Et bilde som inneholder tekst, utklipp

Automatisk generert beskrivelse ****

**The AKM Modeller**

**Training Course Content**

**Introduction**

The AKM Modeller (AKMM) is a generic platform for developing and utilizing knowledge models. It can be customized to support any subject area, if the area can be described by entities of different types and by relationships between those types. Both entities and relationships may be described by relevant properties and associated methods.

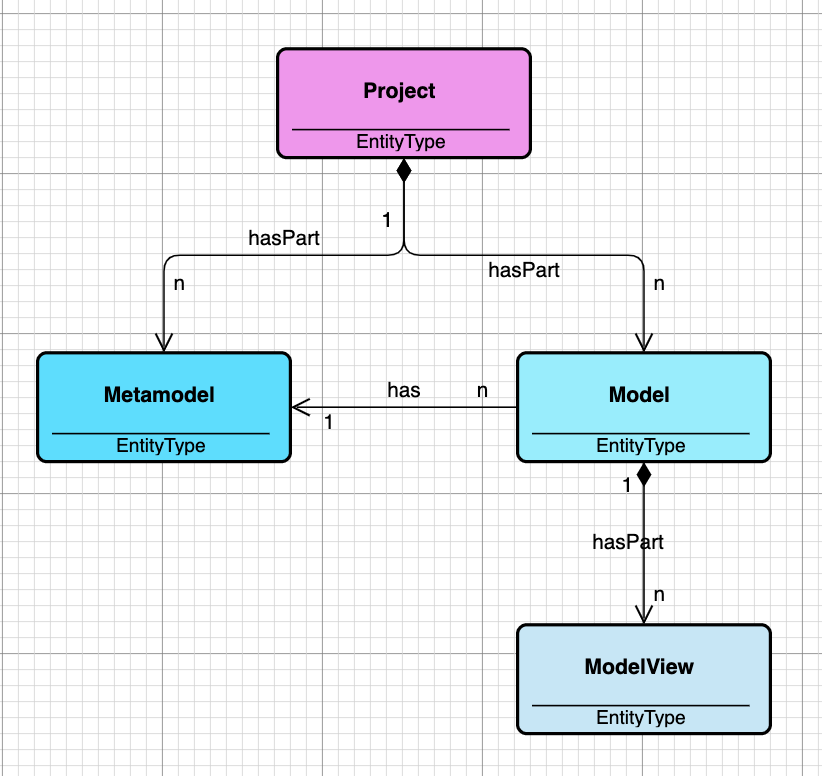
The user interface is implemented as a graphic modelling environment where each concept is visualized graphically to ease communication between users. The visualization of objects and relationships is customizable.

The Modeller differentiates between a meta-model (a modelling language) and a model. The meta-model defines the object types and relationship types - e.g. may *Person* and *Car* be your object types, while *owns* or *rents* may be your relevant relationship types between those object types. Based on such a meta-model you may build a model of actual persons and cars and connect the relationships between the persons and their cars.

**Concepts**

The models in AKMM are organized and managed in **Projects**, as shown in the figure below.

**The Project**



A Project may contain one or more Metamodels, and one or more Models.

A Model is based on one Metamodel, but several Models may share the same Metamodel.

To see the content of a model, Modelviews are used. A Modelview shows visualizations of objects and relationships.

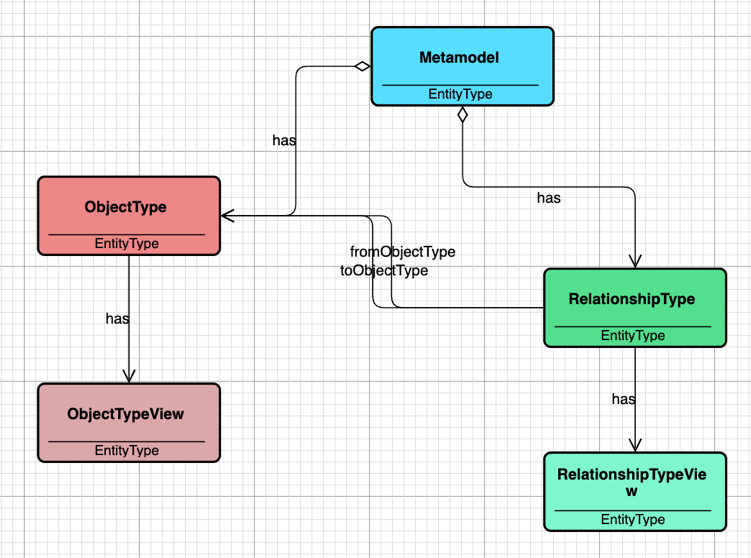
**The Metamodel**

A fundamental concept in AKMM is the **Metamodel** as illustrated in the figure below.

On the base level the Metamodel consists of Object types and Relationship types, and the definition of how they play together.

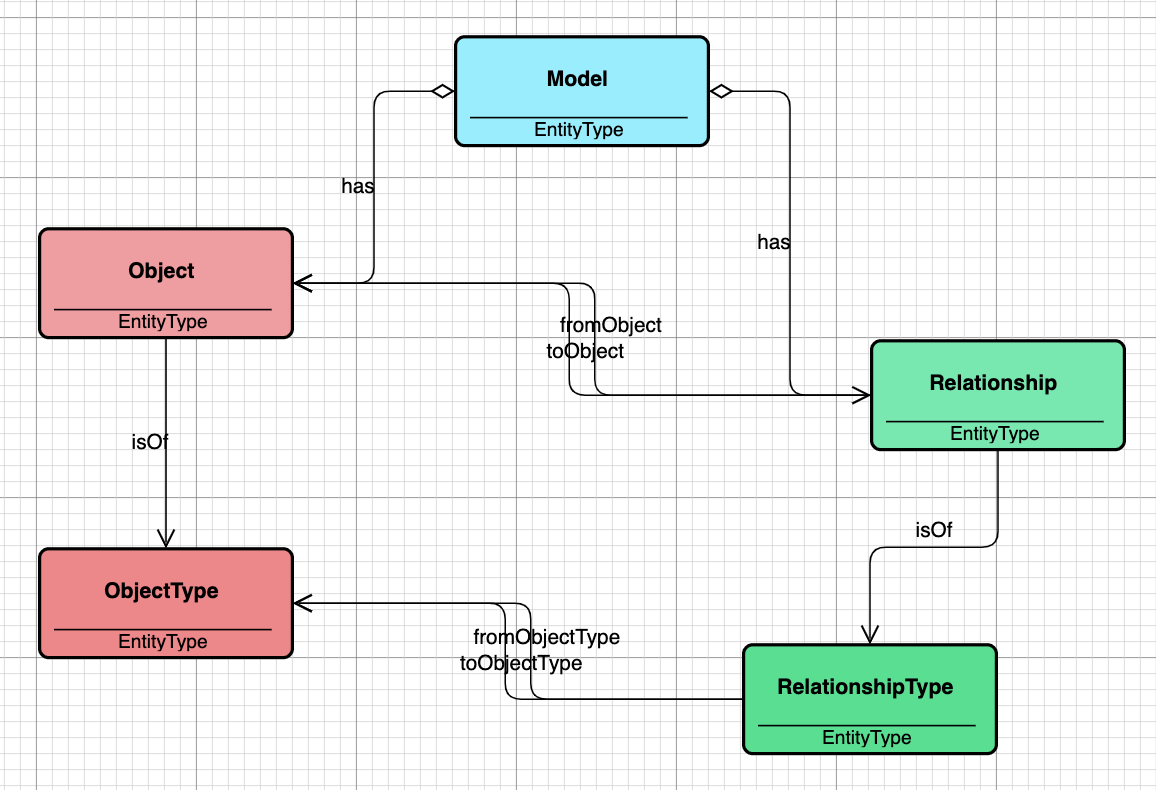
A Relationship type points to two object types, the FROM object type and the TO object type, which tells us that a relationship in AKMM has a direction, which normally is shown in the model views with an arrow pointing to the TO object.

In addition, the Metamodel contains Object Typeviews, that define how the objects of a given type are visualized, and Relationship Typeviews that define how relationships of a given type are visualized.



**The Model**

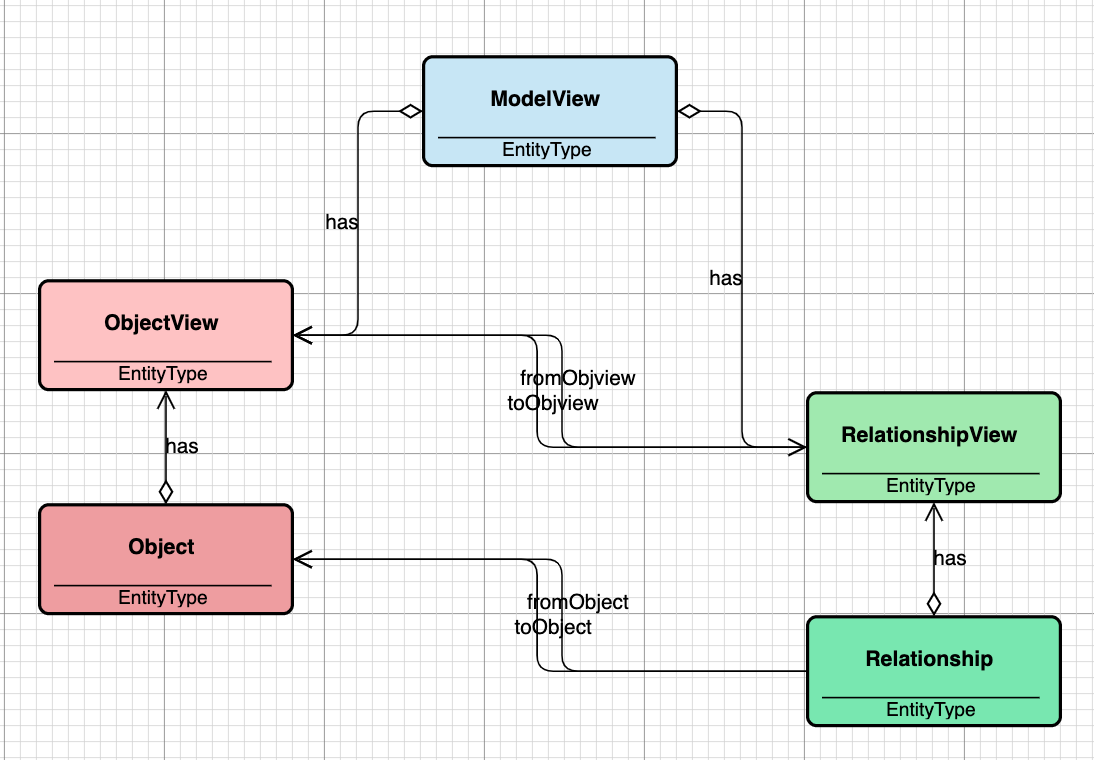
A Model contains objects and relationships. An Object refer to its type – the Object type, while a Relationship refers to its Relationship type, as illustrated below.



**The Modelview**

The model content is visualized in one or more Modelviews. Each Modelview contains Object views and Relationship views as illustrated below. Each Object view refers to the Object it represents, as do the Relationship views.

One Object may have several Object views, even in the same Modelview, while it is most common to be visualized in different Modelviews. The same goes for Relationship views.



**About relationships**

Relationship types have some attributes that are specific to relationships.

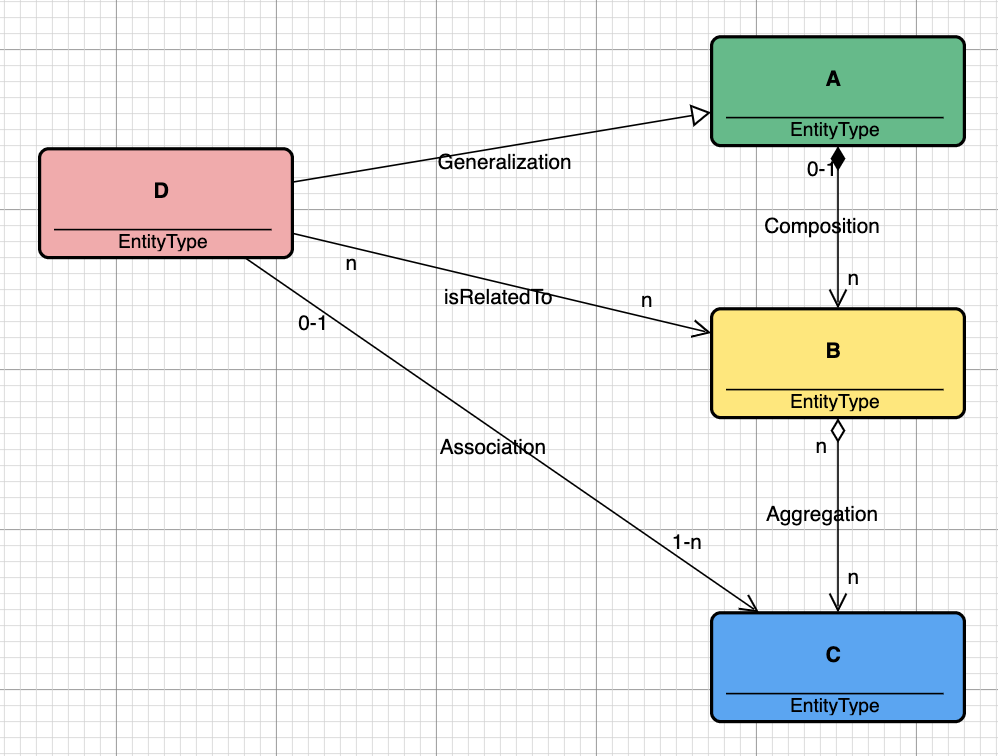
These are:

* Relationship kind
* Cardinality from
* Cardinality to
* Name from
* Name to

The meaning of Relationship kind is derived from UML (Unified Modeling Language), that differentiates between the following kinds of relationships:

* Association
* Generalization
* Composition
* Aggregation

In the figure below both *isRelatedTo* and *Association* are UML associations, in the sense that there are no constraints on the meaning of the relationships.



*Generalization*, on the other hand, has a very specific meaning. It means that “D” in the model above *inherits from* “A”. So, if A has properties, D will have the same properties.

*Composition* describes a “has part” relationship, meaning that if an object of type “A” has several parts of type “B”, and the object of type “A” is deleted, then all the parts of type “B” will also be deleted.

*Aggregation*, on the other hand, describes a “has member” relationships, meaning that if a parent object is deleted, the members will NOT be deleted. They exist independent of the parent.

Relationship *cardinality* is also supported, i.e. the ability to define e.g. one-to one, one-to many, many-to many relationships, or other combinations. The cardinality is specified on both ends of the relationship type.

You may also give a *from-name* and a *to-name* to the relationship type.

**Building Metamodels**

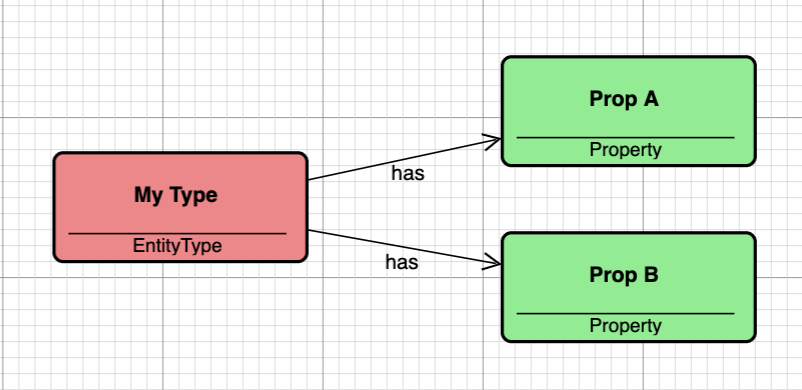
Metamodels are modeled by using ***EntityType*** to represent object types and a relationship of type ***isRelatedTo*** to represent relationship types.

Both are named or renamed to the actual type names the user wants.

**Defining Object types**

To define an object type starts with giving the type a meaningful name – a name that tells users what exactly the type represents. It is good practice to add a description to clarify the purpose.

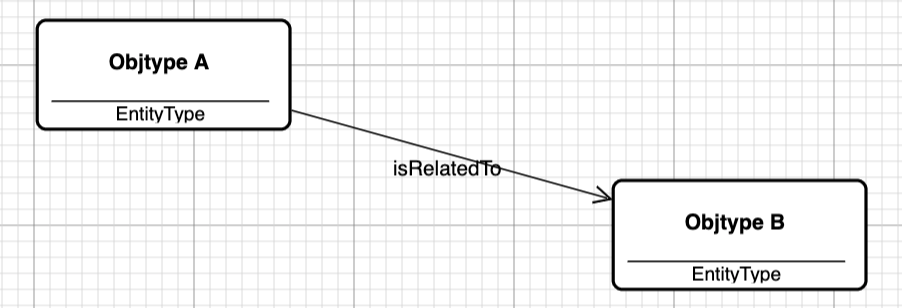
The next step would be to add Properties to the type definition, as shown below.



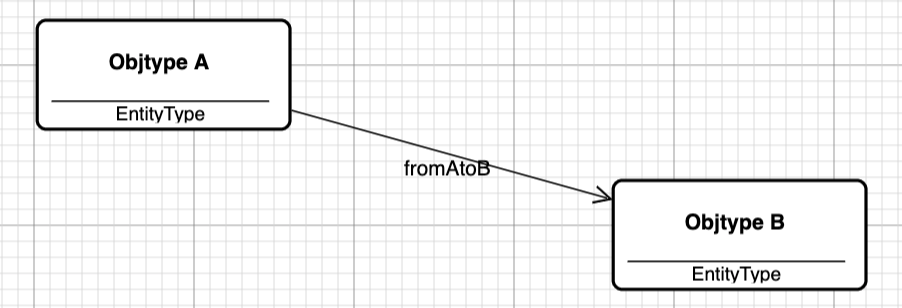
In this case the object type “*My Type*” is specified to have two properties, “*Prop A*” and “*Prop B*”. With no further specification the two properties will be of datatype “string”.

**Defining Relationship types**

Relationship types are defined by drawing a relationship of type “*isRelatedTo*” between object types as shown below.



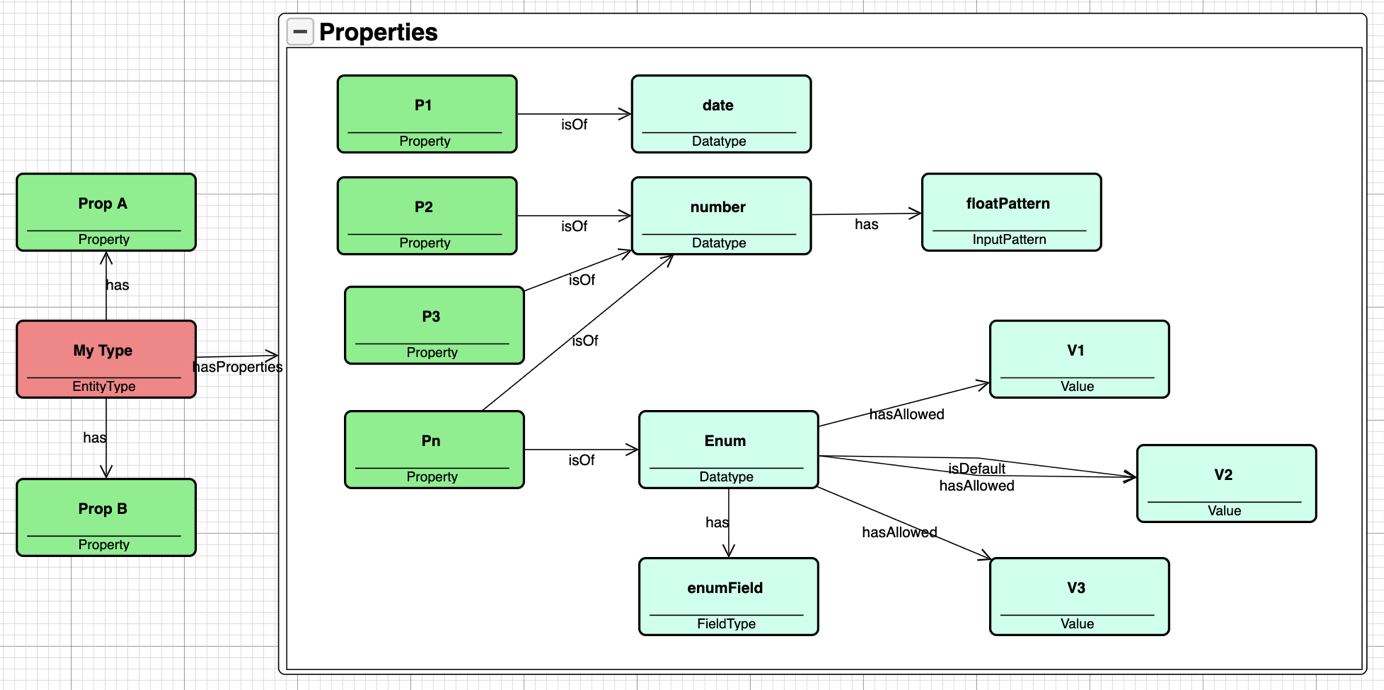
Then change the relationship name so it contains the relationship type name.



**Property modelling**

AKMM offers property modelling to a much more detailed level than just saying that the property is a “string” property. See below.

In this diagram properties have been modeled inside a container. Instead of drawing relationships to each property, a relationship “hasProperties” has been drawn from the type to the container. This implies that all properties inside the container are regarded as properties in “My Type”. This possibility has been added to simplify property modelling in case of many properties.



The figure shows “My Type” with the two properties as defined in the example above. But in addition the type has now 4 other properties, “P1”, “P2”, “P3” and “Pn”. Unlike the first two properties, these 4 properties have been specified to a greater detail.

* “P1” is a date property.
* “P2” is a number property with a pattern specified that limits the allowed values
* “P3” has the same definition as “P2”
* “Pn” is an Enum with 3 allowed values, “V1”, “V2”, “V3”, where “V2” is the default value.

**Property and type methods**

AKMM supports two kinds of methods – property and type methods. This is illustrated in the figure below.

In the figure two methods have been defined:

* “calc A” is a property method that calculates the value of “Prop A” based on a formula or expression that typically contains values of other properties in the same object or in related objects, or a combination of the two
* “Mtd” is a type method, i.e. a method that is meant to be executed on an instance of the actual type. One example of such a method is a method that traverses a hierarchy and executes an action on each node in the hierarchy



Methods are of a type – a method type. Currently there are three method types defined:

* calculateValue
* aggregateValue
* traverse

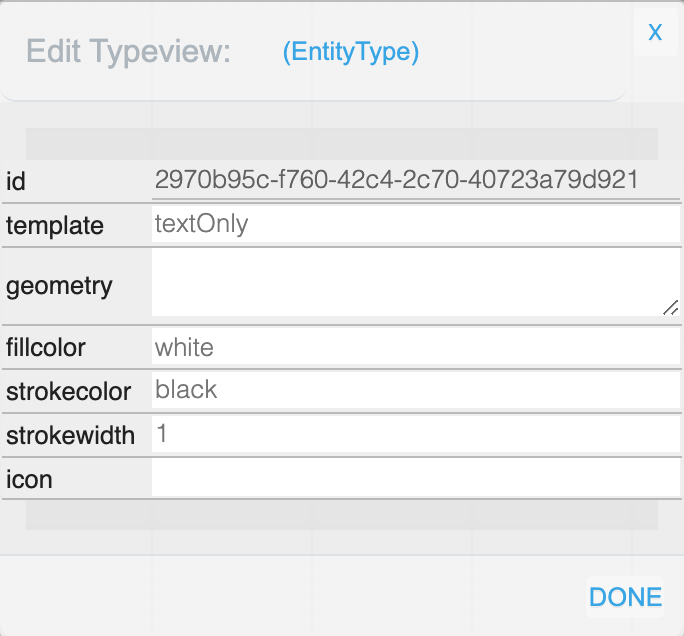
The first two are valid for property methods, while the last is valid for type methods. Documentation of these method types and examples of use are found in a section later in this document.

**Visualization of objects and relationships**

**About Object typeviews**

It is the object typeview that specifies what an instance of a given type will look like when it is first created in a model.

The figure below shows the dialog used to specify a given object typeview.



The fields *fillcolor*, *strokecolor* and *strokewidth* has to do with the *shape* that surrounds the object. In the examples in this document the *shape* is a rectangle. But in principle it can be any shape.

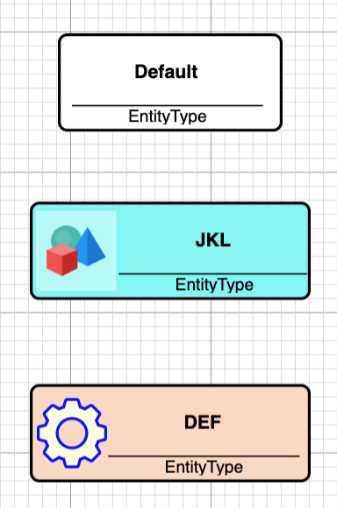
The shape is defined by the node template, i.e. the *template* field in the dialog.

At the time this is written, 3 templates are supported, which are:

* textOnly
* textAndIcon
* textAndGeo

The templates are illustrated in the following figure:

|  |
| --- |
| textOnly |
| textAndIcon |
| textAndGeometry |



*textAndIcon* uses the value of the *icon* field in the typeview dialog to identify the icon. This is a reference to a file that contains the icon, such as a *bmp* file, a *png* file or *svg* file.

The *geometry* used in *textAndGeometry* is stored in the *geometry* field, in a format like what *svg* uses.

**About Relationship typeviews**

It is the relationship typeview that specifies what an instance of a relationship type will look like when it is first created in a model.

The figure below shows the dialog used to specify a given relationship typeview, that contains the default settings.

Et bilde som inneholder tekst

Automatisk generert beskrivelse

The meaning of the fields *strokecolor* and *strokewidth* is self-evident. So is *textcolor*.

The field *dash* has three values: *None*, *Dashed* and *Dotted*, telling whether the line is dashed, dotted or not.

The field *fromArrow* specifies whether the from side of the relationship will have an arrow or not, and if so, what kind of arrow. The same goes for *toArrow*.

If the arrows contain a closed area, like a diamond or a triangle or a circle, the color of that area is specified in the fields *fromArrowColor* and *toArrowColor*.

**Building Metamodels and Models**

**1. Define a new project**

When AKMM is started the user meets a modelling window that looks like the following:

Et bilde som inneholder bord

Automatisk generert beskrivelse

The project is called “INIT-Start Project” which is ok if you want to play with the tool. But if your intention is to build models for a specific purpose, you should start by giving the project a name and description that is meaningful to you. This is done by right clicking the background and select “*Edit Project*”.

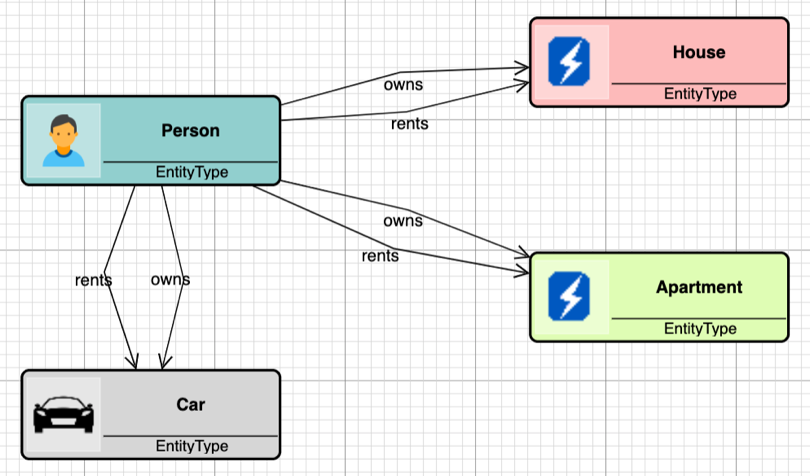
It is also recommended to change the model name (“*Edit Model*”) to a more meaningful name.

When the project and model naming is done, it is recommended to save the model to your local filesystem, which is done by clicking on the “*File*” button above the modelling window. Then choose “*Save Project (all) to File*”.

The next time you start AKMM you should click on “*File*” and then choose “*Import from file*” and select one of the models you previously saved.

**2. Build a model**

In this chapter we will use an example metamodel as defined below. It has four different object types, *Person*, *House*, *Apartment* and *Car*, with relationship types *owns* and *rents* as shown in the next model diagram.



Based on this metamodel the modelling window looks as shown here:

Et bilde som inneholder bord

Automatisk generert beskrivelse

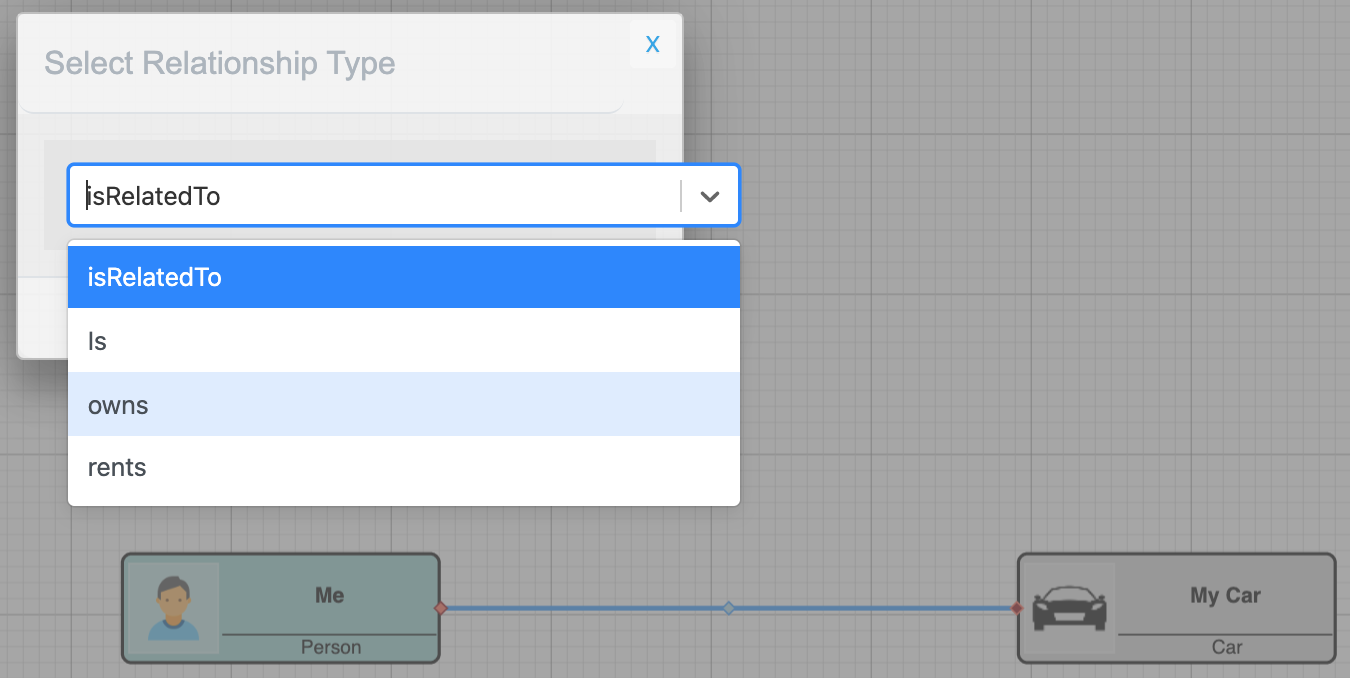
The left pane contains the four object types defined in the metamodel above plus “*Generic*” and “*Container*”, which are found in all metamodels used for modeling in AKMM.

Now we build our model by first dragging and dropping first Person and then Car. We click on each of them to edit their names, and the result is:

Et bilde som inneholder shōji

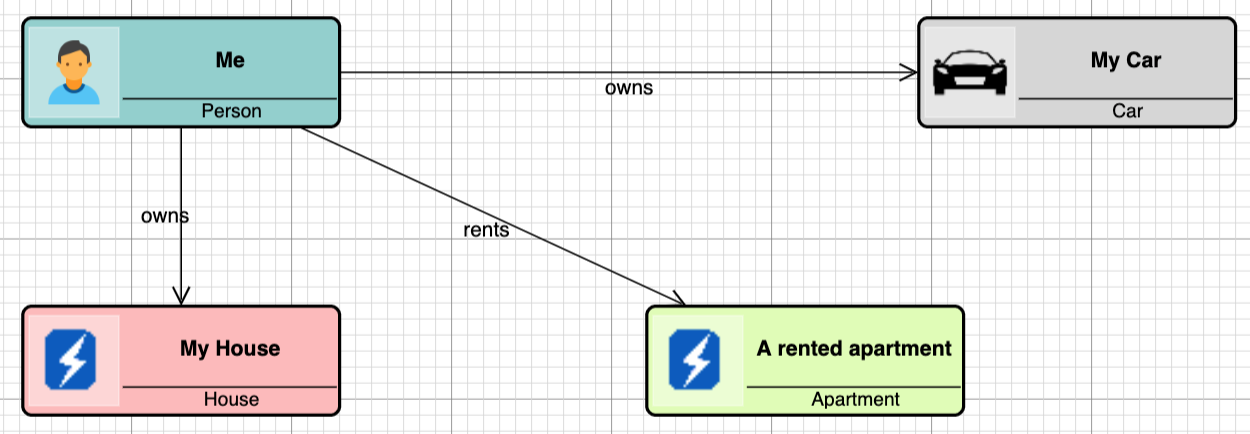
Automatisk generert beskrivelse

Then when we draw a relationship between the two objects, we are asked to choose a relationship type:



We choose “*owns*” in the modal dialog that pops up, clicks on “*Done*” and the relationship is created. (If you click on the “x” in the top right corner of the dialog, the operation is canceled.)

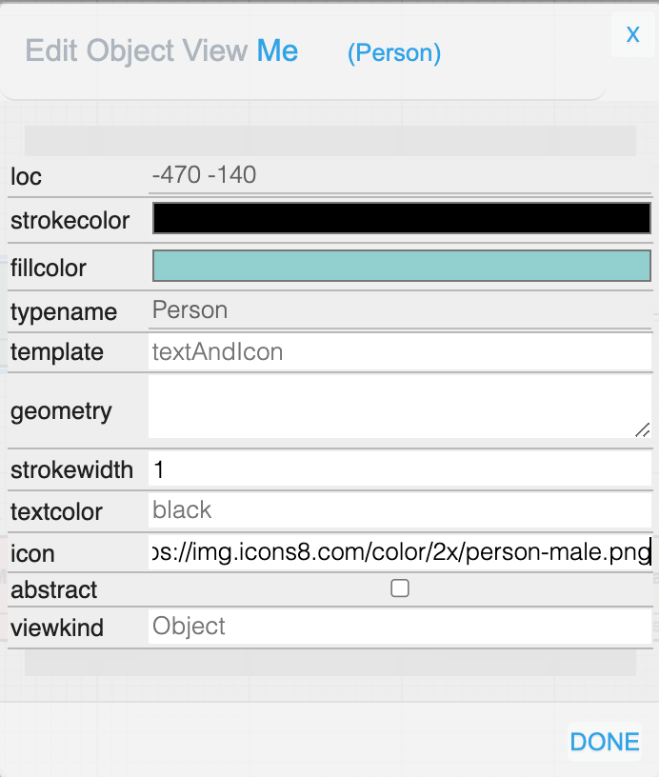
We continue modeling objects and relationships and may end up with a model like this:



**Modifying object and relationship views**

How objects and relationships appear in the model when they are created is decided by the typeview definitions of the respective types. The appearance in the model can be overridden by defining object and relationship views.

The objectview of the person “*Me*” looks like this:



We see that we can modify the *fillcolor* of the object, the “*strokecolor*”, the “*icon*” as so on, and then give the object a specialized look that differs from the default appearance.

We can do the same with relationships, and modify the color, the arrowheads, and so on, when we want to deviate from the default.

**3. Organizing model content using Modelviews**

As explained in the **Concepts** chapter, the model content may be organized in several modelviews. The reason for doing so may vary.

One reason may be to have views of different levels of detail, such as showing high level concepts in one modelview, and detailed views of the different concepts in separate modelviews.

An example of this would be to focus on object and relationship types in one modelview and types with their properties in a second modelview.

In another situation, if you have a large model covering separate areas such as business or technical, it would be natural to use different modelviews for the separate areas.

What to remember is, if the same object appears in two or more modelviews, you should do a “*Copy*” in the first modelview and do a “*Paste View”* in the second modelview.

The same applies if you want the same object-relationship structure to appear in different modelviews. Do a “*Copy*” in the first modelview followed by a “*Paste View*” in the second.

You create a modelview by choosing “New Modelview” in the popup menu on the background. And you can do “Edit Modelview” if you want to change the name or the description.

If you have several modelviews they will be sorted by their names. If you want a specific sequence, you may let the first part of the name be a number. In that way you can control the sequence.

**Layout customization**

In each modelview you can specify:

* automatic layout
* routing and curve
* show cardinality on/off

There are a few built-in layout algorithms that you can choose to use if you want:

* “*Circular*” layout
* “*Grid*” layout
* “*Tree*” layout
* “*ForceDirected*” layout
* “*LayeredDigraph*” layout

The default is “*Manual*” layout.

It is recommended that you test the different layout algorithms to see if any suits your need.

You can also define different kinds of “*Routing*”. The alternatives are:

* Normal
* Orthogonal
* Avoids Nodes

*“Normal”* means straight lines, as shown in most examples in this document.

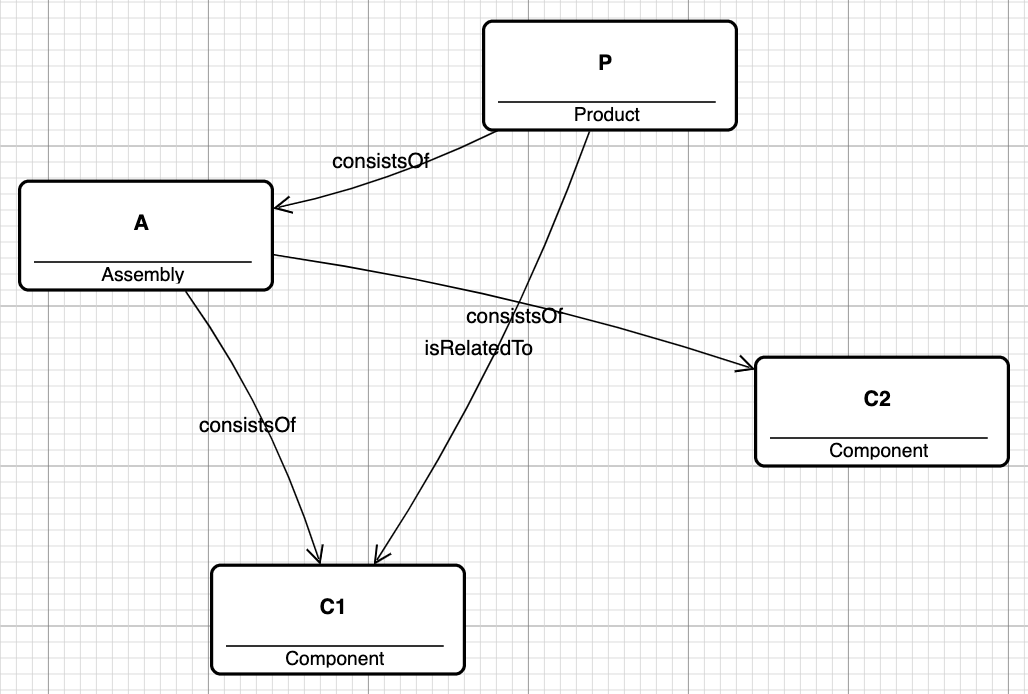
In addition to routing, you can define something called “*Curve*”, with allowed values:

* None
* Bezier
* Jump Over
* Jump Gap

Below you can see how a diagram changes appearance with different combination of the above, and we start with the defaults:



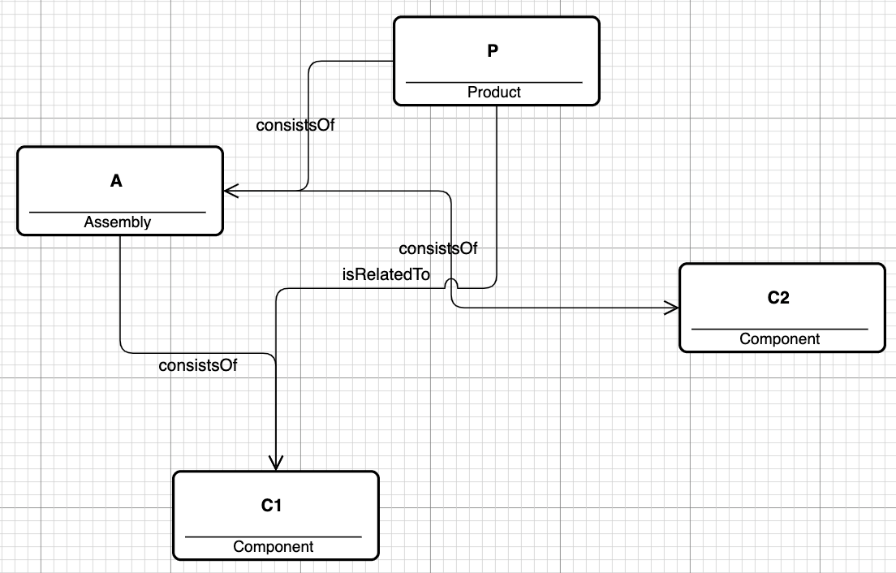
Then with “*Curve*” set to “*Bezier*”:



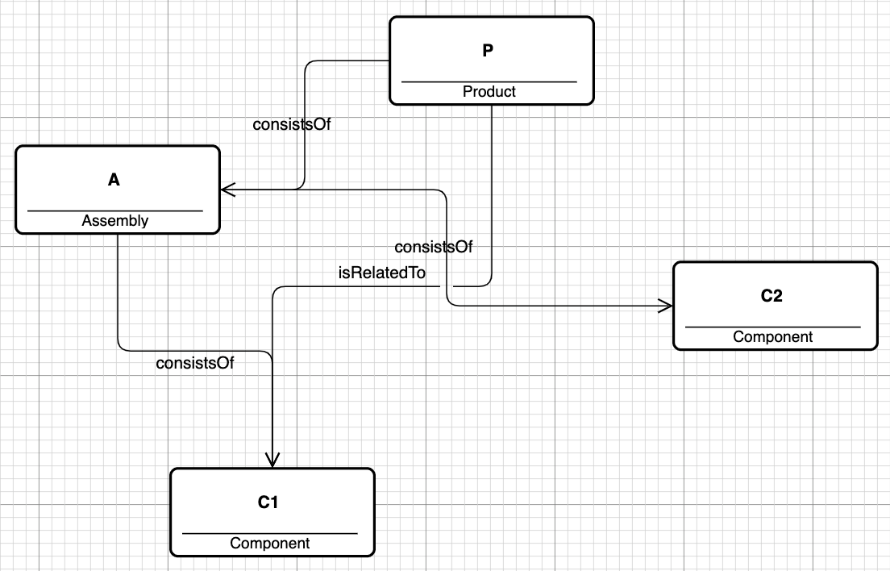
The next example has the following parameters:

* “*Routing*” has been set to “*Orthogonal*”
* “*Curve*” has been set to “*Jump Over*”

Note the two relationships that cross each other, how they are drawn in the diagram.



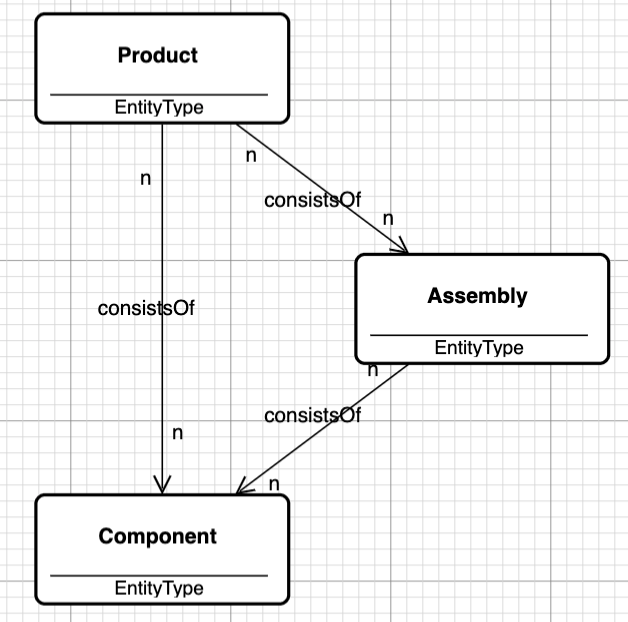
The same as above, except that “Curve” has been set “Jump Gap”. Note the difference.



The last thing to mention regarding modelviews is the ability to turn the visibility of cardinality on and off. You do this from the background menu:

* Toggle Cardinality On/Off

Below is an example of a product metamodel that you will meet later in this document. The “Toggle Cardinality” has been set to “On”:



**The Metamodelling Process**

In this chapter the focus is on the technicalities associated with developing models of metamodels, generating metamodels, creating models based on the generated metamodel and then start over again by making changes in the metamodel design.

**1. Build the metamodel**

We go back to the situation when AKMM is started with the built-in metamodel.

Et bilde som inneholder bord

Automatisk generert beskrivelse

The left pane above contains the object types in the initial metamodel, which user can be used to build a model.

A model can have different purposes, for example, a model can be used to generate a new metamodel. This is often called a concept model.

**Model the metamodel**

When building a concept model the primary type used to represent an object type, is “*EntityType*”. You drag “*EntityType*” into the modelling area and drop it to create an object representing your new type. You set the object name to the typename you decide. You do this for as many types you want.

Relationship types are modelled as relationships of type “*isRelatedTo*” between “*EntityType*s” in the model and renamed to the typename you decide.

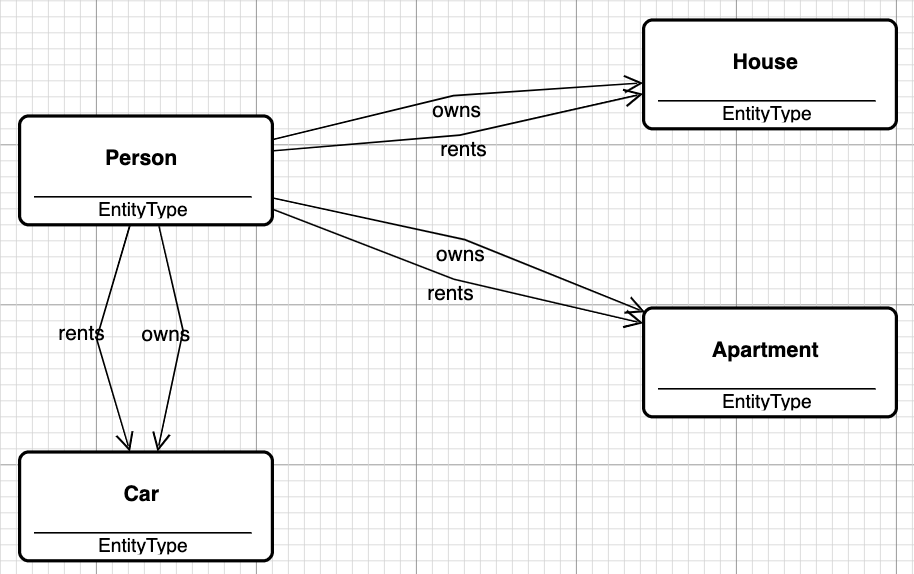
**The example**

In the following we use the same example as described in the “*Build a model*” chapter.

With default typeviews the metamodel looks as shown below.

There are four “*EntityType*” objects, each representing an object type in the intended new metamodel. Their names are the planned object type names.

There are six “*isRelatedTo*” relationships, each representing a relationship type in the new metamodel. Each “*isRelatedTo*” relationship is renamed to its planned relationship type names. These are the names you see in the diagram.



This small model is enough to generate a new metamodel, that will allow you to model people, houses, apartments and cars, and link them together with the appropriate relationships.

**Generate the metamodel**

To generate a metamodel from the model, there are a few things you need to do:

1. Create a new metamodel by right clicking the background and choose “*New Metamodel*” in the popup menu. You will be asked for a metamodel name.
2. Generate the metamodel content by right clicking the background and choose “*Generate Metamodel*”.

You will then be asked two questions:

* “*Do you want to include system types?*” At the time being answer “No” or “Cancel” to that question.
* “*Select Target Metamodel*”. In the dropdown list select the metamodel you created above.

You should then get the message:

“*Target metamodel has been successfully generated!*”.

Now it is time to test your metamodel, to see if you are able to build the desired model based on the type definitions you just created.

**Create a test model**

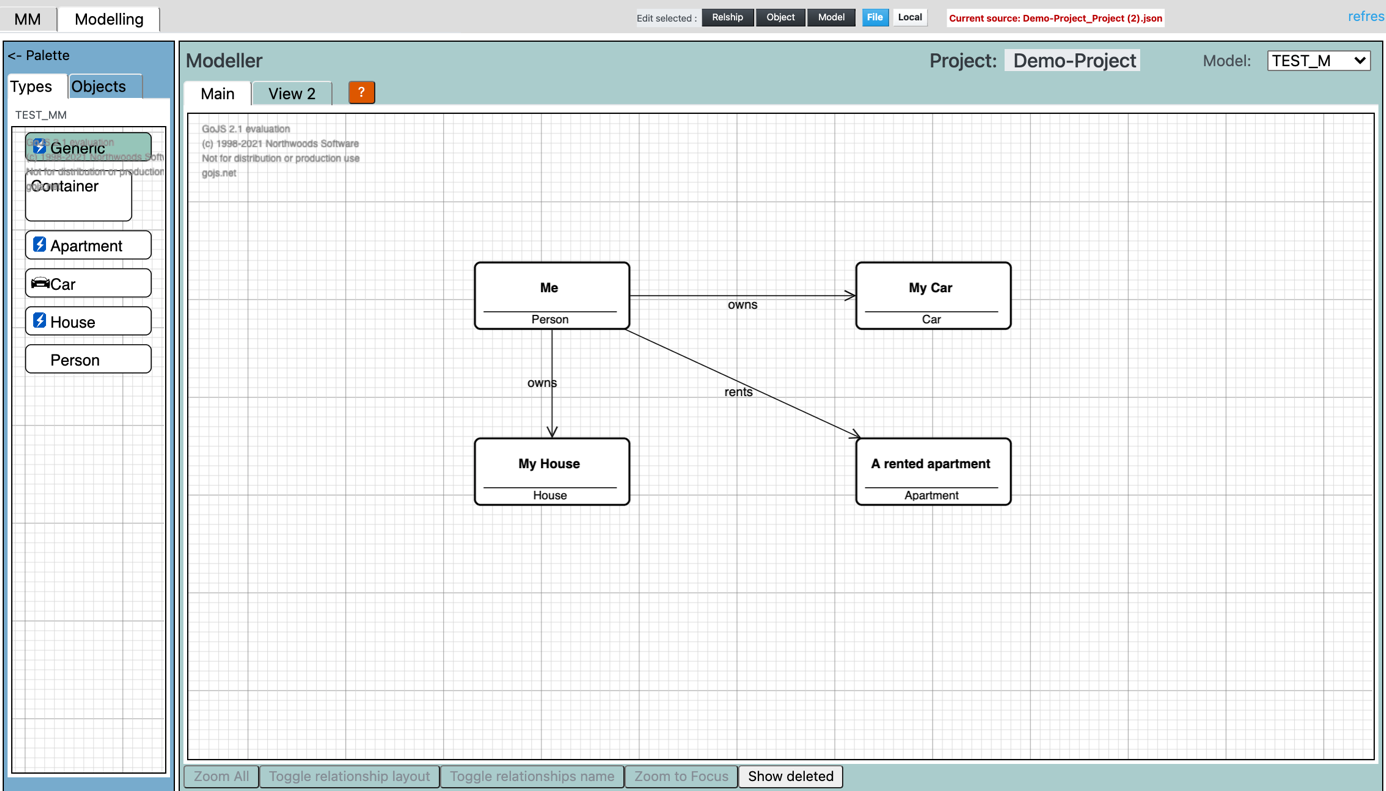
To do the test create a new model based on your new metamodel and start modelling persons, houses, apartments and cars.

If you can create objects of your 4 types, and connect relationships according to your type definitions, you have succeeded.

You create the new model by right clicking the background and choose “*New Model*” in the popup menu. You will be asked to select a metamodel – select the one you just created.

Then you will be asked for a model name and a modelview name. When the model has been created, switch to the model by selecting the model you just created in the pulldown dialog “*Model*:” in the top right corner of the AKMM window.

The object types in the metamodel you created should now appear as object types in the left pane, and you should be ready to test that the metamodel you created works as expected. Create objects and relationships and do the verification as shown below.



**Modifying the metamodel**

If something is wrong, go back to the model of the metamodel, do the necessary changes, and generate the metamodel again.

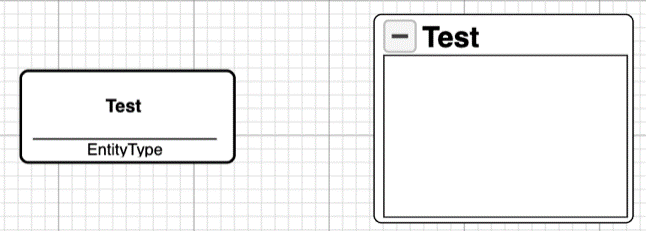
If you want to add symbols and colors to your types, go back to your metamodel to add view specifications, and generate the metamodel again.

The view specifications are done by editing “*Object View*” of the “*EntityType*” objects and by editing “*Relationship View*” of the “*isRelatedTo*” relationships.

The “*Object View*” and “*Relationship View*” definitions are used to define the corresponding “*Object Typeview*” and “*Relationship Typeview*” definitions in the generated metamodel.

There is one variant of object views that has not been mentioned yet. There is an attribute “*viewkind*” that has two values: “*Object*” and “*Container*”. In all examples so far “*viewkind*” has the value “*Object”*. When setting the value to “*Container*” the objectview will start behaving like a container. This is illustrated below:

Et bilde som inneholder shōji

Automatisk generert beskrivelse 

To the left we see two object views of the same object, both with “*viewkind*” having the value “*Object*”. To the right we see the same object views, but in the one to the right “*viewkind*” has the value “*Container*”.

**BUT**: If you want a generated type to behave like a container, you need to give “viewkind” of the **Object** the value “Container”. In this case the value in the “*Objectview”* is ignored when the metamodel is generated.

**Detailing the metamodel**

The first thing you do when you want to add more details to your metamodel than just defining object and relationship types is to add properties to your object types. This is done in the model of your metamodel.

For example, in our test metamodel defined above, we could add properties to “*Person”*, like:

* Phone number
* Email address
* Birth date
* etc

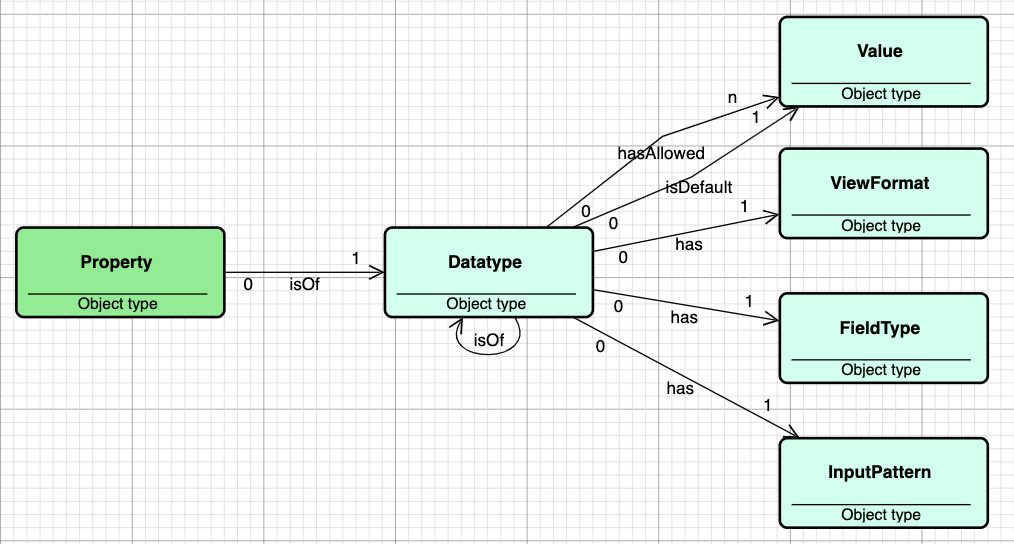
And to the “*House”*:

* Address
* Size
* Year built
* etc

If you do that, and then regenerate the metamodel, those properties will appear as attributes in the “*Edit Object*” dialog of the *Person* and *House* objects.

If nothing further is said about the properties, they will all be “*string*” properties, with no restrictions on the values.

By adding “*Datatype*” to a property, you start adding constraints to the allowed values. The potential constraints are covered by the following part of the AKMM initial metamodel:



The built-in datatypes are:

* string
* integer
* number
* boolean
* date

All custom datatypes will be based on one of these. This means that you as a metamodeller can define your own datatypes that combine the built-in types with additional specifications.

E.g., to define an *enumeration* list, you may define:

* one or more *hasAllowed* relationships to a *Value*
* one *isDefault* relationship to one of the *Values*

Combine this with a *FieldType* “*radio*” or “*select*” and this field will in the property dialog appear as a dropdown list, with the default value being the initial value.

The *FieldTypes* currently supported by the Modeller are the following:

|  |  |
| --- | --- |
| * checkbox * color * date * email * file * image * month * number * password | * radio * range * select * text * textarea * time * url * week |

*InputPatterns* are used to check if values entered by the user are valid / allowed values.

The mechanism to check this is “Regular Expressions”. A regular expression engine has been embedded in AKMM for this purpose.

Ref: *<https://www.w3schools.com/jsref/jsref_obj_regexp.asp>*

and <https://www.regular-expressions.info/examples.html>

*View Formats* are included as a mechanism to format how attribute values are shown in dialogs. The formatting mechanism used is the printf C functions family, implemented in Javascript.

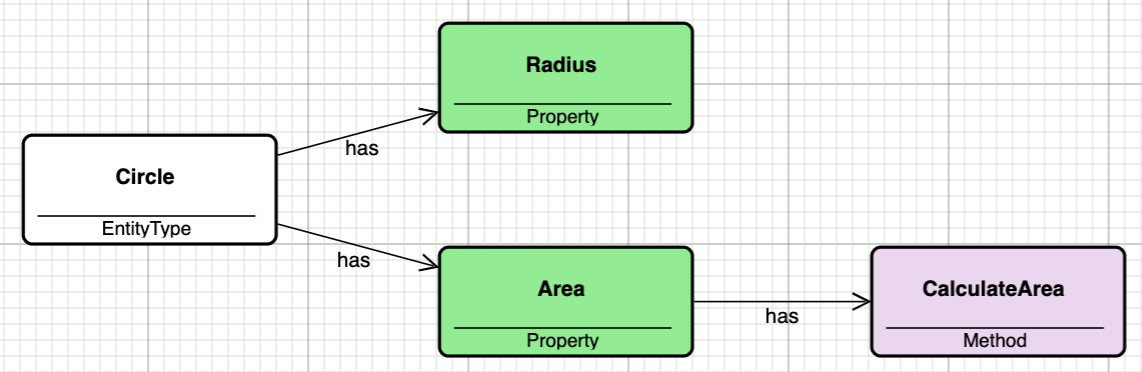
Ref: <https://alvinalexander.com/programming/printf-format-cheat-sheet/>

**Property Methods**

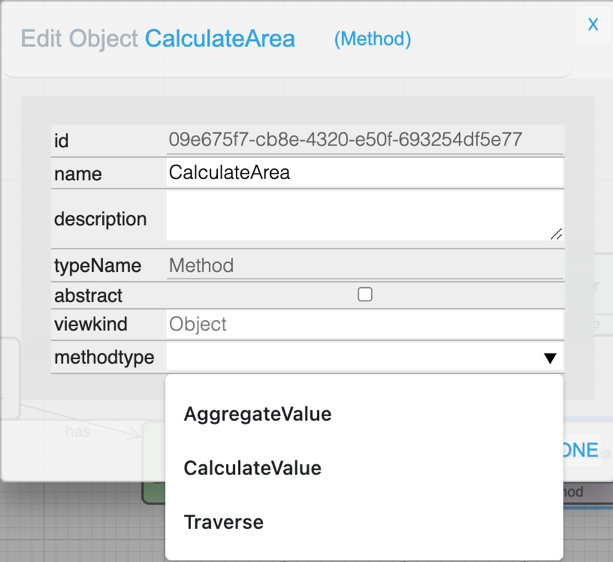
As mentioned earlier AKMM supports property methods. A *Property* may refer to a *Method* that is used to calculate its value.

As an example, assume an object type “*Circle*” with the two properties “*Radius*” and “*Area*”.

Then you can attach a method “*CalculateArea*” to the property “*Area*”:



The method “*CalculateArea*” is defined as shown in the dialogs below:

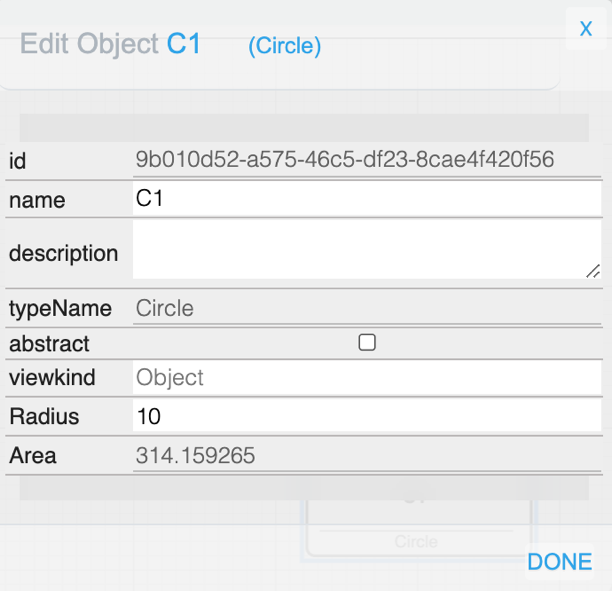
 Et bilde som inneholder tekst

Automatisk generert beskrivelse

To the left, be aware the “*methodtype*” dropdown list. The first two values are types of property methods, and in this case, we choose “*CalculateValue*”.

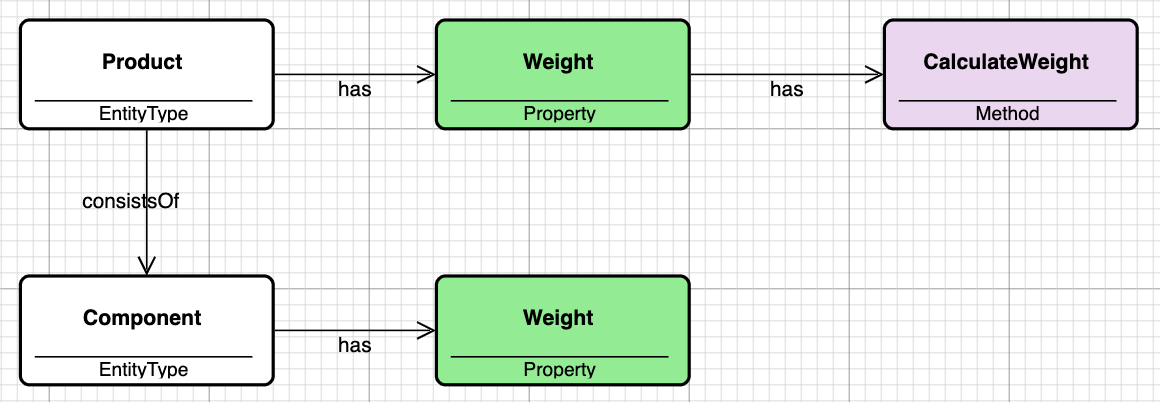
To the right, we see that, when “*CalculateValue*” has been chosen, the field “*expression*” appears, and we can type in the expression to calculate the circle area.

If we now generate the metamodel that contains the “*Circle*” as defined above, we can create an object “*C1*” of type “*Circle*” and edit the object:



We see that “*Area*” is calculated according to the expression in the “*CalculateArea*” method.

As shown above there is also a method type named “AggregateValue”. We will use the example metamodel below to illustrate how this method is used.



A product consists of components, and the weight of the product is equal to the sum of the weights of its components.

The weight of each component is entered manually, while the weight of the product is calculated as a result of the method “CalculateWeight”. The dialog used to define the method is shown below:

Et bilde som inneholder bord

Automatisk generert beskrivelse

The method type used is “*AggregateValue*” that has the arguments:

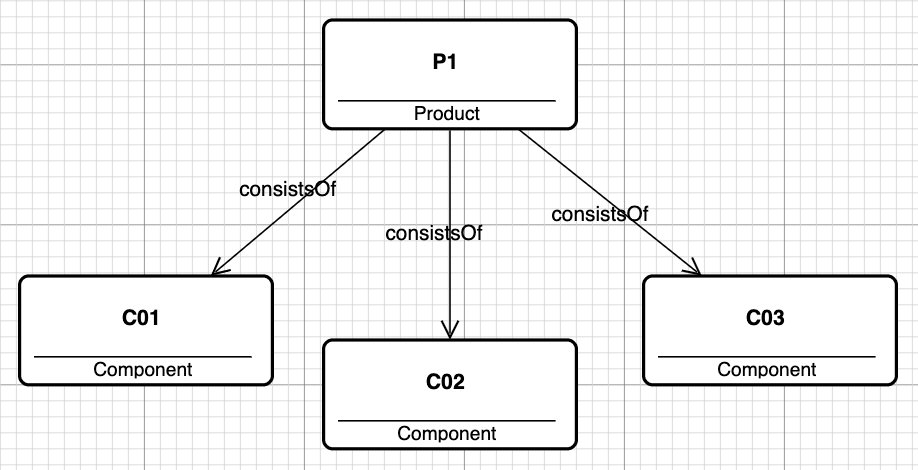
* reltype (may be undefined, meaning: do not check relationship type)
* reldir (in or out)
* objtype (may be undefined, meaning: do not check object type)
* expression (contains property name or expression)

The arguments used in our example are:

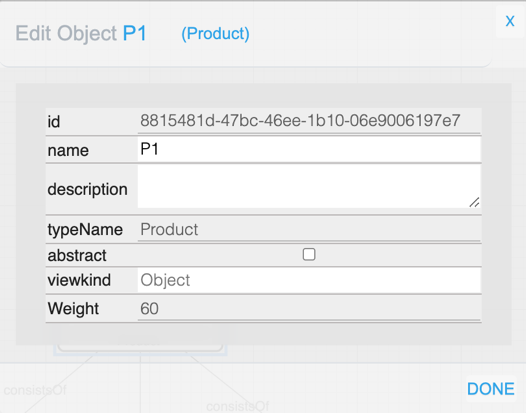
Et bilde som inneholder bord

Automatisk generert beskrivelse

We now generate our Product metamodel and creates a model based on this metamodel. Then we build the following model:



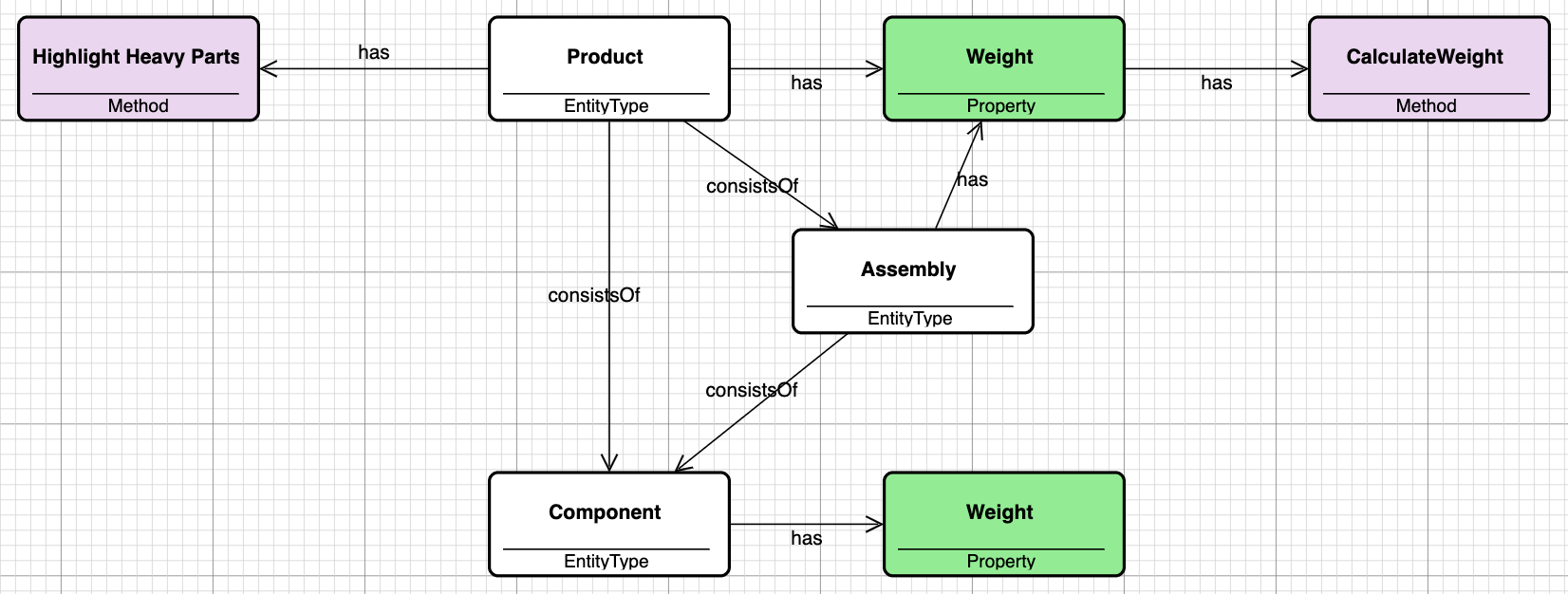
The three components have the weights 10, 20, 30. Then when we open the Edit Object dialog of P1, we see that the calculated weight is 60. Note that we cannot change the value of “*Weight*”, as it is a calculated value, accordingly the field cannot be edited.



**Type Methods**

AKMM also support type methods, i.e. methods that can be activated on an instance of a type.

To demonstrate this, we extend the product model above with a couple of new constructs:



We have added the object type “*Assembly*” and the method “*Highlight Heavy Parts*”.

The method definition is shown in the following dialog:

Et bilde som inneholder bord

Automatisk generert beskrivelse

This is a “Traverse” method, meaning that it will follow relationships of “*reldir*” and “*reltype*” if they are defined. If not, the method will follow any relationship.

In our case the direction has been specified, but not the relationship type to follow.

When the traversal comes to an object, it will check the type if specified (“*typecondition*”), otherwise it does not care what type it is.

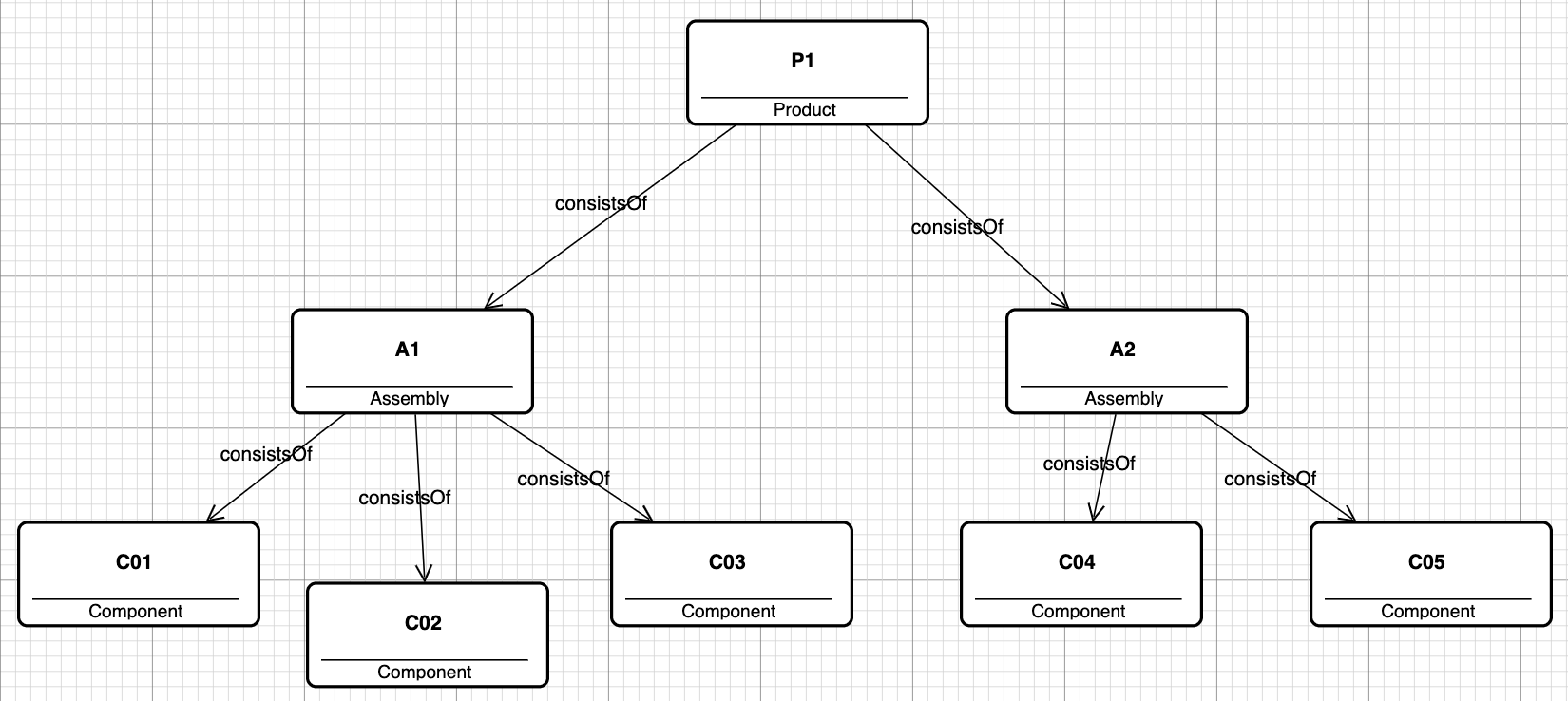
When on an object and “*valuecondition*” is specified, it will check if the condition is fulfilled or not. If fulfilled, the “*preaction*” will be applied when the traversal is moving down the structure. “*postaction*” will be applied when moving up the structure.

As of now, there are only two actions possible. These are:

* Highlight
* Select

Other actions will be added in future releases.

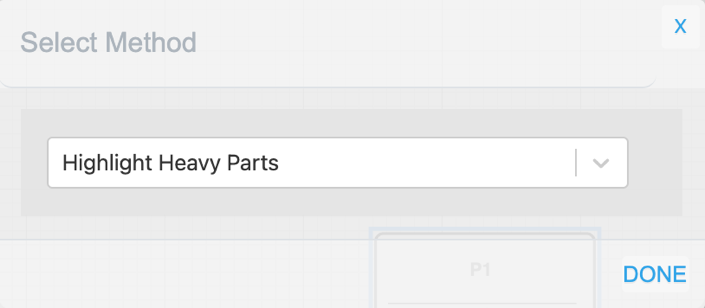
Based on the extended product metamodel the following model has been built:



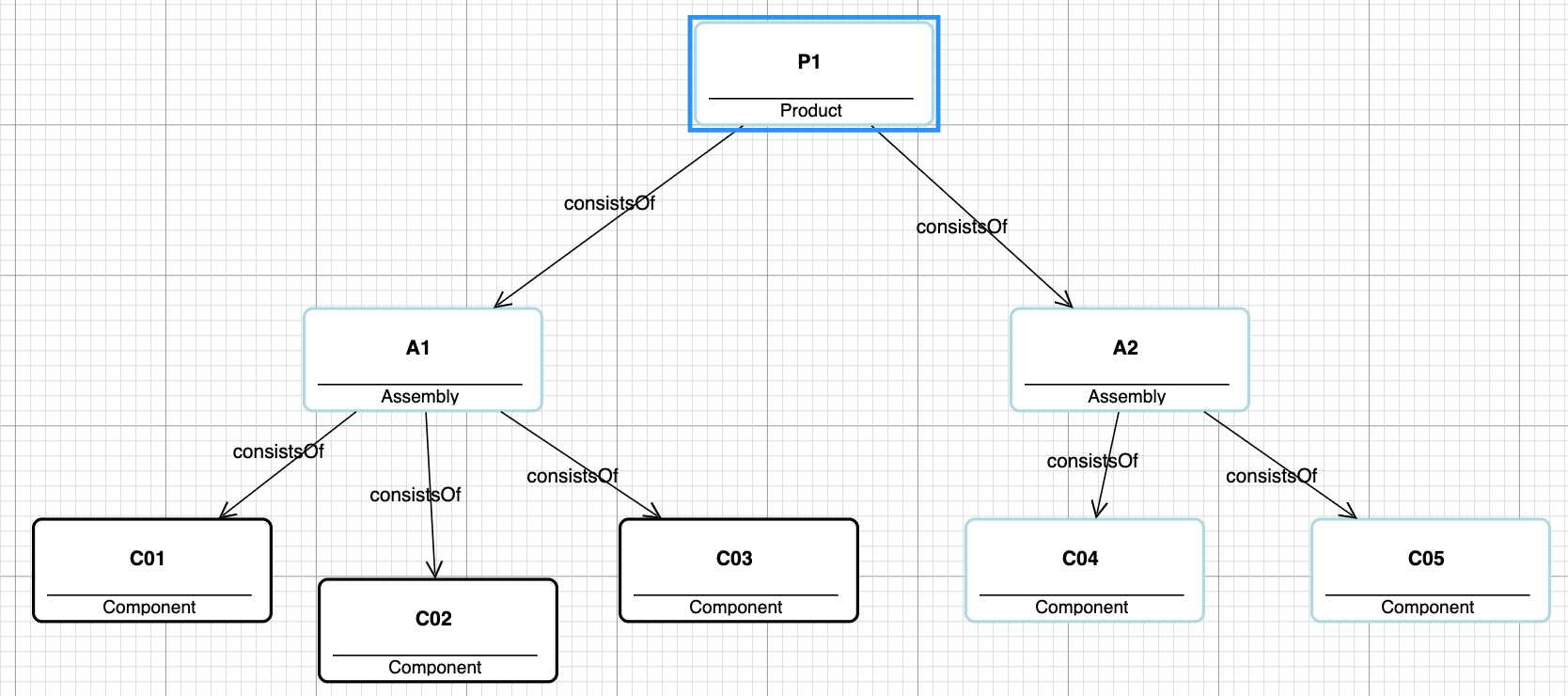
We have added two components, C04 and C05 with weights 40 and 50. And we have added two assemblies, A1 and A2, which now are the two parts the product P1 consists of.

This means that A1 now should have the weight 60, and A2 the weight 90.

If we now rightclick the product P1, we have at the bottom of the pulldown menu a choice “*Execute Method*”, that we choose. We then get a dialog in which we choose the method to execute:



We choose “Highlight Heavy Parts” and clicks “Done”, with a result as shown below. We see that the two assemblies and the two heaviest components are highlighted, as well as the product P1. This is according to the expected result.



Any contents missing and questions can be directed to Dag Rojahn Karlsen, the manager of AKM Modeler team.

KAVCA AS

Norway 25th October 2021