# MODELLING LANGUAGE IMPLEMENTATION ON ADO*xx*

# What is ADO*xx*?

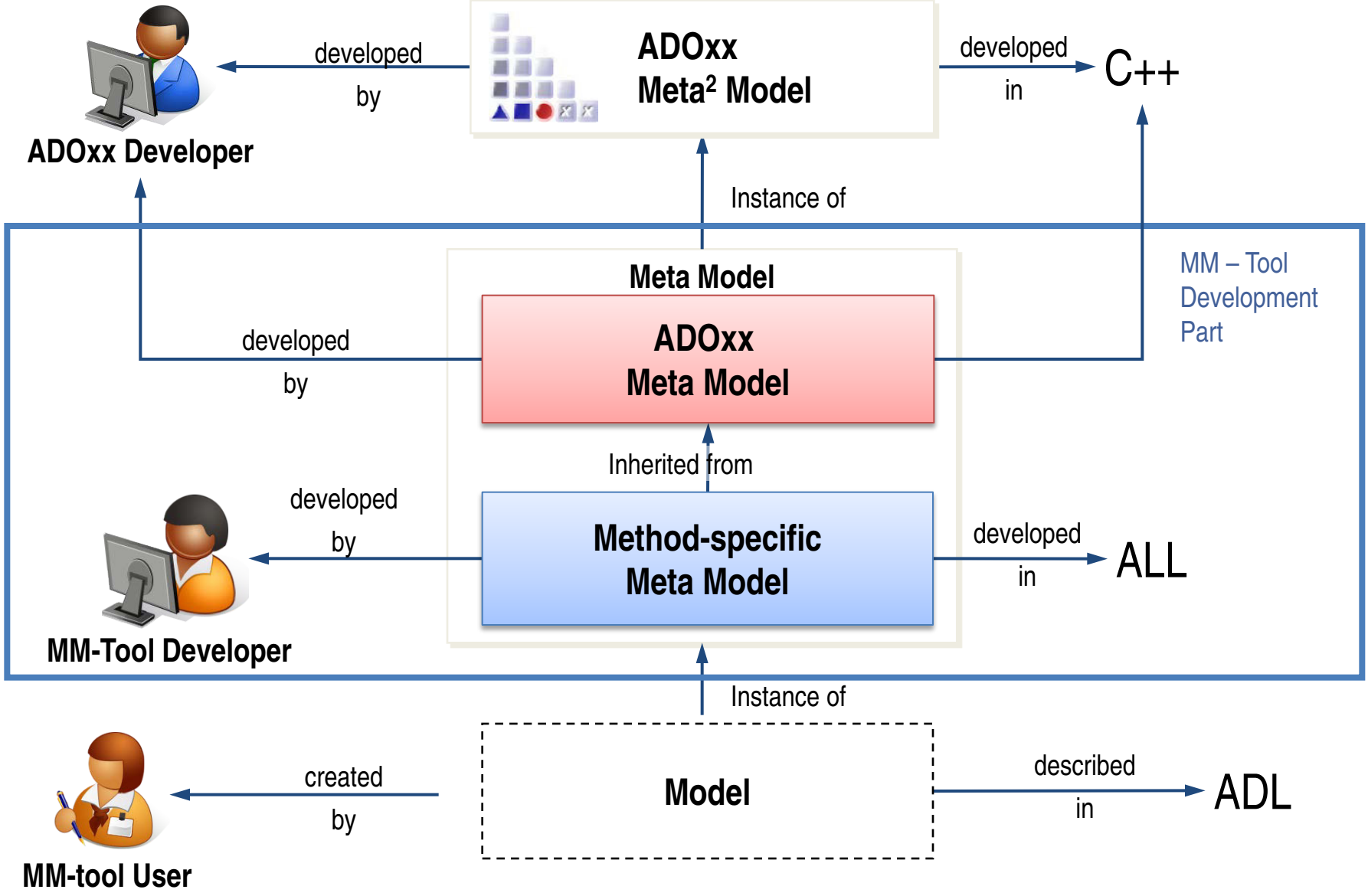
**"ADO*xx* IS A META MODELLING DEVELOPMENT AND CONFIGURATION PLATFORM FOR IMPLEMENTING MODELLING TOOLS."**

| Identified Roles | Major Tasks | Required Skills |
|---|---|---|
|  MM-tool User | Modelling Domain Knowledge | Domain Knowledge<br>Method Knowledge |
|  MM-Tool Developer | Developing an Meta Modelling Tool | Domain Knowledge<br>Method Knowledge<br>Platform Knowledge |
|  ADOxx Developer | Implementation of tool specific and ADOxx functionality | Platform Knowledge<br>ADOxx Technology Skills |

Cases

- Established modelling tools
- Agile development of modelling tool in parallel to modelling tool usage
- Agile development of ADOxx platform in parallel to modelling method development
- . . .

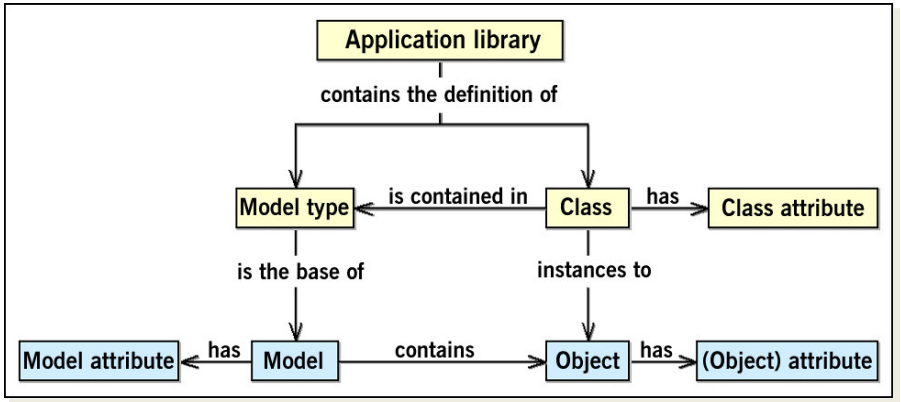
# Meta Modelling Platforms Hierarchy: ADO*xx*




MM … Modelling Method

# Introduction of ADOxx:

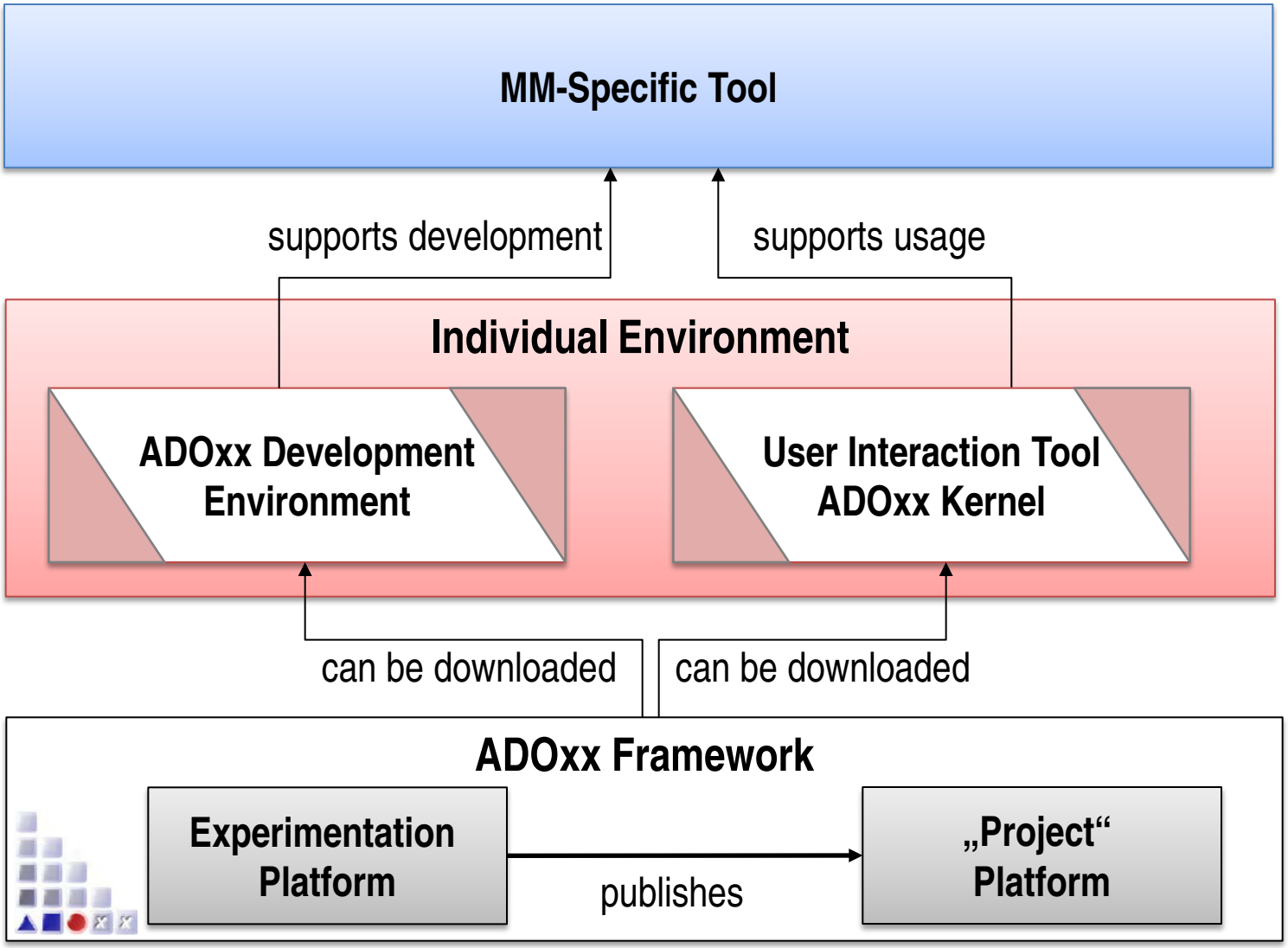
## Definition: Model types, Classes, Attributes and Relations



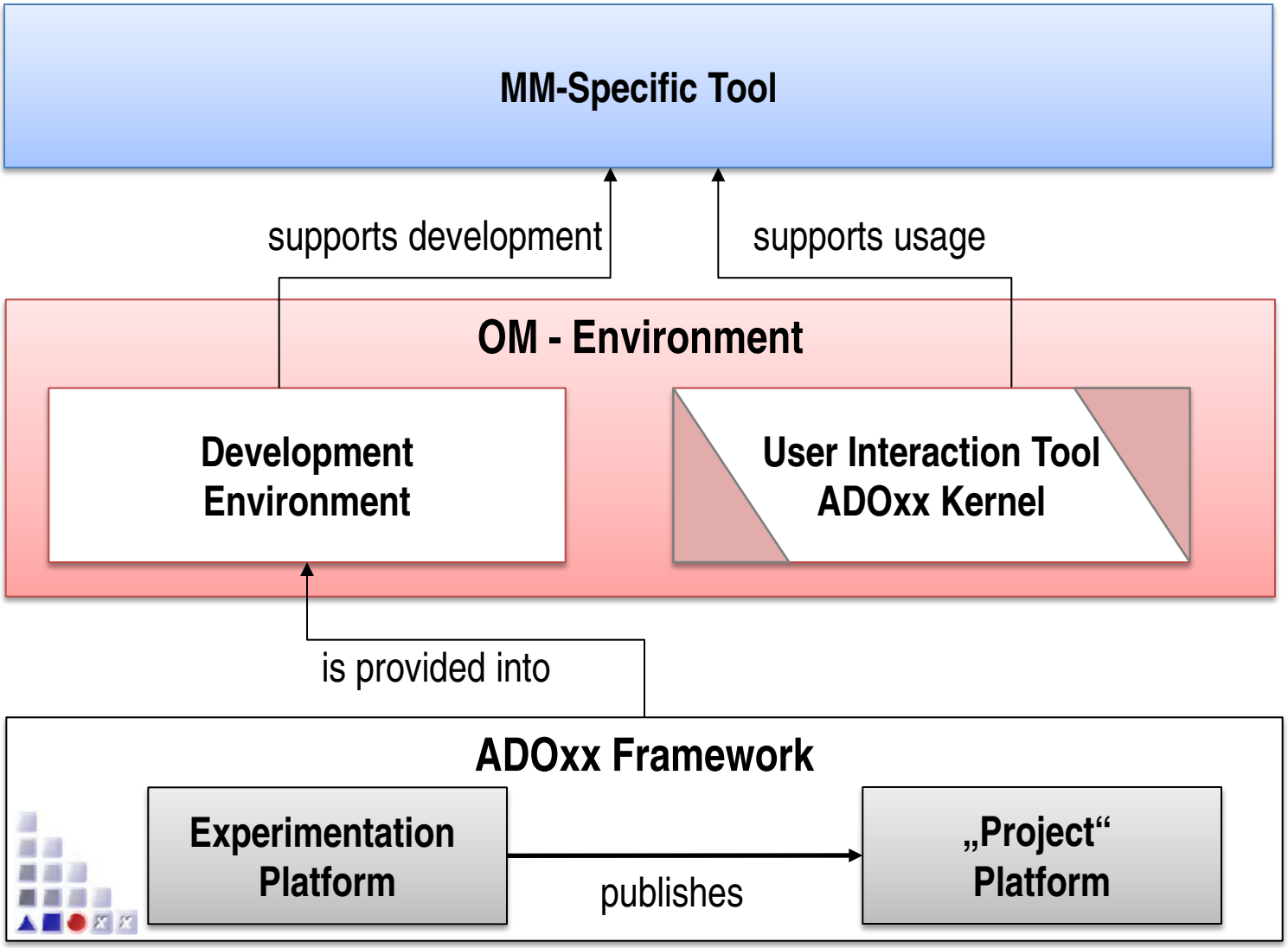

- **Model Types:**
  A model type is a well-defined sub collection of classes and relation classes of a meta model.
- **Classes:**
  A class is a construct that is used as a template to create objects of that class. The objects of a class are alternatively called "instances"
- **Attributes:**
  An attribute is a property of a modelling construct such as a model, object or relation. Each attribute has a type and a value.
- **Relations:**
  A relation class is a construct that is used as a template to create relations between objects. A relation class is defined between classes. A relation is always a directed connection between objects, i.e. each relation has a from-side and a to-side.

# SETUP OF IMPLEMENTATION ENVIRONMENT

# Individual Development Environment from ADOxx.org



# Laboratory Development Environment at OMiLAB

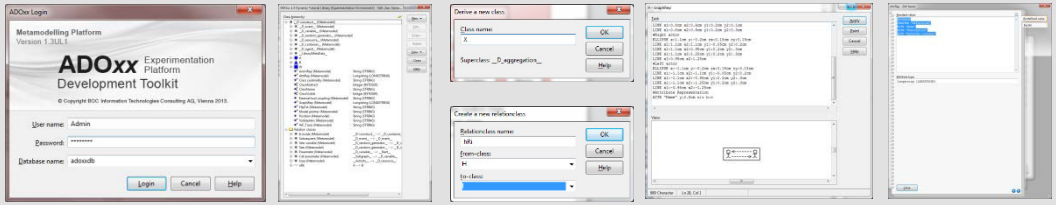


# DEVELOPMENT APPROACHES

## Configuration and Implementation Approach

"Interactive Style"

### Configuration Approach



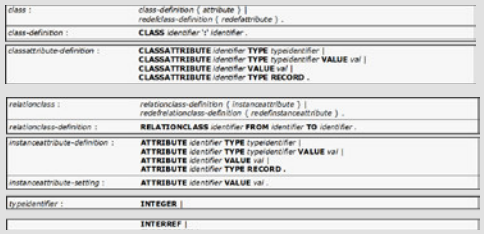

**ADOxx Configuration Tools**

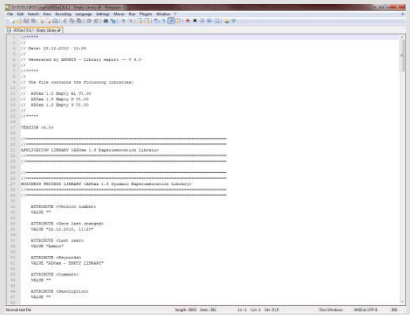
Class Hierarchy Management | Graphrep Notation Editor | AttrRep Notation Editor | Modeltype/View Configuration | Library Validation | External coupling

Metamodel Engineer

„Programming Style"

### Implementation Approach





**ADOxx Library Language Implementation Tools**

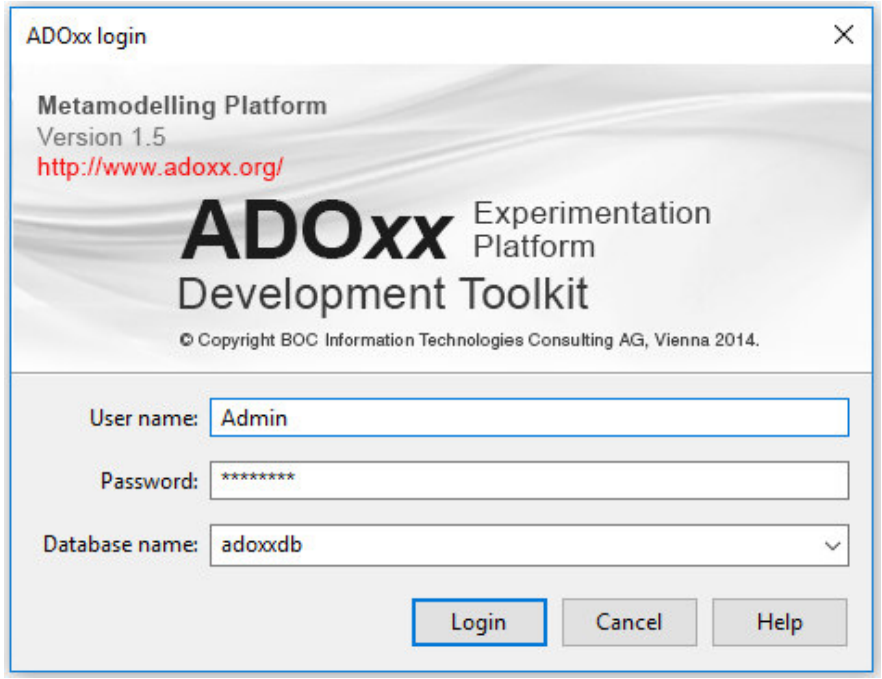
Syntax Highlight AutoComplete | Supporting Development Services

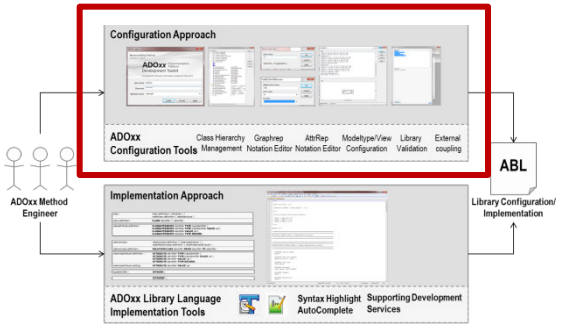
**ABL**

Library Configuration/ Implementation

# Administration Toolkit - STARTUP

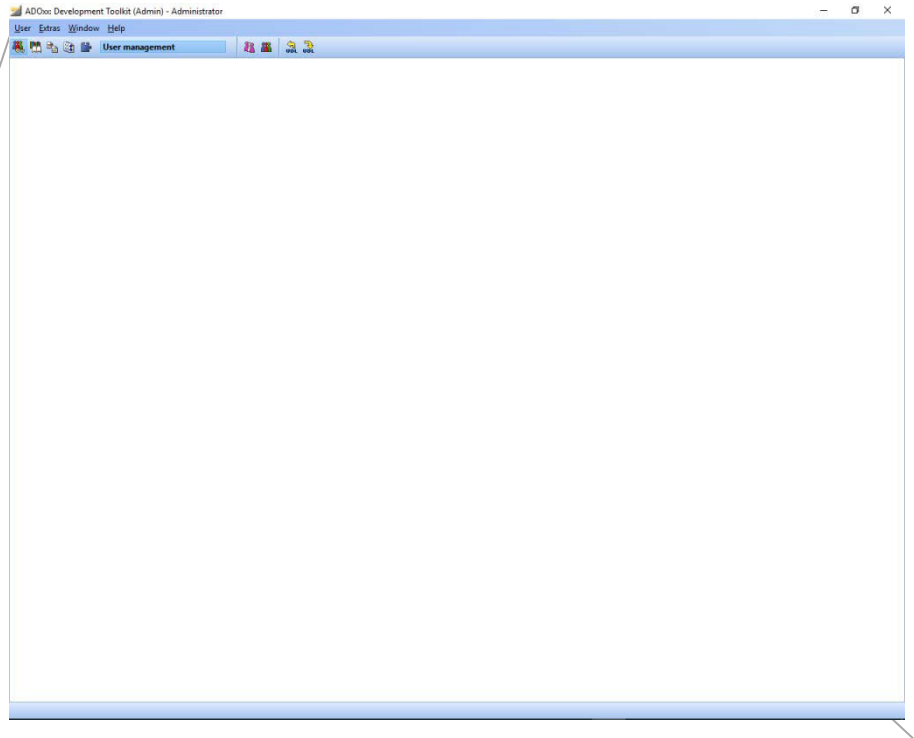
1. Start Administration Toolkit
2. Login into Administration Toolkit
3. Default Development User
4. Username: Admin
5. Password: password
   DB: adoxxdb
6. BACKGROUND: connection to experimentation database hosted on a server platform



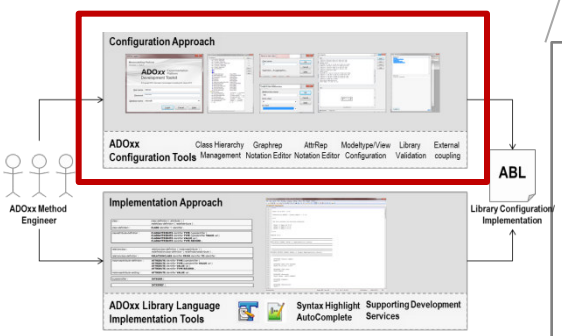



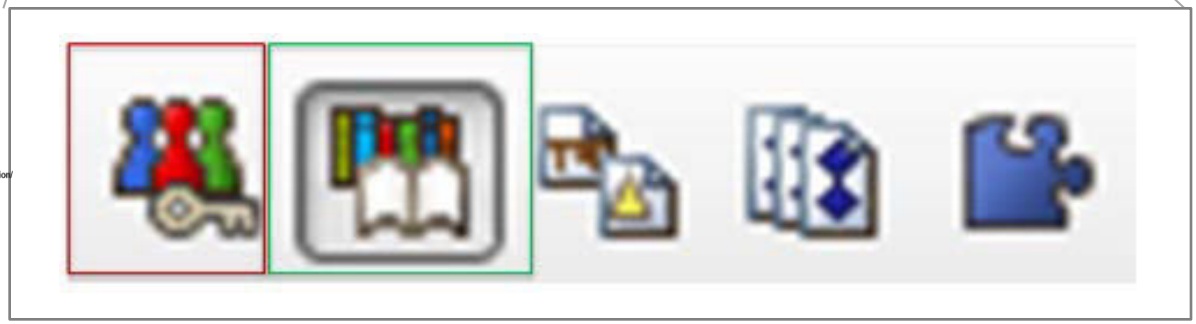
# Development Toolkit - Components

Debug User needed in the database to start modelling toolkit for validation
U: debug
P: debug
Create user in “User Management” component for testing purposes



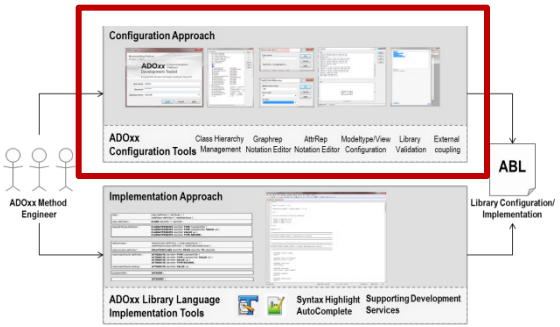

Development Environment: Library Management Component

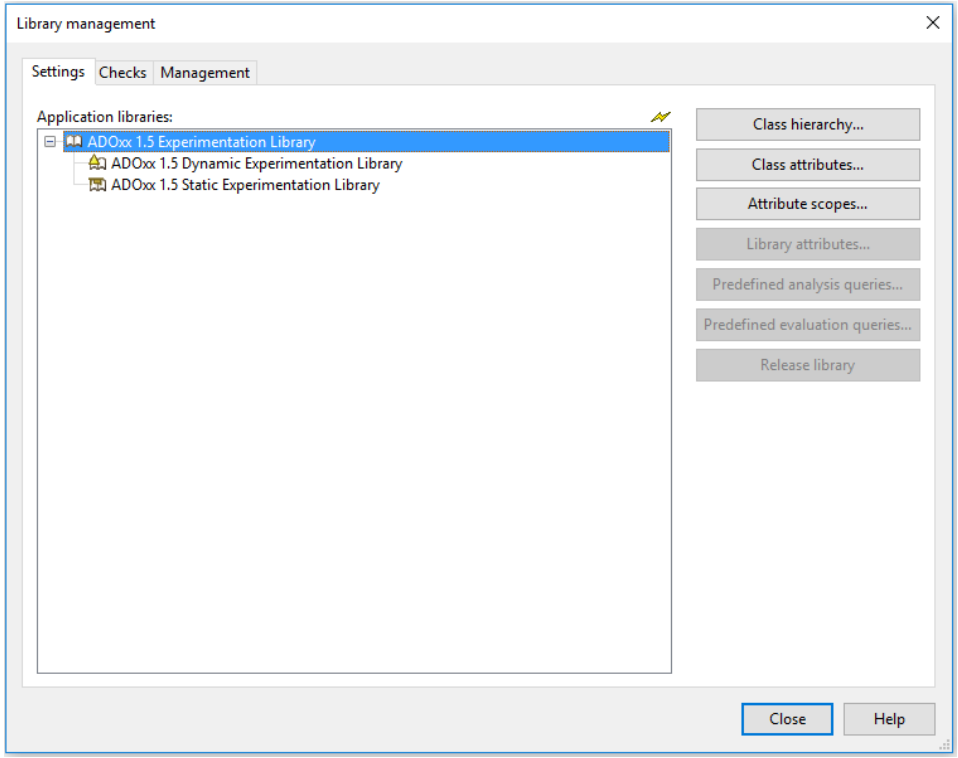


# ADOxx Experimentation Library

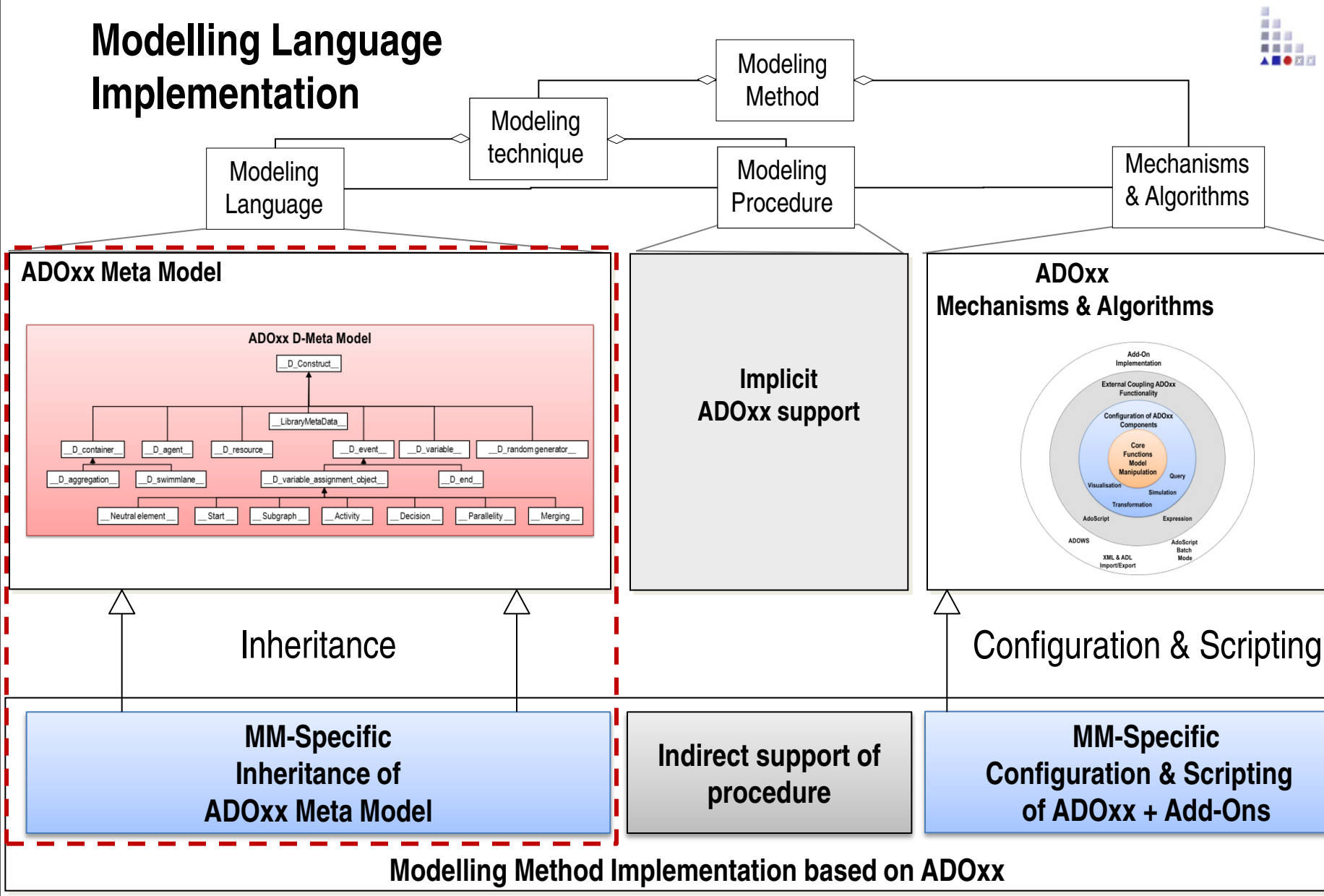
- Development aggregated in “Application Library” consisting of Static and Dynamic sub-library
  - **Dynamic:** ADOxx 1.5 Dynamic Experimentation Library
  - **Static:** ADOxx 1.5 Static Experimentation Library



# MODELLING LANGUAGE IMPLEMENTATION

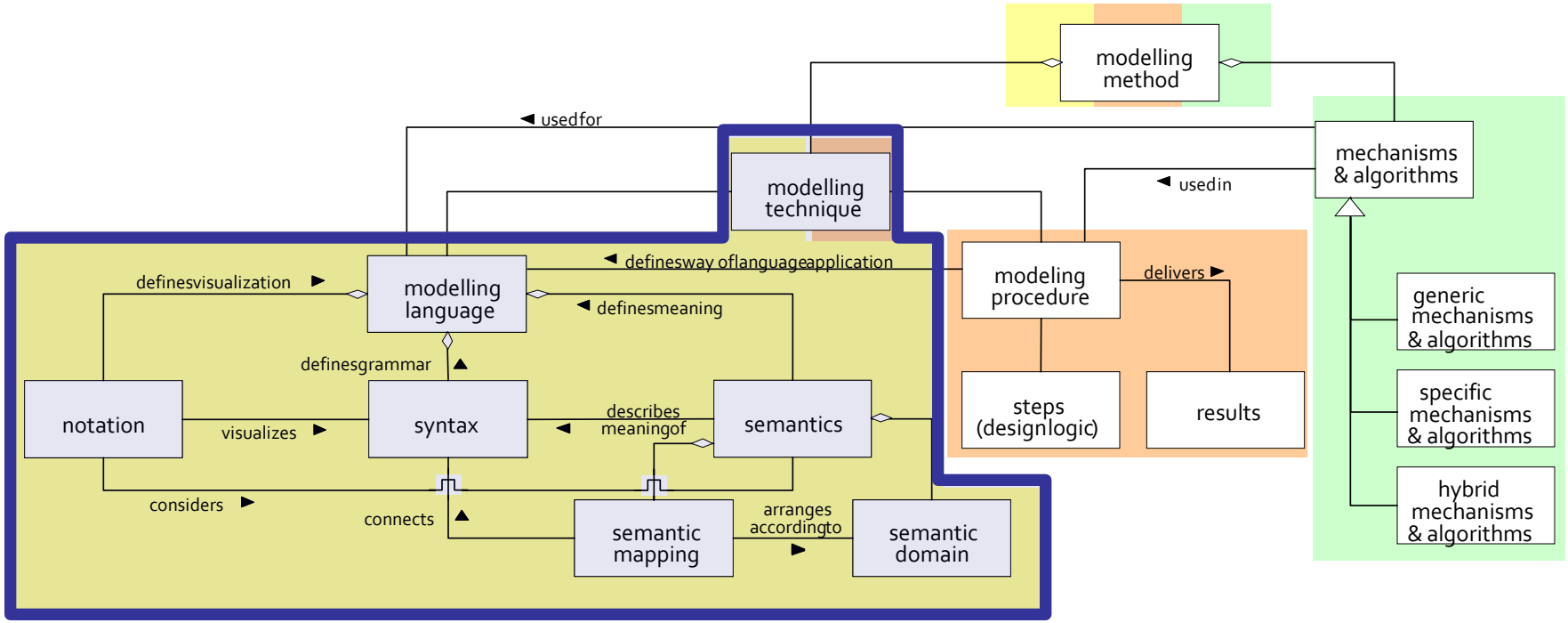
# Modelling Language Implementation




MM … Modelling Method

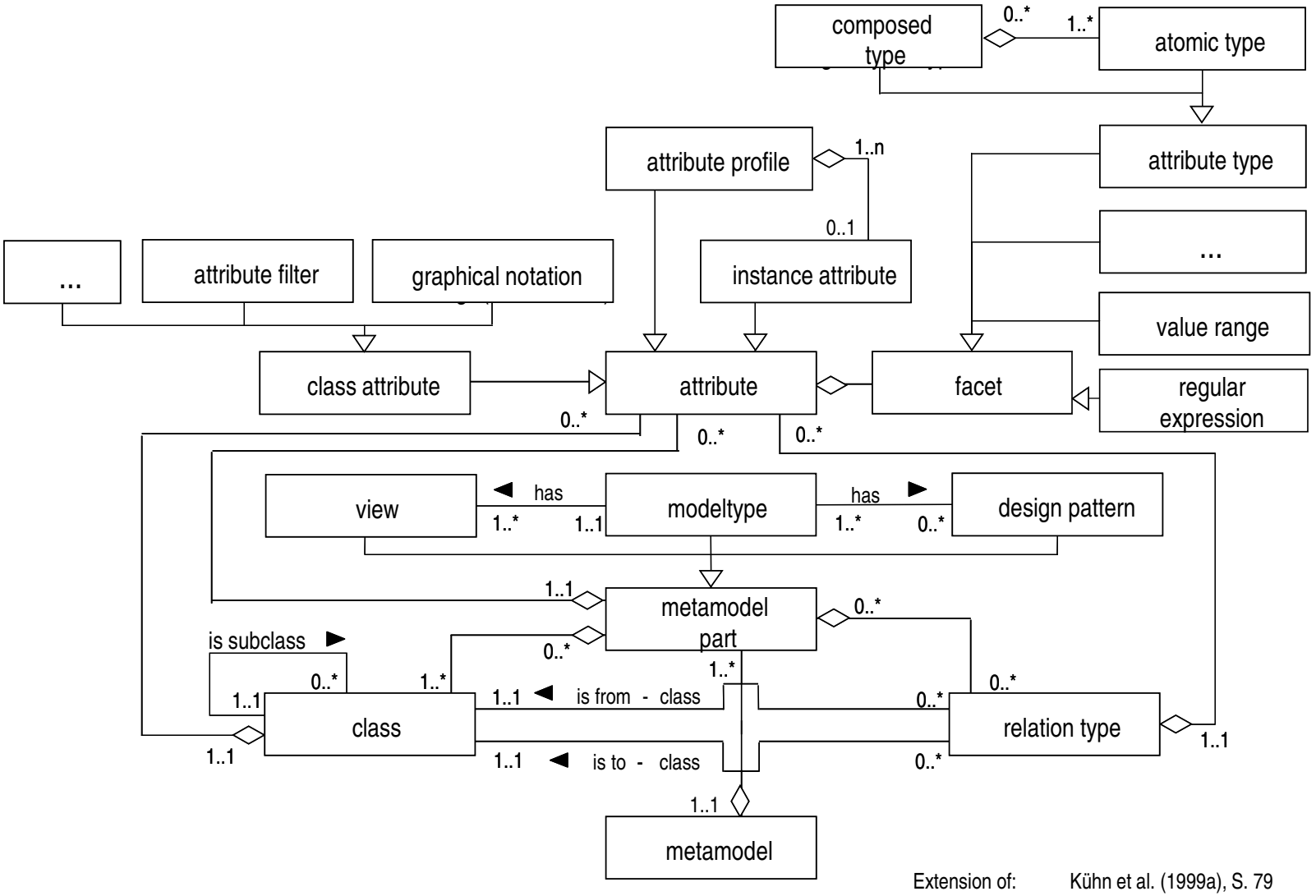
Reference: Kühn, H. (2004). Methodenintegration im Business Engineering. PhD Thesis, University of Vienna

# Generic Modelling Method Framework



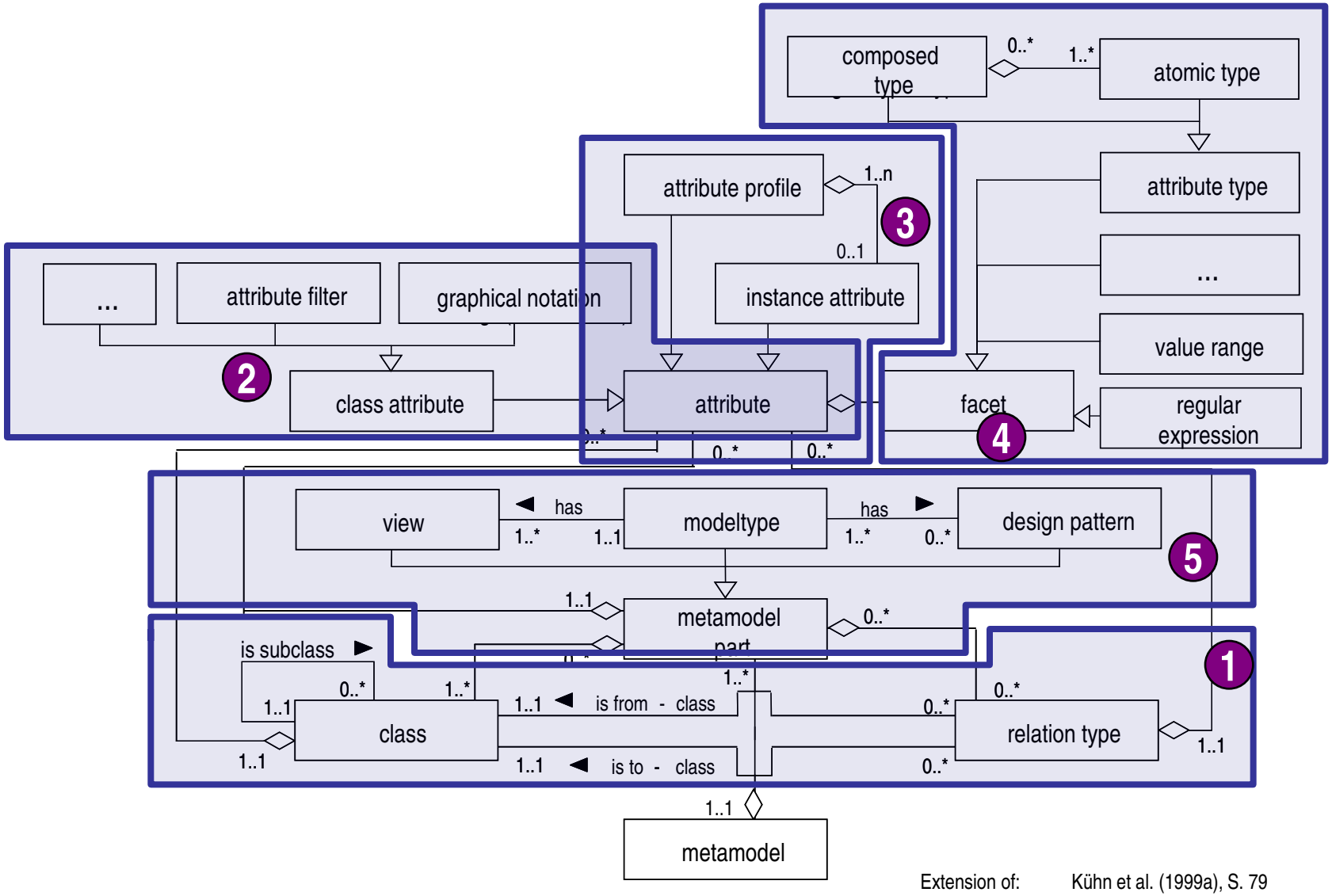

Reference: Karagiannis, D., Kühn, H.: „Metamodelling Platforms“. In Bauknecht, K., Min Tjoa, A., Quirchmayer, G. (Eds.): Proceedings of the Third International Conference EC-Web 2002 – Dexa 2002, Aix-en-Provence, France, September 2002, LNCS 2455, Springer, Berlin/Heidelberg, p. 182 ff.

# Meta Model of Meta Modelling Language

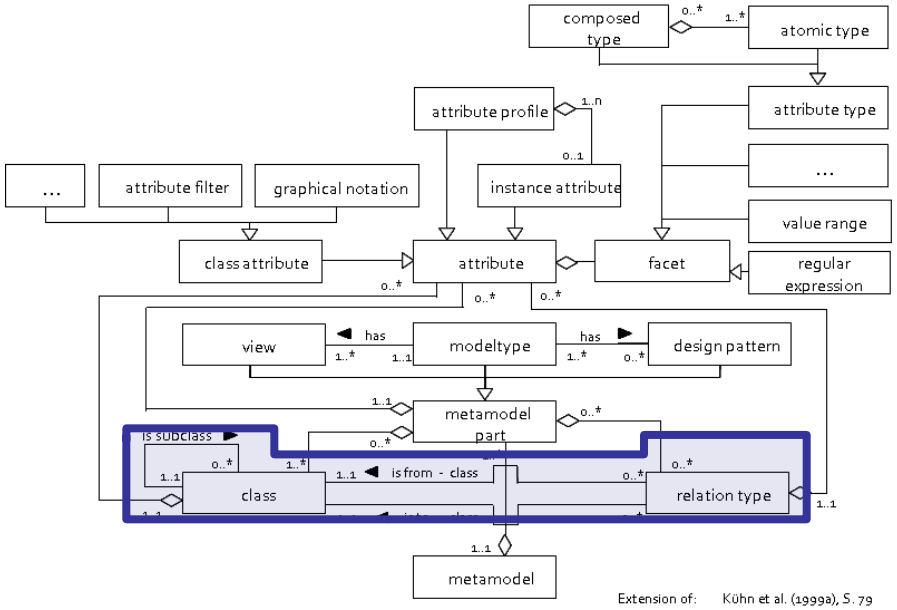



Extension of: Kühn et al. (1999a), S. 79

# Meta Model of Meta Modelling Language

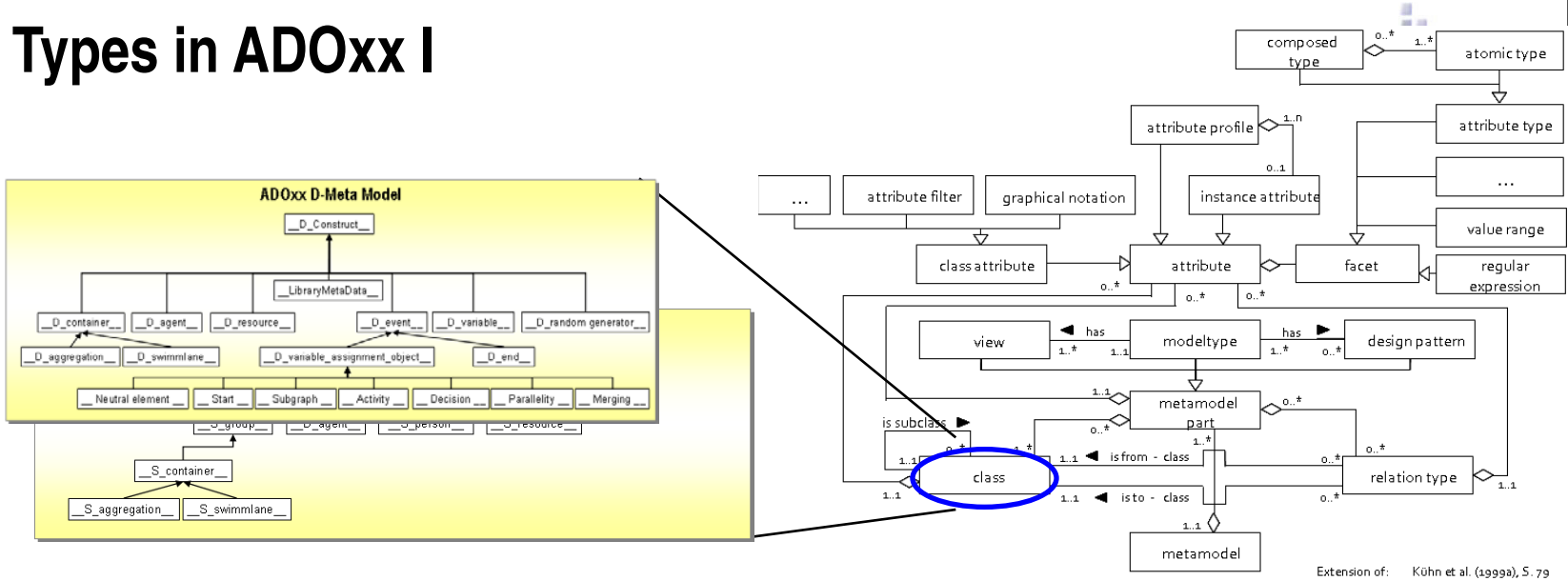



Extension of: Kühn et al. (1999a), S. 79




# 1. CLASSES and RELATIONS

# Class Types in ADOxx I




- **Pre-defined Abstract Classes (ADOxx meta model class)**
  - Pre-defined abstract classes are classes that are provided by ADOxx with a given semantic and basic syntax in form of attributes. They can be used to inherit the pre-defined syntax and the attributes to either self-defined abstract classes or to classes.
  - ADOxx functionality that is provided for the pre-defined abstract classes can be used for any inherited concrete class. Hence pre-defined and provided ADOxx functionality is consumed due to inheritance of such pre-defined abstract classes.
  - Pre-defined abstract classes are the ADOxx meta model, hence they exist in every meta model based on ADOxx.
  - Nomenclature: __ Class Name __

# Class Types in ADOxx II

- **Abstract Classes**
  - Abstract classes are self-defined classes enabling to structure the meta model and define syntax in form of attributes and semantic, which is inherited by sub-classes.
  - Abstract classes either inherit from the root class of the meta model, or from any other class of the meta model. Hence, they inherit the behaviour from their super-class – which is often a pre-defined abstract class from the ADOxx meta model.
  - Abstract classes enable an efficient meta model, hence they may not be in every ADOxx meta model.
  - Nomenclature: _ Class Name _
- **(Concrete) Classes**
  - Classes are self-defined classes defining a concrete modelling class that can be used, when applying the corresponding modelling language. Hence all model objects in every model created on ADOxx is an instance of a class.
  - Classes inherit the semantic and the attributes from the Pre-defined abstract class and additionally - in case of inheriting - from the abstract class.
  - Classes enable the realisation of a concrete meta model.
  - Nomenclature: Class Name

# Selected Pre-defined ADOxx classes for a "Graph-based environment " I

- __ D_Construct ___
  - Super class for „graph-based“ pre-defined meta model.
- __ D_Container __
  - Container class provide the relation „is-inside“, hence every object a drawn on the model having its x/y coordinates within the drawing area of any container b has the relation a Ris-inside b.
- __D_aggregation__
  - Aggregation inherits from __D_Container__, hence also provides the „is-inside“ relation and enables a self-defined „drawing area“. E.g. resizable rectangle.
- __D_swimmlane__
  - Swimmlane inherits form __D_Container__, hence also provides the „is-inside“ relation but only enables either rows (x=0 to x= maximum) or columns (y= 0 to y= maximum) as possible „drawing area“. E.g. three columns one for input, one for processing, one for output

# Selected Pre-defined ADOxx classes for a "Graph-based environment " II

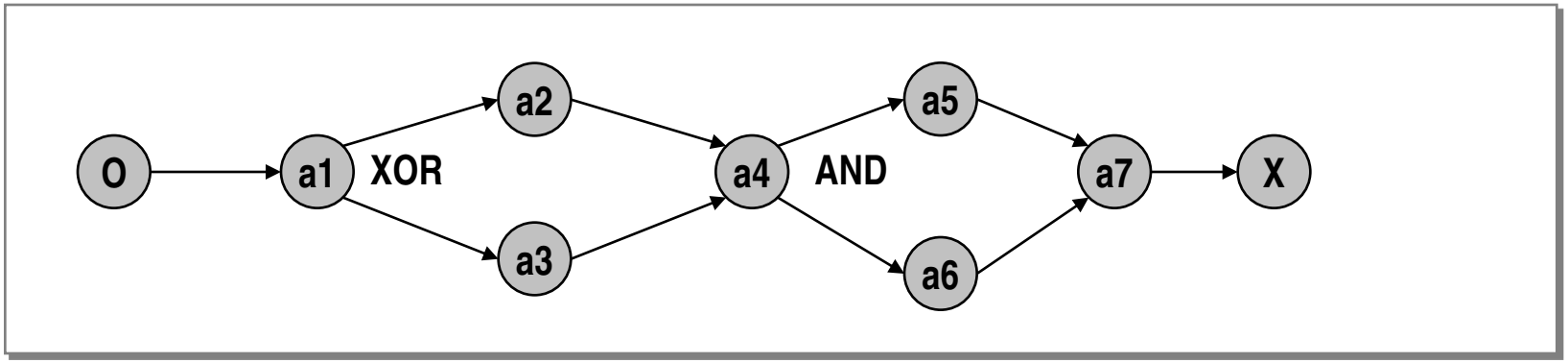
- **__ D_Event ___**
  - Event encapsulates all possible notes of a graph and distinguishes between “D_variable_assignment_object” and “D_end”.
- **__ D_end __**
  - The end concludes the graph and finishes state changes.
- **__D_variable_assignment_objects__**
  - Variable assignment objects enable the change of the state. The state is stored in variables, hence each of the following concepts have the potential to change the status of variables within a graph:
  - Neutral element, start, sub graph, activity, decision, parallelity, merging
- **__D_Neutral element__**
  - Neutral elements do not participate in executing the graph but only display references or state the status.
- **__D_Start__**
  - Start is the starting node of the graph.

# Selected Pre-defined ADOxx classes for a "Graph-based environment " III

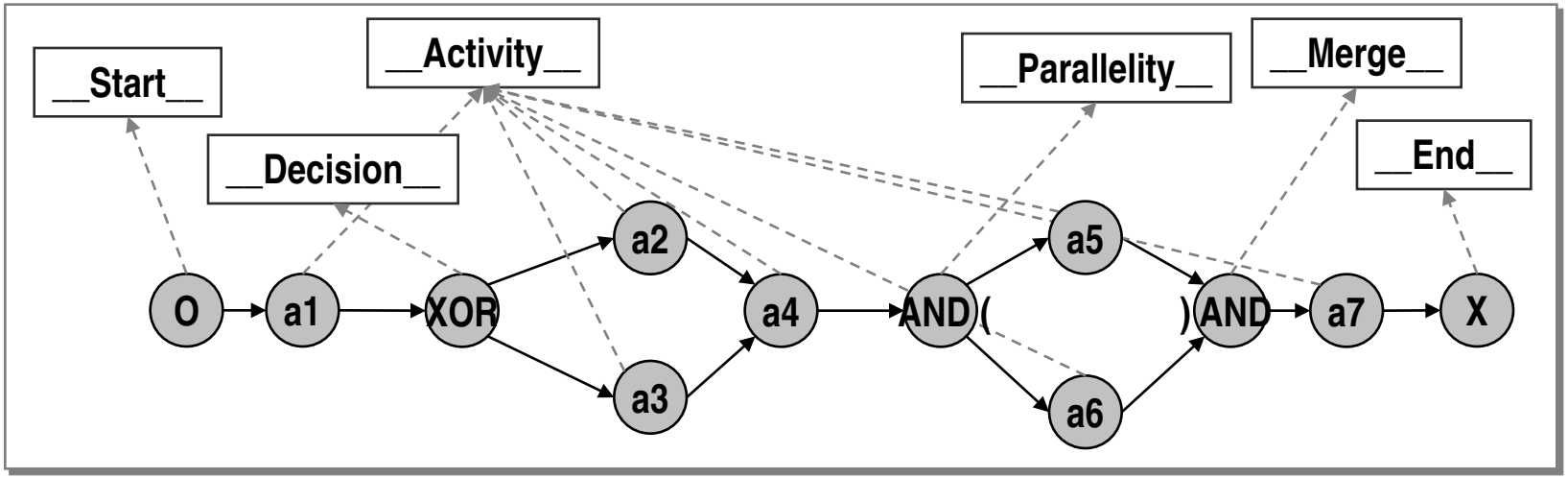
- __ Subgraph ___
  - Subgraph substitutes a sub-graph in the graph to make complex graphs more readable. Technically the subgraph is a pointer to another graph.
- __ Activity__
  - Activity is a node in the graph that performs the typical actions the graph is designed for. Activities are transforming input into output.
- __Decisions__
  - Decisions split the graph in several alternative paths.
- __Parallelity__
  - Parallelity starts a synchronized path of a graph.
- __Merging__
  - Merging ends a synchronized path of a graph.

# Selected Pre-defined ADOxx classes for a "Graph-based environment" IV

## Sample Graph



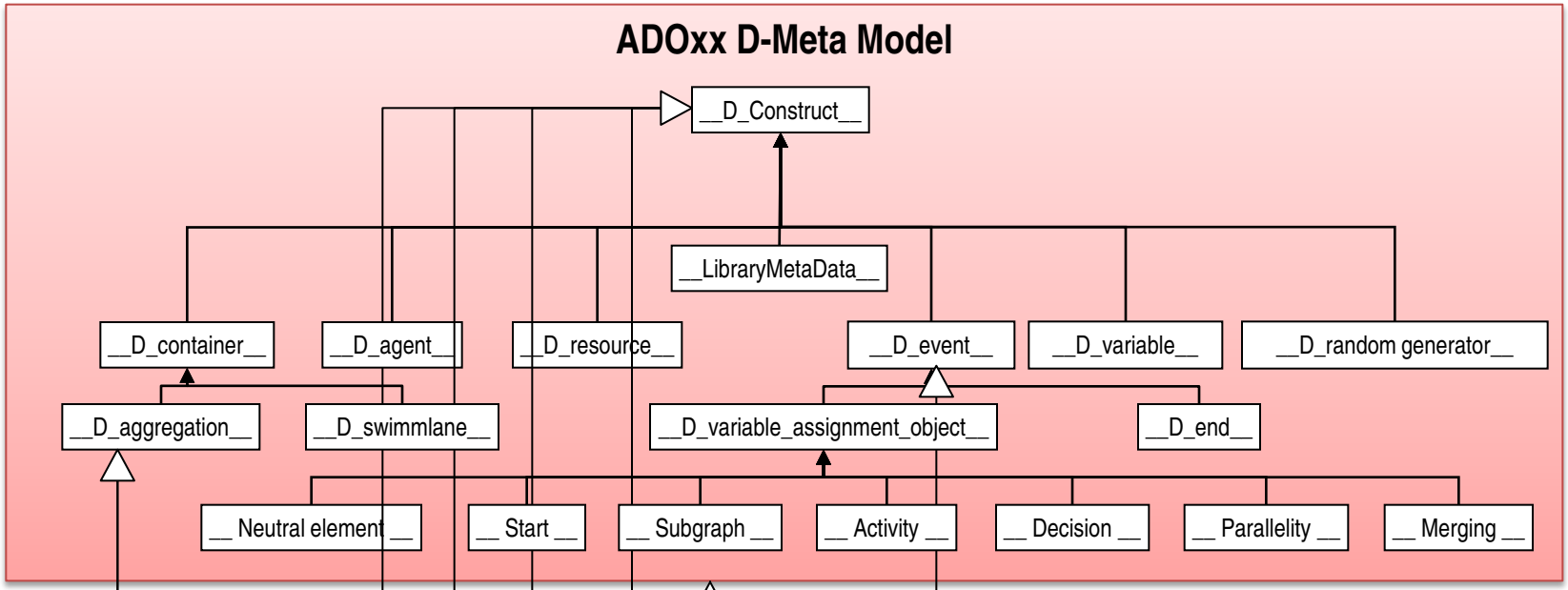

Possible mapping of graph to ADOxx meta model



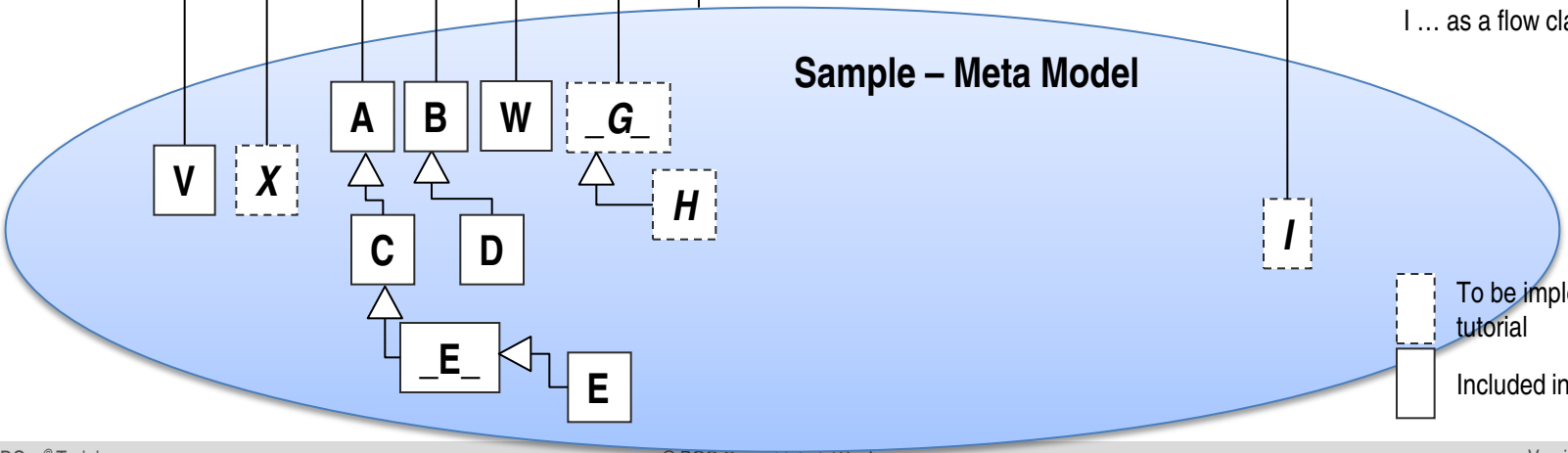
# Selected Pre-defined ADOxx classes for a "Graph-based environment" V

- **__ D_variable ___**
  - Variables are objects that store a certain status of the graph. Hence different variables can be defined, describing different aspects of a graph.
- **__ D_random_generator __**
  - Random generator creates random figures that can be assigned to variables. This is used for simulation.
- **__D_resources__**
  - Resources are properties of graph-nodes represented in an own class hierarchy. Hence descriptive properties need not only be defined as attributes of graph nodes but can be described as classes using class hierarchy from resources.

# Inheritance/Dependencies of ADOxx Dynamic Metamodel




**Inheritance of a sample meta model**




X … as a container class
_G_ … as an abstract class
H … as a modelling class
I … as a flow class

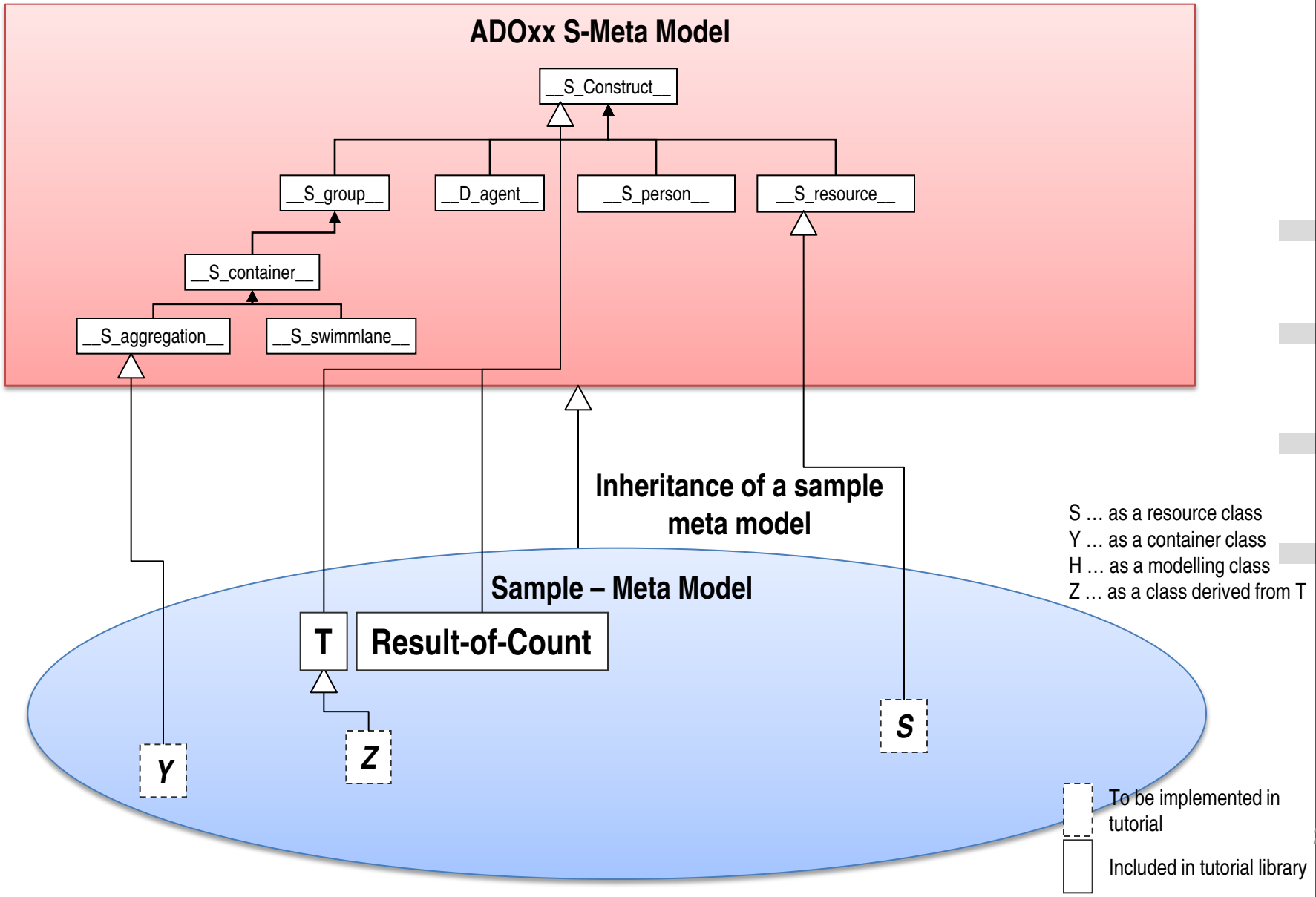
To be implemented in tutorial

Included in tutorial library

# Selected Pre-defined ADOxx classes for a "Tree-based environment"

- **__ S_Construct ___**
  - Super class for „hierarchy" pre-defined meta model.
- **__S_Group__**
  - Group is a tree node
- **__ S_Container __, __S_aggregation__, __S_swimmlane__**
  - Is a special form of a tree-node, same as in __D_Container__
- **__S_resource__**
  - Resources are properties of tree-nodes represented in an own class hierarchy. Hence descriptive properties need not only be defined as attributes of tree nodes but can be described as classes using class hierarchy from resources.
- **__S_person__**
  - In case persons are represented a special class is reserved for implementing person depending behaviour (privacy etc.).

# Inheritance/Dependencies of ADOxx Static Metamodel



# Realisation of Meta Model

## Specification of a meta model in ALL

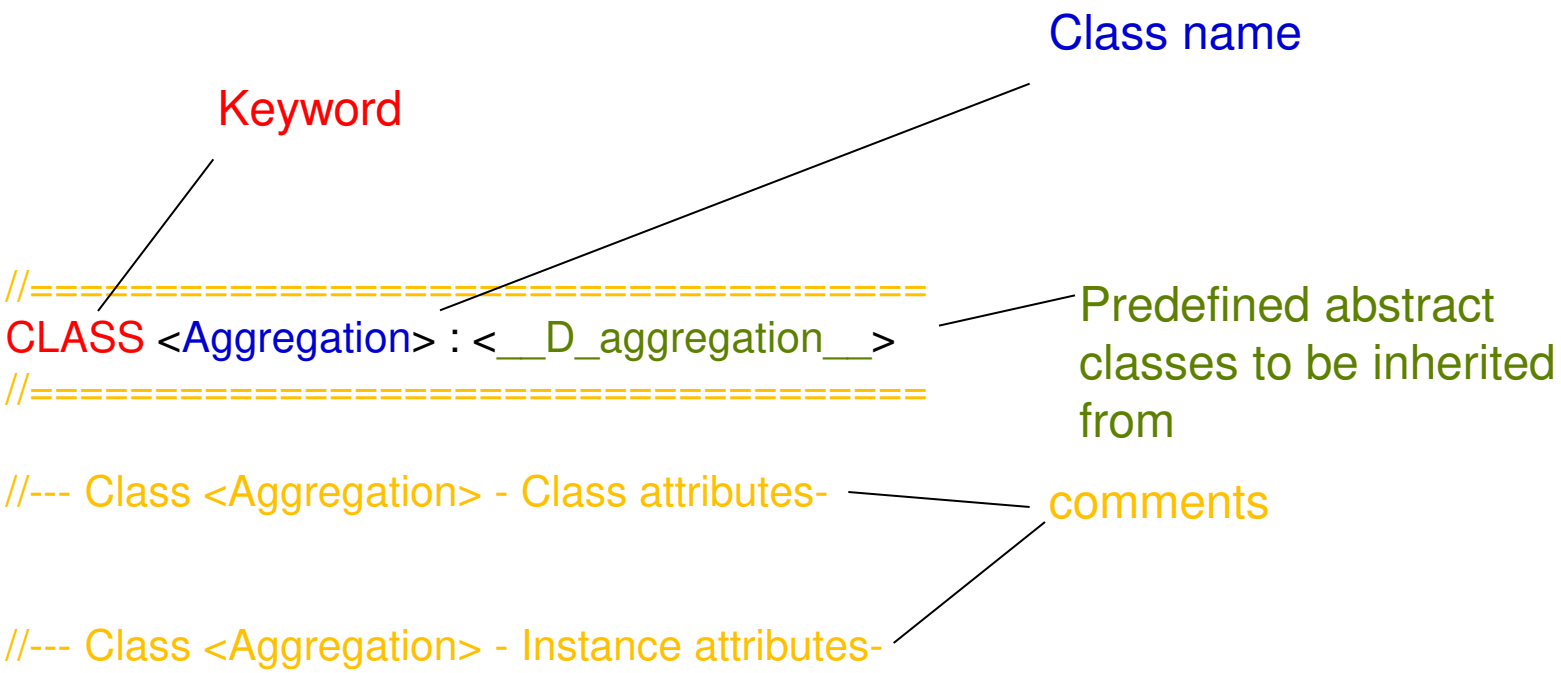
1. Specify the meta model starting from the „Empty Meta Model“ and add classes etc. with ALL using a text editor. Abstract class is defined by the classattribute isabstract.

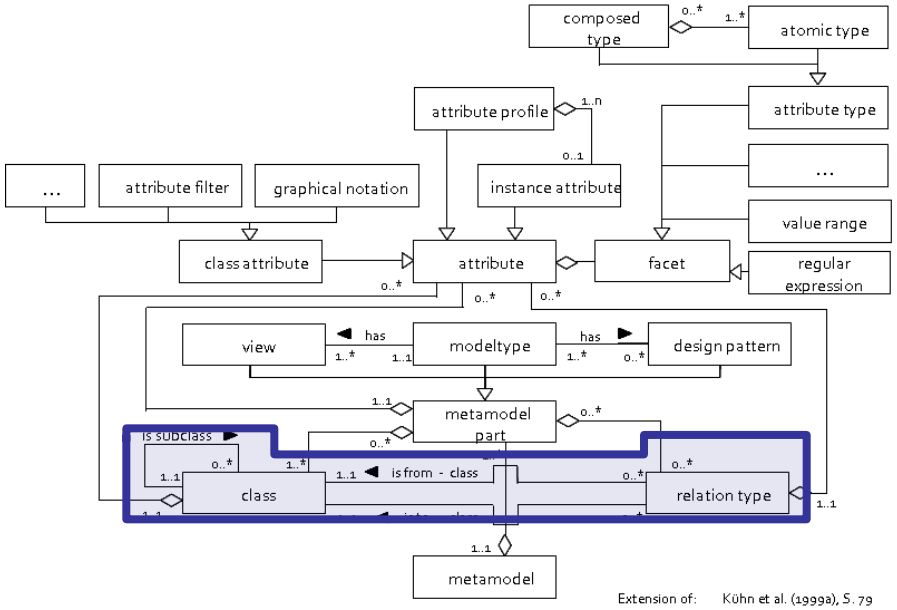
2. Translate ALL into the ADOxx interpretable ABL format and import the meta model into ADOxx.

| | |
|---|---|
| *class* : | *class-definition* { *attribute* } \|<br>*redefclass-definition* { *redefattribute* } . |
| *class-definition* : | **CLASS** *identifier* '**:**' *identifier* . |

| | |
|---|---|
| *classattribute-definition* : | **CLASSATTRIBUTE** *identifier* **TYPE** *typeidentifier* \|<br>**CLASSATTRIBUTE** *identifier* **TYPE** *typeidentifier* **VALUE** *val* \|<br>**CLASSATTRIBUTE** *identifier* **VALUE** *val* \|<br>**CLASSATTRIBUTE** *identifier* **TYPE RECORD .** |

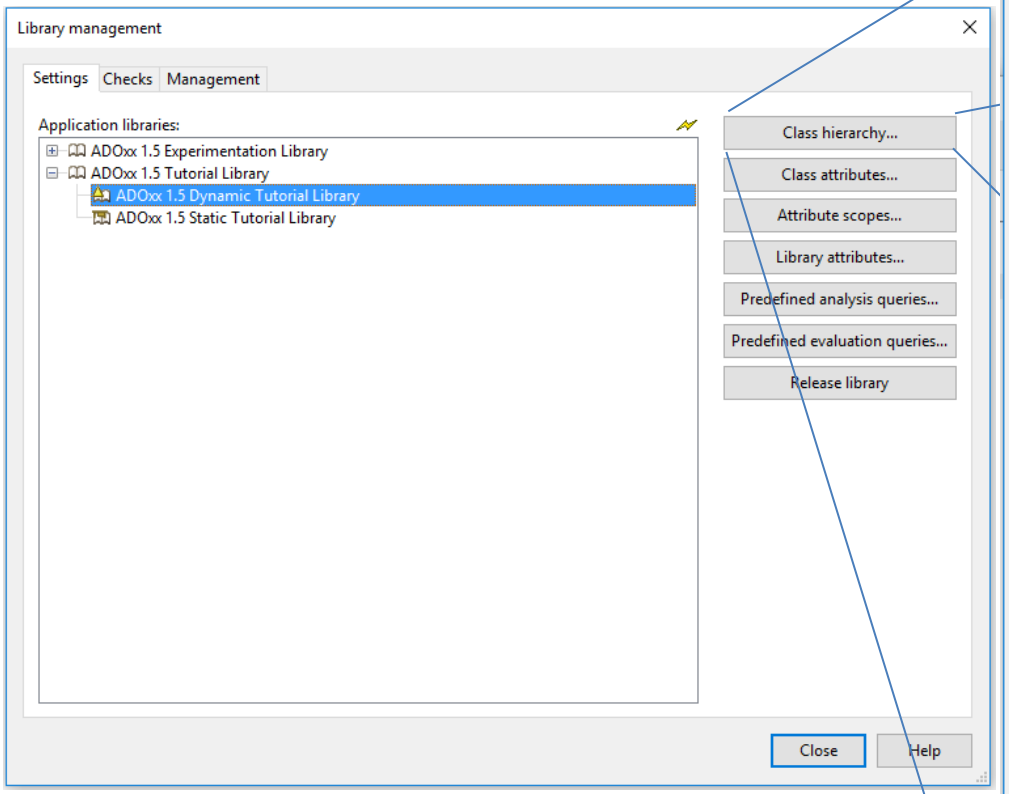
# Definition of a Modeling Class

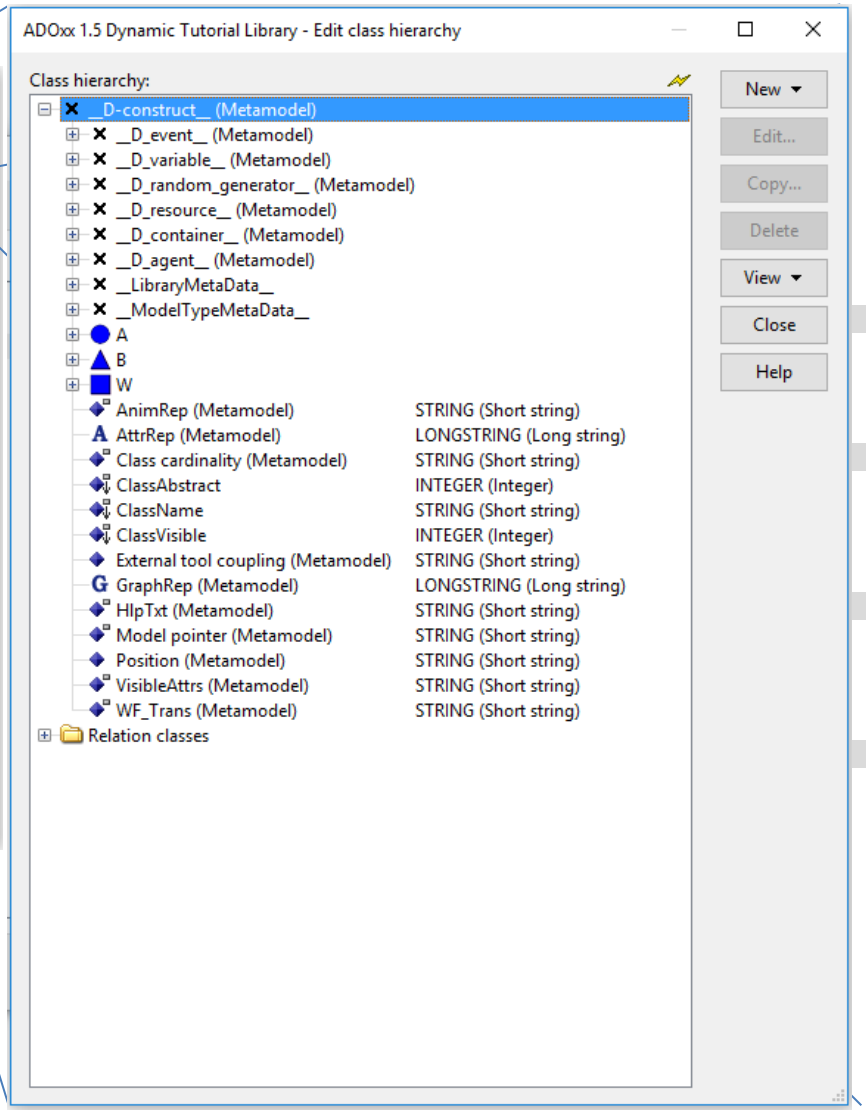


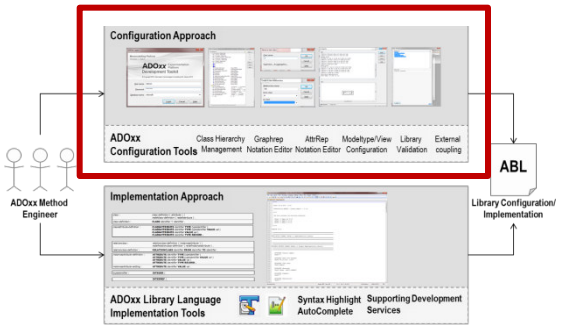


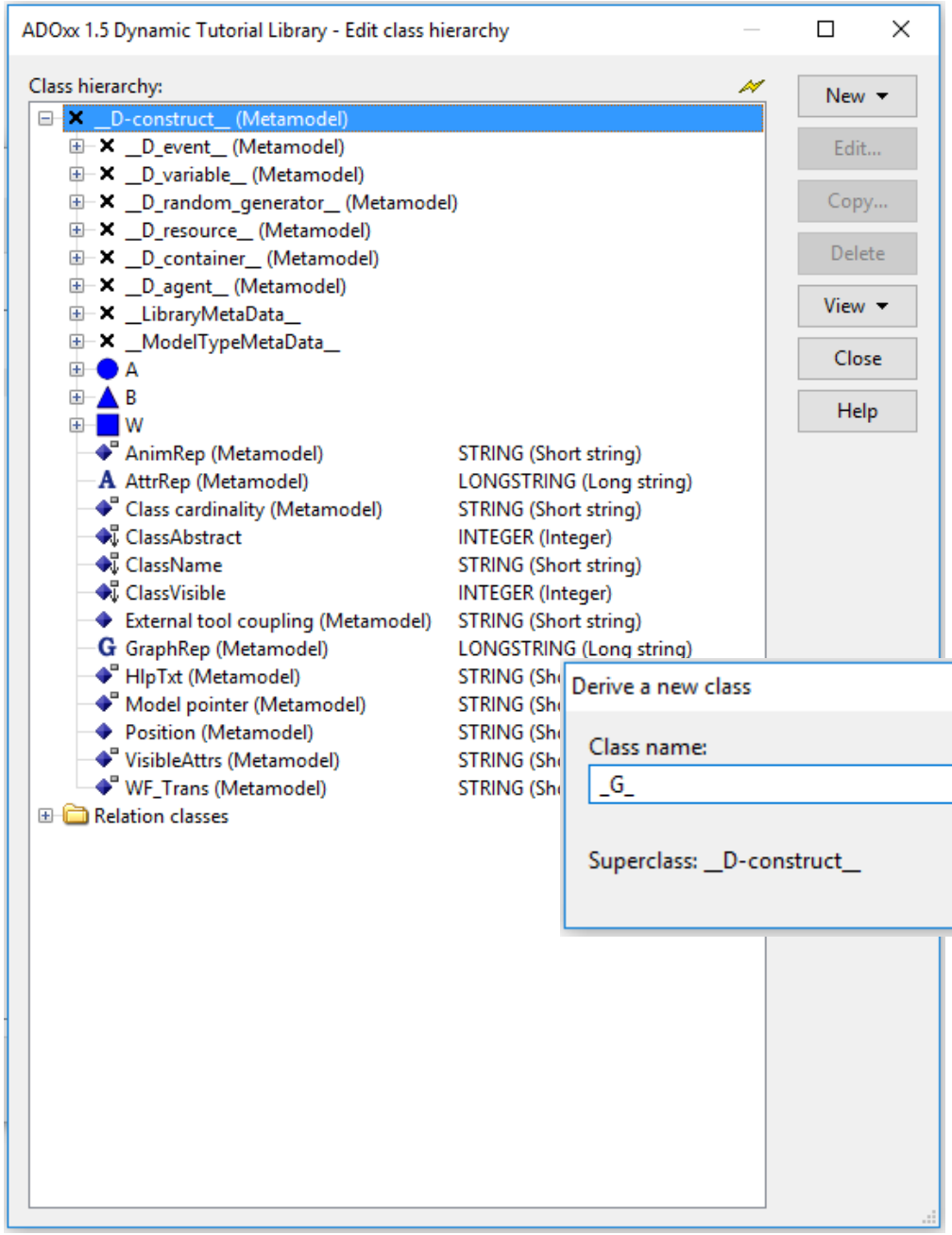

# 1. CLASSES and RELATIONS HANDS-ON

# Modification of class hierarchy of dynamic library



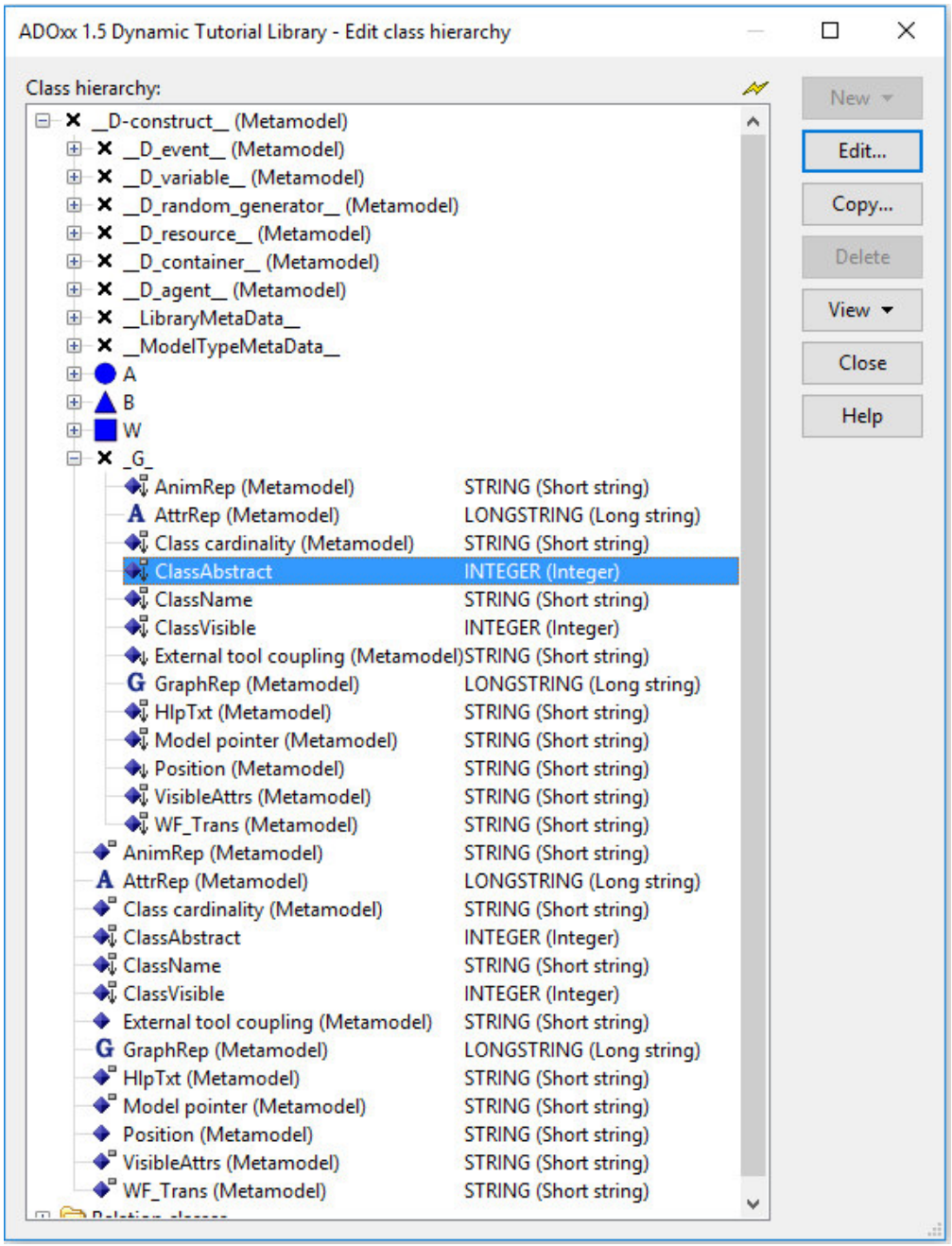






**Add a new abstract class below the root element that is used to define "_G_" related issues**

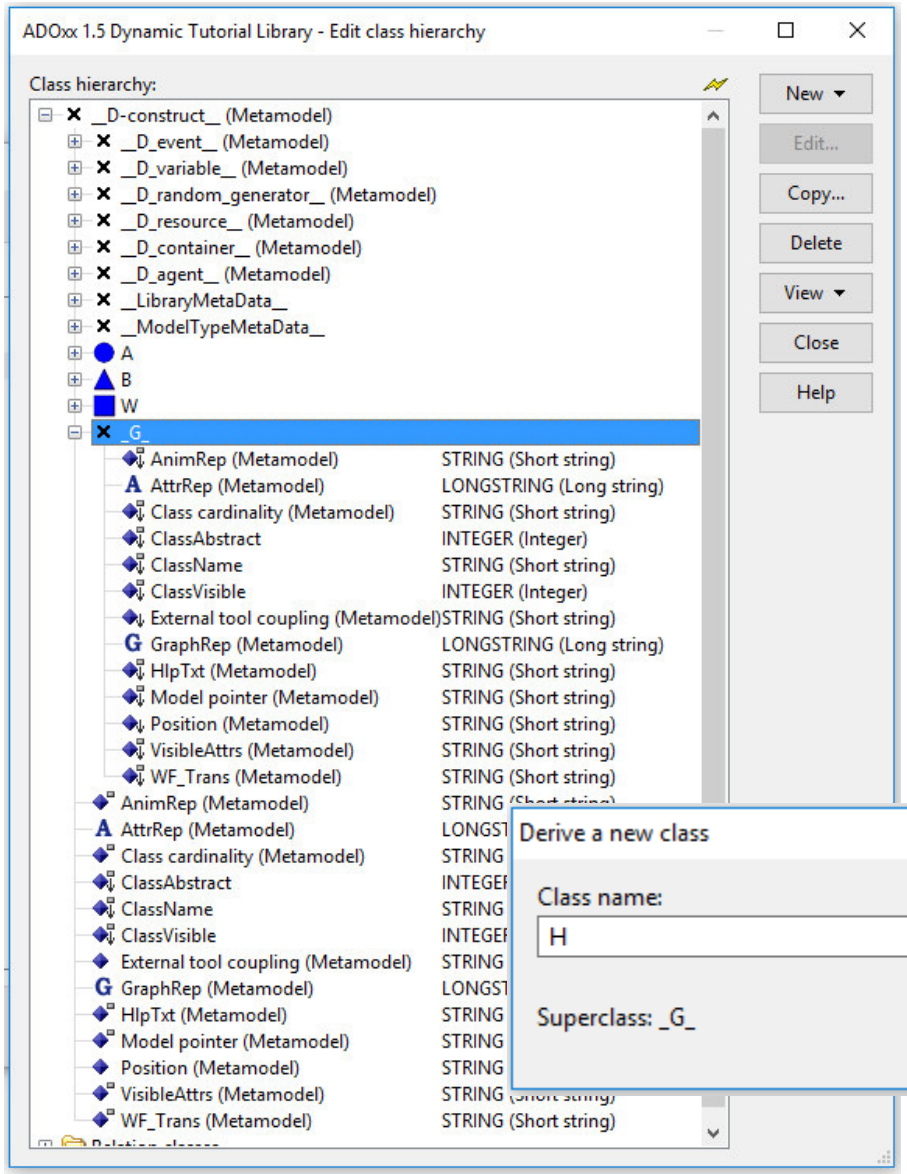
1. **Select root class, click "New" -> "New class"**
2. **Name new class as an abstract class**
   **Naming convention: start and end with "_"**




**Make class abstract using**

**“ClassAbstract“ attribute**

**-> Effect: class can not be instantiated in the modelling tool**

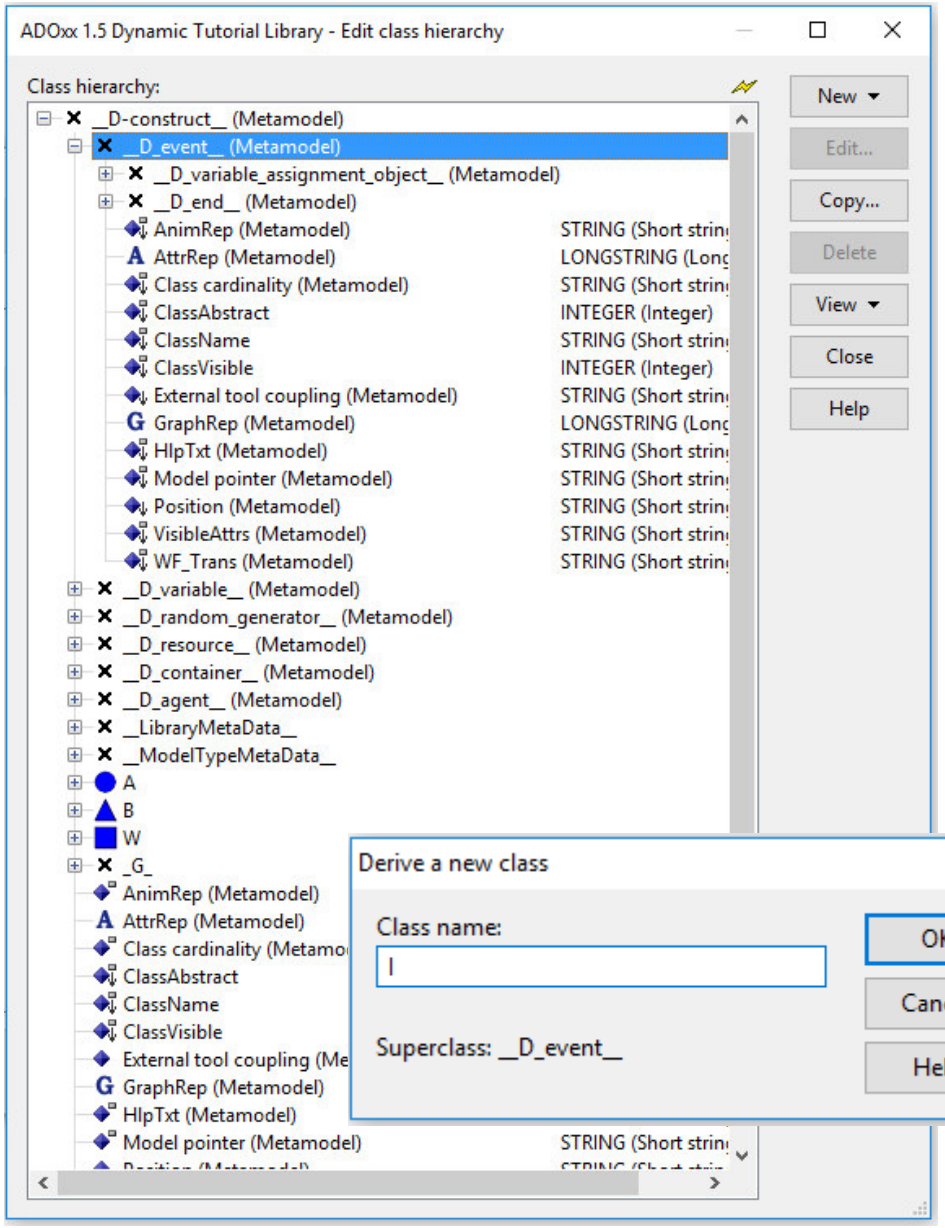



**Add a new concrete class below the abstract element that is used to define a concrete class**

Select the abstract class, click “New” -> “New class”

Name new class

**The new created class can be identified on instance level by the “Name” attribute. This attribute is automatically/implicit available for each class**

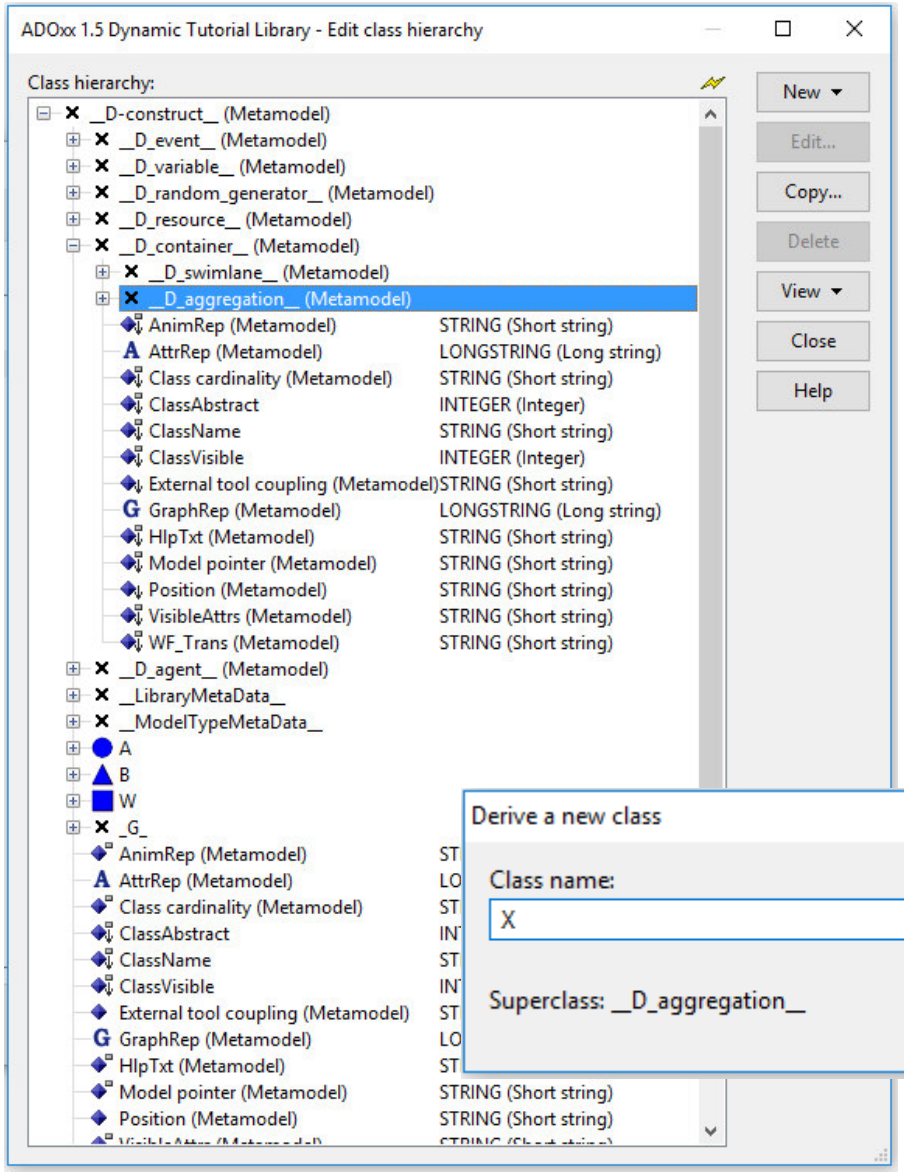



**Add a new concrete class below the __D_event__ element that is used to define a flow class**

Select “__D_event__” class, click “New” -> “New class”

Name new class

**The new created class can be identified on instance level by the “Name” attribute. This attribute is automatically/implicit available for each class**

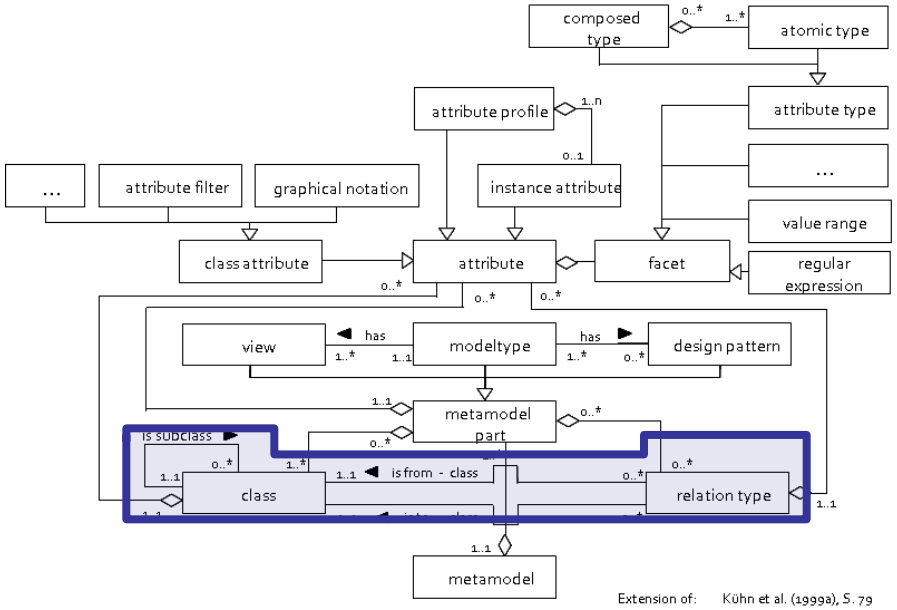



**Add a new concrete class below the __D_aggregation__ element that is used to define Grouping**

Select "__D_aggregation__ "class, click "New" -> "New class"

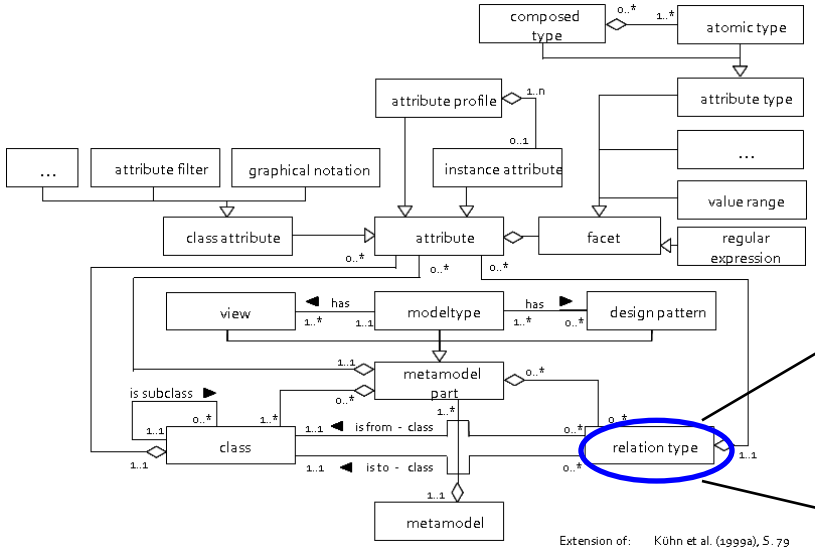
Name new class

**The new created class can be identified on instance level by the "Name" attribute. This attribute is automatically/implicit available for each class**

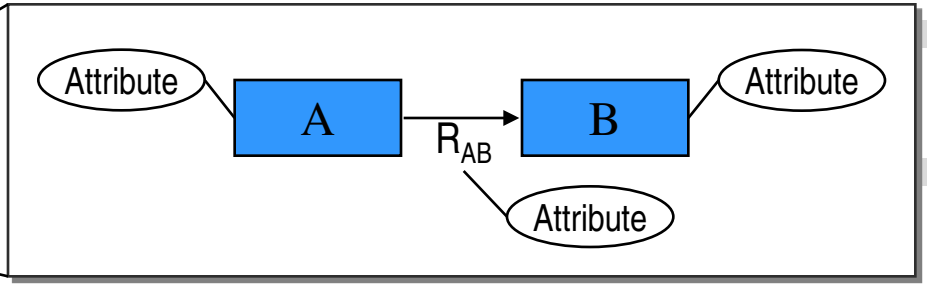



# 1. CLASSES and RELATIONS

# Definition: Relation




Relationship between objects are defined as relation types between classes. Relations are defined by their source and target class, their cardinality, and their attributes.




**Source and Target Class:**

Any class – Pre-defined abstract class, abstract class or class – can act as source class defining where the relation starts from, as well as target class defining where the relations ends.

**Cardinality:**

Cardinality like 1:1, 1:n and n:m relationship is defined in the cardinality of the relation.

**Attributes:**

Attributes are descriptive properties of relations and handled like attribute for classes.

# Relation Types

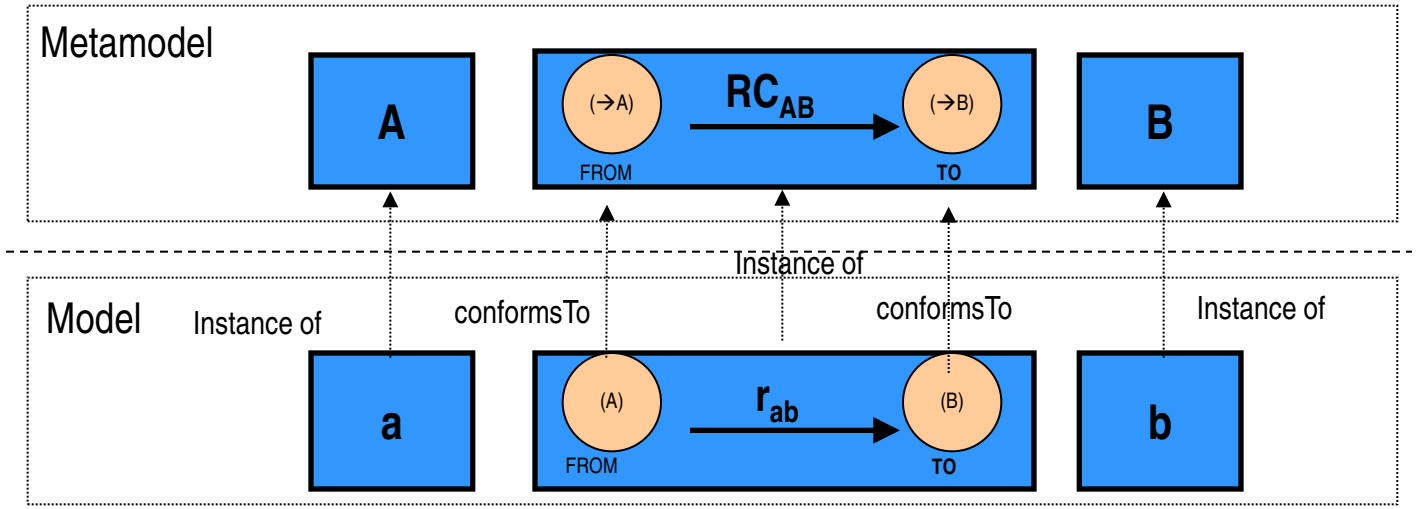
**Relations in ADOxx are expressed either as a class "Relation Class" or as a pointer in form of an attribute called "InterRef".**

**Relation as Class "RC"**

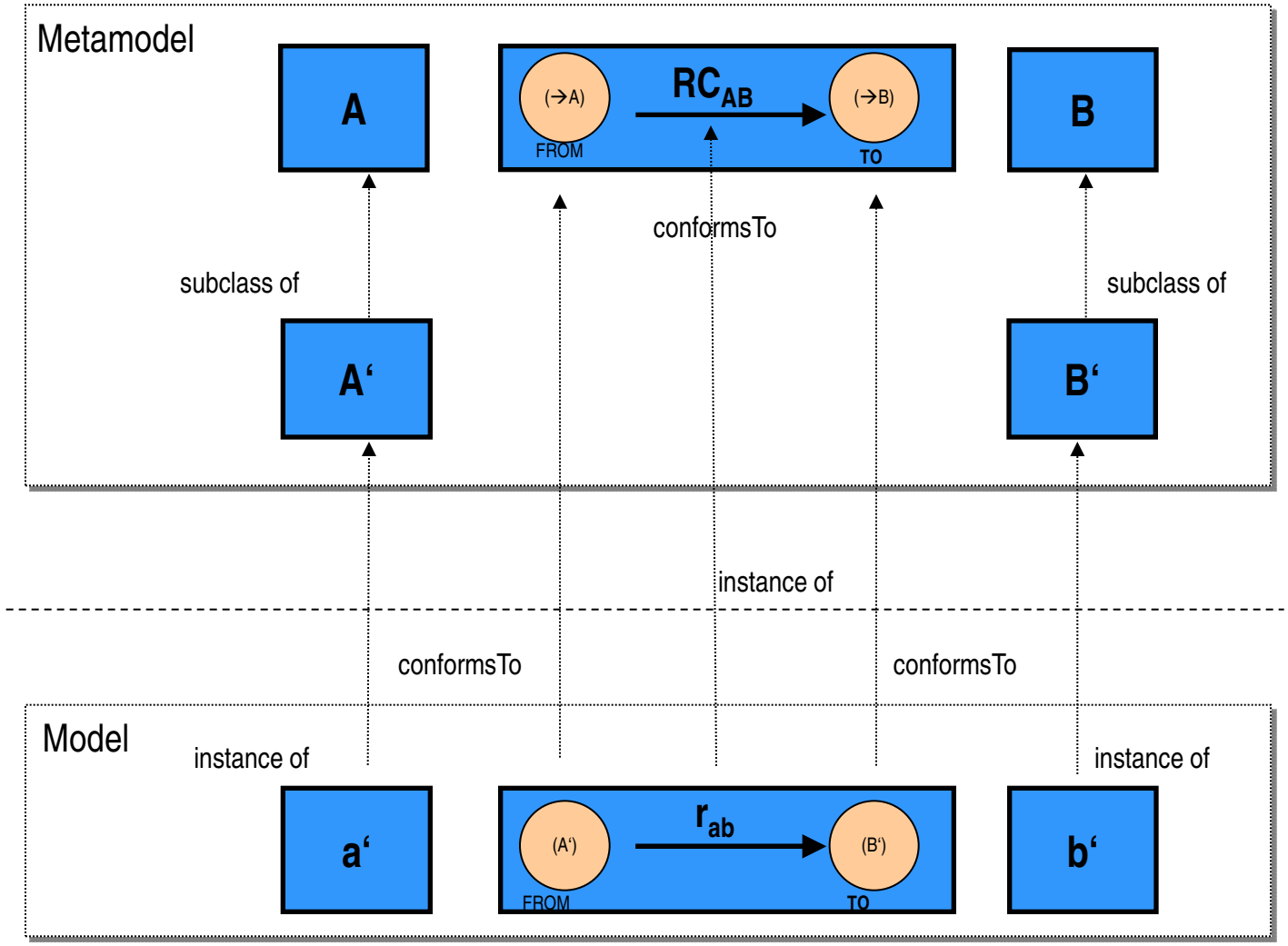
- describes relationship between two objects from two or more classes within one model.
- has start and endpoints define which (abstract) classes a relation can connect
- Cardinality and attribute defined the semantic of the relations class

**Relation as Attribute "InterRef"**

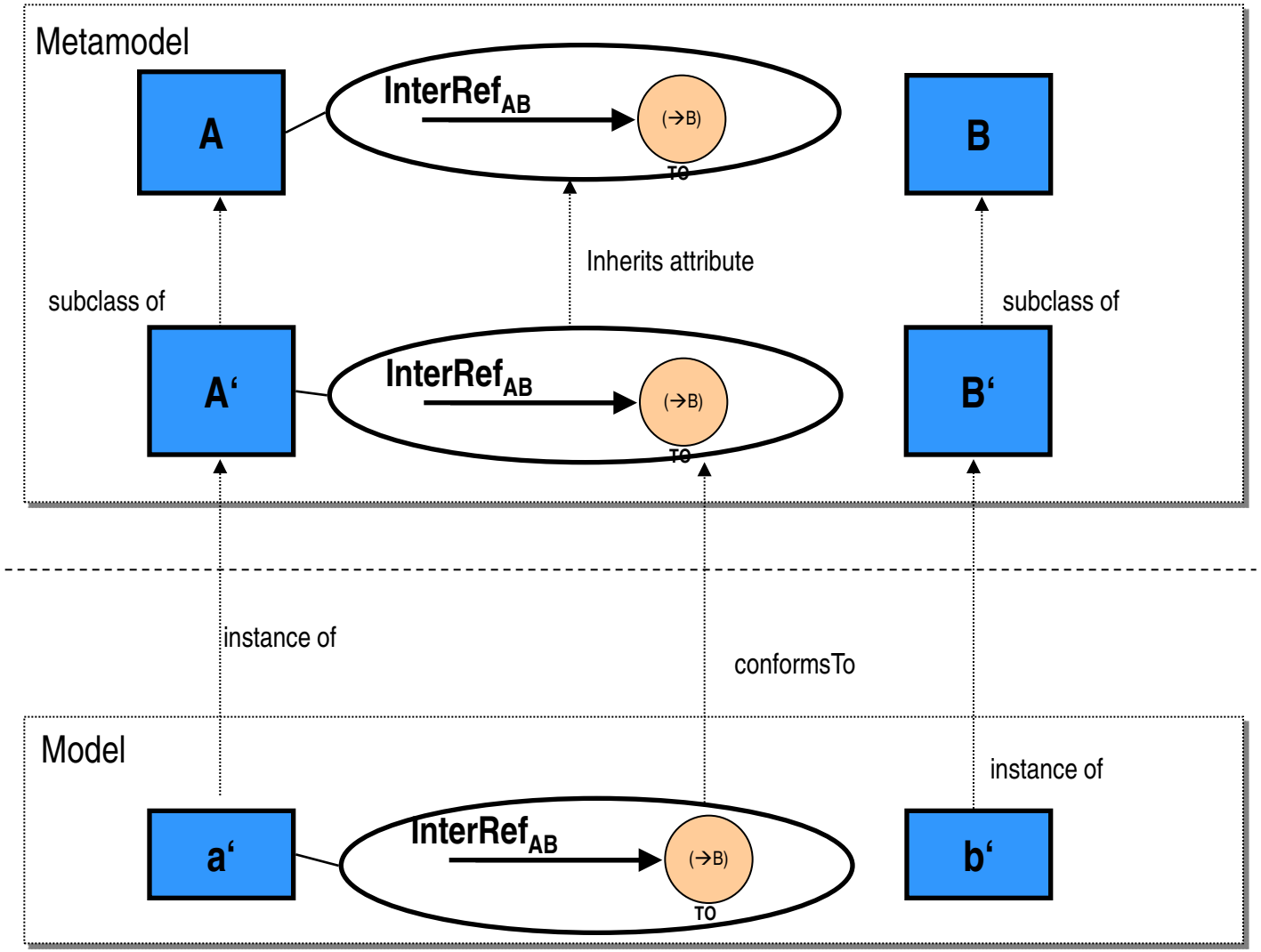
- Is a special configuration of a Relation Class and describes the relationship between two objects from two or more classes within or across models.
- Is a pointer represented as an attributed in the class the relation starts from, with defined classes the relation can point to.
- Cardinality defines the semantic of the InterRef



# Relation Types: Inheritance of Relation Class



# Relation Types: Inheritance of InterRef



# Realisation of Meta Model

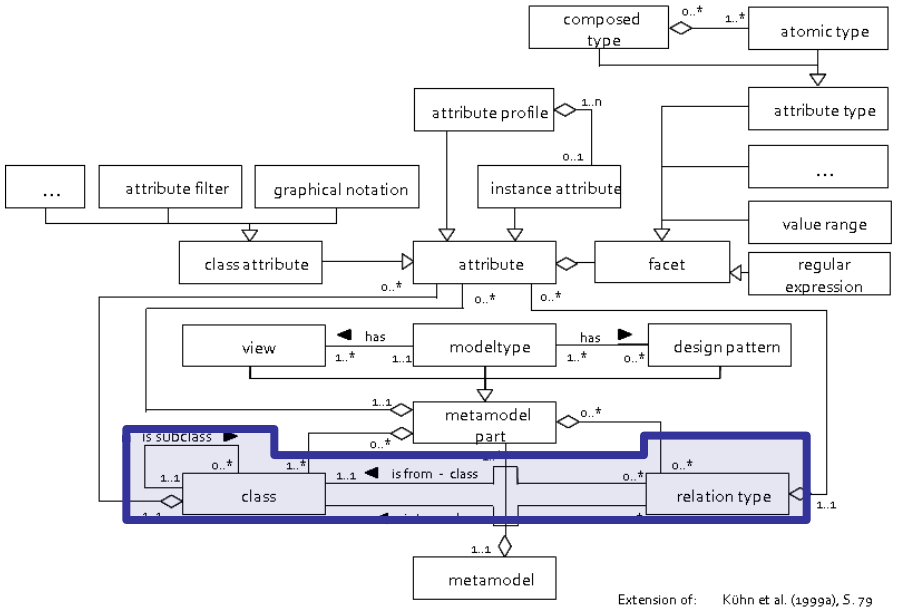
## Specification of a meta model in ALL

1. Specify the meta model starting from the „Empty Meta Model“ and add relation classes and interrefs to classes etc. with ALL using a text editor.
2. Translate ALL into the ADOxx interpretable ABL format and import the meta model into ADOxx.

| | |
|---|---|
| *relationclass* : | *relationclass-definition* { *instanceattribute* } \|<br>*redefrelationclass-definition* { *redefinstanceattribute* } . |
| *relationclass-definition* : | **RELATIONCLASS** *identifier* **FROM** *identifier* **TO** *identifier* . |

| | |
|---|---|
| *instanceattribute-definition* : | **ATTRIBUTE** *identifier* **TYPE** *typeidentifier* \|<br>**ATTRIBUTE** *identifier* **TYPE** *typeidentifier* **VALUE** *val* \|<br>**ATTRIBUTE** *identifier* **VALUE** *val* \|<br>**ATTRIBUTE** *identifier* **TYPE RECORD .** |
| *instanceattribute-setting* : | **ATTRIBUTE** *identifier* **VALUE** *val* . |

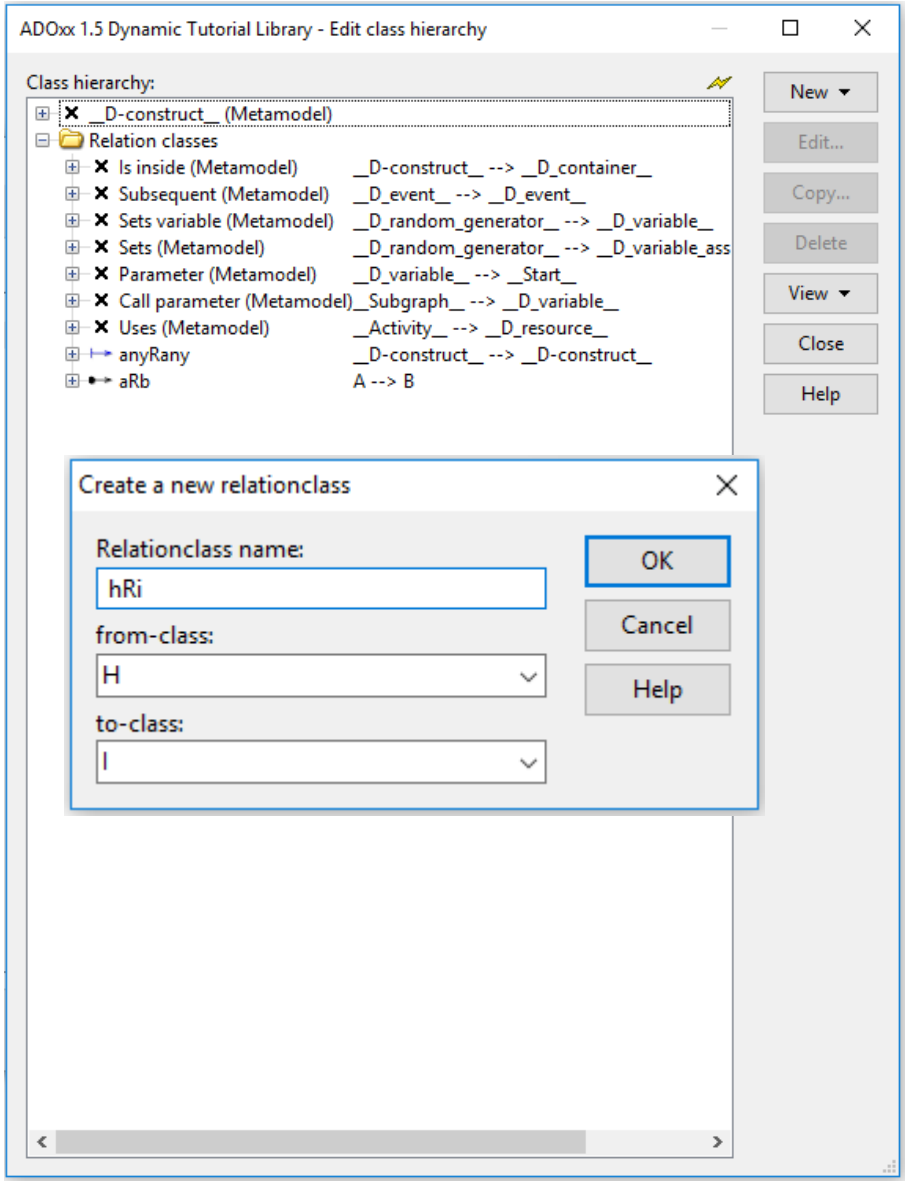
| | |
|---|---|
| *typeidentifier* : | **INTEGER** \| |
| | **. . .** |
| | **INTERREF** \| |




# 1. CLASSES and RELATIONS

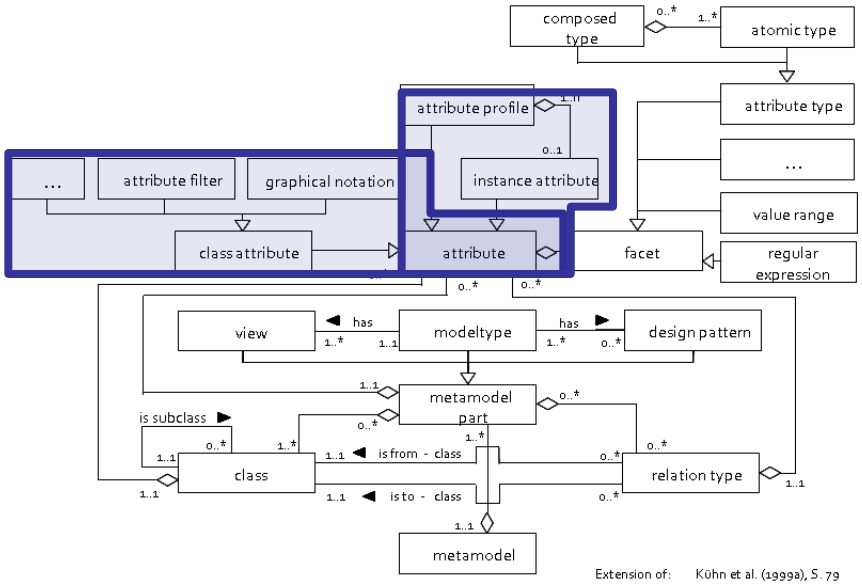
# HANDS-ON

# Definition of Relation Class




**Add a new relation class to connect classes**

- Click “New” -> “New relation class”
- Name new relation class
- Define from-class
- Define to-class




# 2. CLASS ATTRIBUTE & ATTRIBUTE

# Definitions: Data Object Model 1

- A Facet has exactly three properties: a name, a type and a value. Every one of these three properties is saved in one slot. Possible facet types are STRING, INTEGER and DOUBLE.

- Attributes define certain properties of classes or relation classes. Every attribute consists of at least three facets: a name facet (name: "Name", type: STRING, value: "..."), a type facet (name: "Type", type: INTEGER, value: [STRING, INTEGER, DOUBLE, LONGSTRING, DISTRIBUTION, EXPRESSION, TIME, ENUMERATION, ENUMERATIONLIST, PROGRAMCALL, INTERREF, RECORD, PROFILEREFERENCE]) and one value facet (name: "Value", type: [STRING, INTEGER, DOUBLE, RECORD], value: "...").

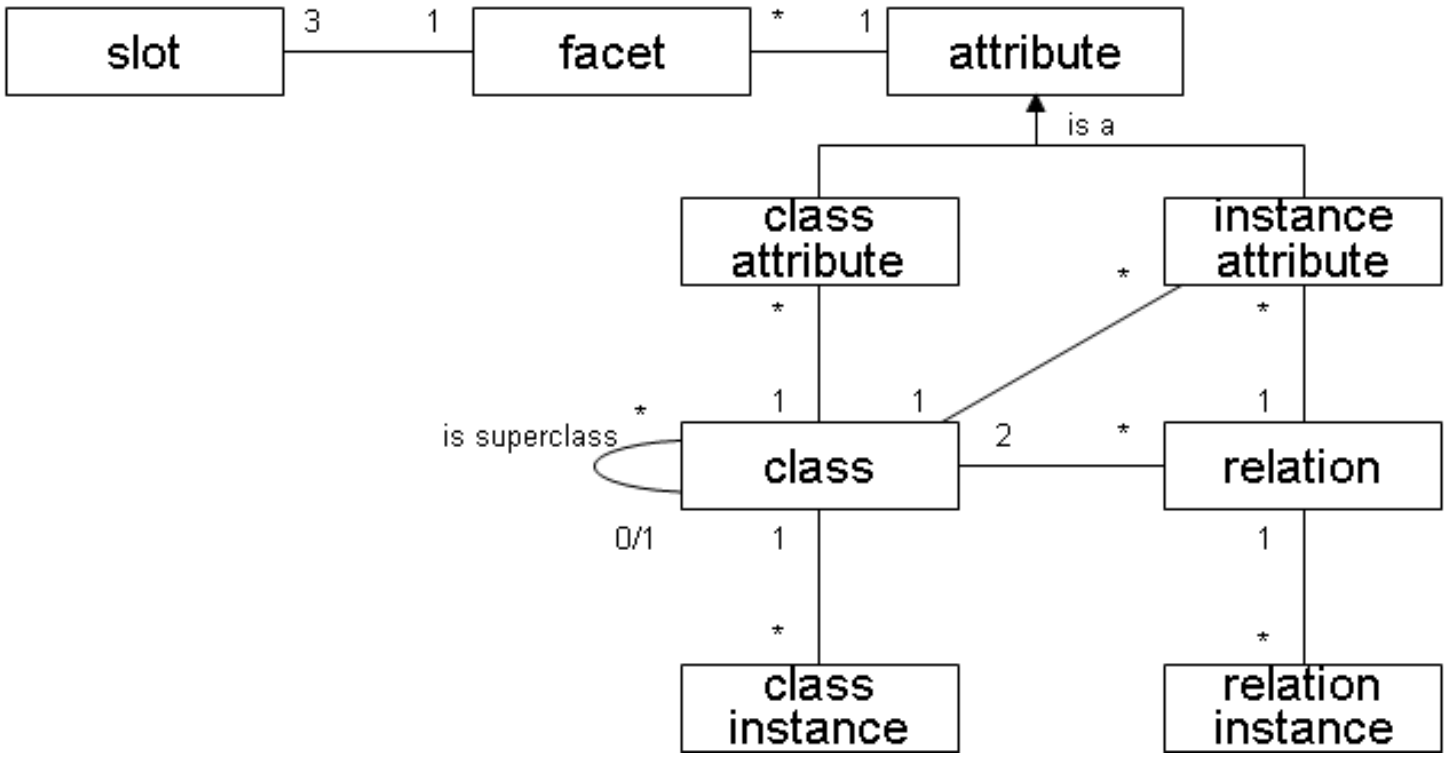
- Every attribute has an additional facet called "AttributeHelpText" which contains user help. Depending on the type of the attribute, additional facets may be defined.

- Attributes can be either class or instance attributes. Class attributes receive one value for every class. Instance attributes receive one value of each instance or relation.

- A Class derived from another class is called subclass and inherits all attributes that are defined in the class from which it is derived. A class from which other classes are derived is called superclass. Relation classes (or just relations) can not be inherited. Relations are always defined between exactly two classes: one source and one target class.

# Definitions: Data Object Model 2

Every object is identified by a unique id. The following chart shows the relations between different objects, used to define concepts like class, relation, instance, attribute ...



# Basic: Definition of Attributes

**Attributes for classes and relation classes have to be defined in the definition section of the class/relation class with 'TYPE'.**

**The following attribute types are possible:**

| | |
|---|---|
| • INTEGER | integer |
| • DOUBLE | floating number |
| • STRING | string – max. 3699 symbols |
| • LONGSTRING | string – max. 32000 symbols |
| • TIME | time |
| • DATE | date |
| • DATETIME | date and time |
| • ENUMERATION | enumeration for selecting a characteristic |
| • ENUMERATIONLIST | enumeration for selecting one or several characteristics |
| • PROGRAMCALL | enumeration for selecting a program |
| • RECORD | a table of attributes |
| • EXPRESSION | a formula |
| • INTERREF | reference on a model or an instance |
| • ATTRPROFREF | a preset set of attribute values |

# Attribute Types and their Appearance

## Numerical Attributes: Integer (INTEGER)


- An attribute of the type "Integer" is defined as an integer from -1,999,999,999 to 1,999,999,999.
- An ADOxx integer is limited to 10 digits plus an optional sign ('+' or '-')
- The standard value of attributes of this type is "0" or a value defined

# Attribute Types and their appearance

## Numerical Attributes: Floating number (DOUBLE)


- The amount of decimal places is defined by the attribute definition
- An attribute of the type "Double" is defined for a float within +/-999,999,999,999,999 for an integer (without decimal places) or +/-999,999,999.999999 for figures with 6 decimals.
- The corresponding attribute value is displayed to 6 decimal places. That means that a double value should not exceed a total of 15 significant digits with at last 6 decimal digits!
- The standard value of attributes of this type is "0.000000" or a value defined in the application library.

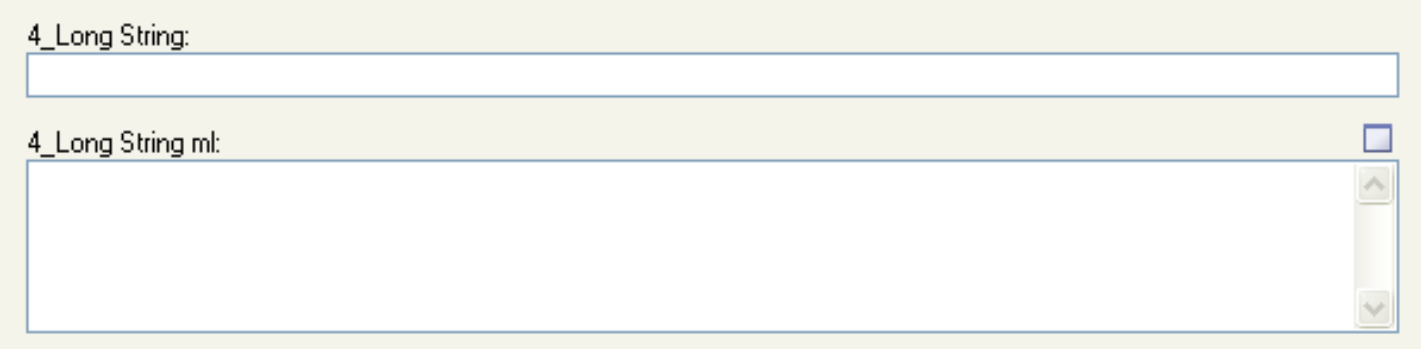
# Attribute Types and their Appearance

## String attributes: String (STRING)


- An attribute of the type "String" is defined for texts up to 3700 characters of any type.
  - Hint: The maximum number of characters is 250 for name. That concerns classes, relation, instances, attributes, application models, libraries and application libraries.
  - Model names have a special rule!
- The standard value of attributes of this type is "" (no entry) or a value defined in the application library.

# Attribute Types and their Appearance

## String attributes: Longstring (LONGSTRING):




- Some text attributes are already defined as „multi-line“. The parameter lines can be used to specify how many lines should be shown in the text field of the Notebook.
- The parameter dialog can be used to specify special input supports in place of the standard one.
- An attribute of type "Longstring" is defined for texts up to 32000 characters of any type.
- The standard value of attributes of this type is "" (no entry) or a value defined in the application library.

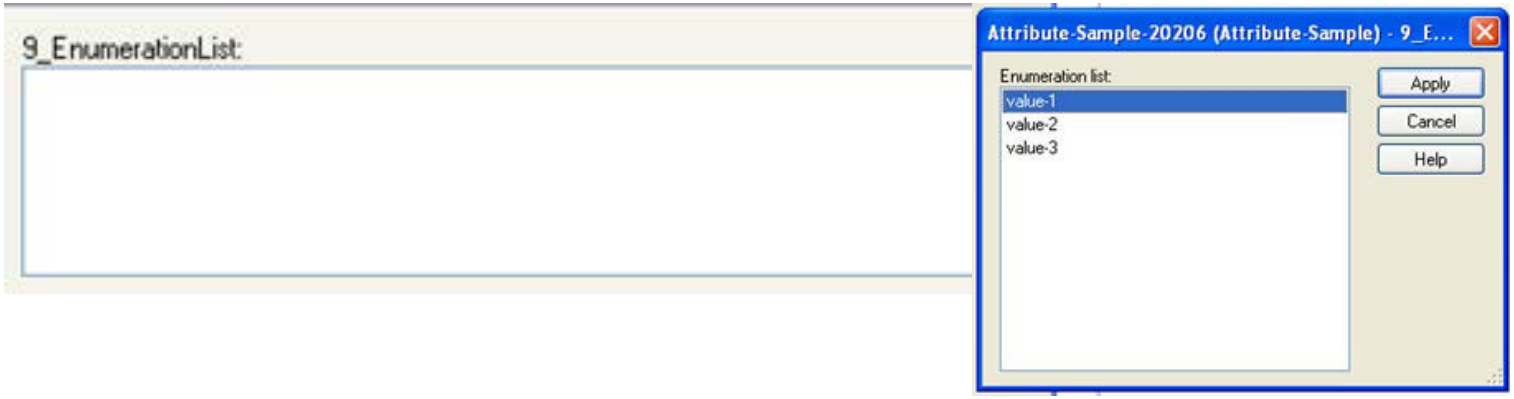
# Attribute Types and their Appearance

## Enumerations / Enumeration lists: Enumeration (ENUMERATION)


- The parameter `ctrltype` sets how the enumeration should appear, as a drop down list, as radio buttons or as checkboxes (only if two possible values).
- An attribute of the type "Enumeration" is characterised by a defined set of values. An "Enumeration" attribute has exactly one value of this set.
- The standard value of this type is specified in the library definition.

# Attribute Types and their Appearance

## Enumerations / Enumeration lists: Enumeration list(ENUMERATIONLIST):




- An attribute of the type "Enumeration list" is characterised by a defined set of values. An "Enumeration list" attribute has either none, one or several values of this set. The difference to an "Enumeration" attribute is, that an "Enumeration list" attribute can have more than one entry selected!
- The standard value of this type must specified in the library definition.

# Attribute Types and their Appearance

**Date / Time: Date (DATE)**


The ADOxx format for date is YYYY:MM:DD

**Date / Time: Time (TIME)**


The ADOxx format time is YY:MM:DDD:HH:MM:SS

**Date / Time: Date and Time (DATETIME)**


The ADOxx format time is YYYY:MM:DD HH:MM:SS

- Time format YY:DDD:HH:MM:SS (years:days:hours:minutes:seconds). Valid day ranges are from 0 to 365, valid hours are between 0 and 23, valid minutes and valid seconds are between 0 and 59.
- The standard value of attributes of this type is "00:000:00:00:00" or a value defined in the application library.

# Attribute Types and their Appearance

## References / Program calls: Intermodel reference (INTERREF)


Syntax of the InterRef domain definition facet:

| | |
|---|---|
| *InterRefDomain* : | [ **REFDOMAIN** [ **max:***intValue* ] ]<br>*Refs* . |
| *Refs* : | *ModRefs* \| *InstRefs* . |
| *ModRefs* : | { **MODREF mt:***modelTypeName* } . |
| *ObjRefs* : | { **OBJREF mt:***modelTypeName* **c:***className* **max:***intValue* } . |

Syntax of InterRef attribute values:

| | |
|---|---|
| *InterRefValue* : | *ModRefs* \| *ObjRefs* . |
| *ModRef* : | { **REF m:***modelName* **mt:***modelTypeName* } . |
| *ObjRef* : | { **REF m:***modelName* **mt:***modelTypeName*<br>**c:***className* **i:***instanceName* } . |

# Attribute Types and their Appearance

## References / Program calls: Programcall (PROGRAMCALL)

- A PROGRAMCALL attribute is characterized by a fixed set of items. These items are related to AdoScripts which can be called via the user interface. A PROGRAMCALL attribute value consists of at most one of the defined items and an optional parameter.


| | |
|---|---|
| *ProgramCallDomain* : | { *ItemDefinition* } . |
| *ItemDefinition* : | **ITEM** *itemText* [ *ParameterDef* inition ] { *FDlgFilter* } *AdoScript* . |
| *ParameterDefinition* : | **param** : *paramText* [ : *defaultTextValue* ] . |
| *FDlgFilter* : | **fdlg-filter**<*i*> : *filterText*<br>**fdlg-type**<*i*> : *filterDescriptionText* . |

*itemText*, *paramText*, *defaultTextValue*, *filterText* and *filterDescriptionText* are string values.

# Attribute Types and their Appearance

## Table: Table (TABLE)

Tables will appear in Notebooks according to the definition of the table class.

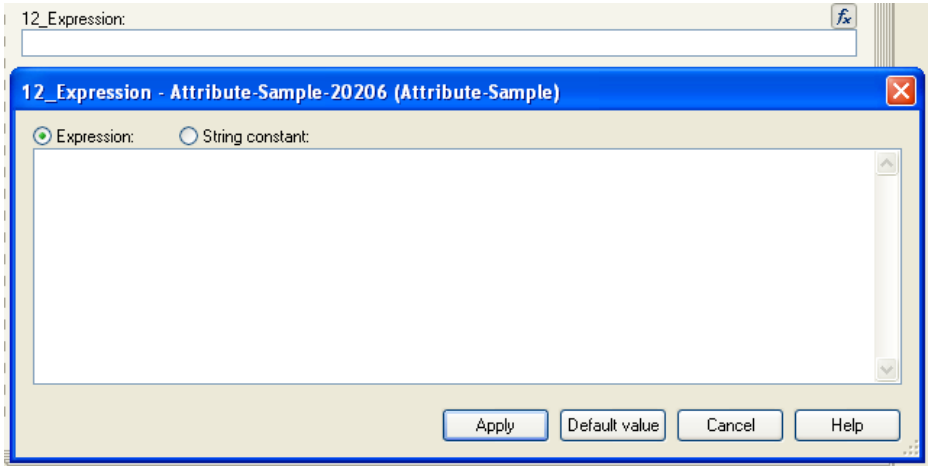
Following adjustments can be done in AttrRep of the table class:

- which columns should be shown
- in what sequence
- Relative width - Parameter `width`


An Attribute of Type "Table" (RECORD) is defined by a flexible List-/Table-Administration of Attribute Types that are put together.
The standard Value for Attributes of this Type depends on the Attribute Types defined in the Table Class.

# Attribute Types and their Appearance

## Expressions / Attribute profile references: Expression(EXPRESSION)




- Every definition of expression attributes is started with the keyword EXPR. The result type is is defined with the attribute type: and the default formula is defined with the attribute expr:. Every time you create an instance (a model, object, or connector), this formula will be used to compute the result value of the expression.
- By setting the modifier fixed:, you make the expression attribute a fixed expression. The user will the not be able to change the formula in the Modelling Toolkit.
- The formula itself (defined in the attribute expr:) must never be longer than 3600 characters.
- For expressions with result type double, the attribute format can be used to specify the number of digits that should be displayed on the user interface. Note: the number of digits displayed on the user interface do not affect the internal precision of the expression result value.

# Attribute Definition

| | |
|---|---|
| *attribute-definition* : | *instanceattribute-definition* \|<br>*classattribute-definition* . |
| *classattribute-definition* : | **CLASSATTRIBUTE** *identifier* **TYPE** *typeidentifier* \|<br>**CLASSATTRIBUTE** *identifier* **TYPE** *typeidentifier* **VALUE** *val* \|<br>**CLASSATTRIBUTE** *identifier* **VALUE** *val* \|<br>**CLASSATTRIBUTE** *identifier* **TYPE RECORD .** |
| *instanceattribute-definition* : | **ATTRIBUTE** *identifier* **TYPE** *typeidentifier* \|<br>**ATTRIBUTE** *identifier* **TYPE** *typeidentifier* **VALUE** *val* \|<br>**ATTRIBUTE** *identifier* **VALUE** *val* \|<br>**ATTRIBUTE** *identifier* **TYPE RECORD .** |

| | |
|---|---|
| *typeidentifier* : | **INTEGER** \|<br>**DOUBLE** \|<br>**STRING** \|<br>**DISTRIBUTION** \|<br>**TIME** \|<br>**ENUMERATION** \|<br>**ENUMERATIONLIST** \|<br>**PROGRAMCALL** \|<br>**INTERREF** \|<br>**EXPRESSION** \|<br>**ATTRPROFREF .** |

# 2. CLASS ATTRIBUTE & ATTRIBUTE

# HANDS-ON

# Example for an instance attribute definition

```
ATTRIBUTE <Description>
TYPE STRING
VALUE ""

        FACET <MultiLineString>
        VALUE 1

        FACET <AttributeHelpText>
        VALUE ""
```

Keyword

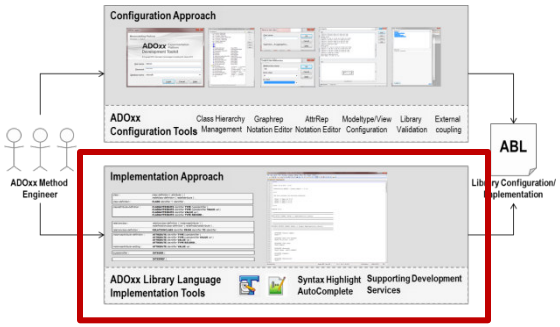
Attribut name:
The name can use alphanumeric characters

Type definition

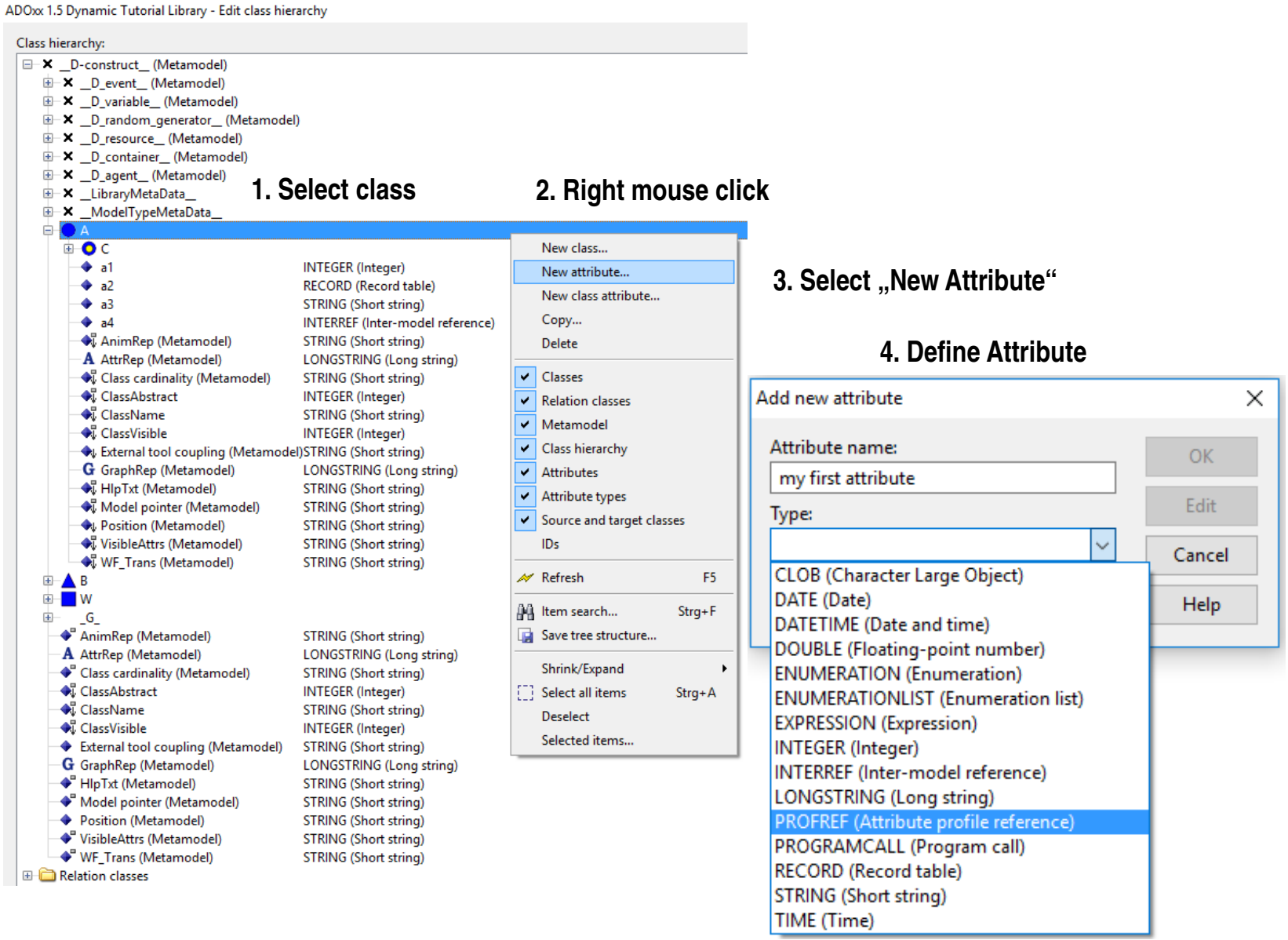
Value:
The concrete value will be determined by the model.

This FACET defines if a text-box can be used.

A help text can be provided for the attribute.



# Example of New Attribute in ADOxx



# 2. SPECIAL CLASS ATTRIBUTE & ATTRIBUTE

# Basics: Selected Special Attributes

**The following class attributes can be customized:**

**AttrRep*:** Notebook-Definition (all classes)

**GraphRep*:** Graphical representation (object- and relation classes)

**Model pointer*:** Relations to other models (object classes)

**Class cardinality*:** Relation constraints (object classes)

**__Conversion__**[X]**:** Conversion from one object to another

*are class attributes from Root Class (D|S_Construct) hence inherited by each class
[X] any class can define this class attribute

# 2. SPECIAL CLASS ATTRIBUTE & ATTRIBUTE GRAPHREP

# Basics: Graphical Notation of Classes

## Static Notation:

- Semiotic Clarity
- Perceptual Discriminability
- Semantic Transparency
- Complexity Management
- Cognitive Integration
- Visual Expressiveness
- Dual Coding
- Graphic Economy
- Cognitive Fitness

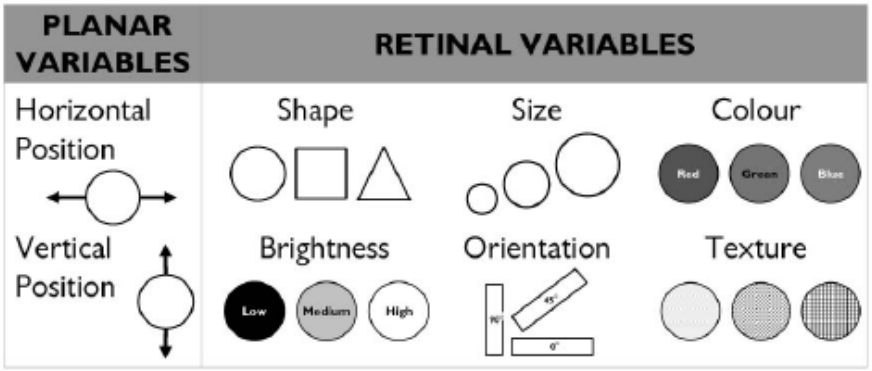



Fig. 7. Visual variables [8]: These define a set of elementary graphical techniques for constructing visual notations. A color version of this figure may be viewed at http://doi.ieeecomputersociety.org/10.1109/TSE.2009.67.

## Dynamic Notation:

- Event based changes of notations (e.g. attribute change)

Reference MOODY: THE "PHYSICS" OF NOTATIONS: TOWARD A SCIENTIFIC BASIS FOR CONSTRUCTING VISUAL NOTATIONS IN SOFTWARE...

# GRAPHREP I

- Class attribute GRAPHREP is of type long string, hence the attribute value is a text that is interpreted as a script by the GRAPHREP interpreter.
- The following types of elements are distinguished:
  - Style elements
  - Shape elements
  - Variable assigning elements
  - Context elements
  - Control elements
- The representation characteristic for following shape elements is modified by style elements:
  - **PEN** sets the characteristics of the outline pen for shape elements.
  - **FILL** sets the characteristics of the fill-in brush for shape elements.
  - **SHADOW** switches the additional shadow of shape elements on or off
  - **STRETCH** switches geometric stretching on or off
  - **FONT** sets the font for displayed texts and attribute values.
- **PEN** determines in which manner the lines and curves are drawn, i.e. how strong, in which color and in which style (e.g. dashed line). For shape elements which can be filled, only the outline of the shape is influenced by the current pen. The filling of shapes is controlled by the current fill-in brush, which can be set with BRUSH.
- **Shape elements** which can not be filled are **POINT, LINE, POLYLINE, ARC** and **CURVE**. Fillable elements are **RECTANGLE, POLYGON, ELLIPSE, PIE and COMPOUND**.
- For shape elements coordinates (positions) have to be specified. Coordinates here are relative to the position of the particular object, i.e. they are added to the object's position.

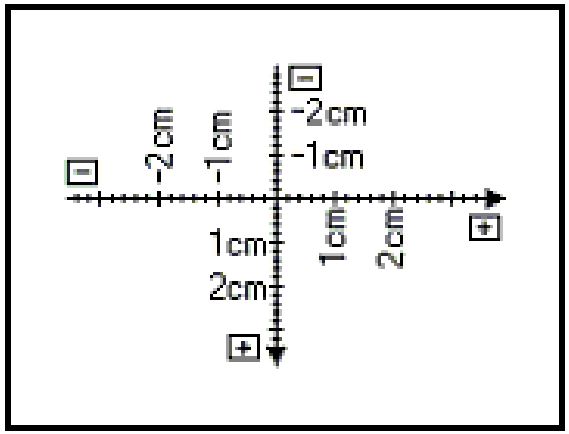
# GRAPHREP II

- Context elements just exist for relations. They specify whether the starting, the middle or the endpoint of the relation is being defined. Keyword "START" defines that the following description refers to the start point of the relation until the next context element START/MIDDLE/END is specified. A fourth context element (EDGE) triggers the drawing of a relation's edge. This is the line from the starting point via possible bend points to the end point of a relation.

- For relations the starting, the middle and the end (point) can be defined. Positions then refer to one of these three points. However, the coordinate system is rotated depending on the direction of the relation instance. On defining a relation's GraphRep, you have to regard the relation as going horizontally from the left to the right. The coordinate system's origin then is the point of the relation for which the graphical representation currently is being defined, i.e. start, middle or end point.

# GRAPHREP III

- On the x-axis the coordinate values increase from the left to the right, on the y-axis they increase from top to bottom (differing from mathematics). Arcs and pies are rotated counter-clockwise.
- ATTENTION: The unit of measure for positions and proportions (cm or pt) has to be specified in every case. Pixel values cannot be used.
- On the drawing of an object, the elements are processed sequentially. However, the control elements make it possible to skip sections during the element processing depending on variables. For example, attribute values of the object to be represented may be assigned to such variables. A graphical representation depending on object attributes can thus be obtained using variable assignment elements combined with control elements. Additional possibilities are given from using variables in graphical elements.



# GRAPHREP IV

## Graph Elements

```
Edge | Start | Middle | End |
Pen | Fill | Shadow | Stretch | Map | Font |
ClipRect | ClipRoundRect | ClipPoly | ClipEllipse | ClipOff |
Point | Line | PolyLine | Arc | Bezier | Curve |
Rectangle | RoundRect | Polygon | Ellipse | Pie |
BeginPath | MoveTo | LineTo | BezierTo |
EndPath | DrawPath |
Compound | Bitmap | GradientRect | GradientTri |
Text | Attr | Hotspot |
Set | Aval | Table | TextBox | AttrBox | BitmapInfo |
IfStatement | WhileStatement |
ForNumStatement | ForTokenStatement | Execute.
```

For detailed explanation see online support for each of the elements

# Some GraphRep-Commands (1)

**GRAPHREP**

- The GraphRep definition must start with this command to be valid. The parameter **layer** defines whether an object will be displayed above or below other objects. The parameter **sizing** specifies if the size can be changed.

**SHADOW**

- Specifies if the class will have a shadow or if it should be drawn “flat”.

**PEN**

- Defines the pens width/color/style.

**FILL**

- Defines the fill color/style and transparency.

**ATTR**

- Shows an attribute value on the drawing area (e.g. object name).

# Some GraphRep-Commands (2)

**POINT**

- Draws a point.

**LINE / POLYLINE**

- Draws a single line (**LINE**) or several lines (**POLYLINE**).

**CURVE / ARC**

- Draws a curve according to a mathematical function or an arc.

**POLYGON**

- Draws a polygon consisting of several straight lines where each corner is defined as a single point.

**RECTANGLE / ROUNDRECT / ELLIPSE / PIE**

- A rectangle, rectangle with rounded edges, an ellipse or a segment of an ellipse.

**COMPOUND**

- A composite filled Form (from **LINE**, **POLYLINE** und **CURVE**-Elements).

# Some GraphRep-Commands (3)

**TEXT**

- Allows to show a specific text on the drawing area (Letters, Symbols …).

**FONT**

- Defines the font style/color for drawn text.

**BITMAP**

- Allows to embed a picture (*.BMP-Format).

**TABLE**

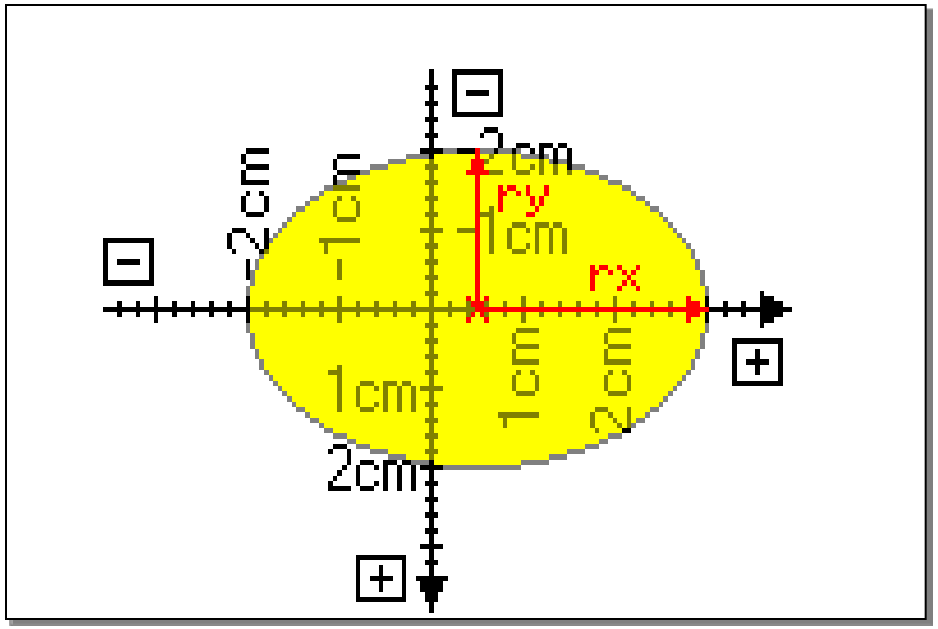
- Creates a table for structuring the attribute representation of an object.

**Hint:**
Graphical elements can be combined for more complex drawing!

# The GraphRep Coordinate Plane

A coordinate plane is used to provide an exact positioning of the GraphRep elements. It is composed of:

- The null coordinate is in the middle
- Positive values go to the right and down
- Negative values go to the left and up




**Hint:**

- It is required to specify the **Unit** (cm or pt). Units in pixels are not possible.
- The direction of rotation progresses counter-clockwise!

# GraphRep Structural Commands

**SET**

Sets a variable with a constant or the result of an expression, which in turn can contain variables.

**AVAL**

Sets variables with the values from an attribute of the instantiated object.

**IF / ELSIF / ELSE / ENDIF**

Allows to change the representation based on values.

**BITMAPINFO**

Reads the height and width of a bitmap file, allowing to properly represent it.

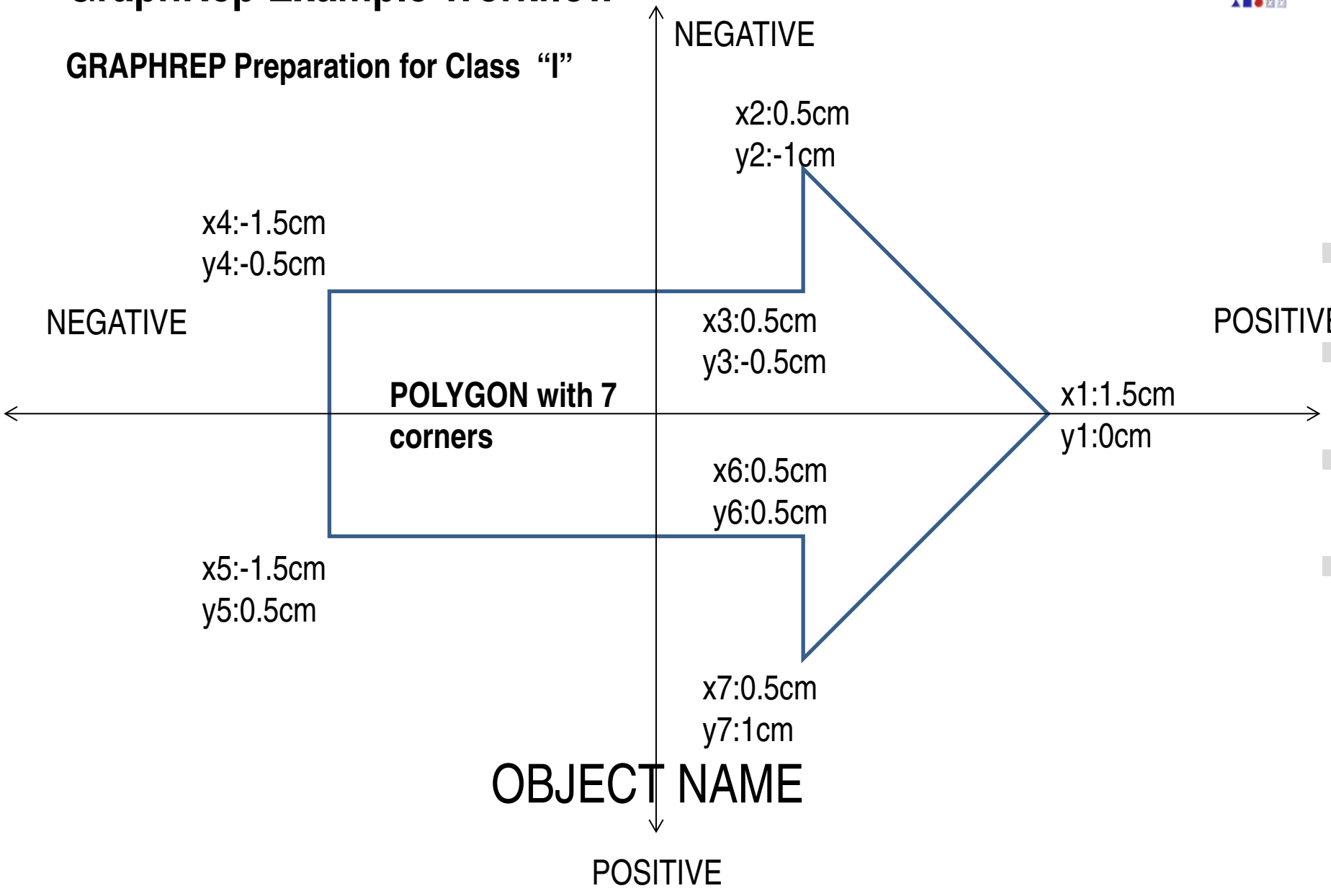
**TEXTBOX / ATTRBOX**

Similar to TEXT and ATTR. However instead of drawing the values it sets specific variables with the rectangle area they would need.

# 2. SPECIAL CLASS ATTRIBUTE & ATTRIBUTE GRAPHREP HANDS-ON

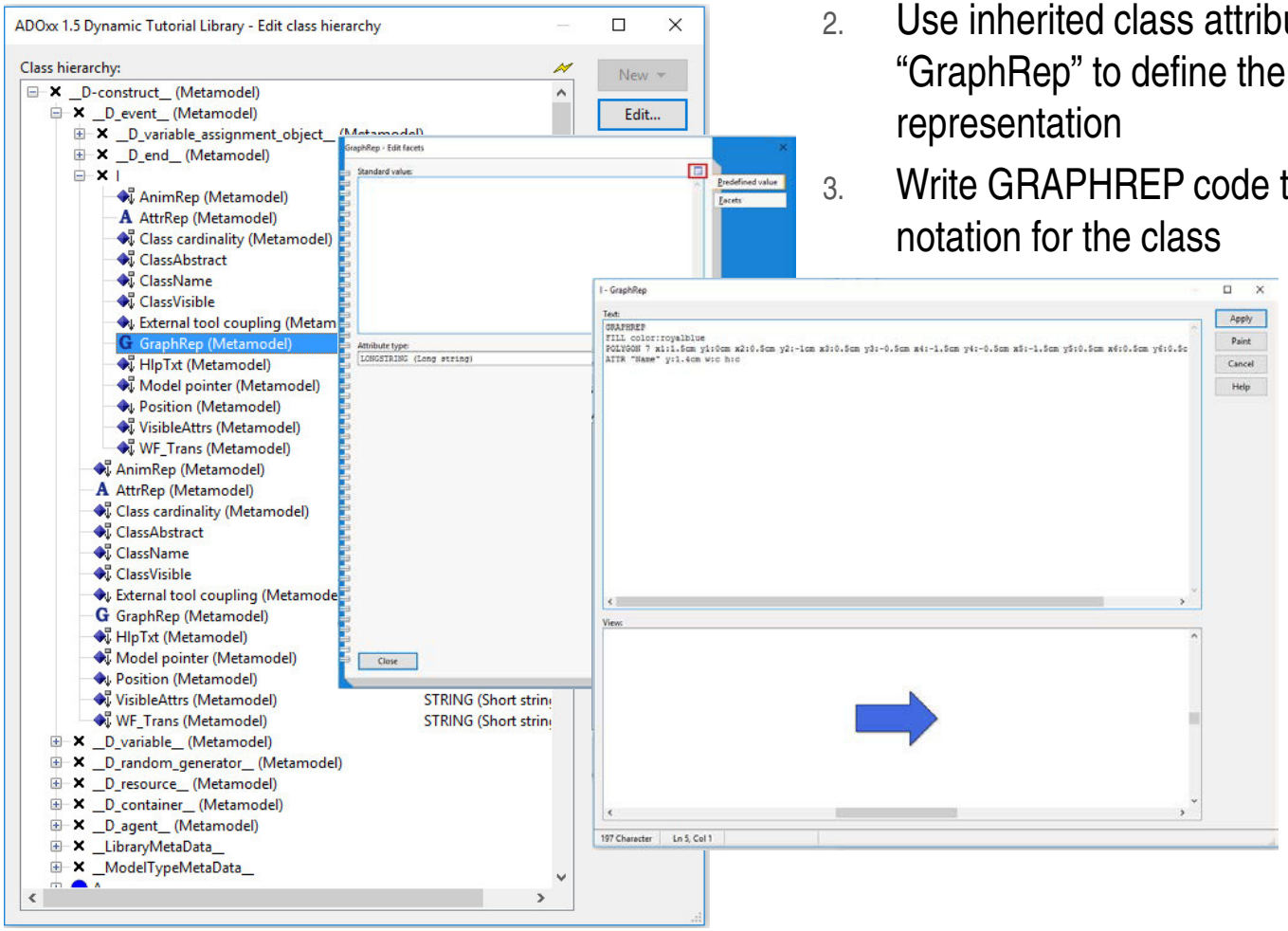
# GraphRep Example Workflow

## GRAPHREP Preparation for Class "I"



# GraphRep Example Workflow

## GRAPHREP Implementation for Class "I"



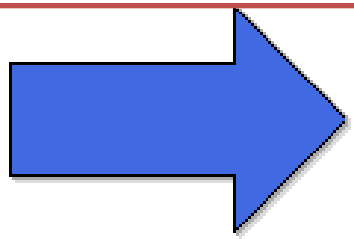

1. Since this class is concrete, a graphical representation needs to be defined.
2. Use inherited class attribute "GraphRep" to define the graphical representation
3. Write GRAPHREP code to provide a notation for the class

# Commented GraphRep Code

## Class: I

```
GRAPHREP
FILL color:royalblue
POLYGON 7 x1:1.5cm y1:0cm x2:0.5cm
y2:-1cm x3:0.5cm y3:-0.5cm x4:-1.5cm
y4:-0.5cm x5:-1.5cm y5:0.5cm
x6:0.5cm y6:0.5cm x7:0.5cm y7:1cm
```

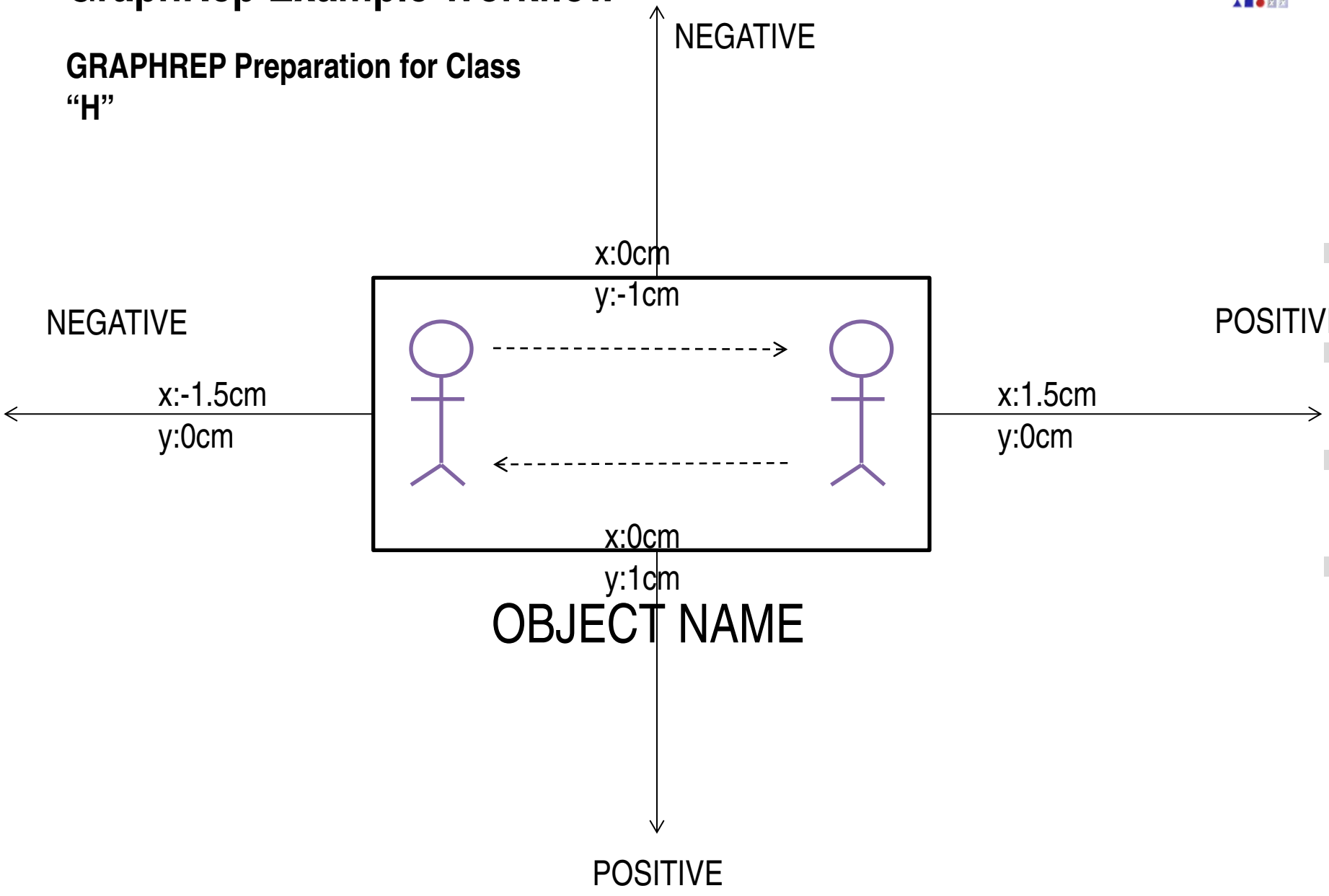
```
ATTR "Name" y:1.4cm w:c h:c
```



In case attribute name is available, it is shown here

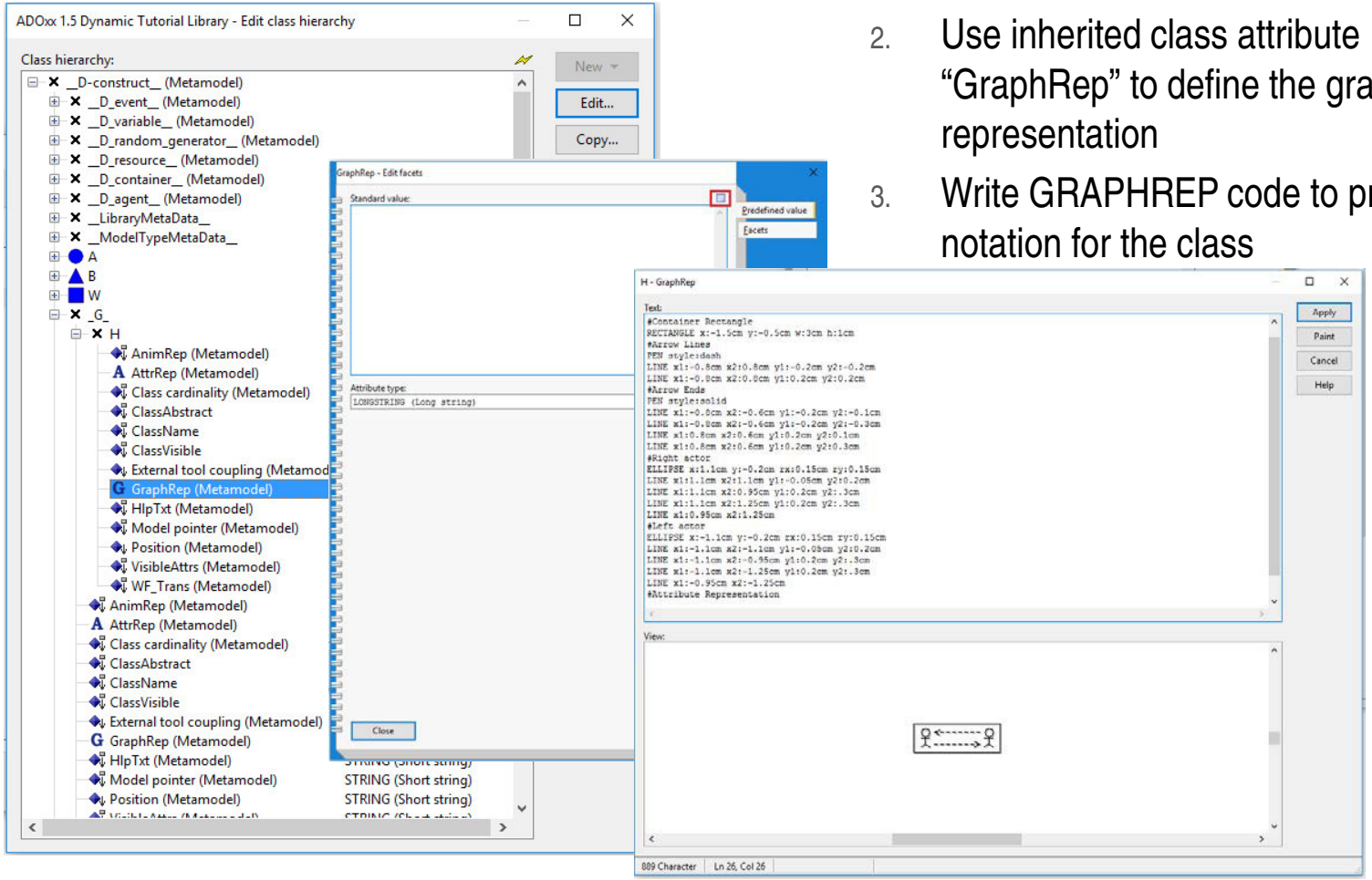
# GraphRep Example Workflow

## GRAPHREP Preparation for Class "H"



# GraphRep Example Workflow

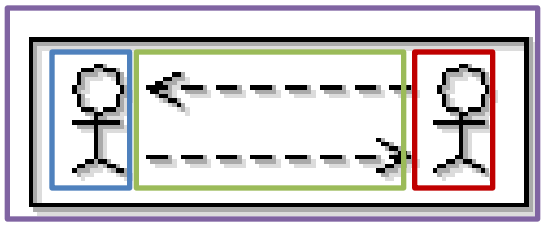
## GRAPHREP Implementation for Class "H"




1. Since this class is concrete, a graphical representation needs to be defined.
2. Use inherited class attribute "GraphRep" to define the graphical representation
3. Write GRAPHREP code to provide a notation for the class

# Commented GraphRep Code: H

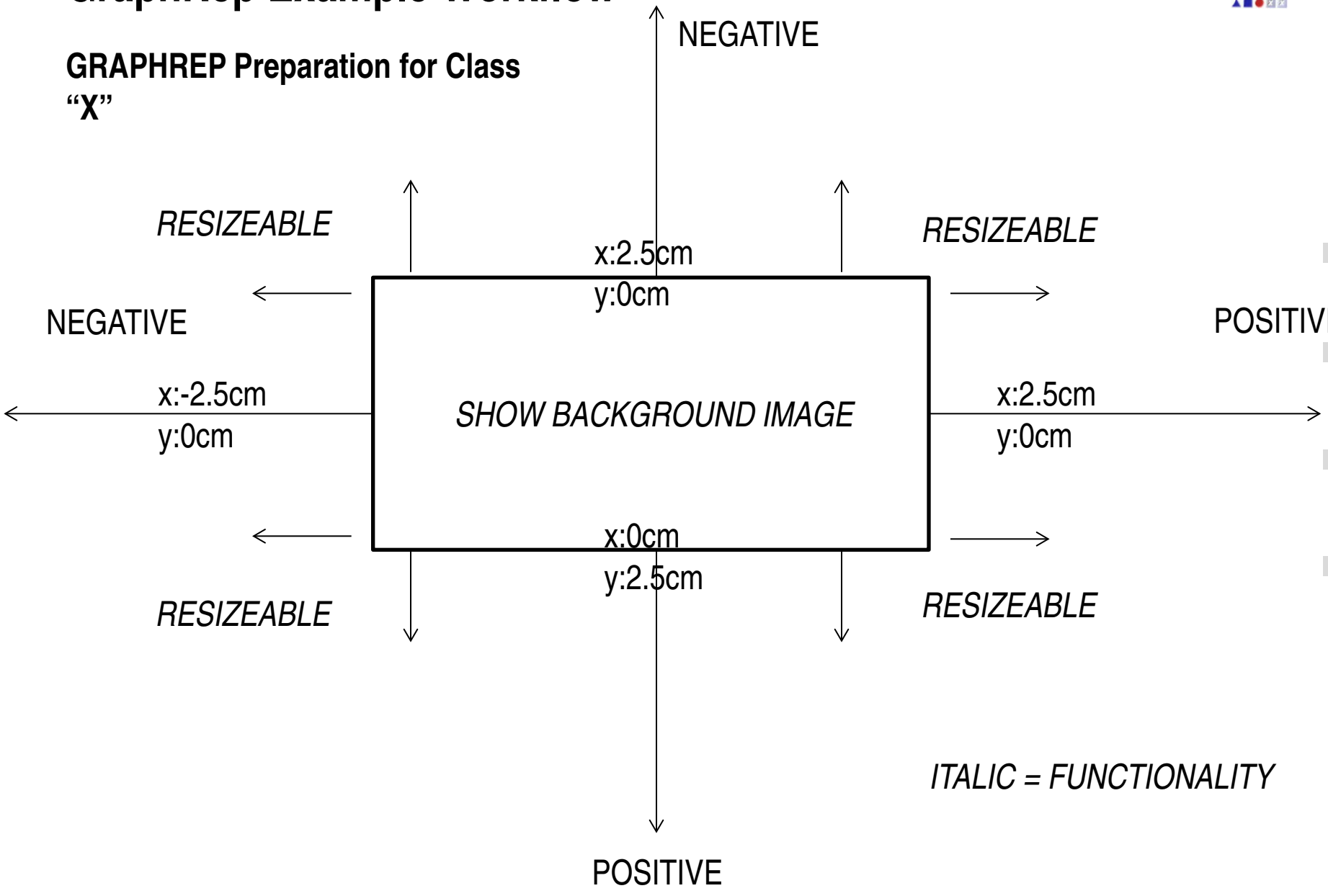
```
GRAPHREP
#Container Rectangle
RECTANGLE x:-1.5cm y:-0.5cm w:3cm h:1cm
#Arrow Lines
PEN style:dash
LINE x1:-0.8cm x2:0.8cm y1:-0.2cm y2:-0.2cm
LINE x1:-0.8cm x2:0.8cm y1:0.2cm y2:0.2cm
#Arrow Ends
PEN style:solid
LINE x1:-0.8cm x2:-0.6cm y1:-0.2cm y2:-0.1cm
LINE x1:-0.8cm x2:-0.6cm y1:-0.2cm y2:-0.3cm
LINE x1:0.8cm x2:0.6cm y1:0.2cm y2:0.1cm
LINE x1:0.8cm x2:0.6cm y1:0.2cm y2:0.3cm
#Right actor
ELLIPSE x:1.1cm y:-0.2cm rx:0.15cm ry:0.15cm
LINE x1:1.1cm x2:1.1cm y1:-0.05cm y2:0.2cm
LINE x1:1.1cm x2:0.95cm y1:0.2cm y2:.3cm
LINE x1:1.1cm x2:1.25cm y1:0.2cm y2:.3cm
LINE x1:0.95cm x2:1.25cm
#Left actor
ELLIPSE x:-1.1cm y:-0.2cm rx:0.15cm ry:0.15cm
LINE x1:-1.1cm x2:-1.1cm y1:-0.05cm y2:0.2cm
LINE x1:-1.1cm x2:-0.95cm y1:0.2cm y2:.3cm
LINE x1:-1.1cm x2:-1.25cm y1:0.2cm y2:.3cm
LINE x1:-0.95cm x2:-1.25cm
#Attribute Representation
ATTR "Name" y:0.8cm w:c h:c
```



In case attribute name is available, it is shown here

# GraphRep Example Workflow

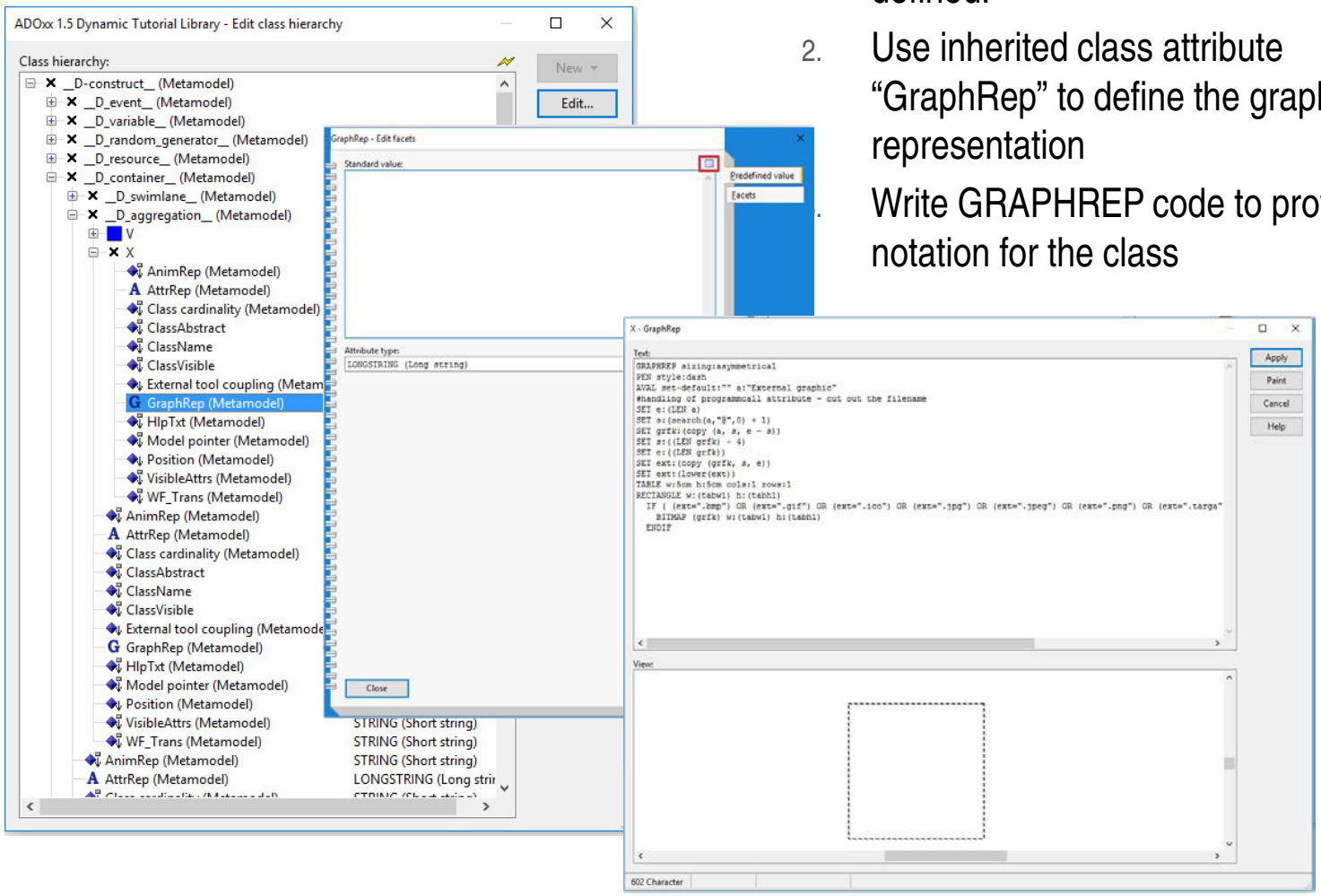
## GRAPHREP Preparation for Class "X"



# GraphRep Example Workflow

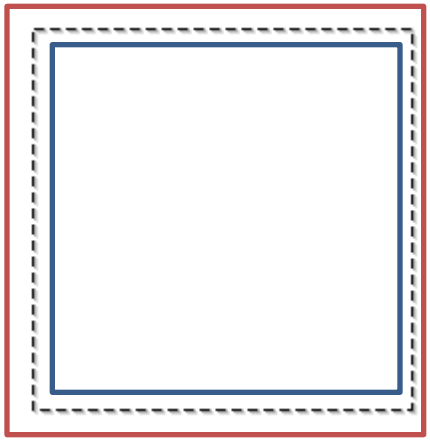
## GRAPHREP Implementation for Class "X"

1. Since this class is concrete, a graphical representation needs to be defined.
2. Use inherited class attribute "GraphRep" to define the graphical representation
3. Write GRAPHREP code to provide a notation for the class

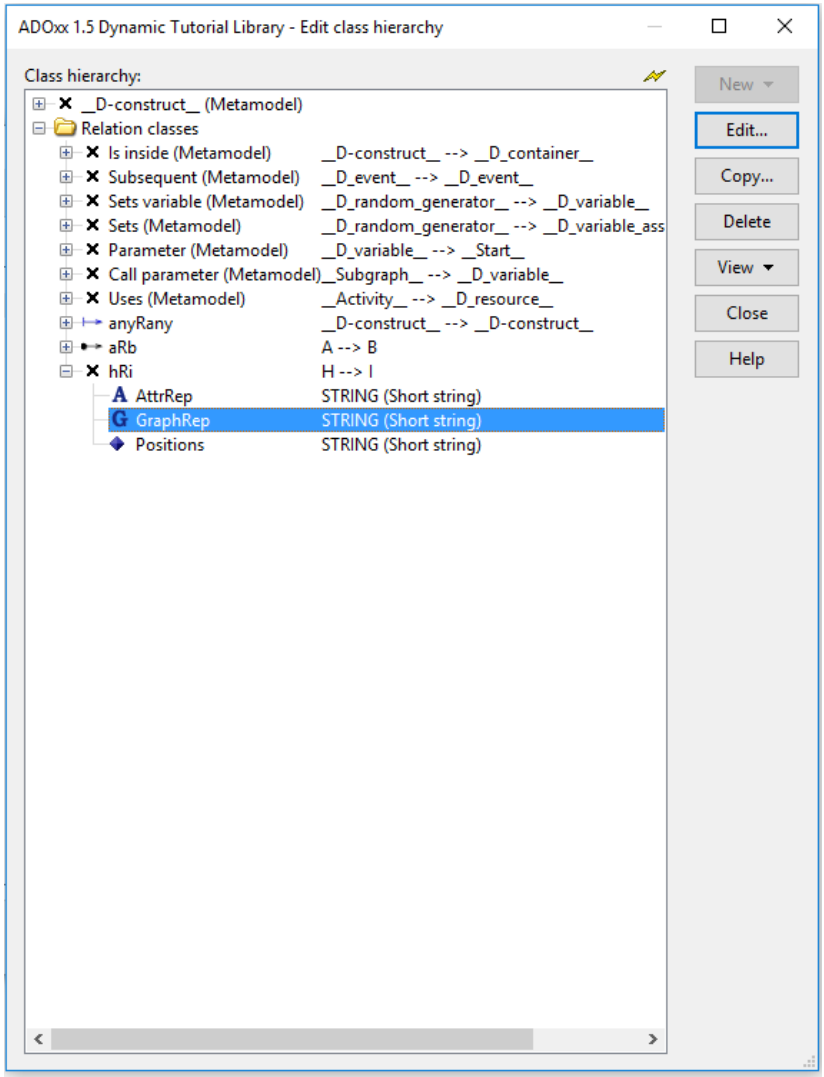


# Commented GraphRep Code: X

```
GRAPHREP sizing:asymmetrical                    RESIZE
PEN style:dash
AVAL set-default:"" a:"External graphic"
#handling of programmcall attribute - cut out
the filename
SET e:(LEN a)
SET s:(search(a,"@",0) + 1)
SET grfk:(copy (a, s, e - s))
SET s:((LEN grfk) - 4)
SET e:((LEN grfk))
SET ext:(copy (grfk, s, e))
SET ext:(lower(ext))
TABLE w:5cm h:5cm cols:1 rows:1          FILE HANDLING
RECTANGLE w:(tabw1) h:(tabh1)
IF ( (ext=".bmp") OR (ext=".gif")OR(ext=".ico")
OR (ext=".jpg") OR (ext=".jpeg") OR
(ext=".png") OR (ext=".targa") OR (ext=".tiff")
OR (ext=".wbmp") OR (ext=".xpm") )
  BITMAP (grfk) w:(tabw1) h:(tabh1)
ENDIF
                                          IMAGE HANDLING
```



# Commented GraphRep: hRi (uni-directional)




```
GRAPHREP rounded:0.05cm
SHADOW mode:off
PEN color:red w:0.02cm color:$727272
EDGE                                    GRPHREP OF EDGE

END
FILL color:red
POLYGON 3 x1:-0.2cm y1:0.11cm x2:0cm y2:0cm
x3:-0.2cm y3:-0.11cm                    GRAPHREP END
```

# Commented GraphRep: bi-directional example

```
GRAPHREP rounded:0.05cm
SHADOW mode:off
PEN color:red w:0.02cmcolor:$727272
    style:dash
START
FILL color:red
POLYGON 3 x1:-0.2cm y1:0.11cm x2:0cm y2:0cm
        x3:-0.2cm y3:-0.11cm
```

GRAPHREP START

```
EDGE
```

GRPHREP OF EDGE

```
END
FILL color:red
POLYGON 3 x1:-0.2cm y1:0.11cm x2:0cm y2:0cm
          x3:-0.2cm y3:-0.11cm
```

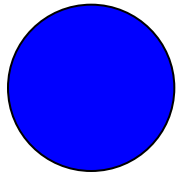
GRAPHREP END

# 2. SPECIAL CLASS ATTRIBUTE & ATTRIBUTE GRAPHREP EXAMPLES

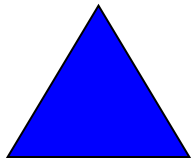
# GraphRep Examples

## Basic Forms

```
GRAPHREP
SHADOW off

FILL color:blue
ELLIPSE x:0.00cm y:0cm rx:1cm ry:1cm

ATTR "Name" x:0.00cm y:1.0cm w:c
```



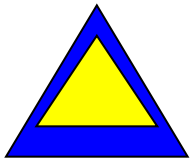
```
GRAPHREP

FILL color:blue
POLYGON 3 x1:-1cm y1:1cm x2:0cm
y2:-1cm x3:1cm y3:1cm

ATTR "Name" x:0cm y:1cm w:c
```



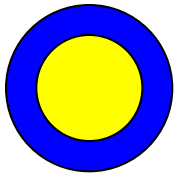
# GraphRep Examples

## Combined Elements 1

```
GRAPHREP

FILL color:blue
POLYGON 3 x1:-1cm y1:1cm x2:0cm y2:-1cm x3:1cm
y3:1cm
FILL color:yellow
POLYGON 3 x1:-0.6cm y1:0.6cm x2:0cm y2:-0.6cm
x3:0.6cm y3:0.6cm

ATTR "Name" x:0cm y:1cm w:c
```



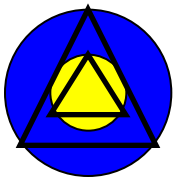
```
GRAPHREP
SHADOW off

FILL color:blue
ELLIPSE x:0.00cm y:0cm rx:1cm ry:1cm

ATTR "Name" x:0.00cm y:1.0cm w:c
```



# GraphRep Examples

## Combined Elements 2

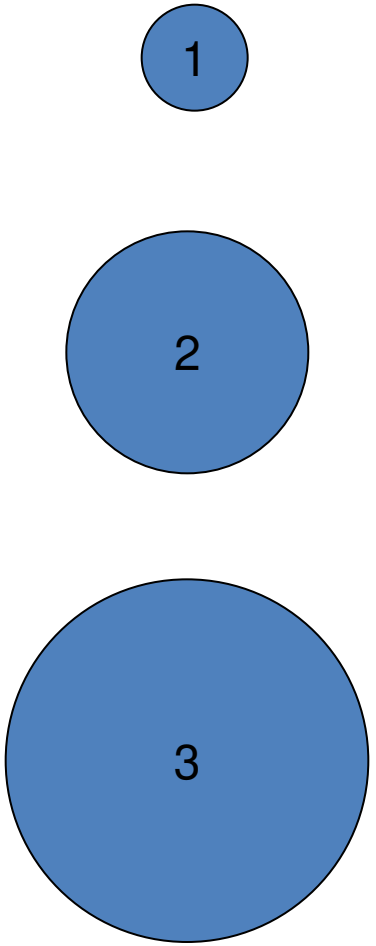
```
GRAPHREP
SHADOW off

FILL color:blue

PEN style:solid w:0.01cm
ELLIPSE x:0.00cm y:0cm rx:1cm ry:1cm
PEN style:solid w:0.1cm
POLYGON 3 x1:-0.8cm y1:0.6cm x2:0cm y2:-1cm x3:0.8cm
y3:0.6cm

FILL color:yellow
PEN style:solid w:0.01cm
ELLIPSE x:0.00cm y:0cm rx:0.5cm ry:0.5cm
PEN style:solid w:0.1cm
POLYGON 3 x1:-0.4cm y1:0.3cm x2:0cm y2:-0.4cm x3:0.4cm
y3:0.3cm

ATTR "Name" x:0.00cm y:1.0cm w:c
```



# GraphRep Examples

## Conditional representation - Sizing

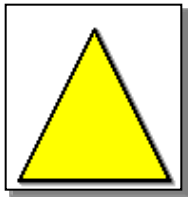
```
GRAPHREP
SHADOW off

AVAL set-default: 2 ar:"number of counts"

TEXT (ar)

FILL color:lightgray
ELLIPSE x:0.0cm y:0cm rx:(CM (ar)) ry:(CM (ar))

ATTR "number of counts" x:0.0cm y:-0.05cm w:c

ATTR "Name" x:0.00cm y:1.0cm w:c
```



# GraphRep Examples:

## Basic forms

```
GRAPHREP
PEN w:0.05cm
FILL color:yellow
POLYGON 3 x1:-.7cm y1:.7cm x2:.7cm y2:.7cm x3:0cm y3:-.7cm
ATTR "Name" y:.8cm w:c:2.8cm h:t
```



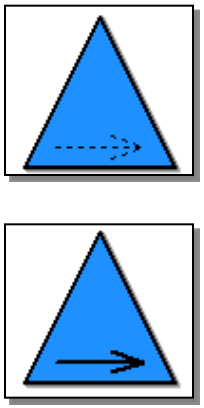
```
GRAPHREP
PEN w:0.05cm
FILL color:dodgerblue
RECTANGLE x:-1.4cm y:-.7cm w:2.8cm h:1.4cm
ATTR "Name" y:.8cm w:c h:t
```

```
GRAPHREP
FILL color:mediumspringgreen
ELLIPSE rx:0.70cm ry:0.70cm
ATTR "Name" y:0.8cm w:c:1.4cm h:t
FONT "Arial" h:32pt color:black
TEXT "V" y:0.13cm w:c h:c
```

# GraphRep Examples:

## Conditional representation (1)

```
GRAPHREP
AVAL col:"fontcolor"
AVAL set-default:"x" p:"referenced process"
AVAL sub:"referenced process "
AVAL i:"Sequence"
AVAL sn:"subprocessname"
FILL color:dodgerblue
PEN w:0.05cm
POLYGON 3 x1:-.7cm y1:.7cm x2:.7cm y2:.7cm x3:0cm y3:-.7cm
SHADOW mode:off
IF (NOT LEN p)
  PEN style:dot
ENDIF
LINE x1:-.4cm y1:.5cm x2:.4cm y2:.5cm
LINE x1:.1cm y1:.4cm x2:.4cm y2:.5cm
LINE x1:.1cm y1:.6cm x2:.4cm y2:.5cm
FONT color:(col)
IF (sub = "")
  ATTR "Name" y:.8cm w:c:2.8cm h:t
ELSE
  FONT "Arial" h:8pt bold
  ATTR "referenced process" y:(texty2 + .1cm) w:c:2.8cm h:t format:"%m"
  FONT
ENDIF
```

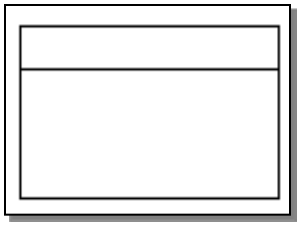


Process call with / without a reference

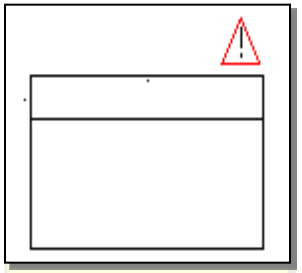
# GraphRep Examples:

## Conditional representation (2)

```
GRAPHREP
AVAL set-default:"Modeling finished" b:"Status"
SHADOW off
FILL style:null
POLYGON 4 x1:-1.54cm y1:0.92cm x2:1.54cm y2:0.92cm
          x3:1.54cm y3:-0.98cm x4:-1.54cm y4:-0.98cm
LINE x1:-1.54cm y1:-0.50cm x2:1.54cm y2:-0.50cm
IF (b = "Modeling not finished")
  LINE x1:1.25cm y1:-1.5cm x2:1.25cm y2:-1.3cm
  LINE x1:1.25cm y1:-1.22cm x2:1.25cm y2:-1.18cm
  PEN color:red
  POLYGON 3 x1:1cm y1:-1.1cm x2:1.25cm y2:-1.6cm
            x3:1.50cm y3:-1.1cm
ENDIF
```



Condition fulfilled



Condition not fulfilled

# GraphRep Examples:

## Tables

```
GRAPHREP sizing:asymmetrical
SHADOW off
PEN color:black
FILL style:null
TABLE x:-3.5cm y:-2cm w:7cm h:4cm
      cols:3 rows:4
      w1:1.3cm w2:50% w3:50%
      h1:1cm h2:0.5cm h3:0.5cm h4:100%
```

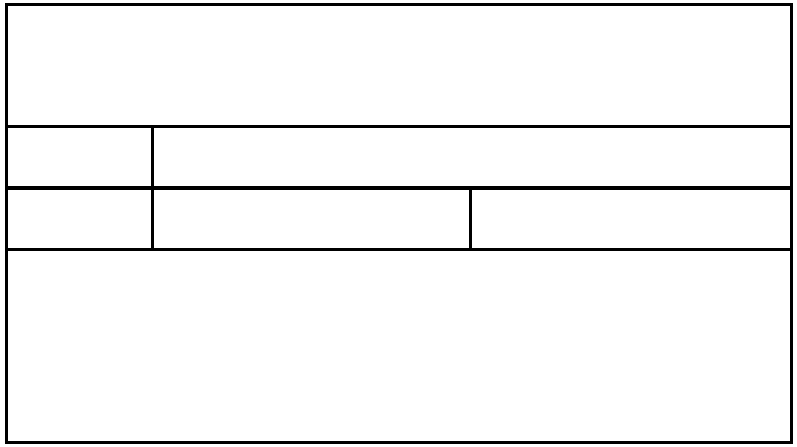


Table with 4 rows and 3 columns

**Hint:**
When manually changing the size of the table only the parameters having values specified as **percent** will change in size. Fields with absolute values will always stay the same.

# GraphRep Examples:

## Table borders

Tables can be drawn with or without borders. Borders are defined as lines using the corners of the tables cells.

For instance: the top left corner of the table has the coordinate (tabx0, taby0), the top right corner of the first cell has (tabx1, taby0) etc.

```
LINE x1:(tabx0) y1:(taby0) x2:(tabx3) y2:(taby0)
LINE x1:(tabx0) y1:(taby1) x2:(tabx3) y2:(taby1)
LINE x1:(tabx0) y1:(taby2) x2:(tabx3) y2:(taby2)
LINE x1:(tabx0) y1:(taby3) x2:(tabx3) y2:(taby3)
LINE x1:(tabx0) y1:(taby4) x2:(tabx3) y2:(taby4)

LINE x1:(tabx0) y1:(taby0) x2:(tabx0) y2:(taby4)
LINE x1:(tabx1) y1:(taby1) x2:(tabx1) y2:(taby3)
LINE x1:(tabx2) y1:(taby2) x2:(tabx2) y2:(taby3)
LINE x1:(tabx3) y1:(taby0) x2:(tabx3) y2:(taby4)
```

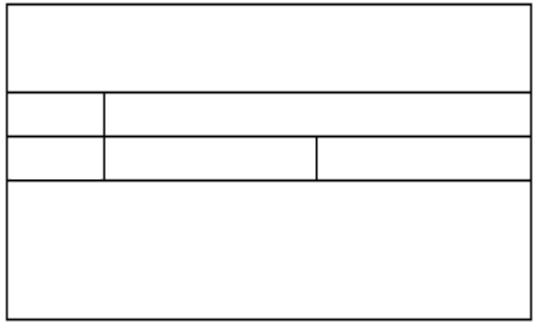


**Table with 4 rows and 3 columns only some lines are arranged**

# GraphRep Examples:

## Complex, attribute dependent representations

```
GRAPHREP
AVAL a:"External Documentation"
PEN w:0.1cm
FILL r:200 g:200 b:200
POLYGON 4 x1:0cm y1:-1cm x2:1cm y2:0cm
          x3:0cm y3:1cm x4:-1cm y4:0cm
ATTR "Name" y:1.2cm w:c:2.8cm h:t
IF (search(lower(a),"winword",0) >= 0)
  PEN w:0.07cm
  FILL r:0 g:255 b:255
  ...

  IF (search(lower(a),".doc",0) >=0)
    ...
  ENDIF
ELSIF (search(lower(a),"powerpnt",0) >= 0)
  ...
ENDIF
```

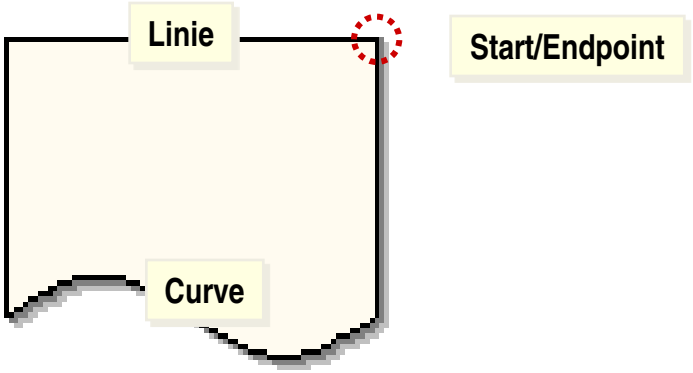
Search for Text pattern

Nested conditions

# GraphRep Examples:

## Compound representation

```
GRAPHREP
COMPOUND 2
LINE x1:1.0cm y1:-.7cm x2:-1.0cm y2:-.7cm
CURVE "t" f:(t) g:(-.2*sin(3.14*(t+1))+.7) from:-1 to:1
```

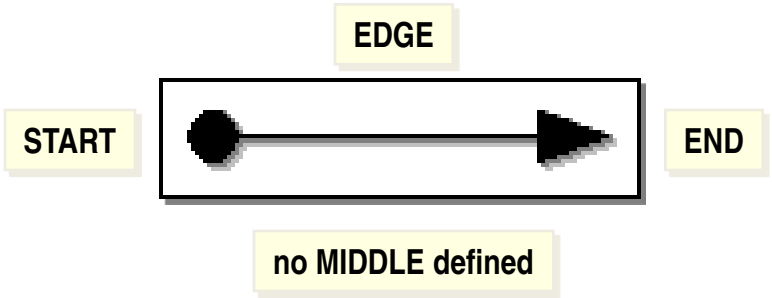



**Hint:**

- The compound consists of ***one*** **line and** ***one*** **curve**.
- The endpoint of the previous element is the start point for the following.
- A connection is made automatically between to elements if necessary. (sequence is important!).

# GraphRep Definition for Relation Classes

- The same commands from normal classes can be used for relation classes as well. In addition the following keywords are available:
- EDGE
  - Defines the representation of the relation edge (line).
- START / MIDDLE / END
  - This command defines the representation of the important edge parts. If **MIDDLE** is defined, then the middle of the edge can be moved in the model.

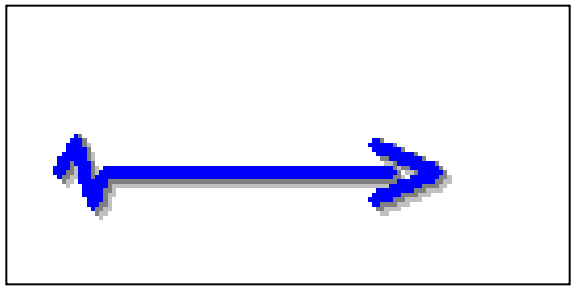


# GraphRep Example:

## Connector

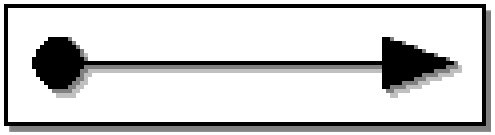
A

```
GRAPHREP
PEN color:lightblue w:.08cm
EDGE start-trans:-.3cm end-trans:-.3cm
START
POLYLINE 4 x1:0cm y1:0cm x2:-.1cm y2:.18cm
           x3:-.2cm y3:-.18cm x4:-.3cm y4:0cm
END
POLYLINE 3 x1:-.4cm y1:.15cm x2:0cm y2:0cm
           x3:-.4cm y3:-.15cm
```



B

```
GRAPHREP
START
FILL color:black
ELLIPSE x:-.1cm rx:.1cm ry:.1cm
END
LINE x1:-.3cm y1:.1cm x2:0cm y2:0cm
LINE x1:-.3cm y1:-.1cm x2:0cm y2:0cm
```



# 2. SPECIAL CLASS ATTRIBUTE & ATTRIBUTE ATTRREP

# Basics: AttrRep

- The class attribute „AttrRep“ controls the availability and structure of the ADOxx-Notebook. If it has no value then the class will have no Notebook.
- The following elements are available to define the Notebook:
  - **Chapter:** Each Notebook must have at least one chapter to show some attributes. Chapters of a Notebook are shown as tabs on the right side.
  - **Attributes:** Attributes are embedded in a chapter where they should be shown. The distribution and sequence of the attributes is also defined in the AttrRep.
  - **Groups:** Attributes can be combined to groups inside of a chapter.

# The AttrRep-Commands

**NOTEBOOK**

The Notebook-Definition must start with this command to be valid. It has no parameters.

**CHAPTER „*chapterName*"**

Chapters can be started with this command. The chapter will have the name <chapterName> (Hint: A command ENDCHAPTER is not necessary)

**ATTRIBUTE „*AttrName*"**

The attribute with the name <AttrName> will be shown in the notebook on this position. Some attribute types also allow different parameters to adapt the actual display.

**GROUP „*groupName*" / ENDGROUP**

The attributes listed between GROUP and ENDGROUP will be enclosed by a group-box with the name <groupName>

**SET_ACCESS usergroup: *userGroupSpec***

Attributes following this command will only be shown to the user group <userGroupSpec>. This restriction can be revoked using „**SET_ACCESS usergroup: all**"

# ATTRREP

- Classattribute “AttrRep“ is of type long string, hence the text entered as value is interpreted as configuration script of the so-called **NOTEBOOK**.
- Each **NOTEBOOK** has **CHAPTERS**, which contains a list of attributes that may be grouped.
- Relations that are allowed for this class can be automatically created as an own chapter.
- Appearance of attributes is defined by **lines**, **dialog**, control types (**ctrltype**), **width** or **format**.
- Access rights per attribute can be defined.

# AttrRep Syntax Reference

| | |
|---|---|
| *Notebook* : | **NOTEBOOK** [ **with-relations** \| **move-relations:***intValue* ]<br>{ *NBElement* \| *SetAccess* \| *Language* } . |
| *NBElement* : | *Chapter* \| *Group* \| *Attribute* \| *Set* \| *AVal* . |
| *Chapter* : | **CHAPTER** *chapterName* [ **color:***ColorSpec* ] . |
| *Group* : | **GROUP** *groupName* [ **color:***ColorSpec* ]<br>{ *Attribute* }<br>**ENDGROUP** . |
| *Attribute* : | **ATTR** *attrName*<br>[ **write-protected** ] [ **format:***strValue* ] [ **dialog:***Dialog* ]<br>[ **lines:***intValue* ] [ **font-family:***FontFamily* ]<br>[ **color:***ColorSpec* ] [ **ctrltype:***ControlType* ]<br>[ **unchecked-value:***strValue* ] [ **checked-value:***strValue* ]<br>[ **no-auto** ] [ **no-param** ] [ **push-button** ]<br>[ **formula-visible** ] [**align**:*Alignment* ]<br>[ **enabled:***boolExpr* ] [ **mandatory:***boolValue* ]<br>[ **notitle** ] [ **width:***realValue* ] [ **h:***measureValue* ] . |
| *FontFamily* : | **decorative** \| **modern** \| **roman** \| **script** \| **swiss** \| **system** . |
| *Dialog* : | **time** \| **date** \| **datetime** \| **distribution** \| **actor** \| **subprocess** \|<br>**person-calendar** \| **processstart-calendar** \| **resource** \|<br>**color** \| **transcond** \| **acfilter** \| **wizard** . |
| *ControlType* : | **radio** \| **dropdown** \| **check** \| **graphrep** . |
| *Set* : | **SET** *var* :*expression* |
| *AVal* : | **AVAL** { *AvalAssignment* } . |
| *AValAssignment* : | [ **set-format:***strValue* ] [ **set-sep:***strValue* ]<br>[ **as-original-type** ] [ **set-count-rows** ]<br>*var* :*attrName* . |
| *SetAccess* : | **SET_ACCESS usergroup:***UserGroupSpec* **mode:***AccessMode* . |
| *UserGroupSpec* : | *userGroupName* \| **all** . |
| *AccessMode* : | **blocked** \| **protected** \| **full** . |
| *Alignment* : | **l** \| **c** \| **r** . |
| *Language* : | **LANG** *LangSpec* . |
| *LangSpec* : | *langName* \| **all** . |

# Example for a AttrRep Definition in ALL

```
CLASSATTRIBUTE <AttrRep>
TYPE STRING
VALUE  "
       NOTEBOOK with-relations
       CHAPTER \"Description\"
       ATTR \"Name\"
       ATTR \"Presentation\"
       ATTR \"Description\" lines:5
       ATTR \"Comment\" lines:5
       ATTR \"Color\" dialog:color
       "
       FACET <MultiLineString>
       VALUE 0

       FACET <AttributeHelpText>
       VALUE ""

       FACET <AttributeRegularExpression>
       VALUE ""
```

Keyword

Attribute name:
“AttrRep“ is a special attribute which defines what other attributes are processed by the ADOxx documentation function

Type definition

Value:
The string of the “AttrRep” attributes is defined as a “Notebook”. Therefore a specific syntax is used.

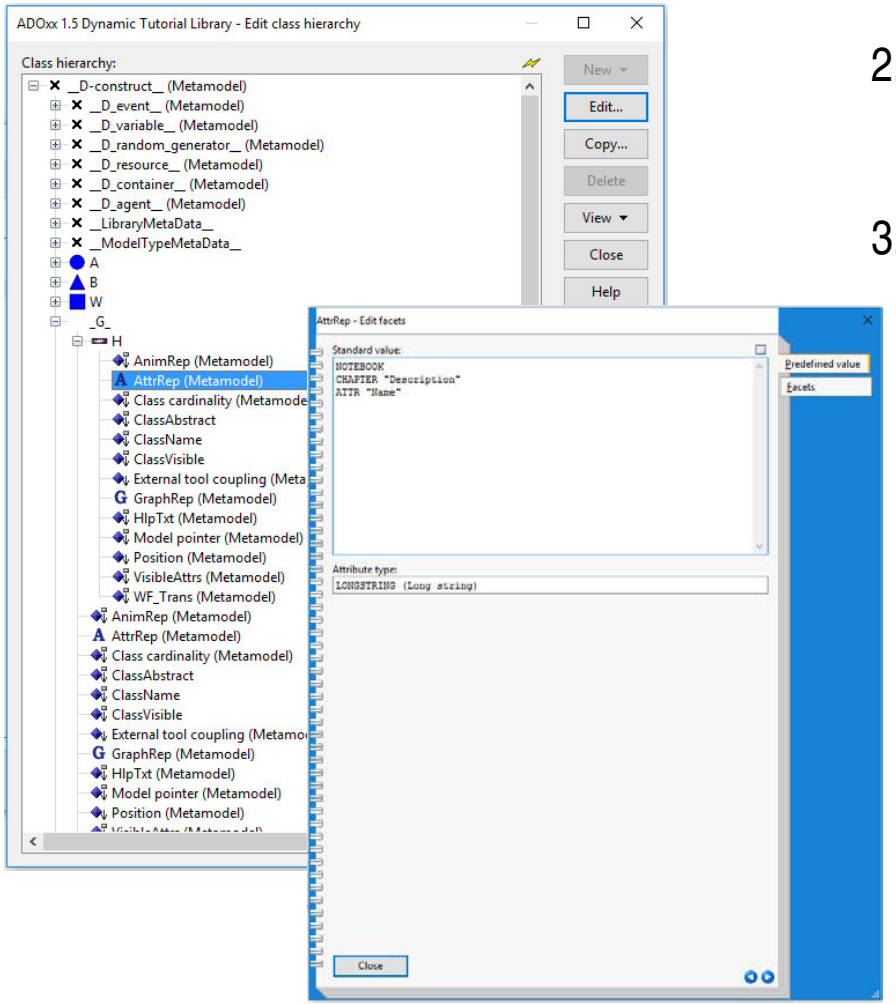
A help text can be provided for the attribute.

# 2. SPECIAL CLASS ATTRIBUTE & ATTRIBUTE

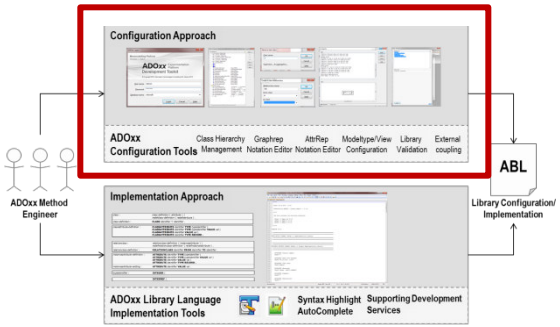
# ATTRREP

# HANDS-ON

# Example Workflow AttrRep




1. Since this class is concrete, a attribute representation needs to be defined.
2. Use inherited class attribute “AttrRep” to define the attribute representation
3. Write ATTRREP code to provide a notation for the class



# Commented AttrRep Code

```
NOTEBOOK
CHAPTER "Definition"
ATTR "Name"
GROUP "Definition"
ATTR "Description"
ATTR "External content"
ENDGROUP
```

```
NOTEBOOK
CHAPTER "Definition"
ATTR "Name"
ATTR "Description"
CHAPTER "Dialectic Influence"
ATTR "Influencing dialectics" lines:10
```

```
NOTEBOOK
CHAPTER "Definition"
ATTR "External graphic"
```

Chapter Structure

Attributes

Grouping of attributes on same chapter

Attribute Representation

# 2. SPECIAL CLASS ATTRIBUTE & ATTRIBUTE CLASS CARDINALITIES

# Class Attribute “Class cardinality”

- The class attribute „Class cardinality “ contains the cardinality definition of the current class.
  The cardinality of a class describes
  - **the minimal/maximal number of objects of this class per model** und
  - **the minimal/maximal number of relations of a specific type**,
    incoming or outgoing from the object.

- If no cardinalities are defined then there are also no restrictions for this class.

**Hint:**

- A validation of the class cardinality can be performed in the toolkit either with each save or only when manually selecting the function (depending on the customizing).
- Please consult the ADOxx-Manual volume 4 for a detailed description of the cardinality definition.

# Commands of the Class Cardinality

**CARDINALITIES**

The cardinality definition must start with this command to be valid. It has no parameters.

**RELATION „*RelationName*"**

Restricts the following commands to the relation class with the name <*RelationName*>.

**FROM_CLASS „*ClassName*" / TO_CLASS „*ClassName*"**

Restricts the following commands to relations with the class of <*ClassName*>.

# Parameters of the Class Cardinality

**min-objects / max-objects**

Specifies how many objects of a class can minimally/maximally be available in the model.

**min-relations / max-relations**

Specifies the minimal/maximal number of relations which can be connected with this object from this class.

**max-outgoing / min-outgoing / max-incoming / min-incoming**

Restricts the number of allowed incoming/outgoing relations; either:

- in general or
- with a preceding **RELATION** command only for this relation or
- with a preceding **FROM_CLASS** or **TO_CLASS** command only for relations to these classes.

# 2. SPECIAL CLASS ATTRIBUTE & ATTRIBUTE CLASS CARDINALITIES HANDS-ON

# Class cardinality: Examples

- Only one object of the class „A" should exist per model.
- As well no connectors anyRany schould exist incomming to objects of class „A" and only one connector anyRany maximum should exist outgoing from objects of class „A".

- The cardinalities of the class „A" have to be defined in the following way:
- CARDINALITIES max-objects:1
  RELATION "anyRany" max-incoming:0 max-outgoing:1

# 2. SPECIAL CLASS ATTRIBUTE & ATTRIBUTE CONVERSION

# Class Attribute „__Conversion__“

The class attribute „Conversion“ defines and controls the conversion of a modeling object from one class to another.

When converting three things happen. First a new object of the defined class is created. Afterwards all attribute values are copied into the new object as defined in the “Conversion” attribute. In the end the old object is deleted.

**Hint:**

- The possibility for the conversion must be defined manually in the metamodel, so it can be used later in the tool.
- The modeler can access the functionality from the context menu in the ADOxx-BPM-Toolkit.

# Commands and Parameters for Conversion

**CLASS "*ClassName*"**

Specifies that an object can be converted into the target class <*ClassName*>. Several target classes can be specified.

**ATTR "*AttrName*"**

Defines the attributes from which the values will be copied during the conversion.

**from**

This parameter is used if values should be copied from the source object to the target object, but the corresponding attributes have different names. **from** specifies the name of the source attribute.

**Hint:**
A detailed description of the Conversion-Grammar can be found in the ADOxx-Manual volume 4.

# The commands and parameters for the conversion

**If you define __*Conversion*__ for the class „A" with**

***CLASS „B"***
***ATTR „ba1"***
***ATTR „ba2" from: „aa3"***

| | |
|---|---|
| *Conversion* : | { *ClassConversion* } . |
| *ClassConversion* : | **CLASS** *className* { *AttrConversion* } . |
| *AttrConversion* : | **ATTR** *attrName* [ **from:***attrName* ] . |

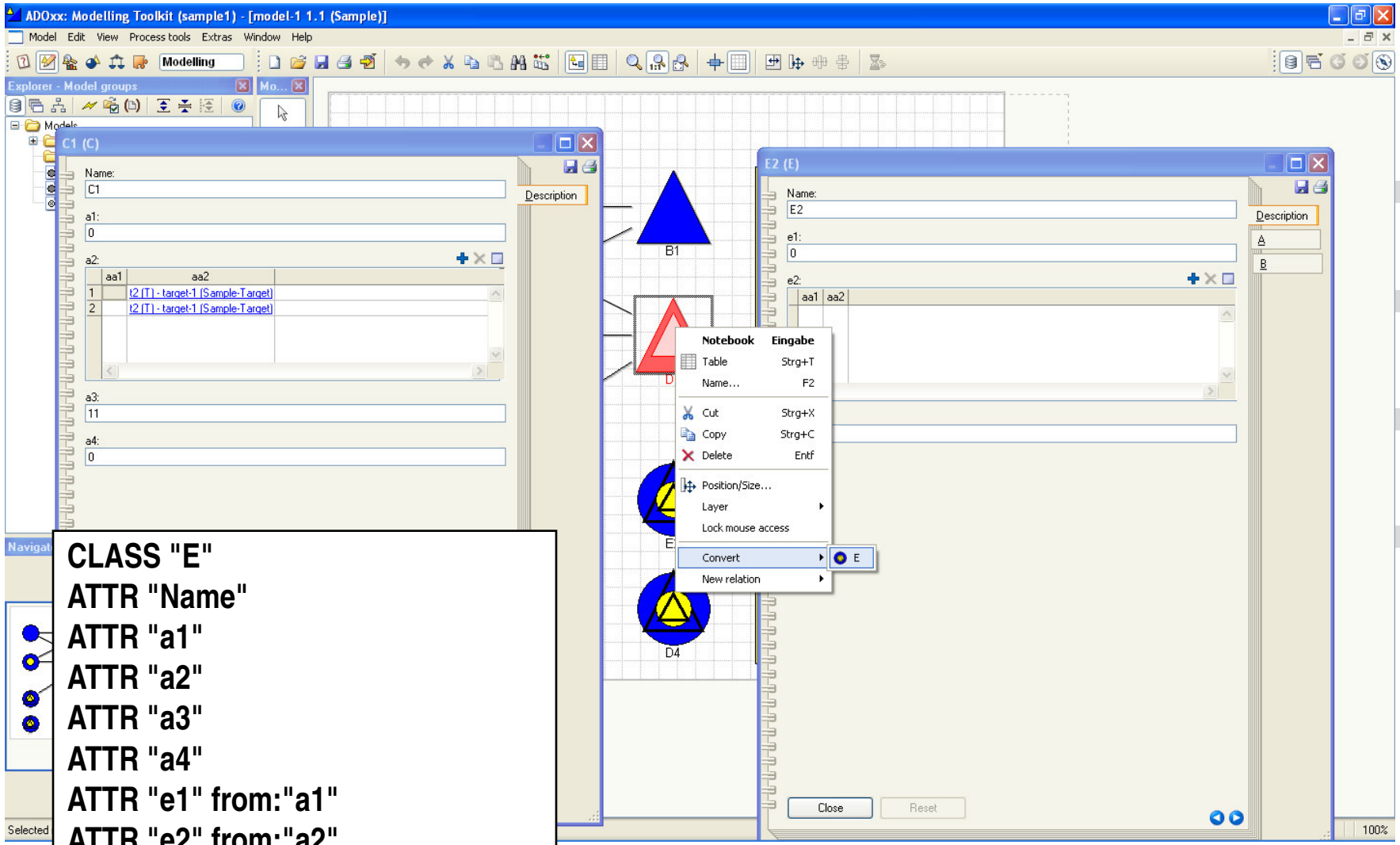
**this means that**

- objects of class „A" can be converted to objects of class „B",
- the aa1 is assigned from A to ba1 in B as the have the same name,
- the aa3 from A is assigned to Ba2 from B as they have different names,

# 2. SPECIAL CLASS ATTRIBUTE & ATTRIBUTE CONVERSION HANDS-ON

# Conversion example:

## Instances of C->E




```
CLASS "E"
ATTR "Name"
ATTR "a1"
ATTR "a2"
ATTR "a3"
ATTR "a4"
ATTR "e1" from:"a1"
ATTR "e2" from:"a2"
ATTR "e3" from:"a3"
```

# 2. SPECIAL CLASS ATTRIBUTE & ATTRIBUTE

# MODEL POINTER

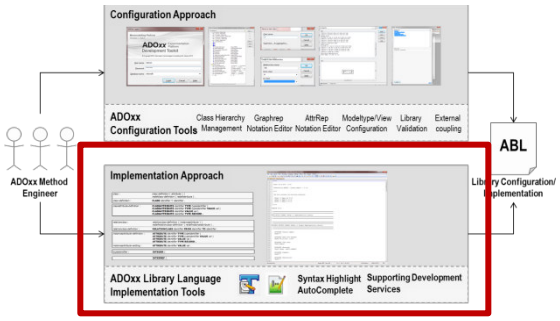
# The class attribute “Model pointer“

The class attribute „Model pointer“ priorities one specified pointer with the ability to get from one object in a model directly to another model.

The name of the attribute which provides the reference to another model or object is specified in the model pointer attribute field.

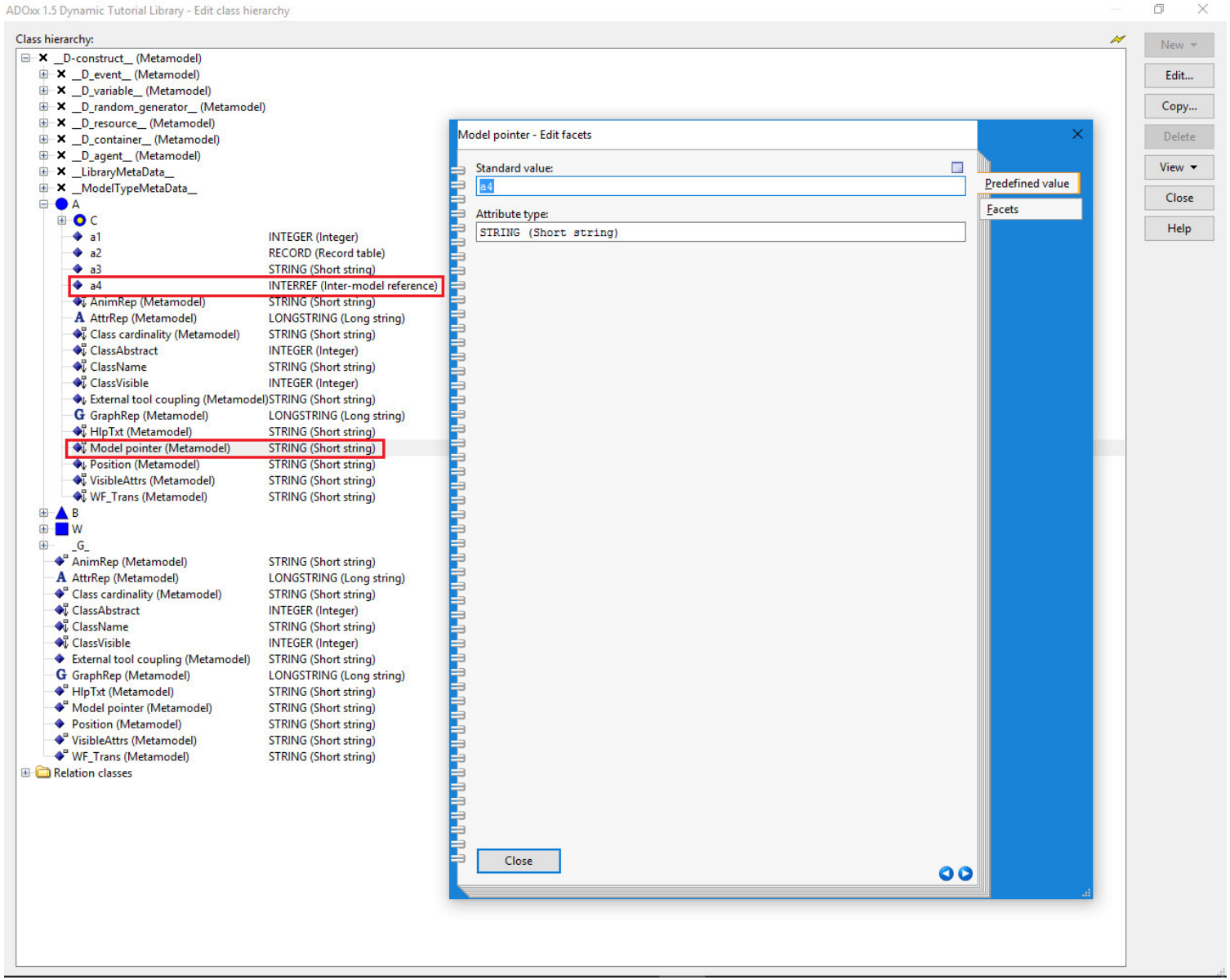
ADOxx provides a short cut with <Ctrl> + double click to follow the pointer

```
CLASSATTRIBUTE <Model pointer>
        VALUE "ra"
        ATTRIBUTE <ra>
        TYPE INTERREF
                FACET <MultiLineString>
                VALUE 0

                FACET <AttributeHelpText>
                VALUE "helptext"

                FACET <AttributeInterRefDomain>
                VALUE "VALUE "REFDOMAIN max:1
                        OBJREF
                        mt:\"my model type\"
                        c:\"my class\"
                        max:1 "
```

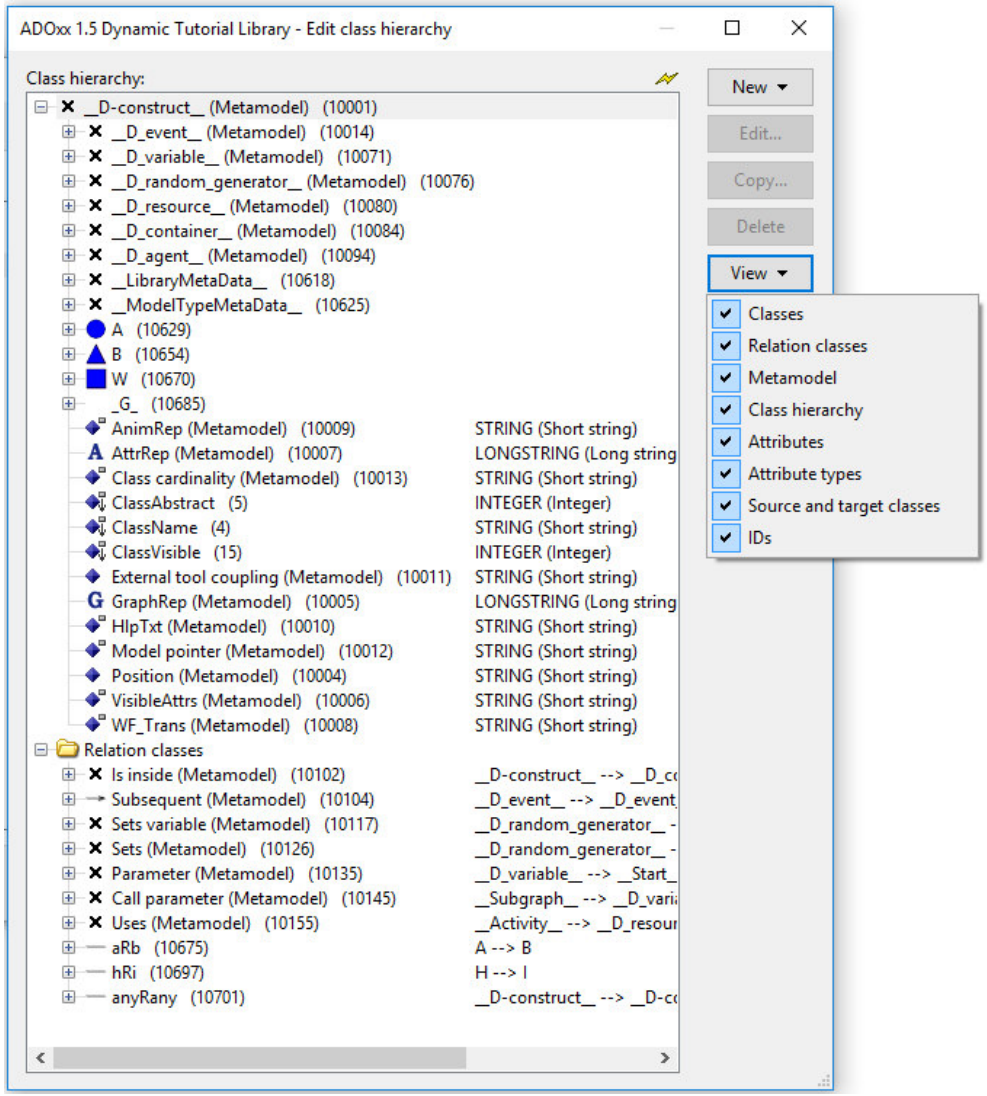


# 2. SPECIAL CLASS ATTRIBUTE & ATTRIBUTE MODEL POINTER HANDS-ON

# Model pointer: Example



# Views of the class hierarchy




***Classes***

**All visible classes will be shown**

***Relation classes***

**All available relation classes will be shown**

***Metamodel***

**All classes will be shown**

***Class hierarchy***

**All classes will be shown with their inheritance in a hierarchy**

***Attributes***

**The attributes of the (relation-)classes will be shown**

***Attribute types***

**The type of each attribute will be shown**

***Source- and Target-classes***

**Shows the endpoints for each relation class, i.e. between which classes it can be used.**

***IDs***

**Shows ID numbers of classes and attributes**

# Icons in ADOxx class hierarchy management


**Class** (the icon shows the graphical definition of the object and can therefore vary)

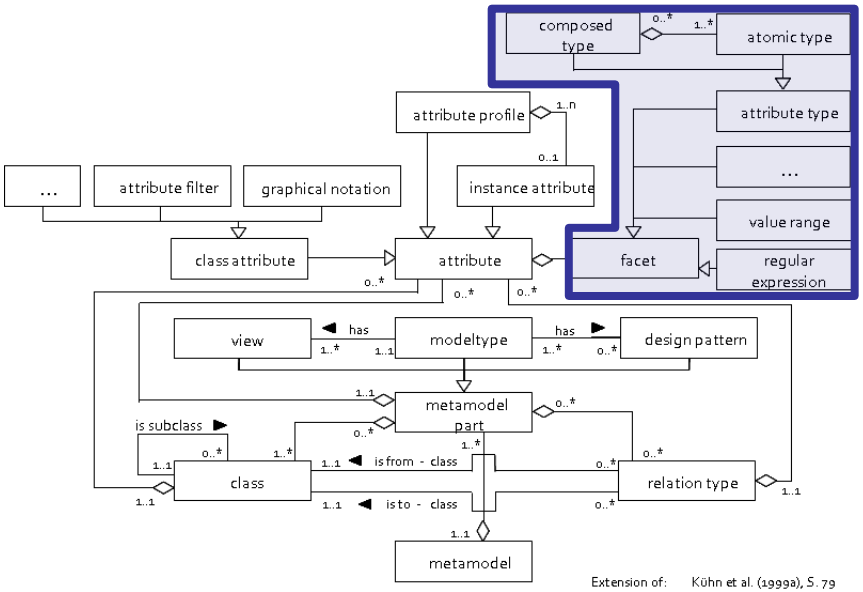

**Class** (without a graphical definition)


**Attribute**


**Attribute** (inherited from another class)


**Class attribute**


**Class attribute** (inherited from another class)




# 3. ATTRIBUTE FACETS

# Attribute Facets Correlation

| | AttributeNumeric Domain | AttributeRegular Expression | AttributeInterref Domain | Enumeration Domain | MultiLineString | AttributeHelp Text | RecordClass Name | RecordClass Multiplicity |
|---|---|---|---|---|---|---|---|---|
| INTEGER | X | | | | | X | | |
| DOUBLE | X | | | | | X | | |
| STRING | | X | | | X | X | | |
| LONGSTRING | | X | | | X | X | | |
| TIME | | | | | | X | | |
| ENUMERATION | | X | | X | | X | | |
| ENUMERATIONLIST | | X | | X | | X | | |
| PROGRAMCALL | | | | X | | X | | |
| RECORD | | | | | | X | X | X |
| EXPRESSION | | | | | X | X | | |
| INTERREF | | | X | | | X | | |

# Attribute Domain Definition 1

## Definition

*AttrDomainDef* : *DomainHead*
{ *DomainInterval* }.
*DomainHead* : **DOMAIN**
**message:**"*domainMessage*".
*DomainInterval* : **INTERVAL**
**lowerbound:***lowerBoundValue*
**upperbound:***upperBoundValue*.

- Example

```
FACET <AttributeNumericDomain>
VALUE "LAYOUT decimals:2"
```

```
FACET <AttributeNumericDomain>
VALUE "DOMAIN
        message:\"Enter a value between 0.25 (quarter of an hour) and 20.\"
        INTERVAL
        lowerbound:0.25
        upperbound:20.0"
```

# Attribute Domain Definition 2

## Example (cont.)

```
FACET <AttributeNumericDomain>
VALUE "DOMAIN
        message:"The valid Value Range of the Attribute lies between 0 and 100
                 and between 1000 and 1100."
        INTERVAL
        lowerbound:0
        upperbound:100
        INTERVAL
        lowerbound:1000
        upperbound:1100 "
```

# Regular Expression Definition

## Definition

*RegExpDef*: *RegExpHead*.
*RegExpHead*: **REGEXP**
**message:**"*regExpMessage*"
**expression:**"*regularExpression*".

- Example

```
FACET <AttributeRegularExpression>
VALUE "REGEXP
        message:\"Enter the time in the format MM.YYYY (Domain 01.1950 to 12.2050).\"
        expression:\"^(0[1-9]|1[0-2])\\.(19[5-9][0-9]|20[0-5][0-9])$\""
```

```
FACET <AttributeRegularExpression>
VALUE "REGEXP
        message:"That is not a valid e-mail address!"
        expression:".*@.*"
        "
```

```
FACET <AttributeRegularExpression>
VALUE "REGEXP
        message:"Input data in format 'DD.MM.YYYY'."
        expression:"^()$|^((((0[1-9]|1[0-9]|2[0-9]).(0[1-9]|10|11|12))|(30.(01|0[3-
9]|10|11|12))|(31.("
                "01|03|05|07|08|10|12))).[0-2][0-9][0-9][0-9]${10})"
        "
```

# InterRef Domain Definition

## Definition

*InterRfDomainDef*: *[ DomainHead ]*
{ *ModRefDomain* } |
{ *InstRefDomain* }.
*DomainHead* : **REFDOMAIN**
[ **max:**totalMaxValue ].
*ModRefDomain* : **MODREF**
**mt:**"*modelTypeName*"
[ **max:**maxValue ].
*InstRefDomain* : **OBJREF**
**mt:**"*modelTypeName*"
**c:**"*className*"
[ **max:**maxValue ].

## Example

```
FACET <AttributeInterRefDomain>
VALUE "REFDOMAIN
        OBJREF
        mt:"My ModelType"
        c:"MyModelClass"
        max:1 "
```

```
FACET <AttributeInterRefDomain>
VALUE "REFDOMAIN max:1
        MODREF mt:\"Knowledge
Management Process Model\" "
```

```
FACET <AttributeInterRefDomain>
VALUE "REFDOMAIN max: 100
        OBJREF
        mt:"MyFirstModelType\"
        c:"MyClassInMyFirstModelType"
        max: 50
        OBJREF
        mt:"MySecondModelType"
        c:"MyClassInMySecondModelType"
        max: 50 "
```

# Enumeration Domain Definition

## Definition

**ITEM** *itemText* [ **param:***varName*:*defaultText* ] [ **fdlg-filterX**:*filterExt* **fdlg-typeX**:*filterName* ]
*AdoScript* .

- Example

```
FACET <EnumerationDomain>
VALUE "value-1@value-2@value-3@value-n"
```

# MultiLineString Definition

## Definition

The attribute-facet 'MultiLineString' (only for attributes of type STRING) specifies, whether the text field for the string has a single line (VALUE 0) or several lines (VALUE 1).

The text field allows entering 3700 symbols maximum. In the attribute 'name' entering 255 symbols maximum is possible. A text field with more lines owns scroll-bars in the notebook and can be enlarged to screen size 640x480 by an enlarging button.

## Example

```
FACET <MultiLineString>
VALUE 0
```

```
FACET <MultiLineString>
VALUE 1
```

# Attribute Help Text Definition

## Definition

The **attribute-facet 'AttributeHelpText'** defines an i-Button (on the right top of the text field), where the info-text (defined in 'VALUE') is deposited.

## Example

```
FACET <AttributeHelpText>
VALUE "You can change the language from English to German and/or vice versa."
```

# Example for Meta-Data

**Attributes can be defined and provided with a descriptive default value.**

**They should not be provided in the „Notebook“ to prevent the user from changing these, making them only accessible through processing.**

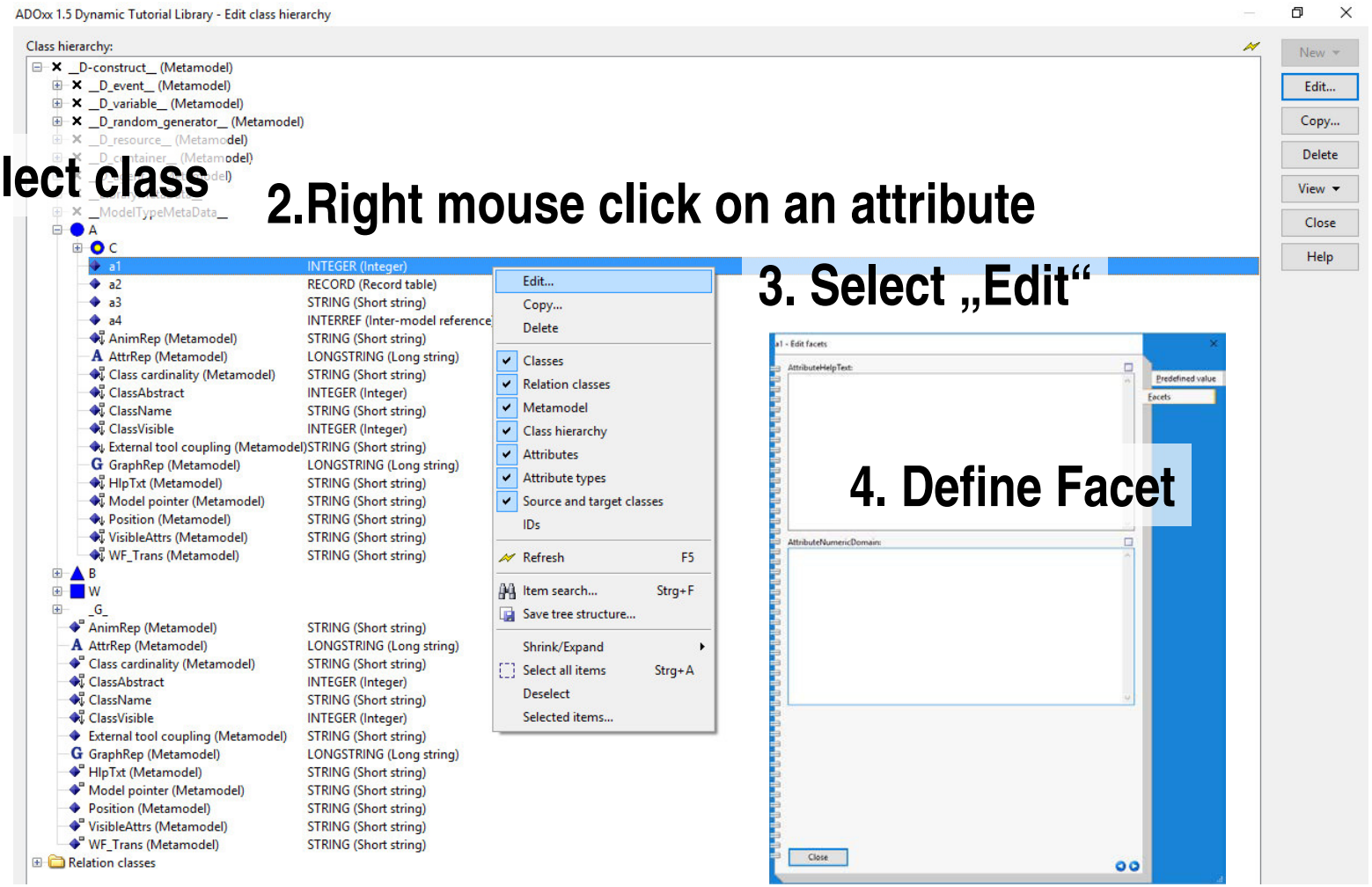
```
ATTRIBUTE <Application>
TYPE STRING
VALUE "All objects of this aggregation belong together and must be considered
as a group by all functions. "

        FACET <MultiLineString>
        VALUE 1

        FACET <AttributeHelpText>
        VALUE "Enter a description for documentation purposes."

        FACET <AttributeRegularExpression>
        VALUE ""
```

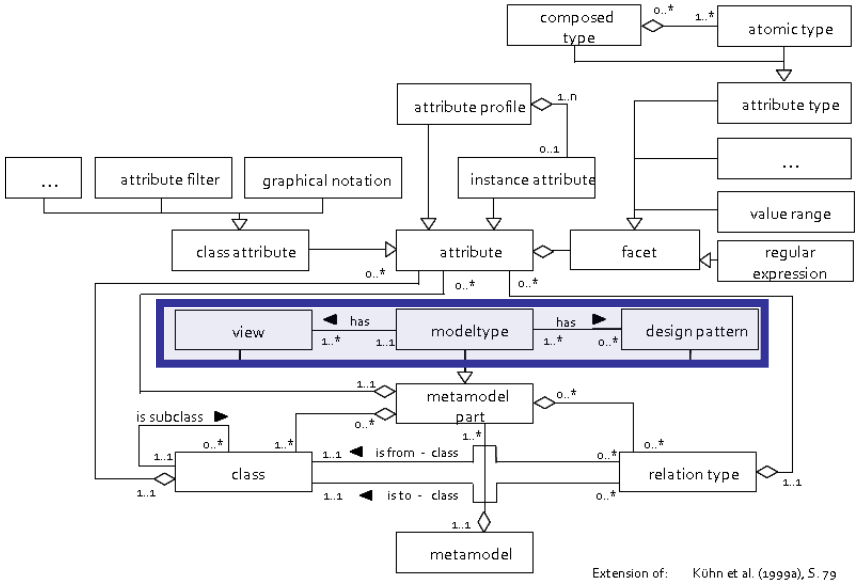
142

# 3. ATTRIBUTE FACETS

# HANDS-ON

# Facet Notebook in Attribute-Edit Mode






# 4. MODELTYPES

# Definition of Model Types

- Model types, model type-groups and views for model types:
  - A **model type** determines a **subset of all instantiable classes and relations**. Each model has a specific model type which can not be changed afterwards.
  - **Model type-groups** should be defined, if the application library consists of many different model types. This allows to group and structure the available model types.
  - A **modus** is a **further restriction of a model type**. It defines a subset of the assigned classes/relations and simplifies modeling by hiding not needed classes. The modus of a model can be changed any time unlike the model type.

# Definition of Model Types

**GENERAL order-of-classes: *OrderOfClasses***

Defines if the sequence of the classes in the modeling tool should be taken from the meta model (<*OrderOfClasses*> = „**default**“) or is specified for each model type explicitly („**custom**“).

**METHOD graphrep: *„attrName“***

Introduces a method diagram.

**GROUP *„GroupName“***

Defines a group of model types with the name <*GroupName*>.

**graphrep: *„attrName“***

Defines a graphical representation for a method diagram. <*attrName*> specifies an attribute which contains the representation using the ADOxx GraphRep language.

# Definition of Model Types Sample

```
GENERAL order-of-classes:custom
METHOD graphRep:"Method GraphRep"
{
  GROUP "Simulation"
  {
    MODELTYPE "My First Model Type"
    MODELTYPE "My Second Model Type"
  }
  GROUP "All modeltypes"
  {
    MODELTYPE "My First Model Type"
    MODELTYPE "My Second Model Type"
    MODELTYPE "My Third Model Type"
    MODELTYPE "My Forth Model Type"
  }
}
```

**Modelling Stack with four model types, grouped into two model type groups.**

# Additional Commands to Define Model types

**MODELTYPE „*modelTypeName*" from *MTSource***

This command defines a model type <*modelTypeName*> and inherits all classes and relations from the source <*MTSource*> (**all**, **none** or a different model type)

**plural: „*modelTypePluralName*"**

Defines the plural name of a model type.

**bitmap: „*fileName*"**

Defines a graphical symbol for the selection list (<*fileName*> = path and file name; backslashes must be masked with an additional backslash, i.e. "\\").

**attrrep: „*attrName*"**

Provides a Notebook (defined in the library as an attribute with the name <*attrName*>) with model attributes for a model type.

**INCL / EXCL**

Adds (except for **all**) /removes (except for **none**) classes and relations to the **MODELTYPE**.

**pos / not-simulateable**

Determines the position in list of model types / excludes the model type from simulation.

# Example: Model type

```
MODELTYPE "My First Model Type"
  from:none
  plural:"My First Model Types"
  pos:1
  not-simulateable
  bitmap:"db:\\MyFirstModelType.bmp"
  attrrep:"Notebook for My First Model Type"
INCL "My Class 1"
INCL "My Class 2"
INCL "My Class 3"
INCL "has relationship 1"
INCL "has relationship 2"
```

# Commands to define Views on Model Types

**MODE "*modeName*" from: "*modeSource*"**

This command defines a view modus with the name <*modeName*>. A list of classes/relations must be specified (either absolute or relative as described above) together with this command. MODE can be extended using several parameters.

**from: "*modeSource*"**

Inherits all the classes and relations from the source <*modeSource*> (**all**, **none** or a different mode). „**all**“ relates to the list from the model type (not the whole metamodel).

**no-modeling**

The defined mode is not applicable for modeling and will not be shown in the menu entry “Modi” of the modeling component.

**no-documentation**

The defined mode is not applicable for creating a documentation.

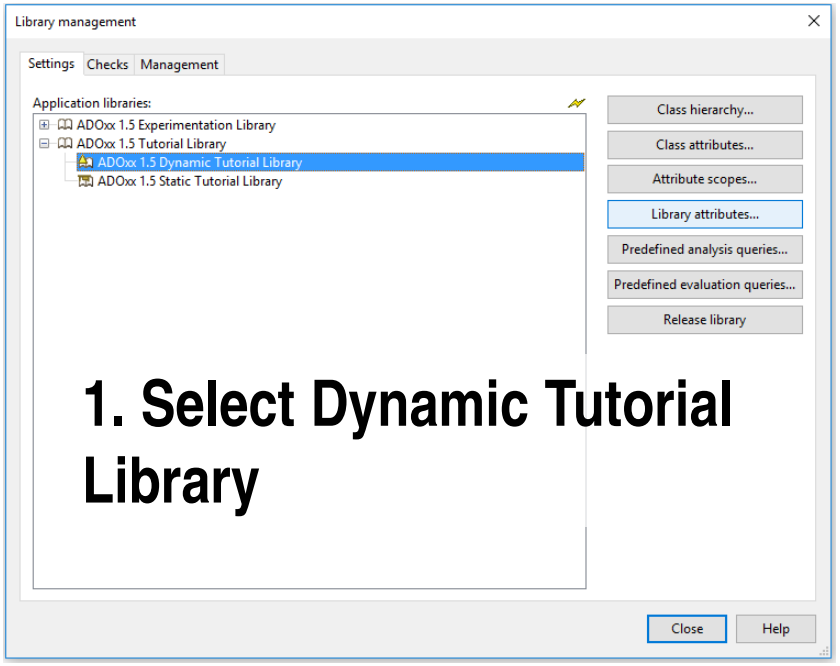
# Example: Model type View

```
MODELTYPE "My First Model Type" from:none plural: "My First
Model Types"
  pos:0 not-simulateable bitmap:"db:\\MyFirstModelType.bmp"
  attrrep: "Notebook of My First ModelType"
  INCL "My Class 1"
  INCL "My Class 2"
  INCL "My Class 3"
  INCL "has relationship 1"
  INCL "has relationship 2"
MODE "Standard" from:all
  EXCL "My Class 3"
  EXCL "has relationship 2"
MODE "Documentation" from:Standard no-modeling
  INCL "My Class 3"
  INCL "has relationship 2"
```

# 4. MODEL TYPES

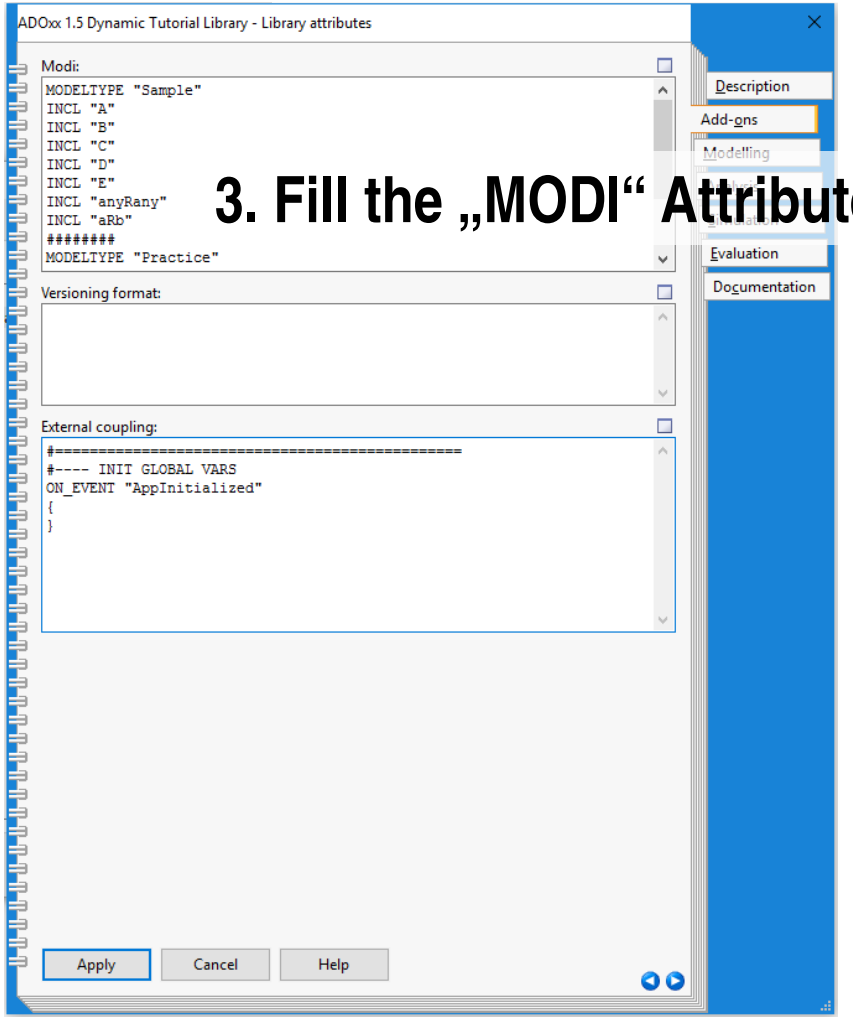
# HANDS-ON

# Definition of the ADOxx MODI Attribute




1. Select Dynamic Tutorial Library

2. Select the Tab Add-Ons

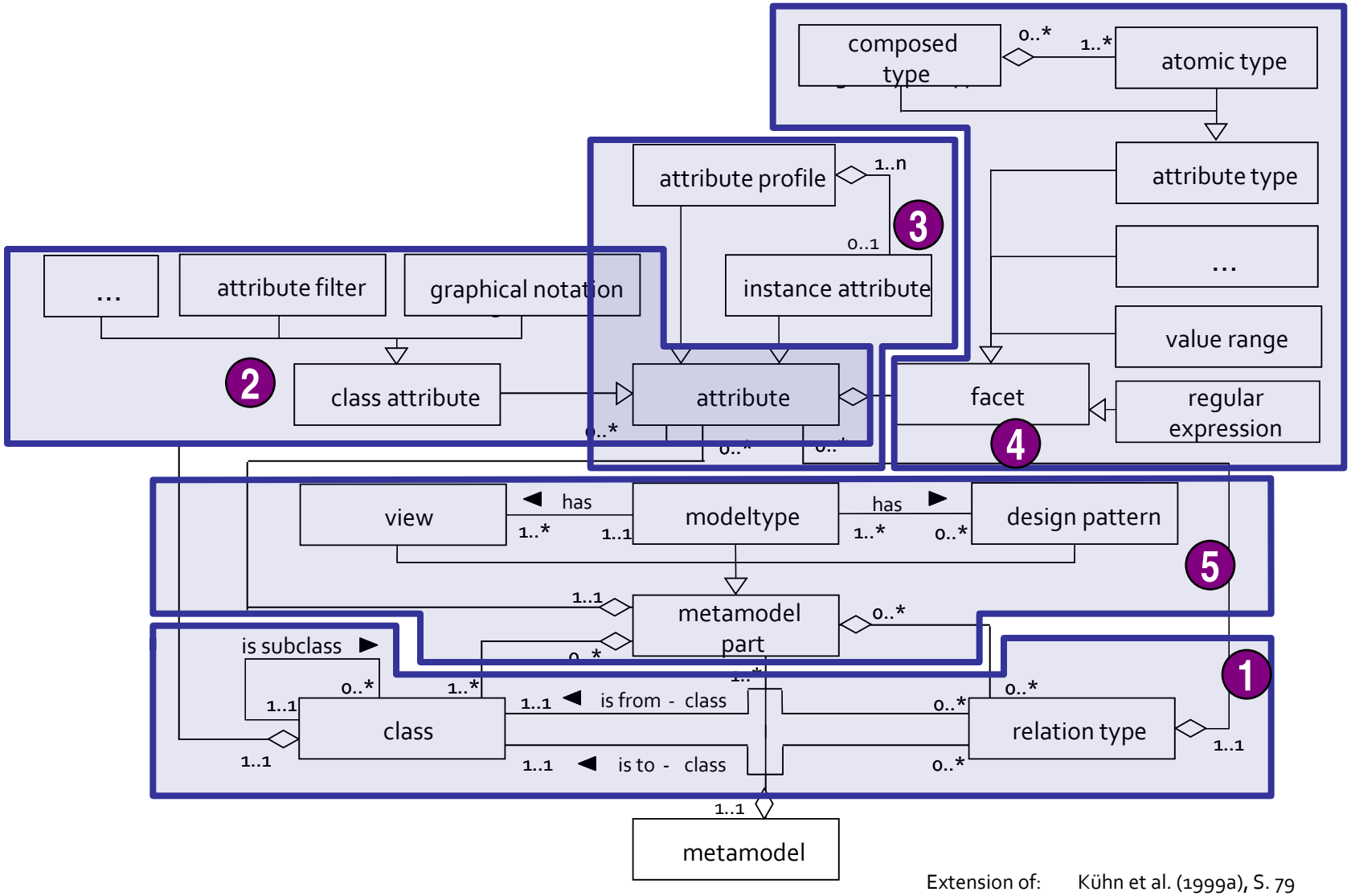



3. Fill the „MODI“ Attribute

# MODELLING LANGUAGE IMPLEMENTATION ON ADO*xx*

# SUMMARY

# Meta Model of Meta Modelling Language




Extension of: Kühn et al. (1999a), S. 79