Implementing Model-based Functionality in ADOxx

Preparation steps

A modelling tool does not provide only the possibility of creating diagrammatic models. It should also process those models in ways that are relevant to the user and to the modelling language. Some functionality is already built-in ADOxx (e.g., model queries, export of models in XML). Other functionality may be programmed by using the internal ADOScript language, which we'll exemplify here. We will still use the cooking library.

In ADOxx Development Toolkit:

- Go to Library Management in the Settings tab
- Select the Dynamic Library component (of the Cooking Library)
- Press the Library attributes button
- Go to Add-ons -> External Coupling

There you will find a default event handler (ON_EVENT "AppInitialized" {}). Under this event handler, add the following code:

```
ITEM "Console" modeling:"My Menu" evaluation:"My Menu"

CC "AdoScript" EDITBOX text:(myscript)
 fontname:"Courier New" fontheight:12 title:"Type here some AdoScript code" oktext:"Run the code"

IF (endbutton="ok") {
 SETG myscript:(text)
 EXECUTE (text)
}

IF (type(myscript)="undefined") {
 SETG myscript:""
}
```

Remarks:

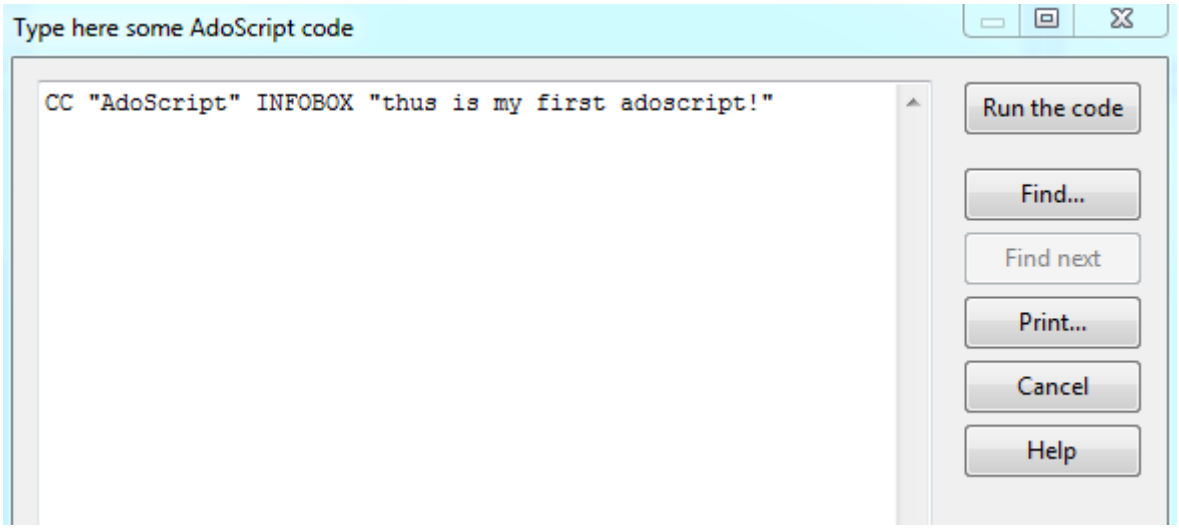
- ITEM defines a new menu option labelled "Console"; the option is included in a new menu labelled "My Menu"; this menu will be included in the modeling and evaluation components (menu bars) of the modelling tool; everything that follows after the ITEM line will be the code to be executed when someone selects the "Console" option
- CC is an inclusion statement (similar to "require" in PHP, "from import" in Python etc.) – ADOScript functions and objects are grouped in something similar to "libraries" - but we'll avoid calling them "libraries" because the word is already used here for something else, as we've seen; instead, they are called "messageports" or "component calls" or APIs. One of the most important ones is "AdoScript", from which we need functions and objects for interfacing both with the UI (e.g., messageboxes) or with the file system (e.g., files on the disk);
- In this case we use (from the AdoScript messageport) the EDITBOX object to create an editable textbox
    - The textbox will be displayed when the menu item Console is clicked (because it is defined after the Console ITEM); its goal is to allow us to test AdoScript code directly

in the modelling toolkit (without switching between the Development and Modelling toolkits for every line we want to test);

- The box has an initial text (parameter *text*, with the value taken from the *myscript* variable, which will remember whatever we type even if we close the box), a title (*title*), a selected font (*fontname, fontheight*) and an OK button to run the code that we type (parameter *oktext* defines the text on the button); other buttons will also exist since they are default (e.g., Help button)
- The first IF statement tests if the user pressed the ok button in the box; IF yes, the typed text is stored in the *myscript* variable (SETG sets a global variable); EXECUTE will then execute the typed text (which should be valid AdoScript code);
- The second IF statement tests if the myscript variable was set and if not, it sets it to an empty string (just to avoid an error in case the box is left empty).

Press *Apply*, confirm saving and switch to the modelling toolkit. You should see your newly created menu and its Console option, which will open an editable textarea. Type the following code:

```
CC "AdoScript" INFOBOX "this is my first adoscript!"
```



Press *Run the code* and you should see a messagebox (created with the INFOBOX object from the same AdoScript messageport). You can search the help for "MessagePorts" to see all of the ports (then select AdoScript to see the functions and objects available in that messageport. Or, you can check the on-line documentation: https://www.adoxx.org/AdoScriptDoc/

Replace the code with the following:

```
SET mytext: "this is a string"
CC "AdoScript" INFOBOX (mytext)
```

Notice

- how a variable can be set (with : and not with = as in other languages)
- how that variable is used (with parantheses)

From now on we can use this box to type and test small pieces of AdoScript code. However, if we want to write complex algorithms, we might prefer to write them in Notepad++[1]

Write the following code in Notepad++, in a file mycode.asc:

```
SET mytext:"this is a message"
CC "AdoScript" INFOBOX (mytext)
```

Save it in a convenient folder – e.g., C:\myscripts\

Now go to the Development Toolkit in External Coupling. Add a new menu ITEM, in a new menu, which should read the .asc file and execute it:

```
ITEM "Display message" modeling:"My Second Menu"
CC "AdoScript" FREAD file:("C:\\myscripts\\mycode.asc")
EXECUTE (text)
```

Now restart the modelling toolkit and you should see the new menu with the new option. Click it to run the code from the file. Any changes saved in the file can be executed with this new option. Type the following code:

```
CC "Modeling" GET_ACT_MODEL
CC "AdoScript" INFOBOX (STR modelid)
```

This time we used, from the Modeling messageport, the function GET_ACT_MODEL which returns the internal identifier for the active model. Then an INFOBOX is used to display it as a string. Test it in the modelling toolkit. You have to open/create a model, otherwise it will return -1 (an exception code for "no active model").

Notice that the string conversion was applied on a variable with a predefined name, *modelid*. Most functions in AdoScript return their values in such predefined constants or variables, and we need to know their predefined names in order to further access them (see also the variable called *text* in prior examples). How can we know these names? There are two options:

- we can consult the Development toolkit help for that function (or search for the function name in https://www.adoxx.org/AdoScriptDoc/)
- we can run the function in debug mode as follows:

Replace those two lines with a single one, including the keyword *debug* before the function name:

```
CC "Modeling" debug GET_ACT_MODEL
```

This will automatically generate an INFOBOX with debugging information: the line of code from the source file, as well as the constant name that is returned by the function (*modelid*). After a while you will end up knowing by heart these names (text, modelid, objid, classid, objname, classname etc.) but the beginners will use quite often the debugging mode.

[1] A syntax highlighter may be installed in Notepad++. See details at https://www.adoxx.org/live/adoxx-development-languages-syntax-support#AdoScript_syntax_highlighting

Scripts that read information from existing models

Replace with the following code:

```
CC "Modeling" GET_ACT_MODEL
SET activemodel:(modelid)
CC "Core" debug GET_ALL_OBJS_OF_CLASSNAME modelid:(activemodel) classname:"CookingStep"
```

The last line displays in debug mode the result of GET_ALL_OBJS_OF_CLASSNAME which:

- takes as input parameters a model id (here the id of the active model, taken from GET_ACT_MODEL) and the name of a concept from that model (here CookingStep);
- returns several values:
    - objids is an array of object identifiers, for all the node elements of the specified type (concept) in that model
    - ecode and errtext are some error indicators (e.g., in case the concept does not exist)

In the next example, we display the names (instead of IDs) for all objects present in the active model. For this, we need to create a FOR loop over the objids array, to extract each separate ID and to read its name attribute with GET_OBJ_NAME

```
CC "Modeling" GET_ACT_MODEL
SET activemodel:(modelid)
CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(activemodel) classname:"CookingStep"
SET message:""
FOR i in:(objids)
{
        CC "Core" GET_OBJ_NAME objid:(VAL i)
        SET message:(message+objname+"\n")
}
CC "AdoScript" INFOBOX (message)
```

(VAL is necessary to convert the ID from string - as it is stored in *objid* -, to a numeric value - as required for the parameter *objid* of GET_OBJ_NAME)

In the next example, we loop again through all objects in the active (open) model. This time we display the list of attributes that are visible in each object and:

- for every attribute of type integer we also add its value
- for any other type of attribute we just display the name followed by the message "has non-integer value"

```
CC "Core" GET_CLASS_ID classname:"CookingStep"
SET cookingstepid:(classid)
CC "Core" GET_ALL_NB_ATTRS classid:(cookingstepid)
SET attributeidentifiers:(attrids)

CC "Modeling" GET_ACT_MODEL
SET activemodel:(modelid)
CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(activemodel) classname:"CookingStep"
SET objectidentifiers:(objids)

SET message:""
FOR i in:(objectidentifiers)
```

```
{
	SET currentobjectid:(VAL i)
	CC "Core" GET_OBJ_NAME objid:(currentobjectid)
	SET message:(message+objname+"\n")
	FOR j in:(attributeidentifiers)
	{
		SET currentattributeid:(VAL j)
		CC "Core" GET_ATTR_NAME attrid:(currentattributeid)
		SET currentattributename:(attrname)
		CC "Core" GET_ATTR_VAL objid:(currentobjectid) attrid:(currentattributeid)
		SET currentattributevalue:(val)
		IF (type(currentattributevalue)="integer")
			{
			SET message:(message+" "+currentattributename+":"+(STR currentattributevalue)+"\n")
			}
		ELSE
			{
			SET message:(message+" "+currentattributename+": has non-integer value\n")
			}
	}
	SET message:(message+"\n")
}
CC "AdoScript" VIEWBOX text:(message) title:"Attributes of all node elements in the model:"
```

This time we displayed the information in a VIEWBOX object (a read-only version of the EDITBOX). Notice the functions that we used to extract various information from the currently open model:

- GET_ACT_MODEL to obtain the id of the current model; no input parameters are necessary, the result is stored in modelid
- GET_ALL_OBJS_OF_CLASSNAME to obtain an array of ids for all node objects in the current model; input parameters are the model id and the class name, the result is stored in objids
- GET_CLASS_ID to obtain the id of a class; input parameter is the class name, the result is stored in classid
- GET_ALL_NB_ATTRS to obtain an array of ids for all attributes visible for a class; input parameter is the class id, the result is stored in attrids
- GET_OBJ_NAME to obtain the name of a node element; input parameter is the object id, the result is stored in objname
- GET_ATTR_NAME to obtain the name of an attribute; input parameter is the attribute id, the result is stored in attrname
- GET_ATTR_VAL to obtain the value of an attribute; input parameter is the object id and the attribute it, the result is stored in val

Notice that we used very often the SET command to capture the result of a function in a local variable named by us (i.e., the output of a function is passed to a variable, then the variable is passed as input to another function). This makes sense if those variables are reused in many places and there's a danger that the output variables created by functions (e.g., modelid, classid, objid) may be overwritten. This is not the case in our example, so we can simplify the code to directly use the output variables created by each function (we already discussed how we can find their predefined names). The example below does the same thing, without using any intermediate variables, so it's much shorter (the predefined output variables are bolded):

```
CC "Modeling" GET_ACT_MODEL
CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(modelid) classname:"CookingStep"
CC "Core" GET_CLASS_ID classname:"CookingStep"
CC "Core" GET_ALL_NB_ATTRS classid:(classid)
```

```
SET message:""
FOR i in:(objids)
{
	CC "Core" GET_OBJ_NAME objid:(VAL i)
	SET message:(message+objname+"\n")
	FOR j in:(attrids)
	{
		CC "Core" GET_ATTR_NAME attrid:(VAL j)
		CC "Core" GET_ATTR_VAL objid:(VAL i) attrid:(VAL j)
		IF (type(val)="integer")
			{
			SET message:(message+" "+attrname+":"+(STR val)+"\n")
			}
		ELSE
			{
			SET message:(message+" "+attrname+": has non-integer value\n")
			}
	}
	SET message:(message+"\n")
}
CC "AdoScript" VIEWBOX text:(message) title:"Attributes of all node elements in the model:"
```

For the next example, we will access the table attribute named "required ingredient quantities". This should be present in all CookingStep objects and should have two columns: "Ingredient" (a hyperlink to an ingredient object from resource models) and "Quantity" (an integer). So in order to test the next example:

- Also create a resource model with 2-3 ingredients with simple names ("Salt", "Water" etc.);
- In your Cooking Steps fill the "required ingredient quantities" table with 2-3 rows linking to those ingredients (plus some quantities); have at least 3 CookingSteps with 2-3 rows filled for each of them.

The following script works like this:

- It loops through all CookingSteps
- From each of them it reads the "requires ingredient quantities" table
    - It loops again through the rows of the table
    - For each row it builds a message showing both the ingredient name and the quantity

```
CC "Core" GET_CLASS_ID classname:"CookingStep"
# returns classid for CookingStep
CC "Core" GET_ATTR_ID classid:(classid) attrname:"requires ingredient quantities"
# returns attrid for the table attribute "requires ingredient quantities" of "CookingStep" concept

CC "Modeling" GET_ACT_MODEL
CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(modelid) classname:"CookingStep"
# returns objids with all instances of CookingStep in the current model
SET message:""
FOR i in:(objids)
{
	CC "Core" GET_OBJ_NAME objid:(VAL i)
	#returns objname for the current object, in order to display it
	SET message:(message+objname+" requires ingredient quantities:\n")
	CC "Core" GET_ALL_REC_ATTR_ROW_IDS objid:(VAL i) attrid:(attrid)
	#returns rowids for all the rows in the table attribute in the current object
	FOR j in:(rowids)
	{
		CC "Core" GET_ATTR_VAL objid:(VAL j) attrname:"Ingredient" format:"%o"
		#returns val of the Ingredient attribute in the current row, in a format that only shows the object name
		SET ingredientname:(val)
		CC "Core" GET_ATTR_VAL objid:(VAL j) attrname:"Quantity" as-string
```

```
                #returns val of the Quantity attribute in the current row, in string format, thus conversion with STR is not neccessary
                SET quantityvalue:(val)
                SET message:(message+" "+ingredientname+":"+quantityvalue+"|")
        }
        SET message:(message+"\n")
}
CC "AdoScript" VIEWBOX text:(message) title:"required ingredient quantities for all elements:"
```

Notice that we kept the code short, without including all those SET commands, but comments are included now to indicate clearly what predefined variable is generated by each function.

We modify the example so that, instead of displaying the ingredient and quantity, we see for each object a list of targeted object ids.

```
CC "Modeling" GET_ACT_MODEL
CC "Core" GET_CLASS_ID classname:"CookingStep"
# returns classid for CookingStep
CC "Core" GET_ATTR_ID classid:(classid) attrname:"requires ingredient quantities"
# returns attrid for the table attribute "requires ingredient quantities" of "CookingStep" concept
CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(modelid) classname:"CookingStep"
# returns objids with all instances of CookingStep in the current model
SET message:""
FOR i in:(objids)
{
        CC "Core" GET_OBJ_NAME objid:(VAL i)
        #returns objname for the current object, in order to display it
        SET message:(message+objname+" references the following objects:\n")
        CC "Core" GET_ALL_REC_ATTR_ROW_IDS objid:(VAL i) attrid:(attrid)
        #returns rowids for all the rows in the table attribute in the current object
        FOR j in:(rowids)
        {
                CC "Core" GET_INTERREF objid:(VAL j) attrname:"Ingredient"
                #returns tobijd, which is the ID of the object targeted by the INTERREF attribute Ingredient
                #(other information is also returned: the target model, the target concept etc.... check the documentation)
                SET message:(message+" "+(STR tobjid)+"|")
        }
        SET message:(message+"\n")
}
CC "AdoScript" VIEWBOX text:(message) title:"referenced objects for all elements:"
```

You may notice that this script only works if the targeted model (Cooking Resources) is open, otherwise the targeted object ids are -1 (which means that the script is not able to access those objects and read their attributes). If you include the debug keyword in GET_INTERREF (as shown in the beginning) you will notice that the function can read the targeted model id (in tmodelid) but not the targeted objects (tobjid).

In order to avoid this obstacle you have two options:

- open manually the targeted model before executing the script;
- extend the script to automatically open the targeted model, by replacing the inner FOR loop with the following code:

```
FOR j in:(rowids)
{
        CC "Core" GET_INTERREF objid:(VAL j) attrname:"Ingredient"
        CC "Modeling" IS_OPENED modelid:(tmodelid)
        IF (NOT isopened=1)
                {
                CC "Modeling" OPEN modelids:(tmodelid)
                }
        CC "Core" GET_INTERREF objid:(VAL j) attrname:"Ingredient"
        SET message:(message+" "+(STR tobjid)+"|")
```

```
}
```

Notice that GET_INTERREF is executed twice:

- first to obtain the targeted modelid (which is available even if the model is closed);
- then the IS_OPENED function tests if this model is open (it generates the isopened variable which must be tested with an IF!)
- IF the model is not open, the OPEN function is executed on it, otherwise nothing happens
- Finally a second GET_INTERREF is executed to retrieve the object ids (and any other information we need from the targeted open model).

An alternative to the OPEN function is the LOAD function, which only opens a model in the computer memory (to access its contents) without displaying it on the screen.

Scripts that change attributes in an existing model

Now switch back to the Development toolkit to add a new menu item. Just like before, add the following code at the end of the Library attribute named External Coupling:

```
ITEM "Increment all costs and extend ingredient tables" modeling:"My Second Menu"
CC "AdoScript" FREAD file:("C:\\myscripts\\mycode2.asc ")
EXECUTE (text)
```

This, of course, means that the next example will be saved in a file mycode2.asc in the same folder myscripts.

Restart the Modelling toolkit to see the new menu item. As the name suggests, the new menu option will increase all costs in the current model by 1. The script is the following:

```
CC "Modeling" GET_ACT_MODEL
CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(modelid) classname:"CookingStep"
FOR i in:(objids)
{
        CC "Core" GET_ATTR_VAL objid:(VAL i) attrname:"Cost"
        CC "Core" SET_ATTR_VAL objid:(VAL i) attrname:"Cost" val:(val+1)
}
CC "AdoScript" INFOBOX "costs are now incremented"
```

Notice that we used SET_ATTR_VAL to increase the value of the attribute Cost. The old value was obtained with GET_ATTR_VAL on the previous line (which stores it in val). Test the script and check if the costs are incremented successfully.

Now we will extend this script so that the table attribute "requires ingredient quantities" is extended with a new row, and that row is filled as follows:

- On the "Ingredients" column we include an INTERREF hyperlink to the object "Salt" from a model "Cooking Resources – new" (make sure that you have an object with that name in a model with that name!)
- On the "Quantity" column we include the value 100

The extended script is the following:

```
CC "Modeling" GET_ACT_MODEL
CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(modelid) classname:"CookingStep"

CC "Core" GET_CLASS_ID classname:"CookingStep"
```

```
CC "Core" GET_ATTR_ID classid:(classid) attrname:"requires ingredient quantities"
SET tableattributeid:(attrid)

CC "Core" GET_REC_CLASS_ID attrid:(tableattributeid)
CC "Core" GET_ATTR_ID classid:(classid) attrname:"Ingredient"
SET ingredientcolumnid:(attrid)

FOR i in:(objids)
{
	CC "Core" GET_ATTR_VAL objid:(VAL i) attrname:"Cost"
	CC "Core" SET_ATTR_VAL objid:(VAL i) attrname:"Cost" val:(val+1)
	CC "Core" ADD_REC_ROW objid:(VAL i) attrid:(tableattributeid)
	CC "Core" ADD_INTERREF objid:(rowid) attrid:(ingredientcolumnid) tmodelname:"Cooking Resources - new"
tmodeltype:"Cooking Resources" tclassname:"CookingIngredient" tobjname:"Salt"
	CC "Core" SET_ATTR_VAL objid:(rowid) attrname:"Quantity" val:100
}
CC "AdoScript" INFOBOX "costs are now incremented and a new row is added"
```

Notice the following:

- We used GET_ATTR_ID twice: first to get the ID of the table attribute, then to get the ID of the Ingredient column, which is also considered an attribute; we used SET to store the two values in separate variables: tableattributeid and ingredientcolumnid respectively
- However, in the second case, in order to obtain a column id (by applying GET_ATTR_ID on the table), we need to know the class id of that table. For this we use GET_REC_CLASS_ID (and not GET_CLASS_ID) because this is a RECORD class (remember how we created this table!)
- Inside the FOR loop we have the following operations:
    - SET_ATTR_VAL increments the cost (kept from the previous example)
    - ADD_REC_ROW adds a new row in the table based on the tableattributeid (created with the first SET)
    - ADD_INTERREF injects a hyperlink in the first column of this new row.
        - The new row is identified with the predefined rowid (which is an output of ADD_REC_ROW).
        - The column is identified with ingredientcolumnid (created with the second SET)
        - The target of the hyperlink must be completely specified with the following fields: tmodelname (the target model name), tmodeltype (the target model type), tclassname (the target concept name), tobjname (the target object name). Make sure that all these names exist and are correct! (the model name and object name can be edited in the modelling tool, the model type and the concept name have been defined in the Development toolkit)
    - The last SET_ATTR_VAL injects the value 100 in the Quantity column; this works similarly to the previous SET_ATTR_VAL – we are allowed to access the column by name (so we did not have to finds its id, as in the case of ADD_INTERREF which only allows us to access a column by ID)

When testing the script, make sure that the target model and object exist with exactly the names defined in the script.

Modify the code so that these changes are only applied on a set of selected elements (notice the bolded lines – GET_SELECTED is used instead of GET_OBJS_OF_CLASSNAME and an IF was included to make sure that the changes are only applied for cooking steps; if other types of objects are also selected, a warning message will be displayed):

```
CC "Modeling" GET_ACT_MODEL
CC "Core" GET_CLASS_ID classname:"CookingStep"
SET cookingstepid:(classid)
CC "Core" GET_ATTR_ID classid:(cookingstepid) attrname:"requires ingredient quantities"
SET tableattributeid:(attrid)
CC "Core" GET_REC_CLASS_ID attrid:(tableattributeid)
SET tableclassid:(classid)
CC "Core" GET_ATTR_ID classid:(tableclassid) attrname:"Ingredient"
SET ingredientcolumnid:(attrid)
CC "Modeling" GET_SELECTED modelid:(modelid)
FOR i in:(objids)
{
        CC "Core" GET_CLASS_ID objid:(VAL i)
        SET selectedclassid:(classid)
        IF (selectedclassid=cookingstepid)
                {
                CC "Core" GET_ATTR_VAL objid:(VAL i) attrname:"Cost"
                CC "Core" SET_ATTR_VAL objid:(VAL i) attrname:"Cost" val:(val+1)
                CC "Core" ADD_REC_ROW objid:(VAL i) attrid:(tableattributeid)
                CC "Core" ADD_INTERREF objid:(rowid) attrid:(ingredientcolumnid) tmodelname:"Cooking Resources - new"
tmodeltype:"Cooking Resources" tclassname:"CookingIngredient" tobjname:"Salt"
                CC "Core" SET_ATTR_VAL objid:(rowid) attrname:"Quantity" val:100
                }
        ELSE
                {
                CC "AdoScript" INFOBOX ("object "+i+" is not a cooking step, nothing will happen to it")
                }
}
CC "AdoScript" INFOBOX "costs are now incremented"
```

Scripts that are triggered by UI events

Until now, we created model-based functionality that was triggered by menu items. Another possibility is to trigger functionality with events. Similarly to how we program event handlers in JavaScript, we can also do it here. The difference is that instead of having browser events (onclick, ondblclick etc.), in ADOScript we work with special events that are customized for the modelling canvas, such as "AfterCreateModelingConnector" which corresponds to the moment when we end the drawing of an "arrow" between two "nodes" (i.e., the last click of the connector creation). The complete list of events can be consulted in the EVENTS chapter of the documentation (https://www.adoxx.org/AdoScriptDoc/).

Go to the Development Toolkit, in the External Coupling attribute. After the menu item definitions include the following code:

```
ON_EVENT "AfterCreateModelingConnector"
{
#you have implicit access to several predefined variables regarding the environment where the event was triggered:
# modelid,objid,classid,fromobjid,toobjid.
SET message:""
CC "Core" GET_CLASS_NAME classid:(classid)
SET message:(message+"You created a connector of type "+classname+" between two objects:\n")
CC "Core" GET_CLASS_ID objid:(fromobjid)
CC "Core" GET_CLASS_NAME classid:(classid)
CC "Core" GET_OBJ_NAME objid:(fromobjid)
SET message:(message+"The source object "+objname+" is instance of the concept "+classname+"\n")
CC "Core" GET_CLASS_ID objid:(toobjid)
CC "Core" GET_CLASS_NAME classid:(classid)
CC "Core" GET_OBJ_NAME objid:(toobjid)
SET message:(message+"the target object "+objname+" is instance of the concept "+classname+"\n")

CC "AdoScript" INFOBOX (message)
}
```

Notice that you have access to some predefined variables that give you access to information about the location where you triggered the event. In this case, you can access the model where the event was triggered (modelid), the connector identifier (objid), the source object (fromobjid), the target object (toobj), the class of the connector (classid). Each event has such predefined contextual data, which may be consulted in the documentation of the event.

The script displays a message indicating three aspects:

- The type (class) of the created connector;
- The name and type of the source node (where the connector starts);
- The name and type of the target node (where the connector ends).

In order to test it you have to restart the modelling toolkit.

Because we have some repeated code, we may prefer to define a procedure. In the procedure version, the event handler looks like this:

```
ON_EVENT "AfterCreateModelingConnector"
{
PROCEDURE GET_INFO someinput:integer someoutput:reference
  {
  CC "Core" GET_CLASS_ID objid:(someinput)
  CC "Core" GET_CLASS_NAME classid:(classid)
  CC "Core" GET_OBJ_NAME objid:(someinput)
  SET someoutput:(someoutput+"The source object "+objname+" is instance of the concept "+classname+"\n")
  }

CC "Core" GET_CLASS_NAME classid:(classid)
SET message:("You created a connector of type "+classname+" between two objects:\n")
GET_INFO someinput:(fromobjid) someoutput:message
GET_INFO someinput:(toobjid) someoutput:message
CC "AdoScript" INFOBOX (message)
}
```

Notice how the repeated code is structure in a procedure named GET_INFO, which has two parameters:

- someinput, declared as integer, will be read by the procedure as input argument; it is the object ID for which a message must be composed; notice:
  - how it is used in the procedure call (it gets values from fromobjid and toobjid)
  - how it is used inside the procedure (to pass relevant values to objid internally)
- someoutput, declared as a reference (pointer), will be written by the procedure (it acts as a "returned value"); notice:
  - how it is used in the call (without parentheses, to indicate by reference in which variable should the value be overwritten by the procedure)
  - how it is used inside the procedure (in a SET statement which overwrites its value)

It is also possible to program the event in a separate .asc file, as we previously did to avoid switching back and forth between the Development toolkit and the Modelling toolkit. However, in the case of events it is a bit more complicated than just reading script contents from the external scripting file.

First go to the Development toolkit, the External coupling attribute. Replace the event handling script with the following:

```
ON_EVENT "AfterCreateModelingConnector"
```

```
{
SET predefined:("SET from:("+(STR fromobjid)+")\n")
SET predefined:(predefined+"SET to:("+(STR toobjid)+")\n")
SET predefined:(predefined+"SET class:("+(STR classid)+")\n")
CC "AdoScript" FREAD file:("C:\\myscripts\\mycode3.asc ")
EXECUTE (predefined+text)
}
```

Notice that in the case of events, the script file contents (mycode3.asc) must be concatenated with several initializations, for all the values that must be read from the predefined variables. In this case, we initialize the variable **from** (taking the value from the predefined fromobjid), the variable **to** (value read from toobjid) and the variable **class** (value from classid). This is necessary because the external script file does not have access to the predefined variables generated by the event! (so we have to read those values in External coupling and then cocatenate them into the script contents –m see the EXECUTE line)

This means that the external file (mycode3.asc) will work with these initialized variables and not with the predefined ones:

```
SET message:""
CC "Core" GET_CLASS_NAME classid:(class)
SET message:(message+"You created a connector of type "+classname+" between two objects:\n")
CC "Core" GET_CLASS_ID objid:(from)
CC "Core" GET_CLASS_NAME classid:(classid)
CC "Core" GET_OBJ_NAME objid:(from)
SET message:(message+"The source object "+objname+" is instance of the concept "+classname+"\n")
CC "Core" GET_CLASS_ID objid:(to)
CC "Core" GET_CLASS_NAME classid:(classid)
CC "Core" GET_OBJ_NAME objid:(to)
SET message:(message+"the target object "+objname+" is instance of the concept "+classname+"\n")

CC "AdoScript" INFOBOX (message)
```

In the next example we will generate a log that writes down, for each creation of a model element (node or connector) a sentence describing the element that was created. Create the empty log file C:\log\log.txt

Now go to Development Toolkit, in External coupling and add the following event handlers:

```
ON_EVENT "AfterCreateModelingNode"
  {
  CC "Modeling" GET_ACT_MODEL
  CC "Core" GET_MODEL_INFO modelid:(modelid)
  CC "Core" GET_CLASS_NAME classid:(classid)
  CC "Core" GET_OBJ_NAME objid:(objid)
  CC "AdoScript" FWRITE file:"C:\\log\\log.txt" text:("In model "+modelname+" you created the object "+objname+" of type
"+classname+"\n") append:yes
  }


ON_EVENT "AfterCreateModelingConnector"
  {
  CC "Modeling" GET_ACT_MODEL
  CC "Core" GET_MODEL_INFO modelid:(modelid)
  CC "Core" GET_CLASS_NAME classid:(classid)
  CC "Core" GET_OBJ_NAME objid:(fromobjid)
  SET sourcename:(objname)
  CC "Core" GET_OBJ_NAME objid:(toobjid)
  SET targetname:(objname)
  CC "AdoScript" FWRITE file:"C:\\log\\log.txt" text:("In model "+modelname+" you created a connector of type "+classname+ " from object
"+sourcename+" to object "+targetname+"\n") append:yes
  }
```

Notice that each event handler collects some contextual information from the predefined variables of the event (current model, current object, source and target of the connector). The names are extracted for each of these IDs, then a message is composed and appended to the text file (notice the append:yes option; otherwise, the text will be replaced).

Restart the modelling toolkit to have the events activated, then create objects or connectors in any model type. After that, go check the text file to see the logged operations.

Scripts that generate model elements

AdoScript also has the ability to create models dynamically out of information provided by the user, or out of other models. Add a new menu item (in the Development toolkit, External coupling):

```
ITEM "Generate model elements" modeling:"My Second Menu"
CC "AdoScript" FREAD file:(" C:\\myscripts\\mycode4.asc")
EXECUTE (text)
```

Create the next script in mycode4.asc, saved as indicated in the code above:

```
CC "Modeling" GET_ACT_MODEL
CC "AdoScript" EDITBOX text:("") title:("Enter some names, one on each row") oktext:("Create")
SET names:(text)
CC "Core" GET_CLASS_ID classname:"CookingStep"
SET stepclassid:(classid)
CC "Core" GET_CLASS_ID classname:"followed By"
SET arrowclassid:(classid)

SET nr:0
FOR i in:(names) sep:("\n")
{
IF (LEN i > 1)
        {
        CC "Core" CREATE_OBJ modelid:(modelid) classid:(stepclassid) objname:(i)
        SET newobj:(objid)
        CC "Modeling" SET_OBJ_POS objid:(newobj) x:(2cm) y:(2cm+CM (5 * nr))
        IF (nr>0)
                {
                CC "Core" CREATE_CONNECTOR modelid:(modelid) fromobjid:(previousobj) toobjid:(newobj) classid:(arrowclassid)
                }
        SET nr:(nr+1)
        SET previousobj:(newobj)
        }
}
```

The effect of the new menu item is the following:

- an EDITBOX is displayed, where the user is asked to type several string values, one on each line;
- the script will generate one CookingStep object for each line that was typed; each line will become the name of a generated object; each object will be positioned under the previous one, with a 2cm distance;
- starting with the second object, the script will also generate a "followed By" connector between the current object and the previous one; in order to accomplish this we need to keep count of how many objects are being generated (nr), as well as to distinguish between the current object id (newobj) and the previous object id (previousobj)

In the following, extend the code to have an additional feature:

- on each line, the user should type the name and the cost of a CookingStep, separated by a comma; when generated, the objects will also get their costs from this input;
- notice in the code how, for each line, we use the token() function to separate the name from the cost; the new lines are bolded (other string functions may be consulted in the Help):

```
CC "Modeling" GET_ACT_MODEL
CC "AdoScript" EDITBOX text:("") title:("Enter one name and one corresponding cost on each row. Separate name and cost with a
comma") oktext:("Create")
SET data:(text)
CC "Core" GET_CLASS_ID classname:"CookingStep"
SET stepclassid:(classid)
CC "Core" GET_ATTR_ID classid:(stepclassid) attrname:"Cost"
CC "Core" GET_CLASS_ID classname:"followed By"
SET arrowclassid:(classid)

SET nr:0
FOR i in:(data) sep:("\n")
{
IF (LEN i > 1)
        {
        SET name:(token(i,0,","))
        SET cost:(VAL token(i,1,","))
        CC "Core" CREATE_OBJ modelid:(modelid) classid:(stepclassid) objname:(name)
        SET newobj:(objid)
        CC "Modeling" SET_OBJ_POS objid:(newobj) x:(2cm) y:(2cm+CM (5 * nr))
        CC "Core" SET_ATTR_VAL objid:(newobj) attrid:(attrid) val:(cost)
        IF (nr>0)
                {
                CC "Core" CREATE_CONNECTOR modelid:(modelid) fromobjid:(previousobj) toobjid:(newobj) classid:(arrowclassid)
                }
        SET nr:(nr+1)
        SET previousobj:(newobj)
        }
}
```

This technique is especially important for deserialization, when models must be generated from some textual input (possibly from an external file containing diagram descriptions structured in some textual form). The reverse of this technique is "code generation", when for certain objects and attributes present in a model, certain patterns of source code are written in a text file (remember the example with logging).

Scripts that are triggered by the context menu (right-click)

We have seen until now how we can trigger new functionality through (a) menu options and (b) events. A third way of triggering functionality is through the right-click context menu. This should be combined with the event "AppInitialized", to make sure that the customized right-click menu is available from the very start (of course, we may also activate such customizations with other events).

Go the Development Toolkit, in the External coupling, where there is already an empty placeholder for the event handler AppInitialized. Fill the empty handler as shown below:

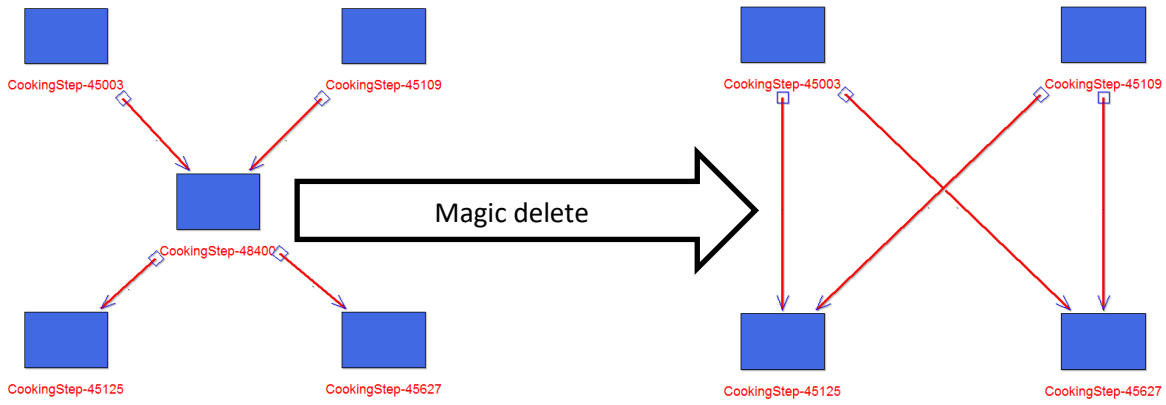
```
ON_EVENT "AppInitialized"
{
CC "Application" INSERT_CONTEXT_MENU_ITEM context:"drawingarea.mobject" item:"My magic delete"
CC "Application" raw SET_CMI_SELECT_HDL context:"drawingarea.mobject" item:"My magic delete"
  {
  CC "AdoScript" FREAD file:("C:\\myscripts\\mycode5.asc")
  EXECUTE (text)
```

```
    }
}
```

Notice:

- INSERT_CONTEXT_MENU_ITEM defines the new option label (item) and the "limited context" in which it should be made available in the right-click menu. The context drawingarea.mobject means that the new option will be active only when right-clicking on a modelling object (node, not on connectors and not on other areas of the user interface; check the documentation of INSERT_CONTEXT_MENU_ITEM to see other such "limited contexts");
- SET_CMI_SELECT_HDL defines the script that should be executed when selecting the new option. Notice that, as before, we externalized the code to a script file mycode5.asc (it is recommended to include keyword *raw* before this function, otherwise there is a risk that some predefined variables will not be accessible.

The script is listed below. It provides an alternative "magic delete" option. When this is used instead of the regular "delete", all the connectors going to the deleted element will be automatically reconnected with any other elements towards which the deleted one had outgoing connectors. Explanations are included in the code as comments.




```
CC "Modeling" GET_ACT_MODEL
CC "Modeling" GET_SELECTED
SET currentobject:(token(objids,0," "))
# beware, GET_SELECTED returns a list of objids, allowing multiple objects to be selected with Ctrl-click
# however, we want to apply the magic delete on a single object, so in case of multiselection only the first will be deleted
# for this reason, the token function extracts the first id from the list of objids

CC "Core" GET_CONNECTORS objid:(VAL currentobject) in
# we get all connectors going in the selected object; notice the "in" parameter
SET sources:({})
# we initialize an array to collect the sources for all incoming connectors
FOR i in:(objids)
        {
        CC "Core" GET_CONNECTOR_ENDPOINTS objid:(VAL i)
        # we get the source endpoint of the connector, made available in the predefined variable fromobjid
        SET newsource:(aappend(sources,fromobjid))
        # we add the detected source to the array of sources
```

```
        }

CC "Core" GET_CONNECTORS objid:(VAL currentobject) out
# we get all connectors outgoing from the selected object; notice the "out" parameter

SET targets:({})
# we initialize an array of targets for all outgoing connectors
FOR i in:(objids)
        {
        CC "Core" GET_CONNECTOR_ENDPOINTS objid:(VAL i)
        SET newtarget:(aappend(targets,toobjid))
        # we get the targets of all outgoing connectors and add them to an array of targets
        }

CC "Core" GET_CLASS_ID relation classname:"followed By"
FOR i from:0 to:(sources.length-1)
        {
        FOR j from:0 to:(targets.length-1)
                {
                CC "Core" CREATE_CONNECTOR modelid:(modelid) fromobjid:(sources[i]) toobjid:(targets[j]) classid:(classid)
                }
        }
# we loop through all the sources and all the targets and create direct "followed  By" connectors for each pair

CC "Core" DELETE_OBJS modelid:(modelid) objids:(currentobject)
# finally, we deleted the selected object
```

Scripts that use model queries

We may create more efficient code if, instead of looping through various object lists we use AQL queries. These are actual **model queries** that can retrieve objects meeting certain complex conditions. Consequently, they can hide a lot of processing effort in simple expressions.

We can learn AQL queries in two ways:

- through its syntax documentation: https://www.adoxx.org/live/adoxx-query-language-aql
- by using the query builder in the Modelling toolkit.

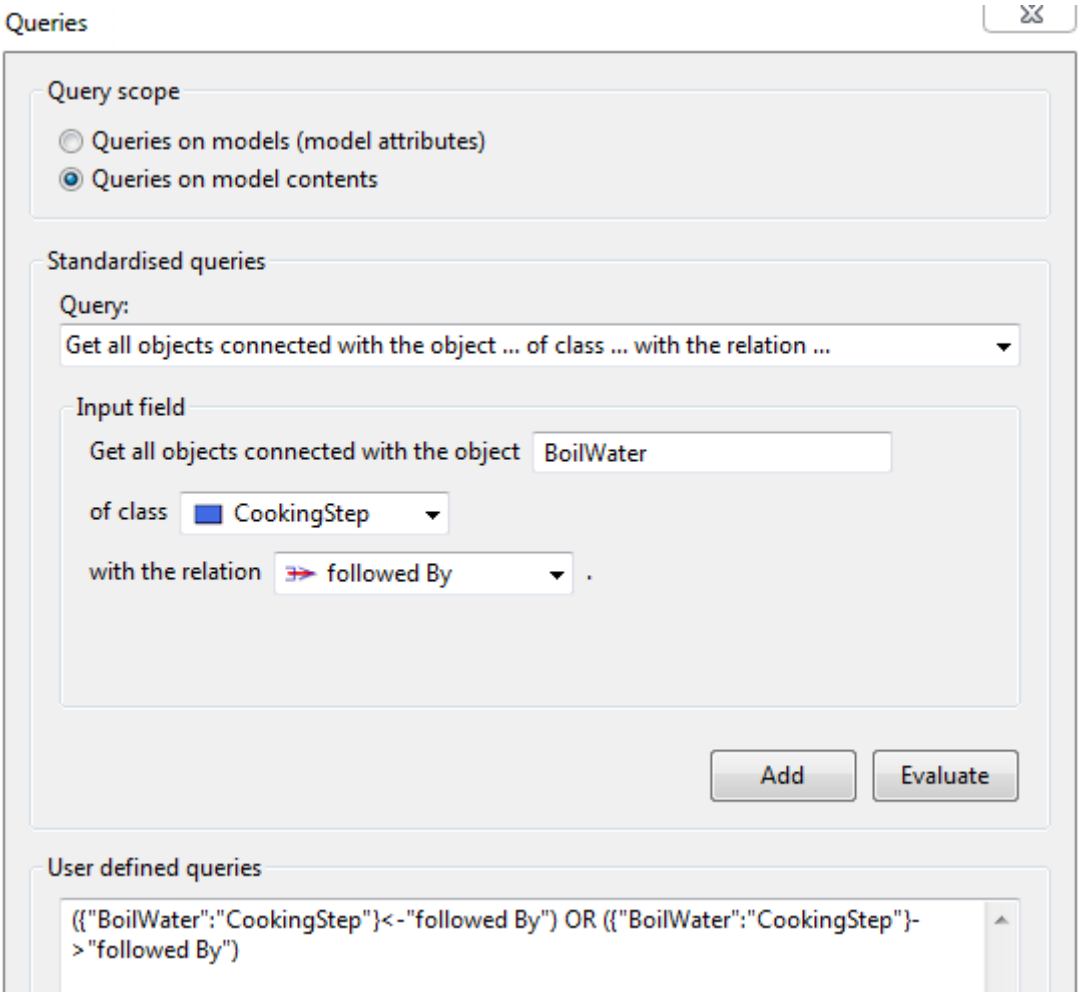
We already used the query builder during the definition of the cooking language syntax. First you have to activate the Analysis menu bar (the top left buttons), then access Analysis-Queries/Reports and select the models you want to query.

Assuming that you have a "BoilWater" cooking step in your model, and you want to retrieve all objects connected to it, you will build the query by using the standardised template "Get all objects connected with the object x.....". Fill all the slots with the settings visible in the next figure and press Add to see the actual AQL code of the query:

Since the AQL contains an OR you may guess that the first half, for example, retrieves all objects FROM which there are incoming connectors to BoilWater:

```
({"BoilWater":"CookingStep"}<-"followed By")
```

The other half retrieves the objects TO which there are outgoing connectors.



You can play around with the query builder to see how other queries look like when written in AQL:

- (({"BoilWater":"CookingStep"}<-"followed By")<-"followed By")
  (all objects that are two incoming connections away from BoilWater)
- ({"BoilWater"}<<-"followed By")
  (all objects from which connections exist to BoilWater, regardless of how far they are)
- ({"BoilWater"}<-"followed By")
  (all objects from which incoming connections to BoilWater exist, without stating the BoilWater type)
- ({"BoilWater"}<-"followed By">"CookingStep"<)
  (all objects from which incoming connections to BoilWater exist, the results must be of type CookingStep)
- (<"CookingStep">)
  (all objects of type CookingStep)
- (<"followed By">[?"Condition" = "yes"])
  (all connections of type followedBy, which have the Condition attribute with a value of "yes")
- ({"BoilWater"}-->"requires ingredients")
  (targets of the "requires ingredients" hyperlink of the BoilWater object)
- (<"CookingStep">[?"requires ingredient quantities"][?"Quantity" = 8])
  (all cooking steps that have, in the table attribute "requires ingredient quantities", a row for which the quantity is 8

As a general rule, the following key delimiters are involved here:

- {...} indicates a specific object
- <...> indicates a specific class (or model) to be selected; if inverted >...< it uses the class as a filter
- [...] indicates a filter condition applied on attributes; if the attribute is a table (record class), then an additional [...] can impose a condition on the columns of the table
- -> filters by outgoing connectors
- <- filters by incoming connectors
- ->> and <<- filter by connector chains of arbitrary length
- --> and <-- filter by incoming or outgoing hyperlinks
- -->> and <<-- filter by hyperlink chains of arbitrary length
- : is used as a qualification delimiter (usually between an object and its class; it's also possible to use it between a class and its model and between a model and its model type if the object is not in the queried model)

All these queries may be executed in AdoScript with **EVAL_AQL_EXPRESSION.** Usually it is a shorter way of retrieving objects that meet certain conditions. However, keep in mind that:

- AQL queries take as input names, not ids (object names, class names, model names);
- When included as a string in AdoScript, quotation marks must be escaped with backslash;
- The output of AQL queries are objids

Once the queries are confirmed to work well in the modelling toolkit, we can include them in the code of our previous script. It will execute two queries: one for sources of incoming connectors and one for targets of outgoing connectors. Compare the code below with the previous version and notice that we got rid of 2 of those FOR loops. We only keep the loop that creates the reconnections.

```
CC "Modeling" GET_ACT_MODEL
CC "Modeling" GET_SELECTED
SET currentobject:(token(objids,0," "))
CC "Core" GET_OBJ_NAME objid:(VAL currentobject)

SET queryincoming:("({\""+objname+"\"}<-\"followed By\")")
CC "AQL" debug EVAL_AQL_EXPRESSION expr:(queryincoming) modelid:(modelid)
SET sources:(objids)
CC "AdoScript" INFOBOX  (VAL sources)

SET queryoutgoing:("({\""+objname+"\"}->\"followed By\")")
CC "AQL" EVAL_AQL_EXPRESSION expr:(queryoutgoing) modelid:(modelid)
SET targets:(objids)

CC "Core" GET_CLASS_ID relation classname:"followed By"
FOR i in:(sources)
        {
        FOR j in:(targets)
                {
                CC "Core" CREATE_CONNECTOR modelid:(modelid) fromobjid:(VAL i) toobjid:(VAL j) classid:(classid)
                }
        }
CC "Core" DELETE_OBJS modelid:(modelid) objids:(currentobject)
```

Notice that:

- queries rely on object names, therefore we have to get the name of the selected object;

- queries contain some quotation marks, which must be escaped by backslash

The next example is a "process stepper". You have to select the Start element in a Cooking Recipe, then the script will step through each element and will display its object id. When it reaches the end it displays a message ("Process finished"). If you don't start by selecting the Start element, it will display an error message.

The example assumes that you don't allow multiple arrows to go out of an element. If you want to allow ramifications, then you need to add additional logic to choose the connector on which to advance through the process.

```
CC "Modeling" GET_ACT_MODEL
CC "Modeling" GET_SELECTED modelid:(modelid)
SET currentobject:(VAL objids)
CC "Core" GET_CLASS_ID objid:(currentobject)
CC "Core" GET_CLASS_NAME classid:(classid)
IF (NOT(classname="Start"))
	{
	CC "AdoScript" INFOBOX ("This is not a Start element, it is a "+classname)
	}
ELSE
	{
	SET flag:1
	WHILE (flag=1)
		{
		CC "Core" GET_CONNECTORS objid:(currentobject) out
		SET currentoutconnector:(VAL objids)
#this line assumed that only one connector goes out of an element (if you make a ramification only one connector will be used)
		CC "Modeling" DYE (currentoutconnector) make-visible
		CC "Core" GET_CONNECTOR_ENDPOINTS objid:(currentoutconnector)
		SET currentobject:(toobjid)
		CC "Core" GET_CLASS_ID objid:(currentobject)
		CC "Core" GET_CLASS_NAME classid:(classid)
		IF (classname="CookingStep")
			{
			CC "Modeling" DYE (currentobject) make-visible
			CC "AdoScript" INFOBOX ("You reached the step "+(STR currentobject))
			}
		ELSIF (classname="Stop")
			{
			CC "AdoScript" INFOBOX ("Process finished")
			CC "Modeling" UNDYE_ALL modelid:(modelid)
			SET flag:0
			}
		}
	}
```

A final remark: when deploying a modelling tool to its end-users, we cannot provide the .asc scripting code as separate files! All the .asc code files must be imported in the Development Toolkit in the same way that we imported custom pictures in the first part of the tutorial (for the CookingStep notation). Then, as already shown, any external files that are imported in ADOxx must be referenced in the FREAD command through a path such as "db:\\myscript.asc".