EasternGraphics
visualize your business

Application Note (2023-11-07)
Notes on OAP Data creation

Contents

References

[an0601] Application Note on Control Data Tables (AN-2006-01). EasternGraphics GmbH

[article] The OFML Interfaces *Article* und *CompositeArticle* (Specification). EasternGraphics GmbH

[methods] Useful OFML methods for OAP data. EasternGraphics GmbH

[oap] OAP – OFML Aided Planning, Version 1.5. EasternGraphics GmbH

[ocd] OCD – OFML Commercial Data (Version 4.3). EasternGraphics GmbH

[ofml] OFML – Standardized Data Description Format of the Office Furniture Industry.
Version 2.0, 3rd revised edition.
Industrieverband Büro- und Arbeitswelt e.V. (IBA)

[property] The OFML Interface *Property* (Specification). EasternGraphics GmbH

[xoi] XOI – Library extending the basic OFML implementation library OI (Documentation).
EasternGraphics GmbH

Except [xoi], the specifications are available via the *pCon Download Center*

https://download-center.pcon-solutions.com

in the category *OFML Specifications*.

1 General

1.1 Extensions in table structures / EBase

In the course of the further development of OAP, it occasionally comes to extensions (changes) of already used tables resp. to the definition of new tables. This necessarily causes the release of a new format version (minor or major)[1]. In order to ensure correct processing of the OAP data, the used format version has to be stored in table *Version*.

With each new major resp. minor version of OAP also a corresponding new EBase table description file `oap_<major>_<minor>.inp_descr` is provided.

1.2 Separate series for OAP data

By means of key `oap_program` in the registration file of an OFML series, another series can be referenced in which the OAP data for the articles of the OFML series are located.

In the following situations it is necessary or advisable to create the OAP data in a separate series:

- The OAP data refers to articles from several OFML series.
- Planning groups are used in the OAP project.
  The easiest way to integrate a planning group into the catalog is to create an own (pseudo) article in the OCD data[2]. (In any case, an own article for the planning group is required if OAP interactors are bound to it.) In most cases, it does not make sense to include the article for the planning group(s) together with the actual articles in a single OCD database unless there is such a close coupling between OCD data and OAP data that also a simultaneous distribution is indicated.
- The OAP data changes more frequently[3] or in a different rhythm than the OFML data, and the manufacturer's distribution processes allow for a separate distribution of the OAP data.
- OFML data and OAP data are created or maintained (in parallel) by different persons[4].

1.3 Name conventions for identifiers

In most tables, identifiers (field type *ID*) are used as access keys. As long as there is no application for OAP data creation, these keys must be manually defined. For a quicker orientation in the tables[5] it is recommended to mark the purpose of an identifier with a standardized prefix. The "*OAP style guide*" contains corresponding recommendations.

[1] Conversely, a new minor version can also contain only enhancements that do not require any changes in the table structures (e.g., new interactor symbols).
[2] usually without price and properties, but possibly with an article short text
[3] which is probably true for the early stages of most OAP projects
[4] in the case of OAP e.g. by an external partner
[5] both for the person creating the data, but also for support staff

2 Conditions/Expressions

2.1 OAP expressions vs. OCD expressions

The syntax of expressions in OAP differs from the syntax in OCD expressions [ocd]. Those who have only created OCD data so far have to retrain a bit in this regard. (However, programmers who have already used the OFML programming language should not have any major difficulties since the syntax is very similar.)

OAP expressions are described in detail in appendix A of the OAP specification. For the start, focus on section A.4. For a quicker switch from OCD to OAP, here is an overview of the operators used in **conditions** (logical expressions):

Operator	OCD	OAP
OR relation	`OR`	`||`
AND relation	`AND`	`&&`
Is lower than	`<` (`LT`)	`<`
Is lower equal	`<=` (`LE`)	`<=`
Is equal	`=` (`EQ`)	`==`
Is not equal	`<>` (`NE`)	`!=`
Is greater equal	`=>` (`GE`)	`>=`
Is greater than	`>` (`GT`)	`>`

It should also be noted that expressions in OCD operate with OCD characteristics, whereas expressions in OAP operate with OFML properties. This particularly affects OCD characteristics with data type `C` (Char): if a list of values is associated with these characteristics, the values of the OFML properties generated for these characteristics are symbols (rather than character strings). For comparison, here two conditions in OCD and OAP identical in content:

OCD: `Legs = 'ALU'`

OAP: `Legs == @ALU`

Another significant difference is the representation of string literals (constants): in OCD, these are enclosed in single quotation marks (see example above), in OAP, however, in double quotation marks (`"ALU"`).

2.2 Using standard methods

In expressions (to determine the position of interactor symbols or to formulate conditions for interactors or actions), the use of project-specific implemented methods should be avoided as far as possible, using property values and/or standard methods instead.

The accompanying document *„Useful OFML methods for OAP data“* [methods] describes useful standard methods.

Following section provides an example of the usage of standard methods.

2.3 Article at top planning level?

In OAP data creation, there is often a requirement that certain interactors are *not* valid (should not be displayed) if the article in question is at the top planning level, i.e., if it is not a sub-article of a *composite article*[6].

Instead of programming a special method in the (specific) class of the article that checks this condition, standard methods *getFather()* and *getRoot()*[7] should be used.

oap_methodcall.csv:

```
MC_GET_FATHER;Instance;@IF_Base;getFather;
MC_GET_ROOT  ;Instance;@IF_Base;getRoot  ;
```

With corresponding actions (table *Action*) the condition in table *Interactor* can be formulated as follows:

```
methodCall("AC_call_GET_FATHER") != methodCall("AC_call_GET_ROOT")
```

or if it is to be linked to a condition for the case that the article is a sub-article of a composite article:

```
methodCall("AC_call_GET_FATHER") == methodCall("AC_call_GET_ROOT") ? 0 : <condition>
```

2.4 Numerical metaproperties

Properties for selecting a value from a list of integers or floating-point numbers created in the metadata (table `go_types`), using formats *chi* and *chf* respectively, for historical/technical reasons return the character string representation of the current value via the Property interface, e.g., instead of the number `1200` the string `"1200"`.

Thus, these properties cannot be used directly in OAP in expressions for comparison with numeric values or for determining a coordinate for the position of interactor symbols! Instead, functions *int()* and *float()* have to be used to convert the string value into an integer or a floating-point number.

2.5 Special Symbols

Occasionally, in expressions Symbols have to be specified that cannot be represented as a Symbol literal in normal form. The alternative known from OFML of using the constructor `Symbol()` is not applicable in OAP! Instead, the conversion function `symbol()` or the alternative form to represent a Symbol literal (see appendix A.3.2 in the OAP specification) have to be used.

An example for a *PropChange* action:

```
PC_SET_OPT_WITHOUT;Value;OPT;symbol("---")
```

or

```
PC_SET_OPT_WITHOUT;Value;OPT;@"---"
```

[6] e.g., of a planning group (see section 5)
[7] For the meaning of these both methods see [methods] or [ofml].

2.6 Expressions with integers

The behavior described in appendix A.4.10 of the OAP specification regarding binary arithmetic operators has a small "pitfall" when using integer operands (type *Int*), especially in division:

> "If both operands have a numeric type and at least one operand has type *Float*, the other operand, if necessary, is converted to *Float* and the calculation is performed in *Float*. The result then also has type *Float*. Otherwise, the result has the same type like both operands."

In other words: If both operands have type *Int*, the result also has type *Int*. As a consequence in division, the possibly occurring non-integer remainder of the division is omitted!

This is particularly important in expressions for positions of interactor symbols (table *NumTripel*) using properties of type "i"!

Example:

The active object has a property DEPTH of type "i", which indicates the current depth of the object in millimeters. The interactor symbol has to be placed centered in depth. Since the corresponding Z coordinate has to be given in meters, possibly the following expression would be specified:

```
DEPTH / 2000
```

At a current depth of e.g. 650 mm, the expression returns 0 instead of the desired 0.325!

Therefore, the expression must be written as follows:

```
DEPTH / 2000.0
```

Or the expression is transformed into a multiplication:

```
DEPTH * 0.0005
```

3 Interactors

3.1 Recommendations for using the 3 abstract symbol sizes

large	Interactors for planning groups
medium	Actions, referring to top-level objects or elements of planning groups
small	Actions, referring to child objects (sub-articles)

3.2 Interactor for planning group or group elements?

If a planning group (section 5) is used in the OAP project, certain actions can be triggered both via an interactor at the group instance and via interactors at the elements.

> A typical example is adding and removing elements. Usually this is allowed only at certain places (elements) of the layout. The interactors with the corresponding actions could be displayed for the elements where the action is possible. However, it would also be possible for the group instance itself to place the interactors at the allowed locations (elements).

When making a decision, aspects of user guidance as well as the respective efforts have to be considered and weighed against each other (if necessary, in consultation with the client). Here are some considerations for decision-making.

From the **user's point of view**, interactors bound to group elements are problematic because they are visible only when the user has selected the element in question. However, the inexperienced user often is not even aware that he must first select elements in order to be able to perform certain actions. Furthermore, in the scenario of adding a new element (see above), the user often wants to continue adding to the new element and must first select it again.
On the other hand, many interactors bound to the group can lead to oversaturation and, when the viewer is further away, to unsightly overlapping effects. In order to avoid or reduce the oversaturation of the group, a compromise could be that interactors, which would only be visible/valid for certain elements (see example above), are bound to the group. In contrast, interactors that are valid for (almost) all elements are bound directly to them.

Element-related interactors (actions) bound to the group usually imply more **effort**, since the positions for the interactor symbols have to be determined based on the position (and rotation) of the respective element within the group (for which appropriate methods have to be implemented in the project-specific class for the planning group). Further effort arises if the group instance itself does not have the required functionality (property, method) to perform the desired action. In this case, methods have to be implemented in the project-specific class, that use (call up) the corresponding functionality of the concerned element.
In the case of interactors bound the elements, on the other hand, there may be an effort involved in creating the validity condition if the interactor has to be visible only for certain elements[8]. Here, however, often existing

[8] This corresponds to the above-mentioned recommendation from the user's point of view to rather bind these interactors to the group

standard methods can be used, i.e., the creation of these conditions may not require any project-specific programming.

If the decision has been made to perform certain element-related actions via interactors bound to the group, but the number of possible interactors and the positions of the interactor symbols are difficult to predict/manage, the use of *dynamic interactors* can/should be considered.

Tip:
According to the recommendation from the previous section, interactors of the planning group that refer to individual elements should have a smaller symbol size than the interactors that actually refer to the group as a whole.

3.3 Dynamic interactors

They are realized by implementing the (parameter less) method ***getDynamicOAPInteractors()*** in the class of the relevant article instance (planning group)[9]. The method is called by the application when the instance is selected and after processing the actions of an activated interactor[10]. The interactors supplied by the method then are displayed in addition to the currently valid static interactors.

The return value of the method must be of OFML type *List*, whose elements are of type *Vector* (each) containing the following 2 elements:

1. Interactor ID *(String)*
2. Interactor information *(Vector)*

The ID for a dynamic interactor can be defined freely, but it must be unique and must not match the ID of a static interactor!

The Interactor information in the second element contains the following elements:

1. NeedsPlanMode *(Int)*
2. Action IDs *(String[])*
3. SymbolType *(Symbol)*
4. SymbolSize *(Symbol)*
5. SymbolDisplay information *(Vector)*

Elements 1-4 correspond to the according fields in OAP table *Interactor*.

The elements in the SymbolDisplay information (element 5) again are of type *Vector* each defining a symbol using the following elements (which correspond to the according fields in OAP table *SymbolDisplay*):

1. HiddenMode *(Int)*
2. Offset *(Float[3])*
3. Direction axis *(Float[3] | Void)*
4. ViewAngle *(Float | Void)*
5. Orientation X *(Float[3] | Void)*

[9] i.e., theoretically also are possible with "normal" articles
[10] see also method *xOiOAPManager::getInteractors()*, section 6.1

Notes:

- Since there is no element for a validity condition, the method must/may supply only those interactors that are valid at the time the method is called.
- The data for the specified actions has to be provided/contained in the OAP database. For further notes on this point see below.
- *SymbolType* and *SymbolSize* must be specified as a *Symbol*, whose value corresponds to the identifier of the desired symbol type resp. the desired symbol size grade according to the OAP specification.
- Elements *NeedsPlanMode* and *HiddenMode* must not contain expressions, but have to be specified as an explicit Boolean value (`0` or `1`).
- *Float* values must be specified explicitly, i.e. no expressions are allowed (and the ID of an entry in the table *NumTripel* cannot be used either).

As noted in the 2nd point above, the actions of the dynamic interactor have to be specified in the OAP data. If an action is semantically identical for all relevant planning elements, placeholder **`$INTERACTOR`** can be used. Otherwise, a separate action would have to be created for each relevant planning element with the appropriate target object (object category *MethodCall*) or an appropriate object as argument of a *MethodCall* action. With an arbitrary or unknown number of relevant elements this is practically impossible.

The basic idea of using the placeholder `$INTERACTOR` is to build up the ID of a dynamic interactor defined for a given element in such a way that the relevant element can be retrieved from it. This can be done, e.g.[11], by encoding the local object name of the element within the group into the ID of the interactor. For this purpose methods *localObjName()* and *localName2Obj()* are available in the base class *xOiPlGroup* for all types of planning groups, see example.

The example shows the implementation of dynamic interactors in a planning group class derived from type *xOiJointPlGroup*), which should allow to change the length of each layout element by means of a PropEdit action:

oap_object.csv:

```
OB_self;Self;;;
OB_dynID;MethodCall;AC_call_getElemByDynIID;;
```

oap_action.csv:

```
AC_PE_Length;;PropEdit;PE_Length;OB_dynID
AC_call_getElemByDynIID;;MethodCall;MC_getElemByDynIID;OB_self
```

oap_methodcall.csv:

```
MC_getElemByDynIID;Instance;::foo::bar::barPlGroup;getElemByDynIID;$INTERACTOR
```

[11] Another possibility would be to encode the index of the element in the list of layout elements into the ID of the interactor.

barplgroup.cls:

```
static var sIA_PE_Length_ID_Prefix = "IA_PE_Length_";

public func getDynamicOAPInteractors()
{
    var tRet = @();

    var tEl;
    foreach(tEl; self.getElOrder()) {
        var tIID = sIA_PE_Length_ID_Prefix + self.localObjName(tEl);
        var tPos = self.getEditInteractorPos(tEl);
        var tDisplayInfo = [0, tPos, NULL, NULL, NULL];
        var tData = [tIID, [0, ["AC_PE_Length"], @Edit, @small, [tDisplayInfo]]];
        tRet.pushBack(tData);
    }

    return(tRet);
}

private func getEditInteractorPos(pEl)
{
    // position above the element in the middle of x dimension

    var tOffset =  0.05;
    var tLBB = pEl.getLocalBounds();

    var tX = tLBB[0][0] + (tLBB[1][0]-tLBB[0][0])/2;
    var tY = tOffset;
    var tZ = 0.0;

    return(xOiTransformObjCoords(pEl, [tX, tY, tZ], self));
}

public func getElemByDynIID(pIID)
{
    var tName = pIID.substr(sIA_PE_Length_ID_Prefix.size());

    return(self.localName2Obj(tName));
}
```

Note:
Currently, object category *MethodCall* is not permitted for the target object of an action whose ID is used as the argument of a call to function *methodCall()* (see [oap]).
Therefore, the object ID `OB_dynID` defined in the example above cannot be used for target objects of actions that are used as arguments of *methodCall()* expressions in conditions!

Assume that in the example above, in addition to the *PropEdit* action, an *ActionChoice* action with a selection of objects to be added is to be applied, where the selection depends on the currently selected group element. Method *canAddObject(pType)* of the class of the group element is to be used to determine whether a specific object from the possible selection list can be added to the selected group element.

The condition for an option in table ActionList would then be formulated as follows:

```
methodCall("AC_call_canAddObj_<type>")
```

As object category *MethodCall* cannot be used for the target object of these actions, the method call must not be made directly on the group element. Instead, a method (of the same name) of the planning group class must be called, which in turn calls the method on the relevant group element. In doing so, placeholder `$INTERACTOR` and method `getElemByDynIID()` described above are used:

oap_action.csv:

```
AC_call_canAddObj_<type>;;MethodCall;MC_canAddObj_<type>;OB_SELF
```

oap_methodcall.csv:

```
MC_canAddObj_<type>;Instance;::foo::bar::barPlGroup;canAddObject;$INTERACTOR,<type>
```

barplgroup.cls:

```
public func canAddObject(pIID, pType)
{
    var tRet = 0;
    var tEl = getElemByDynIID(pIID);

    if (tEl != NULL)
        tRet = tEl.canAddObject(pType);

    return tRet;
}
```

3.4 3D interactor symbols

In the case of interactor symbols which illustrate the effective direction of the action associated with the interactor by means of an arrow, the implementation as a 3D symbol should be considered.

Example:
On the right side of an object there is an interactor with symbol type *ChangeDim2Right*, linked to an action which increases the width of the object.

A "normal" 2D symbol, which always is displayed parallel to the screen plane, would always point to the right. This would illustrate the correct direction of the action to the user (in the case of a non-rotated object) only when viewed from front or from above.

Since for 3D symbols the plane has to be specified in which the (flat) symbol icon is to be located for 3D, one has to decide on a view that is most suitable to deal with the object:

- For objects with a low height, the top view is usually the most suitable, i.e., the symbol icon then has to lie in the X-Z plane of the object.
- In the case of objects with a shallow depth, the front or rear view is usually the most suitable, i.e., the symbol icon then has to lie in the X-Y plane of the object.

A 3D symbol intended for the top view would be defined as follows[12]:

oap_numtripel.csv:

```
NT_RightSide;LENGTH * 0.001;0.0;0.0
NT_POS_X_AXIS; 1.0; 0.0;0.0
NT_POS_Y_AXIS; 0.0; 1.0;0.0
NT_NEG_Y_AXIS; 0.0;-1.0;0.0
```

oap_symboldisplay.csv

```
IA_RESIZE;0;Tripel;NT_RightSide;NT_POS_Y_AXIS;360;NT_POS_X_AXIS
```

[12] see also figure 2 in section 4.6 of the OAP specification

Explanation:
The direction vector of the visibility range (`NT_RightSide`, field 5) runs along the positive Y axis (top view). Thus, the 3D symbol lies in the X-Z plane of the object and, due to the entry in field 7, always is oriented along the positive X axis of the local coordinate system of the object.

Notes on the opening angle in field 6

Without specifying an opening angle (empty field 6), no visibility area is defined, which means that the possibly existing entry in field 5 does not take effect for the direction vector. The result would be a normal 2D symbol! If the visibility range actually is not to be restricted, an opening angle of 360 degrees must be specified, as in the example above. However, since the 3D symbol is becoming "flatter" and therefore less recognizable the closer the viewing angle approaches the X-Z plane of the object, a restriction of the visibility range is recommended for 3D symbols, e.g.:

```
IA_RESIZE;0;Tripel;NT_RightSide;NT_POS_Y_AXIS;150;NT_POS_X_AXIS
IA_RESIZE;0;Tripel;NT_RightSide;NT_NEG_Y_AXIS;150;NT_POS_X_AXIS
```

Then, the interactor would have 2 symbols with mutually exclusive visibility ranges, which would not be visible at a viewing angle of less than 15 degrees to the X-Z plane of the object. (The second entry can be omitted if the interactor should be visible only when the user looks at the object from above.)

Analogous, a 3D symbol intended for the front and rear view (in the example with symbol type *ChangeDim2Right*) would be defined as follows:

oap_numtripel.csv:

```
NT_RightSide;LENGTH * 0.001;0.0;0.0
NT_POS_X_AXIS; 1.0; 0.0; 0.0
NT_POS_Z_AXIS; 0.0; 0.0; 1.0
NT_NEG_Z_AXIS; 0.0; 0.0;-1.0
```

oap_symboldisplay.csv

```
IA_RESIZE;0;Tripel;NT_RightSide;NT_POS_Z_AXIS;150;NT_POS_X_AXIS
IA_RESIZE;0;Tripel;NT_RightSide;NT_NEG_Z_AXIS;150;NT_POS_X_AXIS
```

3.5 Correct configuration context?

The validity condition in OAP table *Interactor* can resp. must be used to ensure that the interactor is only displayed if the current configuration context allows it.

A common requirement is that an interactor should only be displayed if the article in question is a **sub-article** of a composite article, i.e., if it is *not* at the top planning level. This applies, for example, to the following cases:

- An *Add* interactor for attaching a neighbor element in the context of a planning group.
  (see also section 3.2)
- All *Delete* interactors!
  (see also section 4.4)

A typical resp. recommended solution for this requirement is described in section 2.3.

Furthermore, it must be considered that some applications have the function "***Split up article***"[13], during the execution of which a composite article is split up into its components, i.e., individual articles at top planning level.

If the composite article is the special case of a **Meta type** instance, it will no longer exist after executing the function "Split up article"!
One consequence of this is that meta properties can no longer be accessed after splitting up. Accordingly, interactors with *PropEdit* or *PropChange* actions that use meta properties then must no longer be displayed! If an OAP type exclusively has such interactors, this can be achieved in a simple way by assigning the OAP type (exclusively) to one (or more) meta types (ID) via mapping table *Metatype2Type*[14].

[13] or similar, e.g., "Break up article"
[14] i.e., there should/must not be an assignment to articles (mapping table *Article2Type*)

4 Actions

4.1 Creating articles via *xOiCreateArticle()*

In cases where it is not possible/sufficient to use an action of type *CreateObj* to create articles, and the article creation needs to be programmed, it is recommended to use the global functions *xOiCreateArticle()* resp. (as of the application releases in fall 2023) *xOiCreateArticle2()*[15].

In default mode *@extended*, the functions simulate the standard process of article creation in the applications of EasternGraphics, where the so-called *checkAdd()* mechanism is used to determine position and rotation of the new article instance (including the creation of a temporary article instance).

If position and rotation of the new article instance are defined during the programmed article creation itself, the functions should be used in mode *@lite*, as this offers a better performance. Then, after calling the function position and rotation have to be assigned to the article instance returned by the function.

Like actions of type *CreateObj*, functions *xOiCreateArticle()* and *xOiCreateArticle2()* require the specification of a parent article (father object) and, thus, are not suitable for insertion at the top planning level.

4.2 Change direction in *DimChange* actions

As of OAP 1.2, in field *Dimension* in table *DimChange* can be specified in which axis direction a change is permitted. Previously, only a change in the positive direction of the respective dimension axis was possible (implemented). To ensure downward compatibility, for OAP data that is created in accordance with an older OAP version, values `X`, `Y` and `Z` are treated in the same way as `PX`, `PY` or `PZ` in accordance with OAP 1.2.

With dimensions X and Z, in most cases it makes sense to allow a change in both directions (i.e., to specify values `X` resp. `Z` as of OAP 1.2) in order to offer the user an optimal change behavior regardless of the rotation of the object or the camera perspective. If necessary, existing data should be ported to OAP 1.2[16].

4.3 Nesting of method calls

Since the arguments for a method call (table *MethodCall*) can be specified as expressions, it is possible to realize nested method calls by means of the expression *methodCall()*.

Example:

```
MC_METHOD_1;Instance;::foo::bar::aClass;method_1;methodCall("AC_call_METHOD_2")
```

[15] In the XOI documentation [xoi] these functions are found in section "xOiFuncs" in sub-section "Miscellaneous utilities".
[16] considering new fields *Separate* and *ThirdDim* in table *DimChange*

4.4 Notes on *DeleteObj*

Of the currently supported object categories, only *Self* and *MethodCall* can be used for the target object of a *DeleteObj* action[17].

MethodCall is useful in the context of interactors, which are bound to planning groups (see section 5). Then the method call can be used to determine the element of the planning group that is to be removed.

When using ***Self***, after the *DeleteObj* action no further actions may follow, which again need a target object or an object definition as a parameter, because after the action the original active object no longer exists!
(This applies not only to *DeleteObj* actions, but generally to all actions that lead to the removal of the active object.)

This is problematic if, after removing the object, further actions have to be executed that perform certain checks and, if necessary, subsequent treatments[18]. Such a task typically exists in the context of planning groups. If, after all, the *DeleteObj* action cannot be executed in the context of an interactor bound to the planning group[19], there is no alternative but to replace the *DeleteObj* action with a *MethodCall* action for the planning group instance (object category *ParentArticle*) that deletes the element *and* performs the subsequent examinations and treatments. The element to be deleted is passed in a parameter using the placeholder `$SELF`:

```
public func delElem(pObj)
{
    self.remove(pObj);
    // do further checks and act accordingly
    ...
}
```

Note:
If the checks to be performed relate to the neighborhood relationships of the layout elements of the planning group, an OAP data creator might think of calling certain methods for manipulating the internal layout structure of the planning group instance prior to the *DeleteObj* action in order to obtain desired results in the neighborship tests (exclusion of the element to be deleted). However, this is error-prone, since this requires exact knowledge of the processes in the base classes. Since a method for performing the checks has to be created in the (derived) class of the planning group anyway, it then can/should better be extended as described above.

[17] *ParentArticle* and *TopArticle* are not possible because the active object cannot delete a superior article instance.
[18] and these actions cannot be performed before the *DeleteObj* action
[19] and, then, object category *MethodCall* is used to determine the element to be removed

4.5 RG properties

OFML properties generated for OCD properties with scope **RG** can only be changed using actions of types *PropChange* and *PropEdit* if value `1` is specified for option *@NeedValuesForRGProps* in control data table `epdfproductdb`[20].

Caution:
An assignment of a value to an RG property made possible via *@NeedValuesForRGProps* should not lead to a value change of a configurable property that follows the RG property in the list of properties (due to dependencies in OCD product relationships)! Otherwise, problems ("freezing") can occur when the article instance is re-created (e.g., as part of an update)[21].

4.6 Action type *NoAction*

An interactor is not displayed if it currently has no valid action. This may be a handicap in an early phase of project development, when, initially, only the interactor symbols are to be created and positioned. In this case, special dummy action type *NoAction* can be used, which does not require any further data to be specified.

Example:

```
AC_DUMMY;;NoAction;;
```

[20] With this, the value choice list of the generated property contains all the values that are stored for the RG property in the OCD value table. Without the option (or with value `0`) the value list contains only the current value, so the property is practically read-only.
[21] as RG properties are not encoded in the variant code

5 Planning groups

The explanations in this section refer to the classes from the OFML base library XOI used in OAP projects to implement planning groups[22].

Useful methods of these classes that can be used directly in OAP via *MethodCall* actions are described in the document [methods]. This document also contains a brief introduction to the purpose and application of the planning classes.

Control data tables `customplgroup`, `jointplgroup`, `layoutgroup` resp. `tabularplgroup` can be used to control some aspects of the behavior of planning groups. The possible options are described in Application Note AN-2006-01 [an0601].

Note on the use of *xOiJointPlGroup* vs. *xOiLayoutGroup*:
In principle, a pure left-right-oriented joint planning can be realized by means of *xOiLayoutGroup*[23]. However, in most cases the use of *xOiJointPlGroup* is the more "light weight" solution for such planning groups and thus preferable. Certain requirements, which go beyond a pure left-right-oriented joint planning, within limits also can be implemented based on *xOiJointPlGroup*, see sections 5.6 and 5.7.

5.1 Using object category *MethodCall*

In particular, in connection with planning group classes, there are some useful applications for object category *MethodCall*. A common scenario is, for example, that a certain property has to be set on a group element created via a *CreateObj* action. In this case, the target object for the according *PropChange* action cannot be determined by means of object category *Self* (since this refers to the object to which the interactor is bound whose actions currently are being executed). However, if the group element has been added to the last layout element of the group, in the case of class *xOiJointPlGroup*, e.g., method *lastObj()* can be used for that purpose. In the classes *xOiLayoutGroup* and *xOiCustomPlGroup* there are equivalent methods[24].

For the example above, the *PropChange* action could be implemented as follows:

oap_object.csv:

```
OB_PARENT;ParentArticle;;;
OB_LAST_GROUP_OBJ;MethodCall;AC_call_LAST_OBJ;;
```

oap_action.csv:

```
AC_PC_CHANGE_A_PROPERTY;;PropChange;PC_CHANGE_A_PROPERTY;OB_LAST_GROUP_OBJ
...
AC_call_LAST_OBJ;;MethodCall;MC_LAST_OBJ;OB_PARENT
```

oap_methodcall.csv:

```
MC_LAST_OBJ;Instance;::ofml::xoi::xOiJointPlGroup;lastObj;
```

[22] These are currently the classes *xOiCustomPlGroup*, *xOiJointPlGroup*, *xOiLayoutGroup* and *xOiTabularPlGroup*. They are described in detail in the XOI documentation [xoi] in section "Planning Groups".
[23] In this case, the group would consist of only a single branch.
[24] see document [methods]

Note:
A method that is used in object category *MethodCall* has to return an instance that represents an article[25]. This requirement may not be met for special auxiliary objects within planning groups. Then, the solution is not to execute the action directly on this auxiliary object, but to call a method on the group instance (action type *MethodCall*), passing the auxiliary object as an argument to this method by means of a *methodCall()* expression[26] (see section 4.3).

5.2 Changing dimensions of group elements

5.2.1 General

Classes *xOiCustomPlGroup*, *xOiJointPlGroup* and *xOiLayoutGroup* support the modification of the dimension (width/depth) of topological[27] elements of the planning group: connected elements (to the right side[28]) are moved away resp. closer (to re-connect).

For that, the so-called *InsertMode*[29] has to be set to `1` (or greater)[30].

This is done via an entry

```
@InsertMode;;1
```

in the corresponding control data table.

Furthermore, in the article classes of the planning group elements, the handling of a dimension change by the planning group has to be triggered by calling method *dimensionChanged()* on the planning group instance[31].

Typically, a dimension change occurs when the value for certain properties is changed. Accordingly, these properties have to be handled separately in the method *propsChanged()* [property].

In the case of an article class derived from *GoMetaType*, assuming that a change of property *@WidthProp* results in a change in width, a prototypical implementation of this method using the planning group class *xOiJointPlGroup* would look like this:

[25] The technical background for this restriction is that the clients of online apps perform some actions themselves (i.e., do not delegate them to the server), but in turn only "know" articles (no OFML instances).
[26] The same *MethodCall* action is used here that would also be used for object category *MethodCall*.
[27] elements, defining the layout of the group
[28] In the case of *xOiLayoutGroup* and *xOiCustomPlGroup*, the side of the changed element to be adjusted is defined by a parameter of method *dimensionChanged()*.
[29] defined in the base class *xOiPlGroup*
[30] For *xOiCustomPlGroup*, this requires (default) value `1` for option *@StoreNeighborship*.
[31] For details (parameters) see [xoi]. In case of *xOiJointPlGroup* method *dimensionChanged()* is implemented and documented In base class *xOiLRPlGroup*.

```
public func propsChanged(pProps, pCheck)
{
    var tRet = GoMetaType::propsChanged(pProps, pCheck);

    if (!tRet) return (tRet);

    // Do nothing during meta type initialization!
    // (This would cause syntax errors due to the lack of some properties.)
    // Skip this, if the class is not derived from GoMetaType!
    if (!isMetaInitialized() || sInSetAddStateCode)
        return tRet;

    var tFather = self.getFather();

    if (pCheck && tFather.isA(xOiJointPlGroup) &&
        pProps.find(@WidthProp) >= 0 &&
        !tFather.dimensionChanged(self))
        tRet = 0;

    return(tRet);
}
```

For Meta properties that raise a native property of the encapsulated article instance to the level of the Meta type instance (so-called *na-properties*), this does not work because a change of such a Meta property is delegated from the Meta type instance to the encapsulated article instance. Thus, in this case, method *propsChanged()* would have to be overwritten accordingly in the class of the encapsulated article instance. Since in standard OFML data creation normally the standard type *OiOdbPlElement* is used, it would therefore be necessary to derive a specific class from it and assign it to the corresponding articles in the OAM data. Section 5.3 describes a different approach that allows to avoid overwriting *propsChanged()* in the class of the encapsulated article instance by means of an extended Meta data creation.

5.2.2 Considering changes in the bounding volume

The algorithm for handling a dimension change is based – amongst others – on a comparison of current and stored minimum axis-orthogonal bounding volume of the changed element relative to its local coordinate system[32]. The bounding volume is stored by the planning group instance after an element has been inserted and whenever the above-mentioned methods are called. If a change of the bounding volume takes place without calling any of the methods mentioned above, this will lead to incorrect handling in subsequent calls of the methods, because the planning group instance has not noticed anything about the interim change of the bounding volume!

This has to be considered in situations where the bounding volume changes, but the actual dimension of the element does not change and, thus, actually no neighbors need to be moved. An example would be creating a frame foot as a child of a table placed midway under the left neighbor table and the modified table. In order to avoid the problem above, the methods for handling a dimension change should be called anyway, even if this means a small overhead. However, if such a change takes place exclusively during the insertion of an element into the planning group, the call can be omitted. (In the example above, this would be the case, if the foot is generated in the course of the execution of the actions of an *Add* interactor by means of a *PropChange* action that is executed after a *CreateObj* action that actually performs the element's creation.) But then, the planning group instance has to be assisted by other method calls, where different methods must be used for the two above-mentioned classes:

[32] according to method *getLocalGeoBounds()* of OFML interface *Base* [ofml]

- *xOiJointPlGroup*:
  Call of *elementCreated()*[33] after the relevant *PropChange* action
- *xOiCustomPlGroup* and *xOiLayoutGroup*:
  Call of *delayHandleNewElement()* before the *CreateObj* action and *handleNewElement2()* after the relevant *PropChange* action

For details on these methods, see XOI documentation [xoi].

While methods *elementCreated()* and *delayHandleNewElement()* have a simple signature and thus can be integrated directly without special programming by means of a *MethodCall* action[34], in the case of *handleNewElement2()* the encapsulation by a specially programmed method probably is the preferred way.

5.2.3 Notes on the algorithm in *xOiJointPlGroup*

In cases were the origin of the local coordinate system (also) has changed, method *dimensionChanged()*, when indicated, also performs a re-positioning of the changed element before (possible) re-positioning of relevant neighbor elements. An example would be attaching a mounting plate on the left side of a table. Re-positioning of the changed element is also performed if the coordinates of the relevant attach point change accordingly.

In the example with the left mounting plate, therefore, it is assumed that the attach point on the left side of the modified table "migrates" to the left in accordance with the width of the mounting plate.

Another scenario would be the example already mentioned above (in section 5.2.2) of creating a frame foot centered below two adjacent tables. If *dimensionChanged()* is called in this case[35], the coordinates of the attach point on the left side of the table to which the foot is added must not change. Otherwise, there would be an undesirable shift of the changed table!

The re-positioning of the changed element ideally takes place on the basis of the attach point that was used when the changed element was attached to (the right of) its left neighbor. In the case where the left neighbor of the changed element was added to (the left of) the changed element, this attach point is not known (with the standard implementation in *xOiJointPlGroup*/*xOiLRPlGroup*). As a fallback, the changed element then is translated by the amount of the change in the local origin.

Especially with angled or curved elements, the latter can lead to an undesired repositioning of the changed element if *dimensionChanged()* is called as a result of a change in the orientation of the element (in order to accomplish the repositioning of the neighboring elements). This can be prevented by delivering the relevant attach point via overwriting method *getUsedAttPt()*[36]. In the ideal case, where the elements of the planning group are attached to each other with the help of an attach point each on the left and right side and these attach points are named the same for all elements, the overwritten implementation would look something like this:

[33] inherited from base class *xOiLRPlGroup*
[34] In the case of *elementCreated()*, the argument itself also is determined by calling a method (*lastObj()*) via function *methodCall()*, see section 4.3.
[35] see alternative, described in section 5.2.2
[36] The method is defined and documented in the base class *xOiLRPlGroup*.

```
protected func getUsedAttPt(pRefObj, pNeighbor)
{
    var tRet = xOiJointPlGroup::getUsedAttPt(pRefObj, pNeighbor);

    if (tRet == NULL) {
        if (pNeighbor == self.neighbor(pRefObj, @R))
            tRet = @AP_R;
        else
        if (pNeighbor == self.neighbor(pRefObj, @L))
            tRet = @AP_L;
    }

    return(tRet);
}
```

5.3 Dimension change based on *na*-Metaproperties

This section describes a Meta data approach that can be used to avoid overwriting method *propsChanged()* in the class of the encapsulated article instance, as described in section 5.2.1, in the case of a dimension change that is based on a change to a *na*-Metaproperty.

Suppose there is a *na*-Metaproperty `GWidth`, which specifies the width of the article in centimeters:

1.
In table *go_types*, create an ***invisible*** utility property of type (format) "ch" controlling sub-items (mode 24):

```
MT_SE_Name;GWidth;na;60;1;
MT_SE_Name;GWidth2;ch;_60;24;
```

2.
The implementation of *propsChanged()* in the class of the Meta type instance (see section 5.2) uses property key `@GWidth2` (instead of `@GWidth`).

3.
In table *go_childprops*, specify a parameter set for the possible values of the utility property:

```
CHP_SE_GWidth2;GWidth2;_60;
CHP_SE_GWidth2;GWidth2;_70;
...
```

(Before, the parameter set, here CHP_SE_GWidth2, was assigned to the articles of Meta type `MT_SE_Name` in table *go_articles*. Alternatively, an already assigned parameter set can be used.)

Tip:
It is enough to specify just one value (e.g. the start value). This is particularly helpful when there are many possible values.

4.
In file `go_context.ofml`, implement a function converting a value of property `@GWidth` into a value of property `@GWidth2`:

```
func gwidth2gwidth2(pVal)
{
    return(Symbol("_"+String(pVal)));
}
```

5.
In table *go_actions*, add an according action:

```
MT_SE_Name;GWidth;;CON;;SET_PROP;GWidth2;gwidth2gwidth2(GWidth);
```

5.4 Group-specific entries in the control data table

If several group classes derived from an XOI base class exist in a series and different specifications have to be made for these in the respective control data table, the second field (argument) of the table entries can be used for differentiation, provided that (different) articles have been created for the different planning group types.

Example:

```
@StartElement;[@Article,["SINGLE_WP"]];[@foo_bar,"ARTICLE1",@VarCode,"",[]]
@StartElement;[@Article,["DOUBLE_WP"]];[@foo_bar,"ARTICLE2",@VarCode,"",[]]
```

Details see Application Note AN-2006-01.

5.5 Changing articles via *replaceElement()*

Edit interactors are often used to change the article of the object. Ideally, a corresponding property is directly addressed by means of a *PropEdit* action. Sometimes, however, there is no suitable property available. In this case, one can use method *replaceElement()* implemented in the classes *xOiJointPlGroup* and *xOiLayoutGroup* resp. method *replaceField()* in class *xOiTabularPlGroup*, calling it by means of a *MethodCall* action.

For details about using these methods see document [methods].

Notes:

- If the interactor is bound to a group element (not to the planning group instance), after the *MethodCall* action with the call of *replaceElement()* no further actions may follow, which again need a target object or an object definition as a parameter, because after the action the original active object no longer exists! (See also notes on *DeleteObj* in section 4.4.)
- The implementation of *xOiLayoutGroup::replaceElement()* currently does not work if a neighbor of the element to be replaced belongs to a different branch.

5.6 Non-Layout articles with *xOiJointPlGroup*

Occasionally elements have to be inserted in an instance of *xOiJointPlGroup* (or any derived class) that are not part of the topological list, e.g. frames, service tables etc.

Regarding such elements a few hints:

- For elements of the planning group that should not be part of the topological list[37], hook method *isValidForLRPlanning()*[38] has to return value `0`. Accordingly, this method has to be overwritten specific to the project.
  It should be noted that this method is called immediately after element creation, i.e. at a time when the article is not initialized yet. As a consequence, the article number or properties of the instance cannot

[37] This corresponds roughly to the distinction between layout elements and other elements in *xOiLayoutGroup*.
[38] This method is defined in base class *xOiLRPlGroup*.

be used in the method to decide whether or not the given element should be part of the topological list. All that remains is the ability to test for a type using *isA()* or for a category using *isCat()*[39].

- If these elements are to be treated as sub-articles, value `1` has to be specified for option *@AllElements4SubArticle* in control data table `jointplgroup`.
- If these elements are placed by means of attach points, by calling method *xOiJointPlGroup::isBusyAttPt()* in validity conditions of interactors it is possible to determine whether an element from the topological list has a neighbor element at the relevant attach point[40].
  (For details about using this method see document [methods].)
- When changing the dimensions of elements from the topological list (see section 5.2), elements that are not contained in this list are not moved! Therefore, these elements either have to be re-created or class *xOiLayoutGroup* has to be used.

5.7 Additional start elements

Typically, options *@StartElement* or *@StartLayout* in the control data tables are used to initially create one or several start (layout) elements. Occasionally, it may be necessary to initially create also some non-layout elements.

Because this initialization occurs when an article number is assigned to the planning group instance during method *setArticleSpec()*, this method has to be overridden accordingly in derived classes in order to create additional start elements. In the overridden method, first the inherited implementation has to be called. After that global function *xOiCreateArticle()* (see section 4.1) can be used to create further r elements. If required, the reference objects for the additional elements to be created can be determined using the methods for accessing the topological list (*xOiJointPlGroup*[41]) resp. for accessing the layout structure (*xOiLayoutGroup* and *xOiCustomPlGroup*).

Pay attention also to the notes in section 5.10.3 and, in the case of a class derived from *xOiJointPlGroup*, the notes in previous section on creating elements that should not be part of the topological list!

5.8 Removability of group elements

Since the removal of an element from a planning group often is accompanied by further actions (see section 4.4), usually they should be able to be removed only via OAP interactors (symbol type *Delete*), but not via the Delete command of the application. Previously, this explicitly had to be done by directly or indirectly calling the method *setCutable()* (OFML interface *Base* [ofml]) with the value `-1`. This is no longer necessary: The desired state can be specified for all planning group classes, separately for layout elements and other elements (see 4.7), in the respective control data table using options *@CutableState4Layout* and *@CutableState4Other*.

[39] If for some reason this is not possible or not sufficient, class *xOiLayoutGroup* has to be used.
[40] The methods for determining neighbor elements in the topological list do not work here.
[41] via base class *xOiLRPlGroup*

5.9 Common properties / Group properties

With all group types, option *@CommonProps* in the respective control data table can be used to specify the properties of the elements, which can be edited together at the level of the planning group. Then, a change at group level is delegated to all group elements that own the changed property.

There are a number of accompanying options that affect the processing of common properties. These are described in section 6.1 of Application Note AN-2006-01 [an0601]. Some of these are also discussed in the following (sub) sections.

Beyond that – if necessary, in addition to the properties generated via *@CommonProps* – in derived, project-specific classes group properties also can be programmed using OFML interface *Property* [property], see section 5.9.4.

5.9.1 Creation and update of common properties

For which of the properties, specified in option *@CommonProps*, a group property is actually created depends on the group elements used for generating the group properties and on their current configuration. This can be controlled or influenced by the following options:

@AllObjs4CommonProps
@NonLayout4CommonProps,
@Meta4CommonProps
@CommonPropsDepth
@NonVisibleProps4Common
@ROPropsEditable4Common

The initial creation of the common properties is done during the initialization of the group instance after the creation of the initial layout elements[42].

An automatic **update** (re-creation) takes place in case of the following events:

1. When a common property is changed and this has affected at least one group element[43].
   This reacts to possible dependencies between the common properties.
2. When opening resp. updating the property editor for an OFML instance – via method *updateProperties()* of OFML interface *Property*[44] – if the language to be used for product data of the series has changed since the last opening resp. updated. This reacts to a changed language setting on the part of the user.
   As of XOI 1.60 (with the releases in fall 2023), the update is generally also performed during the first call of *updateProperties()* after calling *setAddStateCode()*[45]. This reacts to possible changes in the product data, e.g., changed choice lists in the relevant properties

[42] usually in method *setArticleSpec()*
[43] via method *fixPropsChanged()*
[44] In the overridden implementation in base class *xOiPlGroup*
[45] in the course of restoring an article instance on the basis of a saved basket representation

3. If option *@AllObjs4CommonProps* has value `1`:
   When creating a group element using a *CreateObj* action or using global function *xOiCreateArticle()* in *@extended* mode (see section 4.1) and when removing a group element[46].
   This covers properties that are relevant only for the new element or the element to be deleted.

In certain situations that are not covered by the standard events mentioned above, the update must be triggered from the OFML resp. OAP data. For this purpose, method *updateCommonProperties()* is implemented in the XOI classes for all group types (see document [methods]). Typical use cases are:

- A property has been changed that affects the visibility or the state of common properties, where the changed property itself is not included in the set of common properties.
- A group element is created or removed[47] that has properties listed in the common properties, but the creation or removal is not covered by the standard situation 3 described above.

The method can and should be called directly in the OAP data by means of a corresponding *MethodCall* action after the relevant actions. (Thus, no special programming is required for this.) However, if an element is removed by an action triggered by an interactor of the element itself, this is not possible (see section 4.4). In this case, consider using an interactor of the parent article to remove the element.

5.9.2 Common properties for a new element

When a new element is inserted into the group the element gets the configuration defined by the respective action (as long as there is no Metadata-driven inheritance of property values). This configuration can differ from the current settings of the common properties of the planning group. If the current settings of the common properties of the planning group are to be adopted for the new element, method ***assignCommonPropValues()***[48] has to be called on the planning group instance in an additional action after the action that creates the element.

(Whether actually the current settings of the common properties are to be adopted for the new element, or rather a Metadata-controlled inheritance of property values of the reference object has to be carried out, is to be determined on a project-specific or situation-specific basis[49].)

Method *assignCommonPropValues()* expects as an argument the element to which the procedure is to be applied. In the considered scenario this is the element which was created by means of the *CreateObj* action. The best way to specify this element is to apply object category *MethodCall*, using the appropriate methods to access the neighbor element of the reference object defined by the *CreateObj* action (see also section 5.1).

For example, if the *CreateObj* action is executed in the context of an *Add* interactor bound to the selected group element (active object), and if the active object also is the reference object of the *CreateObj* action, methods *neighbor()* (*xOiJointPlGroup*) resp. *getNeighbor()* (*xOiLayoutGroup* and *xOiCustomPlGroup*) could be used, where in the first argument the active object is passed by means of placeholder `$SELF`, and the second argument specifies the attach direction (*xOiJointPlGroup*) resp. the attach point used in the *CreateObj* action (*xOiLayoutGroup* and *xOiCustomPlGroup*).

[46] via events of types *@ArticleInserted* resp. *@ElementRemoval*
[47] e.g., as a result of a property change
[48] The method is equally defined and implemented in all classes. It is described in the document [methods] (as well as the both methods mentioned below).
[49] Property values of the reference object can deviate from the common properties if the properties of the reference object were changed again after the last change of the common properties.

5.9.3 Property groups

By default, when generating the common properties (option *@CommonProps*), the property classes are not adopted from the elements and no specific class is assigned either. Thus, in the property editor, the generated common properties appear in the standard group, i.e. “Article” or “Other”, if additional programmed properties (see next sub section) are assigned to their own class.

In most projects this behavior is sufficient. However, in the following situations, a class assignment for the generated common properties can be useful or even necessary:

1. With a large number of common properties, classes/groups can be used to increase the clarity for the user.
2. If the common properties are drawn from different object types and these each have disjoint sets of relevant properties:
    a. it makes sense to make this visible to the user as well[50]
    b. separate property groups are necessary if (different) properties of the different object types have the same language-specific name[51]

In the simplest case, the class assignment for the generated common properties is done by setting value `1` for option ***@Classes4CommonProps***[52]. Currently, the classes are adopted from the group elements for all generated common properties. If this is not desired, or if specific classes are to be assigned in connection with additional programmed properties (see next sub section), this must be done in derived classes using method *setPropClass()* of OFML interface *Property* [property].

In both cases, language-specific text resources (`.sr` files) that correspond to the names of the property classes also should or must be created in the series of the planning group article.

5.9.4 Programmed group properties

In many projects, users should be able to control some general planning features of the group, e.g. the maximum permitted dimensions. For this purpose, corresponding properties have to be programmed in the project-specific derived group class. Usually this is done in addition to the automatically generated common properties (option *@CommonProps*).
Occasionally, properties also have to be programmed in the project-specific class because the properties of the group elements do not provide enough functionality or they cannot be adopted one-to-one for the group[53].

Here are some hints and tips for programming group properties:

1. Since the list of keys of the programmed properties is needed in several places of the implementation of the group class, it is recommended to define a corresponding static class variable, e.g.:

    ```
    static var sMyPropKeys = @(@Foo, @Bar);
    ```

2. As already described in Application Note AN-2006-01 for option *@CommonProps*, also the programmed properties should be specified in table `non_pd_properties` for better performance.

[50] It is then easier for the user to see resp. understand that the change of a property from a given property group only affects certain elements of the planning group.
[51] and, thus, cannot be distinguished by the users
[52] The prerequisite for this is, of course, that suitable property classes are assigned in the OFML data of the group elements.
[53] This is often the case when the OAP project is built on top of existing OFML data. In projects where the OFML data and the OAP data are built together, care should be taken from the outset that the properties of the group elements are created in the OFML data already in such a way that specific programming of properties in the group class is avoided

3. Base class *xOiPlGroup* for all planning group types implements some standard handlings (e.g. regarding persistence) for the properties of a group instance, the keys of which are supplied by hook method *getFixProperties()*. Thus, as a rule, this method should be overridden in the project-specific group class and supply the keys of the programmed properties, if necessary, in addition to the keys of the generated common properties supplied by the inherited implementation.

   Example see section 5.10.1.

4. The methods of the new *Property* interface [property] are recommended for defining the properties. (The language to be used for the names of the property and the possibly existing choice list values is determined by means of method *getPDLanguage()*[54].)

5. If the programmed properties are created in addition to the automatically generated common properties (*@CommonProps*), it must be decided whether the programmed properties should be displayed in the property editor ***before*** or ***after*** the generated common properties. By default, the generated common properties are created in the property list starting from position 1[55].
   - If the programmed properties are to follow afterwards, they must be assigned a correspondingly high position number.
   - In the other case, option *@FirstPos4CommonProps* has to be used to specify a position number high enough for the first generated common property.

   However, with the help of option ***@CommonPropsPos***[56], a mixed order can also be realized!

6. The initial definition of the programmed properties is done in the overridden method *setArticleSpec()* after calling the inherited implementation and, if necessary, after generating additional start elements (see section 5.7).

```
public func setArticleSpec(pSpec)
{
    <BaseClass>::setArticleSpec(pSpec);

    initMyProps();
}
```

   To avoid overhead (performance) and side effects, the initial values are not assigned using *setPropValue()*, but by setting the value in the table of dynamic properties (method getDynamicProps() from OFML interface *Base* [ofml]) or by calling the corresponding *set* methods.

[54] For details see [property]
[55] in the order according to option *@CommonProps*
[56] As of XOI 1.60 (fall 2023)

7. The handling of a change of a programmed property by the user is done in the overridden method *fixPropsChanged()*, if necessary, after calling the inherited implementation, which performs the standard handling for generated common properties (see also section 5.9.1):

```
protected func fixPropsChanged(pProps, pDoChecks)
{
    var tRet = <BaseClass>::fixPropsChanged(pProps, pDoChecks);

    var tP;
    foreach(tP; pProps) {
        if (sMyPropKeys.find(tP) < 0) continue; // not my cup of tea

        // handle change of my property tP
        ...
    }
    return(tRet);
}
```

8. If the user changes the **language setting** in the application at runtime and the group instance is selected at this point (open property editor), the language-specific names of the programmed properties including their values and classes have to be changed[57].
   This is done by overriding method *updateOtherFixProperties2()*[58]. This hook method is called by the standard implementation of method *updateProperties()* in base class *xOiPlGroup* if the language to be used for the product data of the series has changed since the last call (see also section 5.9.1, point 2). At this time, the names for the generated common properties have already been changed by the base class. So, these do not have to be handled in *updateOtherFixProperties2()*[59].

   If the programmed properties are created using method *setupProperty2()* of the *Property* interface (see point 4 above), a call of *setPropName()* and, if necessary, a call of *setPropChoiceList()* (if no text resources are used for the values) is sufficient to adapt the texts. If the properties are created using method *setupProperty()* of the old *Property* interface, they have to be re-created using this method. In both cases, the language to be used is determined by means of method *getPDLanguage()*[60].

9. If the choice lists and, if necessary, other attributes of properties of group elements are used in the definition of programmed properties, it is necessary to react to possible changes in the product data.
   This is also done by overwriting method *updateOtherFixProperties2()*[61]. This hook method is called, in addition to the situations described in point 8, by the standard implementation of *updateProperties()* in base class *xOiPlGroup*, if it is the first call after the call of *setAddStateCode()* (see also point 2 in section 5.9.1). At this time the common properties have already been re-created by the base class[62].

   In contrast to the situation from point 8 - method *updateOtherFixProperties2()* has parameters, by which the triggering situation can be recognized - here the simplest and safest solution probably is to completely redefine the concerned programmed properties (see point 6).

[57] This also applies if a saved group instance will be opened for reconfiguration and a different language is set in the application than at the time of saving.
[58] As of XOI 1.60 (fall 2023). Before that, hook method the *updateOtherFixProperties()* was used. This is still supported due to backward compatibility, but is considered obsolete.
[59] Hence the "Other" in the method name.
[60] For details see [property]
[61] As of XOI 1.60 (fall 2023)
[62] So, these do not have to be handled in *updateOtherFixProperties2()*.

5.10 Persistency

The XOI planning group classes store all the information that is necessary to reconfigure the planning group. Essentially, this is done in the methods *getAddStateCode()* and *setAddStateCode()* of the OFML interface *Article* [article]. When implementing derived classes, the following aspects must be taken into account so that the inherited behavior continues to function or to avoid overwriting the two mentioned methods.

5.10.1 Properties

If own properties are programmed in the derived class, the keys of these properties must be supplied via overridden method *getFixProperties()*. If these properties are defined in addition to the properties that are generated by the base class according to option *@CommonProps*, this must be taken into account accordingly:

```
protected func getFixProperties()
{
    var tMyProps = @(@Foo, @Bar);
    var tRet = xOiCopyAggr(<BaseClass>::getFixProperties(), NULL, 0);

    xOiCopyAggr(tMyProps, tRet, 0);

    return(tRet);
}
```

5.10.2 Member variables

If information is stored in member variables that is required for a re-configuration, and if this information cannot be restored in any other way, the values of these member variables have to be encoded in the so-called *AddStateCode* (*getAddStateCode()*), and parsed from the *AddStateCode* and assigned again during restoration (*setAddStateCode()*).

To avoid the rather complex overwriting of the two methods, all XOI planning group classes offer a mechanism that is based on method ***getAddStateMembers()***: For all member variables whose names (String) are supplied by the method (return value is of type *Vector*), the XOI base class takes the encoding in the *AddStateCode* and the restoration from it.

Regardless of whether the encoding is done via *getAddStateMembers()* or not, it should be noted that the member variables to be encoded **must not contain any object references**!

There are two approaches to circumvent this limitation:

1. The two methods mentioned above are overwritten: in *getAddStateCode()*, the object references are converted into storable information (e.g. object name, see below) before calling the inherited implementation, and in *setAddStateCode()* the object references accordingly are restored after calling the inherited implementation.
2. No object references are stored in the relevant member variables, but rather an information from which the respective object reference can be determined if necessary (e.g. object name, see below).

Caution: When using object names, the complete (hierarchical) object name must not be used, only the local name within the planning group instance!

Methods *localObjName()* and *localName2Obj()* are available in all planning group classes to implement these approaches. The methods are specified and implemented in the base class *xOiPlGroup*.

As an alternative to the (local) object name, the index of the instance can be saved under which the instance is managed in the list of elements of the planning group instance (see method *getElements()* of OFML interface *MObject* [ofml]).
If a list of object references has to be saved, global XOI functions *xOiElRefs2ElIdcs()* and *xOiElIdcs2ElRefs()* can be used[63]. For an example how to use these methods see below.

Object references in member variables (currently) also have to be observed and handled in the OFML **persistence rules**[64]!

Assuming there is a member variable *mMyEL* which contains a list of object references, the persistence rules could be implemented as follows:

```
rule START_DUMP(pArg)
{
    mMyEL = xOiElRefs2ElIdcs(self, mMyEL);

    return(0);
}

rule FINISH_DUMP(pArg)
{
    mMyEL = xOiElIdcs2ElRefs(self, mMyEL);

    return(0);
}

rule FINISH_EVAL(pArg)
{
    mMyEL = xOiElIdcs2ElRefs(self, mMyEL);

    return(0);
}
```

5.10.3 Additional start elements

According to section 5.7, overridden implementations of method *setArticleSpec()* can create further elements in addition to the start elements that are created in the inherited implementation according to options *@StartElement* resp. *@StartLayout*.

In order for the restoration to work using method *setAddStateCode()* of the base class, the following has to be taken account of:

- If deriving from *xOiJointPlGroup* and *xOiLayoutGroup*, it must be ensured that overridden method *elRemoveValid()* does not prevent the removal of elements created in *setArticleSpec()*.

- If deriving from *xOiTabularPlGroup* and *xOiCustomPlGroup* , no additional elements may be created in *setArticleSpec()* if the method is called in the course of the restoration of the planning group for re-configuration. The distinction is made using method *xOiBasePlanning::getCreationMode()*[65]:

[63] see section "xOiFuncs", sub-section "DUMP and EVAL rule supporting functions" in the XOI documentation [xoi]
[64] The basket implementation (OBK) used in the applications writes a dump of the article instance to the cut buffer when an article is deleted or copied.
[65] For details see XOI documentation [xoi], section „Main Planning Classes“

```
public func setArticleSpec(pArtNo)
{
    xOiTabularPlGroupVert::setArticleSpec(pArtNo);

    if (oiGetPlanning().getCreationMode() == 0) {
       // add some further elements:
       ...
    }
    // else: nothing to do during re-creation
}
```

5.11 Overriding methods of XOI base classes

Occasionally, before or after a method of the planning group instance is called via an OAP action of type *MethodCall*, further treatments must be carried out. Ideally, this should be done via additional preceding resp. succeeding OAP actions[66]. If this is not possible for certain reasons, or if it is more efficient to do all in one method call, the method in question should not be overwritten directly in the derived class (in order to implement there the additional treatments), as this can hinder the further development of the used XOI base class[67]. Instead, an own method should be defined and implemented in which the relevant method of the XOI base class is called at a suitable point!

```
public func myXoiMethod(...)
{
    // do something special
    ...

    xoiMethod(...);

    // do something other
    ...
}
```

5.12 Managing the property state of group elements

Often the task is that certain, normally editable properties of an article instance should not be editable (or not visible at all) if the article instance is an element of a planning group. For example, it may be required that a material property, e.g. *@Color*, which is raised to the planning group level using option *@CommonProps* (see section 5.9), should not be editable for individual elements of the planning group in order to guarantee a uniform look of the planning group.

Note:
If a property of the group elements is made not editable or not visible with the approaches described in this section, but should be visible and editable as part of the *@CommonProps* on the level of the planning group, option *@ROPropsEditable4Common* resp. option *@NonVisibleProps4Common* has to be used!

[66] according to the principle of avoiding OFML programming
[67] e.g. the introduction of additional optional parameters

One approach to this involves the following steps:

- For the start elements of the planning group, also the state of the relevant properties is specified in option *@StartLayout* (or *@StartElement*), e.g.:

  ```
  @StartLayout;;[\
  [NULL, @man_series,"0815",@VarCode,"",[],[[@Color,1]]],\
  [@AP_R,@man_series,"0815",@VarCode,"",[],[[@Color,1]]]]
  ```

  Here, the property state hast to be specified in accordance with method *setPropState2()* of OFML interface *Property* [property]. There, value `1` specifies a visible but not editable property.

- For elements subsequently inserted into the planning group via an OAP interactor, after the *CreateObj* action in the action list of the interactor (or the relevant entries of an *ActionChoice*) an additional *PropChange* action is specified (see also section 5.1), e.g.:

  ```
  PC_DisableColor;Editability;Color;0
  ```

Note that this approach does not require any additional OFML programming. Unfortunately, this approach **does not work** if the property in question is an original, **commercial property** of the article and the article is encapsulated by a **meta type** instance: after a property change triggered by the planning group instance, the property gets its original state again (editable).

Then, the approach described above has to be modified and expanded as follows:

- The commercial property in question is raised to the level of the meta type instance as a so-called *na-property*. (The name of the property by default then changes to *@GColor* and must be changed accordingly in the places mentioned above.)

- For the start elements, method *setArticleSpec()* must be overwritten in a specifically derived planning group class according to the following pattern (for *xOiJointPlGroup*):

  ```
  public func setArticleSpec(pSpec)
  {
      xOiJointPlGroup::setArticleSpec(pSpec);

      var tEl;
      foreach(tEl; self.getElOrder())
          tEl.updateNAPropState(@GColor, 0);
  }
  ```

- For elements subsequently inserted into the planning group via an OAP interactor, in the relevant action list after the *CreateObj* action and the *PropChange* action described above an additional *MethodCall* action must be specified according to the following pattern:

  ```
  MC_Disable_GColor;Instance;::ofml::go::GoMetaTyppe;updateNAPropState;@GColor,0
  ```

  Remark:
  Method *GoMetaType::updateNAPropState()* guarantees a permanent state change of a na-property. There, the status has to be specified according to method *setPropState()* of old OFML interface *Property*, i.e., a property that is visible but cannot be edited here is specified by the value `0`.

In the case of **meta-type-based articles**, the following approach may be **more suitable**, especially if the state of several (many) properties has to be modified[68]. This is often the case with meta types, since, in addition to the above example with the material property, properties for an article exchange often are not useful in the context of the planning group and must be deactivated:

- An auxiliary (invisible) property is created in the meta type, e.g. *In_PGroup*, with possible values `0` (no) and `1` (yes), where `0` is the initial value.
- In table `go_actions`, the relevant properties[69] are deactivated via a CON_PROP action if the condition `In_PGroup == 1` is met.
- In OAP, property *In_PGroup* is set to `1`. (For the start elements this is done in the corresponding option in the control data table. For additional elements inserted via an interactor this is done via a *PropChange* action after the *CreateObj* action in the relevant action list.)

5.13 Reaction to property changes of elements

Often the group instance has to react to changes of certain properties of group elements. In principle, this can be realized with the following two approaches:

1. In the relevant classes of the group elements, method setPropValue() or propsChanged() is overridden and the change of a relevant property is delegated to a corresponding method of the parent group instance.
2. The group instance registers with the global **change manager** (type *OiChangeManager*) for change events of type *@PropertyChange*, triggered by elements of the group, and reacts to them in method *receivePropertyChange()* (example see below)[70].
   (For commercial properties that are raised to the level of an encapsulating meta type instance as so-called *na*-properties, event type *@MetaMainChildPropChange* must be used if necessary[71].)

In most cases, the 2nd approach is preferable, as it does not require any implementations in project-specific element classes[72]!

In addition, the second approach facilitates the solution to the following problem:

If the group contains Meta type instances as elements, during the (re)creation of the group for reconfiguration property changes are triggered by base class *GoMetaType* when processing the saved variant code. These changes are irrelevant for the group instance, since the group has the correct, saved state when the (re)creation process has finished. Alone for reasons of performance, the group instance should not react to these changes. Even more, in certain cases there may be misbehavior or evaluation errors if the group reacts to them, since at this point not all (other) group elements have been completely restored yet!

[68] Option *@StartLayout* and the action lists then would become very long and confusing with the first approach.
[69] For that, commercial properties such as *@Color* also have to be created as na-properties, as in the first approach.
[70] The concept of the global change manager and the currently supported event types are described in the XOI documentation [xoi] in section "ChangeManager Event Types".
[71] With these properties, the property change is delegated to the so-called main child of the meta type instance. Method *setPropValue()* for the main child (also) reports an event of type @PropertyChange, but at that time no (possibly required and relevant) adjustments in the meta type instance itself took place yet.
[72] which might also have to be first derived from the standard classes just for this purpose

The following is the implementation pattern based on the 2nd approach:

```
public func initialize(pFather, pName)
{
    <BaseClass>::initialize(pFather, pName);

    var tChMgr = oiGetChangeManager();

    if (tChMgr != NULL)
        // consider property changes in (grand) children
        tChMgr.register(self, @(@PropertyChange), [@(self), 0]);
}

public func receivePropertyChange(pPublisher, pKeys)
{
    // Do nothing during Meta type initialization of pPublisher!
    if (pPublisher.isA(GoMetaType) &&
        (!pPublisher.isMetaInitialized() || pPublisher.sInSetAddStateCode))
        return;

    // process pKeys and check for possible re-action:
    ...
}
```

5.14 Collision detection

If a new element is added to a defined attach point of an element already in the group by means of a *CreateObj* action or function *xOiCreateArticle()* (resp. *xOiCreateArticle2()*)[73], a standard check is performed to determine whether the new element collides with other elements in the group[74]. If a collision is detected, the insertion of the new element is rejected.

In the case of inaccurate geometry data resp. inaccurately defined attachment points, collision detection for the group elements is often switched off completely or temporarily (during the insertion process) in the OFML data[75]. This has the following consequences:

1. In the case of groups planned “around the corner”, the new element can collide with other group elements.
2. It is not possible to insert the new element *between* two other group elements (insert mode 2)!

In order to avoid these problems, the geometry data and attach points should be defined as precisely as possible from the outset so that the collision detection need not be switched off!

If the deactivation of the collision detection cannot be prevented, at least the second problem can be circumvented with some programming effort (when using *xOiCreateArticle()* resp. *xOiCreateArticle2()*). The basic idea is to use methods *isBusyAttPt()*[76] resp. *isFreeAttPt()*[77] to check whether the attach point of the reference object is already occupied, and in this case to force the collision detection.

The following implementation pattern deals with the case where, for a **data creation based on *Meta types***, the collision detection is temporarily deactivated for the new element by means of *GSetup* flag `4096`.

[73] Concerns the group types *xOiCustomPlGroup*, *xOiJointPlGroup* and *xOiLayoutGroup*.
[74] via method *checkChildColl()* of OFML interface *Complex* [ofml]
[75] via method *disableCD()* of OFML interface *Base* [ofml]
[76] class *xOiJointPlGroup*
[77] class *xOiLayoutGroup* and *xOiCustomPlGroup*,

In the project-specific class derived from *GoMetaType*, method *isEnabledCD()* must be overwritten:

```
public func isEnabledCD()
{
    var tRes = GoMetaType::isEnabledCD();

    var tFather = self.getFather();
    if (!tRes && tFather.isA(myLayoutGroup))
        tRes = tFather.needCD4Insertion();

    return(tRes);
}
```

In the project-specific class for the planning group *myLayoutGroup*:

```
static var sNeedCD4Insertion = 0;

func needCD4Insertion()
{
    return(sNeedCD4Insertion);
}

// Used in OAP via a MethodCall action
public func myAddElement(pRefObj, pAttPt, pPID, pModel)
{
    if (!isFreeAttPt(pRefObj, pAttpt))
        // enforce re-positioning of previous neighbors (insert mode 2)
        sNeedCD4Insertion = 1;

    var tNewEl = xOiCreateArticle(self, pRefObj, pAttpt, pPID, tModel, "", @VarCode);

    sNeedCD4Insertion = 0;
    ...
}
```

The implementation pattern above assumes that collision detection is switched on (by default) for the child objects of the Meta type instance [78]!

Therefore, the **more general solution** to the problem is not based on overwriting *isEnabledCD()* in the classes of the group elements, but on temporarily activating collision detection during *xOiCreateArticle()* (resp. *xOiCreateArticle2()*):

```
// Used in OAP via a MethodCall action
public func myAddElement(pRefObj, pAttPt, pPID, pModel)
{
    var tEl, tDisableCD = @(); // elements to be disabled again after xOiCreateArticle()

    if (!isFreeAttPt(pRefObj, pAttpt)) {
        // enforce re-positioning of previous neighbors (insert mode 2)
        foreach(tEl; getElements()) {
            if (!tEl.isEnabledCD()) {
                tDisableCD.pushBack(tEl);
                tEl.enableCD();
            }
        }
    }

    var tNewEl = xOiCreateArticle(self, pRefObj, pAttpt, pPID, tModel, "", @VarCode);

    // tNewEl.disableCD(); // ?

    foreach(tEl; tDisableCD) tEl.disableCD();

    ...
}
```

[78] If the collision detection is enabled for the Meta type instance, during the actual check for its child objects, via global function *oiCollision()*, the status applies which is defined for the objects to be checked against each other via *disableCD()* resp. *enableCD()*.

6 OFML-Debugging

The remarks in this section refer to the OFML debugging in the pCon.planner using the OFML Console provided with the plugin X3G-OFC.

6.1 Debugging the *xOiOAPManager*

The *OAP Manager* is the interface between the OAP data and the applications. For that, at runtime there is exactly one instance of the class *xOiOAPManager*. In case of problems during processing the OAP data, an OFML debug log for this class can be helpful[79]. Essentially 3 top-level methods are relevant here:

- When selecting an article the application calls methods *getArticleData()* and *getInteractors()* (in this order).
- When an interactor is activated the application calls method *getActionData()* for each action of the interactor[80].

(The detailed specification of these methods can be found in the XOI documentation [xoi][81].)

Calls of *getArticleData()*/*getInteractors()* occur again - with unchanged selection - if the mouse pointer enters another view of the Planner (because for each view the interactor symbols have to be determined and displayed). In order to avoid unnecessary output in the log file, it is recommended to debug in 1-view mode.

Errors in the OAP tables can be found relatively easily and quickly by looking for the following output in the log file (`debug.out`):

```
W: OAP database access error: ...
```

For this purpose, at least the debug modes `Warn(ing)`, `Func` and `Func2` must be set and the function trace level should be set to at least 5 (if *ActionChoice* actions are involved, at least to 7).

6.2 Debugging *CreateObj*

If a *CreateObj* action with a given attach point (or a *MethodCall* action using global functions *xOiCreateArticle()* and *xOiCreateArticle2()*) does not produce the desired result (object is not created), the cause often is not immediately obvious.

Here are 2 troubleshooting tips:

1.

One reason could be that a collision of the new object with already existing objects would occur at the position of the selected attach point (either due to incorrectly positioned attach point or due to "inaccurate" geometries).

Whether this is the case can be seen quite quickly with the debug setting *Collision*. The log file then contains entries like this

```
c.e1 (myPlGroup) detected collision: c.e1.ch (myClass) >|< c.e1.e1 (myClass) ...
```

[79] The OFML Console already provides a suitable debug profile OAP for this purpose.
[80] If an error occurs while accessing the OAP data of an action, the processing of the actions is terminated.
[81] Class *xOiOAPManager* is contained in section "Utility Classes".

2.

If 1. does not apply, (unfortunately) one must dive a little deeper into the process of object creation. It is important to know that the above actions are based on the so-called *checkAdd()* mechanism. For details about this see the OFML specification. For the moment it is enough to know the following:

Method *checkAdd()* is called on the parent object to determine the position and rotation for the new object. Before this, the attach point specified in the action is activated at the reference object (neighbor) using method *setActiveAttPt()* (interface *AttachPts* [xoi]).

Thus, in general, in case of problems with *CreateObj*, the class of the parent object (and possibly relevant base classes[82]) has to be specified in the debug settings. In the log file then you have to look for clues in the outputs for method *checkAdd()*.

However, in a first step it is recommended to specify only the class *OiGlobal* and to search in the log for the global function *oiGetPosRot4AttachPts()*. This function is used by all standard implementations of *checkAdd()* in the base classes of OI and XOI for the actual determination of position and rotation for the new object..

One cause for a failure of *checkAdd()* could be, e.g., that *there is no definition for the given attach point*. Then you would see this relatively quickly based on the output for the mentioned function:

```
1> OiGlobal::oiGetPosRot4AttachPts(object: c.e1.e1 (GoMetaType), [object: c.e1.ch
(GoMetaType), NULL, 1, 1, 1])
I: is child?: 0
I: list of att pts: @(...)
I: active attach point: @ApOb_12R
W: missing or wrong attach point definition for active attach point @ApOb_12R
1< OiGlobal::oiGetPosRot4AttachPts: NULL.
```

If this is not the case, the cause of the error could be that *there is no suitable opposite for the active attach point*. This can be seen in the log by adding the class *OiFuncs*. Then, the output for function *oiCalcCheckAttPt()* in the log (1 level below oiGetPosRot4AttachPts()) shows the following:

```
no opposite attach point(s) for @ApOb_12R ...
```

In case of attach points defined in meta data table `go_attpt.csv`, the cause could be that they are not defined as opposites in table `go_action.csv`.

6.3 Usage of *xOiOAPManager::setDBMode()*

Normally, the currently accessed OAP database will be kept open until access to the OAP database for another OFML program. This is somewhat inconvenient for the OAP data creator, as he/she cannot test "live" changes in the OAP data of the OFML program currently being edited.

The class *xOiOAPManager* provides the method *setDBMode()* to change this default behavior (mode *@KeepOpen*): With mode *@CloseOpen* OAP databases will be closed and opened at each time an object is selected or after an action was performed.If you enter the command

```
/t.getOAPManager().setDBMode(@CloseOpen)
```

in the command line of the OFML Console, then you can immediately test changes in the OAP data[83].

[82] e.g., in the case of *xOiJointPlGroup* the base classes *xOiLRPlGroup* and *xOiPlGroup*
[83] However, the `oap.ebase` must be rebuilt in the process.

7 Miscellaneous

7.1 EBase for control data tables

If a control data table used by planning group class (see section 5) has to be transferred to the `ofml.ebase` and an EBase version older than 1.2.3[84] is used, the following problem has to be considered:

If an empty variant code is specified in a table entry for option *@StartElement* or *@StartLayout*, like in

```
@StartElement;;[@foo_bar,"ArticleNr",@VarCode,"",[]]
```

the field must be enclosed in double quotation marks (") and double quotation marks in the field contents have to be replaced by 2 consecutive double quotation marks:

```
@StartElement;;"[@foo_bar,""ArticleNr"",@VarCode,""""",[]]"
```

Such a transformation is also required if the article number or the variant code contains a semicolon[85].

Advice:

The elements in the vector after the article number are optional. If these are not relevant, as in the example above, they should be omitted, which makes a transformation superfluous:

```
@StartElement;;[@foo_bar,"ArticleNr"]
```

7.2 Metatype-Type-Mapping

The specification contains no statement which series to specify in field 2 of the *Metatype2Type* table.

It is currently implemented in such a way that the series of the article is expected in this field, not the series in which the Meta type is created!

A consequence of this is that in the case where an OAP database is created for several commercial series, and there are articles from the various series that use one Meta type (ID) (no matter in which series this is created) for this Meta type multiple entries have to be created in the mapping table, for example:

```
man;foo;MTID;;OAP_TYPE
man;bar;MTID;;OAP_TYPE
```

(There may be other/better regulations in the future. In any case, a future change in the OAP core of the applications will contain a downward-compatible fallback, so that OAP data created according to the current state still will be processed correctly.)

[84] contained in version 1.4.0 of the OFML-Binaries
[85] This corresponds to the regulations known from the specifications of OCD and OAP for the representation of a table field.

Appendix

A.1 Document history

2023-11-07:

- Various minor corrections and improvements.
- Additional notes in section 3.3 (dynamic interactors) related to the target object of an action whose ID is used as the argument of a *methodCall()* expression in conditions.
- New section 3.5 on considering the configuration context for interactor visibility.
- In section 4.1 now the new global function *xOiCreateArticle2()* is mentioned.
- Section 5 now considers new class *xOiCustomPlGroup*.
- Added note regarding extended behavior of *xOiPlGroup::updateProperties()* in section 5.9.1.
- More precise description regarding usage of method *getPDLangauage()* in section 5.9.4.
- New note in section 5.9.4 regarding usage of new option *@CommonPropsPos*.
- New note in section 5.9.4 on reacting to possible changes in the product data when definitions of programmed properties are derived from attributes (e.g., choice lists) of properties of group elements.
- New hint on event type *@MetaMainChildPropChange* in section 5.13.
- Additional notes on collision detection in section 5.14.
- New section 6.3.

2022-01-03:

- New section 5.14 with notes on collision detection.
- Correction and addition regarding the debug settings in section 6.1.

2021-11-17:

- First version containing this history.
- Obsolete section 3.2 “Visibility of interactor symbols” has been replaced by a section that addresses the question of when it is better to bind an interactor to the planning group or to the group element.
- Sections 3.3 and 3.4 have been swapped.
- Update of section 3.3 on dynamic interactors, including notes on the use of placeholder `$INTERACTOR`.
- Additional note in section 4.4 on actions that remove the active object.
- New section 4.6 on the use of the dummy action type *NoAction*.
- Class *xOiTabularPlGroup* now is treated in section 5 "on an equal footing" with the older classes for planning groups.
- Additional note in section 5.1 on the use of object category *MethodCall*.
- Additional note in section 5.5 on method *xOiLayoutGroup::replaceElement()*.
- Extensive expansion and restructuring of section 5.9 with many new information and tips on creating and using of group properties (common properties).
- Renaming of section 5.11 (better description of the content) and addition of an example.
- Additional note in section 5.12 on the use of options *@ROPropsEditable4Common* and *@NonVisibleProps4Common*.
- New section 5.13 with notes on the realization of reactions to property changes of elements on the part of a group instance.