View doesn't exist on view editor yet". As before, right click on the error to show the options for reconciliation. The reconciliation options are:

- Commit (sending the model version to the server)
- Ignore (This option lets you choose to ignore the issue, which does not change either the model or the View Editor.)

Running Validate View on a document will also try to validate its hierarchy, which is explained next.

Validate View Hierarchy

This will allow users to compare the ordering of views and viewpoints between EMS server and EMS Project in MagicDraw. Will result in validation offering:

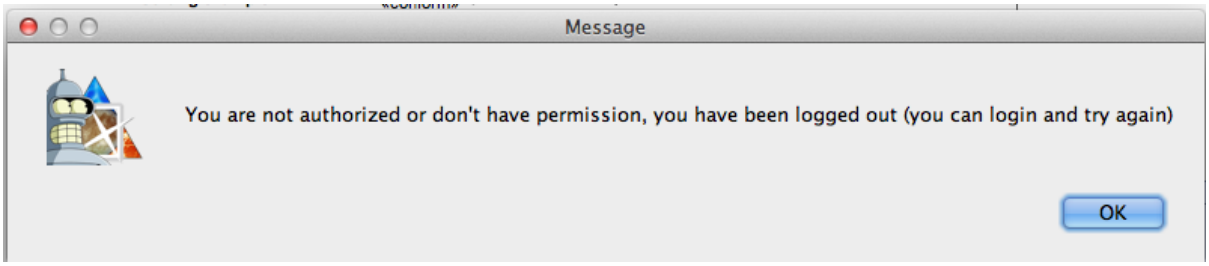
- Accept
- Commit

(NOTE: this must be performed on the document not a view)

Which option you chose is dependent upon your specific project or task. It never hurts to commit more when you are generating the document the first time. However, later on when working with your document and incorporating changes, you do want to avoid committing parts of the model that you have not yet worked on.

You can still see ignored validation errors by making sure the right-most button on the Validation window's top bar is set to "ALL" (vs. Ignored or Not Ignored). This is considered good engineering practice.

NOTE: If you do not have permission to add documents or edit that specific model on the server you will get the following error.



If you feel this server response is incorrect please contact your task manager or the MDev Team.

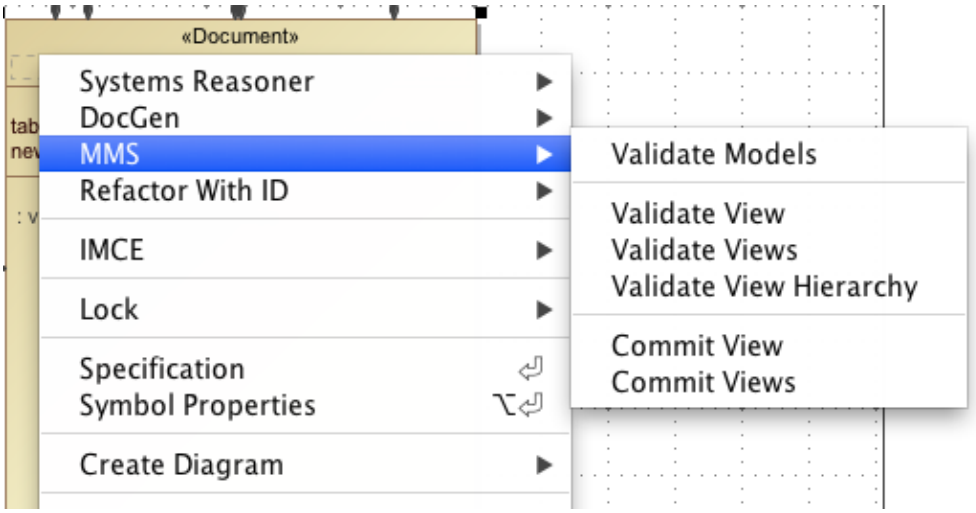
After you have resolved the reconciliation options and committed your document, it should now be visible and editable on the View Editor site.

3.4.2. Manual Synchronization

Manual syncing between MagicDraw and EMS server uses a similar procedure to Crushinator 2.0. However, changes can be made either in EMS View Editor or MagicDraw. In Crushinator 2.0, some changes such as creating views, hierarchy changes, etc. had to be done in the MagicDraw model.

To start, sign into EMS server using the MMS menu on the top bar of MagicDraw. If not already done, configure the model using the steps described in a previous section of the manual. While not strictly necessary for manual syncing, it is highly recommended to create a MagicDraw branch and corresponding EMS task (e.g., workspace) to work in, especially if you might use Dynamic Sync at some point. Now you have two copies of your model, one in MagicDraw and the other in EMS. You can edit the content either from MagicDraw or EMS View Editor. With manual sync you are responsible for transferring the changes from one to the other. If the MagicDraw model was loaded from a Teamwork server, the user is responsible for committing the MagicDraw model to the Teamwork Server after sync between MagicDraw and EMS. The rest of this section explains how you can compare models between MagicDraw and EMS, analyze the discrepancies, and incorporate or reject the differing content.

Before the user can transfer changes to/from between MagicDraw and EMS View Editor, the changes must be identified. The user can compare all of the contents of the MagicDraw model against the server using the right click menu "MMS>Validate Model" on any element. Validating the entire model takes approximately 1 second per element, and it is recommended to do this in the background. You can also just validate a selected view or a view along with its hierarchically-connected views using corresponding right click menus. The latter approach is recommended to save time as well as to ensure you don't accidentally sync someone else's work by mistake. For example, if someone else is changing a different model element, it will show in the validation window if you validated that element. However, you shouldn't accept or commit it since you may not know what the change is about.

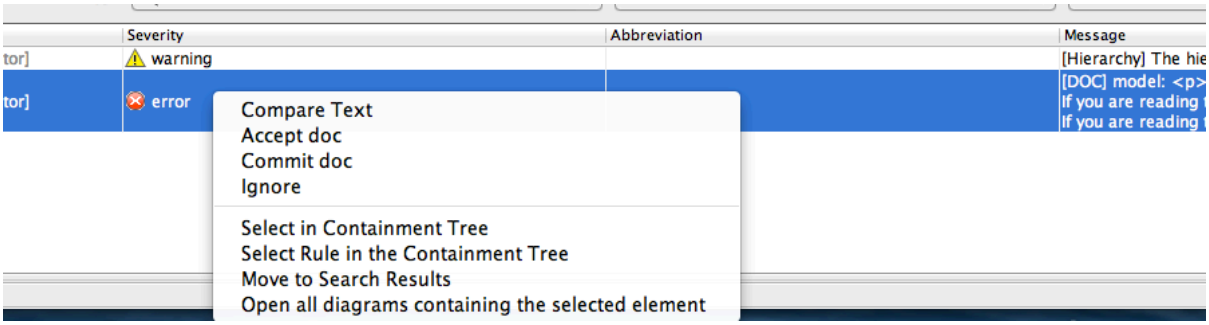


Once completed, any discrepancies or changes between MagicDraw and the server will be reported for each element in the validation window. The validation window message field will explain the differences (in errors or warnings) between web and model. By right-clicking on the message, various options such as commit (send the model version to the View Editor) or accept (replace the model version with the View Editor version) can be executed. This can be done on an element by element basis, or in bulk by selecting multiple entries with the same error message at once. To make bulk changes, it helps to sort errors and warning by messages by clicking on the "Message" header in the validation table. If applicable, you can commit the view with or without the corresponding model elements. You can also commit it recursively, which commits everything hierarchically connected to the view as well. Finally, you can choose to ignore the issue, which does not change either in MagicDraw or in EMS.

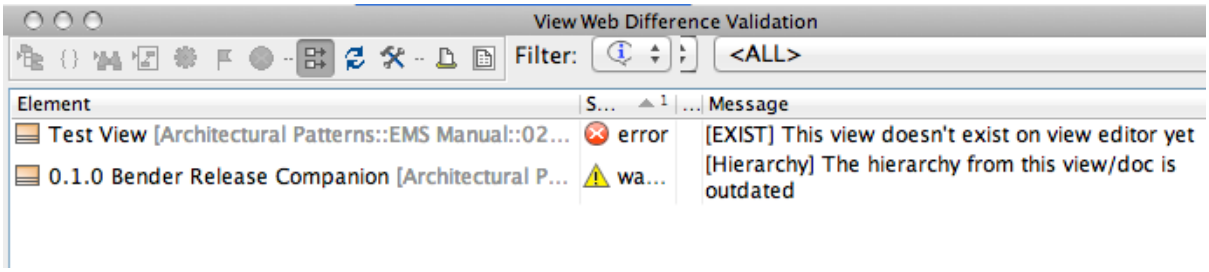
Note: On shared projects, the best practice is to not ignore issues. Others might need to use something you have ignored. Other users may be validating the model at the same time and those errors may correspond to what they are working on. Moreover, once ignored, it is really easy to miss those issues when they do need attention and resolution. You can still see ignored validation errors by using the right-most button on the Validation window's top bar and setting it to "ALL" (vs. Ignored or Not Ignored). This is considered good engineering practice.

Some common discrepancies in the model after validation:

- If the discrepancy is purely documentation, a "Compare Text" option is offered that shows the web and model text side-by-side.



- When the organization of sections of a document was changed, it will be detected as discrepancies in document hierarchy. For example, when a section is added, moved, or removed, validation will give a warning and the change may need to be synced using MDK plug-in. **Note that his has to be done at the Document level.** Validate the <<Document>> hierarchically. (If the organizational change is limited to the direct children of the document, you need to validate only the document.) T he system will return the following hierarchy warning in the validation window. Note that with Crushinator 2.1 the hierarchy can be accepted from the View Editor as well as committed from the MagicDraw model (in Crushinator 2.0, document hierarchy was able to be changed only in MagicDraw).

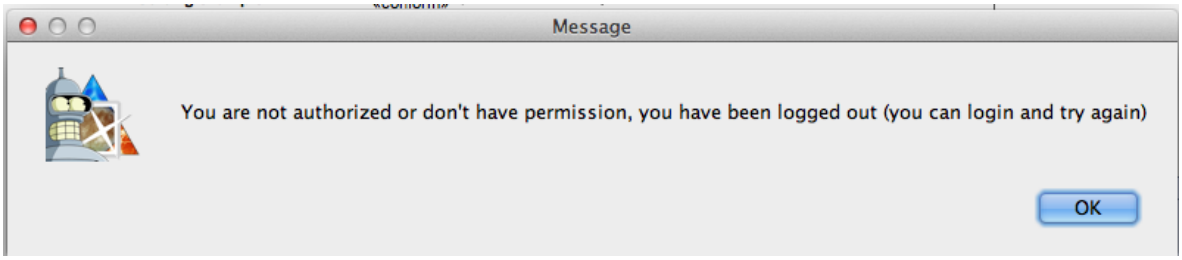


If a new section was added, you will also get an "Exist" error the view. If there is an "Exist" error, first right click and add the view (this will also add the element and its relation to the model database) before syncing the hierarchy. Similarly, if a section was deleted, right click on the error in the validation window and delete the view before syncing the hierarchy. The new view and order should be now on the server and visible in EMS.

- You may get a warning for "Data" saying "The baseline tag isn't set, baseline check wasn't done". This is an artifact of EMS server and can be ignored.

- **NOTE: If you do not have permission to add documents or edit that specific model on the server you will get the following error.**This often occurs based on your permissions for the corresponding

site on EMS server. Please contact your task manager or the MDEV Team to address the problem.



Here are a few guidelines for model syncing.

- **Be sure to know what you commit or accept since either operation will overwrite one source of your information.** For the view editor, previous data usually *cannot* be reverted because last save wins. If you are using a task (e.g., workspace, branch) in EMS, make sure only you make changes to it. For a MagicDraw model managed in TeamWork server, previous data may be retrieved from an earlier version, but reverting may result in *other lost work* .

- **Accept/commit changes in a timely manner.** Unless you are the only person with permission to edit the workspace, others can overwrite your changes if not accepted/committed in a timely manner.

- **Accept/commit only what you changed.** As discussed earlier, you won't be able to know which version to sync if you did not make the change yourself. You wouldn't want others changing your work without permission, so don't do it to others.

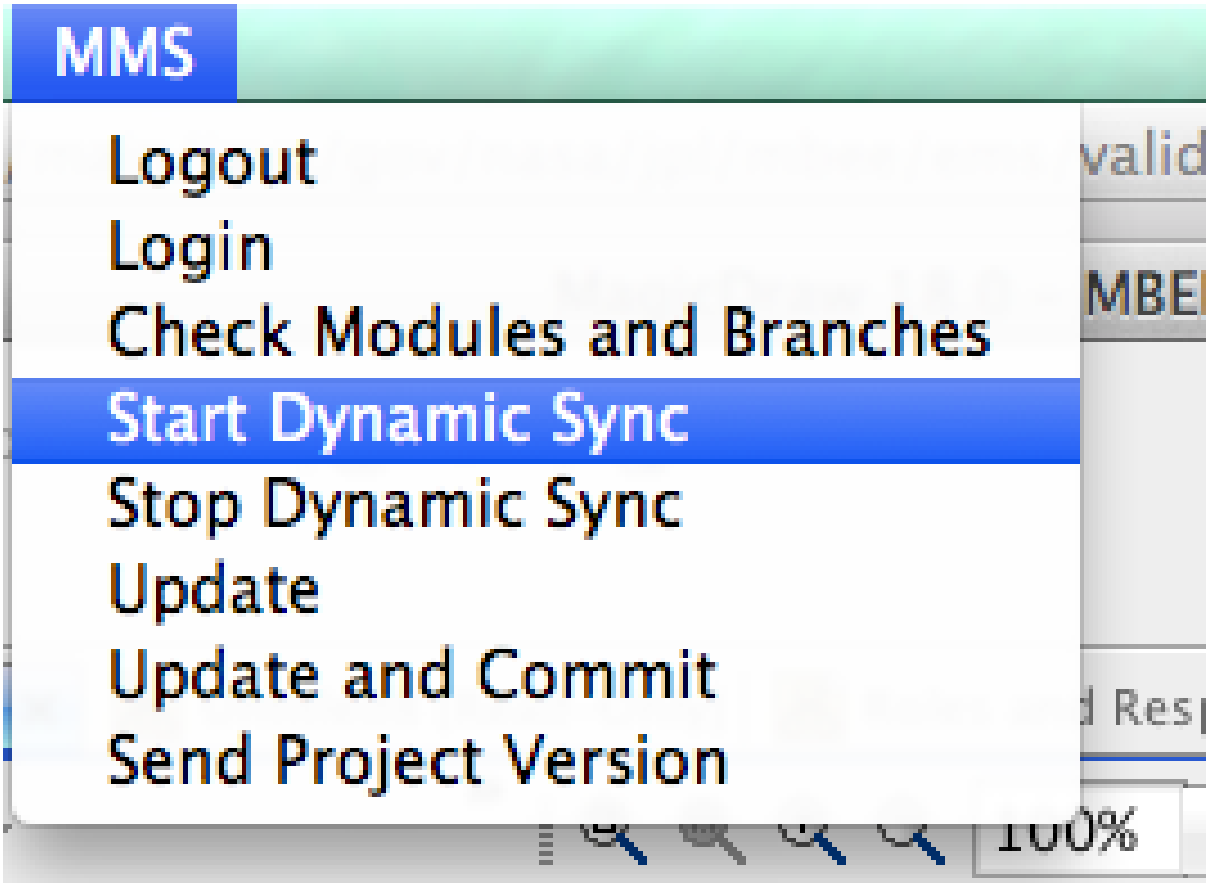
- **Remember the current "source of truth" is still the model stored in the Teamwork server.** After accepting changes from EMS, be sure to commit the MagicDraw model to the Teamwork server.

3.4.3. Dynamic Sync

Check your Project's Modeling Policy for permissions and procedures.

In addition to manual sync, Crushinator 2.1 provides a new capability to automatically sync models as changes are made. **Note that Dynamic Sync works with only models on a branch (or task, workspace). This is because it needs to lock the entire model for dynamic sync to start.**

To start, you must use a branch model in MagicDraw that corresponds to your workspace. You must also have already initialized the model on the EMS server. Sign into EMS server using "MMS>login" menu. If the versions for your branch are not the same, you will be immediately notified, in which case you select "Send Project Version" from the drop down menu. (This tells the server that the project is in sync, don't do this unless you know this to be true like did a model validation.) This will send the current version and allow the sync to be initiated. Once this is done (check notification window), start EMS Session again.



Once you have logged in, use "MMS>Start Dynamic Sync" menu to activate the dynamic sync. Now any changes you make in MagicDraw or EMS View Editor will be immediately synced to the other. This is why it is important to use a branch/workspace for dynamic sync. Otherwise you could end up battling other people's changes. To end the session, use "MMS>Stop Dynamic Sync" menu.

If the MagicDraw model is managed in a Teamwork server, **YOU STILL NEED TO MANUALLY COMMIT THE MODEL CHANGES TO THE TEAMWORK SERVER AFTER SYNC** . Remember that dynamic sync is between MagicDraw and EMS, and it does not push changes automatically to Teamwork server.

Note that it is possible to cause a visible delay in the automatic syncing in certain situations. For example, remote accessing a project via VPN over a slow internet connection may slow down the automatic sync. In this case, we recommend using the manual sync option instead.

3.4.4. Delay-Tolerant Sync

The purpose of delay tolerant sync is to get the benefits of dynamic sync but with multiple people working in the model at the same time, or where dynamic sync isn't possible. Instead of sending and receiving in real time, changes in MagicDraw are saved as part of the model, so a full model comparison isn't needed.

The rule of thumb on how to use this:

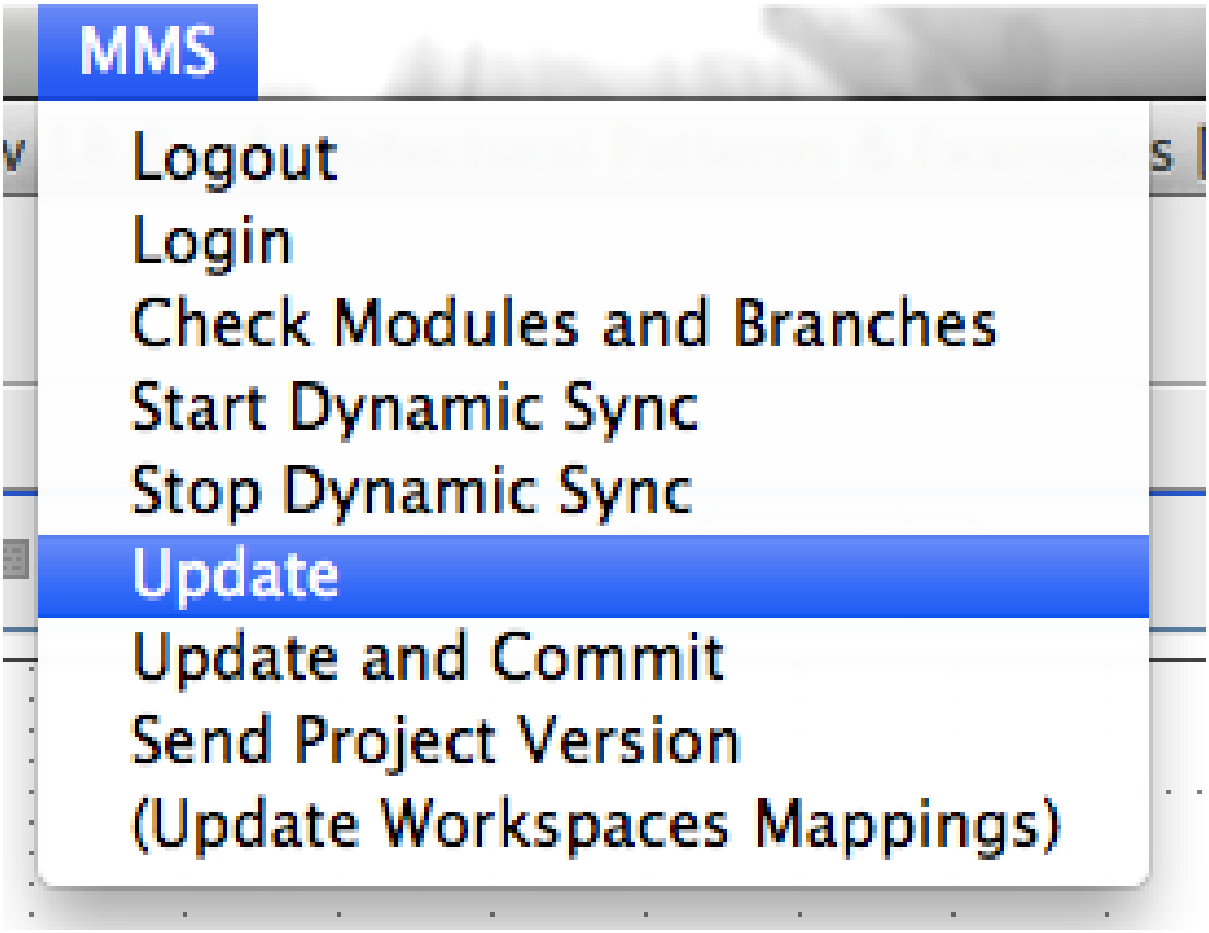
Before selecting "Update" or "Update and Commit" from MMS menu, do an update from teamwork server.

After doing either action, do a commit to teamwork server.

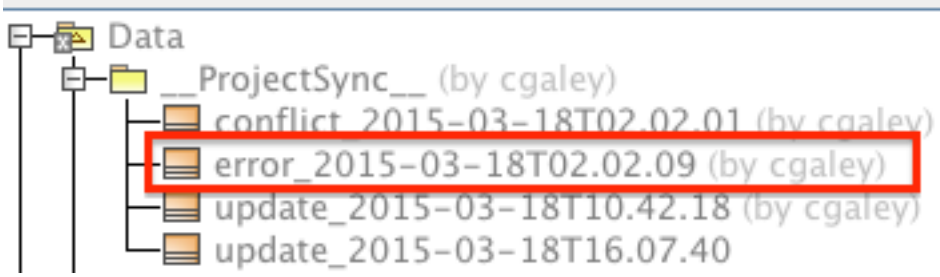
"Update" pulls changes from MMS into Magicdraw, and "Update and Commit" does that plus send any local model changes made in MagicDraw to MMS.

Details

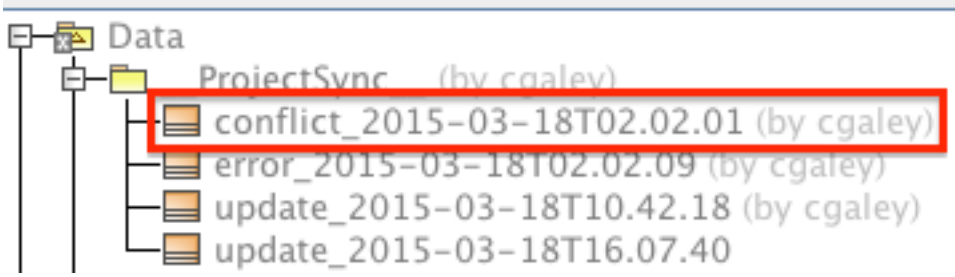
While working in a model, the user can select "Update" from the MMS drop down.



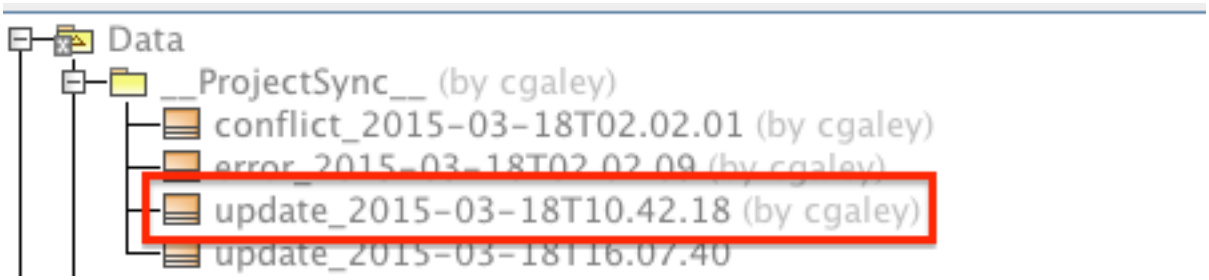
This will get all elements changed on the server since the last update and attempt to apply them. It will attempt to lock the model, but won't abort if some things can't be locked. If it can't be applied (element can't be locked), they will be saved in an error element to be tried the next time. The error block is added automatically at the time of error occurrence.



Update will also look at the local model changes to make sure changes from server doesn't conflict with model changes. If there are conflicts, the validation window will come up to let you resolve the conflicts. Once you've resolve them, run Update again. Any conflicts are stored under the conflicts block and checked again the next time.



Update and Commit does everything that Update does and if there is no conflicts will send to the server all the elements that were updated locally in the model.



If the user made changes in the local model but did not manage to commit them to the server, on "save project" the update block will be updated with all elements that haven't been saved. On a teamwork save it'll attempt to auto unlock the project sync folder. On the MMS update it'll attempt to lock the project sync folder.

When looking at the error and update blocks, it'll always consolidate info from all blocks regardless of timestamp. When updating them, it'll attempt to change the latest timestamped block and update the timestamp, but if it can't, it'll create a new block. When updating blocks, it'll attempt to delete the same type of block whose timestamp isn't the latest. When you have multiple people working in the same project, getting model changes to other people still require using teamwork's commit and update. This can result in multiple update/error blocks being locked by different people. This is ok.

3.5. Manage and Organize EMS Projects in MagicDraw

EMS 2.1 release provides a capability to manage branches of a model both within MagicDraw and EMS and sync between them. Typically branches are used to implement a work package in a model. Once the changes for the work package are approved, they can be merged to the main branch (called master or trunk). The branch and merge process is useful to manage work packages that are being worked on at the same time. Each work packaged can be first implemented on a branch first, and they can be merged in a coordinated way. This reduces the chances of work packages interfering with each other.

This section describes branching mechanism in MagicDraw and explains how branches in MagicDraw can be synced with branches in EMS Server (called task or workspace).

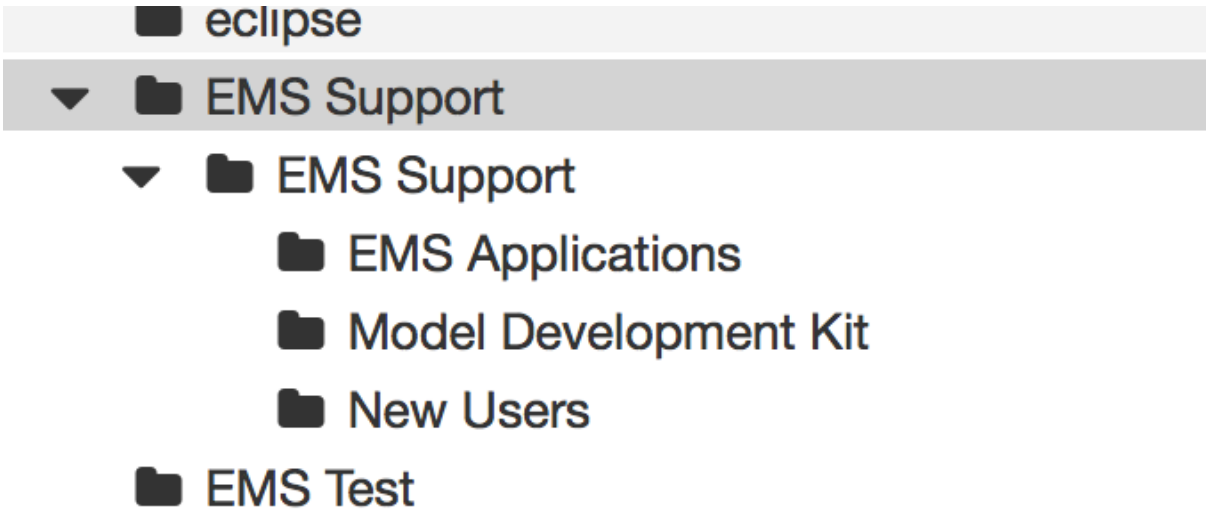
3.5.1. Organizing EMS Projects using MagicDraw

In order to better organize your EMS Projects on the web, MDK incorporates a feature to allow Packages within the model to be appear as if they were sub-sites. There are two advantages to this, better organization for large models and the ability to fine tune access permissions from whole project scale down to individual packages.

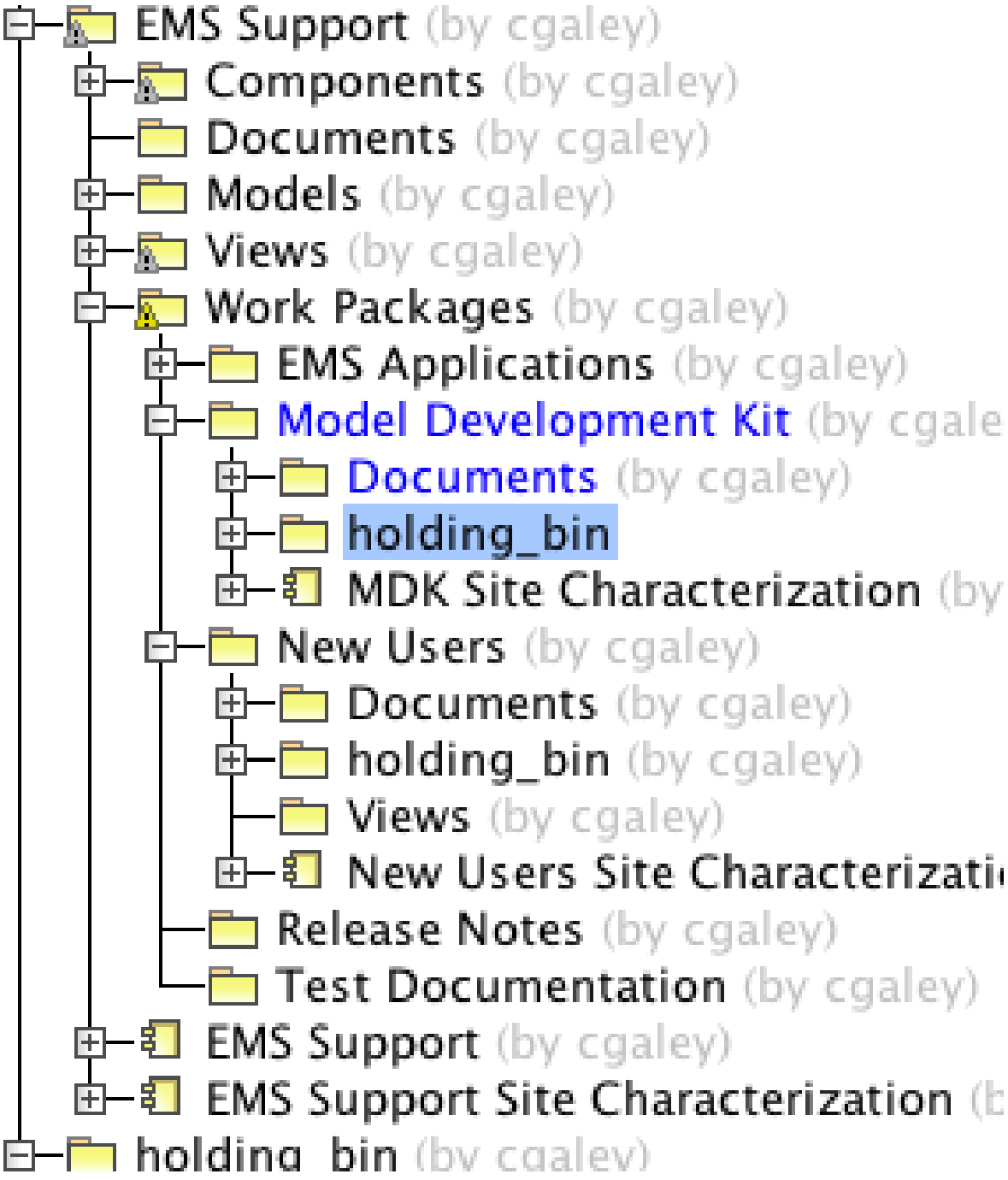
Designation of a package as an sub-site requires two steps, first application characterization of the package as a site and then validation of the package after characterization. To control permissions an optional third step requires use of 'curl' commands in the terminal.

To designate the package as a site first create a UML Component within the scope of the package that you are characterizing. In order to complete the pattern, two relationships need to be created, a generalization to the Site Characterization library element and a <<characerizes>> dependency to the Package that will become the sub-site. The pattern and how to draw it is described in the BDD below.

Site Characterization can be performed on any number of packages to create a MagicDraw Containment Tree like model on the view editor.



Each 'sub-site' will have its own holding-bin for proposed elements and permissions.

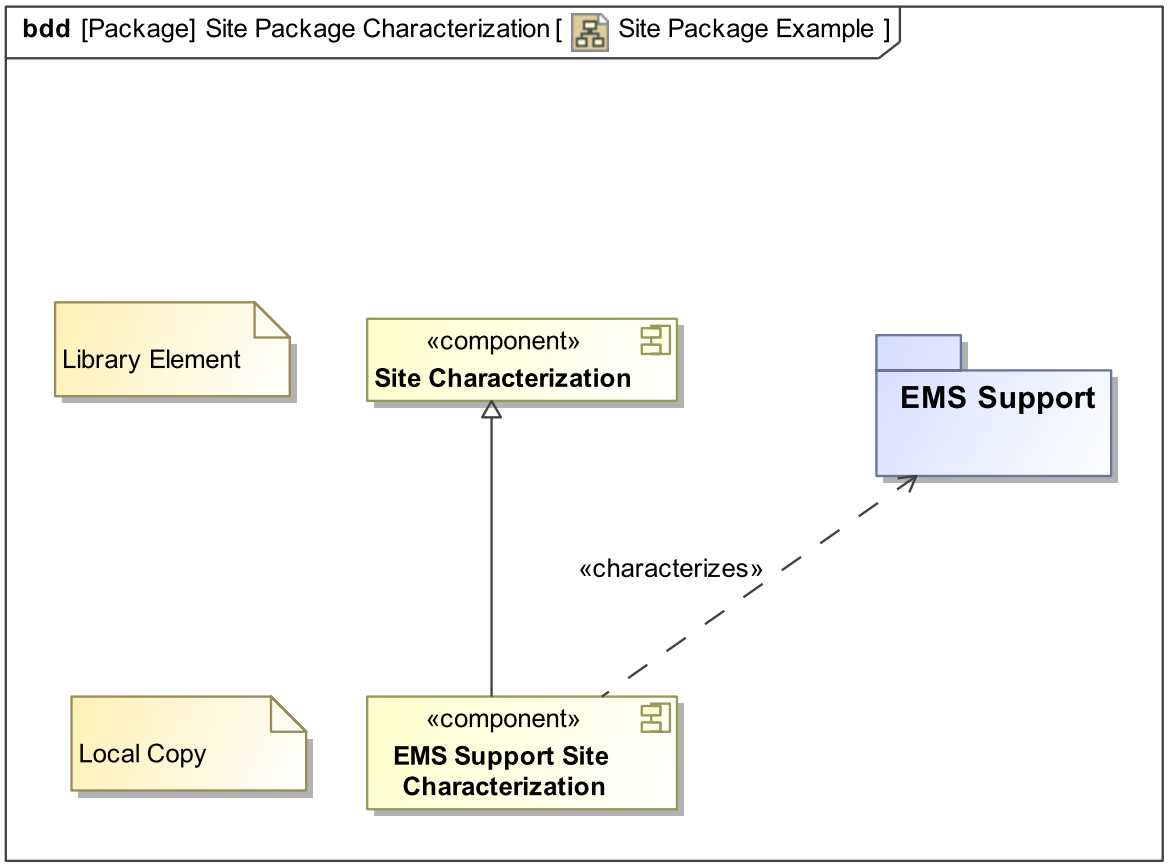


Permissions can be updated so that elements contained within are those for the sub-site rather than the parent site. This is done by the issuing of the following curl command by an EMS Server Admin(Alfresco Share Admin).

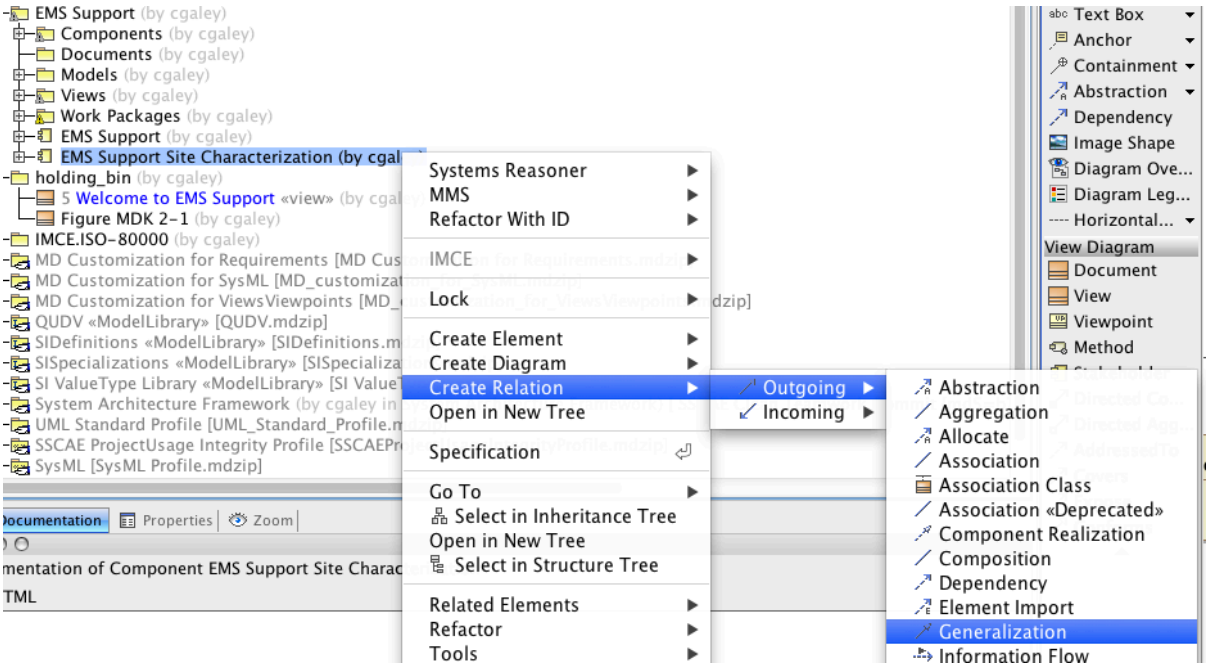
curl -u <username> -k -X POST <EMS Server>/alfresco/service/sites/permissions

This action must be performed each time the permissions of a sub-site changes.

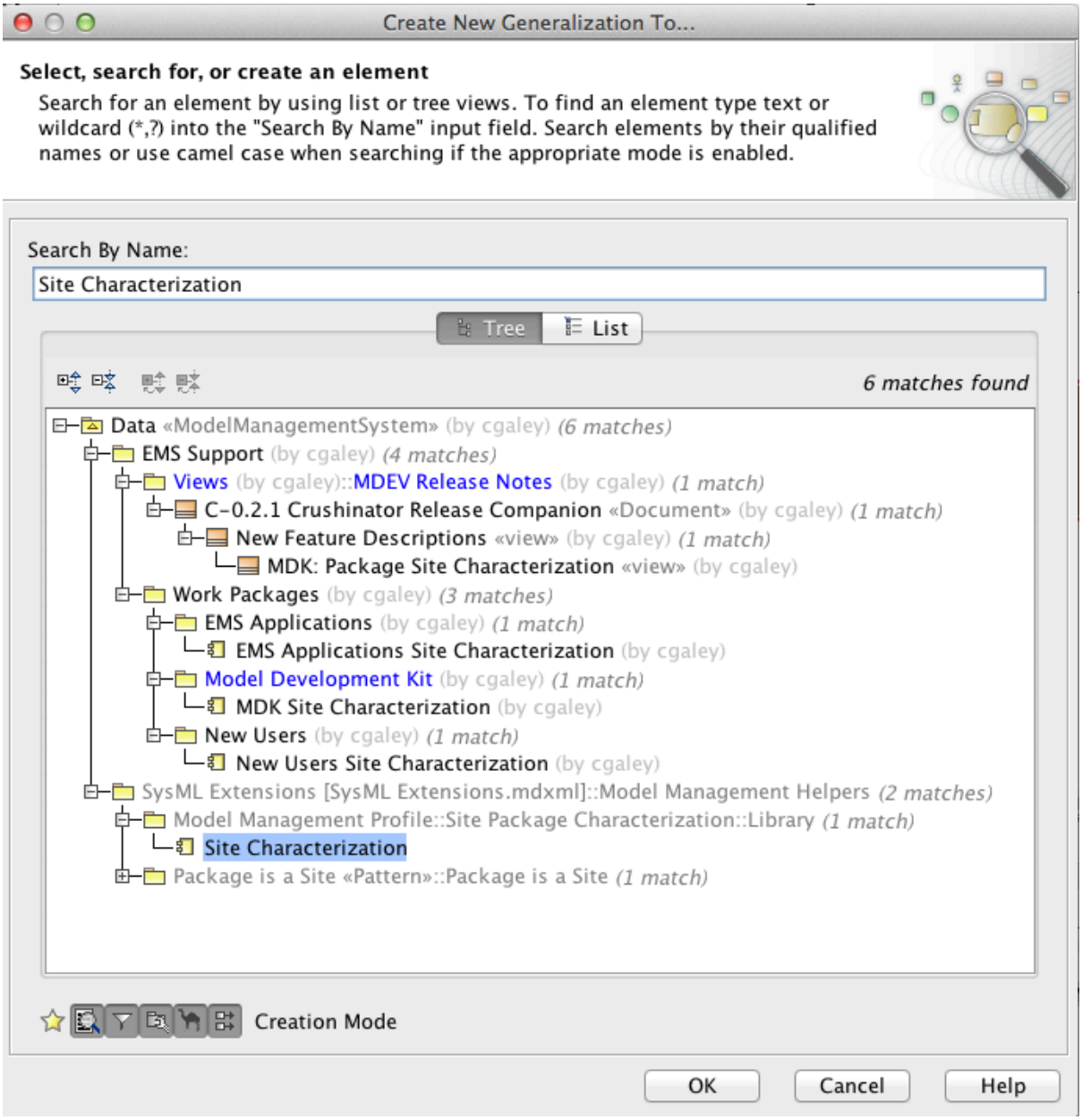
Figure 3.25. Site Package Example



First draw the Generalization from the new sub-site Characterization by right-click Create Relation->Outgoing->Generalization,

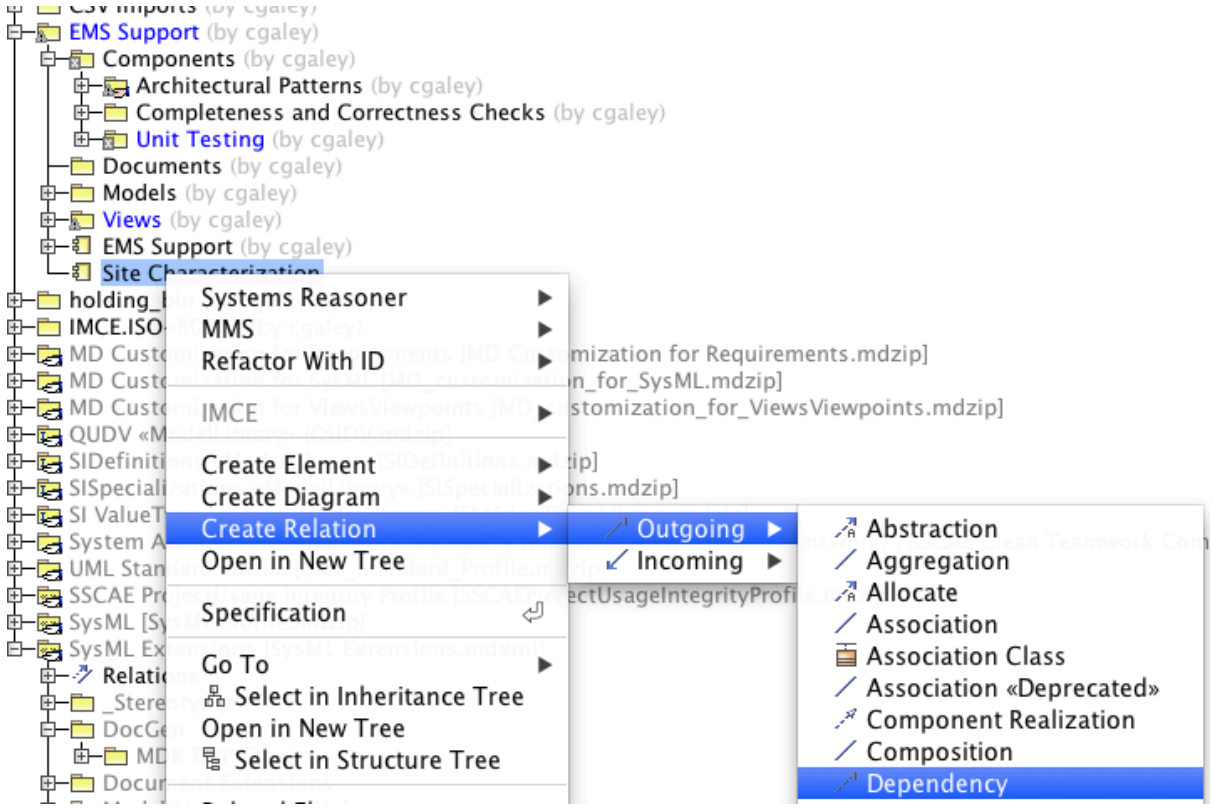


and when the relationship dialog opens search for "Site Characterization" the library element within SysML Extensions->Model Management Helpers->Model Management Profile->Site Package Characterization->Library->Site Characterization.

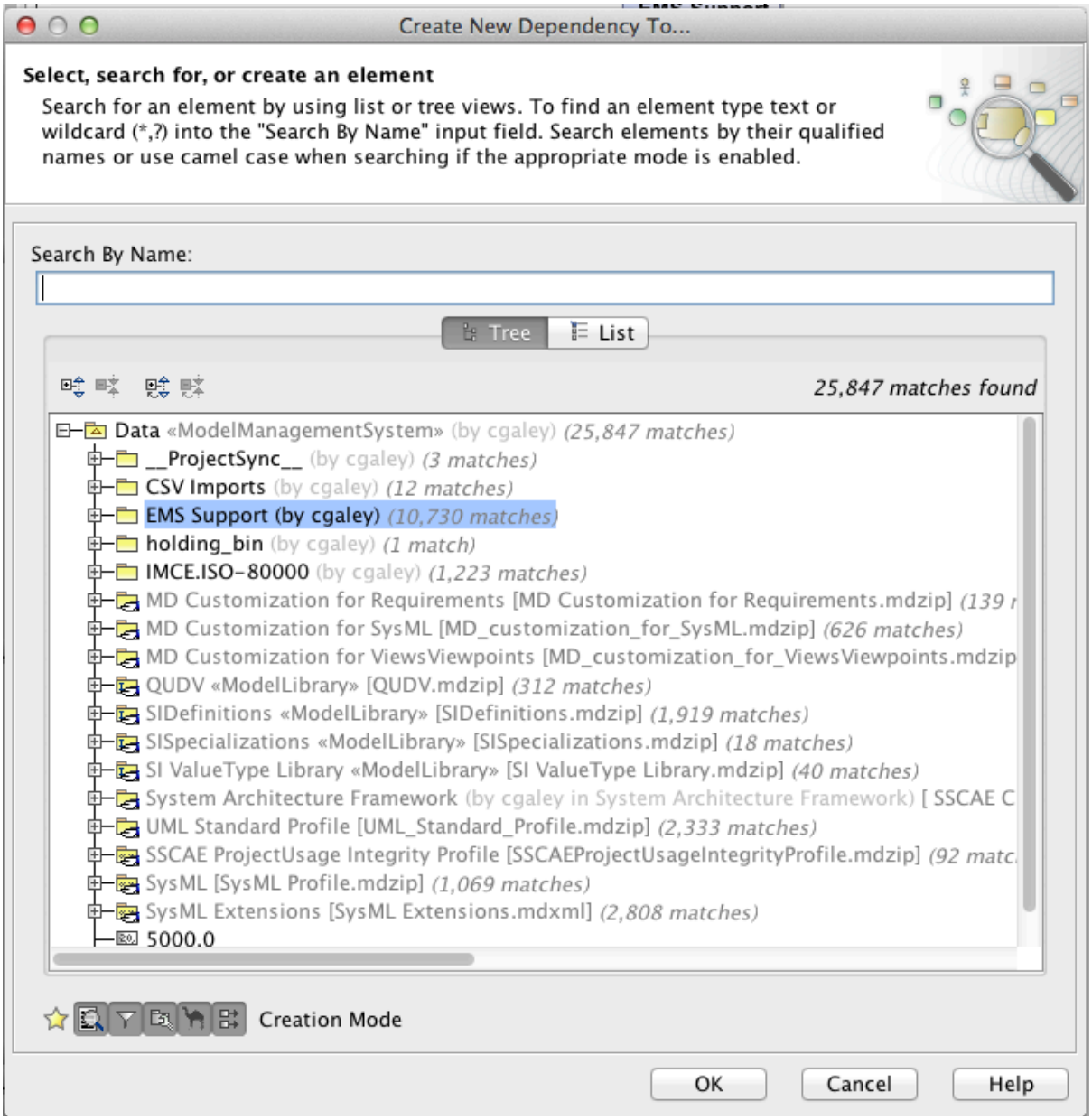


Click 'Ok'.

Next create the <<characterizes>> Dependency by right-clicking on the created site characterization ->Create Relation->Outgoing->Dependency.

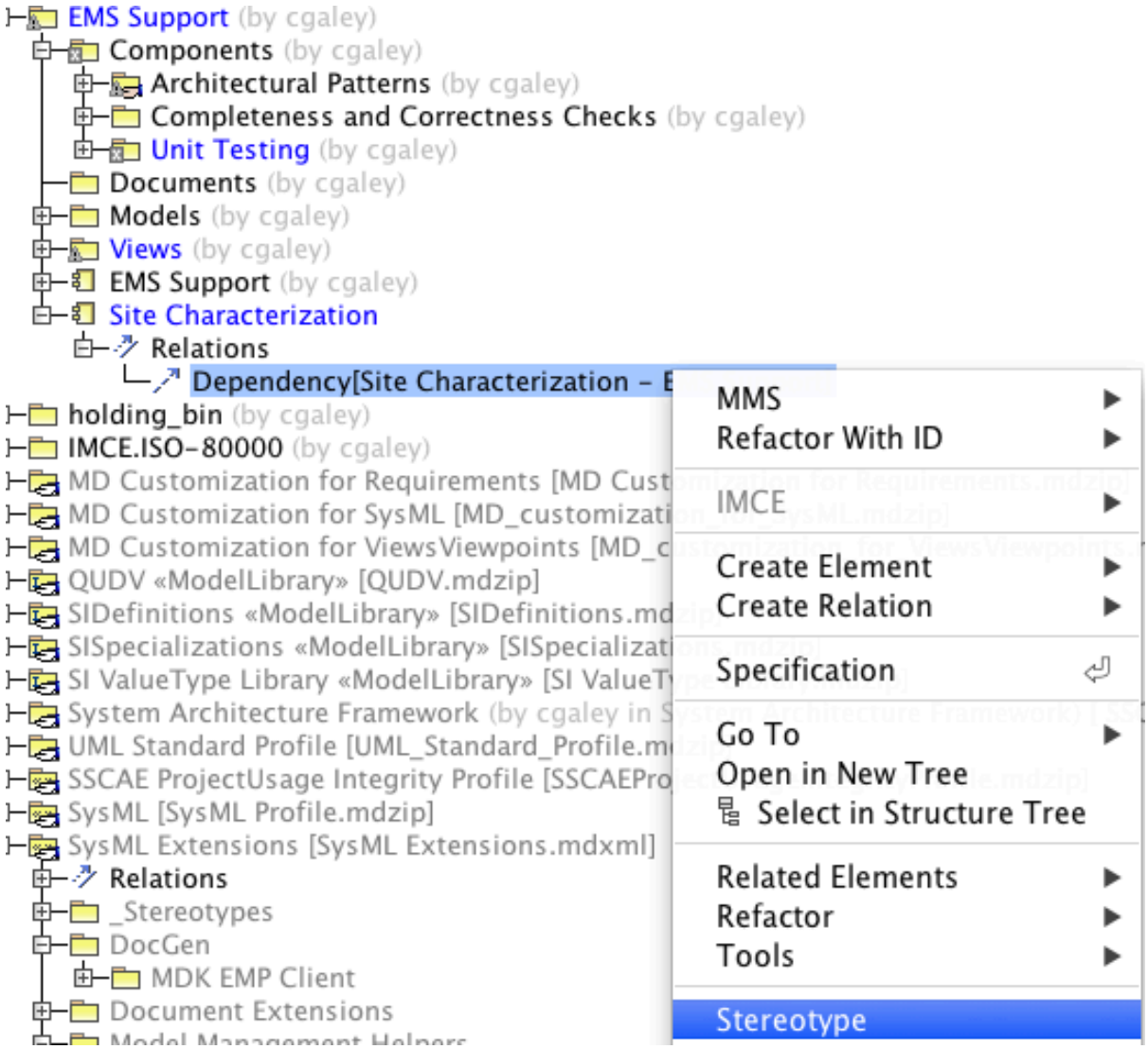


The relationship creation dialog will open, select the Package that is going to be characterized.

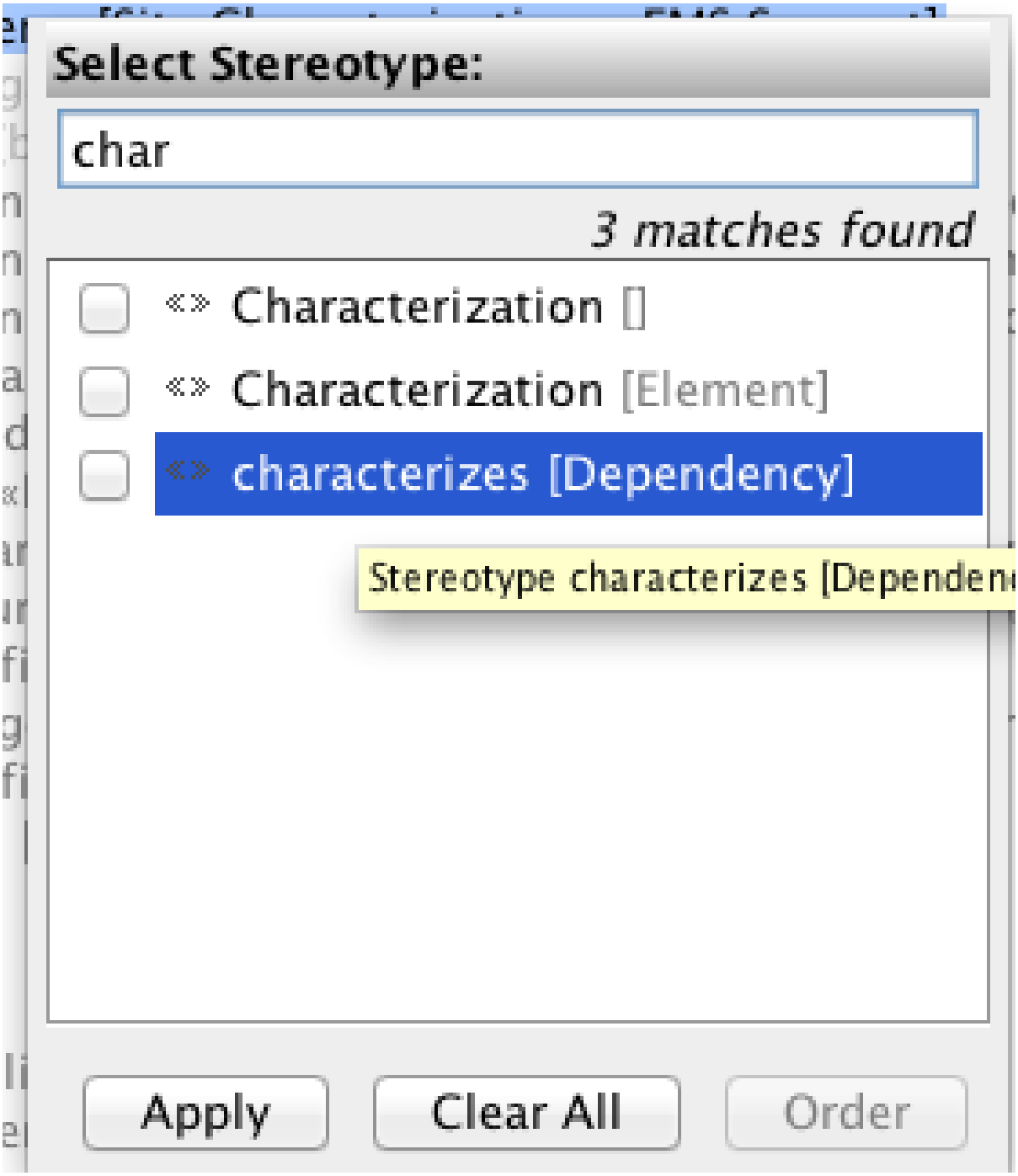


Click 'Ok'.

Next, stereotype the dependency by right-click -> Stereotype

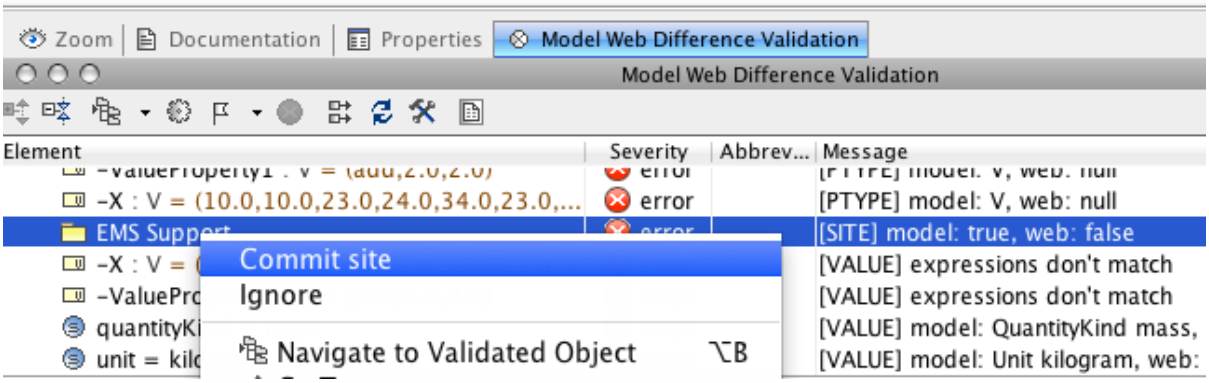


The new MD 18 Stereotype dialog will open, search for 'characterizes' and it will appear. Select it and click 'Apply'.



Finally, right-click the newly characterized package ->MMS->Validate SubModel

The validation window will open and within the results will be [SITE] model: true, web: false. Right-click on the violation and select Commit site



3.5.2. Managing EMS Project Workspaces in MagicDraw

EMS Projects in MagicDraw that are uploaded to NoMagic's Teamwork Server are capable of integrating with the EMS's Model Management suite of tools and applications. These tools are discussed in depth in the EMS Applications Manual's [cf:Model Management.vlink] section.

Workspaces on the EMS Server are handled in MagicDraw using Teamwork's branching and Project Merge capabilities. In order to connect a branch in MagicDraw with EMS, the user first need to create a task (e.g., branch) in EMS as discussed in EMS Application Manual. Once a task is created, the user need to create a corresponding branch on the MagicDraw Teamwork server.

If your project has adopted the use of tasks (e.g., workspace or branch) in EMS to protect the "Trunk" or "baseline" model, it is *strongly* recommended to use the following practices. *Note: Consult your Project Modeling P olicy for your project's model practices.*

1. Never make changes directly to the trunk after initial load to ems. Update the trunk only by merging a branch.

2. User should lock an entire models for edit when working in a branch. This prevents other users from making changes to the branch that may complicate merging process.

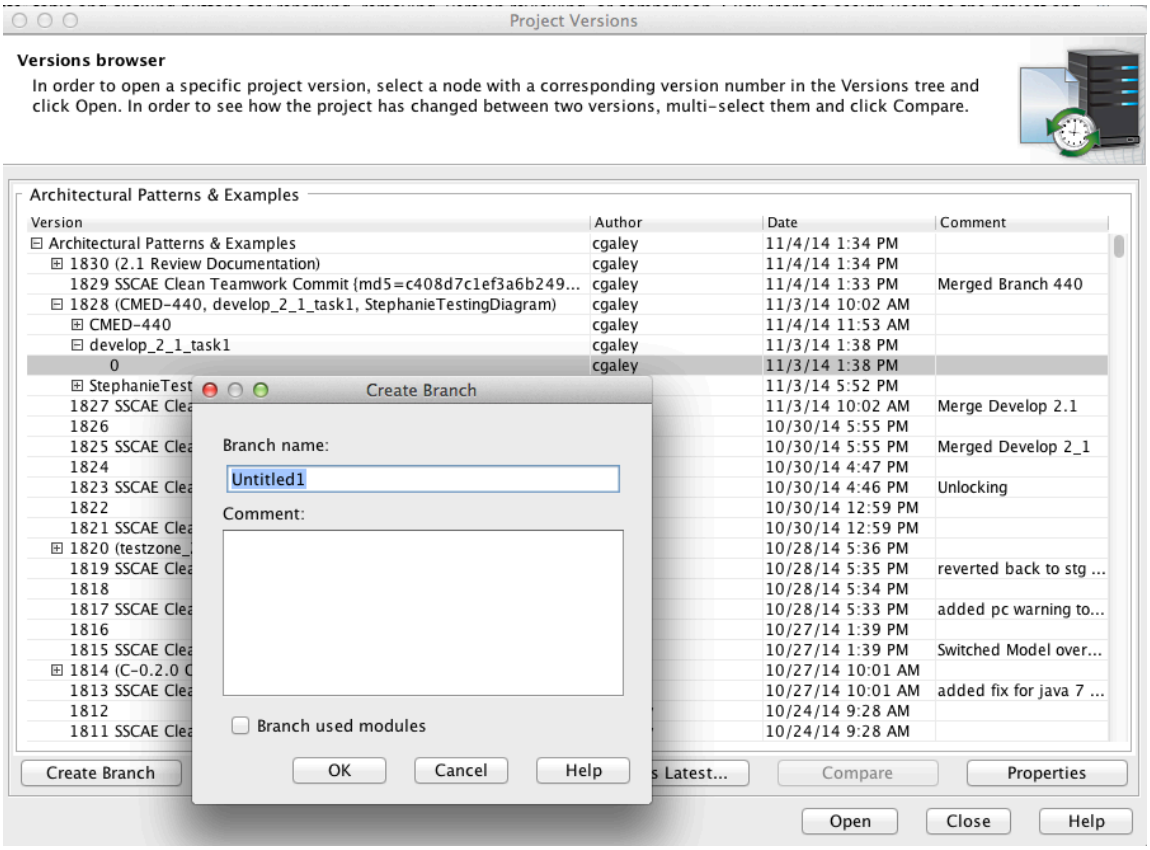
The following explains how to create branches in Teamwork server .

Note: These procedures are based on MD 18.0.

To create a branch:

Note: The ability to create a branch in TeamWork server is subject to project policy. To create a branch you must have "Administer project" access to the mdzip containing the EMS Project on teamwork. Consult your Project Modeling Policy document for more information on what procedure is required to create your branch.

1. Click "Collaborate..." in the upper menu of MagicDraw
2. Select "Projects..."
3. Select the project for which you want to create the branch and right-click "Project History"
4. Then click on the model version from which you want to create a branch. **Give the branch name exactly the same as the task (workspace) name in EMS** .
   - You must create the branch from the latest version to properly sync with EMS
   - If you are creating a MagicDraw branch for a sub-task you must create the branch from the latest version of that branch. Screenshot below shows creating a branch for a sub-task of develop_2_1. The branch hierarchy must match with the hierarchy of tasks (e.g., workspaces) in EMS.



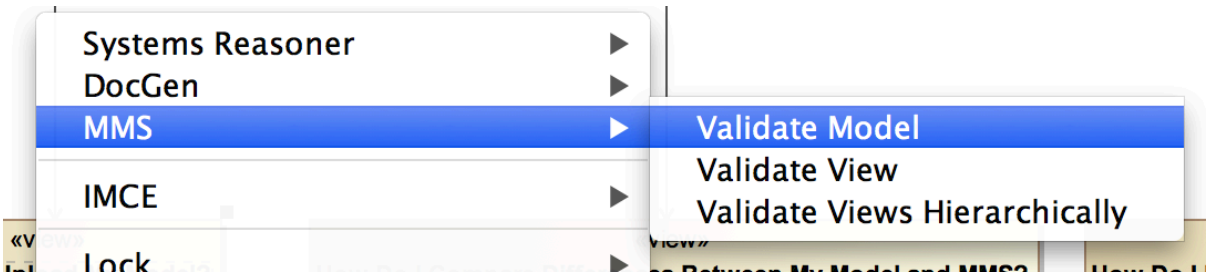
5. Once the branch is created, open it
6. Now the branch is ready to be synced with the corresponding task (workspace) in EMS.

The branch hierarchy on MagicDraw is expected to explicitly mirror the task workspace hierarchy on EMS. Otherwise, sync between MagicDraw and EMS server will not work properly.

3.5.3. Validate and Upload Model

Publish Documents

Once the model is uploaded to EMS server, documents in the model can be published. Note that documents are not automatically uploaded along with the model. This is because to allow the user to selectively upload documents. To publish a document, right click on the document element and select "MMS>Validate Views Hierarchically". Note that if you validate only the view, you will upload an empty document. You can then upload the rest later, but this is an extra step.



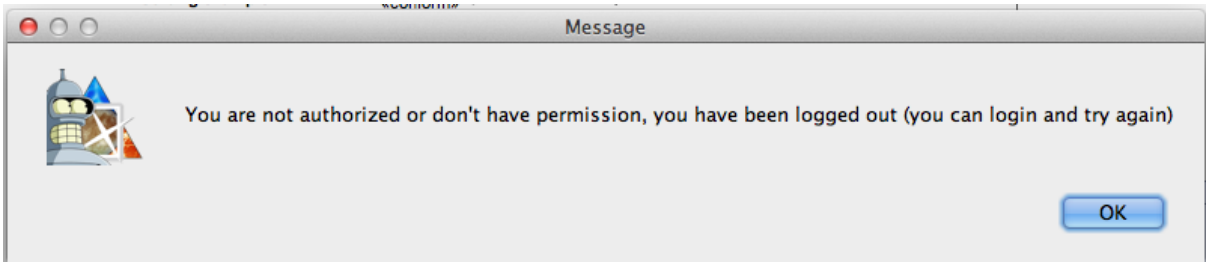
If the document is not already on the server the validation window will return with "[EXIST] This View doesn't exist on view editor yet". As before, right click on the error to show the options for reconciliation. The reconciliation options are:

- **Commit** : to send the MagicDraw model to the server.
  - **Commit with model elements**
  - **Commit without model elements**
  - **Commit recursively** : to commit everything hierarchically connected to the view as well)
- **Ignore** : This option lets you choose to ignore the issue, which does not change either the model or the View Editor.

Which option you chose is dependent upon your specific project or task. It never hurts to commit more when you are generating the document the first time. However, later on when working with your document and incorporating changes, you do want to avoid committing parts of the model that someone else changed.

To see all validation errors including ignored ones, use the right-most button on the Validation window's top bar and set it to "ALL" (vs. Ignored or Not Ignored). This is considered good engineering practice.

NOTE: If you do not have permission to add documents or edit that specific model on the server you will get the following error.



Please consult your task manager to address the permission issue. After you have resolved the reconciliation options and committed your document, it should now be visible and editable on the View Editor of EMS Server.

3.5.4. Merge with MDK Focus

Merging a Branch/Task:

Once a task has been completed a multi-step process is required to ensure all model elements are updated in both Magic Draw and EMS.

Note: The ability to merge a branch with the trunk is subject to project policy, consult your Project Modeling Policy document for more information on what procedure is required to merge your branch.

1. Obtain permission to merge the branch via project policy

2. Perform a Model Manager comparison and merge between the master and the branch as explained in the Model Manager manual (link)

3. Ensure that Magic Draw is open with the "Project Merge" plugin enabled at start-up (see MD Plugin Documentation for more information)

4. Open the source MagicDraw model (task/branch)

5. a. Initiate an EMS link

   b. Sync the model and views with the EMS following the procedures outlined in the MDK Manual (link)

6. Open the destination magic draw model (trunk, parent or other task/branch)

7. Navigate to "Tools->Project Merge"

8. Select 3-way merge and select the task branch to be merged

9. The ancestor should be populated automatically as the version of the trunk that the branch was based on

10. Click "Merge"

11. Use the MD tool accept/reject all conflicts (for more detailed instructions consult the MD user manual)

12. Click "Finish Merge"

13. Initiate an EMS connection on the destination branch/trunk

14. Sync the model and views with the EMS destination following the procedures outlined in the MDK Manual (link)

3.6. 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.

3.6.1. 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.

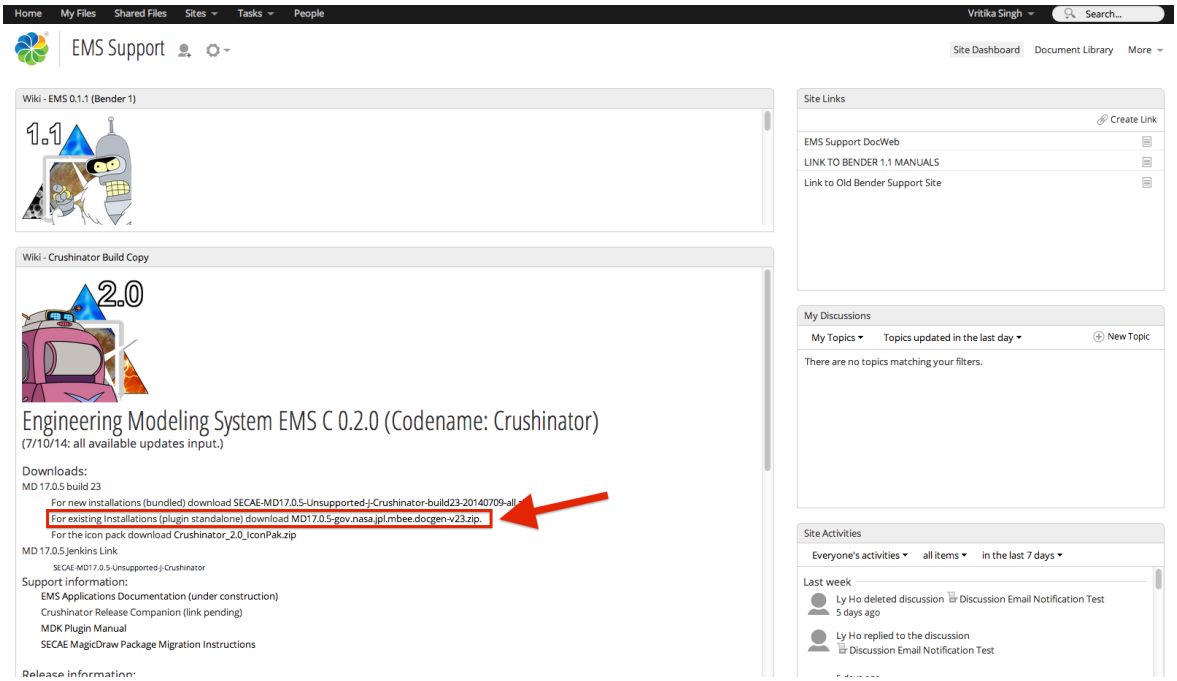
DocGen consists of: a UML profile with elements for creating a document framework; a set of scripts for traversing document frameworks, conducting analysis, and producing the output; and a set of tools to help users validate the correctness and completeness of their documents. Users need to invest the 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.

3.6.2. 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. 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 plugin (the latest will be on the EMS-Support site; for Crushinator on MD 17.0.5, the file is called MD17.0.5-gov.nasa.jpl.mbee.docgen-v74.zip). **Do not unzip the file.**



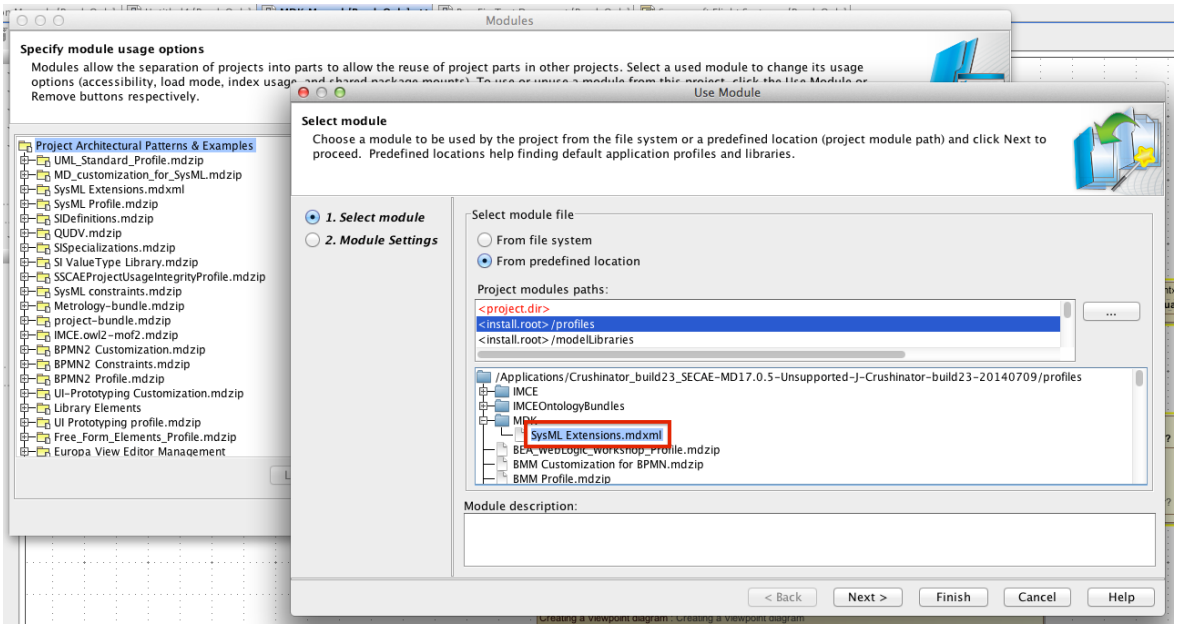
2. Start MagicDraw, go to Help->Resource/Plugin Manager

3. 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)

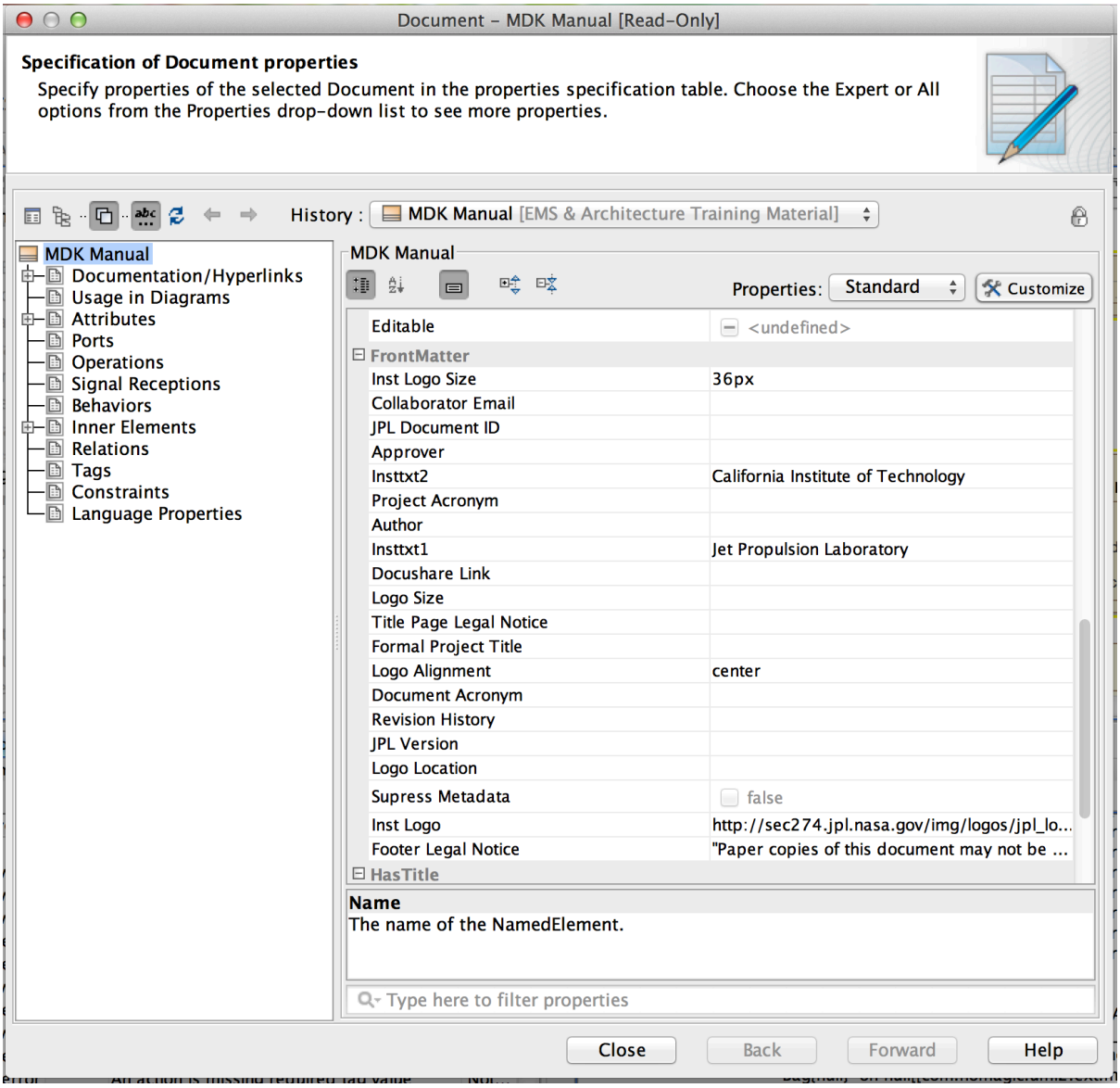


3. Now you can use the stereotypes from Document Profile, inside DocGen>MDK EMP Client

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.

3.6.3. 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. With the inputs shown in the image above, the output would have a front page that looks as shown below:

MDK Manual

Paper copies of this document may not be current and should not be relied on for official purposes. The current version is availible from online at:

"This Document has not been reviewed for export control. Not for distribution to or access by foreign persons."

Wed Aug 06 10:55:13 PDT 2014
JPL D-

Jet Propulsion Laboratory
California Institute of Technology

A reference card for the tags is shown below.

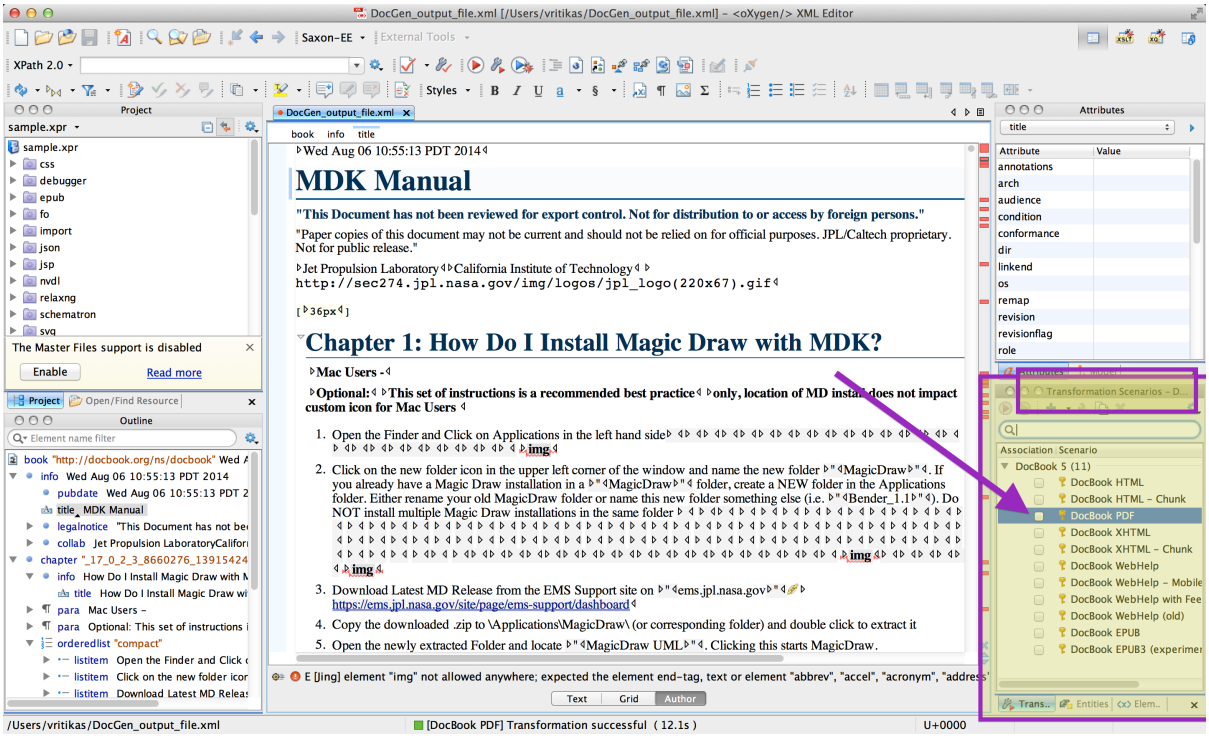
Tag Name	Location	Description	Example
Document Title	Name of Stereotyped Package	This is the title of the document and is set by the name of the documentview stereotyped package name	Stylesheet Documentation
Date	Date and time of generation	This is the date used on the title page and header, it is set by the system when docgen is run	Wed Aug 21 08:31:35 PDT 2013
Logo Size	DocumentView,FrontMatter	Project Logo size on front page	"4in"
Concurrance	DocumentView,FrontMatter	Concurring authors for signature page, multiples are supported	"Firstname,Lastname,jobtitle,organization,section"
Revision History	DocumentView,FrontMatter	Insert revision histories for the page after signatures, multiples are supported	"VERSION|DateEdited|EditorFirstName|EditorLastName|Changes"
Project Acronym	DocumentView,FrontMatter	Short Project name for use in header (Acronym is best)	MEA
Formal Project Title	DocumentView,FrontMatter	Full Project name for use in title page	Modelling Early Adoptors
Docushare Link	DocumentView,FrontMatter	link to where this file will be hosted	http://mbse.jpl.nasa.gov
Approver	DocumentView,FrontMatter	Approving authors for signature page, multiples are supported	"Firstname,Lastname,jobtitle,organization,section"
Title Page Legal Notice	DocumentView,FrontMatter	Legal notice to apper on the fron page	JPL/Caltech Proprietary. Competition Sensitive.
Use Default Stylesheet	DocumentView,FrontMatter	Depreciated, allowed users to select the old stylesheet	False/True
JPL Version	DocumentView,FrontMatter	Version that the document is currently in	DRAFT
Collaborator Email	DocumentView,FrontMatter	distribution email list (if needed), multiples are supported	email@thatguy.com
Logo Alignment	DocumentView,FrontMatter	For smaller logos or specific projects the alighment of the project patch/logo might want to be changed	
Footer Legal Notice	DocumentView,FrontMatter	Legal notice to apper in the footer of every page. Will be italiczied	Paper copies of this document may not be current and should not be relied on for official purposes. JPL/Caltech proprietary. Not for public release.
JPL Document ID	DocumentView,FrontMatter	JPL Document ID for front page and header (will not impact the document if left blank)	JPL D-1234
Document Acronym	DocumentView,FrontMatter	Short name for the document. Used in header along with project acronym (ie MEA V&V)	V&V
Logo Location	DocumentView,FrontMatter	URL/local path where the project patch/logo is stored	http://your logo location -or- users/username/Your logo location
Author	DocumentView,FrontMatter	Author information for signature page, multiples are supported	"Firstname,Lastname,jobtitle,organization,section"
CoverImage	DocumentView,DocumentMeta	URL/local path where a SBU or Classified cover is stored	http://your cover location -or- users/username/Your cover location

3.6.4. Generate Document

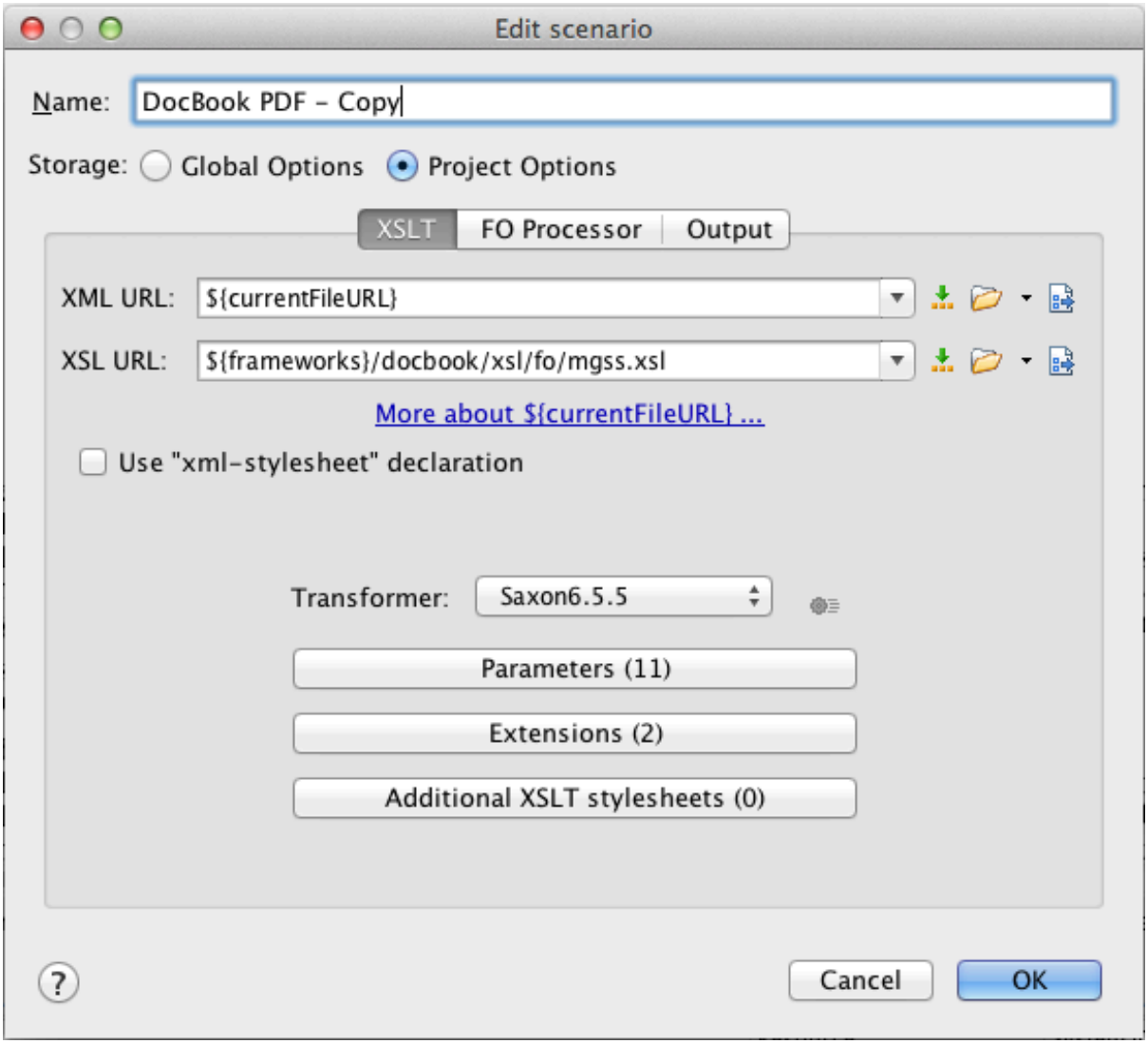
DocGen allow generating a local version of your document without access to EMS server. Select a document, and use a right click menu "DocGen>Generate DocGen3 Document". It will generate a DocBook XML file. Check with your project to see if there is a preferred method of generation. Most practitioners 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.

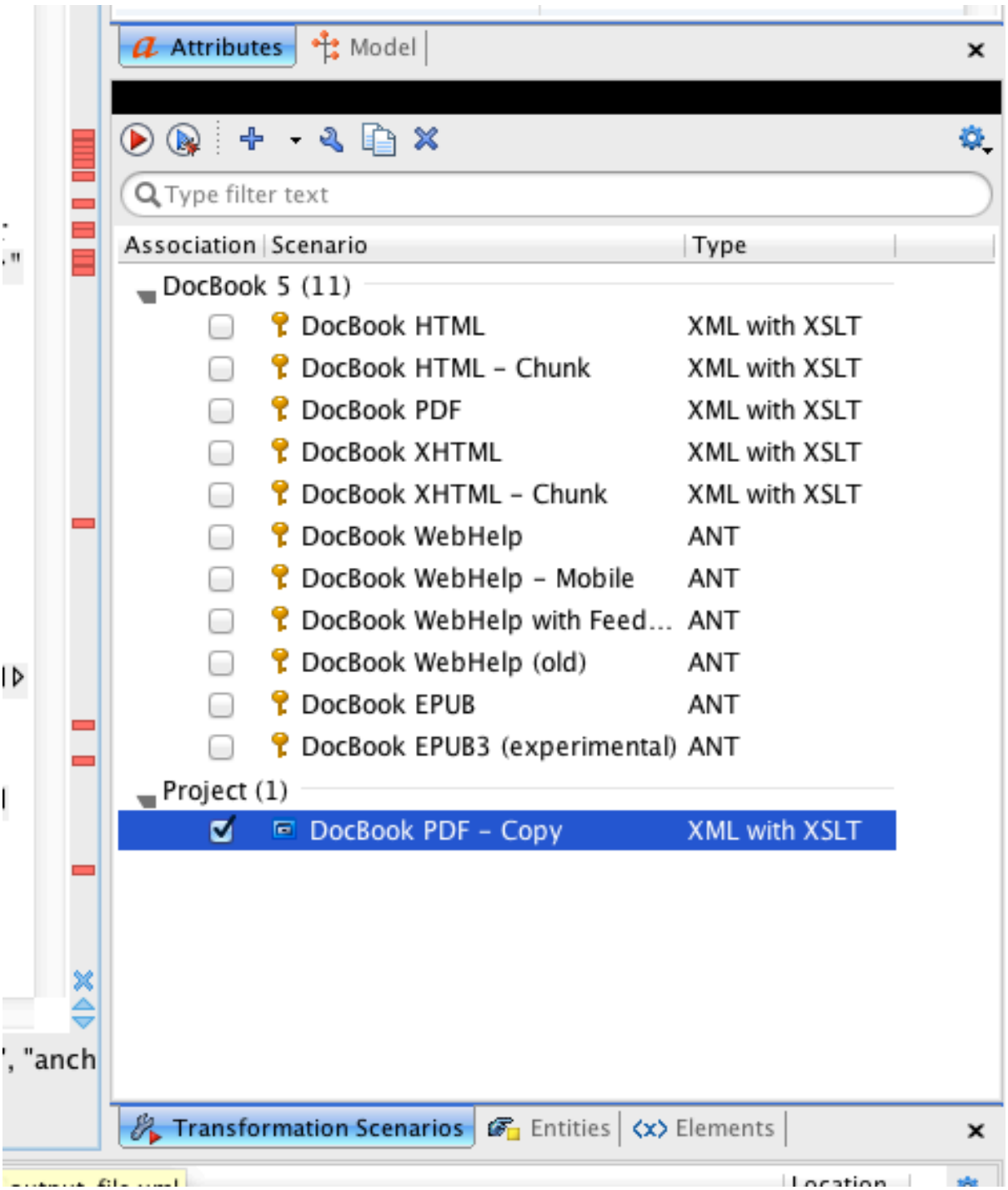
Now you can open your saved .xml output file with oXgen. After opening your file, on the bottom right hand side, right click the "DocBook PDF" option within transformation scenarios, as pictured below.



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 (at the bottom, shown under subsection "Project," as visible in the picture below). You can also click the red "play" button if your DocBook PDF is selected as shown below to generate a PDF.



3.7. 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.

3.7.1. What Is OCL and Why Do I Use It?

OCL Primer By Bradley Clement

What Is OCL?

The Object Constraint Language (OCL) is a text language (a subset of 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.oclAsType(LiteralString)

.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 .

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

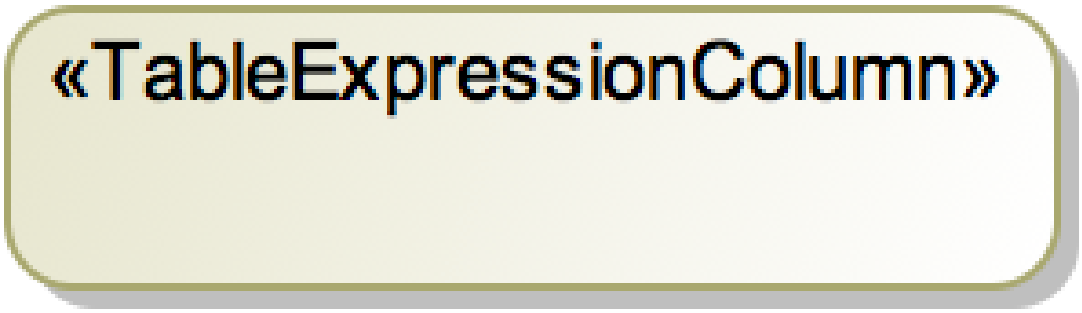
For help with OCL, you can try the OCL Cheat Sheet or search the web for example OCL. Some examples will also be given in the sections below. The OCL Cheat Sheet is very helpful once you get a basic understanding of how OCL works, however, until you have gained some understanding it might be confusing. 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.

3.7.2. 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.

Operation	Description
=	True if *self* and the argument are the same
<>	True if *self* and the argument are not the same
oclIsNew()	True if *sel* was created during the operation
oclIsUndefined()	True if *self* is undefined
oclAsType(type)	*self* as of the given type, *type*
oclIsTypeOf(type)	True if *self* is an instance of the given type, *type*
oclIsKindOf(type)	True if *self* conforms to the given type, *type*
oclisInState(state)	True if *self* is in the given state, *state*
T::allInstance()	Set of all instances of the type *T*

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

The model below is used in the following sections to demonstrate the various capabilities of the black box expressions.

Figure 3.26. Animals

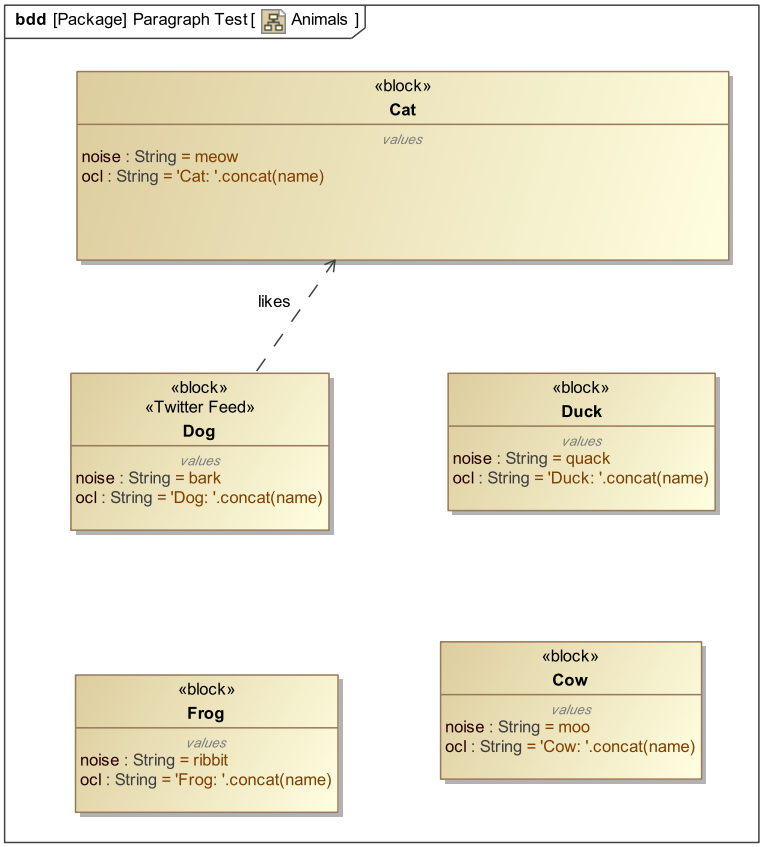


Diagram name	Animals
Author	igomes
Creation date	6/4/14 4:06 PM
Modification date	9/25/14 6:19 PM
Last modified by	cgaley

3.7.2.1. 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.

Figure 3.27. Animals

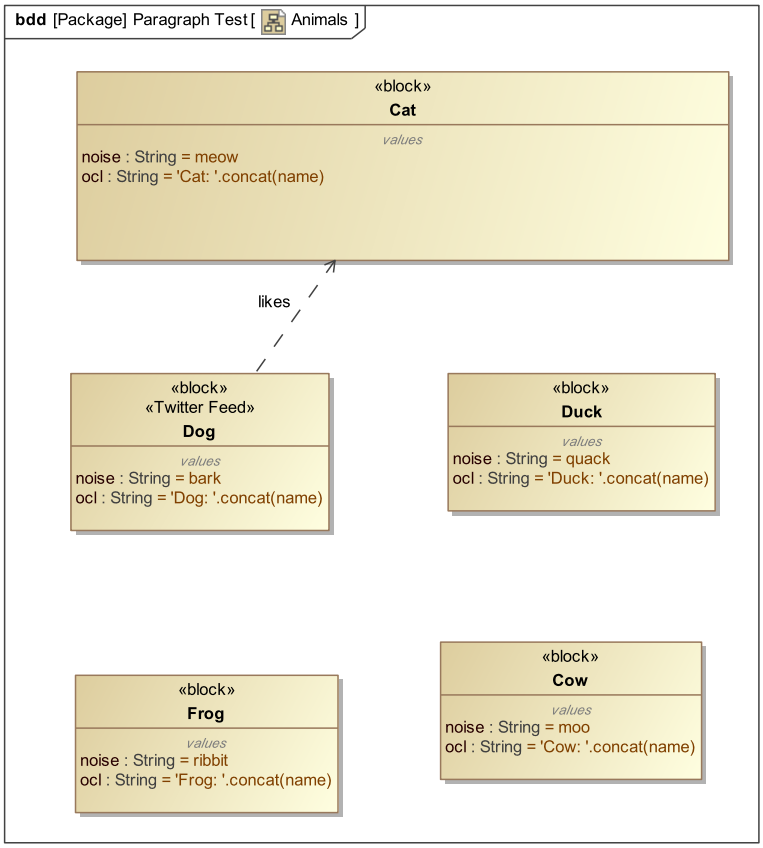


Diagram name	Animals
Author	igomes
Creation date	6/4/14 4:06 PM
Modification date	9/25/14 6:19 PM
Last modified by	cgaley

Figure 3.28. Unit Relationship Collect

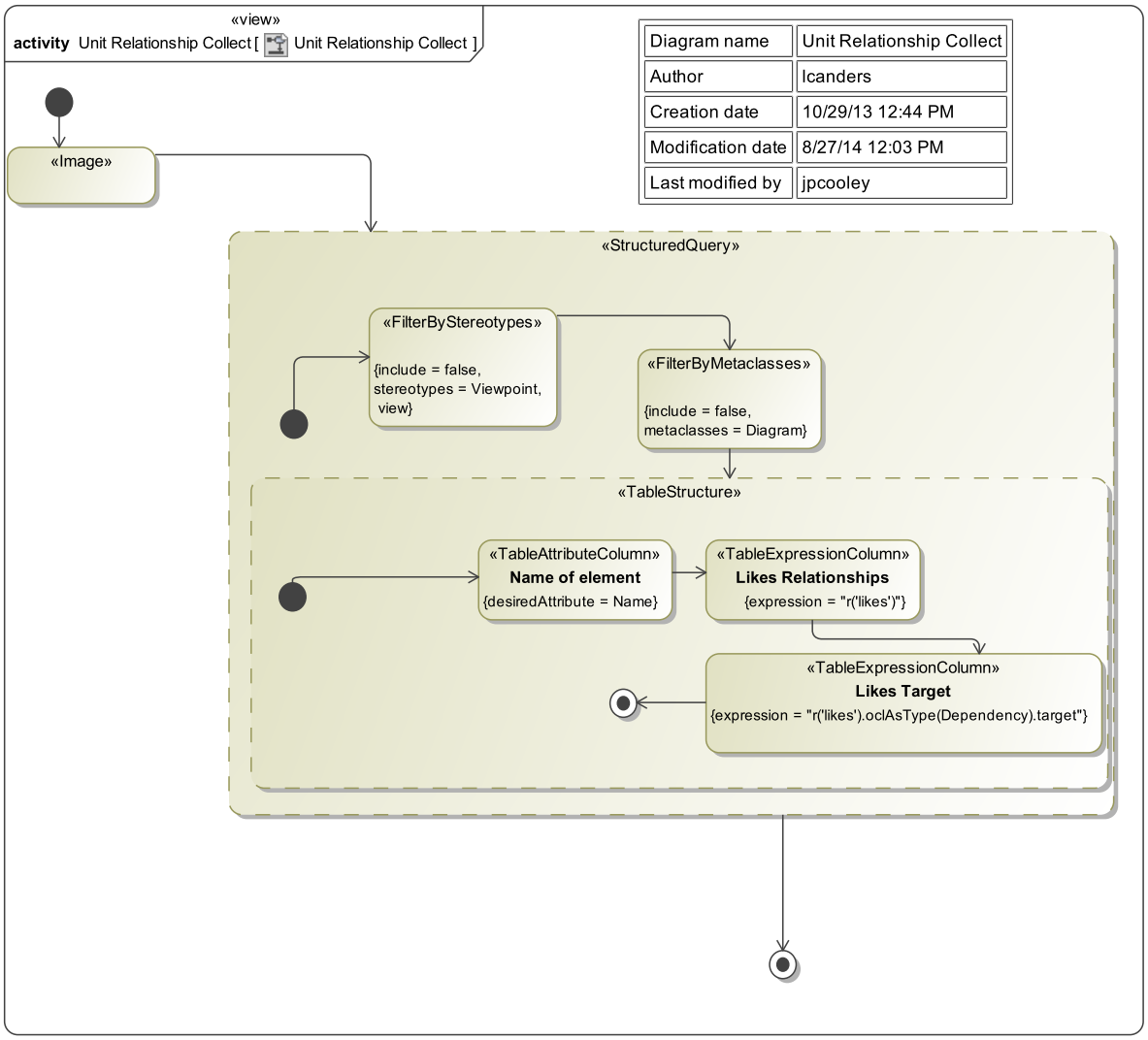


Table 3.6.

Name of element	Likes Relationships	Likes Target
Dog	likes	Cat

3.7.2.2. 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."

see table below.

Figure 3.29. Animals

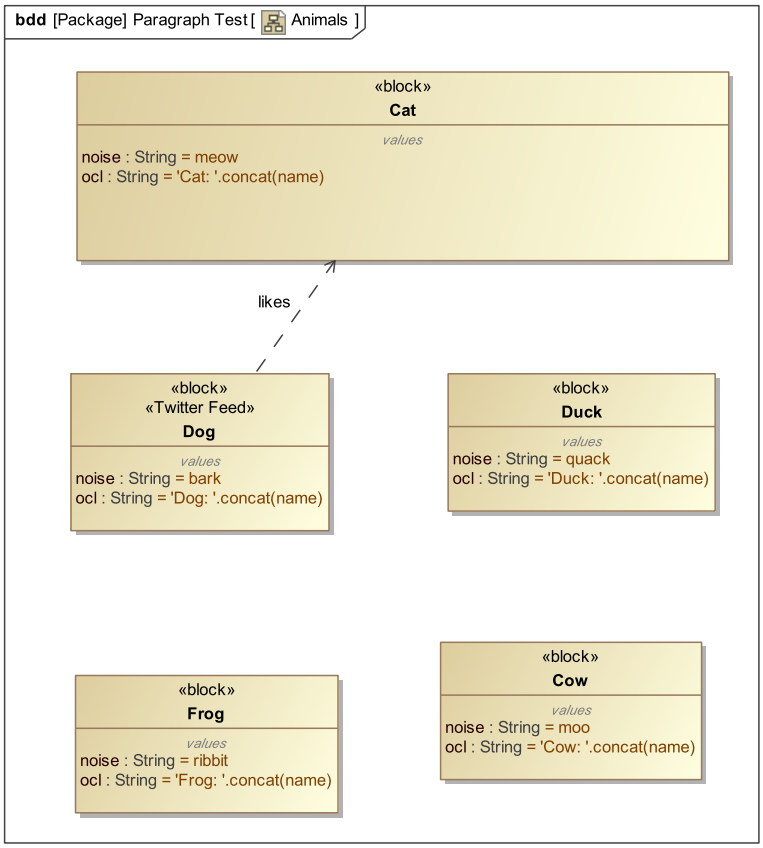


Diagram name	Animals
Author	igomes
Creation date	6/4/14 4:06 PM
Modification date	9/25/14 6:19 PM
Last modified by	cgaley

Figure 3.30. Unit Name Display

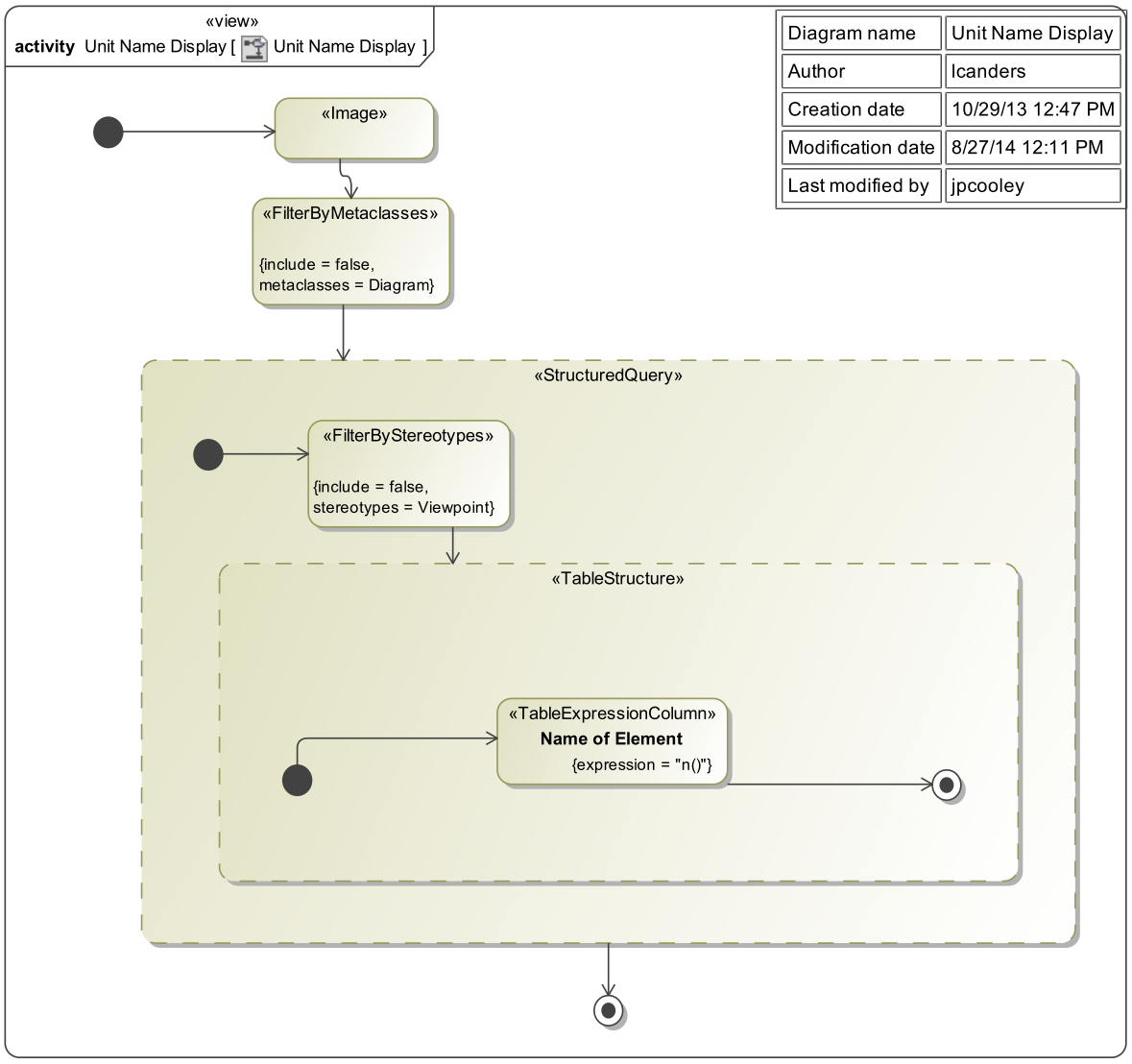


Table 3.7.

Name of Element
Dog

3.7.2.3. 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 the table below.

Figure 3.31. Animals

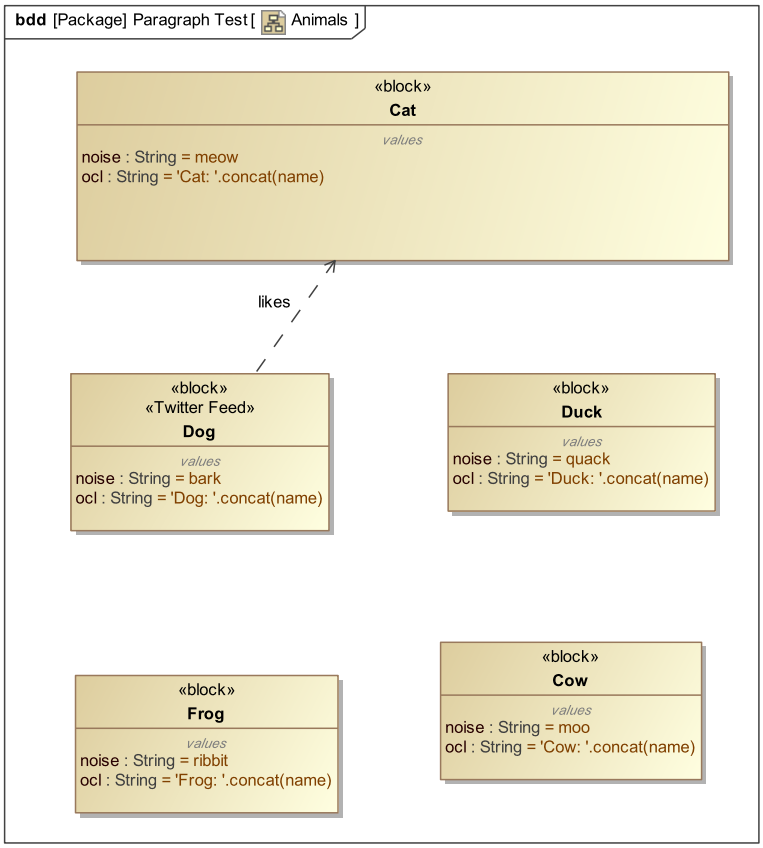


Diagram name	Animals
Author	igomes
Creation date	6/4/14 4:06 PM
Modification date	9/25/14 6:19 PM
Last modified by	cgaley

Figure 3.32. Unit Stereotype Display

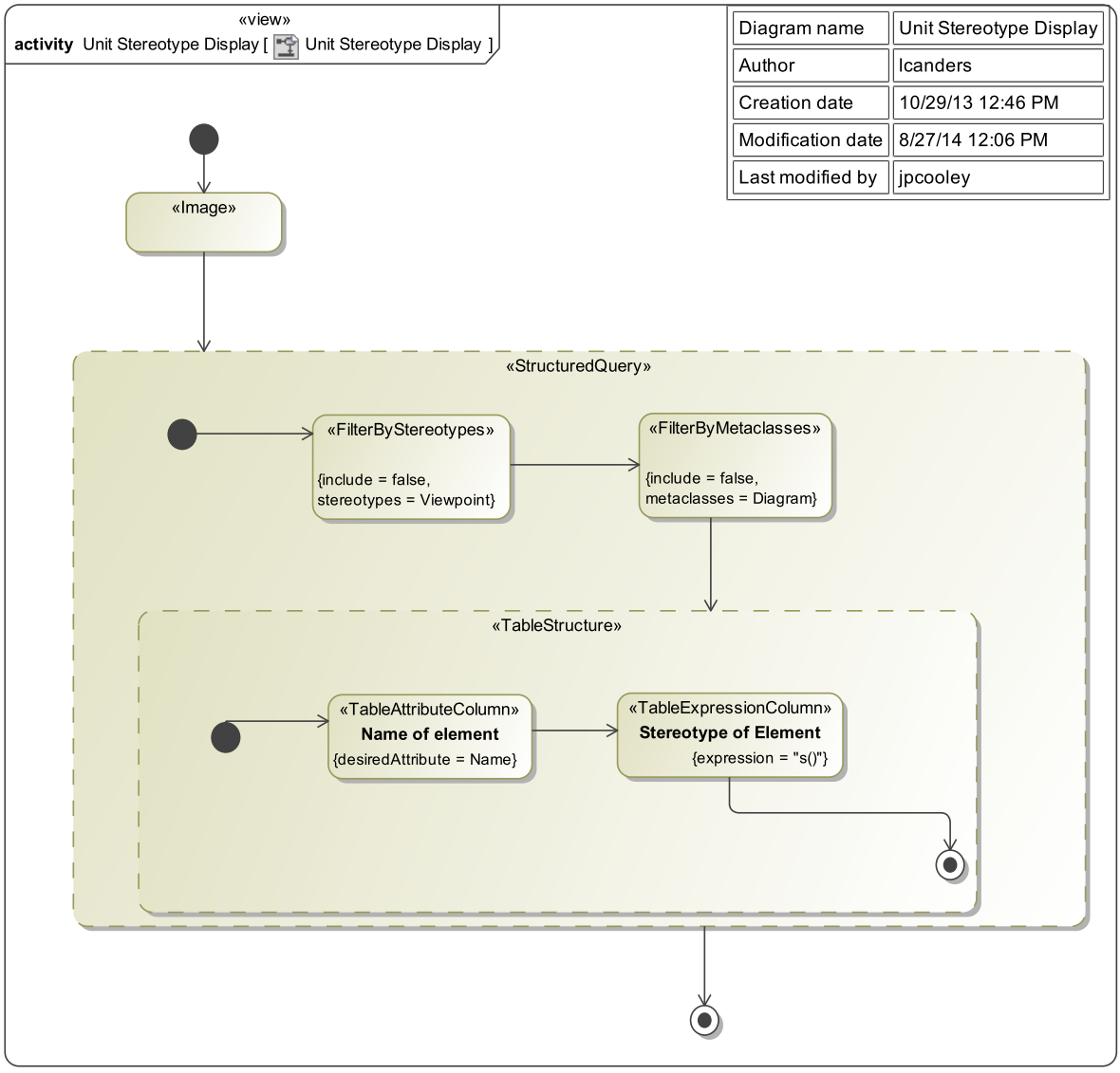


Table 3.8.

Name of element	Stereotype of Element
Dog	Block Twitter Feed

3.7.2.4. 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 the table below.
- 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** .

Figure 3.33. Animals

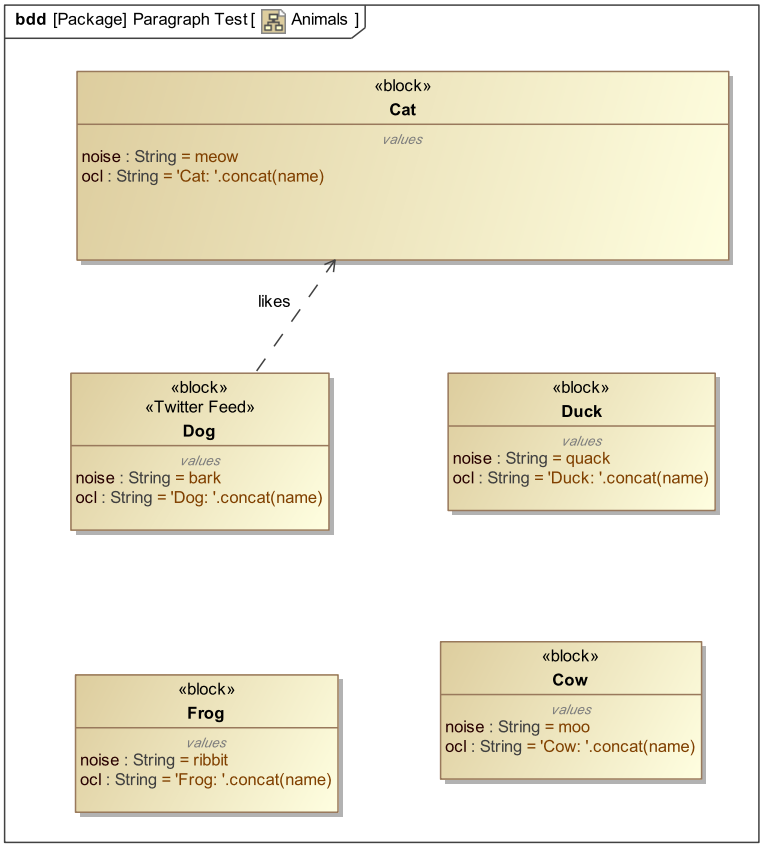


Diagram name	Animals
Author	igomes
Creation date	6/4/14 4:06 PM
Modification date	9/25/14 6:19 PM
Last modified by	cgaley

Figure 3.34. Members Black Box Demo

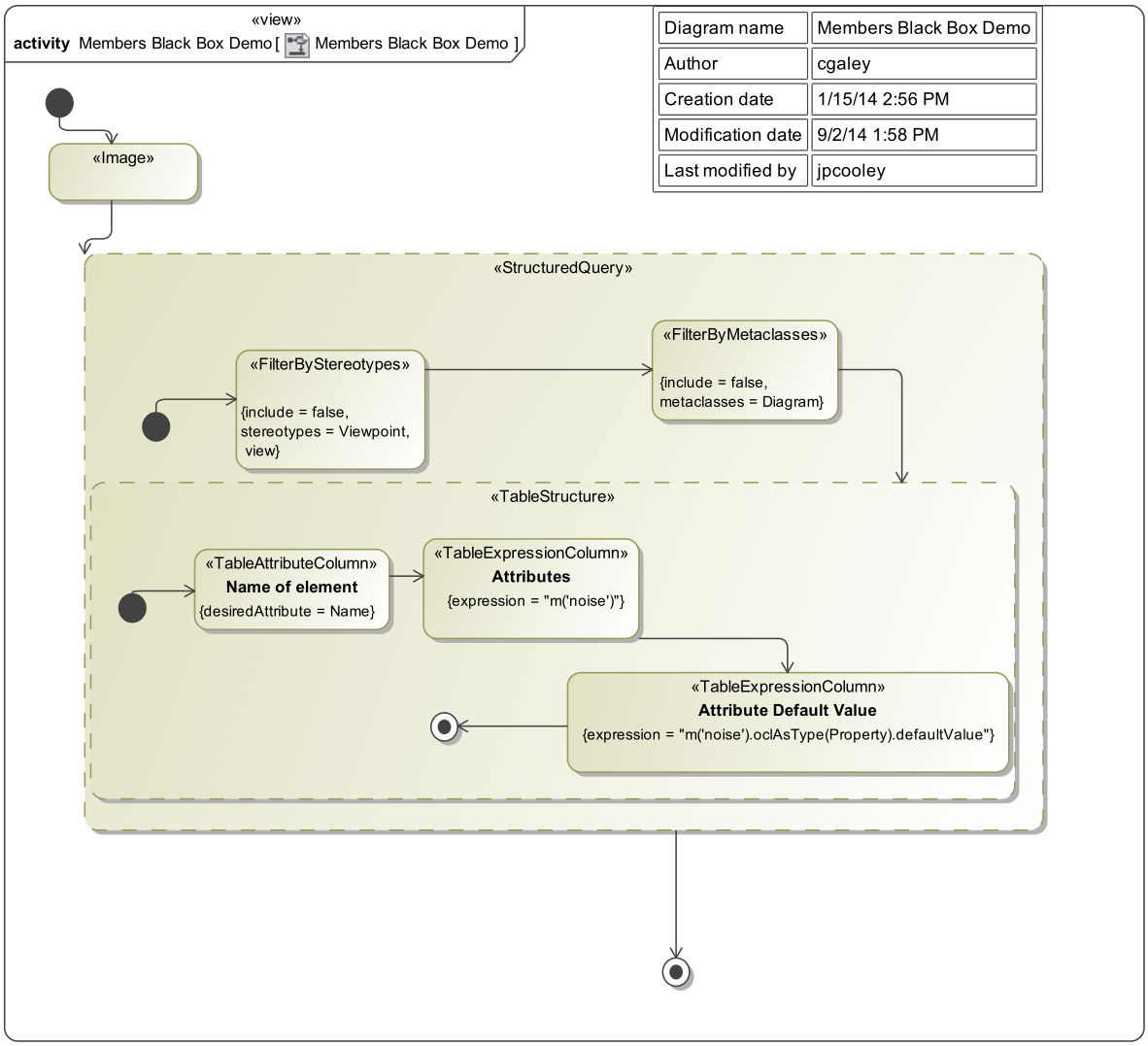


Table 3.9.

Name of element	Attributes	Attribute Default Value
Dog	noise	bark

3.7.2.5. 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>>.

Results shown below.

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

Figure 3.35. Animals

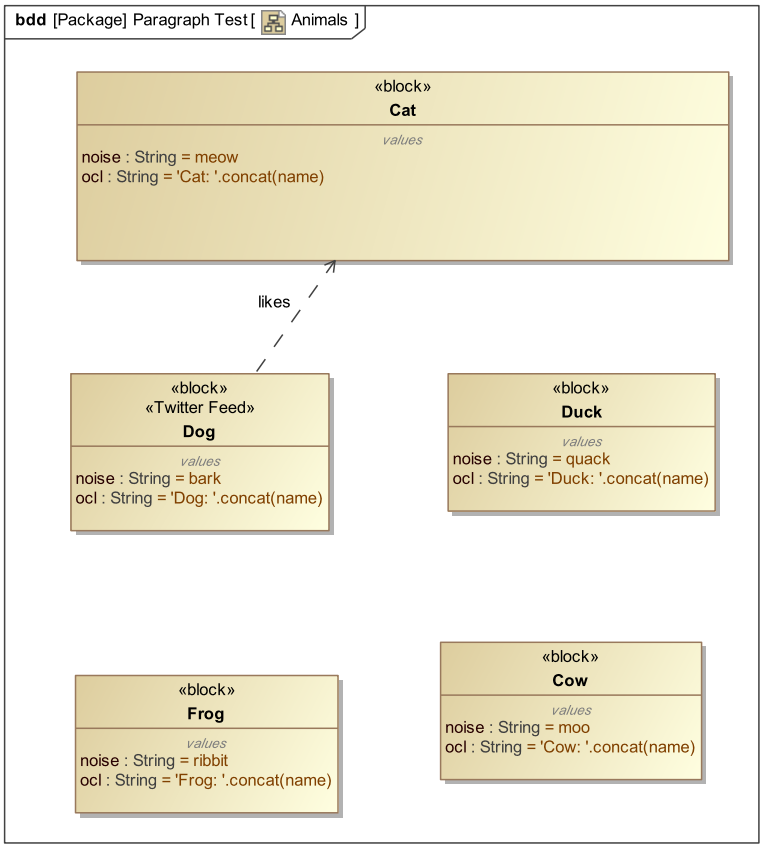


Diagram name	Animals
Author	igomes
Creation date	6/4/14 4:06 PM
Modification date	9/25/14 6:19 PM
Last modified by	cgaley

Figure 3.36. Types Black Box Demo

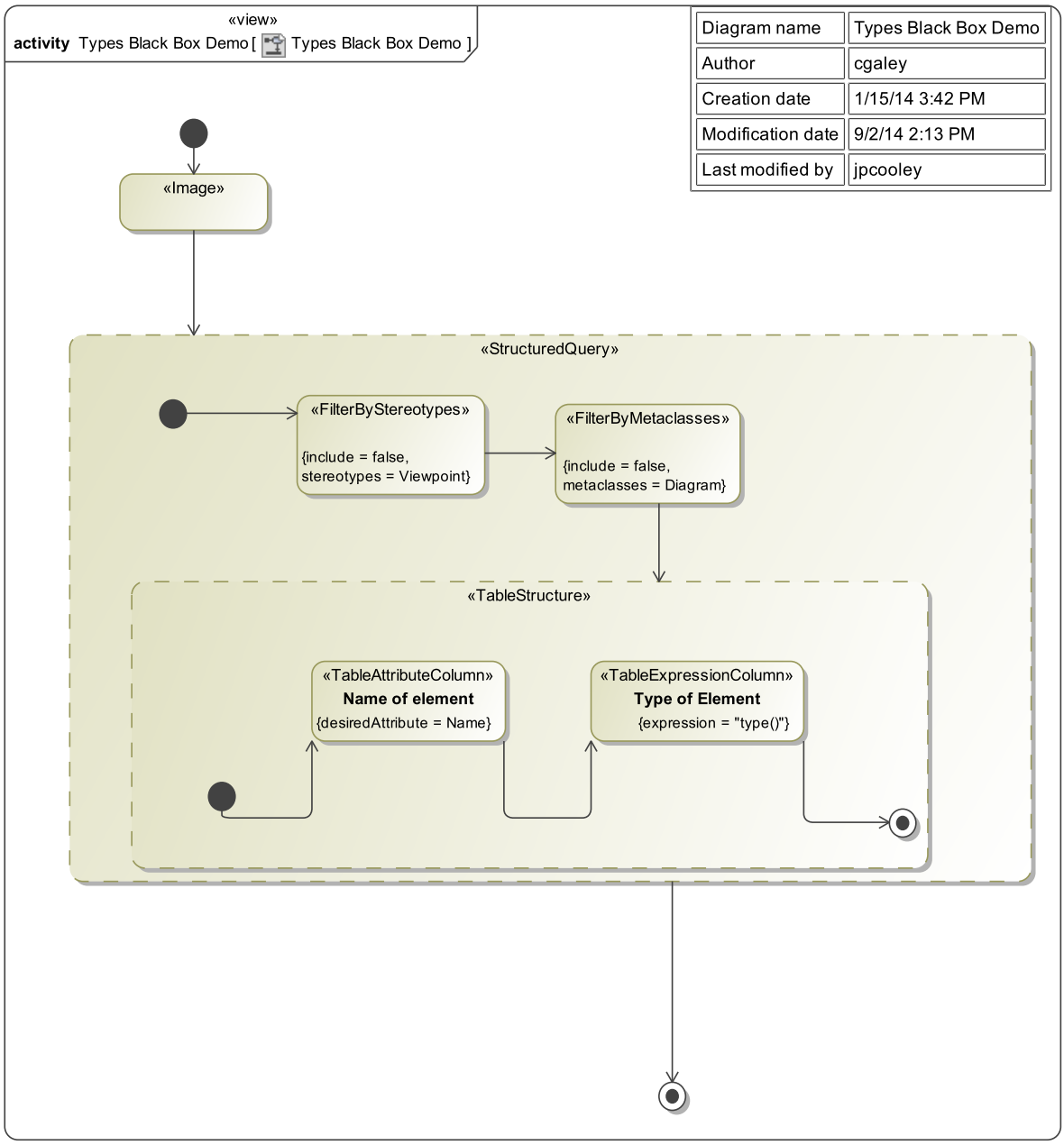


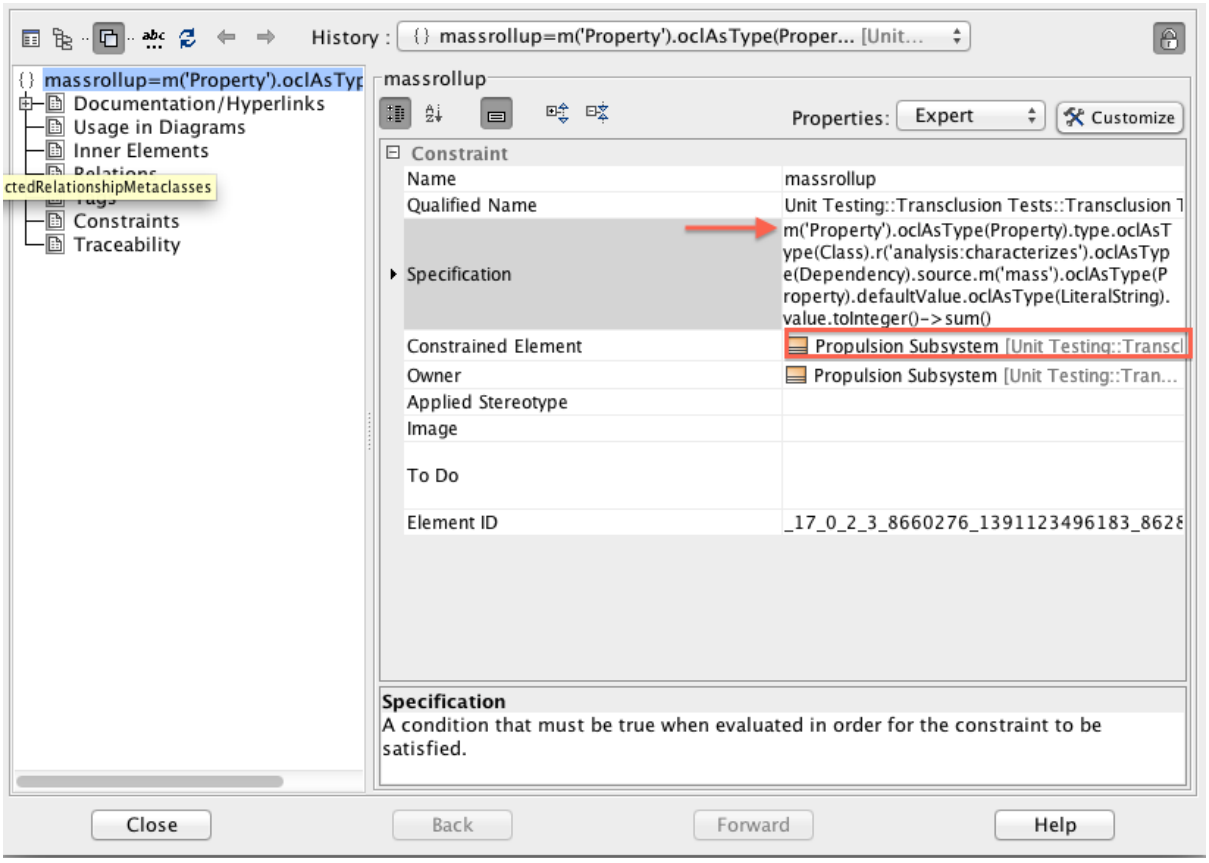
Table 3.10.

Name of element	Type of Element
Dog	interface com.nomagic.uml2.ext.magicdraw.classes.mdkernel.Class

3.7.2.6. Evaluate "eval()"

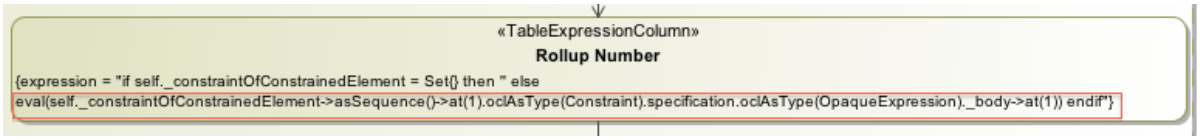
eval() or **e()** evaluates an OCL expression retrieved from an element. This example may seem overwhelmingly complex, however, the main point is to demonstrate the use of the black box expression. The image below shows an element (Propulsion Subsystem) which has a set constraint (red arrow).

This is a constrained element with the constraint specification:



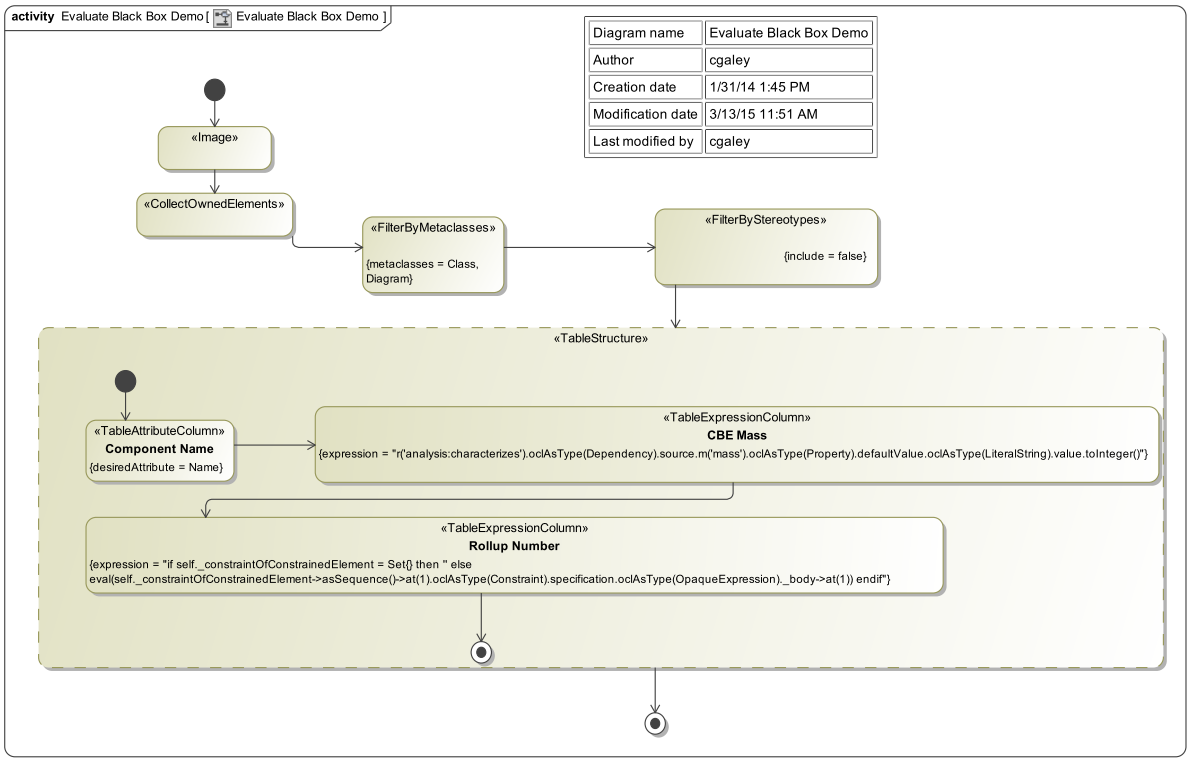
The example expression is long but take the time to try and piece it together to help your understanding. A useful hint is to start with the oclAsType() operations since these will give you a clue as to what was returned by the previous operation.

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



This is the complete viewpoint method diagram:

Figure 3.37. Evaluate Black Box Demo



This is the output of the complete method based on the specified constraints:

Table 3.11.

Component Name	CBE Mass	Rollup Number
Propulsion Subsystem		0
Launch Mass		
Steady-State Power		
On		
Off		
Standby		
Hydrazine Tank		
Launch Mass		
Steady-State Power		
On		
Off		
Standby		
Thruster		
Launch Mass		
Steady-State Power		
On		
Off		
Standby		

3.7.2.7. 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. The results are displayed at the bottom of the page.

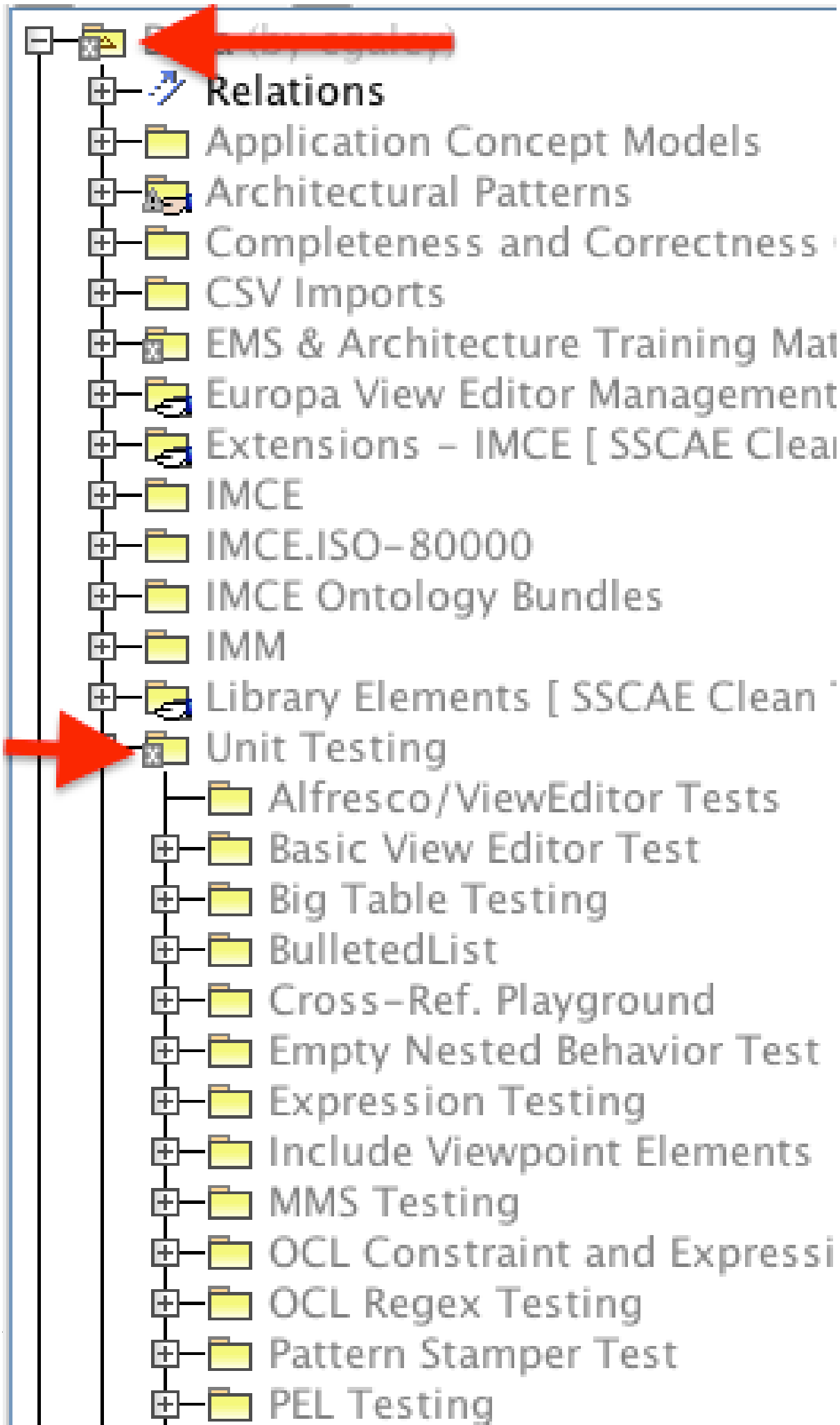

Relations
Application Concept Models
Architectural Patterns
Completeness and Correctness
CSV Imports
EMS & Architecture Training Mat
Europa View Editor Management
Extensions - IMCE [SSCAE Clea
IMCE
IMCE.ISO-80000
IMCE Ontology Bundles
IMM
Library Elements [SSCAE Clean
Unit Testing
Alfresco/ViewEditor Tests
Basic View Editor Test
Big Table Testing
BulletedList
Cross-Ref. Playground
Empty Nested Behavior Test
Expression Testing
Include Viewpoint Elements
MMS Testing
OCL Constraint and Expressi
OCL Regex Testing
Pattern Stamper Test
PEL Testing

Figure 3.38. Owners Black Box Demo

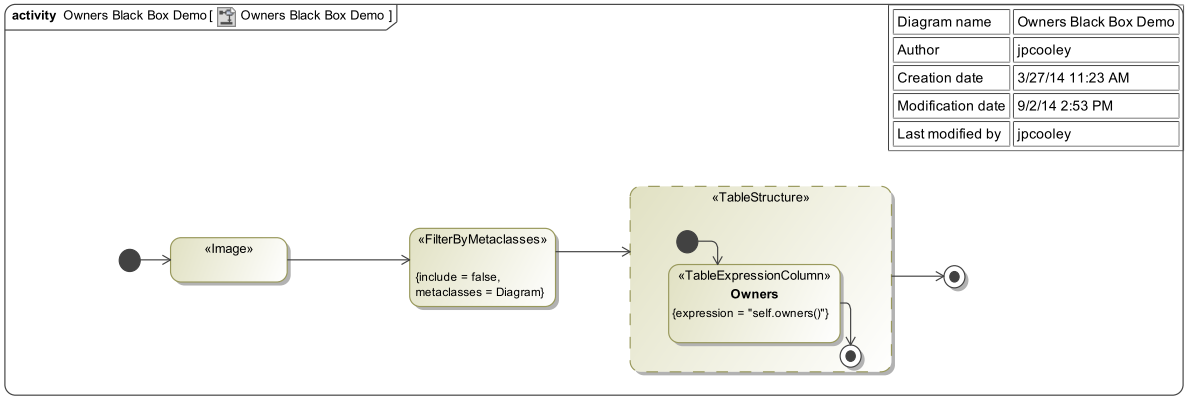


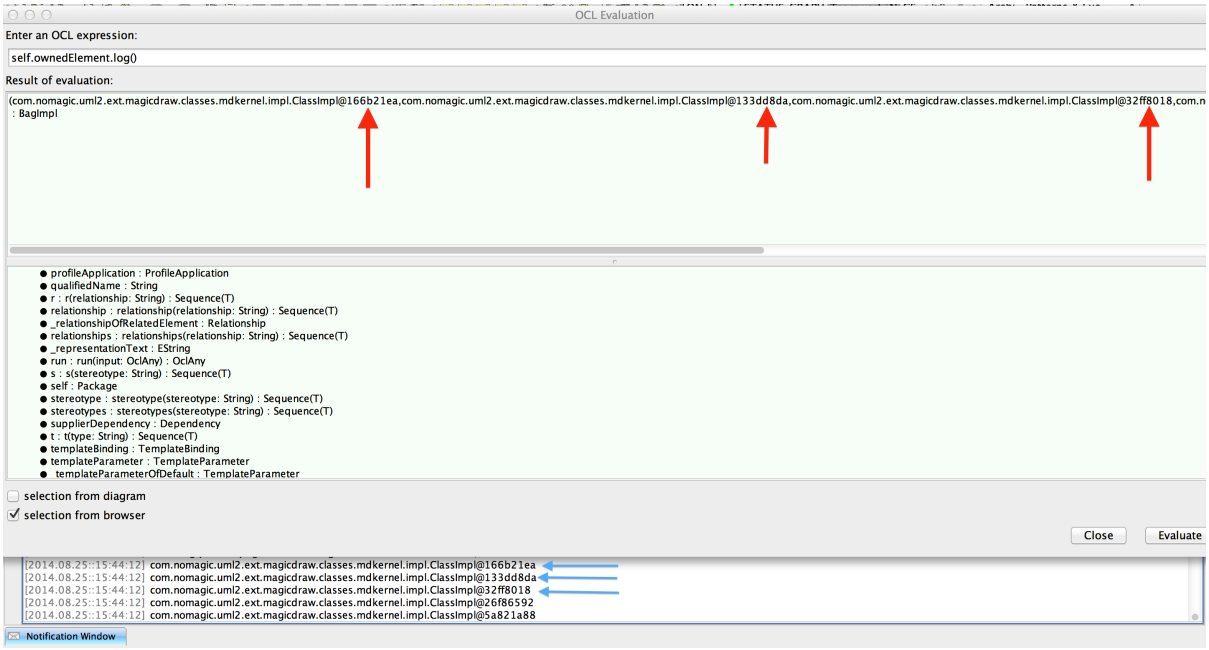
Table 3.12.

Owners
Animals
Paragraph Test
Viewpoint Operations
Unit Testing
Components
EMS Support
Data

3.7.2.8. 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.



3.7.2.9. 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. Results shown below.

Figure 3.39. Run Black Box Demo

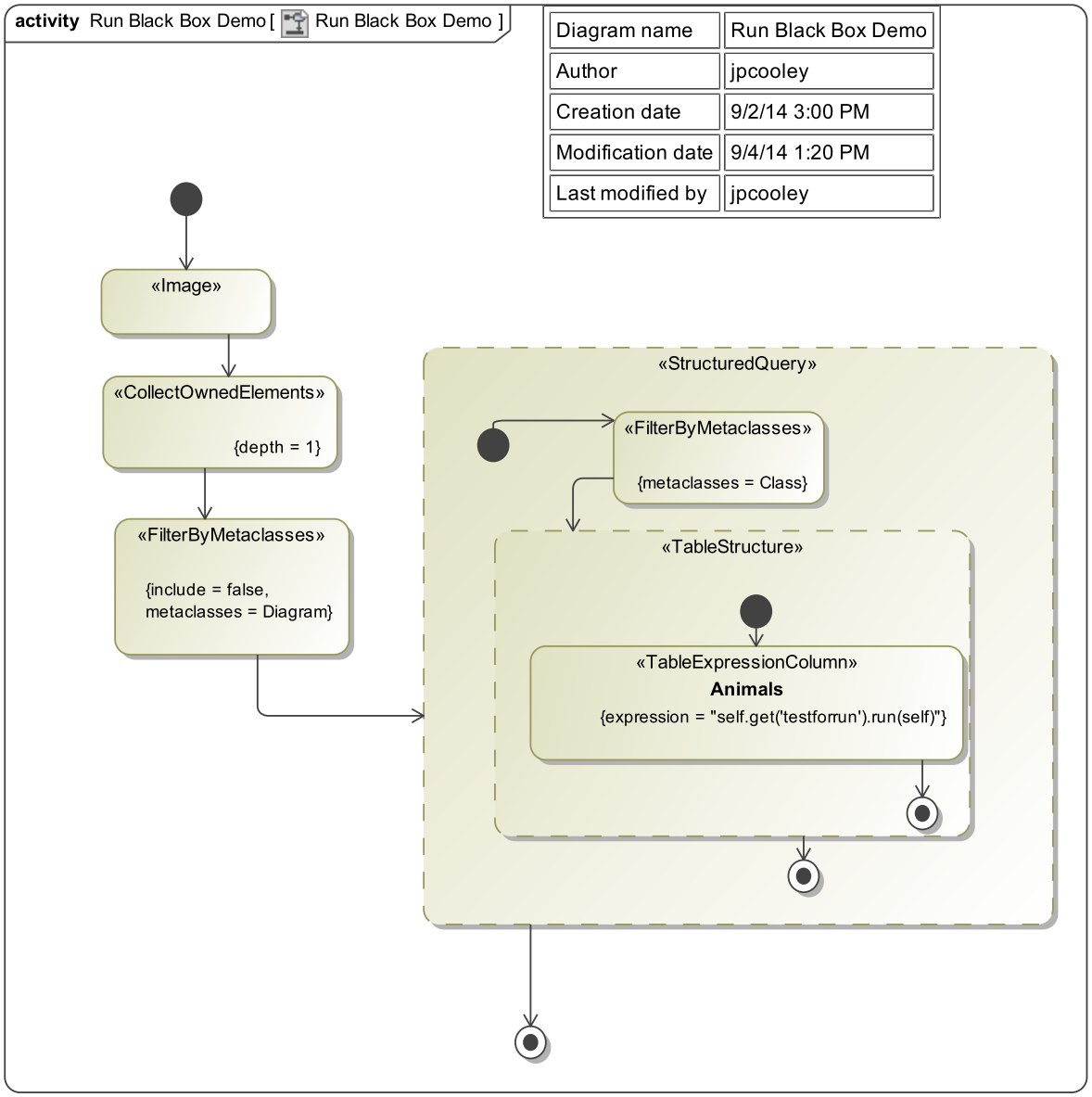


Figure 3.40. testforrun

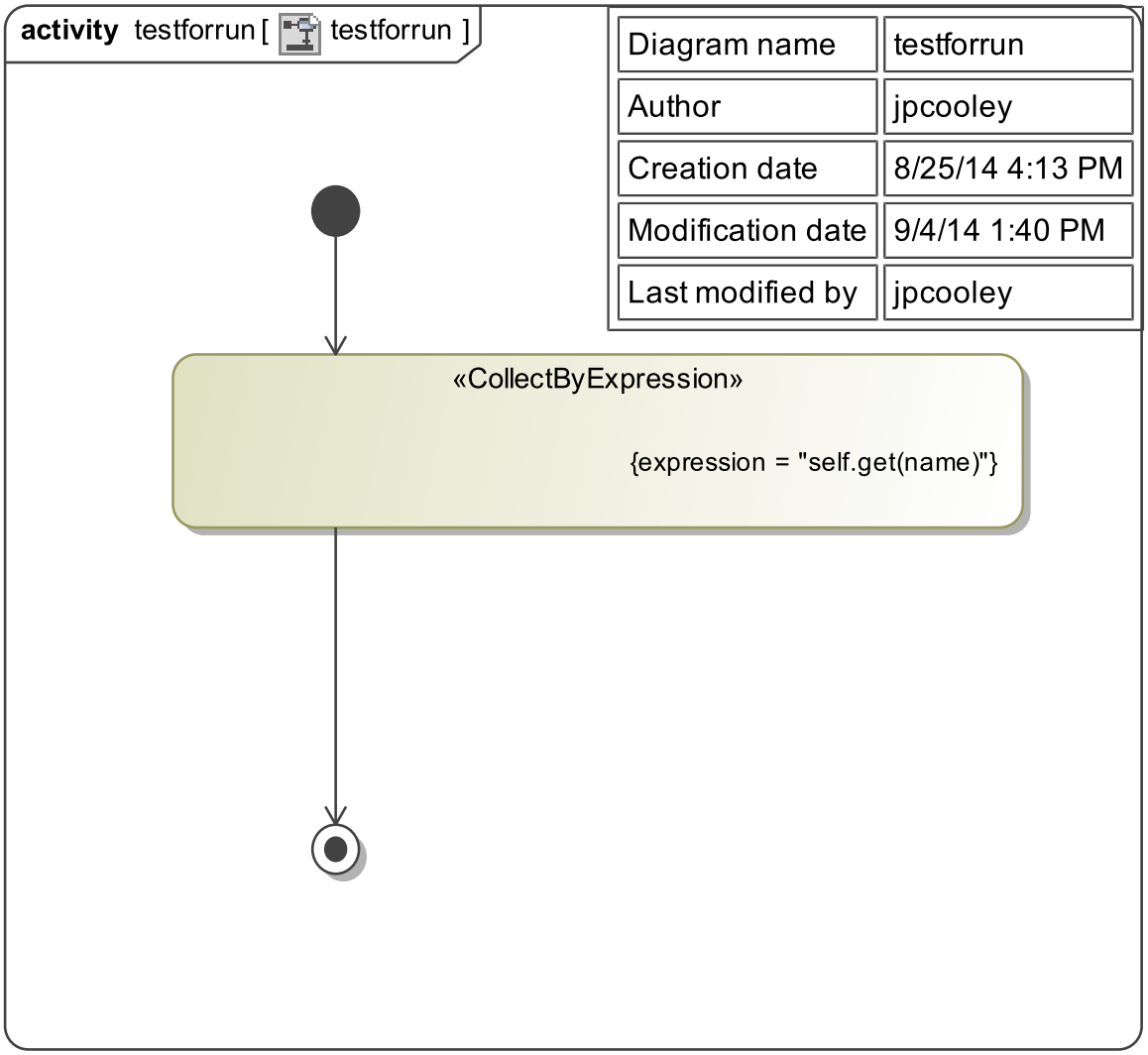


Table 3.13.

Animals
Cat
Duck
Cow
Frog
Dog

3.7.2.10. 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. Results shown below.

Figure 3.41. Animals

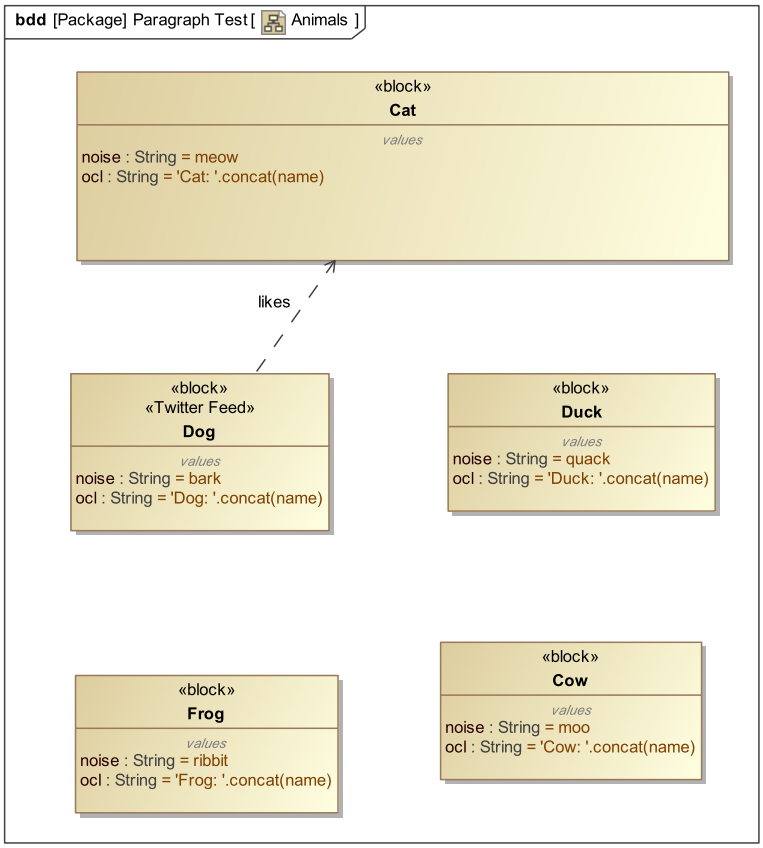


Diagram name	Animals
Author	igomes
Creation date	6/4/14 4:06 PM
Modification date	9/25/14 6:19 PM
Last modified by	cgaley

Figure 3.42. Value Black Box Demo

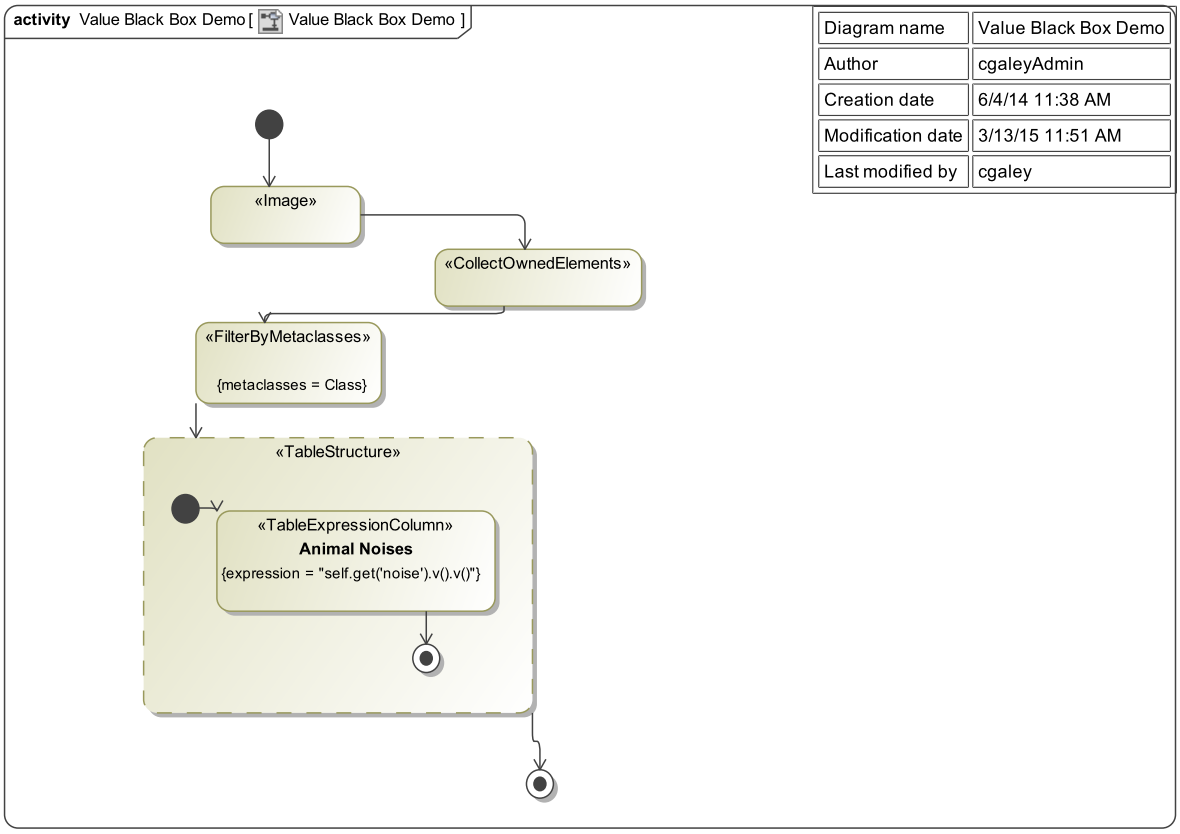


Table 3.14.

Animal Noises
meow
quack
moo
ribbit
bark

3.7.3. How Do I Use OCL Expressions in a Viewpoint?

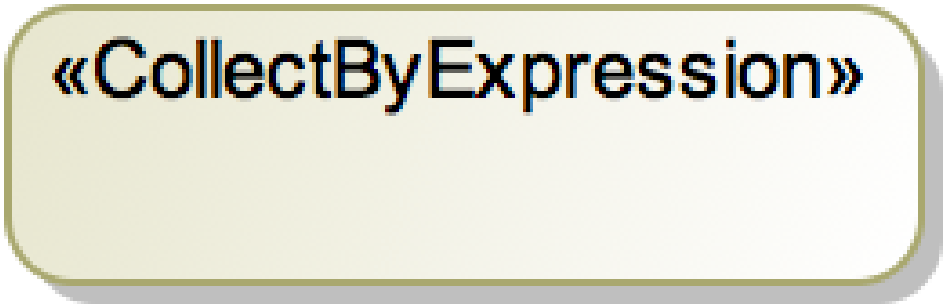
OCL expressions can be used in viewpoint methods in various ways to do various things. The following sections will outline some of these functions.

3.7.3.1. Using Collect/Filter/Sort by Expression

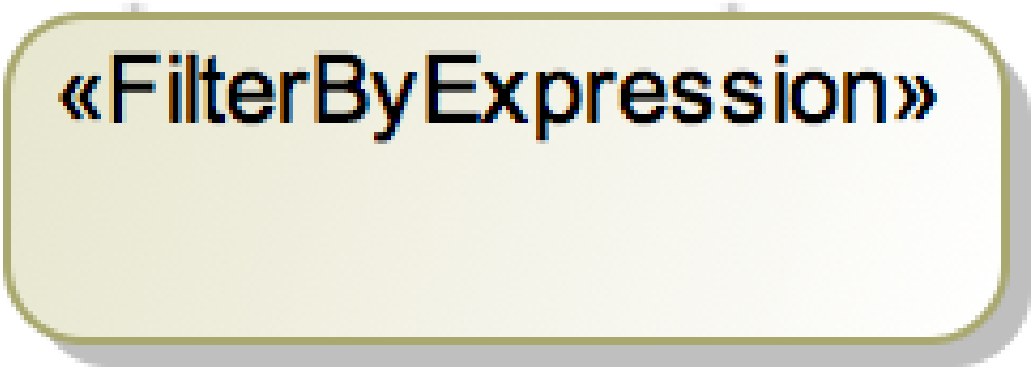
To more easily introduce OCL Expressions into viewpoint methods, several custom actions have been provided and can be found in the tool bar of viewpoint method diagrams along with the other collect and filter actions discussed in previous sections. A brief description of each follows below. Each action's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

Viewpoint Elements

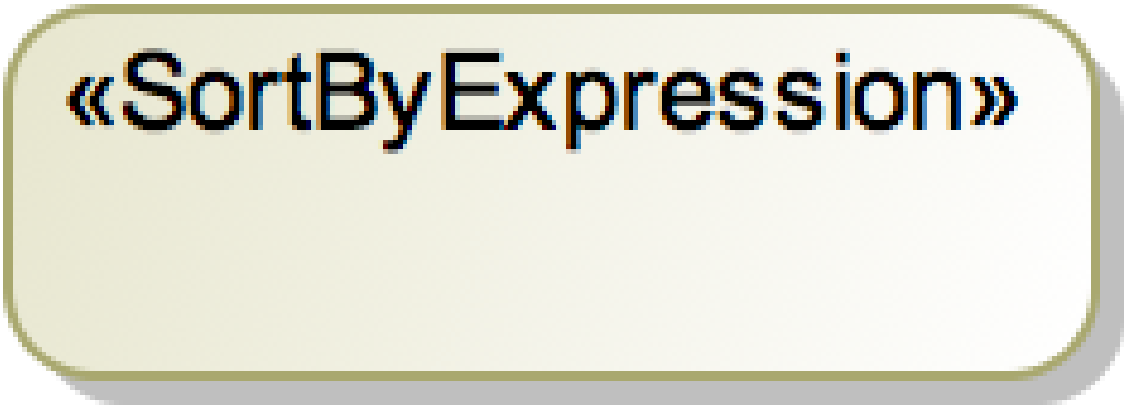
<< **CollectByExpression** >> Collect elements using an OCL expression. This works like other collect stereotypes.



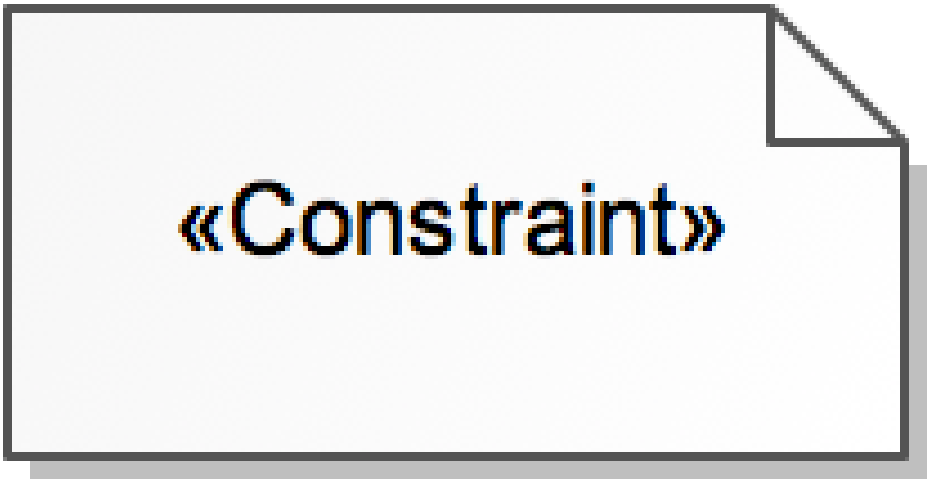
<< **FilterByExpression** >> Filter a collection using an OCL expression. This works like other filter stereotypes.



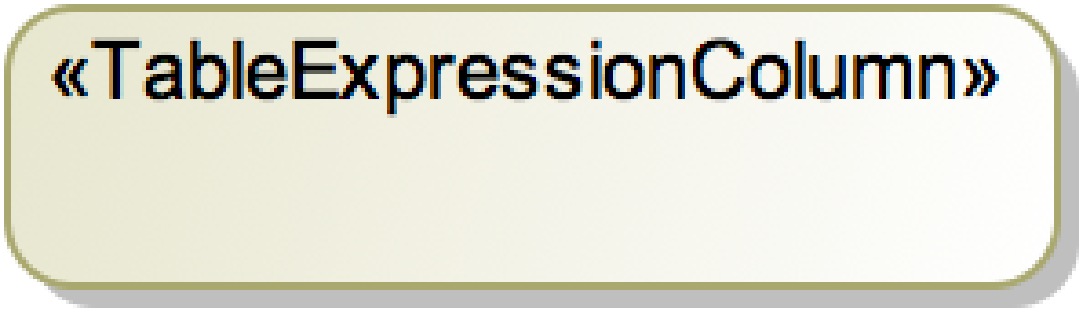
<< **SortByExpression** >> Sort a collection using an OCL expression. This works like other sort stereotypes.



<< **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.

3.7.3.2. Advanced Topics

View Currently Under Construction

3.7.3.2.1. Use of the Iterate Flag

Figure 3.43. Use of the Iterate Flag

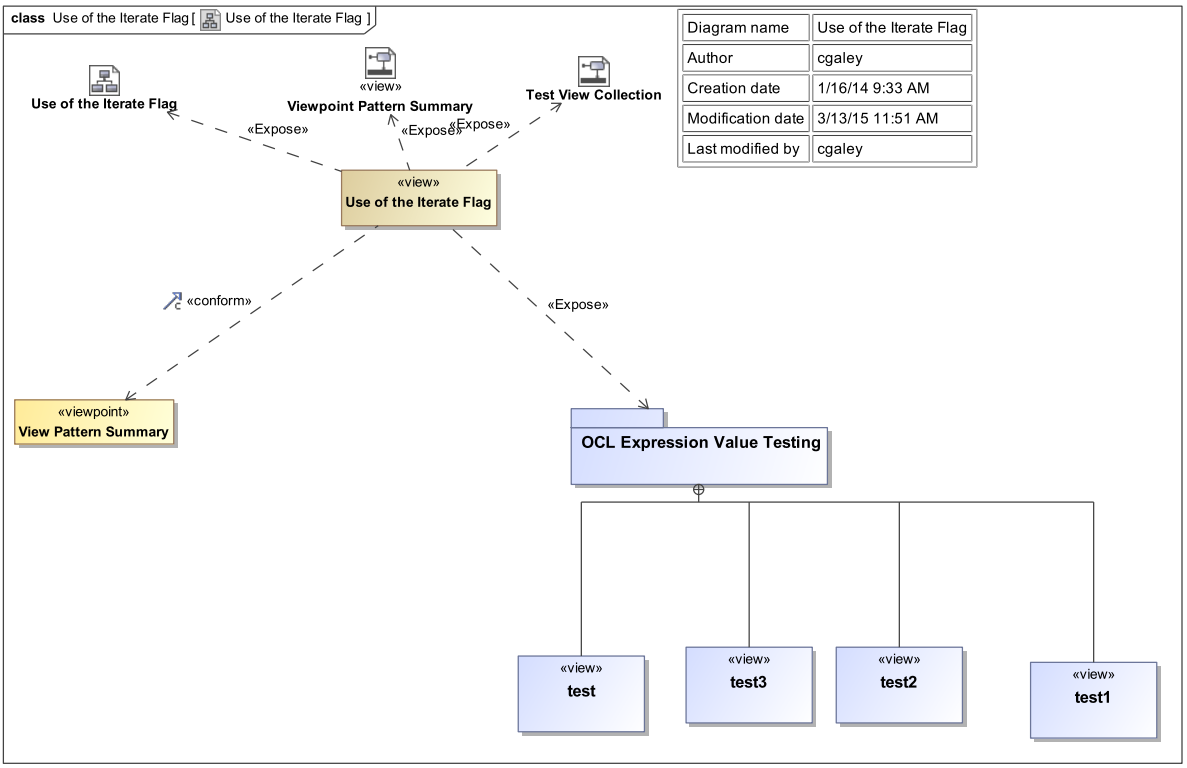


Figure 3.44. Viewpoint Pattern Summary

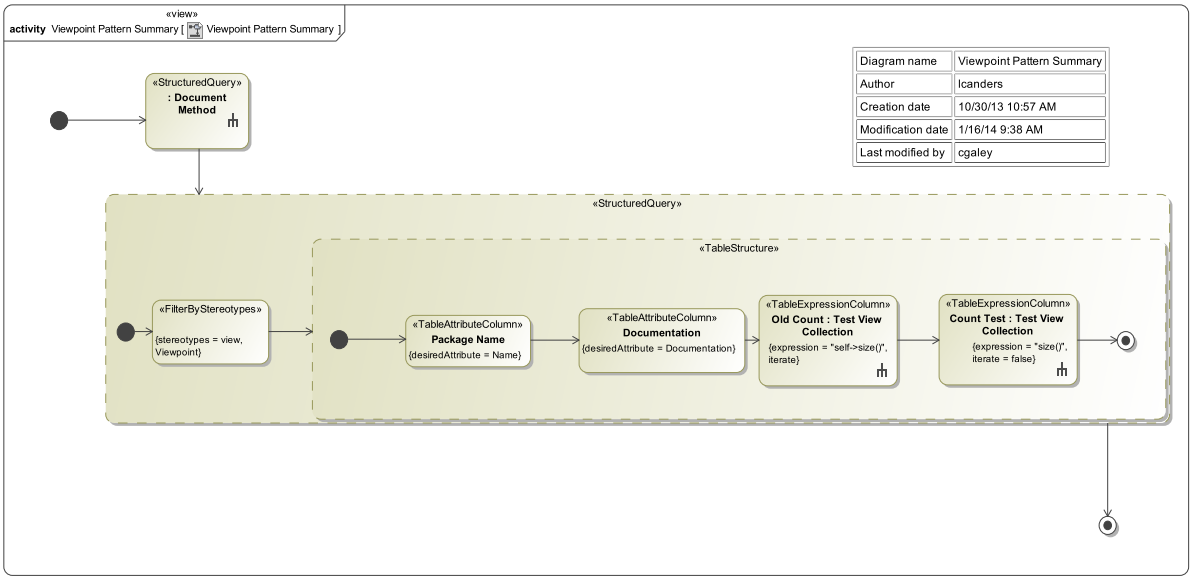


Figure 3.45. Test View Collection

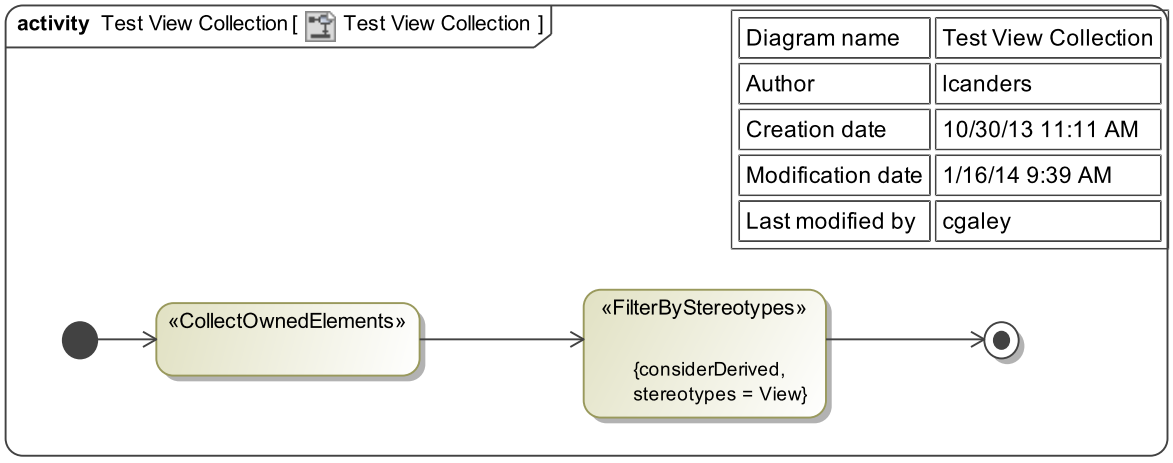
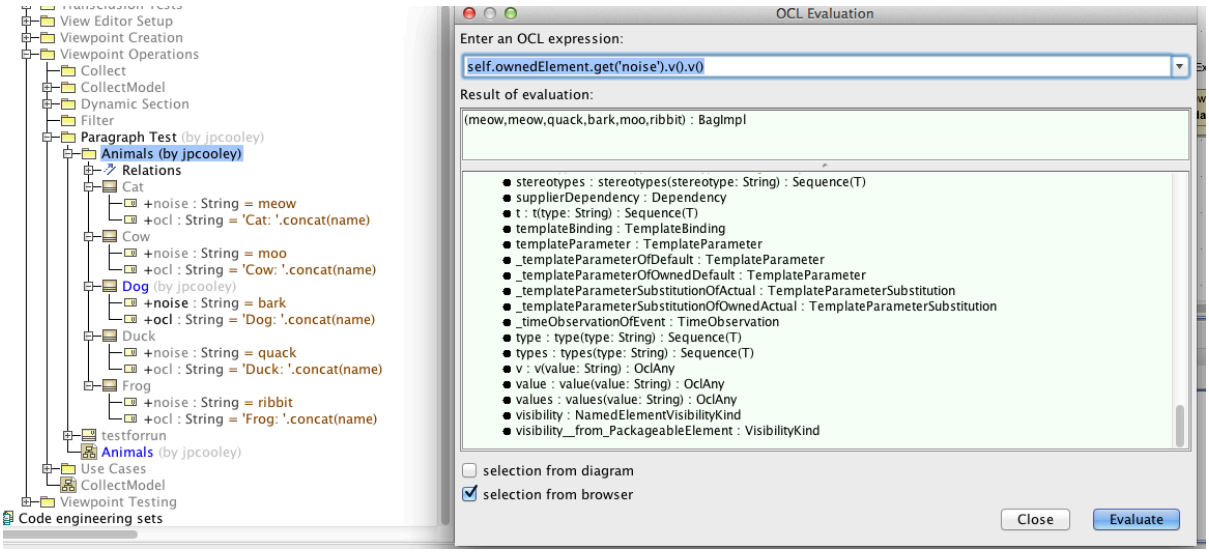


Table 3.15.

Package Name	Documentation	Old Count	Count Test
Viewpoint Pattern Summary			0

3.7.4. 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,

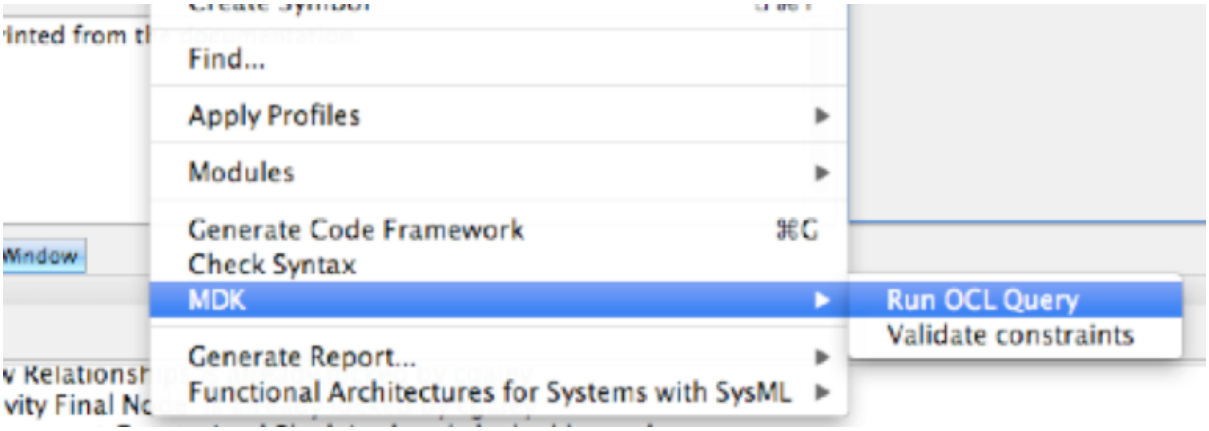
2. finding the MDK menu, and

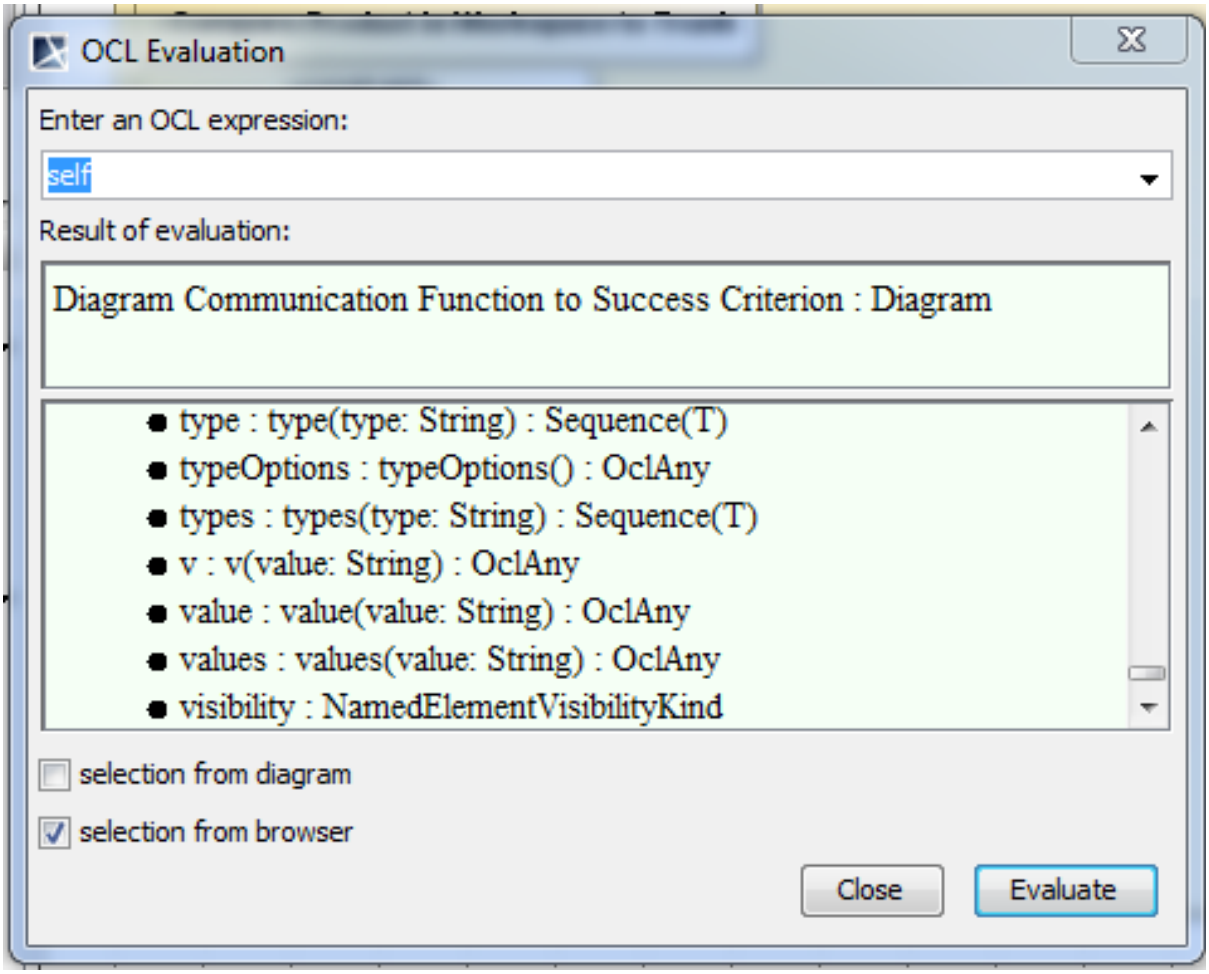
3. selecting "Run OCL Query."

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).

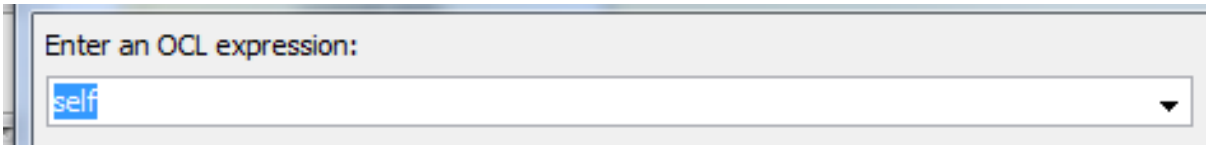
Note: "Self" in the below examples is setting the target of the OCL expression.





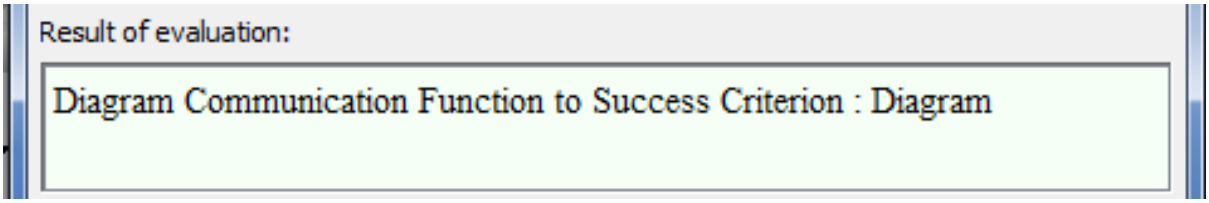
There are four parts to the OCL Evaluator:

1. Expression entry



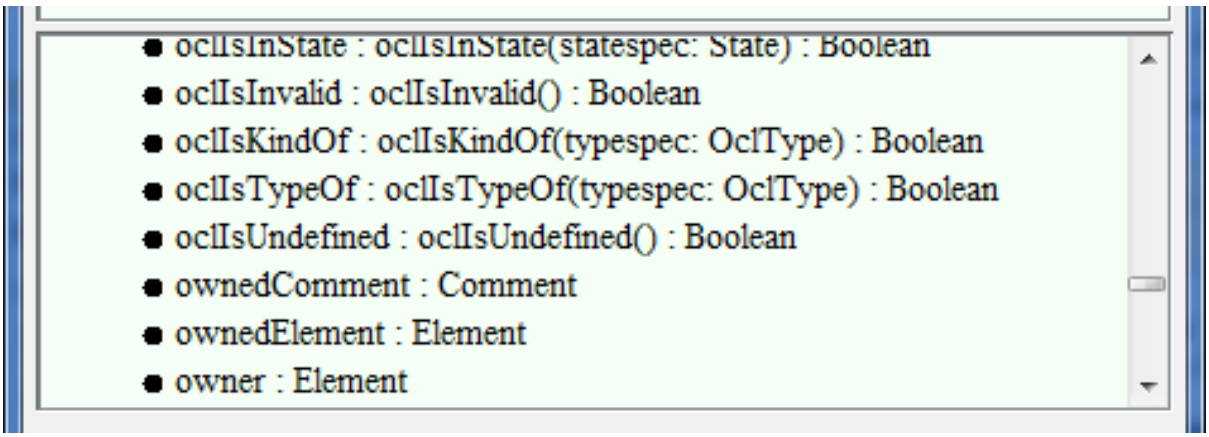
In this area you can enter an OCL query and recall previously evaluated queries by clicking on the arrow on the right hand side of the field.

2. Results



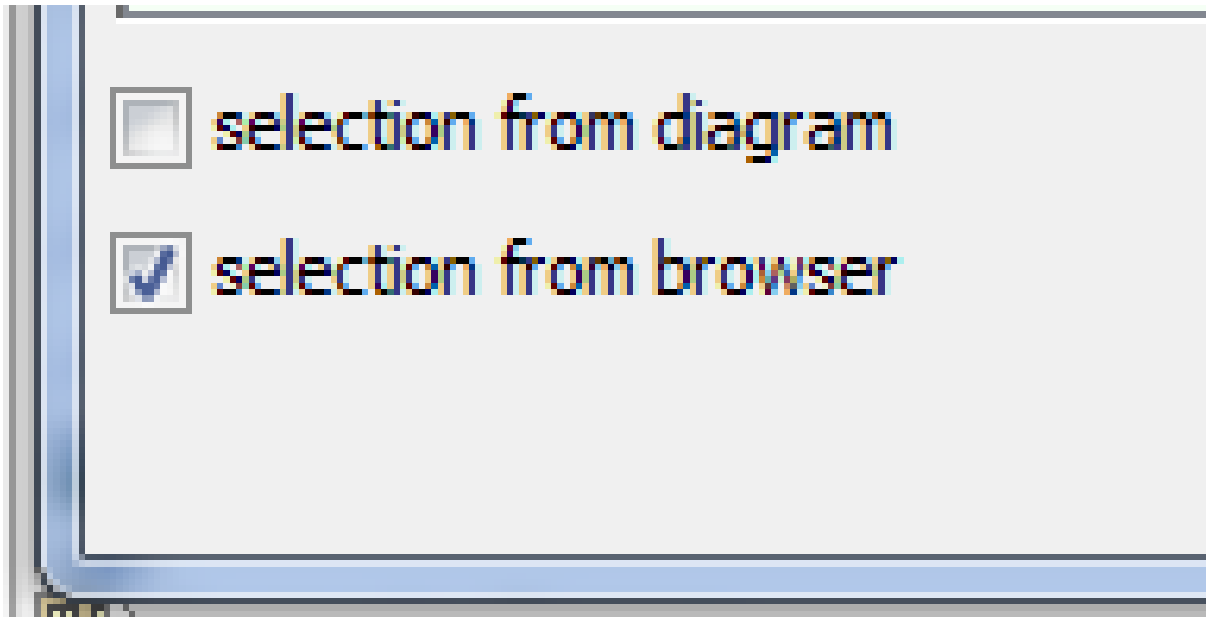
This is where the results of the query are displayed.

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.

3.7.5. 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.

Figure 3.46. How Do I Create Viewpoint Expressions?

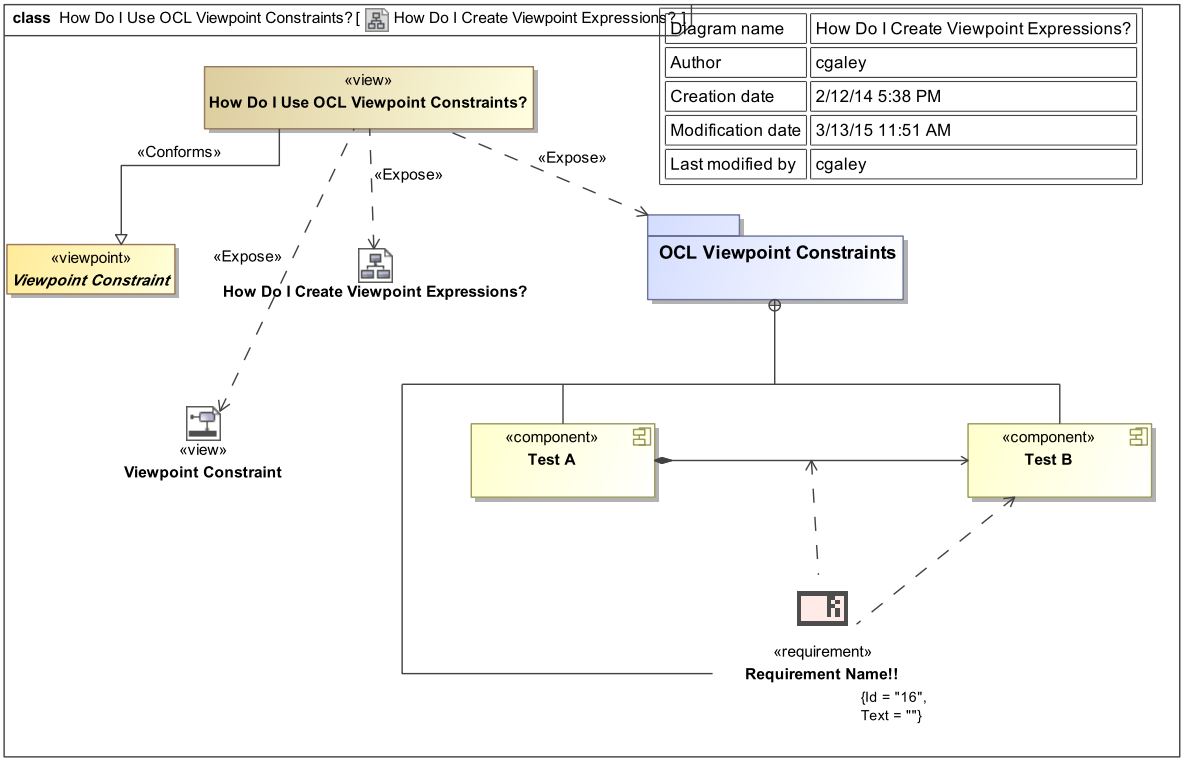
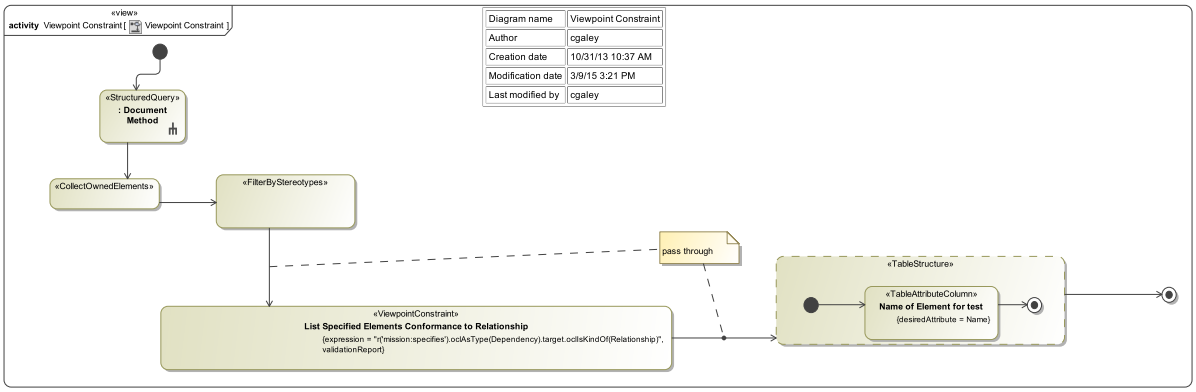


Figure 3.47. Viewpoint Constraint



3.7.6. How Do I Create OCL Rules?

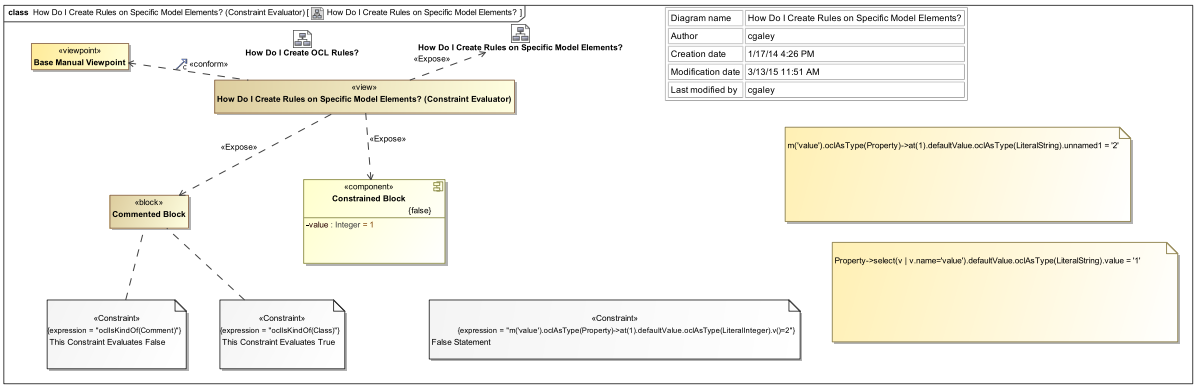
3.7.6.1. How Do I Create Rules on Specific Model Elements? (Constraint Evaluator)

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

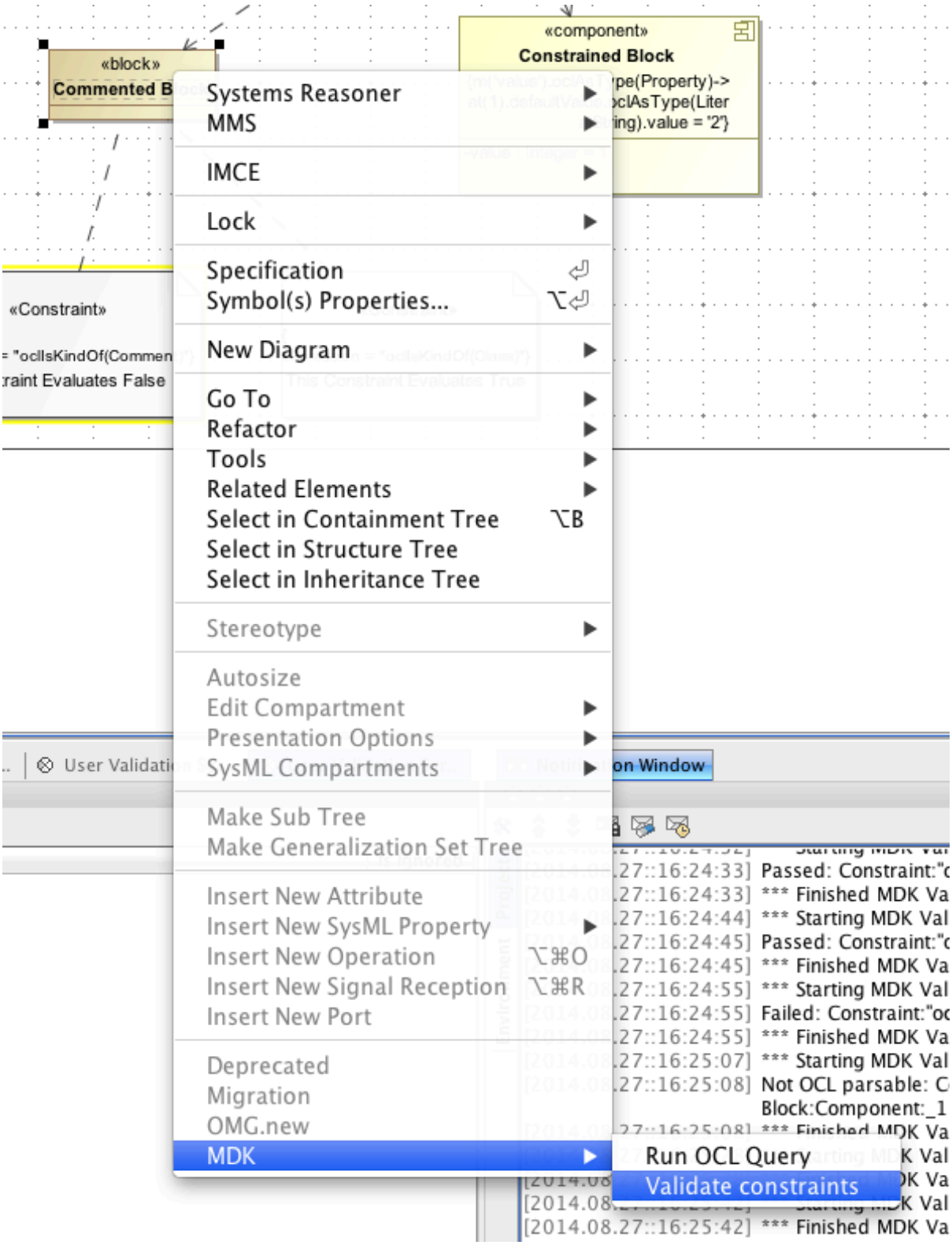
1. oclIsKindOf(Comment)

2. oclIsKindOf(Class)

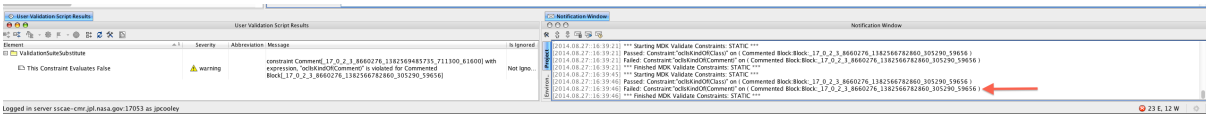
Figure 3.48. How Do I Create Rules on Specific Model Elements?



To validate these constraints directly in the model, select the element and right click, navigate to MDK, Validate constraints. See image below. Obviously, one of these constraints is true and one is false, so only one warning should be thrown. This is shown in the image below. The red arrow points out the failed constraint in the Magic Draw Notification window.



Obviously, one of these constraints is true and one is false, so only one warning should be thrown. This is shown in the image below. The red arrow points out the failed constraint in the Magic Draw Notification window.



3.7.6.2. How Do I Create Rules Within Viewpoints?

3.7.6.3. 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. Creation of validation rules is discussed here .

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

3.7.7. How Do I Create Expression Libraries?

3.7.8. 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

3.7.9. How Do I Create Transclusions with OCL Queries?

Figure 3.49. How Do I Create Transclusions with OCL Queries?

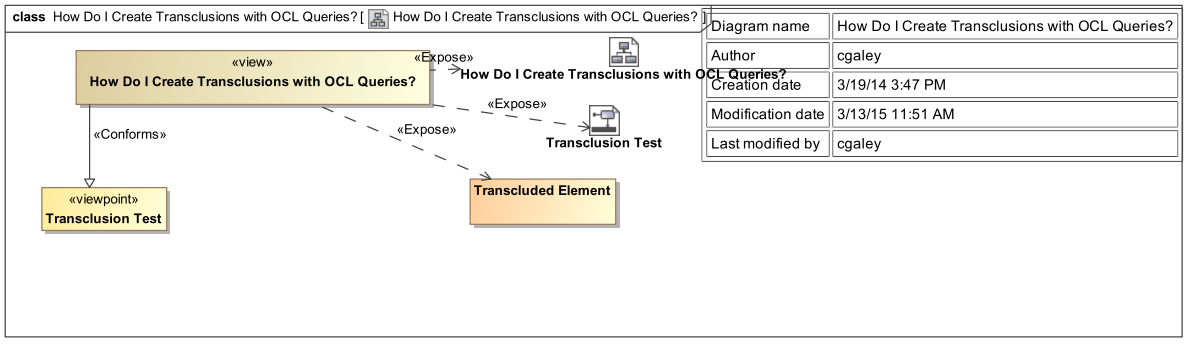


Figure 3.50. Transclusion Test

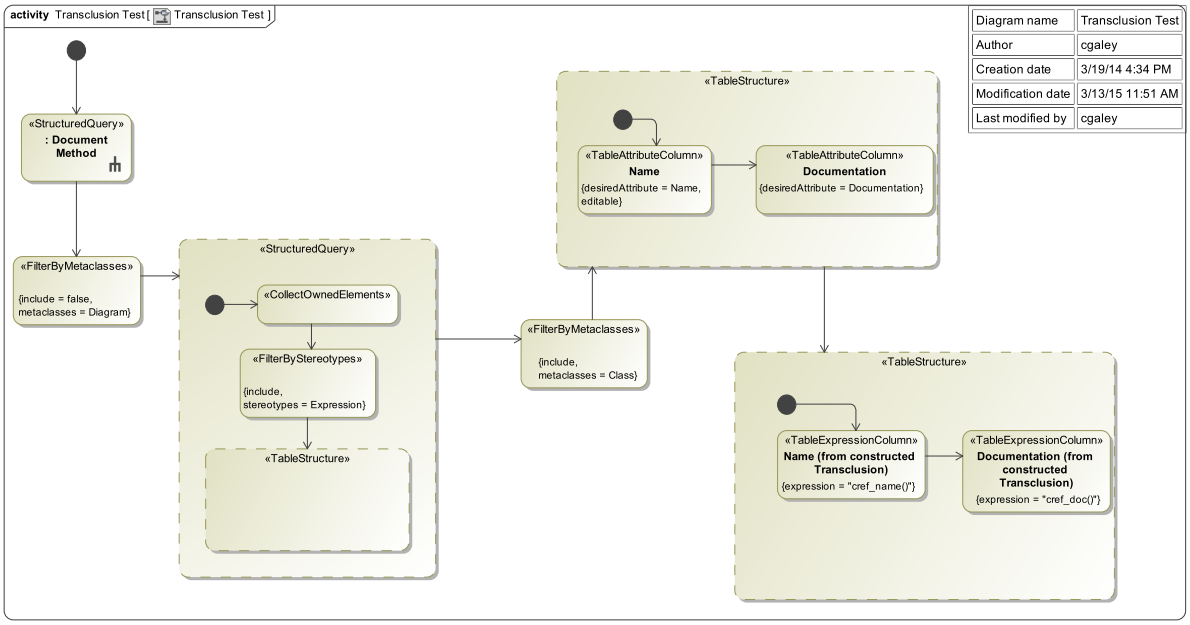


Table 3.16.

Name	Documentation
Transcluded Element	This is the documentation of the Transcluded Element

Table 3.17.

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

3.8. Create 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.

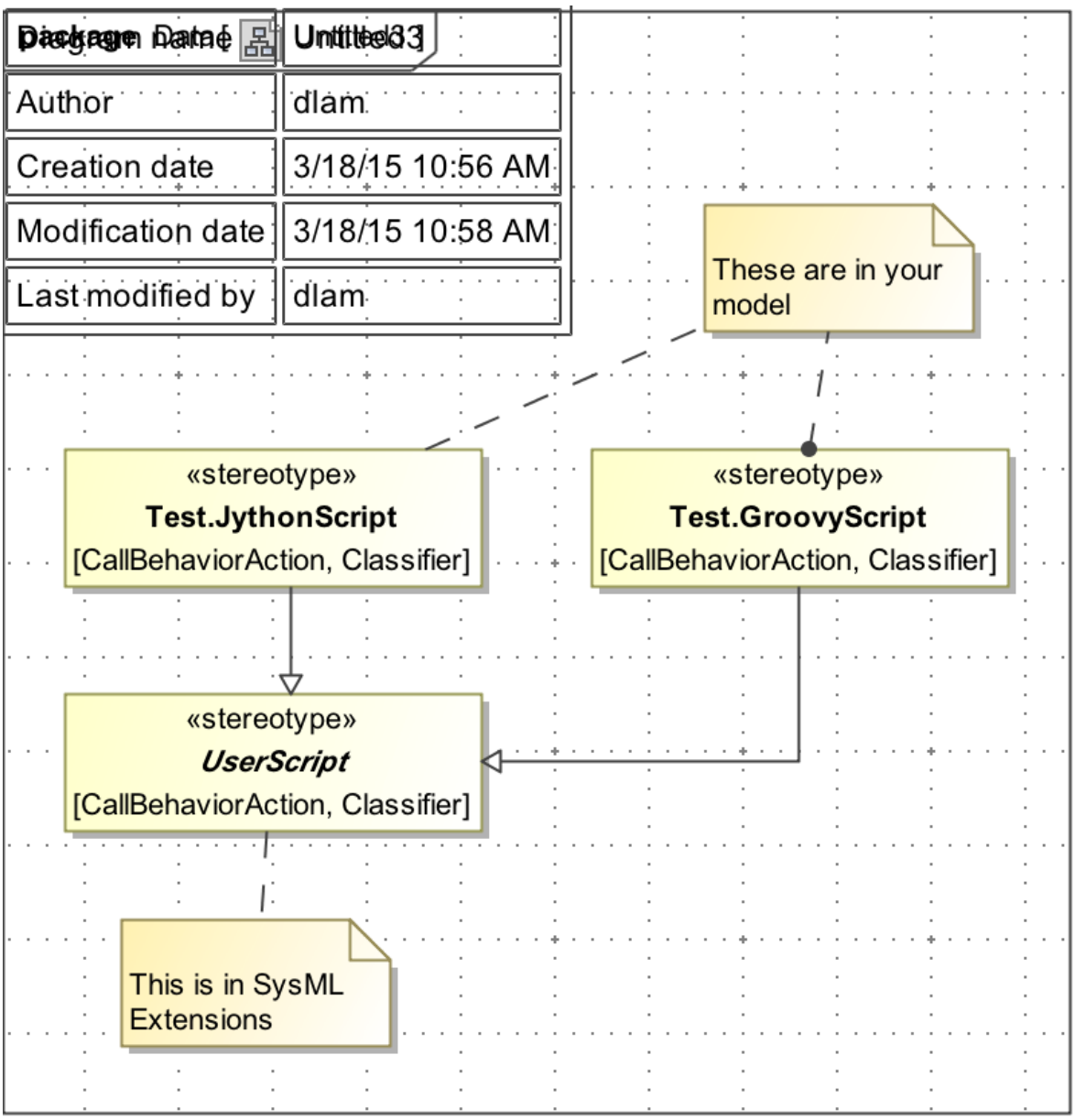
3.8.1. Presentation UserScripts

DocGen allows you to create your own custom queries using Jython, Groovy, or Java extensions. This section details how to create custom queries that result in elements that will be displayed, such as lists, tables, paragraphs.

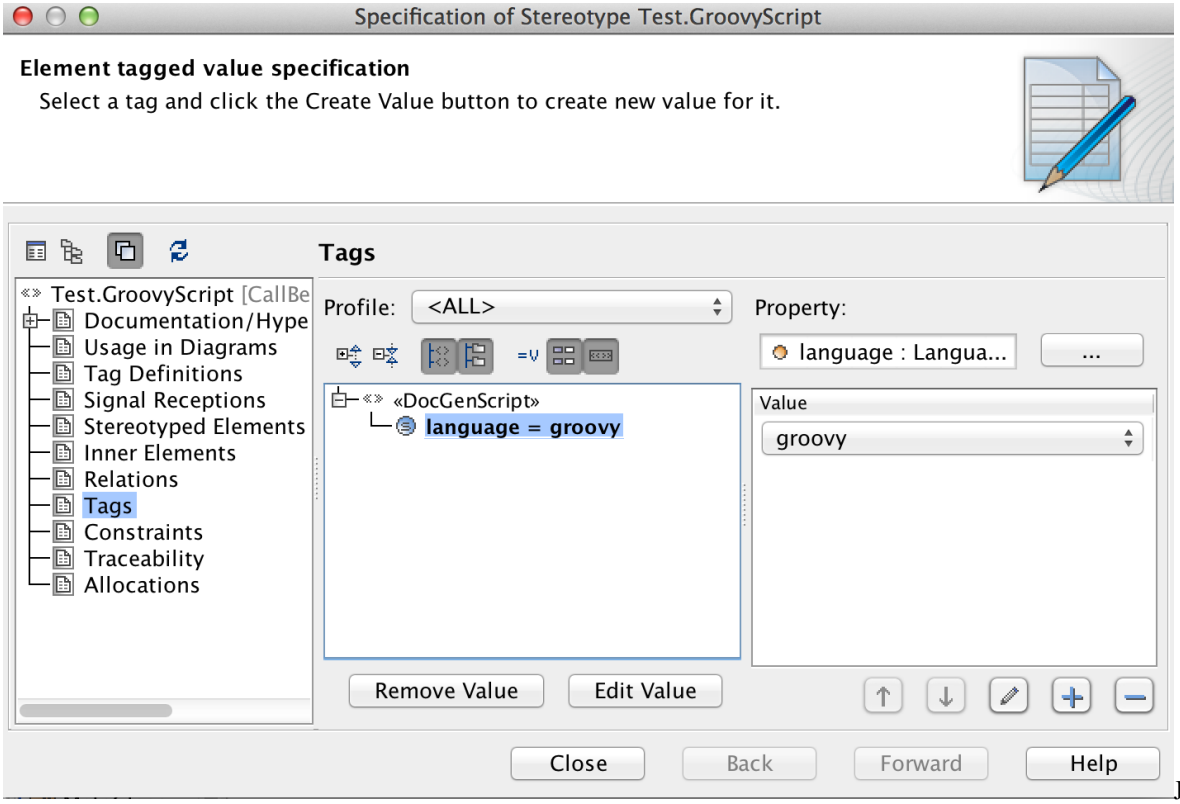
There are multiple steps to creating a Jython or Groovy UserScript:

1. Create a stereotype that specializes the UserScript stereotype in SysML Extensions profile. Name your stereotype like <namespace>.<script name>.

2.

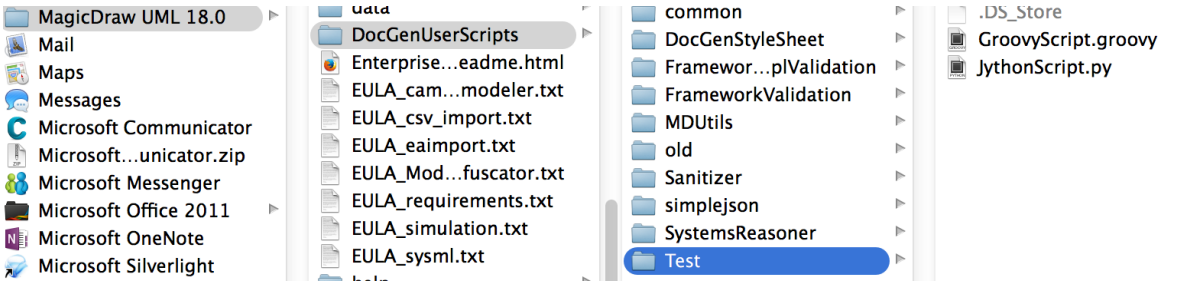


3. Jython is the default language if you don't specify anything. If you want to use Groovy, you need to apply the <<DocGenScript>> stereotype from SysML Extensions on your stereotype and set the language to be the right one under tags.

4. 

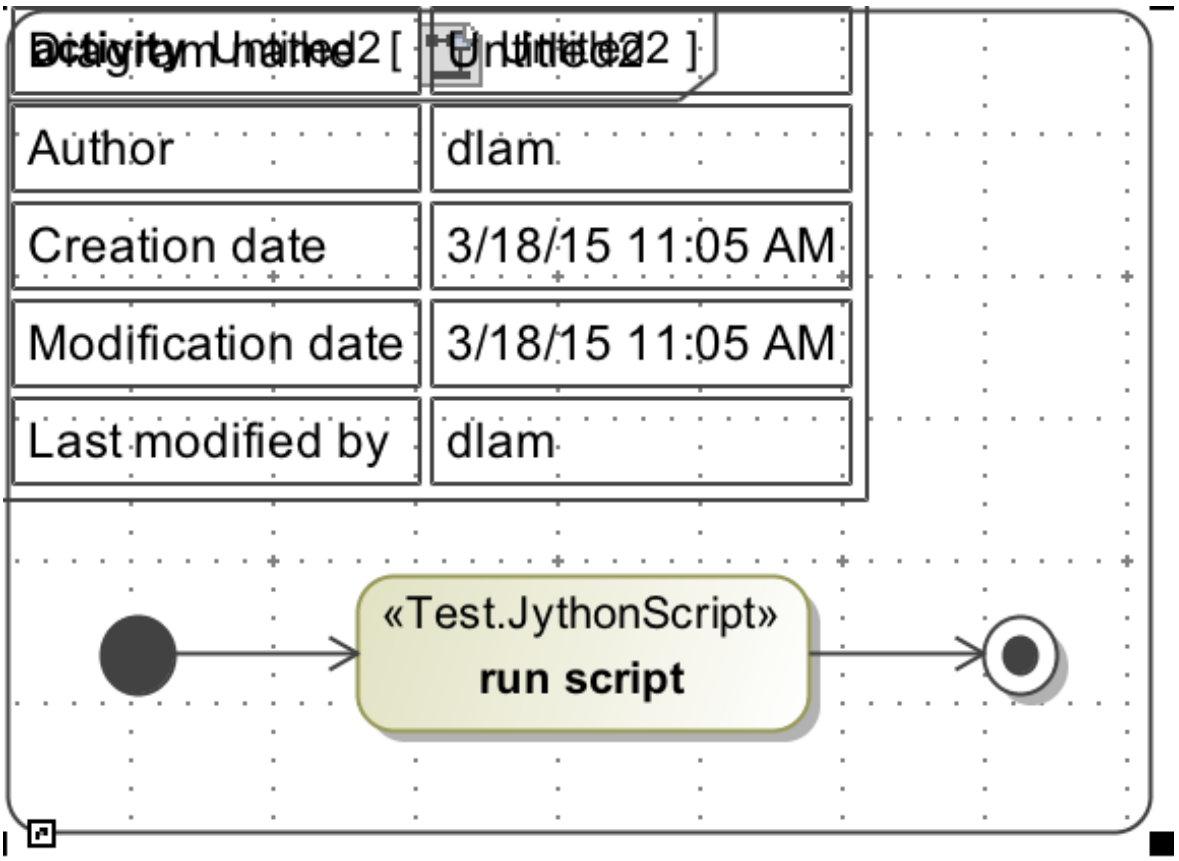
Jyth

5. Create your script under <md.install.dir>/DocGenUserScripts/<namespace>. The namespace and script name should be the same as what you named your stereotype. For Jython, use extension .py, for Groovy, use .groovy.

6. 

7. Use your new stereotype instead of the built in ones like <<BulletList>> or <<Paragraph>>. Instead of using the diagram palette of the viewpoint method diagram, you can create an action and apply the stereotype manually.

8.



Writing the script:

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]

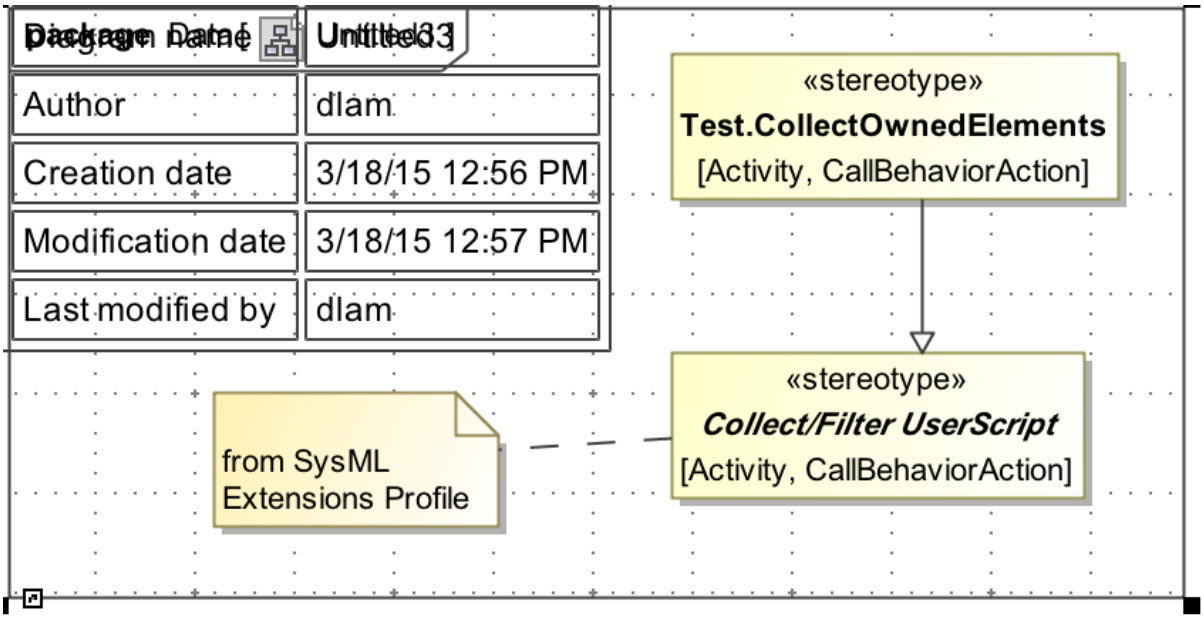
For output, DocGen would be looking for a "scriptOutput" object with a key-value pair of "DocGenOutput" to be a list of DB* objects. The DB* objects are detailed in the javadoc here. They are basically Java class representations of Table, Paragraph, or List. Below is example code that outputs a table:

```
from gov.nasa.jpl.mgss.mbee.docgen.docbook import DBTable
from gov.nasa.jpl.mgss.mbee.docgen.docbook import DBText
from gov.nasa.jpl.mgss.mbee.docgen.docbook import From
#This gives a one column table for editing names of the targets, a target each row
targets = scriptInput['DocGenTargets']
docgenHeaders = [[DBText("names")]] #this is for the DocGen UserScript table headers
docgenBody = [] #this is for the DocGen UserScript table body
for t in targets:
  docgenBody.append([DBText(t.getName(), t, From.NAME)]) #pass in the element and the property to edit
```

```
  #above adds the target's name, if you do DBText(t.getName()),
  #it'll still show the name but it won't be editable on the web
  #the first argument will be shown for local generations
docgenTable = DBTable()
docgenTable.setBody(docgenBody)
docgenTable.setHeaders(docgenHeaders)
docgenTable.setTitle("Example UserScript Table")
scriptOutput = {}
scriptOutput["DocGenOutput"] = [docgenTable] #this is for the docgen output, it can be list of DBText, DBParagraph, DBT
```

3.8.2. 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}
```

3.8.3. Validation UserScripts