CAMEO CONCEPT MODELER PLUGIN

18.0 SP10

Contents

1   Introduction

1.1   MDA

The Model Driven Architecture (MDA) approach as defined by the Object Management Group (OMG) "provides an approach for deriving value from models and architecture in support of the full life cycle of physical, organizational, and I.T. systems [1]."

1.2   Concept Modeling Purpose

When building a system for a business, there are a plethora of methodologies to choose from, as well as numerous existing models, profiles, and plug-ins across any given enterprise. What should be the starting point of the effort, business concepts, is often lost in overwhelming technical detail. Many profiles are at such an intricate technological level (e.g., DDL, XSD, AndroMDA) that a development team is faced with too many technical choices which leads to inconsistent models. Technological concerns drag down the level of abstraction to the point that business concerns can get overlooked. Aligning models becomes too difficult and too much work, almost invariably resulting in disconnected model silos.

A concept model (unifying business concepts across an enterprise) is the basis for a solution to this dilemma. A concept model represents the concepts and defining relationships of the *business*. A concept model is a model of the real world of the business, not the data used by business systems. Additionally, the concept model provides the vocabulary for process models that describe the way the business is run. The concept model is created by capturing the knowledge of business experts, then understood and validated by business experts.

Data models, which define and structure the data used by a system, can be built or generated by "sub-setting" a concept model. The concept model becomes the "Rosetta Stone" for enterprise level semantic integration (i.e., automatically generating data transformations between systems within the enterprise described by the concept model).

1.3   The Role of Ontologies and Reasoners

An ontology is a formal naming and definition of the types, properties, and interrelationship of the entities that exist in some domain. It defines and represents consensual knowledge as a set of concepts within a domain, using a shared vocabulary to denote the types, properties, and relationships of those concepts. Artificial intelligence, the Semantic Web, systems engineering, software engineering, biomedical informatics, library science, enterprise bookmarking, and information architecture all uses ontologies to represent concepts that belong to their domain in very specific ways. Domain ontologies (domain-specific ontologies) plays a significant role in the definition and use of an enterprise architecture framework.

Ontologies are commonly encoded using ontology languages. OWL (Web Ontology Language), produced by the W3C Web Ontology Working Group, is one of the formal languages to construct ontologies. It is an international standard for encoding and sharing ontologies and is

designed to support the Semantic Web. An OWL ontology may include classes, relations, attributes, formal axioms, and instances. OWL can be used to build most kinds of ontologies. The Concept Modeler maps to a subset of the OWL. The following are some examples of what you can do with OWL ontologies using the Concept Modeler:

- Import existing OWL ontologies for concept reuse, and/or as a starting point for the creation of a concept model.
- Export a concept model as an OWL ontology, which can be augmented by the addition of axioms not supported by the Concept Modeler. For example, axioms can be added to constrain model interpretations, or for advanced reasoning (e.g., transitivity) not supported by the Concept Modeler and UML.

A semantic reasoner infers logical consequences from a set of asserted axioms in an ontology, and typically provides automated support for reasoning tasks such as classification and querying. The inferences made by a semantic reasoner over an ontology generated by the Concept Modeler can be used to find logical inconsistencies in the primary concept model. Hence, a semantic reasoner can provide information to validate and improve a concept model.

The logical consistency of a concept model is particularly important if the desired result is a system that classifies information. As stated above, the inferences made by a semantic reasoner can help to ensure that the concept model is logically consistent in its classification.

The Concept Modeler maps to a subset of OWL that is most useful to the business purposes of defining a concept model. Consequently, any attempt to "round trip," (i.e., re-import a possibly modified ontology model that has been exported by the concept model) is very likely to be "lossy", particularly if the ontology generated from a concept model is augmented by additional axioms not supported by the Concept Modeler. Therefore, as a prime tenet of MDA, the concept model is considered to be the "primary" artifact, and the ontology is the "secondary" artifact. Business concept development and changes must be made in the concept model.

1.4   Open World Assumption vs. Closed World Assumption

Concept models built by the Concept Modeler satisfy the Open World Assumption. That is, no one is assumed to have complete knowledge, and a fact may be unknown. The opposite is the Closed World Assumption, where unknown facts are assumed to be false. An *information* model "subsetted" from a concept model would satisfy the Closed World Assumption.

1.5   Information Modeling Purpose

An information model describes what information is stored in a system, and is independent of any particular implementation of data management structure or technology. One way to remember the difference between a concept model and an information model is that while a personnel record can represent a person, a personnel record does not go to jail when a person is found guilty of wrongdoing. An information model has a different purpose than a concept model. An information model is designed to meet a set of requirements for the information in a system.

However, a concept model can provide an excellent starting point for an information model. The elements of a concept model that would fulfill a system's requirements can be cherry-picked or subsetted to create an information model in UML. Doing so can retain the traceability from elements in an information model to their definitions in a more precise language than plain English.

2   Concept Modeler Capabilities

This section provides the capabilities of the Concept Modeler.

2.1   SME Friendly Graphical Notation

- Uses consultant proven "SME (subject matter expert) friendly" graphical notation.
- Facilitates real-time interactions with SMEs to model real world business concepts and their relationships.
- Requires no camel case names.
- Sets, by default, visibility of properties to public.
- Encourages clean, hyperlinked micro-subject-area diagrams.

2.2   Automatic Styling of Concept Models

Large-scale models often contain information that is tangled in a complex web of relationships and therefore are difficult to read and focus on. Even though MagicDraw is the best modeling tool on the market, without the AutoStyler capability, untangling requires quite a bit of effort, so many modelers don't bother to try.

The Concept Modeler, through the AutoStyler plugin, assists the modeler in producing diagrams that are concise, focused, and presentable to stakeholders.

Central to AutoStyler is a style named "Defined Elsewhere" that is applied to diagrammed model elements when they are not defined on the current diagram. Traditionally, this style collapses all compartments and fades the normal element colors, including association ends that are not defined on the current diagram. The modeler can fine tune this style in any way the MagicDraw style system allows, and, if desired, even retain the default style for some or all kinds of elements.

AutoStyler examines two factors to determine whether the current diagram is eligible to be the defining diagram for an element being added to a diagram. The first is whether or not a non-descendant package owns the added element. In other words, when the added element is owned by an element that is not a child of the diagram's owner, it is defined elsewhere. An example of this is when a package other than the diagram's owning package owns the added class. The second is whether or not the added element has a hyperlink to a diagram, which indicates the defining diagram for the element. When an added element's hyperlink is not the current diagram, it is clearly defined elsewhere.

AutoStyler has a mode to automatically assign a hyperlink that points to the current diagram when it is eligible to be the defining diagram for an element being added to it. This automatic mode can be turned off as well.

The "Defined Elsewhere" style can be defined differently for each project, or defined the same across all projects. For example, one might use a UPDM architectural model and a UML software model with one standard "Defined Elsewhere" style that works for both kinds of models. This style is usually a clone of the "Default" style that has been adjusted to fade fill colors, text colors, and line colors, which can usually be done at the top level using opacity settings.

AutoStyler allows the modeler to select one or more elements on a diagram as the defining diagram for them, as long as they meet the criteria described above for defining diagram eligibility. AutoStyler cannot automatically change the style of an element on a diagram when its defining diagram status changes on any but the current diagram. AutoStyler allows the modeler to select these elements and "repair" the styles to reflect this change in status.

The "Default" and "Defined Elsewhere" styles are tuned for working with concept models. The symbol properties within those styles are, by default, set as follows:
- Visibility and stereotypes for properties are suppressed.
- Tagged values for association ends and classes are suppressed.
- Tagged values for attributes are shown.
- Subsets, redefines, and constraints for properties are shown.
- Constraint names for properties are shown (instead of constraint expressions).
- Properties of Property Holders are always shown.
- Association-end properties for Property Holders are shown as attributes when the "Defined Elsewhere" style is applied, and the Association Ends are not shown.

AutoStyler allows the modeler to change any of these settings element by element, or by changing them in the "Default" and "Defined Elsewhere" styles.

AutoStyler is currently a separate plugin for MagicDraw, but is expected to become part of the base MagicDraw product in the future.

2.3   Automatic Glossary Generation

Using a glossary saves time by ensuring consistent usage of terminology in the organization. It also improves the communication between team members since terms are understood in the same way and definitions become visible everywhere the terms are used.

Copyright © 2017, No Magic, Inc.

Depending on project options, automatic glossary generation in a concept model can create a glossary containing the names and descriptions of classes, association ends, attributes, enumerations, and / or enumeration literals used in the owning concept model. These glossaries are generated on import, element creation in the containment tree, or element creation on the diagram. When creating a new class, association end, attribute, enumeration, or enumeration literal in the containment tree or on the diagram, the element will not be added to the glossary until the user names the element.

For automatic glossary generation, a user is provided with five project options:

1. Add classes to a glossary. When a class is created in the Containment tree, created on a diagram, or imported from an ontology, the class name and documentation will be added to a glossary in the owning concept model.
2. Add association ends to a glossary. When an association end is created in the Containment tree, created on a diagram, or imported from an ontology, the class name and documentation will be added to a glossary in the owning concept model.
3. Add attributes to a glossary. When an attribute is created in the Containment tree, created on a diagram, or imported from an ontology, the attribute name and documentation will be added to a glossary in the owning concept model.
4. Add enumerations to a glossary. When an enumeration is created in the containment tree, created on a diagram, or imported from an ontology, the enumeration name and documentation will be added to a glossary in the owning concept model.
5. Add enumeration literals to a glossary. When an enumeration literal is created in the Containment tree, created on a diagram, or imported from an ontology, the enumeration literal and documentation will be added to a glossary in the owning concept model.

You may change these project options at a later time. You can build or rebuild a glossary table containing only the kinds of entries selected in the project options. When you create a glossary for any selected «Concept Model» stereotyped package, the Concept Modeler will add only the elements that exist inside the selected package to the glossary. For example, you have a project that has two packages: package A and package B. When you create a glossary for package A, the glossary only includes data from the selected package A. It does not contain any data from package B. See section 5.9 Create a Glossary Table for the detailed steps.

Just like creating a glossary, rebuilding it works the same way by allowing only the elements from a selected package to be kept. For example, you have created the glossary for package A and you later added some terms or elements (classes, association ends, attributes, enumerations, and enumeration literals) to the glossary manually. When you rebuild that glossary, the terms or elements that you have manually added will not be included in the glossary. The same thing will happen when you move some of the elements from package A to package B and then rebuild the glossary for package A. You will not find the elements that you removed from package A in the glossary. Please see section 5.10 Rebuild a Glossary Table for the detailed steps.

Attributes and association ends can be suppressed from a glossary when their names are too generic. For example, when a property is called "in", every occurrence of that word in any other description, therefore, undesirably becomes a hyperlink to the property called "in". Additionally, automatic glossary generation can be turned off. Existing glossary entries are not removed when automatic glossary generation is turned off.

The glossary table allows for managing the terms of a concept model in a spreadsheet-like form. Each row in the table represents a term, which can be a word, a phrase, or any element of the model.

With the help of this table, a user can easily:
- Create and manage all terms of the model in a single place.
- Customize the representation of the table.
- Export the data into an *.html, *.csv, or *.xlsx file.

For more information, please refer to the user manual for MagicDraw 18.0 SP4 or higher.

2.4   Concept Model Authoring

The Concept Modeler can:
- Create a concept model from scratch or by importing an OWL ontology for reuse and/or as a starting point for the creation of a concept model.
- Graphically represent imported RDFS/OWL 2 ontologies.
- Provide graphical concept model authoring with subject matter experts.
- Integrate with any UML model or UML-based standard, such as the Unified Profile for MoDAF and DoDAF, and NIEM-UML.
- Support the Open World Assumption (i.e., the absence of evidence is not evidence of absence).
- Export a concept model to an OWL 2 ontology for reasoning over and adding further precision to constrain possible interpretations.
- Support the creation of Closed World Assumption information models.
- Automatically style classes defined in other packages and diagrams.

To see all elements that the Concept Modeler can import or export, see sections 3. Concept Modeling Semantics and 4. UML to Equivalent OWL (in OWL Functional Syntax) respectively.

2.5   UML Model Traceability

The Concept Modeler uses UML to build models. Therefore, concept models built by the Concept Modeler can be traced to any UML model, (e.g., NIEM-UML).

2.6   Semantic Integration of Multiple Information Models

A concept model built by the Concept Modeler provides the semantics to integrate multiple information models "subsetted" from the concept model:

- Information at rest (e.g., relational and XML databases).
- Information in motion (e.g., XSD schema and NIEM-UML).

2.7   Natural Language Glossary

In addition to glossary tables, the Concept Modeler provides a separate feature for generating a natural-language glossary. Natural language glossaries are intended for technical and non-technical people alike. For instance, concept modelers can ensure that the model indeed says what was intended. Subject matter experts can ensure that the model captures their business knowledge correctly. And system builders can find definitions for the terms used in requirements in much more detail than usual.

A natural language glossary converts the elements in a concept model into natural-language sentences. Every class creates a hyperlinked glossary entry that describes its superclass(es), necessary and sufficient properties, necessary properties, and optional properties. Any user-supplied documentation is transcribed at the end of each glossary entry. That documentation can add supplemental definitions such as examples and counter-examples.

Users will find that the better the model, the clearer the auto-generated glossary.

2.8   Annotation Properties in the Natural Language Glossary

Cameo Concept Modeler offers a project option that allows the selection of which annotation properties will be shown or hidden in every natural language glossary entry, in addition to the definitions generated from the semantics of a concept model. You can select any number of annotation properties. Elements in the report such as Classes or Properties that are annotated with a «Annotation» stereotyped UML comment that contains one of these annotation properties will display the UML comment body in the report. When no comment body exists the name of the annotation property will display by itself.

In our software, the feature is labeled "Natural Language Glossary annotation property list" and it consists of a list of pre-loaded annotation properties.

2.9   Preferred Annotation Property

Cameo Concept Modeler offers a project option that speeds up the creation of one of many possible kinds of annotations, and specifies which kind of annotation to treat as documentation. You can select one "preferred" annotation property to be assumed whenever the user adds a UML Comment or an annotation. Comments and Annotations that explicitly use that annotation

property will then be shown in the documentation panel, and in the Natural Language Glossary, as the human-specified definition (as opposed to the model-generated definition).

If you import a concept model that contains annotations, they are placed in its owning folder and each annotation has an annotated element and can have an annotation property tagged value. When you select a preferred annotation property, the «Annotation»s owned by the annotated element for the preferred annotation property will appear as documentation in MagicDraw.

Any annotations on ontology itself are imported correctly by CCM as annotations.
- In this new update, any annotations on ontology itself are imported by the Concept Model as annotations.

If your project is a TWC project, Concept Modeler will attempt to lock the project's elements. If any of the elements cannot be locked, whether it is locked by another user, then several message windows will appear, notifiying you of the problem and allowing you to see which elements are not locked. Please refer to 5.20.7 for more information about these messages and further steps.

2.10 Creation of Multiple Data Models from One Concept Model

2.11 Connection of Multiple Existing Data Models to One Concept Model

2.12 Updating Symbol Styles

Cameo Concept Modeler diagrams are intended to be as non-technical as possible for subject matter experts. As new features are added to the Concept Modeler, sometimes symbol styles may expose technical details that are not appropriate for that kind of audience. Cameo Concept Modeler therefore offers to tweak the styles called "Default" and "Defined Elsewhere". However, if you have tweaked those styles yourself, you may wish to either defer this, or make Concept Modeler stop asking you altogether.

This new feature that updates the symbol styles in older projects; more specifically, we added versioning to symbol styles which allows you to programmatically update a project's symbol styles. Please note that this feature only works in 18.2+ and is not compatible in 18.0 and 18.1. Additionally, updating symbol styles will overwrite the existing styles, so if you manually made changes to the styles or added new features to the existing styles, those values will not be retained. Please refer to 5.21.1 for instructions related to updating symbol styles.

In addition to the Concept Modeler's import capability, the software allows for diagram preservation after an ontology import. More specifically, while importing an ontology, the concept modeler will update the existing concept model. An ontology is imported into a CCM project that contains one or more concept models. Each ontology is imported into a concept model that may already be present in the project in which the ontology is imported. Ontology elements get translated into concept model elements.

The following table describes the conditions, evaluated by Concept Modeler, of each resource from ontology that is being imported. The condition determined by Concept Modeler will dictate how Concept Modeler will create/merge/delete the model element.	
New	An element is not present in the model project in which an ontology is being imported. Some parts of the ontology being imported may be already present in the model project but some of the other parts may be brand new.
Deleted	An element may be present in a concept model in the model project in which an ontology is being imported. This element however may be missing from the ontology being imported. This deletion need to be identified and element removed after the import.
Modified	An element may be present in a concept model in the model project in which an ontology is being imported. This element however may have different properties / values in the ontology being imported. This update to its properties need to be identified and updated after the import.
Same	An element remains unmodified after the import of an ontology.

The table below groups all the concept model elements and explains how they handle the "preservation" for each of the four conditions explained above.

Concept Model Element	Type of Update after Import	What can be modified?
Concept Model	New and Modified	Only two relevant conditions
	Deleted and Same	Irrelevant for the concept model.
Concept	New	New Concept is present only in the imported ontology for the given model.
	Modified	The concept's IRI matches but either or both of the name and owner are different.
	Deleted	Concept is present in the model but missing from the ontology. Mark all the new, same and modified ones in the original model. Delete the unmarked ones from the original model as these are the ones that are deleted in the ontology.
	Same	Following concept property

		match - Concept's IRI, name and its owner / model.
	New	New generalization (to be identified by general and special concept) is present only in the ontology
	Modified	Not applicable
	Same	Generalization's general and special concept are same in the ontology as in the model.
Concept Generalization	Deleted	Concept Modeler will mark all the new and same generalizations from the original model and delete all the unmarked ones since those are the generalizations deleted in the ontology. We need to mark all the new and same generalizations (in the original model) and delete all the unmarked ones (from the original model) as these are the generalizations that are deleted in the ontology.
Concept Disjoint Relationship	Explicit disjoint relation between 2 concepts	Identical to concept generalization
	New	New equivalence is present only in the ontology
	Modified	Not applicable
	Same	Equivalent concepts are the same in the ontology as in the model.
Concept Equivalence Relationship	Deleted	Concept Modeler will mark all the new and same generalizations from the original model and delete all the unmarked ones since those are the generalizations deleted in the ontology. We need to mark all the new and same generalizations (in the original model) and delete all the unmarked ones (from the original model) as these are the generalizations that are deleted in the ontology.

Anonymous Unions	Same	Same (anonymous) union to be identified by looking up constituents (set of concepts in the union) of the union against every existing union.
	Modified	Modified union is a union which has either a) same union constituents AND different (uniquely identifiable) properties or b) different constituents AND same (uniquely identifiable) properties
	New	New set constituents AND new properties
	Deleted	Other 3 cases should mark anonymous unions that are same, new or modified. Any unmarked anonymous unions in the original model are to be deleted.
Properties	New	New object property is present only in the imported ontology (to be determined using IRI)
	Same	Following properties of a object property should match- IRI, Domain, type, multiplicities
	Modified	Object property's IRI matches but one or more of the following values differ - Domain, type, multiplicities
	Deleted	Property is present only in the original model but is missing from the ontology being imported. Mark all the new, same and modified ones in the original model. Delete the unmarked ones from the original model as these are the ones that are deleted in the ontology.

3 Concept Modeling Semantics

In order to improve UML's suitability for modeling real-world concepts, the Concept Modeler interprets the UML standard to allow subproperties, existential quantification constraints, and universal quantification constraints. In addition to those interpretations, the Concept Modeler uses a small UML profile to add the capabilities of global properties, necessary and sufficient properties, and other future capabilities. Simply having or applying a «Concept Model» stereotype on a UML package causes anything within that package to have this interpretation, and allows these added capabilities.

The following subsections describe how the Concept Modeler interprets the UML standard and augments it to describe conceptualizations.

3.1 Class

In the concept modeling interpretation of the UML standard, a class is a set or collection of individual things called *members*. The members of a class in a concept model are either things that exist in the real world around us, or things we can imagine to exist, such as unicorns. For example, depending on the stated scope of a concept model, the members of a Chair class would include the one you sit upon to do your work, or the one in a warehouse ready to be shipped to a customer.

3.2 Property Ownership

The concept modeling profile of UML interprets the owner of a property as a context in which that property must conform to certain constraints. These constraints can include multiplicity (which includes a minimum cardinality and a maximum cardinality), a type for the property, existential quantification, and universal quantification, which is the default. When an instance is a member of an owning class, all of that class' constraints must be met.

UML allows the cardinality of a property to be left unspecified. Unlike UML, which interprets unspecified cardinalities as a minimum of one and a maximum of one, the concept modeling profile interprets unspecified cardinalities as being zero to many ("0..*").

An OWL ontology may contain properties in namespaces that are different from their domains. If you import an OWL ontology that has properties with foreign domains defined in such manner, you will see the association ends with cross (x) marks. In the Concept Modeler, these non-navigable association ends mean that the properties belong to foreign domains and therefore, they are owned by the association. The following diagram shows the examples of non-navigable association ends.
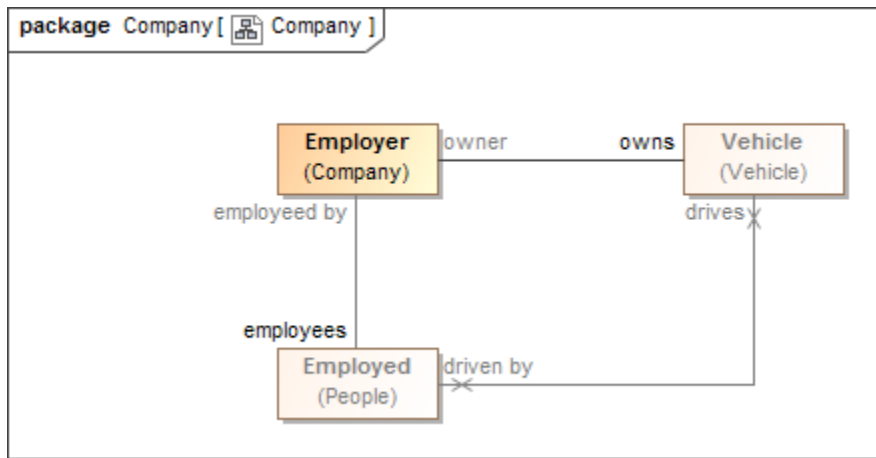
Figure 1   Properties owned by an association

In the diagram, the Employer, Vehicle, and Employed classes belong to three different namespaces, and the association belongs to the same namespace as that of the Employer.

Note	It is recommended that duplicate property names in a concept modeling diagram be avoided because they will result in conflicting definition of domains and ranges when exported to OWL.

3.3   Global Properties

Global properties are property declarations that can be used by any instance. Normally, a UML property cannot be defined outside of a classifier, so a global property declaration is represented as a UML property owned by a class that is stereotyped as a «Anything». The concept of a property holder was introduced in the NIEM-UML standard for a similar purpose. In the concept modeling profile, every property holder is equivalent to one topmost class (⊤) of which all other classes are subclasses. Thus, a property of a property holder is "inherited by" all subclasses and usable in any instances. In addition, while the name of a property holder is irrelevant, consistently naming property holders "Thing", "Concept", or "Entity" in all concept models avoids any confusion with normal classes.
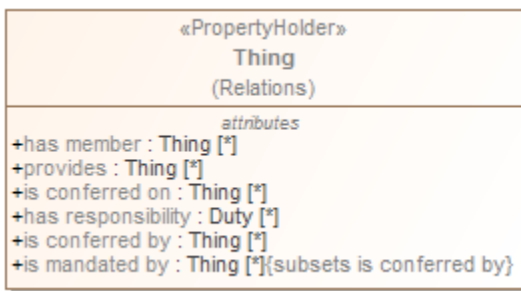

Figure 2   A property holder in Concept Modeler

3.4 Subproperty

A subproperty is a more specific kind of property than some other property, and a super property is a more general kind of property than some other property. For example, "has father" is a more specific property than "has parent", and "has parent" is a more general property than either "has mother" or "has father". In the concept modeling interpretation of UML, subsetting a property creates a subproperty when the subsetting property has a different name than the subsetted property. (See section 3.5 Existential Quantification Constraint, for when the name is the same or is omitted.) UML provides a {subsets} constraint that asserts that the values within a subsetting property are also in the set of values within a subsetted property. To stay as close to standard UML as possible, the concept modeling profile interprets a subsetting property having a different name as a subproperty.

The diagram below shows that the property "is capacity of" (owned by the class "Legal Capacity") is a subset of the global property "is conferred on" (from the property holder "Thing").

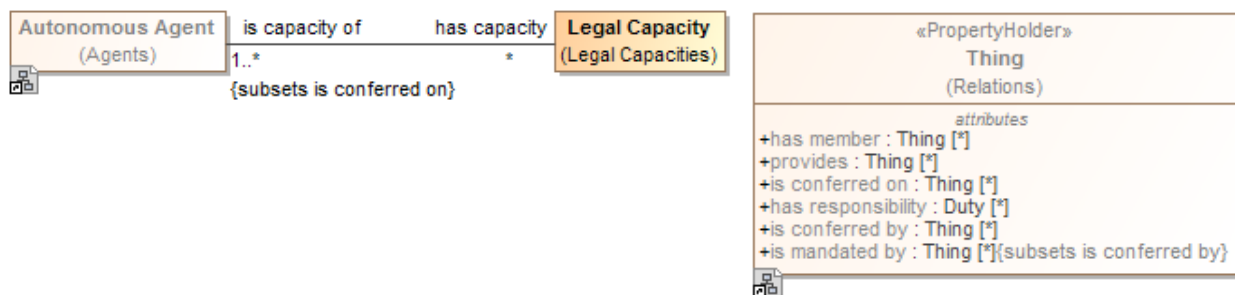| Note	In order to create a subproperty, the subsetting property must have a different name than the property it subsets.

Figure 3   Property 'is conferred on' and subproperty 'is capacity of' having different names

3.5 Existential Quantification Constraint

A property is not limited to a minimum and a maximum cardinality (known as multiplicity) for just one type. A property can have a multiplicity for a superclass, while at the same time having a more specific multiplicity for one or more subclasses of that superclass. This constraint is known as an *existential quantification* (∃) or qualified constraint. This type of constraint is an assertion that, among other possible values, the number of values of one of these subclasses is between some minimum and maximum cardinality. Adding an existential quantification constraint does not define a new property, rather it constrains an existing property. Note that an existential quantification constraint must have a minimum cardinality of at least one in order to meet the definition of "existential" for the constraint. In the concept modeling interpretation of UML, subsetting a property without giving the new property a different name (or leaving off the new property name altogether) creates an existential quantification constraint. As {subsets} with an

Copyright © 2017, No Magic, Inc.

omitted name is not well defined in UML, in the concept modeling profile it is used to state that a subset of values must meet the stated cardinality and type constraints of the subsetting property. It does not create a new property, although it does create a context in which this constraint holds: the owning class and its subclasses.

The next diagram shows an existential quantification constraint on the global property "is conferred by" (from the property holder "Thing"). The multiplicity is such that at least one of the instances of the property constraint must be one of the types in the union.

Note	The property adding the constraint is unnamed. An unnamed property is equivalent, in this case, to naming this property the same as the property being constrained ("is conferred by" from the property holder "Thing").
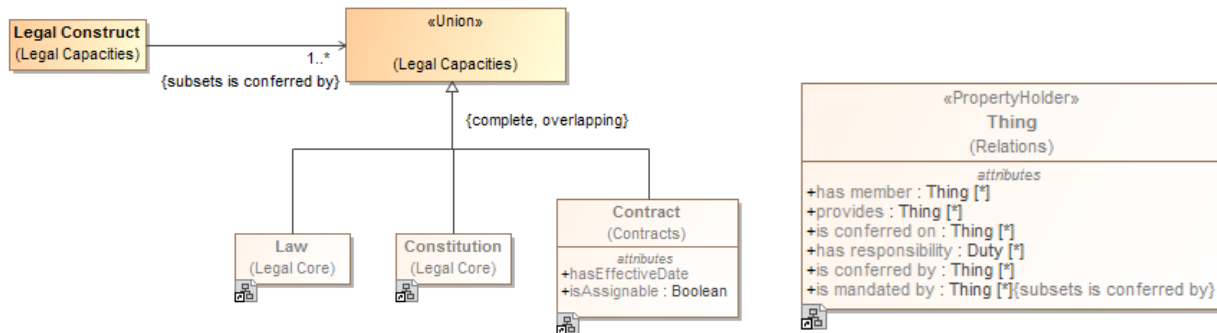
Figure 4 An existential qualification constraint on property 'is conferred by'

Note	• In the Concept Modeler, the existential quantification constraint of a property must have a minimum multiplicity of at least one. If the minimum multiplicity of a property that restricts another property is, for example, 0..5 or 0..*, the Concept Modeler will adjust it to 1..5 or 1..*. • Multiplicity values of *, 0..*, and Unspecified all mean the same thing.

3.6   Universal Quantification Constraint

Sometimes, in the context of some class, it is necessary to constrain *all* the values of a property to a particular type. This constraint is known as a *universal* quantification or *for-all* constraint (∀). This kind of constraint is an assertion that only values of the specified type are valid, and that the number of values must be between some minimum and maximum cardinality. In the concept modeling interpretation of UML, introducing a new property or redefining an existing property creates a universal quantification constraint in the context of the owning class. This interpretation is based on {redefines} in UML, which allows adding more specific constraints to an existing property without defining a new property.

Copyright © 2017, No Magic, Inc.

The diagram below shows the introduction of a new property, "consists of", defining a universal quantification constraint on the property. The constraint states, in the context of Soccer Team and any of its subclasses, that all values of this property must be of the type "Soccer Player", and that there must be between 5 and 11 values of this property.
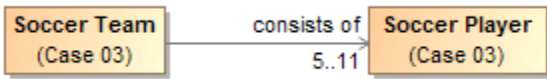

Figure 5   The property 'consist of' defining a universal quantification constraint

The following diagram shows a universal quantification constraint on the property "has" (owned by the class "Person"). It states, in the context of "Dog Owner", that all values of the property "has" must be of type "Dog", and that at least one value of this property must exist.

Note	A property that is redefined must have the same name as the redefined property. In this case, leaving the redefined property unnamed is equivalent to naming the property the same as the one being redefined ("has" from the class "Person").
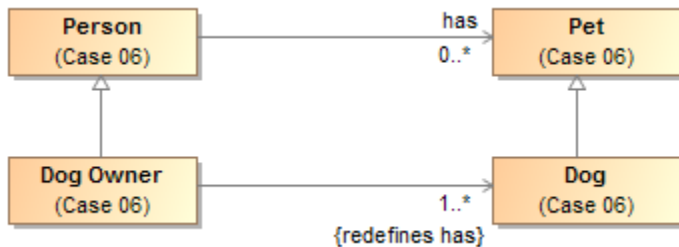
Figure 6   A universal quantification constraint on the property 'has'.

3.7   Necessary and Sufficient Condition

A property's multiplicity or type is declared in the context of an owning class or a property holder. When the minimum cardinality is at least one, these declarations are always *necessary* conditions for an instance to be a member of the owning class, or, in the case of a property holder, for an instance to be valid at all.

Another kind of condition is known as both *necessary and sufficient*. A class with at least one necessary and sufficient condition is known as a *defined* class, which means the differentiating characteristics of the class that make it distinguishable from its parent and sibling classes are defined. Note that using a necessary and sufficient condition on a property with a minimum cardinality of zero is not meaningful.

In the concept modeling interpretation of UML, a property that has the {sufficient} constraint applied to it indicates that when an instance satisfies the multiplicity and type constraints for the property's values, not only is a *necessary* condition for being an instance of the class met, a *sufficient* condition is also met. This necessary and sufficient condition allows an inferencing engine to classify that instance as a member of the class with that condition. Once an instance is classified automatically, the conditions on any other properties that have the {sufficient} constraint, including those inherited from superclasses, merely become *necessary* conditions the instance must meet to be a *valid* member of the owning class. In other words, satisfying any one {sufficient} constraint is enough for an inferencing engine to classify an instance.

The diagram below shows that when an instance with the property "has contract with" satisfies specific multiplicity ("1..*") and type constraints (of type 'Steering Wheel Manufacturer" or "Windshield Manufacturer") for the property's values, the instance meets a necessary and sufficient condition to be a member of the class "Car Manufacturer". Therefore, an inferencing engine would classify this as an instance of the class "Car Manufacturer". As discussed above, an instance meeting any one of these necessary and sufficient conditions is enough to classify the instance regardless of conditions on the values of any other properties with the {sufficient} constraint owned by the class "Car Manufacturer". The conditions on the values of these properties become necessary conditions on an instance for it to be a valid member of class "Car Manufacturer." Also, an instance meeting any one of these necessary and sufficient conditions is enough to distinguish instances of the class "Car Manufacturer' from its parent class "Manufacturer."
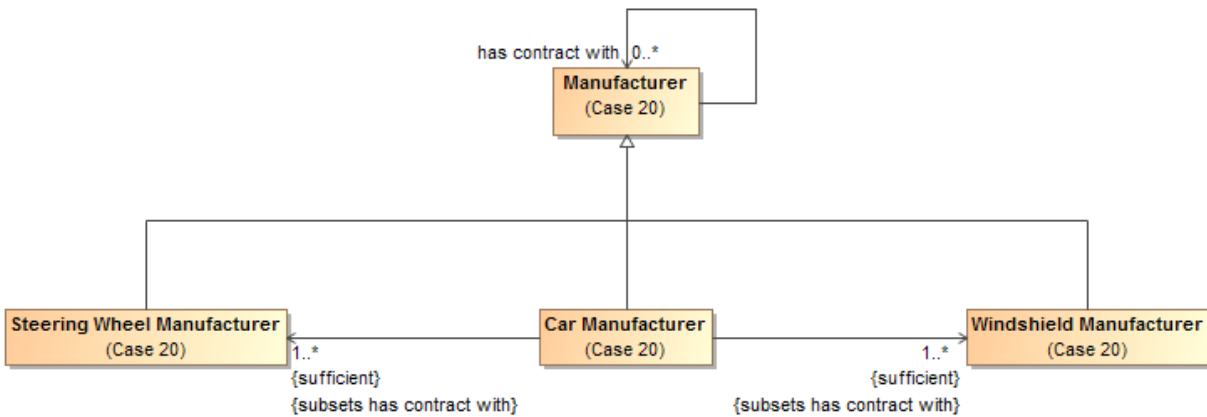
Figure 7   An example of necessary and sufficient condition

3.8   Generalization

A generalization is a subsumption relationship between a more general class and a more specific class. Every instance of the specific class is also an instance of the subsuming general class.

Because of this subsumption relationship, the specific class inherits all of the necessary conditions of the more general classifier.

For a simple example, if we define "Futsal Team" as a subclass of "Soccer Team", then the set of individuals in "Futsal Team" must be a subset of the set of individuals in "Soccer Team".
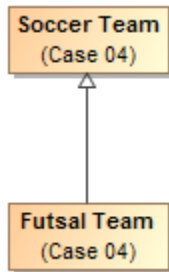

Figure 8   The relation between subclass 'Futsal Team' and class 'Soccer Team' represents generalization

There are four variations on generalization described in the following subsections. The first variation corresponds to the example above: overlapping and incomplete subclasses. That variation is the default in both UML and concept modeling.

3.8.1 Overlapping and Incomplete Subclasses

This variation is the default in both UML and in concept modeling. In this variation, an instance can be a member of the superclass and / or any number of subclasses. In this sense, the classification of instances is "incomplete"—sometimes there is a specific subclass, and sometimes there is not.

For example, the diagram below shows four instances. One is an instance of "Manufacturer", one is an instance of "Windshield Manufacturer", one is an instance of "Car Manufacturer", and one is an instance of both "Windshield Manufacturer" and "Car Manufacturer".
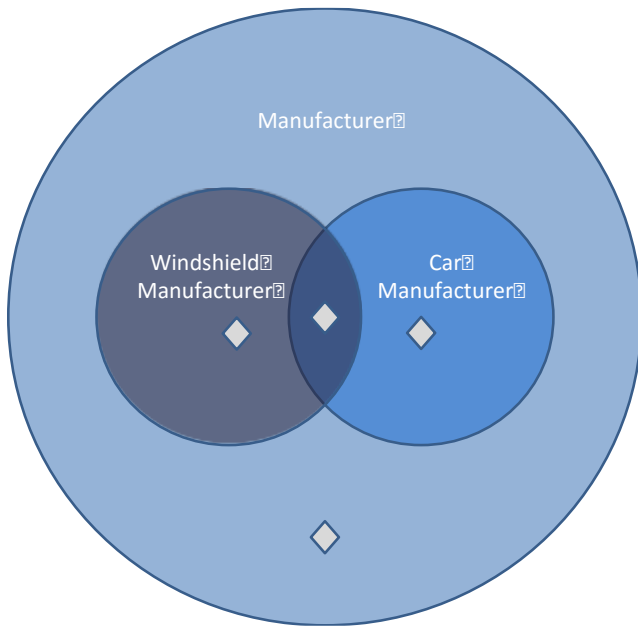
Figure 9   An example of incomplete instances

In both standard UML and in concept modeling, incomplete and overlapping subclasses are shown with either no notation, or with the notation {incomplete, overlapping} near the generalization arrow.
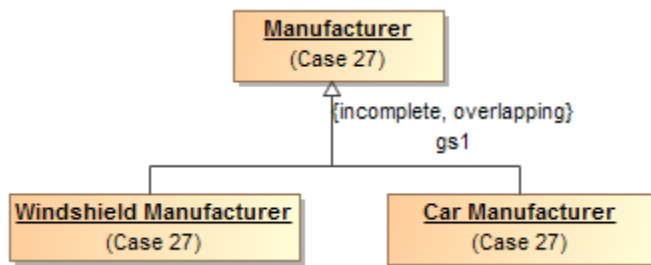

Figure 10   Incomplete and overlapping subclasses in standard UML notation

3.8.2 Disjoint Subclasses

This variation means that an instance can only be classified by one of the disjoint classes. Disjoint classes cannot have any overlap in their instances.

The diagram below shows three instances. One is an instance of "Cat", one is an instance of "Dog", and one is an instance of "Animal". An instance classified as both "Cat" and "Dog" is impossible because there is no overlap between the two classes. In the most basic terms, an instance of a "Cat" cannot be an instance of a "Dog", and vice versa.
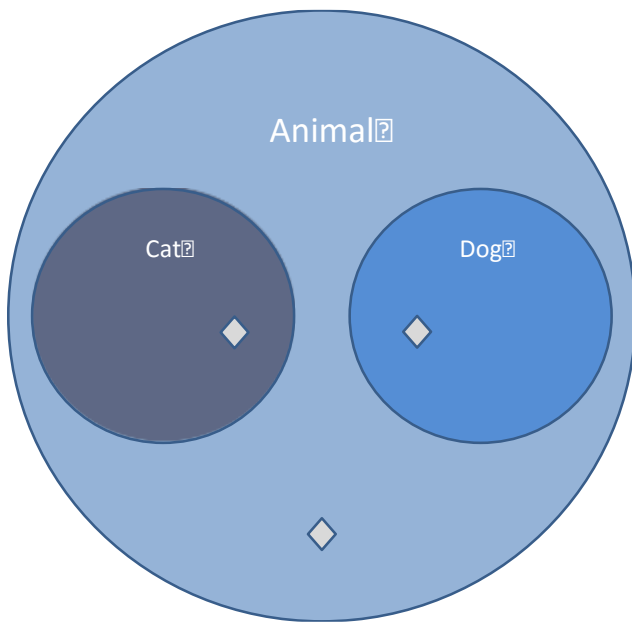
Figure 11   Disjoint instances

The following diagram shows an example of disjoint subclasses in standard UML notation. It shows that "Dog", "Cat", and "Mouse" are all subclasses of "Animal". In addition, the standard UML {incomplete, disjoint} notation declares all of the subclasses to be incomplete and disjoint. Intuitively, an instance of the subclass "Dog" is an instance of the superclass "Animal", but it cannot be an instance of the "Cat" or "Mouse" subclasses. Moreover, a lizard would be an instance of "Animal", but could not be an instance of any of the subclasses "Dog", "Cat", or "Mouse".
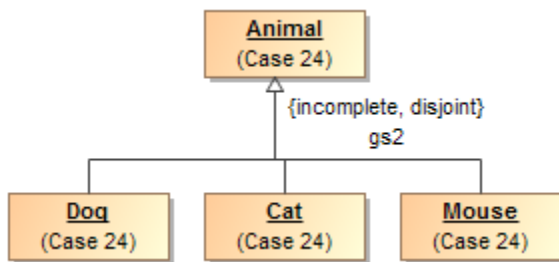
Figure 12   Incomplete and disjoint subclasses in standard UML notation

The Concept Modeler supports importing disjoint classes. A dependency stereotyped as «Disjoint With» will be used to specify disjoint subclasses. For example, the class Animal has two disjoint subclasses, Cat and Dog. When you import them to the Concept Modeler, the diagram will look similar to the example shown in the following figure.
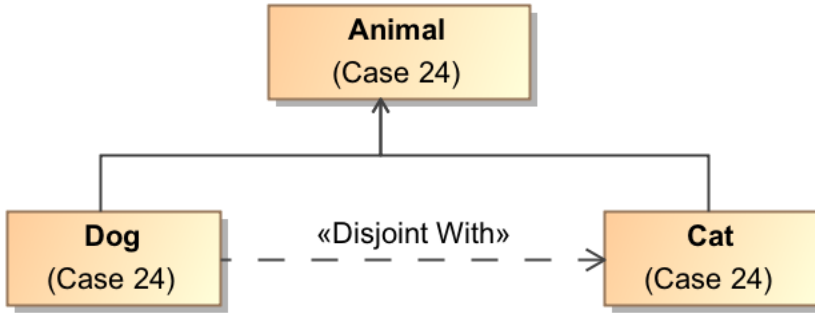
Figure 13   Imported disjoint subclasses are stereotyped with «Disjoint With»

3.8.3   Complete Subclasses

This variation means that an instance can only be classified by one of the subclasses; it cannot be classified by only the superclass. However, an instance of a subclass is indirectly an instance of a superclass at the same time.

For example, the following diagram shows three instances. One is an instance of "Windshield Manufacturer", one is an instance of "Car Manufacturer", and one is an instance of both "Car Manufacturer" and "Windshield Manufacturer". Note that there can be no instance of "Manufacturer" that is not also an instance of one of the subclasses.
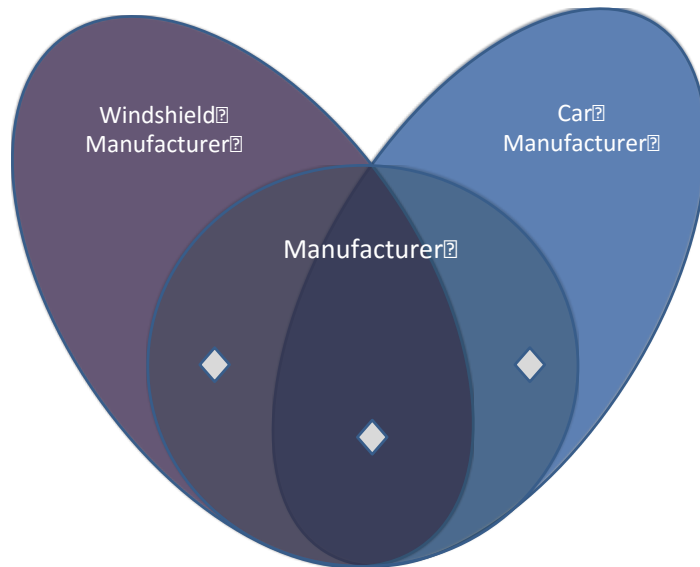
Figure 14   An example of complete subclasses

The diagram below shows an example of complete subclasses in standard UML notation. The diagram shows that "Steering Wheel Manufacturer", "Car Manufacturer", and "Windshield Manufacturer" are all subclasses of "Manufacturer". In addition, the standard UML {complete, overlapping} notation declares that the subclasses are complete and overlapping.
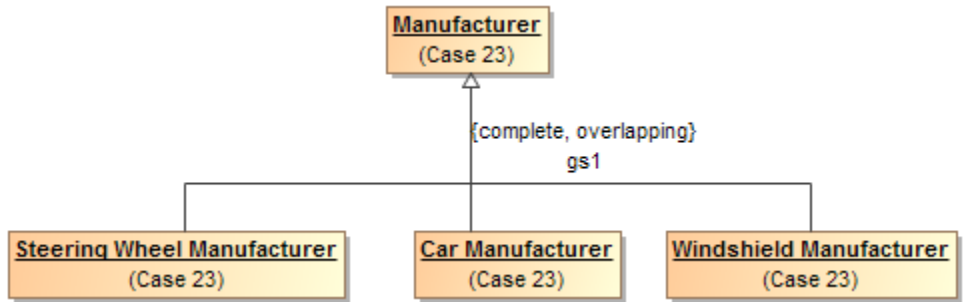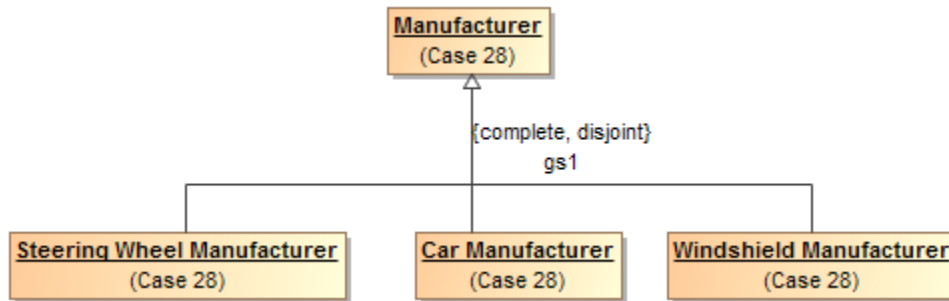
Figure 15   Complete subclasses in standard UML notation

3.8.4 Disjoint and Complete Subclasses

This variation means that an instance can only be classified by one of the subclasses. The instance cannot be classified as only the superclass, and it cannot be classified by two subclasses at the same time.

For example, in the subsequent diagram, two instances are shown. One is an instance of "Windshield Manufacturer", and one is an instance of "Car Manufacturer". There can be no instance of "Manufacturer" that is not also an instance of one of the subclasses, and there can be no instance that is classified as both a "Windshield Manufacturer" and a "Car Manufacturer" at the same time.
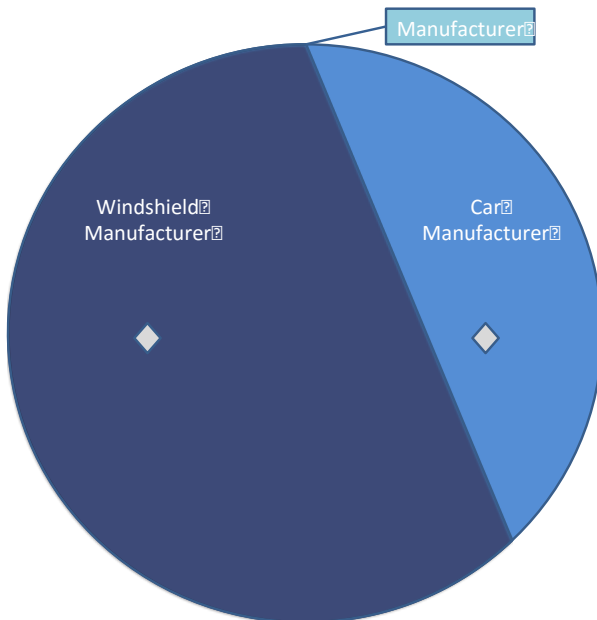
Figure 16   Disjoint and complete instances

The diagram below shows an example of disjoint and complete subclasses in standard UML notation. The diagram shows that "Steering Wheel Manufacturer", "Car Manufacturer", and

"Windshield Manufacturer" are all subclasses of "Manufacturer". In addition, the standard UML {complete, disjoint} notation declares that the subclasses are complete and disjoint.

Figure 17   Disjoint and complete subclasses in standard UML notation

3.9   Anonymous Union Class

An anonymous union is an unnamed class used to represent a set of classes that can be used as a type of a property. An anonymous union class always implies a complete subclass generalization. (See 3.8.3 Complete Subclasses.)

The following diagram states that an instance of a Person may have a value of type Cat or Dog for the *cares for* property. The diagram also states that an instance of a Cat or a Dog may have a value of type Person for the *cared for by* property.
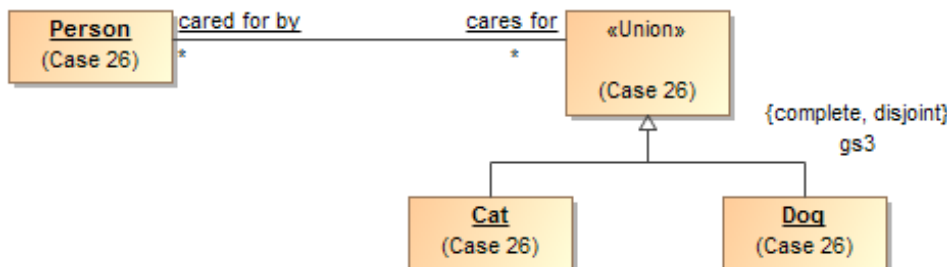
Figure 18   An anonymous union class

In an ontology, if anonymous union, with same classes within the union, is used in multiple places, the Concept Modeler can distinguish it when importing the ontology. In other words, if the anonymous union has the same union members, the Concept Modeler will identify it as the same anonymous union.

3.10 Inverse Properties

A property is a unidirectional relation between two classes, or between a class and a datatype. In the case in which there is a relation between two classes, it is often useful to define a property that goes in the opposite direction. For example, if a Video Game Company **manufactures** Video Game Consoles, the opposite would be that a Video Game Console is **manufactured by** a Video

Game Company. Rather than draw two separate unidirectional associations, properties drawn on opposite ends of one association are *inverses* of one another. When an instance has a value for a property that has an inverse defined, a reasoner can infer that an opposite value also exists, and automatically create it.

The next diagram asserts that for every (Video Game Console, Video Game Company) related by the **manufactures** property, there is a corresponding (Video Game Company, Video Game Console) related by the **manufactured by** property.
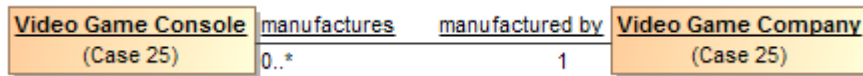
Figure 19 Inverse properties shown on opposite ends of association

In most cases, when importing an OWL ontology, information in OWL is enough to assert that two properties are inverse of each other. However, if the definition is insufficient to prove that two are inverse of each other or which class owns the property and what the type is, the Concept Modeler will create two unidirectional associations and use a stereotyped dependency «inverse of» between the properties to show that they are inverse of one another.

3.11 Property Restrictions

In the Concept Modeler, a property restriction appears as one with a {subsets} or {redefines}. The Concept Modeler will import each property restriction as a unidirectional association between the two concepts.

3.12 Annotation and Annotation Properties

The OWL language provides a way to comment on any subject that has a URI, using *annotations*. One can annotate classes, properties, and ontologies. In addition to providing a way to comment on a subject, the OWL language provides an open-ended way to define *annotation properties*. An annotation property defines a type of annotation with a relatively precise meaning.

Every annotation is a value for an annotation property. An annotation describes some subject URI using an annotation property URI and a (usually textual) value, forming what is called a *triple*. For example, a well-known vocabulary called *Dublin Core* defines an annotation property that has the URI http://purl.org/dc/terms/description. That annotation property is what the Concept Modeler uses by default to document a class called Person. It forms the triple http://example.com/ontology/Person http://purl.org/dc/terms/description "A human being".

The Concept Modeler allows the user to define any number of annotation types. The user does this by declaring that a property is an *annotation property* using the «Annotation Property» stereotype on a UML property. Alternatively, a user can import annotation properties from

existing OWL ontologies. When the Concept Modeler imports annotation properties, it automatically applies the «Annotation Property» stereotype.

Any UML comment can be exported as an OWL annotation. By default, the concept modeler converts UML element documentation, notes, and comments into the Dublin Core annotation property http://purl.org/dc/terms/description. When the user would like to use some other annotation property, he or she can specify which one as a tagged value in a UML comment stereotyped as an «Annotation». Applying this stereotype allows one to use any annotation property to provide more precise meaning for the information the comment contains and to properly export the comment into OWL.

For example, the following diagram illustrates a UML comment stereotyped with «Annotation» to document the concept *Item*. It uses a specific kind of annotation property, called *explanatory note,* to provide context for that documentation. That annotation property definition is shown as the third attribute stereotyped as an «Annotation Property» in the pink property holder. Its usage as a tagged value is shown as "{annotationProperty = explanatory note}" in the UML Comment stereotyped as an «Annotation». Please see the normal MagicDraw documentation for how to create a tagged value for a stereotype in the specification window.

You may have an imported model that contains annotation properties and documentation. A UML Comment can have more than one «Annotation» and each of these «Annotation»s can also have their own annotation property tagged value.

3.13 Preferred Annotation Property

The **Project Options** in the Concept Modeler provides a **Preferred annotation property** option for the UML Comments or «Annotation»s in a model. This capability allows you to control which annotation property becomes the "special" MagicDraw comment providing documentation, when you enter text into the Documentation pane or create a new Annotation. It also allows the documentation to be included in the Natural Language Glossary. If the UML Comments and the «Annotation» Comments do not have annotation property value assigned (annotationProperty Tag from Specifications), then they should be treated as if they have "Preferred annotation property" assigned from Project options.

Changing a preferred annotation property tagged value will cause the ownership of existing UML comments and «Annotation»s to change, for example:

- Any UML comment and «Annotation» that was using the old preferred annotation will be moved from being owned by the annotated element to being owned by the next-higher package.
- Any UML comment and «Annotation» that is using the new preferred annotation property will be moved to being owned by the annotated element.

Note	- Changing the preferred annotation property tagged value of UML comments and «Annotation»s will change their ownership. - Changing in ownership causes a change in the Documentation pane (on the lower-left) and what shows up in the Natural Language Glossary because the first UML Comment or «Annotation» that is owned by an annotated element is considered "documentation" in MagicDraw.

If you do not select an annotation property tagged value for any UML comment or «Annotation» in your model, any time we export it to an OWL ontology, the Concept Modeler will use the preferred annotation property that does not have a tagged value, which is *http://purl.org/dc/terms/description*[3]. This preferred annotation property without a tagged value (unspecified) in the Concept Modeler is active until you change it through the **Project Options** dialog. Selecting **<UNSPECIFIED>** from the **Project Options** dialog will remove the current preferred annotation property tagged value.
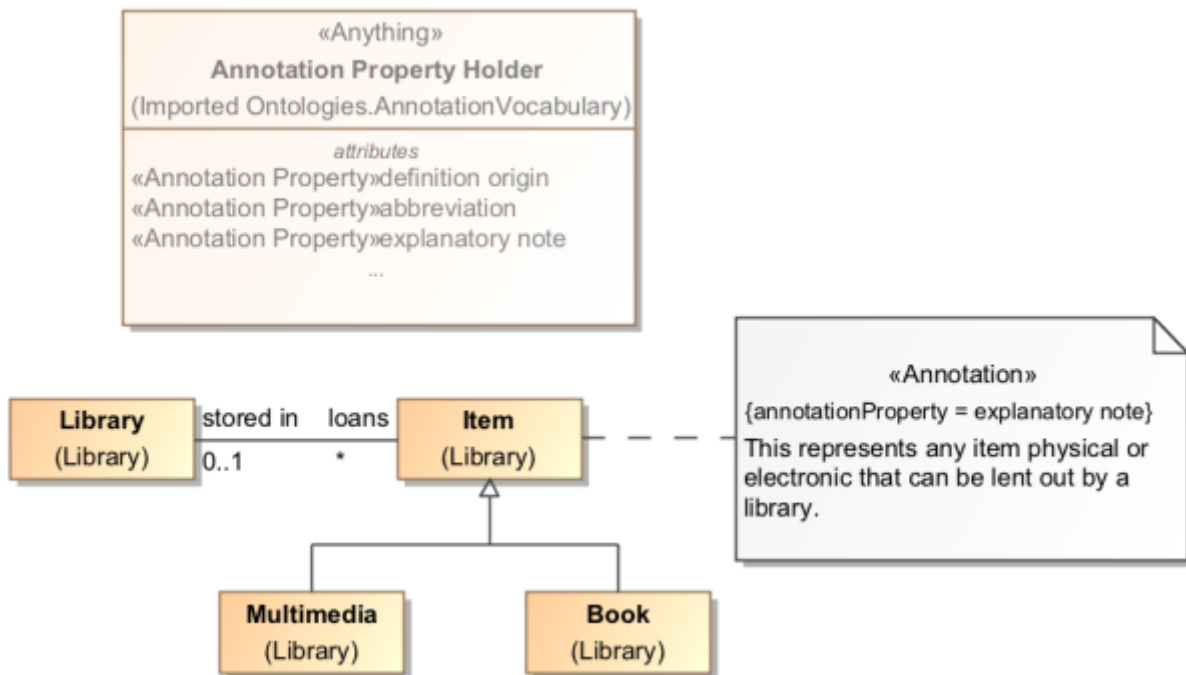
Figure 20   A comment stereotyped with «Annotation» to provide a correct meaning of an item

The following diagram is used as an example to show you how the Preferred annotation property option works. In the figure, a class **autonomous** agent is plotted on the diagram along with five annotations.
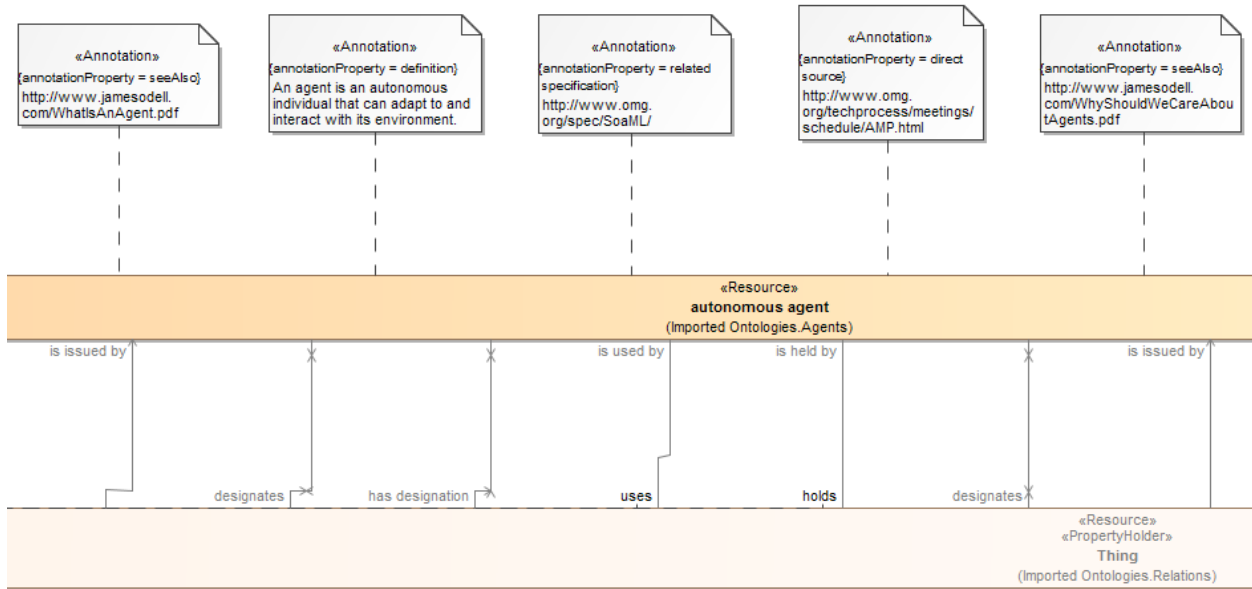
Figure 21   A class with annotations in the diagram pane

The first tree view example shows you how they are structured in the Containment tree before you select a preferred annotation property from the **Project Options** dialog. Notice that the **Documentation** pane shows no annotation when the class is clicked. Using the above instructions, we select the **definition** as the preferred annotation property for the «Annotation» with the (annotationProperty = definition).

Copyright © 2017, No Magic, Inc.
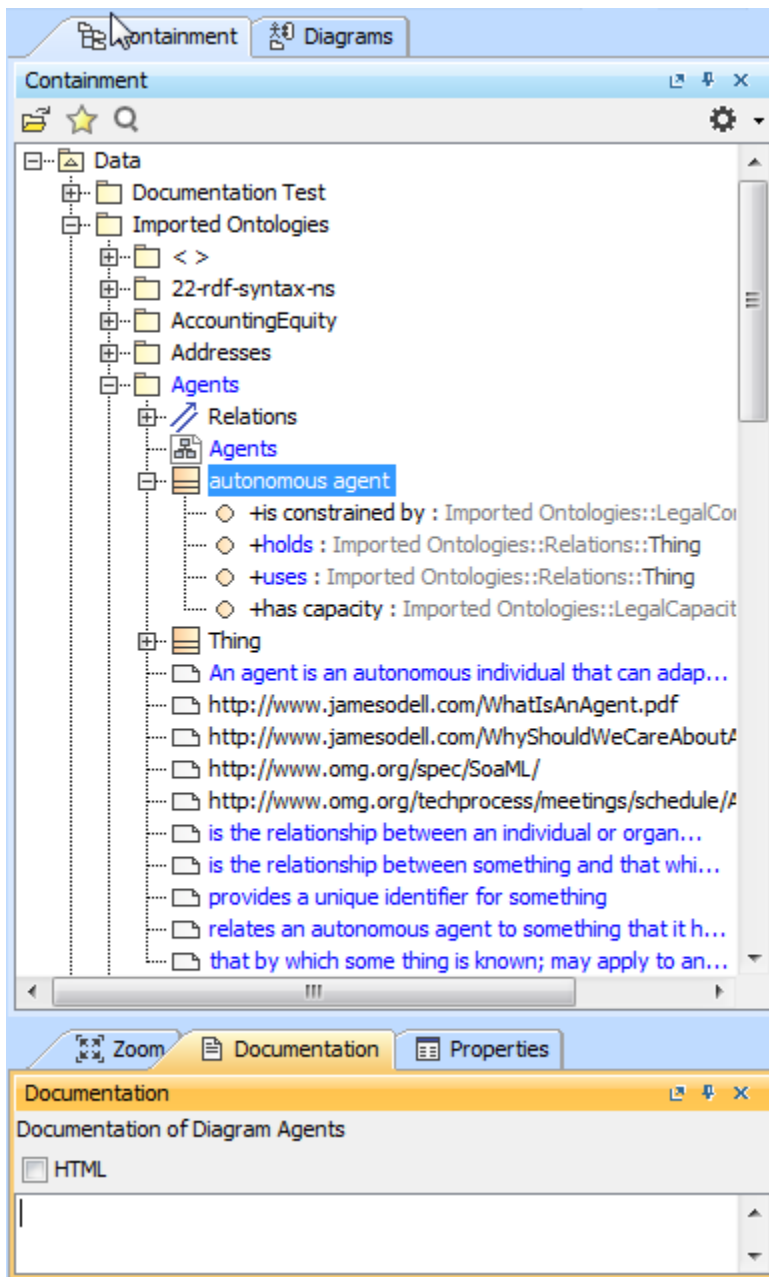
Figure 22   Annotations in the Containment tree before the Preferred annotation property is selected

In the second tree view example below, see how the annotation ownership changed after you selected **definition** as the preferred annotation property. If you open the Specification dialog of the «Annotation» with the (annotationProperty = definition), you will see that the new owner is **autonomous agent** (previously, it was the package **Agents**'). The **Documentation** pane will also show the annotation if you click the class **autonomous agent**.
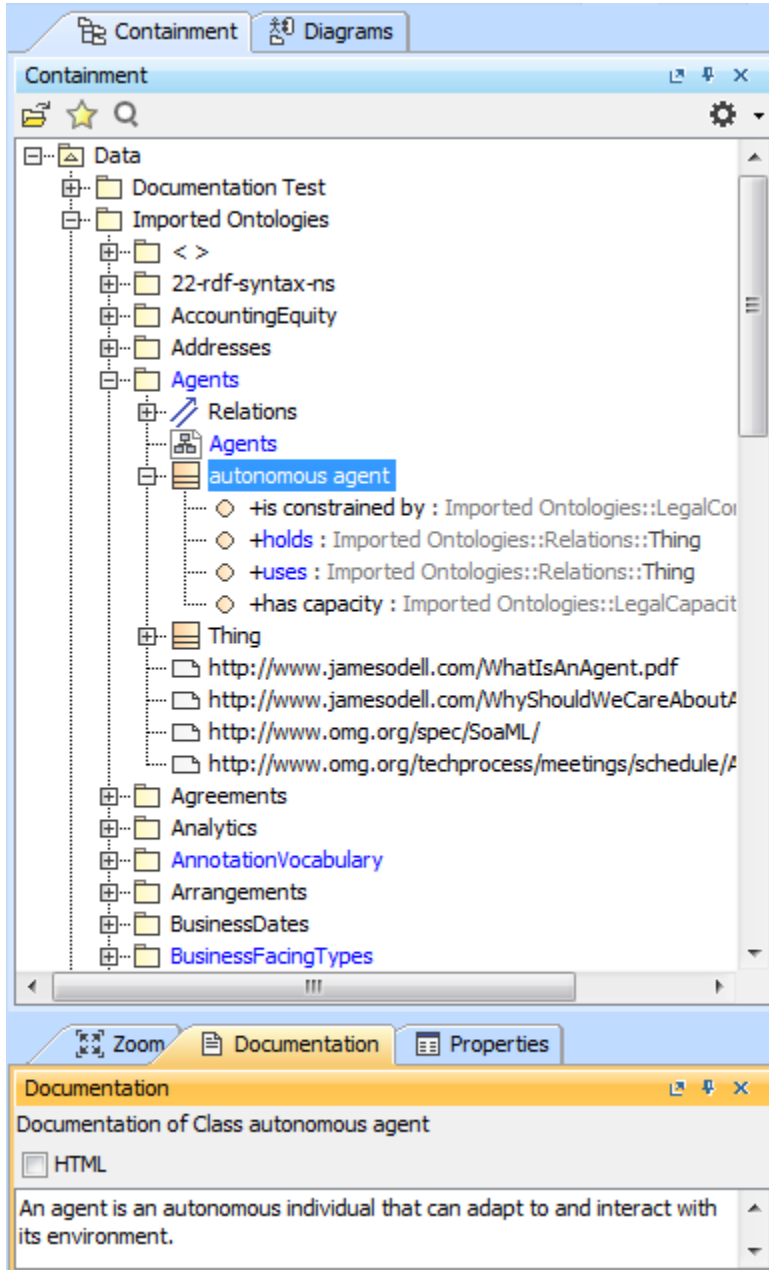
Figure 23   The annotation appears in the Documentation pane after the preferred annotation property is selected

3.14 Property Chain

The Concept Modeler can import property chains. A property chain is useful for composing a property from two or more other properties that are put together in a chain. It defines the property with reference to the other properties. The property chain allows you to navigate from a starting class (the one with the stereotype «Subproperty Chain») through a chain of properties that take a path through multiple classes.

A property chain is an ordered list of composed properties, therefore, it should have two or more properties in the chain. The same property can appear more than once in a chain. For example, "has parent • has parent" is a subproperty of "has grandparent".

Note	- An existential or universal quantification restriction *cannot* have or be a part of a subproperty chain, although the property it restricts *can*. - A sub-property *can* have or be part of a subproperty chain for another property.

The following example describes a Person class that has two instances "Female Person" and "Male Person", and four properties "has parent", "has father", "has uncle", and "has brother". The stereotype of the property "has uncle" will be **«Subproperty Chain»** of **Element** type and the tagged value is **chain = has father, has brother**.
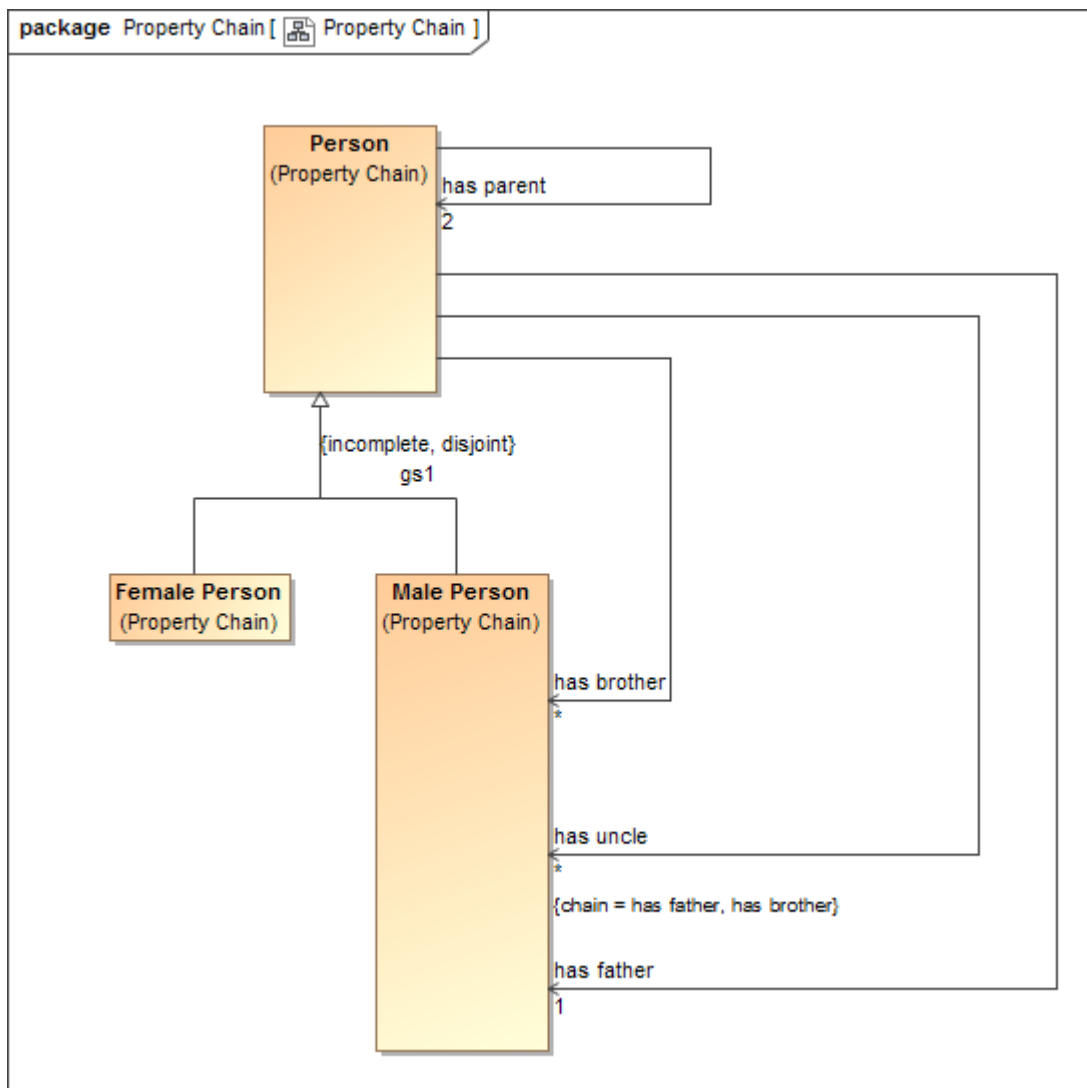
Figure 24   Property Chain in the Concept Modeler

For more information on how to create a property chain, see section 5.2.2 Create a Property Chain.

3.15 Equivalent Properties

The Concept Modeler allows you to represent, import, and export equivalent properties in a model. You can make two or more properties equivalent to each other by applying the stereotype **«Equivalent Property»** to the target property and the tagged value "**equivalent to**" the equivalent property.

Note	• An existential or universal quantification restriction *cannot* have or be an equivalent property, although the property it restricts *can*. • A sub-property *can* have or be an equivalent property.

The following figure shows the equivalent properties in a diagram.
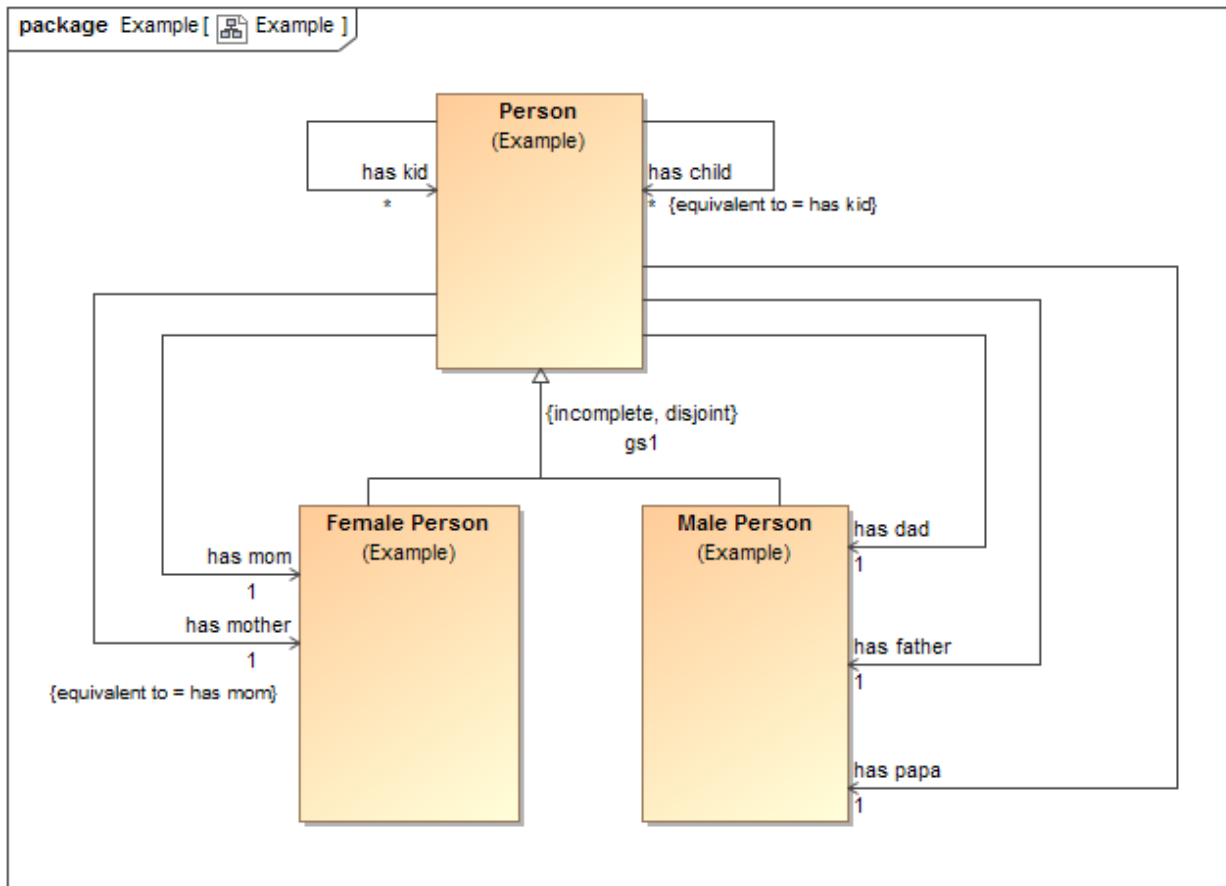
Figure 25   Equivalent properties in the Concept Modeler

In the example, the property "has mother" is equivalent to the property "has mom". For more information on how to create equivalent properties, see section 5.2.3 Create Equivalent Property.

3.16 Equivalent Classes

The Concept Modeler can specify equivalence between two classes, import equivalent classes from OWL, and export equivalent classes to OWL. Class equivalence expresses a generalization relationship stereotyped as **«Equivalent Class»**. The Concept Modeler will draw this with a double-headed arrow.

The following figure shows two equivalent classes in a diagram.
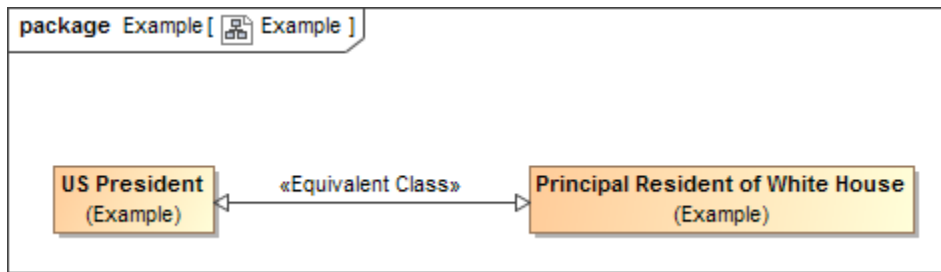
Figure 26   Two Equivalent Classes in the Concept Modeler

In the example, the equivalence class arrow defines that the two classes are semantically equivalent to each other. For more information on how to create equivalent classes, see section 5.2.4 Create Equivalent Classes.

If you would like to look at equivalent class relations which are identified by «Equivalent Class» stereotype, there are several possibilities of results depending on the types of classes. Consider the two related classes called Class 1 and Class 2. After perusing through all of these cases, please refer to the figure below the text to see a brief summary of some of these cases.

- If both classes are Class but they do not share the same name, then one class will show "Equivalent to" and the rest will show "See."
- If both classes are Class and they do share the same name, then it will appear as one class with all the properties from both classes.
- If Class is equivalent to a Union, then the Class will be shown with its properties and the subclasses of Union will be shown under Class.
- If both classes are Unions then both unions will be merged and follow the same pattern as the Class/Union combinations explained above.

Remark: Unions should not have names. If a Union has a name, then it's a Class.

The following information explains more explicitly how to determine which Class will show "Equivalent to" and which will show "See."

- If only one class has documentation:
  - This class will show 'Equivalent to' and list the rest of the classes it is equivalent to.
  - The rest of the classes will show 'See' with the link going back to the class that shows 'Equivalent to'
- More than one class has documentation:
  - First class with documentation in alphabetical order will show 'Equivalent to' and list rest of the classes it is equivalent to
  - The rest of the classes will show 'See' with the link going back to the class that shows 'Equivalent to'
- None of the classes have documentation
  - First class in alphabetical order will show 'Equivalent to' and list rest of the classes it is equivalent to

o   The rest of the classes will show 'See' with the link going back to the class that shows 'Equivalent to'

Remarks:

1. The class that shows 'Equivalent to', will list all properties and annotations from all the classes it is equivalent to.
2. The class that shows 'See', will not show any of the properties or annotations.
3. All same named classes will show '(from package {qualified class name})' to differentiate each same named class.

A

    A

        Model-Generated Definition:

                Equivalent to B.

        Definition:

B

    B

        See A.
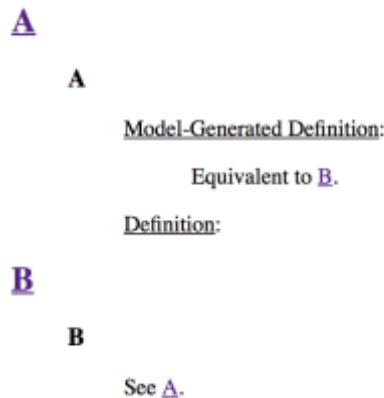
Figure 27   Segmented shots of a report showing the merging of all the equivalent classes in the project.

4   UML to Equivalent OWL (in OWL Functional Syntax)

There are various syntaxes available for encoding OWL ontologies. The Concept Modeler can export UML to an OWL ontology using the following syntaxes:

- **RDF/XML**. It is the originally standard syntax for writing RDF (Resource Description Framework), which is a general-purpose language for representing information in the Web. Though verbose and difficult to read, it is the only syntax that is mandatory to be supported by OWL 2 tools. It provides an XML representation of an RDF graph. This syntax is the default syntax used in the Concept Modeler.
- **JSON-LD** or JavaScript Object Notation for Linked Data is a method of encoding linked data using JSON, which is a concrete RDF syntax. JSON-LD is used to map JSON terms (keys and values) to IRIs, giving them a global context. A JSON-LD document is both an RDF document and a JSON document.
- **OWL Functional**. It is a simple text-based syntax designed to be easier for specification purposes and to provide a foundation for the implementation of OWL 2 tools such as APIs and reasoners. It is used in most of the OWL 2 specification documents as the primary presentation syntax that translates the structural specification into other concrete

syntaxes. A functional-style syntax ontology document consists of sequences of Unicode characters and is encoded in UTF-8.

- **Turtle**. A concrete syntax for RDF, Turtle (Terse RDF Triple Language) is a plain-text RDF representation. It is more concise and easier to read and edit manually than RDF/XML. A Turtle document is a collection of RDF-triples. Each triple has the format: `<subject> <predicate> <object>`. Each statement ends with a period and each element in the triple is an URI, except the `<object>`, which can be a bit of text or a number.
- **Manchester**. It provides a compact textual-based representation of OWL ontologies that is easy to read and write. It uses IRIs as term identifiers. The syntax for annotations and descriptions in the Manchester OWL syntax closely corresponds to the syntax in the OWL Functional syntax. A Manchester OWL document consists of sequences of Unicode characters and is encoded in UTF-8.

Below, examples are given that show the transformation of UML modeled in the Concept Modeler to an exported OWL ontology. The OWL ontologies are presented in OWL Functional Syntax.

For a simple UML class, the diagram below shows that the ontology is transformed as the package containing the UML class. Subsequent diagrams do not show the package in the diagram for the sake of brevity.

Note	- A model may contain elements, for example, classes, properties, datatypes, or generalizations, that belong to other models. When exporting the model, the Concept Modeler will show the OWL declaration of the elements that exist in the current model only, not those of the other models. - However, if the entity that belongs to another model is an object property with an inverse property defined, you will see the OWL declaration of the inverse property in the current OWL ontology upon export.

4.1   Class
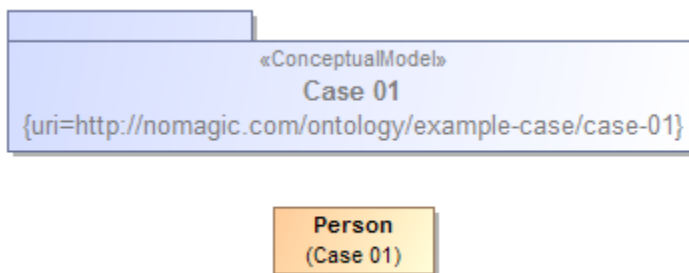
Figure 28   A class diagram in Concept Modeler

Copyright © 2017, No Magic, Inc.

```
Ontology(<http://nomagic.com/ontology/example-case/case-01>
        Declaration(
                Class(:Person)
        )
        AnnotationAssertion(rdfs:label :Person "Person"@en)
)
```

4.2   Class Generalization
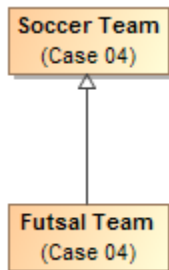
Figure 29   Generalization in Concept Modeler

```
Ontology(<http://nomagic.com/ontology/example-case/case-04>
        Declaration(
                Class(:FutsalTeam)
        )
        Declaration(
                Class(:SoccerTeam)
        )
        AnnotationAssertion(rdfs:label :FutsalTeam "Futsal Team"@en)
        SubClassOf (:FutsalTeam :SoccerTeam)
        AnnotationAssertion(rdfs:label :SoccerTeam "Soccer Team"@en)
)
```

4.3   Generalization with Disjoint Subclasses
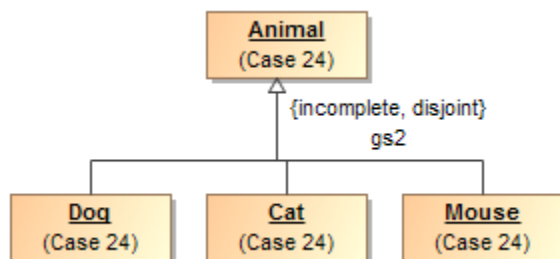
Figure 30   Generalization with disjoint subclasses

Copyright © 2017, No Magic, Inc.

Ontology(<http://nomagic.com/ontology/example-case/case-24>

    Declaration(

        Class(:Animal)

)

    Declaration(

        Class(:Cat)

)

    Declaration(

        Class(:Dog)

)

    Declaration(

        Class(:Mouse)

)

    SubClassOf(:Cat :Animal)

    SubClassOf(:Dog :Animal)

    SubClassOf(:Mouse :Animal)

    DisjointClasses(:Cat :Dog)

    DisjointClasses(:Cat :Mouse)

    DisjointClasses(:Dog :Mouse)

    AnnotationAssertion(rdfs:label :Animal "Animal"@en)

    AnnotationAssertion(rdfs:label :Cat "Cat"@en)

    AnnotationAssertion(rdfs:label :Dog "Dog"@en)

    AnnotationAssertion(rdfs:label :Mouse "Mouse"@en)

)

4.4 Generalization with Subclass Completeness
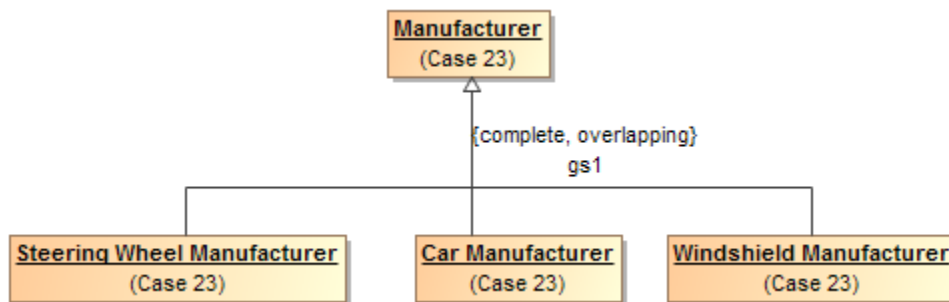
Figure 31 Generalization with complete subclasses

Ontology(<http://nomagic.com/ontology/example-case/case-23>

    Declaration(

        Class(:CarManufacturer)

)
Declaration(
        Class(:Manufacturer)
)
Declaration(
        Class(:SteeringWheelManufacturer
)
Declaration(
        Class(:WindshieldManufacturer)
)
SubClassOf(:CarManufacturer :Manufacturer)
SubClassOf(:SteeringWheelManufacturer :Manufacturer)
SubClassOf(:WindshieldManufacturer :Manufacturer)
EquivalentClasses(:Manufacturer ObjectUnionOf(:CarManufacturer
:SteeringWheelManufacturer :WindshieldManufacturer))
AnnotationAssertion(rdfs:label :CarManufacturer "Car Manufacturer"@en)
AnnotationAssertion(rdfs:label :Manufacturer "Manufacturer"@en)
AnnotationAssertion(rdfs:label :SteeringWheelManufacturer "Steering Wheel
Manufacturer"@en)
AnnotationAssertion(rdfs:label :WindshieldManufacturer "Windshield
Manufacturer"@en)
)

4.5   Anonymous Union Class
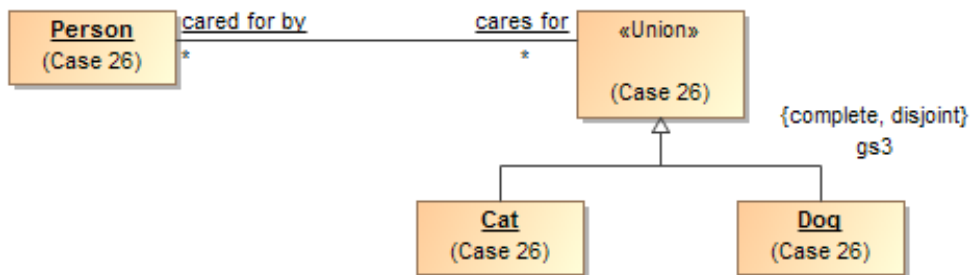
Figure 32   Anonymous union class

Ontology(<http://nomagic.com/ontology/example-case/case-26>
        Declaration(
                Class(:Cat)
)
        Declaration(
                Class(:Dog)
)

```
Declaration(
        Class(:Person)
)
Declaration(
        ObjectProperty(:caredForBy)
)
Declaration(
        ObjectProperty(:caresFor)
)
AnnotationAssertion(rdfs:label :Cat "Cat"@en)
DisjointClasses(:Cat :Dog)
AnnotationAssertion(rdfs:label :Dog "Dog"@en)
AnnotationAssertion(rdfs:label :Person "Person"@en)
AnnotationAssertion(rdfs:label :caredForBy "cared for by"@en)
InverseObjectProperties(:caredForBy :caresFor)
ObjectPropertyDomain(:caredForBy ObjectUnionOf(:Dog :Cat))
ObjectPropertyRange(:caredForBy :Person)
AnnotationAssertion(rdfs:label :caresFor "cares for"@en)
ObjectPropertyDomain(:caresFor :Person)
ObjectPropertyRange(:caresFor ObjectUnionOf(:Dog :Cat))
)
```

4.6   Class with Datatype Property
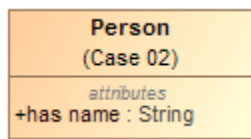
Figure 33   A class with datatype property

```
Ontology(<http://nomagic.com/ontology/example-case/case-02>
        Import(<http://www.omg.org/spec/PrimitiveTypes/20100901>)
        Declaration(
                Class(:Person)
        )
        Declaration(
                DataProperty(:hasName)
        )
        Declaration(
                AnnotationProperty(<http://purl.org/dc/terms/description>)
```

)
Declaration(

       Datatype(xsd:string)

)

AnnotationAssertion(rdfs:label :Person "Person"@en)

SubClassOf(

       :Person

       ObjectIntersectionOf(

              DataMaxCardinality(1 :hasName xsd:string)

              DataMinCardinality(1 :hasName xsd:string)

)

)

AnnotationAssertion(rdfs:label :hasName "has name"@en)

DataPropertyDomain(:hasName :Person)

DataPropertyRange(:hasName xsd:string)

AnnotationAssertion(http://purl.org/dc/terms/description
<http://www.omg.org/spec/PrimitiveTypes/20100901#String> "An instance of String defines a piece of text. The semantics of the string itself depends on its purpose, it can be a comment, computational language expression, OCL expression, etc. It is used for String attributes and String expressions in the metamodel."@en)
)

4.7   Class with Self-Referential Object Property
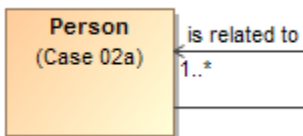
Figure 34   A class with self-referential object property

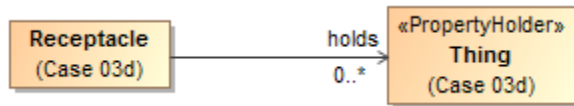Ontology(<http://nomagic.com/ontology/example-case/case-02a>

       Declaration(

              Class(:Person)

)

       Declaration(

              ObjectProperty(:isRelatedTo)

)

       AnnotationAssertion(rdfs:label :Person "Person"@en)

       SubClassOf(

              :Person

```
                    ObjectMinCardinality(1 :isRelatedTo :Person)
        )
        AnnotationAssertion(rdfs:label :isRelatedTo "is related to"@en)
        ObjectPropertyDomain(:isRelatedTo :Person)
        ObjectPropertyRange(:isRelatedTo :Person)
)
```

4.8   Class with Object Property
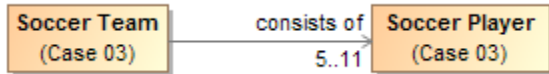
Figure 35   A class with object property

```
Ontology(<http://nomagic.com/ontology/example-case/case-03>
        Declaration(
                Class(:SoccerPlayer)
        )
        Declaration(
                Class(:SoccerTeam)
        )
        Declaration(
                ObjectProperty(:consistsOf)
        )
        AnnotationAssertion(rdfs:label :SoccerPlayer "Soccer Player"@en)
        AnnotationAssertion(rdfs:label :SoccerTeam "Soccer Team"@en)
        SubClassOf(
                :SoccerTeam
                ObjectIntersectionOf(
                        ObjectMaxCardinality(11 :consistsOf :SoccerPlayer)
                        ObjectMinCardinality(5 :consistsOf :SoccerPlayer)
                )
        )
        AnnotationAssertion(rdfs:label :consistsOf "consists of"@en)
        ObjectPropertyDomain(:consistsOf :SoccerTeam)
        ObjectPropertyRange(:consistsOf :SoccerPlayer)
)
```

Copyright © 2017, No Magic, Inc.

4.9 Property Holder with Datatype Property

Figure 36 A property holder with datatype property

Ontology(<http://nomagic.com/ontology/example-case/case-03a>
    Import(<http://www.omg.org/spec/PrimitiveTypes/20100901>)
    Declaration(
        DataProperty(:hasName)
)
    Declaration(
        AnnotationProperty(<http://purl.org/dc/terms/description>)
)
    Declaration(
        Datatype(xsd:string)
)
    SubClassOf(
        owl:Thing
        ObjectIntersectionOf(
            DataMaxCardinality(3 :hasName xsd:string)
            DataMinCardinality(2 :hasName xsd:string)
)
)
    AnnotationAssertion(rdfs:label :hasName "has name"@en)
    DataPropertyRange(:hasName xsd:string)
    AnnotationAssertion(http://purl.org/dc/terms/description
    <http://www.omg.org/spec/PrimitiveTypes/20100901#String> "An instance of String
    defines a piece of text. The semantics of the string itself depends on its purpose, it can be
    a comment, computational language expression, OCL expression, etc. It is used for String
    attributes and String expressions in the metamodel."@en)
)

4.10 Property Holder with Self-Referential Object Property
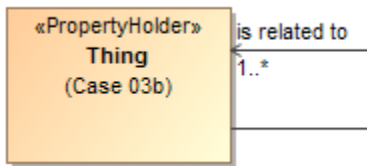
Figure 37   A property holder with self-referential object property

Ontology(<http://nomagic.com/ontology/example-case/case-03b>
     Declaration(
         ObjectProperty(:isRelatedTo)
)
     SubClassOf(
         owl:Thing
             ObjectMinCardinality(1 :isRelatedTo)
)
     AnnotationAssertion(rdfs:label :isRelatedTo "is related to"@en)
)

4.11 Property Holder with Object Property
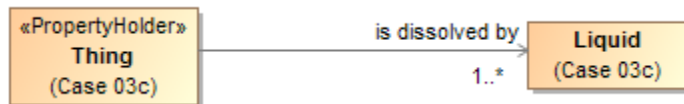
Figure 38   A property holder with object property

Ontology(<http://nomagic.com/ontology/example-case/case-03c>
     Declaration(
         Class(:Liquid)
)
     Declaration(
         ObjectProperty(:isDissolvedBy)
)
     AnnotationAssertion(rdfs:label :Liquid "Liquid"@en)
     SubClassOf(
         owl:Thing
             ObjectMinCardinality(1 :isDissolvedBy :Liquid)
)
     AnnotationAssertion(rdfs:label :isDissolvedBy "is dissolved by"@en)
     ObjectPropertyRange(:isDissolvedBy :Liquid)

)

4.12 Class with Object Property without Range

Figure 39   A class with object property without range

Ontology(<http://nomagic.com/ontology/example-case/case-03d>
    Declaration(
        Class(:Receptacle)
)
    Declaration(
        ObjectProperty(:holds)
)
    AnnotationAssertion(rdfs:label :Receptacle "Receptacle"@en)
    AnnotationAssertion(rdfs:label :holds "holds"@en)
    ObjectPropertyDomain(:holds :Receptacle)
)

4.13 Class with Subproperty
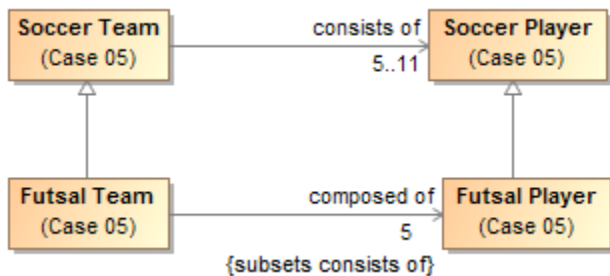
Figure 40   A class with subproperty

Ontology(<http://nomagic.com/ontology/example-case/case-05>
    Declaration(
        Class(:FutsalPlayer)
)
    Declaration(
        Class(:FutsalTeam)
)

```
Declaration(
        Class(:SoccerPlayer)
)
Declaration(
        Class(:SoccerTeam)
)
Declaration(
        ObjectProperty(:composedOf)
)
Declaration(
        ObjectProperty(:consistsOf)
)
AnnotationAssertion(rdfs:label :FutsalPlayer "Futsal Player"@en)
SubClassOf(:FutsalPlayer :SoccerPlayer)
AnnotationAssertion(rdfs:label :FutsalTeam "Futsal Team"@en)
SubClassOf(:FutsalTeam :SoccerTeam)
SubClassOf(
        :FutsalTeam
        ObjectIntersectionOf(
                ObjectMaxCardinality(5 :composedOf :FutsalPlayer)
                ObjectMinCardinality(5 :composedOf :FutsalPlayer)
        )
)
AnnotationAssertion(rdfs:label :SoccerPlayer "Soccer Player"@en)
AnnotationAssertion(rdfs:label :SoccerTeam "Soccer Team"@en)
SubClassOf(
        :SoccerTeam
        ObjectIntersectionOf(
                ObjectMaxCardinality(11 :consistsOf :SoccerPlayer)
                ObjectMinCardinality(5 :consistsOf :SoccerPlayer)
        )
)
AnnotationAssertion(rdfs:label :composedOf "composed of"@en)
SubObjectPropertyOf(:composedOf :consistsOf)
ObjectPropertyDomain(:composedOf :FutsalTeam)
ObjectPropertyRange(:composedOf :FutsalPlayer)
AnnotationAssertion(rdfs:label :consistsOf "consists of"@en)
ObjectPropertyDomain(:consistsOf :SoccerTeam)
ObjectPropertyRange(:consistsOf :SoccerPlayer)
)
```

4.14 Class with Universal Quantification Constraint on Property I
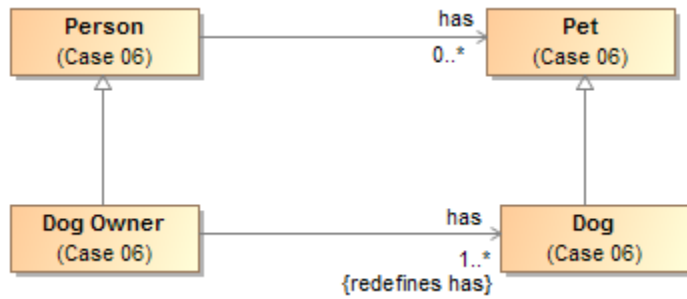
Figure 41   A class with universal quantification constraint on property I

Ontology(<http://nomagic.com/ontology/example-case/case-06>
    Declaration(
        Class(:Dog)
)
    Declaration(
        Class(:DogOwner)
)
    Declaration(
        Class(:Person)
)
    Declaration(
        Class(:Pet)
)
    Declaration(
        ObjectProperty(:has)
)
    AnnotationAssertion(rdfs:label :Dog "Dog"@en)
    SubClassOf(:Dog :Pet)
    AnnotationAssertion(rdfs:label :DogOwner "Dog Owner"@en)
    SubClassOf(:DogOwner :Person)
    SubClassOf(
        :DogOwner
        ObjectIntersectionOf(
            ObjectMinCardinality(1 :has :Dog)
            ObjectAllValuesFrom(:has :Dog)
)
)
    AnnotationAssertion(rdfs:label :Person "Person"@en)

AnnotationAssertion(rdfs:label :Pet "Pet"@en)

AnnotationAssertion(rdfs:label :has "has"@en)

ObjectPropertyDomain(:has :Person)

ObjectPropertyRange(:has :Pet)

)

4.15 Class with Universal Quantification Constraint on Property II

This example differs from the previous example primarily because the superclasses "Person" and "Pet" are from a different package than their subclasses "Dog Lover" and "Dog," respectively. This difference is reflected in the OWL ontology by the import of this namespace.

As shown below in the next diagram, the superclasses "Person" and "Pet", defined in the package "Case 06", are a different color and a lighter shade than the classes defined in the package "Case 07". This color differentiation is to distinguish them from the classes defined on this diagram. MagicDraw's AutoStyler plugin can automatically set the display properties for classes and other UML elements using the "defined elsewhere" style; that is, when they are shown on a non-defining diagram for the UML element (see section 2.2 Automatic Styling of Concept Models).
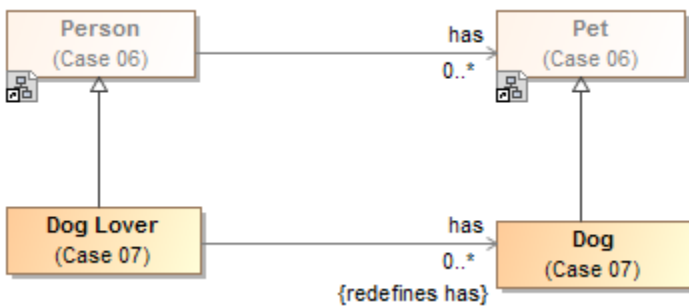
Figure 42   A class with universal quantification constraint on property II

Ontology(<http://nomagic.com/ontology/example-case/case-07>

Import(<http://nomagic.com/ontology/example-case/case-06>)

Declaration(

Class(<http://nomagic.com/ontology/example-case/case-06#Person>)

)

Declaration(

Class(<http://nomagic.com/ontology/example-case/case-06#Pet>)

)

Declaration(

Class(:Dog)

)

Declaration(

Class(:DogLover)

)
Declaration(
       ObjectProperty(<http://nomagic.com/ontology/example-case/case-06#has>)
)
AnnotationAssertion(rdfs:label :Dog "Dog"@en)
SubClassOf(:Dog <http://nomagic.com/ontology/example-case/case-06#Pet>)
AnnotationAssertion(rdfs:label :DogLover "Dog Lover"@en)
SubClassOf(:DogLover <http://nomagic.com/ontology/example-case/case-06#Person>)
SubClassOf(
      :DogLover
          ObjectAllValuesFrom(<http://nomagic.com/ontology/example-case/case-06#has> :Dog)
)
)

4.16 Class with Existential Quantification Constraint on Property
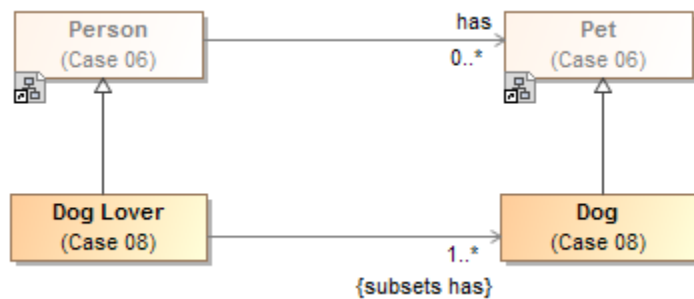
Figure 43   A class with existential quantification constraint on property

Ontology(<http://nomagic.com/ontology/example-case/case-08>
      Import(<http://nomagic.com/ontology/example-case/case-06>)
      Declaration(
          Class(<http://nomagic.com/ontology/example-case/case-06#Person>)
)
      Declaration(
          Class(<http://nomagic.com/ontology/example-case/case-06#Pet>)
)
      Declaration(
          Class(:Dog)
)
      Declaration(
          Class(:DogLover)
)

```
Declaration(
        ObjectProperty(<http://nomagic.com/ontology/example-case/case-06#has>)
)
AnnotationAssertion(rdfs:label :Dog "Dog"@en)
SubClassOf(:Dog <http://nomagic.com/ontology/example-case/case-06#Pet>)
AnnotationAssertion(rdfs:label :DogLover "Dog Lover"@en)
SubClassOf(:DogLover <http://nomagic.com/ontology/example-case/case-06#Person>)
SubClassOf(
        :DogLover
        ObjectIntersectionOf(
                ObjectMinCardinality(1 <http://nomagic.com/ontology/example-
                case/case-06#has> :Dog)
                ObjectSomeValuesFrom(<http://nomagic.com/ontology/example-
                case/case-06#has> :Dog)
        )
    )
)
```

4.17 Property Holder with Self-Referential Subproperty
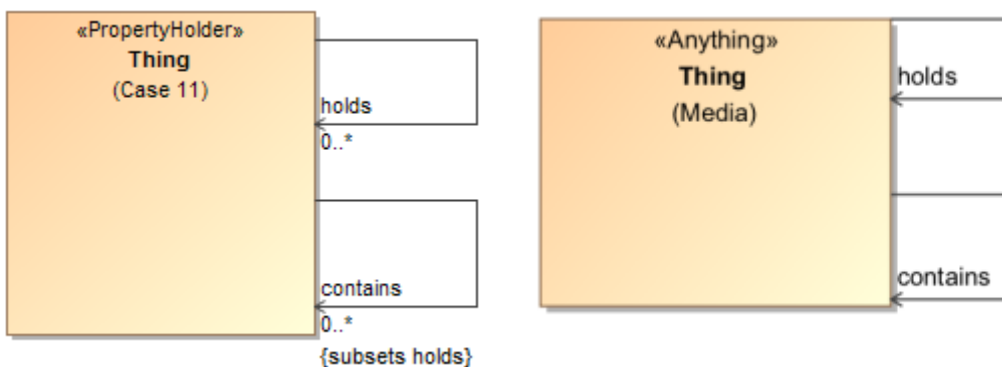
Figure 44   A property holder with self-referential subproperty

```
Ontology(<http://nomagic.com/ontology/example-case/case-11>
        Declaration(
                ObjectProperty(:contains)
        )
        Declaration(
                ObjectProperty(:holds)
        )
        AnnotationAssertion(rdfs:label :contains "contains"@en)
        SubObjectPropertyOf(:contains :holds)
```

AnnotationAssertion(rdfs:label :holds "holds"@en)
)

4.18 Property Holder with Subproperty
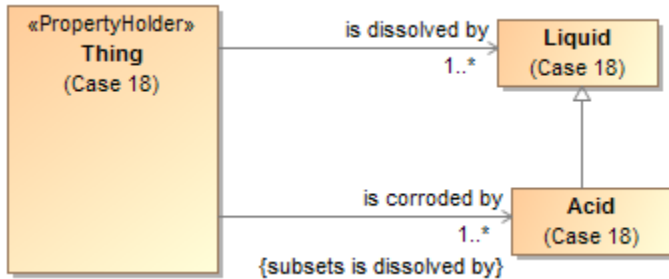
Figure 45   A property with subproperty

Ontology(<http://nomagic.com/ontology/example-case/case-18>
        Declaration(
                Class(:Acid)
)
        Declaration(
                Class(:Liquid)
)
        Declaration(
                ObjectProperty(:isCorrodedBy)
)
        Declaration(
                ObjectProperty(:isDissolvedBy)
)
        AnnotationAssertion(rdfs:label :Acid "Acid"@en)
        SubClassOf(:Acid :Liquid)
        AnnotationAssertion(rdfs:label :Liquid "Liquid"@en)
        SubClassOf(
                owl:Thing
                ObjectIntersectionOf(
                        ObjectMinCardinality(1 :isCorrodedBy :Acid)
)
)
        SubClassOf(
                owl:Thing
                ObjectIntersectionOf(
                        ObjectMinCardinality(1 :isDissolvedBy :Liquid)

```
                )
        )
        AnnotationAssertion(rdfs:label :isCorrodedBy "is corroded by"@en)
        SubObjectPropertyOf(:isCorrodedBy :isDissolvedBy)
        ObjectPropertyRange(:isCorrodedBy :Acid)
        AnnotationAssertion(rdfs:label :isDissolvedBy "is dissolved by"@en)
        ObjectPropertyRange(:isDissolvedBy :Liquid)
)
```

4.19 Class with Subproperty without a Range
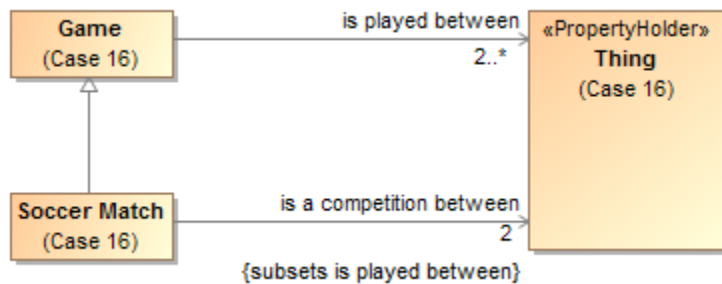
Figure 46   A class with subproperty that has no range

```
Ontology(<http://nomagic.com/ontology/example-case/case-16>
        Declaration(
                Class(:Game)
        )
        Declaration(
                Class(:SoccerMatch)
        )
        Declaration(
                ObjectProperty(:isACompetitionBetween)
        )
        Declaration(
                ObjectProperty(:isPlayedBetween)
        )
        AnnotationAssertion(rdfs:label :Game "Game"@en)
        SubClassOf(
                :Game
                ObjectIntersectionOf(
                        ObjectMinCardinality(2 :isPlayedBetween)
                )
        )
```

Copyright © 2017, No Magic, Inc.

AnnotationAssertion(rdfs:label :SoccerMatch "Soccer Match"@en)

SubClassOf(:SoccerMatch :Game)

SubClassOf(

      :SoccerMatch

      ObjectIntersectionOf(

            ObjectMaxCardinality(2 :isACompetitionBetween)

            ObjectMinCardinality(2 :isACompetitionBetween)

)

)

AnnotationAssertion(rdfs:label :isACompetitionBetween "is a competition between"@en)

SubObjectPropertyOf(:isACompetitionBetween :isPlayedBetween)

ObjectPropertyDomain(:isACompetitionBetween :SoccerMatch)

AnnotationAssertion(rdfs:label :isPlayedBetween "is played between"@en)

ObjectPropertyDomain(:isPlayedBetween :Game)

)

4.20 Class with Necessary and Sufficient Property
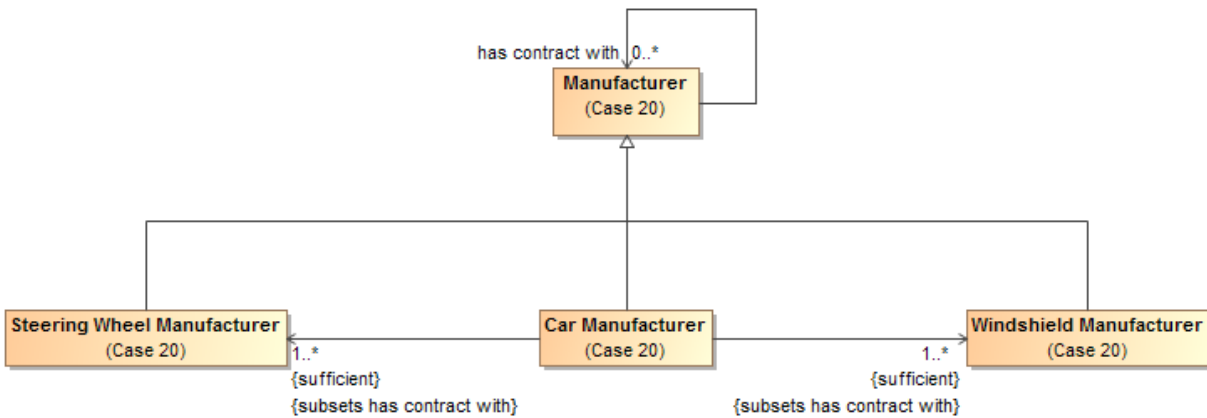
Figure 47   A class with necessary and sufficient property

Ontology(<http://nomagic.com/ontology/example-case/case-20>

      Declaration(

            Class(:CarManufacturer)

)

      Declaration(

            Class(:Manufacturer)

)

      Declaration(

            Class(:SteeringWheelManufacturer)

```
        )
        Declaration(
                Class(:WindshieldManufacturer)
        )
        Declaration(
                ObjectProperty(:hasContractWith)
        )
        AnnotationAssertion(rdfs:label :CarManufacturer "Car Manufacturer"@en)
        EquivalentClasses(
                :CarManufacturer
                ObjectIntersectionOf(
                        ObjectMinCardinality(1 :hasContractWith :SteeringWheelManufacturer)
                        ObjectSomeValuesFrom(:hasContractWith :SteeringWheelManufacturer)
                )
        )
        EquivalentClasses(
                :CarManufacturer
                ObjectIntersectionOf(
                        ObjectMinCardinality(1 :hasContractWith :WindshieldManufacturer)
                        ObjectSomeValuesFrom(:hasContractWith :WindshieldManufacturer)
                )
        )
        SubClassOf(:CarManufacturer :Manufacturer)
        AnnotationAssertion(rdfs:label :Manufacturer "Manufacturer"@en)
        AnnotationAssertion(rdfs:label :SteeringWheelManufacturer "Steering Wheel
        Manufacturer"@en)
        SubClassOf(:SteeringWheelManufacturer :Manufacturer)
        AnnotationAssertion(rdfs:label :WindshieldManufacturer "Windshield
        Manufacturer"@en)
        SubClassOf(:WindshieldManufacturer :Manufacturer)
        AnnotationAssertion(rdfs:label :hasContractWith "has contract with"@en)
        ObjectPropertyDomain(:hasContractWith :Manufacturer)
        ObjectPropertyRange(:hasContractWith :Manufacturer)
)
```

4.21 Class with Property Having Unspecified Multiplicity

UML allows the cardinality of a property to be left unspecified. The concept modeling profile interprets unspecified cardinalities as being zero to many ("0..*").
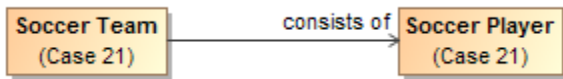
Figure 48   A class with property whose multiplicity is unspecified

Ontology(<http://nomagic.com/ontology/example-case/case-21>
      Declaration(
            Class(:SoccerPlayer)
)
      Declaration(
            Class(:SoccerTeam)
)
      Declaration(ObjectProperty(:consistsOf))
      AnnotationAssertion(rdfs:label :SoccerPlayer "Soccer Player"@en)
      AnnotationAssertion(rdfs:label :SoccerTeam "Soccer Team"@en)
      AnnotationAssertion(rdfs:label :consistsOf "consists of"@en)
      ObjectPropertyDomain(:consistsOf :SoccerTeam)
      ObjectPropertyRange(:consistsOf :SoccerPlayer)
)

4.22 Class with Inverse Property
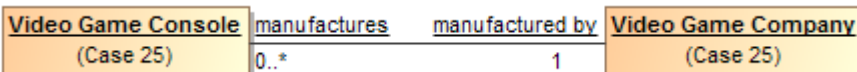
Figure 49   A class with inverse property

Ontology(<http://nomagic.com/ontology/example-case/case-24>
      Declaration(
            ObjectProperty(:manufacturedBy)
)
      ObjectPropertyDomain(:manufacturedBy :VideoGameConsole)
      ObjectPropertyRange(:manufacturedBy :VideoGameCompany)
      Declaration(
            ObjectProperty(:manufactures)
)
      InverseObjectProperties(:manufacturedBy :manufactures)
      ObjectPropertyDomain(:manufactures :VideoGameCompany)
      ObjectPropertyRange(:manufactures :VideoGameConsole)
      Declaration(

```
            Class(:VideoGameCompany)
    )
    Declaration(
            Class(:VideoGameConsole)
    )
    SubClassOf(
            :VideoGameConsole
            ObjectIntersectionOf(
                    ObjectMaxCardinality(1 :manufacturedBy :VideoGameCompany)
                    ObjectMinCardinality(1 :manufacturedBy :VideoGameCompany)
            )
    )
    AnnotationAssertion(rdfs:label :VideoGameCompany "Video Game Company"@en)
    AnnotationAssertion(rdfs:label :VideoGameConsole "Video Game Console"@en)
    AnnotationAssertion(rdfs:label :manufacturedBy "manufactured by"@en)
    AnnotationAssertion(rdfs:label :manufactures "manufactures"@en)
)
```

4.23 Annotation and Annotation Property
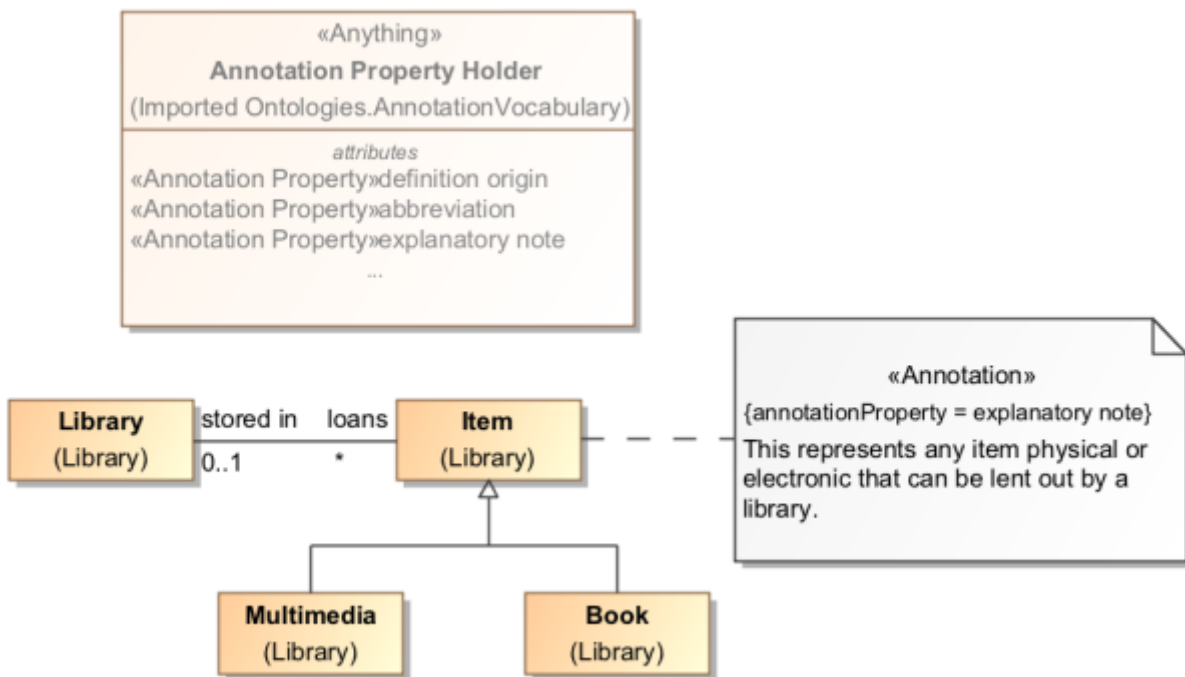
Figure 50   A class with annotation and annotation property

Ontology(<http://nomagic.com/ontology/example-case/case-25>

        Declaration(Class(:Book))
        Declaration(Class(:Item))
        Declaration(Class(:Multimedia))
        Declaration(
                AnnotationProperty(<http://spec.edmcouncil.org/fibo/FND/Utilities/AnnotationV
                ocabulary/explanatoryNote>))
        AnnotationAssertion(rdfs:label :Book "Book"@en)
        SubClassOf(:Book :Item)
        AnnotationAssertion(rdfs:label :Item "Item"@en)
        AnnotationAssertion(<http://spec.edmcouncil.org/fibo/FND/Utilities/AnnotationVocabul
        ary/explanatoryNote> :Item "This represents any item physical or electronic that can be
        lent out by a library."@en)
        AnnotationAssertion(rdfs:label :Multimedia "Multimedia"@en)
        SubClassOf(:Multimedia :Item)
)

4.24 Asymmetrical Inverse Property
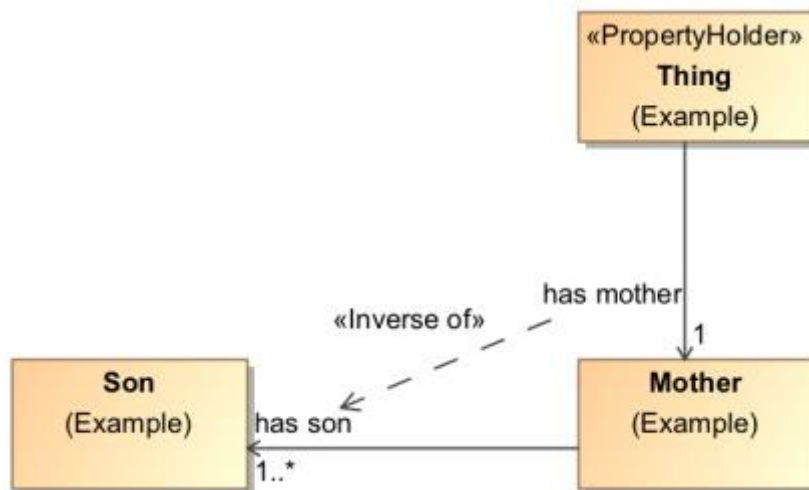
Figure 51   Asymmetrical Inverse Property

Ontology(<http://example.com/ontology/AsymmetricalInverseProperty>

        Declaration(Class(:Mother))
        Declaration(Class(:Son))
        Declaration(ObjectProperty(:hasMother))
        Declaration(ObjectProperty(:hasSon))

AnnotationAssertion(rdfs:label :Mother "Mother"@en)

SubClassOf(:Mother ObjectIntersectionOf(ObjectMinCardinality(1 :hasSon :Son)))

AnnotationAssertion(rdfs:label :Son "Son"@en)

SubClassOf(owl:Thing ObjectIntersectionOf(ObjectMaxCardinality(1 :hasMother :Mother) ObjectMinCardinality(1 :hasMother :Mother)))

AnnotationAssertion(rdfs:label :hasMother "has mother"@en)

ObjectPropertyRange(:hasMother :Mother)

AnnotationAssertion(rdfs:label :hasSon "has son"@en)

ObjectPropertyDomain(:hasSon :Mother)

ObjectPropertyRange(:hasSon :Son)

)

4.25 Disjoint Classes
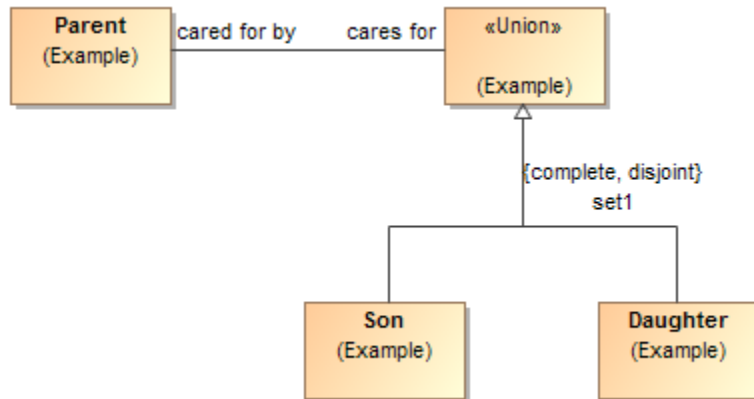
Figure 52   Disjoint Dependency

Ontology(<http://www.example.com/ontology/Disjoint>

    Declaration(Class(:Daughter))
    Declaration(Class(:Parent))
    Declaration(Class(:Son))
    Declaration(ObjectProperty(:caredForBy))
    Declaration(ObjectProperty(:caresFor))
    AnnotationAssertion(rdfs:label :Daughter "Daughter"@en)
    DisjointClasses(:Daughter :Son)
    AnnotationAssertion(rdfs:label :Parent "Parent"@en)
    AnnotationAssertion(rdfs:label :Son "Son"@en)
    AnnotationAssertion(rdfs:label :caredForBy "cared for by"@en)
    InverseObjectProperties(:caredForBy :caresFor)
    ObjectPropertyDomain(:caredForBy ObjectUnionOf(:Son :Daughter))
    ObjectPropertyRange(:caredForBy :Parent)
    AnnotationAssertion(rdfs:label :caresFor "cares for"@en)

ObjectPropertyDomain(:caresFor :Parent)
ObjectPropertyRange(:caresFor ObjectUnionOf(:Son :Daughter))

)

4.26 Property Chain
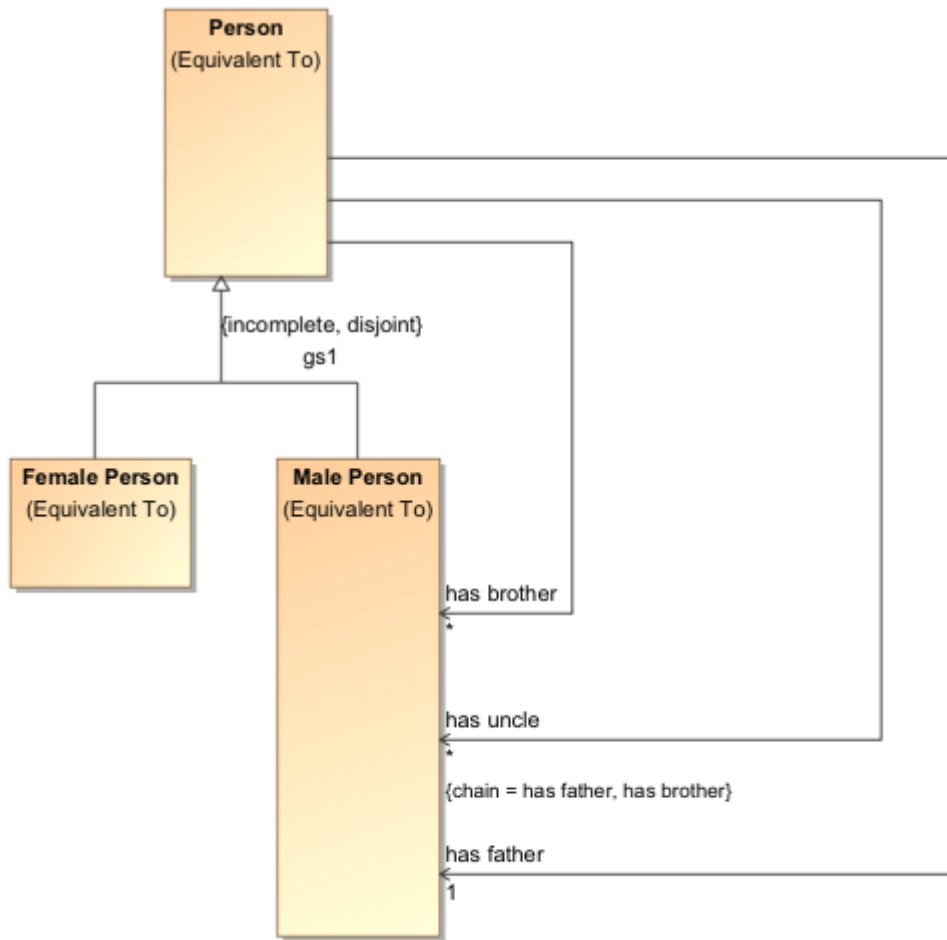
Figure 53   Properties in a property chain

Ontology(<http://example.com/ontology/Unnamed>

    Declaration(Class(:FemalePerson))
    Declaration(Class(:MalePerson))
    Declaration(Class(:Person))
    Declaration(ObjectProperty(:hasBrother))
    Declaration(ObjectProperty(:hasFather))
    Declaration(ObjectProperty(:hasUncle))
    AnnotationAssertion(rdfs:label :FemalePerson "Female Person"@en)

SubClassOf(:FemalePerson :Person)

DisjointClasses(:FemalePerson :MalePerson)

AnnotationAssertion(rdfs:label :MalePerson "Male Person"@en)

SubClassOf(:MalePerson :Person)

AnnotationAssertion(rdfs:label :Person "Person"@en)

SubClassOf(:Person ObjectIntersectionOf(ObjectMaxCardinality(1 :hasFather :MalePerson) ObjectMinCardinality(1 :hasFather :MalePerson)))

AnnotationAssertion(rdfs:label :hasBrother "has brother"@en)

ObjectPropertyDomain(:hasBrother :Person)

ObjectPropertyRange(:hasBrother :MalePerson)

AnnotationAssertion(rdfs:label :hasFather "has father"@en)

ObjectPropertyDomain(:hasFather :Person)

ObjectPropertyRange(:hasFather :MalePerson)

AnnotationAssertion(rdfs:label :hasUncle "has uncle"@en)

ObjectPropertyDomain(:hasUncle :Person)

ObjectPropertyRange(:hasUncle :MalePerson)

SubObjectPropertyOf(ObjectPropertyChain(:hasFather :hasBrother) :hasUncle)

)

4.27 Equivalent Property
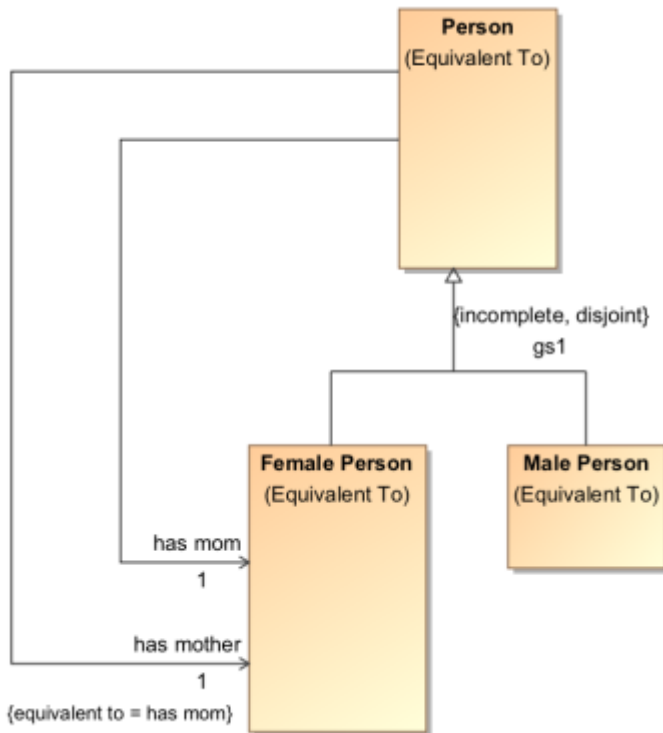
Figure 54   Equivalent properties

Ontology(<http://example.com/ontology/Unnamed>

    Declaration(Class(:FemalePerson))
    Declaration(Class(:MalePerson))
    Declaration(Class(:Person))
    Declaration(ObjectProperty(:hasMom))
    Declaration(ObjectProperty(:hasMother))
    AnnotationAssertion(rdfs:label :FemalePerson "Female Person"@en)
    SubClassOf(:FemalePerson :Person)
    DisjointClasses(:FemalePerson :MalePerson)
    AnnotationAssertion(rdfs:label :MalePerson "Male Person"@en)
    SubClassOf(:MalePerson :Person)
    AnnotationAssertion(rdfs:label :Person "Person"@en)
    SubClassOf(:Person ObjectIntersectionOf(ObjectMaxCardinality(1 :hasMom :FemalePerson) ObjectMinCardinality(1 :hasMom :FemalePerson)))
    SubClassOf(:Person ObjectIntersectionOf(ObjectMaxCardinality(1 :hasMother :FemalePerson) ObjectMinCardinality(1 :hasMother :FemalePerson)))
    AnnotationAssertion(rdfs:label :hasMom "has mom"@en)
    EquivalentObjectProperties(:hasMom :hasMother)
    ObjectPropertyDomain(:hasMom :Person)
    ObjectPropertyRange(:hasMom :FemalePerson)
    AnnotationAssertion(rdfs:label :hasMother "has mother"@en)
    ObjectPropertyDomain(:hasMother :Person)
    ObjectPropertyRange(:hasMother :FemalePerson)
)

4.28 Equivalent Class

Figure 55   Equivalent classes

Ontology(<http://example.com/ontology/Unnamed>

    Declaration(Class(:PrincipleResidentOfWhiteHouse))
    Declaration(Class(:US-President))
    AnnotationAssertion(rdfs:label :PrincipleResidentOfWhiteHouse "Principle Resident of White House"@en)

EquivalentClasses(:PrincipleResidentOfWhiteHouse :US-President)
AnnotationAssertion(rdfs:label :US-President "US President"@en)
)

5 Usage

5.1 Create a Concept Modeling Project

To create a concept modeling project:
1. Click **File** > **New Project**. The **New Project** dialog will open.
2. Select **Concept Modeling Project**.
3. Name your project and select your **Project location**.
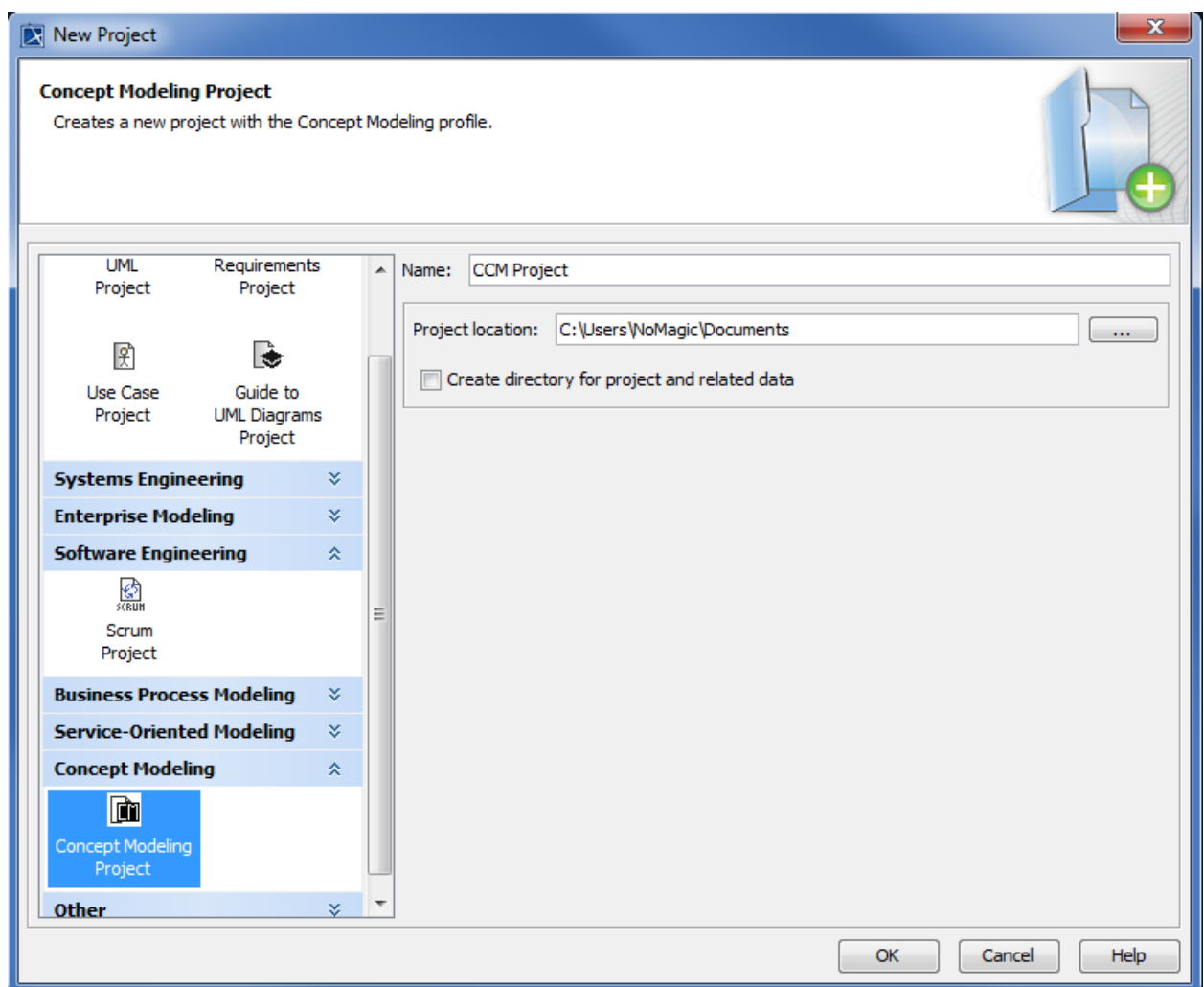
Figure 56   Selecting the Concept Modeling profile

4. Click **OK**. A new Concept Modeling diagram will open, complete with the Concept Modeling diagram palette. This diagram and its palette will also open whenever you create a new Concept Modeling diagram.
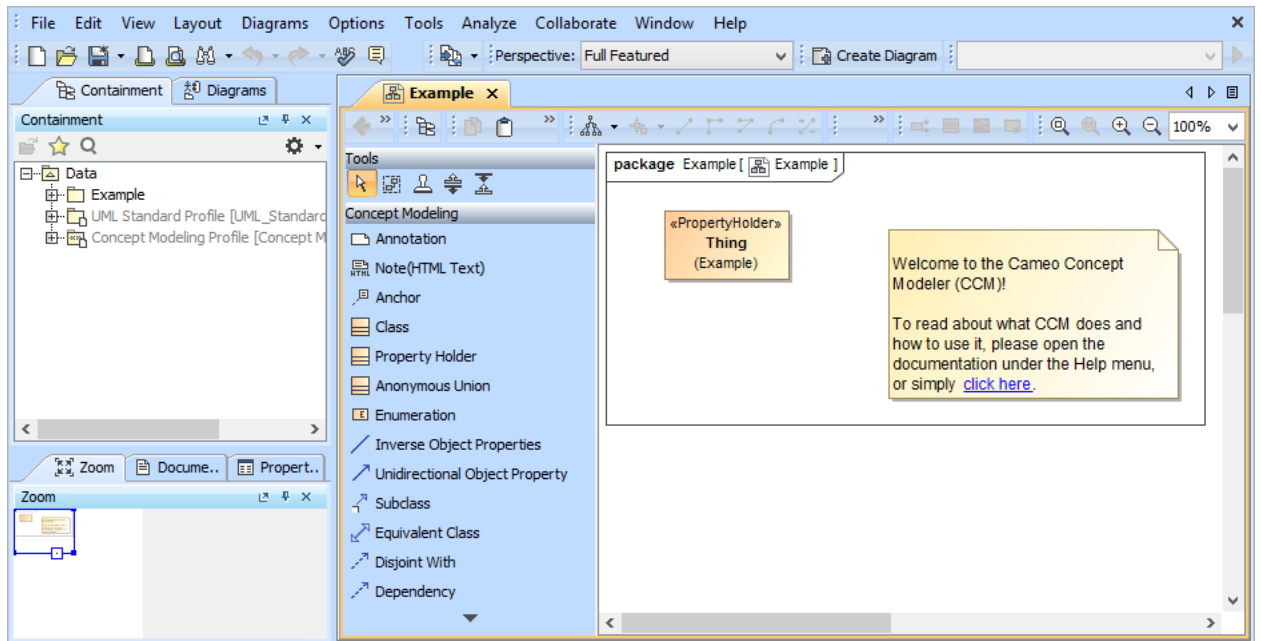
Figure 57   The Concept Modeling diagram and its palette

The following table shows the buttons in the Concept Modeling diagram palette, which represent the elements you use to create a Concept Modeling diagram. You can drag the button to a diagram to create that kind of element. The shortcut key may also make it easier for you to create a specific element.

Buttons	Shortcut Keys
Annotation	A
Note(HTML Text)	Shift + N
Anchor	H
Class	C
Property Holder	P
Anonymous Union	Shift + U
Enumeration	K
Inverse Object Properties	S

↗ Unidirectional Object Property	U
↗ Subclass	G
↗ Equivalent Class	Shift + G
↗ Disjoint With	D
↗ Dependency	Shift + D

If you use either Inverse Object Properties or Unidirectional Object Property, the following things will be created:

(i) When a property's type does not have a name, "unnamed property" will be used as the property's name.
(ii) When a property's type has a name, the name will be written in lower-case letters and prepended with "has " (with a space after). For example, if the property's type name is "Boss Deck", it will be converted to "has boss deck".

5.2   Create a Concept Model

To create a concept model:
1. Right-click a package in the Containment tree.
2. Select **Concept Modeling**.
3. Select **Create Concept Model**.

Copyright © 2017, No Magic, Inc.
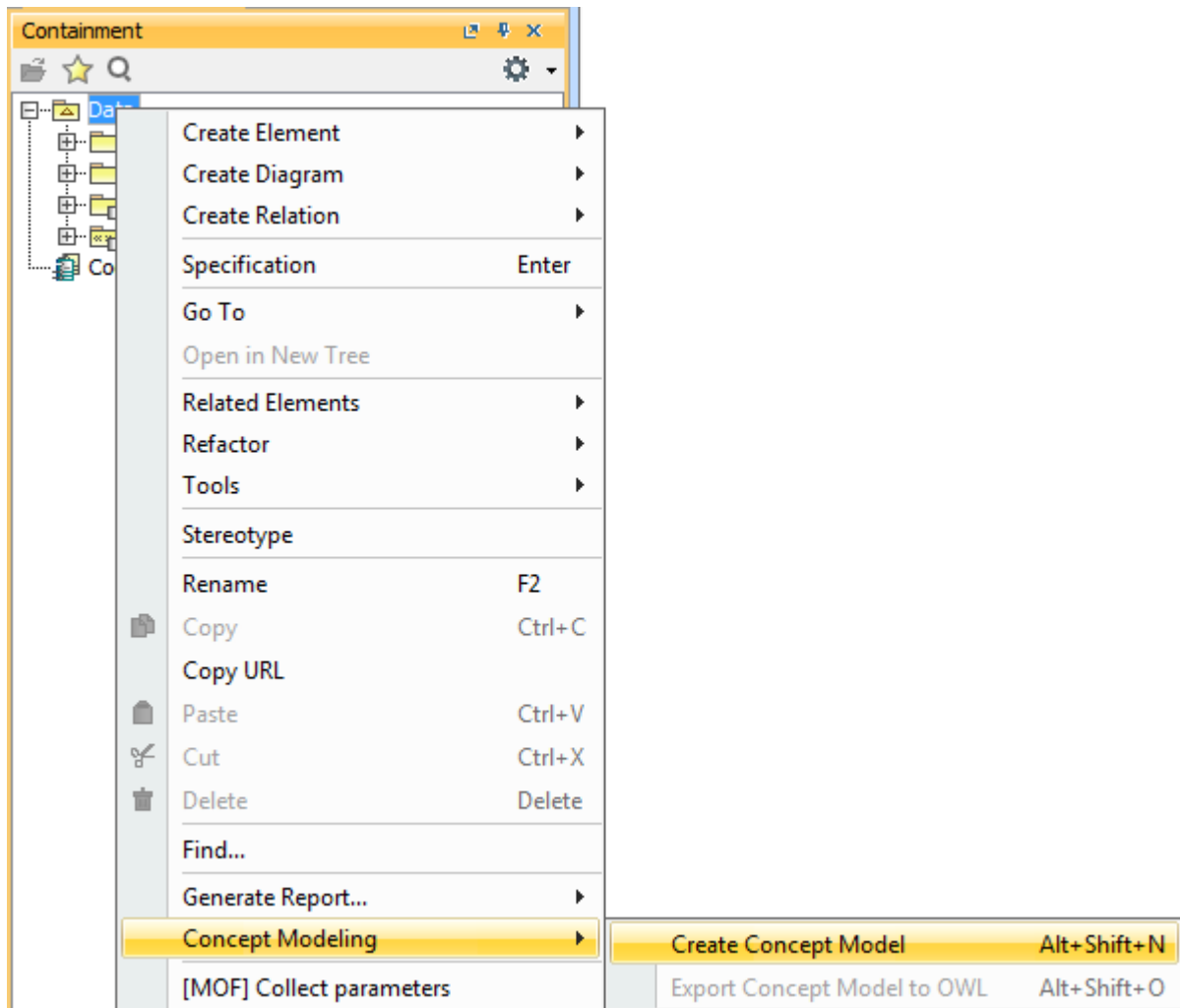
Figure 58   The Create Concept Model shortcut menu

Note	• If an Unnamed package already existed in the Containment tree, a number in the package name will be added or incremented.

5.2.1   Convert a UML Model into a Concept Model

To change a UML model to a concept model:

1. Open an existing UML project.
2. On the main menu, click **File** > **Use Project** > **Use Local Project**. The **Use Project** dialog will open.

Copyright © 2017, No Magic, Inc.
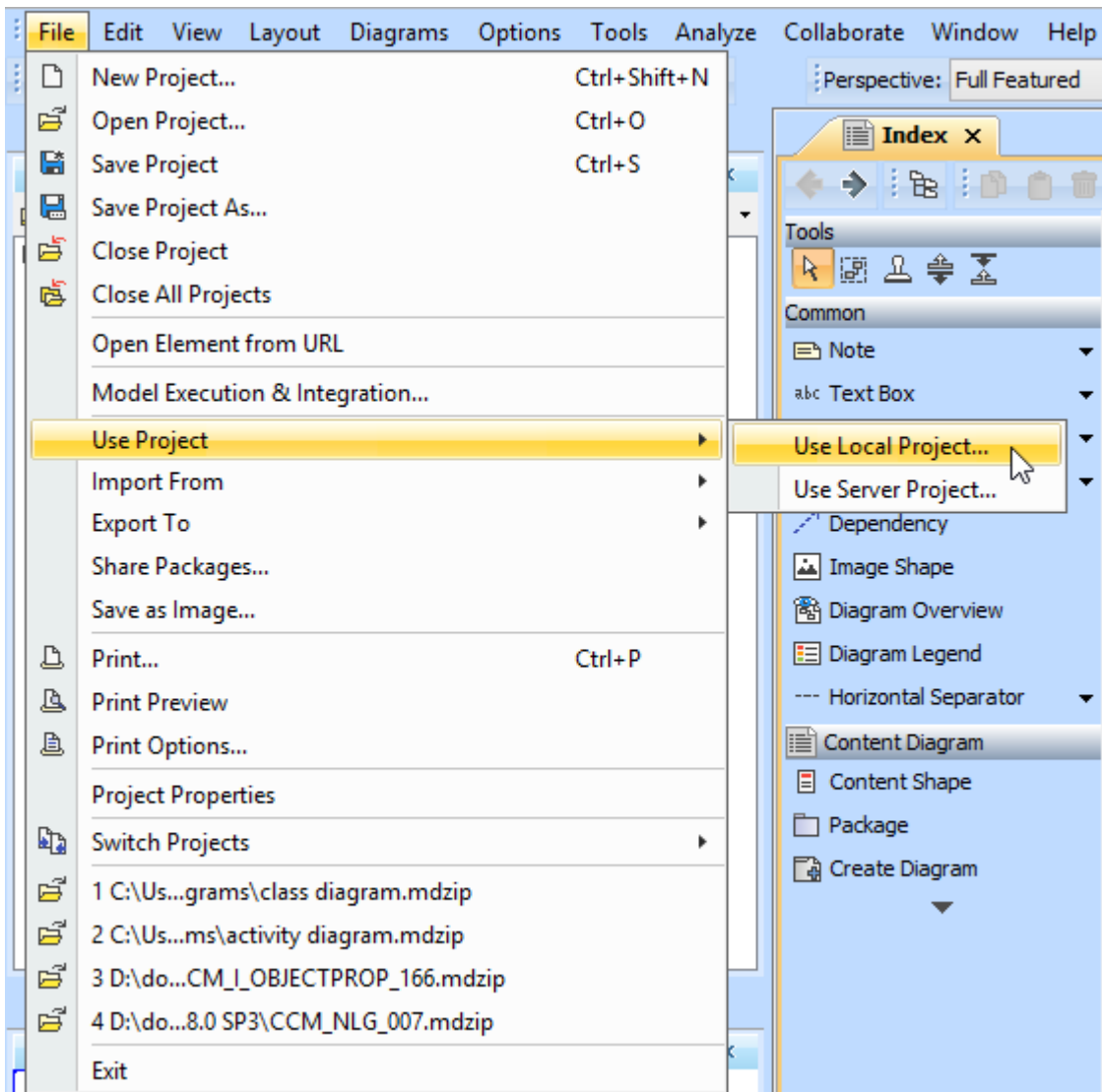
Figure 59   The Use Local Project Menu

3. Select **Profile** and **Concept Modeling Profile**.
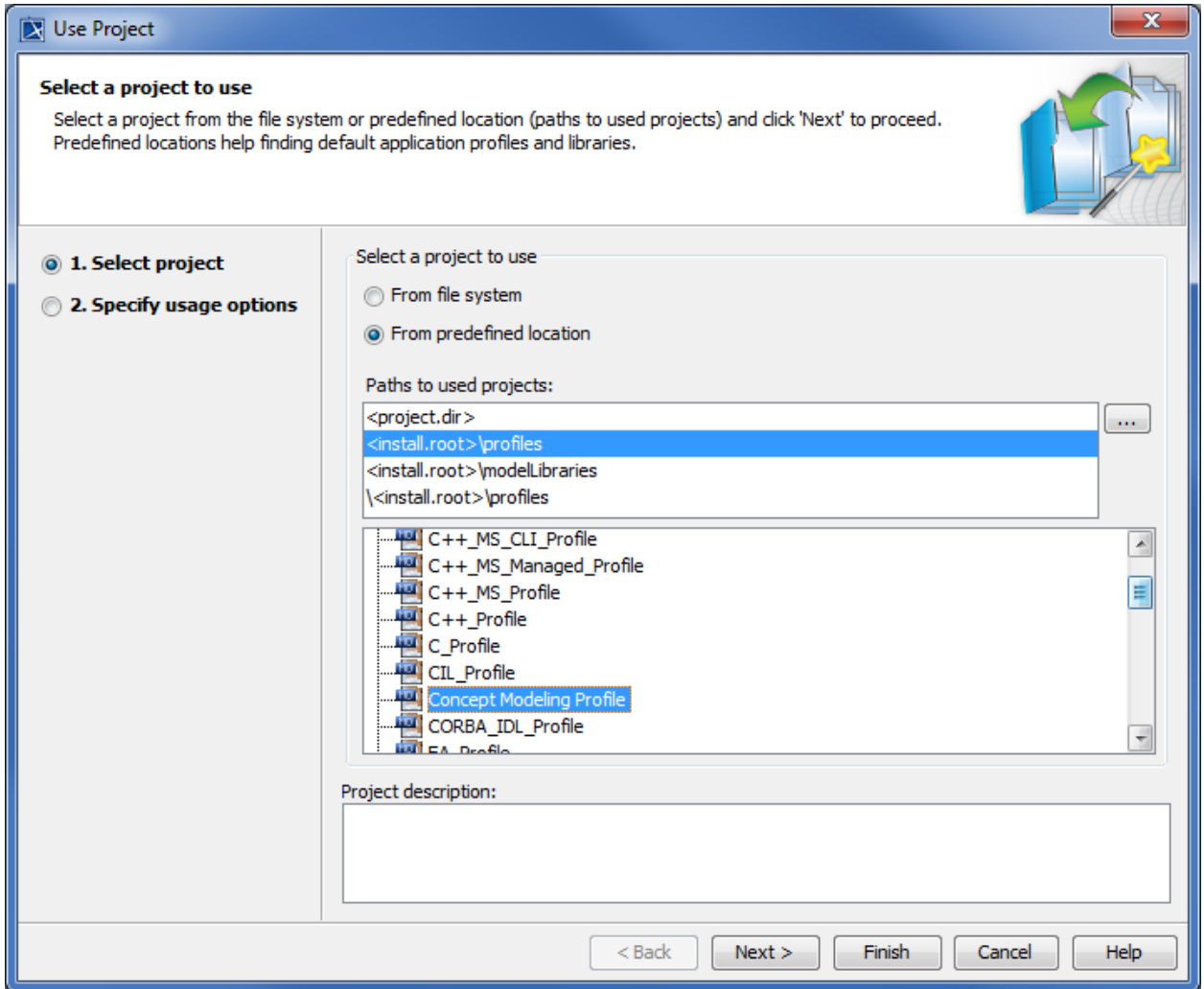
Figure 60   Selecting the Concept Modeling Profile

4. Click **Next**.

Copyright © 2017, No Magic, Inc.
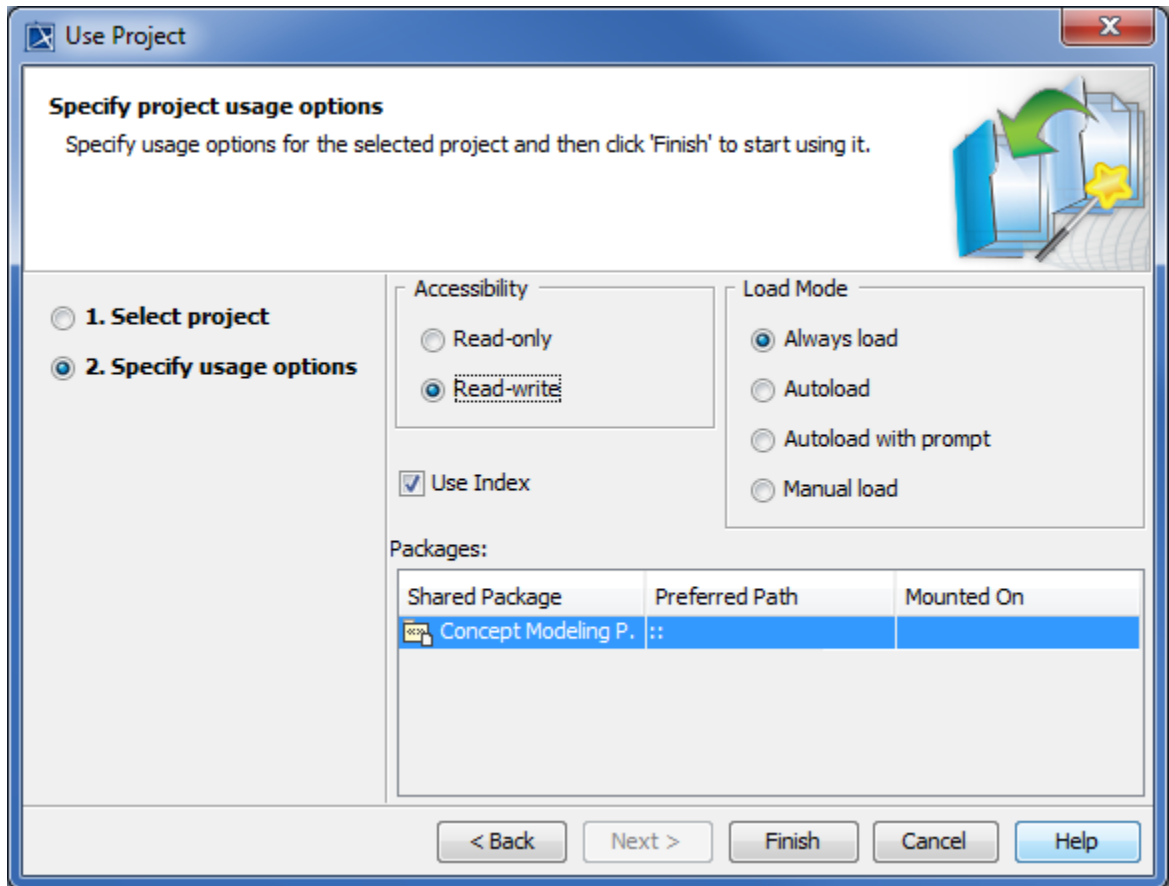
Figure 61   Using the selected Concept Modeling Profile

5. Select usage options and click **Finish**. You will see the Concept Modeling Profile is added to your project in the Containment tree.
6. Create a package in your project.
7. Right-click it and select **Stereotype**.
8. Select the stereotype « » **Concept Model [Package]** and click **Apply**.
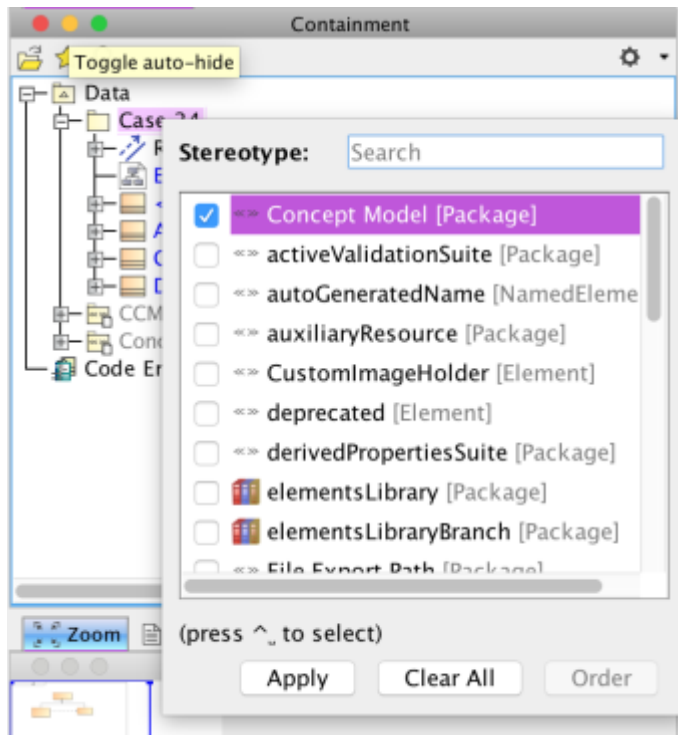
Figure 62   Applying the «　» Concept Model [Package] to the model

9. Open a new concept modeling project (**File** > **Open Project**).
10. Click **Options** > **Project** to open the **Project Options** dialog.
11. Click **Symbol styles** from the tree view, select, and export the **Default** and **Defined Elsewhere** styles to the UML project.
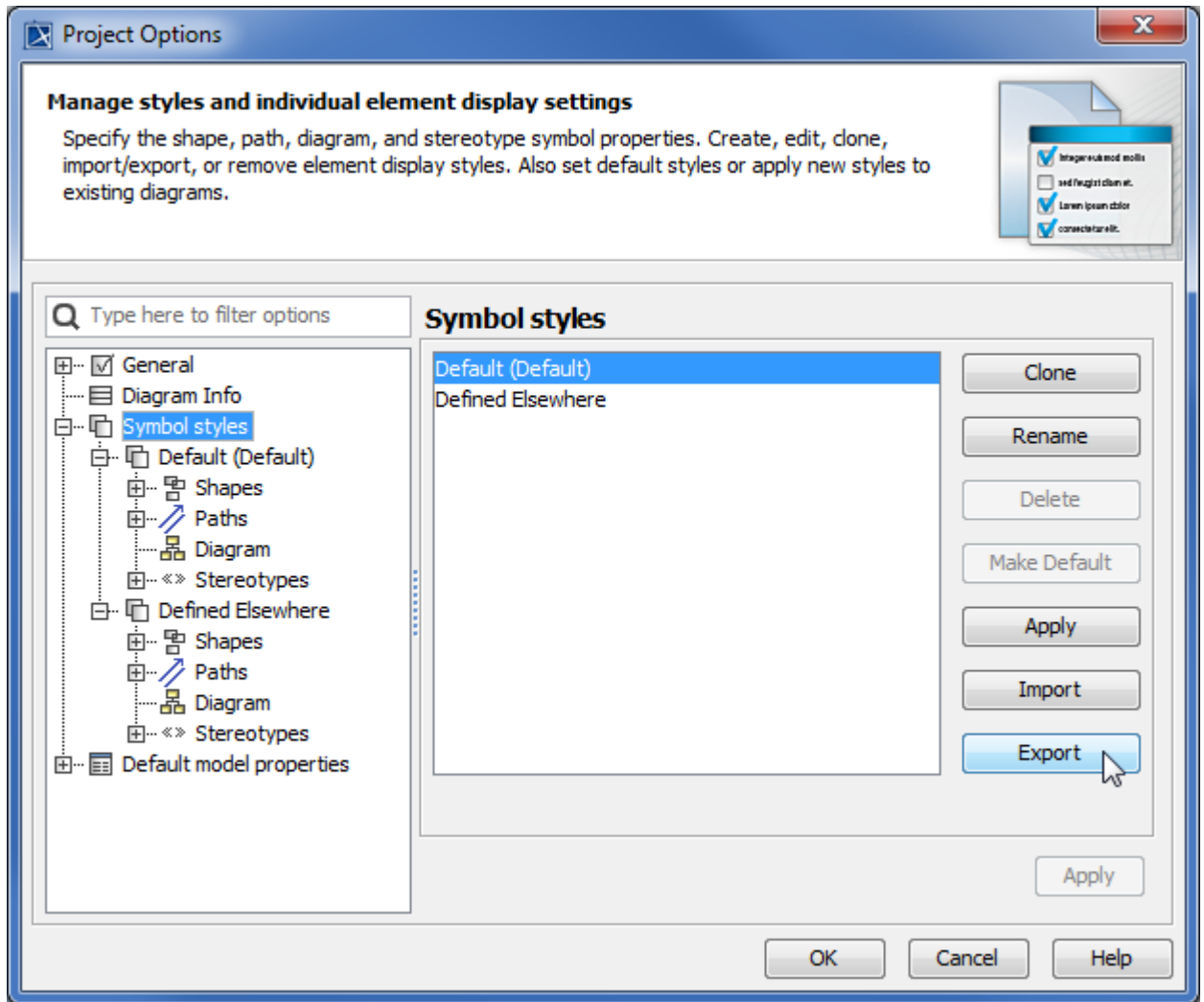
Figure 63   Exporting a project's symbol styles to another project

12. Switch to the previous UML project and click **Options** > **Project** to open the **Project Options** dialog and import the styles to the project.
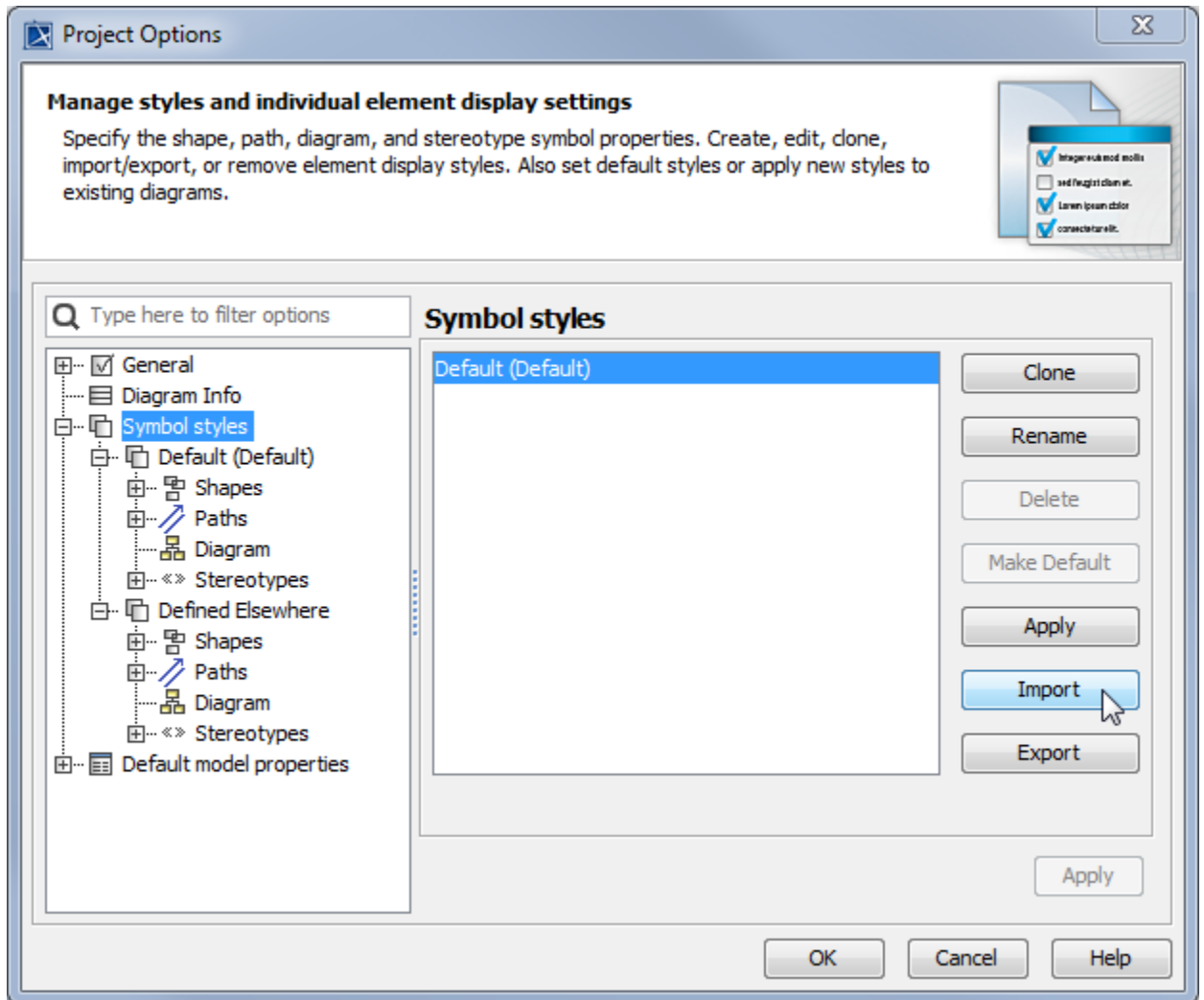
Figure 64   Importing another project's symbol styles into the current project

13. Select **Symbol styles** and click **Import**.
14. Select the exported Default and Define Elsewhere styles and click **OK**. You will see the imported styles added to the **Symbol styles** under the Default (Default) style.
15. Select the imported Default style and click **Make Default**. The imported Default is now the default style.
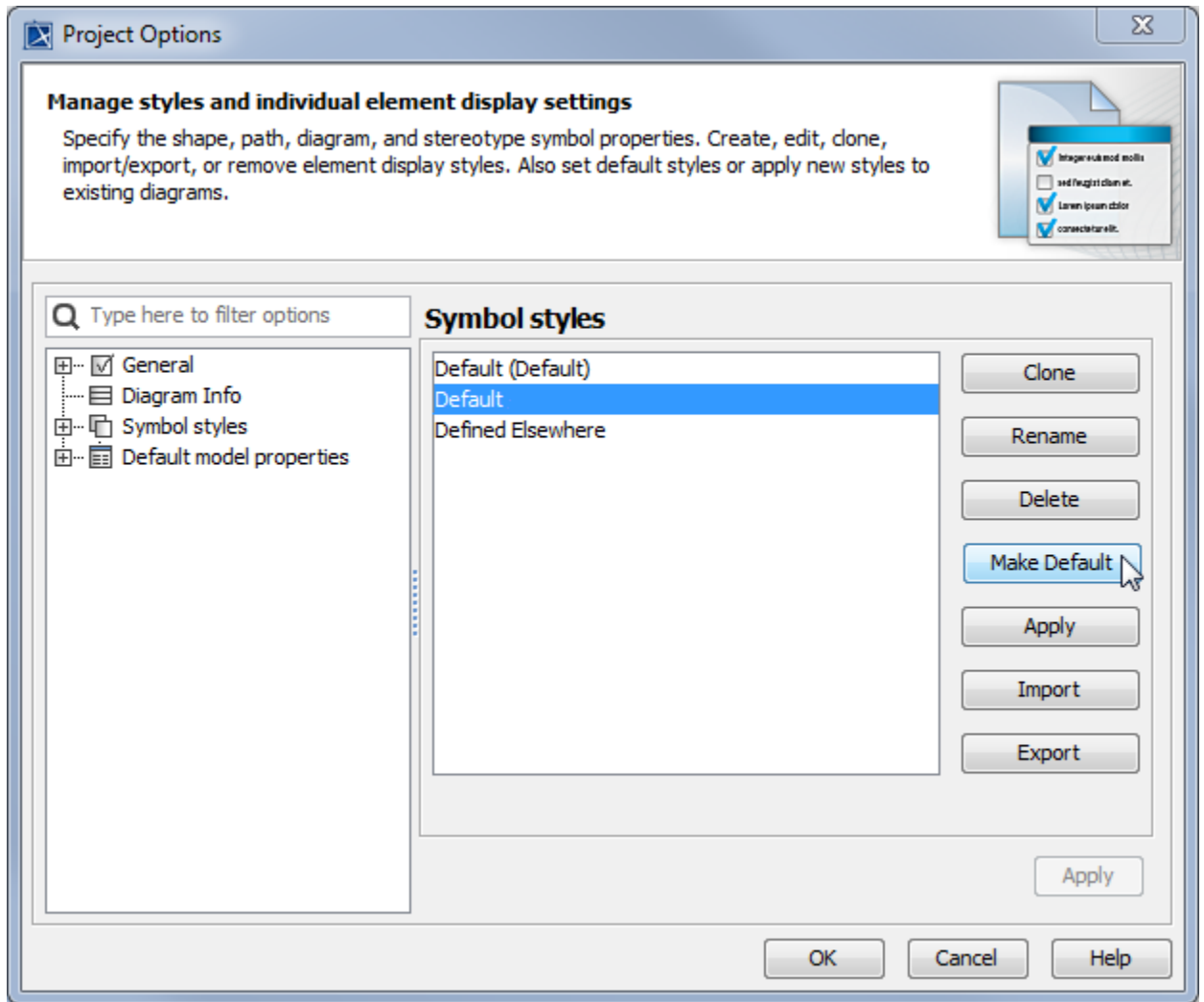
Figure 65   Making the imported symbol style as the default one

16. Select the old **Default** style and click **Delete**.
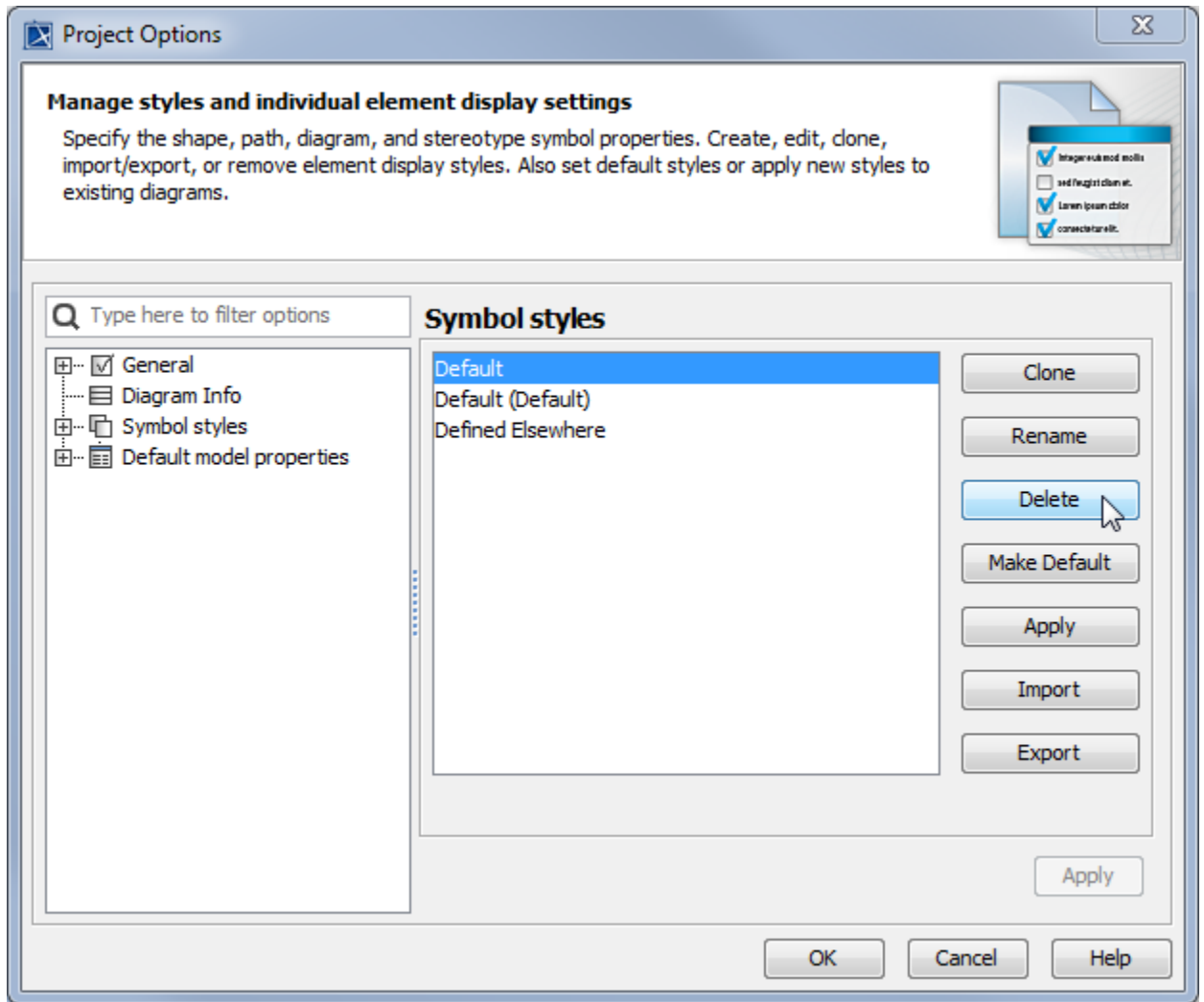
Figure 66   Deleting the old symbol style

17. Click **OK**.
18. Click **Options** > **Project**. The **Project Options** dialog will open.
19. Expand the **Default model properties** and select **Association**.
20. Change the **Association** visibility to **public**.
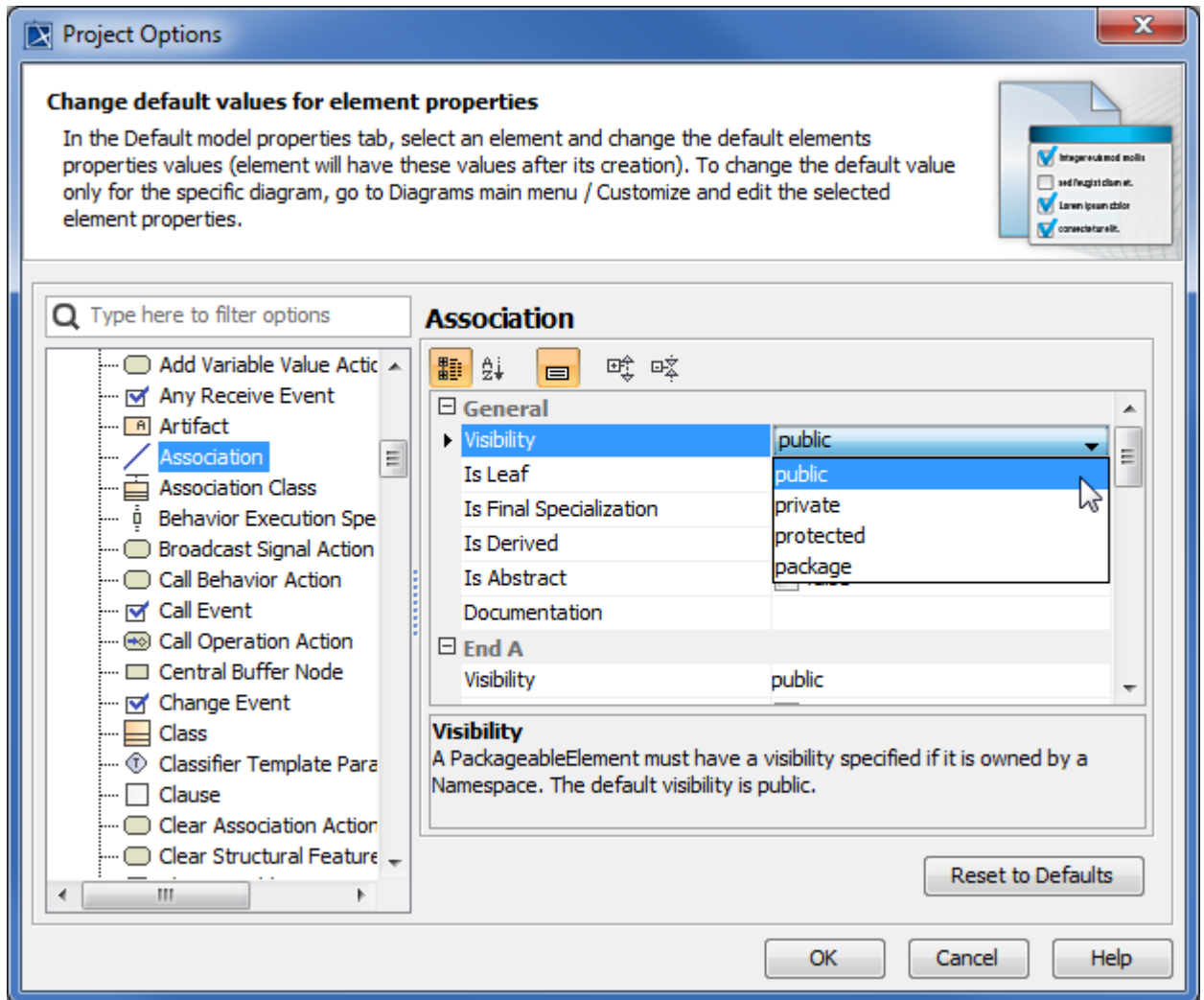
Figure 67   Making the Association's default visibility public

21. Change the **Property** visibility to **public**.
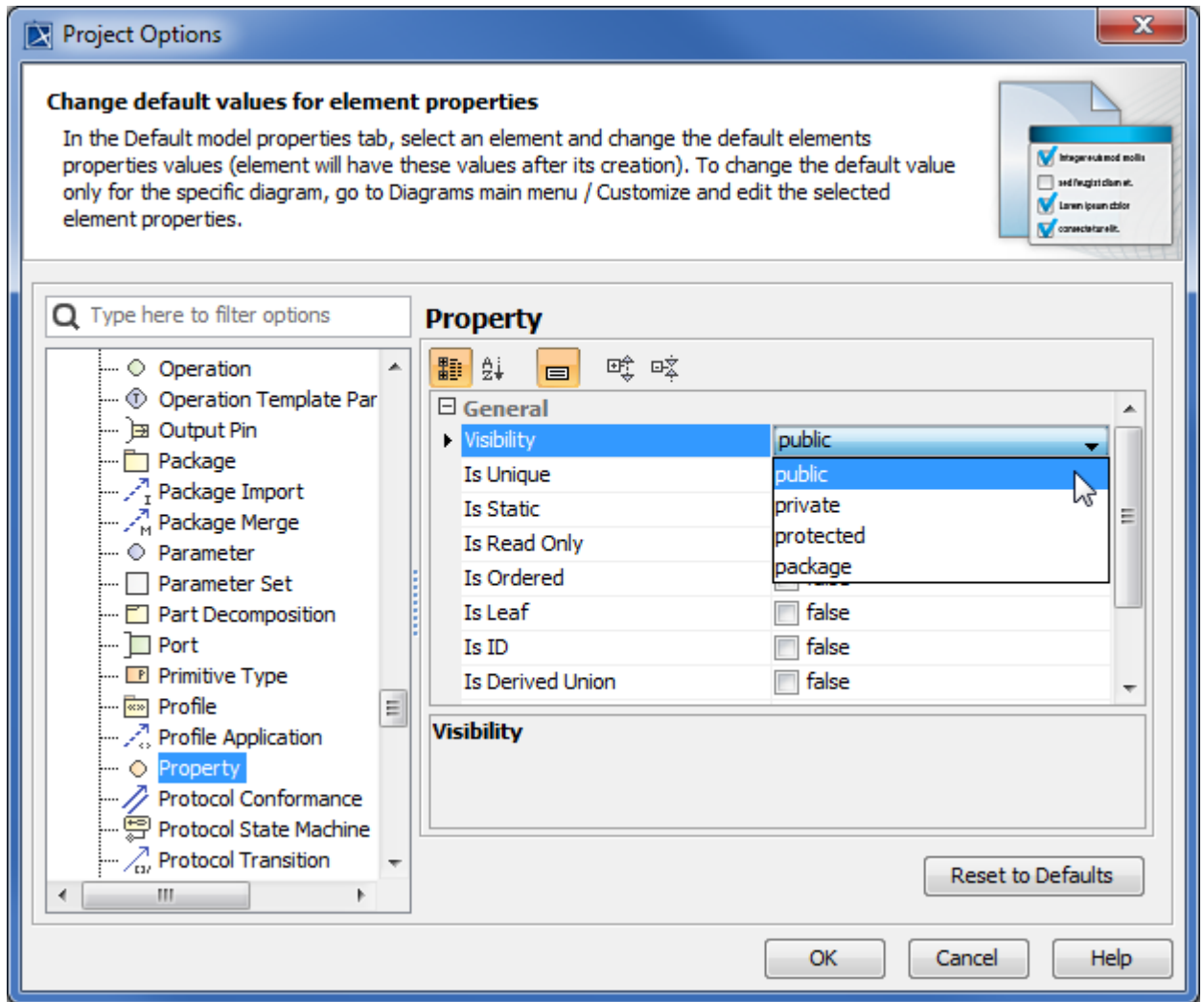
Figure 68   Making the property's default visibility public

22. Click **OK**.

5.2.2   Create a Property Chain

The Concept Modeler allows you to create a property chain by dragging properties, one after another, to a target property. The drag-and-drop action provides two menu options:

    (i)   Create subproperty chain
    (ii)   Add property to subproperty chain

To create a property chain:

1. Drag a property to be composed in the chain, for example, "has father", to a target property, for example, "has uncle", on the diagram pane. A shortcut menu will open.
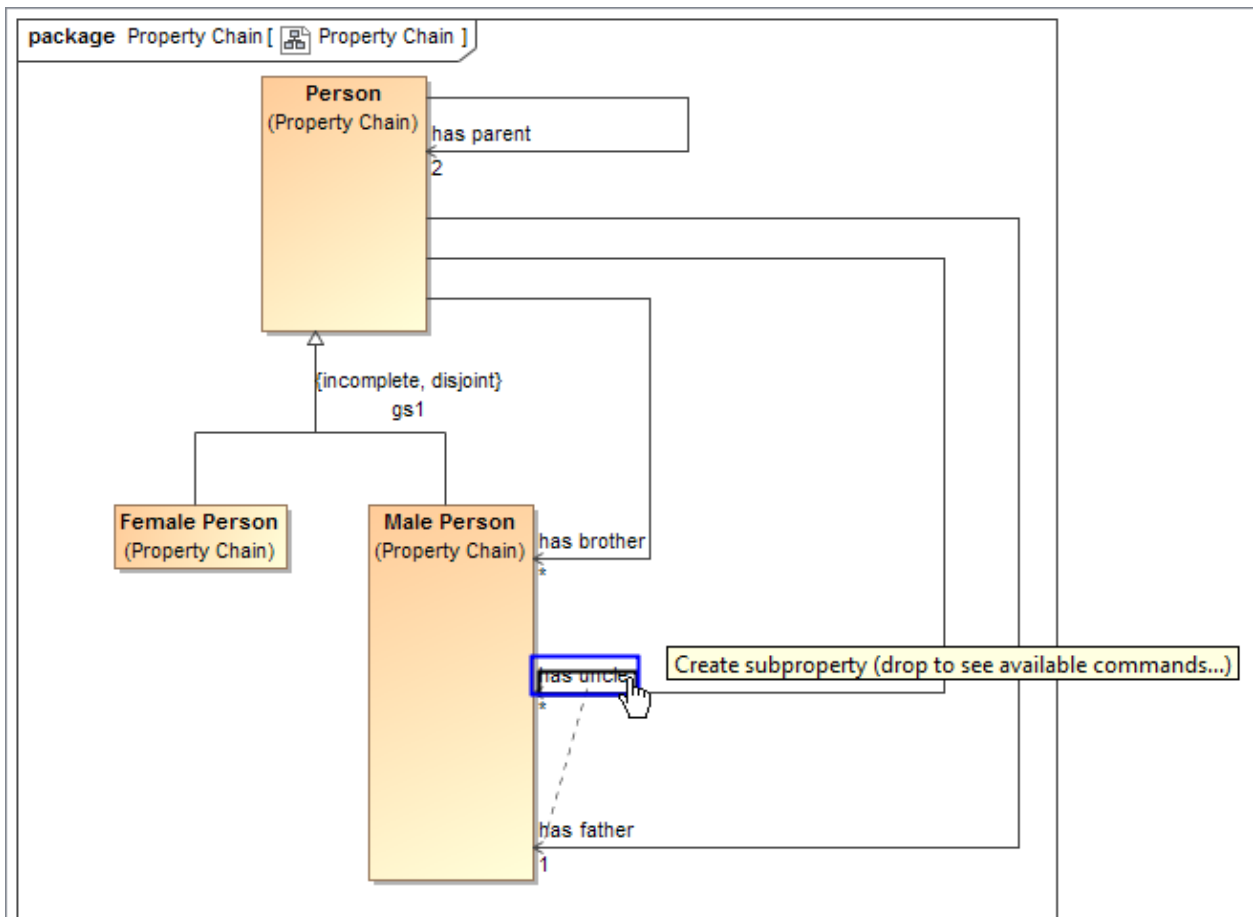
Figure 69 Dragging a property to a target property to create a property chain

Tip	Alternatively, you can create a property chain by right-clicking a target property and select **Create subproperty chain** from the shortcut menu, and select the properties to be composed in the chain from the tree in the **Select Property** dialog.

2. Select **Create subproperty chain** from the shortcut menu.
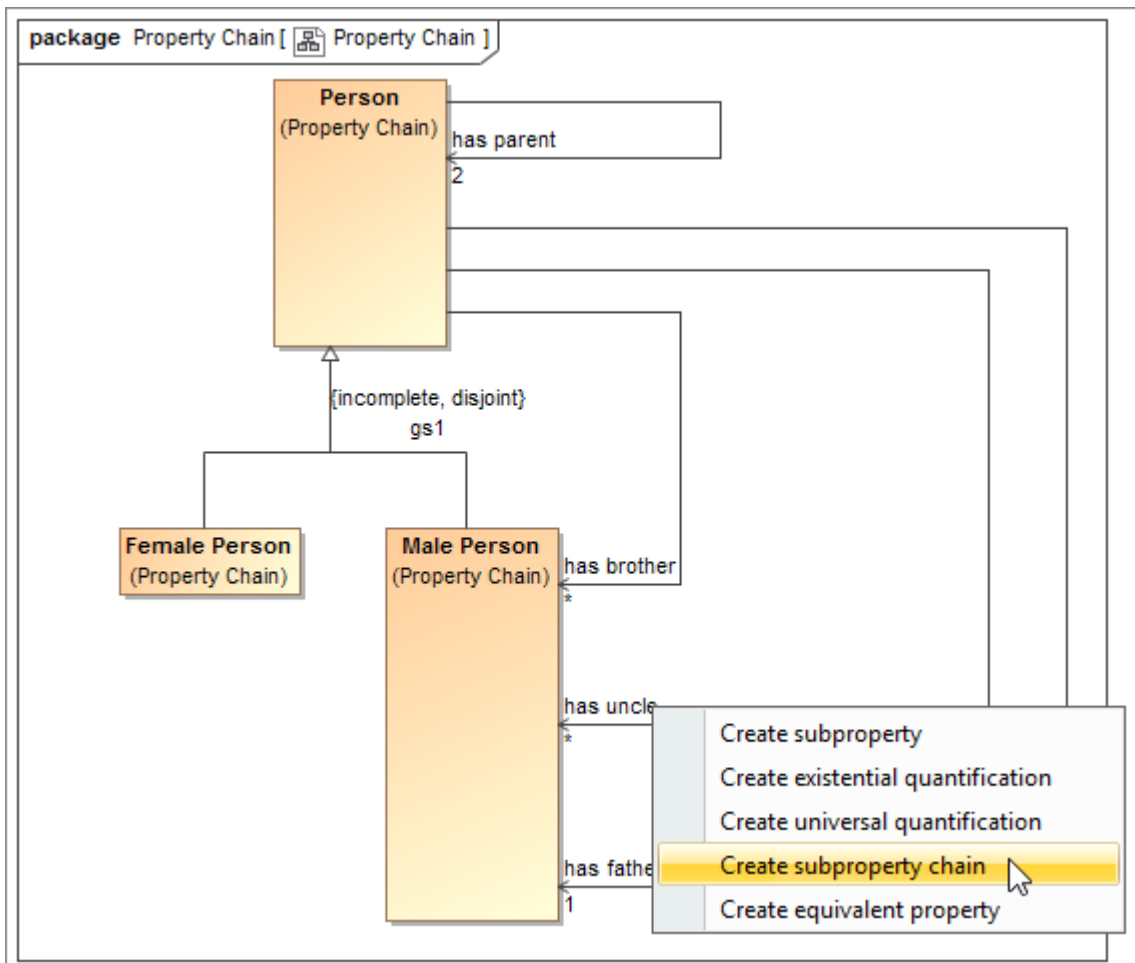
Figure 70   The Create subproperty chain shortcut menu

3. Drag the next property to be composed in the chain, for example, "has brother", to the target property and select **Add property to subproperty chain**. The property chain will be created (Figure 72).
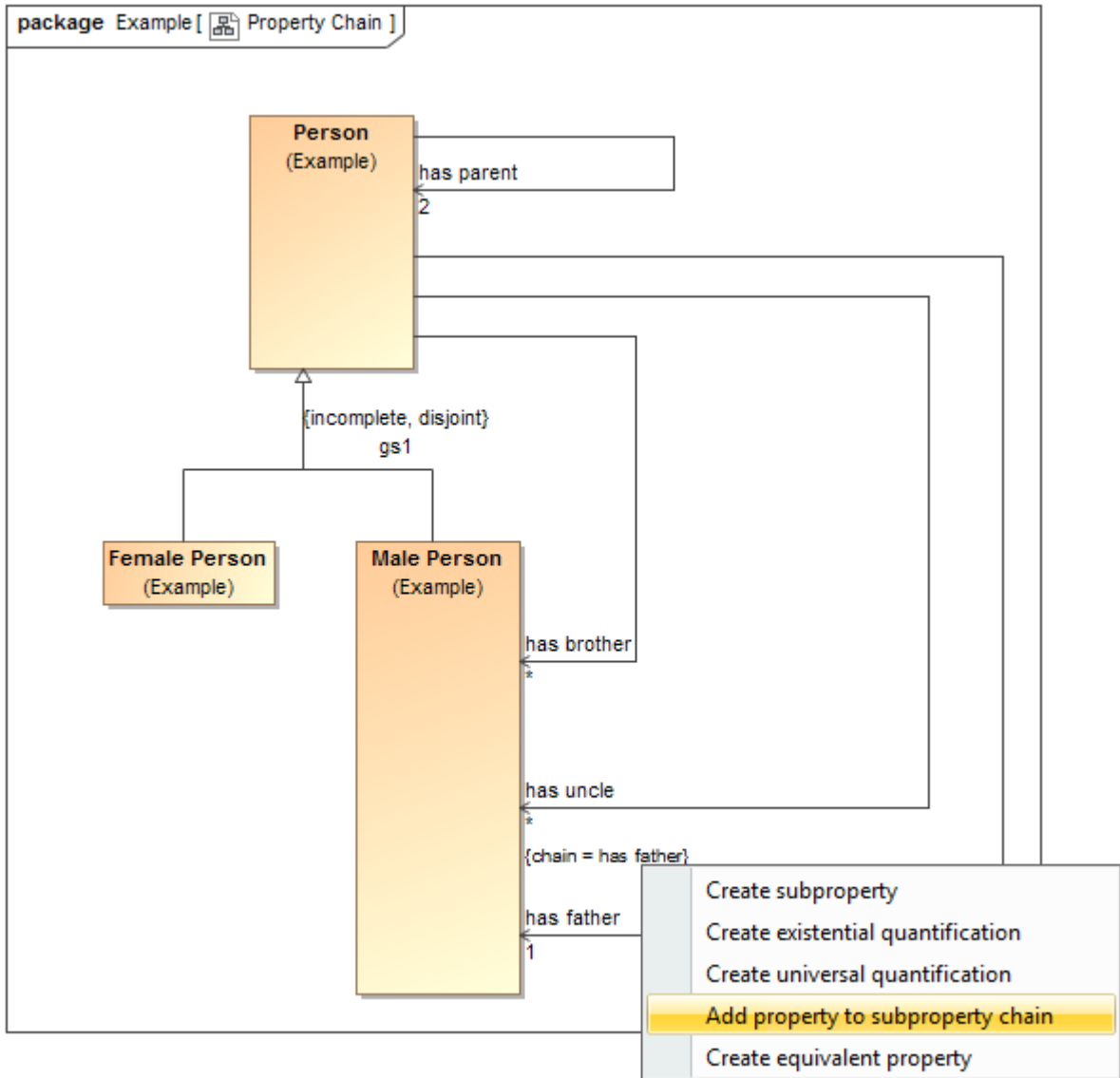
Figure 71   Adding a property to a property chain
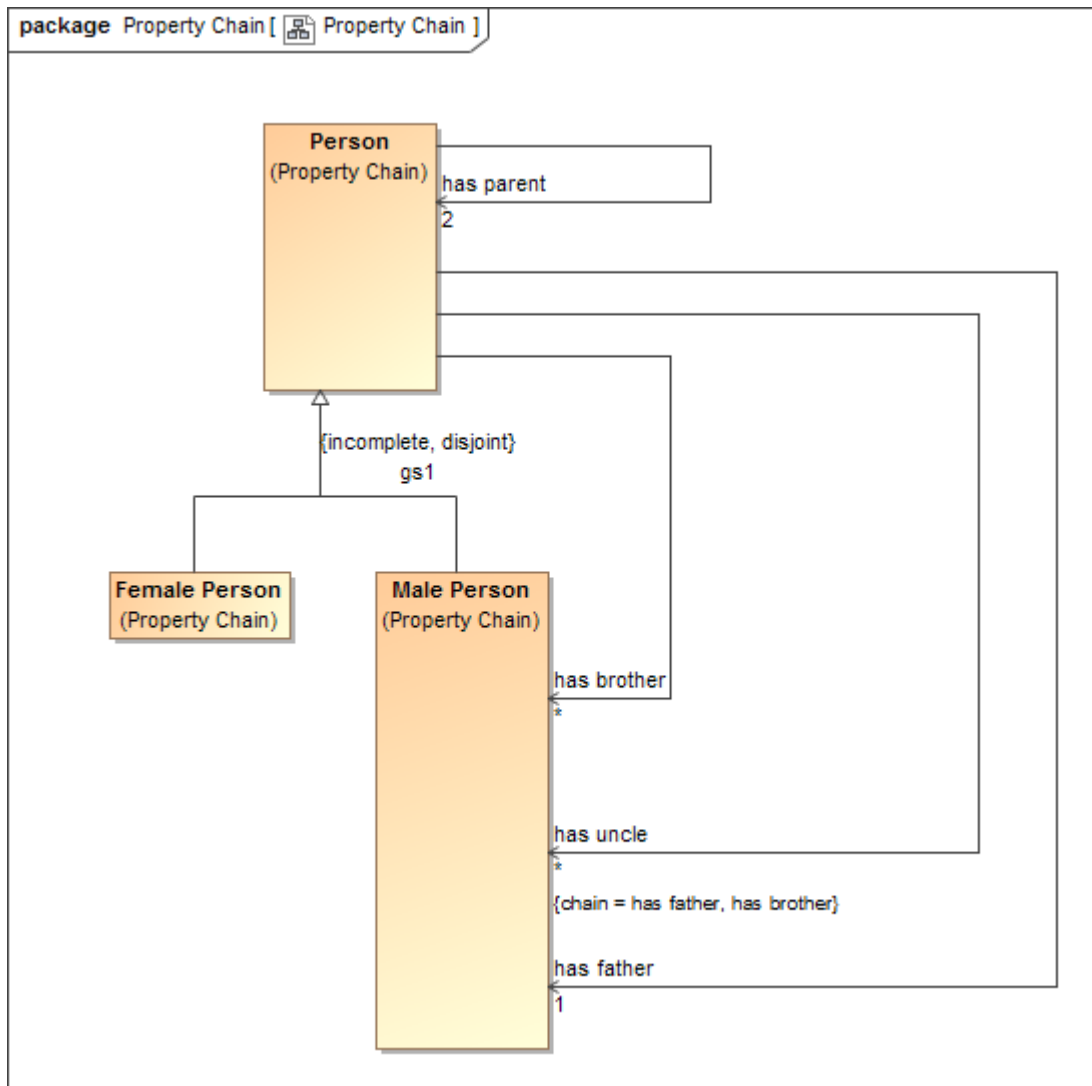
Figure 72   A property chain has been created

You can double-click the property "has uncle" to open its **Specification** window and see the tagged value of the property.

You can delete or edit a property chain using the shortcut menus **Remove subproperty chain** or **Edit subproperty chain**.

To delete a property chain:
1. Right-click a target property or a property chain in the diagram pane.
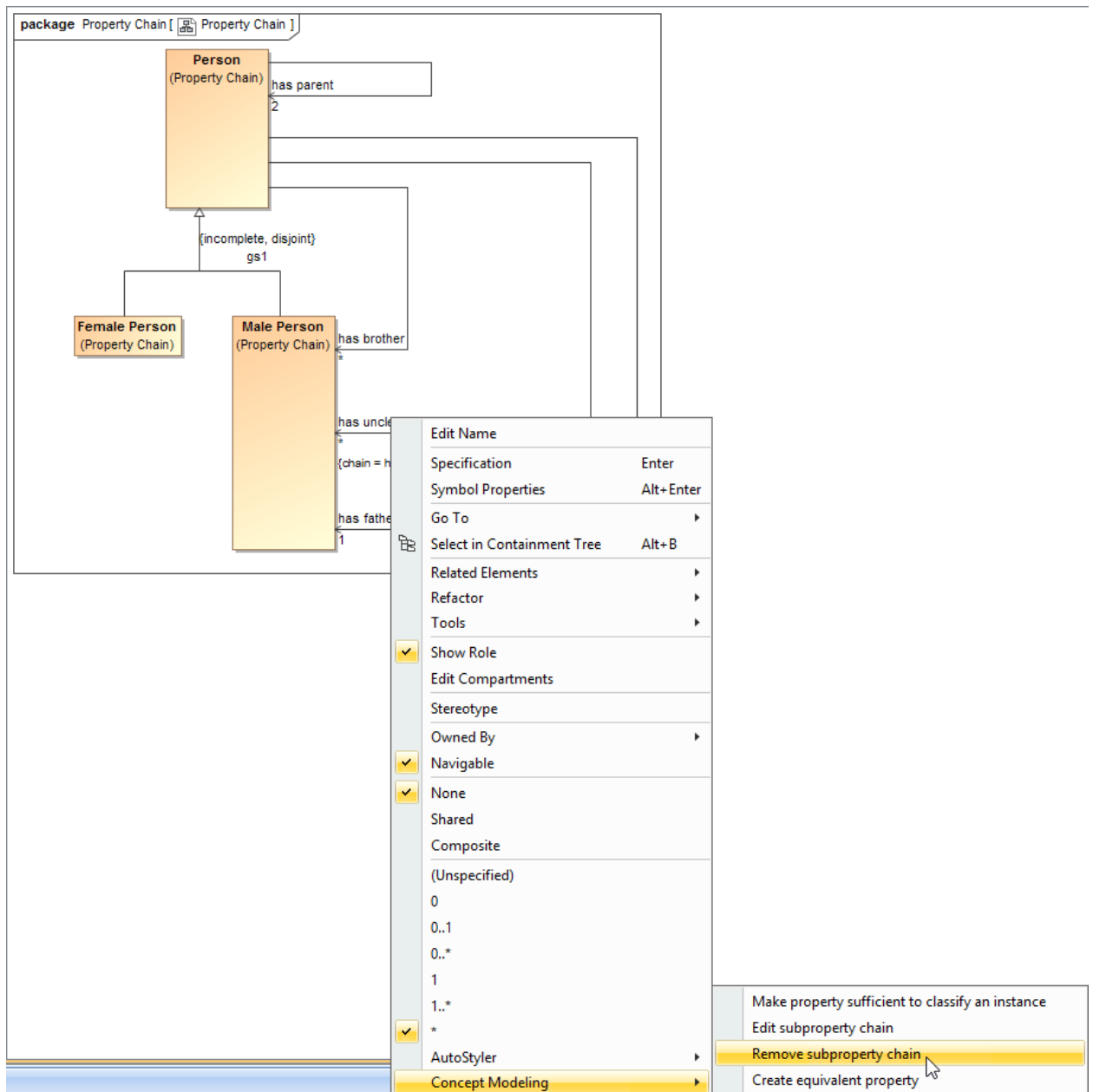
Figure 73   The Remove subproperty chain shortcut menu

2. Select **Concept Modeling** > **Remove subproperty chain** for the shortcut menu. The Concept Modeler will delete all of the properties in the selected property chain.

Tip	Alternatively, you can select **Concept Modeling** > **Edit subproperty chain** and delete the property chain by clicking its tagged value > Remove Value in the **Specification** window of the property.

When editing a property chain, you can add, remove, or reorder properties in a chain by using the **Specification** window.

To edit a property chain:

1. Right-click a property chain in the diagram pane.
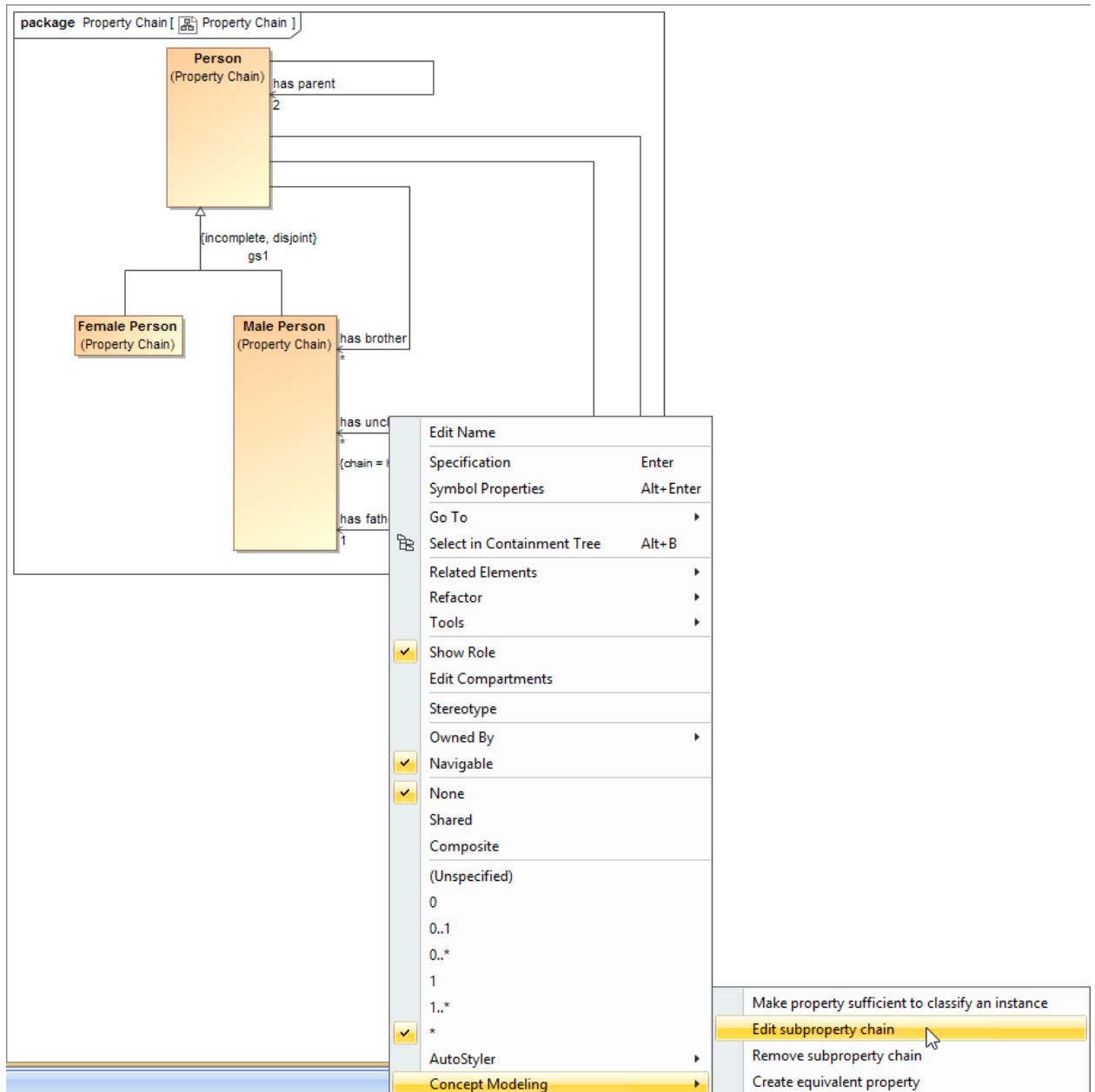
Figure 74   The Edit subproperty chain shortcut menu

2. Select **Concept Modeling** > **Edit subproperty chain**. The **Specification** dialog of the property will open showing the property chain in the **Tags** section.
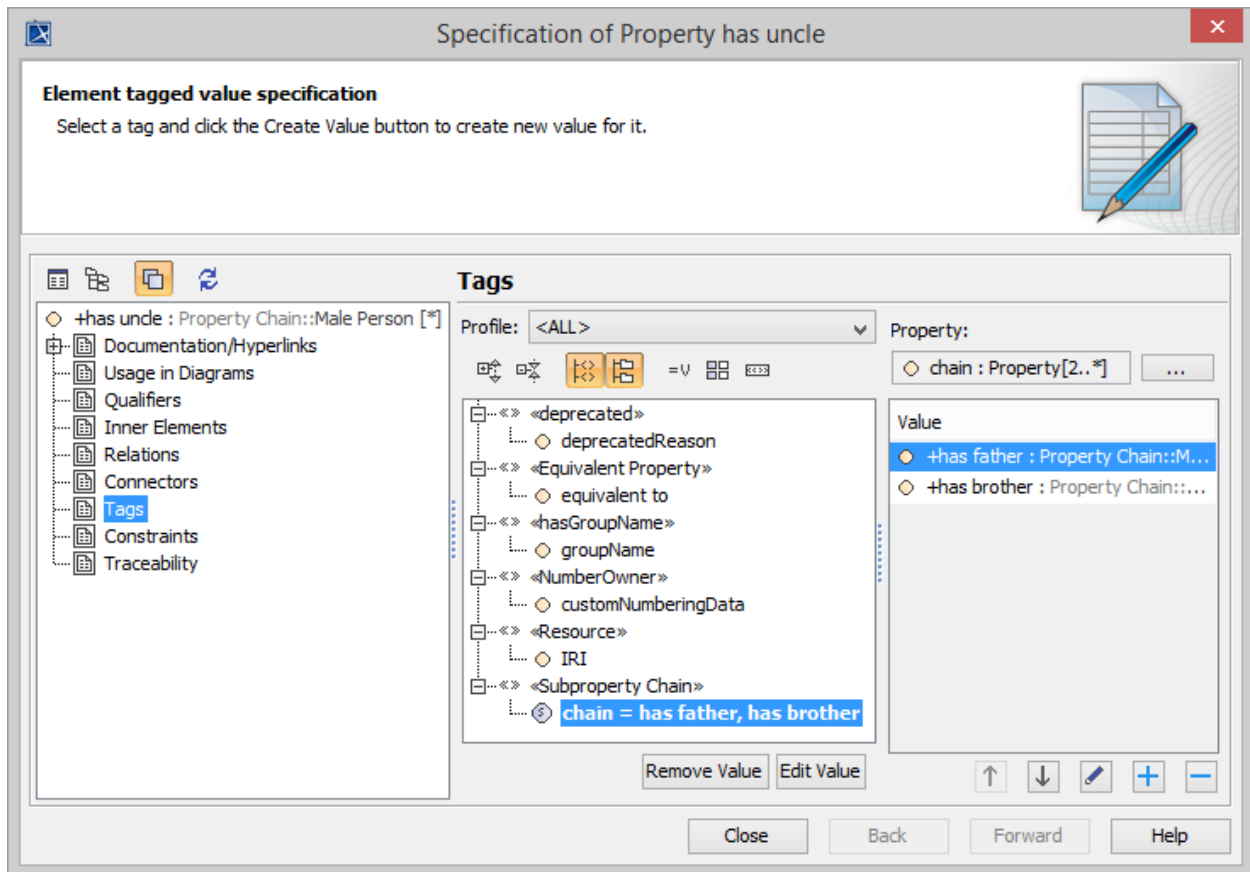
Figure 75   The Specification window of the property "has uncle"

3. Click the tagged value, for example, **chain = has father, has brother**.

4. Click Edit Value. The **Specification of Slot < >** window will open.

Copyright © 2017, No Magic, Inc.
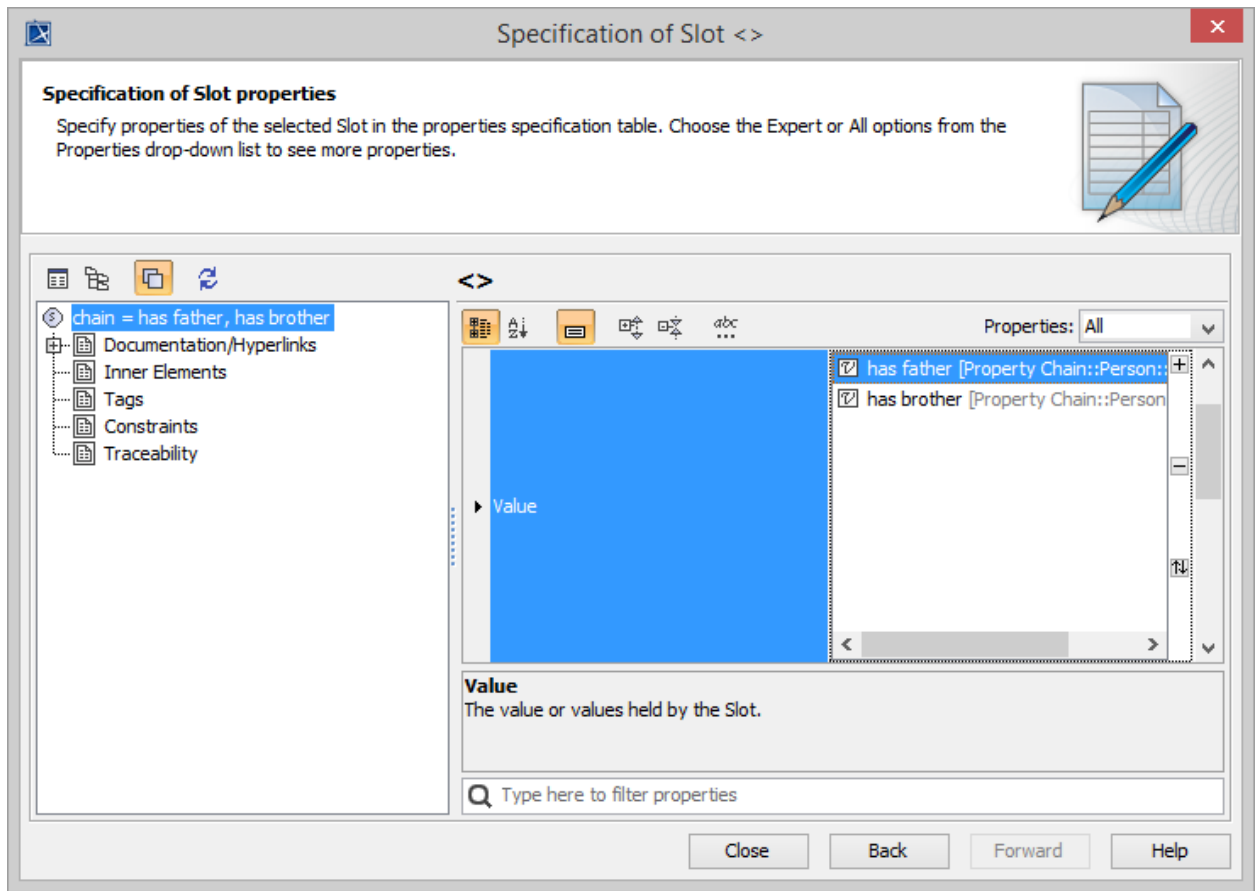
Figure 76   Editing the property chain in the Specification of Slot < > window

5. Click **Value** and click the properties box next to it.
6. You can click:
   (i)   ⊞ to add a property to the chain.
   (ii)   ⊟ to delete a selected property from the chain.
   (iii) ⇅ to order the properties in the chain.

5.2.3   Create Equivalent Property

Properties can be equivalent to each other. You can make two or more properties equivalent by dragging a property to the target property. The Concept Modeler provides you with the following shortcut menus:

   (i)   Create equivalent property
   (ii)   Add property to equivalent property

A property is semantically equivalent to another property if it is stereotyped with **«Equivalent Property»** and its tagged value is **equivalent to**.

Copyright © 2017, No Magic, Inc.

To create two or more equivalent properties:

1. Drag a property, for example, "has dad" to a target property, for example, "has father" in the diagram pane. The shortcut menu will open.
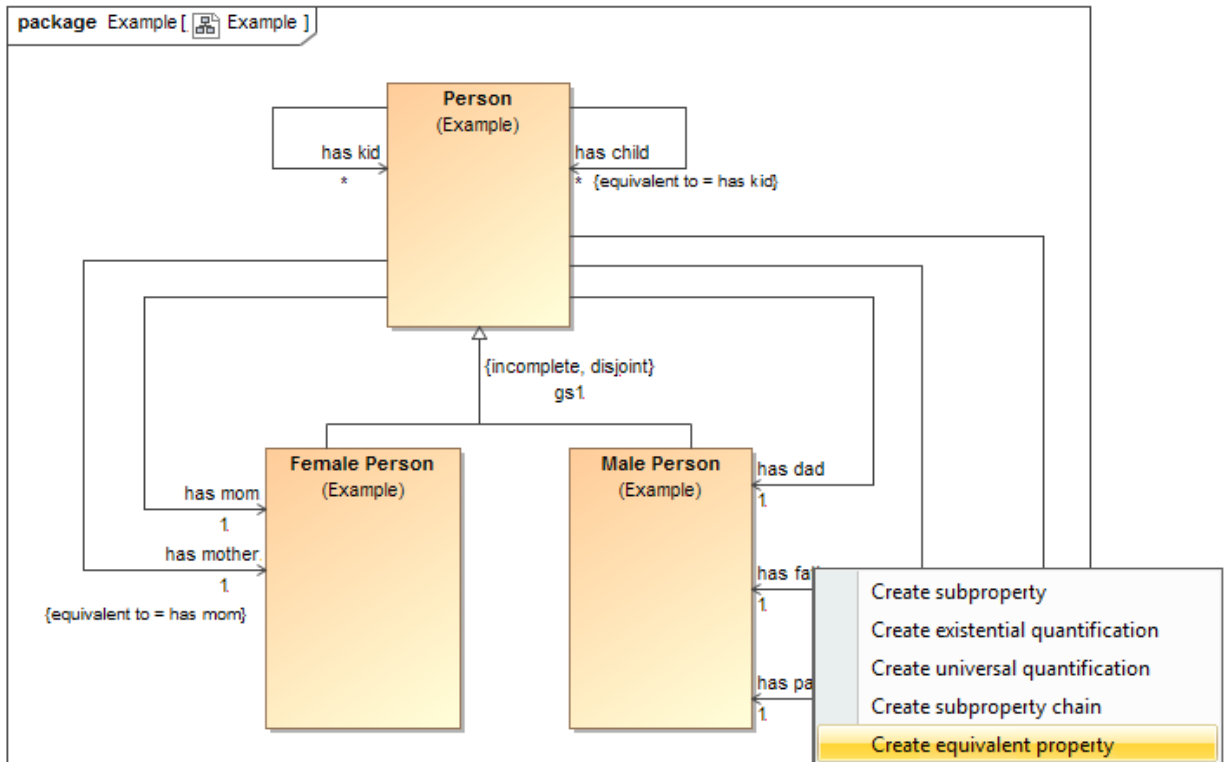
Figure 77   The Create equivalent property shortcut menu

2. Select **Create equivalent property**. The property "has father" is now equivalent to the property "has dad".

Copyright © 2017, No Magic, Inc.
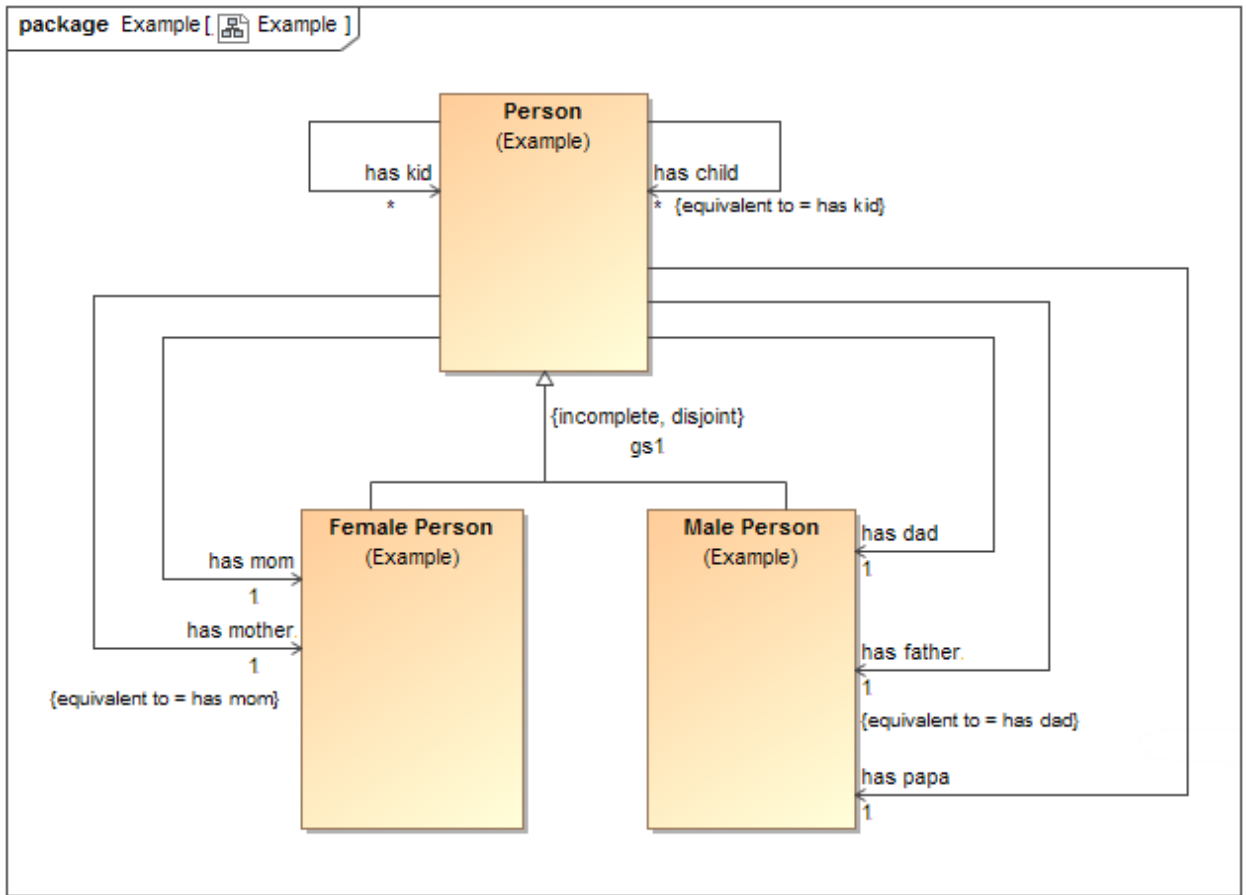
Figure 78   Two equivalent properties "has father" and "has dad"

3. You may add more properties to the existing equivalent properties by dragging the next property, for example, "has papa" to the equivalent property "has father" in the diagram pane.
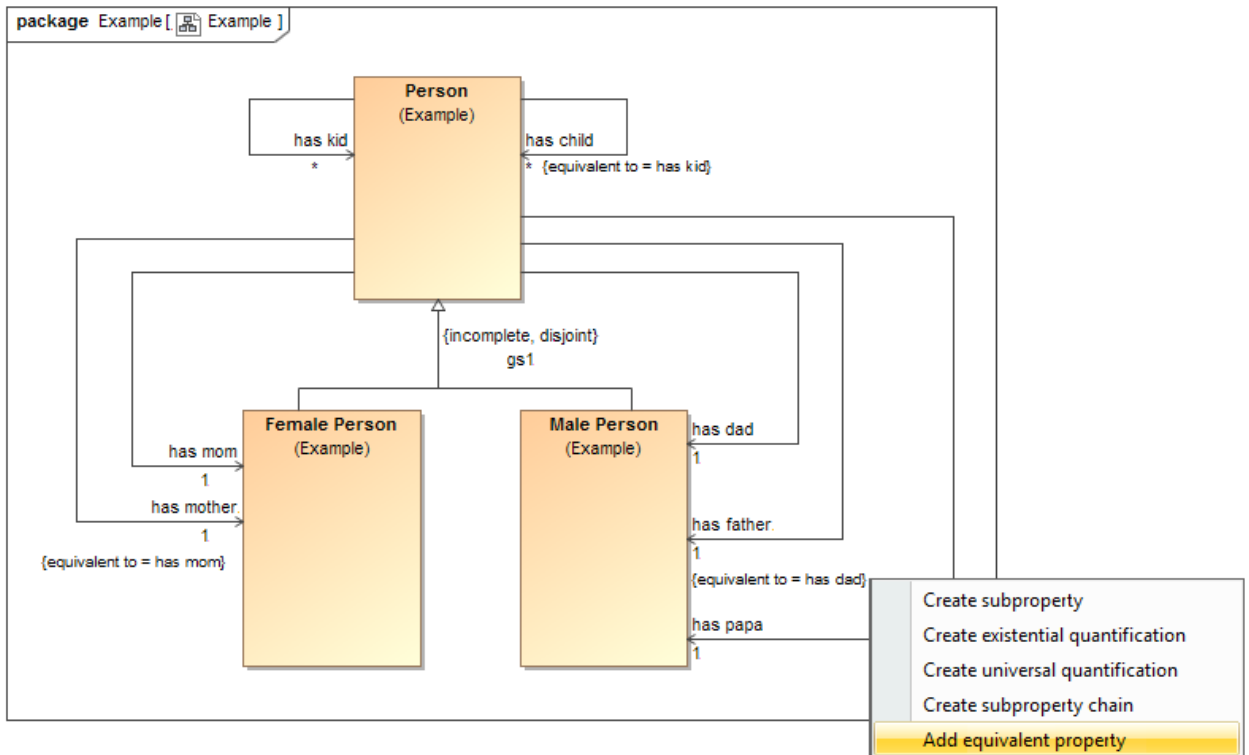
Figure 79 The Add equivalent property shortcut menu

4. Select **Add property to equivalent property**. The property "has papa" is now equivalent
    to the properties "has father" and "has dad".

Copyright © 2017, No Magic, Inc.
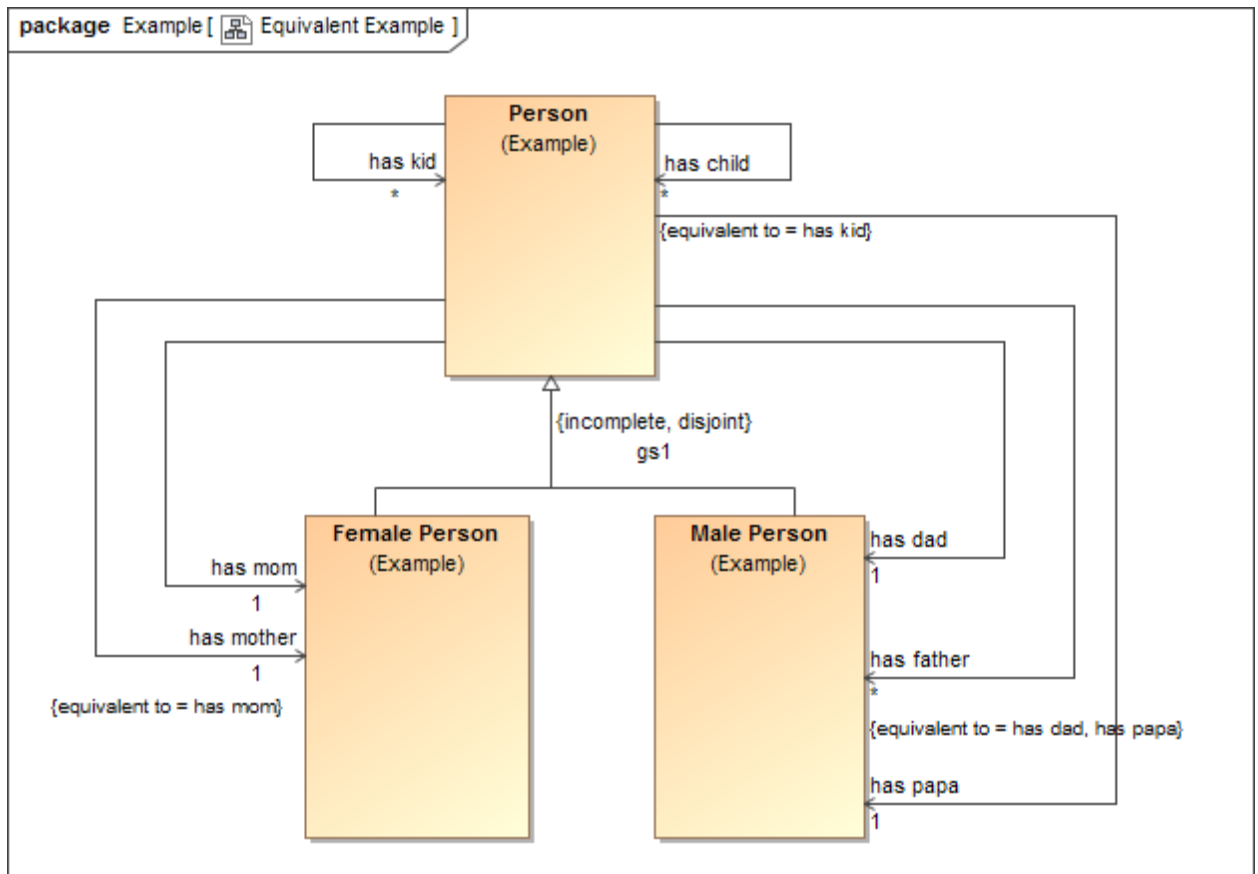
Figure 80   Equivalent properties in the Concept Modeler

Tip	Alternatively, you can make a property equivalent to one or more properties by right-clicking it and select **Concept Modeling** > **Create equivalent property** from the shortcut menu, and select one or more properties from the tree in the **Select Property** dialog.
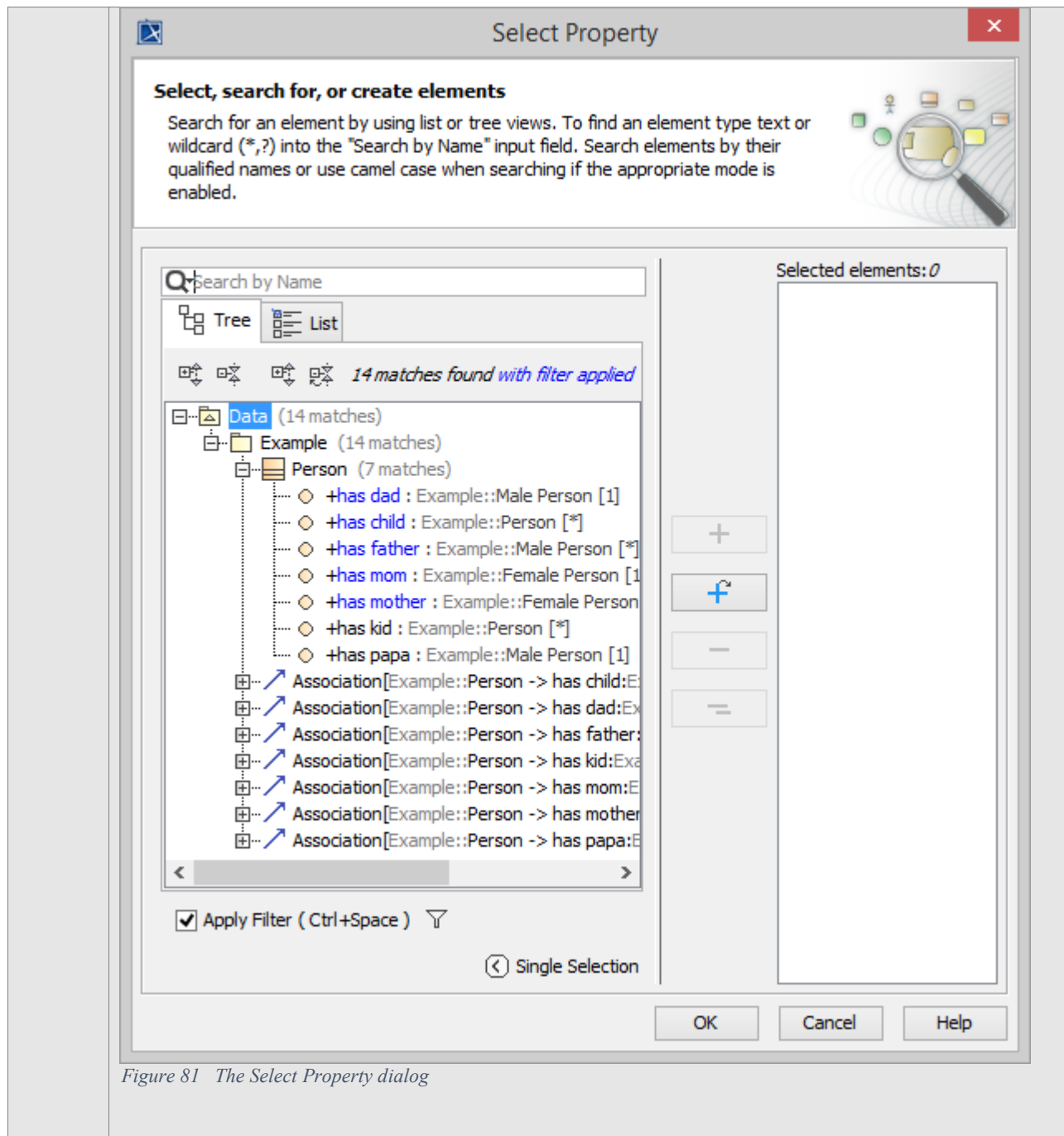
Figure 81 The Select Property dialog

To delete an equivalent property:

1. Right-click the target property, for example, "has father {equivalent to = has dad, has papa}", in the diagram pane. The shortcut menu will open.
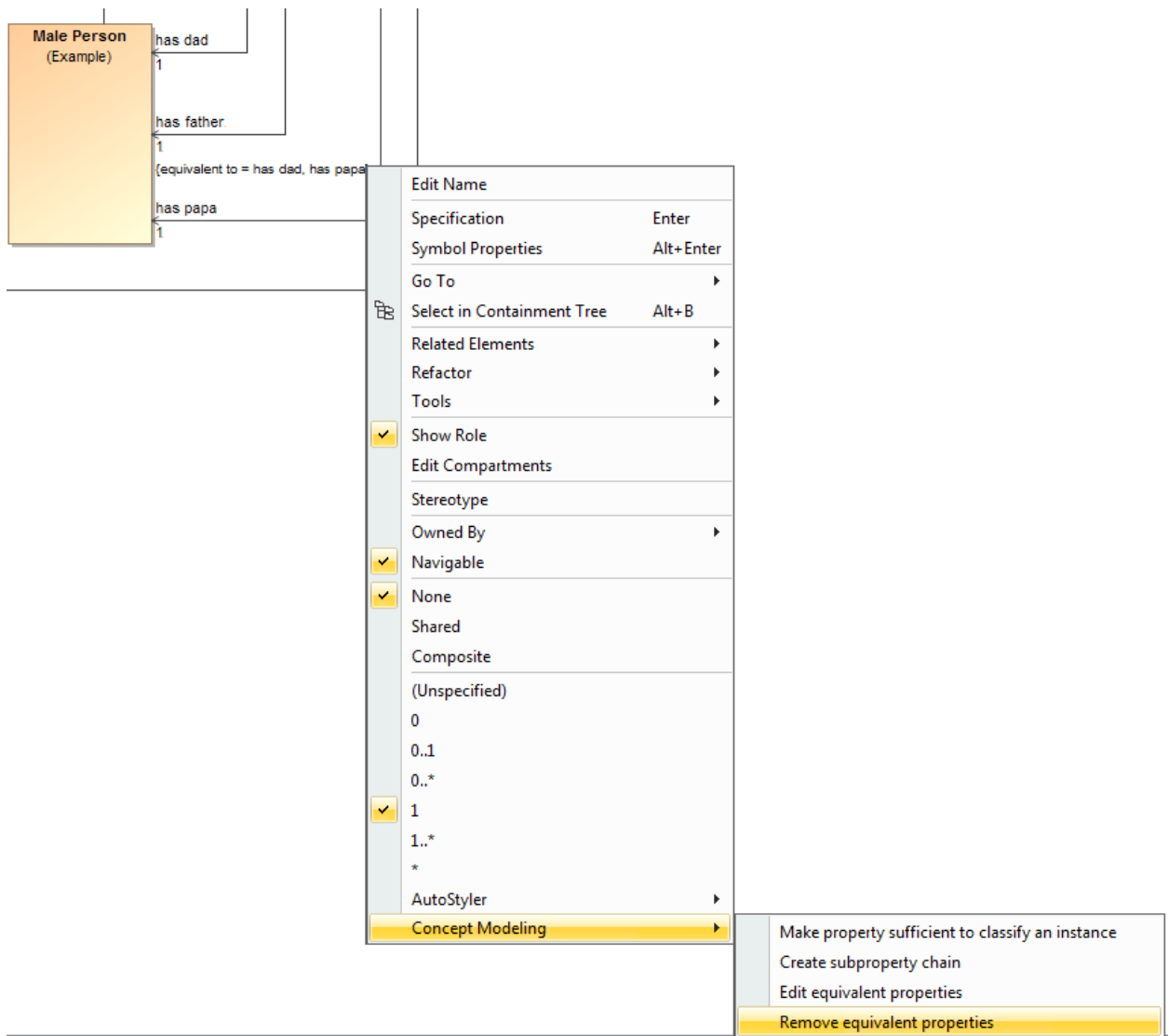
Figure 82 The Remove equivalent property shortcut menu

2. Select **Concept Modeling** > **Remove equivalent properties**. The Concept Modeler will remove all of the equivalent properties.

To edit an equivalent property:
1. Right-click an equivalent property, for example, "has father", in the diagram pane.
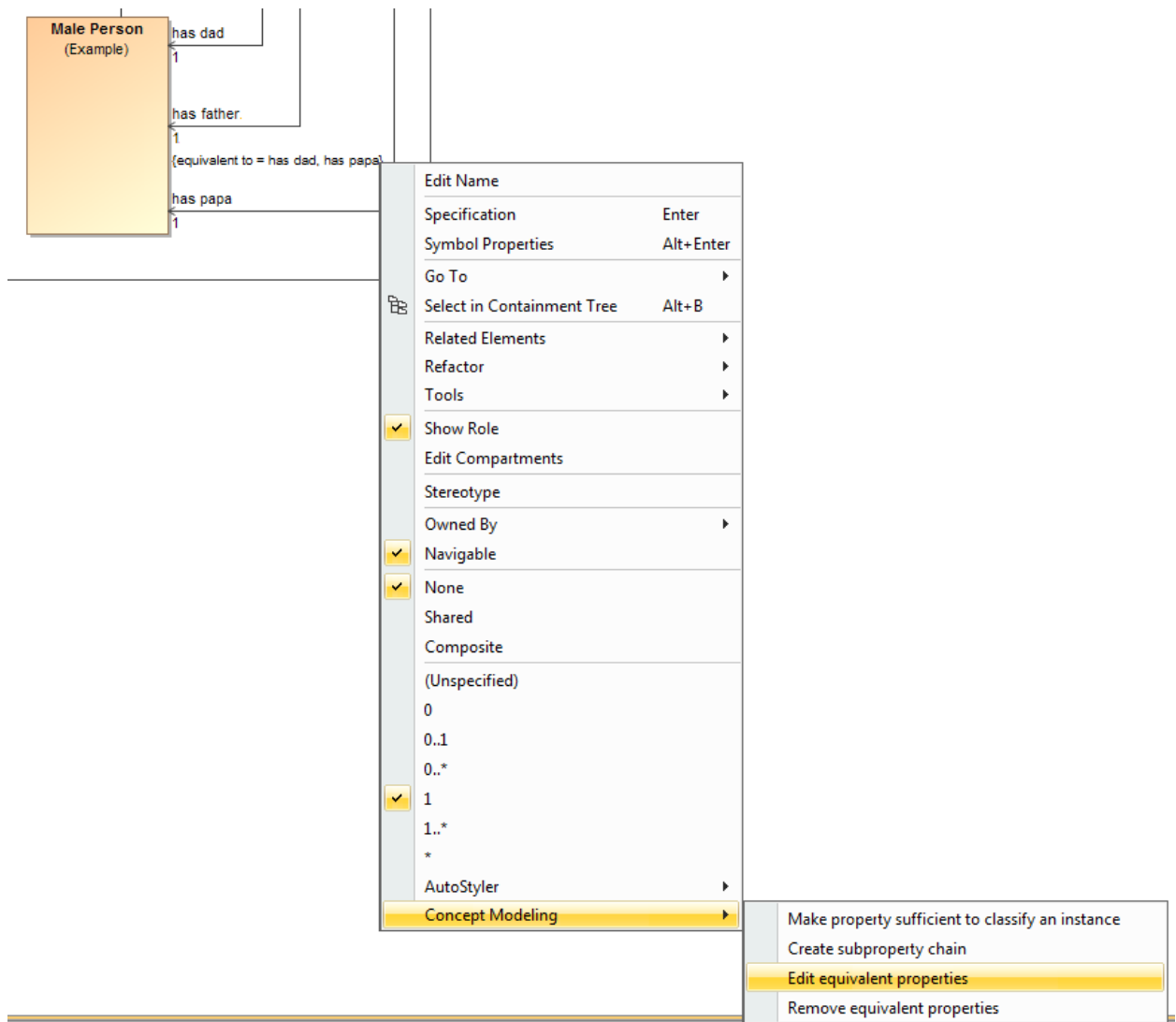
Figure 83   The Edit equivalent properties shortcut menu

2. Select **Concept Modeling** > **Edit equivalent properties**. The **Specification of Property has father** window will open showing the equivalent properties under the section **Tags**.
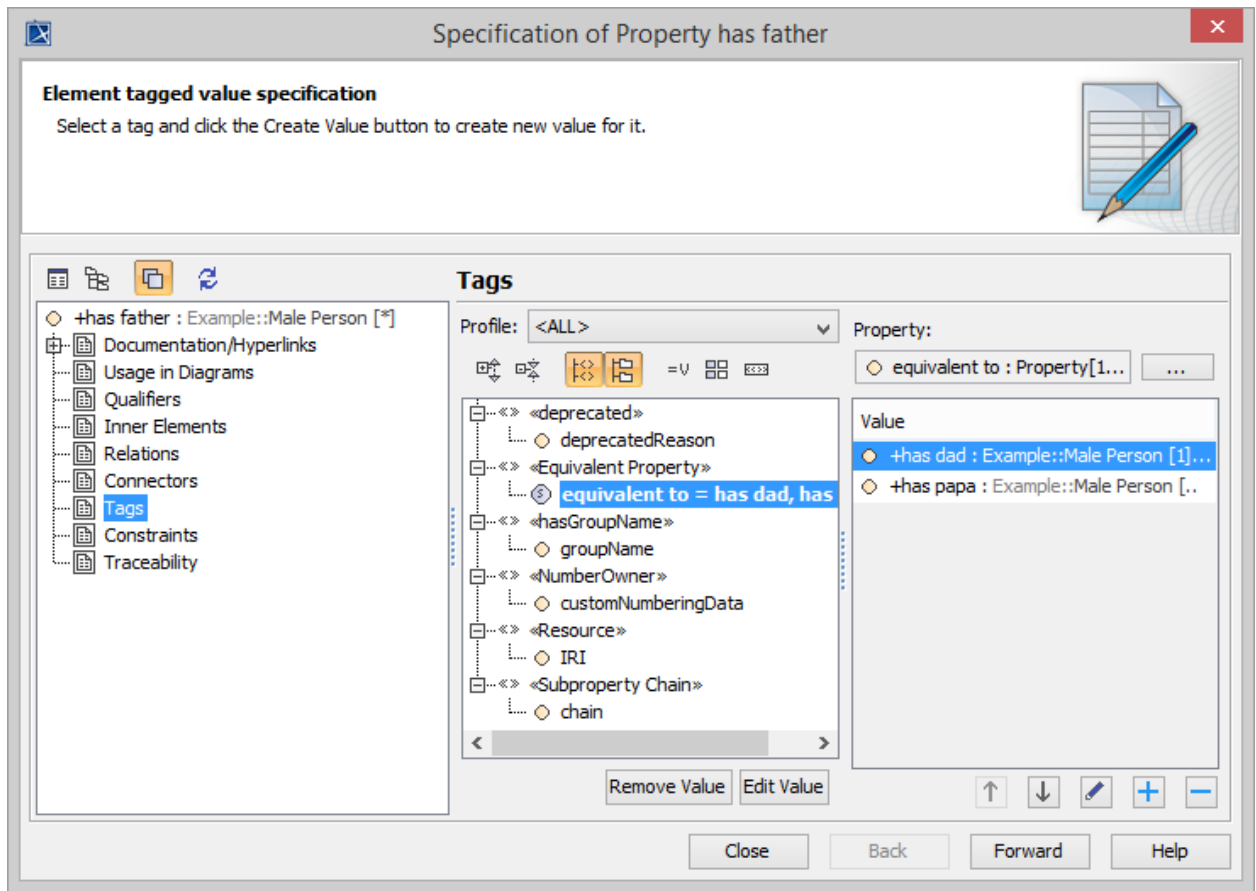
Figure 84   The Specification window of Property has father

3. Click the tagged value, for example, **equivalent to = has dad, has papa**.
4. Click Edit Value. The **Specification of Slot < >** window will open.
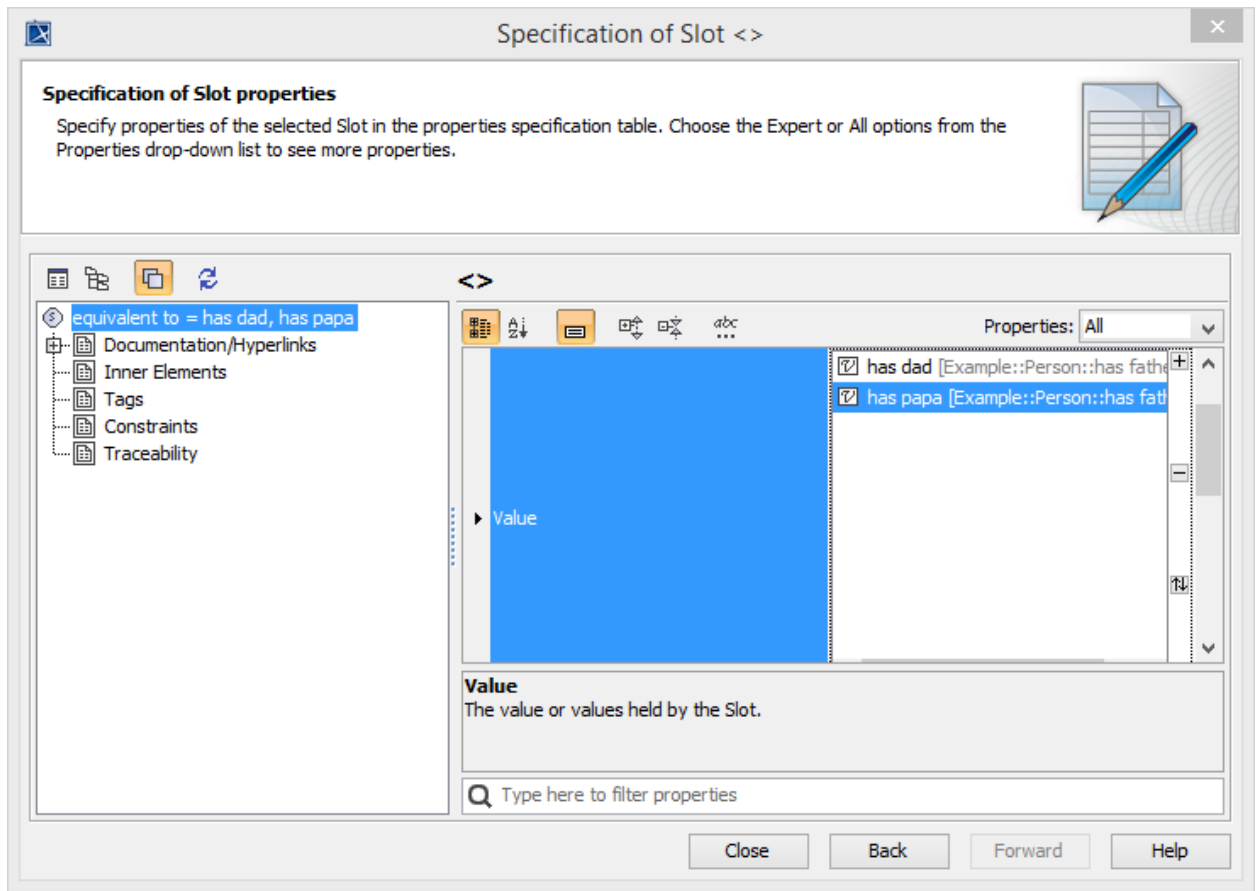
Figure 85   The Specification window of Slot < >

5. Click **Value** and click the properties box next to it.
6. You can click:
   (i)   ⊞ to add another equivalent property.
   (ii) ⊟ to delete a selected equivalent property.
   (iii) ⇅ to order the equivalent properties in the **Order Value** dialog.

5.2.4   Create Equivalent Classes

To create equivalent classes:

1. Click ⬈ Equivalent Class on the Concept Modeling diagram palette.

2. In the diagram pane, click a class and drag the line to another class to make them equivalent to each other. A double-headed arrow will be created between the two classes and the stereotype **«Equivalent Class»** will be visible.
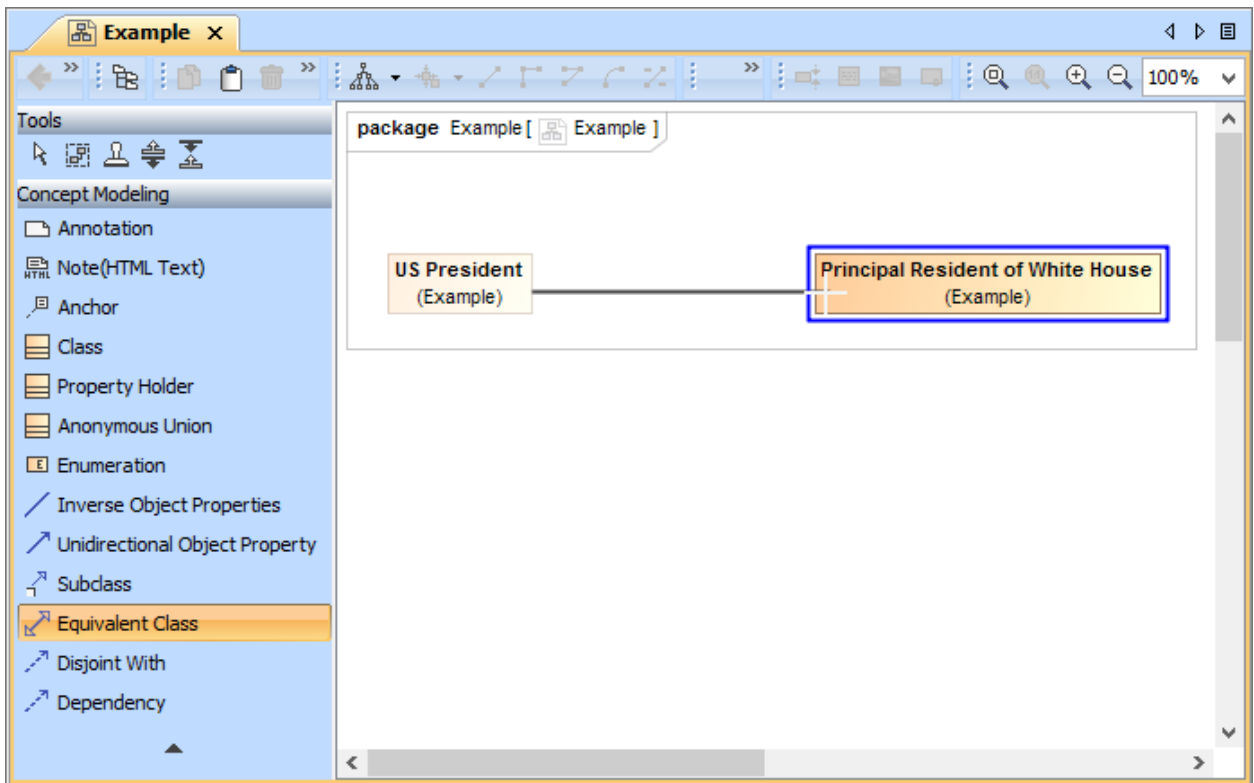
Figure 86   Creating class equivalence between two classes
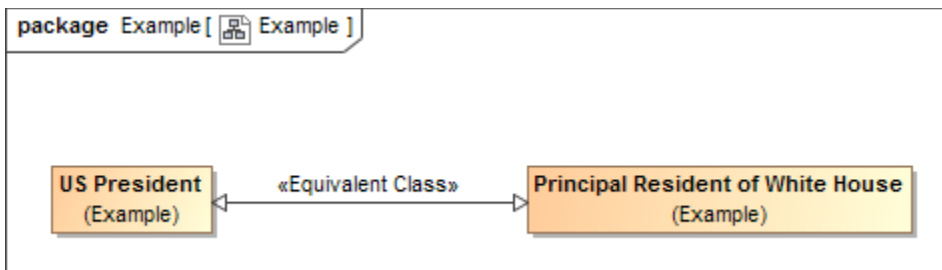
Figure 87   The classes are equivalent to each other

5.3   Set the Concept Model URI

URI stands for Uniform Resource Identifier. A URI can provide identification about a location to a resource (a document, a person, an abstract thing) and a name or both, depending on the context. The URI is used as a single global identification system in the Web.

On the semantic Web, not only can you use URIs for Web documents (to link to and access them in a Web browser), but also for real world objects (such as people, cars, and even abstract ideas).

A concept model must have a valid URI before it can be exported to an OWL ontology. If you forget to change the default URI, the notification window will open and remind you to change it

Copyright © 2017, No Magic, Inc.

when you export that concept model to an OWL ontology. The last part of this URI is used as the filename, and the extension for this file will be derived from the export format. (See section 5.6.2 Set the Concept Model Export URI Style for export format options.)

To set the concept model URI:
1. Right-click on the desired **ontology package** in the Containment tree.
2. Select **Specification**.
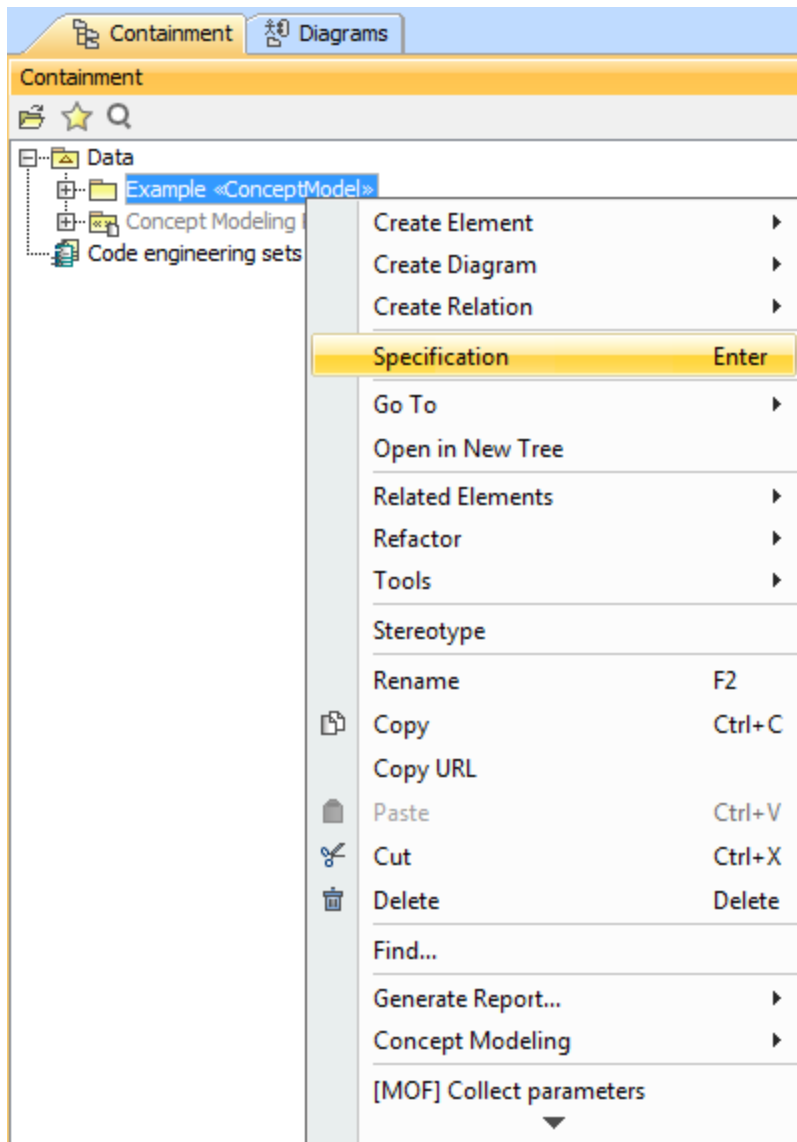
Figure 88   Opening the Specification dialog of a selected package

3. Scroll down to **URI,** or type "**URI**" in the search field at the bottom of page.

Note	A default URI will be set for new concept modeling projects and newly created concept models. Updating the URI to match your ontology is recommended. For situations in which a URI has not been set, or the default has not been changed, a warning message

	will appear in the notification window on OWL export.

4. Click on the field next to **URI**.
5. Update the URI and click **OK**.

5.4   Create the XML Catalog File

The XML catalog file can be created using the application **Protégé**[1]. The version of **Protégé** used in these instructions is version 4.3.0.

To create an XML catalog file for a locally cached set of external ontologies:
1. Open **Protégé**.
2. Select **File** > **New**.

---

[1] **Protégé** is a free, open-source ontology editor and a framework for building knowledge management systems.
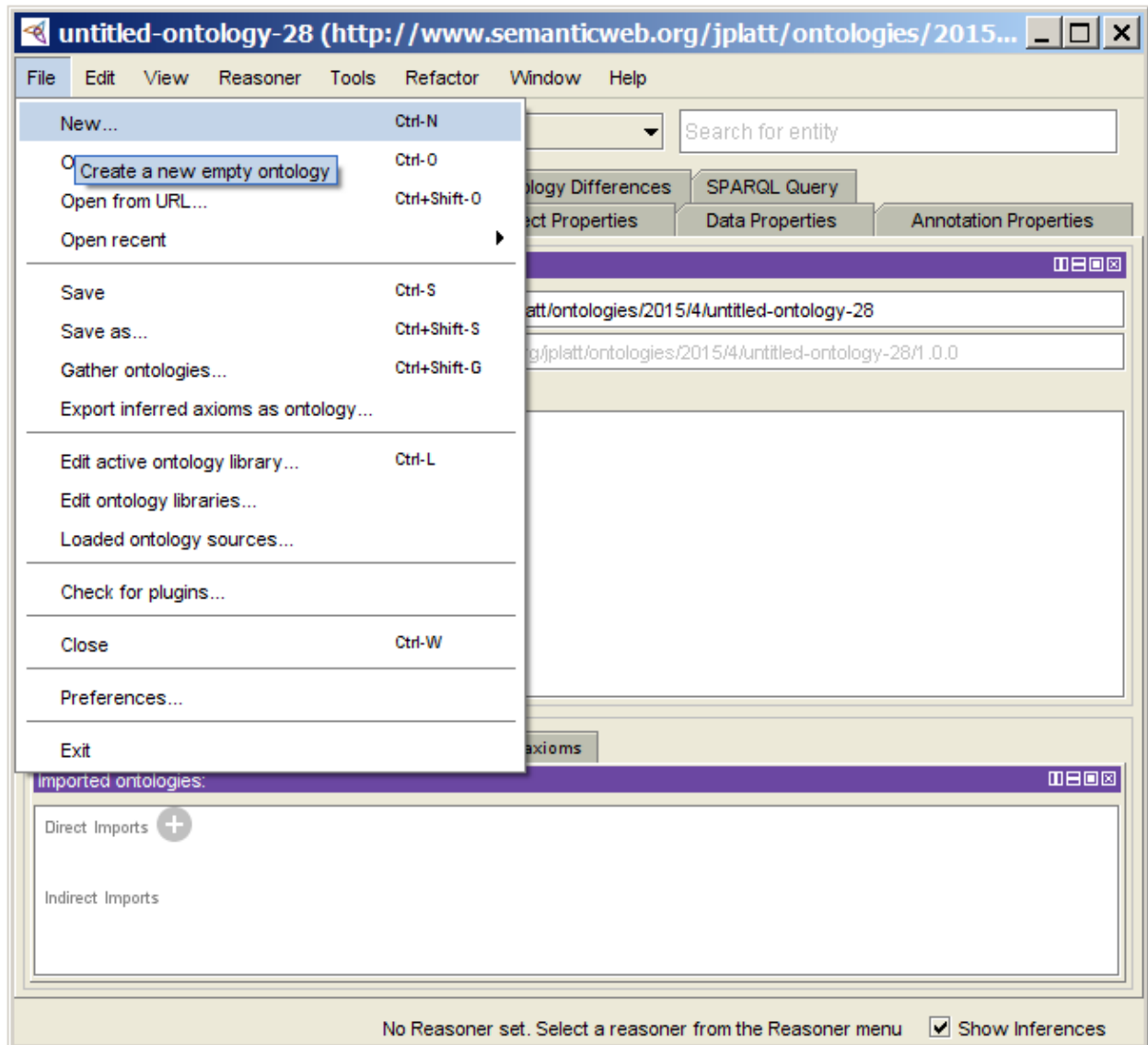
Figure 89 Creating an OWL ontology in Protege application

3. Select **Save as…**
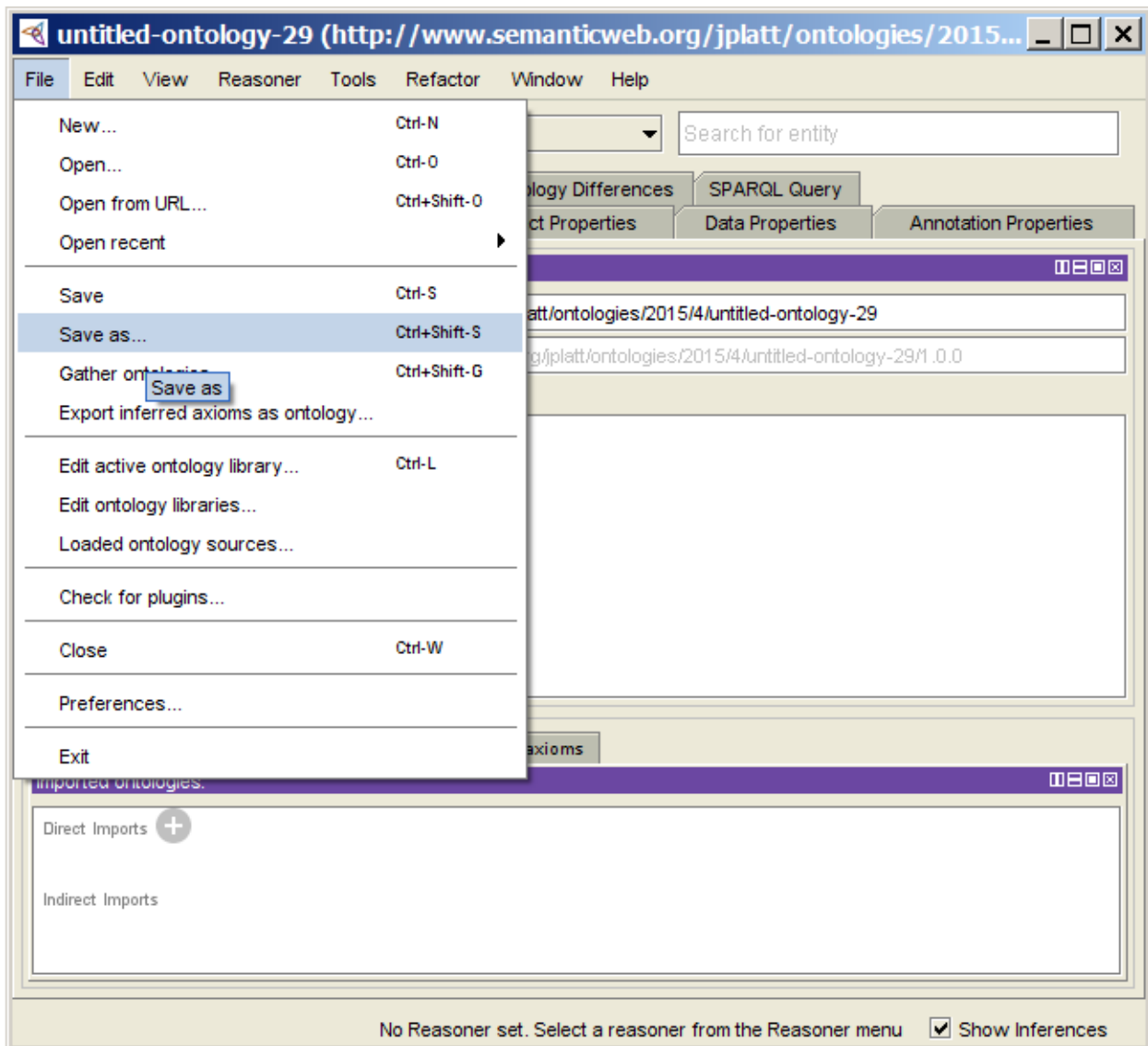
Figure 90   Saving the OWL ontology in Protégé

4. Click **OK.**
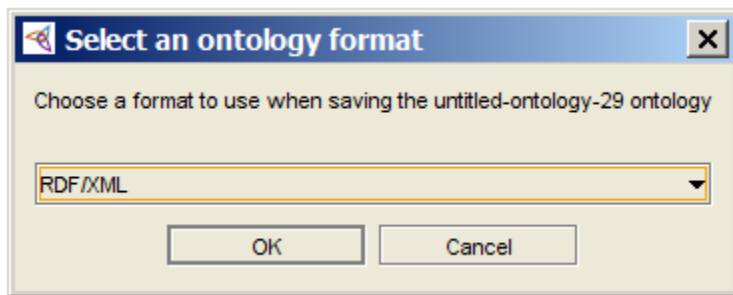
Figure 91   Selecting an ontology file format option

5. Navigate to a folder location.
6. Name the empty ontology.

7. Click **Save**.
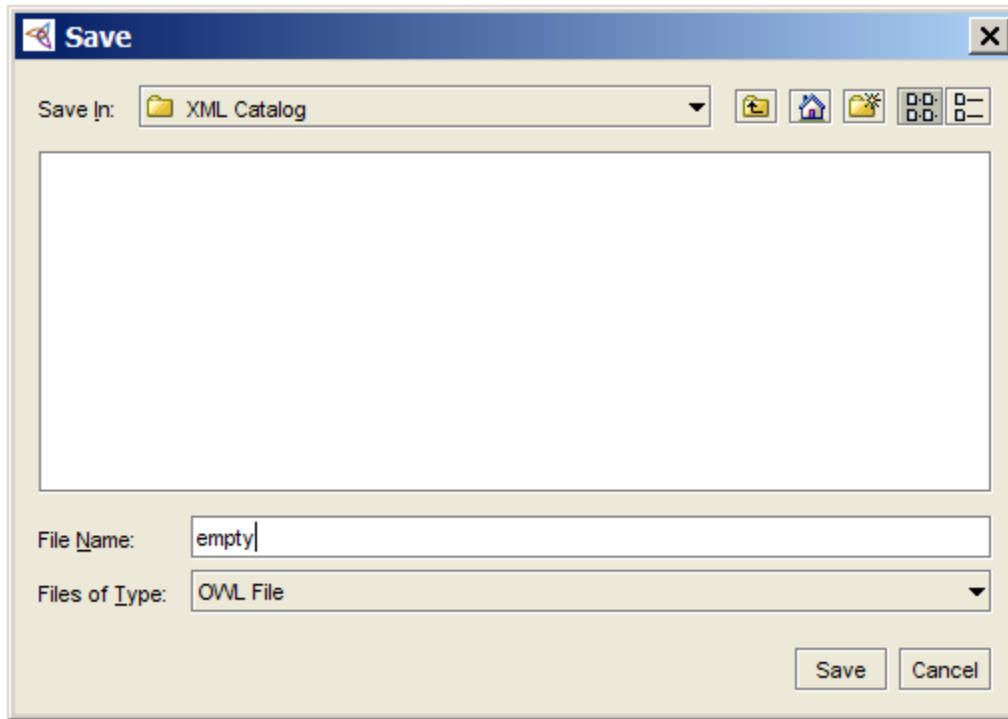
Figure 92   Saving the OWL ontology file to a selected location

Copyright © 2017, No Magic, Inc.

8. Click **File** > **Open**.
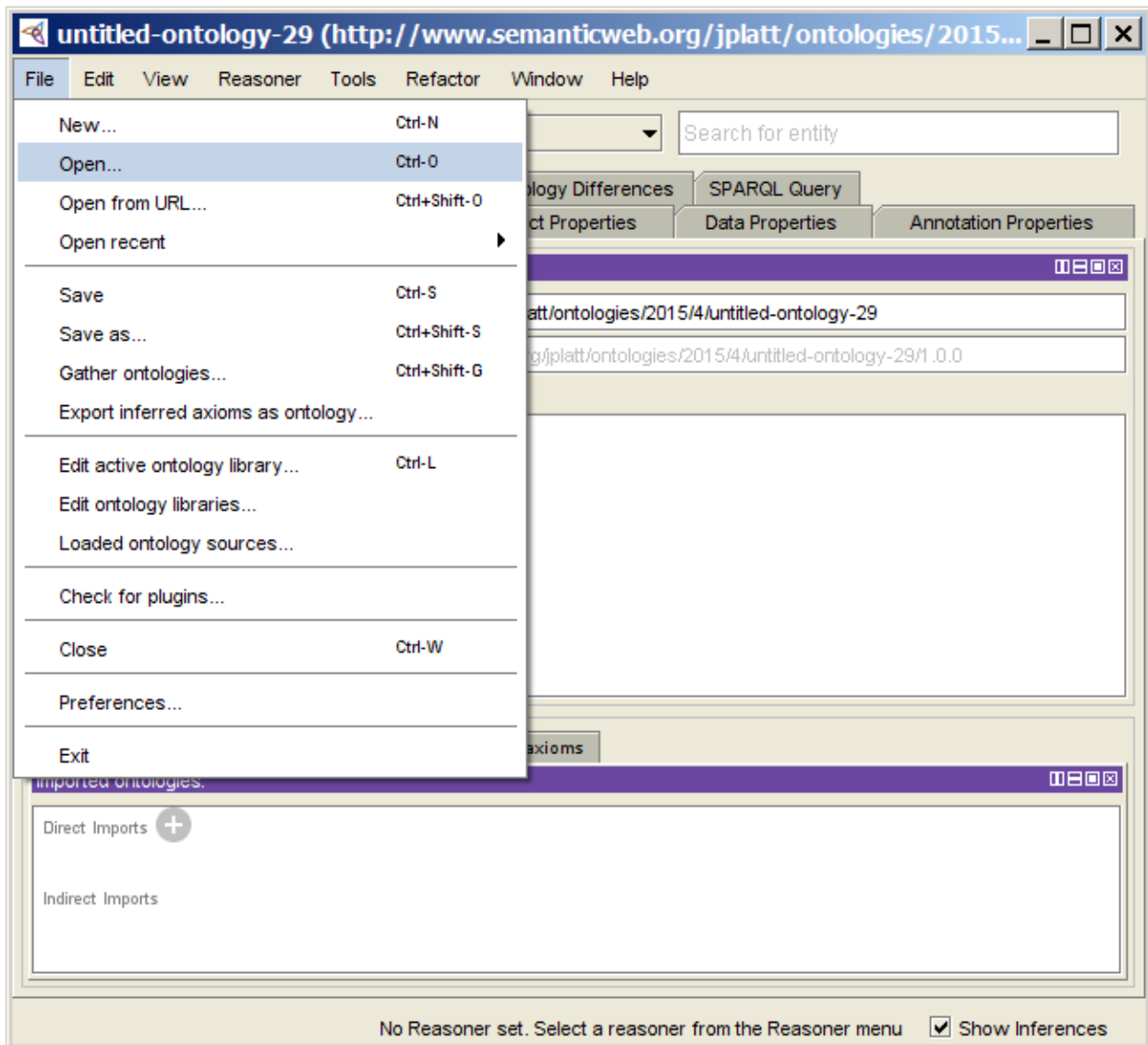
Figure 93   Opening an OWL ontology menu in Protégé

9. Click **Yes**.
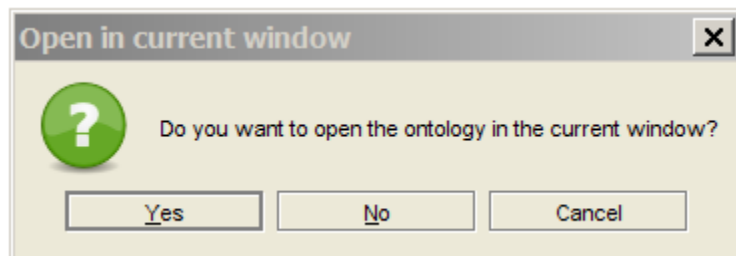
Figure 94   Opening ontology in the current window option

10. Select the newly created ontology.
11. Click **Open**.
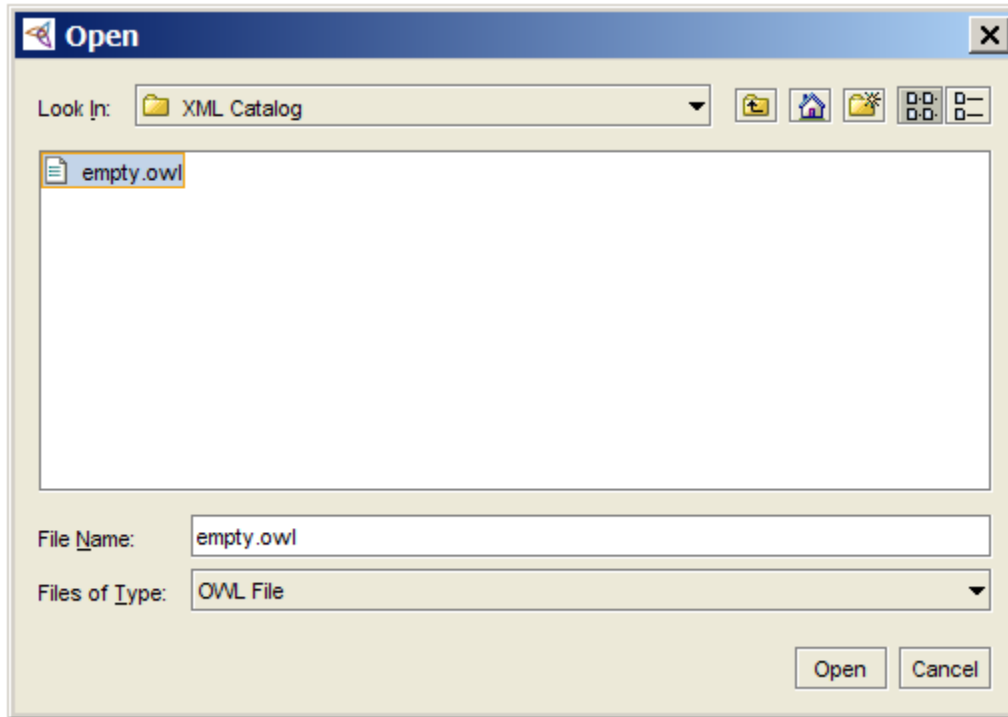
Figure 95   Selecting an OWL ontology to open

To create an XML catalog file for the desired locally cached set of external ontologies:

1. Click **File** > **Edit active ontology library**.
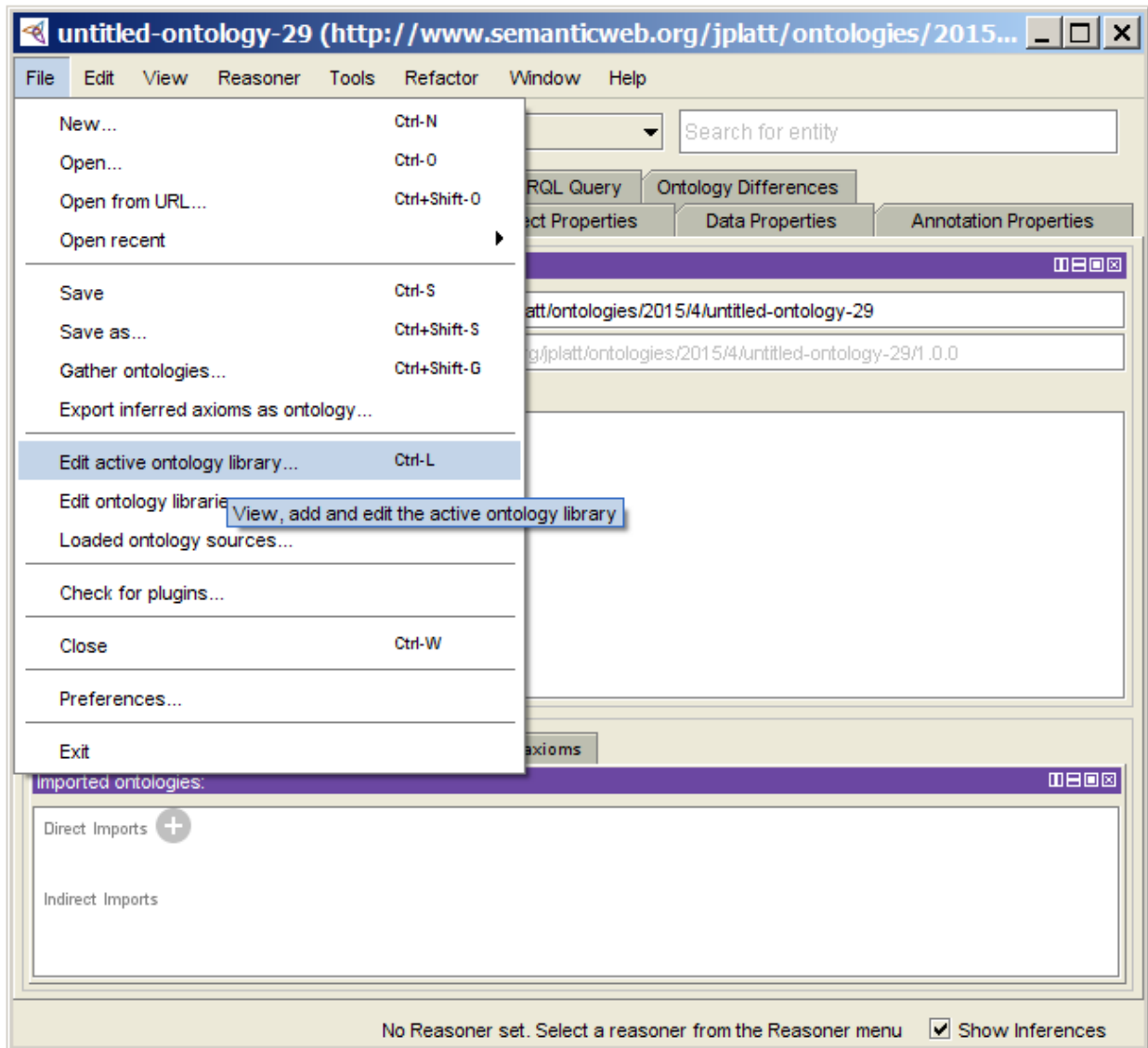
Figure 96   Creating an XML catalog file in Protégé

2. Click on the **Folder Repository** for the empty ontology.
3. Click Delete (-)
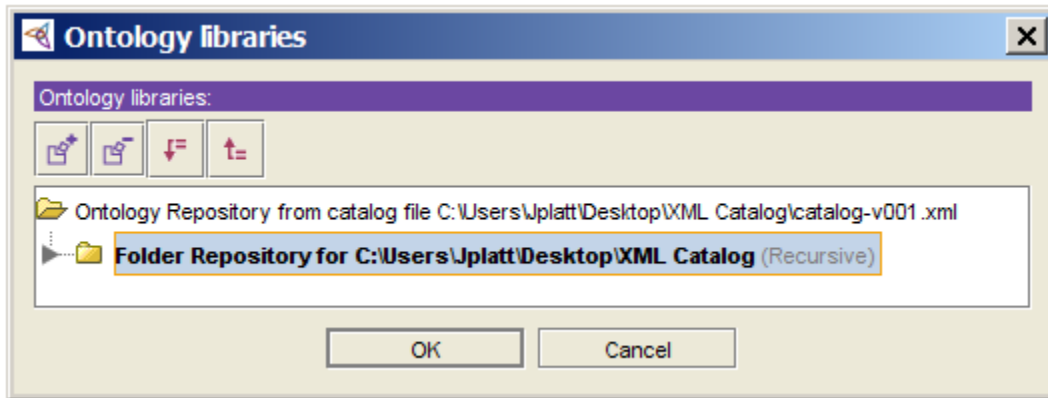
Figure 97   Deleting a folder repository

4. Click Add (+).
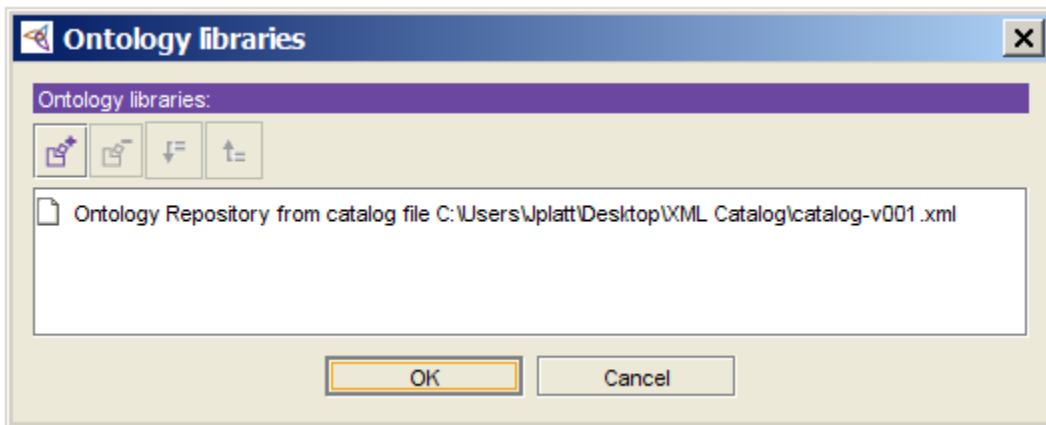
Figure 98   Adding a folder repository

5. Click on **Folder Repository.**
6. Select **Recursively search subdirectories**.
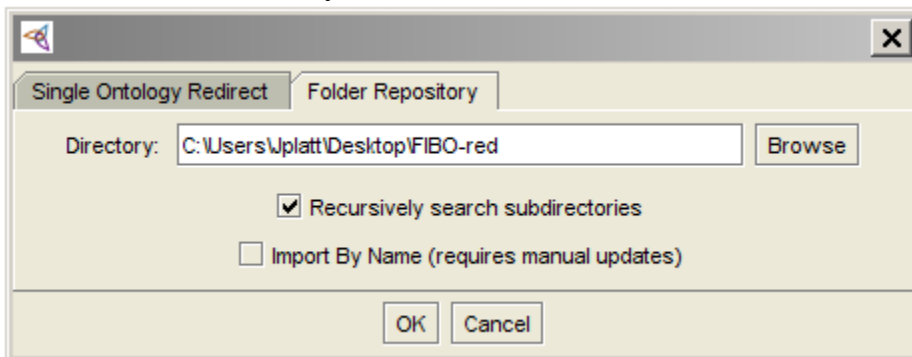
Figure 99 Locating for a folder repository

7. Select the desired folder.
8. Click **Open.**
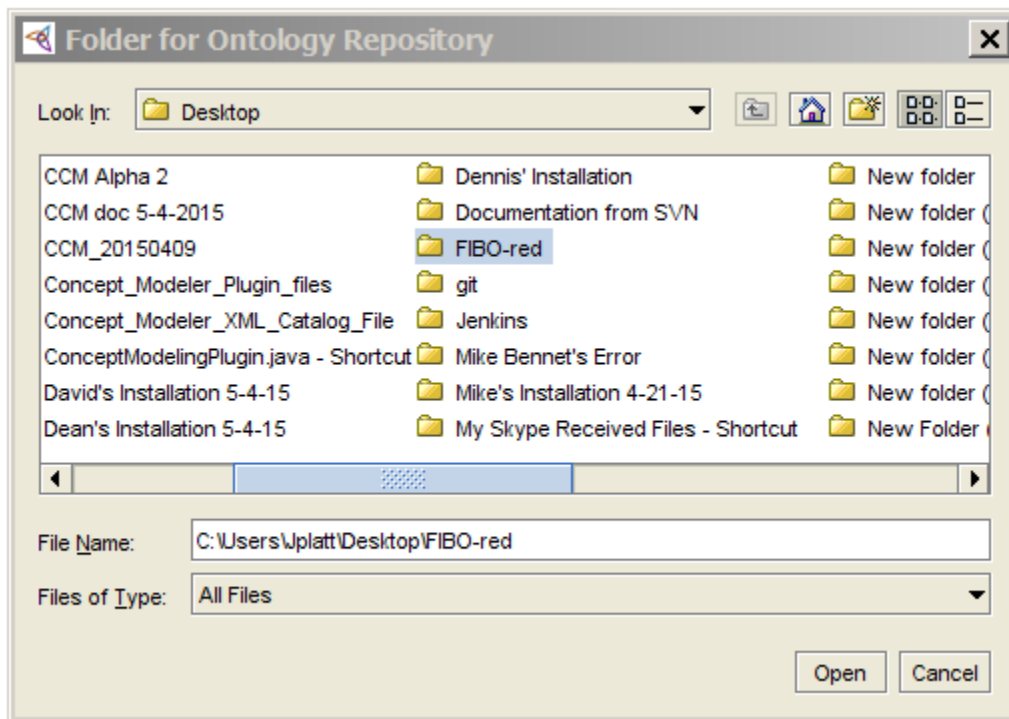
Figure 100   Selecting a folder repository

9. Click **OK**.
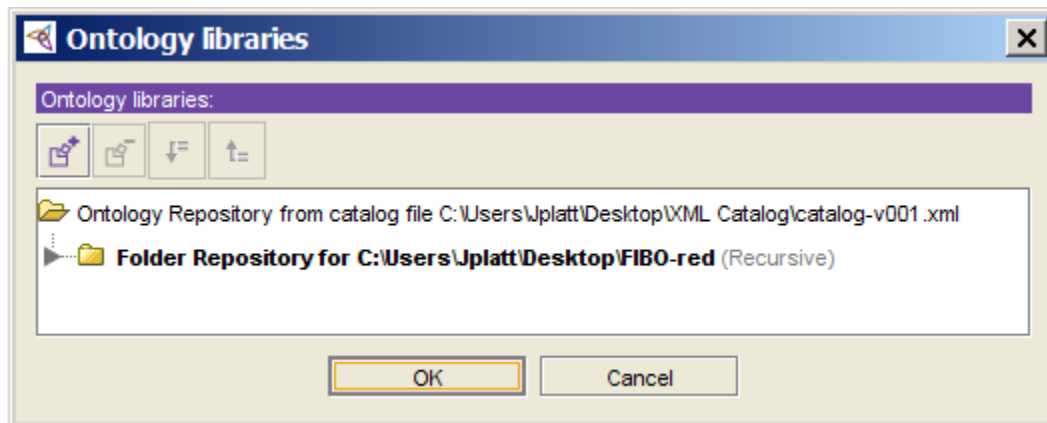
Figure 101   A new folder repository is added

Copyright © 2017, No Magic, Inc.

5.5 Import an OWL Ontology to a Concept Model

5.5.1 Update the XML Catalog File

The Concept Modeler can import a local ontology model into the concept model. Occasionally, an ontology file may contain references to external ontologies (for example, ontologies not stored locally and not available to import directly into the concept model). An XML catalog (OASIS Standard V1.1) may be used to locate these external ontologies as a locally cached equivalent. An XML catalog describes the mapping between external entity references and locally cached equivalents for an XML external resource. In this case, the external entity references are URIs to external ontologies, and the locally cached equivalent is a pointer to a local copy of the root folder containing the external ontologies. For instance, the following entry from an XML catalog for the Red branch FIBO ontologies tells the Concept Modeler to look for the ontology named "http://spec.edmcouncil.org/fibo/red/be/" in the folder "be/be.rdf."

```
<uri id="Automatically generated entry, Timestamp=1425183124824"
name="http://spec.edmcouncil.org/fibo/red/be/" uri="be/be.rdf"/>
```

The local copy of this ontology is located relative to the root folder of the locally cached equivalent. This root folder must be set in the XML catalog file by modifying the "id" attribute of the element "group" in the XML catalog file:

```
<group id="Folder Repository, directory=file:///C:/Users/Jplatt/Desktop/FIBO-red/,
recursive=true, Auto-Update=true, version=2" prefer="public"
xml:base="file:///C:/Users/Jplatt/Desktop/FIBO-red/">
```

The identical, bolded file URIs above point to the folder "FIBO-red" on the Windows desktop of user "Jplatt." These file URIs must be set to the location of the local folder containing the locally cached equivalent of the external ontologies. Windows users should note the use of forward slashes, as well as the triple forward slash before the Windows file location.

5.5.2 Set the OWL Import Catalog

The OWL import catalog must be set for a MagicDraw project if an XML catalog is used to import an ontology model.

To set the OWL import catalog to the XML catalog file for the desired external ontologies:
1. Click **Options** > **Project**.

Copyright © 2017, No Magic, Inc.
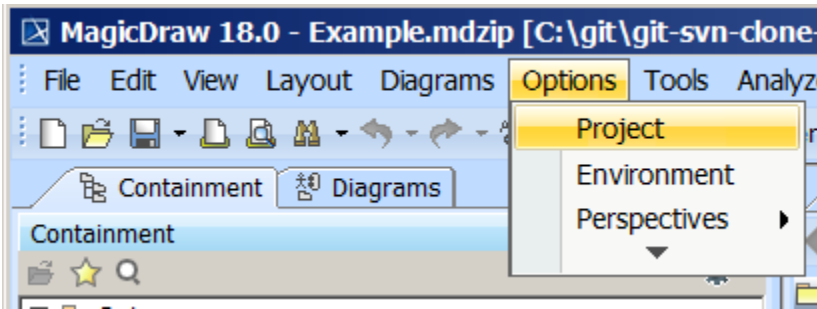
Figure 102   The Concept Modeler's Project Options menu

2. Select **General Project Options**.
3. Click in the field next to **OWL Import Catalog**.
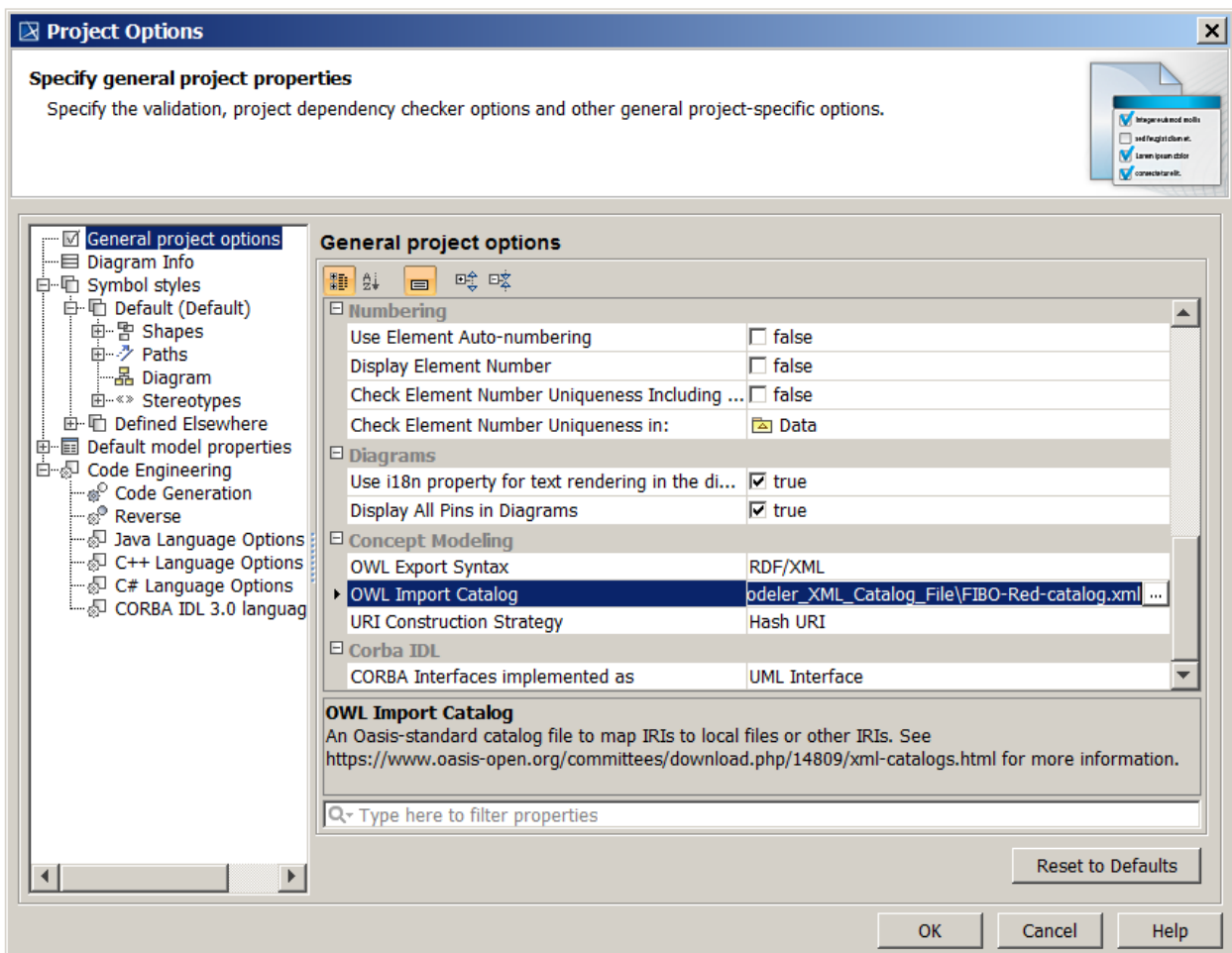4. Click the "…" button.

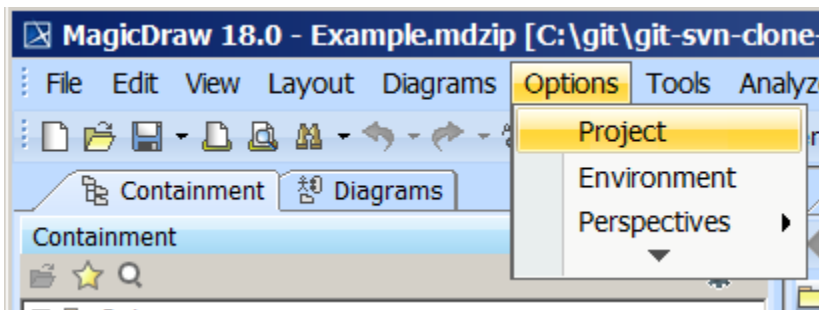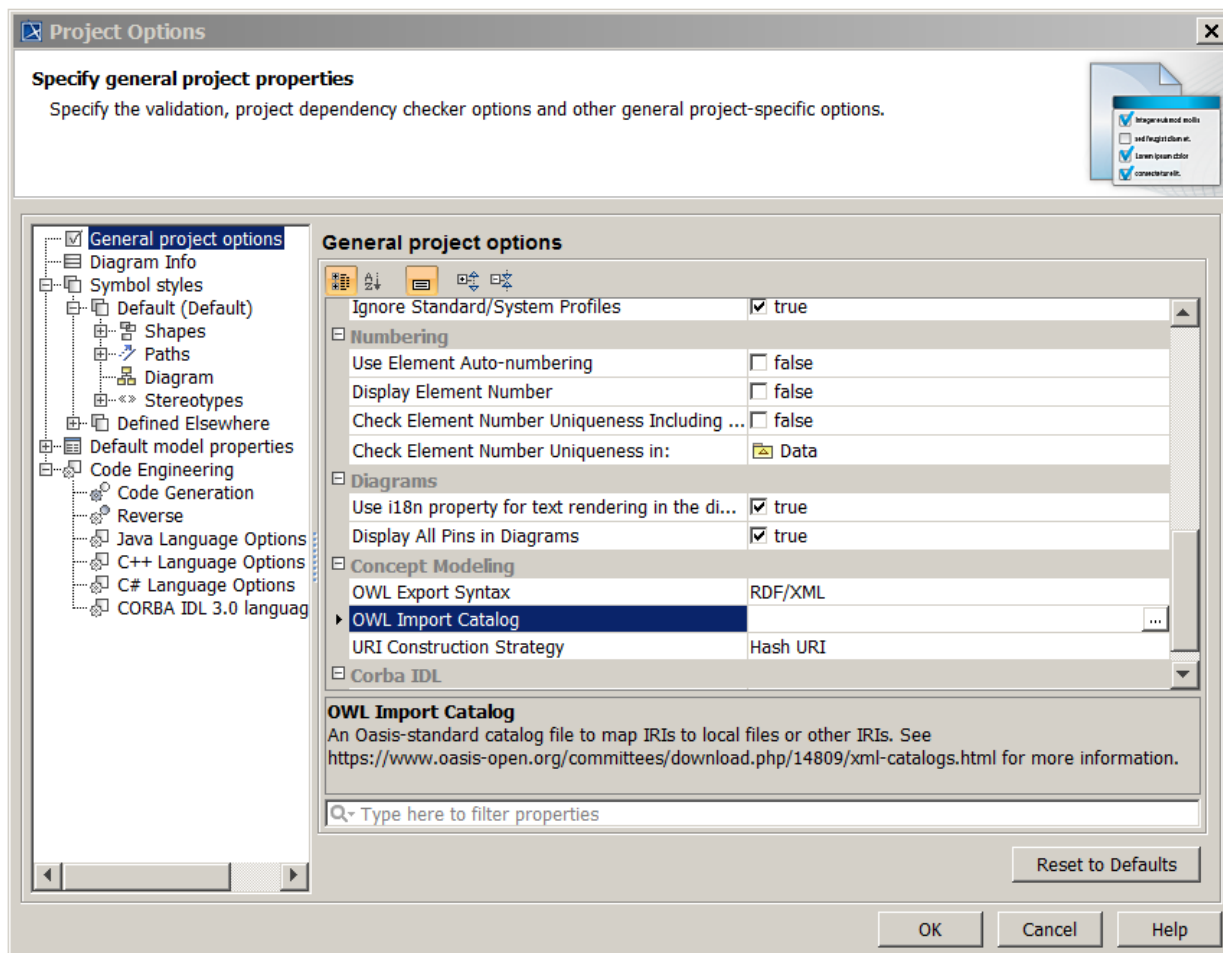
Figure 103   Selecting an XML catalog file as the OWL import catalog

5. Select the XML catalog file.
6. Click **Open**.

7. Click **OK**.

5.5.3 Set a Path Variable to Share OWL Import Catalog Files

By permitting a user to set a path variable to a local directory containing OWL import catalog files, the Concept Modeler allows users to easily share OWL import catalogs and the MagicDraw projects that use them. Without such a variable, each user may have a different path to the file, which causes the users to change the path back and forth. To resolve this issue, a user needs to define a path variable to this local directory on his or her computer that corresponds to the directory containing the same OWL import catalog files on another user's computer.

To define a path variable:

1. Click **Options** > **Environment**.
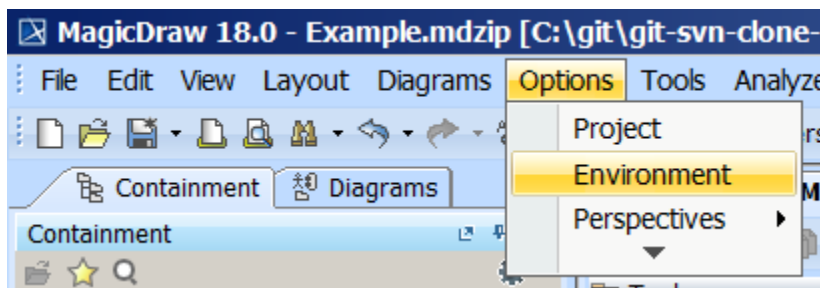
Figure 104   The Concept Modeler's Environment Options menu

2. Select **Path Variables.**
3. Click **Add.**
4. Name the path in the **Name** field.
5. Click the "…" button next to the **Value** text box.
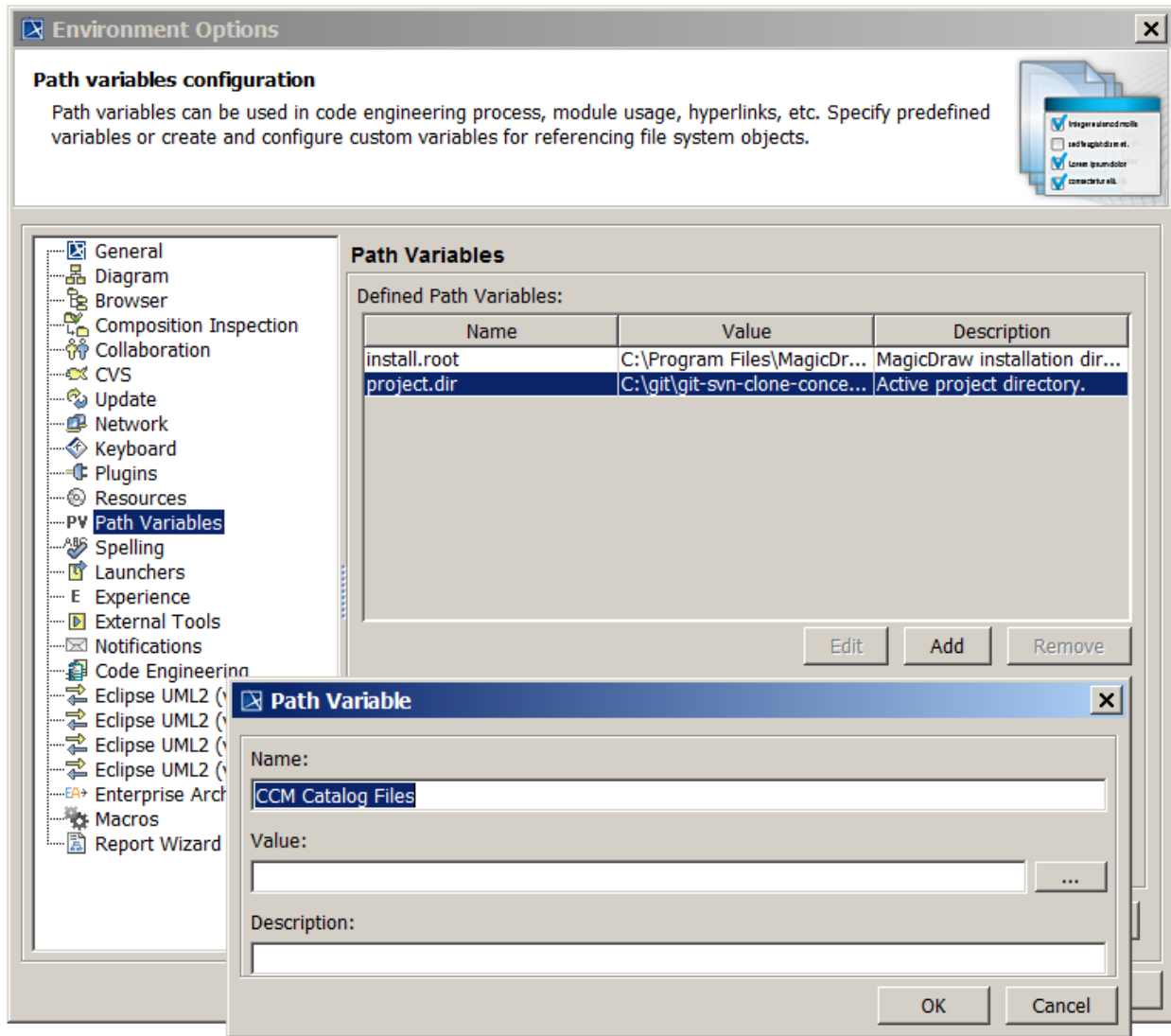
Figure 105   Locating the OWL import catalog

6. Select desired root directory containing OWL import catalog files.
7. Click **Open**.
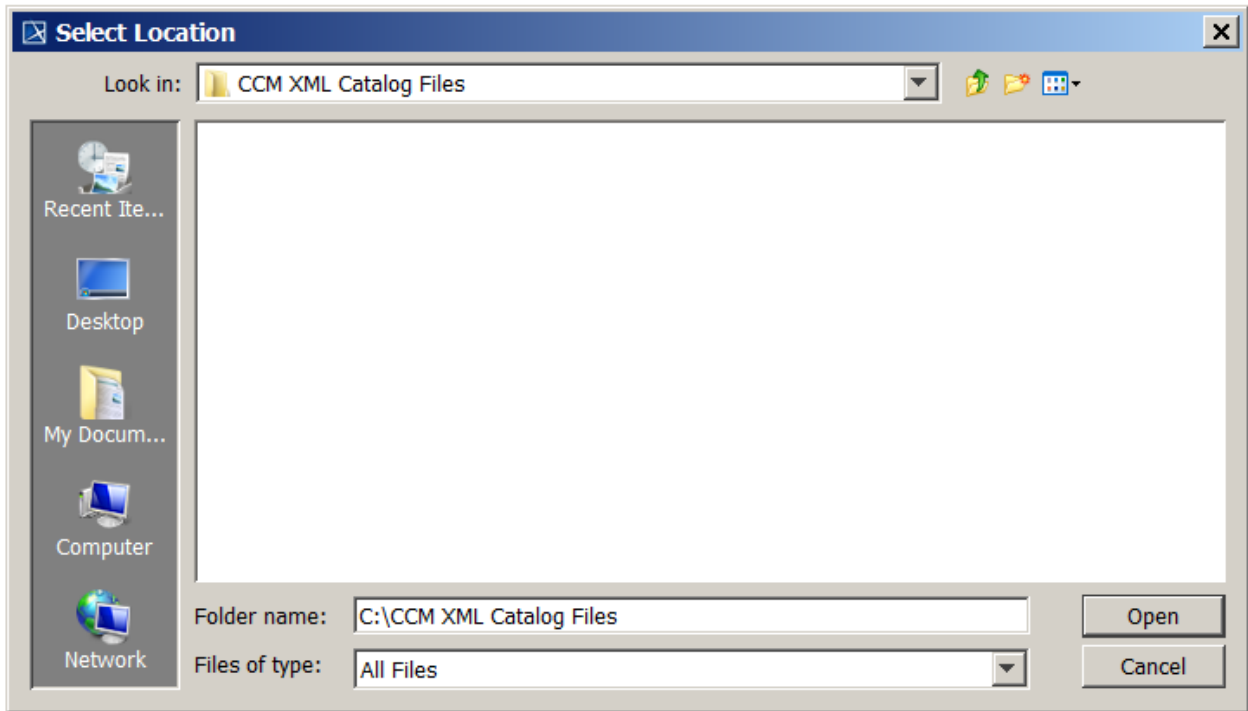
Figure 106   Selecting the OWL import catalog

8.   Click **OK**. You will see the created directory appear on the Path Variables list (see the following figure).

Copyright © 2017, No Magic, Inc.
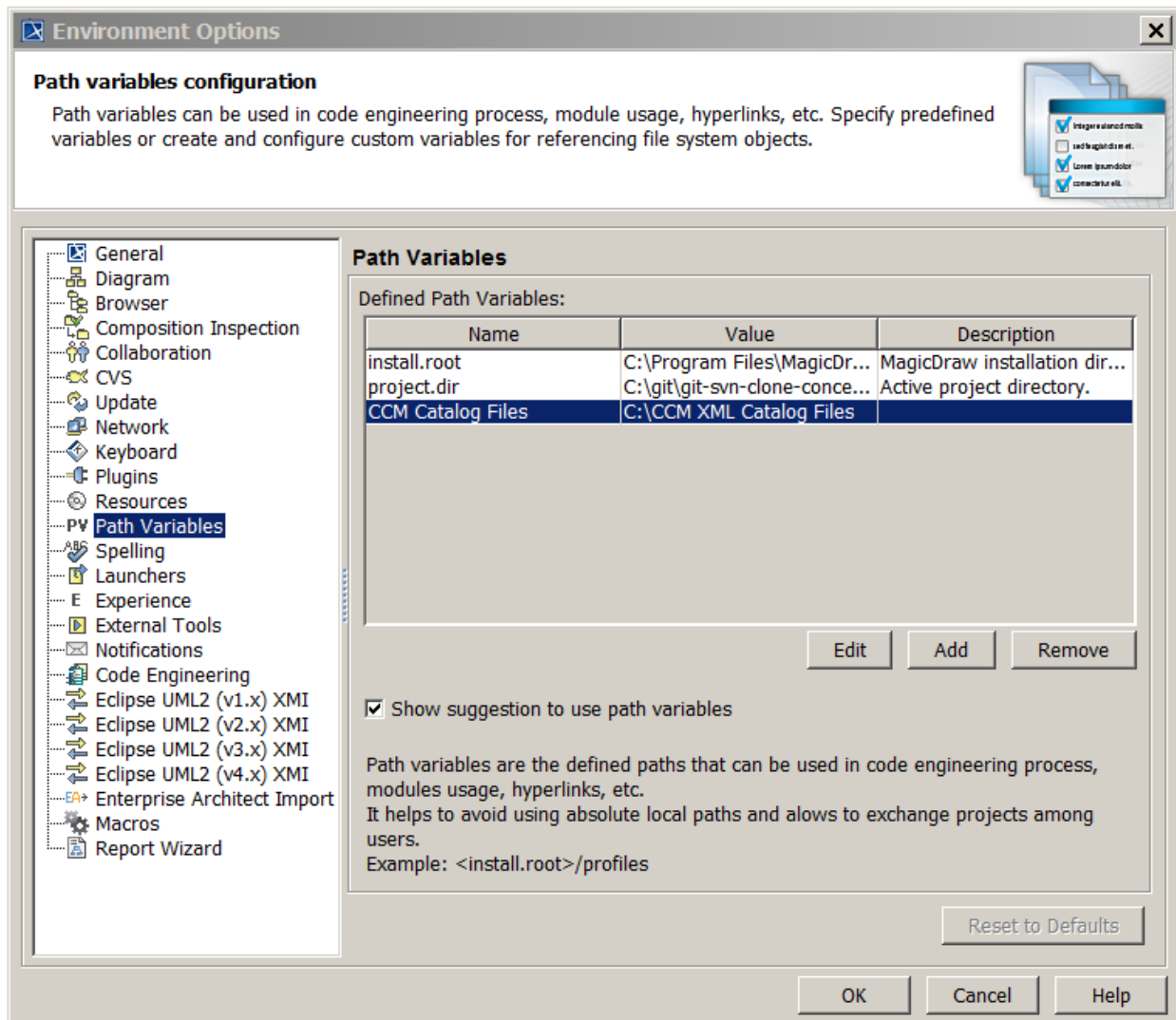
Figure 107 The OWL import catalog is defined as the path variables

8. Click **OK**.

5.5.4 Use a Path Variable to Share OWL Import Catalog Files

To use a path variable to share OWL import catalog files:
1. Click **Options** > **Project**.

Figure 108   Opening the Project Options Dialog

2. Select **General project options.**
3. Click in the field next to **OWL Import Catalog.**
4. Click the "…" button.

Figure 109   Selecting the path variables to use in the Project Options dialog

5. Select the XML catalog file.
6. Click **Open**.
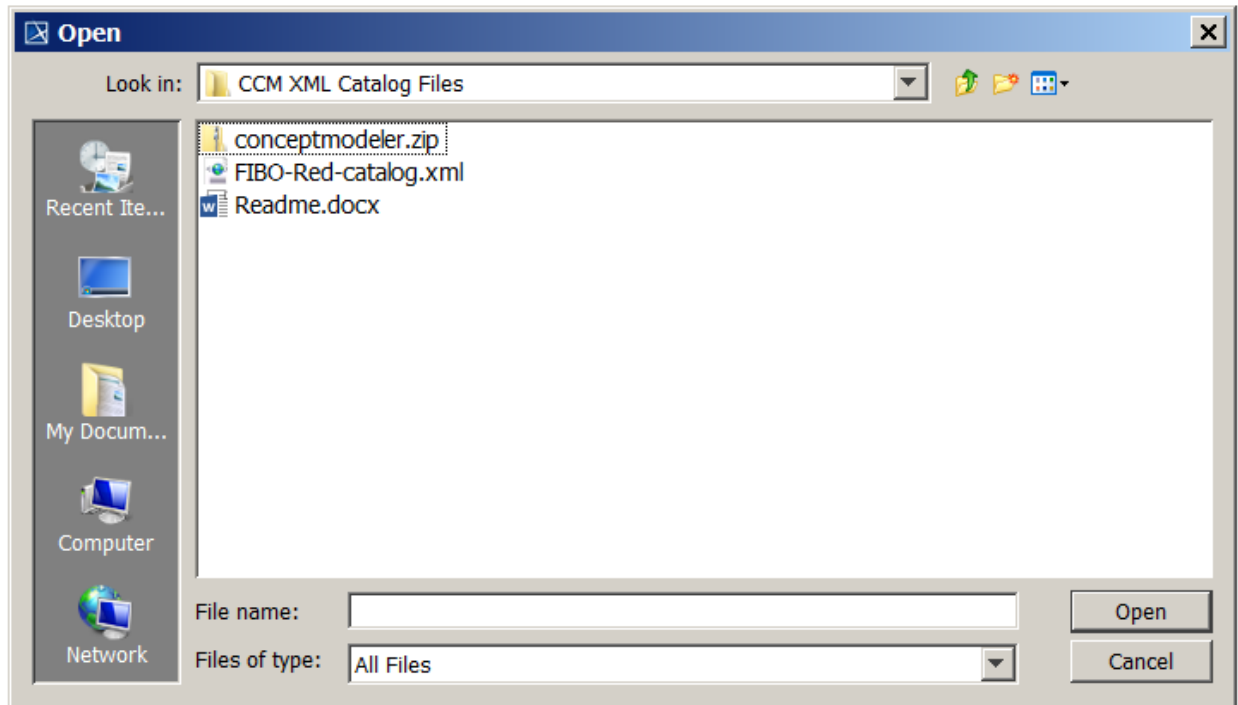
Figure 110   Selecting the XML catalog file

7. Select the path to the OWL import catalog that includes the defined path variable.
8. Click **Use Selected**.
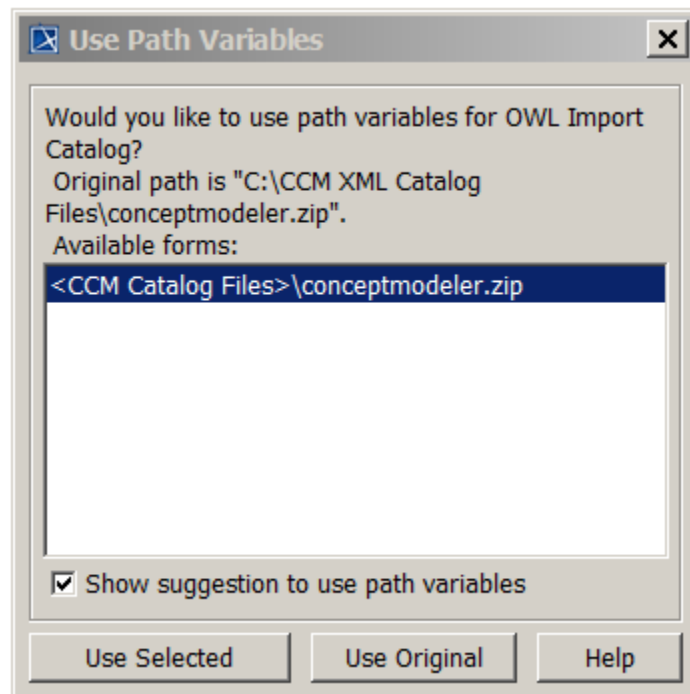
Figure 111   Using the selected path variables

You can import an OWL ontology file (after setting the OWL import catalog, if necessary) and reuse or augment it in the Concept Modeler.

When you import an OWL ontology file, the Concept Modeler preserves the URI/IRI for every OWL class and property and imported it as a tagged value of the corresponding UML class or property. The tagged value, called IRI, is part of a «Resource» stereotype applied to each UML element. This tagged value is generally used only for an imported OWL ontology. It allows you to refer to an OWL ontology from a concept model that has been exported to OWL. The exported concept model directly imports the original OWL ontology file(s) and can use the classes and properties defined there with all the correct URIs/IRIs.

Note	The Concept Modeler enables you to import an OWL ontology, change the package's «Model» stereotype to «Concept Model» and edit the classes or properties within that package, then export it back to OWL. This round-trip OWL ontology editing (OWL ontology to a concept model to OWL ontology) gives priority to an IRI tagged value and therefore, it preserves URIs/IRIs from the original OWL ontology on export. If you need to use the concept model from this point forward as the source model for an OWL ontology, you should probably remove the tagged values so that changing class and property names will keep the URIs/IRIs in sync.

To import an OWL ontology file into a concept model:
1. On the main menu, click **File** > **Import From**.
2. Select **OWL Ontology File**.

Copyright © 2017, No Magic, Inc.
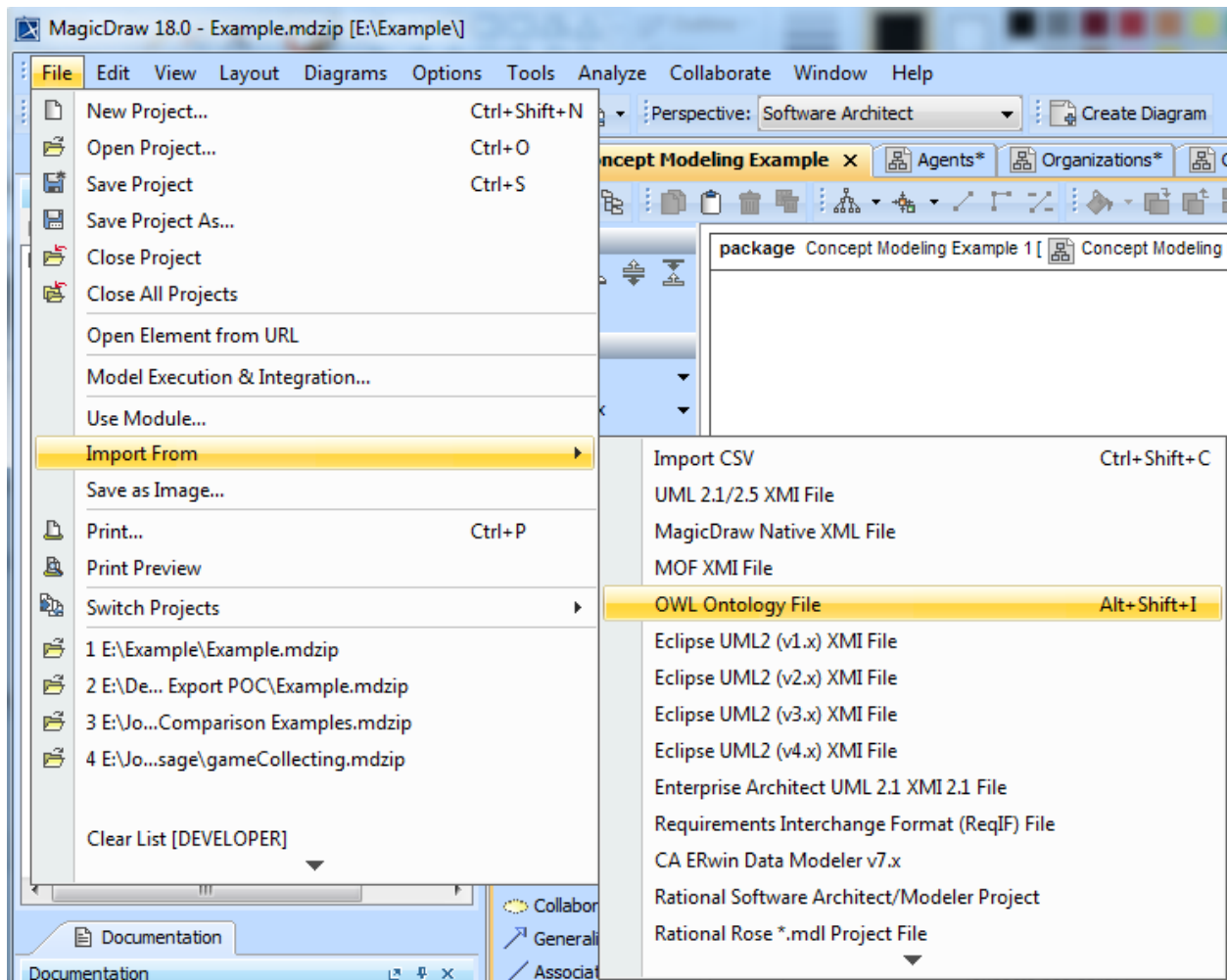
Figure 112   The Concept Modeler's import ontology menu

3. Select an ontology file.
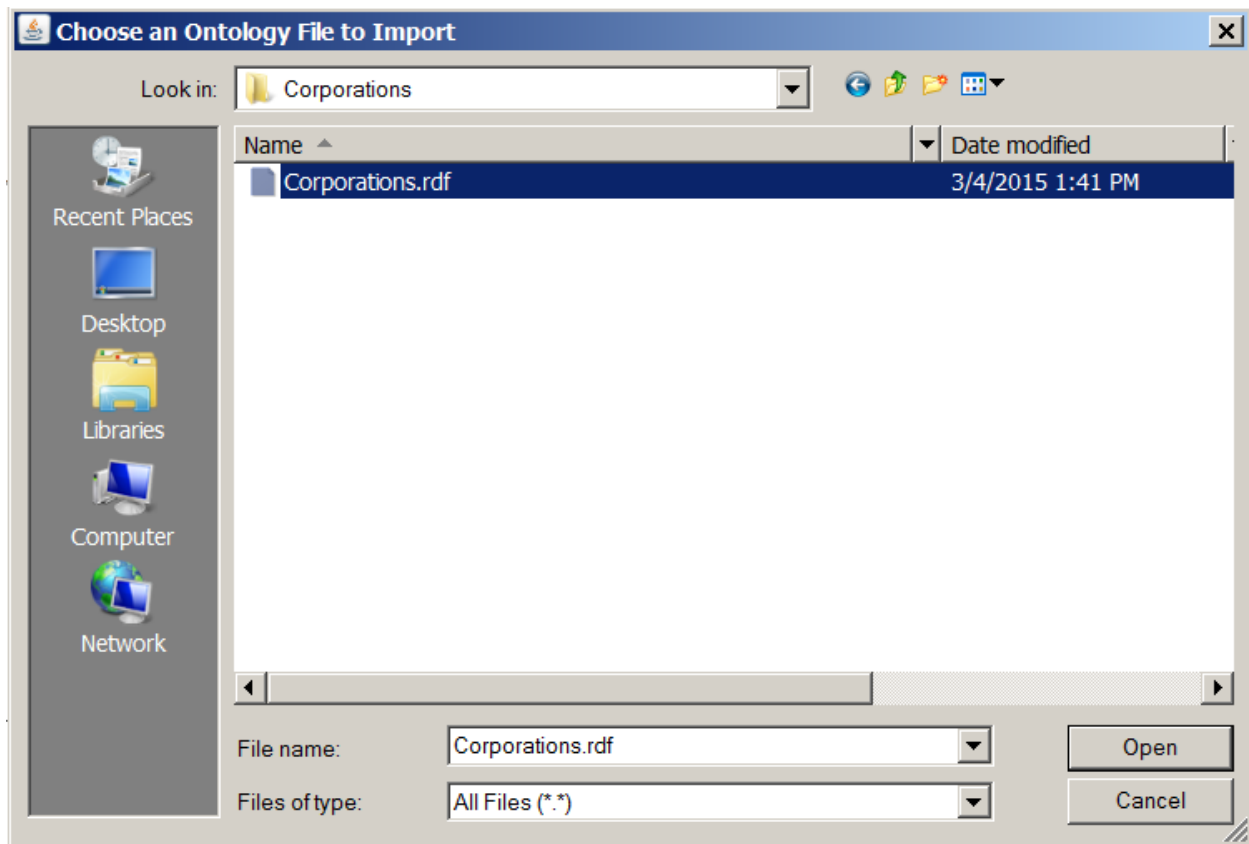4. Click **Open** (see the following figure).

Figure 113   Selecting the ontology file to import

If the OWL import catalog is set using a path variable (as described in section 5.5.4 Use a Path Variable to Share OWL Import Catalog Files), and this path variable is not defined (as described in section 5.5.3 Set a Path Variable to Share OWL Import Catalog Files), the Concept Modeler will not be able to locate it. Consequently, the following dialog box will be displayed:
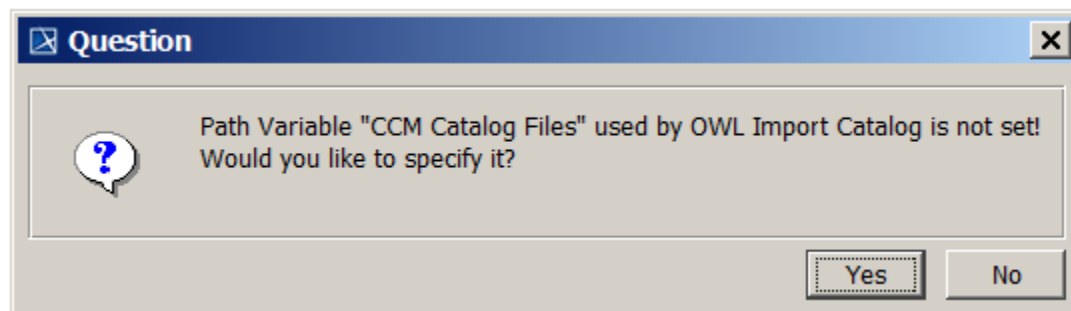
Figure 114   A dialog prompting you to specify a path variable

Clicking **Yes** will allow the user to set the path variable to a root directory containing OWL import catalog files, and the import of the OWL ontology to proceed.

Copyright © 2017, No Magic, Inc.
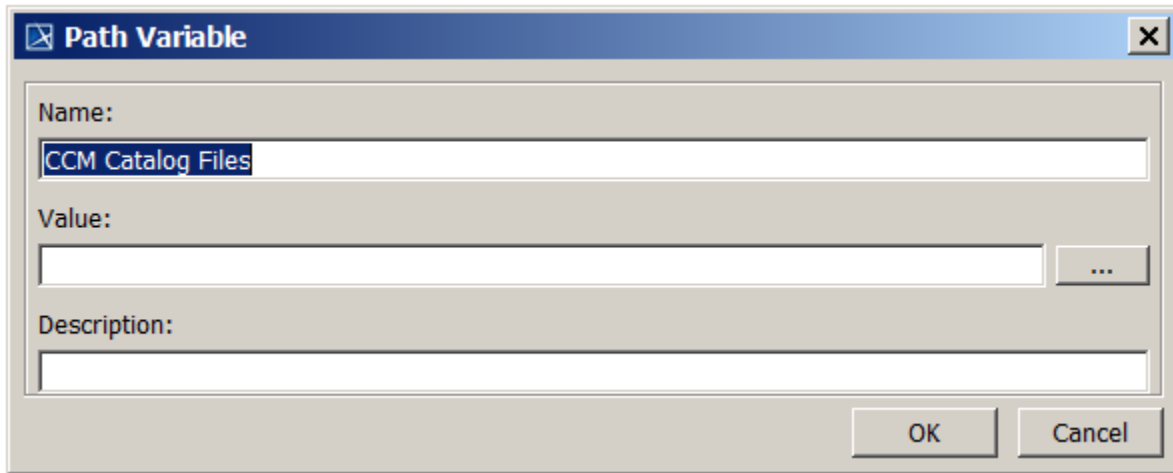
Figure 115 Setting the path variable to the OWL import catalog files

Once an OWL file has been successfully imported, an **Imported Ontologies** package will appear in the Containment tree window containing the imported OWL data.
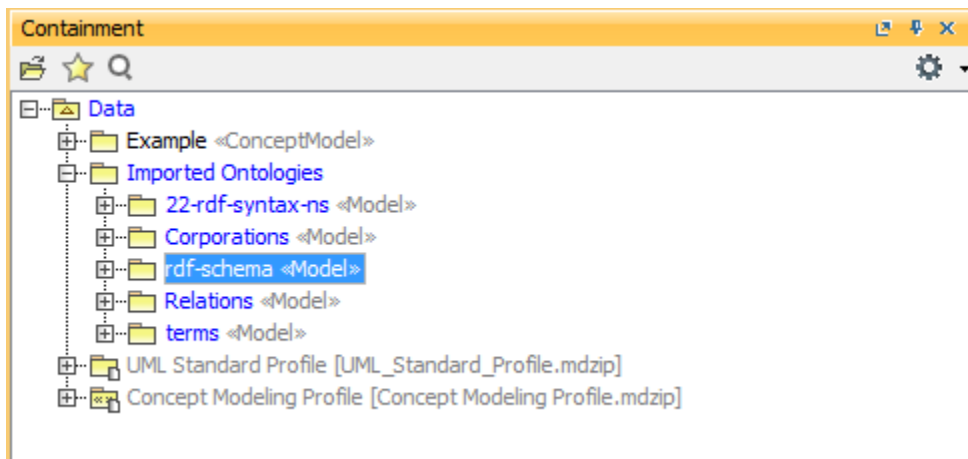
Figure 116 The imported ontology package appears in the Containment tree

Note	The Concept Modeler supports importing classes, properties and packages that have the same label but different URIs.

5.5.6   Import annotations on an OWL Ontology to a concept model

An OWL ontology may have one or more annotations added to itself. Concept Modeler can import the annotations as annotations on a concept model. If there is only one annotation on the OWL ontology, it will show u p in the concept model's Documentation pane in MagicDraw upon import. When there are more than one annotations imported, they will not show up automatically

Copyright © 2017, No Magic, Inc.

in the Documentation pane. You need to specify which one gets to be displayed as the package's default documentation by selecting it from the **Preferred annotation property** option in the **Project Options** dialog. The rest of the annotations will become the package owned comments (UML comments tagged as Annotation).

5.5.7   Version IRI

An ontology can have multiple published versions. To identify various version separately OWL provides a mechanism to specify "version IRI". The version IRI may be, but need not be, equal to the ontology IRI. For instance, an ontology document of an ontology that contains an ontology IRI <http://www.example.com/my>, a version IRI would look like <http://www.example.com/my/2.0>
Owl version IRI is specified as follows:
<owl:Ontology rdf:about="http://www.example.com/my">
     <owl:versionIRI rdf:resource="http://www.example.com/my/2.0"/>
</owl:Ontology>
The version IRI of an ontology will show up under corresponding concept model tag 'versionIRI'.

5.5.8   Display and Hide IRI

The IRIs of classes and properties may not be visible in the diagram pane. You can display them by using the shortcut menu **Display IRI tagged value**. To hide the IRIs from the diagram pane, you can select the shortcut menu **Hide IRI tagged value**.

To display or hide the IRI tagged value of a class or an association end in the diagram pane:
1.   Right-click a class or an association end in the diagram pane.
2.   Select **Concept Modeling** and select either **Display IRI tagged value** or **Hide IRI tagged value**.
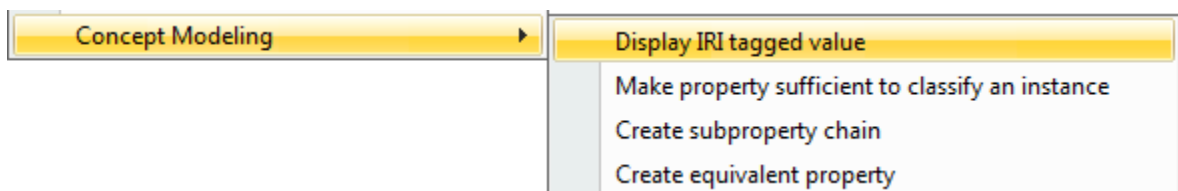
Figure 117   The Display IRI tagged value shorcut menu

Copyright © 2017, No Magic, Inc.

Figure 118   The Hide IRI tagged value shortcut menu

<u>To display or hide the IRI tagged values of all classes or association ends in the diagram pane:</u>

1. On the main menu, click **Edit** > **Select All**. All of the elements in the diagram pane will be selected.
2. Right-click on any element.
3. Select **Concept Modeling** and select either **Display IRI tagged value** or **Hide IRI tagged value**.

Note	• The menu **Display IRI tagged value** will appear when you right-click an element whose IRI is hidden from the diagram pane. • The menu **Hide IRI tagged value** will appear when you right-click an element whose IRI is displayed in the diagram pane.

5.6 Export a Concept Model to an OWL Ontology

5.6.1 Set the Concept Model Export Syntax

The Concept Modeler provides many syntaxes (see section 4. UML to Equivalent OWL (in OWL Functional Syntax)) that you can select to export your concept model project to an OWL ontology.

If you export your model without selecting a syntax, the Concept Modeler will export it using RDF/XML, which is the default syntax.

To set the syntax with which to export a concept model for a MagicDraw project:
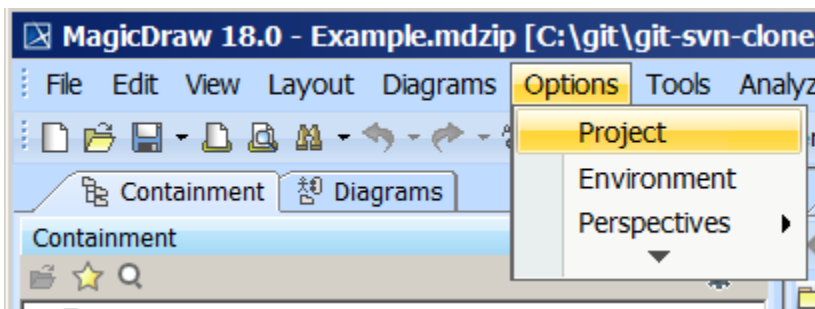 1. Click **Options** > **Project**.
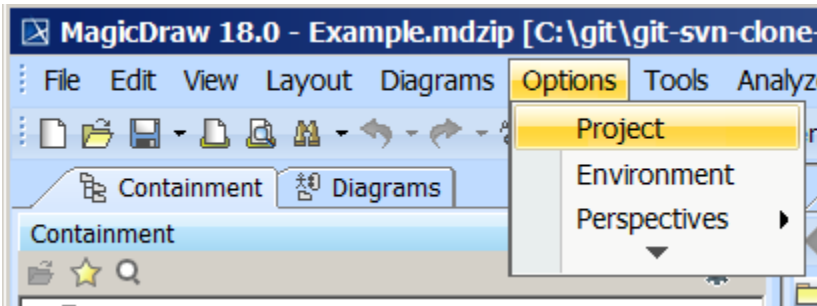
Figure 119   The Concept Modeler's project options menu

  2. Select **General project options**.
  3. Click in the field next to **OWL Export Syntax.**
  4. Select a syntax to export the concept model (see the following figure).

Copyright © 2017, No Magic, Inc.
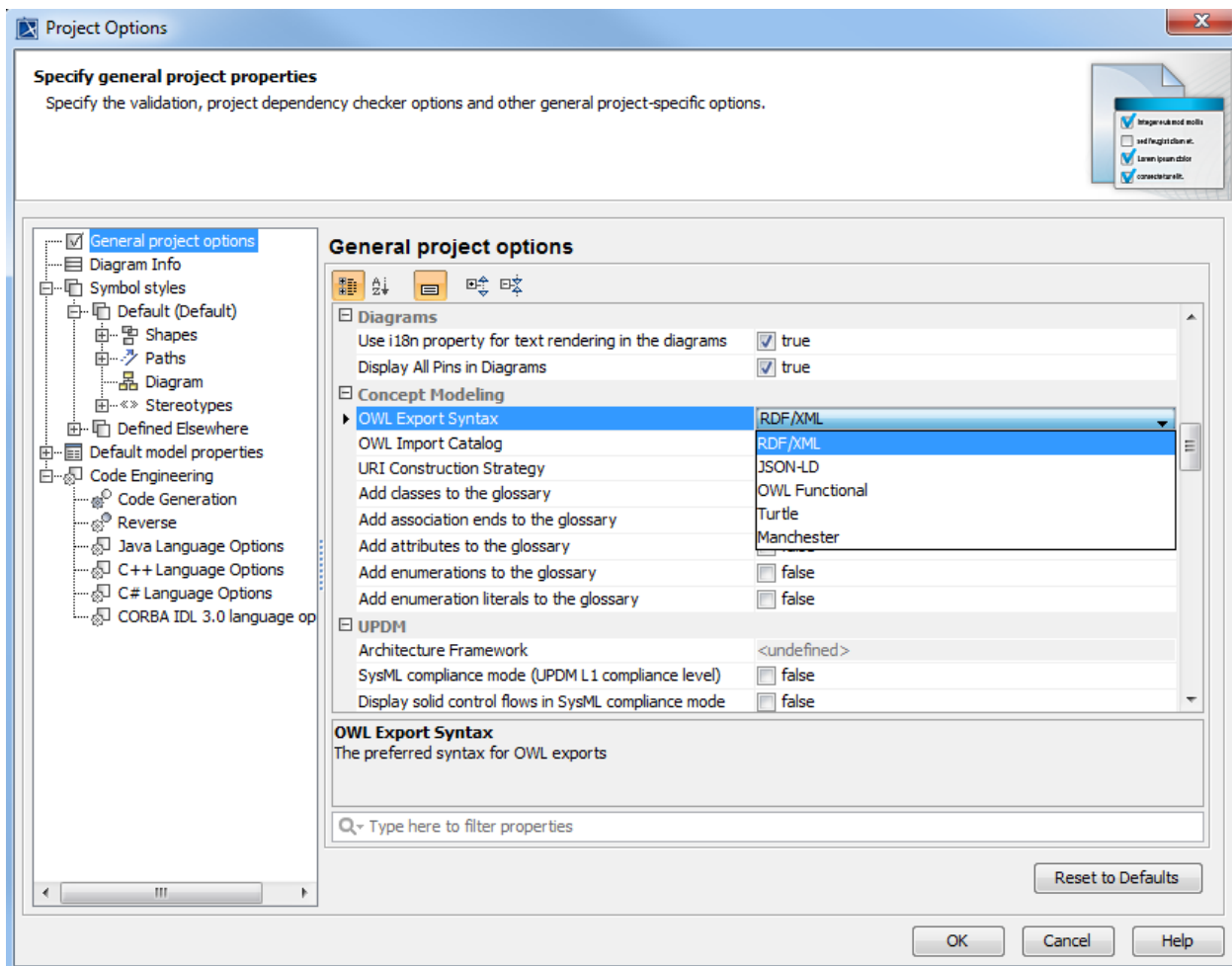
Figure 120 The OWL export syntax options

5.6.2   Set the Concept Model Export URI Style

The Web uses Uniform Resource Identifiers (URIs) as a global identification system. A URI is used to identify a resource, such as a document or an abstract thing, either by a location, such as a DNS host name and a path on that machine, or a name.

When identifying real-world objects using a URI, you can choose between (i) Hash URI and (ii) 303 URI. The differences are as follows:

(i)   Hash URI: for smaller and stable sets of resources that evolve together, for example, RDF Schema vocabularies and OWL ontologies. The advantage is that all resources are in the same file because the redirection target cannot be configured separately for each resource.

(ii)   303 URI: for large-scale data sets that are likely to grow over time. One document can be used for describing either each or all resources. When using 303 URI for an ontology, it can reduce a client's performance and cause higher latency.

Copyright © 2017, No Magic, Inc.

Note	The Concept Modeler imports and preserves the URI or IRI for every OWL class and property from an OWL ontology file as the tagged value on the corresponding UML class and property in the Concept Modeler. When exporting this particular model back to OWL, the Concept Modeler will not apply the normal automatic URI/IRI generation and preserve the URI or IRI from the original OWL ontology so that the classes and properties can be used with their correct URIs/IRIs. The export URI style option also has no effect on the preserved URI or IRI.

To select a concept model export URI style for a MagicDraw project:

1. Click **Options** > **Project**.

Figure 121 The Concept Modeler's project options

2. Select **General project options**.
3. Click in the field next to **URI Construction Strategy**.
4. Select either **Hash URI** or **303 URI**.

Copyright © 2017, No Magic, Inc.
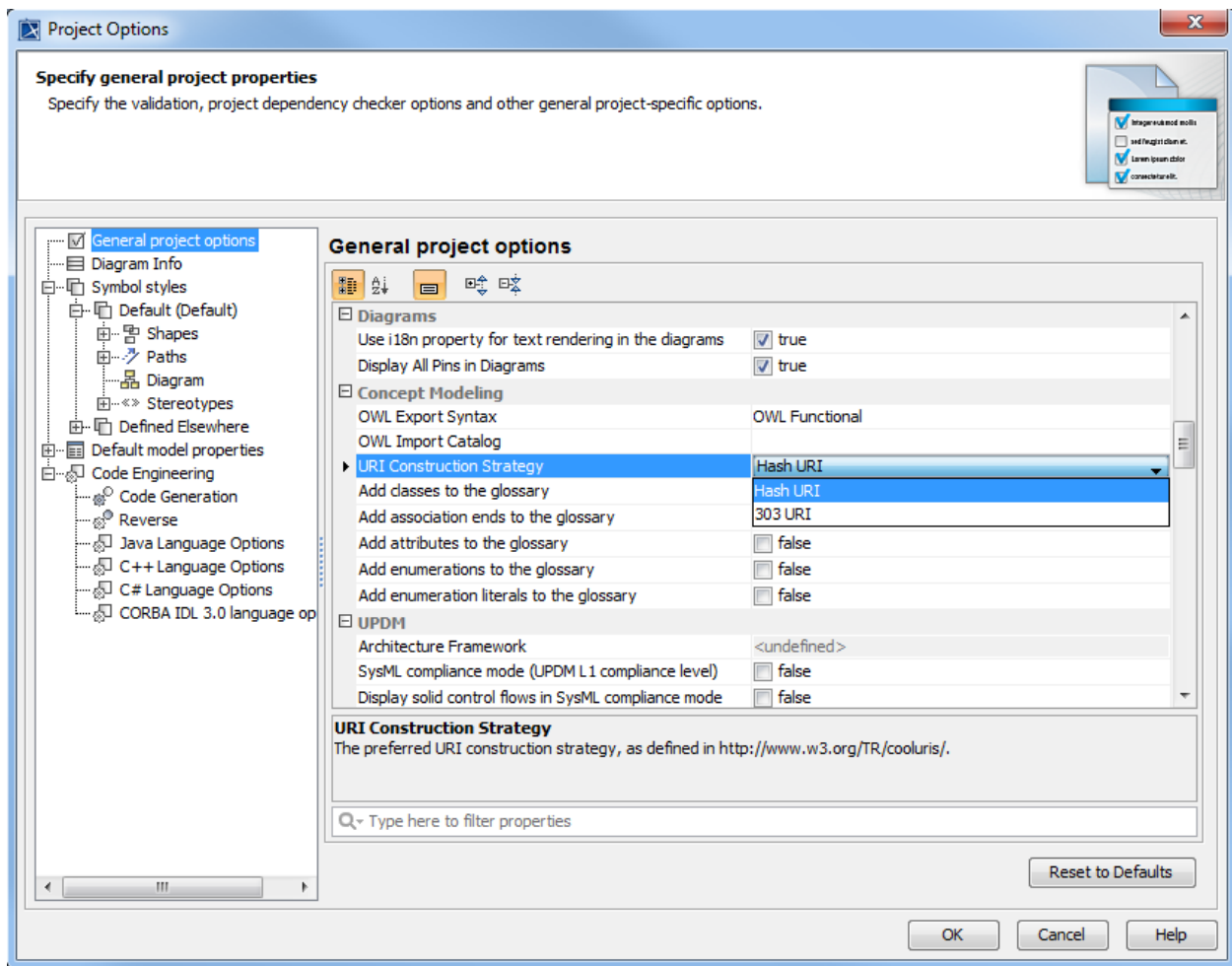
Figure 122   Selecting a URI construction strategy

5.6.3   OWL Export Folder

Every time you export a concept model to an OWL ontology, it will go into a default export location. The default location is an OWL directory, which is created automatically next to the project file. However, you can disable this default directory and enable the option that will allow you to choose your desired destination folder whenever you export a concept model.

Selecting **Always prompt for a file destination when exporting OWL** in the **Project Options** dialog allows the Concept Modeler to prompt for an export directory to store your «Concept Model» every time you export one.

- If you have previously selected an export location, a window will open showing the file path of the former saved location.
- If the previously saved export location is invalid, a message will show in the notification window, the previously saved location will clear, and a prompt for a file destination will open to a default location.

Copyright © 2017, No Magic, Inc.

- If the previously saved export location no longer exists for any reason, the Concept Modeler will revert to the default location and prompt for file destination.

Turning off this project option (default) allows the Concept Modeler to automatically remember and select the last export location for your «Concept Model» without asking you first.

- If the export location no longer exists, it will export your file to the default OWL directory.
- If there is a valid saved export location, the Concept Modeler will export to the saved export location.
- If the saved export location is not valid and the default location exists, then the Concept Modeler will revert to the default location and display an error message; otherwise, it will create a new OWL folder and revert the saved export location back to the default location.
- If there is no saved export location and the default location exists, then the Concept Modeler will set the export location to the default location; otherwise, it will create a new OWL folder at the project location and set the export location to the default location.

When you would like to select multiple packages, you must right-click on one of the packages to export. You will always receive a prompt for file destination which is applied to all of your packages and the same file location will be loaded to all the packages. You are able to select different file destinations for each package, but the software always loads the same starting location for each package.

The error that appears in the Notification Window when trying to export the concept model to an OWL ontology can be caused by entering an incorrect path name or the path name to the previous export location does not exist anymore. If you encounter this type of error, you need to open the **fileExportPath** tagged value in the model's **Specification** dialog and correct the path name, or select another location (Figure 7).

To prompt for a dialog that allows you to select a destination folder on export:
1. Click **Options** > **Project** on the main menu. The **Project Options** dialog will open (see the following figure).
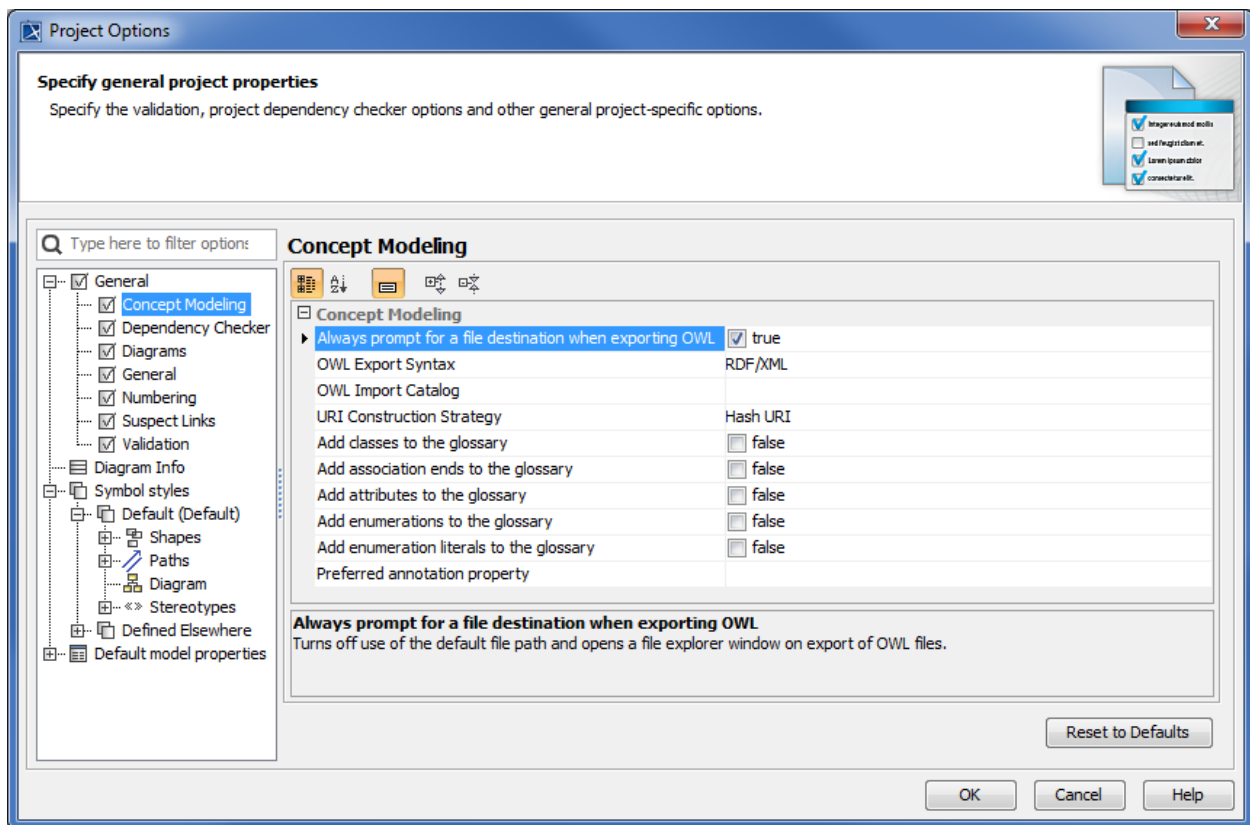
Figure 123   The OWL export destination folder option

2. Select **General** > **Concept Modeling**.
3. Select the check box **Always prompt for a file destination when exporting OWL**.
4. Click **OK**.

5.6.4 Export a Concept Model to OWL

Before exporting a model to an OWL ontology, you can specify the file export path in the Specification window. The **fileExportPath** property allows you to store the file export path as a tagged value in the Concept Modeling package (see the following figure).
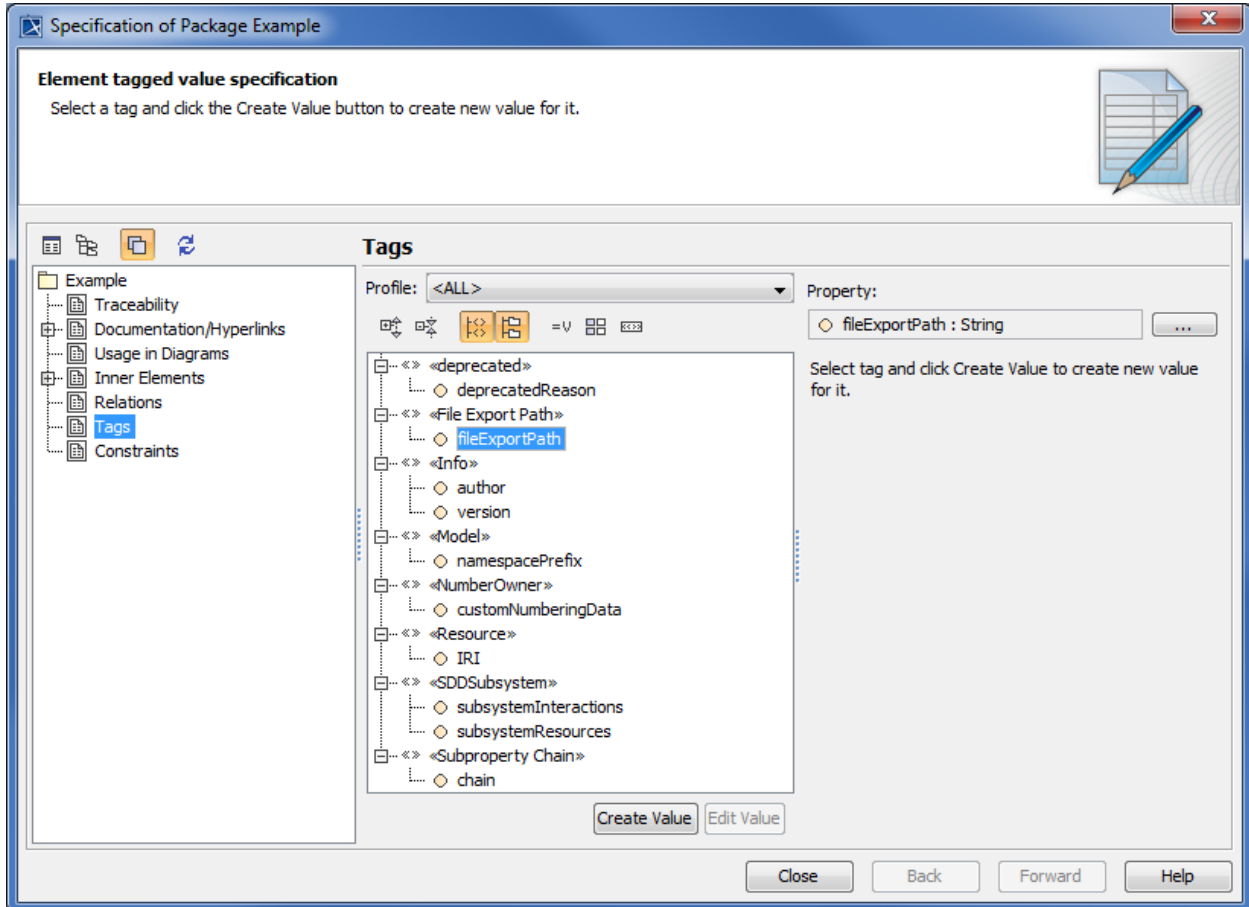

Figure 124   Specifying the file export path before exporting a concept model to an OWL ontology

To export a concept model to an OWL ontology:
1. Right-click on a concept model in the Containment tree.
2. Select **Concept Modeling.**
3. Select **Export Concept Model to OWL**.
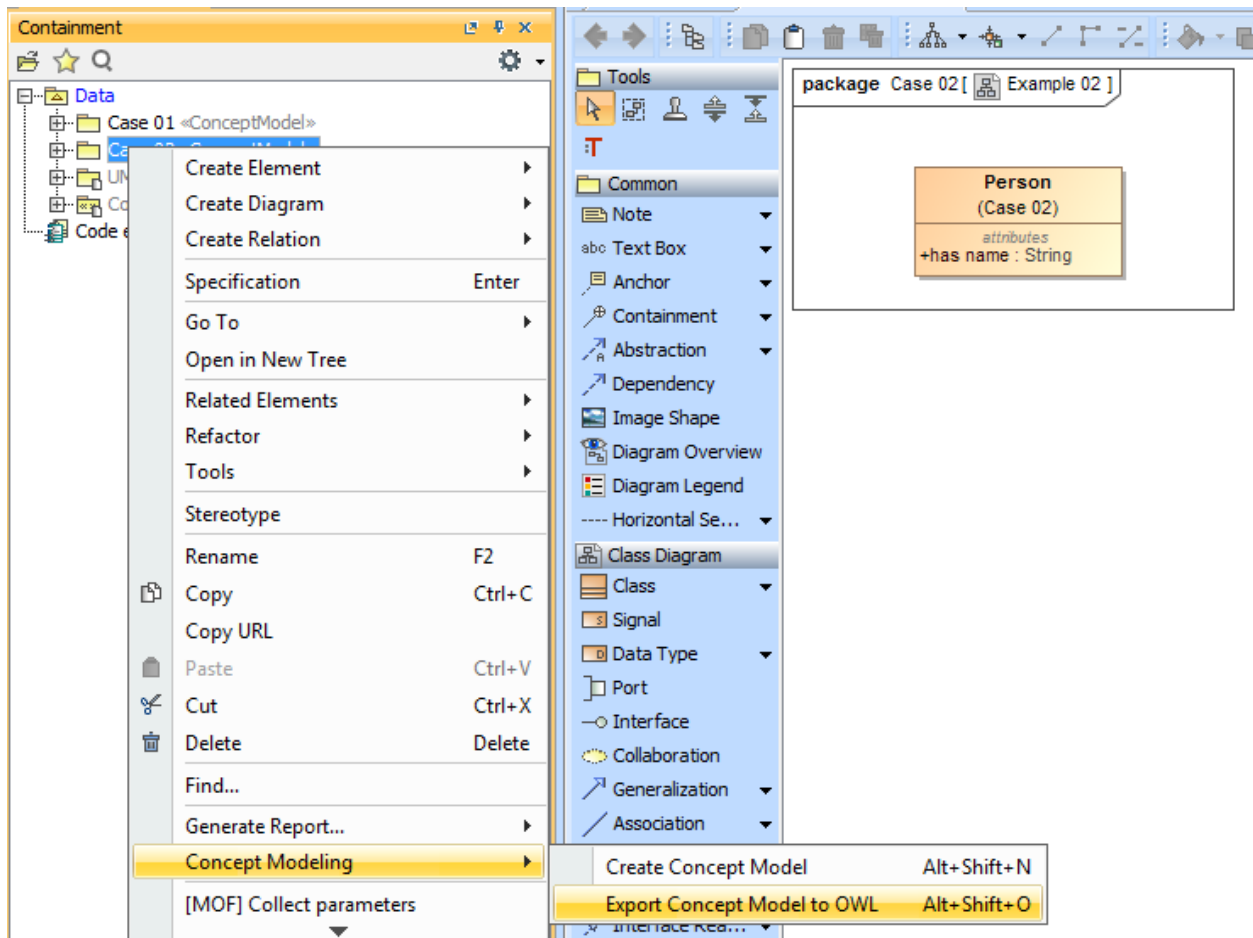
Figure 125   The Export Concept Model to OWL menu

5.6.5   Use Path Variables to Export a Concept Model to an OWL Ontology

When you export a «Concept Model» stereotyped package, you must have the 'Always prompt for a file destination when exporting OWL' set to true. Please refer to section 5.6.3 to see how to enable this option. This new support allows for users to collaborate their projects with other team members without having to keep in mind the exact destination of the file.

To use Path Variables to export the concept model to OWL:

1. Find your Concept Model package you wish to export.
2. In your Containment Tree, right click on that package and select **Concept Modeling** and then **Export Concept Model to OWL**.
3. Once clicking those, you should be prompted to select a folder from your directory. Select the desired the location.
4. The Use Path Variables should pop up next.
   a. Note: This popup will appear if and only if the selected destination folder has the same path destination that is defined for path variable.

Copyright © 2017, No Magic, Inc.

Would you like to use path variables?
Original path is "/Applications/CCM".
Available forms:

<CCMtest>

☑ Show suggestion to use path variables

Use Path Variables

Use Selected   Use Original   Help
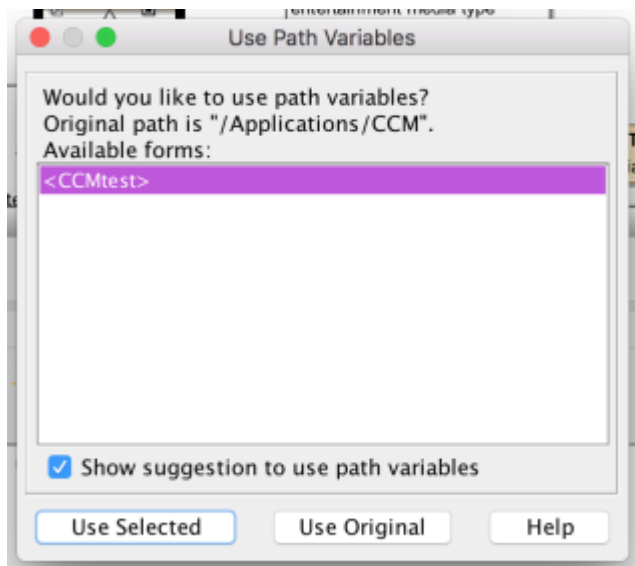
5. Now, you may do one of the following:
   a. Select **Use Selected** to show the form highlighted in purple.
   b. Select **Use Original** to show the path shown in quotes right above the highlighted portion.
6. After clicking the button of your choice, you should have the file generated, exported, and saved inside the directory path described.

5.7   Add a Concept Model to Teamwork Cloud and Export it as an OWL Ontology

You can collaborate with your team members in constructing a concept model. The collaboration feature in MagicDraw allows you to add a concept model to the TWCloud server so that everyone on the team can access the shared model, make changes to it, and commit them to the server. You can easily update the model every time someone commits the changes to the server.

Once your model or project has been added to TWCloud, you can start a collaborative session. You can save the model locally so that you can continue working on the model even though you are not connected to the server (offline mode). This offline mode feature allows you to commit the changes the next time you are online and connected to the server. (For more information about offline projects, see http://docs.nomagic.com/display/MD184/Offline+modeling.)

Exporting either a concept model on your local machine or the one in TWCloud to an OWL ontology works the same way. The Concept Modeler allows you to export it to the default location, a previous location, or a selected destination folder. Prior to exporting the model, you can enable the prompt to export OWL to a selected destination so that you can select a desired location every time you export an OWL ontology.

5.7.1 Add a Concept Model to Teamwork Cloud

Teamwork Cloud (or TWCloud) is a new generation of server that is designed to work with large amounts of data. TWCloud provides a modeling repository standard that can be

transparently scaled from a single workstation to hundreds of servers. It enables multiple servers to interconnect and share resources (see Teamwork Cloud Documentation for more information about TWCloud).

Before adding the concept model to the TWCloud server, you must first log into the server. When adding the concept model to the server, you need to select a category for the model, because TWCloud groups projects into categories. If you do not select any category, your model will be stored under the **Uncategorized** category by default. You can later move your model to another category using Teamwork Cloud Admin (TWAdmin), which is the user interface of TWCloud (for more information about moving a project category, see http://docs.nomagic.com/display/TWCloud184/Moving+projects+from+one+category+to+another).

To log into the TWCloud server:
1. Click the main menu **Collaborate** > **Login**. The **Login** dialog will open.
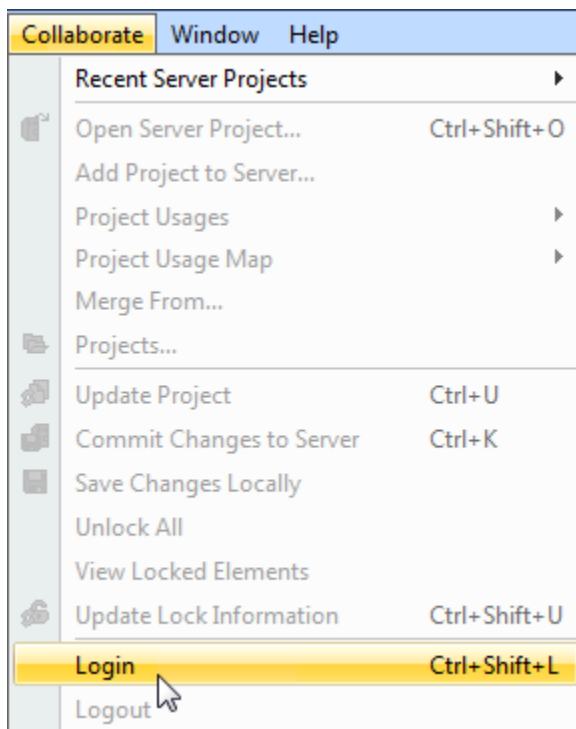
Figure 126   Logging into the Teamwork Cloud server.

Copyright © 2017, No Magic, Inc.
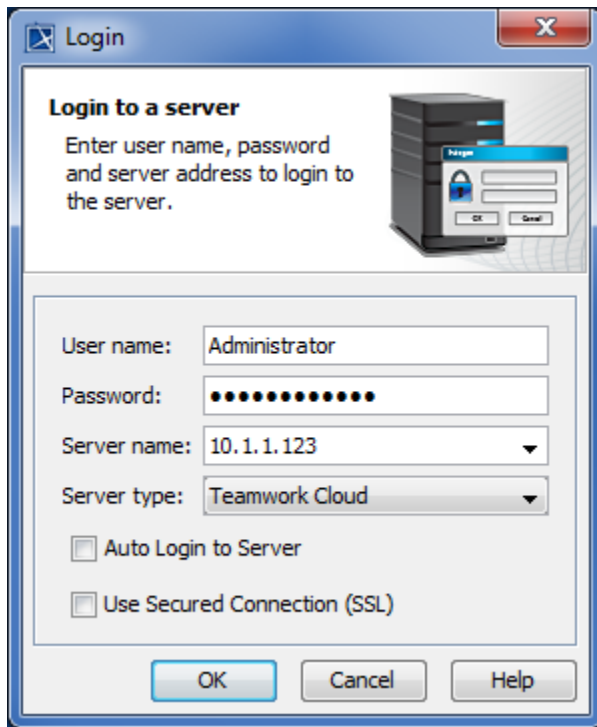
Figure 127   The Login to the Teamwork Cloud server dialog.

2. Type your username and password, for example, Administrator.
3. Enter the server address, for example, 10.1.1.123.
4. Select **Teamwork Cloud** as the server type.
5. Click **OK**. You will be connected to the server.

Once you are logged into TWCloud, you can add a concept model to the server. The following instructions use the concept model **StereotypeDisjointSample** as an example.

To add a concept model to the TWCloud server:
1. Open a concept model project.
2. Click the main menu **Collaborate** > **Add Project to Server**. The **Add Project to Server** dialog will open.

Copyright © 2017, No Magic, Inc.
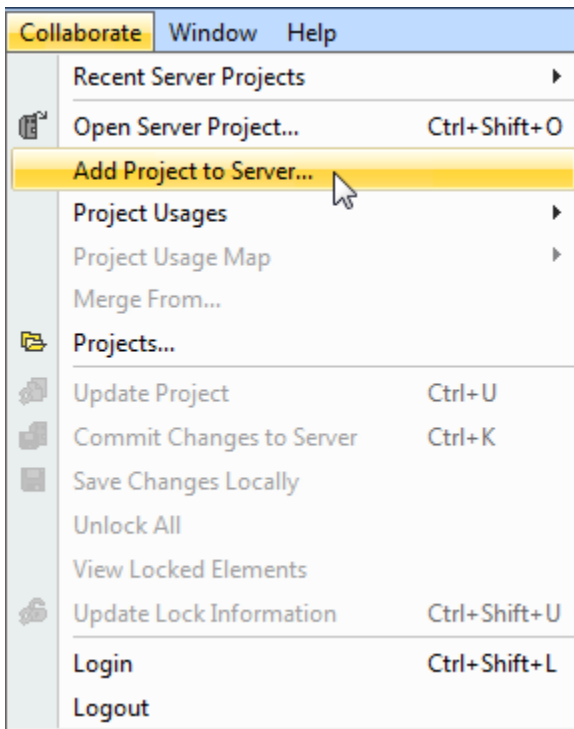
Figure 128   The Add Project to Server menu allows project export to Teamwork Cloud.
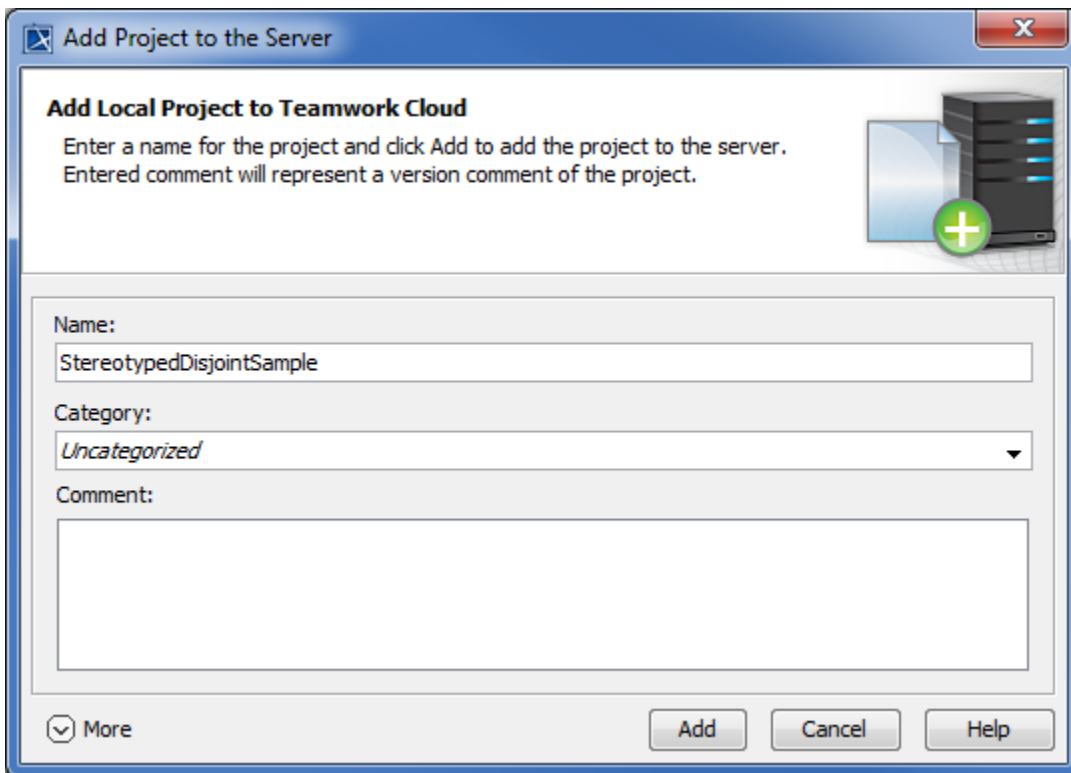
Figure 129   Exporting a concept model to Teamwork Cloud.

The name of the active concept model that you are going to add to the server will appear in the dialog by default. The concept model **StereotypeDisjointSample** is used in this example.

3. Select a category for your concept model. The default option is **Uncategorized**.
4. Click **Add**. The concept model will be added to the server.

To check if the concept model has been successfully added to TWCloud:
- Click the main menu **Collaborate** > **Projects**. The concept model that was added to the server will appear in the **Manage Projects** dialog. The concept model used in this example is **StereotypeDisjointSample**.
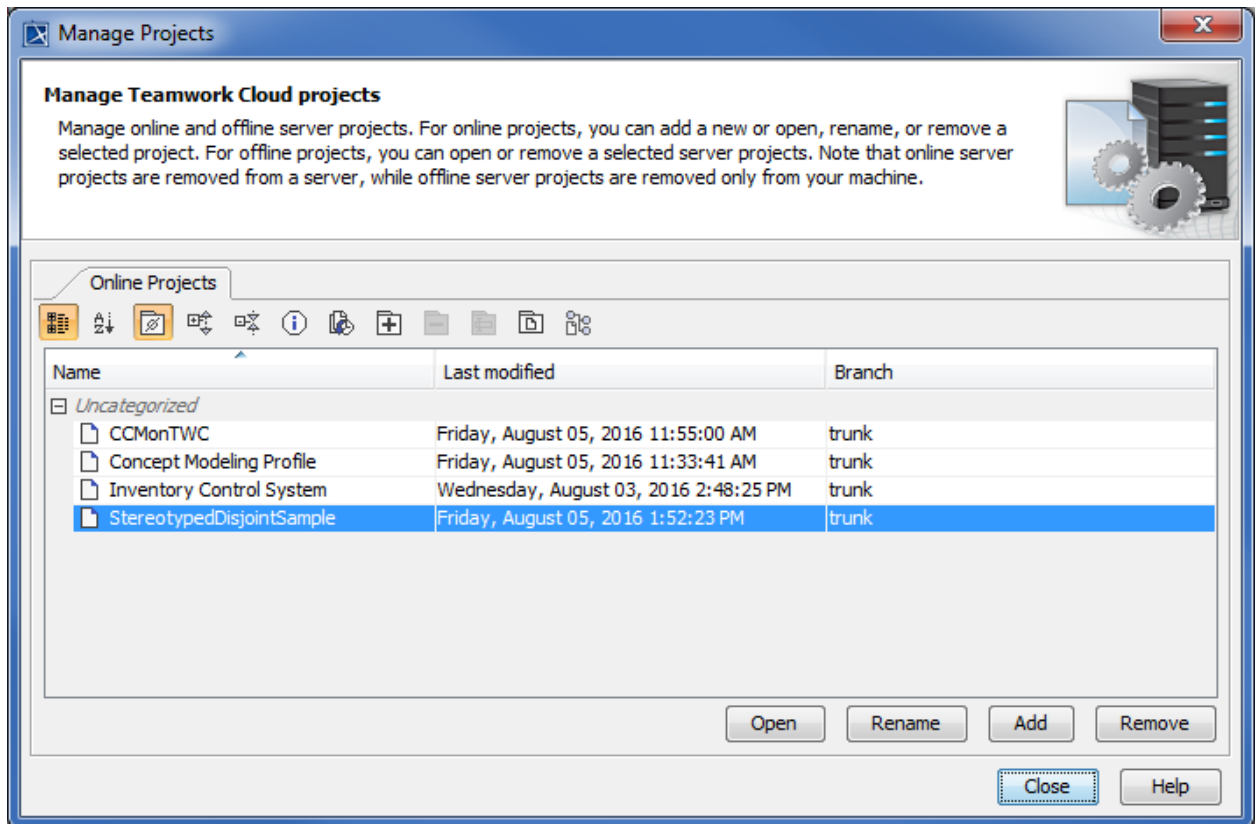
Figure 130   The Manage Projects dialog showing the exported concept model in Teamwork Cloud.

A concept model that has been added to TWCloud is saved on the server as a server project. Users with the permission to work on server projects can access the concept model. Before editing the concept model, you first need to lock it to prevent others from editing it at the same time. (For more information on locking models for editing, see http://docs.nomagic.com/display/MD184/Locking+model+for+edit.)

To open a concept model in TWCloud

Copyright © 2017, No Magic, Inc.

1. Log into the TWCloud server.
2. On the MagicDraw main menu, Click **Collaborate** > **Projects** to open the **Manage Projects** dialog.
3. Select a concept model that you want to open and click **Open**. The concept model will open in read-only mode.
4. To edit the model, right-click **Data** in the Containment tree, and select **Lock** > **Lock Elements for Edit Recursively**. The concept model will be locked for editing.

If you make changes to the model, you need to commit them so that others can update the model with your changes.
(See http://docs.nomagic.com/display/MD184/Committing+changes+to+CEDW for the instructions to commit changes to TWCloud.)

5.7.2 Export a Concept Model in Teamwork Cloud to an OWL Ontology

When exporting your model to an OWL ontology, you can let the Concept Modeler export it to the previous OWL export location or your selected directory. See 5.6.3 OWL Export Folder to learn more about the OWL export location options.

If the option **Always prompt for a file destination when exporting OWL** in the **Project Options** dialog is enabled, the Concept Modeler will always prompt you to select an OWL export file location every time you export the concept model. Figure 6 below shows the prompt for OWL export folder option is enabled (set as "true").
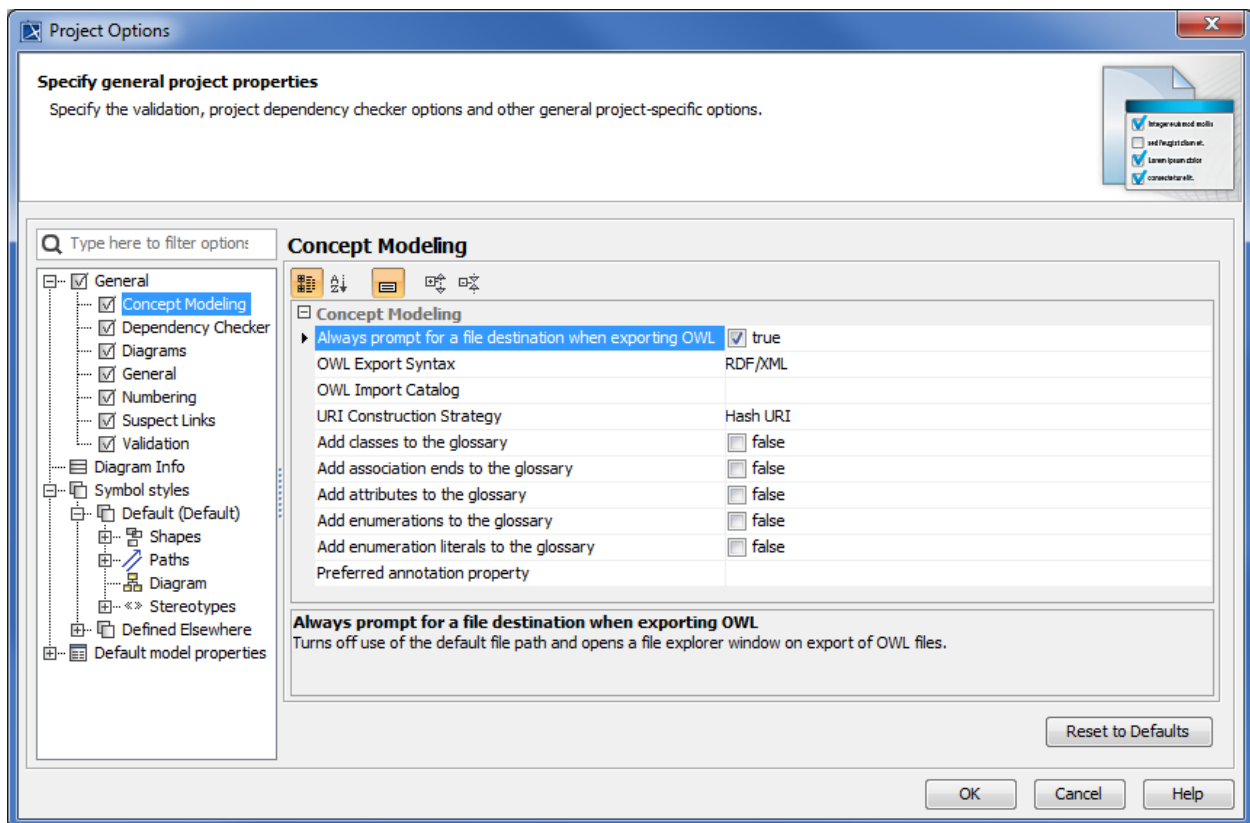
Figure 131 The prompt for an OWL export file destination option in the Project Options dialog.

You do not have to lock the concept model to export it to an OWL ontology. The following steps explain how to export a concept model that you have added to TWCloud.

How to export a concept model in TWCloud to an OWL ontology

1. Right-click a concept model in the Containment tree.
2. Click **Concept Modeling** > **Export Concept Model to OWL** to export the concept model. Any one of the following will happen:

   - If the prompt for export location is either enabled or disabled and you never export the project to an OWL ontology before, a dialog will open to prompt you to select a desired location. This location will be set as default.
   - When saving new export location and package is unlocked, Concept Modeler will try to lock it and show message in notification window. If locks cannot be obtained, due to package being locked by another user, a message will show in notification window and export location will not be saved, but concept model will still export.

- If the prompt for export location is enabled and you have exported a project before, and the location is valid, a dialog will open prompting you to select either the previous location or a new location.
- If the prompt for export location is enabled and you have exported a project before, and the location is invalid, an error will open in the Notification Window, and a dialog will open prompting you to select a new location.
- If the prompt for export location is disabled and you have exported a project before, and the location is valid, the Concept Modeler will export it directly to the location.
- If the prompt for export location is disabled and you have exported a project before, and the export location is invalid, an error will open in the Notification Window, and the Concept Modeler will prompt you to select another location.

When you would like to select multiple packages, you must right-click on one of the packages to export. You will always receive a prompt for file destination which is applied to all of your packages and the same file location will be loaded to all the packages. You are able to select different file destinations for each package, but the software always saves the same starting location for each package.

The error that appears in the Notification Window when trying to export the concept model to an OWL ontology can be caused by entering an incorrect path name or the path name to the previous export location does not exist anymore. If you encounter this type of error, you need to open the **fileExportPath** tagged value in the model's **Specification** dialog and correct the path name, or select another location (Figure 7).
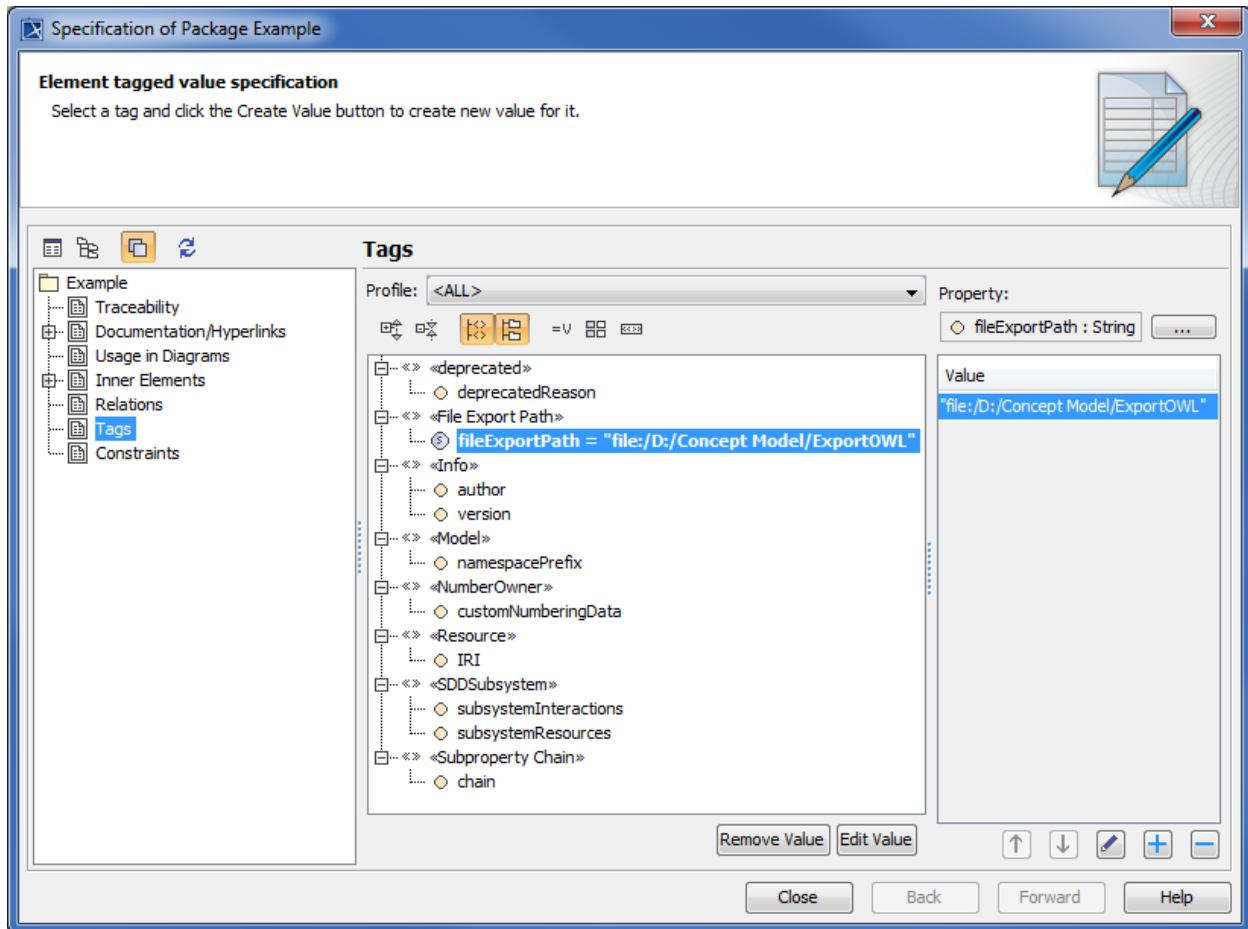
Figure 132   The OWL export destination directory in the model's Specification dialog.

For more information about changing the OWL export location, see 5.6.4 Export a Concept Model to OWL.

5.8   Automatically Generate Glossaries

The Concept Modeler can automatically generate glossaries for classes, association ends, attributes, enumerations, and enumeration literals in a concept model upon importing an OWL ontology. Additionally, it can generate glossary entries when those concept model elements are created or edited in the diagram or the containment tree.

To set the options for automatic generation of glossaries:

1. Select **Project** from the **Options** menu.

Copyright © 2017, No Magic, Inc.
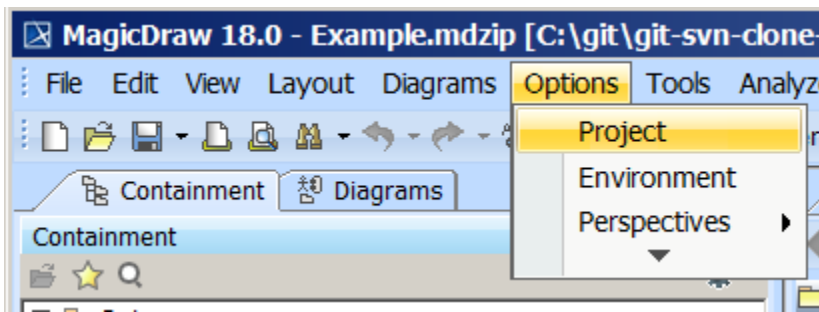
Figure 133   The Concept Modeler's project options

2. Select **General project options.**
3. Click on the corresponding checkbox for the option to enable or disable the following options (by default, these options are disabled):
   a. Add classes to the glossary.
   b. Add association ends to the glossary.
   c. Add attributes to the glossary.
   d. Add enumerations to the glossary.
   e. Add enumeration literals to the glossary.

4. Click **OK**.
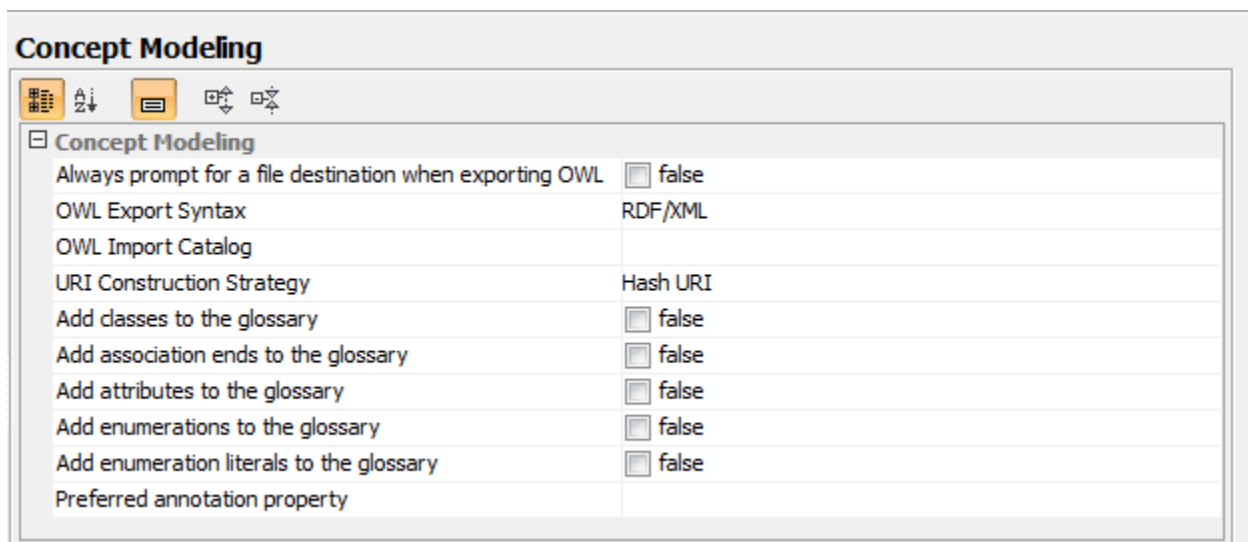
Figure 134   The Concept Modeler's glossary options

5.9   Create a Glossary Table

In a concept model, a glossary table contains the names and descriptions of classes, association ends, attributes, enumerations, and enumeration literals that are defined in the concept model.

Copyright © 2017, No Magic, Inc.

To create a glossary table:

1. Right-click the owning package.

Note	The owning package must have the correct «Model» or «Concept Model» stereotype for these menus to appear.

2. Select **Concept Modeling** > **Create Glossary Table** (see the following figure).

Note	If a glossary table already exists in the owning package, the **Create Glossary Table** menu option will not be available.
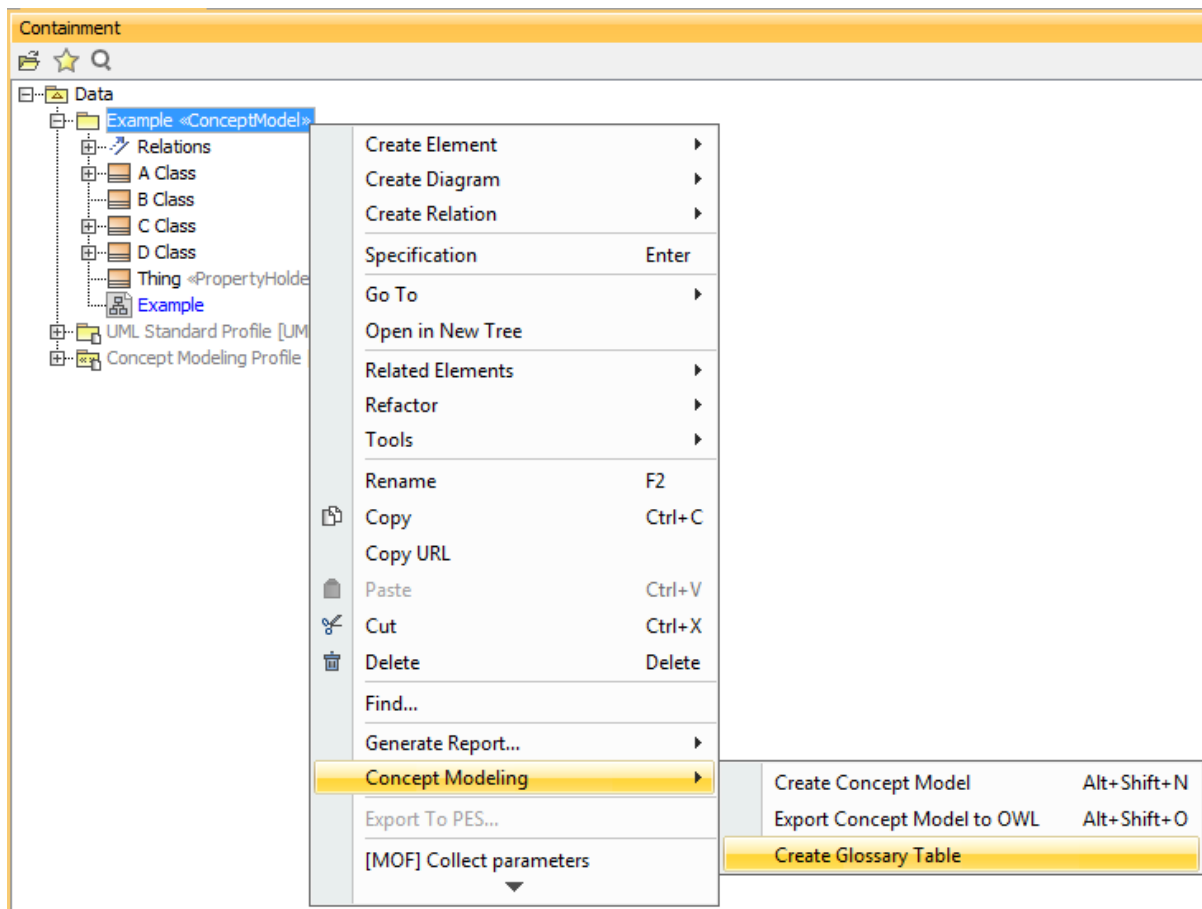
Figure 135   The Concept Modeler's Create Glossary Table menu

At least one glossary generation project option must be enabled. If all of the options are turned off, the **Create Glossary Table** menu option will be disabled.

Create Concept Model	Alt+Shift+N
Export Concept Model to OWL	Alt+Shift+O
Create Glossary Table (Nothing selected in Project Options)	

Figure 136 The glossary creation menu is disabled

5.10 Rebuild a Glossary Table

When the glossary-generation options change, one may want to rebuild existing glossary tables to only contain the specified kinds of entries.

To rebuild an existing glossary table for a concept model:

1. Right-click on the existing concept modeling generated glossary.
2. Select **Concept Modeling**.
3. Select **Rebuild Glossary Table**.

| Note | The owning package must have the correct «Model» or «Concept Model» stereotype and a glossary table must not already exist for this menu item to appear. |

Copyright © 2017, No Magic, Inc.
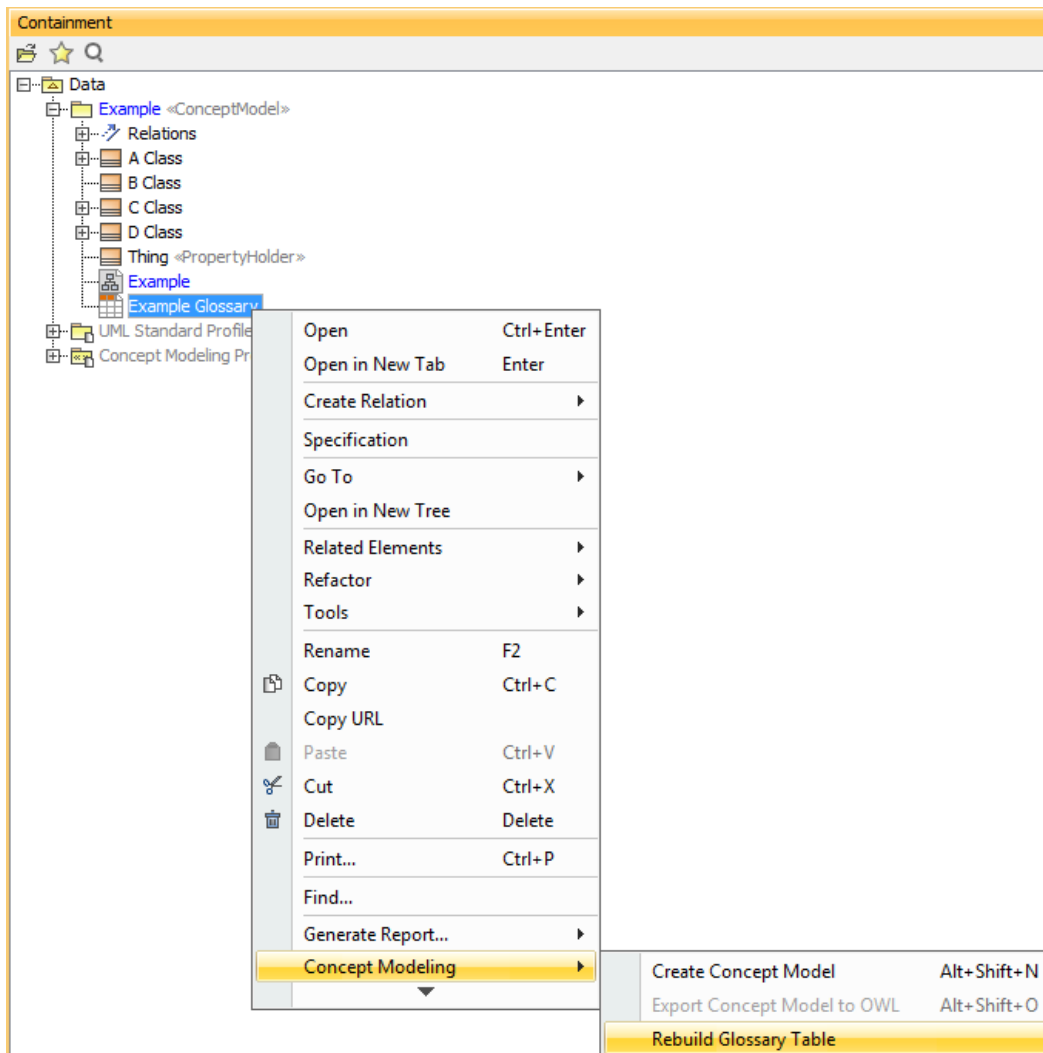
Figure 137   The Concept Modeler's Rebuild Glossary Table menu

Note	At least one glossary generation project option must be enabled for the **Rebuild Glossary Table** menu item to be enabled, otherwise it will look like the last menu item in the following figure.
	Figure 138 The Rebuild Glossary Table menu is disabled
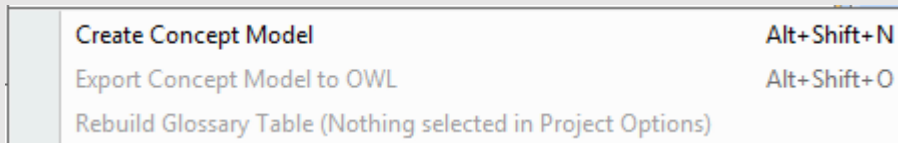
Copyright © 2017, No Magic, Inc.

5.11 View a Glossary

The generated glossary is created in the package of the owning concept model.

To view a glossary's contents:

- Double-click the glossary in the Containment tree.
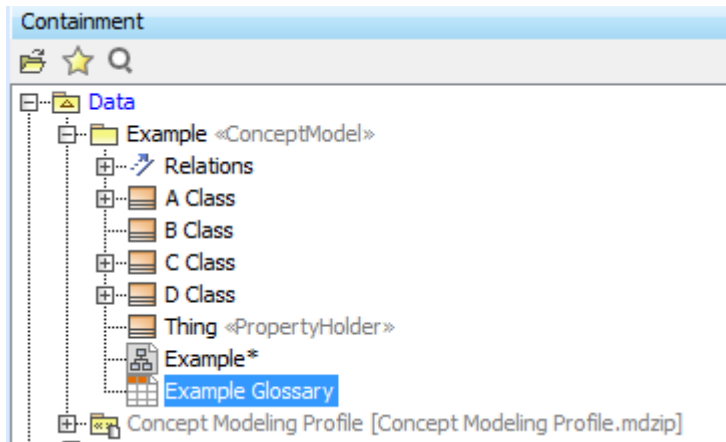
Figure 139   A generated glossary in the Containment tree

Figure 140   A Concept Modeler's glossary table

Editing the glossary name or a description in the glossary will automatically update the corresponding element in the concept model.

Furthermore, clicking the name of a class, association end, attribute, enumeration, or enumeration literal will display the provided element's description.

Copyright © 2017, No Magic, Inc.
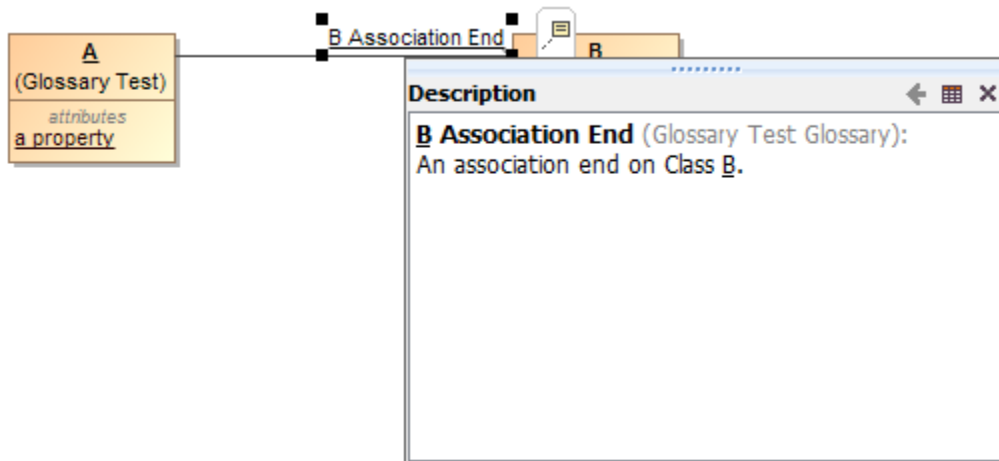
Figure 141   A element's description in the Concept Modeler

For additional information on how to manually create, delete, and update elements in the glossary, please refer to the user manual for MagicDraw 18.0 SP4 or higher.

5.12 Create a Property Holder

To create a property holder (UML class stereotyped as «Anything»):
1. Create a UML class (named "Thing" below).
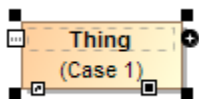
Figure 142   A UML class

2. Right-click on the created class.
3. Click **Stereotype.**

Copyright © 2017, No Magic, Inc.
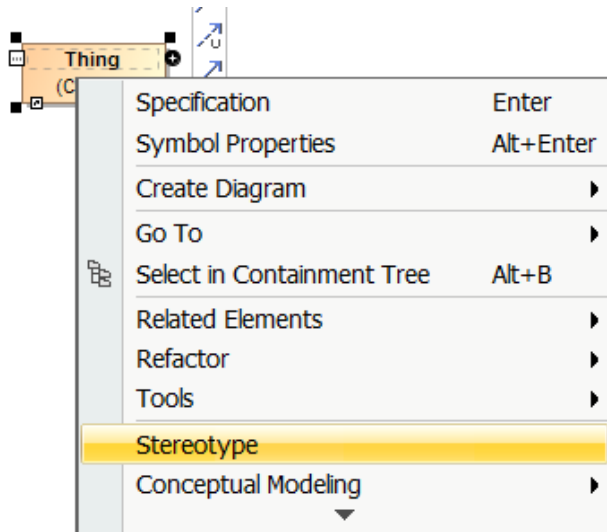
Figure 143   The Stereotype shortcut menu of the class

4. Type "Anything" in the search box.
5. Select the stereotype "Anything".
6. Click **Apply**.
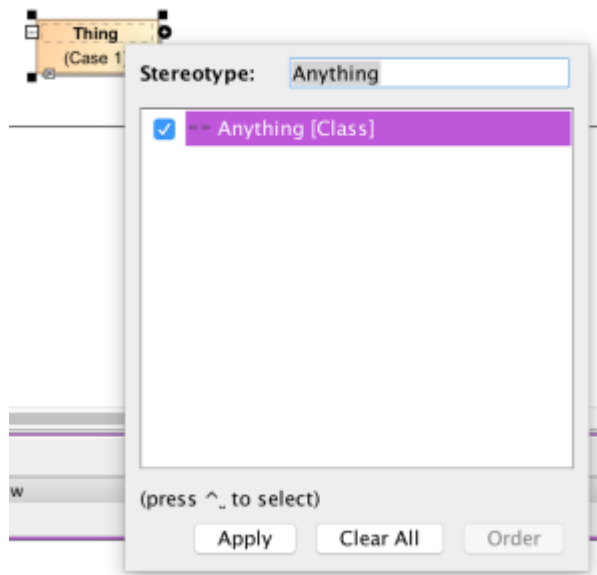
Figure 144   Selecting the Anything stereotype for the class

5.13 Universal Quantification Constraints for an Existing Property

In the concept modeling interpretation of UML, redefining an existing property creates a universal quantification constraint in the context of the owning class (see section 3.6 Universal Quantification Constraint). This interpretation is based on {redefines} in UML, which allows for more specific constraints to be added to an existing property without defining a new one.

5.13.1 Add a Universal Quantification

To add a universal quantification:
1. Drag and drop a property to be redefined (for example, "has" from "Person") onto a redefining property (for example, "has" from "Dog Lover").

Note	The property is owned by the class at the opposite end of the association. Additionally, the target can have the same name as the source or be unnamed. The resulting redefinition's multiplicity is adjusted to conform to the multiplicity of the dragged, redefined property.
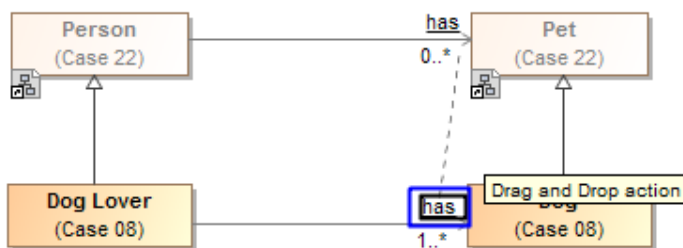
Figure 145   Dragging the property to be redefined to the redefining property

2. Click **Create universal quantification.**
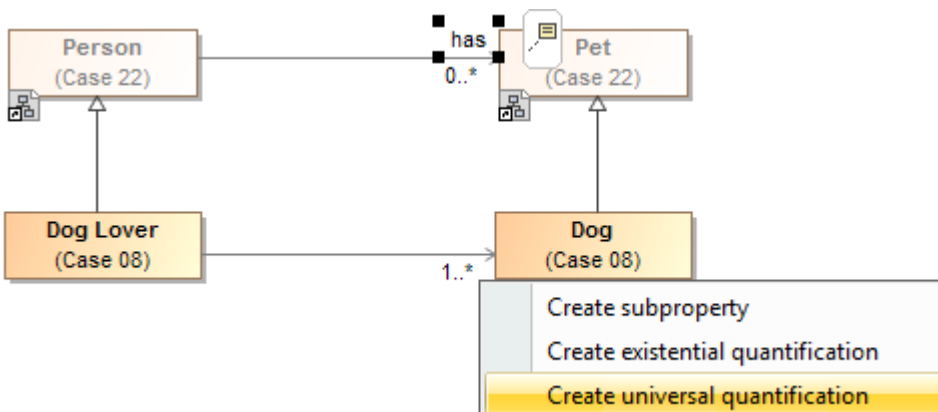
Figure 146   The Create universal quantification shortcut menu

To remove a property redefinition from a property:
1. Right-click a redefining property.
2. Select **Concept Modeling.**
3. Select **Remove universal quantification**.
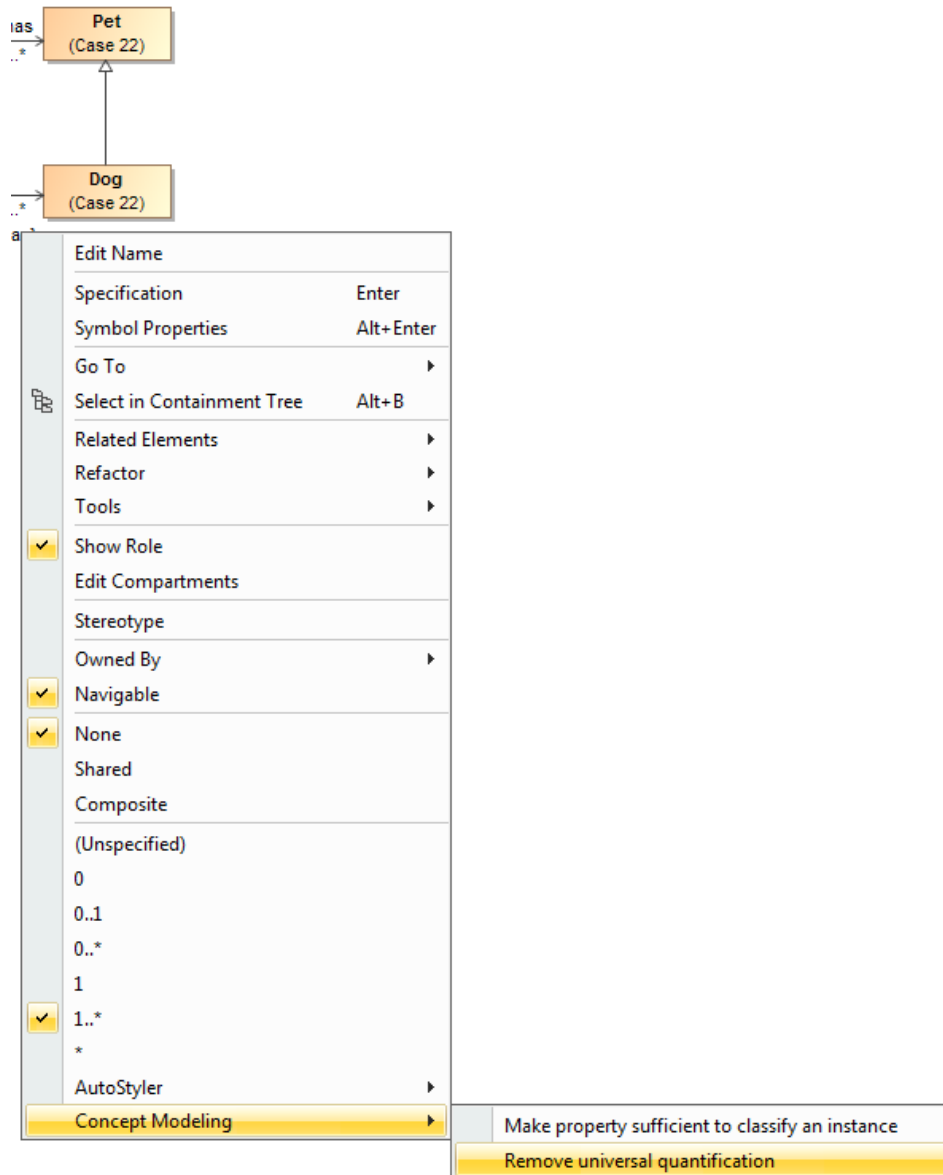
Figure 147   Removing a universal quantification constraint from a property

5.14 Subproperties

In the Concept Modeling interpretation of UML, subsetting a property creates a subproperty when the subsetting property has a different name than the subsetted property (see section 3.4 Subproperty). UML provides a {subsets} constraint that asserts that the values within a

subsetting property are also in the set of values within a subsetted property. The concept modeling profile interprets a subproperty as a subsetting property that has a different name.

5.14.1 Add a Subproperty

<u>To add a subproperty:</u>
1. Drag and drop a subsetted property (for example, "consists of" from "Soccer Team") onto a property (for example, "composed of" from "Futsal Team").

Note	The property is owned by the class at the opposite end of the association. Additionally, the target can have the same name as the source or be unnamed. The resulting redefinition's multiplicity is adjusted to conform to the multiplicity of the dragged, subsetted property.

2. Click on **Create subproperty**.
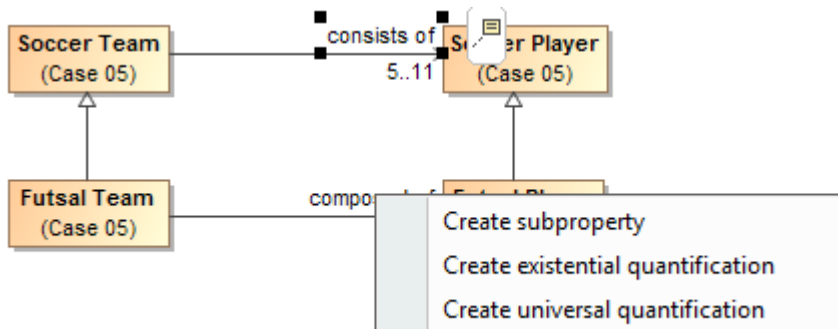
Figure 148 Dragging a subsetted property to another property to create a subproperty

5.14.2 Remove a SubProperty

<u>To remove a property subsetting from a property:</u>
1. Right-click a subsetting property (for example, "composed of" from "Futsal Team").
2. Select **Concept Modeling > Remove subproperty.**
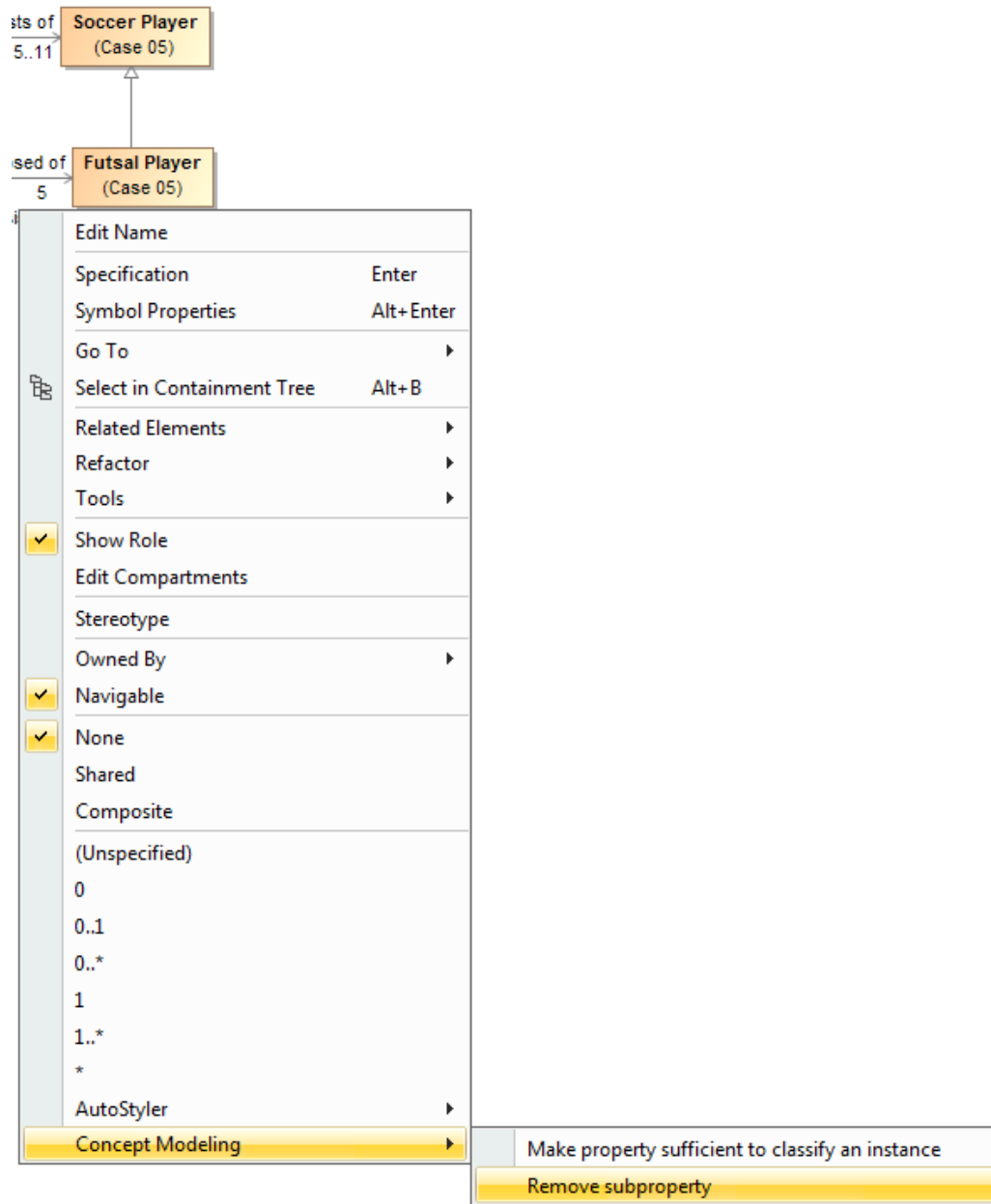
Figure 149   Removing a property subsetting from a property

5.15 Create an Existential Quantification (Qualified) Constraint for a Property

In the Concept Modeling interpretation of UML, subsetting a property without giving the new property a different name (or leaving off the new property name altogether) creates an existential quantification constraint (see section 3.5 Existential Quantification Constraint). As {subsets} with an omitted name is not well defined in UML, it is used in the Concept Modeling profile to express that a subset of values must meet the stated cardinality and type constraints of the subsetting property.

5.15.1 Add an Existential Quantification

<u>To add an existential quantification constraint to a property:</u>

1. Drag and drop a subsetted property (for example, "has" from "Person") onto the cardinality of the property that will subset another property (for example, "unnamed" from "Dog Caretaker"). Note that the target can have the same name as the source or be unnamed.

Note	The resulting subsetting property's multiplicity is adjusted to conform to the dragged, subsetted property, and to have a minimum cardinality of at least one.
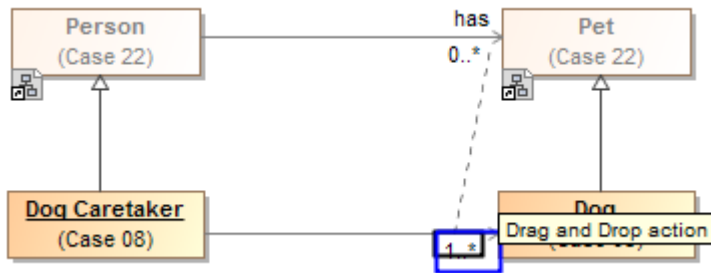
Figure 150   Dragging a subsetted property to a cardinality of another property to create an existential quantification constraint

2. Right-click the subsetted property and select **Create existential quantification.**
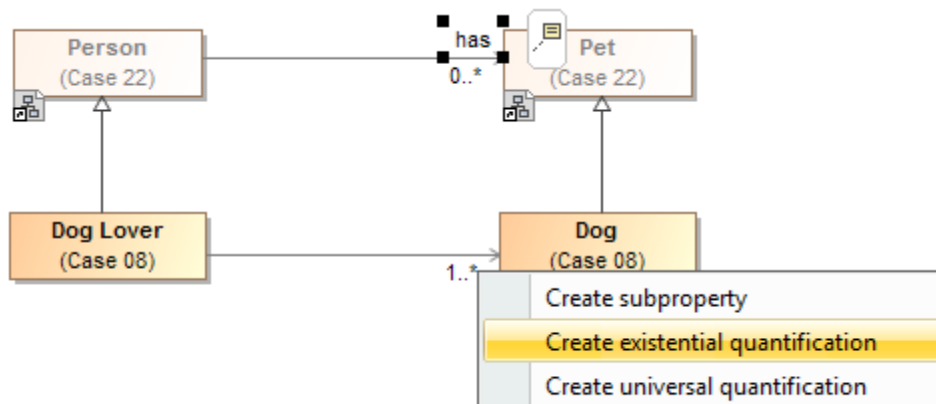
Figure 151   The Create existential quantification shortcut menu

5.15.2 Remove an Existential Quantification

<u>To remove an existential qualification constraint:</u>
1. Right-click a subsetting property.
2. Select **Concept Modeling.**
3. Select **Remove existential quantification**.
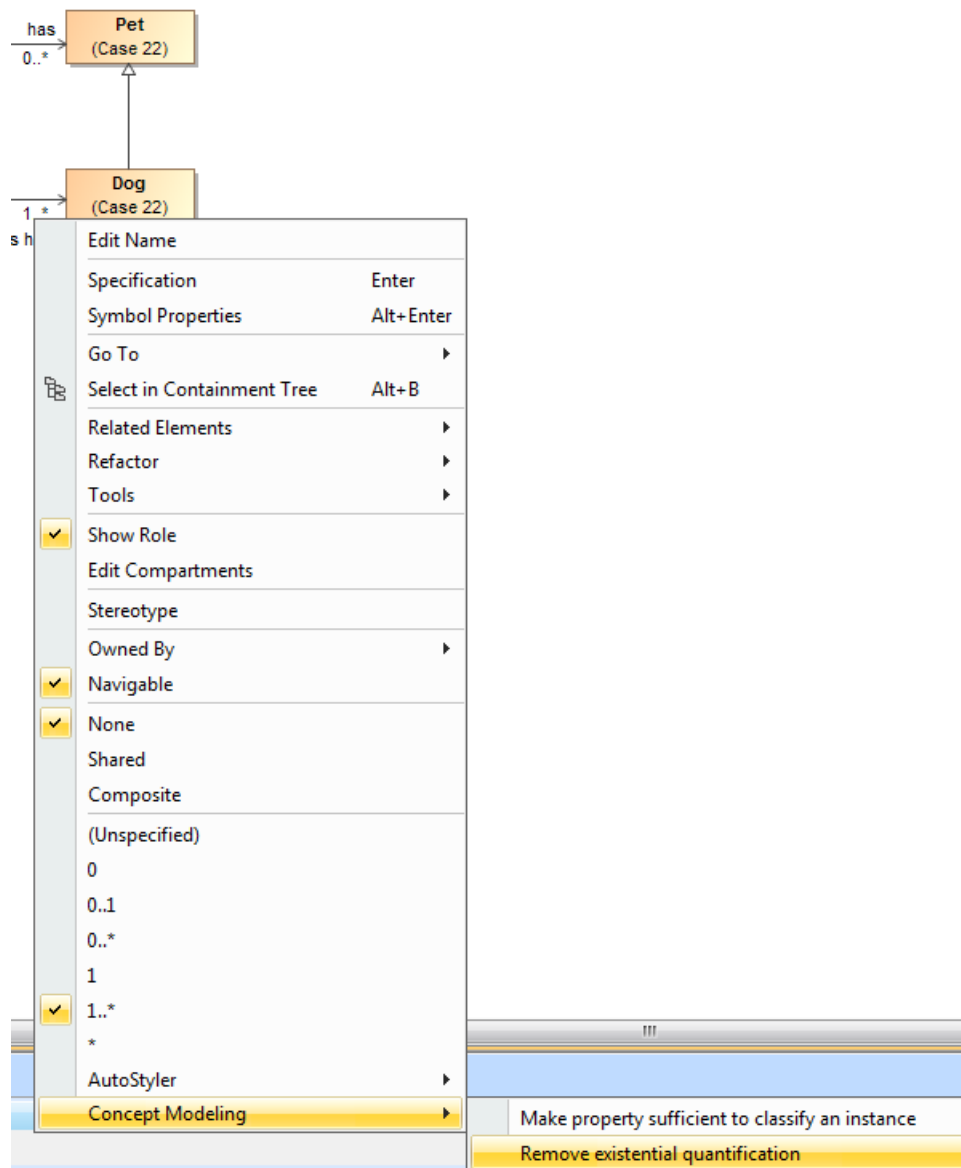
Figure 152   The Remove existential quantification shortcut menu

Copyright © 2017, No Magic, Inc.

5.16 Go to Redefined Property

It is often useful to go to a redefined property to see its original definition. There are two ways to do so. The first is to go to the redefined property in the Containment tree. The second is to go to the redefined property on a diagram.

5.16.1 Go To Redefined Property in Containment Tree

To go to a redefined property in the Containment tree:
1. Right-click a redefining property, its multiplicity, or its redefinition.
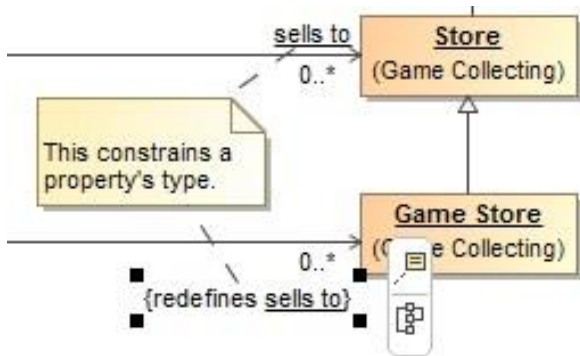
Figure 153   A redefining property in Concept Modeler

2. Select **Go To > Redefined property in containment tree**.
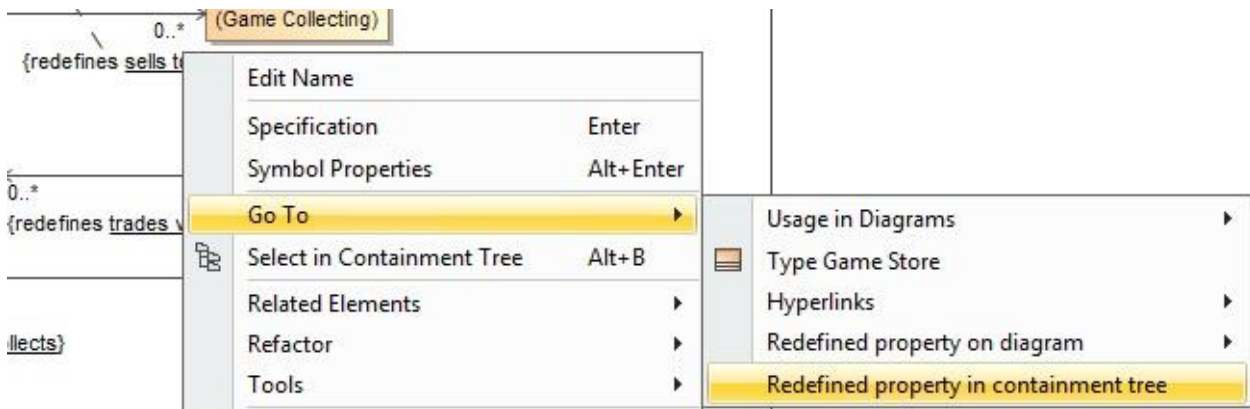
Figure 154   The Redefined property in containment tree shortcut menu

The focus will jump to the redefined property in the containment tree, as shown in the diagram below.
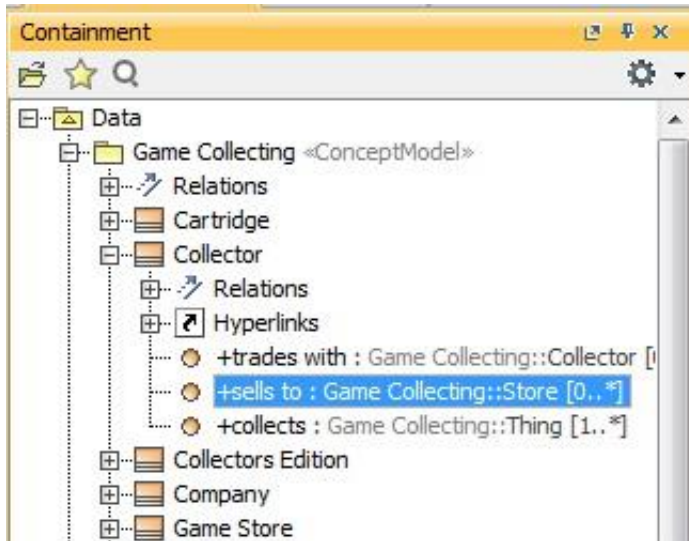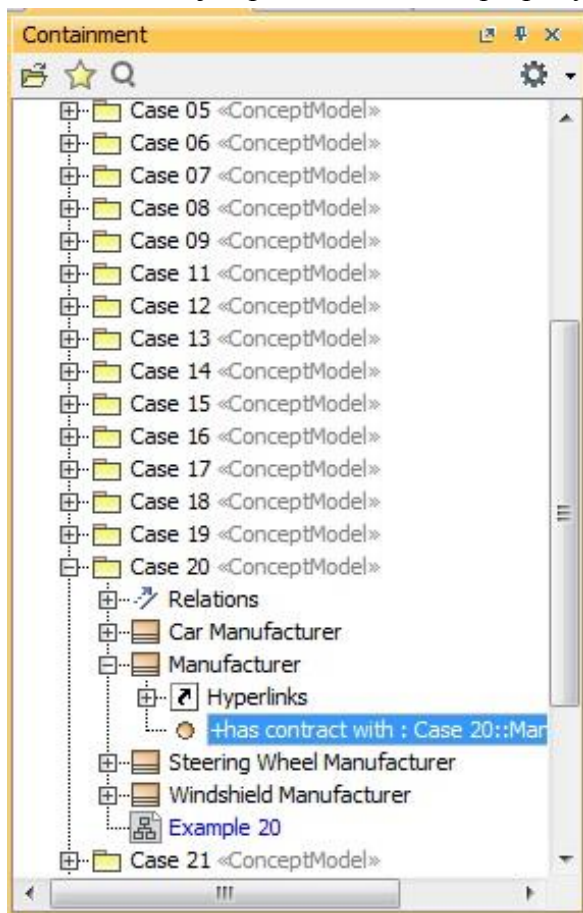

Figure 155   The Concept Modeler highlights the redefined property in the Containment tree

5.16.2 Go To Redefined Property on Diagram

To focus on a redefined property on a diagram:

1. Right-click a redefining property, its multiplicity, or its redefinition.
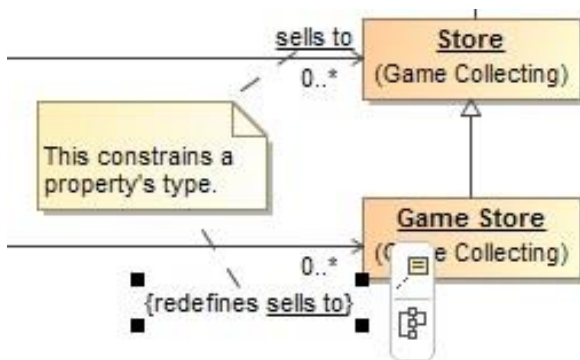
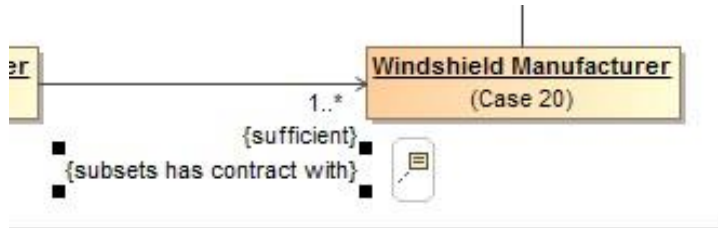
Figure 156   A redefining property in the Concept Modeler

2. Select **Go To > Redefined property on diagram** and choose a diagram.

Copyright © 2017, No Magic, Inc.
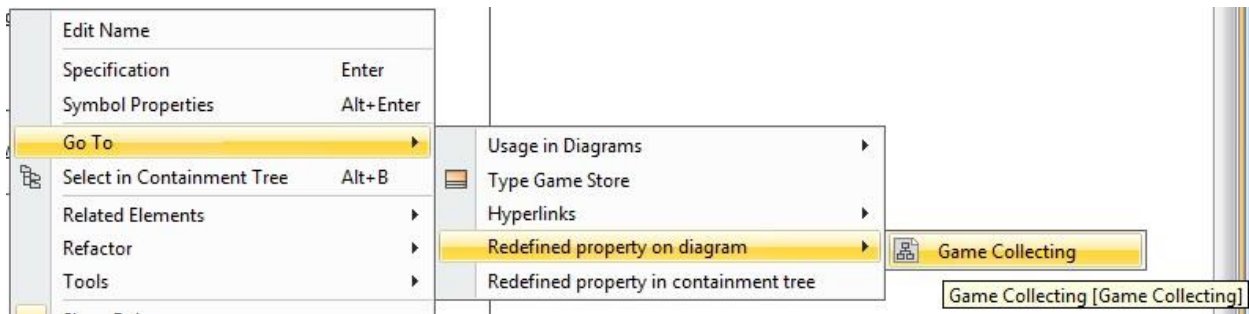
Figure 157   The Redefined property on diagram shortcut menu

Focus will jump to the redefined property on the chosen diagram.
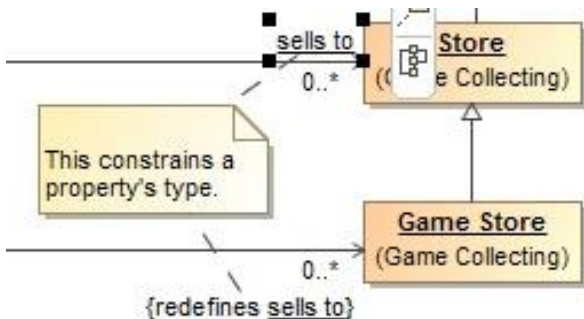
Figure 158   The Concept Modeler highlights the redefined property on the selected diagram

5.17   Go To Subsetted Property

It is often useful to go to a subsetted property to see its original definition. There are two ways to do so. The first is to go to the subsetted property in the Containment tree. The second is to go to the subsetted property on a diagram.

5.17.1 Go To Subsetted Property in Containment Tree

To focus on a subsetting property in the Containment tree:
   1. Right-click on the subsetting property, its multiplicity, or its {subsets}.
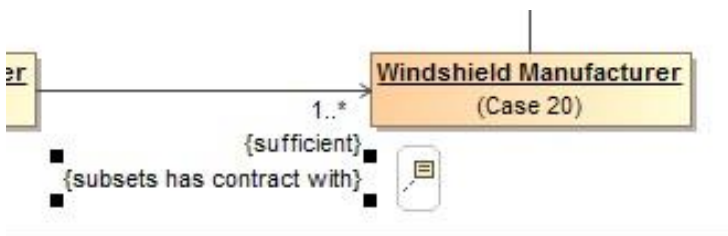
Figure 159   A subsetting property in the Concept Modeler

Copyright © 2017, No Magic, Inc.

2. Select **Go To > Subsetted property in containment tree**.
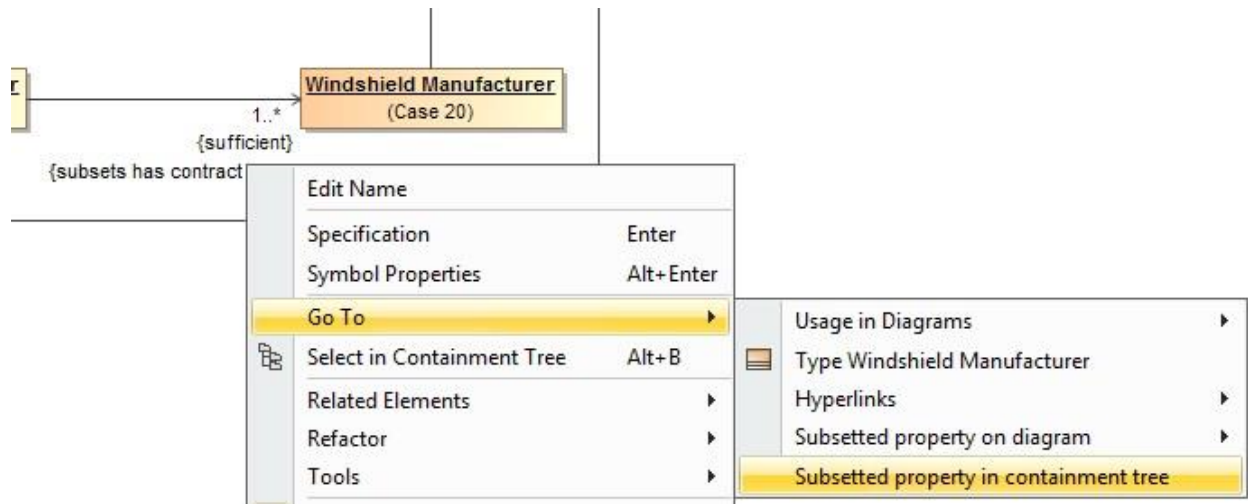
Figure 160   The Subsetted property in the Containment tree

The focus will jump to the subsetted property in the Containment tree.

Figure 161   The Concept Modeler highlights the subsetted property in the Containment tree

Copyright © 2017, No Magic, Inc.

5.17.2 Go To Subsetted Property on Diagram

To focus on a subsetted property on a diagram:
1. Right-click on the subsetting property, its multiplicity, or its {subsets}.

Figure 162   A subsetting property in the Concept Modeler

2. Select **Go To > Subsetted property on diagram** and select a diagram.
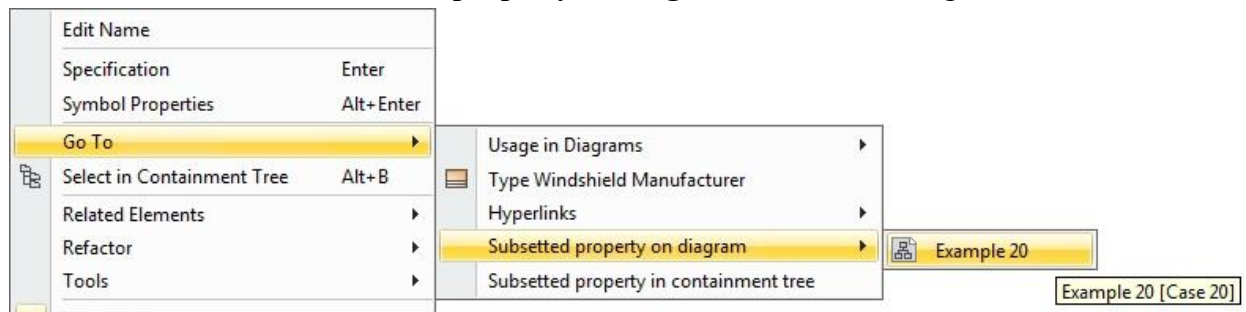
Figure 163   The Subsetted property on diagram shortcut menu

The focus will jump to the subsetted property on the chosen diagram.
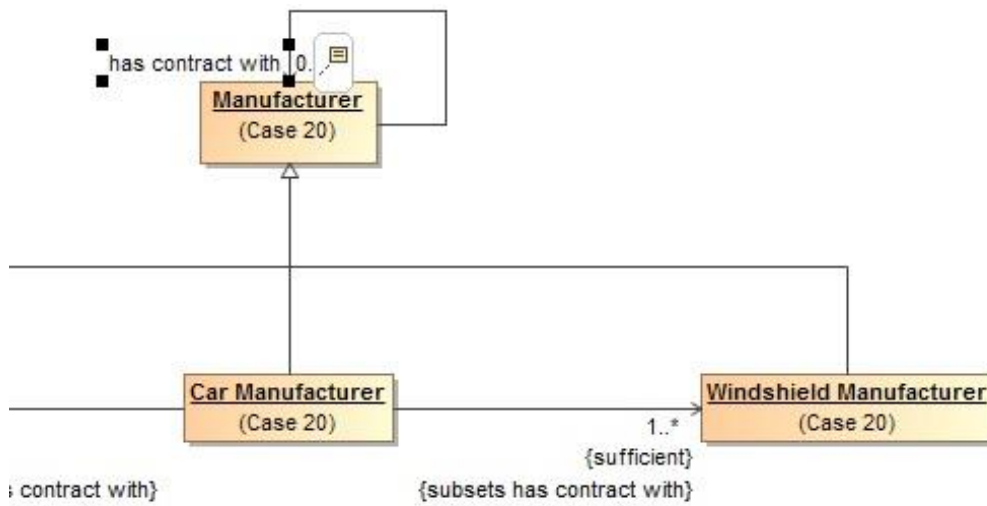
Figure 164   The Concept Modeler highlights the subsetted property on the selected diagram

Copyright © 2017, No Magic, Inc.

5.18 Create a Necessary and Sufficient Condition

5.18.1 Add a Sufficient Condition

In the Concept Modeling interpretation of UML, a property that has the {sufficient} constraint applied to it indicates that when an instance satisfies the multiplicity and type constraints for the property's values, not only is a *necessary* condition to be an instance of the class met, a *sufficient* condition is also met (see section 3.7 Necessary and Sufficient Condition).

To create a sufficient condition:

1. Right-click the association end for the property to which the {sufficient} constraint will be applied (unnamed from "Dog Owner"). Remember that the property is owned by the class at the opposite end of the association.
2. Select **Concept Modeling** > **Make property sufficient to classify an instance** in the shortcut menu.
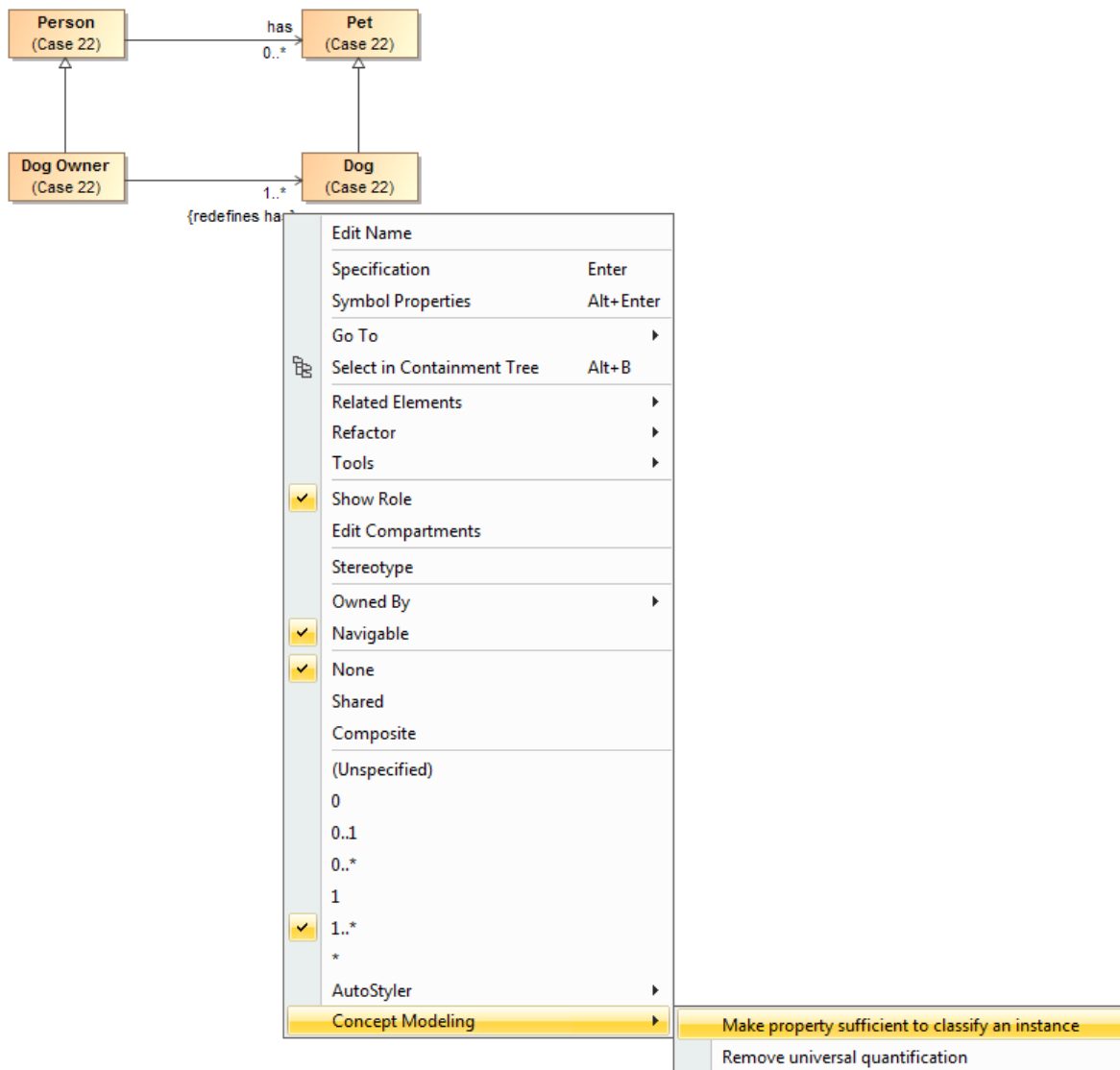
Figure 165 Make property sufficient to classify an instance shortcut menu

Copyright © 2017, No Magic, Inc.

The {sufficient} constraint is toggled on for the property.
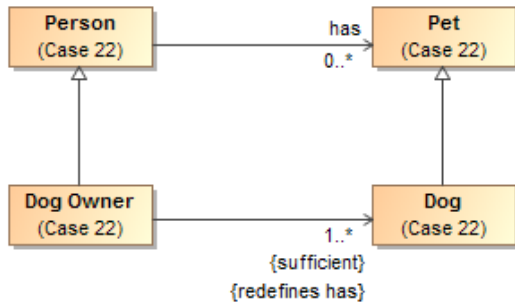
5.18.2 Remove a Sufficient Condition

To remove a sufficient condition on a property:

1. Right-click the association end for the property to which the {sufficient} constraint will be removed (unnamed from "Dog Owner").
2. Select **Concept Modeling > Make property insufficient to classify an instance.**
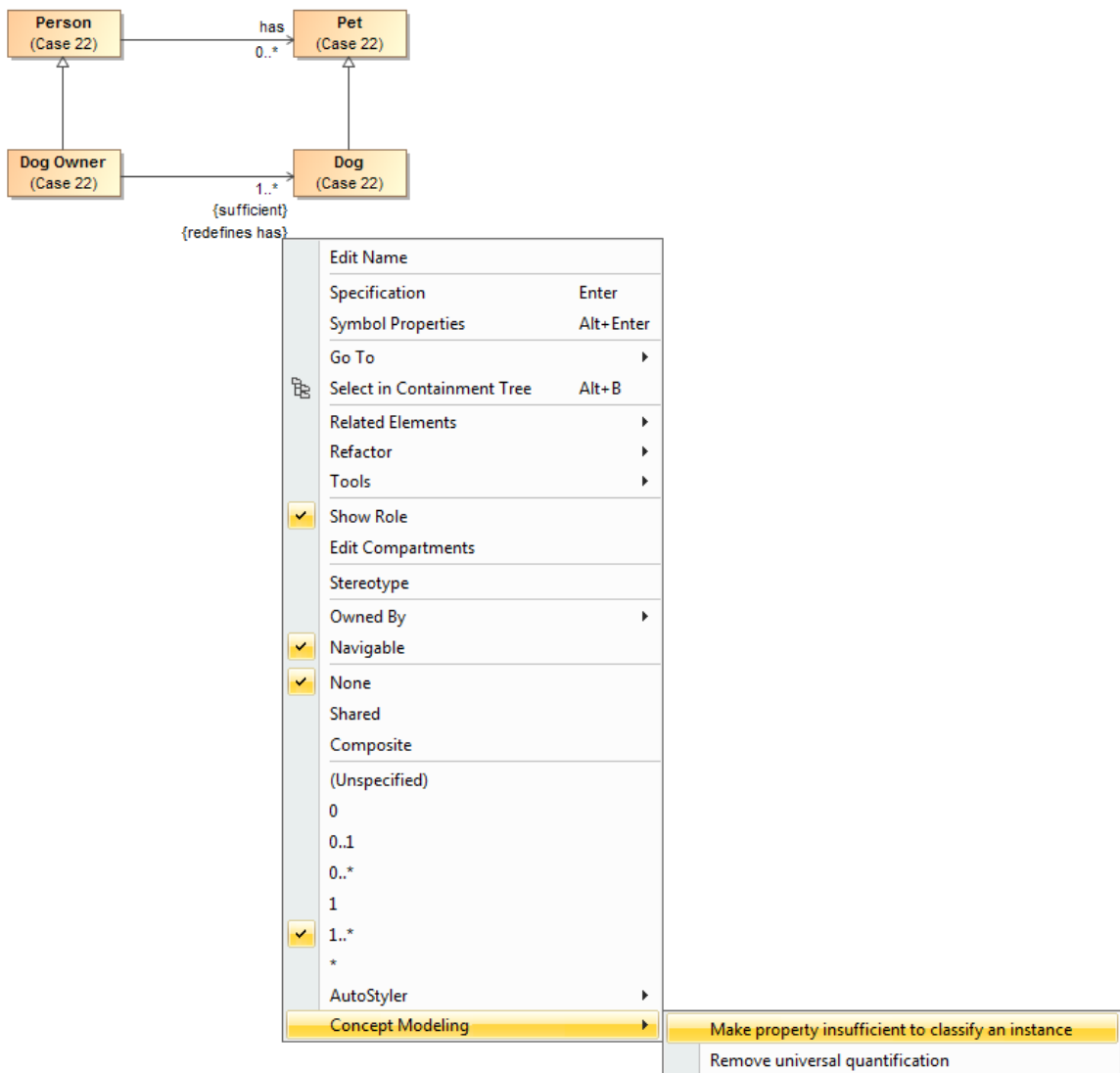
Figure 166 Make property insufficient to classify an instance

5.19 Working with Subclasses

Users may want to make a set of existing subclasses disjoint, overlapping, complete, or incomplete. The Concept Modeler provides a quick method for adding a generalization set to your concept model and setting its properties.

Note	• Creating generalization sets through the Concept Modeler is only applicable to generalization relationships connected together through the shared target notation. The manual method of creating generalization sets will still be available through the Specification window. Please see the MagicDraw user guide for additional information.
	• Anonymous unions are incompatible with {incomplete} because an instance can only be classified by one or more classes in a union, not the union itself.

5.19.1 Make Subclasses Disjoint

To make subclasses disjoint:

1. Right-click on the generalization relationship.
2. Select **Concept Modeling**.
3. Select **Make subclasses disjoint**.
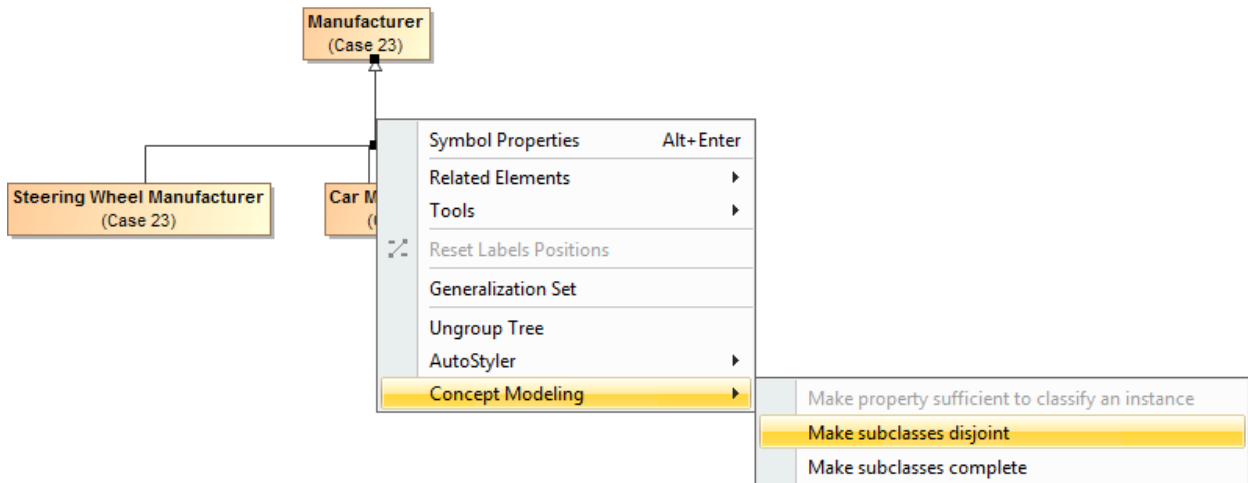
Figure 167   Make subclasses disjoint shortcut menu

5.19.2 Make Subclasses Complete

To make subclasses complete:

1. Right-click on the generalization relationship.
2. Select **Concept Modeling.**
3. Select **Make subclasses complete.**
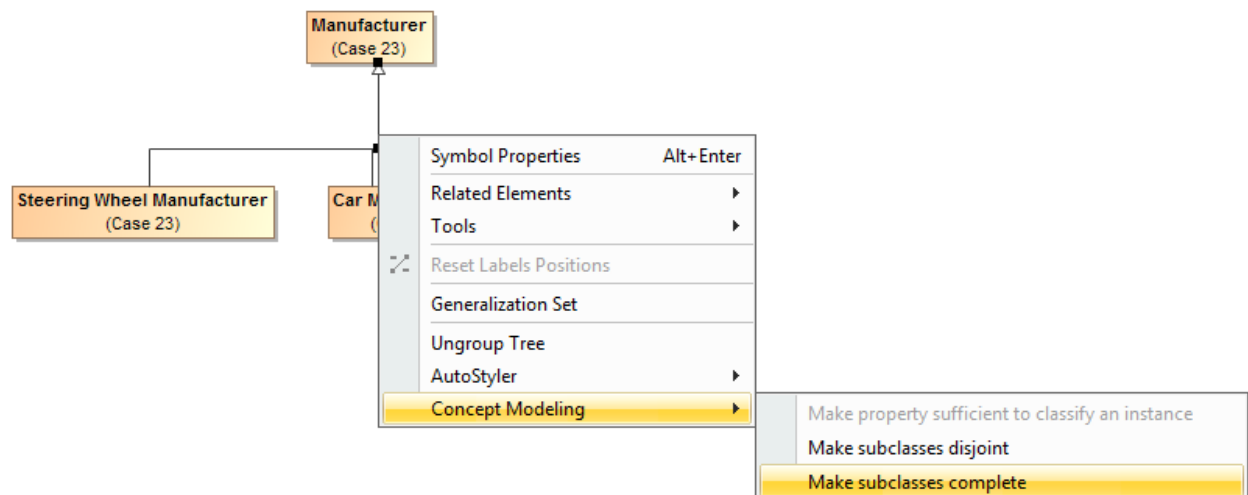
Figure 168   Make subclasses complete shortcut menu

5.19.3 Make Subclasses Overlapping

To make subclasses overlapping:
1. Right-click on the generalization relationship.
2. Select **Concept Modeling.**
3. Select **Make subclasses overlapping**.
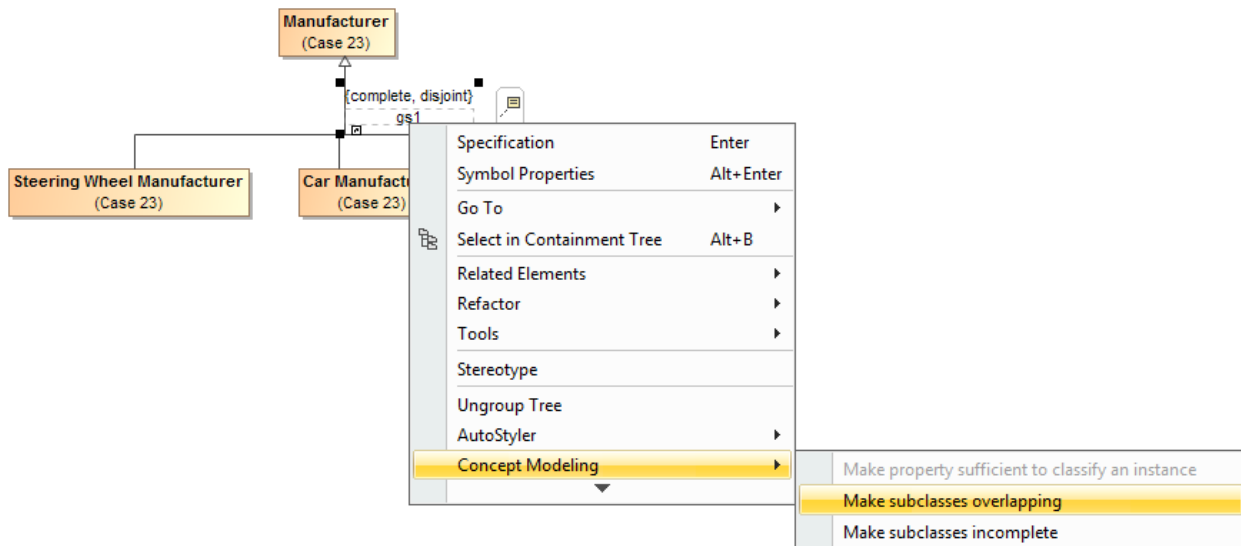
Figure 169   Make subclasses overlapping shortcut menu

Note	• Setting the {incomplete, disjoint} constraint back to the default setting of {incomplete, overlapping} will result in the removal of the generalization set, which has the same meaning.
	• Starting from MagicDraw 18.3, the **Concept Modeling** menu is disabled when you right-click a tree or a generalization set on a diagram because the menu options for creating a generalization set have been moved outside the **Concept Modeling** menu (see the following menu example).

Copyright © 2017, No Magic, Inc.
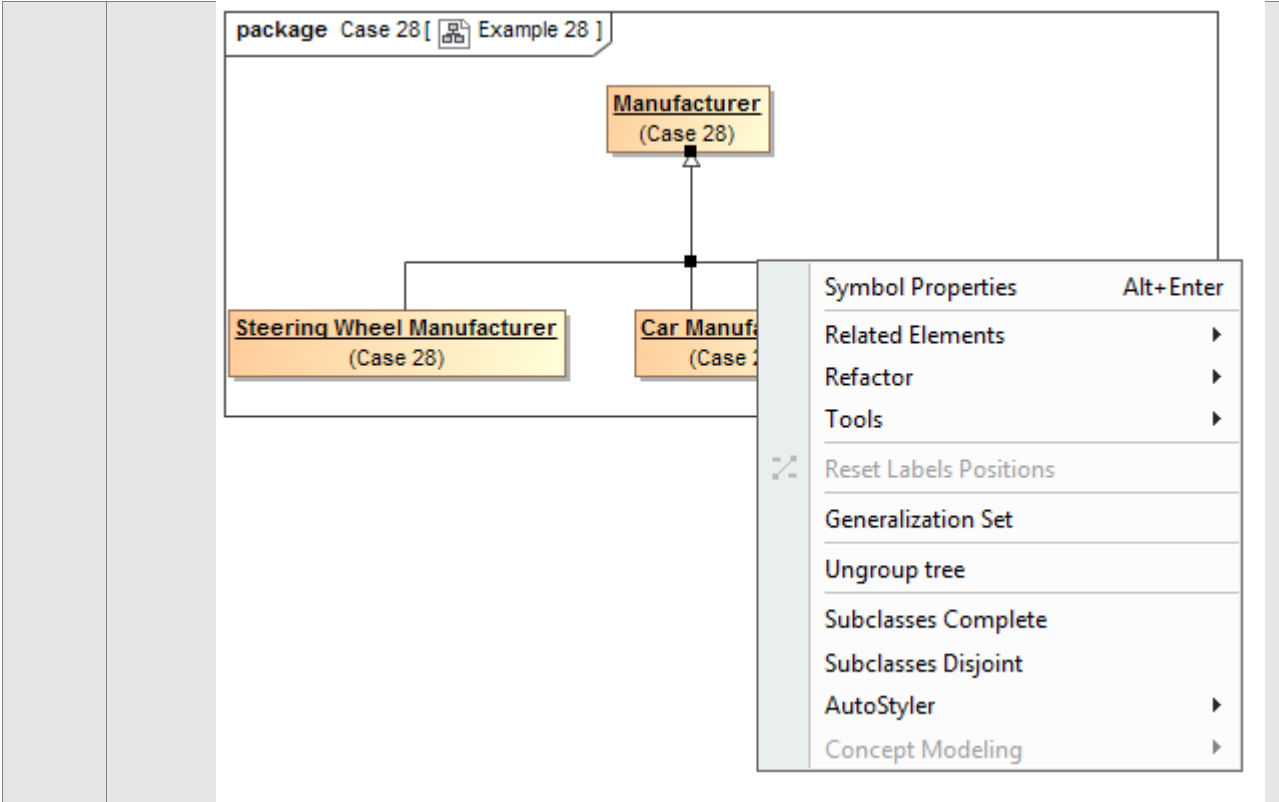
Figure 170   The Concept Modeling menu is disable in MagicDraw 18.3

- Starting from MagicDraw 18.3, the Concept Modeling menu options (i) **Make subclasses complete** and (ii) **Make subclasses disjoint** have been replaced with **Subclasses Complete** and **Subclasses Disjoint** respectively (see the following figure).
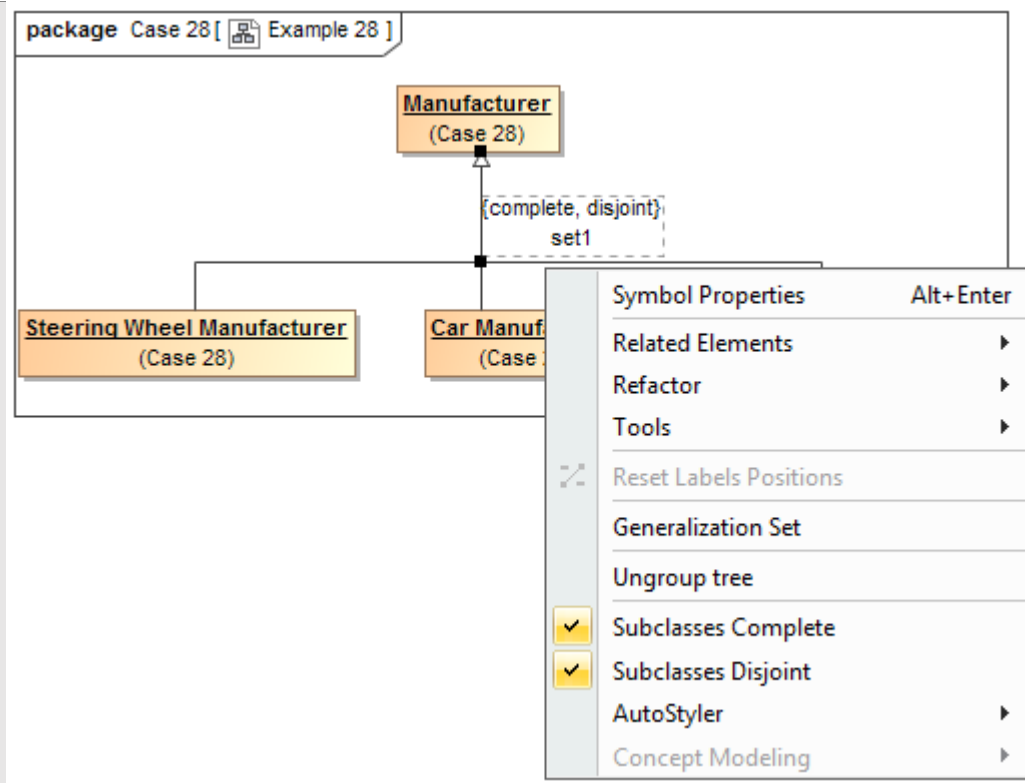
Figure 171   The Concept Modeling menu options

The following is how the menu options work:

(i)   When you make subclasses complete, there is a check mark before **Subclasses Complete**. If it is not complete, there is no check mark before the menu item.

(ii) When you make subclasses disjoint, there is a check mark before **Subclasses Disjoint**. If it is not disjoint, there is no check mark before the menu item.

5.19.4 Make Subclasses Incomplete

To make subclasses incomplete:

1. Right-click on the generalization relationship.
2. Select **Concept Modeling.**
3. Select **Make subclasses incomplete**.
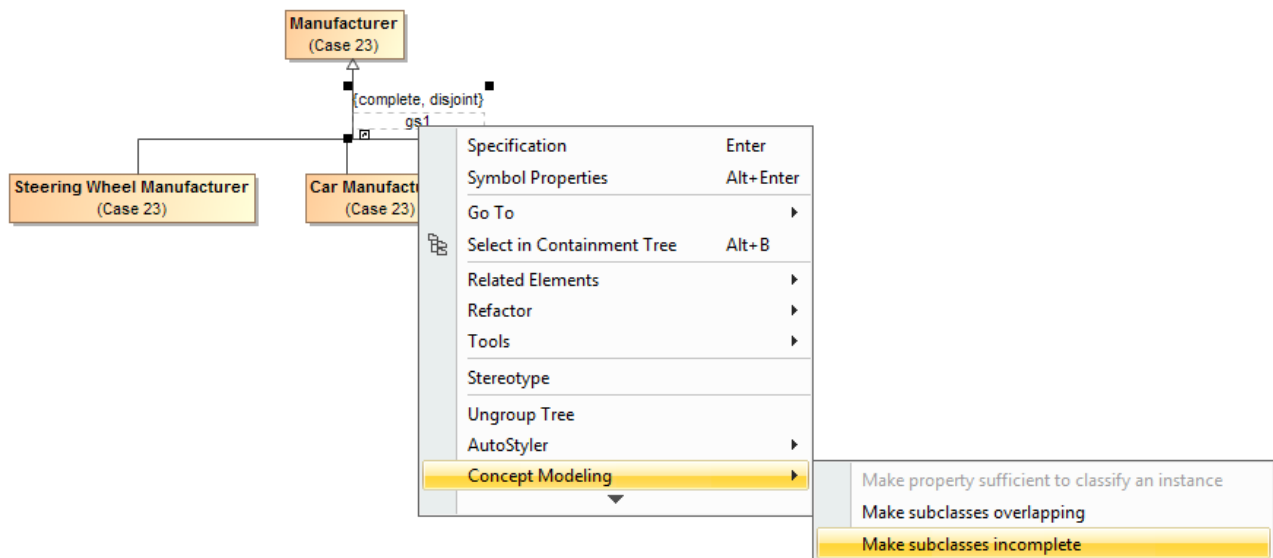
Figure 172 Make subclasses incomplete

Note	Setting the {complete, overlapping} constraint back to the default setting of {incomplete, overlapping} will result in the removal of the generalization set, which has the same meaning.

5.20 Working with Annotations

UML comments can be stereotyped as an «Annotation», then tied to a property that is stereotyped as an «Annotation Property». When a concept model is exported to OWL, these stereotyped UML comments become OWL annotations.

There are two ways that a user can add annotation properties to annotations: by importing an OWL ontology that defines annotation properties, or by defining a property and stereotyping it as an «Annotation Property».

5.20.1 Import an Ontology that Defines Annotation Properties

To import an ontology into an existing concept model:
1. Select **File** > **Import From** > **OWL Ontology File**.
2. Browse and select an OWL Ontology file.

Copyright © 2017, No Magic, Inc.
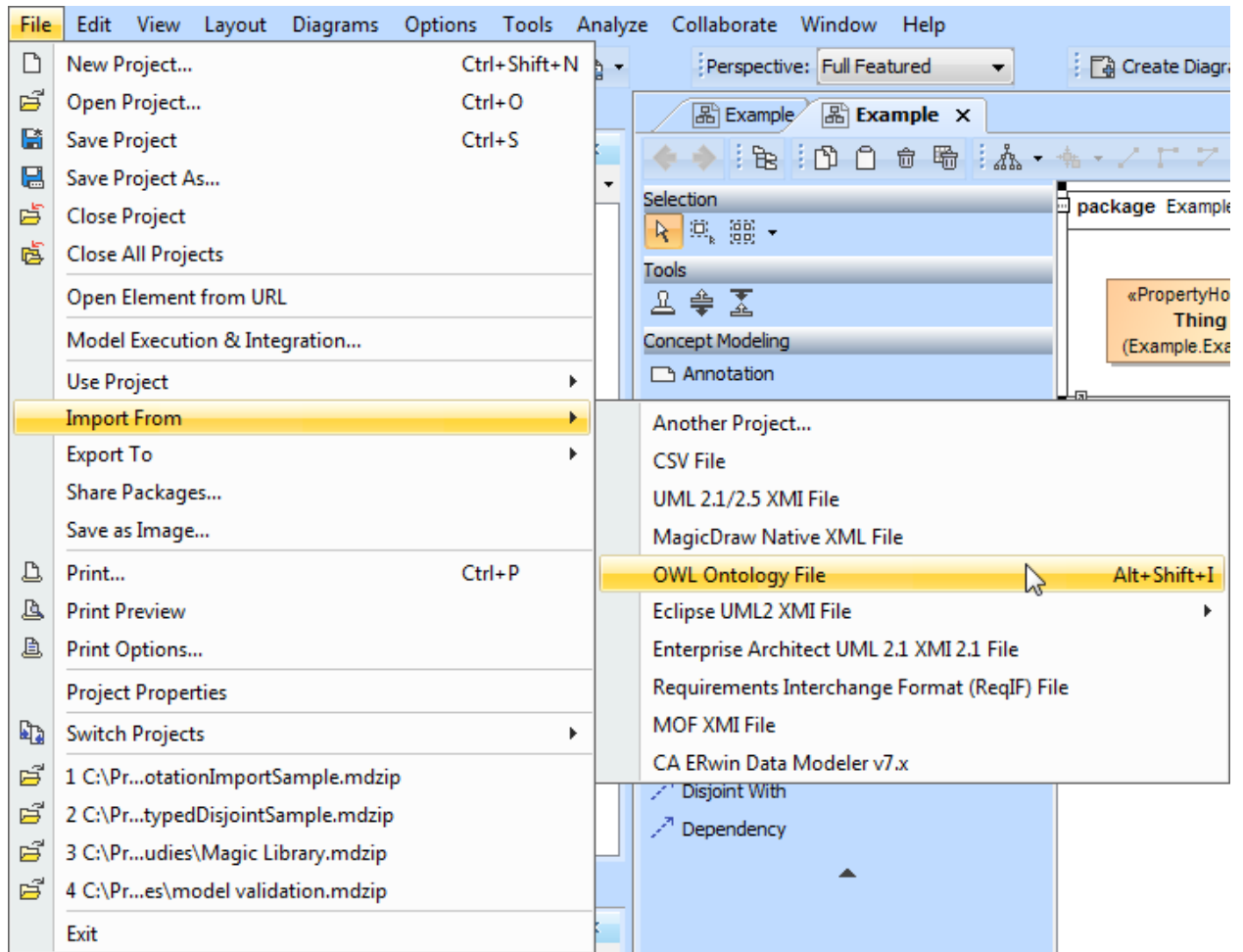
Figure 173   Importing an OWL ontology file to the Concept Modeler

3. Annotation properties imported from the OWL ontology will be displayed in the Containment tree under **Imported Ontologies** as shown in the following figure.
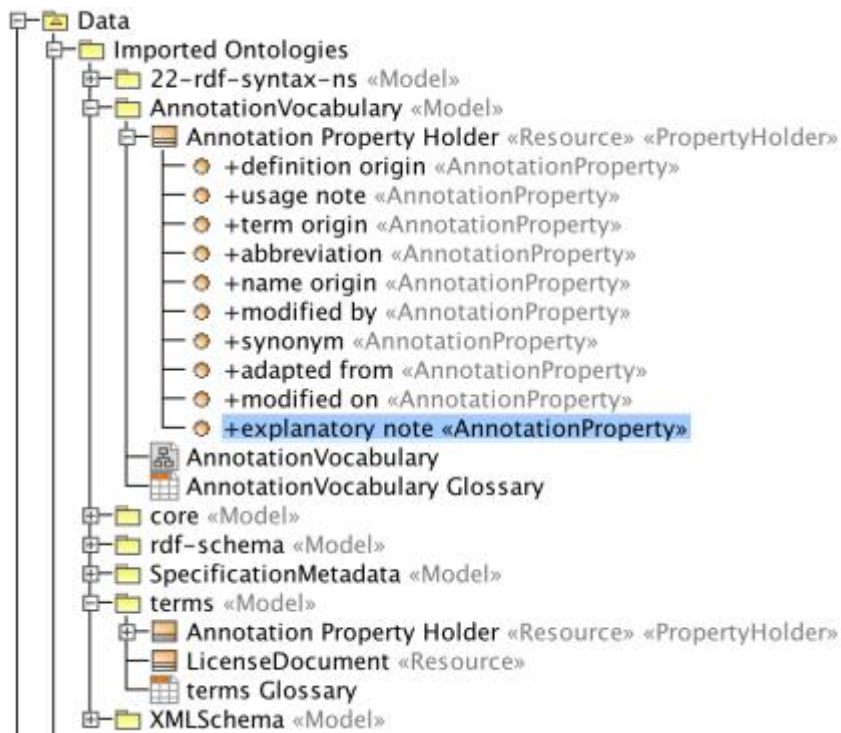
Figure 174 The imported ontology file is highlighted in the Containment tree

5.20.2 Define an Annotation Property

To define an annotation property for a property:

1. Create a UML property.
2. Open the property's Specification by double-clicking the property in a diagram or the Containment tree.
3. In the Specification window, select **Annotation Property** under the **Applied Stereotype** option.
4. Click **Apply**.
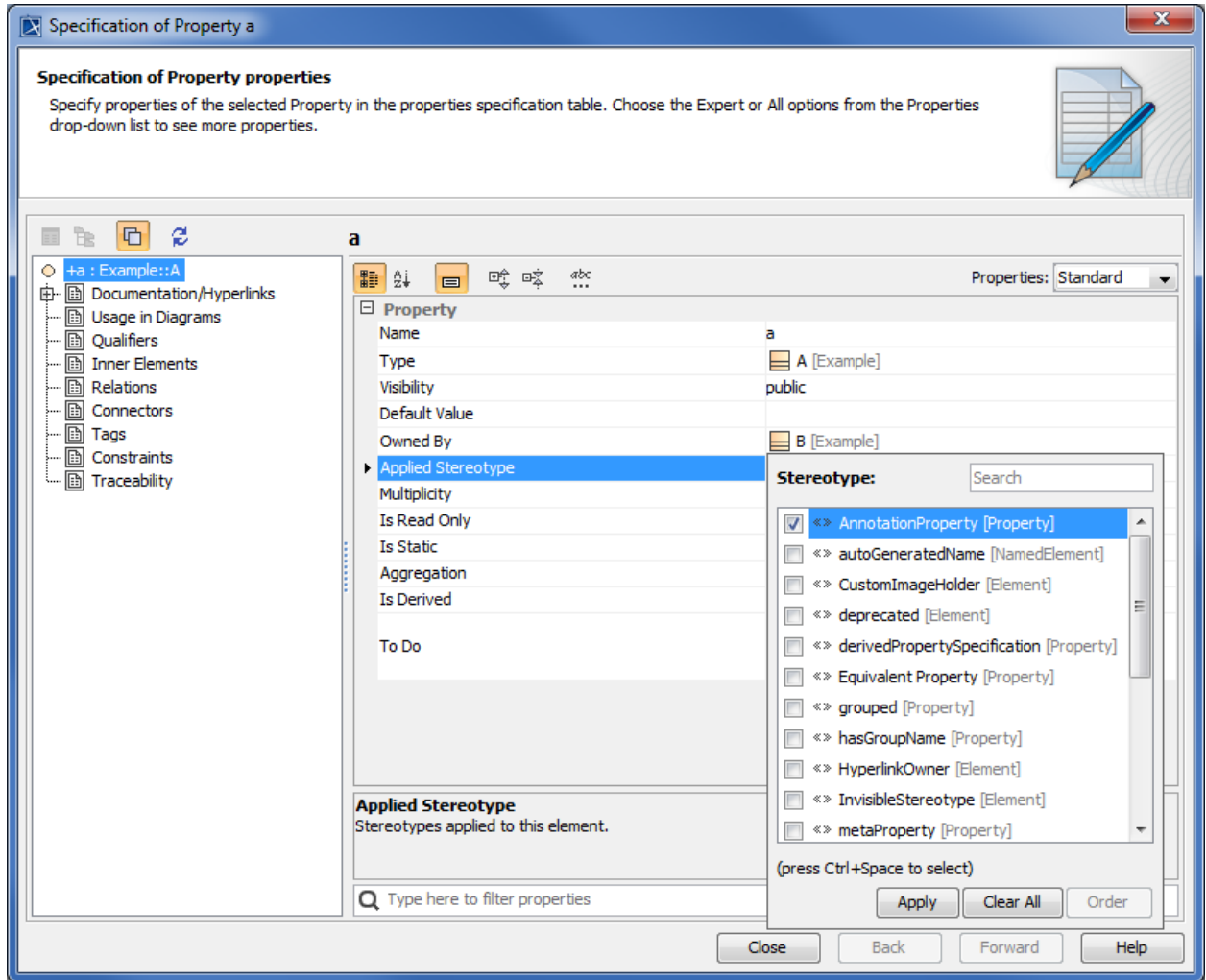
Figure 175   Applying an Annotation Property stereotype to a property

5.20.3 Apply an Annotation Stereotype

To apply an annotation stereotype to a comment:

1. Create a UML Comment containing whatever text you like, and anchor it to the element to be annotated.
2. Double-click the Comment to open its specification.
3. In the specification window, select **Annotation** under the **Applied Stereotype** option.
4. Click **Apply**.
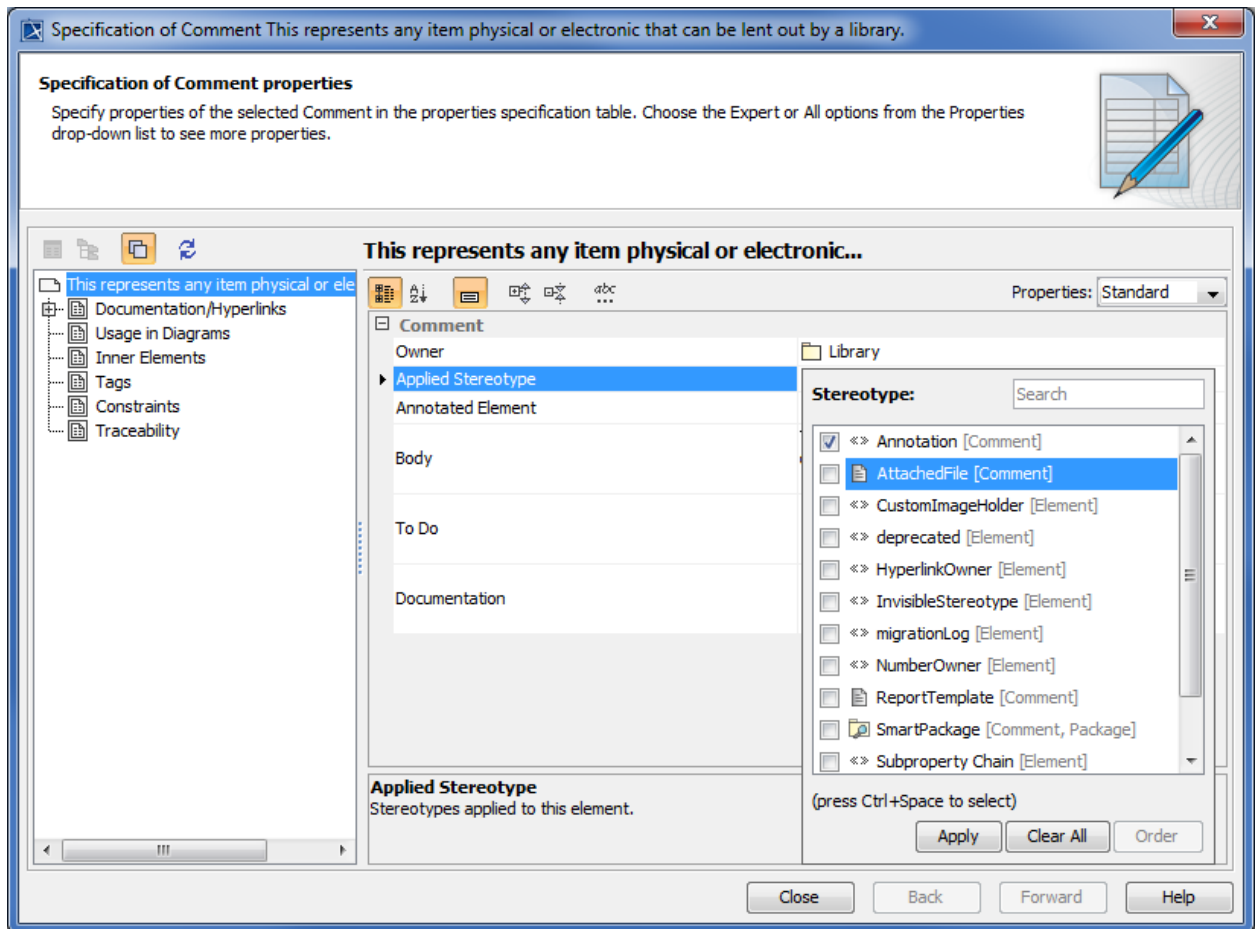
Figure 176   Applying Annotation stereotype to a comment

5.20.4 Associate an Annotation Property with an Annotation

The UML comment specification dialog allows you to select a particular kind of annotation property for each annotation.

To select a type of annotation property for an annotation:
1. Double-click an annotation on the diagram pane. The Specification window of the selected annotation will open (see the following figure).

Copyright © 2017, No Magic, Inc.
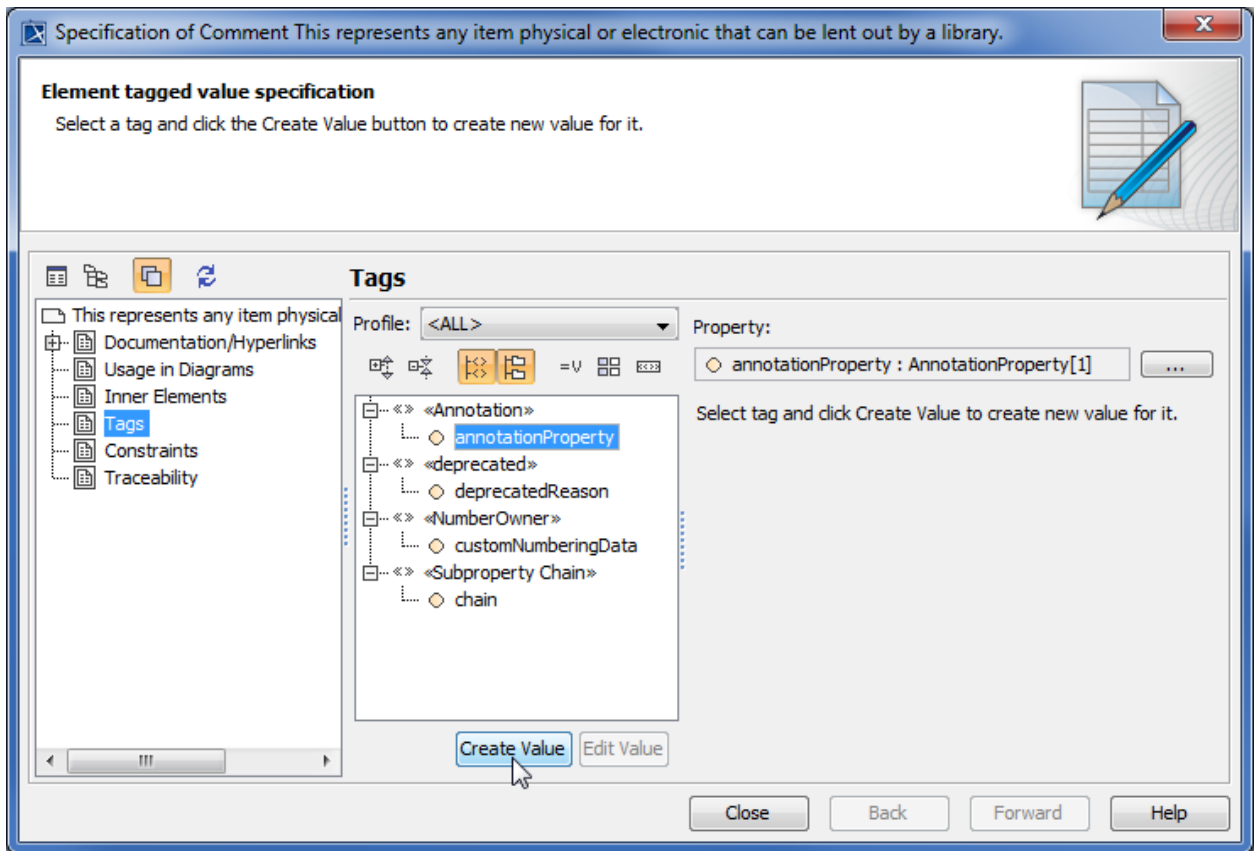
Figure 177   Selecting an annotation property tagged value

2. Select **Tags** on the left-hand side list and select **annotationProperty**.
3. Click **Create Value**. The **Select Property** dialog will open (see the following figure).
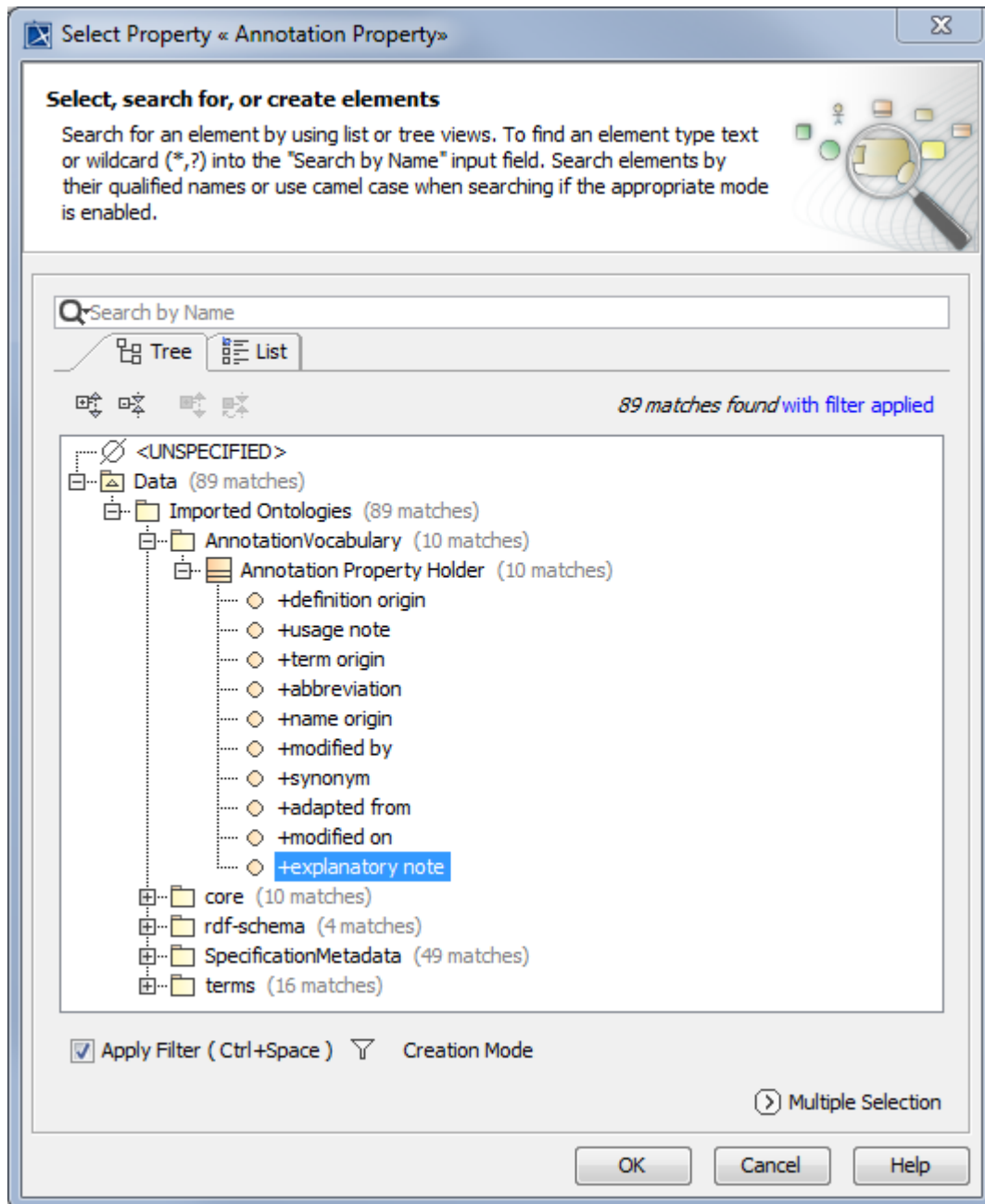
Figure 178   Selecting explanatory note as the annotation property tagged value

Copyright © 2017, No Magic, Inc.

4. Select an annotation property and click **OK**. The selected annotation property will be created for the annotation (see the following figure).
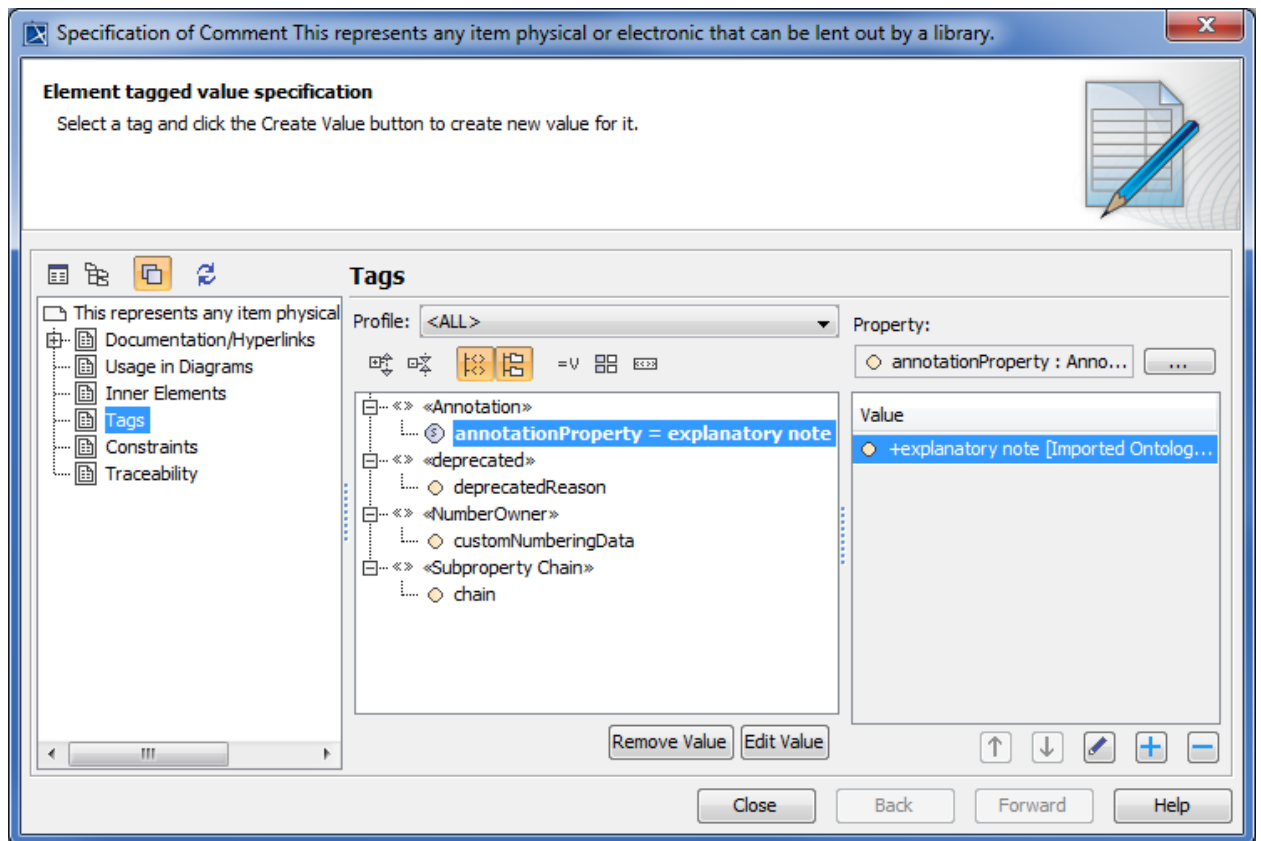
Figure 179   Types of annotation property available in the Specification window

In this example (see the preceding figure), the annotation property ***explanatory note*** is a UML property stereotyped with «Annotation Property».

5.20.5 Show Annotations on the Diagram

A diagram may contain annotations for a class or a property. They may not appear on the diagram pane, but you can see them in the Containment tree. The following steps will show you how to make them appear on the diagram.

To show an annotation(s) for a class on a diagram:
1. Right-click a class and select **Related Elements** > **Display Related Elements**.
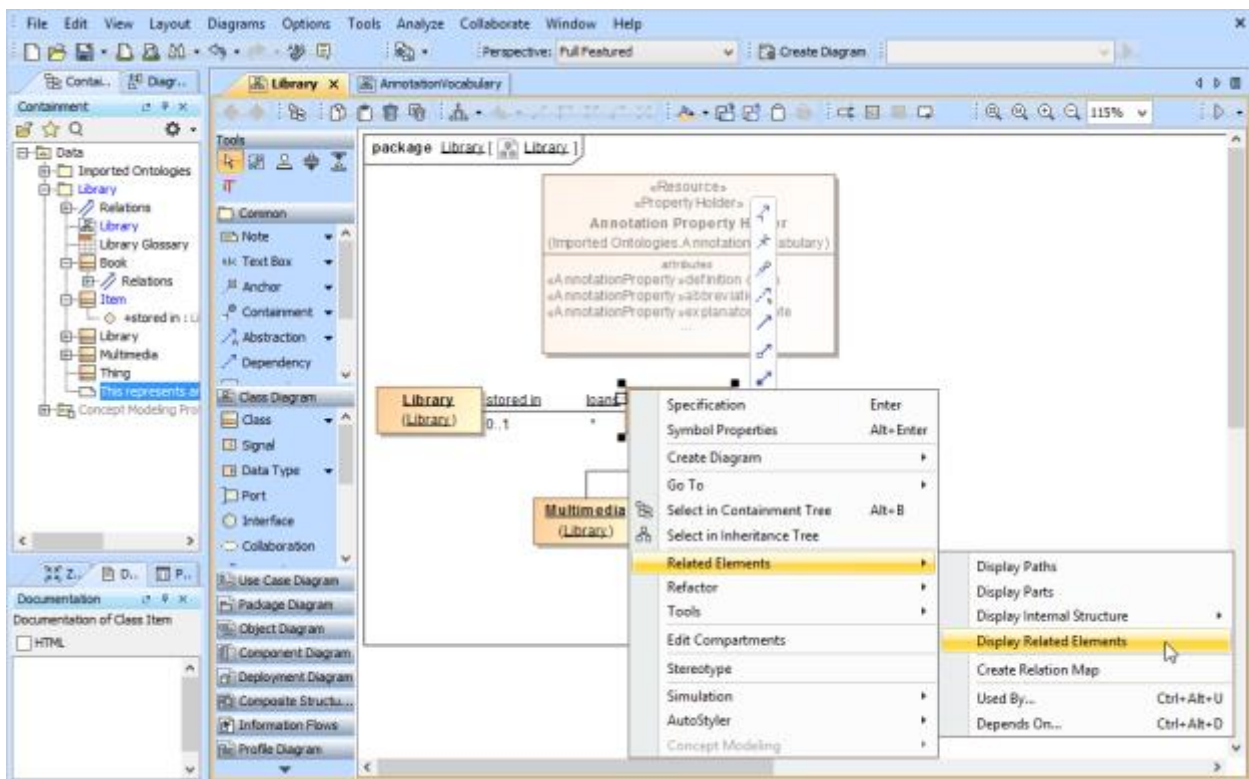
Figure 180   The Display Related Elements menu

2. In the **Display Related Elements** dialog, select **Comment** > **OK** (see the following figure). The annotation(s) for the class will appear on the diagram.
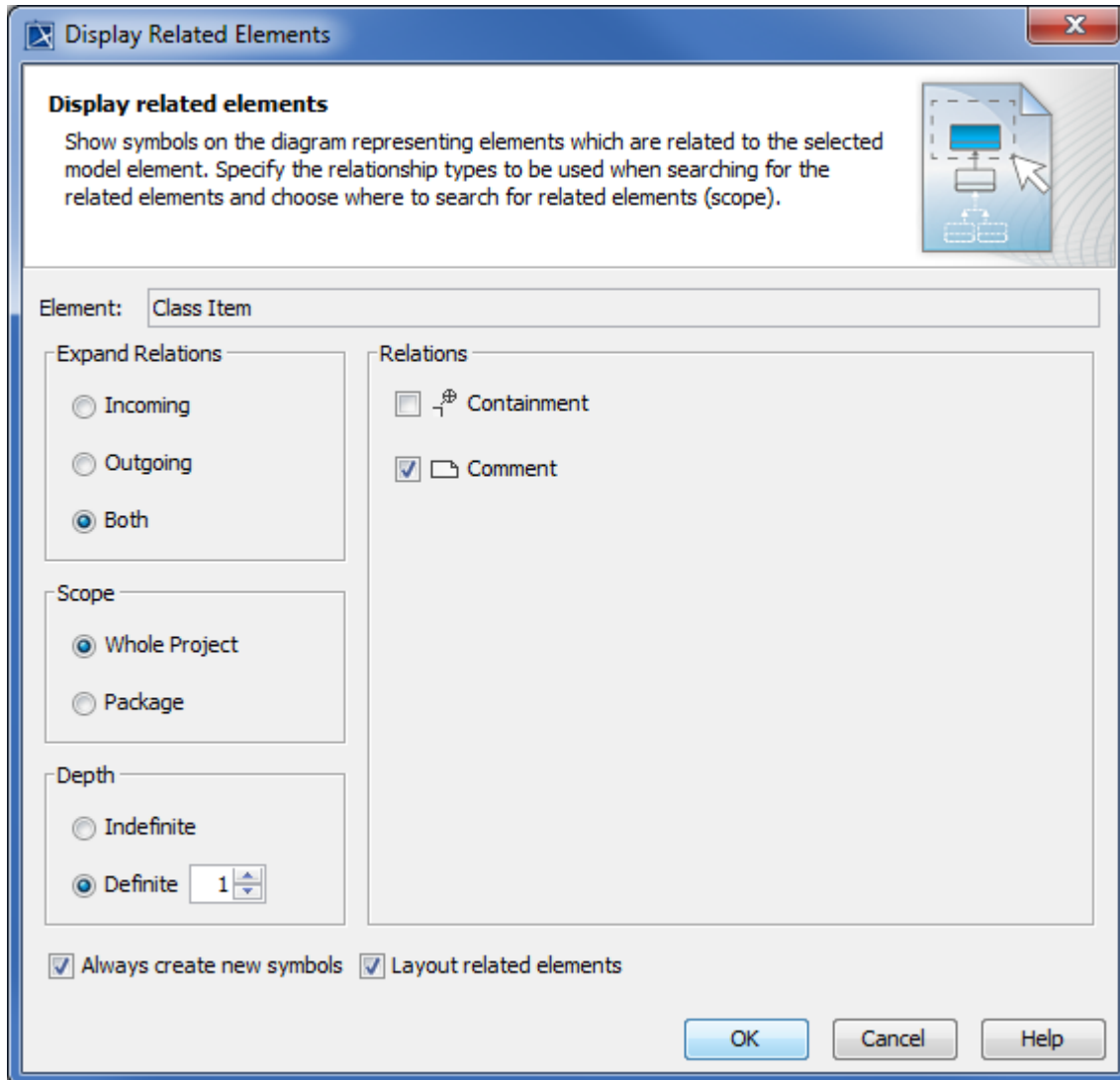
Figure 181   The Display Related Elements dialog

To show an annotation(s) for a property on a diagram:

1. Drag an annotation(s) from the Containment tree to the diagram pane.
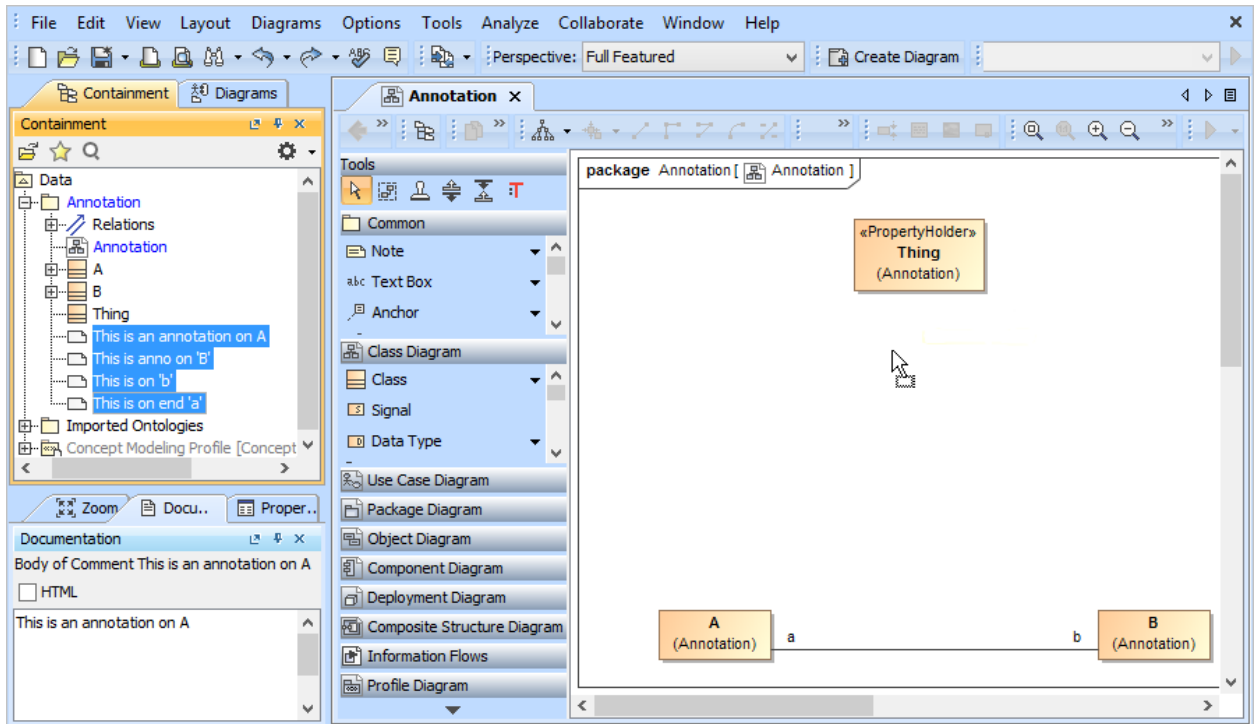
Figure 182   Dragging annotations from the Containment tree to the diagram pane

2. Right-click the annotation(s) and select **Related Elements** > **Display Paths** (see the following figure). A question dialog will open.
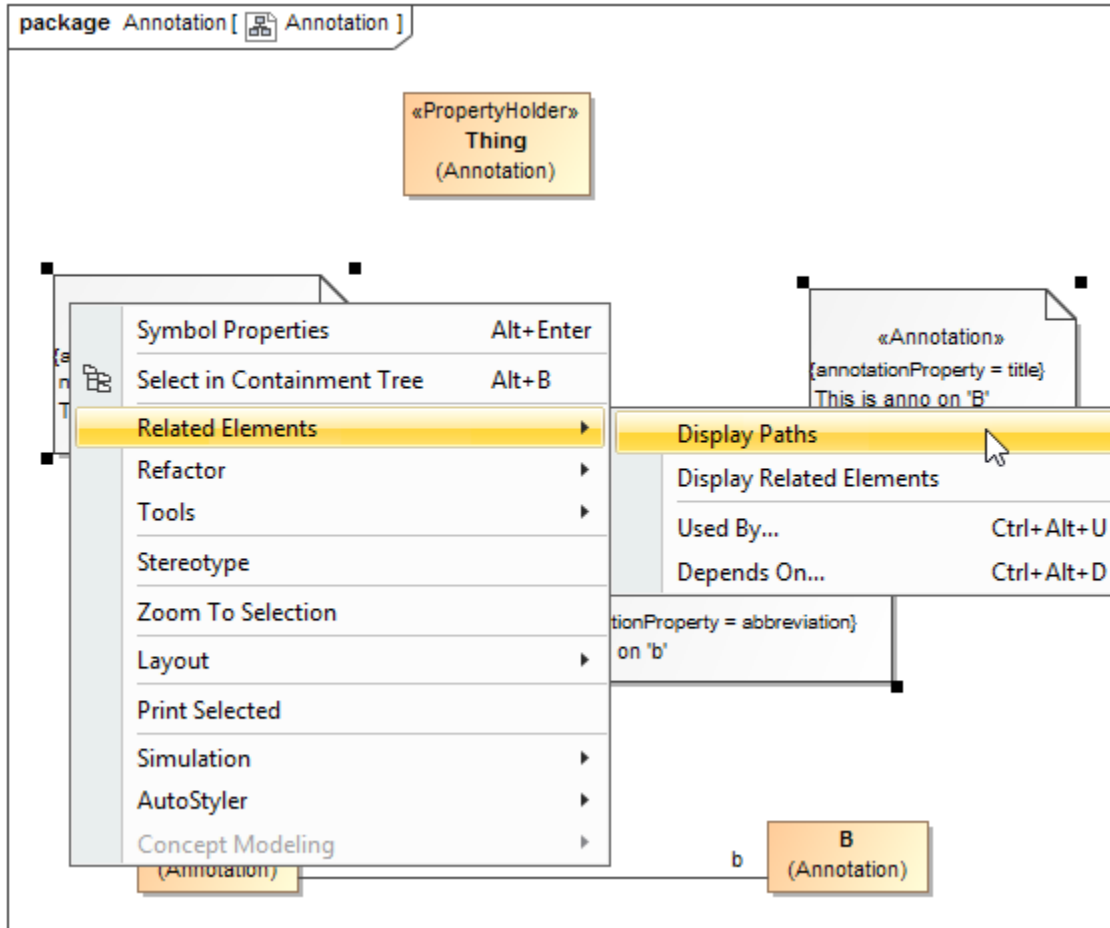
Figure 183   The Display Paths menu item

3. Click **No**.
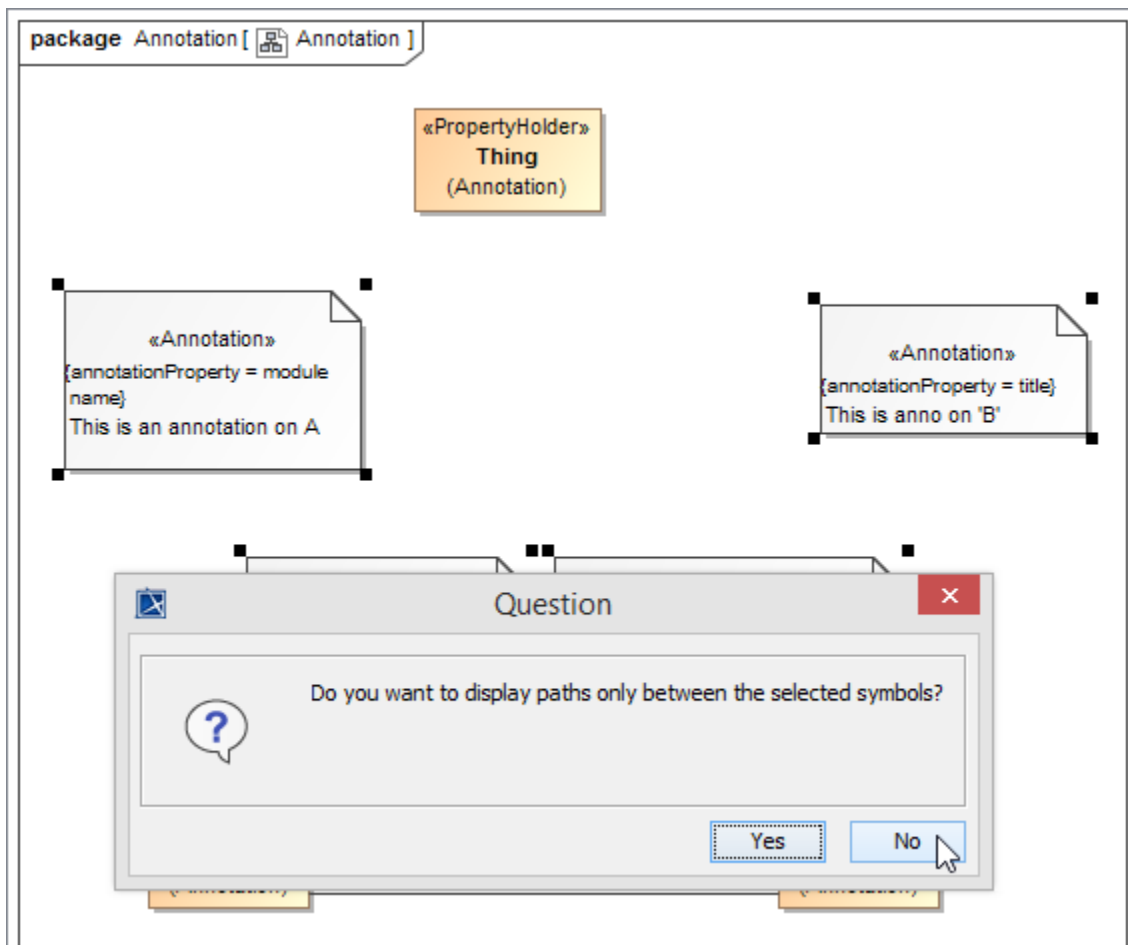
Figure 184   The Display Related Elements menu of a property

The annotation(s) for the property(ies) will appear on the diagram (see the following figure).
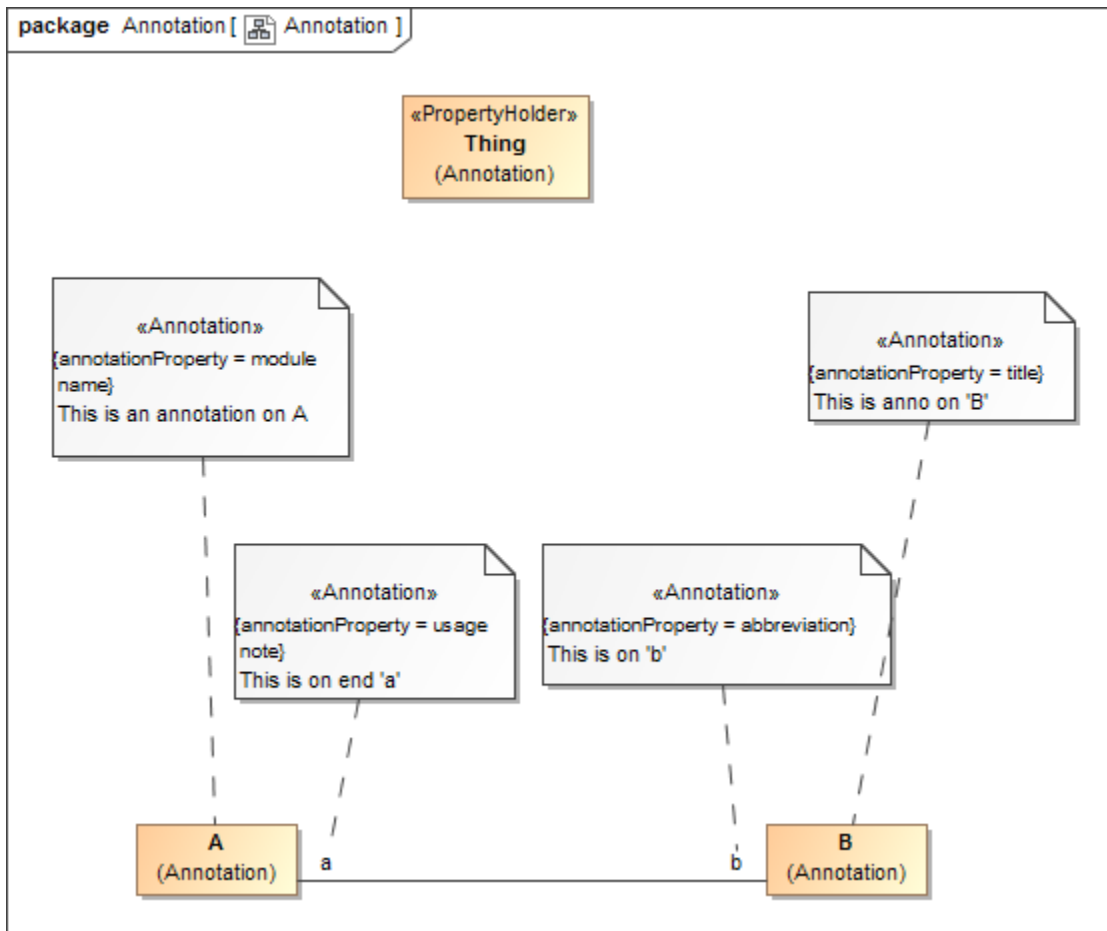
Figure 185   The annotations for the properties showing on the diagram

There are several ways to make an annotation for a class or a property appears in the Documentation pane any time you click the annotation in the Containment tree or in the diagram pane.

To show an annotation for a class in the Documentation pane when you click the class:
1. Drag the annotation (either on the diagram or in the Containment tree) to the class. (This makes it owned by the class.)
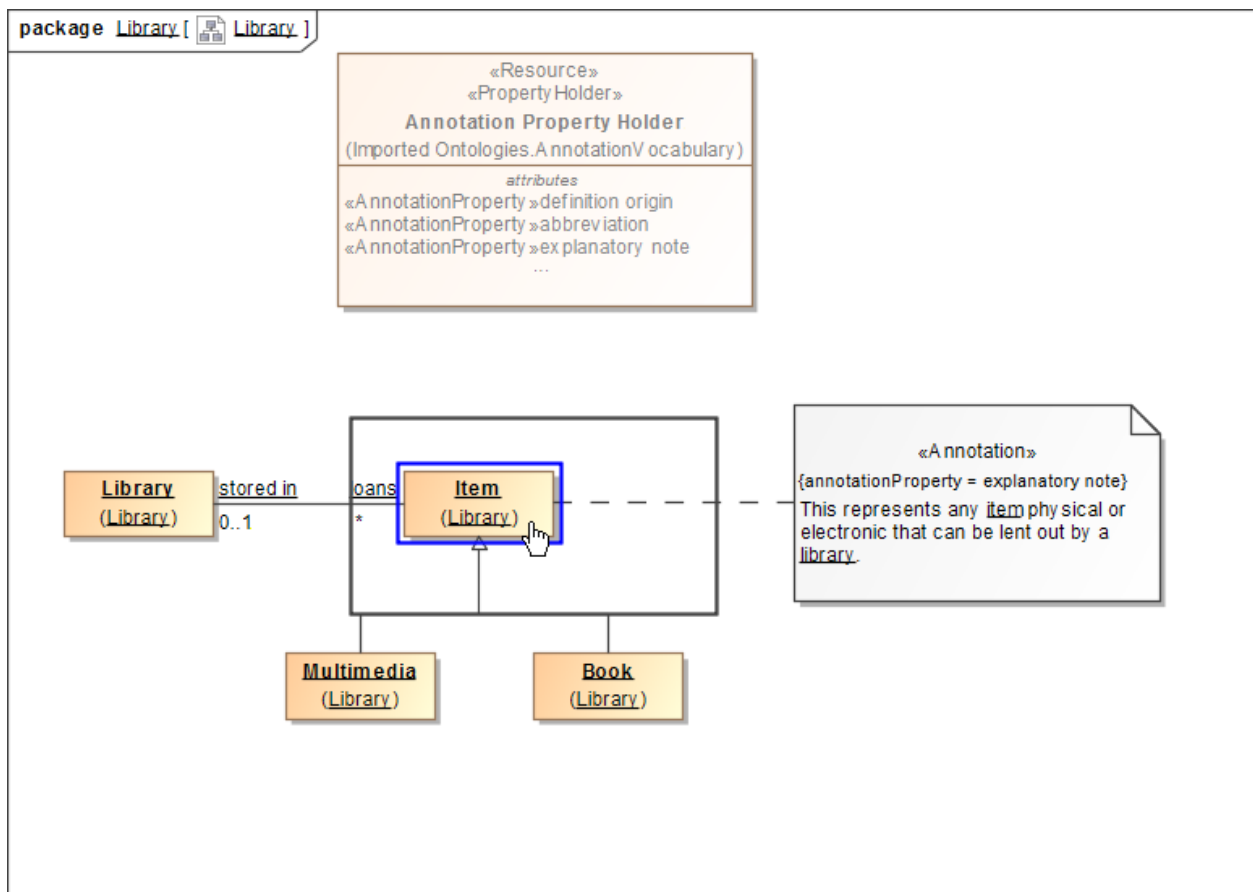
Figure 186   Dragging an annotation to a class

Copyright © 2017, No Magic, Inc.

2. Click the class. The annotation will show up in the **Documentation** pane.
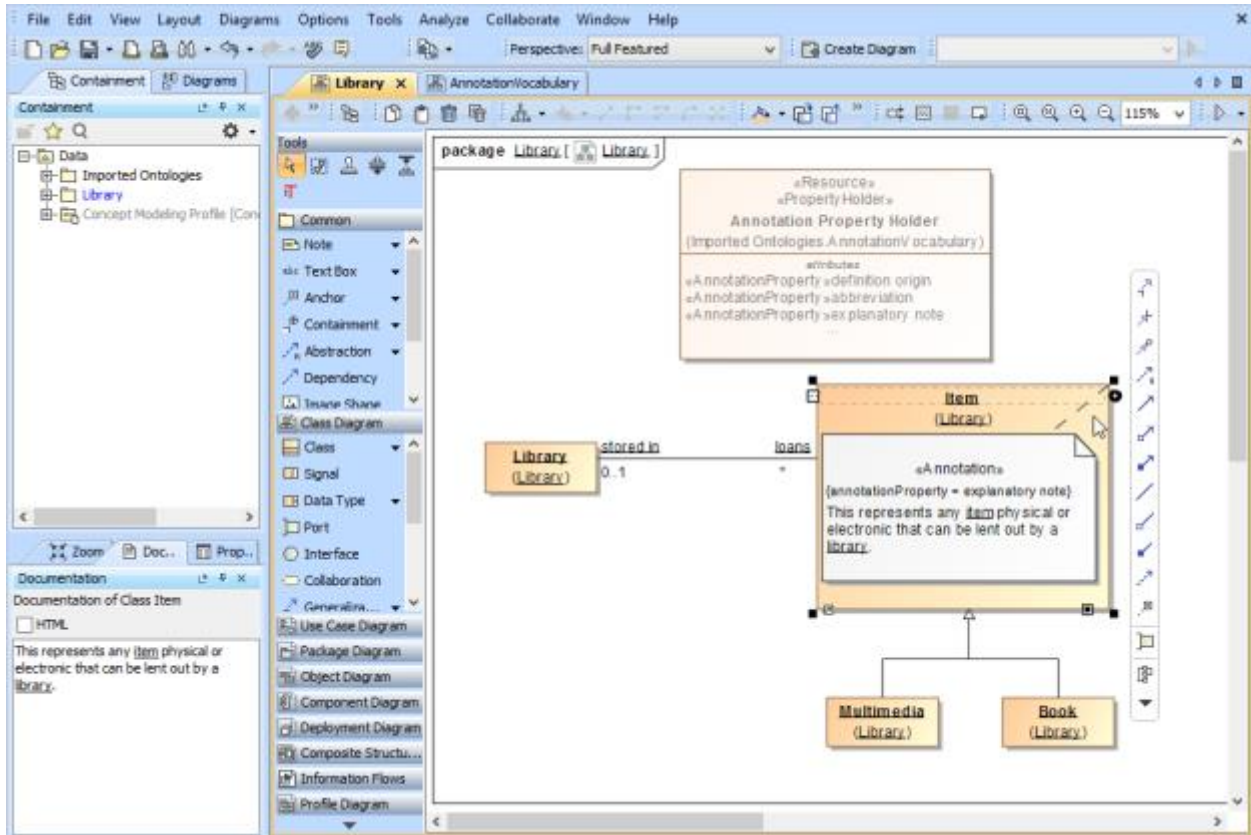
Figure 187   The annotation owned by the class shows in the Documentation pane

<u>To make an annotation for a property appear in the Documentation pane when you click the property:</u>

1. Double-click an annotation in the Containment tree to open its **Specification** dialog.
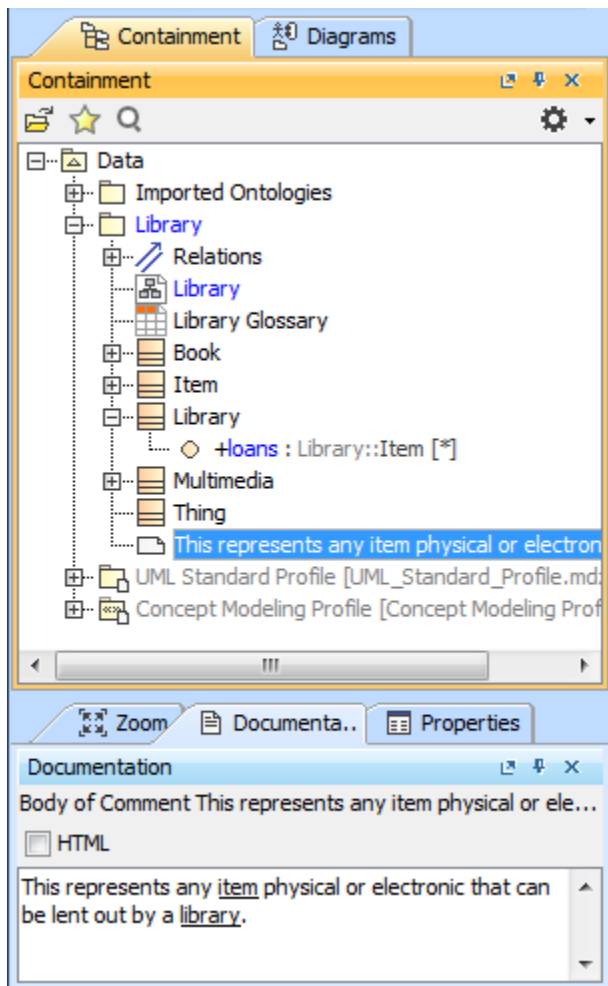
Figure 188   An annotation in the Containment tree

2. In the **Specification** dialog, click [...] next to **Annotated Element**. The **Select Elements** dialog will open.
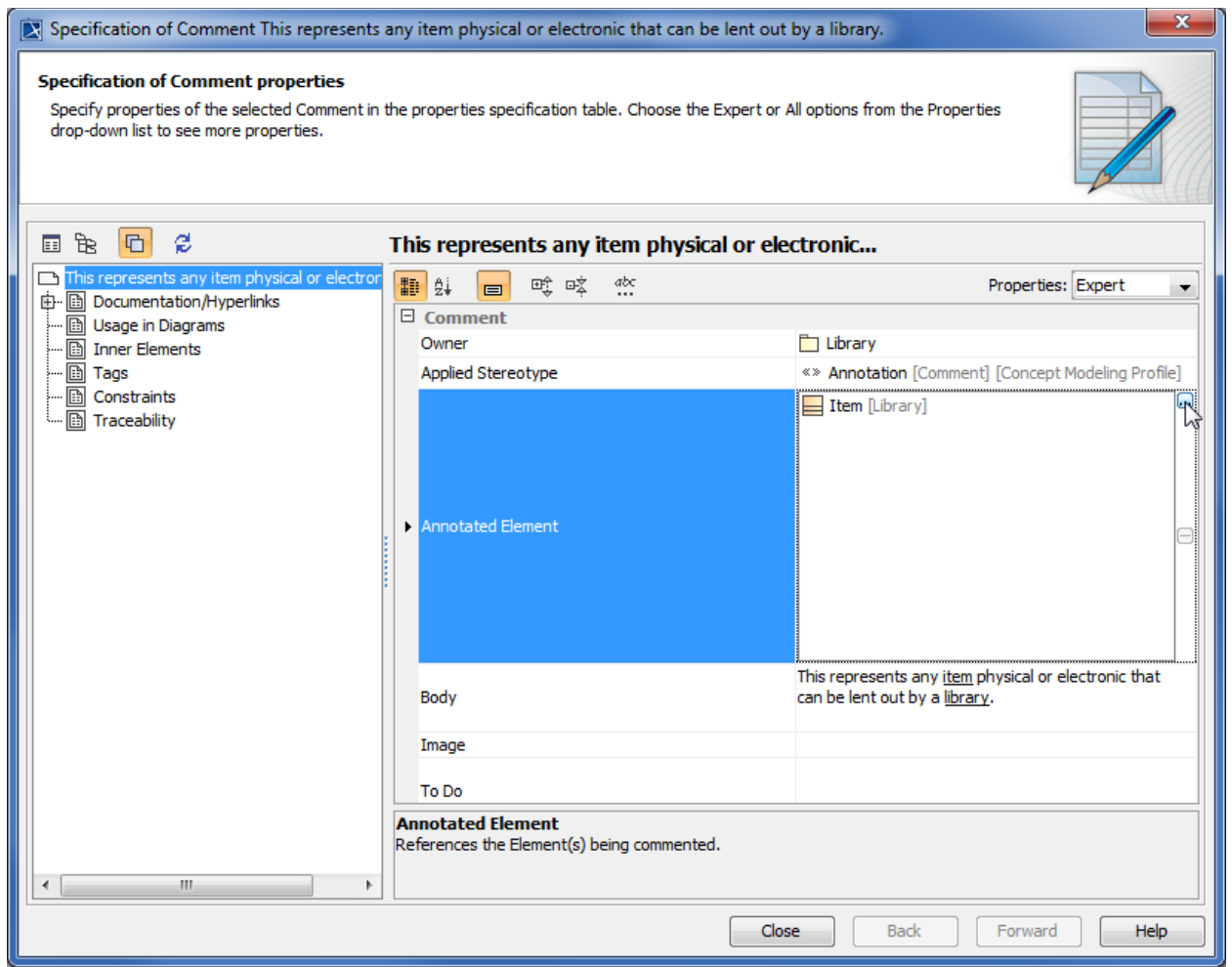
Figure 189   The Specification dialog of a selected annotation

3. Remove the current element, in this example, **Item**, from the **Selected elements** pane by selecting it and click [—].

4. Select a property that will be the new annotated element of the annotation from the Tree view list, for example, **loans**, and click [+].
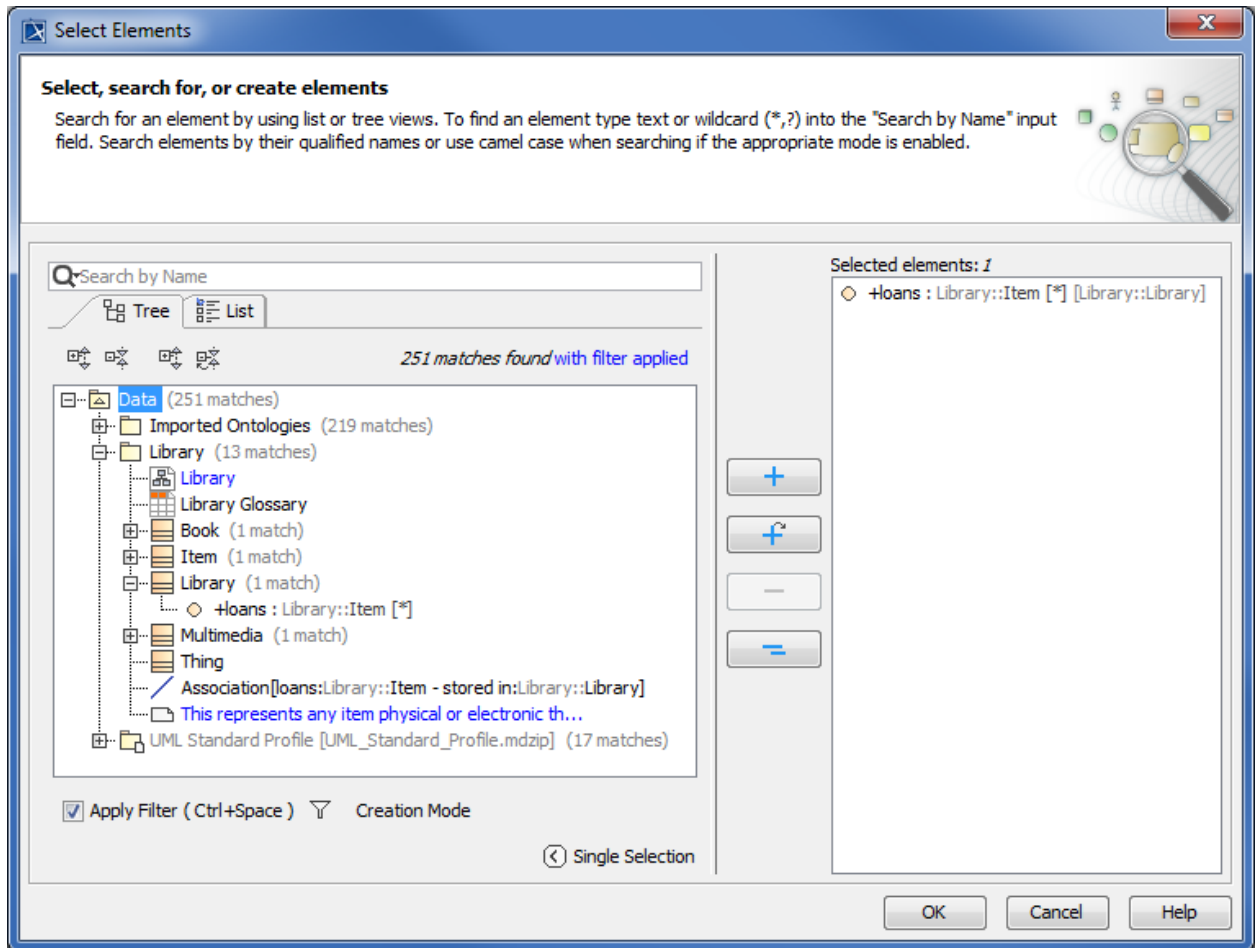

Figure 190   Selecting annotated element for an annotation in the Select Elements dialog

5. Click **OK**. The Annotated Element of the annotation has been changed, in this example, from **Item** to **loans**.
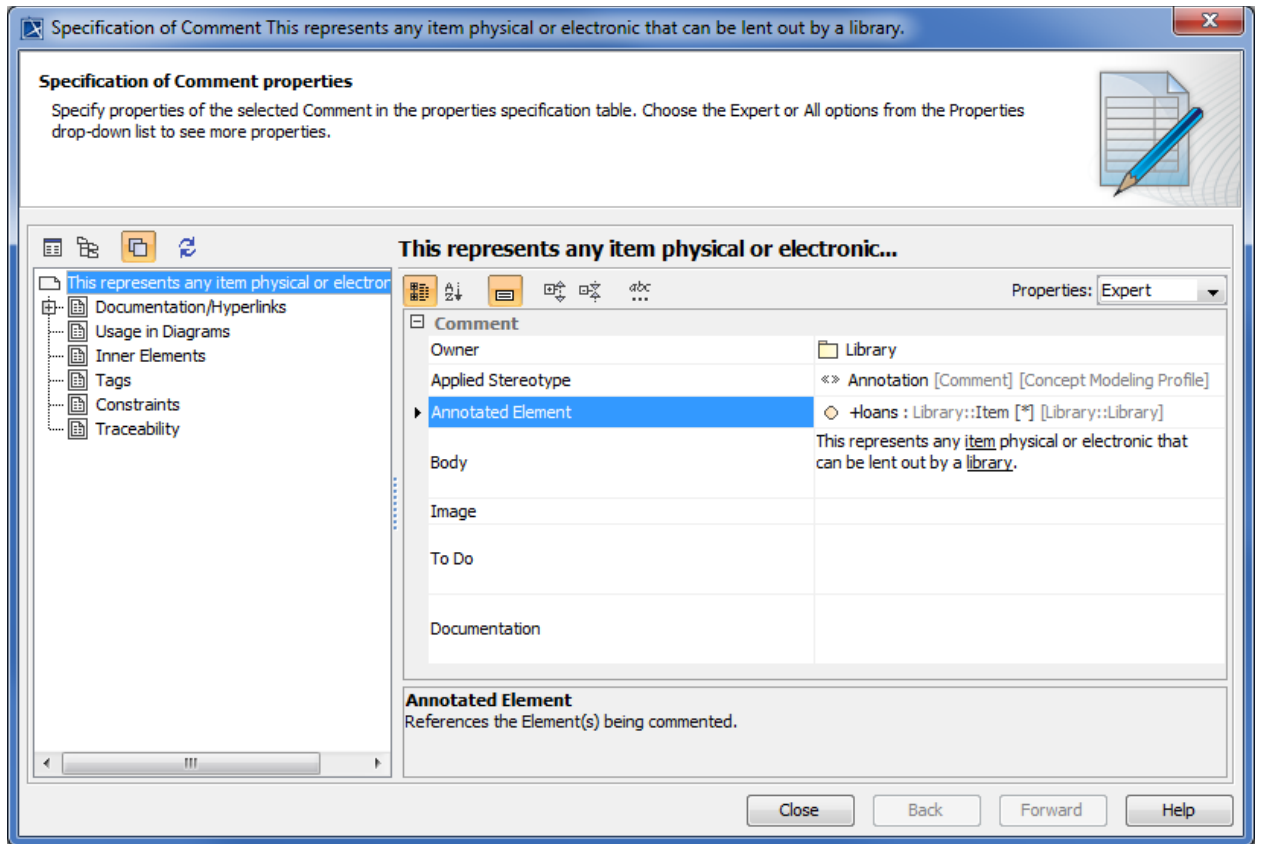

Figure 191   The annotated element for the annotation is set to the selected property

6. Click **Close** to close the **Specification** dialog.
7. On the main menu, click **Options** > **Project** to open the **Project Options** dialog.
8. Click **General** > **Concept Modeling**.
9. Click ... next to **Preferred annotation property**.
10. Select the preferred annotation property tagged value (the value must be the same as that of the selected annotation), for example, **explanatory note**, and click **OK**. The **Preferred annotation property** tagged value is now **explanatory note**.
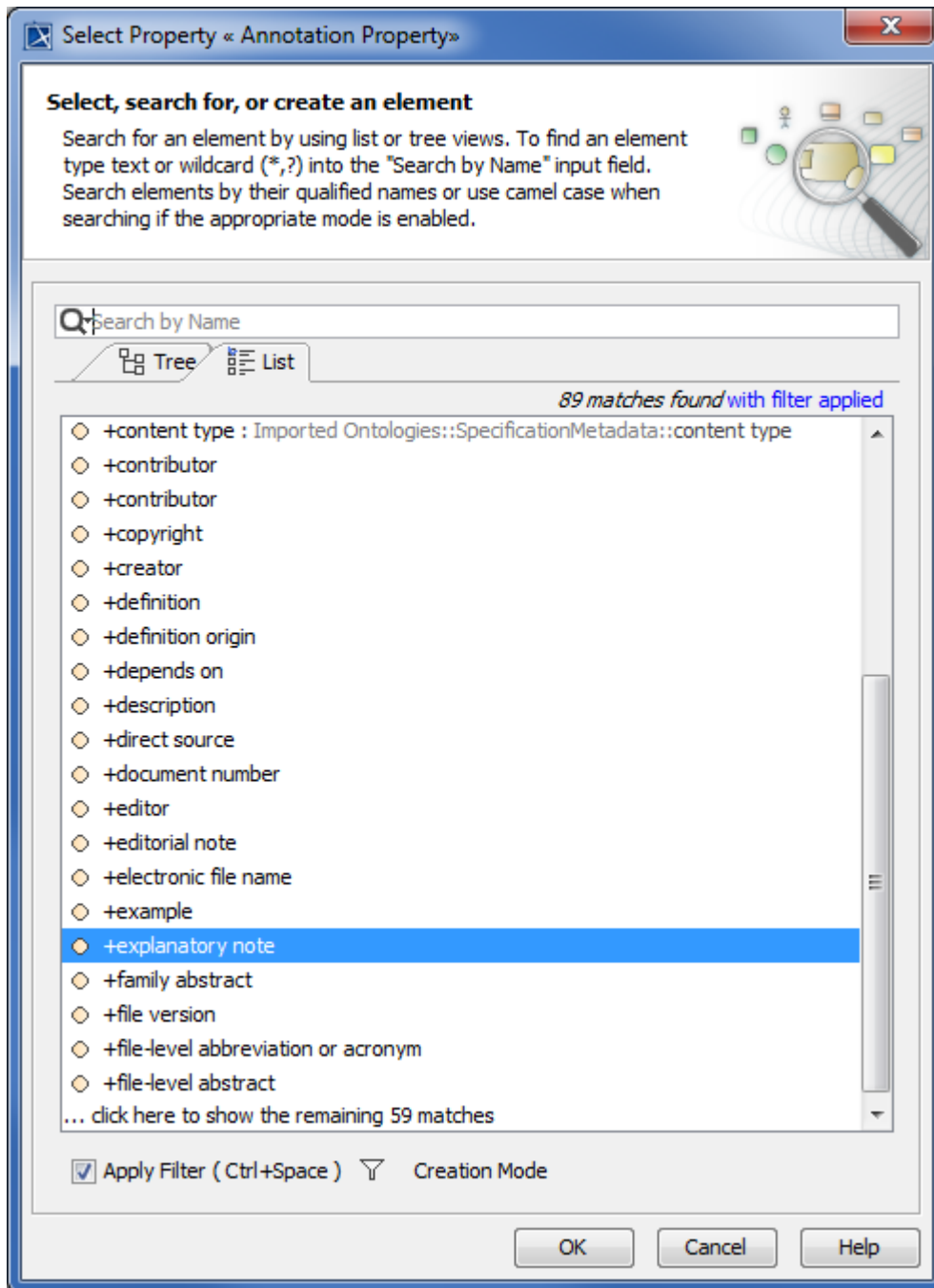

Figure 192   Selecting the preferred annotation property tagged value

Copyright © 2017, No Magic, Inc.

11. Click **OK** to close the **Project Options** dialog. The annotation will be moved to the property **loans**. Any time you click **loans** in either the Containment tree or the diagram pane, the annotation will appear in the **Documentation** pane.
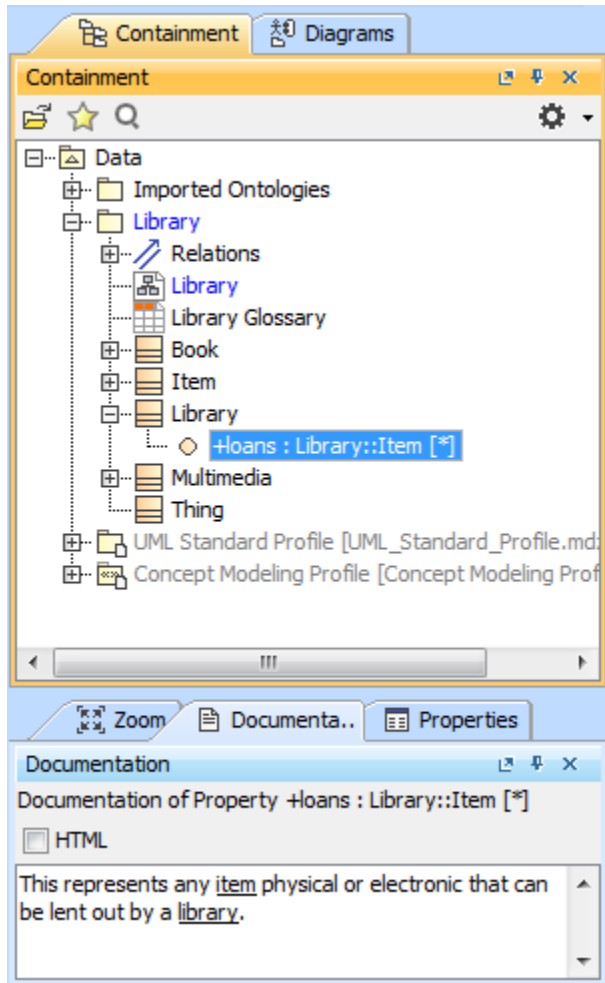

Figure 193   The annotation for the property shows in the Documentation pane

Note	• If you drag more than one annotation to a class or a property, only the first created annotation will appear in the Documentation pane and in the class' **Specification** window (under the **Documentation/Hyperlink** property).
	• An annotation for a property will appear in the **Documentation** pane only if its annotated element is set to the property and its preferred annotation property tagged value is specified or updated.
	• In this current release, only annotations that have been adjusted to show in the **Documentation** pane will appear in the Natural Language Glossary.

You can also create a new property and a new annotation for the property, and make the annotation appears in the Documentation any time you click the property in either the Containment tree or the diagram pane. The *Library loans.mdzip* sample is used in the following instructions.

5.20.7 Select a Preferred Annotation Property for a UML Comment or «Annotation»

To select a preferred annotation property tagged value for an existing «Annotation» of an element:

1. With your project open, on the main menu, click **Options** > **Project**. The **Project Options** dialog will open.
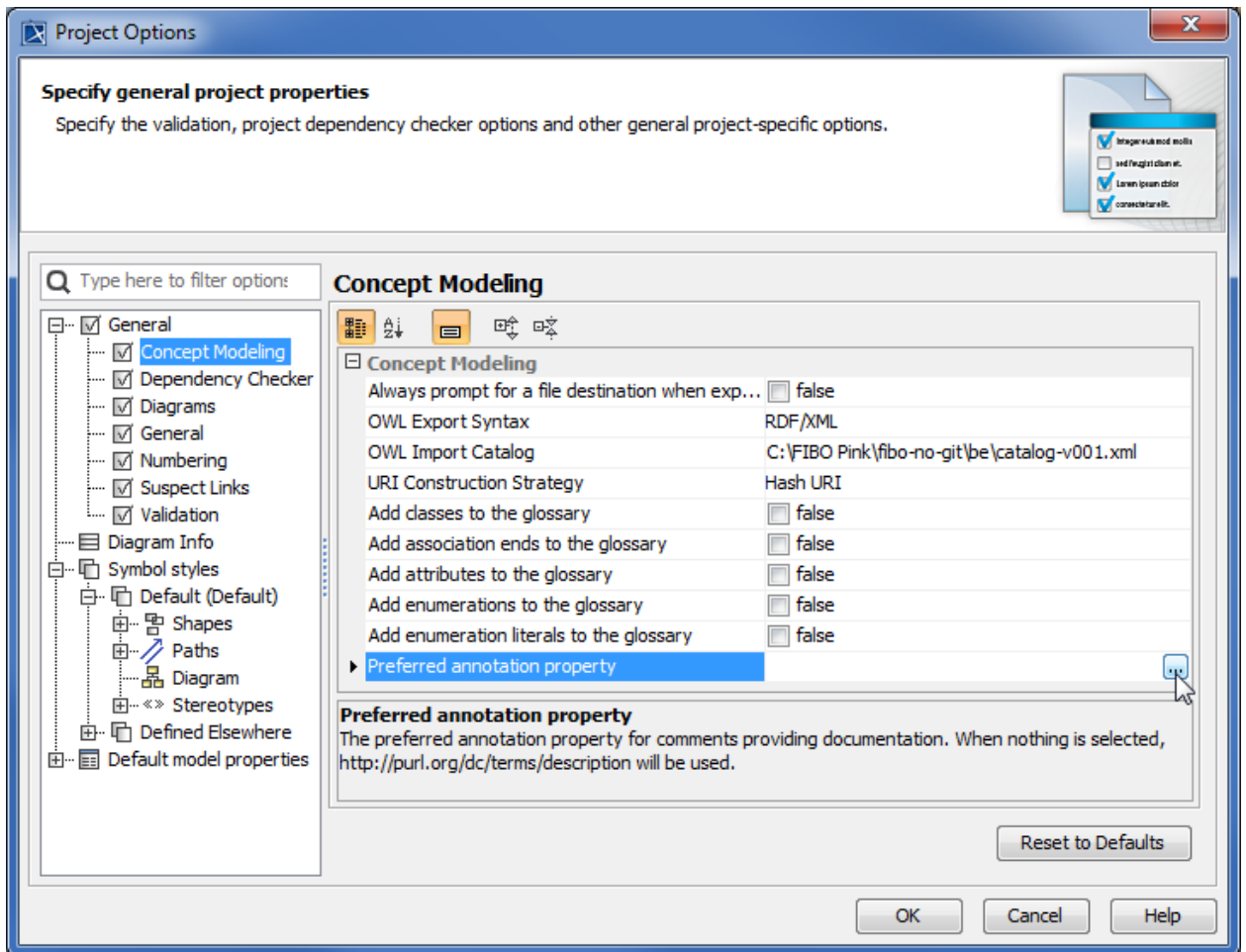2. Select **General** > **Concept Modeling** (see the following figure).

Figure 194   The Preferred annotation property option in the Project Options dialog

Copyright © 2017, No Magic, Inc.

3. Click ![icon]. The **Select Property «Annotation Property»** dialog will open (see the following figure).
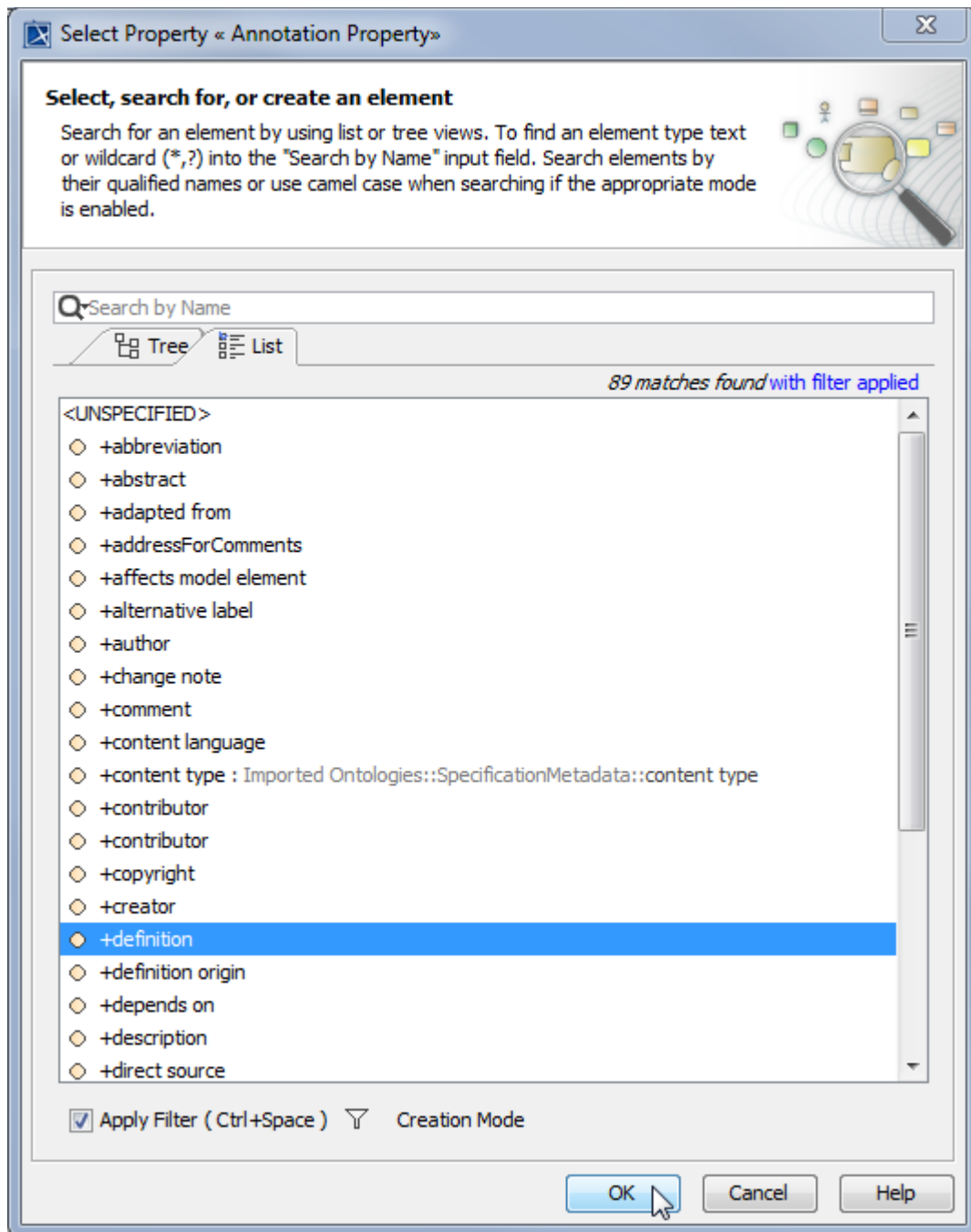

Figure 195   Selecting a preferred annotation property

4. Click the **List** tab and select an annotation property for the comments, for example, **definition**.
5. Click **OK** > **OK**. The selected annotation property tagged value **definition** will be made as the current preferred annotation property for all comments/annotations in your model (see the following figure).
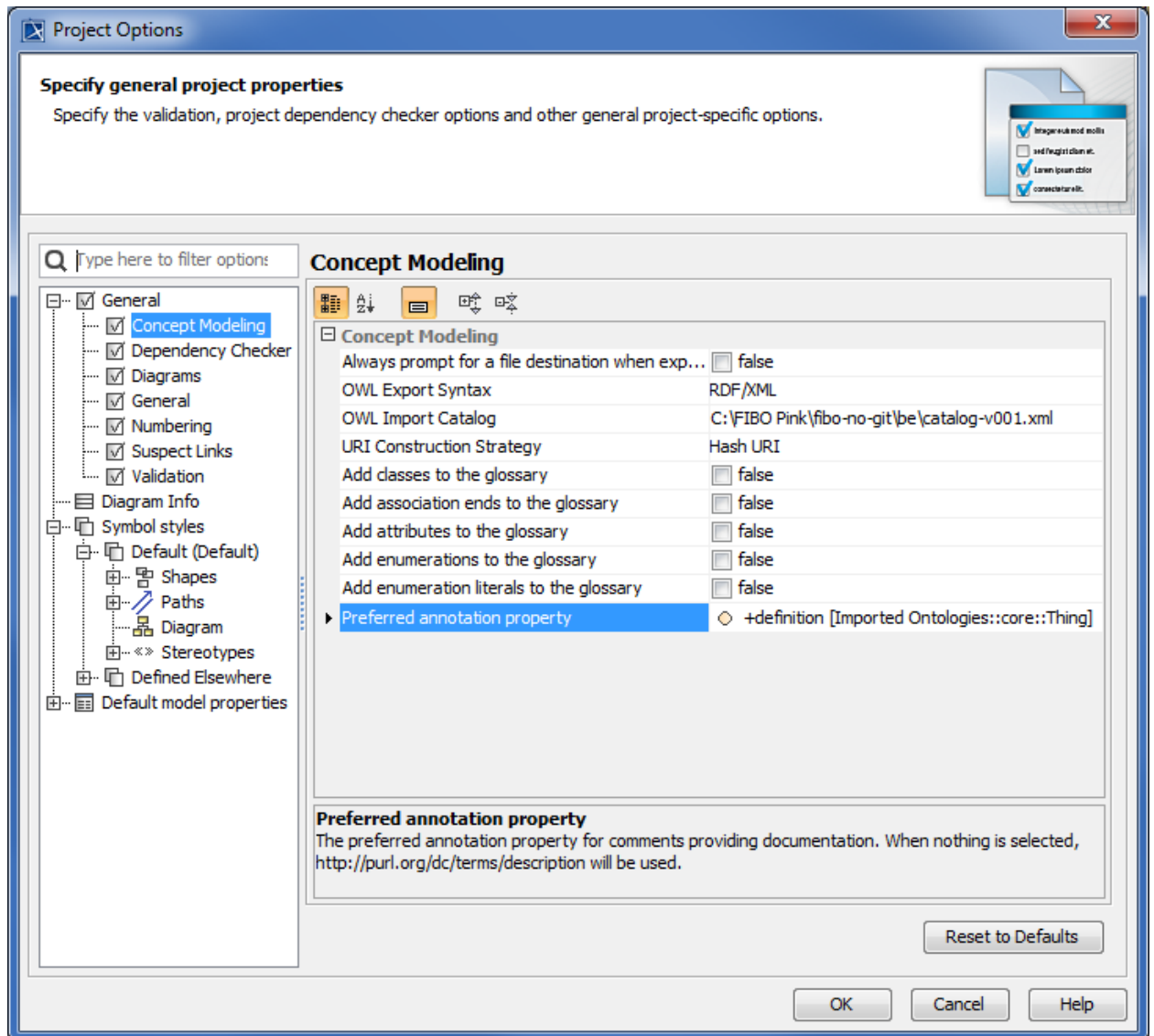

Figure 196   The selected preferred annotation property

After clicking OK, a progress bar will appear. If your project is a TWC project, Concept Modeler will attempt to lock the project's elements. If any of the elements cannot be locked, whether it is locked by another user, then the dialog box with the OK button will say "Cannot lock all elements for edit to allow preferred annotations to be used as element documentation. You may refer to the Lock View tab to see what still needs to be locked." Furthermore, an additional message will appear in the notification window saying "The preferred annotation property

change has been reverted." After these two messages will appear, the preferred annotation property will revert back to its previous value.

So after they click OK, they should see progress bar and if it is TWC project, concept modeler will try and lock elements, then you need to mention what happens if locks cannot be acquired.

The following example shows you how to change the tagged value **definition** to an unspecified preferred annotation property.

To change a current preferred annotation property tagged value to an unspecified preferred annotation property tagged value:

1. On the main menu, click **Options** > **Project**. The **Project Options** dialog will open.
2. Select **General** > **Concept Modeling.**
3. Click next to **Preferred annotation property**. The **Select Property «Annotation Property»** dialog will open.
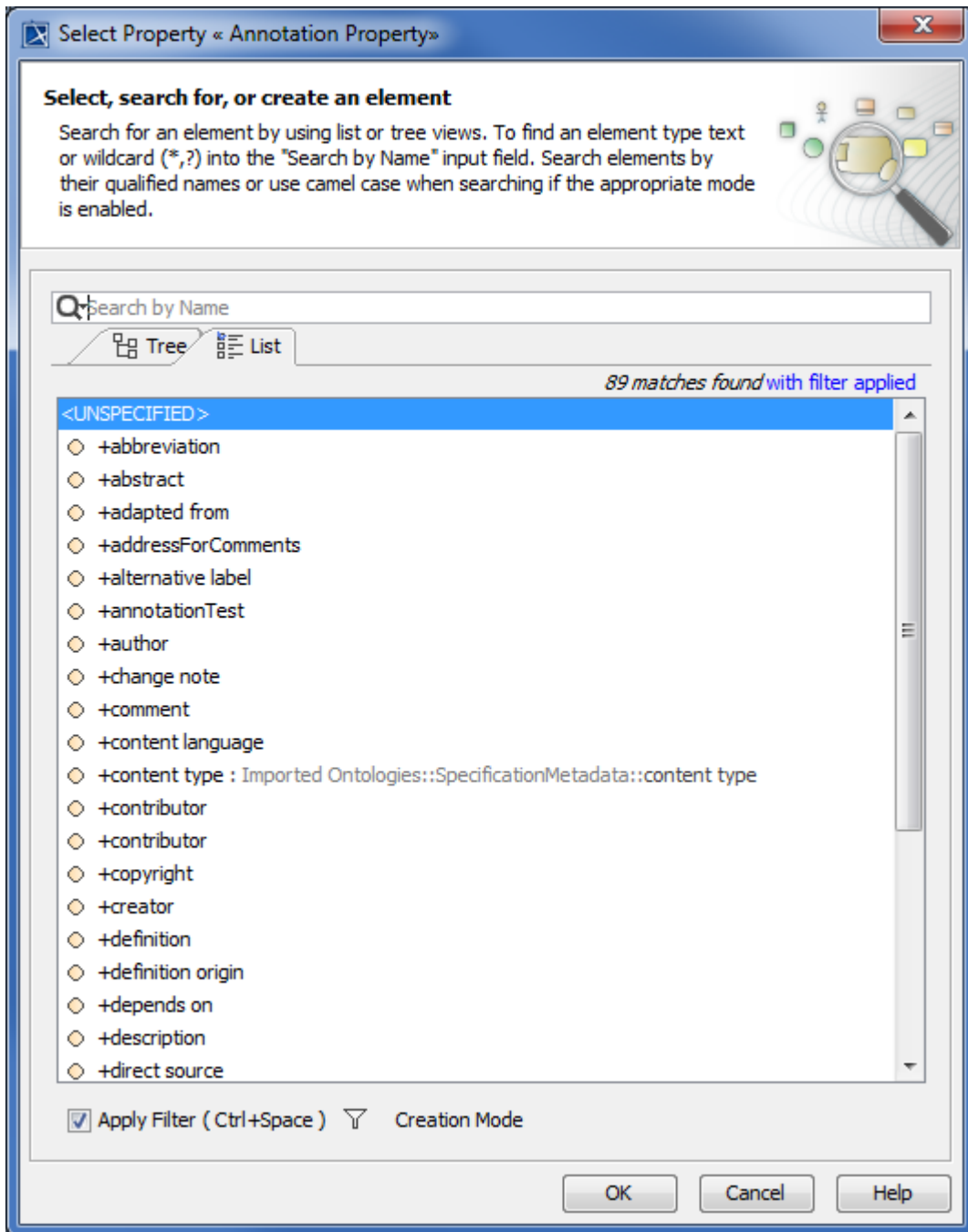4. Select **<UNSPECIFIED>** and click **OK** (see the following figure).

Figure 197   Selecting the default <UNSPECIFIED> preferred annotation property

5. Click **OK**. The **definition** tagged value will be removed from **Preferred annotation property** box and the annotation will be moved back under the owning folder, in this example, it is the package **Agents.**

You can also add annotation properties manually in your Concept Modeling project. The following instructions show you how to create an annotation property in your model.
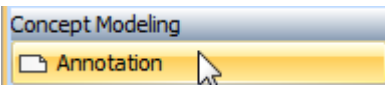
To add documentation to your model by using the Documentation pane:

1. Right-click the **Data** package in the Containment tree and select **Concept Modeling** > **Create Concept Model** to create a new concept model.
2. Create a property under the Anything and name it, for example, annotationTest.
3. Right-click the property and select Annotation Property as its stereotype.
4. Create a class.
5. Click it and click the **Documentation** pane.
6. Type, for example, This is a test class.
7. Right-click the class and select **Display** > **Display Related Elements**. The **Display Related Elements** dialog will open.
8. Select the **Comment** check-box.
9. Clear the **Always create new symbol** check-box.
10. Click **OK**.
11. The annotation "This is a test class." will appear above the class.

To add documentation to your model by using the Specification window:

1. Right-click the **Data** package in the Containment tree and select **Concept Modeling** > **Create Concept Model** to create a new concept model.
2. Create a property under the Anything and name it, for example, annotationTest.
3. Right-click the property and select Annotation Property as its stereotype.
4. Create a class.
5. Double-click it to open the **Specification** window.
6. Type in the **Documentation** pane, for example, This is a test class.
7. Right-click the class and select **Display** > **Display Related Elements**. The **Display Related Elements** dialog will open.
8. Select the **Comment** check-box.
9. Clear the **Always create new symbol** check-box.
10. Click **OK**.
11. The annotation "This is a test class." will appear above the class.

To add documentation to your model by using the Concept Modeling Diagram palette:

1. Right-click the **Data** package in the Containment tree and select **Concept Modeling** > **Create Concept Model** to create a new concept model.
2. Create a property under the Anything and name it, for example, annotationTest.
3. Right-click the property and select Annotation Property as its stereotype.
4. Create a class.
5. Drag from the diagram palette to the diagram pane to create an «Annotation».
6. Click the created «Annotation» in the diagram pane.

7. Type, for example, This is a test class, either in the **Documentation** pane in the bottom left of the screen or in the Annotation itself.

8. Click ![Anchor] from the diagram palette and click «Annotation» and the class. The documentation will be created. Any time you click the class, the documentation will appear in the **Documentation** pane.

5.21 Generate a Natural Language Glossary

To generate a Natural Language Glossary:

1. Select **Tools** > **Report Wizard** from the main menu.
2. Expand the **Concept Modeler** folder.
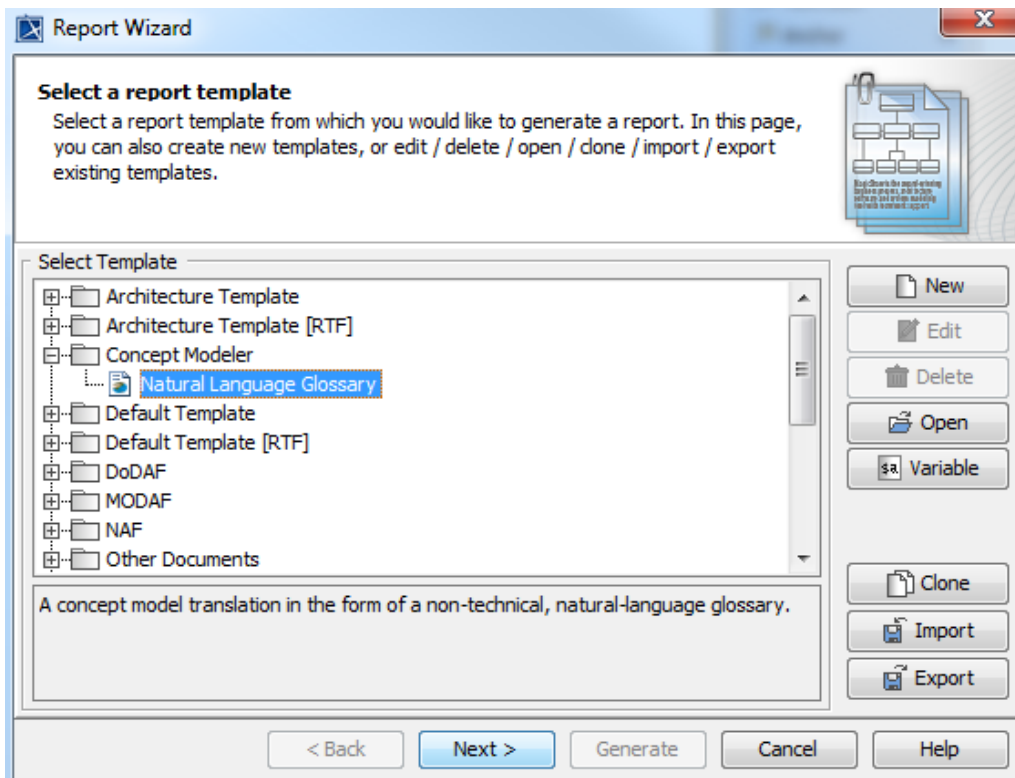3. Select **Natural Language Glossary**.

Figure 198   The Natural Language Glossary option in the Report Wizard dialog

4. Click the **Next** button.
5. Select **Built-in**.
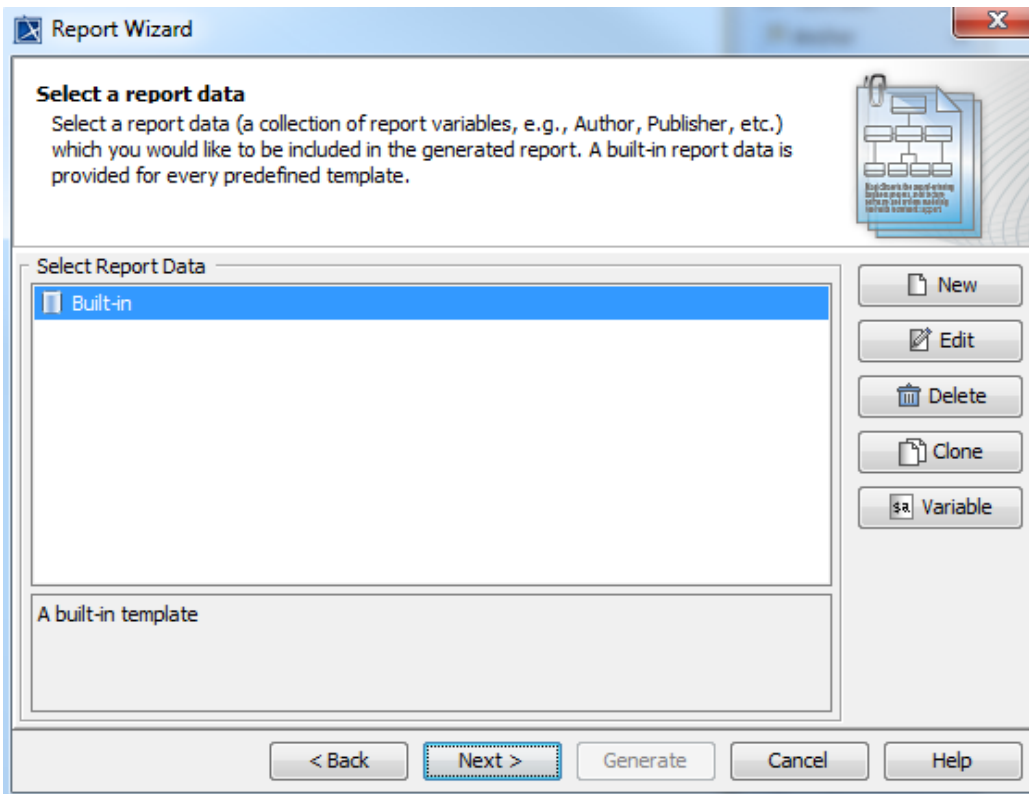
Copyright © 2017, No Magic, Inc.

Figure 199   Selecting the built-in glossary creation option

6. Click the **Next** button.
7. Select the package(s) you want to generate a natural language glossary for.
8. Click the **Add** button.
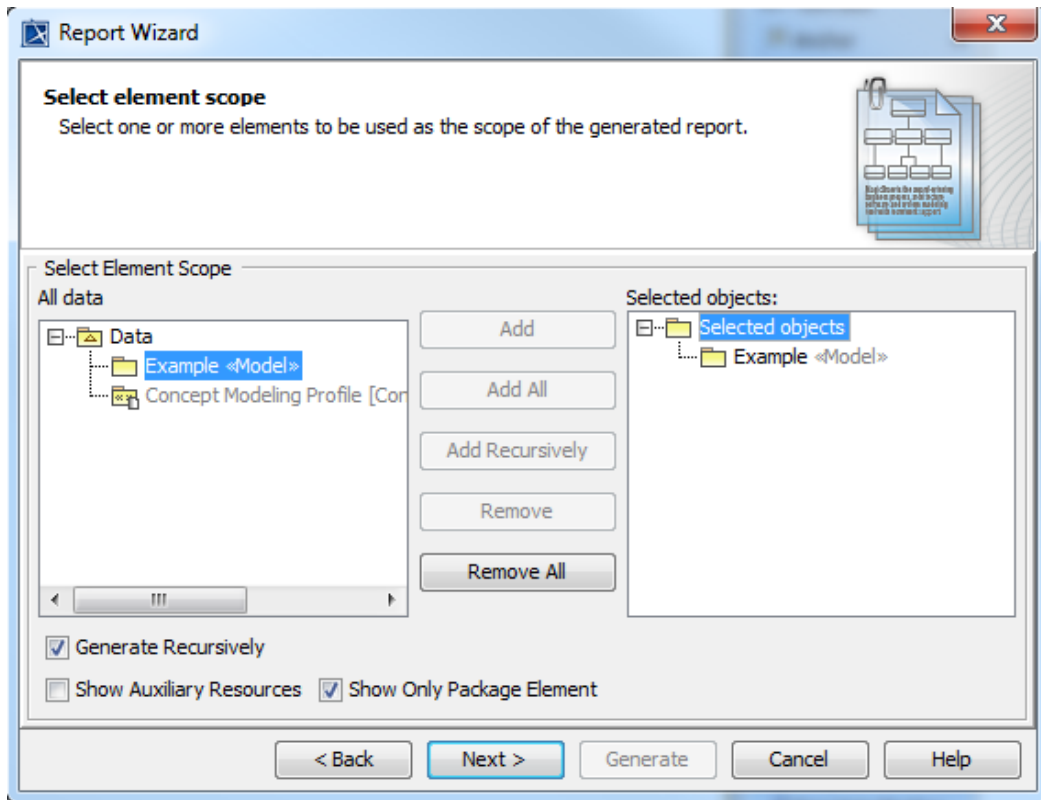9. Click the **Next** button.

Figure 200   Selected the scope of the glossary

10. Name your file and file location for your file.
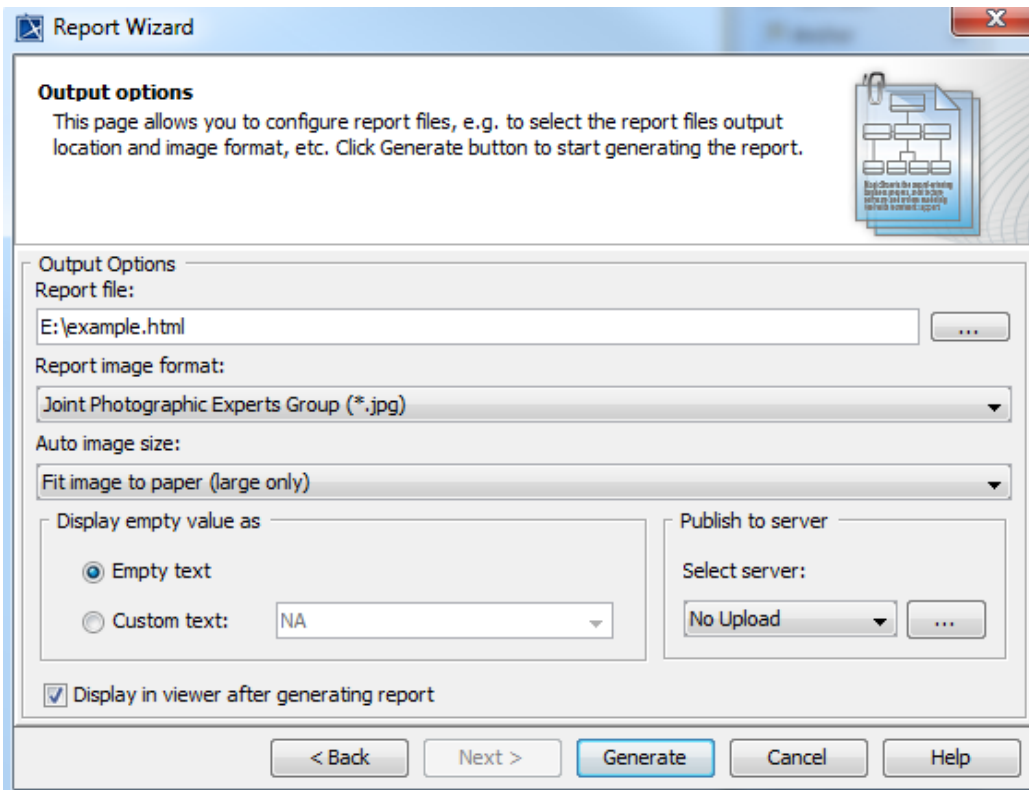11. Click the **Generate** button.

Figure 201   Generating a glossary

5.21.1 Updating symbol styles in older projects

In the message prompt, if you select No, the prompt will always pop up; otherwise, Concept Modeler will update your styles. The option called "Ask to update outdated symbol styles" which prompts you to update symbol styles when styles are out of date is set to True by default. However, if you set it to False, then the prompt window will not show.
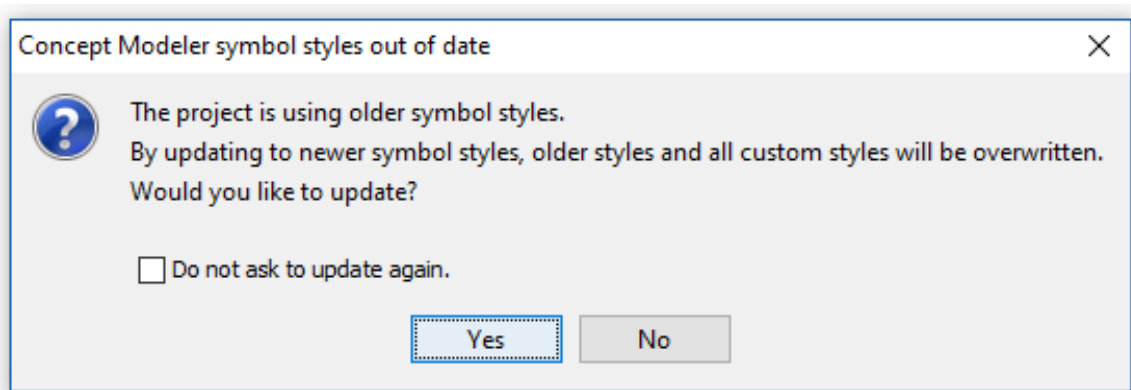

Figure 202 You will get this popup message when you load an older Concept Modeling project with an older set of symbol styles.

Copyright © 2017, No Magic, Inc.

To enable/disable the "Ask to update outdated symbol styles" option:
1. On the main menu, click Options then Project.
2. In the Project Options window, click on General then Concept Modeling.
3. In that window, find the "Ask to update outdated symbol styles" field and check the box so it says true.
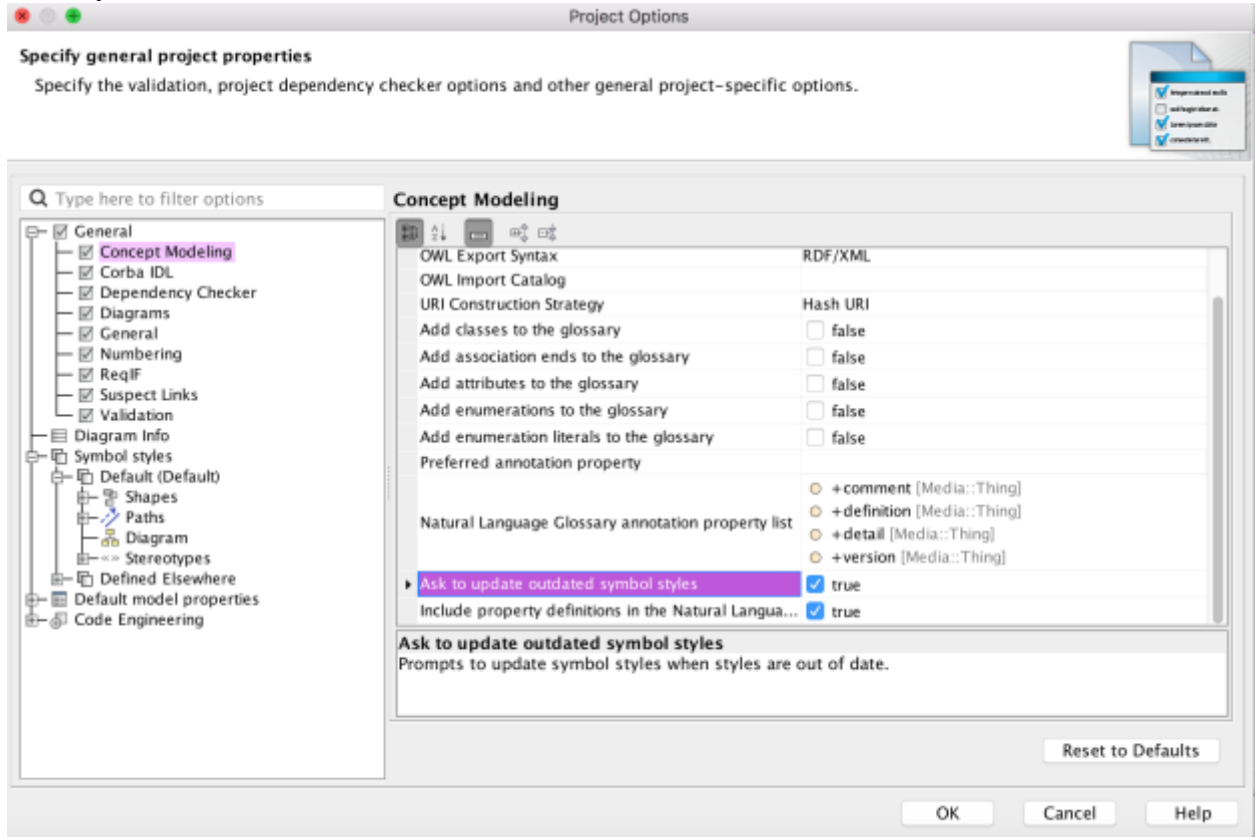
4. Click OK.

5.21.2 Selecting a List of Ordered Annotation Properties

We have added a new feature which allows you to select an ordered list of annotation properties which will be displayed in the Natural Language Glossary.

To select an ordered list of annotation properties:
1. Click **Options** then **Project**.
2. In the Project options window, select **General** then click on **Concept Modeling**.
3. Find the **Natural Language Glossary annotation property list** field and add properties.
4. Click OK.
5. You should see the changes in effect when you generate a Natural Language Report.

5.21.3 Include Property Definitions in the Natural Language Glossary

You must enable the option labeled "Include property definitions in the Natural Language Glossary" which lists property definitions in addition to class definitions in the Natural Language Glossary.

To enable the "Include property definitions in the Natural Language Glossary" option:
1. Click Options then Project.
2. Click on General and select Concept Modeling.
3. Scroll down through the Concept Modeling screen and find the Include property definitions in the Natural Language Glossary option.
4. Click the checkbox so it says "true."
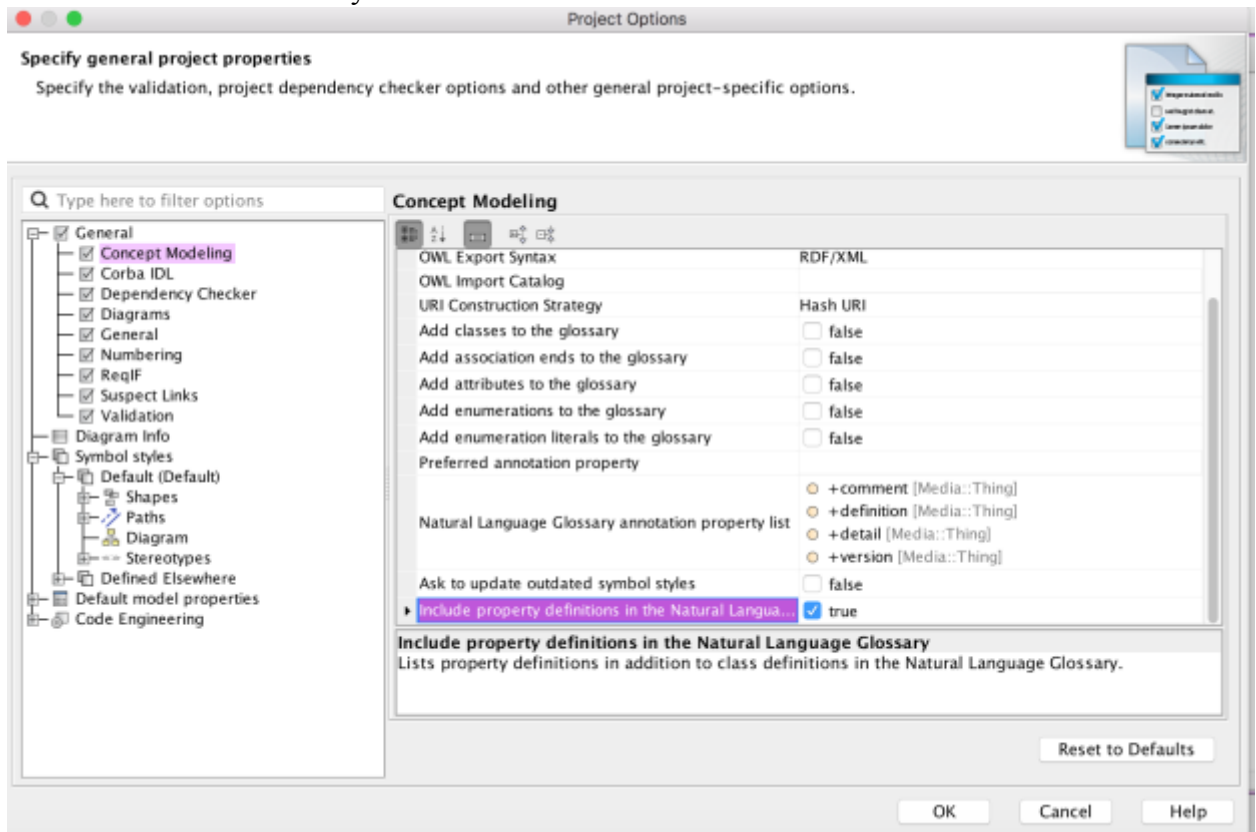
5. Click OK.
6. You should see this change in your NLG report.

Properties (jump to **Classes**)

C

    comment

        A property that can be used by any class.

        Definition:

D

    definition

        A property that can be used by any class.

        Definition:

    detail

        A property that can be used by any class.

        Definition:

Figure 203   Segmented shot of a report showing the property definitions corresponding to the annotation property list.

6   References

[1] OMG, MDA Guide rev. 2.0, OMG Document ormsc/2014-06-01

[2] https://www.ietf.org/rfc/rfc3987.txt

[3] http://dublincore.org/documents/2012/06/14/dcmi-terms/?v=terms#description