EasternGraphics
visualize your business

Application Notes (2023-09-18)

AN-2006-01: Control Data Tables

Contents

1. Introduction

The behaviour of standard implementations of OFML base types, used by the OFML runtime system, may be controlled by means of special data tables, the so called **control data tables**. This approach avoids the need for deriving special sub classes in order to achieve a specific behaviour.

Each OFML base type which introduces a control data table, implements a method *openDataTable()* which declares the name of the table. Derived subclasses may declare and use additional information (data elements) for/from this table.

The table may exist as a single CSV table or may be included in the EBase database `ofml.ebase`, which takes precedence over the single CSV table.

Each data table has the same structure:

- Field 1 specifies the **type of information** provided in the record.
- Field 2, the **argument field**, is optional and may be used to differentiate the usage and/or the scope of the information.
- Field 3, the **value field**, contains the actual information.

Here is the table description for the EBase configuration file:

```
table <name> <name>.csv mscsv
fields 3
field  1 type  vstring delim ; trim hidx link
field  2 args  vstring delim ; trim
field  3 value vstring delim ; trim
```

The tables or the ofml.ebase database have to be located - if not otherwise specified - in the library path of current OFML program (series). ISO-8859-1 (Latin-1) is used as the character set.

The possible/required data types and the according format of argument and value fields are declared in the specification of the methods of the OFML base types using a specific information type. In the subsequent sections, however, the description of the various information types takes place on the basis of their affiliation to a concrete table.

This version of application note reflects the state of OFML base packages OI version 1.43.0 and XOI version 1.60.0 as well as of EAI 1.32.0 (native OCD implementation). The corresponding required application versions are pCon.planner >= 8.9, pCon.basket >= 1.13.10 and EAIWS >= 4.14 (online apps)[1].

[1] Releases in October 2023

2. Table *proginfo*

This table is used to control the behaviour of the so called *ProgInfo object* registered for a given OFML series (program). This object realizes common behaviour concerning the series as a whole. The concrete type of the *ProgInfo* object is specified in the DSR configuration file for the given OFML series under key `proginfo`.

The table is introduced by OFML base type *OiProgInfo* and is used also by derived class *xOiProgInfo*.

If the table is not found in the regular library directory of current OFML series it will be looked up in the directory specified in DSR configuration file for current series under key `proginfodb_path`.

2.1. General options

@AutoDecorationPath

- Value field specifies the path of the directory containing specific templates to be used during automatic decoration for articles belonging to the program (series).
- For more information about automatic decoration see separate application note.
- Used by method *xOiProgInfo::getAutoDecorationPath()*.

@CheckPrice4Consistency

- If value field contains `1`, articles belonging to this program will be marked as inconsistent, if method *getArticlePrice()* of interface *Article* returns `NULL` for a given article.
- Note: inconsistent articles cannot be ordered!
- Used by method *xOiProgInfo::checkObjConsistency()*.

@DynamicAttPt

- Used by method *OiProgInfo::getDynamicAttPts()*, which delivers a list of `[key, dir]` pairs for all attach points occurring in the program (series) and that cannot be determined by calling *getAttPtsOrder()* (OFML Interface *AttachPts*) on initial configurations of the articles of the program due to dynamic determination in *getAttPtsOrder()*, e.g. based on height raster. (This information may be used by tools generating feedback data for special 3D insertion feedback modes of an OFML application.)
- Value field contains one or more `[key, dir]` vectors to be added to the result list of *getDynamicAttPts()* (multiple pairs have to be separated by commas).

@MatPackage

- Value field specifies the name of the OFML package containing the material definition files to be used for the articles belonging to this program.
- Used by methods *OiProgInfo::getMatPackage()* and *OiProgInfo::getMatName()*.

@PricesOnRequest

- If value field contains `1`, articles belonging to this program will be tagged with message “price on request”, if method *getArticlePrice()* of interface *Article* for a given article returns `NULL` or a sales price

with value `0.0`[2].

- Note: this entry has no effect, if there is an entry of type `@CheckPrice4Consistency` with value `1`!
- Used by method *xOiProgInfo::getPricesOnRequest().*

2.2. Property related options

`@EPDFPropValPrefix`

- Value field contains the prefix that has to be put in front of the names of EPDF resp. OCD property values starting with a numeric character when converting them into OFML Symbols for choice lists of according OFML properties.
- Note: OFML Symbols may not start with a numeric character.
- Default prefix is "S". Another prefix has to be used, e.g., if there are property values whose names themselves start with "S"!
- Used by method *xOiProgInfo::getEPDFPropValPrefix().*

`@ForceDynamicProperties`

- If there is an entry with this data type and if its value is `1` (true), method *forceDynamicProp()* of OFML interface *Property* will be applied for all planning elements (articles) belonging to this program (series).
- Note: if method *forceDynamicProp()* returns True for a given property key, the value of the property will be stored in the table for dynamic properties, thus deliberately ignoring possibly existing get- resp. set-methods in order to avoid conflicts with properties dynamically generated by the global product manager for configurable properties specified in OCD data.
- Example: if an OFML type, used to represent articles, implements methods *setFoo()* and *getFoo()*, there is a certain chance, that the OCD data creator adds a property `Foo` to an article represented by the OFML type. The global product manager then generates an according OFML property with key `@Foo`. If *forceDynamicProp()* would not return `1` for key `@Foo`, the value of the property would be stored and retrieved using the methods *setFoo()* resp. *getFoo()*, what might not be the purpose of these methods. Therefore, it is recommended to implement *forceDynamicProp()* and return `1` for all set-/get-method pairs, which could be misused by an according OCD property. However, for reasons of backward compatibility, this handling has to be enabled by an entry of this type with value `1`.
- Used during standard implementations of methods *getPropValue()* and *setPropValue()* of OFML interface *Property.*

`@Property`

- Used during initialization of an instance of *OiProgInfo.*
- Defines a fix property (in contrast to common properties set up dynamically, see section 2.4).
- The value field has to be of following format:
  `[prop_key, prop_def, position, init_value, state],`

[2]In the current implementation the „tag“ will be appended to the article long text.

where the first three vector elements specify the parameters to be passed to method *setupProperty()* of OFML interface *Property*.

The elements `init_value` and `state` are optional. If `init_value` is given it will be assigned as initial value to the property after *setupProperty()*. If `state` is given it will be passed to *setPropState()*.

The following property types may be specified in `prop_def`: “s”, “b”, “i”, “u chi”, “f” and “u chf”.

@UseUnspecPropVal4OdbInfo

- Specifies, whether properties with value `@UNSPECIFIED` have to be represented in the ODB info hash table (`1`) or not (`0`). Default: `1`.
- Used by method *OiProgInfo::useUnspecPropVal4OdbInfo()* called from hook method *OiOdbPlElement::useUnspecPropVal4OdbInfo()* in *OiOdbPlElement::getBasicOdbInfo()*.

@UseVoidPropVal4OdbInfo

- Specifies, whether properties with value `@VOID` have to be represented in the ODB info hash table (`1`) or not (`0`). Default: `1`.
- Used by method *OiProgInfo::useVoidPropVal4OdbInfo()* called from hook method *OiOdbPlElement::useVoidPropVal4OdbInfo()* in *OiOdbPlElement::getBasicOdbInfo()*.

@NeedPropGroupDescriptions

- Specifies, whether (possible) different property groups of current OFML instance have to be displayed in the property editor (`1`) or not (`0`). Default: `1`.
- Used by hook method *OiProgInfo::needPropGroupDescriptions()* called from standard implementation of method *getPropGroupDescriptions()* of OFML interface *Property*.
- Property groups will be displayed, if there is no control data table `proginfo`, or if the table does not contain a matching entry of this type, or if the first matching with a valid value in the value field there contains the value `1`.
- An entry matches if the argument field is empty or if all conditions specified in the argument field are satisfied. A condition has to be given as a vector `[condition_type,condition_value]`. Multiple conditions have to be separated by commas.
- Currently, the following **condition types** are supported:

  `@Category`
  : is true, if the instance belongs to one of the categories specified in the value (Vector of Symbols)

  `@CategoryNOT`
  : is true, if the instance *does not* belong to one of the categories specified in the value (Vector of Symbols)

`@Article`

is true, if the instance represents an article[3] and if its base article number starts with one of the article number root strings specified in the value (Vector of Strings)

`@ArticleNOT`

is true, if the instance represents an article[4] and if its base article number *does not* start with one of the article number root strings specified in the value (Vector of Strings)

`@Type`

is true, if the type of the instance is one of the types specified in the value (Vector of Strings) or if it is a sub type of them.

Types have to be fully qualified (i.e., including the OFML package name).

`@TypeNOT`

is true, if the type of the instance is *not* one of the types specified in the value (Vector of Strings) and if it is *not* a sub type of them.

Types have to be fully qualified.

2.3. 2D layer name

The information types described in this subsection are used by method *OiProgInfo::get2DLayer()*, which delivers the name of 2D layer to be used if 2D data for a given article does not define an explicit layer. The method creates a layer name that is composed of the following (optional) components:

1. prefix
2. OFML manufacturer ID and/or commercial manufacturer ID (normally, only one of both)
3. separator
4. OFML series ID and/or commercial series ID (normally, only one of both)
5. suffix

Which of these components will be included into the layer name is controlled by the appearance of following information types:

`@2DLayerPrefix`

- value field contains the prefix for layer name

`@2DLayerSuffix`

- value field contains the suffix for layer name

`@2DLayerSeparator`

- value field contains the String to be used to separate manufacturer and series IDs

[3] The instance represents an article, if the class of the instance implements OFML interface *Article* and the base article number returned by method *getArticleSpec()* is not an empty String.
[4] see above

@2DLayerUseManufacturerID

- value field contains `1`, if commercial manufacturer ID has to be included into layer name

@2DLayerUseSeriesID

- value field contains `1`, if commercial series ID has to be included into layer name

@2DLayerUseManufacturer

- value field contains `1`, if OFML manufacturer ID has to be included into layer name

@2DLayerUseSeries

- value field contains `1`, if OFML series ID has to be included into layer name

2.4. Property value pictures

@PropInfoPicPrefix

- The value field specifies a prefix for picture file names for values other than `@VOID` (or for all values including `@VOID` if table does not contain entries of type *@PropInfoPic4Void*, see below).
- The file name then will be constructed from the prefix and the string representation of the property value (without leading character `@`). If the property is associated with a property in product database, the value ID from product database will be used instead.
- Used by method *OiProgInfo::getPropInfo4Obj()*.

@PropInfoPic4Void

- The value field specifies the picture file name for value `@VOID`.
- Used by method *OiProgInfo::getPropInfo4Obj()*.

There may be multiple entries of types *@PropInfoPic4Void* and *@PropInfoPicPrefix*. Whether a given entry will be used may be controlled by the conditions given in argument field (2. field). If this field is empty, the entry will be used without checks. If it contains conditions, they all have to be evaluated as True in order to make entry effective. The first matching entry will be used.

Each condition is given as vector `[condition_type, condition_value]`. Multiple conditions have to be separated by commas. Currently, the following condition types are supported:

`@PropKey`

is True, if the key of affected property starts with one of the strings specified in the condition value.

Condition value may be a single string or a vector of strings.

`@PropKeyNOT`

is True, if the key of affected property does NOT start with one of the strings specified in the condition value.

Condition value may be a single string or a vector of strings.

`@PropName`

is True, if affected property is associated with a property in product database and the name of that property starts with one of the strings specified in the condition value.

Condition value may be a single string or a vector of strings.

`@PropNameNOT`

is True, if the affected property is associated with a property in product database and the name of that property does NOT start with one of the strings specified in the condition value.

Condition value may be a single string or a vector of strings.

`@PropValue`

is True, if string representation of property value is one of the strings specified in the condition value.

Condition value may be a single string or a vector of strings.

Examples

1.

Use prefix for all properties whose key string starts with "Mat", but not for properties `@MatGroup` and `@MatType`:

```
@PropInfoPicPrefix;[@PropKey,"Mat"],[@PropKeyNOT,["MatGroup","MatType"]];::foo::bar::
```

2.

Within a furniture program property value "LG" has a different meaning in two different properties `OPTION1` and `OPTION2`. Therefore, for `OPTION1` the normal prefix has to be used, for `OPTION2` a special prefix:

```
@PropInfoPicPrefix;[@PropName,"OPTION2"];::foo::bar::OPTION2
@PropInfoPicPrefix;;::foo::bar::
```

Note: Because the first matching entry will be used, the order of both entries has to be as stated above in order to achieve the desired result.

2.5. Common properties

See also application note on global variant configuration.

`@CommonProps4ProgInfo`

- Common properties for the ProgInfo object itself are generated only if the table contains an entry with this type and if its value is `1`.
- Used by method *xOiProgInfo::getProperties().*

`@NonPlElements4CommonProps`

- If there is an entry with this info type and its value is `1`, objects belonging to the program will be considered for generation of common properties even if they are not located on top level of planning hierarchy.

- Background: normally, only elements on top planning level will be considered for generation of common properties for a temporary marked group of objects. However, some applications or application modes might allow to mark objects not on top planning level. In this case it might be useful, to consider non top planning elements, too.
- Used by method *xOiProgInfo::nonPlElements4CommonProps()*.

`@IgnorePClass4CommonProps`

- If there is an entry with this info type and its value is `1`, property classes of properties of objects belonging to the program will be ignored during collection of common properties.
- Note: normally, properties with same name but belonging to different property classes will get different corresponding properties in the set of common properties, because – conceptually – they are considered as different properties. However, in some data setups, the different property classes may not have a special meaning. In these cases it might be more appropriate to use a single common property for these properties.
- Used by method *xOiProgInfo::ignorePClass4CommonProps()*.

`@CommonProperty`

- Value field contains a comma separated list of property names. All properties with the given names are denoted as common ones.
- Special property name "ALL_PROPERTIES" marks all properties as common ones.
- Used by method *xOiProgInfo::isCommonProperty2()*.

`@CommonPropPrefix`

- Value field contains a comma separated list of property name prefixes. All properties whose names start with the given prefixes are denoted as common ones.
- Used by method *xOiProgInfo::isCommonProperty2()*.

There may be multiple entries for both types `@CommonProperty` and `@CommonPropPrefix`.

If a property is associated with property in product data, in the value fields have to be specified the property names resp. prefixes as used in the product database rather than property keys generated by the global Product Data Manager.

Note:
If using an own class derived from *xOiProgInfo*, make sure there is no implementation of *isCommonProperty()*: this implementation would not have any effect because method *isCommonProperty2()* (implemented in *xOiProgInfo*) takes precedence.

2.6. Order generation (article lists)

@OrderInfo

- Entries of this type may be used in order to control the appearance of generated order listings. (However, the setting affects only the articles belonging to this OFML series.)
- Used by method *xOiProgInfo::getOrderInfo()*, called during generation of order listings.
- The argument field is used to determine, for which values of method's parameters the value field of the entry has to be used.

  The format of the argument field is a Vector containing `[parameter,value]` pairs, where the possible parameters are `@OrderMode`, `@OrderDepth`, `@Region` and `@InfoType`:

  - Parameter `@OrderMode` specifies the mode of order generation used by the application (typically, this is `@Standard`) whereas parameter `@OrderDepth` specifies the maximum depth of generated order structure (where `0` denotes no restriction for order structure depth).

    These parameters are optional. If no value for `@OrderMode` or `@OrderDepth` is given, the entry matches each value for the corresponding order generation parameter. This is the normal usage for entries of these types (because special parameters are used only in special applications).

  - (Optional) parameter `@Region` specifies the distribution region. It will be compared with the value which is given under key `distribution_region` in the registry file of given OFML package. If no `@Region` parameter is given, the entry matches each distribution region.

  - Parameter `@InfoType` specifies a concrete option influencing the appearance of the order. The supported options and their corresponding possible values are described below. The specification for `@InfoType` is required. If there are multiple table entries for a given `@InfoType` with different values for `@OrderMode`, `@OrderDepth` and `@Region`, the table entry will be used, which has the “best match” for the corresponding parameters of ongoing order listing generation resp. for current distribution region of registered OFML package. If there are multiple equally matching table entries, the first in the table will be used.

- The supported options for parameter `@InfoType` and their corresponding possible values are:

  - `@NeedSumOrder`: the integer value (`0` – *no*, `1`- *yes*) specifies whether identical articles (equal configuration) have to be summarized, using a single (shared) position in the order listing.

    Default behaviour (if no value is given for this option) is `0`, i.e. no summarizing will be performed.

    Note: this option has an affect only if the application generally allows summarizing of identical articles during order listing generation.

  - `@DeepSumOrder`: the integer value (`0` – *no*, `1`- *yes*) specifies whether sub-articles should be taken into account when summarizing identical articles (see option `@NeedSumOrder` above): In the case of 0 (standard), two equal articles are not considered to be identical if at least one of them has sub-articles.

    Note: If specified, parameter `@OrderMode` must be `@Standard` and parameter `@OrderDepth` must be `0`.

- `@ArtSpecMode`: specifies which article specification (number) has to be printed in the order listing. The value is a Vector `[mode, max_length]`, where `mode` may be:

  `0` – base article number (default)
  `1` – final article number

  The element `max_length` specifies the maximum number of characters to be used for printing the article number, where `NULL` (default) denotes no restriction.

- `@ArtTextMode`: specifies which form of the article text has to be used in the order listing. The possible integer values are:

  `0` – short text
  `1` – long text (default)
  `2` – both

- `@NeedsAddPrices`: the integer value (`0`, `1`) specifies whether a detailed pricing information (e.g. including extra charges) has to be printed (`1`, default) or whether only the total price has to be printed for an order listing position (`0`).
  Whether the individual price components (with value of `1`) are actually displayed depends on the respective application!

- `@MaxOrderDepth`: the integer value specifies the allowed maximum depth of generated order structure, i.e. this option may be used to overrule the general maximum depth used for ongoing order listing generation.
  Value `0` (default) denotes no restriction for order structure depth.

  Examples:

  ```
  @OrderInfo;[[@InfoType,@ArtSpecMode]];[1,NULL]
  ```

  specifies, that final article number has to be printed (instead of base article number) without length restriction, regardless of order mode and depth of ongoing order listing generation.

  ```
  @OrderInfo;[[@InfoType,@NeedsAddPrices],[@Region,"ANY"]];1
  @OrderInfo;[[@InfoType,@NeedsAddPrices]];0
  ```

  Specifies, that separate price components have to be listed in distribution region `ANY`, but not in all other regions. Note: the order of both table entries is significant, because the second entry also matches region `ANY` (and would be used in case of reversed order of entries).

  ```
  @OrderInfo;[[@InfoType,@NeedsAddPrices],[@Region,"ANY"]];0
  ```

  Specifies, that separate price components must not be listed in distribution region `ANY`. In all other regions they will be listed (default behavior, see above).

@OrderRemoveType, @OrderRemoveArticle, @OrderRemoveCategory, @OrderSumArticlesByType, @OrderSumArticlesByArticle, @OrderSumArticlesByCategory

- Entries of these types specify which articles have to be removed from order structure resp. for which articles summarizing of equal configurations has to be performed.
- Used by method *xOiProgInfo::prepareOrder()*.

- Note: the information types regarding summarizing of identical articles have an effect only if the application generally allows summarizing of identical articles during order listing generation.
- Further note: if option `@NeedSumOrder` in entry of information type *`@OrderInfo`* (see above) has value `1`, the order generation algorithm after the calls of *xOiProgInfo::prepareOrder()* also will summarize identical articles in a general way not applying any restrictions. Therefore, if using the information types *`@OrderSumArticles*`*, you probably should make sure, that option `@NeedSumOrder` in entry of type *`@OrderInfo`* has value `0`.
- Articles may be specified by the OFML types used to represent them, by base article specification or by category.
- An article specification may be incomplete in which case all article whose specification start with given string will be affected.
- The value field of each of these information types may contain a comma-separated list of full type names, base article specification roots or category symbols, respectively.
- There may be given more than entry for each type.
- Whether a given entry will be used may be controlled by the conditions given in argument field (2. field). If argument field is empty, the entry will be used without checks. If it contains conditions, they all have to be evaluated as True in order to make entry effective.
- Each condition is given as vector `[condition_type,condition_value]`. Multiple conditions have to be separated by commas.
- Currently, the following condition types are supported:

  `@OrderMode`
  : is True if given value is equal to mode of ongoing order listing generation (see parameter `@OrderMode` of information type *`@OrderInfo`* above)

  `@OrderDepth`
  : is True if given value is equal to allowed maximum hierarchy depth of ongoing order listing generation (see parameter `@OrderDepth` of information type *`@OrderInfo`* above)

  `@LevelGT`
  : is True if effected order item belongs to an order hierarchy level greater than the given value (highest level is 1).

`@OrderPackFolderLabel`

- Specifies the label for the order folder into which generated pack article items have to be inserted. For more detailed information see Application Note on generation of pack articles.
- The pack article generation process scans the current planning hierarchy for items referencing planning elements of this program, which have to be grouped to form so called *pack articles.*
- The pack article generation process itself is controlled by entries of type *`@OrderPackArticle`* (see below), but if there is no entry of type *`@OrderPackFolderLabel`* no pack article items will be generated at all (for this manufacturer).
- All OFML programs of a certain manufacturer should use the same label, because there will be created only one pack folder item for each manufacturer.

- The value field may contain a text resource ID.
- Used by method xOiProgInfo::*prepareOrder()*.

`@OrderPackArticle`

- Entries of this type control the pack article generation process (see also entry type `@OrderPackFolderLabel` above).
- The value field contains a Vector specifying the base article number of each pack article corresponding to a respective count (pack size) of planning elements forming the pack article. Thereby, first article number specifies the pack article containing one planning element, the second article number specifies the pack article containing two planning elements and so on.

  If there is no pack article for a given count of planning elements the according article number in the Vector is an empty String whereby the last article number in the Vector must not be empty.

  Thus, pack generation schemes may be specified involving non serial pack sizes, e.g. 3 and 7.
- The argument field specifies the OFML class, article parameters and/or ODB type for planning elements subject to pack generation. There are given in the common form for conditions as described above for entry types `@OrderRemove*`. The according condition types and value types are:

  `@Class`
  : is True if name of class of current planning element returned method *getClass()* is equal to class name (String) given as value.

  `@ArticleParams`
  : is True, if article parameter code returned by method *getArticleParams()* of current planning element (article) is equal to article parameter code (String) given as value.

  `@ODBType`
  : is True, if ODB type (if any) of current planning element is equal to ODB type name (String) given as value.

  The class specification is required, the other specifications are optional.

  Additionally the same conditions as for removing items may be specified (see `@OrderRemove*` types above), if necessary.
- Used by method *xOiProgInfo::prepareOrder()*.

3. Table *plelement*

This table is used to control the behaviour of instances representing articles. It is introduced by OFML base type *OiPlElement* and is used also by derived classes *OiOdbPlElement*, *xOiPlElement* and *xOiBTGPlElement3*. Some entry types are used only by derived classes. Thus, not all of the entry types described below may be useful in a given OFML program (series), depending on the OFML types mapped to the articles of the series (see OAM).

@Article4PropTitle

- Specifies usage of article specification for property editor title.
- Used by method *OiPlElement::getPropTitle().*
- Value field has to be of following format: `[spec_type,max_length]`, where `spec_type` may be one of

  `@Base` for using the base article number, or

  `@Final` for using the final article number

  and `max_length` is an Integer, specifying the maximum length of article number allowed (`0` specifying no restriction).

  Default: `[@Base,0]`

@ArtText4PropTitle

- Specifies usage of article text for property editor title.
- Used by method *OiPlElement::getPropTitle().*
- Value field can contain the following value:

  `@s` for using the short description

  `@l` for using the long article text.

  Default: `@s`

The property editor title string will be constructed from determined article specification (*@Article4PropTitle*) and article text (*@Article4PropTitle*) separated by a space character.

@NoAddAsSibling

- Standard implementation of base type *OiOdbPlElement* allows to add a new planning element next to a selected child object of an *OiOdbPlElement*, e.g. next to an accessory placed on top of a table. However, in some planning situations this behaviour may not be appropriate. Then, entries of type *@NoAddAsSibling* may be used to specify the situations when adding an element as a sibling (next) to a selected child object of an *OiOdbPlElement* is **not** allowed.
- A planning situation is described by a set of conditions in the argument field. (The value in field 3 always has to be integer value `1`.)
  A condition has to be given as a vector `[condition_type, condition_value]` and the conditions have to be separated by commas. An entry matches if all conditions specified in the argument field are satisfied. If no conditions are specified (empty argument field) the entry matches in any case.

- The following *terms* are used in the subsequent description of possible condition types:
  - *element*: the element to be added to the planning
  - *implicit instance*: the object (instance of type *OiOdbPlElement*) which becomes the father object of new element
  - *reference object*: the selected child object of implicit instance (next to which the new element will be added)
- Currently, the following **condition types** are supported:

  `@ElType`
  : is True, if the type of new element is one of the types specified as condition value (Vector of Strings) or if it is a sub type of them.

    Types have to be fully qualified (i.e. including the according OFML package specifier).

  `@SelfType`
  : is True, if the type of implicit instance is one of the types specified as condition value (Vector of Strings) or derived from them.

    Types have to be fully qualified.

  `@SelfCategory`
  : is True, if implicit instance belongs to one of the categories specified as condition value (Vector of Symbols).

  `@SelfArticle`
  : is True, if the class of implicit instance implements interface *Article* and if its base article number starts with one of the strings specified as condition value (Vector of Strings).

  `@RefType`
  : is True, if the type of reference object is one of the types specified as condition value (Vector of Strings) or derived from them.

    Types have to be fully qualified.

  `@RefCategory`
  : is True, if the reference object belongs to one of the categories specified as condition value (Vector of Symbols).

  `@RefArticle`
  : is True, if class of reference object implements interface Article and if its base article number starts with one of the strings specified as condition value (Vector of Strings).

- Used by method *OiOdbPlElement::doNotAddAsSibling().*

`@ShowAddAttPts`:

- Controls, whether additional (non-standard) attach points will be shown or not.
- Used in *PICK* rule of *xOiPlElement.*
- Additional attach points will be shown, if there is no control data table `plelement`, or if it does not contain an entry of this information type or if first matching entry contains integer `1` in its value field.
- An entry of this type *matches*, if all conditions specified in the argument field are satisfied. A condition has to be given as a vector `[condition_type,condition_value]`. Conditions have to be separated by commas.

- Currently, the following **condition types** are supported:

  `@CountGT`
  : is True, if the count of currently defined additions attach points is greater than the given value (Integer)

  `@Category`
  : is True, if the class of the article instance belongs to one of the categories specified in value (Vector of Symbols)

  `@Article`
  : is True, if base article number of the article starts with one of the strings specified in value (Vector of Strings)

  An empty argument field is identical to condition `[@CountGT,0]`.

`@FixDimensions`

- Used during initialization of instances of *xOiBTGPlElement3.*
- Value field contains integer value for member variable *mXoiFixDimensions*, specifying whether fields "width", "depth", "height" and "ins_angle" in `btgmlist` table are defined as Floats. If value is `0` (false/no), they are defined as String fields and may contain symbolic names to be resolved by current value of according property.

`@UseVarKeys`

- Used during initialization of instances of *xOiBTGPlElement3.*
- Value field contains integer value for member variable *mXoiUseVarKeys*, specifying whether so called *varkeys* defined in additional `var` table are used as variant code in XCF and in field "variant" of `btgmlist` table (`1` for *yes*, `0` for *no*).

`@UseOAM`

- Used during initialization of instances of *xOiBTGPlElement3.*
- Value field contains integer value for member variable *mXoiUseOAM*, specifying whether OAM (instead of tables `bgtmlist` and `mat`) is used in order to specify additional ODB parameters (`1` for *yes*, `0` for *no*).

`@AppInteractorDefs`

- Used in standard implementation of method *getAppInteractorDefs().*
- For details of usage see Application Note AN-2013-001.

4. Table *anyarticle*

The entries in this table allow to control the creation and appearance of the dummy geometry of instances of base type *xOiAnyArticle*, used to represent articles without a given specific graphical representation.

@GeoCreationMode

specifies, whether a dummy geometry has to be created:

`1` - always create dummy geometry

`0` - create dummy geometry only if father is root object of hierarchy.

Default: `1`

@GeoType

specifies the OFML type of dummy geometry.

Default: `OiBlock`

@GeoArgs

specifies the parameters for *initialize()* function of OFML type given in entry of type *@GeoType*.

Default (if there is no *@GeoType* entry): `[1,1,1]`

@GeoMat

specifies the material symbol for geometry (to be resolved via *OiProgInfo::getMatName()*).

Default: white

@CreateText

specifies, whether a text has to be displayed beside actual geometry.

Values: `0` (not), `1` (yes).

Currently, the text contains the article short description of represented article.

Default: `0`.

@TextPos

specifies the (local) position for text primitive.

Default: right bottom back corner of bounding box of actual geometry.

@TextMat

specifies the fully qualified material name for text primitive.

Default: white

@TextAlignment

specifies the type of text alignment.

Values according to *Mtext::setAlignment()* (see OFML Specification).

Default: `@LEFT`

5. Table *epdfproductdb*

This table is used to control some aspects of the behaviour of the EPDF/OCD product database object registered for a given OFML series. The concrete type of the product database object is specified in DSR configuration file for given OFML series under key `productdb` and depends on the concrete data format used for the series, as well as – in case of OCD – on the implementation to be used (native vs. non-native). Some of the options only are usable with the native implementation (see subsection 5.3).

If the table is not found in standard location (library path) it will be looked up in the path specified in DSR configuration file for given OFML series under key `proginfodb_path`.

The name of the table is derived from the historically older proprietary product data format EPDF, which is/was the predecessor of OFML standard product data format OCD.

5.1. General options

`@SafePropertyNames`

Specifies, whether identifiers (names) of OCD characteristics in given OFML series comply with the OCD specification, where integer value `1` means *Yes/True*, and `0` means *No/False*. Default: `0`.

According to the OCD specification only alphanumeric characters including the underscore may be used for identifiers of OCD characteristics, where the first character must not be numeric. Furthermore, no reserved keywords may be used. (In OFML-based applications, to the keywords mentioned in the OCD specification, the reserved OFML keywords are added, see Appendix A.1.)

When the OCD data is converted from an external ERP system, however, this requirement cannot be met under certain circumstances. Therefore, OFML applications (in mode `0`) try to handle OCD characteristic identifiers that are not standard compliant in a robust manner. For that purpose, the following replacements are made during the conversion of an identifier of an OCD characteristic into the ID of the associated OFML property[5]:

- If the identifier of the OCD characteristic is an OFML keyword, the character '_' is preceded.
- If the identifier of the OCD characteristic starts with a numeric character, the letter 'S' (for "Symbol") is preceded.
- Non-alphanumeric characters are replaced by a sequence "_XX", where XX is the hexadecimal representation of the character.
- The character '_' is replaced by "__".

During the determination of the associated OCD characteristic identifier for a given OFML property ID then the corresponding inverse modifications will be made.

If the identifiers of OCD characteristics comply with the OCD specification, but value `1` is *not* specified for this option, the following problems may occur[6]:

- If the identifier of an OCD characteristic starts with letter 'S' followed by a numeric character, a corresponding OFML property ID is used, e.g. `@S2SNA`, but backwards the (non-existent) OCD characteristic identifier "2SNA" will be determined (with the consequence that the property cannot be changed by the user).

[5]Note: These modifications must be taken into account in ODB data, because there OFML property ID's have to be used.
[6]what should be avoided by explicitly specifying value `1`

- If the identifier of an OCD characteristic contains underscores, these are replaced in the associated OFML property ID by "__" (e.g. "_2SNA" => `@__2SNA`). This must be taken into account in ODB data, i.e., there the OCD characteristic identifier may *not* be used one-to-one.

@PreselectRestrictable

Specifies the behaviour with respect to automatic value selection for restrictable properties. Without automatic value selection the (initial) state (value) of a restrictable property is “not specified” (“???”).

`1`
: pre-select restrictable properties which have a value marked as default (and which is valid in current configuration)

`2`
: pre-select all restrictable properties (use first valid value, if no default value is specified or if it is not valid in initial configuration)

`3`
: no pre-selection

Default behaviour is `1`.

Note: A restrictable property keeps its value after changes of other properties as long as the value remains valid in the current configuration and is not overwritten by product relationships. If the value becomes invalid, the new value of the property will be determined according to the method specified with this option.

@ObligatoryPropCheck

Specifies, whether unspecified obligatory properties have to be reported as an inconsistency, where integer value `1` means *Yes/True*, and `0` means *No/False*. Default: `0`.

Note:
An inconsistent article cannot be ordered. Therefore, use value `1` only after careful consideration. An obligatory property may be unspecified, if all possible values currently are not valid due to preconditions. Normally, this has to be considered as a data error because an obligatory property should have at least one valid value in each possible configuration. Value `1` would prevent the user from ordering the article, and the manufacturer has to rely on an error report by side of the user in order to get informed about the error.

Value `1` might be useful only in some complex data constellations where it is hard to assure a valid value in all possible configurations. I.e., the data creator deliberately accepts such inconsistent intermediate configuration states. However, the user still might not know, that this an intermediate state, and how to get out of this state. Therefore, with OCD 4.0 the data creator should consider the application of user messages in order to inform the user about the situation.

@MultiplyPricingFactors

Specifies, whether multiple pricing factors assigned for a given variant condition within a single pricing condition have to be multiplied. There, integer value `1` means *Yes/True*, and `0` means *No/False*. In case of 0 only the last assigned pricing factor will be used. Default: `1`.

Note:
This option is relevant only for OCD versions 2.0, 2.1 and 3.0. In format versions 2.2 and 4.0 the prescribed behaviour corresponds to value `0`, because the assignment of multiple pricing factors for a given variant condition probably is not really intended by the data creator, and because, on the other hand, the desired/required end factor could be set with a single assignment as well.
(The default for OCD versions 2.0, 2.1 and 3.0 is `1` for historical reasons and due to backward compatibility.)

@InsignificantPropClasses

Specifies, whether OCD property classes are used to qualify OCD properties. There, integer value `1` means *No*, and `0` means *Yes*. Default: *Yes*.

Value `1` should be assigned, if OCD data will be exported from an ERP system, which does not employ the concept of property classes, and where the export routine generates the OCD property classes in a random way. If value `1` is not assigned in these cases, saved projects may not be updated after installation of newly exported OCD data, even if there were no significant changes in the original ERP system.

@GenerateLimits4NumProps

Specifies, whether min/max limits have to be generated for numeric properties, if no limit was set via interval values. There, integer value `1` means *Yes/True*, and `0` means *No/False*. Default: *False*.

The limits will be generated based on the count of digits specified for the affected property. For instance, if the specified count of digits for an integer property is 3, the corresponding limits are `-99` (min) and `999` (max). Thus, this option (value `1`) may be used in order to avoid the input of invalid values by the user.

@CompleteEAN

Specifies, whether end (final) article number contains all information in order to recreate the current configuration, where integer value `1` means *Yes/True*, and `0` means *No/False*. Default: `0`.

This option is relevant only for user-defined OCD coding schemes[7]. If value is `1`, the end (final) article number will be used to describe and store the current configuration of the article, otherwise an extended variant code will be used/generated. Use value `1` only, if you are sure, that the end (final) article numbers of all articles in the series contain a representation for all configurable properties.

5.2. Coding schemes

The following options are used to specify the coding scheme for end (final) article number (including the variant code). However, in case of OCD product data these options will not be used if a code scheme is specified for a given article in the according OCD table itself. (It is recommended to use the OCD coding scheme table instead of this control data table in order to specify a coding scheme.)

[7] Also for user-defined coding schemes, this option can be considered obsolete now, since starting with the releases of the pCon applications in March 2010, the manufacturer-independent, so-called OFML-VarCode is used to store and restore the configuration of an article. For projects that have been saved with older versions, or for projects where by special programming manufacturer-specific variant codes or final article numbers are stored and assigned, this option should however be further maintained.

Conceptually, the end (final) article number consists of the base article number, a separator string and the variant code describing the current configuration of the article.

@VarCodeType

Specifies the type of the variant code. The currently supported types are:

`@Complete`

The variant code contains property **and** value IDs for all currently defined properties. Each property is represented in the form `prop_class.property=prop_value`. The property representations are separated by semicolons.

This is the default type.
It corresponds to predefined OCD coding scheme `KeyValueList`.

`@ValuesOnly`

The variant code only contains the value IDs for all currently defined properties. The values may be separated by client defined separators (see *@VarCodeValueSep* below).

This type corresponds to predefined OCD coding scheme `ValueList`.

@VarCodeMode

Modifies the behaviour with respect to a defined variant code type.

For type `@Complete` currently there is no special mode defined.

For type `@ValuesOnly` the mode specifies the range of properties to be represented in the variant code. The possible modes are:

`0` - only currently defined properties (default)
`1` - all configurable properties

@SpecSeparator

Specifies the String to be used to separate base article number and variant code in the end (final) article number.

Default: a single space.

@VarCodeValueSep

Specifies the String to be used to separate property values in variant codes of type `@ValuesOnly`.

Default: a single space.

@VoidValVarCodeChar

Specifies the character to be used to represent unspecified optional properties in variant codes of type `@ValuesOnly`.

In the variant code there will be placed as much instances of this character as the specified number of digits for value representation.

Default: '`X`'.

@InvalidVarCodeChar

For variant codes of type `@ValuesOnly` in mode `1` (see *@VarCodeMode*) specifies the character to be used to represent currently invalid configurable properties.

In the variant code there will be placed as much instances of this character as the specified number of digits for value representation.

Default: '–'.

@FixVarCodeValLen

For variant codes of type `@ValuesOnly` specifies whether value codes for each property have a fix length according to the count of digits specified for each property. There, integer value `1` means *Yes/True*, and `0` means *No/False*.

In the case of `0`, padding space characters are stripped.

Default: `0`, if value separator (see *@VarCodeValueSep*) is not an empty string, otherwise `1`.

5.3. Specific options of native OCD implementation

@AllowConsecValsInTrimmedCode

This option refers to the processing of final article numbers (in order to re-create an encoded article variant) which are encoded according to a user-defined scheme where the trim flag is set (i.e., spaces at the end of a property value are removed). Such end article numbers cannot be processed with the standard method if two properties immediately follow each other in the final article number (i.e., are not separated by separator characters). With this option, an extended method can be enabled, which – at the expense of the performance – makes it possible to process even such end article numbers with consecutive properties.

`0` (or `false`)
: The extended procedure *should not* be used.

  This is the default behavior (if there is no table entry for this option).

`1` (or `true`)
: The extended procedure is to be used.

@DeactivateValuePreconds

With this option, the evaluation of value preconditions generally can be prevented.

The value of the option (3rd field) is treated and evaluated like an OCD condition (logical expression)[8]. Among other things, it is therefore possible to reference properties.

Example: `@DeactivateValuePreconds;;SPECIAL = '1'`

The result of the expression must be definitely true so that the deactivation of the preconditions becomes effective[9].

[8]The condition is evaluated before the evaluation of the relationship knowledge of the processed article.
[9]For more on undefined logical expressions see OCD specification.

The option aims at a very concrete scenario: Given is an article with many properties, which in turn have very large sets of values. By setting a special property (in the example above `SPECIAL`), the article should be ordered in a special version, for which all values must/should be selectable (valid). For this, all value preconditions would have to be supplemented by the sub-condition `SPECIAL <> '1'`. Due to the immense amount of values, this is not only costly in terms of data creation, but would also lead to poor performance in the evaluation of the relationships at run-time. This can be avoided by means of this option.

@IgnoreUnknownPropInKeyValueVC

This very special option is only needed in very rare cases, namely when, in the context of an online shop system, an article generated via the interface of the OnlineConfigurator is to be initialized by means of a variant code generated from the leading ERP system, and this variant code is built-up according to the OCD schema *KeyValueList*, but in which the characteristics are not qualified with a class. In this case, the OCD implementation expects that a characteristic contained in the variant code is included in the (current) classes of the article in question. As soon as this does not apply to a characteristic, the processing of the variant code is aborted and there is no initialization of the article with the values coded in the variant code.
This option now controls whether the default behavior described above should be used or whether missing characteristics should be ignored. The latter could be necessary, e.g., if the variant code is generated from an older order where characteristics were used that no longer exist in the current OCD data.

`0` (or `false`)
: An unqualified characteristic in the variant code that is not contained in the classes of the article causes the processing of the variant code to be aborted.

  This is the default behavior (if there is no table entry for this option).

`1` (or `true`)
: An unqualified characteristic in the variant code that is not contained in the classes of the article is ignored when processing the variant code.

@NeedValuesForRGProps

If the option is present and has the value `1`, all values of properties with `RG` are passed to the OFML property generated for the OCD property[10]. This allows for a value change by calling the method *setPropValue()* (OFML interface *Property*). Especially, this is interesting for OAP projects.

@OCDUserMessageMode

This option is used to control the behavior of the application with respect to messages to the users, which are output in the OCD data using the function *USER_MESSAGE()*.

`@MessageOnly`
: The messages should always be output in a modal message dialog (which must be confirmed by the user).

[10] Otherwise, the value choice list contains only the current value of the OCD property, which makes the OFML property read-only.

`@HintPreferred`

The messages should preferably be displayed in a non-modal information window (hint). The messages are to be output in a modal message dialog only if the application does not support hints.

This is the default behavior (if there is no table entry for this option).

`@HintOnly`

The messages may only be displayed in a non-modal information window (hint), that is, if the application does not support hints, the messages are not output.

`@OptPropsWithBaseValues`

This option controls the behavior regarding properties declared as optional for which there are values specified in the article base table. The OCD specification does not contain clear regulations with respect to this situation. There are 2 possible interpretations, which can be chosen by the according value of this option:

`0`

Because the article base table defines the fixed resp. allowed values for selected properties of the article, the optional flag will be ignored, i.e. the property cannot be in the state “not specified” but must always have one of the specified values.

This is the default behavior (if there is no table entry for this option).

`1`

For configurable properties the article base table serves only to restrict the set of values compared to those specified in the value table, i.e., a property declared as optional also can be in the state “not specified”.

`@RelEvalOptimization`

With this option, certain methods for optimizing the evaluation of relationship knowledge can be activated. Currently, only one method is supported, which is activated with the value `2`:

`2`

This optimization refers to the automatic value selection of restrictable properties (see option *`@PreselectRestrictable`*): The value set of a pre-selected property is "frozen", so it cannot be changed by the pre-selection of subsequent properties. Therefore, this optimization requires that pre-selected properties have no dependencies on properties that are further down in the feature list[11].
With this method, a noticeable performance gain results when many properties with large value sets are pre-selected and many of the values have preconditions[12]. If the value sets of the pre-selected properties are restricted exclusively via constraints (which should be the normal case in the OCD data creation), the performance gain is low.

[11] Such dependencies are poor from the user's point of view and should be avoided anyway in the data creation.

[12] Actually, the performance gain results from the fact that significantly fewer preconditions have to be evaluated with this method.

@SetDefaultMode **(obsolete[13])**

Specifies the behavior regarding Builtin-function

`$SET_DEFAULT($self,<property>,<term>)`

in SAP—language subsets[14]:

`@AssignValue`

The value of the expression is assigned to the property if the property currently has no value.

So, the behavior is identical to that of an action of the form

`<property> = <term> if <property> not specified`

This is the default behavior (if the option is not given).

`@SetDefault`

The value of the expression is set as the default value for the property.

So there is no direct value assignment to the property, however, for restrictable properties the default value can be effective in the context of a subsequent automatic value selection (see option *@PreselectRestrictable*).

For optional properties this mode has no meaning. The behaviour for these properties is always as in mode `@AssignValue`.

@SetVisibilityMode

This option defines the scope for function *SET_VISIBILITY()*:

`@Standard`

Characteristics that are declared nonvisible by the function are *not* displayed both in the Property Editor and in the variant text.

This is the default behaviour (if the option is not given).

`@PropEditOnly`

Characteristics that are declared nonvisible are *not* displayed only in the Property Editor (but are included in the variant text).

@ShowExtraCharge4Value

For properties with a value choice list this option may be used in order to display the extra charge which will be imposed if a given value from the list will be selected[15].

For that purpose, in the value field a scheme for relevant variant conditions has to be specified. This scheme consists of arbitrary characters and has to contain the placeholder %V, which will be substituted at run time by the identifier of the property value. Optionally, the scheme may contain the placeholder %P, which will be substituted at run time by the identifier of the property. During initialization of an article, for all possible values (of all properties) the price table will be looked up for entries of level `X` (extra charge) with a fixed price value for a variant condition which results from applying the given

[13]This is no longer required/supported as of the releases in fall 2019. See also application note AN-2014-04.
[14]Note: Officially, this Builtin-function is *not* supported by the OCD—Standard!
[15]Currently, this option is supported only by the property editor of pCon.planner >= 7.

scheme to the property value. If there is such an entry, the extra charge from this entry will be appended to the value description in the choice list.

Example:

```
@ShowExtraCharge4Value;;%P_%V
```

Let's assume, an article possesses a property `FOO` with a value `BAR` and in the price table exists an entry of level `X` with a fixed price value for variant condition `FOO_BAR`, then the extra charge from this entry will be appended to the description of value `BAR` in the choice list.

@ShowPrecondInvalidValues

This option is related to restrictable properties:

If there is an entry for this option and its value is `1`, then property values currently invalid due to preconditions will be displayed in the value choice list as inactive (greyed out).

Normally (no entry for this option or in case of value `0`), currently invalid property values will not be presented in the value choice list.

Note: This option has no impact on property values which became invalid due to constraints. These values in any case will *not* be presented in the value choice list.

@UnlockBackwardRestriction

Occasionally, OCD relationship knowledge includes so-called backward dependencies. These lead to the fact that the value set of a property is influenced by the value of another property which is further down in the property list. In extreme cases, it can happen that the dependent property has only one valid value after the property further down in the list has been changed and, therefore, can no longer be changed.

A (typical) example of a backward dependency is the following *Constraint*:

```
Objects:
    T is_a MAT_PROPS.
Restriction:
    Table STEEL_COLOR_59 (
        STEEL_COLOR_59 = T.STEEL_COLOR_PGR,
        STEEL_COLOR    = T.STEEL_COLOR).
Inferences:
    T.STEEL_COLOR_PGR, T.STEEL_COLOR.
```

Since both the price group property `STEEL_COLOR_PGR` and the associated finish property `STEEL_COLOR` are listed under `Inferences`, both properties restrict each other. If the user changes the property `STEEL_COLOR`, which is further down in the list of properties, property `STEEL_COLOR_PGR` may be blocked as described above.

From the user's point of view, such backward dependencies should be avoided in principle.

In the example above, ideally only the finish property should be dependent on the price group property.

If this is not possible[16] and if the properties that imply a backward dependency are restrictable properties, this option (value `1`) can be used to enable an exception handling in the OCD implementation:
A temporary reset of the current value of the changed property and an additional evaluation cycle of the relationship knowledge unlocks properties higher up in the list.

Implications of this exception handling are:

[16] e.g. due to peculiarities of the ERP system from which the OCD data is exported

1. Any existing relationships of the type *Reaction* for the changed restrictable property may become ineffective under certain circumstances!!
2. In extreme cases, the value just set can become invalid (and a different value can be assigned to the property).
3. Slightly poorer performance due to the additional evaluation of relationship knowledge.

The possible problems as a result of implications 1 and 2 require extensive testing by the manufacturer (data creator) when using this option!

Note:
The additional evaluation cycle has no effect if the relevant properties depend on another, restrictable property by a precondition and only become valid/visible when this property is pre-selected[17]!

`@UnfixPreselectedChoiceList`

When automatically pre-assigning values to a restrictable property (see option `@PreselectRestrictable`), by default the set of values of the property shown to the user is "frozen", i.e., corresponds to the set of valid values defined at that time (thus cannot be changed due to the pre-assignment of subsequent characteristics).

However, the underlying assumption that the properties are or can be arranged in the property list according to a logical order from top to bottom is not always fulfilled, see example below. In that case, this option can be used with the value `1`. The set of values visible to the user then corresponds to the set of valid values defined after the pre-assignment of all restrictable characteristics.

Note: If the option `@RelEvalOptimization` has the value 2, this option has no effect.

An application **example** for this option would be a media cabinet with a built-in TV. Depending on the selected model, a certain number of HDMI and USB ports are available for additional devices. Each added device consumes a certain number of ports. If all ports are occupied, no further additional device can be installed. The add-on devices are configured/inserted via corresponding properties, but there is no defined priority order for the different types of devices.

[17] see option `@PreselectRestrictable` in section 5.1

6. Tables for Planning Groups

Tables `jointplgroup`, `layoutgroup`, `tabularplgroup` and `customplgroup` can be used to control some aspects of the behavior of planning group instances whose type is *xOiJointPlGroup*, *xOiLayoutGroup*, *xOiTabularPlGroup* resp. *xOiCustomPlGroup*[18].

If the tables are not found in standard location (library path) they will be looked up in the path specified in DSR configuration file for given OFML series under key `proginfodb_path`.

If argument field (2) is empty the value specified in field 3 for a given info type (field 1) is valid for any instance of this class. If argument field is *not* empty it has to contain a condition in the form of a vector

```
[condition_type, condition_value].
```

If multiple conditions are specified, they have to be separated by a comma.

Currently, the following **condition types** are supported:

`@Article`
: is true, if the instance represents an article and if its base article number starts with one of the strings specified in the condition value.
The strings in the condition value must be separated by a comma and enclosed in square brackets.

`@ArticleX`
: is true, if the instance represents an article and if its base article number is *exactly* the same as one of strings specified in the condition value.
The strings in the condition value must be separated by a comma and enclosed in square brackets.

  This condition type should/can be used instead of `@Article` if there are several planning group articles with a common article number root, in order to avoid dependency on the order of the table entries.

`@ArticleNOT`
: is true, if the instance represents an article and if its base article number does ***not*** start with any of the strings specified in the condition value.
The strings in the condition value must be separated by a comma and enclosed in square brackets.

`@Property`
: is true, if the key of a processed common property starts with one of the strings specified in the condition value.
The strings in the condition value must be separated by a comma and enclosed in square brackets.

`@PropertyNOT`
: is true, if the key of a processed common property does ***not*** start with any of the strings specified in the condition value.
The strings in the condition value must be separated by a comma and enclosed in square brackets.

There may be only one article-related condition[19].

[18] or is derived from these types
[19] If more article conditions are specified, all except of the first will be ignored.

Property-related conditions currently are supported for the following options[20]:

- `@CommonPropsChLMode`
- `@SortUnionCommonProps`
- `@NonVisibleProps4Common`
- `@ROPropsEditable4Common`
- `@Classes4CommonProps`
- `@ArticleFeatures4Common`

If the group instance does ***not*** represent an article, table entries with non-empty argument fields will be ignored and first entry with an empty argument field for a given info type rules. (I.e., neither article-related nor property-related conditions may be applied for group instances not representing an article.)

If the group instance represents an article, the processing of the table entries with respect to a requested option depends on whether the option supports property-related conditions (see above):

- If the option *does not support* property-related conditions, the first entry with a matching article-related condition will be used. If there is no entry with a matching article related condition, the first entry with an empty argument field rules (if any).
- If the option *supports* property-related conditions, all entries with property related conditions *and* a matching article condition are considered first. For a given common property, then the first of these entries is used whose property conditions are satisfied. If there is no such an entry, the first entry with an empty argument field rules (if any).

6.1. Common options

`@InsertMode`

The insert mode controls the behavior regarding insertion of new elements between two existing layout elements[21] and dimension change of elements.

This option is not relevant for class *xOiTabularPlGroup* (table `tabularplgroup`)!

The following modes are defined:

`0` Insertion between two layout existing elements and dimension change of elements is not allowed.
This is the default mode.

`1` Dimension change of elements is allowed.

`2` Both insertion and dimension change are allowed.

Note:
The algorithms for handling modes 1 and 2 are based on the detection of collisions between the elements of the planning group. Therefore, for a correct functioning, the elements of the planning group must not be excluded from collision detection using method *disableCD()* (OFML interface *Base*)!

[20] Any property condition specified for other options will be ignored.
[21] For class *xOiJointPlGroup* this refers to elements contained in the topological order list.

@SelectableState4Layout

Specifies, whether layout elements[22] are selectable by the user, where integer value `1` means *Yes/True*, and `0` means *No/False*. Default: `1`.

Initial state according to OFML interface *Base* is `1`, occasionally, however, it is desirable (necessary) that the layout elements are not selectable, e.g. if the elements should (may) only be changed via methods of the planning group.

@SelectableState4Other

Specifies, whether non-layout elements[23] are selectable by the user, where integer value `1` means *Yes/True*, and `0` means *No/False*. Default: `1`.

Initial state according to OFML interface *Base* is `1`, but non-layout elements (such as frames) typically should not be selectable.

@CutableState4Layout

Specifies the state according to method *setCutable()* (OFML interface *Base*) to be assigned to layout elements[24].

If there is no matching table entry for this option, elements get the initial (default) state `1`, i.e. they can be deleted from the group by using the *Cut* or *Delete* operation of the application.

If the elements only are allowed to be deleted in the context of a specific interaction (for example, an OAP Delete action), then this option should be used to assign the status `-1`.

@CutableState4Other

Specifies the state according to method *setCutable()* (OFML interface *Base*) to be assigned to non-layout elements[25].

It's not necessary to specify this option if option *@SelectableState4Other* (see above) has value `0`.

See also the notes above for option *@CutableState4Layout*.

@ShowAsArticle

Specifies, whether the group instance – if it represents an article – has to be displayed in the basket as an article (`1`) rather than a group/folder (`0`).
Default: `0`

@IsPseudoArticle

Specifies whether the group instance represents a pseudo article, i.e. belongs to category *@PseudoArticle*[26] (`1`) or not (`0`).

[22] For class *xOiJointPlGroup* this refers to elements contained in the topological order list.
[23] For class *xOiJointPlGroup* this refers to elements not contained in the topological order list.
[24] For class *xOiJointPlGroup* this refers to elements contained in the topological order list.
[25] For class *xOiJointPlGroup* this refers to elements not contained in the topological order list.
[26] see specification of OFML interface *Article*

Default is `1` if the group instance represents an article and option `@ShowAsArticle` (see above) has the (default) value `0`. Otherwise, the default is `0`.

`@IsNonOrderArticle`

Specifies whether the group instance belongs to category *@NonOrderArticle*[27] (`1`) or not (`0`).
Default: 0

Note: Category *@NonOrderArticle* presupposes category *@PseudoArticle*!

`@CommonProps`

Specifies the common properties to be pulled from the element level to the planning group level.

The value is a listing of the keys of the relevant properties (OFML symbols) separated by a comma and enclosed in square brackets[28].

Option `@AllObjs4CommonProps` (see below) specifies whether the attributes of the properties (type, value set etc.) will be taken from all layout elements or only from the first layout element. The latter is the default behavior, which assumes, that all layout elements[29] possess the same attributes with respect to these properties.

Additionally, option `@NonLayout4CommonProps` (see below) can be used to specify that non-layout elements also have to be considered for common properties.

If various elements are considered and two elements have the same (common) property but with different attributes, the first element in the list "wins". But see option `@CommonPropsChLMode`, specifying the way the choice lists of common properties are built.

Only those properties will be added to the group instance which are actually defined for the relevant elements. In doing so, the order specified in `@CommonProps` is taken into account. However, if option `@AllObjs4CommonProps` has the value `1`, the order may differ if the first layout element does not have all the properties specified in `@CommonProps`. (See also options `@FirstPos4CommonProps` and @`CommonPropsPos` below.)

After a change of a common property the set of the common properties will be updated in order to react on possible dependencies.

This option is available/active only if at least one initial sub article is specified in option `@StartLayout` (resp. `@StartElement` with *xOiJointPlGroup*) and if it was created successfully.

Note:
For a better performance, the common properties should also be specified in the table `non_pd_properties` (see appendix A.2), if the article of the planning group does not possess these properties (what probably is true in most of the cases).

[27] see specification of OFML interface *Article*
[28] OFML-Vector representation
[29] For class *xOiJointPlGroup* this refers to elements contained in the topological order list.

@FirstPos4CommonProps

Specifies the position in the property list to be used for first common property[30].
Default: 1

Note: If further (fix) properties will be defined in derived classes, they should be positioned either after or before the common properties[31].

See also options *@FirstPos4CommonProps* and *@CommonPropsPos* below.

@CommonPropsPos

This option can be used to specify specific positions for common properties.

The value field contains the OFML String Notation of a *Vector* whose elements are either of type *Int* or of type *Void*.

The order in this option corresponds to the order of the properties specified in option *@CommonProps* (see above), i.e., first entry in *@CommonPropsPos* specifies the position for the first property in *@CommonProps* etc.

If an entry in the Vector is an integer >= 0, the corresponding common property gets this position (according to the regulations specified for method *setPropPos()* of interface *Property*). Otherwise, if the entry has type *Void*, the position of the corresponding common property is determined based on option *@FirstPos4CommonProps* (see above) and the position of the property in option *@CommonProps*.

If there are fewer entries in this option than in *@CommonProps*, the missing entries in *@CommonPropsPos* are treated as if they had a value of type *Void*.

@AllObjs4CommonProps

Specifies whether all layout elements have to be considered for common properties (1) or only the first layout element (0).
Default: 0

@NonLayout4CommonProps

Specifies whether also non-layout elements have to be considered for common properties (1) or not (0).
Default: 0

Note: Value 1 has an effect only if option *@AllObjs4CommonProps* also has the value 1 !

@CommonPropsChLMode

Specifies the way the choice lists of common properties are built.

The following modes are supported:

@FirstEl
: The first element possessing a given common property rules, i.e. the property definition is taken from that element (including a possible choice list).

[30] With instances of class *xOiTabularPlGroup*, the common properties actually get the positions starting with the value of this option plus 3 as the first 3 positions of the properties generated by this class are occupied by the *Resize* properties.
[31] Consider the regulations regarding property positions according to specification of method *setPropPos()* of interface *Property*.

This is the default if option *@AllObjs4CommonProps* (see above) is not given or has the value 0.

This mode offers a slightly better performance (than the other 2 modes), but should only be used if option *@AllObjs4CommonProps* does not have the value 1 and if all layout elements actually possess the same attributes with respect to the common properties.

@Intersection

If various elements have the same attributes with respect to a given common property but different choice lists, the choice list of the common property is formed by the intersection of the choice lists of the elements.

If the resulting choice list is empty, the property will be ignored[32]!

This is the default if option *@AllObjs4CommonProps* has the value 1.

@Union

If various elements have the same attributes with respect to a given common property but different choice lists, the choice list of the common property is formed by the union of the choice lists of the elements.

Using the property-related condition types @Property and @PropertyNOT, different values can be set for this option for various common properties.

@SortUnionCommonProps

Specifies whether the values in a choice list of a common property built with mode @Union (see option @CommonPropsChLMode above) have to be sorted (1) or not (0).
Default is 1 for numeric properties, 0 for symbolic properties (types Y and YS).

@Classes4CommonProps

Specifies whether the property class for a common property has to be taken from the property at element level (1) or not (0).
Default: 0 (i.e., common properties get no class)

@Meta4CommonProps

Specifies whether properties of a (possibly) encapsulating instance (e.g. a Meta type instance) should be considered for common properties (1) or rather the properties of the actual article instance (0).
Default: 0

@CommonPropsDepth

Specifies the levels of sub articles below the layout elements which have to be considered for common properties.
Default is 0, i.e., only properties of the layout elements are considered.

@NonVisibleProps4Common

Specifies whether non-visible properties should be included in the set of common properties (1) or not (0).
Default: 0

[32] i.e. will not be defined for the group instance

If included, they will be editable by the user.

Using the property-related condition types `@Property` and `@PropertyNOT`, different values can be set for this option for various common properties.

`@ROPropsEditable4Common`

Specifies whether properties which originally are read-only have to be editable if included in the set of common properties (`1`) or not (`0`).
Default: `0`

Using the property-related condition types `@Property` and `@PropertyNOT`, different values can be set for this option for various common properties.

`@ArticleFeatures4Common`

Specifies whether a language dependent description for the common properties of the first layout element has to be included in the result of method *getArticleFeatures()*[33] (`1`) or not (`0`).
Default: `0`
Note: For this option to be effective, option *`@ShowAsArticle`* must have value 1.

6.2. Specific options for *`xOiJointPlGroup`*

`@_2DMode`

Specifies the mode for the 2D representation of non-empty planning group:

`@Elements`
: Use (only) the 2D representation of the elements themselves (recommended).

`@Group`
: Generate 2D representation for elements (ignoring own 2D representation of the elements).

`@Combined`
: Use 2D representation of the elements and 2D representation generated by the group instance.

  This is the default (due to backward compatibility).

`@Rectangle`
: Generate a simple bounding volume rectangle.

`@IgnorePlanDir4Remove`

Specifies whether first resp. last element of topological order can be removed independent of current planning direction of the application (`1`) or not (`0`).

In the case of default value `0` the last element only can be removed if current planning direction is "*to the right*" (and analogical for the other direction).

Note: Default planning direction is "*to the right*". Not all applications allow the user to specify a different direction.

[33] OFML interface *Article*

@AllElements4SubArticle

Specifies whether all group elements (which represent an article) should be considered as sub articles of the group article.

In the case of default value `0` only the elements from the topological order list will be considered as sub articles.

If elements (such as back walls, cover plates and alike) which are not included in the topological order list also should be sub articles, value `1` has to be specified.

@StartElement

Specifies information to be used to create an initial sub article right after the planning group instance was created (and initialized with an article number).

The value field contains the OFML String Notation of a Vector containing the following elements:

1. OFML program (*Symbol*)
2. Base article number *(String*)
3. optional: Variant code type (*Symbol*: `@VarCode`, `@OFMLVarCode`, `@Final`)
4. optional: Variant code (*String*) , may be incomplete
5. optional: Property values (*Vector* of [key, value] pairs)
6. optional: Property states (*Vector* of [key, state] pairs)

   Currently, the following state values can be specified[34]:
   `0` – property is not visible
   `1` – property is visible, but not editable
   `3` – property is visible and editable

Examples:

```
@StartElement;;[@foo_bar,"ArtNr1"]
@StartElement;;[@foo_bar,"ArtNr2",@OFMLVarCode,"PROP1=814"]
@StartElement;;[@foo_bar,"ArtNr3",@VarCode,"",[[@Prop1,1],[@Prop2,@abc]]]
```

@StartLayout

Specifies information to be used to create initial sub articles (as layout elements) right after the planning group instance was created (and initialized with an article number).

Note: Option *@StartLayout* takes precedence over (possibly simultaneously specified) option *@StartElement*!

[34] according to the specification of method *setPropState2()* of OFML interface *Property*

The value field contains the OFML String Notation of a Vector of Vectors each containing the following elements:

1. Attach point of predecessor to be used to position this element (*Void/Symbol*[35])
2. OFML program (*Symbol*)
3. Base article number (*String*)
4. optional: Variant code type (*Symbol*: `@VarCode`, `@OFMLVarCode`, `@Final`)
5. optional: Variant code (*String*) , may be incomplete
6. optional: Property values (*Vector* of [key, value] pairs)
7. optional: Property states (*Vector* of [key, state] pairs)
   Currently, the following state values can be specified[36]:
   `0` – property is not visible
   `1` – property is visible, but not editable
   `3` – property is visible and editable

Example:

```
@StartLayout;;\
[[NULL,@foo_bar,"ArtNr1"],\
 [@ATTPT_R1,@foo_bar,"ArtNr2",@OFMLVarCode,"PROP1=814"],\
 [NULL,@foo_bar,"ArtNr3",@VarCode,"",[[@Prop1,1],[@Prop2,@abc]]]]
```

6.3. Specific options for *xOiLayoutGroup*

@ChildBranchGrowthMode

Specifies whether child branches of the layout structure may grow only in one direction (`@OneDir`) or in both directions (`@BothDirs`).
Default: `@OneDir`

@ContinueBranch4SingleForkEl

Specifies whether the branch of a fork element has to be continued if a neighbor is added to it and the fork element is currently the single element in the branch.

Due to backward compatibility, the default behavior is defined by the value `0` (no/false) since in principle a new branch is started when attaching a neighbor to a fork element.

However, in the given scenario, the behavior defined by the value `1` (yes/true) is probably the desired behavior in most cases.

@UseAllAttPts

Specifies whether all attach points (including the standard ones) are used to connect layout elements (`1`) or only additional attach points (`0`).

For better performance, use value `1` only if standard attach points actually are used to connect layout elements.
Default: `0`

[35] If an attach point is specified (Symbol) it will be used to position this article instance beside of the predecessor (if any), otherwise the position is determined via a standard procedure.
[36] according to the specification of method *setPropState2()* of OFML interface *Property*

@StartLayout

Specifies information to be used to create initial sub articles (as layout elements) right after the planning group instance was created (and initialized with an article number).

The value field contains the OFML String Notation of a Vector of Vectors each containing the following elements:

1. Attach point of predecessor to be used to position this element (*Symbol*[37])
2. OFML program (*Symbol*)
3. Base article number (*String*)
4. optional: Variant code type (*Symbol*: `@VarCode`, `@OFMLVarCode`, `@Final`)
5. optional: Variant code (*String*) , may be incomplete
6. optional: Property values (*Vector* of [key, value] pairs)
7. optional: Property states (*Vector* of [key, state] pairs)

   Currently, the following state values can be specified[38]:

   `0` – property is not visible

   `1` – property is visible, but not editable

   `3` – property is visible and editable

Example:

```
@StartLayout;;\
[[NULL,@foo_bar,"ArtNr1"],\
 [@ATTPT_R1,@foo_bar,"ArtNr2",@OFMLVarCode,"PROP1=814"],\
 [@ATTPT_R2,@foo_bar,"ArtNr3",@VarCode,"",[[@Prop1,1],[@Prop2,@abc]]]]
```

6.4. Specific options for *xOiTabularPlGroup*

@MaxColumns

Specifies the maximum allowed count of columns, where default value `0` denotes no restriction[39].

@MaxRows

Specifies the maximum allowed count of rows, where default value `0` denotes no restriction [40].

@UniformColumnWidth

Specifies whether all columns have the same width (`1`) or not (`0`).
Default: `1`

The uniform width can be specified in option *@ColumnWidth*.

@ColumnWidth

Specifies the uniform width of the columns (in *meter*), if option *@UniformColumnWidth* has value `1`.
Default: `1.0`

[37] `NULL` in case of the first start element
[38] according to the specification of method *setPropState2()* of OFML interface *Property*
[39] The value initially set by this option can be overwritten using method *setMaxColumns()*.
[40] The value initially set by this option can be overwritten using method *setMaxRows()*.

@ColumnWidths

Specifies the widths of the columns (in *meter*), if option *@UniformColumnWidth* has value 0.

In the value field a vector of float values is specified, with the index of the vector element corresponding to the index of the relevant column.

If a new column is inserted into the layout for whose index the Vector in the value field of this option does not contain an element, the value from the last Vector element is used for this column.

Default: [1.0]

@UniformRowSize
@UniformRowDepth
@UniformRowHeight

Specifies whether all rows have the same depth resp. height (1) or not (0).
Default: 1

Option *@UniformRowSize* can be used with both orientations, option *@UniformRowDepth* may be used only with horizontal orientation and option *@UniformRowHeight* may be used only with vertical orientation. If two options are specified, option *@UniformRowSize* takes precedence.

@RowSize
@RowDepth
@RowHeight

Specifies the uniform depth resp. height of the rows (in *meter*), if options *@UniformRowSize*, *@UniformRowDepth* resp. *@UniformRowHeight* have the value 1.
Default: 1.0

Option *@RowSize* can be used with both orientations, option *@RowDepth* may be used only with horizontal orientation and option *@RowHeight* may be used only with vertical orientation. If two options are specified, option *@RowSize* takes precedence.

@RowSizes
@RowDepths
@RowHeights

Specifies the depths resp. heights of the rows (in *meter*), if options *@UniformRowSize*, *@UniformRowDepth* resp. *@UniformRowHeight* have the value 0.

In the value field a vector of float values is specified, with the index of the vector element corresponding to the index of the relevant row.

If a new row is inserted into the layout for whose index the Vector in the value field of this option does not contain an element, the value from the last Vector element is used for this row.

Default: [1.0]

Option *@RowSizes* can be used with both orientations, option *@RowDepths* may be used only with horizontal orientation and option *@RowHeights* may be used only with vertical orientation. If two options are specified, option *@RowSizes* takes precedence.

@AlignmentX

Specifies the alignment of the elements within the fields along X axis.

The possible values are: `@Left` (default), `@Right`, `@Centered`

@AlignmentY

Specifies the alignment of the elements within the fields along Y axis.

The possible values are: `@Bottom` (default), `@Up`, `@Centered`

In case of a group with a vertical orientation the alignment relates to the height of the fields, otherwise to the local bounding box of the field element.

@AlignmentZ

Specifies the alignment of the elements within the fields along Z axis.

The possible values are: `@Back` (default), `@Front`, `@Centered`

In case of a group with a horizontal orientation the alignment relates to the depth of the fields, otherwise to the local bounding box of the field element.

@DefaultArticle

Specifies information to be used to create articles if no specific article information is specified/available.

The value field contains the OFML String Notation of a Vector containing the following elements:

1. OFML program (*Symbol*)
2. Base article number *(String*)
3. optional: Variant code type (*Symbol*: `@VarCode`, `@OFMLVarCode`, `@Final`)
4. optional: Variant code (*String*) , may be incomplete
5. optional: Property values (*Vector* of [key, value] pairs)
6. optional: Property states (*Vector* of [key, state] pairs)
   Currently, the following state values can be specified[41]:
   `0` – property is not visible
   `1` – property is visible, but not editable
   `3` – property is visible and editable

Examples see option *@StartElement* in section 6.2.

@EmptyFieldsPlaceholder

Specifies whether empty fields should be filled by transparent and selectable placeholder objects (`1`) or not (`0`).
Default: `0`

If the option has value `1`, options *@FieldPlaceholderThickness*, *@FieldPlaceholderMat* and *@FieldPlaceholderArticle* (see below) can be used to influence certain aspects of the placeholder objects.

[41] according to the specification of method *setPropState2()* of OFML interface *Property*

`@FieldPlaceholderThickness`

Specifies the height[42] resp. depth[43] of placeholder objects.

Default: `0.1`

`@FieldPlaceholderMat`

Specifies the material to be used for placeholder objects.

In the value field, a fully qualified OFML material name is expected.

Default is a material of type *Glass*.

`@FieldPlaceholderType`

Specifies the fully qualified OFML type (*String*)to be used for placeholder instances instead of default type *xOiTabularPlGroupPlaceholder*.

The specified type must be an immediate subtype of *xOiTabularPlGroupPlaceholder*!

`@FieldPlaceholderArticle`

Specifies the article information for placeholder objects.

The value field contains the OFML String Notation of a Vector containing the following elements:

1. OFML program (*Symbol*)
2. Base article number (*String*)
3. commercial manufacturer ID (*String*)
4. commercial series ID (*String*)

Default: no article information, i.e., the placeholder objects don't represent an article (and, thus, do not appear in article listings).

Note:
The article information is not used to create an instance of the type mapped to the article in OAM data (if any) but will be assigned to the placeholders represented by instances of internal type *xOiTabularPlGroupPlaceholder*.

`@FieldsPlaceholder4SubArticle`

Specifies whether placeholders representing articles[44] should be represented in the basket structure as sub articles (`1`) or not (`0`).
Default: `1`

If the option has value `0`, there is no need to specify the placeholder article number in overridden method *getInvalidSubArticleSpecs()* (see specification of base class *xOiPlGroup* in XOI documentation).

[42] with horizontal orientation of the planning group
[43] with vertical orientation of the planning group
[44] according to option `@FieldPlaceholderArticle`

@StartLayout

Specifies information to be used to create initial sub articles (as field elements) right after the planning group instance was created (and initialized with an article number).

The value field contains the OFML String Notation of a Vector containing the following elements:

1. Creation mode (*Symbol*):

   `@Uniform`
   Creates as many fields as specified in second element filling them with articles according to option *@DefaultArticle*.

   `@Specific`
   Creates individual field elements as specified in the second element.

2.
   - With mode `@Uniform`, a two-digit vector of integer values defining the numbers of columns (1st element) resp. rows (2nd element) to be created.
   - With mode `@Specific` a Vector of Vectors each containing the following elements:
     1. Field address of the layout element: [column index (*Int*), row index (*Int*)][45]
     2. Article information according to the specification of option *@DefaultArticle*

6.5. Specific options for *xOiCustomPlGroup*

@StartLayout

Specifies information to be used to create initial sub articles (as layout elements) right after the planning group instance was created (and initialized with an article number).

The value field contains the OFML String Notation of a Vector of Vectors each containing the following elements:

1. Attach point of predecessor to be used to position this element (*Symbol*[46])
2. OFML program (*Symbol*)
3. Base article number (*String*)
4. optional: Variant code type (*Symbol*: `@VarCode`, `@OFMLVarCode`, `@Final`)
5. optional: Variant code (*String*) , may be incomplete
6. optional: Property values (*Vector* of [key, value] pairs)
7. optional: Property states (*Vector* of [key, state] pairs)
   Currently, the following state values can be specified[47]:
   `0` – property is not visible
   `1` – property is visible, but not editable
   `3` – property is visible and editable

Example:

```
@StartLayout;;\
[[NULL,@foo_bar,"ArtNr1"],\
 [@ATTPT_R1,@foo_bar,"ArtNr2",@OFMLVarCode,"PROP1=814"],\
 [@ATTPT_R2,@foo_bar,"ArtNr3",@VarCode,"",[[@Prop1,1],[@Prop2,@abc]]]]
```

[45] indexes start with 0
[46] `NULL` in case of the first start element
[47] according to the specification of method *setPropState2()* of OFML interface *Property*

@StoreNeighborship

Specifies whether neighbor-relationships between layout elements have to be stored in an internal data structure (`1`) or not (`0`).
Default: `1`

If neighbor-relationships based on OFML attach points are not relevant in a specific project, value `0` should be specified in order to avoid overhead (saving memory and improving performance).

Appendix

A.1 OFML keywords

The keywords below may not be used as identifiers for OCD properties[48].
Note: the reservation here only refers to the upper and lower case spelling used below, i.e., other spellings are allowed.

```
abstract
break
case
catch
class
continue
default
do
else
final
finally
for
foreach
func
goto
if
import
instanceof
native
ODB_NAME
operator
package
private
protected
public
return
rule
self
static
super
switch
throw
transient
try
var
while
```

[48]See also OFML specification 2.0.3, section 3.2.4

A.2 Table `non_pd_properties`

In this table the properties of an article instance are specified, which are **not** associated with a property in product data (OCD), i.e. which are defined by programming in the OFML class of the article instance. This information is used by the global Product Data Manager to avoid calls to the product database in order to process a change of these properties (thus improving performance).

The table has the following structure:

- Field 1: fully qualified name of the class
- Field 2: property keys (OFML symbols), comma-separated

ISO-8859-1 (Latin-1) is used as the character set.

There may be multiple entries for a given class, in which case the lists of property keys will be concatenated.

The table has to be contained in database `pdate.ebase` of the according OFML series (i.e., together with the OCD data for the articles of the series).

The table description for the EBase configuration file is:

```
table  non_pd_properties_tbl non_pd_properties.csv variable_width
fields 2
field  1 class vstring delim ; trim hidx link
field  2 pkey  vstring delim ; trim
```

As mentioned in section 6.1, for a better performance, the properties specified in option *`@CommonProps`* in the control data table for a planning group should also be specified in table `non_pd_properties` (if the article of the planning group does not possess these properties).

Example:

jointplgroup.csv:

```
@CommonProps;[@Article,["GroupXY"]];@Foo,@Bar
```

oam_article2ofml.csv:

```
GroupXY;::ofml::xoi::xOiJointPlGroup;;
```

non_pd_properties.csv:

```
::ofml::xoi::xOiJointPlGroup;@Foo,@Bar
```

A.3 Document history

2023-09-18:

- New option *@UnfixPreselectedChoiceList* in table epdfproductdb (section 5.3)
- Control data tables for planning groups now support the condition types @Property and @PropertyNOT also for options *@Classes4CommonProps* and *@ArticleFeatures4Common* (section 6)
- New options *@CommonPropsPos* and *@SortUnionCommonProps* for all planning groups (section 6.1)
- More precise description of option *@ArticleFeatures4Common* in section 6.1
- New option *@FieldPlaceholderType* as well as more precise descriptions of options *@AlignmentY* and *@AlignmentZ* in table tabularplgroup (section 6.4)
- New section 6.5 for table customplgroup (class *xOiCustomPlGroup*)

2022-09-30:

- More precise description and correction regarding information type @NeedsAddPrices in option *@OrderInfo* in table proginfo (section 2.6)
- Control data tables for planning groups now support the condition type @ArticleX as well as condition types @Property and @PropertyNOT for some options related to common group properties (section 6)
- New options *@IsPseudoArticle*, *@IsNonOrderArticle* and *@CommonPropsChLMode* for planning groups as well as a more precise description regarding option *@CommonProps* in section 6.1
- New option *@ContinueBranch4SingleForkEl* in table layoutgroup (section 6.3)

2021-12-22:

- More precise description of option *@PreselectRestrictable* in table epdfproductdb (section 5.1)

2021-10-11:

- New option *@UnlockBackwardRestriction* in section 5.3
- New options *@SelectableState4Layout* and *@NonLayout4CommonProps* in section 6.1
- New option *@FieldsPlaceholder4SubArticle* in section 6.4

2021-03-23:

- New options *@FirstPos4CommonProps*, *@AllObjs4CommonProps* and *@Classes4CommonProps* in section 6.1
- Revision of option *@StartLayout* in table jointplgroup (section 6.2)
- New appendix describing table non_pd_properties (avoiding reference to XOI documentation in section 6.1)

2020-11-04:

- New option *@NeedPropGroupDescriptions* in table proginfo
- Added remarks regarding options *@InsertMode* and *@CommonProp* in section 6.1 (Common options in the control data tables for planning groups)
- New option *@ArticleFeatures4Common* in section 6.1
- New option *@StartLayout* in table jointplgroup (section 6.2)
- New section 6.4 describing the options in table tabularplgroup

2019-11-29:

- Explicit description of options for tables jointplgroup and layoutgroup (instead of just referring to XOI documentation)

2019-10-24:

- New options *@DeactivateValuePreconds*, *@NeedValuesForRGProps* and *@RelEvalOptimization* in table epdfproductdb
- Option *@SetDefaultMode* (table epdfproductdb) is obsolete as of the releases in fall 2019

2019-07-05:

- New options *@IgnoreUnknownPropInKeyValueVC* and *@SetVisibiltyMode* in table epdfproductdb
- More precise description of option *@SafePropertyNames* in table epdfproductdb incl. new appendix A.1 specifying the OFML keywords
- New section regarding tables jointplgroup and layoutgroup

2017-01-24:

- Added description for Infotype @DeepSumOrder in option *@OrderInfo* in table proginfo
- Added remark for option *@AppInteractorDefs* in table plelement
- New options *@OCDUserMessageMode* and *@AllowConsecValsInTrimmedCode* in table epdfproductdb

2014-07-28:

- Adopted to new CI style
- Some reorganization in section 2
- New options *@UseUnspecPropVal4OdbInfo* and *@UseVoidPropVal4OdbInfo* in table proginfo
- New option *@OptPropsWithBaseValues* in table epdfproductdb

2013-10-25:

- Placeholder %P in option *@ShowExtraCharge4Value* (table epdfproductdb) now is optional
- New options *@SafePropertyNames* and *@SetDefaultMode* in table epdfproductdb

2012-05-23:

- Additional remark about option *@PricesOnRequest* in table proginfo.
- New parameter type @Region for option *@OrderInfo* in table proginfo (incl. examples).
- New subsection for specific options of native OCD implementation (table epdfproductdb)

2010-12-17:

- Added option *@InsignificantPropClasses* in table epdfproductdb

2010-09-14:

- Document structure now is centered around the tables (instead around the OFML types) in order to be more data creator-friendly
- Some stylistic changes and small fixes
- Updated section regarding tables proginfo, plelement and epdfproductdb
- Removed section for obsolete table eplproductdb

2007-08-24:

- Added a lot of stuff for class *xOiProgInfo*

2007-01-08:

- Added stuff and corrections

2006-11-09:

- Initial Release