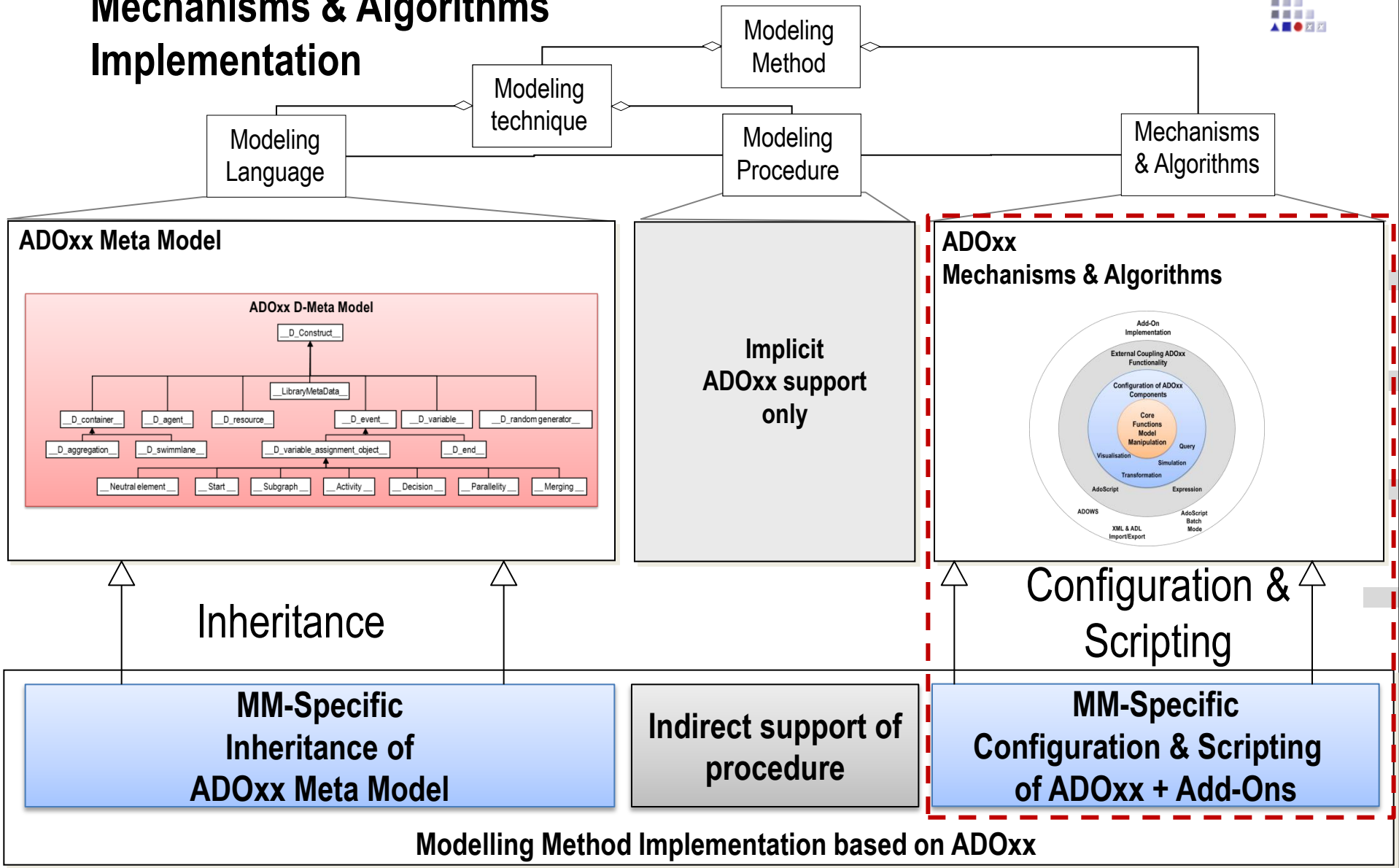
MECHANISMS & ALGORITHMS IMPLEMENTATION ON ADOxx

What is ADO*xx*?

"ADO*xx* IS A META MODELLING DEVELOPMENT AND CONFIGURATION PLATFORM FOR IMPLEMENTING MODELLING METHODS."

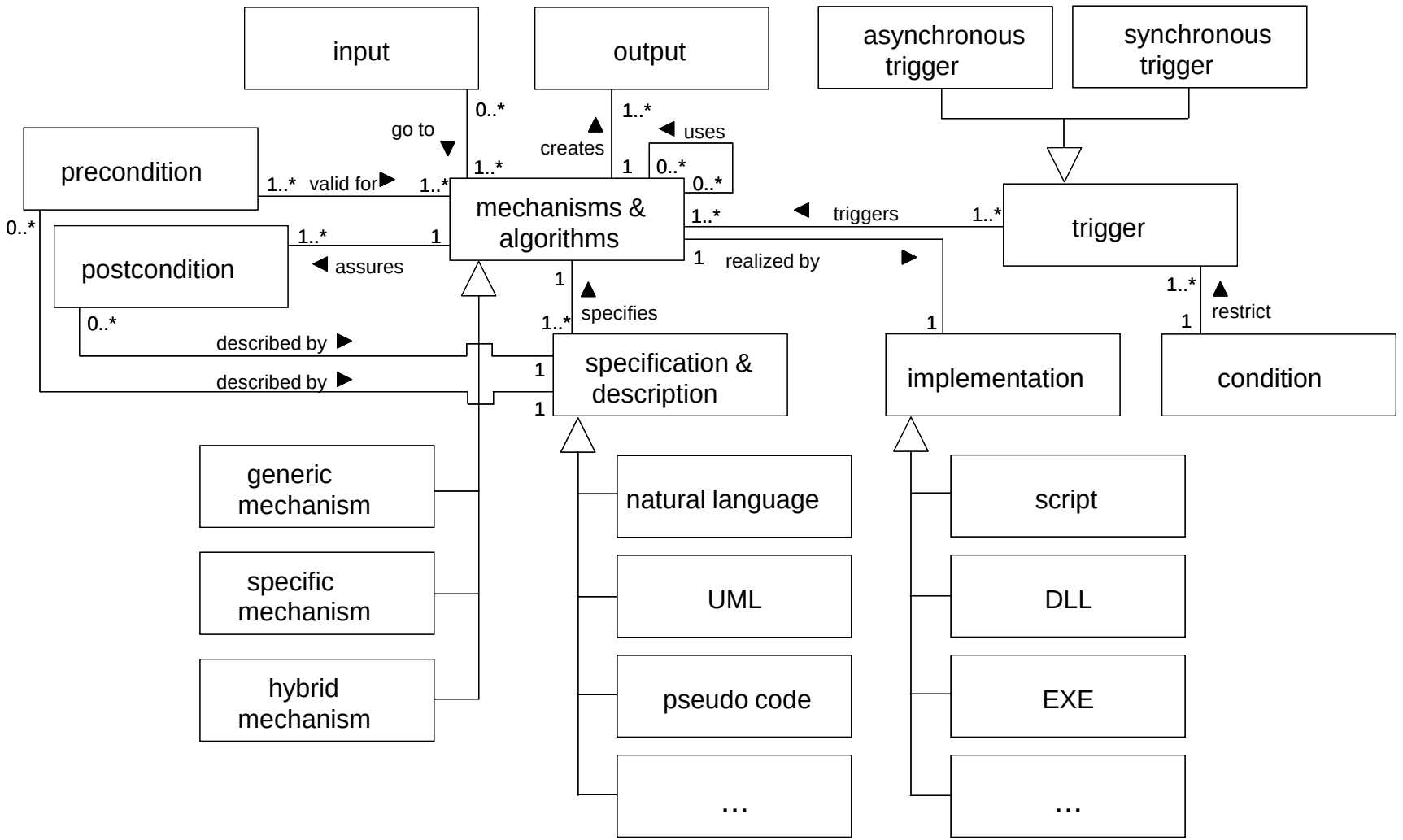
MECHANISMS & ALGORITHMS IMPLEMENTATION

Mechanisms & Algorithms Implementation

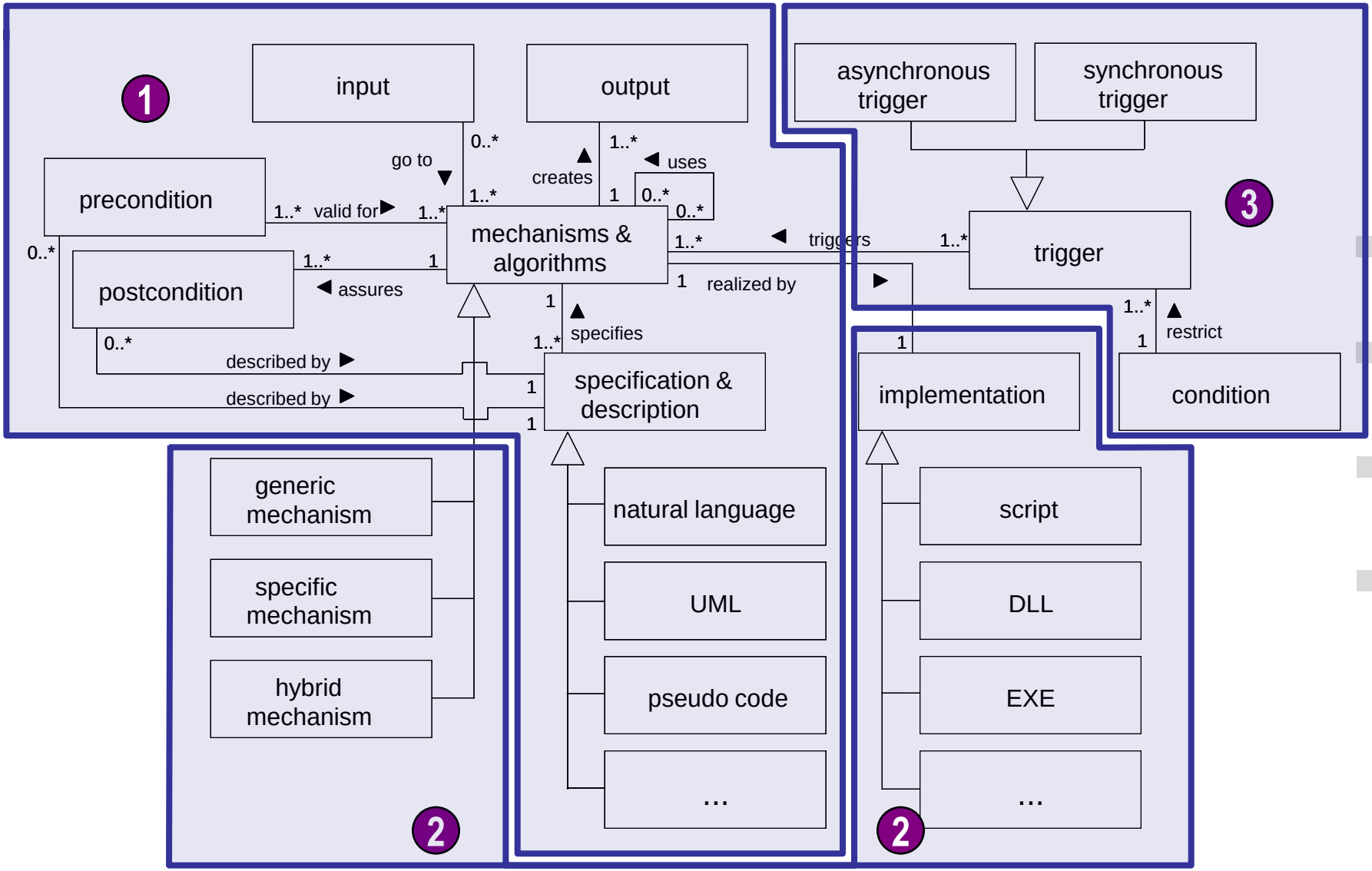


Reference: Kühn, H. (2004). Methodenintegration im Business Engineering. PhD Thesis, University of Vienna

Meta Model of Mechanisms & Algorithms



Meta Model of Meta Modelling Language

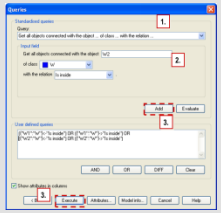


DEVELOPMENT APPROACHES

Configuration and Implementation Approach

"Interactive Style"

Configuration Approach

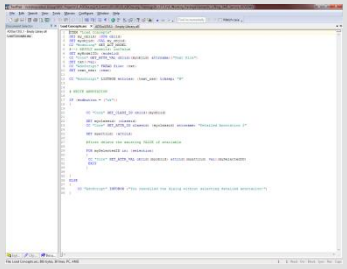


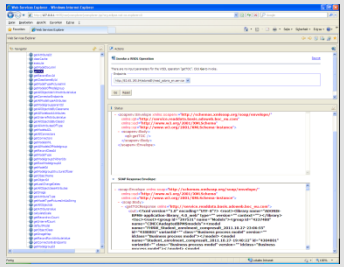
ADOxx Configuration Tools

AQL Query Editor | Simulation Configuration | Documentation Configuration | External coupling

Metamodel Engineer

Implementation Approach





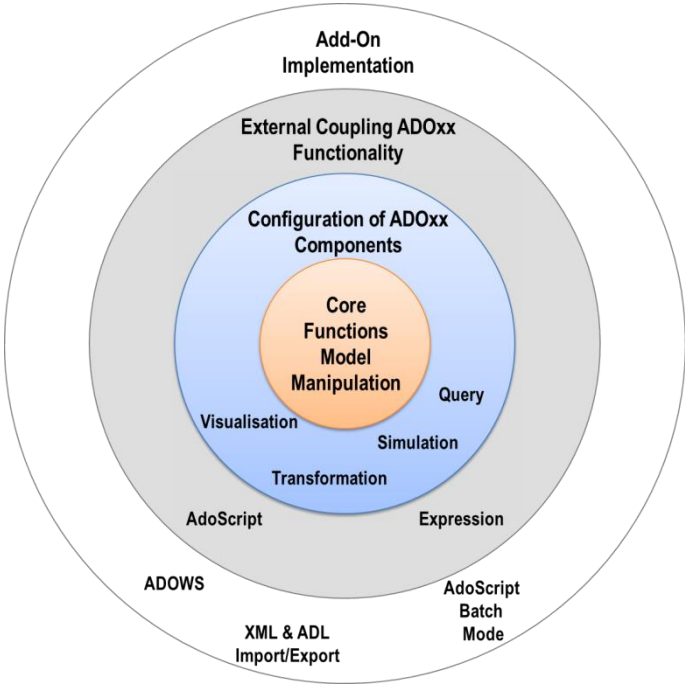
ADOxx Library Language Implementation Tools

Syntax Highlight AutoComplete | Supporting Development Services

„Programming Style"

ABL

Library Configuration/ Implementation



1. CORE FUNCTIONS FOR MODEL MANIPULATION

Core Functions for Model Manipulation

Data Base

Visualisation

Query

Transformation

Simulation

1. CORE FUNCTIONS FOR MODEL MANIPULATION

DATA BASE

Core Functionality

- **User Management**
  create, delete, define access rights
- **Model Group Management**
  create, delete, rename
- **Model Management**
  create, delete, rename, version
- **Library Management**
  import/export, check,
  manipulate
- **Component Management**
  switch on / off components
- **Efficient Storage**
  user, model groups, models and libraries

1. CORE FUNCTIONS FOR MODEL MANIPULATION

VISUALISATION

Graphical Representation

- Graphical Presentation of models in the user interface
- Drag and Drop: Creation and Move, Delete, Edit
- Cardinality conformity check
- Notebook representation
- Grid visualisation, Snap Grid
- Generation of graphic files (bmp, jpg, png, etc.)
- Zoom Functionality (zoom, world-area, right mouse, etc.)
- Table based representation
- Printer Functionality
- Page Layout
- Connector behaviour

1. CORE FUNCTIONS FOR MODEL MANIPULATION

QUERY

Platform basic functionalities in the analysis component

Standardized queries:

Standardized queries as „to complete text“, which are completed by the user. For execution, no AQL knowledge is required.

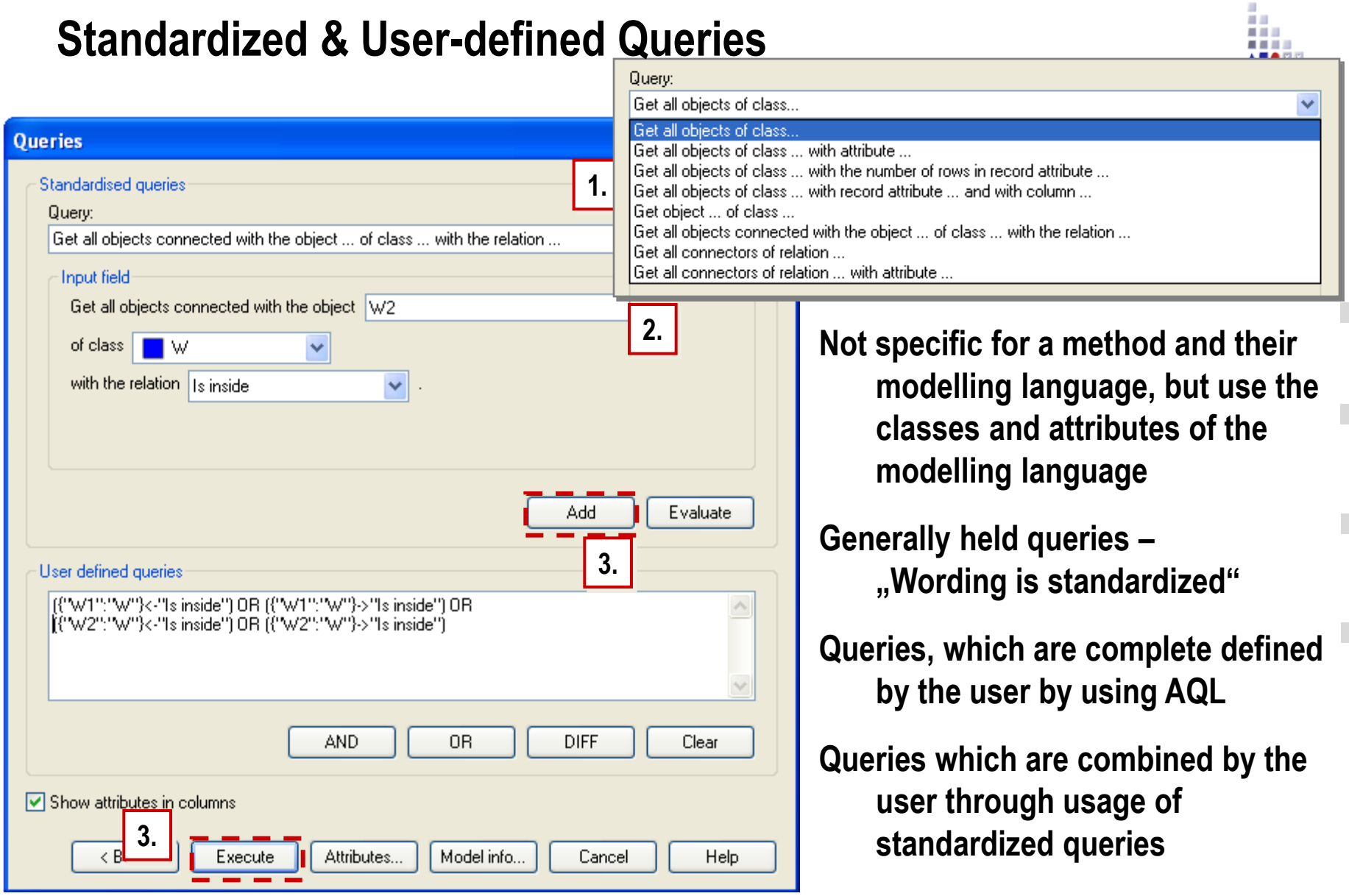
User-defined queries:

Queries which are defined by the user through standardized queries in AQL syntax. For execution AQL knowledge is required.

15

[1] **AQL** = **A**DOxx **Q**uery **L**anguage

Standardized & User-defined Queries



Not specific for a method and their modelling language, but use the classes and attributes of the modelling language

Generally held queries – „Wording is standardized“

Queries, which are complete defined by the user by using AQL

Queries which are combined by the user through usage of standardized queries

Session dependent, regarding of evaluation parameters

1. CORE FUNCTIONS FOR MODEL MANIPULATION

b. TRANSFORMATION

Standard Export / Import

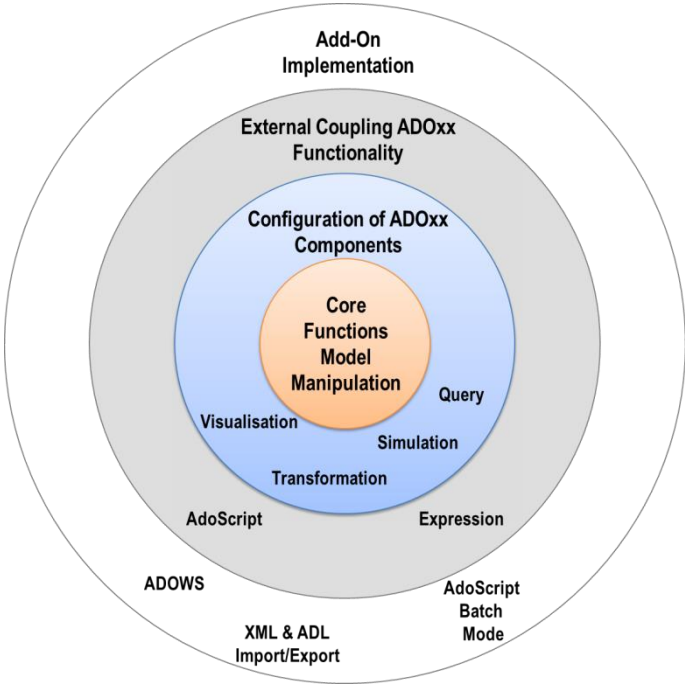
- **Generation of ADL**
  Text file in complimentary ADOxx Definition Language
- **Generation of XML**
  Text file in complimentary ADOxx defined XML syntax

XML Export Sample

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE ADOXML (View Source for full doctype...)>
- <ADOXML version="3.1" date="28.06.2012" time="13:32" database="adoxx13" username="sample1" adoversion="Version 1.0">
  - <MODELS>
    - <MODEL id="mod.13813" name="model-1" version="1.1" modeltype="Sample" libtype="bp" applib="ADOxx 1.3 Dynamic Experimentation Library - START">
      + <MODELATTRIBUTES>
      - <INSTANCE id="obj.13814" class="E" name="E1">
          <ATTRIBUTE name="Position" type="STRING">NODE x:4cm y:11cm w:2cm h:2cm index:1</ATTRIBUTE>
          <ATTRIBUTE name="External tool coupling" type="STRING" />
          <ATTRIBUTE name="a1" type="INTEGER">0</ATTRIBUTE>
          <RECORD name="a2" />
          <ATTRIBUTE name="a3" type="STRING" />
          <ATTRIBUTE name="b1" type="INTEGER">0</ATTRIBUTE>
          <RECORD name="b2" />
          <ATTRIBUTE name="b3" type="STRING" />
          <ATTRIBUTE name="e1" type="INTEGER">0</ATTRIBUTE>
          <RECORD name="e2" />
          <ATTRIBUTE name="e3" type="STRING">11</ATTRIBUTE>
          <ATTRIBUTE name="a4" type="INTEGER">0</ATTRIBUTE>
          <ATTRIBUTE name="b4" type="STRING" />
        </INSTANCE>
      + <INSTANCE id="obj.13817" class="A" name="A1">
      + <INSTANCE id="obj.13826" class="B" name="B1">
      + <INSTANCE id="obj.13832" class="C" name="C-13010">
      + <INSTANCE id="obj.13835" class="D" name="D-13013">
      + <INSTANCE id="obj.16408" class="B" name="B-16408">
      + <INSTANCE id="obj.16604" class="V" name="V1">
      + <INSTANCE id="obj.17004" class="W" name="W1">
      + <INSTANCE id="obj.17007" class="B" name="B-16408-17007">
      + <INSTANCE id="obj.17291" class="E" name="E-17291">
      + <INSTANCE id="obj.17294" class="E" name="E-17294">
      + <INSTANCE id="obj.17297" class="E" name="E-17297">
      + <INSTANCE id="obj.17328" class="E" name="D-13013-17321">
      + <INSTANCE id="obj.17334" class="E" name="C-13010-17318">
      + <CONNECTOR id="con.13841" class="aRb">
      + <CONNECTOR id="con.13842" class="aRb">
      + <CONNECTOR id="con.13843" class="aRb">
      + <CONNECTOR id="con.13844" class="aRb">
      + <CONNECTOR id="con.13845" class="aRb">
      + <CONNECTOR id="con.16607" class="Is inside">
      </MODEL>
    </MODELS>
</ADOXML>
```

ADL Export Sample

```
INSTANCE <E1> : <E>
    ATTRIBUTE <Position>
    VALUE "NODE x:4cm y:11cm w:2cm h:2cm index:1"
    ATTRIBUTE <External tool coupling>
    VALUE ""
    ATTRIBUTE <a1>
    VALUE 0
    ATTRIBUTE <a2>
    VALUE
    ATTRIBUTE <a3>
    VALUE ""
    ATTRIBUTE <b1>
    VALUE 0
    ATTRIBUTE <b2>
    VALUE
    ATTRIBUTE <b3>
    VALUE ""
    ATTRIBUTE <e1>
    VALUE 0
    ATTRIBUTE <e2>
    VALUE
```



2. CONFIGURATION OF ADOxx COMPONENTS

2. CONFIGURATION OF ADOXX COMPONENTS

VISUALISATION

Attribute Dependent Graphical Notation

```
GRAPHREP layer:-1 sizing:asymmetrical

# select type for color
AVAL atype:"Type-Selection"

# set default color
SET f:"white"

IF (atype = "type-1")
  SET f:"blue"          # if type 1 selected => color is blue
ELSIF (atype = "type-2")
  SET f:"yellow"        # if type 2 selected => color is yellow
ELSIF (atype = "type-3")
  SET f:"red"           # if type 3 selected => color is red
ENDIF

FILL color:(f)          # set color with variable

# drawing main box
RECTANGLE x:-3cm y:-3cm w:6cm h:6cm
ATTR "V-Text" x:-0cm y:-2.5cm w:c h:t # type text from atribute "V-Text"
```

2. CONFIGURATION OF ADOXX COMPONENTS

QUERY

Platform basic functionalities in the analysis component

Predefined queries:

Professional queries, which are business or method specific defined. For the execution of these, no AQL knowledge required.

Relation tables:

Relation tables make relations (connectors or references) between two objects of same or different models.

[1] **AQL** = **A**DOxx **Q**uery **L**anguage

Predefined queries are…

- Queries on models, model content and dependencies of models
- Session-independent
- Defined by the user and therefore easy to use
- Specific for the used model and their modelling language
- A configuration of the basic functionalities of the ADOxx Platform. These queries can:
  - Be defined for models of a specific type
  - Combined into groups (topics)

Creation of method-specific Queries

Steps which are necessary:

1. Define menu item
2. Define input fields
3. Define AQL queries
4. Define result attributes

Start Analysis Query Function

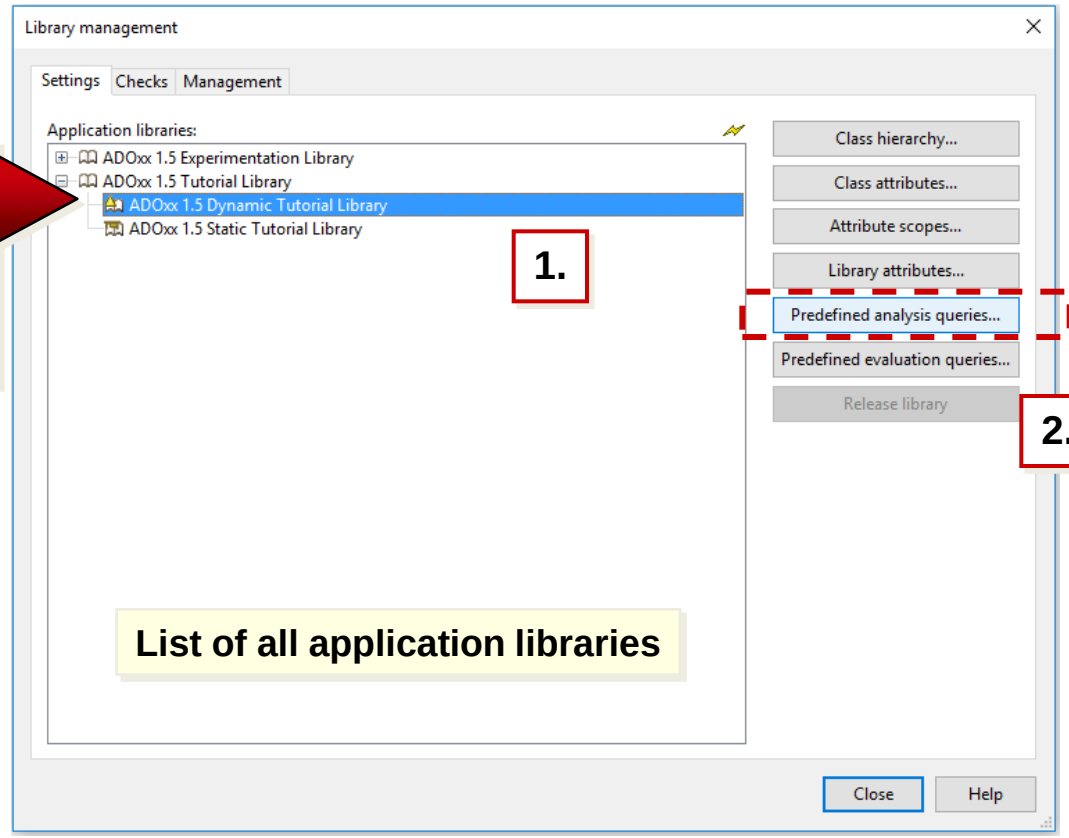
Choose Library
(D- or S-Library)

„Predefined analysis queries“

- SmartIcon

or

- Menu **„Library“**
- Menu item **„Configuration“**



In order to edit the analysis queries you can use selection dialogs in the library management

Query Functions

Available functions:

New: Create a new menu item or a new query

Edit: Edit an existing query

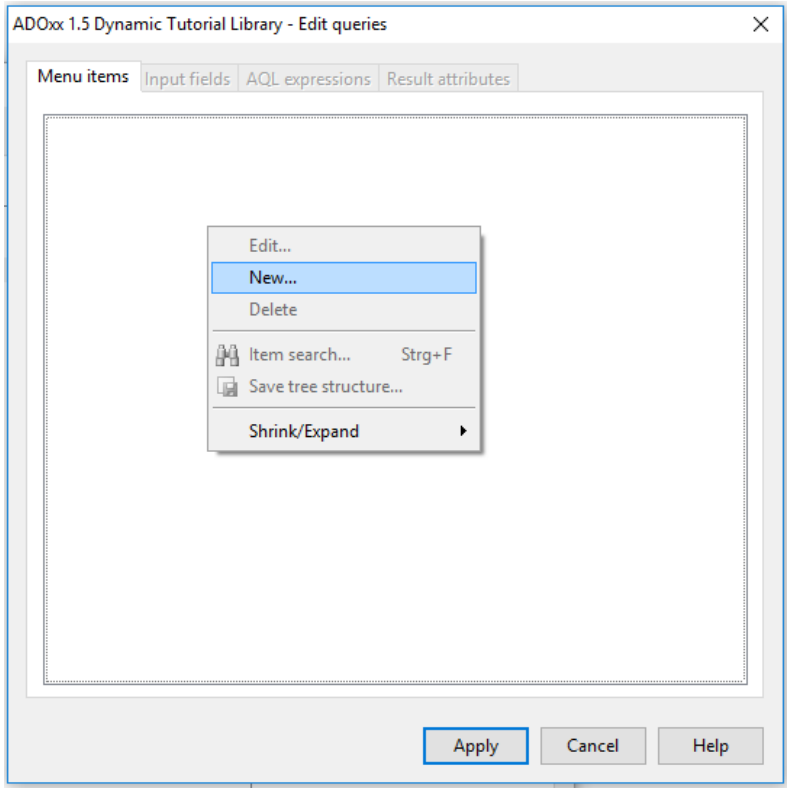
Delete: Delete an existing query from the library

Search entries: Call of the ADOxx search function

Save tree-structure: Save the content of the text file

Show/close: Diverse view options

The appeal of all these function is under the context menu of the list

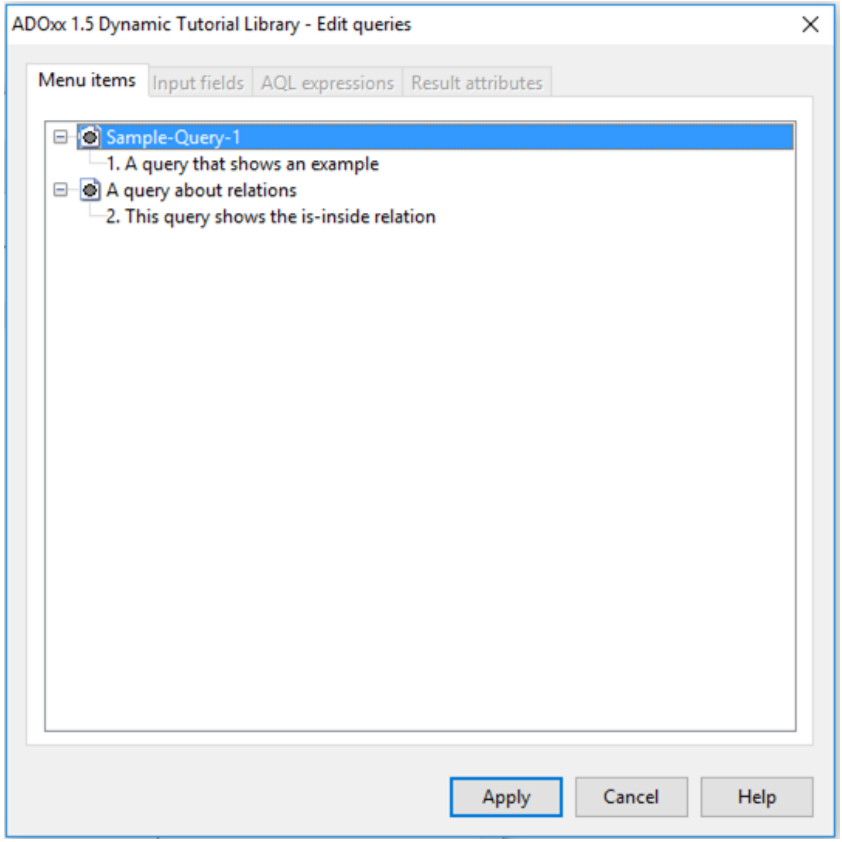


1. Define Menu Item

The creation of a new analysis query is defined through following steps:

- Create query
- Define input fields
- Define AQL-queries
- Define result attributes

For all of these steps there is a chapter in the dialog



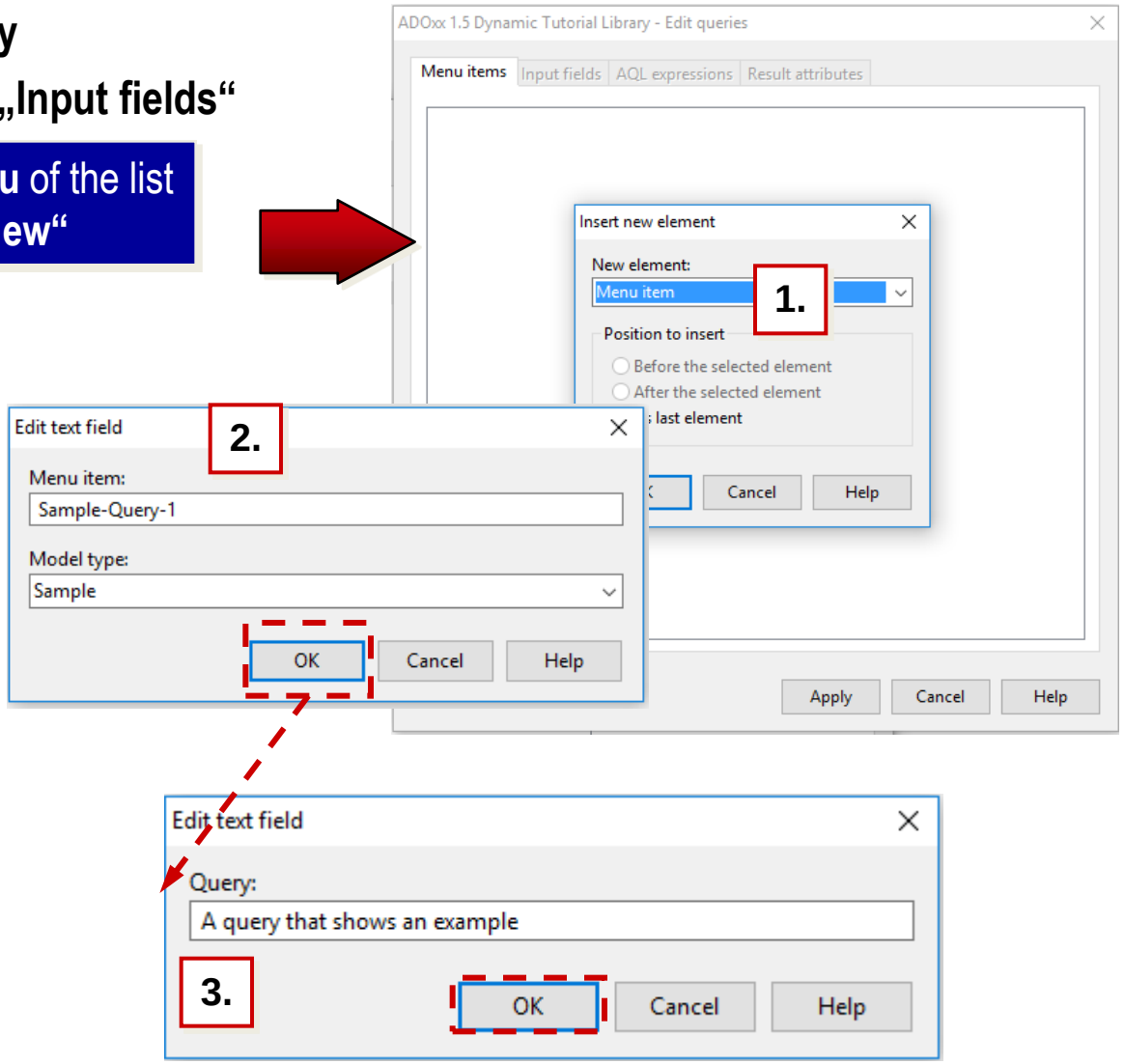
Note: Before creating a new predefined query, it is necessary to put it manually together and test it in the ADOxx-BPM-Toolkit with the „Query/reports“ function. By using this approach you get als the AQL code which is necessary in the later procedure.

Define Query Appearance

Mark the new query

Switch to Chapter „Input fields“

- **Context menu** of the list
- Menu item **„New“**



2. Define input fields

Each query consists of an individual set of parts:

Text

Input Field

Enumeration Field

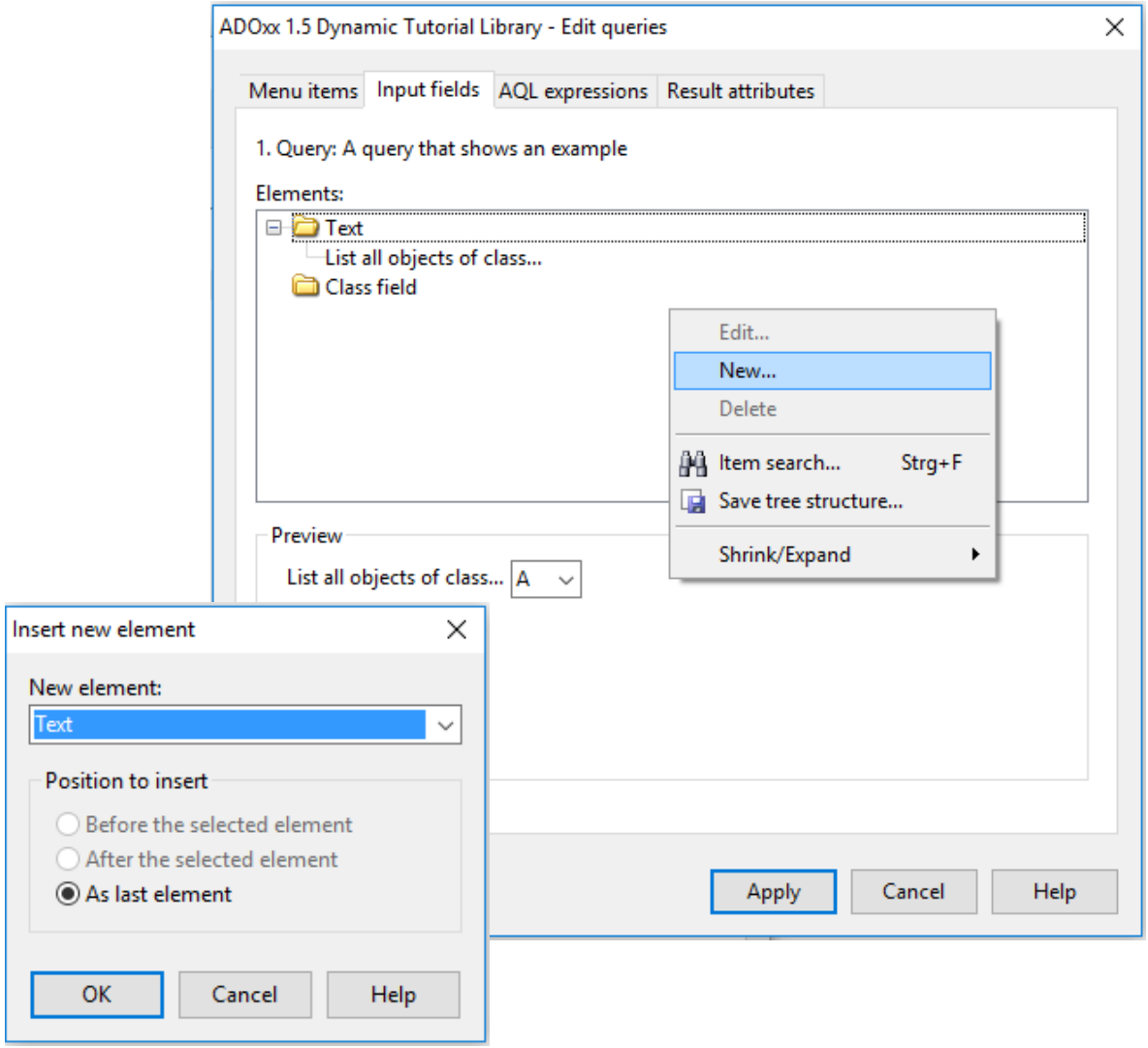
Attribute Value Field

Attribute Enumeration Field

Attribute Field

Class Field

Relation Field



Types of Input Fields

Following types of fields are available:

Input fields: For Attributes of type Text (**`STRING`**, **`LONGSTRING`**), time (**`TIME`**), date (**`DATE`**), date and time (**`DATETIME`**), integer (**`INTEGER`**) und double (**`DOUBLE`**).

Enumeration value field: For attributes of Type enumeration (**`ENUMERATION`**) and enumeration list (**`ENUMERATIONLIST`**).

Attribute value field: For the takeover of attribute values from attributes of different classes.

Enumerated attribute field: For the takeover of attribute values from enumerated attributes of different classes.

Attribute field: For choosing of attributes from a list of all attributes of all classes.

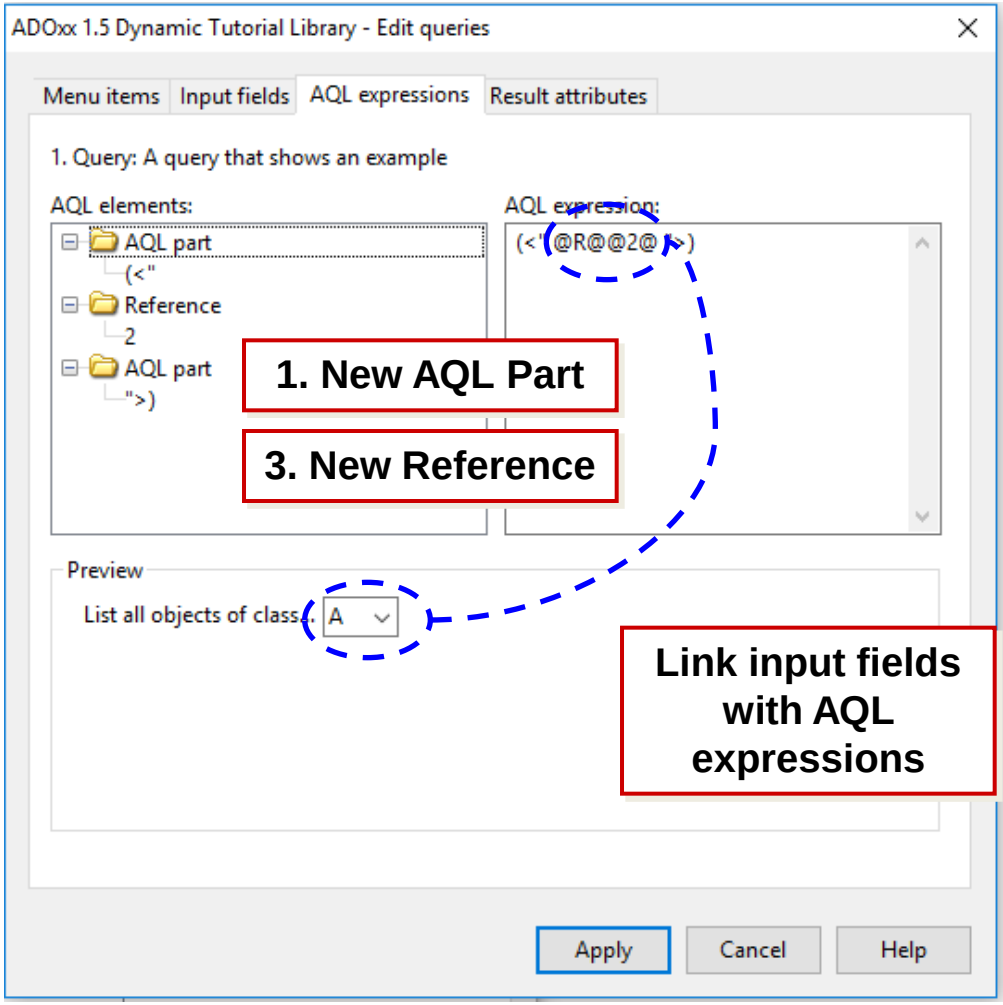
Class field: For choosing of a class from a list of all classes of the active model type.

3. Define AQL-Queries (1)

To make your query functional, it is necessary to deposit it as an AQL-query.

Switch to the chapter „AQL expressions“

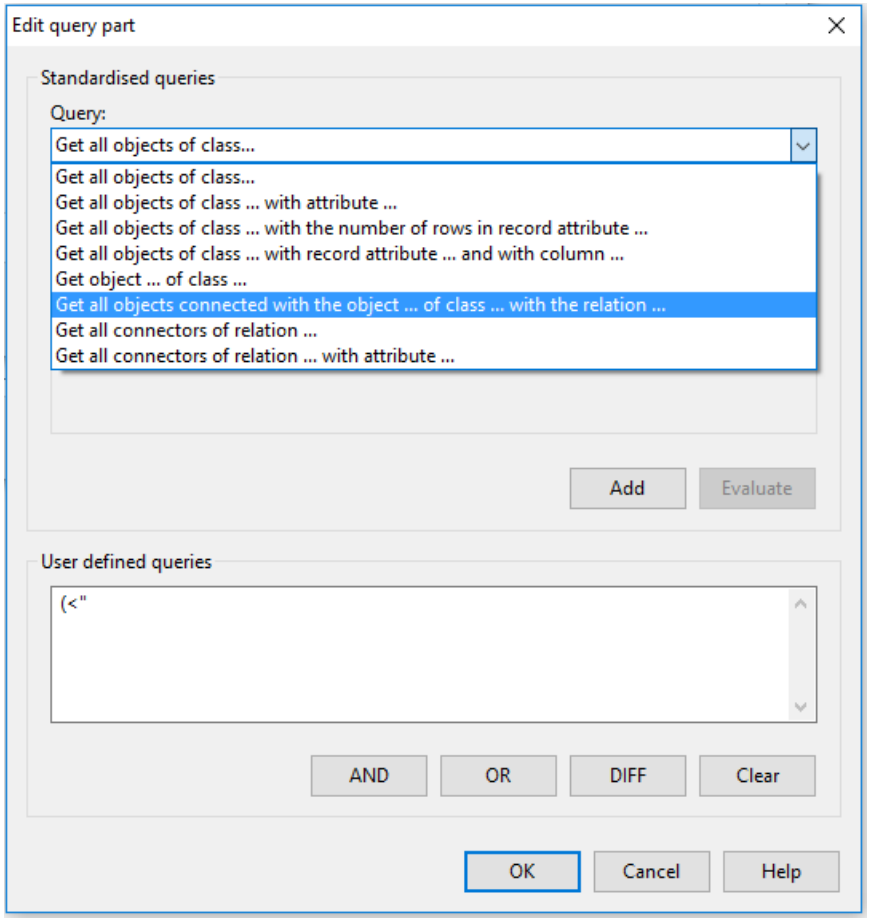
1. Choose Option „AQL Part“
2. Copy manually designed AQL statement
3. Add „References“ to link input fields with query
4. Follow proposal to place input fields into query statement



Define AQL-Queries (2)

Detail view on AQL part:

Either manually type in the statement or click on the query.

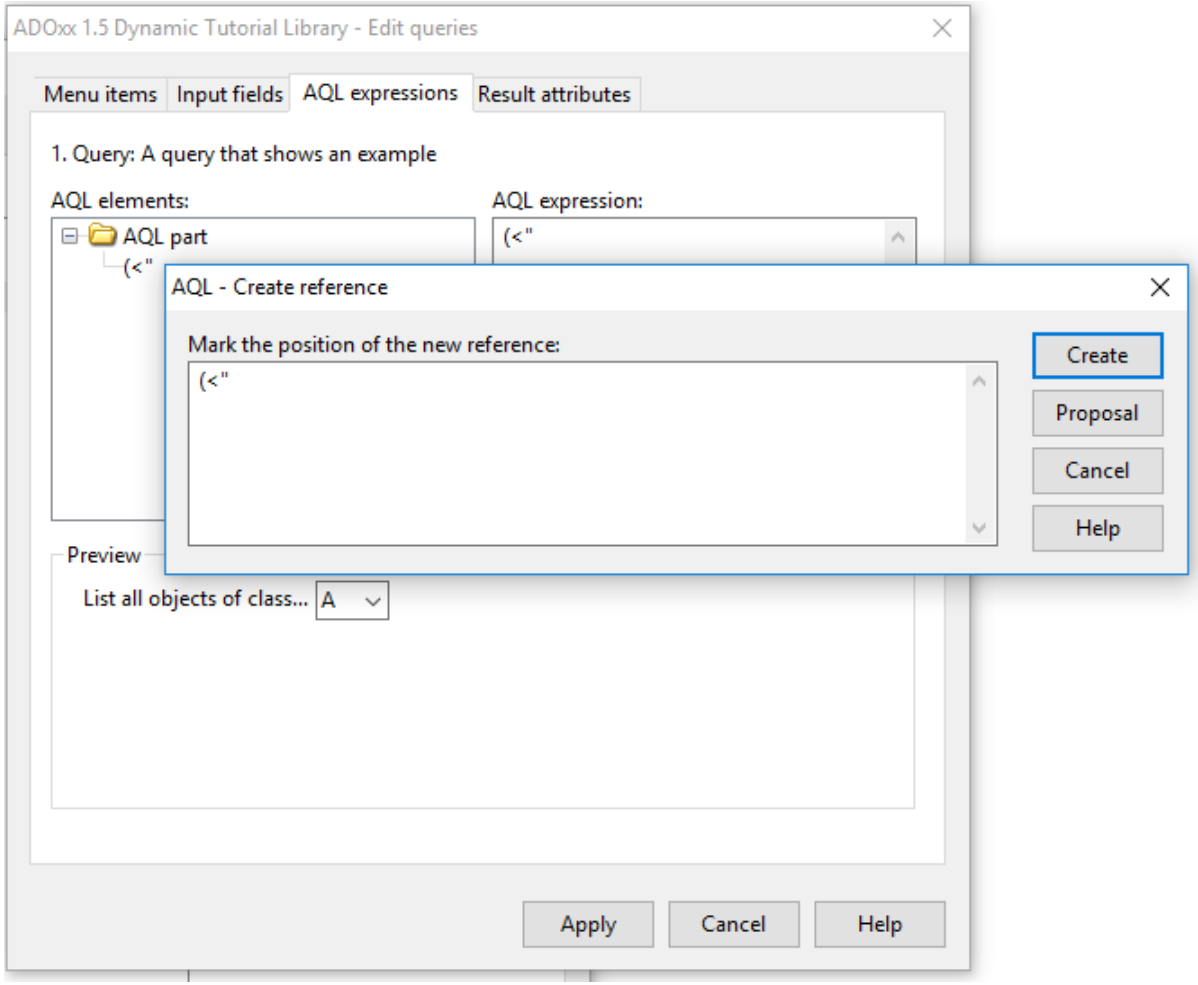


Note:
The shown AQL input support is from the construction same as the function „Query/reports“ in the ADOxx-BPM-Toolkit.

Define AQL-Queries (3)

In the next steps it is necessary to transform the query part.

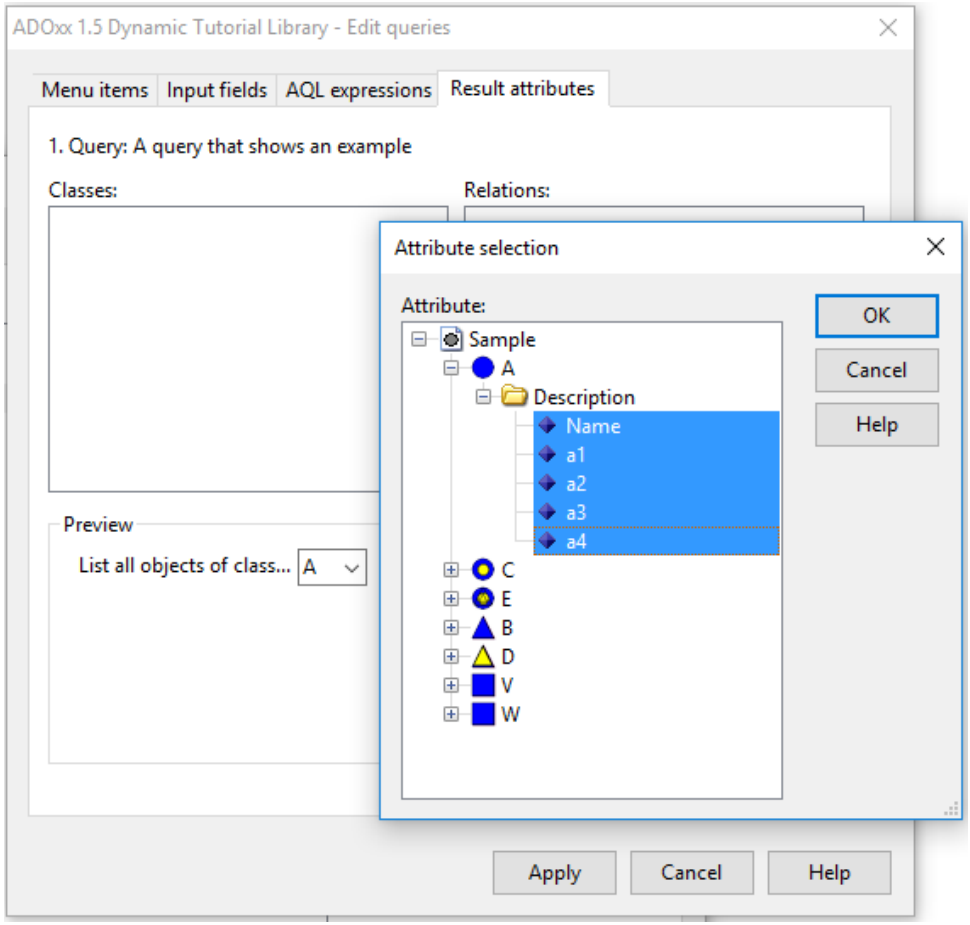
1. Context menu
   of the list entry „AQL part“
2. Menu point „New“



4. Result Attributes (1)

In the chapter „Result attribute“ it is specified which objects and attributes should be in the result representation.

1. Switch to chapter „Result attributes“
2. Choose Option „Attribute“
3. Determine Position
4. Confirm



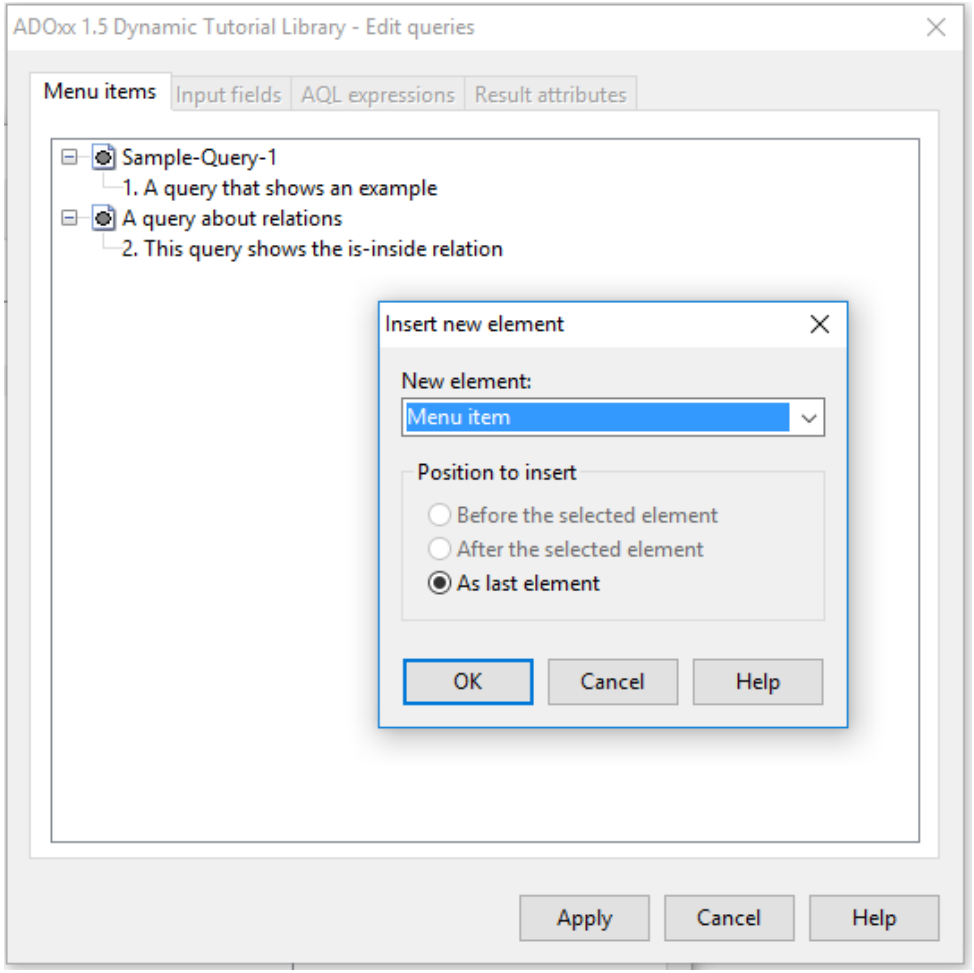
Contextmenu of the list „Classes“
Menu item **„New“**

New Menu Item

Besides the creation of queries, you can also create new menu items, in order to structure the queries.

In the Query-choose window select a query group

- Select option „Menu point“
- Determine position
- Confirm
- Input title
- Choose model type
- Confirm



↳ Contextmenu of shortlist
↳ Menu item **„New“**

New Menu Item – Details

When creating new menu items should be noted:

- every menu point is exactly **assigned to one model type**
- The menu items in the selection dialog correspond exactly to the ADOxx-based Toolkit
- After the creation of the menu item a query must be added to it, because the menu item won‘t be saved.
- Through input of tilde (~) in the menu name the following word will be an accelerator (keyboard shortcut)

AQL Notation

- Extended Backus Naur Form (EBNF) notation is used for describing the AQL syntax
- Terminal symbols
  - symbols which cannot be split up further
  - are included by inverted commas ´…´
- Non-terminal symbols
  - are included by <...>
- Symbols {...}, [...] and | serve to formulate rules in a more compact form :
  - **{...}** arbitrary number of iterations (even 0-times)
  - **[...]** optional (0- or 1-time)
  - | alternative
- Rules start with a non-terminal symbol, followed by "::=" and the symbol's definition

HINT: Keep in mind that AQL is cAse sEnsiTivE

AQL Terminal symbols (I)

'<' and '>' in this order are used to represent a class, a relation, a model, a model type, etc. (e.g. <"Rectangle"> , <"My Model 01"> , <"My First Model Type">)

':' is used for specifying the class of a certain object, or the model where a specific class is included (e.g. <"Red Rectangle 01">:<"Rectangle"> , <"Rectangle">:<"My Model 01">:<"My First Model Type">)

'>' and '<' in this order are used for filtering the results of a query by a specified class (e.g.: <"B"><<-"requires" >"A"< has as results all objects that fullfill the query criteria <"B"><<-"requires" AND are of class A)

'->' , '<-' , '->>' '<<-' , '-->' , '-->>' , '<--' are used for creating AQL expressions that involve relations in the query criteria (will be detailed in the future slides)

AQL Terminal symbols (II)

‘{‘ and ‘}‘ are used to represent the object with the specified name (e.g.: {“Rectangle“})

‘[‘ and ‘]‘ are used to introduce a criteria to an AQL expression ([?"Radius">"10"])

‘(‘ and ‘)‘ are used for deciding the order in which logical operators are evaluated i.e. ‘a OR b AND c‘ and ‘ (a OR b) AND c ‘ return different results

‘?‘ is used for imposing a condition on an attribute in the query criteria (e.g.: <"A">[?"Description" like "*OK*"] returns all objects of class A, whose attribute „Description“ contains the word „OK“)

‘!‘ is used for imposing a condition on a variable during the simulation (e.g.: (<"A">[!"objectCount">"10"]) OR (<"B">[!"objectCount"<="10"]) returns all objects of class A, if the variable object Count is higher than 10 and all objects of class B otherwise.

AQL Non-terminal symbols and key words (I)

Names of classes, names of objects, names of relations, names of attributes, names of variables and constants are denoted by inverted commas (e.g.: "A" , "Rectangle" , "requires" , "Colour")

<Class> ::= '<'*class_name*'>'

Represents a class (e.g. <"requires">)

If the class is not included in the current model, the class has to be denoted explicitly through model name and model type

<Class>':'<ModelName>':'<ModelType>

(e.g.: <"Rectangle">:<"My Model 01">:<"My First Model Type">)

<Object> :: = '<'*object_name*'>'

Represents an object within a concrete model (e.g. <"Red Rectangle 01">)

Should the name of the objects be ambiguous, the name of the class has to be appended to the object's name: <Object>':'<Class>

(e.g.: <"Red Rectangle 01">:<"Rectangle">)

if the referenced object is not part of the current model, the object has to be denoted explicitly through model name and model type:

<Object>':'<Model name>':'<Model type>

(e.g.: <"Red Rectangle 01">:<"My Model 01">:<"My First Model Type">)

<Object>':'<Class>':'<Model name>':'<Model type>

(e.g.: <"Square 01">:<"Square"><"My Model 02">:<"My First Model Type">)

AQL Non-terminal symbols and key words (II)

<Relation> ::= '<'*relation_name*'>'

represents a relation class (e.g. <"requires">)

<Attribute> ::= '<'*attribute_name*'>'

represents the name of an attribute (e.g. <"Color">, <"Description">)

<Value> ::= <Constant> | '!' <Variable> | '?' <Attribute>

a value is either a constant, a variable preceded by the symbol '!' or an attribute preceded by the symbol '?'

<Operator> ::= '>' | '>=' | '=>' | '=' | '<=' | '=<' | '<' |'!='| 'like' | 'unlike'

'like' and 'unlike' are used for alphanumerical signs

? and * are wildcards:„*" substitutes for any zero or more characters and „?" substitutes for any one character or less (123??? will match 12313 or 1233, but not 1239919991)

the other operators are used for comparing numerical values

<LogicalOperator> ::= 'AND' | 'OR' | 'DIFF'

'AND' : the result is the intersection set of the two expressions

'OR' : the result is the union set of the two expressions

'DIFF' : the result is the difference of the two expressions

'AND' links more strongly than 'OR' and 'OR' more strongly than 'DIFF'

AQL Non-terminal symbols and key words (II)

<AQL expression> ::= '{' [<Object>] [',' <Object>] '}'

the result of an AQL expression is a set of objects (0,1 or more)

<AQL expression> ::= '(' <AQL expression> ')'

expressions may contain parentheses

<AQL expression> ::= <AQL expression> {<LogicalOperator> <AQL expression>}

expressions can be linked to one or more expressions by logical operators

<AQL expression> ::= <AQL expression> '>'<Class>'<'

expression results can be filtered by a specific class

AQL Statements (I)

'<' <class> '>'

the result is all objects of the specified class

Example:

<"A">

<"Square":"SecondModel001":"My Second Model Type">

<"A"> [?"Name" like "????e?"]

<"B"> <- "requires"

AQL Statements (II)

- <AQL expression> '->' | '<-' | '->>' | '<<-' <Relation>
  - The result contains all objects which are linked through the given relation with at least one object from the AQL expression
  - '->' returns all direct targets of the relation
  - '<-' returns all direct start objects of the relation
  - '->>' returns all transitive targets of the relation
  - '<<-' returns all transitive start objects of the relation

Example:

- {"A1"}->"requires "
- {"A4"}<-"requires"
- <"A">->"requires"
- <"A"><-"requires"
- {"Rectangle01"}<-“owns“
- ({"A2"}->>"requires") -> "has list"
- {"A4"}<<-"requires“
- <"Rectangle"><<-"owns“
- <"A">->"requires" >"B"<

AQL Statements (III)

<AQL expression> '->' | '<-' '<' <Relation> '>'

The result contains all connectors of the specified relation which have as start or target object one of the objects in the AQL expression

'->' returns all connectors originating from the objects of the AQL expression

'<-' returns all connectors ending in the objects of the AQL expression

Please note similarities and differences with before: if you use the '<' and '>' symbols, the result contains the connectors and if you don't use them, it contains the objects

Example:

<"B">-><"requires">

<"A"><-<"requires">

{"List003"} <- <"has list">

{"A1"} -> <"has list">

AQL Statements (IV)

<AQL expression> '-->' | '-->>' <Attribute>

The result contains all objects which are referenced in the specified attribute of any of the objects in the AQL expression

The '-->>' operator returns is all objects which are transitively referenced in the specified attribute of any of the objects in the AQL expression

Example:

<"A"> --> "IsRunBy"

<"A"> -->> "IsRunBy"

{"A5"} --> "IsRunBy"

{"A5"} -->> "IsRunBy"

{"A5"} -->> "IsRunBy" >"Rectangle"<

Statements (V)

<AQL expression> '<--'

The result contains all objects which refer any of the objects in the AQL expression

Example:

<"Rectangle"> <--

AQL Statements (VI)

- <AQL expression> '[' <Value> <Operator> <Value> ']'
  - The result contains all objects, whose attributes fulfill the defined criteria
  - Constants (numbers, strings) can only be at the right of the operator
  - To the left of the operator there are only attributes or variable references
  - Note: Queries with variable references as dynamic components in the performer assignment are only allowed in the simulation

Example:

- (<"A"> [?"Description" like ""]) AND (<"A"> [?"A_cost" >=10])
- <"A">[?"Description" like "*Test*"]
- (<"Rectangle">[?"Name" like "M*"]) AND (<"Rectangle">[?"Area" <= 20])
- <"A">[?"Name" like "????e?"]

AQL Statements (VII)

<AQL expression> '['<Value>']' '['<Value> <Operator> <Value>']'

The result contains all objects of the start query where their record attribute or attribute profile fulfills the defined criteria

The first value specifies the name of the record attribute or attribute profile.

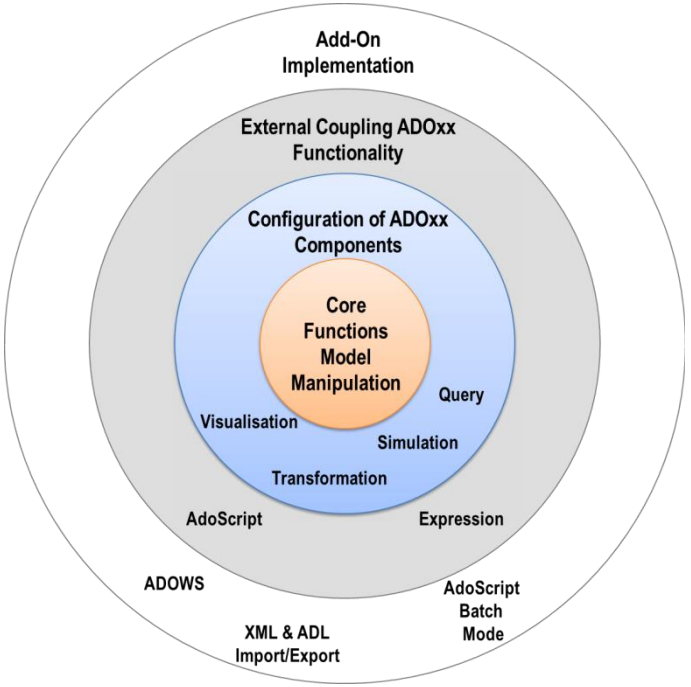
See above the rules for the second expression

Note: In case of a record attribute, the criteria is always fulfilled, if at least a table row of the record attribute meets the defined criteria.

Example:

record attribute: <"List">[?"Classification"][?"State" = "Authorized"]

attribute profile: <"A"> [?"Availability"][?"Days per week" >= 3]



3. EXTERNAL COUPLING ADOXX FUNCTIONALITY

What is AdoScript?

AdoScript is the macro language of ADOxx. It is based on LEO and is build procedural. Through AdoScript the user has access to a huge number of ADOxx functionalities.

AdoScript is a mighty tool which allows huge extension possibilities with low programming effort.

Examples:

- New menu entries
- Integration of new tools
- Realisation of specific model checking
- Realisation of new interfaces
- Additional add-on-programming

How is AdoScript used?

AdoScript can be executed on different ways. So it can be used where it is needed:

As menu entry: For manual execution
(e.g. transformation procedures, evaluation scenarios)

In events: If specific actions are executed, an AdoScript can be automatically called.
(e.g. a special dialogue replaces the standard dialogue window)

Notebook via Programmcall

Automatic over Command prompt

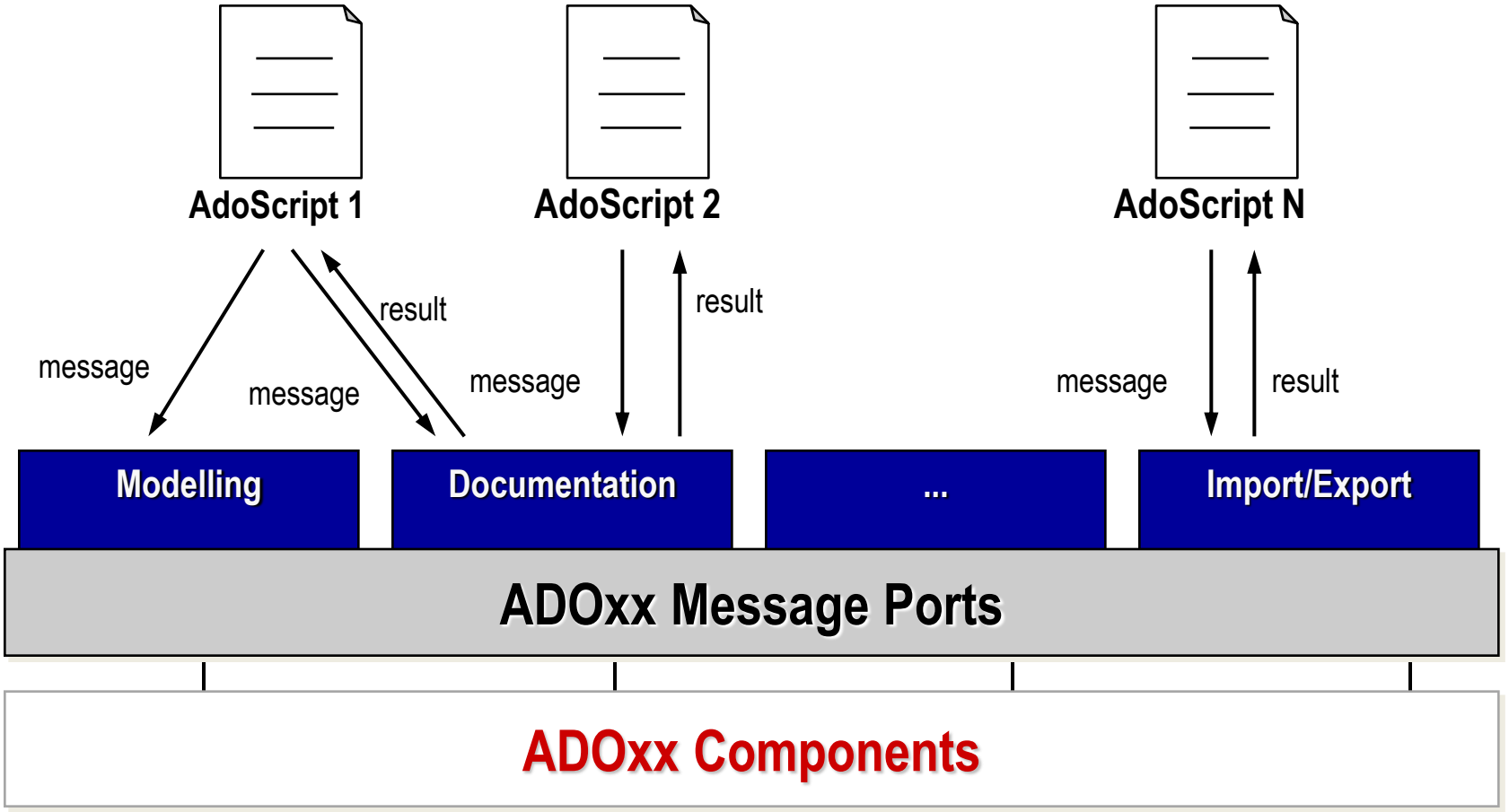
```
ECHO CC "AdoScript" FREAD file:("batchupd.adoscript")
EXECUTE (text) CC "Application" EXIT |
areena -ubatchupd -pbatchupd -dADOxxdb -ssqlserver -e
```

From AdoScript-Shell

Integration of AdoScript:

The Message Port-Concept

AdoScript can be integrated with „External binding“ or „Programm call“.



Programmable through scripting APIs

- **Method-specific development of functionalities through scripting**
- Function calls/APIs of the platform (realized in C++) are possible through scripting language AdoScript.
- Categorisation of APIs called „Messageport“.

Component APIs

Messageport **Acquisition**
Messageport **Modeling**
Messageport **Analysis**
Messageport **Simulation**
Messageport **Evaluation**
Messageport **ImportExport**
Messageport **Documentation**
Messageport **AQL**

UI APIs

Messageport **AdoScript**
Messageport **CoreUI**
Messageport **Explorer**

Manipulation APIs

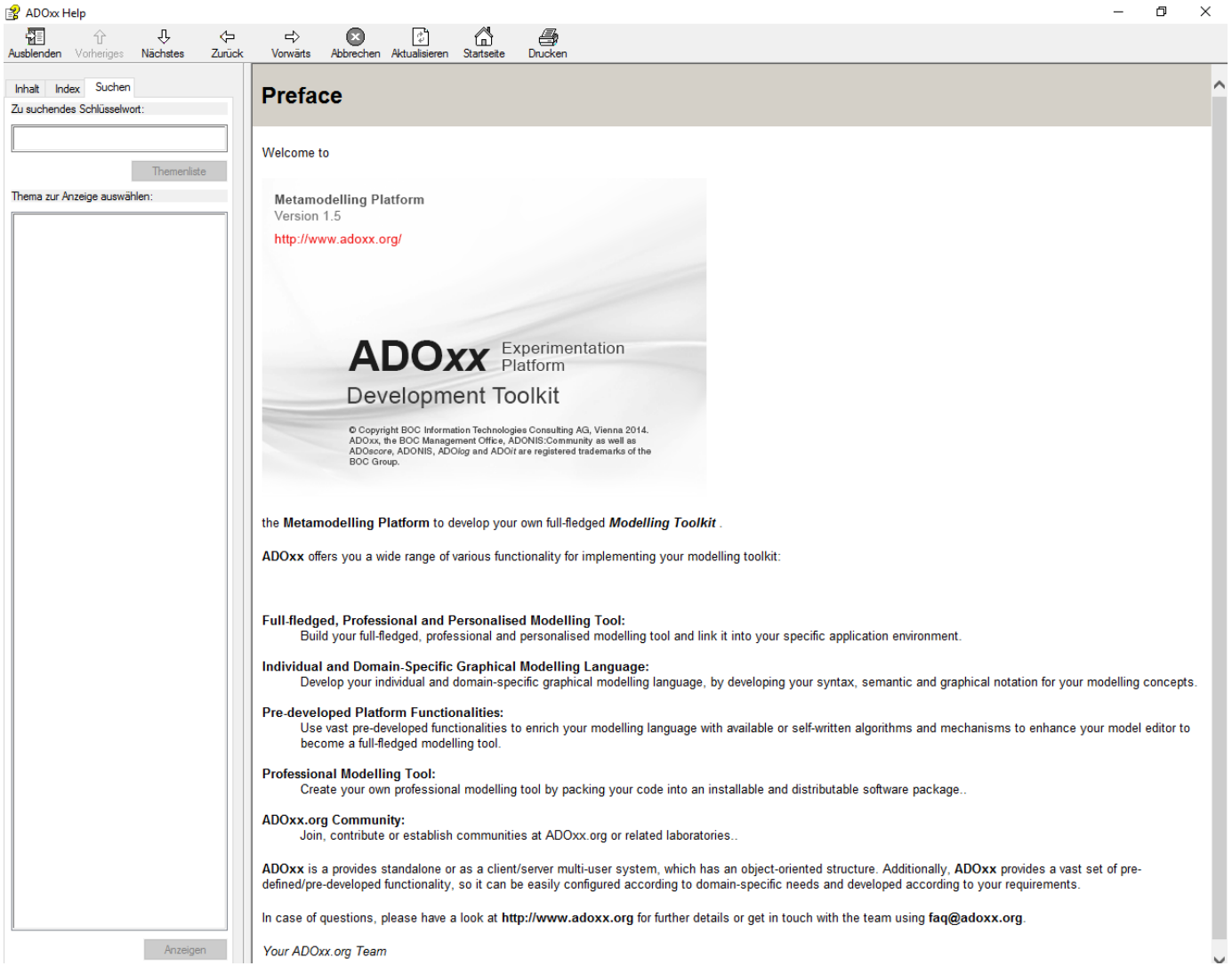
Messageport **Core**
Messageport **DB**
Messageport **UsrMgt**

Application APIs

Messageport **Drawing**
Messageport **Application**

About 400 APIs are available.

Documentation of MessagePorts and AdoScript Call Signatures



Useful Hint

Every Command Call stores the result in global variables

=> HINT:

Store right after the CC required global variables in local variable to avoid overwriting by the next CC

Tracking the global variables with “debug”

=> HINT:

Use the keyword debug during CC “CC “xxx” debug” to track the status of variables

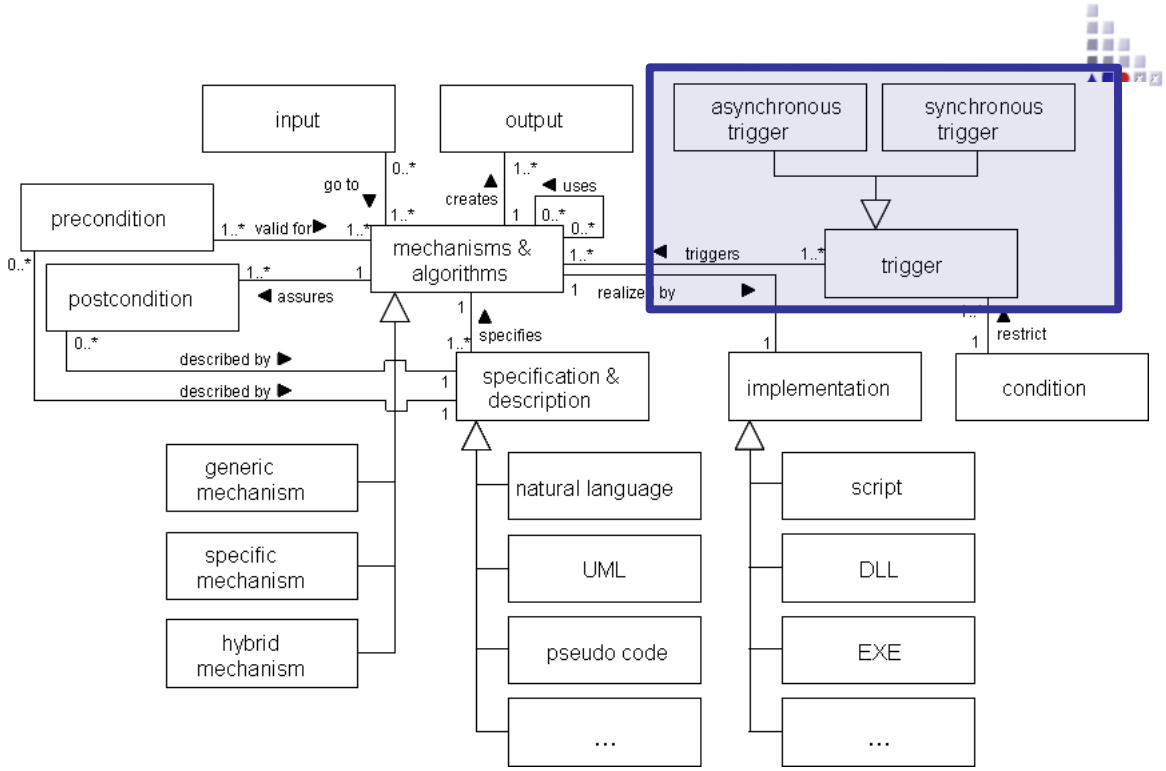
Variables are allocated with values, distinguish if you manipulate the variable v1, or the value of the variable (v1)

=> HINT:

- use VAL and STR to convert strings to integer and vice versa
- use tokcnt to count tokens in a result list
- use () to get the value of a variable
- use CM to convert into centimetre

Data Type Conversion

- **STR** *val* — Converts a *value* into a string.
-
- **VAL** *str* — Parses the string and returns that value.
- **CM** *realVal* — Converts a real value in centimetres into a centimetre
- **PT** *realVal* — Converts a real value in points into a measure value.
- **uistr (***val, digits***)** — Converts a real value in a string
- **uival (***str***)** — Converts a string value in a real value
- **CHR** *intVal* — Returns the character of for the character code provided in *intVal*. Return type is *str*. For example: **CHR 65** = **"A"**.
- **ASC** *str* — Returns the character code for the character passed in *str*. For example: **ASC "A" = 65**.
- **INT** *realVal* — Returns an *intVal*. The *realVal* is converted to integer by truncating digits after the decimal point.



ADOSCRIPT TRIGGERS

Library Attribute "External binding"

Through the library attribute „External binding“ it is possible to extend the available ADOxx functionality. For every component a library specific menu item can be defined, which is bind to a executable script. Besides in the external coupling the coupling to software of other producers can be configured.

```
ON_EVENT "AppInitialized"
{
  #do smthg.
}
```

Ex. Event

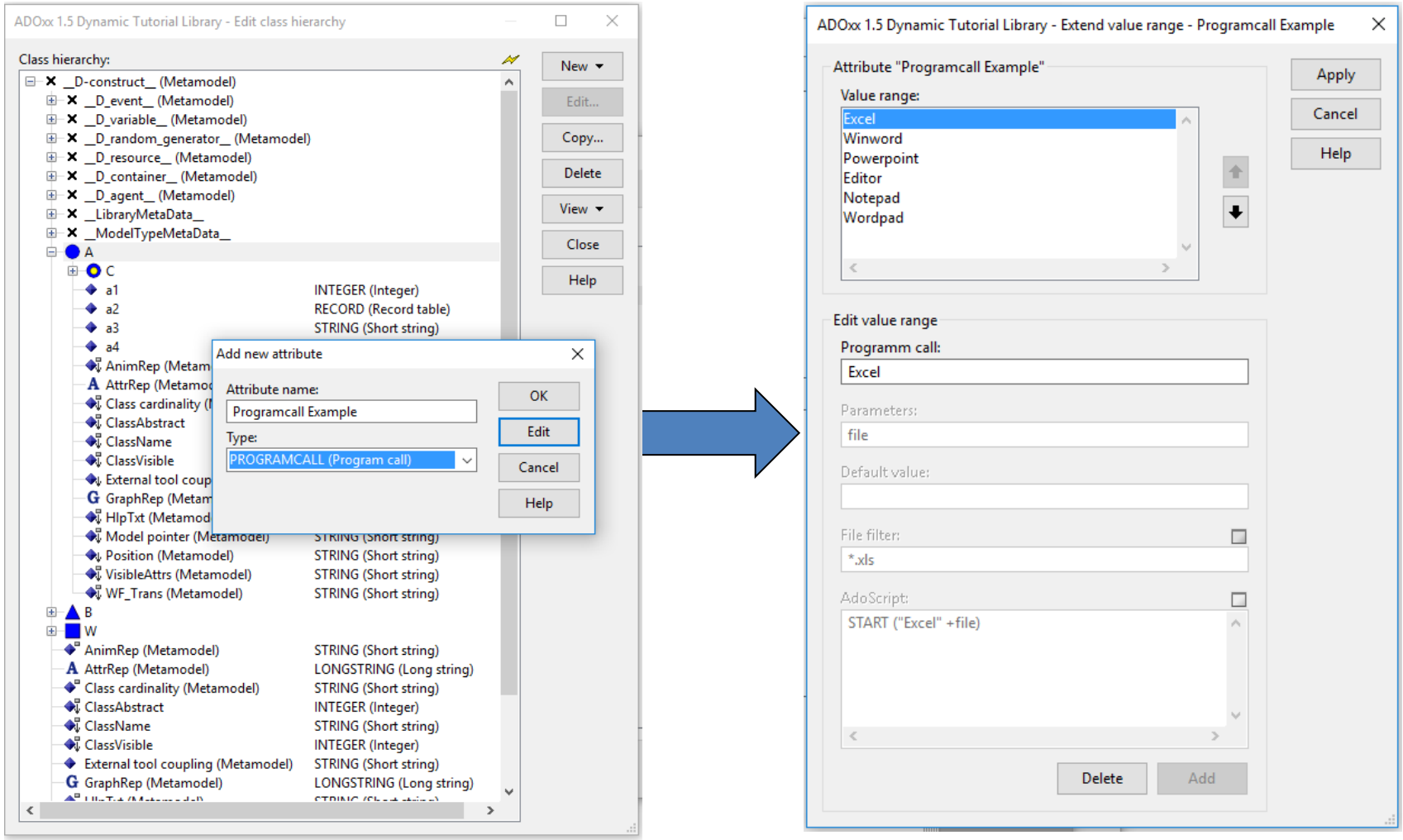
```
ITEM "ADOxx-Standard-Methode„
     acquisition:"~Hilfe„
     modeling:"~Hilfe„
     analysis:"~Hilfe„
     simulation:"~Hilfe„
     evaluation:"~Hilfe„
     importexport:"~Hilfe"
CC "Application" GET_PATH "adostd.hlp"
START ("\"" + path + "\"")
```

Ex. Menu

AdoScript via Notebook

Attribute type PROGRAMCALL

Execution of AdoScript commands over program calls attributes in the notebook of objects.



Automatically over the command prompt

AdoScript integration over command line parameter on startup of tools:

```
ECHO CC "AdoScript" FREAD file:("batchupd.adoscript")
EXECUTE (text) CC "Application" EXIT |
areena -ubatchupd -pbatchupd -dADOxxdb -ssqlserver -e
```

Command Line Parameter

- -uUser
- -pPassword
- -dDatabase
- -sDBServerSystem
- -eInlineHandOver vs. File Execution

AdoScript Shell Window

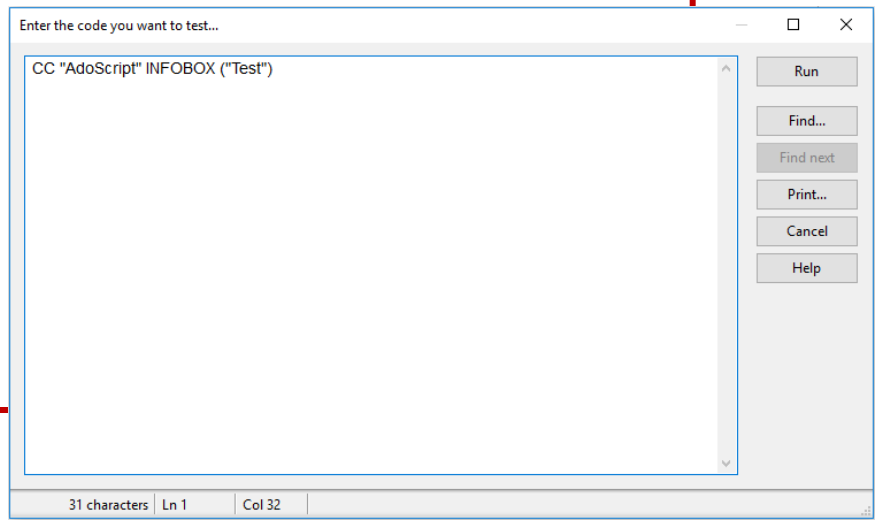
Debug/Development Facility

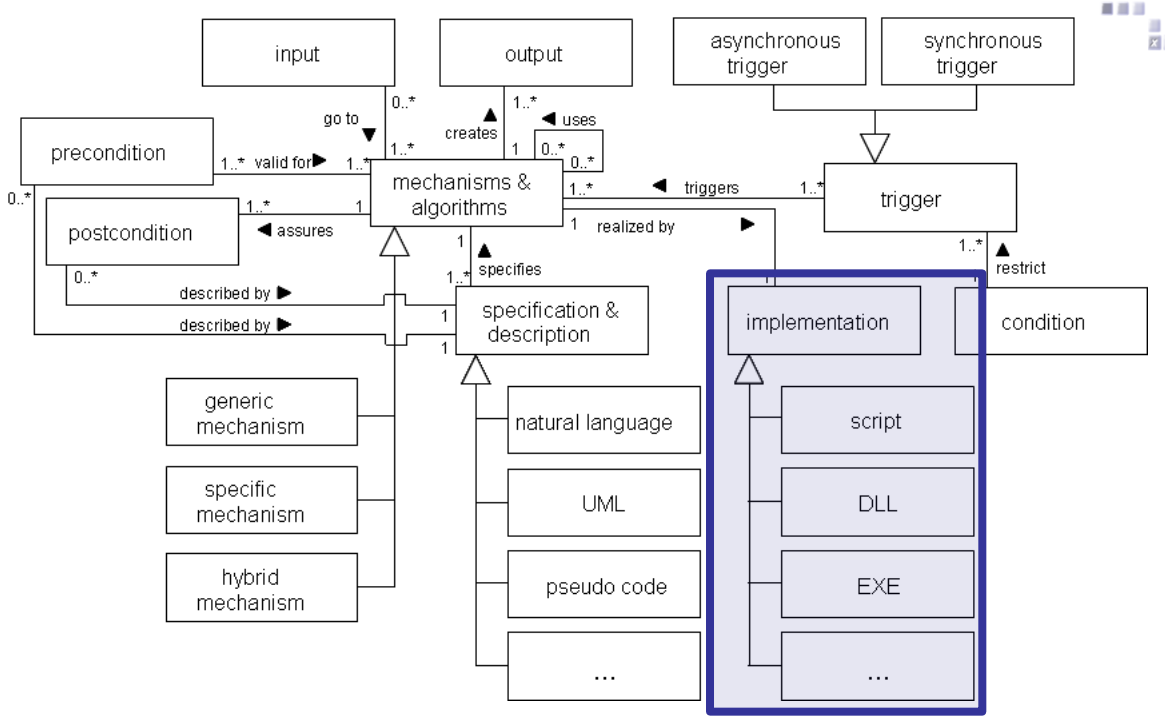
Menu Entry

```
ITEM "Shell window" modeling:"Extras"
```

Script Code for Shell Window

```
SET endbutton:"ok"
WHILE (endbutton = "ok") {
  CC "AdoScript" EDITBOX text: (adoscript)
     title:"Enter the code you want to test..."
     oktext:"Run"
     fontname:"Arial" fontheight: 14
  IF ( endbutton = "ok") {
    SETG adoscript:(text)
    EXECUTE (text)
  }
}
```

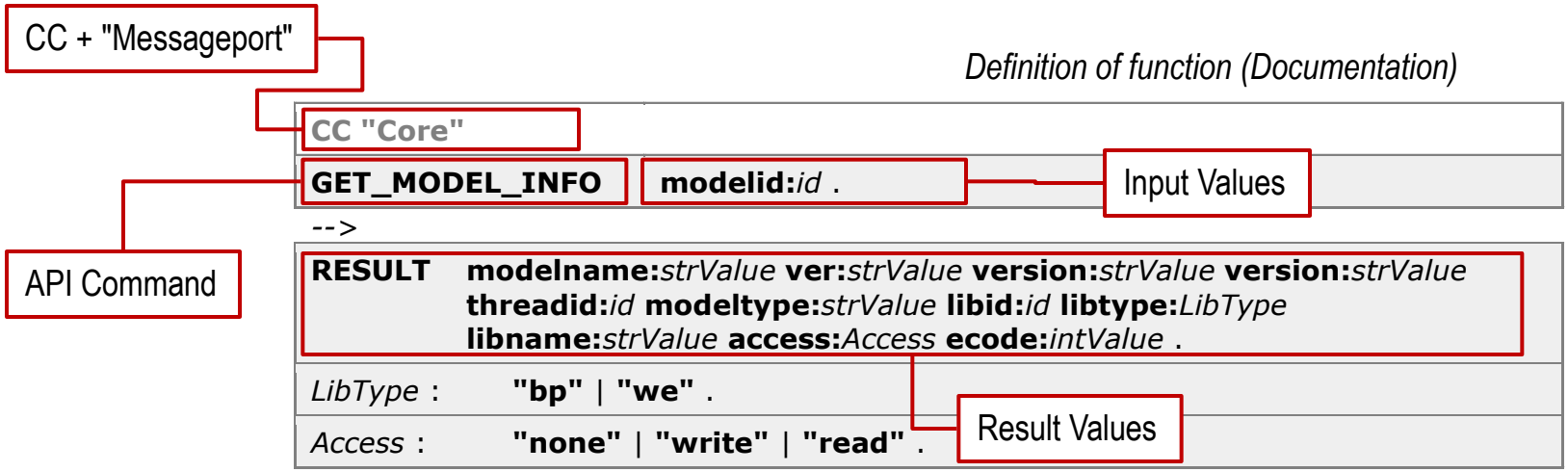




ADOSCRIPT LANGUAGE CONSTRUCTS

API Command Structure

Example of script command- Read of the model information



Code Example

```
# Reading out of the ModelID of a model currently open
CC "Modeling" GET_ACT_MODEL
# Errorcheck ecode
IF (modelid != -1) {
  # Command Call(Keywords in Capitals)
  CC "Core" GET_MODEL_INFO modelid:(modelid)
  # Handling of Return Values
  CC "AdoScript" INFOBOX ("The active model is \"" + modelname + "\" (" +
     modeltype + ")")
}
ELSE {
  # ecode returned
  CC "AdoScript" ERRORBOX "No model is opened!"
}
```

AdoScript Basics

Variable declaration

```
SET, LEO
```

Control structures

```
IF/ELSIF/ELSE, WHILE, FOR, BREAK, EXIT, PROCEDURE,
   FUNCTION
```

External programs / File callings (AdoScript, EXE, DLL)

```
EXECUTE, SYSTEM, START, CALL
```

Sending of messages to ADOxx Component (Messageports)

```
CC, SEND
```

LEO Expressions

Usage of expressions for call parameters.

AdoScript Operators

Logical

```
AND, OR, NOT
```

Comparison

```
< > <= >= = <> !=
```

Arithmetical

```
+ - * / - (unary)
```

Strings

```
s + t, n * s, s / t, s SUB i, LEN s
```

Converting

```
STR value, VAL string
```

AdoScript Functions

Arithmetical

```
abs(x), max(x, y), min(x, y), pow(x, y), sqrt(x),
exp(x), log(x), log10(x)
```

Strings

```
search(source,pattern,start),
bsearch(source,pattern,start),
copy(source,from,count),
replall(source,pattern,new), lower(source),
upper(source)
```

Lists

```
token(source,index[,separator]),
tokcnt(source[,separator]),
tokcat(source1,source2[,separator]),
tokdiff(source1,source2[,separator]),
tokisect(source1,source2[,separator]),
tokunion(source1,source2[,separator]),
```

AdoScript: Procedure Concepts

```
ProcedureDefinition :       PROCEDURE [global]
ProcedureName
[ MainParameter ] { FormalProcParameter }
{ StatementSequence } .
MainParameter :       TypeName:paramName .
FormalProcParameter :
        paramName:TypeNameOrReference .
TypeNameOrReference :       TypeName | reference .
TypeName :    string | integer | real | measure | time
|
array | expression | undefined .
ProcedureName :       keyword .
ProcedureCall :       anyLeoElement .
```

```
PROCEDURE MYPROC integer:n val:string result:reference
{
   SET result:(val + STR n)
}
```

AdoScript: Functions concept

FunctionDefinition ::= FUNCTION functionName[:global]
{ FormalFuncParameter }
return:expression .

FormalFuncParameter ::= paramName:TypeName .

TypeName ::= string | integer | real | measure |
time | expression | undefined .

Example:

```
FUNCTION fak n:integer
  return:(cond (n <= 1, 1, n * fak (n - 1)))

SET m:(fak (10))
```

Expressions in AdoScript

Expressions can be used direct as arguments in calls.

Use closures () in order to delineate arguments of an expression.

Example

```
SET n:(copy (vn, 0, 1) + ". " + nn)
IF (cond(type ( n ) = "integer", n = 1, 0 ) ) {
   ...
}
EXECUTE ("SET n:(" + n + ")")
```

Summary of AdoScript Language Elements

AdoScript language elements

Command execution

EXECUTE	SEND
CC	SYSTEM
START	CALL

Allocation elements

SET	SETG
SETL	

Control elements

IF	ELSIF
ELSE	WHILE
FOR	BREAK
NEXT	EXIT

Definition of Procedures/Functions

PROCEDURE	FUNCTION

LEO (Return Format) Handling

LEO	LEO parse

StatementSeq :	{ *Statement* } .
Statement :	*Execute* | *Send* | *CC* | *System* | *Start* | *Call* | *Set* | *SetL* | *SetG* | *Leo* | *IfStatement* | *WhileStatement* | *ForStatement* | *BreakStatement* | *ExitStatement* | *FunctionDefinition* | *ProcedureDefinition* | *ProcedureCall* .
Execute :	*ExecuteFile* | *ExecuteEx* .
*Execute*File :	**EXECUTE file:***scriptText* [**scope:***ScopeSpec*] .
ExecuteEx :	**EXECUTE** *scriptText* [**scope:***ScopeSpec*] .
ScopeSpec :	**separate | same | child** .
Send :	**SEND** *msgText* **to:***msgPortName* [**answer:***varName*] .
CC :	**CC** *msgPortName* [**debug**] [**raw**] *anyLeoElement* .
System :	**SYSTEM** *strExpr* [**with-console-window**] [**hide**] [**result:***varName*] .
Start :	**START** *strExpr* [**cmdshow:***CmdShow*] .
CmdShow :	**showmaximized** | **showminimized** | **showminnoactive** | **shownormal** .
Call :	**CALL dll:***strExpr* **function:***strExpr* { *InputParam* } [**result:***varName*] [**freemem:***strValue*] .
InputParam :	**varName:***anyExpr* .
Set :	**SET** { *VarAssignment* } .
SetL :	**SETL** { *VarAssignment* } .
SetG :	**SETG** { *VarAssignment* } .
VarAssignment :	**varName:***anyExpr* .
Leo :	**LEO** [*parseCmd*] { *accessCmd* } .
parseCmd :	**parse:***stringExpr* .
accessCmd :	**get-elem-count:***varName* | **set-cur-elem-index:***intExpr* | **get-keyword:***varName* | **is-contained:***varName* [**:***strExpr*] | **get-str-value:***varName* [**:***strExpr*] | **get-int-value:***varName* [**:***strExpr*] | **get-real-value:***varName* [**:***strExpr*] | **get-tmm-value:***varName* [**:***strExpr*] | **get-time-value:***varName* [**:***strExpr*] | **get-modifier:***varName***:***strExpr* .
IfStatement :	**IF** *booleanExpr* **{** *StatementSequence* **}** { **ELSIF** *booleanExpr* **{** *StatementSequence* **}** } [**ELSE {** *StatementSequence* **}**] .
WhileStatement :	**WHILE** *booleanExpr* **{** *StatementSequence* **}** .
ForStatement :	*ForNumStatement* | *ForTokenStatement* .
ForNumStatement :	**FOR** *varName* **from:***numExpr* **to:***numExpr* [**by:***numExpr*]

AdoScript Programm Guidelines

- If you are programming AdoScript, please consider following rules:
  - Files which contain AdoScript should be named file.asc
  - The returning result of a message port command should be under the command.

```
CC "Core" GET_CLASS_NAME classid:(intValue)
# --> RESULT ecode:intValue classname:strValue isrel:intValue
```

  - Indent a block with two spaces
  - Don‘t use the tabulator to indent blocks
  - Decompose the complexity of the program by using procedures and functions.

VISUALISATION AdoScript

3. EXTERNAL COUPLING ADOXX FUNCTIONALITY

Sample on Visualisation with AdoScript

1. **Count how many objects of class E have been modelled in actual model**
2. **Create a new model in a selected model group with Target-Result as an object**
3. **The radius of the target result is the count of modelled Es.**

⇒ **The more Es have been modelled, the bigger is the resulting circle.**

Requirements: Extension of the Tutorial Library

- Create a new Model type named “Result-Type 1”
- Create a new Class named “Result-of-Count”
- Include the Class to the new Model type
- Create a new Attribute named “number of counts” (Integer) for the Class
- Add a GraphRep to the Class
  (https://www.adoxx.org/live/adoxx-graphrep-repository-wiki/-/wiki/GRAPHREP+Repository/CLASS+Gray+circle)

Visualisation AdoScript - Example

```
## Get active Model
CC "Modeling" GET_ACT_MODEL
SETL nStartmodelID:(modelid)

# make an info box for debuggin reasons - convert value of id_actmodel
into a string
CC "AdoScript" INFOBOX ("Hello " + STR(nStartmodelID) + "!")
   title:"Start model id!"

## count how many objects of class "E" have been modelled in that model

# get the id of class "E"
CC "Core" debug GET_CLASS_ID classname:"E"
SETL nClassID:(classid)

#-------------------- This GET_CLASS_ID sets following global variables
#--------------------[ecode:intValue] [classid:intValue]

# get all objectss of class "E"
CC "Core" debug GET_ALL_OBJS_OF_CLASSID modelid: (nStartmodelID)
   classid:(nClassID)
```

Visualisation AdoScript – Example (cont'd)

```
IF (LEN(objids) = 0) {
  CC "AdoScript" ERRORBOX ("You have no instances of class E!")
  EXIT
}
SETL debug nCount_of_objects: (tokcnt(objids))

CC "AdoScript" INFOBOX ("Hello " + STR(nCount_of_objects) + "!")
   title:"Count of objects of class E!"

## Creating a new model
CC "CoreUI" MODEL_SELECT_BOX mgroup-sel without-models
   title:"Zielmodellgruppe" boxtext:"Selektieren Sie die Ziel-
   Modellgruppe in der Datenbank:"

# --------------- This MODEL_SELECT_BOX sets a couple of global variables
# ---------------[ modelids: idList | threadids: idList ]
# ---------------[ mgroupids: idList ] [ appmodelids: idList ]
# ---------------[ extraValues ]
# ---------------the global variable mgroupids is used in CREATE MODEL
```

Visualisation AdoScript - Example (cont'd)

```
CC "Core" CREATE_MODEL modeltype:"Result-Type 1"
   modelname:"My First own result" version:"1.0"
   mgroups:(mgroupids)

# open the new created model AND to make the new model ACTIVE
IF (ecode = 0) {
  CC "Modeling" CREATE_WINDOW_FOR_LOADED_MODEL modelid:(modelid)
}

## Create objects in the new model

# get the model id of the new model
CC "Modeling" GET_ACT_MODEL
SETL nResultmodelID:(modelid)

# make an info box for debugging reasons - convert value of id_actmodel
into a string
CC "AdoScript" INFOBOX ("Hello " + STR(nResultmodelID) + "!")
   title:"Result model id!"
```

Visualisation AdoScript - Example (cont'd)

```
# get the id of class "Result-of-Count"
CC "Core" debug GET_CLASS_ID classname:"Result-of-Count"
SETL nClassID:(classid)

# create the object
CC "Core" debug CREATE_OBJ modelid:(nResultmodelID) classid:(nClassID)
   objname:"A new Result-of-Count"
SETL nObjectID:(objid)

IF (ecode != 0) {
  CC "AdoScript" ERRORBOX ("The object could not be created. \n" +
     "Maybe one with the same name already exists?")
}

# get the attribute "number of counts" of the class
CC "Core" GET_ATTR_ID classid:(nClassID) attrname:"number of counts"
SETL nAttrID:(attrid)
```

Visualisation AdoScript - Example (cont'd)

```
IF (ecode != 0) {
  CC "AdoScript" ERRORBOX ("The selected object does not contain an
     attribute called number of counts !")
  EXIT
}

# set the name of the selected object
CC "Core" debug SET_ATTR_VAL objid:(nObjectID) attrid:(nAttrID)
   val:(nCount_of_objects)
IF (ecode != 0) {
  CC "AdoScript" ERRORBOX "Could not set the attribute value!"
  EXIT
}
CC "Modeling" SET_OBJ_POS objid:(nObjectID) x:(7cm) y:(7cm)
```

VISUALISATION EXPRESSION

3. EXTERNAL COUPLING ADOXX FUNCTIONALITY

Expressions

AdoScript vs. Expressions

AdoScript	Expressions
• Allows embedding external functionality	• No external functionality
• Read and write access to most attributes	• Read access to most attributes, write access only to own attribute
• Must be triggered explicitly by the user	• Are triggered automatically
• Can embed Expressions	• N/A
• Cannot be changed by the modeler	• Can be changed by the modeler if not defined as “fixed”
• Usually synchronous execution	• Can be synchronous or asynchronous (idle-processing)
• Any complexity	• Usually less complex than AdoScripts
	• Careful with closed models (values can be outdated)

3 Types of Expressions

- **LeoExpressions:**
  - Provide a basic set of functions and operators
  - Support for calculation of values, manipulation of strings and other basic operations
  - Used inside LEO based languages

- **CoreExpressions:**
  - Extension of LeoExpressions
  - Only used in EXPRESSION attributes

- **AdoScriptExpressions:**
  - Extension of LeoExpressions
  - Additional functions can be created (using the keyword FUNCTION)
  - Only used in AdoScripts

Expressions – Operations (1)

Logical Op.	`AND, OR, NOT`	Boolean expressions
Comparison Op.	`< > <= >= = <> !=`	Bigger, smaller, equal, diverse
Arithmetic Op.	`+ - * / - (unary)`	
String Op.	`s + t`	Concatenation of Strings s and t
	`n * s`	Replication: String s is replicated n-times
	`s / t`	Count: how often can String s be found in t
	`s SUB i`	The i-th character in String s
	`LEN s`	Length of Strings s

Expressions – Operations (2)

Conversion Op.	`STR val`	String representation of Value val
	`VAL str`	Numerical representation of Strings str
	`CMS measure` `PTS measure`	Conversion of a Unit (in cm or points) to a real number (e.g.: CMS 3.5cm → 3.5).
	`CM real` `PT real`	Conversion of a real number to a Unit (in cm or points; e.g.: CM 3.5 → 3.5cm).
	`uistr(val, n)`	Conversion of a real number to a string in the local format (OS) with n digits.
	`uival( str )`	Conversion of a String in the local format (OS) to a real number.
Sequence Op.	`,`	The comma is used to define a sequence of expressions. The result is always the value of the last expression.

Expressions – Predefined functions (1)

Arithmethic Functions	`abs(x)`
	`max(x, y)  min(x, y)`
	`pow(x, y)  sqrt(x)`
	`exp(x)`
	`log(x)  log10(x)`
		Arithmetic functions

	`sin(x)  cos(x)  tan(x)`
	`asin(x)  acos(x)  atan(x)`
	`sinh(x)  cosh(x)  tanh(x)`
		Trigonometric functions

	`random()`	Random value 0 >= n < 1
	`round(x)`	Round-to-nearest, i.e. if decimal >= 0.5
	`floor(x)  ceil(x)`	Round up/down

Expressions – Predefined functions (2)

String-func.	`search(source, pattern, start)`	Searches in *source* for *pattern*, starting at *start* (0-based), returns index or -1
	`bsearch(source,pattern,start)`	Search begins at end of source string (backwards)
	`copy(source, from, count)`	Copies *count* characters from *source* beginning at *from* (0-based)
	`replall(source, pattern, new)`	Replaces all occurrences of *pattern* in *source* with *new*
	`lower(source)`	Transforms to lower-case
	`upper(source)`	Transforms to upper-case
	`mstr(string)`	Puts the string between “” and escapes special characters

Expressions – Predefined functions (3)

List Funct	**tokcnt(source[,sep])**	Counts tokens in *source* separated by *sep* (default = single whitespace)
	tokcat(source1, source2 [,separator])	Concatenates two lists
	tokunion(source1, source2[, separator])	Union of two lists
	tokisect(source1, source2 [, separator])	Intersection of two lists
	tokdiff(source1, source2 [, separator])	Difference of two lists
Color Funct	**rgbval(colorname)**	24bit RGB-Value of the color (by name)
	rgbval(r, g, b)	Calculates the RGB-Value for the provided color values.

Expressions – Control structures

Expressions	`set(var, expr)`	*Expr* will be stored in *var*. Variable *var* is created implicitly.
	`cond(cond1, expr1, ..., expr_else)`	Evaluate *cond1*, if true return *expr1*, if false return next condition or return *expr_else*.
	`while(cond, loopexpr[, resultexpr])`	While *cond* is true, evaluate *loopexpr*. Return *resultexpr*.
	`fortok(varname, source, sep, loopexpr [, resultexpr])`	For each element in the list *source*, evaluate *loopexpr*. The current element is stored in *varname*. The list elements are separated by *sep*. Return *resultexpr*.

Expressions – Error handling, type checks

Error handling	`try(expr, failexpr)`	Returns *expr*, if it succeeds, otherwise returns *failexpr*.
Type check	`type(expr)`	Returns the type of the expression. Possible values: "string", "integer", "real", "measure", "time", "expression„ or "undefined„.

Expressions in AdoScript

Types of expressions

Core Expressions:

- Are used to define attributes with the type EXPRESSION
- Can access functions for Core Expressions

AdoScript Expressions:

- Are used in AdoScript
- Can be externalized in functions
- Can access externalized function (defined through keyword FUNCTION)

Core Expressions

Functions for Core Expressions

- The following functions can be used in Core Expressions

```
aval()          rcount()        asum()
avalf()         row()           amax()
maval()         rasum()         awsum()
paval()         prasum()        pmf()
pavalf()        allobjs()       class()
irtmodels()     aql()           mtype()
irtobjs()       prevsl()        mtclasses()
profile()       nextsl()        mtrelns()
ctobj()                         allcattrs()
cfobj()                         alliattrs()
conn()                          allrattrs()
```

- Additionally all LEO expressions and functions can be used

Core Expressions

Attributes of the type EXPRESSION

- An expression attribute contains both a formula and the calculated value
- There are two modes for using expression: fixed and editable
- Fixed expressions store the formula in the default value of the attribute
- An error massage will be returned, if an error occurs when evaluating a formula.
- The last valid result is returned, if an inter-model expression cannot be evaluated (when trying to access a not loaded model)
- Expression attributes are always evaluated when an event occurs which can change the value. The changes are shown directly in the user interface

Core Expressions

Attributes of the type EXPRESSION: Definition of expressions as an attribute

Syntax

```
ExprDefinition:     EXPR type:ResultType
                    [format:FormatString]
                expr:[fixed:]CoreExpression
          ResultType :              double | integer | string | time
```

Example

```
EXPR type:string expr:("\"Name = \" + aval(\"Name\")")
```

AdoScript Expressions Application

Expressions can be used directly as arguments of calls and be embedded directly in AdoScript code.

Parentheses are used to delimit the arguments of an expression.

```
SET n:(copy(vn, 0, 1) + ". " + nn)
IF (cond(type(n) = "integer", n = 1, 0)) {
   ...
}

EXECUTE ("SET n:(" + n + ")")
```

Expressions can also be moved to dedicated functions so they can be reused.

AdoScript Expressions

Functions in AdoScript

It is possible to define LEO expressions as reusable functions through the keyword FUNCTION.

Syntax

```
FunctionDefinition   ::= FUNCTION functionName[:global]
                         { FormalFuncParameter }
                         return:expression .
FormalFuncParameter  ::= paramName:TypeName .
TypeName             ::= string | integer | real | measure |
                         time | expression | undefined .
```

Example

```
FUNCTION helloWorld world:string
    return:("Hello " + world + "!")

SET hello:(helloWorld("world"))
CC "AdoScript" INFOBOX (hello)
```

QUERY AdoScript

3. EXTERNAL COUPLING ADOXX FUNCTIONALITY

AdoScript for Queries

EvalAqlExpression : **EVAL_AQL_EXPRESSION expr:***strValue* (**modelid:***intValue* | **modelscope**) .
--> **RESULT ecode:***intValue* **objids:***strValue*

EVAL_AQL_EXPRESSION will evaluate the AQL string specified by the argument **expr**.

The return variable **ecode** is 0 if the evaluation yielded no error.
The list of found objects or models is returned in the variable **objids** (separated by blanks).

AdoScript for Queries (Examples)

Example 1: Get all objects of class „A" in a certain model

```
CC "Modeling" GET_ACT_MODEL
#-->RESULT modelid:intValue
CC "AQL" EVAL_AQL_EXPRESSION expr:"<\"A\">" modelid:(modelid)
IF (ecode = 0) {
  CC "AdoScript" INFOBOX ("Found objects: " + objids)
}
ELSE {
  CC "AdoScript" INFOBOX "An error has occured!"
}
```

Example 2: Get all models of modeltype „Sample"

```
CC "AQL" EVAL_AQL_EXPRESSION expr:"<\"Sample\">" modelscope
IF (ecode = 0) {
  CC "AdoScript" INFOBOX ("Found models: " + objids)
}
ELSE {
  CC "AdoScript" INFOBOX "An error has occured!"
}
```

AdoScript for Queries (Examples)

```
CC "Modeling" GET_ACT_MODEL
# make first query with all objects inside the class V
CC "AQL" EVAL_AQL_EXPRESSION
   expr:"(<\"V\"><-\"Is inside\")"
   modelid:(modelid)
SETL sObjids1:(objids)
# make second query with all objects of class A in model
CC "AQL" EVAL_AQL_EXPRESSION expr:"(<\"A\">)"
   modelid:(modelid)
SETL sObjids2:(objids)
# union of the two AQL result sets
SETL sAllobjectids:(tokunion(sObjids1, sObjids2))
IF (ecode = 0) {
  CC "AdoScript" INFOBOX ("Found objects:" + sAllobjectids)
}
ELSE {
  CC "AdoScript" INFOBOX "An error has occured!"
}
```

AdoScript for Queries (Examples)

Following Interref a4 of an object named A1 (of class A)

```
CC "Modeling" GET_ACT_MODEL
CC "AQL" debug EVAL_AQL_EXPRESSION
    expr:"({\"A1\"} --> \"a4\")"
    modelid:(modelid)
SETL sObjids:(objids)
IF (ecode = 0) {
  CC "AdoScript" INFOBOX ("Found objects: " + sObjids)
}
ELSE {
  CC "AdoScript" INFOBOX "An error has occured!"
}
```

TRANSFORMATION AdoScript

3. EXTERNAL COUPLING ADOXX FUNCTIONALITY

Transformation Example 1

```
## Open Model
CC "Modeling" GET_ACT_MODEL
SETL nSourceModelID:(modelid)

SETL sClassnameSource:("A")
SETL sClassnameTarget:("E")

# BEGIN set new model
CC "CoreUI" MODEL_SELECT_BOX mgroup-sel without-models
   title:"Zielmodellgruppe"
   boxtext:"Selektieren Sie die Ziel-Modellgruppe in der Datenbank:"

CC "Core" CREATE_MODEL modeltype:"Sample"
   modelname:"My First sample"
   version:"1.0"
   mgroups:(mgroupids)
SETL nTargetModelID:(modelid)

# END set new model

CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(nSourceModelID)
    classname:(sClassnameSource)
SETL sObjIDs:(objids)
```

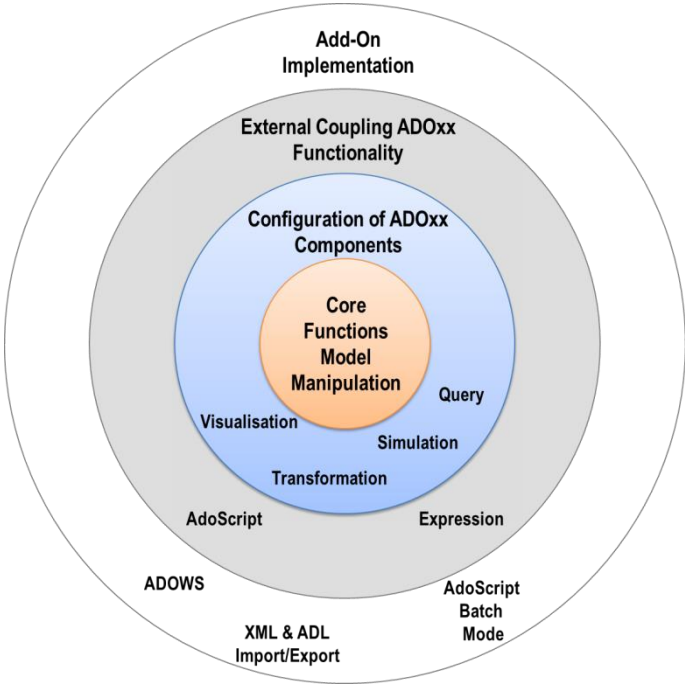
Transformation Example 2

```
# BEGIN set x, y pos
SETL nXpos:(5)
SETL nYpos:(5)
FOR i in:(sObjIDs) {
  # get class ID from class name
  CC "Core" GET_CLASS_ID classname:(sClassnameSource)

  # get the attribute "Name"
  CC "Core" GET_ATTR_ID classid:(classid) attrname:("Name")

  # and show it
  CC "Core" GET_ATTR_VAL objid:(VAL (i)) attrid:(attrid)
  CC "AdoScript" INFOBOX (val)
  SETL sAttrName:(val)

  #Make new model
  CC "Core" GET_CLASS_ID classname:(sClassnameTarget)
  SETL nClassTargetID:(classid)
  CC "Core" debug CREATE_OBJ modelid:(nTargetModelID) classid:(nClassTargetID)
     objname:(sAttrName)
  CC "Core" debug SET_ATTR_VAL objid:(objid) attrname:("Position") val:("x:"
     + STR (nXpos) +"cm y:" + STR (nYpos) + "cm")
  SETL nXpos:(nXpos + 2.5)
}
```

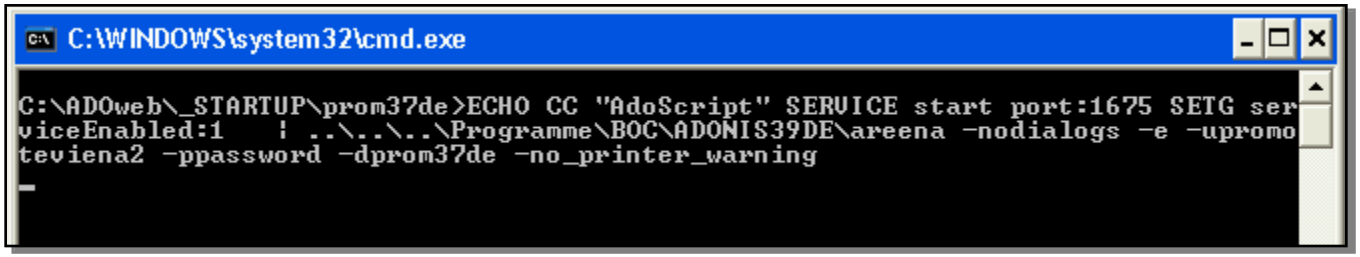


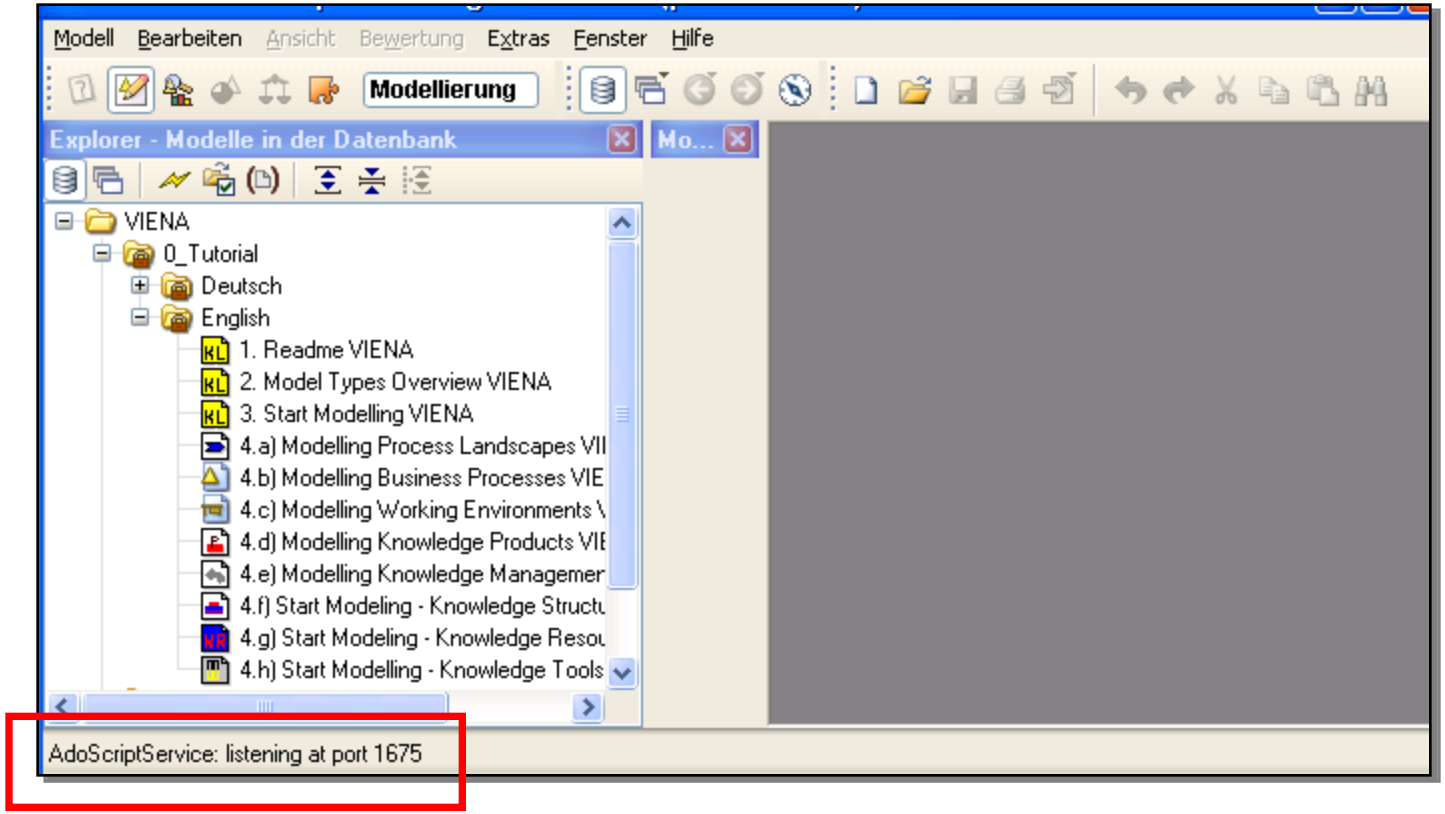
4. ADD-ON IMPLEMENTATION

ADOxx WebService

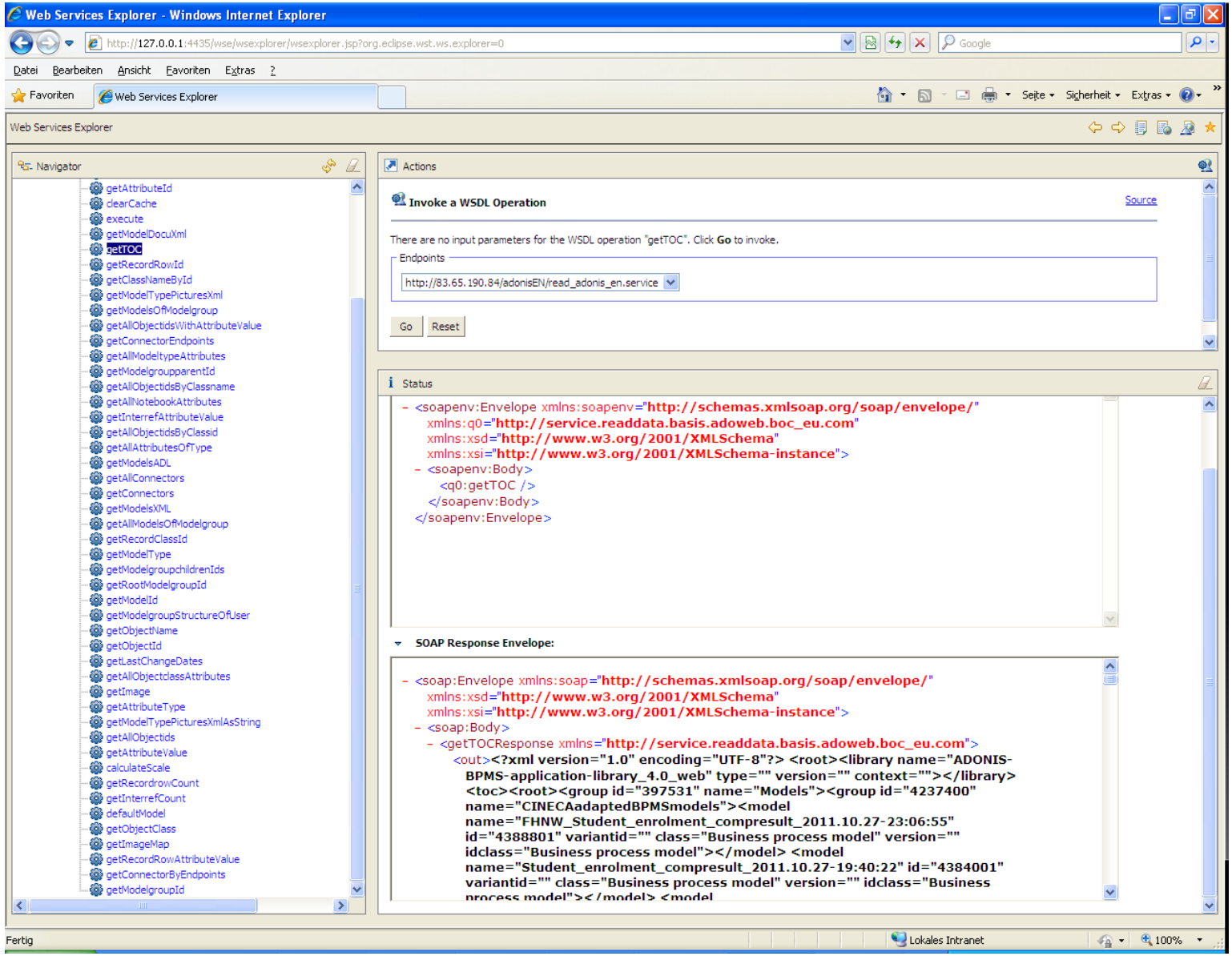
4. ADD-ON IMPLEMENTATION

ADOxx Web Service Port





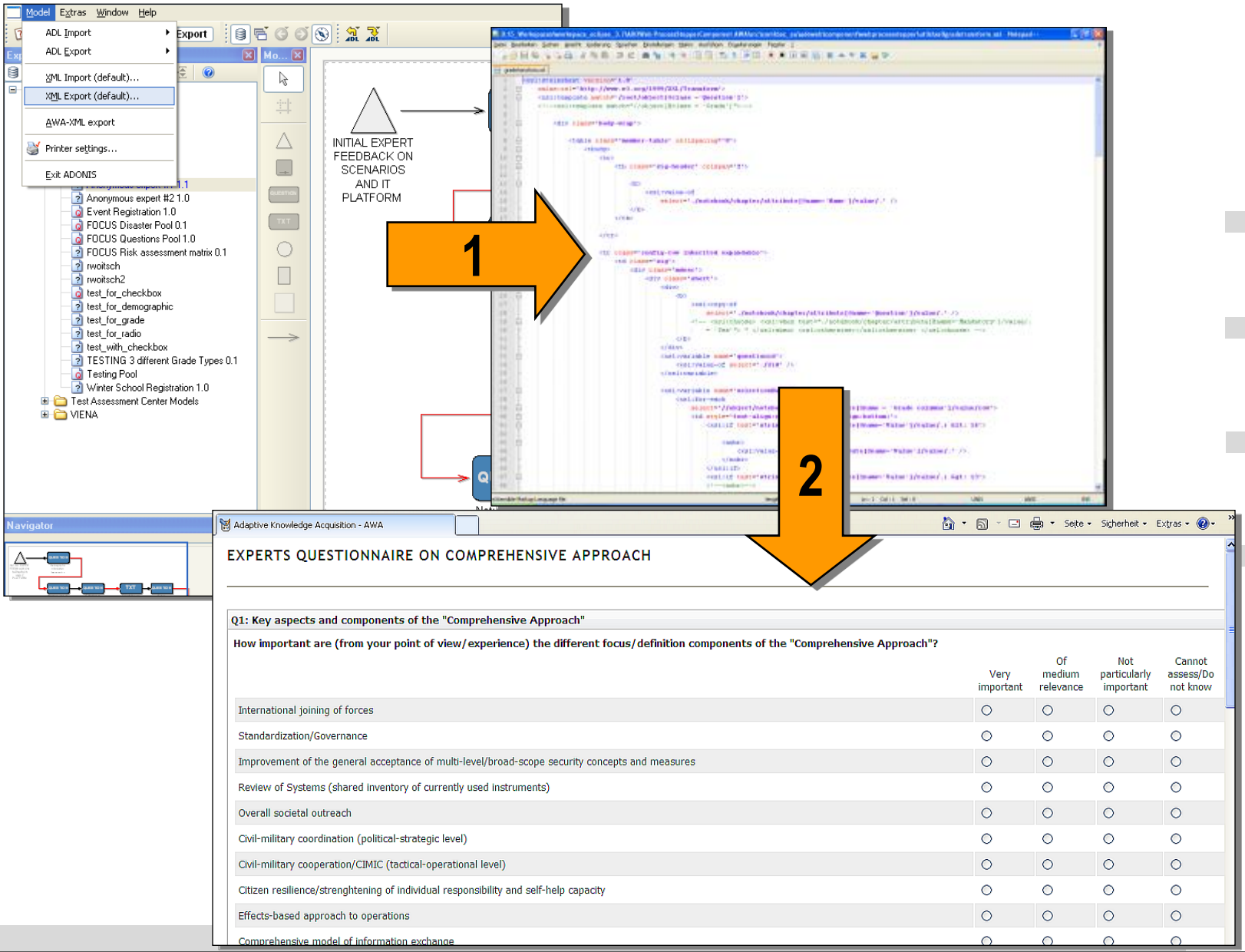
ADOxx Web Service Interaction



XML/ADL IMPORT-EXPORT

4. ADD-ON IMPLEMENTATION

XML – ADD-ON Example



AdoScript BATCH MODE

4. ADD-ON IMPLEMENTATION

Batch Mode - example

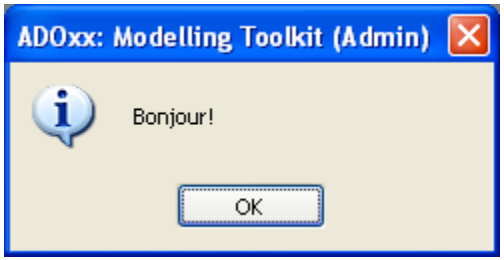
An AdoScript can be specified on the areena command line. It is executed after the initialization of ADOxx is completed. The AdoScript is read from a file or from the standard input stream. In both cases the -e option has to be set, eventually together with a file name specification.

- **Example 1**

```
# Execute the AdoScript contained in startup.asc
areena -uAdmin -ppassword -dado35 -sdb2 -estartup.asc
```

- **Example 2**

```
# Specifying an AdoScript via stdin
echo CC "AdoScript" INFOBOX "Bonjour!" | areena -uAdmin -
ppassword -dado35 -sdb2 -e
```



We thank you for your attention!

In case of any questions, please contact

tutorial@adoxx.org