MetaModelAgent

**for IBM Rational® Software Architect Designer and RealTime Edition,
HCL RealTime Software Tooling and Eclipse Papyrus™**

Metamodeling

UML-profile for Metamodels
to be used in MetaModelAgent
Version 4.5.0

AO-MMA-022 • 2020-09-25

ADOCUS

Contents

1 Introduction

This manual defines a UML-profile for metamodels expressing UML-based domain-specific model languages (DSMLs) to be used in MetaModelAgent.

MetaModelAgent is a modeling tool extension which based on these kind of metamodels brings domain-specific modeling and model analysis capabilities to the host tool.

MetaModelAgent are available for the following host tools: *IBM Rational Software Architect family* (RSAD and RSARTE), *HCL RealTime Software Tooling* (RTist) and open-source *Eclipse Papyrus*.

For information on how to use MetaModelAgent for domain-specific modeling, please refer to the **MetaModelAgent Modeling User Manual**.

1.1 Reading Instruction

The readers of this manual are primary those persons involved in designing a UML-based domain-specific modeling language to be used in MetaModelAgent.

Good knowledge in UML is required to understand this manual.

Chapter 2: *Create* a new metamodel, describes how to create a new metamodel using your modeling tool.

Chapter 3: *Introduction to the UML-profile*, describes how this UML-profile is constructed, what it can express and how it fits in to a model hierarchy.

Chapter 4: *Defining Model Items*, describes how model items (elements, diagrams and relations) should be specified in a metamodel.

Chapter 5: *Defining Property Constraints*, describes how constraints on item properties should be specified in a metamodel.

Chapter 6: *Reuse between metaclasses*, describes how inheritance can be used within a metamodel.

Chapter 7: *Defining Model Structure Constraints*, describes how a model structure built up of elements and diagrams should be represented in a metamodel.

Chapter 8: *Defining Relationships*, describes how relations between elements should be represented in a metamodel.

Chapter 9: *Exemptions from UML2,* a summary of those parts of UML2 that is represented differently in the metamodel notation.

Chapter 10: *Defining Connectors*, describes how connectors such as transitions, object flows, control flows and messages should be represented in a metamodel.

Chapter 11: *Defining Custom Diagrams*, describes how custom diagrams in Papyrus can be represented in a metamodel.

Chapter 12: *Defining Diagram Content*, describes how diagram content should be expressed in a metamodel.

Chapter 13: *Combined Property Constraints*, describes how to express composite property constraints that makes it easy for the end user.

Chapter 14: *Constraint Constraints*, describes how to explicitly express Constraint constraints.

Chapter 15: *Real-Time modeling support*, describes how real-time specific concepts in Rational Software Architect Real-Time Edition should be expressed in a metamodel.

Chapter 16: *Organizing metamodels*, describes how metamodels can be divided and structured into packages, and extended in other metamodels. This chapter also introduces how to document a metamodel.

Chapter 17: *Validating a metamodel* describes several complementary techniques for validating the correctness of a metamodel.

Chapter 18: *Deploying a metamodel* describes how to create a new metamodel based in the provided template and how to populate, test and deploy the metamodel to the users.

Chapter 19: *Models included in MetaModelAgent*, describes the models, metamodels, templates and profiles that are included in the MetaModelAgent product package.

Appendix A: *Stereotypes*, a reference list of all stereotypes defined in this profile.

Appendix B: *Meta class attributes,* reference lists of all valid attributes representing UML element properties and other MetaModelAgent-specific element properties.

Appendix C: *Meta class operations,* reference list of all valid operations in a metamodel that's are used for controlling MetaModelAgent behavior.

Appendix D: *Regular expressions,* a summary of the regular expression language that can be used to express property constraints.

Appendix E: *Pre-defined Keywords,* a list of pre-defined keywords in the metamodel notation.

Appendix E: *Known Limitations,* a summary of known limitations in the current version of the metamodel notation and also a list of exceptions from the UML 2.x specification.

Appendix F: *The Metamodel template,* a diagram of the metamodel template that is included in MetaModelAgent.

Appendix G: *Example of a Metamodel,* a complete example of a metamodel for use-case modeling.

Appendix H: *Migrating Instructions* contains guidance on how to migrate a metamodel from an earlier version.

All diagram examples in this document are produced in IBM Rational Software Architect Designer, they may look different in Eclipse Papyrus.

1.2 References

[1] *Eclipse UML2 documentation*
(www.eclipse.org/uml2)

[2] *IBM Rational Software Architect*
(www-03.ibm.com/software/products/en/rational-software-architect-family)

[3] *Papyrus Modeling Environment*
(www.eclipse.org/papyrus)

[4] *MetaModelAgent*
(www.metamodelagent.com)

1.3 Terminology

The following terms are commonly used in this manual.

Term	Explanation
DSML	Domain-specific modeling language. Modeling language that is tailored for a specific business domain.
Item	An *item* is the generic term for all kinds of elements, relations and diagrams.
Item type	The term *item type* is used to denote the UML type of a specific item. Examples of Item types are: Class, Activity and Dependency.
Property	A property is a characteristic thing on an item. A property is often adding semantics to the item. Examples of properties are: Name, Visibility and Abstract.
Model	A *model* is a complete description of a system from a specific view on a constant level of abstraction.
UML-profile	A *UML-profile* is a well-defined extension of UML within a specific modeling domain, where the built in extension mechanisms are used. A UML profile normally contains stereotype definitions, property definitions and constraints. A UML-profile must provide unambiguous semantics for all included definitions.
Metamodel	A *metamodel* defines a DSML in terms of metaclasses and their relationships for a specific kind of models. A model is an instance of its metamodel.
Metaclass	A *metaclass* is a class in a metamodel that defines the rules for valid items in a model, by defining the constraints that must hold for the items. Each item within a model is an instance of a metaclass in a metamodel.
Metaclass attribute	A *metaclass attribute* is an attribute on a metaclass. A metaclass attribute defines the constraints of a property´s value for items that are instances of the metaclass. Metaclass attributes can specify constraints on both standard UML-properties and user-defined properties defined on stereotypes in a UML-profile.
Property Definition	Same as Metaclass attribute
Metaclass operation	A *metaclass operation* is an operation on a metaclass. A metaclass operation represents some behavior or characteristic constraints for items that are instances of the metaclass.

1.4 What's New

From v4.3.0 to v4.3.1

- Support for definition of nested models using separate metamodels, see chapter 7.6.
- Support for *stylesheet* property definitions in Papyrus, see Appendix B, ***Fel! Hittar inte referenskälla.***.

From v4.3.1 to v4.4.0

- No changes in the metamodel notation.
- Extension point added in MetaModelAgent for registering the mapping from UML elements to a user-defined metamodel. Dependency relationship will then not be needed between a user model and the registered metamodel, see chapter 18.

From v4.4.0 to v4.4.1

- The extension point for registering the mapping from UML elements to a user-defined metamodel, introduced in v4.4.0, has been improved to also support metamodels stored in the workspace, see chapter 18.

v4.4.1 to v4.5.0

- Metamodel templates based on built in General UML Guidelines and General UML-RT guidelines (available in RSA RTE and HCL RTist only) are available when creating a new model. See chapter 2.1 and 2.2.
- Support for enforced nested elements. See chapter 7.4.
- Support for metaclass attributes representing redefined properties. See Appendix B.
- Simplified entering of explanation of valid property values. See chapter 16.3.
- Redesigned metaclass attribute appendix in this manual. See Appendix B.

1.5 Contact

Please contact Adocus (support@adocus.com) if more information, education or support around this metamodeling profile is needed, or if there are any detected defects or ideas of improvement.

Please visit Adocus web site www.adocus.com for information about new releases.

Please visit MetaModelAgent web site www.metamodelagent.com for more detailed information about MetaModelAgent.

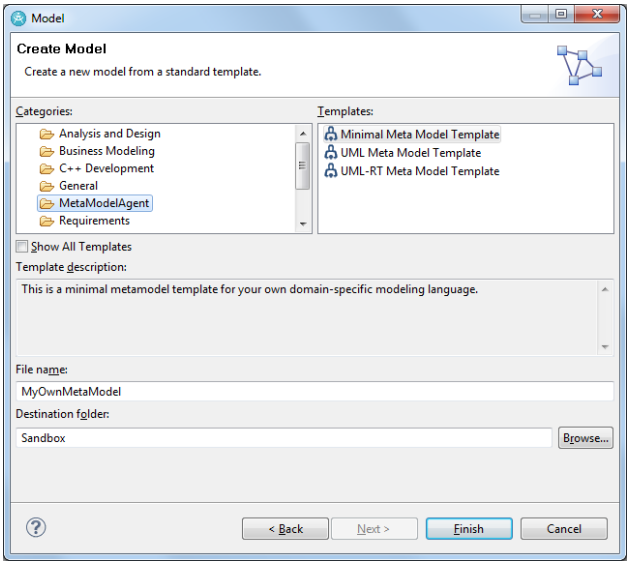
2 Create a new metamodel

2.1 Using RSAD/RSARTE or HCL RTist

MetaModelAgent provides pre-filled metamodel templates that can be used to create a metamodel for your own DSML. Follow these steps:

1. Select *File→New→Model* from the main menu to open the New Model Wizard.
2. Select to create a model from a Standard Template and press *Next.*

Select the category *MetaModelAgent* and select one of the available templates. The templates available are described below.



3. Fill in an appropriate name and select a destination for your new metamodel
4. Press *Finish* and your new metamodel will be created based on the template.

The new metamodel will make use of the MetaModelAgent profile which provides all the necessary stereotypes.

Metamodel templates

The following metamodel templates are available:

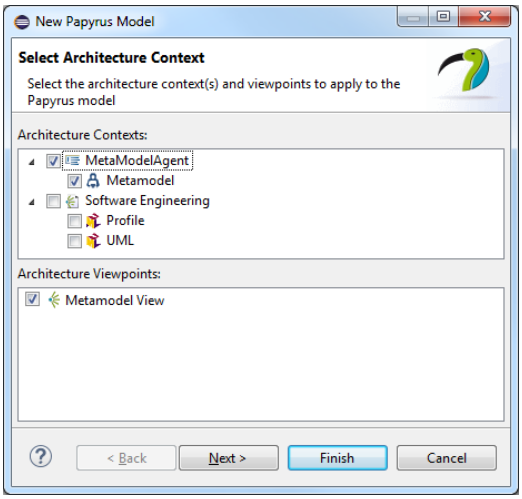
Metamodel Template	**Description**
Minimal Meta Model Template	A minimal metamodel with only the mandatory metaclasses that must exist in a metamodel. Use this template if you would like to define your own UML-based DSML from start.
UML Meta Model Template	A complete metamodel which cover all concepts in UML supported by MMA. Use this metamodel if you want to use most of UML but make some minor reductions, adjustments and/or extensions for your own DSML. This template is the same model as the built-in General UML Modeling Guidelines which can be used for UML models out-of-the-box.

UML-RT Meta Model Template	A complete metamodel which cover all concepts in UML-RT supported by MMA. Use this metamodel if you are using RSARTE or HCL RTist and want to use most of UML-RT but make some minor reductions, adjustments and/or extensions for your own DSML. This template is the same model as the built-in General UML-RT Modeling Guidelines which can be used for UML-RT models out-of-the-box.

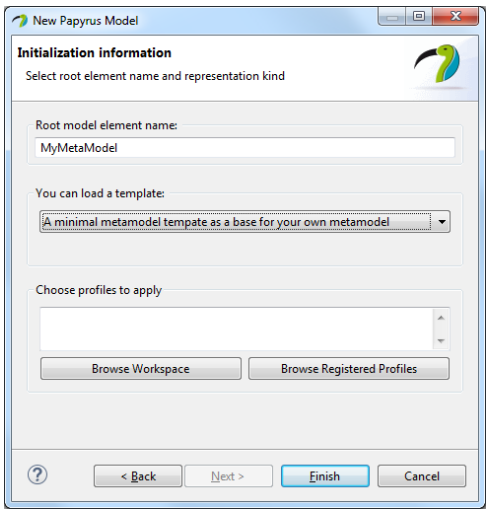
2.2 Using Eclipse Papyrus

MetaModelAgent provides pre-filled metamodel templates that can be used to create a metamodel for your own DSML. Follow these steps:

1. Select *File→New→Papyrus Model* from the main menu to open the New Papyrus Model Wizard.



2. Select *Metamodel* as architectural context and *Metamodel View* as the architecture viewpoint and press Next.
3. Select the parent folder and set the file name of the new metamodel and press *Next*.
4. Set the root model element name for the Metamodel.



5. Select a metamodel template to use from the dropdown list of available templates. Read more about the available metamodel templates below.
6. Press *Finish* to add a new blank metamodel or select the pre-filled template from the drop down list and press *Finish* to add a new pre-filled metamodel. You may be prompted for a profile migration after pressing *Finish*.

The new metamodel will make use of the MetaModelAgent profile which provides all the necessary stereotypes.

Metamodel templates

The following metamodel templates are available:

Metamodel Template	Description
Minimal metamodel template	A minimal metamodel with only the mandatory metaclasses that must exist in a metamodel. Use this template if you would like to define your own UML-based DSML from start.
Built-in metamodel for standard UML	A complete metamodel which cover all concepts in UML supported by MMA. Use this metamodel if you want to use most of UML but make some minor reductions, adjustments and/or extensions for your own DSML. This template is the same model as the built-in General UML Modeling Guidelines which can be used for UML models out-of-the-box.

2.3 Populate the Metamodel

You are now ready to populate your metamodel so it represents your DSML. Use this manual as a guide on how to develop the metamodel.

- Create metaclasses for all kind of UML items that should be valid in your metamodel. Select readable names of the metaclasses to reflect the concept that they represent in your domain.
- Create metaclass attributes for all significant properties for each kind of item.
- If your metamodel is large, organize your metamodel in metaclass packages, and maybe separate it into several linked metamodels.
- Document all metaclasses and metaclass attributes so that the user understand what they represent and how they should be used.

2.4 Validate and Test the Metamodel

There are several complementary techniques for validating and testing a metamodel before deploying it to the users. Chapter 0 describes in details the different techniques that can be used.

3 Introduction to the UML-profile

3.1 Position in the OMG Model Hierarchy

The table below shows the levels of models defined in the OMG model hierarchy.

The UML-profile defined in this manual state the design of metamodels on level M2. The UML-profile itself can therefore be defined by a meta-metamodel on level M3. The meta-metamodel is delivered as part of the MetaModelAgent tool.

Level	Content	Explanation	Typical items
M3	Meta-metamodel	A metamodel defining how metamodels on level M2 should be designed.	Metaclasses
M2	Metamodel	A metamodel defining how models on level M1 should be designed.	Metaclasses
M1	Model	A model defining executable model instances on level M0.	Classes Use Cases
M0	Instantiation of a model	An executing instance of a model on level M1.	Objects

3.2 The Scope of this Profile

This UML-profile defines a simple-to-use metamodel notation, optimized to express UML-based domain-specific modeling languages for Eclipse UML2-models. For details about Eclipse UML2, see ref. [1].

The profile can be used to express UML-based domain-specific languages including the following kind of constructions:

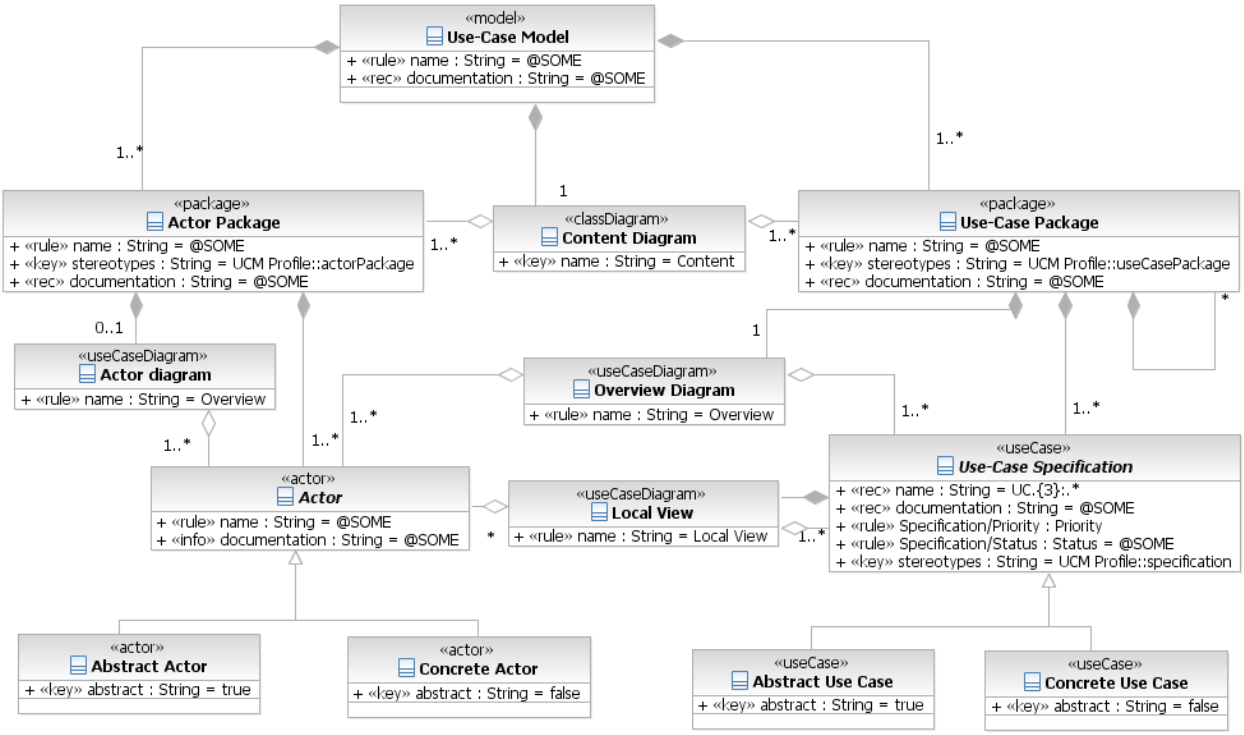
- All kind of model elements, except a few kinds of elements within activities, interactions and state machines.
- All kind of diagrams.
- All kind of relations between all kinds of supported elements.
- Constraints on all built-in properties for all supported elements, relations and diagrams.
- Constraints on all user-defined properties defined by profile stereotypes.
- Constraints on the model structure including elements, relations, and diagrams.
- Constraints on diagram contents for all diagrams.
- All kind of modeling constraints can be expressed as mandatory rules as well as recommendations and information issues.

The UML-profile also contains mechanisms for organizing and reusing metamodels.

3.3 Summary of the Metamodel Notation

A metamodel is expressed by using classes, attributes, operations, interfaces, relations and enumerations. A metamodel can therefore be visualized in one or several class-diagram.

The diagram below shows a subset of a realistic metamodel specifying a subset of UML and some minor extensions for use-case modeling.



Example 1: *A subset of a metamodel specifying a DSML for use-case models*

Classes in a metamodel, also called metaclasses, are used to specify *item definitions*. An item definition defines a specific kind of items which are valid in the kind of models the metamodel defines.

- The *stereotype of a metaclass* indicates the UML base type of the items. The *name of a metaclass* should give a hint of what the items represent.
- An *attribute of a metaclass* defines a significant property for the items and often also constraints on that property. The stereotype of an attribute indicates the severity of the constraint.
- An *operation of a metaclass* is used to express specific characteristic of the items.
- *Composite aggregations* represent constraints on the model structure.
- *Shared aggregations* represent constraints on diagram contents.
- *Plain associations* (not included in the diagram above) represent constraints on relationships suppliers.
- *Generalization relationships are used to inherit and make refinements between metaclasses.*

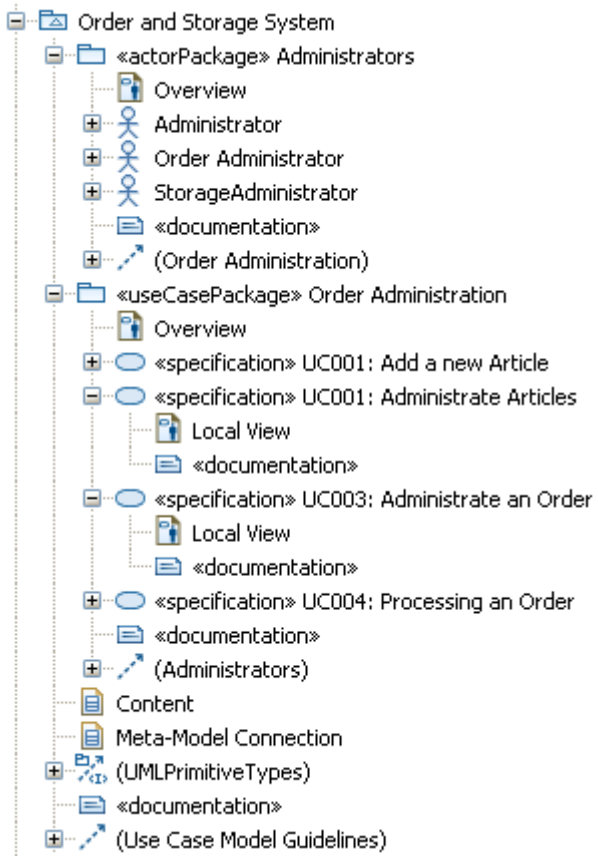
- *Enumerations* (not included in the diagram above) are used to express lists of valid property values.
- *Usage relationships* (not included in the diagram above) may optionally be used to express that a metaclass is referring an enumeration or another metaclass.
- *Interfaces* (not included in the diagram above) are used to express concepts that are realized by two or several private (invisible) metaclasses.
- *Realization relationships* (not included in the diagram above) are used to express that a metaclass realizes an interface.
- *Constraints* (not included in the diagram above) are used to express exclusive-or constraints between aggregated metaclasses, or advanced constraints on diagram content.

The left part of the metamodel example above should be interpreted in the following way:

- A *Use-Case Model* is a model. A Use-Case Model should have some name and is recommended to have some documentation. A Use-Case Model should contain one or several Actor Packages.
- An *Actor Package* is a package that must have the stereotype «actorPackage» from a “UCM profile”. An Actor Package should have some name and is recommended to have documentation. An Actor Package should contain one or several Actors and may optionally contain an Actor Diagram.
- An *Actor* is a UML-actor which can be either a Human Actor or an External System depending on the keyword. The common things about all actors are that they should have some name and are recommended to have documentation. The abstract property is also significant, but it is allowed to have any value.
- Finally an *Actor Diagram* is a use-case diagram which must have the name “Overview”.

The complete notation of metamodels is described in detail with several examples in the following chapters of this manual.

The metamodel *General UML Modeling Guidelines* that are included in the MetaModelAgent package contains a complete metamodel of the part of UML that this notation support. This metamodel can be used to understand how to build our own metamodels.



Example 2: *A use-case model that follows the metamodel example.*

4 Defining Model Items

Classes representing metaclasses are used to define the kind of items (elements, diagrams and relationships) that can exist in a model.

One metaclass should exist for each unique kind of item that should be valid in a model. A metaclass defines which item type that is expected and property constraints on that item type.

An item in a model is an instance of its metaclass and should fulfill all its constraints for the model to be regarded as correct.

Tip: name the metaclasses so that it indicates the concept represented by the metaclass. The name should be unique within the metamodel.

4.1 Item Types

The stereotype of a metaclass defines the kind of UML item represented by the metaclass.

There is one unique stereotype for each type of item in UML. Examples of stereotypes representing items are **«package»**, **«class»**, **«useCase»**, **«dependency»** and **«classDiagram»**.

Example 3: *The metaclass **ControlClass** represents a class in a model.*

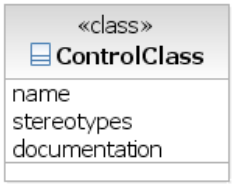
See *Appendix A* for a complete list of stereotypes representing items.

5 Defining Property Constraints

An attribute of a metaclass is used to express a constraint on a significant property for the kind of items that the metaclass represents. There are pre-defined attribute names for all kind of properties for all kinds of items. Examples of attributes are: **name**, **stereotypes**, **keywords**, **visibility** and **documentation**.

See *Appendix B* for a complete list of predefined attribute names.

Only those properties for which there are an attribute in the metaclass, are regarded as significant. A lack of a valid attribute indicates an insignificant property.



Example 4: *The metaclass **ControlClass** represents a kind of class that has three significant properties.*

Each type of item supports its own subset of properties. Only those attributes that represents supported properties for the item kind should be expressed in the metaclass.

For detailed information of valid properties for each specific item kind, see ref. [1].

The order of the attributes within the metaclass has no significance.

5.1 Property Values

The default value of a metaclass attribute is used to express constraints on the values that the corresponding property must hold.

The default value can be one of the following:

- A regular expression that the property value must fulfill, for example: **UC.{3}:.***.
- A constant string representing a fix property value, for example: '**1..***'. Constant strings should be surrounded with single quote characters. If constant values do not contain any characters representing regular expression semantics, the surrounding single quote characters may be omitted.
- A keyword beginning with @, indicating a special kind of constraint. For example **@SOME** indicates that the property value should not be left empty.
- References to other property values. See chapter 5.11.

The possibility to use regular expression makes it possible to define very complex property value constraints. See *Appendix* D for more information on how to define regular expressions.

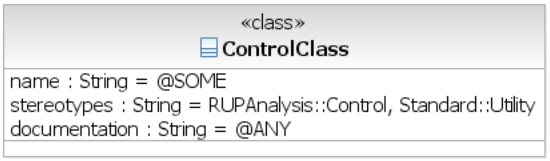
The following keywords can be used as pre-defined constraints on a property value:

Keyword	Semantics for the item property
@SOME	The property value must not be empty.
@NONE	The property value must be empty.
@ANY	The property can have any value.

5.2 Stereotypes Property Values

A value of the stereotype property should be given in the following format: <*Profile name*>::<*Stereotype name*>, where the profile name must be the name of an applicable profile for the model.

The stereotypes property and the keywords property also allow multiple values. That can be expressed by a comma separated list of values as the default value of the metaclass attribute.



Example 5: *The stereotype Control from the RUPAnalysis profile and the stereotype Utility from the Standard profile must be set for classes that should be regarded to be control classes.*

The icon to be presented in MMA menus, wizards and views is taken from the first stereotype in the list. If there is no icon defined for the first stereotype in the list, the standard icon for the UML-element is used.

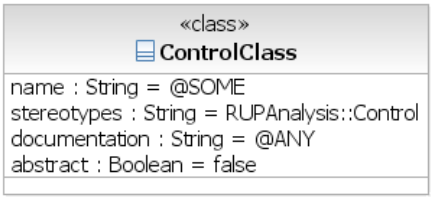
The list of stereotypes can have the keyword @ANY as the last value. @ANY means that any additional stereotypes besides the explicit specified ones are allowed.

Important: Stereotypes from the built-in standard profile should be named with an initial upper-case letter, even if they seem to have an initial lower case letter in the profile. E.g. the attribute name *Standard::Subystem* should be used to refer to the subsystem stereotype in the Standard profile.

5.3 Property Types

All metaclass attributes should set the **type** property. The valid values of the type property are:

- **String** from the built-in library of primitive types.
- **Boolean** from the built-in library of primitive types for metaclass attributes representing properties with boolean values.
- Enumerations defined in the metamodel, which contains literals for the property values. See chapter 5.6 for more information on how and when to use enumerations as the type of a property.
- Meta classes defined in the metamodel, which instances must match the value of the corresponding property. See chapter 5.7 for more information on how and when to use metaclasses as the type of a property.



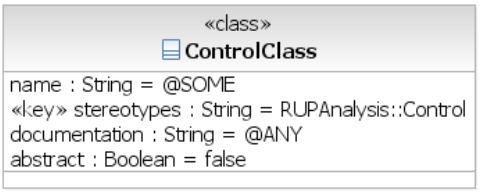
Example 6: *The metaclass **Control Class** represents a type of class with a non-empty name, the stereotype **«Control»** from the RUPAnalysis profile, non-abstract and with an optional documentation.*

5.4 Key Properties

The metaclass attributes are used to express constraints on item properties. If there are several metaclasses in the metamodel defining variants of the same item type, there is a need to decide which metaclass to use for validation.

The stereotype **«key»** is used for those metaclass attributes which constraints must hold for an item to be regarded as an instance of the metaclass.

A metaclass can have several **«key»** attributes. If any of them is not fulfilled by an item, that item is not regarded to be an instance of the metaclass.



Example 7: *The metaclass **Control Class** represents classes that must have the stereotype **«Control»**.*

The metamodel is as ambiguous if there is more than one metaclass, despite the existence of **«key»** attributes, which are matched by a single item.

5.5 Severity of a Constraint

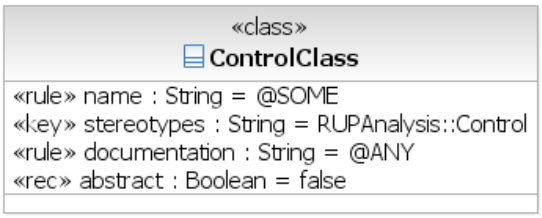
There is often a need of specifying mandatory rules as well as more liberated recommendations and perhaps even information issues. It is for example normal to regard the existence of documentation in a model as recommendations.

This profile supports three different severity levels on a metaclass attribute. Stereotypes are used to indicate the severity:

«rule» Represents a mandatory rule which results in an error if not fulfilled.

«rec» Represents a recommendation which results in a warning if not fulfilled.

«info» Represents an information issue which results in an information-reminder if not fulfilled.



Example 8: *The metaclass **Control Class** represents a class construction, where:*
*- the prerequisite is that the stereotype is **«Control»***
- the name must exist
- there is a recommendation that the class must not be abstract
- documentation may exist or not.

All metaclass attributes must have the stereotype set to either **«key», «rule», «rec»** or **«info»**.

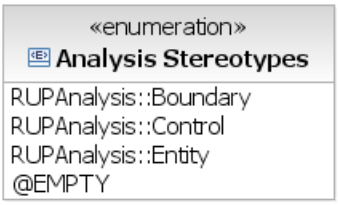
5.6 Alternative Property Values

There is often a need to define several alternative valid values for a property. This can be achieved by defining one separate metaclass for each alternative value, where only the default values for the specific metaclass attribute will vary. However, this is a bit clumsy and may end up in a very large set of similar metaclasses.

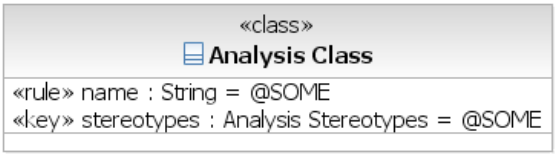
A better solution provided is to define an enumeration that specifies a set of alternative values. Each enumeration literal represents an alternative value.

The type property of the metaclass attribute is then set to the enumeration. The same enumeration can be reused to define valid sets of property values for different kind of properties.

The keyword **@EMPTY** can be used as a literal in the enumeration to denote an empty string as a valid property value.



Example 9: *A declaration of an enumeration class that defines four alternative values, where one of the values is the empty string.*



Example 10: *The stereotypes declaration makes use of the enumeration. Items are regarded as instance of this metaclass if they have one of the stereotype values from the enumeration or no stereotype at all.*

Ordering the literals

For each literal, the value property could be set to indicate the presentation order of the literals in MetaModelAgent. If no literal values are specified, alphabetic order applies. The first presented literal will become the default value when using the Add Wizard.

Use of external literals

If you want to manage the alternative property values without having to update the metamodel, you can let one or several literals refer to an external text file that contains a plain list of alternative property values. This is very useful if the values often changes.

This is done by adding an enumeration literal where the name of the literal is a file reference using one of the following two notations:

@file:*<absolute file path>*	Refers to a file in the local file system or in a shared location.
@platform:/*<plugin-id>*/*<relative file path within the plugin>*	Refers to a file in a deployed plugin.
@platform:/resource/*<project name>*/*<relative file path within the project>*	Refers to a file in the current workspace.

The referenced files should be plain text files with one valid property value on each line. The order of the literals in the MetaModelAgent user interface will be the same as the order in the file.

You can be combine enumeration literals representing property values with literals representing external files defining additional property values within the same enumeration.

Predefined keywords

The following keywords can be used as default values to define constraints in combination with an enumeration as the attribute type:

Keyword	Semantics for the item property
@SOME	The property should have one of the values listed in the enumeration.
@OTHER	The property should have any value, except the ones listed in the enumeration.
@ANY	The property can have any value, independent of the ones listed in the enumeration.

Enumeration Usage

A *usage* relationship may be drawn from the metaclass to the enumeration to make it clearer that a metaclass attribute uses an enumeration. This is however optional.

Generalization between enumerations

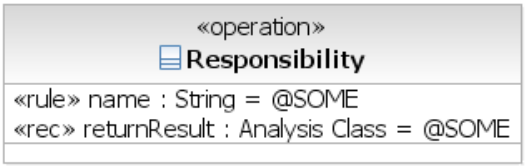
Generalizations between enumerations can be used to extend an enumeration with more literals. An enumeration that has a generalization relationship to another enumeration will inherit the other enumerations public literals.

OBS: this is an extended semantics to UML used by MetaModelAgent. According to the UML specification, Enumeration Literals are not inheritable.

5.7 Elements as Property Values

Some item properties can have an element as the property value, for example the *type* property of attributes and the *return result* property of operations.

To specify that a property should have a particular kind of element as its value: Set the *type* of the attribute to a metaclass defining the valid elements.



Example 11: *The returnResult declaration specifies that the property value set to an item that is an instance of the Analysis Class metaclass.*

The following keywords can be used as default values to define constraints in combination with a metaclass as the attribute type:

Keyword	Semantics for the item property
@SOME	The property value must be an element that is an instance of the metaclass given as the type of the current metaclass attribute.
@OTHER	The property value can be anything, except an element that is an instance of the metaclass as the type of the current metaclass attribute.
@ANY	The property can have any value, independent on the metaclass specified as the attribute type.
@INSTANCE	If the property definition represent a default value of a UML Property. The value must be an instance of the UML Property's type. If the property definition represents a slot value, the value must be an instance of the type of the defining feature of the slot, The behavior is undefined for all other kind of property definitions
@NONE	The property value must be null, e.g. no reference to any element.

An optional *usage* relationship may be drawn between the metaclasses to make it clearer that a metaclass attribute uses another metaclass.

You may also represent these constraints as directed plain associations to the metaclass representing the valid elements. If so the name of the association end should express the kind of property. In this case you may omit the severity stereotype for the association end. If omitted the constraint is regarded to be a rule.

5.8 User-Defined Properties

A UML Profile may contain stereotypes with user–defined properties. Constraints on those user-defined properties can be defined analogue with constraints on build-in properties by use of metaclass attributes.

If you, for example, have a profile with the stereotype *«subsystem»* defined, and within that stereotype the user-defined property *Responsible* is defined as an attribute of type String**.** To constraint this property to not be left empty, you can express the following in a metaclass:

«rule» subsystem/Responsible :String=@SOME

The name of the metaclass attribute denotes the user defined property, as defined in a profile stereotype. The name of the metaclass attribute should follow the naming convention:

<stereotype or category>/<property name>[.<field name>].

The stereotype must refer to a concrete stereotype, even if the property is defined on an inherited abstract stereotype. The category can be used instead of the stereotype if the Category[1] property is not empty.

The field name denotes a field if the property is defined by a class in the metamodel. User defined properties where the type is a two or more level composition are not supported.

If a user-defined property within a profile is using an enumeration for its valid values, an identical enumeration or subset enumeration has to be defined in the metamodel and used as the type of the metaclass attribute.

5.9 Optional Property Values

Normally the multiplicity of a property definition should is set to 1, except for multi-valued property definitions, see below. However, multiplicity 0..1 can be used to express that an empty value is allowed, beside the valid values defined by the property definition.

This means, for example, that a property definition with default value @SOME and multiplicity 0..1 is exactly the same as a property definition with default value @ANY and multiplicity 1, for property definitions with the type set to String or Boolean.

5.10 Multi-Valued User-Defined Properties

If the user-defined property has a multiplicity others than 0..1 or 1..1, it is regarded to be a multi-valued user-defined property. Multi-valued user-defined properties are specified in the meta-model as described above with the following additions.

- The *multiplicity* is used to indicate how many values that are allowed.
- The *unique* flag indicates if several values are allowed to be identical or not.
- The *ordered* flag indicates if the order of the values is significant or not.

The metamodel may introduce further limitation to the profile usage by specifying a more narrow multiplicity than the profile and the unique and ordered flag set even if they are not set in the profile.

«rule» subsystem/Author :String [1..5]=@SOME

Example 12: *The user-defined property "Author" associated with the stereotype subsystem must have one to five string values that are not empty.*

[1] Category is a standard property on Stereotypes in RSAD/RSARTE and HCL RTist but not in Papyrus. By using the General UML Guidelines metamodel in MetaModelAgent when editing profiles in Papyrus, the Category property will be available in the MMA Property View and Add Wizard.

5.11 Reference to a Context Item Property Value

Sometimes a property value is dependent on the parent items property value, or on another property value in the same item, therefore the metamodel notation makes it possible to refer to other property values when defining constraints on a property value.

For example: assume that a package should contain an interface which starts with the same name as the package. This can be expressed by the following metaclass attribute definition for the metaclass representing the interface:

«rule» name:String="{PARENT.name}."*

This kind of expression can be used to refer to any kind of property on the parent and can be used to define any kind of property constraint for the child item.

A similar notation can be used to refer to another property value in the same item. For example: To express that the documentation of a specific item should contain the name of the item:

«rule» documentation:String=".{THIS.name}.*"*

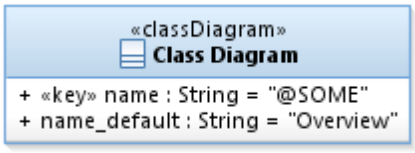
5.12 Default property values

It is possible to specify a default value for a property, separate from the property rule. The property default value is defined in a separate metaclass attribute. The attribute should have the following syntax:

<property name>_default : <Type> = <default value>

- The name of the attribute should be the name of the property followed by the postfix "_default".
- The type of the attribute should be the type of the default value, i.e. String.
- The default value of the attribute should be the default value of the corresponding property:
- References to property values in the parent´s element properties can be included in the default value definition. The syntax is the same as for parent references in property value rules, e.g. {PARENT.name}.

Please note there should be no stereotype applied to a property default value definition attribute.



Example 13: *The rule says that the element must have a name and that the default name should be Overview.*

The standard visibility semantics applies to property default value attributes as well, public attributes may be inherited by sub metaclasses.

6 Reuse between Metaclasses

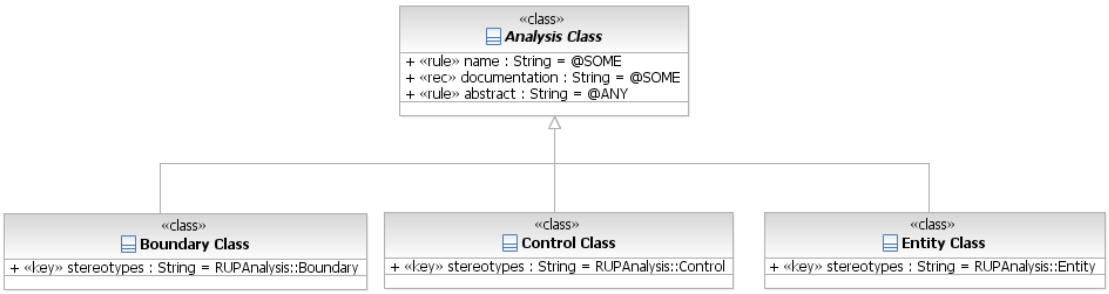
6.1 Metaclass Inheritance

Generalizations can be used between metaclasses to reuse *features* (attributes, relations and operations) between them.

The semantics of a generalization relationship between two metaclasses are as expected. If a feature has the *visibility* property set to *public* it will be reused, if it is set to *private* it will not. The *protected* and *package* visibility has no significance.

Features in a sub metaclass will override the ones with the same name in the super metaclass. That makes it possible to change any constraint in a sub metaclass.

A metaclass can be *abstract*, indicating that there cannot be any instances of it in the model. An abstract metaclass must be inherited by other concrete metaclasses for which instances can exist.



Example 14: *There are three variants of Analysis Classes, Boundary, Control and Entity. All three shares the same name, documentation and abstract constraints, but differ in stereotypes.*

You are not allowed to have ambiguous concrete meta-classes in an inheritance tree, e.g. many metaclasses that may match the same element. If so, you must add/change **«key»**-properties to ensure that there are no ambiguities left.

Multiple inheritances are allowed and may cause the normal sorts of troubles. However, they may add important semantics to the metamodel and are therefore often necessary.

6.2 Generic metaclasses

The super class of a generalization must have the same stereotype as the sub class, or to a so called *generic* metaclass,

A generic metaclass is a metaclass with the stereotype **«Item»**. It represents an arbitrary item of any type. Generic metaclasses act as super metaclasses in a generalization tree and must be abstract.

Generic metaclasses provide a powerful notation to express *or*-constraints between heterogenic items within the model.

A generic metaclass may have attributes defining property constraints. However, each potential sub-meta class must define an item type that supports all these attributes; otherwise the metamodel is regarded incorrect.

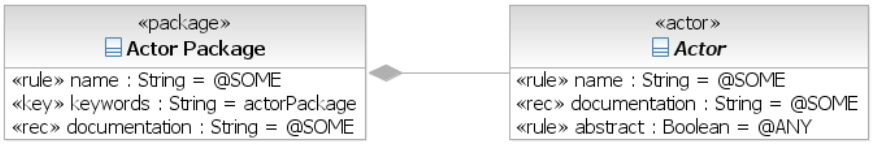
7 Defining Model Structure Constraints

The structure in a UML model is defined in a metamodel by the use of *composite aggregation* relationships between metaclasses.

Composite aggregation relationships between metaclasses can define the following kind of constraints on the model structure:

- The existence of elements in packages.
- The existence of other elements or element details in elements.
- The existence of different kind of diagrams in different kind of elements.
- The existence of relations drawn from elements.

Each kind of structural composition is described by a navigable composite aggregate between the metaclasses representing the participating items.



Example 15: *An Actor Package should contain actors*

When specifying model structures, inherited properties and operations of classifiers are regarded as being owned by the classifier that inherits them. The same does not hold for any other inheritable feature. Properties and operations are also regarded to be redefined by properties and operations in a sub-classifier with the same name. The value of the Redefined element property is not respected.

The severity of the composition can be expressed by using the stereotypes **«key»**, **«rule»**, **«rec»** and **«info»** on the aggregation. The semantic of the stereotypes is the same as when defining metaclass attributes representing properties:

«key» Must be fulfilled for an element to match the compositor metaclass, see chapter 7.4 for details.

«rule» Represents a mandatory rule which results in an error if not fulfilled.

«rec» Represents a recommendation which results in a warning if not fulfilled.

«info» Represents an information issue which results in an information-reminder if not fulfilled.

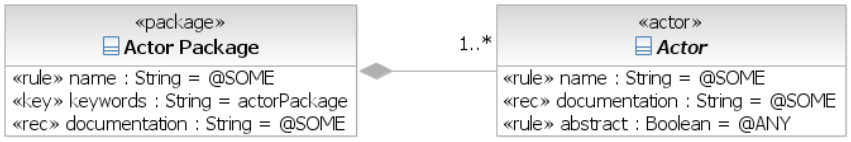
If no stereotype is set, the aggregation represents a mandatory structure rule.

A concrete Metaclass with the stereotype **«model»** or **«package»** that is not composite part in any composition aggregation represent a potential root element in a UML-model.

7.1 Cardinality Constraints

The multiplicity property on the aggregated role is used to define a specific number or range of valid number of sub-items in a model structure.

The multiplicity property defines how many sub-item instances of the metaclass which are valid in the model structure.



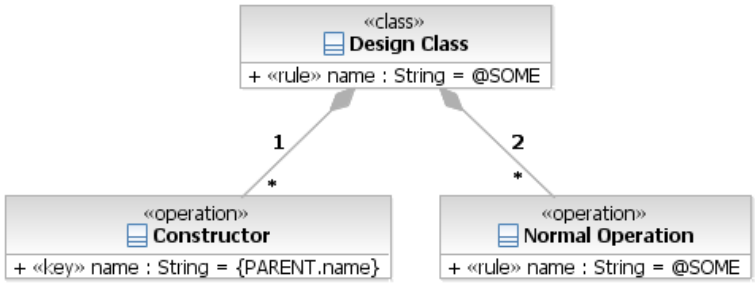
Example 16: *An actor package should contain at least one actor.*

7.2 Avoiding Ambiguous Structure Constraints

A metaclass with aggregations to several other metaclasses can result in a situation where those aggregated metaclasses are not defining mutual exclusive sub-items. In other words, more than one metaclass may match the same sub-item.

If this situation occurs, there is a need to express a priority-order between the aggregated metaclasses. This is done by setting a numeric order number as the name[1] of the composite aggregation relationship.

A low number denotes a high priority. Aggregated metaclasses without any specified order number will have the lowest priority.



Example 17: *An operation should in the first place be regarded as instances of "Constructor", if not possible they should be instances of "Normal Operation".*

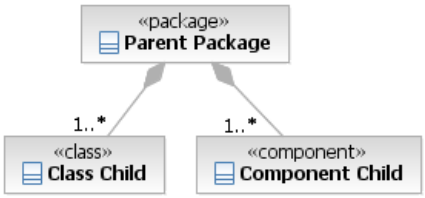
7.3 And, Or, Xor and Not in Structure Constraints

The common logical operators AND, OR, XOR and NOT is possible to express when defining constraints on the model structure. See explanation and examples below. You may of course also combine those patterns to even more complex structure constraints.

Logical AND

Logical AND is expressed by having an aggregation for each part (operand) of the AND-expression.

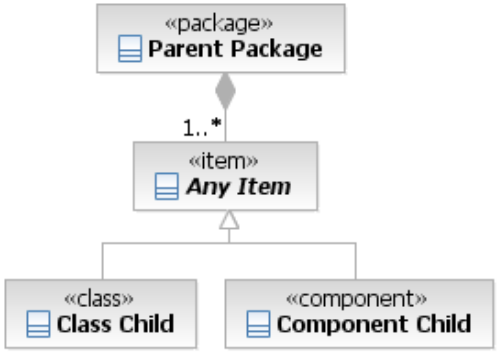
[1] Having several associations with the same name in a package may lead to a live validation warning that you should neglect.



Example 18: *Logical AND: A parent package should have at least one class child and at least one component child.*

Logical OR

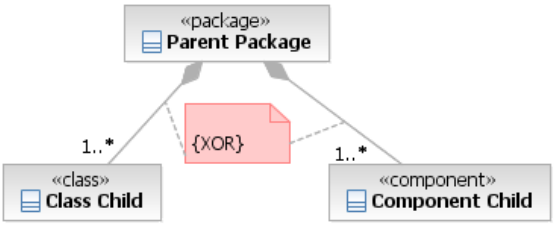
Logical OR is expressed by having one aggregation to an abstract metaclass, which have specialized sub-classes for each alternative. If the sub metaclasses denote different item types, the stereotype of the abstract metaclass should be «item».



Example 19: *Logical OR: A parent package should have at least one class child or component child.*

Logical XOR (Exclusive-OR)

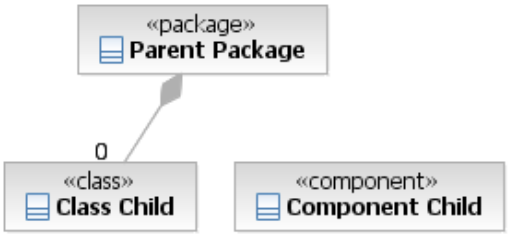
To express XOR you need to define a constraint with the value set to XOR and connect this constraint to two or more alternatives.



Example 20: *Logical XOR: A parent package should have at least one class child or at least component child, but both classes and components are not allowed in the same package.*

Logical NOT

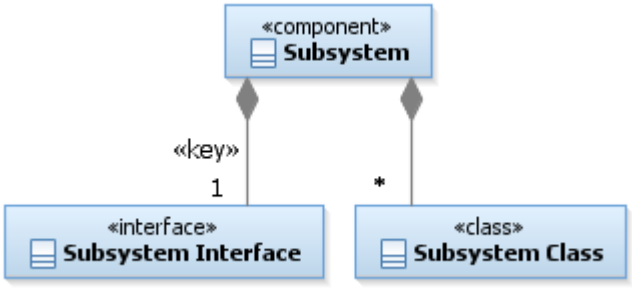
Logical NOT is default if a composite aggregation is not drawn. NOT can also be explicitly expressed by setting the multiplicity to 0 in the aggregation. The latter variant may give better problem explanations when used in MetaModelAgent



Example 21: *Logical NOT: A parent package should neither have class children nor component children.*

7.4 Enforced nested elements

By setting the composition stereotype to **«key»** you define that the composition must be fulfilled for the element to match the compositor metaclass. This enforces the element to have the correct nested elements to be identified itself.



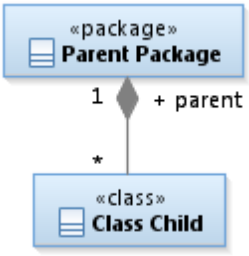
Example 22: *A component must have an interface to be able to be identified as a Subsystem.*

This kind of rules should be used sparingly and only when there are no other way to define the same behavior. Nested elements can be created in a single operation when using the MMA Add Wizard but not when using the standard UI. Refactoring using MMA Refactor feature cannot be used as it might requires creation of nested elements which is not supported.

There are also several limitations in MetaModelAgent's validation capabilities related to **«key»** compositions. There are no support for XOR-compositions and only one level of **«key»** compositions are supported.

7.5 Explicit parent references

The parent role in a meta-class composition may be named "parent" and be assigned a **«key»** or **«rule»** significance stereotype. That will explicitly indicate that the parent of the item represented by the composited metaclass must be an item represented by the metaclass that holds the composition. This may be useful in cases where there are different concepts, but with the same characteristics, in different positions in the model and in the same time references to those concepts that MetaModelAgent could not resolve correctly otherwise.



Example 23: *Example of an explicit parent reference, that may have **«key»** stereotype (not visible) indicating that a class must have an instance of the metaclass Parent Package as its owning package to be regarded being an instance of the metaclass Class Child..*

You may observe that as an alternative to name the compositor role as “parent”, you may use a separate directed plain association, or even an attribute, named “parent” to achieve the same behavior.

7.6 Nested models using separate metamodels

If a model contains nested models, these nested models can use separate metamodels.

To define the usage of separate metamodels o be used for a metaclass, simply draw a dependency between the metaclass and the separate metamodel.

7.7 Unconstrained Model Structure

Sometimes there is a need to allow an unconstrained model structure beneath a specific item. In other words, any kind of sub-items are allowed as part of another item.

This can be useful if one wants to release incomplete metamodels.

In a metamodel, this is achieved by the metaclass operation **permitAll()**. This operation denotes the possibility for an item to have arbitrary number of any kind of sub-items.

The operation should be public and should not have any arguments.

7.8 Suppressing Item Creation

It is possible to mark a metaclass so that MetaModelAgent will not be able to create new instances of it. This may be useful if the guideline designer will force the user to use the built in add-function in Software Architect and Software Modeler to create the corresponding item.

To mark a metaclass to not be able to be instantiated, add the operation **suppressAdd()** to the metaclass. The effect is that the metaclass will not be available as an selectable menu item in the MetaModelAgent Add submenu.

If you add some non-empty documentation to the **suppressAdd()** operation, the metaclass will be presented as a selectable menu item, but selecting the menu item will bring up a message dialog with the text that has been entered as the documentation. This makes it possible to customize information to the user that the concept defined by the metaclass is valid but that instances of it cannot be created by using MetaModelAgent.

You may notice that the **suppressAdd()** operation will not have any impact on the validation of models.

7.9 Suppressing Change

It is also possible to mark a metaclass so that MetaModelAgent will not consider it as an alternative in the *Change To* sub menu and in the automatic change wizard. This may be useful for elements that are automatically added when adding a certain element.

To mark a metaclass to not be an alternative classification for an element, add the operation **suppressChange()** to the metaclass. The effect is that the metaclass will not be considered as an alternative in the *Change To* menu and in the automatic change Wizard.

8 Defining Relationships

As already mentioned, metaclasses are also used to define different kind of relations that can exist between elements in a model.

8.1 Relationship Source

The source of a directed relationship is regarded to be the owner of the relationship.

The metaclass representing a source element should have a two-way navigable composite aggregate relation to the metaclass representing the directed relationship.

Example 24: *An instance of "Design class" can have arbitrary number of realization relationships to interfaces.*

Mandatory rules as well as recommendations can be specified by setting the aggregation stereotype to either **«rule»**, **«rec»** or **«information»**. If no stereotype is given, the aggregation represents a mandatory rule.

Multiplicities should be used to denote the number of relationship instances.

8.2 Relationship Target

A plain navigable association is drawn from the metaclass representing the relationship, towards a metaclass representing the supplier element to specify valid targets of a directed relationship. The target role name shall be **target**.

8.3 Associations

An Association in UML is a kind of element that is referred from two classifiers, through nested property elements, also known as *association ends*.

Both metaclasses representing association-ends of the association and metaclasses representing the association itself are therefore used to specify constraints around the usage of associations within a model,

A composite aggregated metaclass representing the association end specifies the fact that an element has an association connected to another element in the metamodel.

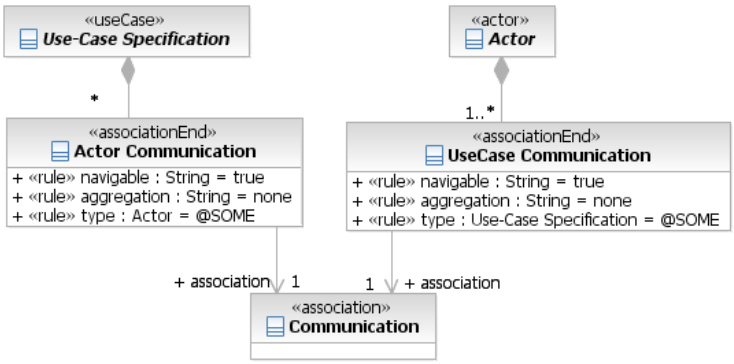
The metaclass representing the property element representing the association end should have the stereotype **«associationEnd»**, or **«property»** if it may represent an attribute only.

Severity can be specified by setting the aggregation stereotype to either **«rule»**, **«rec»** or **«info»**. If no stereotype is given, the aggregation represents a mandatory rule.

Important metaclass attributes for **«associationEnd»**, or **«property»** metaclasses are:

- **type**, referring to the metaclass for the element in the opposite end of the association.
- **association,** referring to the metaclass for the association element itself**.**

- **navigable,** defining if an association end is navigable or not.
- **opposite,** defining valid association ends in the other end of the association.



Example 25: *A use case can have arbitrary number of communication associations to an actor. An actor must have a communication association with at least one use case.*

Navigability

All associations ends are regarded to be owned by the element at the other end of the association, even thus the ownership according to UML 2.x specification depends on both the kind of element and whether the association end is navigable or not. As you see in the example above the association ends are regarded to be owned by the Use Case and he Actor respectively, even thus the ends of an association between use cases and actors always are owned by the association, according to the UML 2.x specification.

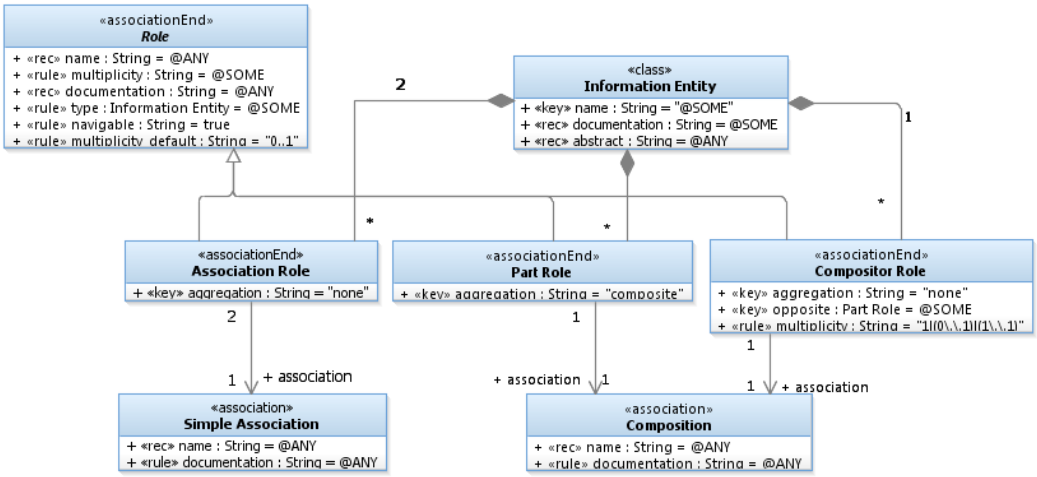
To define the navigability, you use the **navigable** property, as shown above. You may however notice that if you defined a mandatory association end from an element that is non-navigable. MetaModelAgent may report a validation error if the model containing the association is unloaded. The same will happen with navigable association ends between actors and use cases.

Association end limitations

You may define restrictions on which kind of association ends that may participate in the same association. This is done by setting the multiplicity property in the un-navigable end of the association from an association end metaclass to the association metaclass. See example below.

Opposite association ends

A useful attribute when defining association ends is the **opposite** attribute. You use **opposite** to define rules about the valid association ends in other end of the association. Be careful when defining opposite rules. If you add the stereotype **«key»** and let the default value of the **opposite** attribute be a association end meta class that also has a **«key» opposite** rule to your first metaclass, MetaModelAgent validation will lead to an infinite loop,



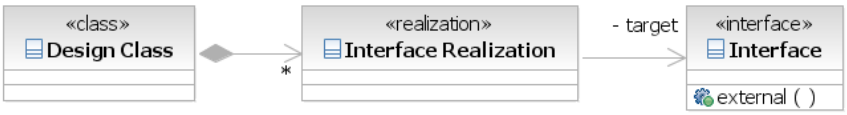
Example 26: *This is an example that separates the concepts of simple association from composition, and that guarantees that there must be exact one association end of type **Part Role** and one of type **Compositor Role** in a composition. **Simple associations** however must have exactly two **Association Roles**.*

*The example also shows the usage of an **opposite** rule in the **Compositor Role** metaclass to make it unique from the **Association Role** metaclass.*

8.4 References to External Elements

When specifying constraints on properties with elements as values and relationship targets, there are often a need to refer to elements that are not part of the same model. Those elements can be part of some other model but are accessed by the current model. An example of this is a Collaboration in an analysis model referring to a use case in a use-case model.

If those elements are defined by metaclasses in the metamodel, they will automatically be regarded as owned by the model. To avoid this to happen, the metaclasses can be annotated with the operation **external()**. The existence of this operation means that the elements represented by the metaclass can be referred from within the model, but cannot be owned by the model.



Example 27: *A design class can realize interfaces defined in some other model.*

9 Defining Activity Nodes

9.1 Actions, Pins, Control Nodes, Object Nodes

There are a lot of defined variants of these kinds of elements in UML2. The metamodel notation simplifies that fact by having only one metaclass stereotype for each base element.

To indicate a specific variant of a base element, you define a metaclass attribute with the following properties:

- The name should be the name of the base element followed by the word *Kind.*
- The stereotype should be **«key»**.
- The type should be **String** or some own-defined enumeration.
- If the type is **String**, the default value should denote the valid kind of variant. If the type is an enumeration, the default value may be a valid keyword.

Example 28: *A metaclass representing a send signal action*

Example 29: *A metaclass representing an output pin*

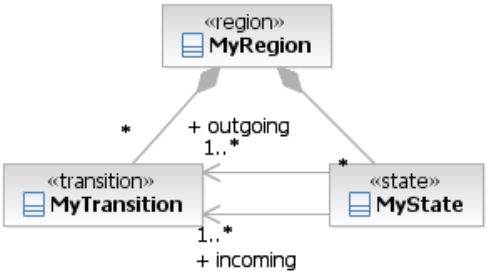
10 Defining Connectors

Transitions in state machines, object and control flows in activities and messages are all kind of “connectors” or “edges” that you may find hard to define properly in a metamodel. This chapter will give guidance on how constraints on these kind of connectors and the nodes they may connect, should be defined.

10.1 Transitions & Guards

Transitions connect states in a state chart diagram. Transitions are however owned by an enclosing Region element, in the same way as states are owned by the region. To express the existence of transitions within a region a composite aggregation should be used.

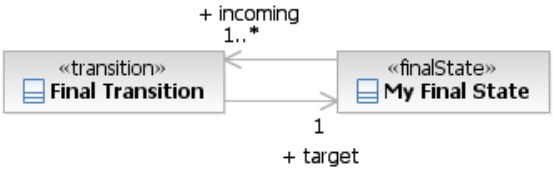
To express that a state (normal state, pseudo state or final state) has incoming and/or outgoing transitions, plain navigable associations are used, where the role name denotes incoming or outgoing, and where the multiplicity denotes the valid number of transitions.



Example 30: *A Region may contain arbitrary number of states and transitions and a State should have at least one incoming and one outgoing transition.*

You may notice that a metaclass representing a state may have several plain associations with the same role name[1] referring to metaclasses representing different kind of transitions. Logical AND, OR, XOR (exclusive-OR) and NOT constraints can be achieved in the same way as described in chapter 7.3.

If you want to express what kind of states that are related to a specific kind of transition, you can express that in the metamodel by using navigable plain associations from the transition metaclass to metaclasses representing states defining the source and targets of a transition. In those cases the role name *source* and *target* should be used.



Example 31: *A final transition must have a final state as the target and a final state must have at least one incoming final transition.*

[1] This may lead to a live validation warning that you should neglect.

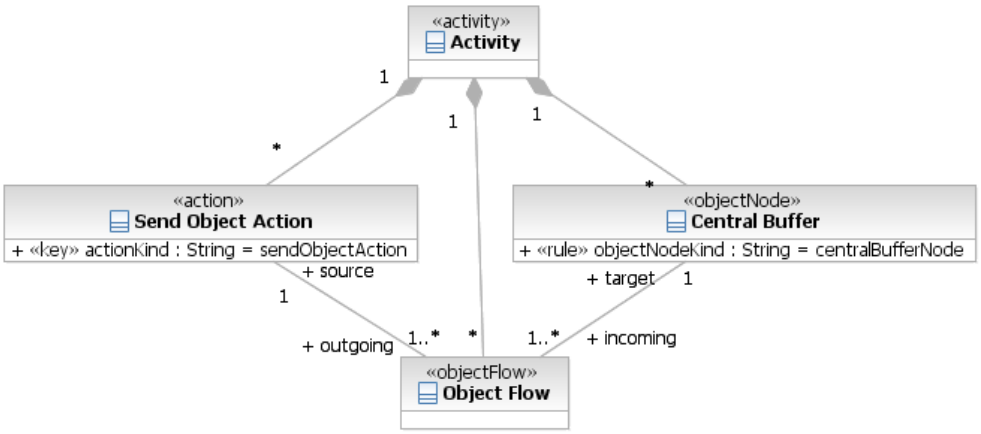
A constraint on a transition in UML2 represents a guard on that transition.

Constraints are not generally supported in the current version of the metamodel notation. Instead there is a dedicated metaclass stereotype **«guard»** to be used for representing guards.

Example 32: *A transition may have a guard*

10.2 Object and Control Flows

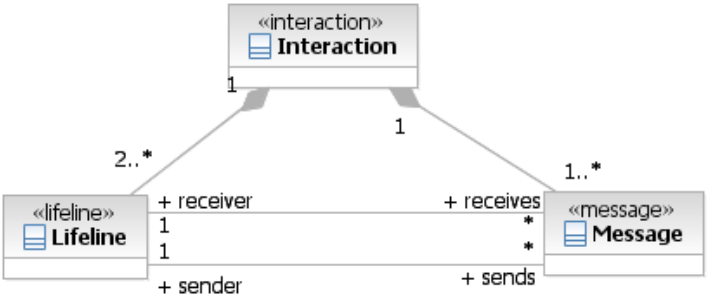
Object and controls flows are the connecting elements within an activity. They are owned by activities or by structured activity nodes. The same metamodeling principles as for transitions also holds for object and control flows.



Example 33: *A Send Object Action must have at least one outgoing Object Flow where a central buffer is the target.*
A Central Buffer must have at least one incoming Object Flow, where a Send Object Action is the source.

10.3 Messages

A Message is the connecting elements within an interaction. It is owned by the interaction, but connects the lifelines within the interaction. The same metamodeling principles as for transitions also holds for messages.



Example 34: *An Interaction must have at least two lifelines and one message.*
A Lifeline receives and sends arbitrary number of messages.
A Message has a lifeline as both a sender and receiver (however it does not have to be the same lifeline)

Example 35: *Messages*

10.4 More about Connector Constraints

As you have seen, it is possible to define constraints for both the source and target side of a transition, flow or message connector, and also constraints on outgoing and incoming connectors from the different kind of nodes. The constraints are expressed by association ends which may as well be showed as attributes in a metamodel.

The association ends should have a multiplicity, indicating how many participating connectors which are valid.

Important: The absence of an association end representing a connector constraint does not indicate that there are no valid connectors. Instead it means that any kind of any numbers of connectors are valid. If you want to express that a specific kind of connector is invalid, you should use 0..0 multiplicity.

The association ends may also be given an explicit severity stereotype; **«key»**, **«rule»**, **«rec»** or **«information»**, with the same semantics as expected. If the stereotype is omitted, it is considered to be a rule. Unfortunately stereotypes on association ends will not be visible in a diagram.

There may be several association ends with the same name indicating different kind of possible connectors. You may use XOR-constraints between these association ends, as described in chapter 7.3, and also ordering numbers of the corresponding associations, as described in chapter 7.2.

11 Defining Custom Diagrams

11.1 Custom diagrams in Papyrus extensions

Domain-specific modeling extensions to Papyrus may have defined custom diagrams based on standard UML-diagrams. To represent such a custom diagram in a metamodel add a metaclass attribute with the name *diagramKind* and a value that denotes the name of the custom diagram kind.

The simplifies way to find out the correct value of the *diagramKind* attribute is to activate a model containing a custom diagram towards the built-in *General Modeling Guidelines* and look for the current value of the *diagramKind* property in the MMA Property View.

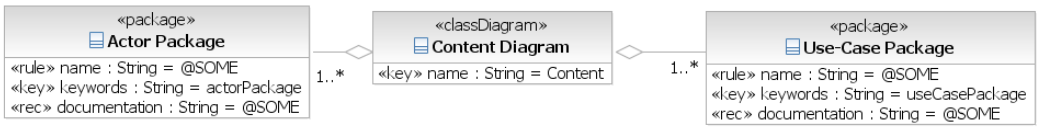
12 Defining Diagram Content

12.1 Classifier Diagrams and Object Diagrams

By classifier diagrams we mean class diagrams, component diagrams, use-case diagrams, deployment diagrams and freeform diagrams.

Constraints on which kind of elements that should be visible in a classifier diagram or in an object diagram are defined by establishing navigable *shared aggregation* relationships from the metaclass representing the diagram towards the metaclasses representing the items that can be visible in the diagram.

Multiplicity should be used to denote valid numbers of visible elements.



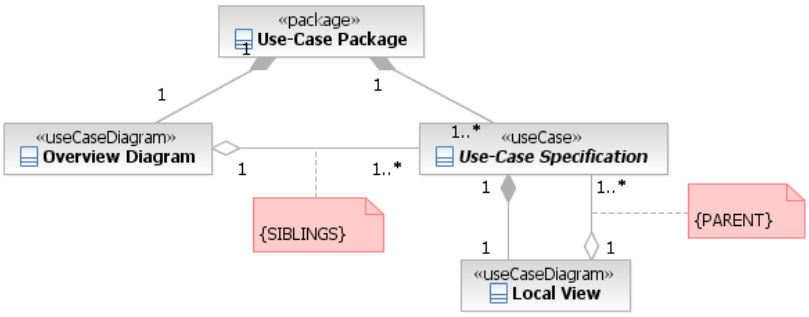
Example 36: *A class diagram with the name "Content" must contain at least one visible actor package and one visible use case package.*

The severity can be specified by setting the shared aggregation stereotype to either **«rule»**, **«rec»** or **«info»**. If no stereotype is given, the shared aggregation represents a mandatory rule.

Logical AND, OR, XOR (exclusive-OR) and NOT constraints can be achieved in the same way as described in chapter 7.3.

References to Surrounding Elements

When defining constraints on diagram content, you can also define in more details which elements that should exist in the diagram. You may refer to the element owning the diagram or the owning elements child elements. This kind of constraints is achieved by connecting UML constraint elements to the shared aggregations.



Example 37: *An Overview Diagram must at least show all Use Cases in the same Use-Case Package holding the diagram.*
A Local View diagram must at least show the Use-Case that holds the diagram.

Summary of the legal keywords in the body of UML-constraints and their semantics:

Constraint Body	Semantics for the item property
SIBLINGS	Indicates that at least all elements of the target kind in the same context as the diagram (its siblings) must be visible in the diagram
PARENT	Indicates that at least the element holding the diagram must be visible in the diagram.

12.2 Other Diagrams

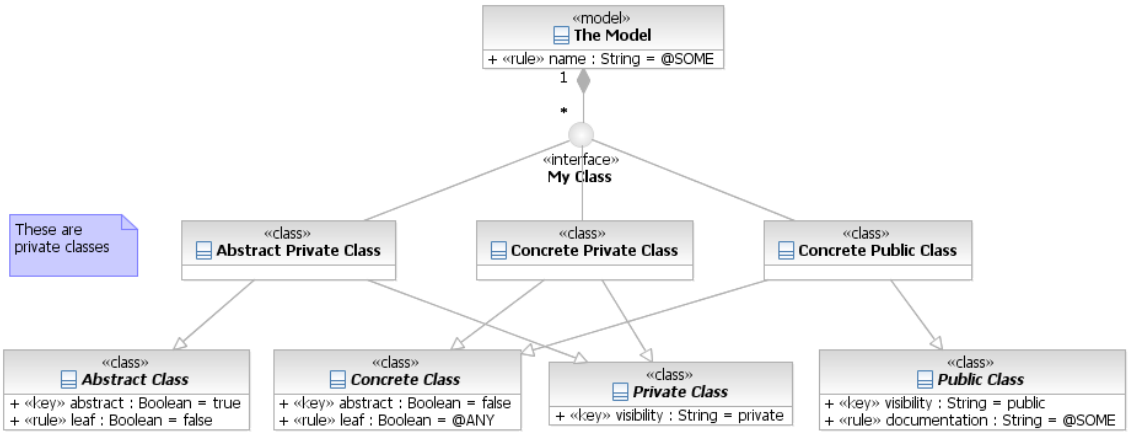
Constraints on the content of other diagrams, e.g. composite structure diagrams, communication diagrams, sequence diagrams, activity diagrams and state-chart diagrams should not explicitly be defined. The contents of these diagrams are implicitly defined by defining constraints on the contents in the elements that holds the diagrams, e.g. classifiers, interactions, actions and state machines.

13 Combined Property Constraints

If you have element properties, which values or significance is dependent on other properties in the same element. You may express that in metamodel by defining one metaclass for each valid combination of the properties and letting all those metaclasses be a sub-class to an abstract metaclass representing the overall concept.

This solution is clumsy for the end user, because he will be aware of all the different kind of metaclasses. For example they all will appear in the Add-menu.

A better solution is to capture the overall concept in an interface element in the metamodel, and letting all the metaclasses representing the different valid combinations be declared as private and have interface realization relations to that interface.



Example 38: *A class may be private or public and abstract or concrete, but not abstract and public in the same time. If the class is abstract, it cannot be a leaf. If the class is public the documentation is significant and it cannot be empty.*

The benefit of using interfaces in the metamodel is that the end-user will only be aware of the concept the interface represents. All private metaclasses realizing the interface is hidden for the end-user. The effect in MetaModelAgent is that only one add-option will be available and that the MetaModelAgent Add-wizard and the MetaModelAgent Property Tab will dynamically toggle the presence, change valid values and presented guidance for each property, to reflect the valid combinations.

If you add the operation **default()** to one of the private meta-classes, that meta-class will be chosen as the default variant in the add-wizard.

13.1 Restrictions on using Interfaces

There are some important restrictions in using interfaces in the metamodel.

- All metaclasses realizing an interface must be declared as private.
- A private metaclass must only realize one interface
- All private metaclasses realizing an interface must have the same set of properties defined as **«key»**-properties.
- An interface can't have any composite aggregations to metaclasses.

14 Constraint Constraints

As default, UML Constraints are regarded to be valid anywhere in a model. This may however be overridden in a metamodel, as described below

14.1 Defining Constraint Constraints

If a metamodel contains at least one metaclass with stereotype **«constraint»,** constraints are not valid anywhere in the model. Constraints will then only be valid in those positions that are explicitly expressed in the metamodel with **«constraint»** metaclasses, composited by metaclasses that represents valid owners of the constraints.

If a **«constraint»** metaclass exist in a metamodel without being composited by any other metaclass, constraints will be invalid anywhere in the model.

If no **«constraint»** metaclass exist in a metamodel, MetaModelAgent will upon activation automatically add a **«constraint»** metaclass to the internal representation of the metamodel and add all compositions needed for Constraints to be valid in any position in the model, making it possible to add constraints using the MMA Add Wizard. The metaclass will have all standard Constraints properties defined as well, making it possible to edit the Constraint properties using the MMA Property View.

15 UML Real-Time Modeling Support

This chapter only applies for those who use MetaModelAgent for UML-RT in RSARTE or HCL RTist.

There is a built-in metamodel in MetaModelAgent for UML-RT in those host tools that covers the whole language. However, if there is a need to restrict or adapt the usage of UML-RT a user-specfic metamodel can be developed and applied. This chapter covers what need be known to develop a custom metamodel for UML-RT.

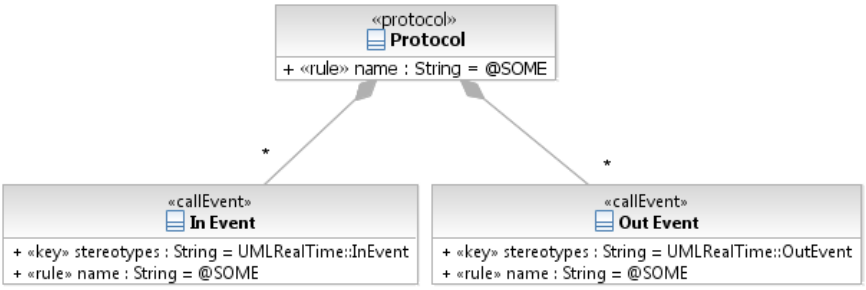
15.1 Protocols

A protocol is a UML-RT concept that is not represented as a single UML-element. A protocol is represented by a package, containing a collaboration and two interfaces. However, when looking on a protocol in the project explorer it seems like it is only a collaboration, stereotyped as **«protocol»** from the UMLRealTime-profile, the other elements are hidden from the user.

MetaModelAgent has native support for protocols by providing the «protocol»-stereotype in the MMA-profile. When developing metamodels you therefore can express the occurrence of a protocol by a metaclass with the stereotype «protocol». When using the Add-wizard to create a new protocol, a complete protocol representation is being created in the model.

15.2 Call Events

In Events and Out Events are allowed in a protocol in UML-RT. These events are represented by metaclasses with the stereotype set to **«callEvent»** and the metaclass attribute *stereotypes* with the default value set to *UMLRealTime::InEvent* and *UMLRealTime::OutEvent* respectively.



Example 39: *A metamodel representation of a Protocol that may have arbitrary number of In Events and Out Events.*

The UML-RT representation of an event also comprises a corresponding interface operation that is hidden from the user in the project explorer. The name of the event as seen in the project explorer is actually the name of the corresponding operation. The name of the event is always empty. This can be seen in the general tab of the Property View for the event.

Defining a name rule of an event in the metaclass actually means defining a name rule of the corresponding hidden operation. MetaModelAgent will therefore show the operation name as the event name in wizards and in the MMA Property View.

15.3 Capsules

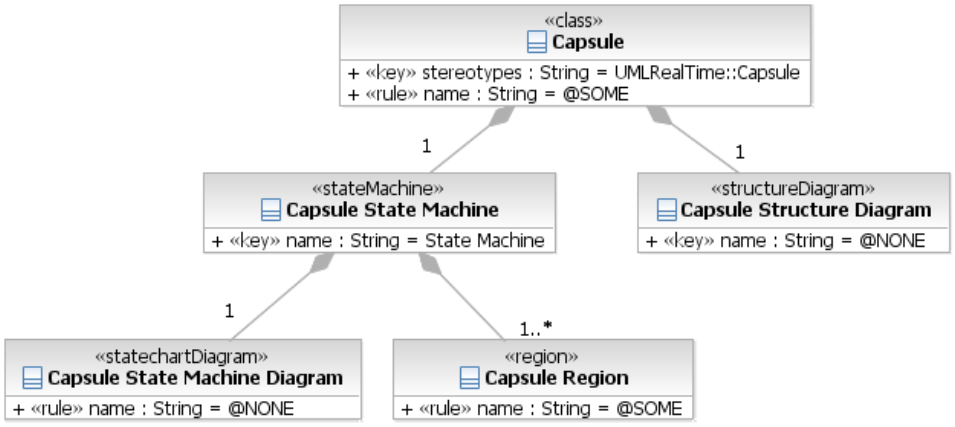
A capsule is a concept in UML-RT that is represented by a class with the stereotype «Capsule» from the UMLRealTime-profile.

However, when adding a new capsule using RSARTE or HCL RTist. The tool automatically adds a structure diagram and a state machine. Within the state machine a state chart diagram and a region are also automatically created, this is visible in the project explorer.

The nested elements of a capsule are crucial for the capsule to behavior properly in a model.

MetaModelAgent has no built in support for automatically generating the elements within a capsule. Instead you are supposed to express the correct element structure of a capsule in your metamodels.

The diagram below shows how you should represent a capsule in a metamodel, for UML-RT to behavior correctly.



Example 40: *Minimal representation of a UML-RT Capsule in a metamodel. You may notice that the diagram names should be empty. RSARTE and HCL RTist will present derived names in the project explorer.*

16 Organizing Metamodels

A metamodel defines the domain-specific language for a specific type of model. Normally, several different metamodels are needed to define the language for different kind of models. Several versions of the same metamodel may also be necessary to handle in parallel.

Each metamodel should be contained in an own UML-model. The model holding the metamodel should have the stereotype **«metaModel»**.

The metaclasses within a metamodel, can be structured into arbitrary number of sub-packages with the stereotype **«metaClassPackage»**. These sub-packages may also be nested.

16.1 Import between Metamodels

A metamodel can *import* the content from another metamodel or from a metaclass package within another metamodel. This is expressed by a *package import* relationship between the packages holding the metamodels.

A metamodel can import from arbitrary number of other metamodels.

Example 41: *A Metamodel importing from a metaclass package in another metamodel.*

The import mechanism is useful to reuse metaclass definitions and enumerations between several metamodels.

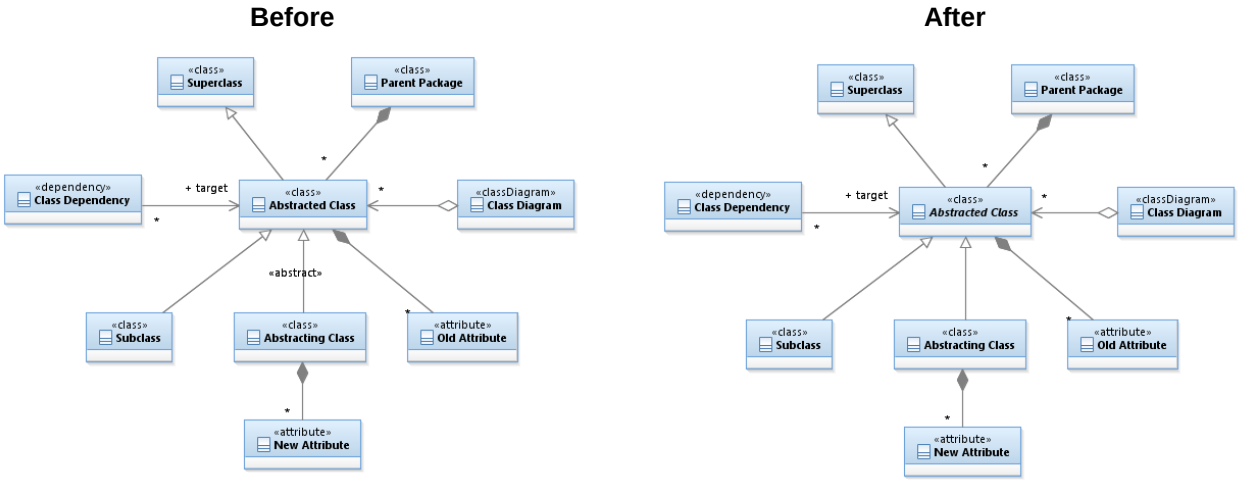
16.2 Extend and Adjust an Existing Metamodel

The metamodel notation used by MetaModelAgent today supports extensions of existing metamodels by the use of package import between metamodels and inheritance between their metaclasses. By doing so, new concepts could easily be added in the new metamodel based on concepts in an existing metamodel.

However, there is often a need for doing some adjustment, replacement or even removal of metaclasses in the metamodel being extended. This fact is supported by two different stereotypes on the generalization relationship and a special stereotype on metaclasses.

Override a Metaclass

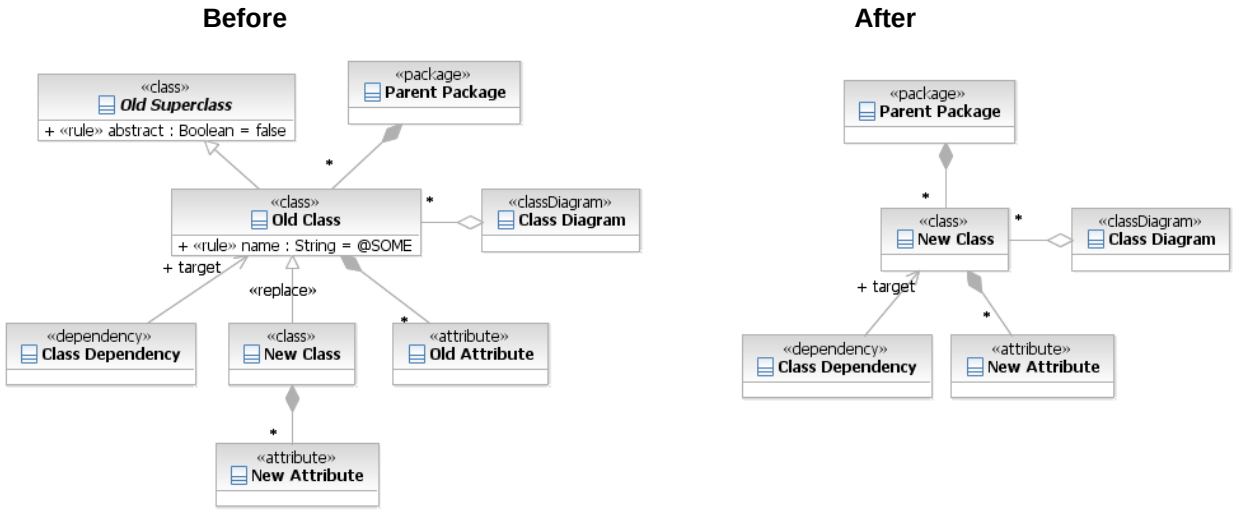
The stereotype **«abstract»** on a generalization between two metaclasses means that the generalized metaclass is regarded to be abstract. Besides that the generalization will have the standard semantics.



Example 42: *The metaclass "Abstract Class" will be treated as abstract and the "Abstracting Class" will inherited all attributes, relationships and aggregation from the "Abstract Class", as seen in the diagram to the right.*

Replace a Metaclass

The stereotype **«replace»** on a generalization between two metaclasses means that the generalized metaclass is regarded to be replaced entirely by the specialized metaclass in all occurrences and that no attributes, operations, aggregations, generalization or other relationships starting from the replaced metaclass will be considered any more.



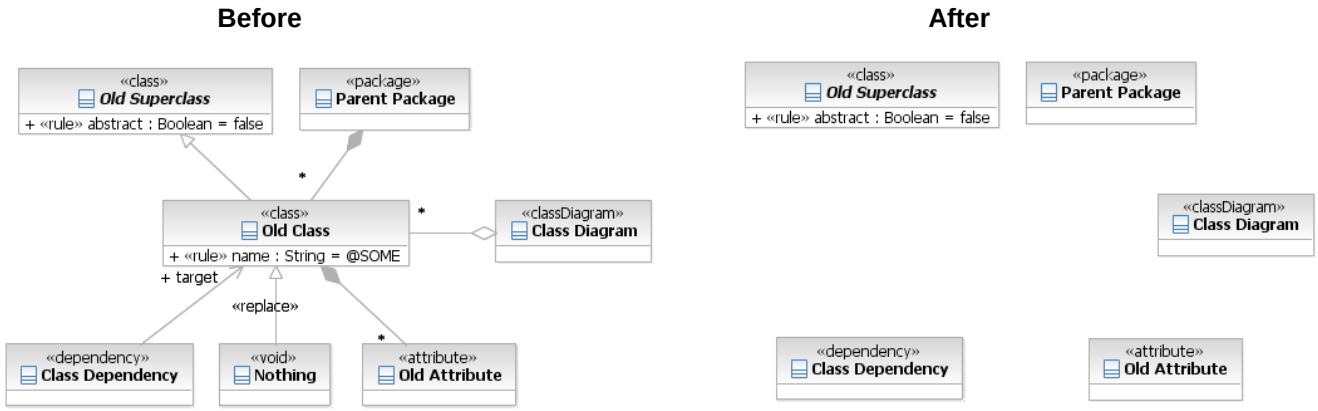
Example 43: *The "New Class" will replace the "Old Class" in all occurrences but not inherit is attributes, aggregates and other relationships, as seen in the diagram to the right.*

There are a few constraints to be aware of when using the **«replace»** stereotype:

- A metaclass must not have more than one **«replace»** generalization
- A metaclass must not be the super class of more than one **«replace»** generalization in the same time

Remove a Metaclass

The stereotype **«replace»** on a generalization can be combined with the stereotype **«void»** on the specialized metaclass. That means that the generalized metaclass will be replaced with nothing, e.g. it will not be considered, as if it has not exists at all.



Example 44: *The "Nothing"-class will remove the "Old Class" in all occurrences, as seen in the diagram to the right.*

There are a few constraints to be aware of when using the **«void»** stereotype:

- The **«void»** stereotype is only allowed on metaclasses that has a **«replace»** generalization towards another metaclass.
- If there are any other metaclasses referring to the removed metaclass directly, for example as target of a relationship, these metaclasses have to be replaced by a **«void»** metaclass as well, for the metamodel to be well-formed.

16.3 Documenting a Metamodel

A metamodel should be documented to make it easier to understand what all the constructions defined by the metaclasses represents.

The metamodel is documented by filling in the documentation property for packages, metaclasses and metaclass attributes within the metamodel:

- The documentation of a model holding a metamodel should explain what kind of model the metamodel represents.
- The documentation of a metaclass package should explain what is common between the metaclasses within the package.
- The documentation of a metaclass should explain what kind of item the metaclass represents. As part of the documentation of a metaclass, the keyword **@SUPER** can be inserted. The semantics of that keyword is that it will be replaced by the metaclass documentation from all inherited super metaclasses to the current metaclass

- The documentation of a metaclass operation named **suppressAdd** should explain why the corresponding elements cannot be added using MetaModelAgent and preferable some tips on how to add those elements.
- The documentation of a metaclass attribute should explain how to use the property defined by the attribute.
- For non-trivial property rules defined by the default values in metaclass attributes, you can provide an explicit explanation of valid values by entering the explanation in the **Default value name** property of the metaclass attribute. This property is derived from the name property of the attribute's default value.

RSAD/RSARTE and HCL RTist only: Besides documenting a metamodel in text, free-form diagrams can be used to show graphical examples on how the domain-specific language should be used. One or several free-form diagrams can be created in a metaclass package or in a metaclass. Both the name of the diagram and its documentation are presented in the published metamodel.

16.4 Using a Metamodel

A dependency relationship should be used to specify that a model or a package is an instance of a specific metamodel.

Example 45: *The use-case model "Order and Storage System" is an instance of the metamodel "Use-Case Modeling Guidelines"*

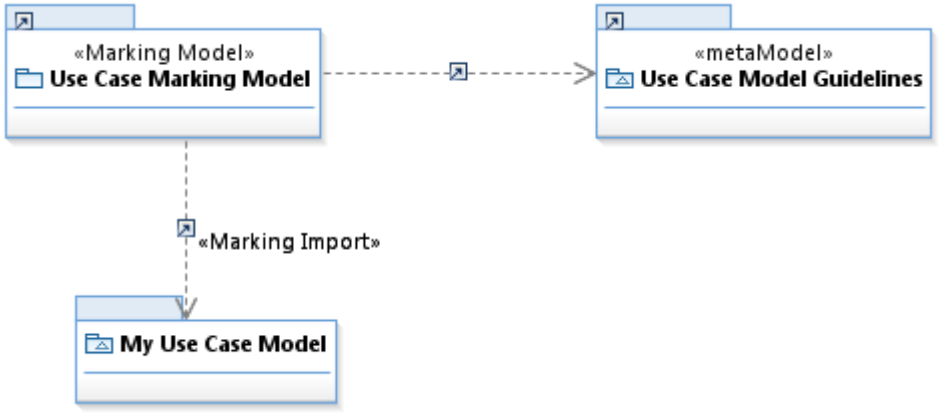
Several metamodels can in this manner be assigned to the same model or package. Each such metamodel must hold the complete DSML, but can be focused on different topics.

If the metamodel is deployed in a plugin and registered using the extension point provided by MetaModelAgent, you can omit the dependency-relationship as MetaModelAgent will found out which metamodel to use for which model. See chapter 18 for details

You may use the built-in meta-metamodel that comes with MetaModelAgent for any user-defined metamodel without an established dependency relationship. You may also use the built-in *General UML Guidelines* metamodel for any user-defined model without any established dependency relationship, see next chapter for details.

Support for Marking Models

If the concept of marking models is used you may assign a metamodel to the marking model instead of assigning it to the target of the marking model. If the marking model is opened when activating MetaModelAgent for the target model, the metamodel is available, otherwise not.



Example 46: *The metamodel is assigned to a marking model and not to the target model.*

17 Validating a metamodel

There are several complementary techniques for validating a metamodel to make sure that it will work as expected in MetaModelAgent.

This chapter describes the different techniques and how they are used

17.1 Built-in model validation (RSAD/RSARTE and HCL RTist)

RSAD/RSARTE and HCL RTist comes with a built-in model validation feature that checks a UML-model for some standard UML violations. This model validation feature can also be used for metamodels as they should also conform to standard UML

The model validation feature is initiated by selecting a model in the explorer view and then invoke *Model→Run Validation* from the main menu. Any problem will be listed in the standard Eclipse Problem View.

As the metamodel notation make use of some specific naming conventions for attributes and enumeration literals, you may get warnings that should be ignored.

17.2 Using the meta-metamodel

MetaModelAgent comes with a built-in meta-metamodel that contains a lot of rules for how metamodels should be constructed.

To validate the metamodel using the meta-metamodel, select the metamodel in the explorer view and select *MetaModelAgent*→Activate→Meta-metamodel from the context menu. Any problem will be listed in the MMA Problem View.

17.3 Using the Metamodel validation

MetaModelAgent has a built-in metamodel-validation feature that checks for ambiguities, conflicts and unused concepts in a metamodel.

You may notice that this metamodel validation feature is still in a beta stage, that means that the presentation of the validation result is very rough and not user-friendly, the presentation will be improved in future releases of MetaModelAgent.

- To validate a whole metamodel using the built-in metamodel validation feature, select the metamodel in the explorer view and select *MetaModelAgent→Activate→Validate Metamodel (Beta stage)* from the context menu. Any problem will be listed in the MetaModelAgent section of the standard Eclipse Console View.
- A comparison for ambiguities between the properties of two single metaclasses is performed by selecting the two metaclasses from the same metamodel in the context menu and then *MetaModelAgent→Activate→Compare Metaclasses (Beta stage)* from the context menu. The result will be a popup-dialog where overlaps between each restricted property definition are presented.
- A comparison for ambiguities in the generalization hierarchies below two metaclasses is performed by selecting the two metaclasses from the same metamodel in the context menu and then *MetaModelAgent→Activate→Compare Metaclass Generalization Trees (Beta stage)* from the context menu. The result will be a popup-dialog where overlapping metaclasses from the two hierarchies will be listed.

If the two metaclasses to be compared resist in two different metamodels, where one metamodel imports the other one, the metaclass selection must be made in a diagram, not in the explorer view, and the metaclass in the importing metamodel must be selected prior to selection of the metaclass in the imported metamodel. Otherwise there might be some metamodel error dialogs appearing.

The reliability of the metamodel validation feature is not 100%. There might be invalid problems presented as well as missing problems in the validation result. Especially the use of regular expression in property rules may result in invalid problems being presented. The validation result should therefore only be treated as an indication of problems that must be further inspected manually.

17.4 Using a test model

Some problems in the metamodel may only be detected by testing the metamodel by creating a model, activate the model using the metamodel and The final variant of validating a metamodel is to use MetaModelAgent and the metamodel to create and populate a model using the MMA Add Wizard and the Property View.

Create a new model based on the metamodel by using the MMA Create Model Wizard and then try to populate the model with all kinds of valid nested elements using the MMA Add Wizard. Try to edit the properties of each added element using the MMA Property View. For more information on how to use the wizards and views in MetaModelAgent, see the User Manual. Any metamodel-related problem will be revealed by popup-dialogs or by problems in the MMA Problem View.

Use the standard UI in the host tool for negative testing. Add elements and property values that are supposed to be invalid according to the metamodel and check that a corresponding problem is reported in the MMA Problem View.

This is a strenuous task that might be needed to make sure that the metamodel behaves as expected. Especially the application of profiles and usage of stereotypes, user defined properties and regular expressions should be tested this way.

17.5 Reviewing the metamodel

The final variant of validating a metamodel is to use the Guideline Publisher in MetaModelAgent to produce a web site of the metamodel that can be manually inspected.

To generate a web site of the metamodel, select the metamodel in the explorer view and then select *MetaModelAgent→Publish Guidelines…* from the context menu to bring up the Guidelines Publishing wizard, see the User Manual for details.

The resulting web site will provide a readable textual representation of the metamodel. A manual review can be used to detect invalid rules and missing rules.

.

18 Deploying a Metamodel

18.1 As a workspace model

The metamodel is deployed as any other model file that should be accessible by several users.

1. Remove the reference from the metamodel to the meta-metamodel so that the meta-metamodel is not needed in the workspace for the users.
2. Put the metamodel file in a common read-only area on your file system
3. Make sure that each user imports the metamodel into their workspaces
4. Optionally, publish the guidelines from the metamodel on your intranet.

   You may notice that if your metamodel does not contain any **«constraint»** metaclass, MetaModelAgent will automatically add a **«constraint»** metaclass as a valid nested element to all other metaclasses representing elements that can hold a constraint. This will be visible in the web published metamodel.

The users are ready to use MetaModelAgent and the metamodel to support them in using the DSML.

18.2 In a plugin

An alternative way to deploy a metamodel is to include it as a model library in an Eclipse plug-in. The plug-in is then installed for all users that should use the metamodel. The benefit of this alternative of this solution is that the metamodel will not be visible in the users work space. The drawbacks are that menu operations on the metamodel, for example creating a new model based on the metamodel is not available.

For a metamodel to become a model library, just add the stereotype **«modelLibrary»** to the model and register the model library using the following extension points:

Tool	Extension Point
RSAD/RSARTE and HCL RTist	`com.ibm.xtools.uml.msl.UMLLibraries`
Eclipse Papyrus	`org.eclipse.papyrus.uml.extensionpoints.UMLLibrary`

A Metamodel deployed in a plugin or in the workspace could also be registered into MetaModelAgent by the same plugin using the extension point `com.adocus.mma.metamodel.provider`.

Registering the metamodel requires that you provide an implementation of the Java interface `com.adocus.mma.api.IMetaModelProvider`.

The interface defines a single method, `getMetaModel()`, that takes an `EModelElement` object as the only parameter and returns the name of the corresponding metamodel to be used for that element.

If the metamodel to be referred is stored in the user's workspace, the name returned should be the file path to the metamodel-file following the following syntax:

`platform:/resource/`*`project-name/metamodelfile`*`.emx` (RSAD/RSARTE)

`platform:/resource/`*`project-name/metamodelfile`*`.uml` (Papyrus)

By registering a metamodel in MetaModelAgent, there will be no need for a dependency relationship from the user's models to the metamodel. When MetaModelAgent is about to activate your model, the corresponding metamodel, according to your implementation of the `getMetaModel()`-method, will automatically be used.

For more information on model libraries and plug-ins, please refer to the online documentation for your specific host tool.

19 Models included in MetaModelAgent

When MetaModelAgent has been installed, the following profile and models are available, however they are not visible in the workspace

19.1 MetaModelAgent Profile

The profile provides stereotypes for metamodels and should be applied to all metamodels that should be used by MetaModelAgent. See *Appendix* A for a summary of all stereotypes.

19.2 Meta-Metamodel

This meta-metamodel is the metamodel for metamodels, i.e. it defines the DSML for metamodels. All user-defined metamodels may be validated towards the meta-metamodel. There is no need for a dependency from the user-defined metamodel to the meta-metamodel to be able to use MetaModelAgent.

19.3 General UML Guidelines Metamodel

This metamodel is a complete metamodel for standard UML, as supported by MetaModelAgent. An end user may use MetaModelAgent with this metamodel for any kind of UML model. There is no need for a dependency between the model and the *General UML Guidelines* metamodel to be able to use MetaModelAgent.

19.4 General UML-RT Guidelines Metamodel

This metamodel is an extension to the *General UML Guidelines* metamodel adding UML-RT specific concepts such as Capsules and Protocols. This metamodel is only available in MetaModelAgent for RSAD/RSARTE and HCL RTist.

There is no need for a dependency between an UML-RT model and the *General UML-RT Guidelines* metamodel to be able to use MetaModelAgent.

19.5 Metamodel Template

This is a model template of a metamodel which can be used as a base when developing a user-defined metamodel. The template is available in the New Model Wizard.

Appendix A Stereotypes

This is a summary of all stereotypes that are defined in the MMA UML-profile to be used when defining metamodels for MetaModelAgent

Class stereotypes representing UML elements

The following stereotypes are applicable for classes which are metaclasses and represents UML elements:

Category	Stereotypes		
Containers	«model»	«package»	«profile»
Classifiers	«activity» «actor» «artifact» «association» «class» «collaboration» «communicationPath» «component» «dataType»	«deploymentSpecification» «device» «enumeration» «executionEnvironment» «extension» «informationItem» «interaction» «interface» «node»	«opaqueBehavior» «primitiveType» «protocol»[1] «protocolStateMachine» «signal» «stateMachine» «stereotype» «useCase»
Features	«attribute» «associationEnd» «collaborationUse» «connector» «connectorEnd»	«extensionEnd» «extensionPoint» «operation» «port» «parameter»	«property» «reception» «templateParameter» «templateSignature»
State Machine elements	«finalState» «guard» «protocolTransition»	«pseudoState» «state»	«transition» «trigger»
Activity elements	«action» «activityPartition» «controlFlow»	«connectionPointReference» «controlNode» «objectFlow»	«pin» «objectNode» «region» «structuredActivityNode»
Interaction elements	«combinedFragment» «gate»	«interactionUse» «message»	«lifeline»
Relationships	«abstraction» «componentRealization» «dependency» «deployment» «elementImport» «extend»	«generalization» «include» «informationFlow» «interfaceRealization» «manifestation» «packageImport»	«packageMerge» «protocolConformance» «realization» «substitution» «templateBinding» «usage»
Others	«callEvent» «constraint»	«instanceSpecification» «slot»	

[1] Protocols can only be used in Rational Software Architect Real Time Edition.

Class stereotypes representing Diagrams

Tool support	Stereotypes		
All tools	«activityDiagram» «communicationDiagram» «componentDiagram» «deploymentDiagram»	«InteractionOverviewDiagram» «sequenceDiagram» «stateChartDiagram» «structureDiagram»	«TimingDiagram» «useCaseDiagram»
RSAD, RSATE and HCL RTist	«freeformDiagram» «objectDiagram»		
Papyrus	«packageDiagram» «requirementDiagram»	«blockDefinitionDiagram» «InternalBlockDiagram»	«profileDiagram» «parametricDiagram»

Other Class Stereotypes

Besides the stereotypes representing UML elements and diagrams there are two other stereotypes applicable to metaclasses:

Stereotype	Description
«item»	Represents an arbitrary UML-item and is only applicable for abstract metaclasses.
«void»	Denotes the removal of a metaclass when combined with a «replace» generalization.

Attribute Stereotypes

The following stereotypes are applicable for metaclass attributes that represents properties:

Stereotype	Description
«key»	Represents a key property for the items the metaclass defines. The attribute value must be fulfilled for the item to be considered an instance of the meta class.
«rule»	Represents a mandatory rule for the items the metaclass defines. It is regarded to be an error if this value is not fulfilled.
«rec»	Represents a recommendation for the items the metaclass defines. It is regarded to be a warning if this value is not fulfilled.
«info»	Represents an information issue for the items the metaclass defines.

Aggregation Stereotypes

The following stereotypes are applicable for composite and shared aggregations between metaclasses:

Stereotype	Description
«rule»	A mandatory rule for the items the metaclass defines. It is regarded to be an error if this value is not fulfilled.
«rec»	A recommendation for the items the metaclass defines. It is regarded to be a warning if this value is not fulfilled.
«info»	An information issue for the items the metaclass defines.

Aggregations without any applied stereotype are regarded to be a mandatory rule, therefore the «rule»-stereotyped may be omitted.

Generalization Stereotypes

The following stereotypes are applicable for generalizations between metaclasses.

Stereotype	Description
«abstract»	The generalized metaclass is regarded as abstract, besides that the standard generalization semantics applies.
«replace»	The generalized metaclass is replaced by the specialized metaclass in all occurrences. No attributes, aggregations or other relationships are owned by the specialized metaclass.

Generalizations without any applied stereotype have standard generalization semantics.

Container Stereotypes

Stereotype	Applicable for	Description
«metaModel»	Model	The model defines a metamodel. This stereotype is mandatory for all models in the profile.
«metaClassPackage»	Package	The package contains metaclasses. This stereotype is mandatory for all packages in the profile.

Appendix B Meta Class Attributes

Attributes representing standard UML element properties

The table below shows valid metaclass attributes representing standard UML element properties. Each attribute represents a specific property for the kind of item the metaclass represents. Attributes where the type is a UML Element can alternatively be modeled as an association to the metaclass representing the type. For more information see ref. [1].

UML Elements in *italic* style in table denotes abstract concepts.

Attribute	Applicable for	Type
association	Association End	Association
abstract	*Behavioral Feature*	Boolean
abstract	*Classifier*	Boolean
active	Class	Boolean
aggregation	Property	Aggregation Kind Enumeration
behavior	Port	Boolean
behavior	Call Behavior Action	Behavior
body	Opaque Behavior	String
classifier	Instance Specification	String
composite	State	Boolean
concurrency	*Behavioral Feature*	Call Concurrency Kind Enumeration
connectorKind	Connector	Connector Kind Enumeration
constrainedElement	Constraint	Element *(multi-valued)*
context	*Behavior*	String
contract	Connector	Behavior *(multi-valued)*
control	Pin	Boolean
controlType	*Object Node*	Boolean
conveyed	Information Flow	Classifier *(multi-valued)*
covered	Combined Fragment	Lifeline *(multi-valued)*
defaultValue	Property	String
defaultValue	Parameter	String
definingEnd	Connector End	Property
definingFeature	Slot	Property
derived	Association	Boolean
derived	Property	Boolean
derivedUnion	Property	Boolean
direction	Parameter	Parameter Direction Kind Enumeration
edge	Activity Partition	Activity Edge *(multi-valued)*
effect	Parameter	Parameter Effect Kind Enumeration
entry	Connection Point Reference	Pseudo State
event	Trigger	String
exit	Connection Point Reference	Pseudo State
exception	Parameter	Boolean

Attribute	Applicable for	Type
fileName	Model	String
guard	*ActivityEdge*	String
incoming	*ActivityNode*	*ActivityEdge (multi-valued)*
inGroup	*ActivityNode*	*ActivityGroup*
inPartition	*ActivityNode*	*ActivityPartition*
incoming	*Vertex*	Transition
interactionOperator	Combined Fragment	InteractionOperatorKind Enumeration
keywords	*Element*	String *(multi-valued)*
language	Guard	String
language	Opaque Behavior	String
leaf	*Redefinable Element*	Boolean
message	Gate	Message
messageKind	Message	Message Kind Enumeration
messageSort	Message	Message Sort Enumeration
multicast	Object Flow	Boolean
multiplicity	*Multiplicity Element*	String
multireceive	Object Flow	Boolean
mustIsolate	Structured Activity Node	Boolean
name	*Named Element*	String
name	*Diagram*	String
navigable	Property	Boolean
node	Activity Partition	Activity Node *(multi-valued)*
operation	Call Event	Operation
operation	Call Operation Action	Operation
ordered	*Multiplicity Element*	Boolean
ordering	*Object Node*	Ordering Kind Enumeration
orthogonal	State	Boolean
outgoing	*Vertex*	Transition *(multi-valued)*
outgoing	*Activity Node*	*Activity Edge*
partWithPort	Connector End	Property
port	Trigger	Port
pseudoStateKind	PseudoState	PseudoState Kind Enumeration
query	Operation	Boolean
raisedException	*Behavioral Feature*	Type *(multi-valued)*
readOnly	Activity	Boolean
readOnly	Property	Boolean
receiver	Message	Message End *(multi-valued)*
receives	Lifeline	Message *(multi-valued)*
redefinedBehavior	*Behavior*	*Behavior*
redefinedClassifier	*Classifier*	*Classifier*
redefinedConnector	Connector	Connector
redefinedEdge	*ActivityEdge*	*ActivityEdge*
redefinedInterface	Interface	Interface

Attribute	Applicable for	Type
redefinedNode	*ActivityNode*	*ActivityNode*
redefinedOperation	Operation	Operation
redefinedPort	Port	Port
redefinedProperty	Property	Property
redefinedState	State	State
redefinedTransition	Transition	Transition
redefinitionContext	*Redefinable Element*	*Redefinable Element*
reentrant	*Behavior*	String
referred	Protocol Transition	Operation *(multi-valued)*
refersTo	Interaction Use	Interaction
render		*(multi-valued)*
represented	Information Item	*Classifier (multi-valued)*
represents	Lifeline	String
role	Connector End	*Connectable Element*
selection	*Object Node*	*Behavior*
selection	Object Flow	*Behavior*
service	Port	Boolean
sender	Message	Message End
sends	Lifeline	Message *(multi-valued)*
signal	Send Signal Action	Signal
signal	Reception	Signal
signature	Message	String
signature	Template Binding	Template Signature
simple	State	Boolean
singleExecution	Activity	Boolean
source	*Activity Edge*	*Activity Node*
source	*Directed Relationship*	*Element*
source	Transition	*Vertex*
specification	Guard	String
specification	*Behavior*	String
static	*Feature*	Boolean
stereotypes	*Element*	String *(multi-valued)*
stream	Parameter	Boolean
subject	Use Case	*Classifier (multi-valued)*
submachineState	State	Boolean
substitutable	Generalization	Boolean
transitionKind	Transition	Transition Kind Enumeration
target	*Activity Edge*	*Activity Node*
target	*Directed Relationship*	*Element*
target	Transition	*Vertex*
transformation	Object Flow	*Behavior*
type	*Typed Element*	*Type*
type	Connector	Association

Attribute	Applicable for	Type
unique	*Multiplicity Element*	Boolean
unmarshall	Accept Event Action	Boolean
upperBound	Object Node	Integer
value	Value Pin	String
value	Enumeration Literal	String
value	Slot	String
weight	Activity Edge	String
viewKind	Diagram	String
viewpoint	Model	String
visibility	Package Import	Visibility Kind Enumeration
visibility	Element Import	Visibility Kind Enumeration
visibility	*Named Element*	Visibility Kind Enumeration

Enumeration values

List of valid values of the enumeration types in the table above

Enumeration	Valid values
aggregation	composite, none, shared
concurrency	concurrent, guarded, sequential
connectorKind	assembly, delegation
controlNodeKind	activityFinalNode, decisionNode, finalNode, flowFinalNode, forkNode, initialNode, joinNode, mergeNode
direction	in, out, inout
effect	create, delete, read, update
interactionOperator	Alt, assert, break, consider, critical, ignore, loop, neg, opt, par, seq, strict
messageKind	complete, found, lost, unknown
messageSort	asynchCall, asynchSignal, createMessage, deleteMessage, reply, synchCall
objectNodeKind	activityParameterNode, centralBufferNode, dataStoreNode, expansionNode
ordering	fifo, lifo, ordered, unordered
pinKind	actionInputPin, inputPin,outputPin,valuePin
pseudoStateKind	choice, deepHistory, entryPoint, exitPoint, fork, initial, join, junction, shallowHistory, terminate
transitionKind	external, internal, local
visibility	public, protected, private, package

Special attributes

Beside the attributes representing standard UML properties, some special attributes have been defined in the metamodel-notation. These are listed below

Attribute	Applicable for	Type	Description
documentation	*Element*	String	Represents the body property of the first owned comment of an element.
actionKind	*Action*	Enumerated *(read-only)*	Represents the kind of action that the metaclass represents. See valid values in table below.
controlNodeKind	*Control Node*	Enumerated *(read-only)*	Represents the kind of control node that the metaclass represents. See valid values in table below.
fileName	Package	String *(read-only)*	represents the file name that are used to store the model (or package, if the model is fragmented)
ObjectNodeKind	*Object Node*	Enumerated *(read-only)*	Represents the kind of object node that the metaclass represents. See valid values in table below.
pinKind	*Pin*	Enumerated *(read-only)*	Represents the kind of object node that the metaclass represents. See valid values in table below.
parent	*Element*	*Element*	Can be used to explicitly represent the parent of the element. Often combined with «key» stereotype to indicate that the parent (owner) must be fulfilled for an element to match.
structuredActivity NodeKind	*Structured Activity*	Enumerated *(read-only)*	Represents the kind of structured activity node that the metaclass represents. See valid values in table below.
upper	Multiplicity Element	String	The upper level of a multiplicity, an alternative to multiplicity.
lower	Multiplicity Element	String	The upper level of a multiplicity, an alternative to multiplicity.

Enumeration values

List of valid values of the enumeration types in the table above

Enumeration	Valid values
actionKind	acceptCallAction, acceptEventAction, addStructuralFeatureValueAction, addVariableValueAction, broadcastSignalAction, callBehaviorAction, callOperationAction, clearAssociationAction, clearStructuralFeatureAction, clearVariableAction, conditionalNode, createLinkAction, createLinkObjectAction, createObjectAction, destroyLinkAction, destroyObjectAction, expansionRegion, loopNode, opaqueAction, raiseExceptionAction, readExtentAction, readIsClassifiedObjectAction, readLinkAction, readLinkObjectEndAction, readLinkObjectEndQualifierAction, readSelfAction, readStructuralFeatureAction, readVariableAction, reclassifyObjectAction, reduceAction, removeStructuralFeatureValueAction, removeVariableValueAction, replyAction, sendObjectAction, sendSignalAction, sequenceNode, startClassifierBehaviorAction, structuralFeatureAction, structuredActivityNode, testIdentityAction, unmarshallAction, valueSpecificationAction, writeStructuralFeatureAction
controlNodeKind	activityFinalNode, decisionNode, finalNode, flowFinalNode, forkNode, initialNode,

	joinNode, mergeNode
objectNodeKind	activityParameterNode, centralBufferNode, dataStoreNode, expansionNode
pinKind	actionInputPin, inputPin,outputPin,valuePin
structuredActivity NodeKind	conditionalNode, expansionRegion, loopNode, sequenceNode

Papyrus-specific attributes

The following attributes are specific for metaclasses representing diagrams in Papyrus:

Attribute	Type	Description
diagramKind	String	Refers to a custom DSML-diagram kind provided by some external DSML-extension.
rootElement	Element	Refers to the root element, or context, of the diagram. An activity diagram should for example has the root element set to an activity. If omitted in the metamodel, the root element will automatically be set to the owner of the diagram.
stylesheet	String	Path to a CSS-stylesheet to be used for the diagram

Appendix C Meta Class Operations

Operations on metaclasses are used to affect MetaModelAgent´s behavior.

Operation name	Behavior
default	Indicates the default meta-class in a context with several private meta-classes realizing an interface.
external	Instances of the metaclass are not allowed in the model.
permitAll	Instances of the metaclass may contain any other kind of item without any restrictions.
suppressAdd	Instances of the metaclass should not be able to be added by using MetaModelAgent wizards.
suppressChange	Instances of the metaclass should not be able to be changed by using MetaModelAgent wizards.
unique	If the metaclass represents a classifier, this operation indicates that the name of the classifier must be unique in the model. If the metaclass represents a directed relationship, this operation indicates that there must not be more than one instance of this relationship between the same pair of elements. This operation has no effect on metaclasses representing other elements than classifiers and relations.

Appendix D Regular Expressions

Regular expressions can be used as default values for metaclass attributes.

By using regular expression it is easy to specify complex patterns that the item properties, represented by the metaclass attribute, must fulfill.

Meta class attribute expression	Explanation
name : String = (A-Z).*	The name must start with an uppercase alphabetic letter
name : String = UC\d{3}:.*	The name (of a use case) must start with 'UC' followed by three digits, a colon, and an arbitrary text string

The following summary is copied from the Java 2 Standard Edition 5.0 API documentation:

Regular expression	Matches
`x`	The character x
`\\`	The backslash character
`[abc]`	`a`, `b`, or `c` (simple class)
`[^abc]`	Any character except `a`, `b`, or `c` (negation)
`[a-zA-Z]`	`a` through `z` or `A` through `Z`, inclusive (range)
`[a-d[m-p]]`	`a` through `d`, or `m` through `p`: `[a-dm-p]` (union)
`[a-z&&[def]]`	`d`, `e`, or `f` (intersection)
`[a-z&&[^bc]]`	`a` through `z`, except for `b` and `c`: `[ad-z]` (subtraction)
`[a-z&&[^m-p]]`	`a` through `z`, and not `m` through `p`: `[a-lq-z]`(subtraction)
`.`	Any character
`\d`	A digit: `[0-9]`
`\D`	A non-digit: `[^0-9]`
`\s`	A whitespace character: `[ \t\n\x0B\f\r]`
`\S`	A non-whitespace character: `[^\s]`
`\w`	A word character: `[a-zA-Z_0-9]`
`\W`	A non-word character: `[^\w]`
`\p{Lower}`	A lower-case alphabetic character: `[a-z]`
`\p{Upper}`	An upper-case alphabetic character:`[A-Z]`
`\p{Alpha}`	An alphabetic character:`[\p{Lower}\p{Upper}]`
`\p{Digit}`	A decimal digit: `[0-9]`
`\p{Alnum}`	An alphanumeric character:`[\p{Alpha}\p{Digit}]`
`\p{Punct}`	Punctuation: One of `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`

Regular expression	Matches
`\p{Blank}`	A space or a tab: `[ \t]`
`\p{Space}`	A whitespace character: `[ \t\n\x0B\f\r]`
X?	*X*, once or not at all
*X**	*X*, zero or more times
X+	*X*, one or more times
X{*n*}	*X*, exactly *n* times
X{*n*,}	*X*, at least *n* times
X{*n*,*m*}	*X*, at least *n* but not more than *m* times
XY	*X* followed by *Y*
X|*Y*	Either *X* or *Y*
(*X*)	X, as a capturing group

Appendix E Pre-defined Keyword

The following keyword can be used for common property constraints:

Keyword	Semantics for the item property
@SOME	The property value must not be empty.
@NONE	The property value must be empty.
@ANY	The property can have any value.
@OTHER	The property can have any value besides those represented by the type
@INSTANCE	The property must have an instance corresponding to the element type as the value
@EMPTY	Only allowed as value of an enumeration literal. Represents an empty literal.

Appendix F Known Limitations

UML Element types not supported

Category	Unsupported elements
Interaction Fragments	ActionExecutionSpecification, BehaviorExecutionSpecification, ConsiderIgnoreFragment, Continuation, ExecutionOccurrenceSpecification, ExecutionSpecification, InteractionOperand, MessageOccurrenceSpecification, OccurrenceSpecification, PartDecomposition, StateInvariant
Events	AnyReceiveEvent, ChangeEvent, CreationEvent, DestructionEvent, ExecutionEvent, MessageEvent, ReceiveOperationEvent, ReceiveSignalEvent, SendOperationEvent, SendSignalEvent, SignalEvent, TimeEvent
Template Parameters	ClassifierTemplateParameter, OperationTemplateParameter, ConnectableElementTemplateParameter
Structured Activity Nodes	ConditionalNode, ExpansionRegion, LoopNode, SequenceNode
Value Specifications	Duration, DurationInterval, Expression, InstanceValue, Interval, LiteralBoolean, LiteralInteger, LiteralNull, LiteralSpecification, LiteralString, LiteralUnlimitedNatural, OpaqueExpression, StringExpression, TimeExpression, TimeInterval
Constraints	DurationConstraint, InteractionConstraint, IntervalConstraint, TimeConstraint
Intervals	DurationInterval, TimeInterval
Observations	DurationObservation, TimeObservation
Link End Data	LinkEndCreationData, LinkEndDestructionData
Directed Relationships	ProfileApplication
Others	Clause, ExceptionHandler, ExtensionEnd, GeneralizationSet, GeneralOrdering, InterruptibleActivityRegion, ParameterSet, QualifierValue, TemplateParameterSubstitution, Variable

UML Elements without an own Metaclass Stereotype

There are some UML2-elements that are not represented by an own metaclass stereotype in the metamodel notation. They can however be represented by a metaclass stereotype representing a more general element in the UML2 element inheritance hierarchy. This can be used for validation but are not useful for element creation. That means that for example an Association Class can be validated (as an Association) but not created using the MetaModelAgent Add wizard.

The following elements can be represented by the listed metaclass stereotypes:

Element	Corresponding stereotype
ClassifierTemplateParameter	«templateParameter»
ConnectableElementTemplateParameter	«templateParameter»
OperationTemplateParameter	«templateParameter»
AssociationClass	«association»
FunctionBehavior	«opaqueBehavior»
RedefinableTemplateSignature	«templateSignature»

Other Limitations

- The order of different kind of parameters to an operation cannot be specified in a metamodel.
- Entry, exit and do-activities on a state cannot be specified separately in a metamodel.
- Whether a property or operation is inherited or owned by a classifier cannot be specified in a metamodel.
- Only one source element and one target element of a dependency are supported.
- Only one classifier of an instance specification is supported.
- Associations with more than two association ends are not supported.
- The meta-metamodel does not allow Enumerations in the metamodel to be referred by directed association from a metaclass. Enumerations must be referred by a metaclass attribute, where the enumeration is the type of the attribute.
- Enumeration literals are incorrectly regarded to be inheritable between Enumerations.

Exceptions from the UML2.x metamodel

Most of the metamodel notation defined in this manual are consistent the UML 2.x specification when it comes to representing elements in metaclasses and element properties in metaclass attributes as well as compositions to represent model structure. However there are a few exceptions to the consistency with UML 2.x, which are important to understand when defining metamodels. These exceptions are summarized in this appendix.

- Dependencies, and all variants of dependencies, are regarded to be owned by the source element instead of owned by a package. This means that there should be a composition from the metaclass representing the source element to the metaclass representing the dependency. The consequence of this is also that only one source element of a dependency is supported.
- The owner of an association is not respected. Metaclasses representing association should not participate as part in a composition. Metaclasses representing association should be referred by navigable associations from metaclasses representing its association ends. The navigable end should have the name *association*, referring to the UML property with the same name.
- Association ends are always regarded to be owned by the element in the opposite end. E.g. they should always be part of a composition from a metaclass representing the element in the opposite end.

Appendix G The Metamodel Template

This is diagram showing a generic base pattern for a metamodel. All created metamodels should start with this pattern.

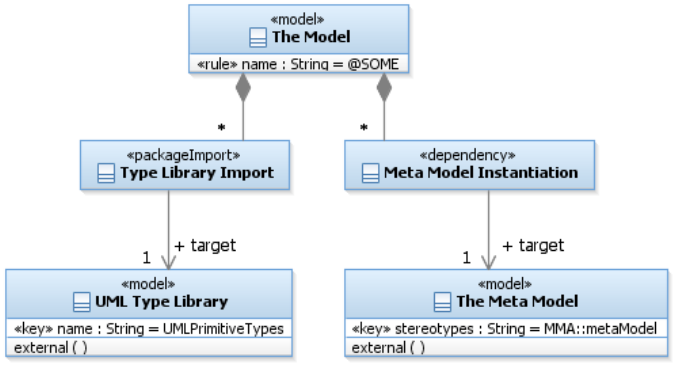


Figure 1: *The fundamentals of a metamodel.*

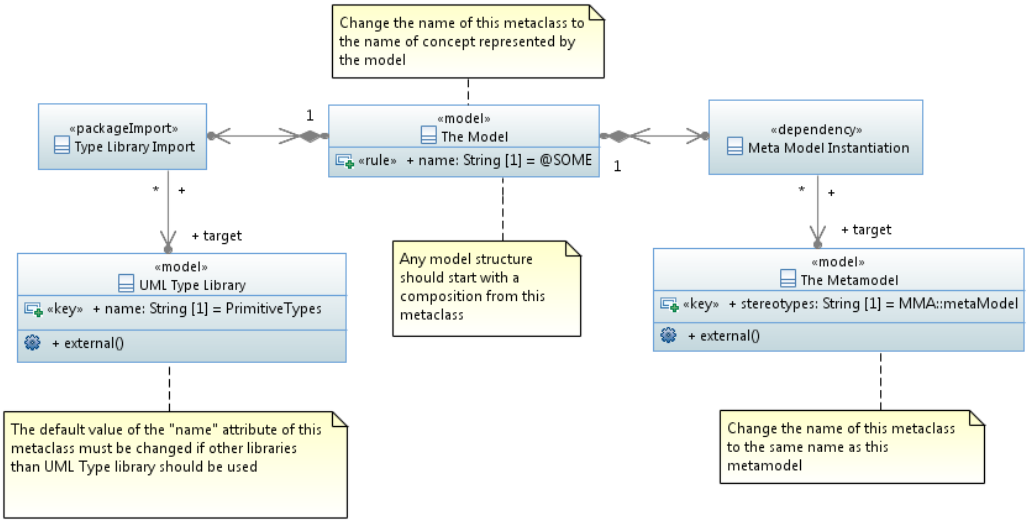


Figure 2: *The fundamentals of a metamodel in Papyrus*

This template is provided as a model template in MetaModelAgent.

The Meta Model Template contains five mandatory metaclasses

- *The Model* represents the top element in a model based on the metamodel.
- *Type Library Import* indicates that the model may have relationships to models containing type libraries.
- *UML Type Library* represents an external model containing UML primitive types to be used in the model. In Papyrus, the name of this library is “PrimitiveTypes”
- *Meta Model Instantiation* indicates that the model must have at least on dependency to a metamodel.
- *The Meta Model* represents an external model holding the DSML for the model. The Meta Model actually represents the Meta Model Template.

Appendix H Example of a Metamodel

This is an example of a complete DSML for use-case modeling expressed as a metamodel using the profile in this document. This example is available for download at www.metamodelagent.com.

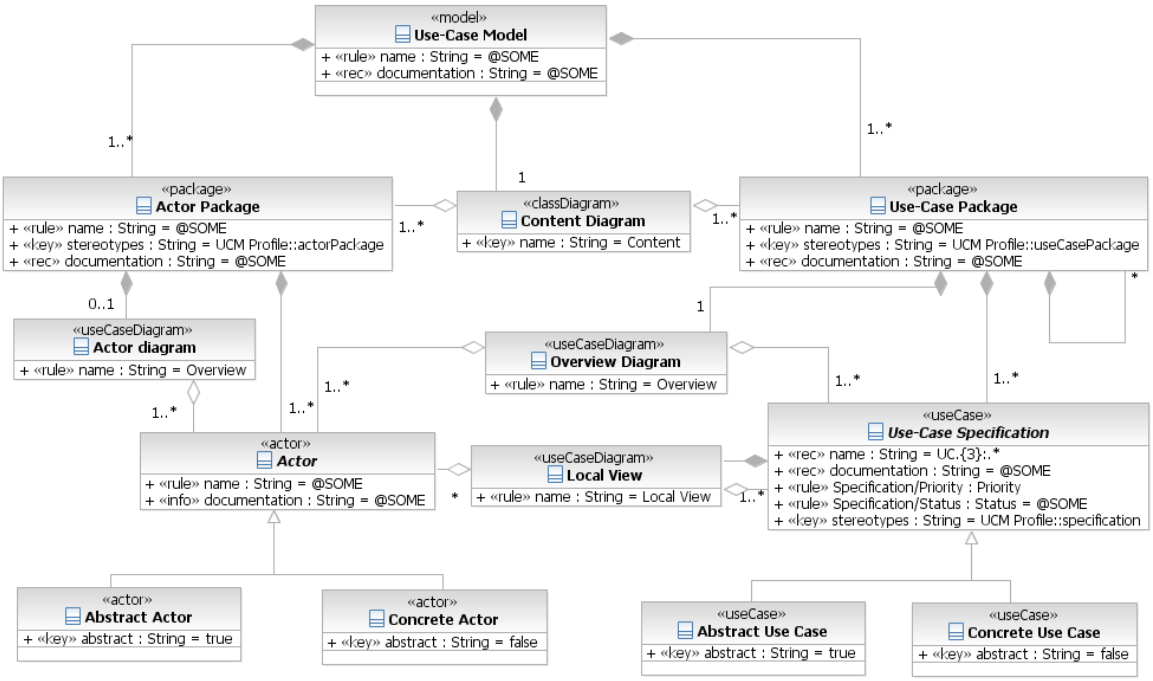


Figure 3: *The constraints on the model structure of a use-case model.*

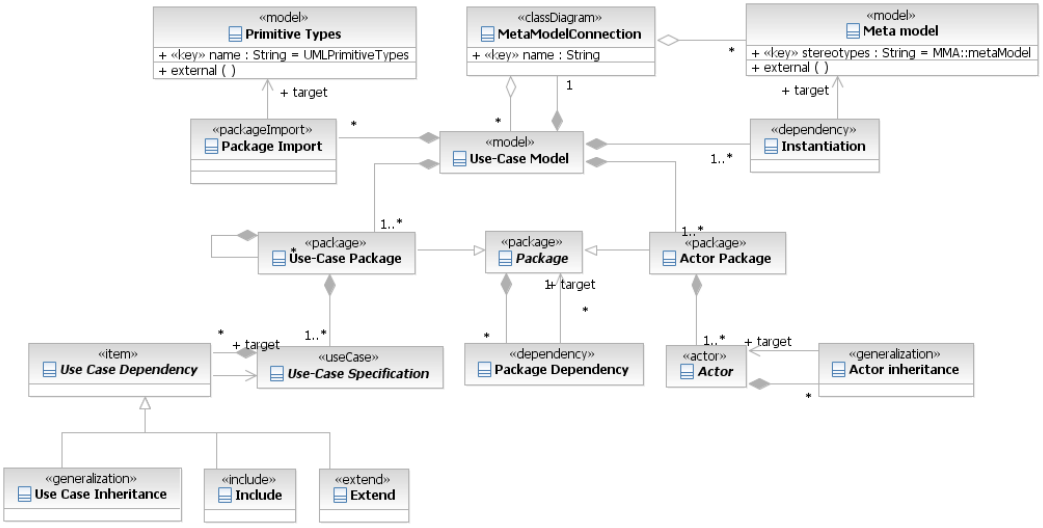


Figure 4: *The constraints on relations, except communication, within a use-case model.*

Appendix I Migrating Instructions

Below you find migrations needed to be made in metamodels when upgrading from an older version of MetaModelAgent.

From version 4.2.0 to version 4.2.1

No migration needed.

From version 4.2.1 to version 4.2.2

When using RSAD/RSARTE and Papyrus 2.0: No migration needed.

When migrating from Papyrus 2.0 to Papyrus 3.0 the MMA Metamodel architecture context should be set for the metamodels to be used. This can be made by the Architecture Switch menu entry in the Model Explorer view.

From version 4.2.2 to version 4.5.0

No migration needed.