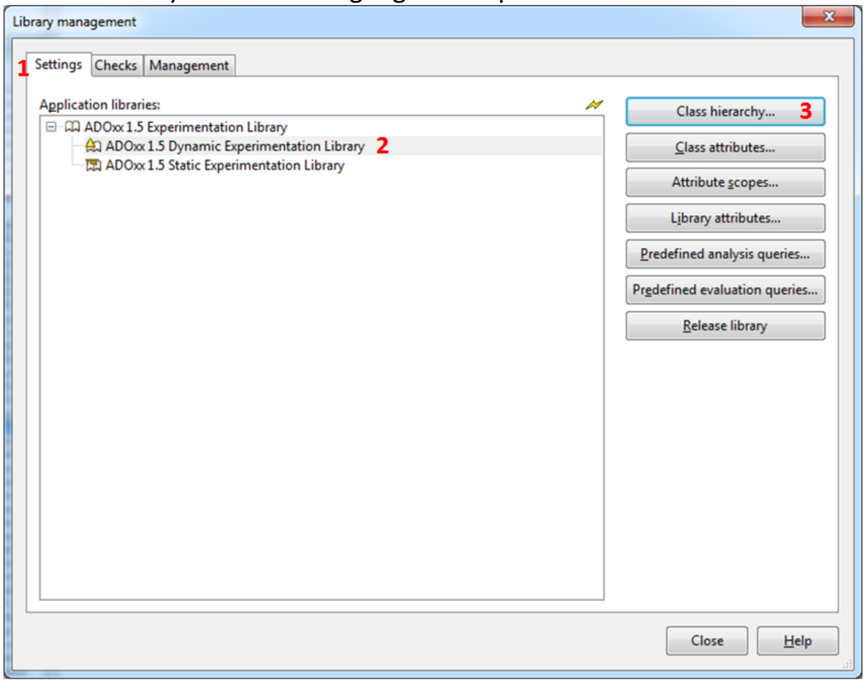
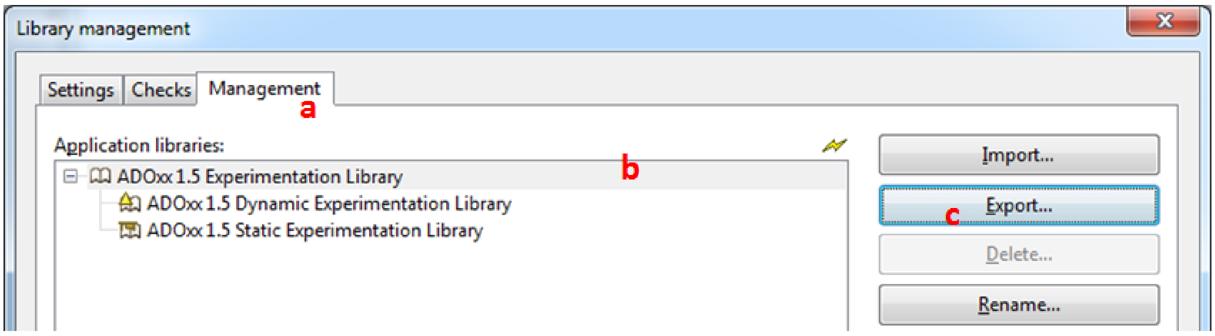
ADOxx Exercises: Notations, Syntax, Semantics[1]

In this exercise we will define a modelling method for describing cooking recipes in the form of diagrams. For the following exercises, start the **ADOxx Development Toolkit (**default credentials are user: "Admin", password:"password"**)** and open the Library Management window.

1. Go to the Settings tab. This will list all the modelling methods created in your ADOxx installation - each of them is called a "library" and each of them will generate its own modelling tool, for its own modelling language. However, in a fresh installation you will find only one – the **Experimentation Library**. This is an empty modelling method skeleton that comes included with ADOxx. You will work by extending this "empty modelling language skeleton" with your own concepts and symbols. So if you want to also keep this empty version (for future projects, or to be able to reset your work back to the empty skeleton), you should export it:
    a. Go to the Management Tab
    b. Select the Experimentation Library
    c. Export it to an external ABL file. Keep this file for the future, in case you will need to start from scratch with a new modelling method (in that case you will have to import it back and change its name during the import).
2. In the following we will work on this Experimentation Library. Select the Dynamic part of the library.
3. Select Class hierarchy to see the language concepts

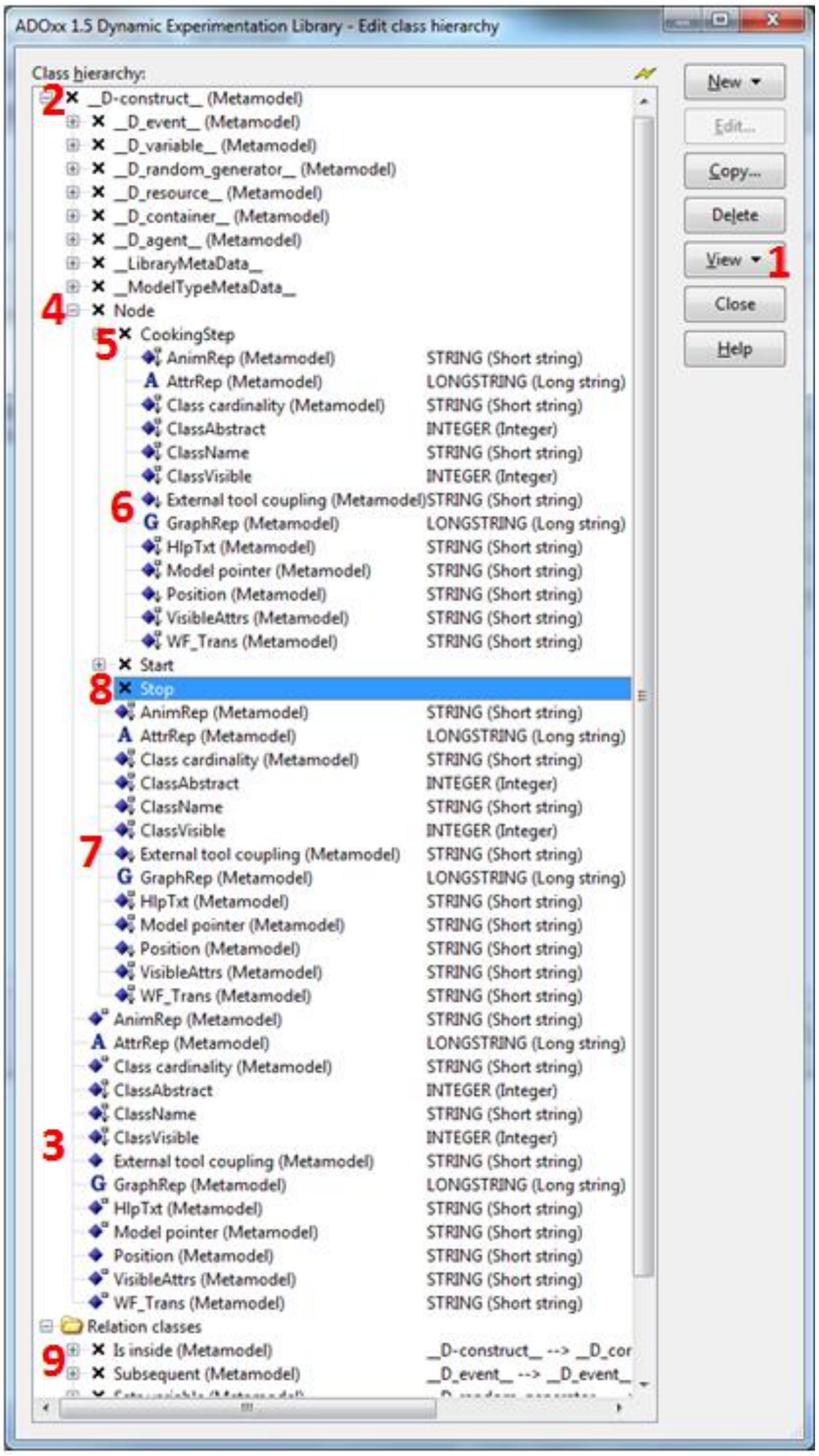

Library management
1
Settings
Checks
Management
Application libraries:
ADOxx 1.5 Experimentation Library
ADOxx 1.5 Dynamic Experimentation Library
2
ADOxx 1.5 Static Experimentation Library
Class hierarchy...
3
Class attributes...
Attribute scopes...
Library attributes...
Predefined analysis queries...
Predefined evaluation queries...
Release library
Close
Help

[1] Tested on ADOxx v. 1.5


Library management
Settings
Checks
Management
a
Application libraries:
b
ADOxx 1.5 Experimentation Library
ADOxx 1.5 Dynamic Experimentation Library
ADOxx 1.5 Static Experimentation Library
Import...
Export...
c
Delete...
Rename...

In the Class hierarchy window follow the numbering of the explanations below:

1. Select View – Metamodel, then View – Class hierarchy to see the full concept hierarchy of ADOxx, including its predefined concepts (the ones starting with _D_). You may also work without seeing the full hierarchy, if you don't care about inheritance (for example if you create a "flat" language where the concepts don't have to inherit from each other) or if you want to see strictly the concepts of your language and not the ones predefined in ADOxx. That's why the full hierarchy is not visible by default;
2. If you made the hierarchy visible, you may notice that **the root of all concepts is _D_construct**, which provides the predefined skeleton for all concepts. Any concept that you define for your own language will inherit some built-in properties / behaviors from _D_construct.
3. In area 3 of the screenshot below you can see what this skeleton is – several predefined attributes that help us to define notation, syntax and semantics for every concept;
4. Right-click on _D_construct and select New class, to define a new concept – Node. This will position the new concept in the hierarchy, right under _D_construct.
5. Right-click on Node, again New class and define a new concept – CookingStep
6. In area 6 you can see the attributes of CookingStep, inherited from Node
7. In area 7 you can see the attribures of Node, inherited from _D_construct (whose attributes are visible in area 3)
8. Right-click on Node and create 2 additional subconcepts: Start and Stop. These will also inherit the same attributes. The goal of these concepts is to create a modelling language that allows us to describe cooking recipes in a diagrammatic form. A Cooking Recipe diagram will be a sequence of Cooking Steps. In addition, the Start concept will indicate where the recipe starts and the Stop concept will indicate where the recipe ends. All these 3 are specializations of Node.
9. Besides Nodes, a cooking diagram will also have connectors (arrows) to indicate the order of the Nodes. These arrows will be defined as relations in area 9 of the screenshot (later). For now, let’s focus on the Nodes.


ADOxx 1.5 Dynamic Experimentation Library - Edit class hierarchy
Class hierarchy:
New
Edit...
Copy...
Delete
View
Close
Help

```
2 _D-construct_ (Metamodel)
    _D_event_ (Metamodel)
    _D_variable_ (Metamodel)
    _D_random_generator_ (Metamodel)
    _D_resource_ (Metamodel)
    _D_container_ (Metamodel)
    _D_agent_ (Metamodel)
    _LibraryMetaData_
    _ModelTypeMetaData_
4   Node
5     CookingStep
        AnimRep (Metamodel)                      STRING (Short string)
        AttrRep (Metamodel)                      LONGSTRING (Long string)
        Class cardinality (Metamodel)            STRING (Short string)
        ClassAbstract                            INTEGER (Integer)
        ClassName                                STRING (Short string)
        ClassVisible                             INTEGER (Integer)
6       External tool coupling (Metamodel)       STRING (Short string)
        GraphRep (Metamodel)                     LONGSTRING (Long string)
        HlpTxt (Metamodel)                       STRING (Short string)
        Model pointer (Metamodel)                STRING (Short string)
        Position (Metamodel)                     STRING (Short string)
        VisibleAttrs (Metamodel)                 STRING (Short string)
        WF_Trans (Metamodel)                     STRING (Short string)
      Start
8     Stop
      AnimRep (Metamodel)                        STRING (Short string)
      AttrRep (Metamodel)                        LONGSTRING (Long string)
      Class cardinality (Metamodel)              STRING (Short string)
      ClassAbstract                              INTEGER (Integer)
      ClassName                                  STRING (Short string)
      ClassVisible                               INTEGER (Integer)
7     External tool coupling (Metamodel)         STRING (Short string)
      GraphRep (Metamodel)                       LONGSTRING (Long string)
      HlpTxt (Metamodel)                         STRING (Short string)
      Model pointer (Metamodel)                  STRING (Short string)
      Position (Metamodel)                       STRING (Short string)
      VisibleAttrs (Metamodel)                   STRING (Short string)
      WF_Trans (Metamodel)                       STRING (Short string)
    AnimRep (Metamodel)                          STRING (Short string)
    AttrRep (Metamodel)                          LONGSTRING (Long string)
    Class cardinality (Metamodel)                STRING (Short string)
    ClassAbstract                                INTEGER (Integer)
    ClassName                                    STRING (Short string)
3   ClassVisible                                 INTEGER (Integer)
    External tool coupling (Metamodel)           STRING (Short string)
    GraphRep (Metamodel)                         LONGSTRING (Long string)
    HlpTxt (Metamodel)                           STRING (Short string)
    Model pointer (Metamodel)                    STRING (Short string)
    Position (Metamodel)                         STRING (Short string)
    VisibleAttrs (Metamodel)                     STRING (Short string)
    WF_Trans (Metamodel)                         STRING (Short string)
Relation classes
9   Is inside (Metamodel)                        _D-construct_ --> _D_cor
    Subsequent (Metamodel)                       _D_event_ --> _D_event_
```

Most concepts of a modelling language will need to have a graphical symbol defined, to distinguish them from other concepts and to communicate their meaning. This is done in the predefined GraphRep attribute.

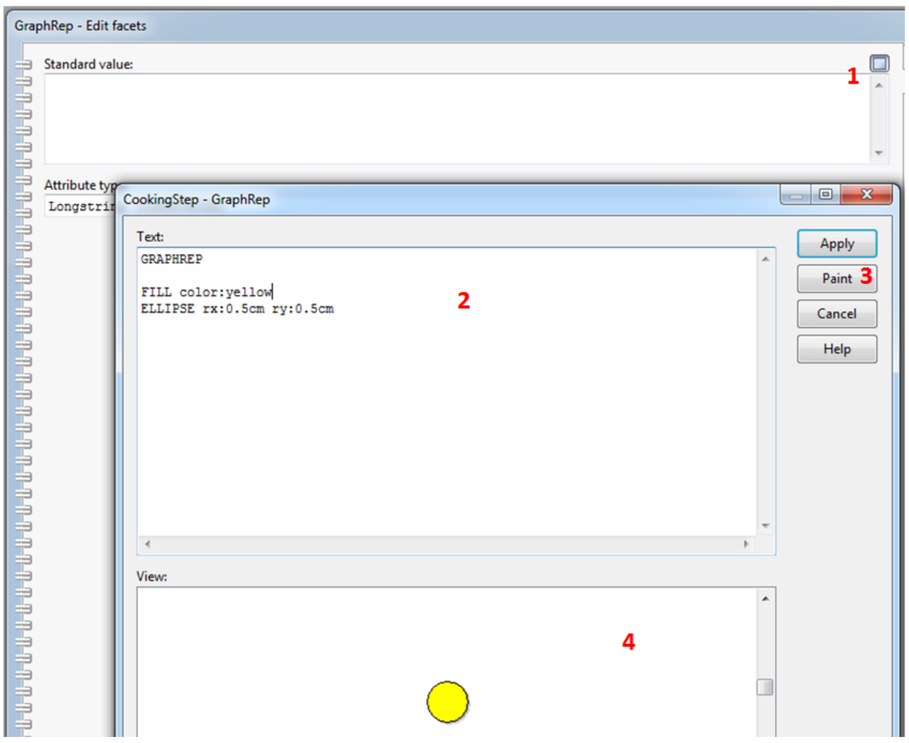
When opening the GraphRep, take care to open it for CookingStep, and not for Node or _D_construct. GraphRep is inherited, therefore it is present for every concept. You have to define it only for those concepts that will be present (instantiated) in diagrams. Node and _D_construct are present in the hierarchy just to facilitate inheritance – they will not be present in the diagrams, therefore they do not need a graphical notation!

Edit the GraphRep attribute of CookingStep to define its graphical symbol. This will display the window below:

1. Press button 1 to open the GraphRep definition window
2. Use the Text area to write GraphRep code (you may use the Help button to consult the syntax).

GRAPHREP
FILL color:yellow
ELLIPSE rx:0.5cm ry:0.5cm

   GraphRep is a vector graphics language that allows you to define shapes in a fixed system of coordinates, whose center is the point of click (where you will place the symbol on the modelling canvas). Details on the example in the figure:
   a. The keyword GRAPHREP is mandatory at the beginning
   b. The FILL command defines the fill color for the shapes that follow (therefore order is important! If you define the FILL color after the shapes it will not be applied!)
   c. The ELLIPSE command defines an ellipse with the radius indicated on the x and y axis. If they are equal, we get a circle.
3. Use the Paint button to get a visual suggestion of how the symbol will look. This will only work for static definitions (in dynamic or interactive notations, not all the details will be visible here)
4. Use area 4 to see the effect of Paint.



```
GRAPHREP

FILL color:yellow
ELLIPSE rx:0.5cm ry:0.5cm
```

Close the GraphRep window with Apply and go back to the class hierarchy. Now create a GraphRep for the Start concept, with the following code:

GRAPHREP
TEXT "Start"

This notation is quite minimal, since it does not even define a graphical shape. The symbol for Start will be just a static text label, with the word “Start”. It is also possible to mix graphical notation with such textual labels in order to have the meaning communicated on a visual level, as clearly as possible.

Now go back to the class hierarchy and create a similar GraphRep for the Stop concept:

```
GRAPHREP
TEXT "Stop"
```

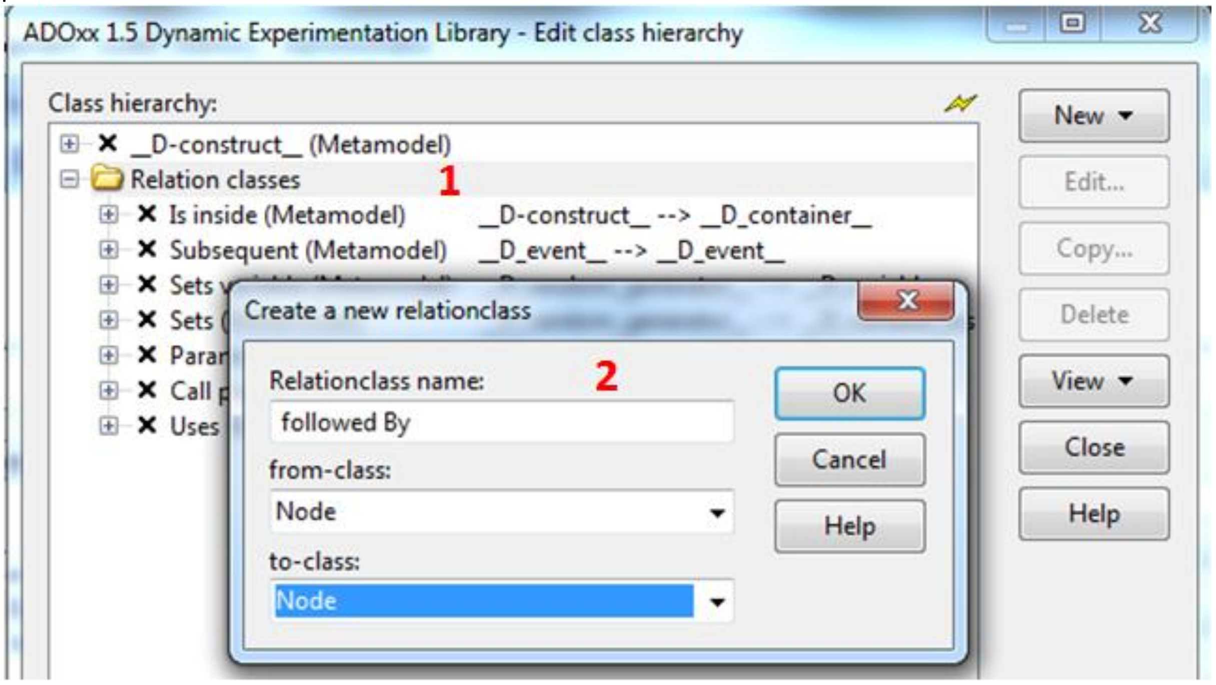
Go back to the class hierarchy and create the arrow that will connect cooking steps to indicate their order. This is called in ADOxx a "relation class":

1. Right-click on Relation classes and New relationclass
2. Indicate the relation name (followed By) and the allowed source and target (Node for both)

Now it should be clearer why we created the intermediate Node concept (although it will have no GraphRep => it will not appear directly in diagrams): a relation in ADOxx can have only one source concept and one target concept, so if we want to allow multiple concepts to be connected with the same relation we need to unify them in an abstract class (abstract class=class with no GraphRep).

With the "followed By" connector we will be able connect any Node to any Node (of any of the instantiable subclasses – Start, CookingStep, Stop). Later we will add some restrictions – for example to forbid an arrow to go into a Start element.

Notice that, unlike concepts, the relations are not arranged in a hierarchy, therefore no inheritance is possible between relations.


ADOxx 1.5 Dynamic Experimentation Library - Edit class hierarchy

Class hierarchy:
- _D-construct_ (Metamodel)
- Relation classes 1
  - Is inside (Metamodel) _D-construct_ --> _D_container_
  - Subsequent (Metamodel) _D_event_ --> _D_event_
  - Sets
  - Sets
  - Para
  - Call
  - Uses

Create a new relationclass
Relationclass name: 2
followed By
from-class:
Node
to-class:
Node

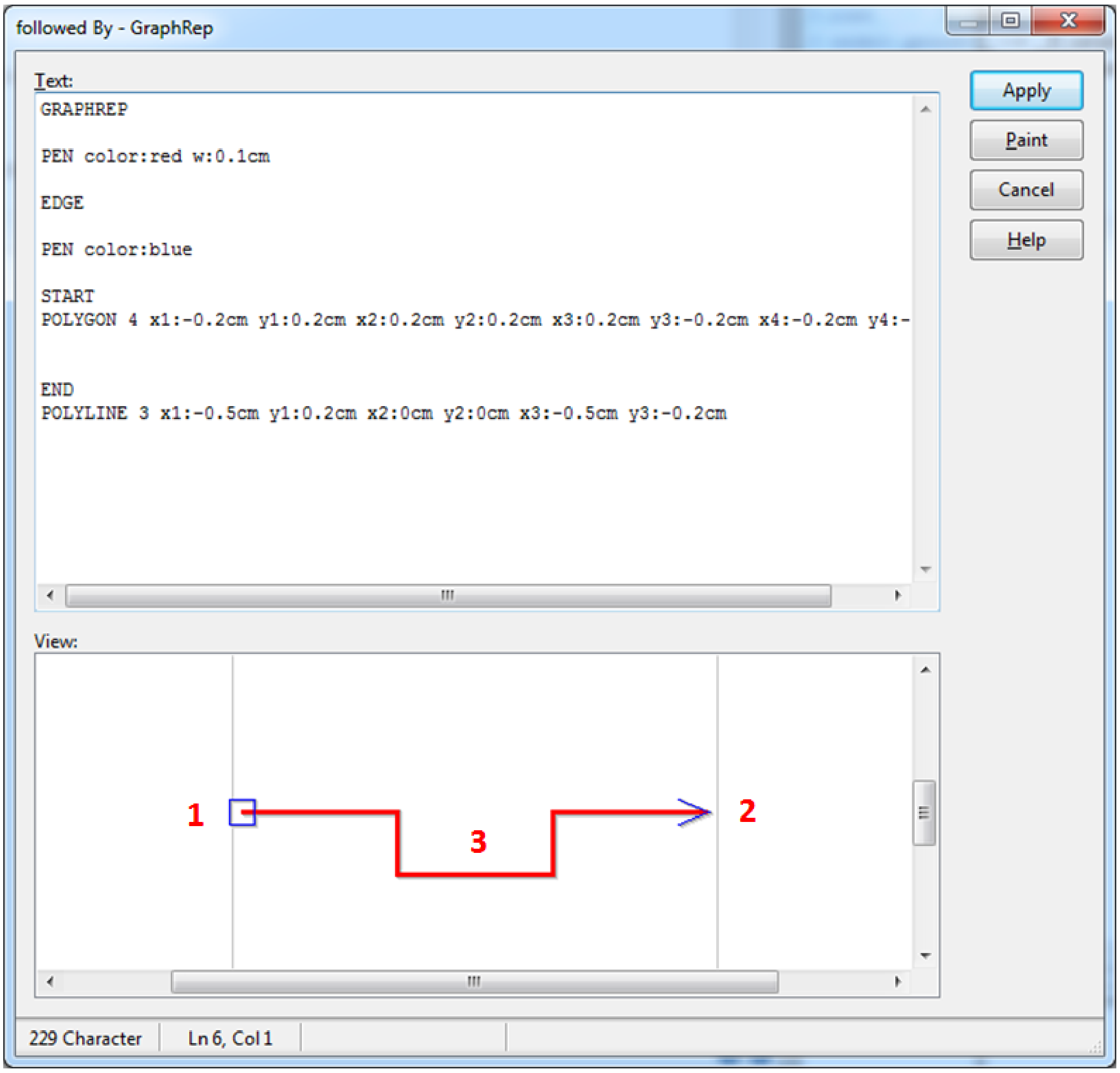
Notice that relations also have their own GraphRep, which is defined in the same syntax as before. Just like before, PEN allows us to define the line (color, width, style etc.) and FILL allows us to define a filling color. Multiple PEN/FILL definitions may change the settings for different parts of the graphical symbol.

There is, however, a difference – for relations we need to define 3 sections of the graphical symbol:

```
GRAPHREP
```

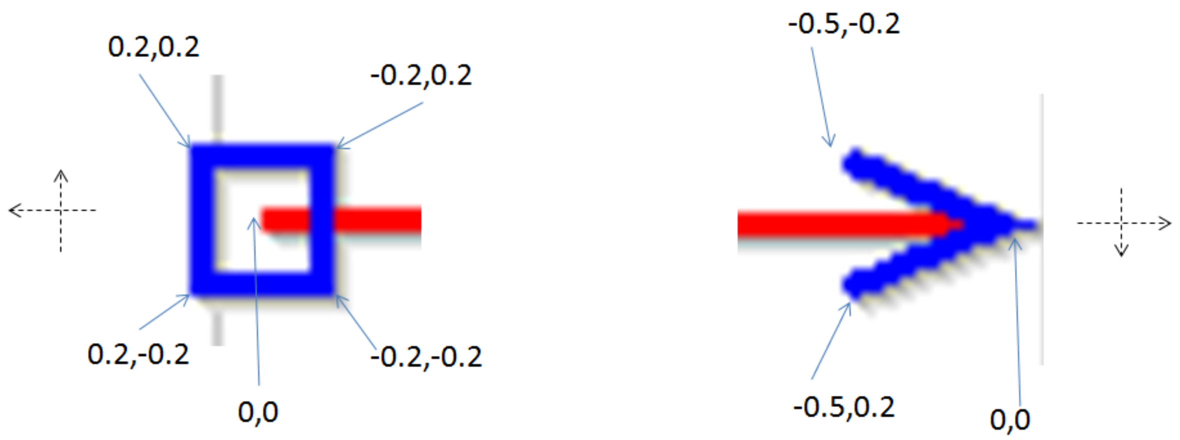
```
PEN color:red w:0.1cm
EDGE

PEN color:blue
START
POLYGON 4 x1:-0.2cm y1:0.2cm x2:0.2cm y2:0.2cm x3:0.2cm y3:-0.2cm x4:-0.2cm y4:-0.2cm

END
POLYLINE 3 x1:-0.5cm y1:0.2cm x2:0cm y2:0cm x3:-0.5cm y3:-0.2cm
```

1. The line itself is defined with EDGE. In our case, the first PEN command applies to it, which states the color red and the width of the line. It is also possible to include here a graphical shape or a text, which will be drawn in the middle of the line.
2. The beginning of the connector is defined with START. In our case, it is a square polygon with 4 points. The coordinates of the points are measured relative to where the line starts. The last PEN command applies here, which states the color blue.
3. The end is defined with END. In our case, this is a polyline (which in contrast to the polygon is not closed) with 3 points, to create the appearance of an arrow head. The coordinate center is where the line ends. The last PEN command applies again, therefore the color is blue.



```
GRAPHREP

PEN color:red w:0.1cm

EDGE

PEN color:blue

START
POLYGON 4 x1:-0.2cm y1:0.2cm x2:0.2cm y2:0.2cm x3:0.2cm y3:-0.2cm x4:-0.2cm y4:-


END
POLYLINE 3 x1:-0.5cm y1:0.2cm x2:0cm y2:0cm x3:-0.5cm y3:-0.2cm
```

Warning: you have to be careful about how to measure coordinates, because it is a bit counter-intuitive. The coordinate system has a different orientation for the START and for the END, as shown in the picture below:



The general coordinate system (also applied for EDGE and for concept GraphReps) is the one on the right, with x growing towards right and y growing downwards. However, for the START of the arrow the coordinate system is inverted!

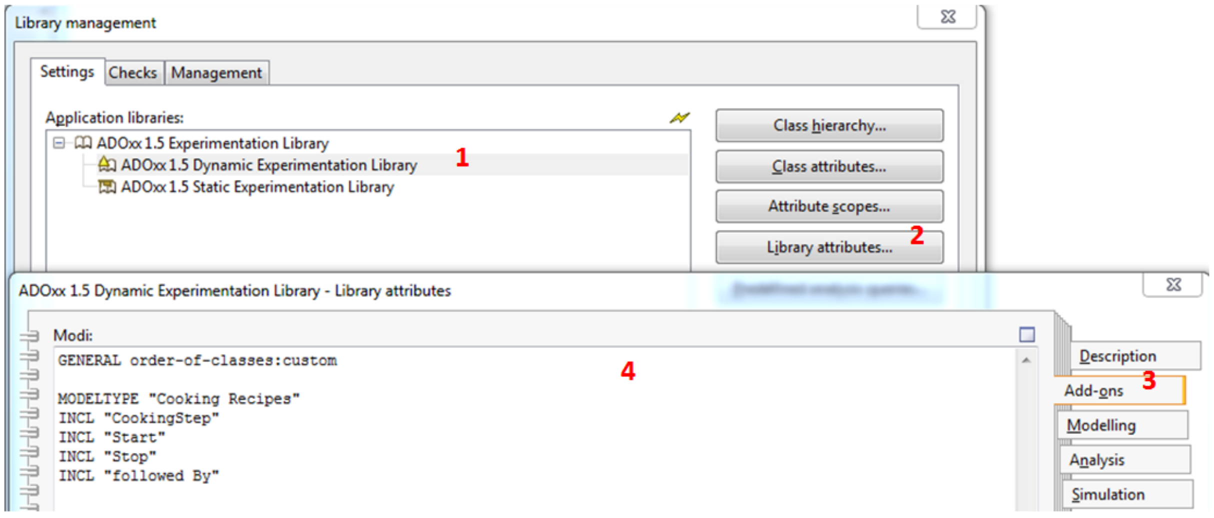
After the concepts and relations are defined, we may close the Class hierarchy window. A message will warn us that changes have been made to the "library". Make sure you answer positively, in order to save the changes.

Now we need to group our concepts in model types. Follow the steps in the picture below:

1. Select the library part where you worked until now
2. Select Library attributes
3. Select the Add-ons group of attributes
4. Write the code in the Modi attribute

GENERAL order-of-classes:custom

MODELTYPE "Cooking Recipes"
INCL "CookingStep"
INCL "Start"
INCL "Stop"
INCL "followed By"



```
GENERAL order-of-classes:custom

MODELTYPE "Cooking Recipes"
INCL "CookingStep"
INCL "Start"
INCL "Stop"
INCL "followed By"
```

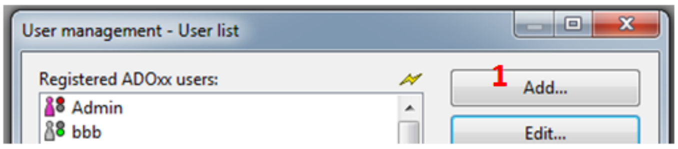
This example is minimal:

- The GENERAL command indicates that the order of classes listed on the toolbar (in the modelling tool) should be the one provided here.
- The MODELTYPE command defines the name of the model type
- The INCL command includes the concepts and the relation. Notice that the Node concept is not included. It does not have a GraphRep to be used in diagrams and it was created only as a unifying concept (to represent all types of nodes of the diagram)

Close the windows by confirming with Yes all requests for saving changes. You might get an error if you mistype the name of a concept or relation!

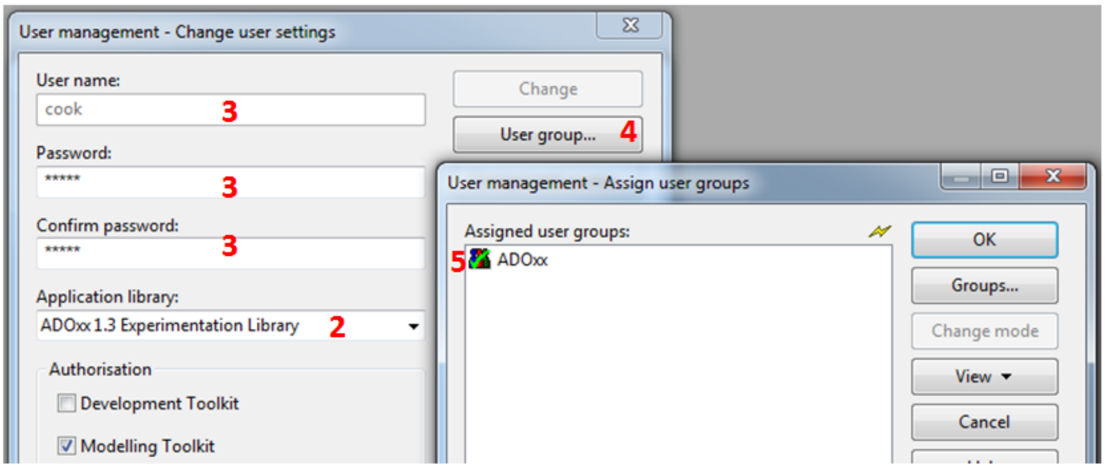
The final step before generating the modelling tool is to assign a user for the Experimentation Library. Go to the User management screen and follow the steps below:

1. Add a new user
2. Select the Experimentation Library where you worked
3. Define a user and a password
4. Go to User groups
5. Click the ADOxx group which includes all the user rights (to avoid going into too much detail)



User management - User list

Registered ADOxx users:
Admin
bbb



User management - Change user settings

User name: cook

Password: *****

Confirm password: *****

Application library: ADOxx 1.3 Experimentation Library

Authorisation
- [ ] Development Toolkit
- [x] Modelling Toolkit

User management - Assign user groups

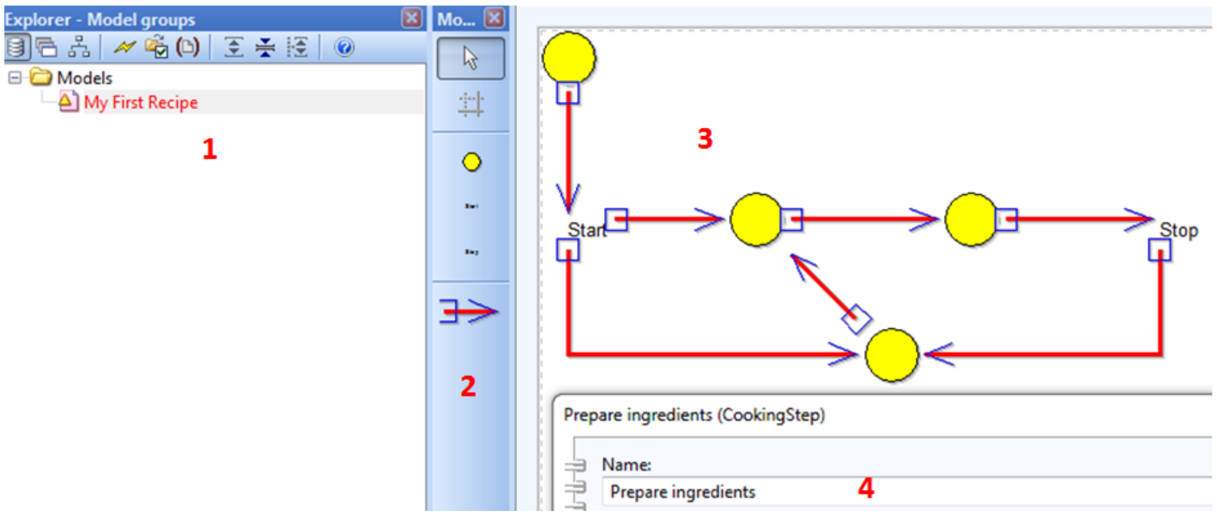
Assigned user groups:
ADOxx

Now start the modelling toolkit with the user credentials that have just been created. After logging in the modelling tool, notice the following:

1. Right click on the Models folder and you will see the Model Type that you defined (Cooking Recipes)
2. Create such a model and you will notice the concept toolbar where you can select your symbols. Remember that, in order to have a symbol visible on the concept toolbar, you need to:
   a. define a GraphRep for it;
   b. include it in the model type with the INCL command. Therefore you will not see here the predefined concepts from the concept hierarchy (such as _D_construct – these are called abstract classes = useful for inheritance of semantics, but not instantiated in diagrams)
3. Play around on the modelling canvas to see how you can create a model. Notice that you can connect anything to anything, any number of times: you can create an incoming arrow for Start, an outgoing arrow for Stop, multiple arrows going from the same Node – this means

that no syntactic constraints have been defined! The only restriction you have comes from ADOxx: you cannot draw an arrow multiple times between the same pair of nodes, in the same direction!

4. Another problem is that the modelling language **has no semantics**. Besides the words Start and Stop (which are part of the notation), no one can understand what is meant with such a diagram. Double click one of the CookingStep nodes and you will get an almost empty set of properties. The only editable property is the Name (inherited from _D_construct) and not even the name is visible on the canvas (you have to open the property sheet to see it). This means that our model does not have machine-readable semantics = a human reader can give them any interpretation he likes and there are no properties available for queries (other than the name). In other words, the model is only a visual diagram, with no useful knowledge captured in it. Moreover, the visual level (the notation) is also very weak, since you don't even see the names of the cooking steps directly in the diagram!



Prepare ingredients (CookingStep)

Name:
Prepare ingredients

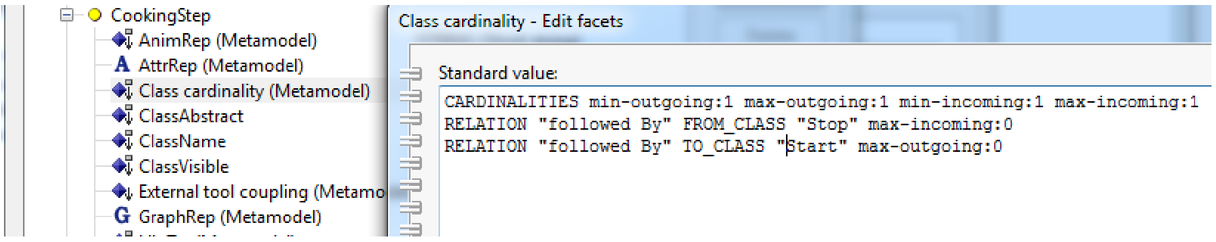
So we need to solve two problems in order to create syntax and semantics for our language:

- To include some well-formedness syntactic constraints:
    - no arrow should link a Step to the Start point, nor the Stop point to a Step;
    - the steps should be in sequence, so we should not allow two arrows to come out of the same Step or to go in the same Step;
    - a model must have exactly one Start and one Stop.
- To include some semantics (properties):
    - a cooking step should have a Cost property slot to indicate estimated costs (e.g., with the consumptions of ingredients) as an integer value between 0 and 10000 (just an arbitrary interval to demonstrate the restriction)
    - a connector should have a Condition property slot to describe under which condition the cook should progress to the next step, as a string of maximum length 20 (again, an arbitrary length limit for demonstration purposes)

Remember that if you return to the Development Toolkit to make changes, you should close the Modelling Toolkit then log in again only after the changes in the metamodel are saved.

Go back to the Development Toolkit, open the CookingStep concept and define its Class cardinality with the following code:

```
CARDINALITIES min-outgoing:1 max-outgoing:1 min-incoming:1 max-incoming:1
RELATION "followed By" FROM_CLASS "Stop" max-incoming:0
RELATION "followed By" TO_CLASS "Start" max-outgoing:0
```



```
CARDINALITIES min-outgoing:1 max-outgoing:1 min-incoming:1 max-incoming:1
RELATION "followed By" FROM_CLASS "Stop" max-incoming:0
RELATION "followed By" TO_CLASS "Start" max-outgoing:0
```

This code defines the following constraints:

- no cooking step should be left unconnected and no cooking step should be connected with more than one connector (both minimum and maximum limits are set to 1 for both incoming and outgoing arrows)
- a cooking step should have zero connectors coming from a Stop element
- a cooking step should have zero connectors going into a Start element

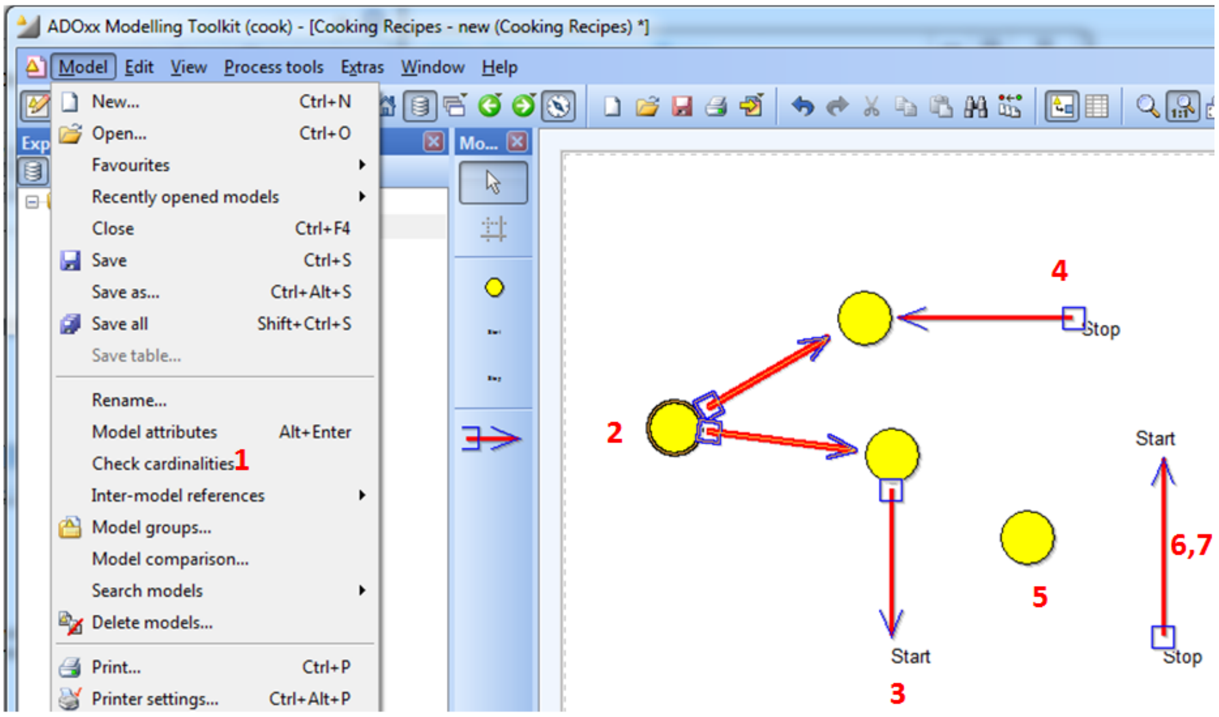
Save this and go to the Modelling Toolkit. Create the model below and following the numbering for explanations:

1. By default, the act of modelling is unconstrained, in order not to annoy the modeler with very frequent error messages, especially if it's possible to temporarily create incomplete models. In this case, the cardinality check is applied explicitly through a menu option: Model – Check Cardinalities. For the example in the image, the following errors should be detected:
2. Two arrows outgoing from the same step
3. Arrow going into the Start element
4. Arrowing coming out of the Stop element
5. Unconnected step

However, notice that some errors are still not covered:

6. It is possible to have more than one Start and more than one Stop in a model (and zero Start or Stop elements)
7. It is possible to make a connector to a Start if you don't draw it from a CookingStep (and also to make a connector from a Stop, if you don't draw it to a CookingStep).

The reason why the last errors are permitted is because they should be constrained in the Start and Stop concepts (our constraints were defined only for CookingSteps).



Go back to the Development Toolkit to change this. Now redefine all cardinalities as follows:

- For CookingStep, reduce it to the following (we will move all constraints about Start and Stop):

CARDINALITIES min-outgoing:1 max-outgoing:1 min-incoming:1 max-incoming:1

- For Start:

CARDINALITIES max-objects:1 min-objects:1
RELATION "followed By" max-incoming:0 max-outgoing:1

- For Stop:

CARDINALITIES max-objects:1 min-objects:1
RELATION "followed By" max-incoming:1 max-outgoing:0

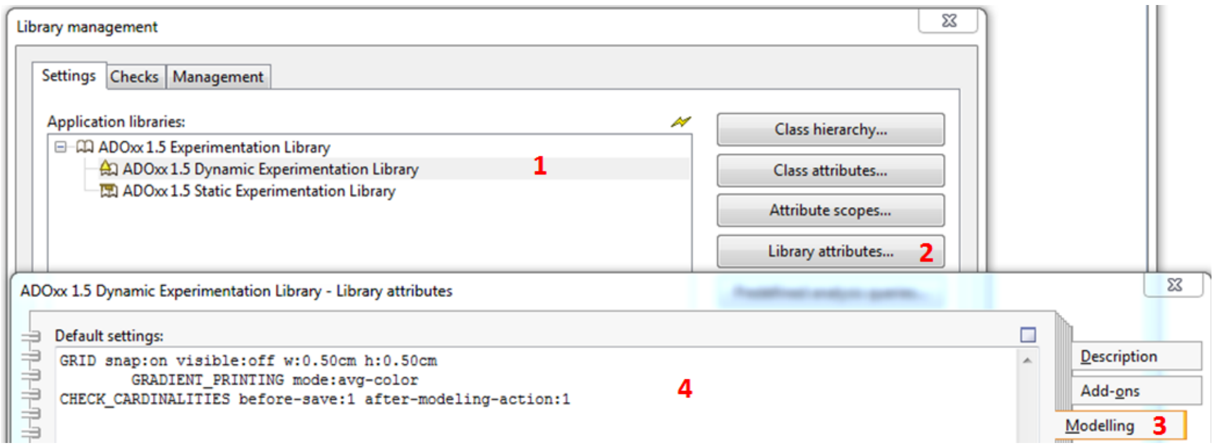
In addition, let's change the moment when cardinalities are checked. In addition to the menu option used before, we will make sure that cardinality is also checked:

- on mouse events (e.g., the moment when you draw a connector, it will be checked if it violates the cardinalities)
- on saving (e.g., to capture any violations that cannot be detected on mouse events, e.g., the absence of a type of elements).

This is defined as a Library attribute:

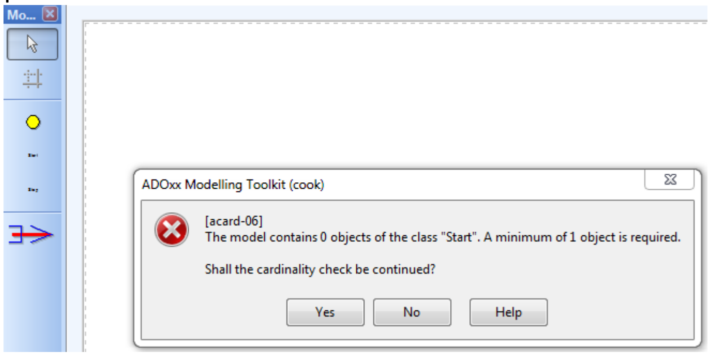
1. Select the library where you work
2. Open Library attributes
3. Select the Modelling section
4. Edit the Default setting attribute by adding the following line, which activates cardinality checks on both mouse events and save operations:

CHECK_CARDINALITIES before-save:1 after-modeling-action:1



```
GRID snap:on visible:off w:0.50cm h:0.50cm
        GRADIENT_PRINTING mode:avg-color
CHECK_CARDINALITIES before-save:1 after-modeling-action:1
```

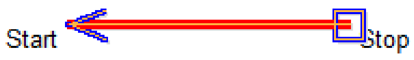
Save and go back to the Modelling Toolkit. Create a new model and try to save it without adding anything to it. You will see errors such as the one saying that at least one Start element should be present.

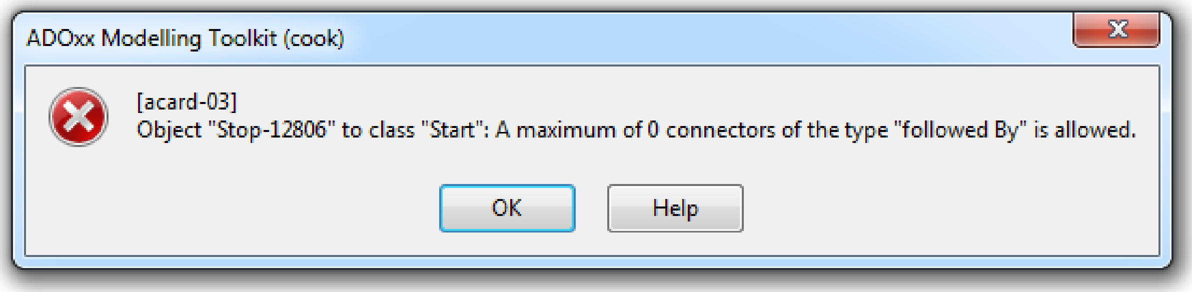


[acard-06]
The model contains 0 objects of the class "Start". A minimum of 1 object is required.

Shall the cardinality check be continued?

Now try to create an arrow from a Stop to a Start. This will trigger a mouse event checking.



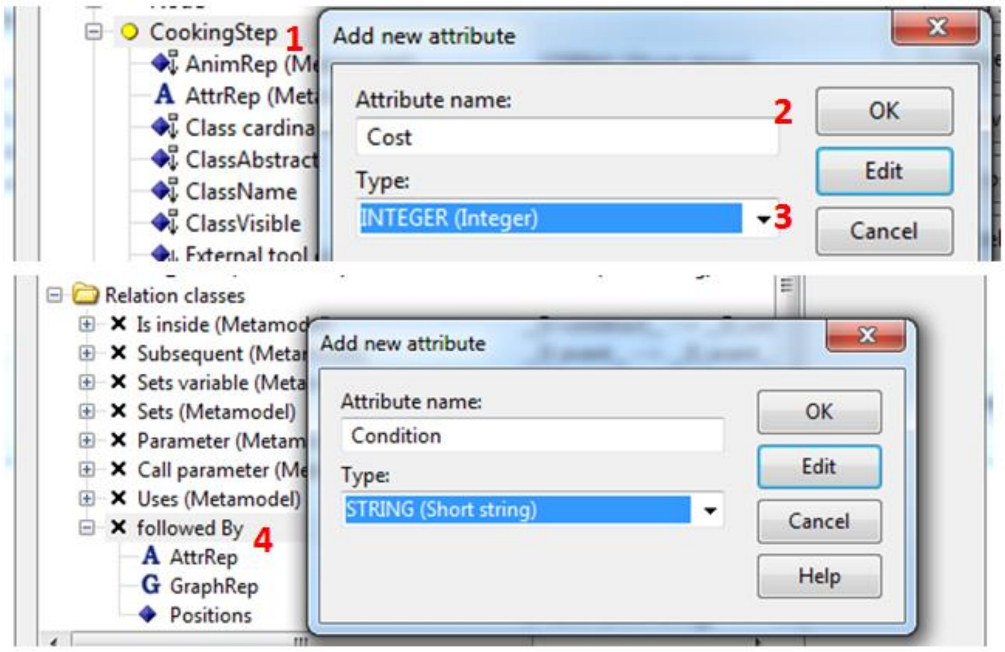

ADOxx Modelling Toolkit (cook)

[acard-03]
Object "Stop-12806" to class "Start": A maximum of 0 connectors of the type "followed By" is allowed.

All the other constraints are still active. If you find that the error messages are too annoying remove the CHECK_CARDINALITIES line from the library attributes (you will probably need to save some incomplete models for some exercises).

Next, let’s take care about the semantics. Go to the Development Toolkit, to the Experimentation Library, and enrich the semantics of your language, as in the picture below:

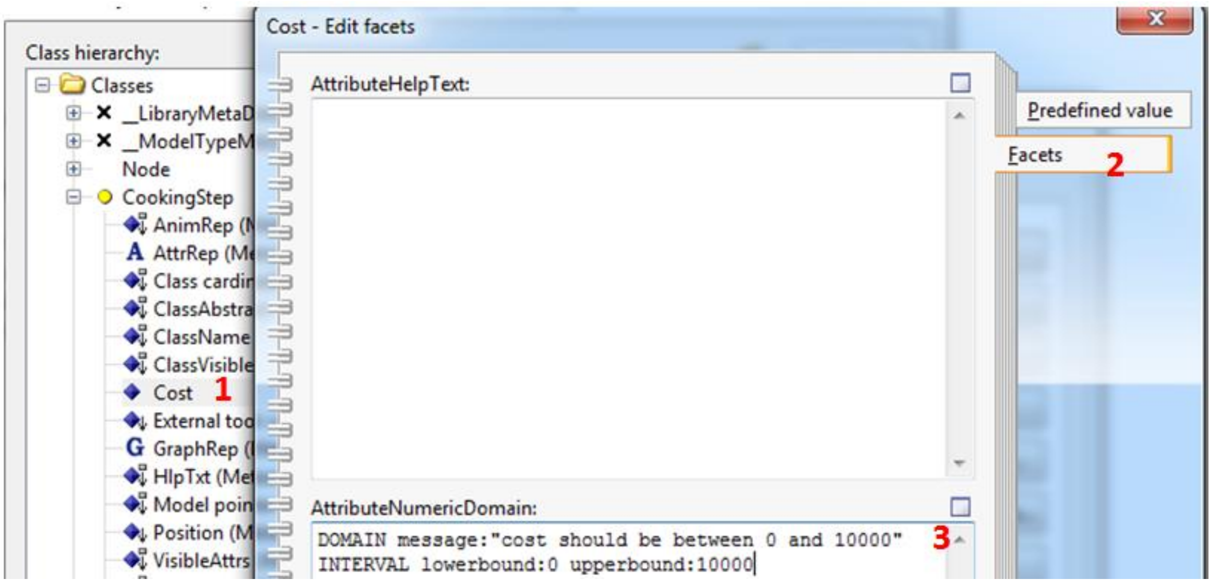
1. Right-click on the CookingStep concept and select New attribute.
2. Let's say that we want to store an estimation of the time for each cooking step. Create a Cost attribute.
3. Select the type Integer (notice the many types of attributes you can define – for example you may also include a Time attribute, for which you have the predefined type Time).
4. Do the same for the “followed By” arrow, where we will store the Condition attribute of type String (to indicate in text form under which condition you should advance to the next cooking step, e.g., “if boiling is finished”).



Now add some extra constraints (the value interval for Cost, the string length for Condition):

1. Double click the Cost attribute
2. Select the Facets
3. Type the following in AttributeNumericDomain (the code sets the interval restriction and an error message):

DOMAIN message:"cost should be between 0 and 10000"
INTERVAL lowerbound:0 upperbound:10000

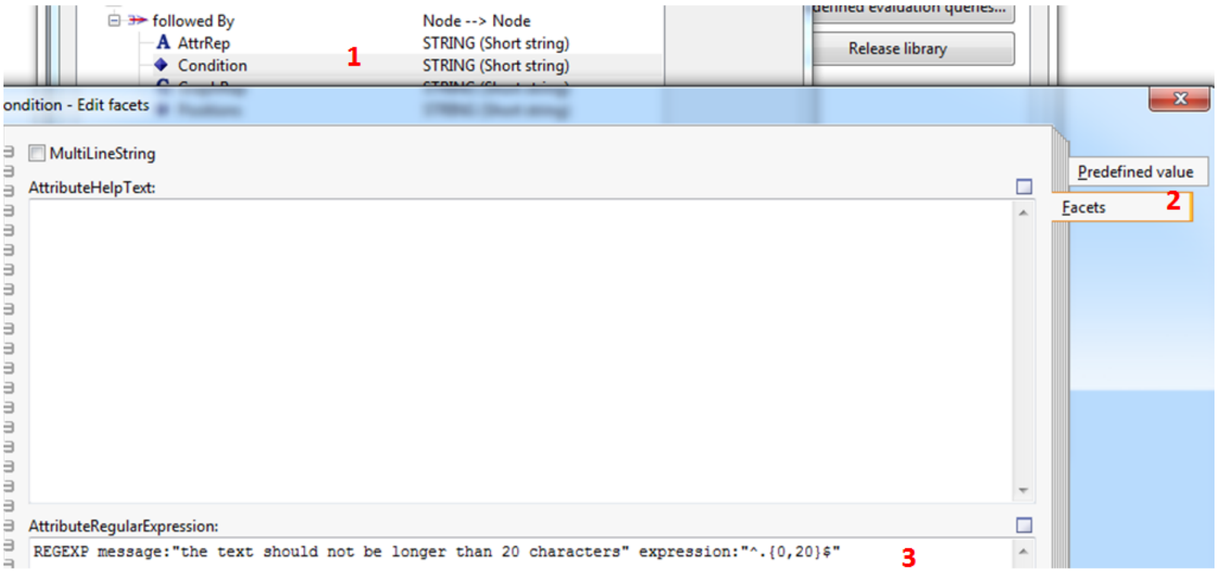


```
DOMAIN message:"cost should be between 0 and 10000"
INTERVAL lowerbound:0 upperbound:10000
```

Do the same to restrict the string length of the Condition attribute. This can be done with regular expression constraints:

1. Double click the Condition attribute
2. Select the Facets
3. Type the following in AttributeRegularExpression, to impose a number of characters between 0 and 20:

REGEXP message:"the text should not be longer than 20 characters" expression:"^.{0,20}$"



```
REGEXP message:"the text should not be longer than 20 characters" expression:"^.{0,20}$"
```

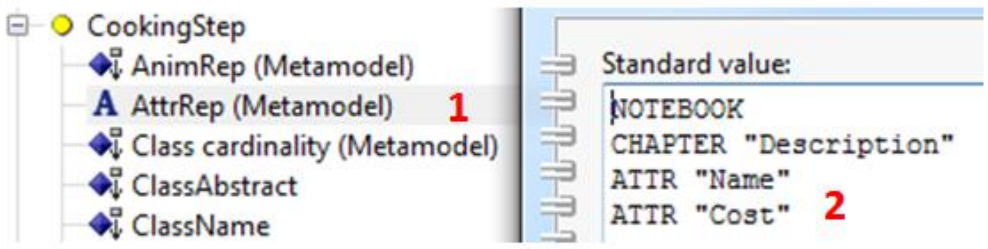
Now the language is richer in semantics, but the properties must become **editable by the user**. In order to do this, you need to indicate, in the inherited AttrRep, which of the properties should be visible to the user:

1. Select AttrRep for the CookingStep
2. Use the NOTEBOOK syntax to define the visible attributes. With CHAPTER you may group attributes in several pages (if there are too many attributes). With ATTR you must indicate exactly the attribute name to be made visible.

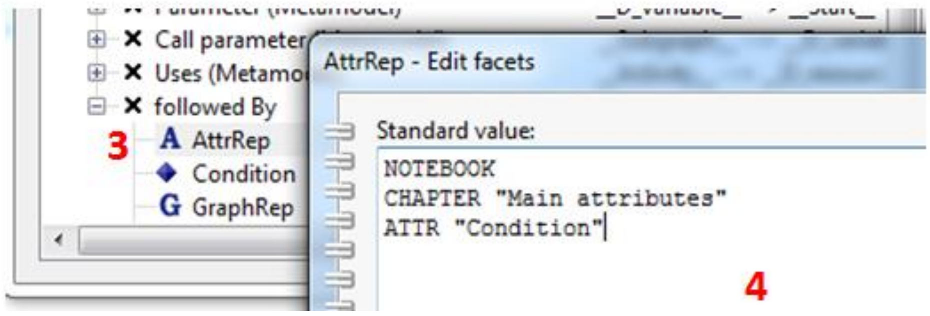
NOTEBOOK
CHAPTER "Description"
ATTR "Name"
ATTR "Cost"

3. Do the same for the AttrRep of the relation
4. Make visible the Condition attribute:

NOTEBOOK
CHAPTER "Main attributes"
ATTR "Condition"



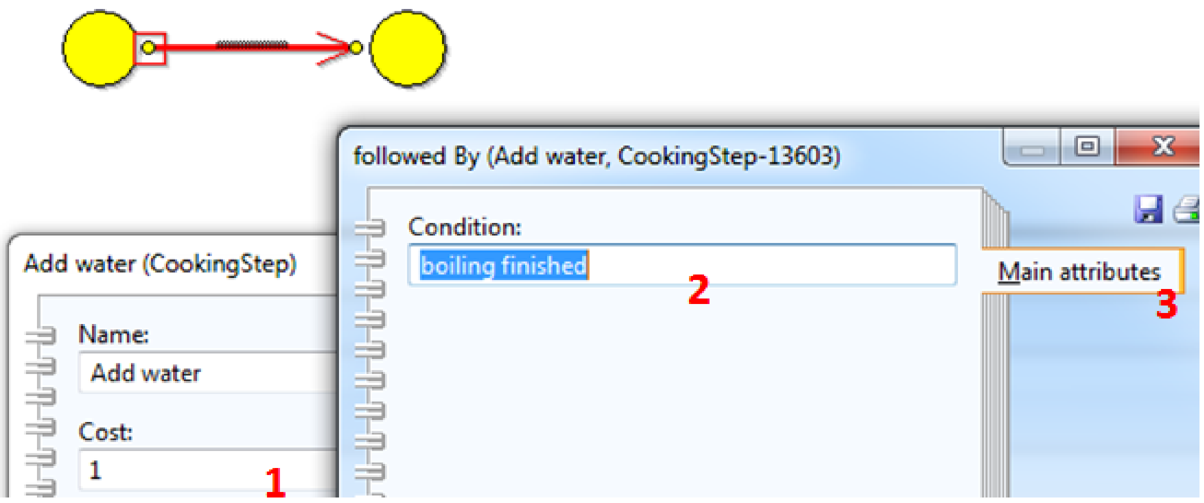
```
NOTEBOOK
CHAPTER "Description"
ATTR "Name"
ATTR "Cost"
```



```
NOTEBOOK
CHAPTER "Main attributes"
ATTR "Condition"
```

Save the changes and log back in the modelling tool. Now you will see:

1. On double clicking a step, you will see its editable properties (Name and Cost)
2. On double clicking an arrow, you will see the Condition.
3. Also notice the chapter "Main attributes" that you defined to group the attributes. Multiple chapters can be defined to paginate attributes (when they are too many).



Name: Add water

Cost: 1

Condition: boiling finished

Now we have a language with richer semantics, but the notation is still weak, since it does not even display the names of the elements. We have to enrich the graphical notation to make it communicate some semantics (property values).

Go back to the Development Toolkit and replace the GraphRep of the CookingStep with the following code[2]:

```
GRAPHREP

FONT "Arial" h:12pt color:red
PEN color:red

FILL color:yellow
ELLIPSE rx:0.5cm ry:0.5cm

ATTR "Name" y:1cm w:c
ATTR "Cost" y:-1cm w:c
```

Notice that we added:

- The FONT definition to be used for all the textual displays that follow
- The ATTR includes the values of the attributes Name and Cost. The position of their display is also indicated: the name will be below the center of the symbol (y:1cm), the cost will be above (-1cm). The w parameter defines the horizontal position – to avoid setting the x coordinate, we use the "c" value which will center the text horizontally.

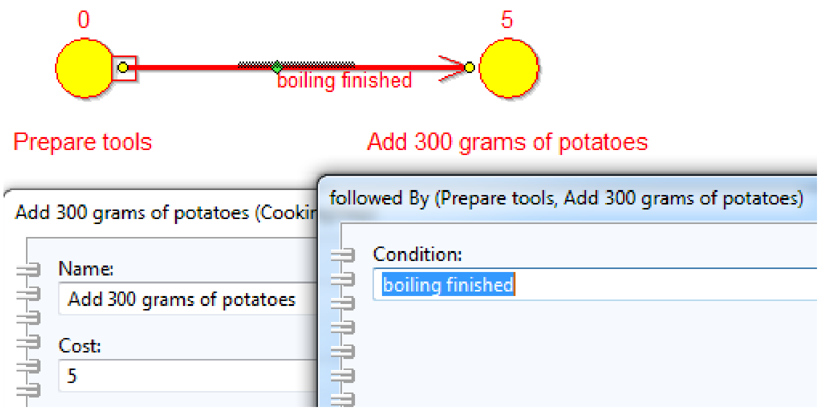
We do something similar to the “followed By” relation, by including the Condition attribute in the EDGE component of the arrow. Since we do not specify position, it will get a default position (the text will be displayed starting from the middle point of the arrow).

```
GRAPHREP

PEN color:red w:0.1cm

EDGE
ATTR "Condition"

PEN color:blue

START
POLYGON 4 x1:-0.2cm y1:0.2cm x2:0.2cm y2:0.2cm x3:0.2cm y3:-0.2cm  x4:-0.2cm y4:-0.2cm

END
POLYLINE 3 x1:-0.5cm y1:0.2cm x2:0cm y2:0cm x3:-0.5cm y3:-0.2cm
```

Note: Please check the ADOxx help for the complete syntax of the declarative languages used in GRAPHREP, ATTRREP and MODI. This exercise will only illustrate some of the more frequently used parameters.

Log back in the modelling tool and see how the notation was extended with semantics (add some values in the Cost and Condition to have some semantics for display).

[2] If you want to keep the old code as a comment, include it between (* and*)



Add 300 grams of potatoes (Cookin

Name:
Add 300 grams of potatoes

Cost:
5

followed By (Prepare tools, Add 300 grams of potatoes)

Condition:
boiling finished

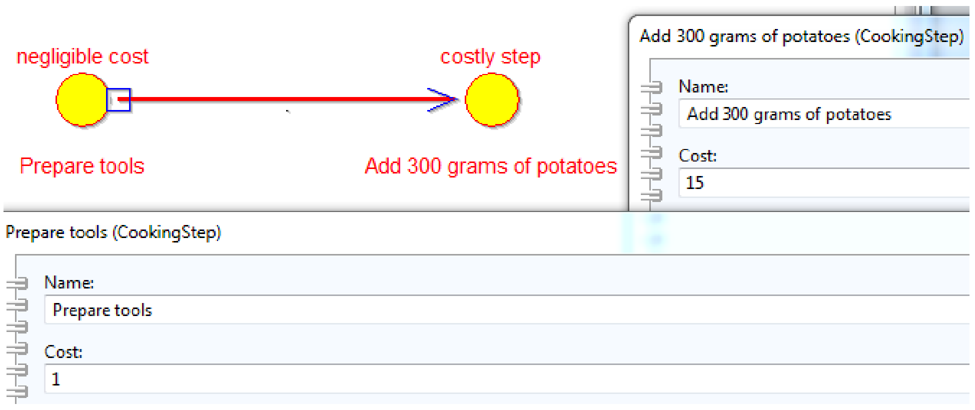
Notational variability is achieved when the information available on a purely visual level changes dynamically together with the semantics. The simplest case of variability is this, when the notation displays directly an attribute value. In other cases, we might want more subtle ways of suggesting changes in properties. In the next example, change the GraphRep of the CookingStep as follows:

```
GRAPHREP

AVAL c:"Cost"

FONT "Arial" h:12pt color:red
PEN color:red

FILL color:yellow
ELLIPSE rx:0.5cm ry:0.5cm

ATTR "Name" y:1cm w:c
TEXT (cond(VAL c>10,"costly step","negligible cost")) y:-1cm w:c
```

With AVAL, we create the variable c to store the value of the Cost attribute. The variable will allow us to do some programming inside the GRAPHREP code! (IF conditions, FOR loops etc.).

In our case, the ATTR "Cost" command was replaced with a TEXT command. This, as used before, will display a static text. However, now the static text is computed based on an IF test (the function cond()) which checks if the cost value is higher than 10 or not. If it is, the displayed text will be "costly step". If not, the text will be "negligible cost". The test is performed on VAL c, which converts a string to a number (by default, every attribute value is imported in the GraphRep code as a string)

Check the effect in the modelling tool, by setting different costs on different cooking steps:



Add 300 grams of potatoes (CookingStep)

Name:
Add 300 grams of potatoes

Cost:
15

Prepare tools (CookingStep)

Name:
Prepare tools

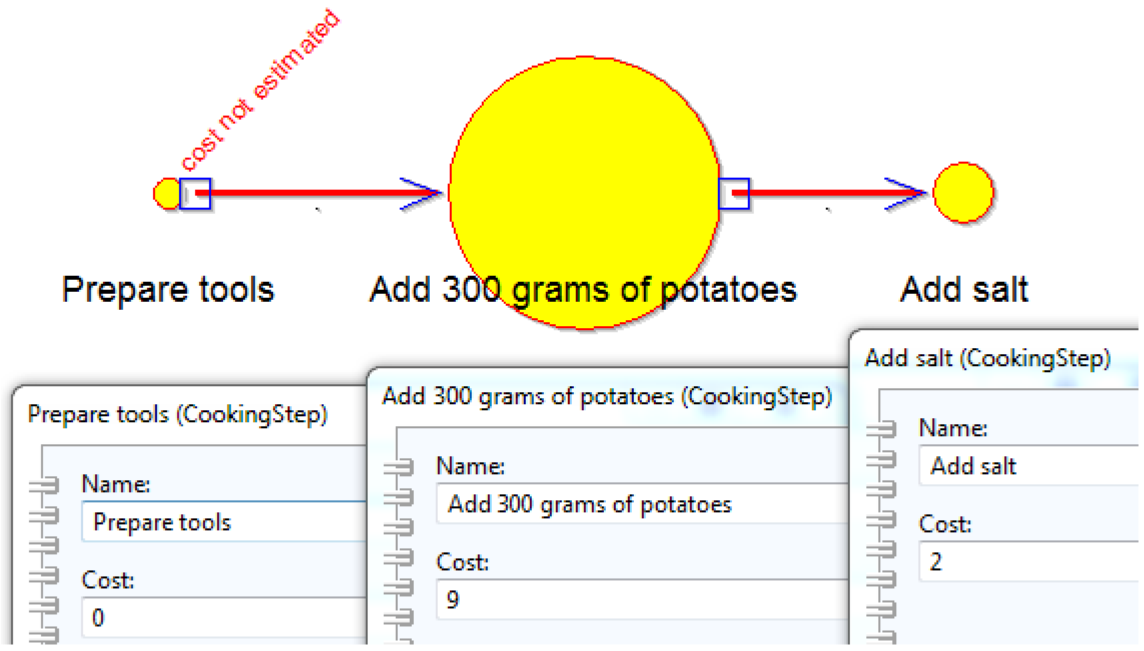
Cost:
1

Notational variability can be extended to influence the graphical shapes. For this, a numerical property of the graphical shape (e.g., size, position, color code, font size) must be computed based on some attribute value. In the next example, the size (radius) of the yellow circles will be proportional with the cost attribute – 1/5 of the cost value. However, we must also consider what should happen if the Cost is zero (a zero-sized circle will be invisible!). In order to avoid this situation, we use an IF structure, to create a default GraphRep for the case when Cost=0.

Replace the GraphRep of a Cooking Step with the following:

```
GRAPHREP

AVAL c:"Cost"

PEN color:red
FILL color:yellow

IF (c="0")
  FONT "Arial" h:10pt color:red line-orientation:45
  ELLIPSE rx:0.2cm ry:0.2cm
  TEXT "cost not estimated" y:-0.5cm
ELSE
  ELLIPSE rx:(CM c/5) ry:(CM c/5)
ENDIF

FONT "Arial" h:14pt color:black
ATTR "Name" y:1cm w:c
```

Notice the IF test which checks if the Cost is zero. If yes, a default circle is drawn, with a text on diagonal: “cost not estimated”. If not, the Cost value is divided by 5 and converted in CM (centimeter values).



Prepare tools (CookingStep)

Name: Prepare tools

Cost: 0

Add 300 grams of potatoes (CookingStep)

Name: Add 300 grams of potatoes

Cost: 9

Add salt (CookingStep)

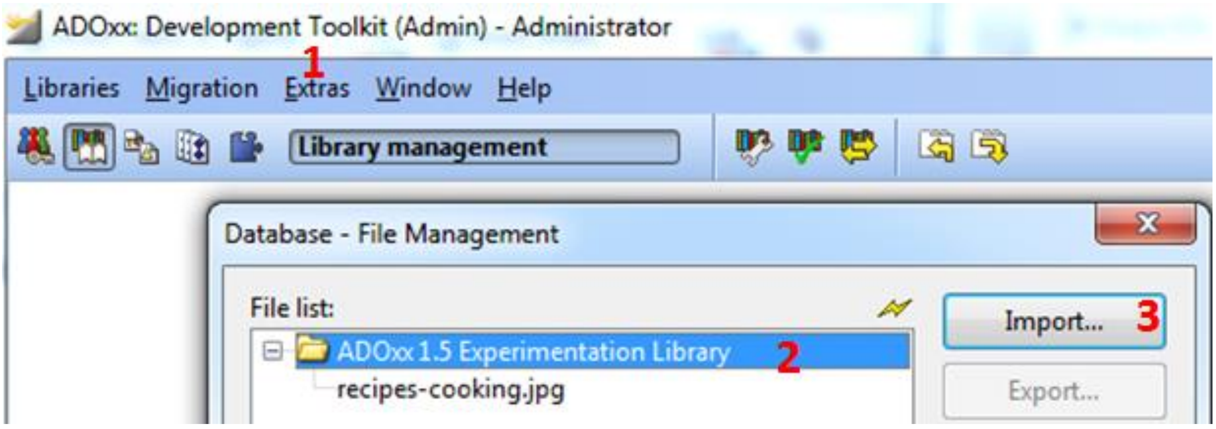
Name: Add salt

Cost: 2

Of course, with an IF structure the notation could be completely changed based on some condition placed on attribute values (or a combination of conditions). So you may have multiple notations prepared for the same concept, to communicate different “states” of a model element.

You also have the possibility to mix vectorial shapes (created with the GRAPHREP syntax) and external graphic files (JPG, PNG etc.). If you want to include external graphic files in your notation, first you need to import them in the ADOxx resource database:

1. In the Extras menu, you will find a File Management option
2. Select the Experimentation library
3. Press Import and choose a graphics file from your disk.

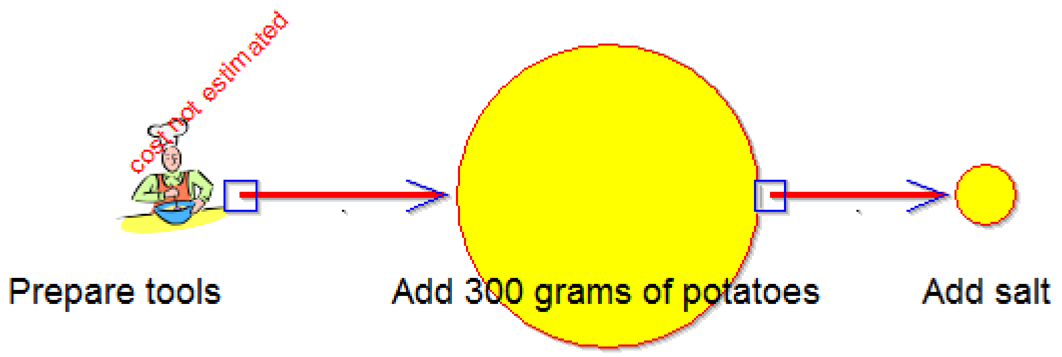

ADOxx: Development Toolkit (Admin) - Administrator
Libraries
Migration
1
Extras
Window
Help
Library management
Database - File Management
File list:
ADOxx 1.5 Experimentation Library
2
recipes-cooking.jpg
Import...
3
Export...

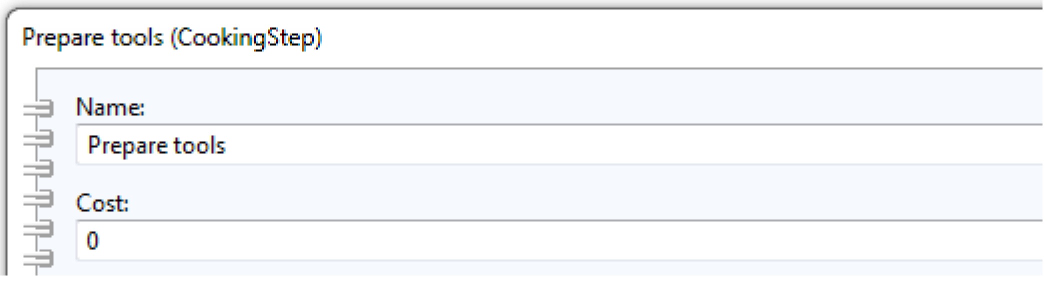
Now change the GraphRep definition for CookingStep, by replacing the first IF branch as shown below:

```
GRAPHREP

AVAL c:"Cost"

PEN color:red
FILL color:yellow

IF (c="0")
  FONT "Arial" h:10pt color:red line-orientation:45
  BITMAP ("db:\\recipes-cooking.jpg") y:-1cm w:1.5cm h:1.5cm
  TEXT "cost not estimated" y:-0.5cm
ELSE
  ELLIPSE rx:(CM c/5) ry:(CM c/5)
ENDIF

FONT "Arial" h:14pt color:black
ATTR "Name" y:1cm w:c
```

The example assumes that we imported in ADOxx an external JPG file called "recipes-cooking.jpg". The standard path for imported files is "db:\\.....".

The BITMAP command allows us to include in the notation the imported file and also to control its size and position. Notice that this replaces the default ELLIPSE that we had before, for the case when the Cost attribute is zero. Now in the modelling tool you can see that the JPG file replaces the circle for the nodes where Cost was not specified. Thus you may include any number of external graphical files, to be displayed based on different values in your concept semantics!




Prepare tools (CookingStep)

Name:

Prepare tools

Cost:

0

Note: you may change an attribute value for multiple model elements at the same time. Select all the elements with Ctrl-click, then go to *Edit-Change attributes*.